1 Einführung

Ich kann mich noch sehr gut daran erinnern, wie ich vor über 20 Jahren angefangen habe, Programmieren zu lernen. Meine ersten Programmiersprachen waren Basic, Assembler, Pascal und C++. Damals war OOP (objektorientierte Programmierung) das, was man heutzutage als »the next big thing« bezeichnen würde. Und ich muss zugeben, nachdem ich gelernt hatte, wie man objektorientiert programmiert, konnte ich mir nicht mehr vorstellen, jemals wieder anders zu programmieren.

Zu dieser Zeit, Anfang der 90er-Jahre, kamen die besten Programmierumgebungen noch von Borland. Und ich kann mich auch noch genau erinnern, wie sehr ich mich über eine neue Version von Turbo Pascal gefreut habe, weil sie ein Feature einführte, das heutzutage in jeder IDE selbstverständlich ist: Syntax-Highlighting.

Es gibt nicht viele Features in IDEs, die das Programmieren sehr viel angenehmer machen und die man – nachdem man einmal in ihren Genuss gekommen ist – einfach nicht mehr missen möchte. Heute zähle ich neben dem Syntax-Highlighting außerdem noch dazu: Code-Completion, Code-Folding, Refactoring-Support, Hintergrund-Kompilierung, einen integrierten Debugger und die Möglichkeit, einen Test auf Tastendruck oder Mausklick ausführen zu können – alles Dinge, die mich als Programmierer um Größenordnungen produktiver sein lassen, als wenn ich nur einen Texteditor hätte.

Mit der testgetriebenen Programmierung (siehe Kap. 4) ist es bei mir ähnlich wie mit der objektorientierten Programmierung: Nachdem ich vor inzwischen über 10 Jahren gelernt habe, wie man Unit-Tests mit JUnit schreibt, konnte ich mir nicht mehr vorstellen, jemals wieder ohne Tests zu programmieren. Und genauso programmiere ich auch heute noch!

In meinen Augen ist das testgetriebene Programmieren eine genauso große Errungenschaft wie das objektorientierte Programmieren. Beide Vorgehensweisen sind aus der modernen Softwareentwicklung nicht mehr wegzudenken. Und während Java als Programmiersprache ein gutes Werkzeug ist, um objektorientiert zu programmieren, ist JUnit ein gutes Werkzeug, um testgetrieben zu programmieren.

Und genau um dieses Werkzeug dreht sich dieses Buch, das man auch als Handbuch für JUnit bezeichnen könnte. Ich hoffe, wenn Sie es lesen, sind Sie besser in der Lage, das vielseitige Werkzeug JUnit in verschiedenen Situationen ideal einzusetzen.

1.1 Automatisierte Tests

Im vorigen Jahrtausend, als die testgetriebene Programmierung noch nicht Mainstream war, war es stattdessen üblich, Programme (nachdem man sie endlich erfolgreich kompiliert hatte) einfach auszuführen und nachzuschauen, ob das, was man gerade programmiert hatte, auch tatsächlich so funktionierte, wie man es sich vorstellte.

Schön, wenn dem so war.

Aber falls nicht, dann war entweder eine Sitzung mit dem Debugger fällig oder man spickte seinen Quellcode mit diversen Logmeldungen und führte ihn noch einmal aus – in der Hoffnung, durch die ganzen Logausgaben den Programmfluss nachvollziehen zu können und so die Stelle zu finden, wo etwas schiefging.

Da man jedoch immer nur die Funktionalität manuell testete, an der man gerade programmiert hatte, konnte es leicht passieren, dass man aus Versehen etwas anderes kaputt machte, ohne dass es bemerkt wurde.

Auch heutzutage ist es nach wie vor üblich, seinen Code mit Logmeldungen zu versehen, damit man im Falle eines Fehlers hoffentlich in der Logdatei einen Hinweis darauf finden kann, was schiefgegangen ist. Neben Open-Source-Bibliotheken wie Log4J, Commons Logging oder Logback bietet sogar das JDK seit der Version 1.4 hierfür (neben `System.out.println`) ein Logging-Framework im Package `java.util.logging`.

Ebenfalls immer noch üblich ist es – sehr zu meinem Unverständnis –, Software manuell zu testen. Dabei ist das manuelle Testen fehleranfällig, langsam, teuer und (wenn man es wiederholt macht) ziemlich langweilig.

Schreibt man hingegen automatisierte Tests, so kann man jederzeit auf Knopfdruck reproduzierbar, schnell (im Vergleich zum manuellen Testen) und ohne die Fehlerquelle Mensch feststellen, dass die selbst entwickelte Software noch das macht, was sie soll. Zugegeben, die Fehlerquelle Mensch ist immer noch da, schließlich werden automatisierte Tests von Menschen programmiert, aber das stupide Ausführen von Testschritten und das Vergleichen von Ist-Werten mit Soll-Werten übernimmt bei einem automatisierten Test ein Computer.

Die entscheidende Idee dabei ist, dass ein automatisierter Test völlig autark laufen kann, ohne dass ein Mensch irgendwelche Ausgaben lesen und interpretieren muss. Der automatisierte Test muss dazu nicht nur den Programmcode ausführen, sondern zusätzlich noch entscheiden, ob das Programm auch richtig funktioniert. Dies wird mit sogenannten *Assertions* (auf Deutsch: Behauptungen) gemacht. Eine Assertion vergleicht typischerweise das Ergebnis eines Methoden-

aufrufs oder den Zustand eines Objekts mit einem vom Testautor definierten Erwartungswert. Der Einsatz von *Mock-Objekten* (siehe Kap. 6) erlaubt Ihnen außerdem auch das Verhalten des Codes zu überprüfen, also ob bestimmte Methoden überhaupt aufgerufen werden und ob dabei die richtigen Parameter übergeben werden.

Wird ein automatisierter Test ausgeführt, so gibt es nur zwei Möglichkeiten: Entweder er war erfolgreich (wenn kein Laufzeitfehler aufgetreten ist und alle Assertions wahr waren) oder eben nicht.

Schlägt ein automatisierter Test fehl, der mithilfe der Testbibliothek JUnit geschrieben wurde, so trifft JUnit noch die Unterscheidung zwischen *Error* (ein unerwarteter Laufzeitfehler ist aufgetreten) und *Failure* (eine Assertion war falsch). Aber in der Praxis ist diese Unterscheidung nicht besonders relevant. Wichtig ist nur, dass alle Ihre Tests erfolgreich sind.

1.2 Der grüne Balken

Als JUnit noch nicht von den Java-IDEs unterstützt wurde, beinhaltete es neben einer Klasse zum Ausführen von Tests auf der Kommandozeile auch noch eine kleine GUI:

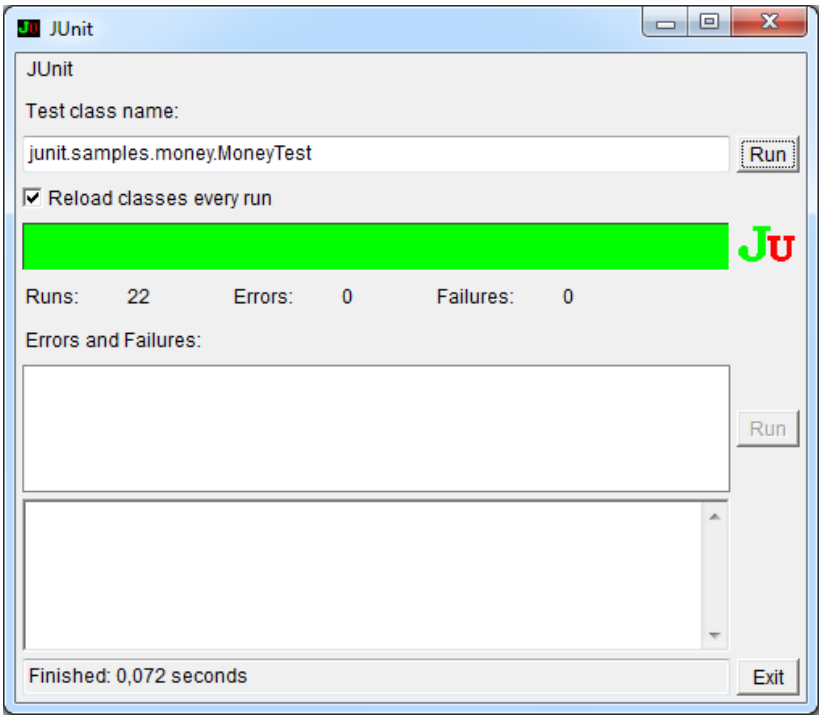

Abb. 1–1 *AWT TestRunner von JUnit 3.8.1 mit grünem Balken*

Das Schöne an dieser einfachen GUI war: Egal, wie viele automatisierte Tests ausgeführt wurden, es war immer sofort erkennbar, ob alle Tests erfolgreich waren (grüner Balken) oder ob zumindest ein Test fehlgeschlagen war (roter Balken).

Diese Metapher – grüner Balken = alle Tests erfolgreich; roter Balken = ein oder mehrere Tests sind fehlgeschlagen – wurde bei der Integration von JUnit in alle Java-IDEs beibehalten. Auch in Eclipse, IntelliJ IDEA oder Netbeans sieht man heutzutage entweder einen grünen oder einen roten Balken, je nachdem, ob alle Tests erfolgreich ausgeführt wurden oder nicht.

Und auch bei Builds auf Continuous-Integration-Servern spricht man kurz von grünen und roten Builds, wobei man mit einem »grünen Build« einen erfolgreichen Build bezeichnet und mit einem »roten Build« einen fehlgeschlagenen Build.

Dabei muss die Ursache für einen fehlgeschlagenen Build nicht unbedingt ein fehlgeschlagener Test sein. Auch Kompilierfehler oder andere Build-Fehler bewirken, dass der Build-Versuch mit der Farbe Rot markiert wird. Nur bis zum Ende erfolgreich durchgelaufene Builds bekommen die Farbe Grün.

1.3 Funktionale Tests

Bevor ich Ihnen in den folgenden Kapiteln erkläre, wie Sie mit JUnit automatisierte Tests schreiben können, möchte ich Ihnen zunächst noch kurz aufzeigen, welche Vielzahl von Testarten es gibt. Wenn Sie einen automatisierten Test schreiben, sollte Ihnen nämlich stets bewusst sein, welche Art von Test Sie gerade schreiben, da dies Auswirkungen darauf hat, welchen Testansatz Sie wählen sollten und wie die Testumgebung, die Sie für Ihren Test aufbauen, aussehen sollte.

Als Erstes kann man die Unterscheidung in *funktionale Tests* und *nichtfunktionale Tests* treffen. Mit einem funktionalen Test überprüfen Sie (wie der Name schon vermuten lässt), ob die von Ihnen entwickelte Software funktioniert, also ob Ihr Code das macht, was er machen soll.

Funktionale Tests lassen sich wiederum weiter unterteilen nach dem Umfang des getesteten Codes. Auf der untersten Ebene sind da die sogenannten *Unit-Tests* zu nennen. Ein Unit-Test überprüft das Funktionieren einer einzelnen Unit. Das kann in Java eine einzelne Klasse sein, viel öfter aber ist es sogar nur eine einzige Methode, manchmal auch eine Teilmenge der Methoden einer Klasse. Entscheidend bei einem Unit-Test ist, dass bei der Ausführung des Tests möglichst nur der *Produktionscode* der zu testenden Methode bzw. Klasse durchlaufen wird. (Der Begriff Produktionscode bezeichnet den Quellcode derjenigen Klassen, aus denen Ihre Applikation besteht, die also später tatsächlich auch ausgeliefert werden. *Testcode* ist hingegen der Code, der lediglich zum Testen Ihres Produktionscodes dient. Die durch Testcode erzeugten Klassen werden nicht ausgeliefert.)

Insbesondere sollten Unit-Tests nicht zu Festplatten-, Datenbank- oder anderen Netzwerkzugriffen führen. Um dies zu erreichen, werden die Abhängigkeiten

der getesteten Methode bzw. Klasse typischerweise durch sogenannte *Mock-Objekte* ersetzt, was ausführlich in Kapitel 6 beschrieben wird.

Der Vorteil von korrekt programmierten Unit-Tests ist, dass sie sehr, sehr schnell sind. Ein Unit-Test sollte nur wenige Millisekunden dauern. Nur so ist es möglich, in großen Projekten mit Tausenden von Unit-Tests innerhalb weniger Minuten einen Build durchzuführen, der das gesamte Projekt kompiliert und alle Unit-Tests ausführt.

Da es aber nicht ausreicht, nur einzelne Methoden oder Klassen zu testen, sondern auch das Zusammenspiel mehrerer Klassen getestet werden sollte, gibt es auf der nächsten Ebene die sogenannten *Integrationstests*. In einem Integrationstest wird typischerweise auf den Einsatz von Mocking verzichtet und eine Testumgebung aufgebaut, in der alle nötigen Abhängigkeiten zur Verfügung stehen, wie beispielsweise ein Spring `ApplicationContext` und/oder eine Testdatenbank. Manchmal ist aber auch bei Integrationstests der Einsatz von Mocking nützlich, nämlich immer dann, wenn Sie Komponenten simulieren wollen, die explizit nicht mitgetestet werden sollen, wie beispielsweise ein externer Webservice.

Dabei wird im Testcode von Integrationstests (genauso wie bei Unit-Tests) direkt auf die Klassen des Produktionscodes zugegriffen. Integrationstests sind also sogenannte *Whitebox-Tests*.

Sowohl Unit- als auch Integrationstests werden meistens von Entwicklern im Rahmen der testgetriebenen Entwicklung (siehe Kap. 4) geschrieben, um sicherzustellen, dass der gerade implementierte Produktionscode auch tatsächlich so funktioniert wie gedacht. Das ist bei den Tests auf der nächsthöheren Ebene – den sogenannten *Szenariotests* – anders: Szenariotests werden typischerweise nicht während der Entwicklung geschrieben, sondern hoffentlich davor, manchmal auch parallel (wenn Sie beispielsweise einen Test-Ingenieur in Ihrem Team haben), selten danach. Mit einem Szenariotest wird nicht eine bestimmte Klasse oder ein bestimmtes Modul getestet, sondern es wird ein kompletter *Use Case* durchgespielt.

Ein Szenariotest kann ebenfalls als Whitebox-Test geschrieben werden – oder aber als *Blackbox-Test*. Das heißt, er spricht die zu testende Applikation nur über solche Schnittstellen an, die auch einem Benutzer zur Verfügung stehen. Dann hätten Sie einen Szenariotest, der gleichzeitig auch ein *Systemtest* ist. Wenn Sie beispielsweise eine Desktop- oder Mobile-Applikation entwickeln, sollten Ihre Systemtests die Applikation über die Bedienoberfläche kontrollieren. Falls Sie eine Webapplikation programmieren, sollten Ihre Systemtests einen Webbrowser fernsteuern. Man spricht in diesen Fällen auch von *GUI-*, *Frontend-* oder *Webtests*. Entwickeln Sie hingegen einen Webservice, dann sollten Ihre Systemtests diesen über HTTP-Requests testen.

Während Unit-Tests auf der untersten Ebene angesiedelt sind, da sie nur sehr wenig Produktionscode testen, sind Systemtests auf der höchsten Ebene zu finden, da diese den Code aller beteiligten Komponenten testen. Während ein Szena-

riotest (wenn er als Whitebox-Test geschrieben wurde) nur Ihre Applikationslogik testet, testet ein Systemtest auch Ihren Frontendcode mit. Dieser muss dabei gar nicht in Java geschrieben sein. Im Falle einer Webapplikation könnte der Frontendcode beispielsweise in *.jsp- oder *.jsf-Dateien sowie in JavaScript-Dateien stecken.

Dabei sollten Sie Ihre Systemtests immer in einer Testumgebung laufen lassen, die möglichst nah an der Produktivumgebung ist. Während zum Beispiel bei Integrationstests aus Performance-Gründen gerne In-Memory-Datenbanken eingesetzt werden, sollten Sie bei Systemtests darauf achten, dass die Testumgebung genau das gleiche Betriebssystem, das gleiche JDK und die gleiche Datenbank verwendet wie die Produktivumgebung, und zwar jeweils in der gleichen Version. Falls Sie verschiedene Betriebssysteme, Datenbanken, Webbrowser oder was auch immer unterstützen wollen, sollten Sie für jede mögliche Kombination eine Testumgebung haben, in der Sie Ihre Systemtests laufen lassen.

Neben den bisher genannten funktionalen Testarten, die sich gut anhand des Umfang des getesteten Codes unterscheiden lassen, gibt es noch drei besondere Arten funktionaler Tests, die sich schlecht in dieses Unterscheidungsschema pressen lassen, die ich hier aber trotzdem erwähnen möchte:

- Ein Szenariotest, wenn er als Blackbox-Test geschrieben wurde, ist gleichzeitig auch ein Systemtest. Um die Sache noch komplizierter zu machen, könnte ein solcher Test zusätzlich auch noch ein *Akzeptanztest* sein.

  Einen Akzeptanztest zeichnet aus, dass er die Akzeptanzkriterien einer Programmieraufgabe testet. Egal wie diese Aufgabe konkret aussieht – es könnte beispielsweise eine Anforderung, eine User Story, eine Task Card oder ein Bug-Report sein. Der entscheidende Punkt ist: Jede Programmieraufgabe sollte klar definierte Akzeptanzkriterien haben, deren Erfüllung bedeutet, dass die Aufgabe erledigt ist.

  Und wenn Sie immer automatisierte Akzeptanztests schreiben, die direkt aus den Akzeptanzkriterien abgeleitet sind, dann können Sie jederzeit sicher sein, dass Ihre Applikation alle bisher implementierten Anforderungen immer noch erfüllt, wenn diese Tests alle grün sind. Ein schönes Gefühl, das Sie nicht mehr missen möchten, wenn Sie es einmal erlebt haben. Ich komme auf das Thema Akzeptanztests noch einmal im letzten Abschnitt von Kapitel 4 zurück, wo es um ATDD (*Acceptance Test Driven Development*) geht.
- Als Nächstes sind die sogenannten *Regressionstests* zu nennen. Mit *Regression* wird die Art von Fehler bezeichnet, wenn etwas bereits einmal funktioniert hat, aber in einer neueren Version Ihrer Software plötzlich nicht mehr geht. Im weiteren Sinne sind eigentlich alle Tests immer auch Regressionstests, da sie ja verhindern sollen, dass irgendetwas plötzlich kaputt ist. Im engeren Sinne verstehe ich unter einem automatisierten Regressionstest jedoch einen Test, der explizit dafür geschrieben wird, das Auftreten einer Regression zu verhindern.

Hierfür ein Beispiel: Angenommen, Sie benutzen Java-Serialisierung, um Objekte in Ihrer Anwendung zu speichern. Ein typischer Unit-Test dafür sieht so aus, dass ein Objekt erzeugt, danach serialisiert und schließlich wieder deserialisiert wird. Anschließend wird überprüft, ob das deserialisierte Objekt die gleichen Eigenschaften hat wie das ursprünglich erzeugte. Nun stellen Sie sich vor, Sie fügen ein weiteres Feld zu der Klasse des Objekts hinzu. Dadurch wird der Unit-Test wahrscheinlich nicht rot werden. Aber wenn Ihre Anwendung im Livebetrieb versucht, eine alte Version zu deserialisieren, die das Feld noch nicht hatte, kommt es sehr wahrscheinlich zu einem Laufzeitfehler. Um dies zu verhindern, sollten Sie, sobald Sie Serialisierung einsetzen, immer auch einen Regressionstest schreiben, der versucht, eine alte serialisierte Version eines Objekts, die als Binärdaten vorliegt, zu deserialisieren.

Ein weiteres gutes Beispiel für Regressionstests sind *Migrationstests*: Wenn Ihre Applikation beispielsweise eine Datenbank verwendet, benötigen Sie eine Strategie dafür, wie Sie mit Schemaänderungen umgehen – also zum Beispiel wenn eine zusätzliche Spalte zu einer Tabelle hinzugefügt werden muss.

Eine mögliche Strategie ist der Einsatz von Open-Source-Bibliotheken wie Liquibase[1] oder flyway[2], die gegebenenfalls Migrationsskripte ausführen, um die Datenbank auf den neusten Stand zu bringen. Wenn Sie diese Strategie verfolgen, dann sollten Sie immer auch einen Migrationstest schreiben, der diese Migrationsskripte testet und wie folgt funktionieren könnte:

Zunächst erzeugt der Test eine Testdatenbank, die mit einem möglichst alten Schema initialisiert wird, zum Beispiel durch ein SQL-Skript, das den Zustand des Schemas am Anfang der Entwicklung widerspiegelt. Danach werden alle Migrationsskripte ausgeführt. Tritt dabei keine Exception auf, ist das schon einmal ein gutes Zeichen dafür, dass die Migrationsskripte syntaktisch richtig sind. Wenn Sie anschließend noch das Schema der Testdatenbank mit dem erwarteten Schema vergleichen, können Sie sogar überprüfen, ob die Migrationsskripte auch inhaltlich richtig sind. Dafür benötigen Sie natürlich einen Mechanismus, mit dem Sie einfach an das erwartete Schema herankommen. Vielleicht pflegen Sie ja in Ihrem Produktionscode ein SQL-Skript zum Initialisieren einer leeren Datenbank oder das von Ihnen verwendete Persistenzframework bietet hierfür etwas an.

Wie auch immer: Mit Regressionstests testen Sie die Abwärtskompatibilität Ihrer Anwendung, und dazu benutzt ein Regressionstest immer irgendwelche Testdaten, die von einem älteren Stand Ihrer Software erzeugt wurden.

1. *http://www.liquibase.org/*
2. *http://flyway.googlecode.com/*

- Die letzte Testart, die ich Ihnen hier vorstellen möchte, sind die sogenannten *Learning-Tests*. Wenn Sie eine neue Bibliothek in Ihrem Projekt einsetzen, auf eine externe Schnittstelle (wie beispielsweise einen Webservice) zugreifen müssen oder vielleicht auch nur eine API des JDK benutzen wollen, mit der Sie vorher noch nie zu tun hatten, müssen Sie als Erstes lernen, wie die neue Bibliothek, externe Schnittstelle oder API funktioniert. Hoffentlich steht Ihnen dafür irgendeine Dokumentation zur Verfügung, die Sie lesen können. Oder noch besser: Vielleicht gibt es sogar Beispielcode, auf dem Sie aufbauen können.

  Aber weder Dokumentation noch Beispielcode können Ihnen garantieren, dass Sie alles richtig verstanden haben und sich die Bibliothek, externe Schnittstelle oder API auch so verhält, wie Sie es erwarten. Deshalb sollten Sie in einem solchen Fall, während Sie die Dokumentation lesen und lernen, immer auch gleich ein paar automatisierte Learning-Tests mit JUnit schreiben, die beweisen, dass die Third-Party-Komponente in Ihrer Laufzeitumgebung auch tatsächlich so funktioniert, wie Sie sich das vorstellen.

  Ein automatisierter Learning-Test hat mehrere Vorteile gegenüber dem sofortigen Einsatz einer neuen Third-Party-Komponente im Produktionscode: Zum einen können Sie die neue Komponente in Isolation testen, und wenn Sie dabei ein Problem feststellen, wissen Sie, dass es nicht an Ihrem Produktionscode liegt. Darüber hinaus haben Sie sofort einen JUnit-Test, den Sie an einen Bug-Report anhängen können, falls Sie den festgestellten Fehler in den Bug-Tracker des Herstellers bzw. Open-Source-Projekts eintragen.

  Sie sollten die Learning-Tests übrigens nicht wegwerfen. Auch wenn Learning-Tests nicht von Ihrem Continuous-Integration-Server ausgeführt werden sollten – schließlich testen Sie ja damit Third-Party-Komponenten, die außerhalb Ihrer Kontrolle sind –, sollten Sie die Learning-Tests trotzdem in Ihre Versionskontrolle einchecken. So können Sie später einmal, zum Beispiel falls von einer Open-Source-Bibliothek eine neue Version erscheint, ganz einfach durch das nochmalige Ausführen Ihrer Learning-Tests feststellen, ob Ihre Anwendung auch mit dieser neuen Version noch funktioniert. Oder Sie können, falls ein externer Webservice plötzlich Probleme macht, ebenfalls einfach die Learning-Tests noch einmal ausführen, um die Fehlerursache schnell einzukreisen.

  Ihre Learning-Tests sind sozusagen Ihre ausführbare Dokumentation der Third-Party-Komponente. Und sollte die Dokumentation einmal von der Realität abweichen (was leider gar nicht so selten vorkommt, zum Beispiel wenn die Dokumentation veraltet ist), können Sie das jederzeit durch das Ausführen Ihrer Learning-Tests feststellen.

1.4 Nichtfunktionale Tests

Auch wenn Sie viel Aufwand in funktionale Tests gesteckt haben, ein funktionierendes Programm ist noch lange kein gutes Programm. Software sollte außerdem schnell, ressourcenschonend, stabil, sicher, fehlertolerant sowie benutzerfreundlich sein und darüber hinaus auch noch ansprechend aussehen. Und genau diese Aspekte können durch *nichtfunktionale Tests* überprüft werden.

Dies müssen keine manuellen Tests sein, JUnit eignet sich auch zum Schreiben von automatisierten *nichtfunktionale*n Tests: Ob Ihr Programm schnell ist und vor allem, dass es mit der Zeit nicht langsamer wird, können Sie mithilfe automatisierter *Performance-Tests* prüfen. Und ob Ihre Applikation ressourcenschonend und stabil ist, lässt sich durch automatisierte *Lasttests* bzw. *Stresstests* überprüfen. Zwar gibt es hierfür auch spezielle Tools, wie beispielsweise das bekannte JMeter[3]. Aber der Vorteil von automatisierten Performance- und Stresstests mit JUnit ist, dass Sie nicht nur Ihre gesamte Applikation, sondern auch dedizierte Komponenten, Klassen oder Methoden testen können. Diesem Thema widmet sich das Kapitel 10.

Um zu gewährleisten, dass Ihre Software auch robust und fehlertolerant ist, sollten Sie nicht nur Tests für den sogenannten *Happy Path* schreiben, also für den Normalfall, sondern auch für mögliche Fehlerfälle. Am einfachsten ist es natürlich, wenn Ihre Methoden im Produktionscode Exceptions gar nicht erst fangen, sondern einfach weiterwerfen, Sie also die Fehlerbehandlung anderem Code überlassen. Auch wenn ich das generell für eine gute Strategie halte, haben Sie sicherlich ein paar Stellen in Ihrem Produktionscode, wo Sie Exceptions behandeln. Auch dieser Code sollte getestet werden. Wie das geht, erkläre ich Ihnen in Kapitel 9.

Für einige *nichtfunktionale* Qualitätsmerkmale, wie Sicherheit, Benutzerfreundlichkeit oder ansprechendes Aussehen, ist es sehr schwer, automatisierte Tests zu schreiben, aber zumindest für Benutzerfreundlichkeit und Aussehen gibt es Ansätze, wie beispielsweise die Open-Source-Projekte web-accessibility-testing[4] und Fighting Layout Bugs[5]. Trotzdem werden solche Qualitätsmerkmale typischerweise durch manuelle *Penetrationstests* und *Usability-Tests* überprüft.

Über die nach außen hin sichtbaren Qualitätsmerkmale hinaus sollte Ihre Software aber auch über eine gewisse *innere Qualität* verfügen, das heißt, sie sollte zum Beispiel leicht skalierbar und wartbar sein. Dafür sollte Ihr Quellcode gut verständlich sein und sich an Ihre Coding-Konventionen halten. Klassen, Felder, Methoden, Parameter und Variablen sollten aussagekräftige Namen haben. Ihre Architektur sollte einfach sein und die bekannten Prinzipien der objektorientierten Programmierung befolgen, wie beispielsweise das Single-Responsibi-

3. *http://jmeter.apache.org/*
4. *http://web-accessibility-testing.googlecode.com/*
5. *http://fighting-layout-bugs.googlecode.com/*

lity-Prinzip, das Liskov'sche Substitutionsprinzip oder das Dependency-Inversion-Prinzip.

Zur Überprüfung dieser inneren Qualitätsmerkmale dienen für gewöhnlich manuelle Codereviews. Es gibt dafür aber auch zahlreiche kommerzielle sowie Open-Source-Tools, von denen ich einige in Abschnitt 4.4 nenne. Sie können jedoch auch JUnit benutzen, um sogenannte *Architecture Conformance Tests* (oder kurz: *Architekturtests*) zu schreiben, mit denen Sie überprüfen können, ob sich der Quellcode Ihrer Applikation auch bei fortschreitender Entwicklung an alle von Ihnen aufgestellten Architekturregeln hält. Mehr dazu erfahren Sie in Abschnitt 10.4.