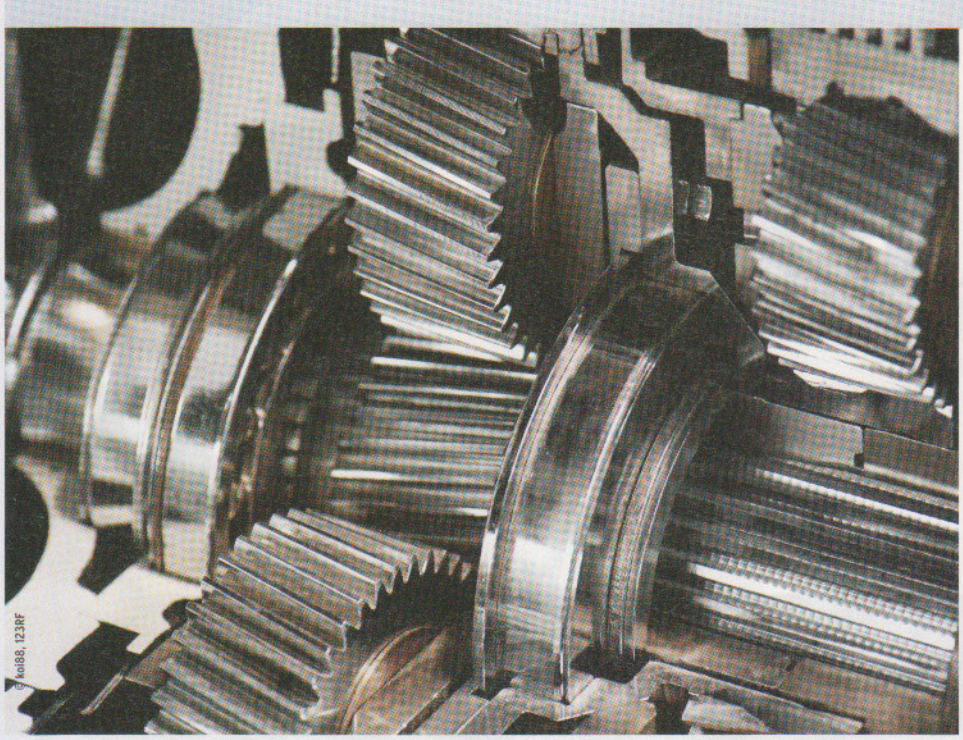
Testgetriebene Software-Entwicklung

Präziser Antrieb

Das Credo des Test Driven Development (TDD): Wer zuerst Tests schreibt und dann erst programmiert, erzeugt sauberen Code und geprüfte Programme. Wie das in Java und weiteren Sprachen geht, zeigt dieser Artikel, der auch die Vor- und Nachteile der Methode abwägt. Gunnar Wrobel


© aoi88, 123RF

Programmieren macht Spaß. Der Entwickler schreibt ein paar Zeilen Code, und schon erscheint eine Grafik auf dem Bildschirm oder ein Roboter bewegt seinen Arm. Besonders im Bereich freier Software ist das die Herangehensweise fast jedes Programmier-Einsteigers.

Rot und Grün

Die Verfechter testgetriebener Programmierung predigen jedoch seit Jahren, dass so Software entsteht, die keine hohe Qualität aufweist. Die neue Lehre propagiert das Mantra „red, green, refactor" [1]. Diesem folgend schreibt der Programmierer erst einen Test. Der schlägt natürlich fehl – und weist damit gemäß Konvention den Status Rot auf. Im zweiten Schritt fügt der Entwickler den eigentlichen Code hinzu. Erfüllt dieser den vorher erstellten Test, wechselt der Status auf Grün. Dabei muss der Code zum Erfüllen der Testbedingung nicht zwingend bereits sauber geschrieben sein. Der letzte Schritt besteht aus Geradeziehen der Codestruktur, Umstrukturieren und Refactoring. Er muss allerdings unter Beachtung der geschriebenen Tests erfolgen: Der Status darf durch Refactoring nicht von Grün auf Rot wechseln.

Das angestrebte Ziel dieses Verfahrens: sauberer Code mit hoher Qualität. Der Entwickler hat nicht mehr einfach nur Ideen oder Ziele vor Augen, die er sofort in Code umsetzt. Stattdessen treibt ein mit Hilfe der Tests definiertes Korsett den eigentliche Code. Entsprechend heißt diese Vorgehensweise testgetriebene Entwicklung.

Die Überlegungen hinter diesem Verfahren, die Vorteile und auch die Nachteile, diskutiert dieser Artikel gegen Ende noch einmal ausführlich. Zunächst soll ein praktisches Beispiel die Vorgehensweise anschaulich machen.

Das Beispiel verwendet so genannte Unit-Tests. Dabei handelt es sich um sehr kleinteilige Tests, die nah am Code nur Bruchstücke oder Einheiten (Units) des Programms testen. Der Begriff testgetriebene Entwicklung bezieht sich so gut wie immer auf Units.

Werkzeuge für Unit-Testing existieren für praktisch jede Programmiersprache. Das folgende Beispiel könnte sich also jeder beliebigen Sprache bedienen. Da Java jedoch sehr verbreitet ist und für viele Entwickler gut lesbar, demonstrieren die folgenden Zeilen testgetriebene Entwicklung in dieser Sprache.

Schritt für Schritt

Die Programmieraufgabe besteht darin, mit dem Avatar-Service Gravatar zu kommunizieren und einer E-Mail-Adresse ein Avatar-Bild zuzuordnen. Das Verfahren ist unter [2] beschrieben. Ein sehr ähnliches Beispiel hat der Autor bereits in einem früheren Linux-Magazin in der Sprache PHP umgesetzt [3]. Unter Java ist Junit [4] die Referenz in Sachen Unit-Testing. Wer die folgenden Schritte selbst durchführen möchte, installiert also das Paket in der Linux-Distribution seiner Wahl. Unter Ubuntu liefert das Paket »junit4« das Testwerkzeug.

Der Entwickler sollte an dieser Stelle erst einmal das grundlegende Setup prüfen. Zu diesem Zweck schreibt er einen ersten Test in die Datei »GravatarTest.java« (Listing 1). Stimmt alles, kompiliert der Java-Compiler den Testfall, der sich anschließend fehlerfrei ausführen lässt. Der

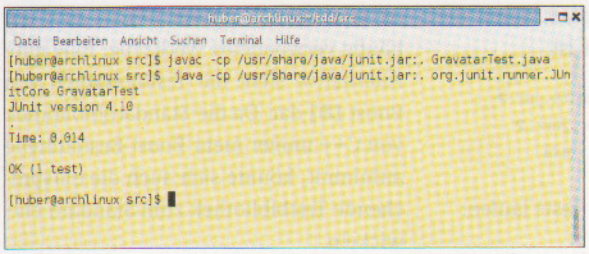


Abbildung 1: Der erste Test ist erfolgreich, bestätigt aber lediglich die korrekte Installation von Junit.

folgende Befehl kompiliert den Test in der Java-Quelldatei zu Bytecode:

```
javac -cp /usr/share/java/junit4.jar:.➥
 GravatarTest.java
```

Danach sollte sich eine Datei »GravatarTest.class« im Verzeichnis befinden. Der nächste Befehl startet Junits »TestRunner« (»org.junit.runner.JUnitCore«) mit »GravatarTest« als Testfall:

```
java -cp /usr/share/java/junit4.jar:.➥
 org.junit.runner.JUnitCore GravatarTest
```

Der einzige definierte Test (»public void testTrue«) prüft mit »assertTrue(true)« nur, ob »true« auch wirklich wahr ist. Läuft der Test wie in Abbildung 1 ab, funktioniert die Junit-Installation wie erwartet.

In beiden Befehlszeilen legt die Option »-cp« die Pfade fest, in denen Java-Klassen zu finden sind. Dieser Klassenpfad nimmt an, dass das Jar-Archiv für Junit unter »/usr/share/java/junit4.jar« installiert ist. Jeder Ort, an dem benötigte Bibliotheken liegen, muss hinter »-cp« angegeben sein, auch das aktuelle Verzeichnis, kurz ».«.

Listing 2 erweitert die einzelne Prüfung zu einer Testsuite, die mehrere Tests aufnimmt. In der Praxis würde der Programmierer langsamer vorgehen und nur einen oder zwei Tests hinzufügen, dann den passenden Code schreiben, weitere Tests hinzufügen und so weiter. Bei Skriptsprachen ließe sich eine solche Testsuite schon ausführen.

Java ist etwas anspruchsvoller: Ohne eine Klasse »Gravatar« weigert sich der Compiler, die Suite in Bytecode zu konvertieren. Schließlich verwendet die Testsuite diese Klasse und ruft die Funktionen »getId()« sowie »getAvatarUrl()« auf. So muss der Entwickler die Klasse in der Datei »Gravatar.java« zumindest mit ein bisschen Code ausstatten (Listing 3).

Alles auf Rot

Der folgende Befehl auf der Kommandozeile sollte nun sowohl Code als auch Testsuite fehlerfrei kompilieren:

```
javac -cp /usr/share/java/junit4.jar:.➥
 Gravatar.java GravatarTest.java
```

Das folgende Kommando führt die Tests der Suite aus:

```
java -cp /usr/share/java/junit4.jar:.➥
org.junit.runner.JUnitCore GravatarTest
```

Das endet aber im Desaster: Alle fünf Tests schlagen wegen der unvollständigen Implementierung der »Gravatar«-Klasse fehl (Abbildung 2).

Es ist an der Zeit, den eigentlichen Code zu schreiben, und zwar getrieben durch die Tests. Den ersten Test, nämlich ob die E-Mail-Adresse »test@example.org« den Hash »0c17bf66e649070167701d2d3cd71711« ergibt, erfüllt der Entwickler auf ganz naive Art und Weise:

```
public String getId(String mail) {
  return "0c17bf66e649070167701d2d3cd71711";
}
```

Dabei geht es wirklich nur darum, dass der Code den Test erfüllt. Solange der Entwickler keinen zweiten Test definiert hat, der eine komplexere Lösung erzwingt, spricht nichts dagegen, zunächst die einfachste Form der Implementierung zu wählen.

Das ist einer der zentralen Aspekte testgetriebener Entwicklung: Die Software sollte nur so komplex wie nötig sein. Verzichtet der Entwickler auf die testgetriebene Entwicklung, ist dies selten der Fall. Spontane Ideen beim Coden führen schnell zu zusätzlichen Features, neuen Abstraktionsebenen oder unnötiger Modularität. Daraus ergibt sich fast zwangsläufig unnötige Komplexität, die sich in

Listing 1: Der erste Test

```
01 import junit.framework.TestCase;
02
03 public class GravatarTest extends TestCase {
04     public void testTrue() {
05         assertTrue(true);
06     }
07 }
```

Listing 2: Testsuite

```
01 import junit.framework.TestCase;
02
03 public class GravatarTest extends TestCase {
04
05     public void testAddressesOne()
06     {
07         Gravatar g = new Gravatar();
08         assertEquals("0c17bf66e649070167701d2
   d3cd71711", g.getId("test@example.org"));
09     }
10
11     public void testAddressesTwo()
12     {
13         Gravatar g = new Gravatar();
14         assertEquals("ae46d8cbbb834a85db7287f
   8342d0c42", g.getId("x@example.org"));
15     }
16
17     public void testIgnoreCase()
18     {
19         Gravatar g = new Gravatar();
20         assertEquals(
21 "55502f40dc8b7c769880b10874abc9d0",
22                 g.getId("Test@EXAMPLE.com")
23         );
24     }
25
26     public void testTrimming()
27     {
28         Gravatar g = new Gravatar();
29         assertEquals(
30 "55502f40dc8b7c769880b10874abc9d0",
31                 g.getId(" Test@Example.com ")
32         );
33     }
34
35     public void testAvatarUrl()
36     {
37         Gravatar g = new Gravatar();
38         assertEquals(
39             "http://www.gravatar.com/avatar/
   0c17bf66e649070167701d2d3cd71711",
40             g.getAvatarUrl(" Test@
   Example.orG ")
41         );
42     }
43 }
```

der weiteren Entwicklung der Software als hinderlich erweisen kann.

Für den zweiten Test der Testsuite in Listing 2 reicht der naive Ansatz schon nicht mehr aus. Um den Rückgabewert einer zweiten E-Mail-Adresse zu ermitteln, muss der Entwickler tatsächlich die MD5-Prüfsumme berechnen. Die Bibliothek »org.apache.commons.codec.digest.DigestUtils« bietet sich an, um das unter Java zu erledigen. Unter Ubuntu Linux findet sie sich im Paket »libcommons-codec-java«.

Listing 4 erfüllt als Inhalt der Datei »Gravatar.java« die beiden ersten Tests. Der Entwickler kompiliert erneut, diesmal mit erweitertem Klassenpfad, damit die neue Bibliothek verfügbar ist:

```
javac -cp /usr/share/java/junit4.jar:?
/usr/share/java/commons-codec.jar:.?
 Gravatar.java  GravatarTest.java
```

Anschließend lässt er die Tests laufen:

```
java -cp /usr/share/java/junit4.jar:?
/usr/share/java/commons-codec.jar:.?
 org.junit.runner.JUnitCore GravatarTest
```

Das ergibt aber immer noch drei gescheiterte Prüfungen.

Der dritte Test fordert, dass die Software den Hash unabhängig von Groß- und Kleinschreibung der E-Mail-Adresse bildet. Das ist mit »toLowerCase()« rasch implementiert, wie die Zeilen 6 bis 8 in Listing 5 zeigen. Wie vom vierten Test gefordert, ignoriert der Code mit »trim()« zudem Leerzeichen vor oder nach der E-Mail-Adresse. Der fünfte Test schließt das Beispiel ab und fordert eine Methode, die dem Aufrufenden die komplette Avatar-URL für eine E-Mail-Adresse zurückgibt (Zeilen 10 bis 12).

Sind alle Verbesserungen schließlich wie in Listing 5 eingearbeitet, wechselt der Status der Testsuite von Rot auf Grün: Alle Tests laufen durch (Abbildung 3).

Listing 3: »Gravatar.java«

```
01 public class Gravatar {
02     public String getId(String mail) {
03         return "";
04     }
05
06     public String getAvatarUrl(String mail) {
07         return "";
08     }
09 }
```

Listing 4: Weitere Methoden

```
01 import org.apache.commons.codec.digest.DigestUtils;
02
03 public class Gravatar {
04     public String getId(String mail) {
05         return DigestUtils.md5Hex(mail);
06     }
07
08     public String getAvatarUrl(String mail) {
09         return "";
10     }
11 }
```

Listing 5: Die fertige Klasse

```
01 import org.apache.commons.codec.digest.DigestUtils;
02
03 public class Gravatar {
04     private final static String URL = "http://www.
   gravatar.com/avatar/";
05
06     public String getId(String mail) {
07         return DigestUtils.md5Hex(mail.toLowerCase().
   trim());
08     }
09
10     public String getAvatarUrl(String mail) {
11         return URL + getId(mail);
12     }
13 }
```

Andere Sprachen, andere Sitten

Java mag das Standardbeispiel sein, doch Unit-Test-Werkzeuge gibt es für praktisch jede Programmiersprache. Das Tool Junit [4] ist nicht Teil von Java, sondern ein separates Programm. Wer es nicht mag oder sich nicht nur auf Unit-Tests beschränken möchte, findet für Java auch alternative Ansätze. Eine sehr gute, Sprachen-unabhängige Referenz zu Unit-Tests liefert die Wikipedia-Seite [5].

Bei C ist die Situation nicht ganz so einfach. Hier existiert eine ganze Reihe alternativer Werkzeuge mit unterschiedlicher Ausrichtung, da die Sprache C in sehr verschiedenen Szenarien zum Einsatz kommt. Das Werkzeug Check (Abbildung 4) bietet sich für viele Standardsituationen an [6]. Ähnliche Vielfalt bietet auch C++. Cppunit [7] ist hier ein häufig verwendetes Werkzeug. Eine gute Alternative stellt die Testbibliothek von Boost [8] dar. Da die Standardbibliothek von C++ immer mehr Boost-Bestandteile aufnimmt, könnte sich auch die entsprechende Testbibliothek zum Standard entwickeln.

Unter PHP ist die Software PHP Unit [9] der Standard in Sachen Unit-Testing. Gelegentlich verwenden Projekte auch Simple Test [10]. Die restlichen Alternativen im PHP-Land rangieren derzeit jedoch unter ferner liefen.

Ruby, Python, Javascript

Ruby liefert das Unit-Testing schon als Element der Sprache in der Bibliothek Test::Unit [11] mit. Auch bei Ruby gibt es externe Alternativen, doch die Standardausstattung ist der unangefochtene Platzhirsch. Außerdem ist Test::Unit sehr gut erlern- und bedienbar.

Ähnlich sieht es bei Python aus: Hier kommt die Bibliothek Unittest [12] gleich mit der Programmiersprache mit und darüber hinaus sogar noch Doctest [13]. Während Unittest eher die standardmäßigen Unit-Tests bedient, bietet Doctest die interessante Möglichkeit, solche Tests in die Dokumentation einzelner Methoden zu schreiben.

Bei Javascript ist die Situation weniger übersichtlich. Hier sind allerdings auch die Einsatzgebiete sehr heterogen. Wer gerne überprüfen möchte, ob eine Bibliothek sauber in allen Browsern läuft, braucht naheliegenderweise ein Testwerkzeug, das diese mit in die Tests einbeziehen kann, beispielsweise Js-test-driver [14]. Gerade weil es verschiedene Engines gibt, die Javascript verarbeiten, erweist sich in diesem Bereich das Unit-Testing als besonders hilfreich. Wer dagegen Server-seitigen Javascript-Code für

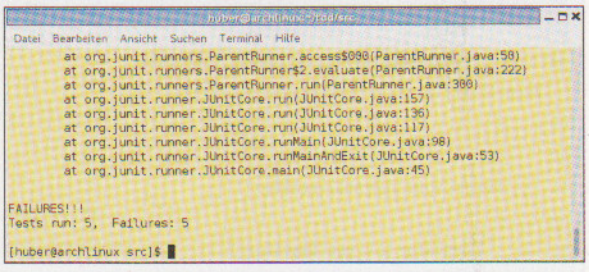


Abbildung 2: Der Programmierer hat noch viel zu tun. Alle fünf Tests der Suite schlagen zunächst fehl.

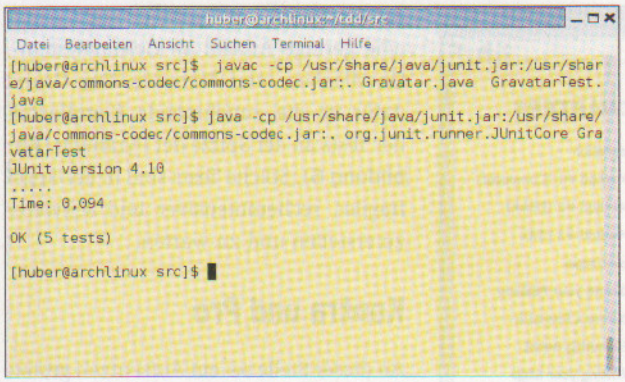


Abbildung 3: Alle Tests sind bestanden: Der Entwickler hat die Java-Klasse für den Gravatar-Client testgetrieben fertiggestellt.

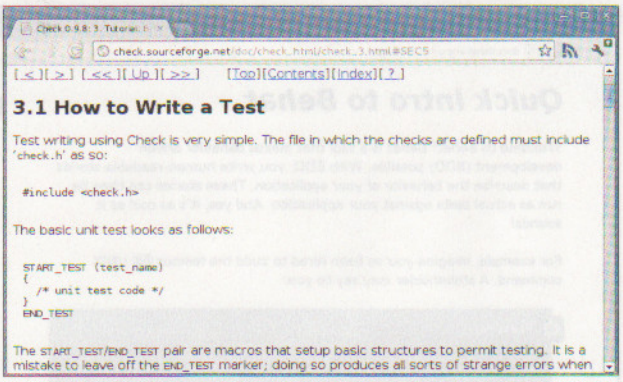


Abbildung 4: Unit-Test-Werkzeuge gibt es für fast alle Sprachen. Das Tool Check erledigt Unit-Tests für C-Programme.

Node.js testen möchte, greift passenderweise zu Nodeunit [15].

Für die meisten Programmiersprachen existiert eine Auswahl an Werkzeugen. Es bleibt die Aufgabe des Entwicklers, die Vor- und Nachteile abzuwägen und das passende für das jeweilige Einsatzszenario auszuwählen. Das Feld des Unit-Testing besitzt zahlreiche Facetten und lässt sich nicht in eine universell gültige Herangehensweise packen.

Das große Ganze

Unit-Tests drängen sich für die testgetriebene Entwicklung geradezu auf. Schließlich steht jeder Code-Einheit der entsprechende Test-Code gegenüber. Und derselbe Programmierer kann im Entwicklungsprozess Code und Tests parallel bearbeiten. Umfangreichere Codeblöcke, Klassen oder Module deckt das Unit-Testing jedoch im Normalfall nicht ab. Arbeiten die Schnittstellen von zwei Klassen sauber zusammen? Diese Frage sollten Unit-Tests an sich nicht stellen, für das Zusammenspiel der Softwarekomponenten ist sie aber essenziell.

Man spricht bei Tests auf dieser Ebene eher von integrativen Tests oder Modultests. Auch diese lassen sich vielfach noch mit den gleichen Unit-Test-Werkzeugen erledigen. Allerdings sind die dafür formulierten Tests umfangreicher, da nicht selten eine Vielzahl von Komponenten erforderlich ist, um das Zusammenspiel überhaupt prüfen zu können.

Wäre noch zu klären, was passiert, wenn der Entwickler die Grenzen einer Komponente verlässt und beispielsweise Service-übergreifend testen möchte. Auch hierbei lauern Fehler, die sich dank testgetriebener Entwicklung vermeiden lassen, falls die Tests auch diese Kommunikation einbeziehen. Man spricht dann von Systemtests. Im einfachsten Fall einer Applikation, die mit einer Datenbank spricht, lassen sich diese Tests häufig auch noch im Rahmen der Unit-Test-Werkzeuge abhandeln.

Das Problem bei derartigen Tests: Sie laufen nur bei verfügbarer Datenbank. Dazu benötigen sie im Normalfall eine Konfiguration und sind insgesamt fragiler als der herkömmliche Unit-Test. Die Anzahl denkbarer Fehler, die zur Testzeit zur Unerreichbarkeit der Datenbank führen können, ist sehr hoch.

Sobald es um Systemtests einer Architektur mit mehreren Komponenten geht, die zum Beispiel via REST miteinander reden, passt der Kontext eines Unit-Test-orientierten Testwerkzeugs häufig nicht mehr. Sinnvoller ist es, ein anderes Werkzeug zu verwenden, das beispielsweise ganz speziell auf REST-Schnittstellen zugeschnitten ist.

Oberflächentests

In die Kategorie Systemtests fallen auch Oberflächentests. Für die verschiedenen Desktopumgebungen gibt es hier häufig spezialisierte Werkzeuge. Von zunehmendem Interesse sind aber natürlich

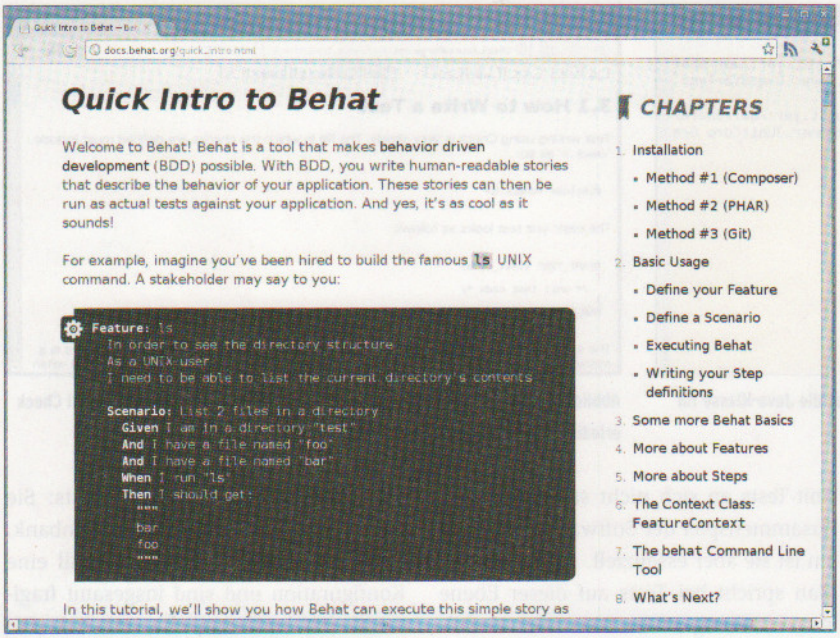


Abbildung 5: Das Werkzeug Behat formuliert verhaltensgetriebene Tests.

die Tests von Weboberflächen. Hier ist die herausragende Referenz das Testen mit Selenium [16].

Ähnlich breit angelegte Systemtests bestehen in fachlichen Tests, die der Kunde oder das Produktmanagement formulieren oder die die Entwicklungsabteilung zusammen mit dem Kunden aufstellt. Treiben solche Tests die Entwicklung an, spricht man von Behaviour Driven Development (BDD, verhaltensgetriebener Entwicklung, [17]).

Wer? Was? Wozu?

Der Kunde könnte beispielsweise formulieren, dass „die Nutzerin Lieselotte Müller den Knopf »Absenden« drücken möchte, damit ihre Nachricht an den Support verschickt wird". Diese Formulierung nach dem Schema „Wer? – Was? – Wozu?" lässt sich in BDD-Werkzeugen recht einfach in eine vorgegebene Form pressen:

```
given: a user Lieselotte Müller
when: the user presses the button 'Send'
then: the message is being sent to support
```

Die Einführung in das in PHP geschriebene Werkzeug Behat [18] demonstriert diese Form der Testfalldefinition sehr schön (Abbildung 5). Nachdem der Kunde seine Erwartung formuliert hat, ist es am Entwickler, die natürliche Sprache mit Code zu hinterlegen, sodass ein ausführbarer Test entsteht. Dieser schlägt anfänglich fehl und kann dann mit dem eigentlichen Code zum Funktionieren gebracht werden.

Unit-Testing, Modultests, Systemtests, Oberflächentests und BDD – wo fängt der normale Entwickler an, wo hört er auf? Eine Einordnung ermöglicht die so genannte Testpyramide [19]: Die überwiegende Mehrzahl der Tests sollten Unit-Tests sein, mit zunehmender Zahl der in die Tests einbezogenen Komponenten sollte die Menge der Tests abnehmen.

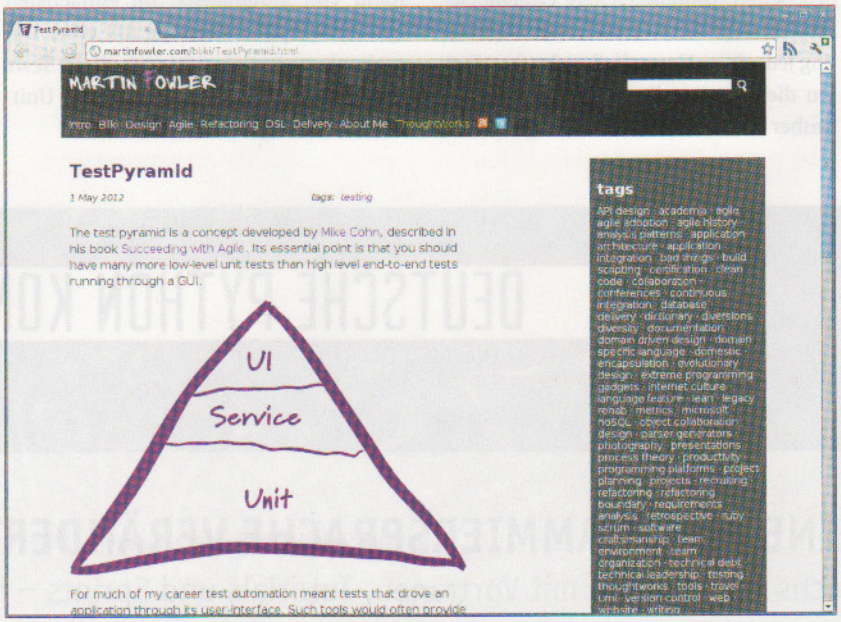


Abbildung 6: Am Fußende der Testpyramide stehen viele Unit-Tests, in der Mitte die Dienste und an der Spitze wenige komplexe Tests wie die der Benutzeroberfläche (UI).

An der Spitze der Pyramide steht damit eine geringe Anzahl an Gesamtsystem-, Oberflächen- und Fachtests. Der Grund dafür ist die zunehmende Komplexität der Tests an der Spitze der Pyramide (Abbildung 6). Solche Tests sind in der Regel fragiler, rechenintensiver und schwerer zu erstellen und zu warten.

Kontra und Pro

An dieser Stelle sei ein Aufseufzen gestattet. Es ergibt sich die schöne Gelegenheit, einen realen Satz aus einem Projekt zu zitieren – gesprochen von einem Entwickler mit hochrotem Kopf und halb erstickter Stimme: „Ich habe in zwei Wochen vier Zeilen realen Code geschrieben! Das ist doch völlig absurd!" Wer den vollen Testaufwand betreibt, auf allen Ebenen testet und diese Tests mit jedem Code-Commit automatisiert durchlaufen lässt, der kommt nicht umhin, einen guten Teil der Arbeitszeit in die Tests zu investieren.

Schon auf der Ebene der Unit-Tests ist in einer Pro-und-Kontra-Diskussion der Zeitbedarf ein naheliegendes Argument gegen die testgetriebene Entwicklung. Der Entwickler schreibt offensichtlich die doppelte Menge Code: den Test, dann die eigentliche Funktionalität. Je komplexer die getestete Funktionalität und das Testsetup, desto mehr Zeit kosten die Tests. Das Gegenargument der Testverfechter:

Den Zeitaufwand für Tests wiegt die erhöhte Codequalität wieder auf. Testen reduziert Fehler und spart daher Zeit beim Ausbessern. Darüber hinaus resultiert testgetriebene Entwicklung in saubereren Codestrukturen, die sich besser warten und weiterentwickeln lassen.

Ein weiterer Punkt, der das Testen erschwert: Es ist natürlich auch beim Schreiben von Tests möglich, gravierende Fehler zu machen. Eine gut wartbare Testsite, die problemlos mit der Entwicklung der Software mitgeht und keinen hohen Wartungsaufwand einfordert, erfordert Erfahrung mit der Testentwicklung. Genauso wie sich eine Software strukturell vor die Wand fahren lässt, ist dies bei einer Testsuite möglich.

Fazit

Testgetriebene Entwicklung ist kein Allheilmittel, das plötzlich alle Komplexität aus der Software-Entwicklung eliminiert. Sie hilft aber dabei, überlegter an die Entwicklung heranzugehen, und liefert im Schnitt die bessere Codequalität. Jeder Entwickler, der sich noch nicht mit der Thematik beschäftigt hat, sollte dies nachholen. Vor allem in größeren Unternehmen gehört die testgetriebene Entwicklung zum erwarteten Standardrepertoire.

Wer aber behauptet, nur testgetrieben entwickelter Code sei guter Code, der lehnt sich allerdings zu weit aus dem Fenster, besonders bei freier Software. Letztlich lebt ein gutes Projekt in erster Linie von den guten Ideen der Entwickler und der Begeisterung, mit der sie bei der Sache sind. Ein Großkonzern mag mit viel Geld eine neue Software in hoher Qualität konzipieren und testgetrieben entwickeln. Bei reinen Open-Source-Projekten dagegen schreibt ein Programmierer schon mal ein Feature rein aus Spaß und präsentiert es dann den Anwendern zum Ausprobieren.

Dass die ersten Versionen von Google oder Facebook testgetrieben entwickelt wurden, darf man bezweifeln. Heute bieten beide Unternehmen aber Musterumgebungen dieses Entwicklungstyps. Neuentwicklungen werden an Millionen von Nutzern ausgeliefert, ohne dass zum Schluss ein Heer von Testern jedes Mal die Gesamtumfang der Software getestet hätte oder jedes neue Deployment die Produktivumgebung regelmäßig lahmlegt [20]. Das ist nur dank testgetriebener Entwicklung möglich. *(mhu)* ■

Infos

[1] Beck, K., „Test-Driven Development by Example": Addison-Wesley/Vaseem, 2003

[2] Gravatar, „Creating the Hash": [http://de.gravatar.com/site/implement/hash/]

[3] Gunnar Wrobel, „Erfolgreich getestet": Linux Magazin 08/11, S. 96

[4] Junit: [http://www.junit.org]

[5] Übersicht zum Unit-Testing: [http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks]

[6] Check: [http://check.sourceforge.net]

[7] Cppunit: [http://cppunit.sourceforge.net]

[8] Boost Test Library: [http://www.boost.org/doc/libs/1_42_0/libs/test/doc/html/index.html]

[9] PHP Unit: [http://www.phpunit.de]

[10] Simple Test: [http://simpletest.org]

[11] Test::Unit in Ruby: [http://ruby-doc.org/stdlib-1.9.3/libdoc/test/unit/rdoc/Test/Unit.html]

[12] Unit Testing in Python: [http://docs.python.org/library/unittest.html]

[13] Doctest in Python: [http://docs.python.org/library/doctest.html]

[14] Js-test-driver: [http://code.google.com/p/js-test-driver/]

[15] Nodeunit: [https://github.com/caolan/nodeunit]

[16] Selenium: [http://seleniumhq.org]

[17] BDD: [http://de.wikipedia.org/wiki/Behavior_Driven_Development]

[18] Behat: [http://docs.behat.org/quick_intro.html]

[19] Testpyramide: [http://martinfowler.com/bliki/TestPyramid.html]

[20] Video zu Facebook-Engineering: [http://www.facebook.com/video/video.php?v=10100259101684977&oid=9445547199&comments]

[21] Listings zum Artikel: [http://www.linux-magazin.de/static/listings/magazin/2012/08/tdd]

Der Autor

Gunnar Wrobel ist PHP-Entwickler und Teilhaber der Horde LLC, einem Unternehmen, das für PHP-Entwicklung im und um das Horde Project herum gegründet wurde.