mehr zum thema:
http://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/
http://artofunittesting.com/
de.wikipedia.org/wiki/Clean_Code

12 TIPPS FÜR GUTEN UNIT-TEST-CODE: BEST PRACTICES ZUM SCHREIBEN GUTER UNIT-TESTS

der autor

Sven Thiergen
[s.thiergen@itcampus.de]
ist Senior Architekt im Bereich Individuelle Softwareentwicklung bei der itCampus GmbH in Leipzig, einem Tochterunternehmen der Software AG. Er hat langjährige Erfahrung in der Softwareentwicklung; in den letzten Jahren mit dem Schwerpunkt auf agilen Methoden.

Unit-Tests – als unverzichtbares Pendant zum Produktionscode – sind in den letzten Jahren zu einem allgemein akzeptierten Standard geworden. Viele kluge Köpfe haben sich viele kluge Gedanken gemacht, um uns Entwicklern diese Erkenntnis als Mantra unauslöschlich einzuprägen. Doch wie viele kluge Gedanken werden zum eigentlichen Entwickeln von 50, 200 oder 10.000 Unit-Tests aufgewendet? Offensichtlich nicht genug, wie die ganz normalen, profanen Unit-Tests eines Standardprojekts zeigen. Der Grund hierfür sind einerseits fehlende Praxiserfahrung beim Schreiben von Unit-Tests und andererseits falsche Annahmen darüber, was Unit-Tests leisten können und zu welchem Zweck sie daher eingesetzt werden sollten.

Unit-Tests müssen den gleichen Qualitätsstandards wie Produktionscode genügen. Sie sind die Basis des Produktionscodes und die Versicherung für den Entwickler, dass Erweiterungen am Produktionscode nicht zu Fehlern an bestehenden Funktionen führen. Wird der Produktionscode erweitert, muss er leicht lesbar sowie modular und sauber entworfen sein, damit die Erweiterungen in einem wirtschaftlich sinnvollen Rahmen durchgeführt werden können.

Erweiterungen am Produktionscode führen zwangsläufig zu Erweiterungen am Testcode. Das heißt, dieser muss ebenso leicht lesbar sowie modular und sauber entworfen sein, damit die Wirtschaftlichkeit des Projekts nicht langfristig gefährdet wird.

Zum Schreiben guter Unit-Tests benötigen Entwickler besondere Erfahrung, denn Unit-Tests sind nicht gleich Produktionscode – für sie gelten zum Teil andere Regeln. Die folgenden Tipps sollen helfen, von Anfang an gute Unit-Tests zu entwerfen, die schnell mit dem Produktionscode wachsen können.

Tipp 1: Einheitliche Namensgebung

An die Namen von Unit-Tests werden besondere Anforderungen gestellt. Abgesehen von den Unit-Tests, die Sie gerade bearbeiten, interessieren Sie sich für einen Unit-Test vor allem dann, wenn er fehlschlägt. Sie wollen dann schnell wissen:

- Was wird getestet (das Testsubjekt)?
- Was sind die Testumstände (das Szenario)?
- Was ist das erwartete Resultat?

Ohne den Code zu untersuchen, stehen Ihnen als Informationen zunächst nur der Testklassen- und der Testmethoden-Name zur Verfügung. Definieren Sie daher zu Beginn des Projekts Namensstandards – insbesondere für die Namen der Methoden –, die den durchgeführten Test beschreiben.

In diesem Artikel wird der Standard *Subject_Scenario_Result* verwendet, der vergleichbar ist mit dem Standard given when then (Listing 1 zeigt ein Beispiel). Vergleichen Sie dazu einen Testmethoden-Namen, wie man ihn ohne besondere Vorgaben vergeben würde (Listing 2).

Tipp 2: Test- und Produktionscode separieren

Zwar gehören Tests untrennbar zum Produktionscode dazu. Lagern Sie die Unit-Tests dennoch in einen separaten Quellcode-Ordner aus oder sogar in ein eigenes Projekt. Das erhöht die Übersichtlichkeit und vereinfacht das Deployment sowie die Auslieferung. Das Kompilieren und Ausführen der Tests auf dem Test-Server (z. B. als *Ant-* oder *Maven-Task*) vereinfacht sich, wenn diese in einem eigenen Ordner liegen. In den ausgelieferten Bibliotheken sollen die Testklassen wiederum üblicherweise nicht enthalten sein.

Des Weiteren benötigen Sie für die Tests häufig eine modifizierte Ausführungsumgebung, beispielsweise eine andere Konfiguration, andere Bibliotheken, andere Zielartefakte wie Testreports usw.

```
@Test
public void PersonSorter_SortByName_PersonsSortedByName() {
  ...
}
```

Listing 1: Beispiel für einen Unit-Test mit Methodenname nach dem Schema „Subject_Scenario_Result“.

```
@Test
public void TestSortByName() {
  ...
}
```

Listing 2: Beispiel für einen Unit-Test mit Standard-Methodennamen.

```
public class TestCleanSetup {

  private Adder adder;

  @Test
  public void Adder_AddPositiveIntegers_Works() {
    adder = new AdderImpl();
    int sum = adder.add(5, 3);
    Assert.assertEquals(8, sum);
  }

  @Test
  public void Adder_AddNullIntegers_Works() {
    int sum = adder.add(5, 0);
    Assert.assertEquals(5, sum);
  }
}
```

Listing 3: *Beispiel für Unit-Tests, die ihre Umgebung nicht vollständig selbst initialisieren.*

Tipp 3: Tests autark schreiben

Diesen Tipp hat sicher jeder schon einmal gehört oder gelesen. Aber da er allzu gern umgangen, ignoriert oder aus „Zeitgründen" vernachlässigt wird, sei hier doch noch einmal explizit daran erinnert: Schreiben Sie Unit-Tests so, dass sie autark laufen.

Schreiben Sie keine Unit-Tests, die direkt als Voraussetzung für andere notwendig sind, etwa für die Erzeugung interner Objektstrukturen oder für die Vorbereitung für externe Module (wie Datenbankinhalte). Schreiben Sie ebenso wenig Unit-Tests, die in einer bestimmten Reihenfolge ablaufen müssen, damit sie funktionieren. Unit-Tests dürfen gemeinsamen Basiscode nutzen. Wenn Sie jedoch Unit-Test A löschen oder abändern, müssen die Unit-Tests B, C und D weiterhin unverändert laufen. Das gilt natürlich nicht, wenn Sie Änderungen an einem etwaigen, gemeinsamen Basiscode vornehmen.

Sobald Sie feststellen, dass Abhängigkeiten zwischen Unit-Tests entstehen bzw. dass es schwierig ist, Abhängigkeiten zu vermeiden, betreiben Sie womöglich kein Unit-Testing mehr, sondern gehen darüber hinaus in Richtung Integrationstests.

Tipp 4: Jeder Test soll seine Umgebung selbst initialisieren

Dieser Tipp ergibt sich im Grunde genommen automatisch aus dem vorherigen, Tests autark zu schreiben. Schreiben Sie jeden Unit-Test derart, dass er seine Testumgebung vollständig neu aufsetzt. Das umfasst üblicherweise das Initialisieren der zu testenden Objekte sowie das Vergeben der entsprechenden Eigenschaften.

Damit sparen Sie langfristig Wartungsaufwand. Wenn Sie ein Objekt für mehrere Tests nur einmalig initialisieren, können diese für den Moment funktionieren. Langfristig jedoch kann es durch Änderungen am inneren Verhalten des Objekts oder durch Änderungen an nur einem der betreffenden Tests dazu kommen, dass ein Folgetest einen unsauberen oder unpassenden Zustand des Objekts vorfindet und dass er fehlschlägt.

Die Regel des sauberen Aufsetzens kann einfach umgesetzt werden, indem alle Klassenvariablen im setup()-Teil vor jedem Testlauf mit dem new-Konstruktor neu erzeugt werden bzw. – sofern es sich um primitive Datentypen handelt – auf einen definierten Initialwert gesetzt werden. Ein Beispiel für eine nicht sauber aufgesetzte Testumgebung zeigt das Codefragment in Listing 3 (der interessierte Leser möge den Fehler selbst suchen). In der zweiten Testmethode wird das Feld adder nicht initialisiert. Wenn also der erste Test nicht vor dem zweiten läuft oder wenn der erste Test gelöscht wird, schlägt der zweite Test mit einer NullPointerException fehl.

Tipp 5: Test-Setup-Code auslagern

Unit-Tests befassen sich zwangsläufig mit einem Setup. Objekte werden in bestimmten Zuständen getestet – dazu müssen Sie die Objekte in diesen Zustand bringen. Schnell jedoch kann der Setup-Code am Beginn der Testmethode den eigentlichen Sinn derselben überdecken. Vergleichen Sie dazu die beiden Testvarianten in Listing 4 und Listing 5, die fachlich dieselbe Funktionalität sicherstellen. In der Variante B) wurde der komplette Initialisierungscode in die Methode setup() ausgelagert. Diese Initialisierungsmethode wird durch die Assertion @Before zu Beginn jedes Tests ausgeführt. In den Tests selber werden nur die Variablen gesetzt, die für den Test relevant sind.

Der erste Test FooTest_BarIs5_IsValid() setzt explizit einzig bar auf den Wert 5. Das ist an sich unnötig, da setup() auf eben diesen Wert initialisiert. Es erhöht jedoch die Lesbarkeit, weil der Testmethoden-Name eine solche Belegung suggeriert.

Dieses Vorgehen ist in vielen Fällen empfehlenswert: Initialisieren Sie die zu testenden Objekte in separaten Methoden vollständig (mit sinnvollen Vorbelegungen) und setzen Sie in den eigentlichen Testmethoden nur die Objekteigenschaften, auf die sich der Test explizit bezieht.

Tipp 6: Kein falscher Alarm

In Projekten mit hoher Testabdeckung – also guten Projekten – sind für den Entwickler nur wenige Dinge so frustrierend wie grundlos fehlschlagende Unit-Tests, oder genauer fehlschlagende Unit-Tests, die dem Entwickler nicht dabei helfen, tatsächliche Fehler in der Software zu finden, sondern die aus irgendeinem anderen Grund fehlschlagen.

Wenn Sie nach einer zeitintensiven Fehlersuche feststellen, dass im Produktivcode alles in Ordnung ist, der Alarm also nichts zur Behebung eines echten Problems in der Software beigetragen, sondern einem nur die Zeit gestohlen hat, schauen Sie sich das wahrscheinlich maximal zwei oder drei Mal an.

Haben Sie den Unit-Test bis dahin nicht stabilisiert (die beste Lösung) oder gelöscht

```
public class FooTestBad {
  @Test
  public void FooTest_BarIs5_IsValid() {
    Foo foo = new Foo();
    foo.setBaz("abcde");
    foo.createHighLow();
    foo.getHighLow().setHigh(3);
    foo.getHighLow().setLow(2);
    foo.setBar(5);
    // ... another 20 lines of initialization
    foo.calculate();
    Assert.assertTrue(foo.isValid());
  }
  @Test
  public void FooTest_BarIs7_IsNotValid() {
    Foo foo = new Foo();
    foo.setBaz("abcde");
    foo.createHighLow();
    foo.getHighLow().setHigh(3);
    foo.getHighLow().setLow(2);
    foo.setBar(7);
    // ... another 20 lines of initialization
    foo.calculate();
    Assert.assertFalse(foo.isValid());
  }
}
```

Listing 4: *Beispiel für Unit-Tests mit zu viel Initialisierungscode in den Testmethoden.*

```
public class FooTestGood {
  private Foo foo;
  @Before
  public void setup() {
    foo = new Foo();
    foo.setBaz("abcde");
    foo.createHighLow();
    foo.getHighLow().setHigh(3);
    foo.getHighLow().setLow(2);
    foo.setBar(5);
    // ... another 20 lines of initialization
  }
  @Test
  public void FooTest_Bar5_IsValid() {
    foo.setBar(5);
    foo.calculate();
    Assert.assertTrue(foo.isValid());
  }
  @Test
  public void FooTest_Bar7_IsNotValid() {
    foo.setBar(7);
    foo.calculate();
    Assert.assertFalse(foo.isValid());
  }
}
```

Listing 5: *Beispiel für Unit-Tests mit ausgelagertem Initialisierungscode.*

(schlechte Lösung), dann werden Sie zumindest diesen spezifischen Unit-Test ab dann ignorieren (schlechteste Lösung). Schlagen auch noch Unit-Test Y und Z „immer mal" fehl, ist es nicht mehr weit, und fehlschlagende Tests werden generell nicht mehr ernstgenommen. Und wenn sie nicht ernstgenommen werden, muss man sie ja auch nicht (mehr) schreiben.

Folgende Warnsignale deuten auf unsaubere, nicht autarke Unit-Tests hin:

- Unit-Tests schlagen sporadisch fehl, mutmaßlich weil die Ausführung im Vergleich zu anderen Unit-Tests um Millisekunden früher oder später erfolgt.
- Unit-Tests schlagen fehl, ohne dass Sie das Verhalten der getesteten Komponente nach außen hin geändert haben.
- Sie ändern Unit-Test A und B, worauf-hin C oder D fehlschlagen.

Stabilisieren Sie solche Unit-Tests umgehend und ersparen Sie sich und Ihren Kollegen unnötige Aufwände, Frust oder gar ein gänzliches Infragestellen der „ganzen Unit-Testerei".

Tipp 7: Keinen 3rd-Party-Code testen

Hin und wieder meint es der Entwickler zu gut und testet auch Code, der gar nicht Bestandteil des Projekts ist. In einfachen Fällen sind die Folgen verschmerzbar. Der unnötige Test ist schnell geschrieben – und ebenso schnell wieder gelöscht (**Listing 6**).

Dass die Klasse ArrayList des *Java Development Kits (JDKs)* die add()-Methode korrekt implementiert, kann sich jeder vorstellen. Trotzdem finden sich derartige „Tests" hin und wieder in Projekten. In jedem Fall sind sie unnötig. Weitaus ineffizienter sind Tests von 3rd-Party-Code, in die viel Zeit gesteckt wird und in denen nicht so schnell überblickt werden kann, welche Tests den eigenen Code betreffen und welche den Code von Drittanbietern (3rd-Party-Code).

Hat der 3rd-Party-Code keine eigenen Unit-Tests, verwenden Sie diese Bibliothek – wenn möglich – nicht. Ist es ein Open-Source-Projekt, können Sie die Unit-Tests dort hinzufügen. Fügen Sie diese jedoch keinesfalls dem eigenen Projekt hinzu. Kurz und knapp: 3rd-Party-Code hat eigene Unit-Tests (zu haben)!

Tipp 8: Datenbanktests sind Integrationstests

Oft wird für „Unit-Tests", die die ORM-Schicht und die Business-Service-Schicht betreffen, die Datenbank verwendet. Der ORM-Kontext wird zu Beginn der Unit-Tests hochgefahren (z. B. Lesen der Hibernate-Mappings, Bereitstellung der Datenquelle). Viel Zeit wird in die Konzeption der Testdatenbasis investiert und ganze Mini-Projekte werden für das initiale Befüllen der Test-Datenbasis und die nachfolgende Pflege aufgesetzt.

In Großprojekten gibt es komplexe Richtlinien, welches Team welche Testdaten-Menge hat, unter welchen Umständen es diese erweitern kann und wie es sicherstellt, dass die Testdaten-Basis nach jedem Testdurchlauf wieder einen definierten Ausgangszustand hat. Unterm Strich funktioniert dieser Prozess am Ende oft eher weniger als mehr – in jedem Fall ist er mit einem als belastend und ineffizient empfundenen Verwaltungsaufwand verbunden.

Hinzu kommt die Komplexität der Synchronisierung der Testdaten-Basis in der Entwicklungsumgebung und im Testsystem.

Klingt ziemlich kompliziert für ein kleinen „Unit-Test"? Ist es auch. Tatsächlich geht es hier nicht um Unit-Tests, sondern um Integrationstests. Das Hinzuziehen eines komplexen Systems wie einer Datenbank ist die Integration von Projektkomponenten.

Die *Data Access Objects (DAOs)* bzw. deren Methoden und Ergebnisse sollte man daher immer „mocken", z. B. mit Mocking-Frameworks wie Mockito, PowerMock, EasyMock etc.

Der eine oder andere schreckt anfangs womöglich vor dem Mocking-Aufwand zurück, denn bei Verwendung der Datenbank hätte man ja „alles schon". Den einmal investierten Aufwand hat man jedoch schnell wieder wettgemacht, da Mocking-Tests langfristig gesehen wesentlich weniger Wartung benötigen als Tests mit einer echten Datenbank.

Des Weiteren fragen Sie sich womöglich, wozu überhaupt testen, wenn man ohnehin definierte Mock-Werte zurückgibt. Worin besteht denn die ursprüngliche Testintention? Darin, einen Wert in der Datenbank zu speichern? Oder ein Objekt im ORM-Framework anzulegen und es zu speichern (gegebenfalls mit Transaktionen) und dann zu schauen, ob der Wert oder das Objekt korrekt in der Datenbank angelegt sind? Dann macht das Ganze mit gemockten Datenbankobjekt in der Tat kaum Sinn.

Der fehlende Sinn hat jedoch seine Ursache darin, dass 3rd-Party-Funktionalität getestet wird – nämlich die des

```
@Test
public void Array_AddElement_SizeIsOne() {
  ArrayList<String> list = new ArrayList<String>();
  list.add("foo");
  Assert.assertEquals(1, list.size());
}
```

Listing 6: *Beispiel für einen Unit-Test, der externen Code testet.*

```
public class TestAdder {
  @Test
  public void testSum() {
    Adder adder = new AdderImpl();
    // can it add positive numbers?
    assert (adder.add(1, 1) == 2);
    assert (adder.add(1, 2) == 3);
    assert (adder.add(2, 2) == 4);
    // is zero neutral?
    assert (adder.add(0, 0) == 0);
    // can it add negative numbers?
    assert (adder.add(-1, -2) == -3);
    // can it add a positive and a negative?
    assert (adder.add(-1, 1) == 0);
    // how about larger numbers?
    assert (adder.add(1234, 988) == 2222);
  }
}
```

Listing 7: Beispiel für eine Testmethode, die mehr als eine Funktionalität testet.

ORM-Frameworks und der Datenbank. Verlässt man sich darauf, dass beide ihre Aufgabe wie definiert erfüllen, kann man sich auf die Tests des eigenen Codes konzentrieren. Alle diese Betrachtungen gelten übrigens ebenso für Web-Services.

Tipp 9: In-Memory-Datenbanken verwenden

Erwägen Sie die Verwendung von In-Memory-Datenbanken für Integrationstests mit Datenbanken, die ein komplettes Setup des Datenbestandes beinhalten. Die Tests laufen wesentlich schneller und durch die Eigenschaften einer In-Memory-Datenbank (keine Persistierung) wird das Team zu einem sauberen Setup der Datenbasis zu Beginn der Tests gezwungen.

Tipp 10: Nur ein funktionaler Test pro Unit-Test

Getreu dem Clean-Code-Prinzip soll ein Unit-Test genau eine Sache tun und nicht mehrere. Widmet sich ein Unit-Test mehreren Funktionen, steigt für Sie die Komplexität beim Lesen des Codes – sowohl aus technischer als auch aus fachlicher Sicht.

Das Schreiben eines separaten Unit-Tests für jede Funktion bringt zwar auch (etwas) mehr Schreibarbeit mit sich, andererseits haben Sie dadurch die Möglichkeit, den verschiedenen Testmethoden wesentlich treffendere Namen zu geben. Die Gesamtheit der Unit-Tests wird dadurch zum einem Teil der Softwaredokumentation.

Das klingt (zu) trivial? Dann schauen wir uns doch einmal gemeinsam auf Wikipedia um (vgl. http://en.wikipedia.org/wiki/Unit_testing), wo das in Listing 7 gezeigte Codebeispiel zu finden ist. Hierbei handelt es sich um ein typisches Beispiel für einen Alles-Tester. Die Kommentare deuten schon in die Richtung, dass es um verschiedene Anforderungen an die Summenfunktion geht, die durch Teilfunktionen der Software abgebildet werden. Die Summenfunktion soll für folgende Zahlen funktionieren:

- Positive Zahlen
- Null(en)
- Negative Zahlen
- Eine negative und eine positive Zahl
- Große Zahlen

Somit kann man den einen großen Test, wie in Listing 8 dargestellt, in fünf kleine Tests zerlegen und spezifischere Namen vergeben.

Tipp 11: Testen privater Methoden

Private Methoden sind ein schwieriger Fall, was das Testen betrifft.

Einerseits werden sie nie direkt aufgerufen und dienen nur indirekt dazu, eine Funktionalität sicherzustellen. Nur die public-Methoden liefern Funktionalität, die Sie benötigen. Wenn Sie dies absichern, sollte das reichen.

In der Praxis werden Sie jedoch oft feststellen, dass interessante Details in privaten, geschützten Methoden liegen und dass es eine viel direktere und natürlichere Herangehensweise wäre, die privaten Methoden selbst mit Tests zu versehen. Öffentliche Methoden haben oft ein zu hohes Abstraktionsniveau.

Am besten lösen Sie dieses Problem dadurch, dass Sie mittels Refaktorisierung

```
public class TestAdderPartioned {
  private Adder adder;
  @Before
  public void setup() {
    adder = new AdderImpl();
  }
  @Test
  public void Adder_AddPositiveIntegers_Works() {
    Assert.assertEquals(2, adder.add(1, 1));
    Assert.assertEquals(3, adder.add(1, 2));
    Assert.assertEquals(4, adder.add(2, 2));
  }
  @Test
  public void Adder_AddZeroIntegers_Works() {
    Assert.assertEquals(0, adder.add(0, 0));
  }
  @Test
  public void Adder_AddNegativeIntegers_Works() {
    Assert.assertEquals(-3, adder.add(-1, -2));
  }
  @Test
  public void Adder_AddPositiveAndNegativeInteger_Works() {
    Assert.assertEquals(2, adder.add(4, -2));
  }
  @Test
  public void Adder_AddLargeIntegers_Works() {
    Assert.assertEquals(2222, adder.add(1234, 988));
  }
}
```

Listing 8: Beispiel für mehrere Testmethoden, wobei jede nur eine Funktionalität testet.

```
public static String getMD5(String clearText) {
  try {
    MessageDigest md = MessageDigest.getInstance("MD5");
    md.update(clearText.getBytes());
    return byteArrayToHex(md.digest());
  } catch (NoSuchAlgorithmException nsa) {
    return "";
  }
}
private static String byteArrayToHex(byte hashBytes[]) {
  StringBuffer sb = new StringBuffer(hashBytes.length * 2);
  for (byte hashByte : hashBytes) {
    if (firstBytePositionIsZero(hashByte)) {
      sb.append("0");
    }
    sb.append(byteToHex(hashByte));
  }
    return sb.toString();
}
private static String byteToHex(byte theByte) {
  return Long.toString(theByte & 0xFF, 16);
}
private static boolean firstBytePositionIsZero(byte theByte) {
  return (theByte & 0xFF) < 0x10;
```

Listing 9: *Beispiel für eine zu testende, private Methode.*

die vormals geschützten Implementierungsdetails in einen kleineren Codeblock auslagern und als eigenständige, öffentliche Funktionen dieses Codeblocks zur Verfügung stellen und damit auch Unit-testbar machen.

Als Beispiel hier ein Fall aus meinem Projektalltag: Die öffentlich zugängliche Funktion war String getMD5(), die den MD5-Wert eines Eingabe-Strings in hexadezimaler, lesbarer Form zurückgeben sollte (**Listing 9**).

Die Hauptmethode getMD5() war für Testzwecke nicht relevant, da die verwendete Klasse MessageDigest 3rd-Party-Code ist. Die tatsächlich abzusichernde Methode war byteArrayToHex(), die ein reines Byte-Array in einen Hex-String wandelt (jedes Byte als zweistelliger Hex-String, d. h. mit führender Null, wenn nötig).

Die Lösung ist im Grunde einfach: Das Wandeln eines Byte-Arrays in einen Hex-String ist eine reine StringUtil-Funktion und kann als solche in die entsprechende Klasse als öffentliche Funktion verschoben werden.

Von Lösungen wie *Reflection* rate ich im Übrigen dringend ab. Zwar beeinflusst dies nicht den Produktionscode, aber es ist eine sehr umständliche und wartungsanfällige Lösung.

Tipp 12: Unit-Tests treiben die Funktionalität voran

Abschließend noch eine Bemerkung zum Sinn der Unit-Tests: Unit-Tests sind (auch) dazu da, um Fehler zu finden. Und einige naheliegende Extremfälle sollen durch Unit-Tests abgeprüft werden.

Hauptsächlich jedoch stellen Unit-Tests sicher, dass jede Funktionseinheit für sich auf einer atomaren Ebene eine bestimmte Funktionalität bereitstellt. Unit-Tests sollen als treibende Kraft bei der Umsetzung der Funktionalität und nicht als abwehrende Kraft zur Fehlervermeidung verstanden werden. Eine ganz konsequente Vorgehensweise ist hier die testgetriebene Entwicklung (*Test Driven Development*): Der Test wird zuerst geschrieben und bezieht sich auf eine noch nicht implementierte Funktionalität, die der Unit-Test *verlangt*. Durch die tatsächliche Implementierung erfüllt der Produktionscode die Anforderungen des Testcodes. Der Testcode wird bei diesem Vorgehen zum *Contract*, den der Produktionscode erfüllen muss.

Wenn Sie in Ihrer beruflichen Praxis keine Gelegenheit haben, testgetrieben zu entwickeln, dann tun Sie dies unbedingt einige Male privat. Es ist eine der lohnenswertesten Erfahrungen, die Sie als Entwickler sammeln können.

Sie können danach immer noch entscheiden, ob Sie ab jetzt nur noch testgetrieben entwickeln wollen. Aber wie sich der Fokus beim Schreiben der Tests auf die eigentliche, umzusetzende Funktionalität verschiebt und wie das Nachdenken über mögliche Fehler in den Hintergrund rückt, bemerkt man häufig erst, wenn man testgetriebenes Entwickeln selbst ausprobiert hat.

Zum Schluss

Die Tipps und Best Practices in diesem Artikel resultieren aus meinen Erfahrungen in der Umsetzung zahlreicher Projekte sowie aus den fruchtbaren Diskussionen mit anderen Software-Ingenieuren. Einen Anspruch auf absolute Wahrheit erhebt der Artikel nicht. Wenn die Tipps helfen, Denkanstöße zu geben und Diskussionen in Gang zu setzen, wie Sie Unit-Tests für Ihr Projekt möglichst effektiv schreiben können, freue ich mich sehr. ■