Testgetriebene Entwicklung: Entschlüsselt

Wie man Code und Design kontrolliert weiterentwickelt

Johannes Link

#tdddecrypted

Testgetriebene Entwicklung: Entschlüsselt

Wie man Code und Design kontrolliert weiterentwickelt

Johannes Link

Dieses Buch können Sie hier kaufen http://leanpub.com/tdd-entschluesselt

Diese Version wurde auf 2015-02-05 veröffentlicht

Twittern Sie dieses Buch!

Bitte unterstützen Sie Johannes Link, indem Sie dieses Buch auf Twitter weiterempfehlen!

Vorschlag: Verwenden Sie den folgenden Hashtag, wenn Sie über dieses Buch twittern: #tdddecrypted.

Was sagen Andere über dieses Buch? Klicken Sie hier, um nach diesem Hashtag auf Twitter zu suchen:

https://twitter.com/search?q=#tdddecrypted

Diese Buch wurde ermöglicht durch die Unterstützer meiner Startnext-Kampagne: Görge Albrecht, Ruben Artus, Achim Bangert, Florian Bugiel, RA Malte T. Burchard, Jakob Bysewski, Catalysts GmbH, Jens Coldewey, Roger Cugini, Carsten Drossel, Christoph Ertl, Jan Fellien, Andy Fischer, Thomas Gauweiler, Thorsten Göckeler, Vanessa Hagge, Benjamin Haupt, Heidelberg Mobil International GmbH, Chrisitian Heiss, Markus Holland-Letz, Florian Hopf, Jerko Horvat, Daniel Hunziker, IDOS AE GmbH, Marco Klemm, Martin Klose, Markus Knittig, Andreas Kumlehn, Udo Langer, Helmut und Ellen Link, Michel Löhr, Sabine Lorenz, Frank Lux, Stefan Mallmann, Krishan Mathis, Sandra Rebholz, Timm Reinstorf, Thilo Riegel und Sylvie Dénarié, Philipp Riemer, Roland Sand, Claude Schmidhuber, Niko Schmuck, Christian Schubert, Marko Schulz, Tomi Schütz, Christoph Stock, Swen Stötzer, Frank Conrad Strauss, Conrad Thukral, Klaus Unger, Alexander Volk, Holger Voormann, Felix Wenz

Mein herzlicher Dank gilt außerdem den Reviewern: Patrick Koglin, David Völkel

Inhaltsverzeichnis

Vorwort

Testgetriebene Entwicklung (engl. Test Driven Development, *TDD*) ist vermutlich die meist zitierte Praktik des *Extreme Programming*. Am Sinn oder Unsinn dieser Programmiertechnik scheiden sich die Geister. Während die einen TDD für unverzichtbar halten, um dauerhaft wartbare Software zu entwickeln, sehen andere darin ein umständliches, unintuitives und seinen Zweck verfehlendes Vorgehen.

Mein Ziel ist es nicht, diese Sinnfrage allgemein zu klären, sondern Entwicklern, die TDD grundsätzlich positiv gegenüberstehen, dabei zu helfen, diese Frage für sich selbst, für ihr Team und für ihren derzeitigen Kontext zu beantworten. Dazu gehören auch die Aufdeckung von Fallstricken, typischen Fehlern, notwendige Abwägungen und die Warnung vor Situationen, in denen testgetriebene Entwicklung kontraproduktiv wirken kann.

Zielpublikum

Dieses Buch ist kein Buch für den TDD-Anfänger, der nach einem einfachen Rezept für die Erstellung perfekter Software sucht. Zwar werden auch die theoretischen Grundlagen der testgetriebenen Entwicklung im Detail behandelt, dafür werden andere typische Anfängerfragen, wie etwa nach der Installation und Benutzung eines bestimmten Testframeworks, gar nicht berührt oder lediglich angerissen.

Das Buch zielt auf Leserinnen und Leser mit solidem Wissen in Programmierung, Design und Architektur, die bereit sind, in Codebeispiele auch dann einzutauchen, wenn diese nicht in ihrer Lieblingssprache verfasst wurden. Es möchte dem TDD-Anfänger die Möglichkeit geben, den Sinn hinter den Regeln zu verstehen. Der Praktiker wird erfahren, dass auch beim Einsatz testgetriebener Entwicklung unterschiedliche Ziele unterschiedliches Vorgehen erfordern. Und der erfahrene TDDler soll Intuition dafür entwickeln, wann die vorgestellten Prinzipien hilfreich sind, wann sie angepasst werden müssen und wann sie im Wege stehen.

Codebeispiele

Die meisten Beispiele sind in einer (überwiegend) typisierten Variante der JVM-Sprache Groovy verfasst. Damit wird der Quellcode deutlich knapper als beispielsweise in Java und ist dennoch für den “normalen” Programmierer gut lesbar. Ungewöhnlichere Fähigkeiten von Groovy, die über den Überblick in Anhang A hinausgehen, setze ich nur ein, wenn es für ein Thema wesentlich ist; dann selbstverständlich mit ausführlicher Erklärung.

Als Testframework kommt in den meisten Beispielen JUnit zum Einsatz; auch hier verzichte ich auf esoterische Eigenschaften. Andere Bibliotheken, z.B. für die Erzeugung von Test-Doubles, kommen bei Bedarf hinzu und werden an Ort und Stelle erklärt. Eine kompakte Einführung in JUnit findet sich in Anhang B.

Kryptographie

Alle Aufgaben und Codebeispiele stammen aus dem Gebiet der Kryptographie. Dieses Thema ist nicht nur hochaktuell und spannend, sondern auch so vielschichtig, dass sich für jeden Aspekt der testgetriebenen Entwicklung eine passende Anwendung aus dem Reich der Verschlüsselung findet.

Aber keine Angst! Das für die Beispiele nötige Wissen, wird im Buch vermittelt; die vorherige Lektüre eines Buchs über Kryptographie ist nicht notwendig, im Anschluss allerdings sehr empfohlen.

Erfahrung der langjährigen Praktiker

Im Laufe der Recherchen für dieses Buch habe ich einige “alte Hasen” der testgetriebenen Entwicklung befragt, die zum Teil schon seit mehr als 15 Jahren TDD einsetzen. Abschriften der Interviews habe ich ins Buch eingestreut - und meine Erkenntnisse aus diesen Gesprächen natürlich auch in die “normalen” Kapitel mit eingearbeitet.

Feedback und Kommentare

Ich freue mich über jedes Lob und über alle konstruktiven Kommentare - alle anderen ignoriere ich einfach. Einfach über Twitter (@johanneslink #tdddecrypted) oder per E-Mail an business@johanneslink.net.

Einführung

Der erste Teil des Buches besteht aus drei kurzen Kapiteln. Eine Einführung in die Grundideen der testgetriebenen Entwicklung wird gefolgt von einem Überblick über Kryptographie. Die ersten beiden Interviews mit TDD-Veteranen schließen diesen Buchteil ab. So gesehen handelt es sich um eine Auswahl an Kapiteltypen, die dem Leser einen Vorgeschmack auf den Stil des Buches geben können.

1. Testgetrieben in 10 Minuten

Laut Kent Beck, dem Erfinder bzw. Wiederentdecker des Test Driven Development besteht TDD aus drei Teilen (siehe Beck 2010):

- Entwickler schreiben automatisierte Tests während sie programmieren.
- Die Tests werden vor dem zugehörigen Produktionscode geschrieben.
- Design findet in kleinen Schritten statt.

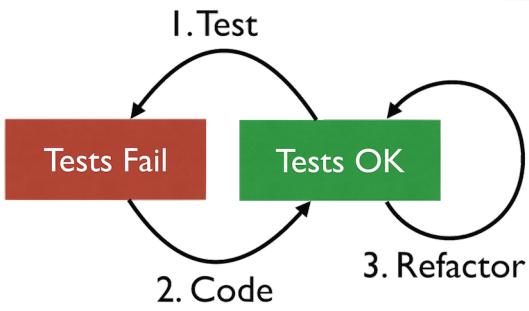
Schauen wir uns diese drei Punkte nacheinander genauer an:

Dass **automatisierte Entwicklertests** unabdingbar sind, haben die meisten Entwickler im Laufe des vergangenen Jahrzehnts gelernt. Viele haben auch erfahren, dass diese Tests manchmal unerwartet Probleme aufzeigen und so deren Beseitigung erst möglich machen. Aus Management-Sicht ist eine umfangreiche Testautomatisierung – oft unter dem Stichwort *Regressionstests* – eine Investition, die sich durch einen geringeren manuellen Testaufwand und schnellere aussagekräftige Testergebnisse auszahlt.

Der zweite Bestandteil testgetriebener Entwicklung wird schon wesentlich kontroverser diskutiert. Warum sollte es einen Unterschied machen, ob ich meine Tests **vor oder nach** dem Produktionscode erstelle? Um die spätere Diskussion zusammenzufassen: In der Praxis macht es einen erheblichen Unterschied. Nachträgliche Tests werden häufiger weggelassen. Darüber hinaus ist Produktionscode, der ohne Rücksicht auf Testbarkeit entworfen wurde, oft schwer testbar und führt zu aufwändigerem Testcode. Entscheidend ist jedoch der Einfluss von frühzeitig erstellten Tests auf das Schnittstellendesign des Codes: Der Testcode dient als erster Verwender des Produktionscodes und gibt daher direkt Feedback über die Adäquatheit der Schnittstelle.

Und schließlich ist da noch der Gedanke, **Design in kleinen Schritten** vorzunehmen, und nicht nur einmal zu Beginn des Entwicklungsprojekts (siehe Kapitel xx). Inkrementell und evolutionär das Design entstehen zu lassen, ist das Gegenteil des Draufloshackens und erfordert diszipliniertes Arbeiten. Das Ziel ist es, Designentscheidungen so spät wie möglich zu treffen, denn später haben wir mehr Wissen. So entwerfen wir eine Schnittstelle dann, wenn wir sie das erste mal benötigen; nämlich beim Schreiben des ersten betroffenen Testfalls. Und wir passen das Design dann an, wenn ein weiterer Testfall uns zusätzliche Informationen liefert. *In kleinen Schritten* designen heißt eben auch *ständig* designen.

1.1 Mechanik



Test-Code-Refactor-Zyklus

Die innere Mechanik der testgetriebenen Entwicklung, beschreibt der *Test-Code-Refactor-Zyklus*. Ausgangspunkt ist immer ein Zustand, in dem alle existierenden Testfälle erfolgreich durchlaufen. Nur von hier aus begeben wir uns in die Weiterentwicklung und starten den nächsten Zyklus. Dieser besteht aus drei Schritten:

1. Schreibe einen **Testfall**, der, sobald er ausgeführt wird, fehlschlägt. In diesem Schritt können sowohl existierende Schnittstellen, Funktionen und Objekte verwendet werden, oder neue geschaffen werden, die es geben sollte.
2. Bringe den Test zum Erfolg, indem du den **Produktionscode** mit **minimaler Implementierung** dazu bringst, das im Test geforderte Verhalten zu erfüllen.
3. Nun räume das Programm auf – und zwar nicht nur den neuen, sondern auch den bereits vorhandenen Code. Dieser Schritt wird typischerweise der **Refactor**-Schritt genannt, weil hier die Idee des Refaktorisierens zur Anwendung kommt: Verbessere den existierenden Code und das existierende Design ohne dabei das nach außen sichtbare Verhalten zu verändern.

Nun beginnt der Zyklus **wieder von vorne** mit einem neuen Test. Dies geschieht solange, bis für die vollständige Erfüllung der aktuelle Aufgabe kein neuer Testfall mehr notwendig ist.

Ein solcher Mikro-Zyklus dauert häufig nur wenige Minuten, nämlich immer dann, wenn weder die fachliche Anforderung noch die technische Umsetzung große Überraschungen bereithalten. Dies bedeutet, dass ein in TDD geübter Entwickler in der Lage sein muss, umzusetzende Features in sehr kleine Stücke zu zerschneiden.

1.2 Erste Praxis

Anhand eines einfachen Beispiels möchte ich diese Mechanik verdeutlichen. Greifen wir Kapitel 2 etwas vor und implementieren einen einfachen *monoalphabetischen Substitutionschiffre* auf testgetriebene Weise. Wir beginnen - natürlich - mit einem Testfall:

```
class ReplacementCipherTest {
    @Test
    void emptyTextIsEncryptedAsEmptyText() {
        def cipher = new ReplacementCipher([:])
        assert cipher.encrypt('') == ''
    }
}
```

Aus funktionaler Sicht ist der Test sehr schwach, da er lediglich die Verschlüsselung eines leeren Klartexts in einen leeren Schlüsseltext spezifiziert. Allerdings mussten wir bei der Formulierung des Testcodes *Schnittstellen-Design* betreiben. Zu diesem Zeitpunkt arbeiten wir im "Wunsch-Modus", d.h., wir entwerfen die Schnittstelle so, wie wir sie als Verwender gerne hätten. Es ist daher nicht verwunderlich, dass sich große Teile nicht (statisch) kompilieren lassen; weder existiert die Klasse `ReplacementCipher` noch die `encrypt`-Methode, und auch der postulierte Konstruktor mit `Map`-Parameter der zu verschlüsselnden Buchstaben fehlt. Aber das lässt sich schnell beheben:

```
class ReplacementCipher {
  ReplacementCipher(Map replacements) {}
  def encrypt(clearText) {}
}
```

Nun können wir den Test anwerfen und das erwartete Ergebnis genießen:

```
Assertion failed:
assert cipher.encrypt('') == ''
       |      |
       |      null
       ReplacementCipher@1661db97
```

Genießen? Jawohl, schließlich haben wir Schritt 1 des Test-Code-Refactor-Zyklus' erfolgreich durchlaufen: Wir haben einen Testfall erstellt, der uns zeigt, welches kleine Stück Funktionalität wir als nächstes programmieren wollen.

Schritt 2, den Test mit minimalen Änderungen im Applikationscode zu erfüllen, ist jetzt ganz einfach:

```
class ReplacementCipher...
  def encrypt(clearText) {
```

```
      return ''
```

```
  }
```

Die Änderung war trivial und wir sehen dem Code an, dass das noch nicht der Endzustand ist, wollen aber auch keine Logik hinzufügen, die nicht von Tests gefordert wird. Daher bleibt uns in Schritt 3 lediglich, den Parameter- und Rückgabe-Typ zu konkretisieren:

```
class ReplacementCipher...
  String encrypt(String clearText) {...}
```

Zweiter Durchgang

Stürzen wir uns nun auf den zweiten Test-Code-Refactor-Durchgang und damit auf einen neuen Testfall:

```
class ReplacementCipherTest...
  @Test
  void replaceSingleMappedCharacter() {
      def cipher = new ReplacementCipher(['a': 'A'])
      assert cipher.encrypt('a') == 'A'
  }
```

Wir nähern uns **mit einem kleinen Schritt** dem Ziel und fordern im Test die Ersetzung eines einzigen Buchstabens. Auch jetzt starten wir selbstverständlich den Test-Runner und sehen unseren Test wunschgemäß fehlschlagen.

Die nächste, möglichst einfache Code-Änderung muss nun nicht nur diese neue Anforderung umsetzen, sondern darf den vorherigen Test nicht versehentlich zerstören:

```
class ReplacementCipher...
  String encrypt(String clearText) {
    if (clearText.isEmpty())
      return ''
    return 'A'
  }
```

Das ist wirklich schnell und schmutzig, bringt uns aber in den *grünen Bereich*, in dem alle Testfälle laufen und der Puls nach unten geht. Im **Refactor-Schritt** können wir nun schon ein klein wenig Abstraktion betreiben und statt des konstanten Rückgabewertes uns den hinterlegten Verschlüsselungswert greifen:

```
class ReplacementCipher...
  private final Map replacements
  ReplacementCipher(Map replacements) {
    this.replacements = replacements
  }
  String encrypt(String clearText) {
    if (clearText.isEmpty())
      return ''
    return replacements.get(clearText[0])
  }
```

Dass das ein Schritt in die richtige Richtung ist, erkennen wir, wenn wir den Testfall leicht verändern und damit zeigen, dass der Applikationscode nicht mehr ganz so stark von den konkreten Testdaten abhängt:

```
class ReplacementCipherTest...
  @Test
  void replaceSingleMappedCharacter() {
    def cipher = new ReplacementCipher(['z': 'X'])
    assert cipher.encrypt('z') == 'X'
  }
```

Dritter Durchgang

Bohren wir den Test so auf, dass er nicht nur das erste Vorkommen, sondern alle Vorkommen eines Buchstabens ersetzt:

```
class ReplacementCipherTest...
  @Test
  void replaceSingleMappedCharacterEverywhere() {
    def cipher = new ReplacementCipher(['z': 'X'])
    assert cipher.encrypt('z') == 'X'
    assert cipher.encrypt('zzz') == 'XXX'
  }
```

Neben der zusätzlichen Zeile, haben wir auch noch den Test umbenannt, um Intention und Umfang klarer zu beschreiben. Auch die Änderung im Anwendungscode ist etwas umfangreicher:

```
class ReplacementCipher...
  String encrypt(String clearText) {
    String encrypted = ""
    for(int i = 0; i < clearText.size(); i++) {
      encrypted += replacements.get(clearText[0])
    }
    return encrypted
  }
```

Das Kernstück der Logik hat jedoch noch überlebt:

```
replacements.get(clearText[0])
```

Schlussrunde

Und erst ein weiterer - zunächst fehlschlagender - Test wird uns zur entscheidenden Verallgemeinerung zwingen:

```
class ReplacementCipherTest...
  @Test
  void replaceAllMappedCharactersEverywhere() {
    def cipher = new ReplacementCipher(['a': 'X', 'b': 'Y'])
    assert cipher.encrypt('ba') == 'YX'
    assert cipher.encrypt('babbaa') == 'YXYYXX'
  }
```

Um diesen Testfall zu erfüllen, müssen wir aus

```
encrypted += replacements.get(clearText[0])
```

ein

```
encrypted += replacements.get(clearText[i])
```

machen. Und das wiederum können wir im abschließenden Refactoring-Schritt ein wenig idiomatischer gestalten:

```
class ReplacementCipher...
  String encrypt(String clearText) {
    String encrypted = ""
    for(letter in clearText) {
      encrypted += replacements.get(letter)
    }
    return encrypted
  }
```

Fertiger Code des Kapitels

- ReplacementCipher.groovy
- ReplacementCipherTest.groovy

Was haben wir erreicht?

In drei kurzen Durchläufen des Test-Code-Refactor-Zyklus haben wir eine Klasse `ReplacementCipher` ins Leben getestet, die bereits einen Text verschlüsseln kann, und deren augenblickliche Funktionalität durch die drei Testfälle dokumentiert und überprüft wird. Einige offene Enden, wie beispielsweise der Umgang mit unbekannten Buchstaben, existieren noch. Doch diese lassen sich durch eine Handvoll weiterer Test-Code-Refactor-Runden leicht schließen.

1.3 Was testgetriebene Entwicklung sein kann - und was nicht

TDD ist ein Vorgehen, das uns Softwareentwicklern dabei helfen möchte, Code und Design *kontrolliert* und *mit hoher innerer Qualität* zu erstellen und weiterzuentwickeln. Seine Effekte wirken dabei auf zahlreichen Ebenen; teilweise benutzt es die typischen menschlichen Schwächen, teilweise arbeitet es gegen unsere erworbene Intuition. So führt testgetriebene Entwicklung im Idealfall…

- zu einer **hohen Abdeckung** durch automatisierte Unit-Tests,
- zu einer **durchgängig testbaren** Codebasis,
- zu **sparsam gekoppeltem** Design und
- zu Programmeinheiten (engl. units) mit **sinnvollem Namen**.

Oder in drei Worten: zu **dauerhaft wartbarer Software**.

TDD kann jedoch nicht alle Wunder vollbringen, die ihm zuweilen zugeschrieben werden. Insbesondere ist TDD…

- **kein** ausfallsicherer Prozess, um das beste Design oder gar den besten Algorithmus zu entwickeln.
- **keine** Methode, um aus schlechten oder unerfahrenen Programmierern gute Entwickler zu machen.
- **kein** Vorgehen, das die Produktivität eines Softwareteams im Alleingang verdoppelt, verzehnfacht oder auch nur um 10% erhöht.
- **kein** Ersatz für ein gründliches Verständnis der fachlichen Probleme und der sinnvollen Lösungsansätze.

Testgetriebene Entwicklung ist ein äußerst wirksames *taktisches* Konzept. Jede Taktik muss jedoch zur gewählten *Strategie* passen, und nicht jede Strategie findet in TDD den richtigen Partner[1].

[1] Die Frage, wann testgetriebenes Vorgehen nicht zur Strategie passt, wird später in einem eigenen Kapitel diskutiert.

2. Kryptographie in 5 Minuten

Seit Jahrtausenden versuchen Menschen den wahren Inhalt ihrer Nachrichten an andere so zu verstecken, dass der Inhalt nur dem gewünschten Adressaten zugänglich ist. Königreiche sind zu Grunde gegangen, Schlachten gewonnen worden, weil diese Geheimhaltung gelang - oder eben nicht[1].

Die Wissenschaft von der Ver- und Entschlüsselung (engl. *Encryption* und *Decryption*) von Informationen zum Zwecke der Geheimhaltung bezeichnet man als *Kryptographie*. Das Gegenstück stellt die *Kryptoanalyse* dar, das Studium der Methoden, um verschlüsselten Informationen wieder ihre ursprüngliche Bedeutung zu entlocken, auch wenn man das verwendet Verschlüsselungsverfahren bzw. den verwendeten Schlüssel nicht kennt[2].

2.1 Codes und Chiffren

Als *Code*[3] bezeichnet man ein Verfahren, um die Form oder die Darstellung von Informationen zu ändern; so dient der *Morsecode* dazu, die Übertragung von Texten über "dünne" Kanäle - beispielsweise Telegrafenleitungen - zu ermöglichen. Wer immer das Codebuch besitzt, kann die Verwandlung von Text in Morsecode und umgekehrt vornehmen. Andere bekannte Kodierungen sind UTF-8 oder auch alle verlustlosen Kompressionsverfahren wie z.B. Zip und Gzip.

Im Gegensatz zum Code hat eine *Chiffre* (engl. *cipher*), den expliziten Zweck, eine Nachricht zu verschlüsseln, d.h. für einen unbefugten Leser unverständlich zu machen. Eine Chiffre arbeitet typischerweise auf Ebene der Buchstaben, d.h. die Buchstaben des *Klartextes* (engl. *cleartext*) werden durch andere ersetzt, sodass ein chiffrierter Text (engl. *ciphertext*) dabei herauskommt, den nur ein Eingeweihter wieder in den ursprünglichen Klartext zurückverwandeln kann.

Eine bekannte Chiffre ist die *Caesar-Verschiebung*, die jeden Buchstaben einer Nachricht durch einen anderen Buchstaben ersetzt, indem man sich im Alphabet um N Stellen nach vorne bewegt. Kommt man dabei über den letzten Buchstaben hinaus, dann fängt man im Alphabet wieder von vorne an. So wird aus dem Klartext

```
die schlacht beginnt bei morgengrauen
```

und einem N=9 der folgende Chiffretext[4]:

[1] Ein lesenswertes Buch, das den Kampf zwischen "Codemaker" und "Codebreaker" von der Antike bis heute anschaulich schildert ist Singh 2001.

[2] Eine Lernsoftware, um sich mit den gängigen Kryptografie- und Kryptoanalyse-Verfahren vertraut zu machen, ist Cryptool. Cryptool ist frei und in mehreren Sprachen verfügbar.

[3] Umgangssprachlich nennt man oft nur das Ergebnis der Abbildung "Code" und nicht das Verfahren selbst.

[4] Häufig stellt man in Beispielen den Klartext in Kleinbuchstaben und den Chiffretext in Großbuchstaben dar, um Verwechslungen zwischen den beiden zu vermeiden.

```
MRN BLQUJLQC KNPRWWC KNR VXAPNWPAJDNW
```

Die Erkenntnis, dass die Sicherheit eines Verschlüsselungsverfahrens nicht von der Geheimhaltung des verwendeten Algorithmus abhängen sollte, sondern von der Geheimhaltung des Schlüssels, nennt man *Kerckhoffs Prinzip*. Daraus folgt direkt, dass die Anzahl aller möglichen Schlüssel ein wichtiges Kriterium für die Sicherheit einer Chiffre darstellt. So existieren bei der Caesar-Verschiebung in einem Alphabet der Länge 26 lediglich 25 unterschiedliche Schlüssel; dies macht das Verfahren leicht knackbar, da im Schnitt nach nur 13 Entschlüsselungsversuchen der richtige Klartext zum Vorschein kommt.

2.2 Substitution und Transposition

Chiffrier-Verfahren kann man grob in zwei Gruppen aufteilen: Bei *Substitutionsverfahren* werden Buchstaben durch andere ersetzt. Repräsentiert ein Buchstabe des Chiffretextes immer den gleichen Buchstaben des Klartextes, dann spricht man von *monoalphabetischen Substitutionsverfahren*; auch hier ist die Caesar-Verschiebung ein typischer Vertreter. Monoalphabetische Chiffren sind sehr anfällig für ein Grundverfahren der Kryptoanalyse: die Häufigkeitsanalye (engl. *frequency analysis*).

Polyalphabetische[5] Substitution ist für Angriffe durch Häufigkeitsanalyse nicht in gleichem Maße anfällig, da hierbei der gleiche Buchstabe im Chiffretext aus unterschiedlichen Buchstaben im Klartext hervorgehen kann. Beispielsweise benutzt die *Vigenère-Verschlüsselung* die Idee der Caesar-Verschiebung, setzt jedoch auf die periodische Anwendung unterschiedlicher Verschiebungen, die durch ein Codewort spezifiziert werden. Das Vigenère-Verfahren galt lange Zeit als „Le Chiffre indéchiffrable“ bis schließlich Friedrich Wilhelm Kasiski 1863 eine Lösung veröffentlichte.

Neben der Substitution ist die *Transposition* ein wichtiges Element der Verschlüsselung. Von Transposition spricht man, wenn die Position eines Buchstaben im Chiffretext anders ist als im Klartext. Eine einfache Transposition ist z.B. die Umkehrung der Reihenfolge aller Buchstaben; so einfach, dass bereits ein mäßig geübter Leser die Entschlüsselung mühelos im Kopf vornehmen kann. Komplexere Transpositionen lernen wir in Kapitel 5.2 kennen.

Alle bislang besprochenen Chiffre sind Vertreter *symmetrischer* Kryptographie; dies bedeutet, dass Verschlüssler und Entschlüssler denselben geheimen Schlüssel verwenden, über den sie sich zuvor einigen müssen.

Die meisten modernen symmetrischen Chiffren wie DES und AES kombinieren Substitution und Transposition miteinander, was ein Brechen des Codes im Vergleich zu reinen Verfahren schwieriger macht. Im Gegensatz dazu bietet die Mehrfachverschlüsselung mit dem gleichen Verfahren häufig keine höhere Sicherheit. Wenden wir beispielsweise die Caesar-Verschiebung mehrfach mit unterschiedlichen Schlüsseln an - z.B. erst mit 7, dann mit 9 -, so entspricht das exakt einer Verschlüsselung mit 16.

[5] *Poly* (griechisch *viel, mehrere*) sagt aus, dass mehrere unterschiedliche Alphabete für die Abbildung von Klartextbuchstaben auf Chiffrebuchstaben zum Einsatz kommen.

2.3 Die perfekte Chiffre

Während letztendlich auch die Vigenère-Verschlüsselung mittels einer erweiterten Häufigkeitsanalyse geknackt wurde, reifte kurz nach Ende des ersten Weltkriegs die Idee einer perfekten Chiffre heran: Benutzt man für polyalphabetische Substitution **zufällig generierte** Schlüssel der Länge des zu verschlüsselnden Klartextes und setzt man diese Schlüssel auch nur **ein einziges mal** ein, dann ist jeder Versuch, diese Nachricht ohne Besitz des ursprünglich verwendeten Schlüssels zu entziffern, zum Scheitern verurteilt.

Claude Shannon konnte in den 1940er Jahren beweisen, dass dieses *One-Time-Pad* genannte Verfahren tatsächlich informationstheoretisch sicher ist, da jede denkbare Zeichenfolge im Chiffretext gleich wahrscheinlich ist, und sich damit jeder denkbare Klartext der gleichen Länge hinter einer verschlüsselten Nachricht verbergen kann. Das One-Time-Pad-Verfahren betrachten wir in Kapitel XXX noch genauer.

2.4 Moderne Kryptographie und Mathematik

Die moderne Kryptographie geht deutlich über das reine Verschlüsseln und Entschlüsseln von Nachrichten hinaus. Neben der Vertraulichkeit übermittelter Daten, sind Integrität und Authentizität ebenso wichtig. Auch die Techniken zur Autorisierung beim Zugriff auf Computersysteme rechnet man mittlerweile zum Gebiet der Kryptographie, ebenso wie die diversen Verfahren, die bei Kryptowährungen wie etwa Bitcoin zum Einsatz kommen. Die Mathematik ist dabei zum wichtigsten Werkzeug geworden. Mathematische Verfahren helfen dabei, u.a. folgende Fragen zu beantworten:

- Wie erzeugt man zufällige Folgen von Zahlen? Dies ist beispielsweise wichtig, um Schlüssel zu generieren, die für so genannte Wörterbuch-Angriffe (engl. *dictionary attack*) unanfällig sind.
- Welche Hashfunktionen sind geeignet, um die Integrität von Daten zuverlässig zu bestimmen, ohne dass man diese Byte für Byte mit dem Original vergleichen müsste?
- Gibt es mathematische Berechnung, die sich leicht durchführen lassen, aber nur mit sehr hohem Aufwand umkehren lassen? Solche Falltürfunktionen stellen die Grundlage jeder asymmetrischen Verschlüsselung dar.

Darüber hinaus muss sich die angewandte Kryptographie im Zeitalter von Computern mit einer Reihe praktischer Probleme herumschlagen: Verschlüsselung beliebiger Daten statt buchstabenbasierter Nachrichten, Anforderungen an Performance, Ressourcenverbrauch und Bandbreite, sowie die pragmatische Abwägung zwischen dem Grad der erreichten Sicherheit und der Benutzerfreundlichkeit kryptographisch geschützter Systeme.

2.5 Schlüsselverteilung

Die Entdeckung der nicht knackbaren One-Time-Pad-Chiffre war keineswegs die Lösung aller kryptographischen Probleme, da deren Sicherheit - und auch die aller anderen symmetrischen Verschlüsselungsverfahren - davon abhängt, dass alle eingesetzten Schlüssel vom Sender zum Empfänger einer Nachricht gelangen, ohne dass sie dabei ausgespäht werden. Dieses *Schlüsselverteilungsproblem* beschäftigt die Kryptographen seit Jahrhunderten.

Eine Algorithmus zur Lösung dieses Problems wurde 1976 an der Stanford University entwickelt und ist unter dem Namen Diffie-Hellman-Merkle-Schlüsselaustausch (DHE) bekannt. Das Verfahren kombiniert die bereits erwähnten Falltürfunktionen mit Modulo-Arithmetik und ermöglicht, dass zwei Parteien - häufig Alice und Bob genannt - sich durch den Austausch zweier Nachrichten auf einen Schlüssel einigen können, den eine Dritte - Eve - auch dann nicht errechnen kann, wenn Eve alle Nachrichten zwischen Alice und Bob abhört[6].

DHE kommt u.a. in einer Variante von TLS/SSL zum Einsatz. TLS, die *Transport Layer Security*, ist für den Webanwender durch ihre Verwendung im HTTPS-Protokoll besonders wichtig. Um die Anfälligkeit des Protokolls für einen Man-in-the-Middle-Angriff zu mildern, werden dort zusätzlich Zertifikate[7] verwendet, um die Authentizität der Server-Nachrichten sicherzustellen.

2.6 Asymmetrische Verschlüsselung und Public Key Infrastrukturen

Eine Schwäche von DHE und seinen diversen Varianten ist die Notwendigkeit, sich vor dem Nachrichtenaustausch auf einen Schlüssel zu einigen. Für asynchronen Nachrichtenaustausch, wie beispielsweise E-Mails, ist das lästig. Viel praktischer wäre es, wenn man statt der üblichen symmetrischen Chiffre auch *asymmetrische* Verschlüsselungen hätte: Während die Verschlüsselung mit einem Empfänger-spezifischen, aber öffentlich zugänglichen Schlüssel erfolgt, kann nur der gewünschte Empfänger mit seinem privaten und geheim gehaltenen Schlüssel den Klartext einer Nachricht wiederherstellen.

Die Suche nach einem solchen Verfahren war schließlich 1977 von Erfolg gekrönt. Das nach seinen Schöpfern Rivest, Shamir und Adleman *RSA* genannte Kryptosystem greift wiederum in die Trickkiste der Falltürfunktionen und nutzt dabei die Tatsache, dass die Primfaktorzerlegung sehr großer Zahlen äußerst rechenintensiv ist, während deren Multiplikation sehr schnell funktioniert.

Ein RSA-Schlüssel besteht immer aus einem Schlüsselpaar: dem *öffentlichen Schlüssel* (engl. *public key*), den alle kennen sollten, und dem *privaten Schlüssel* (engl. *private key*), den nur die rechtmäßige Besitzerin kennen darf. Aber auch RSA kommt nicht ohne Probleme:

[6] Hier ist ein 5-Minuten-Video, das DHE anschaulich erklärt.

[7] Ein digitales Zertifikat ist ein elektronischer Datensatz, das dem Leser - in der Regel ein Computerprogramm - beweisen soll, dass eine gegebene Internet-Domäne tatsächlich einer bestimmten Person oder Organisation gehört, und dass man daher TCP/IP-Paketen von dieser Domäne vertrauen kann.

- Zum einen ist das Verfahren sehr rechenaufwändig, was es für eine Echtzeit-Entschlüsselung großer Datenmengen unpraktikabel macht. Dies löst man durch hybride Verschlüsselung, d.h. der Kombination mit einem symmetrischen Verfahren (z.B. AES). Dabei wird lediglich ein zufällig generierter symmetrischer Schlüssel per RSA verschlüsselt, die eigentliche Nachrichtenverschlüsselung erfolgt anschließend mit diesem symmetrischen Schlüssel.
- Zum anderen muss man die Zuordnung der öffentlichen Schlüssel zu einer gewünschten Empfängerin gewährleisten. In anderen Worten: Man muss eine Public-Key-Infrastruktur (PKI) aufbauen.

Auch auf asymmetrische Verschlüsselungen werden wir im Laufe des Buches noch intensiver eingehen.

2.7 Signaturen und Zertifikate

Da bei RSA der öffentliche und der private Schlüssel algorithmisch die gleiche Rolle spielen, kann man das Verfahren auch zur Authentifizierung von Nachrichten einsetzen: Benutzt Alice ihren eigenen privaten Schlüssel zur Verschlüsselung einer Nachricht, dann kann diese Nachricht mit Hilfe des öffentlichen Schlüssels von jedermann entschlüsselt werden. Gelingt die Entschlüsselung, dann weiß Bob, dass diese Nachricht tatsächlich von Alice stammt – oder von jemandem, der sich Zugang zu Alice' privatem Schlüssel verschafft hat.

Ähnlich wie bei den hybriden Verfahren wird in der Praxis nicht die vollständige Nachricht verschlüsselt, sondern lediglich ein kollisionsresistenter Hash-Wert der Nachricht. Damit ist die Nachricht für jedermann lesbar, und wer möchte kann die Urheberschaft überprüfen. In diesem Falle spricht man von einer *Signatur* (engl. *signature*).

Ein Sonderfall der Signatur stellen *Zertifikate* dar. Diese sind nichts anderes als *signierte öffentliche Schlüssel*. Durch die Signatur bestätigt die signierende Partei (z.B. eine allgemein anerkannte Zertifizierungsstelle) die Zugehörigkeit eines öffentlichen Schlüssels zu einer bestimmten juristischen oder tatsächlichen Person. Zertifikate werden sowohl in diversen PKIs eingesetzt, als auch zur Sicherung der TLS/SSL-Verbindungen im Internet.

2.8 Kryptographie als Bürgerrecht

Die jüngere Vergangenheit hat gezeigt, dass unverschlüsselte Daten im Internet und auf unseren Computern nicht nur in der Theorie missbraucht werden können, sondern dass es tatsächlich Institutionen gibt, die beinahe jeden Aufwand betreiben, um an diese Informationen flächendeckend zu gelangen. Der Schutz unserer Daten durch sichere Kryptotechnik ist eines der Mittel, um diesem demokratie-gefährdenden Handeln einige große Steine in den Weg zu legen. Doch dafür müssen wir sie benutzen[8]! Dass uns die Nutzung selbst wiederum verdächtig macht, ist ein Dilemma; doch es beweist, dass wir die Technologie verstehen müssen, um sinnvoll mit ihr umzugehen.

[8] Dass uns auch angewendete Kryptographie nur sehr begrenzt schützt, zeigt diese Präsentation von Peter Gutmann.

2.9 Die Zukunft der Kryptographie

Ob die heutigen Verschlüsselungsverfahren ausreichend Sicherheit bieten - sowohl prinzipiell als auch was die Länge der Schlüssel angeht - wissen wir nicht. Die Quantencomputer der Zukunft könnten sowohl manche heutige kryptographische Algorithmen wertlos machen, aber auch zur Entwicklung absolut sicherer Verfahren, wie beispielsweise dem Quantenschlüsseltausch führen.

3. Old-Timer der testgetriebenen Entwicklung: Ron und Steve

Sich mit den Meistern eines Fachs zu unterhalten, bringt immer neue Aha-Erlebnisse. Daher habe ich aus reinem Eigeninteresse einige der TDD-Verfechter der frühen Stunde interviewt. Die Interviews beginnen stets mit einigen Standardfragen und enden mit Themen, die sich auf die Spezialitäten der Interview-Partner beziehen.

3.1 Ron Jeffries

Ron ist ein Pionier des Extreme Programming und Mitautor des agilen Manifests. Seine Aktivitäten kann man auf www.XProgramming.com verfolgen und ihn dort kontaktieren.

Das Interview [1]

Wann war dein erster Kontakt mit TDD und was hast du damals davon gehalten?

> **Ron**: Das war 1996 im C3-Projekt. Ich war bereit es auszuprobieren, weil Kent Beck sehr überzeugend sein kann. Vermutlich glaubte ich, dass es nicht viel nutzen würde, weil ich mich bereits für einen großartigen Programmierer hielt.

Was hat dich letztendlich überzeugt, dass testgetriebene Entwicklung ein lohnenswerter Ansatz ist?

> **Ron**: Es tatsächlich zu tun, und dabei genau hinzuschauen. Ich fand, dass es die Arbeitsweise verändert; als ob man Karten richtig stapelt, anstatt sie auf ihre Kanten zu stellen.

Was hat sich seit den frühen Tagen in der Art und Weise verändert, in der du TDD praktizierst und anderen beibringst?

> **Ron**: Ich glaube, ich versuche es weniger anzupreisen, weil ich alles weniger anpreise als früher. Ich zeige den Leuten, was ich tue, erkläre, wie ich es tue und warum, und überlasse ihnen den Rest.

[1] Hier geht's zum englischen Originaltext des Interviews

Ist deine TDD-Technik eine andere als zu Beginn? Sind deine einzelnen Schritte größer oder kleiner geworden?

> **Ron**: Meine Schritte werden immer kleiner. Natürlich denke ich immer nach, aber ich lasse den Code mir sagen, was ich tun muss, nicht irgendwelche Theorien über Design.

Gehst du an Abhängigkeiten mit anderen Werkzeugen und veränderter Taktik heran? Wie z.B. Test-Doubles und Mocks?

> **Ron**: Als Mitglied der "klassischen TDD-Schule" verwende ich normalerweise keine Test-Doubles, höchstens um mich von einer Datenbank zu entkoppeln oder um eine größere Objektmenge darzustellen. Stattdessen lasse ich Objekte an Ort und Stelle wachsen - und teste sie während sie größer werden.

Gibt es Situationen, in denen du TDD nicht für den richtigen Entwicklungsansatz hältst? Und wenn ja, welche Techniken und Ansätze empfiehlst du in diesen Fällen?

> **Ron**: Ich finde TDD schwierig, wenn ich Software schreibe, die die echte Welt direkt manipuliert. Wenn Systemaufrufe Gegenstände bewegen oder anderes "physisches" Verhalten auslösen, dann fallen mir Tests mit TDD schwer, da es keine Stelle gibt, an der man dieses Verhalten unterbrechen und überprüfen könnte, ohne dass es tatsächlich passiert.
>
> Ich finde TDD auch dann schwierig, wenn es noch kein entsprechendes Framework gibt und wenn das Basissystem zu nackt ist, um schnell eines zu bauen, beispielsweise ohne Reflection und ohne Objekte. Dann mache ich sehr kleine Schritte und drucke die Zwischenergebnisse aus. Häufig überprüfe ich sie dann lediglich visuell; selten schreibe ich dann einen kleinen Test im TDD-Stil. Einen Debugger verwende ich jedoch nie, in den meisten Umgebungen wüsste ich nichtmal, wie er funktioniert.

Was ist deiner Meinung nach die Bedeutung von testgetriebener Entwicklung in der heutigen Welt - mit Lean Startups, funktionaler und nebenläufiger Programmierung, Continuous Delivery und den allgegenwärtigen Mobilgeräten?

> **Ron**: TDD ist immer noch die beste Art zu programmieren, die ich kenne. Wo sie eingesetzt werden kann - also fast überall - hilft TDD den Code gleichmäßig wachsen zu lassen; sie führt meist zu einfachem aber gutem Design, und sie hält mich davon ab, Karten auf ihre Kante zu stellen.

Herzlichen Dank, Ron, für die Beantwortung der Fragen!

Persönliches Fazit

Trotz mehrerer Jahrzehnte an Programmiererfahrung gibt es für Ron keinen besseren Entwicklungsansatz. Das macht mir noch einmal deutlich, dass alternative Entwicklungsansätze zu TDD, die ähnliche Ergebnisqualität liefern und gleichzeitig eine feine Kontrolle über die Entwicklung erlauben, nicht auf der Straße liegen.

3.2 Steve Freeman

Steve ist der prominenteste Vertreter der "London School of TDD", auch wenn er behauptet, dass diese gar nicht existiert. Zusammen mit Nat Pryce hat er das Buch "Growing Object-Oriented Software, Guided by Tests" geschrieben. Dies ist eines der einflussreichsten TDD-Bücher der vergangenen Jahre. Steve twittert als @sf105 und kann über http://www.higherorderlogic.com kontaktiert werden.

Das Interview[2]

Wann war dein erster Kontakt mit TDD und was hast du damals davon gehalten?

> **Steve:** Das war irgendwann 1997 oder 1998. Ich arbeitete bei OTI in London und wir verfolgten die anschwellende Diskussion um das C3-Projekt auf dem C2-Wiki. Zunächst war unsere allgemeine Reaktion "Das kann überhaupt nicht klappen", aber wir haben immer weiter Dinge ausprobiert und es stellte sich heraus, dass sie doch funktionierten. Außerdem war Kent Beck in jenem Jahr Sprecher auf der SPA Konferenz, so dass wir mehr darüber herausfinden konnten, wie die diversen Praktiken tatsächlich durchgeführt wurden.

Was hat dich letztendlich überzeugt, dass testgetriebene Entwicklung ein lohnenswerter Ansatz ist?

> **Steve:** Ausprobieren und feststellen, dass es für mich hilfreich war. Außerdem gab es ein paar schöne Gegenbeispiele, bei denen ein Teammitglied sich die Nacht um die Ohren schlug, um etwas "hinzubekommen", was dann das Team einen ganzen Vormittag kostete, die eingebauten Fehler wieder zu entfernen.

Was hat sich seit den frühen Tagen in der Art und Weise verändert, in der du TDD praktizierst und anderen beibringst?

[2] Hier geht's zum englischen Originaltext des Interviews

Steve: Einiges ist reine Technik, wie etwa das Zerschneiden eines Feature in Inkremente und das Arbeiten auf unterschiedlichen Testebenen. Heutzutage verwende ich viel Mühe darauf, die Tests lesbar und ausdrucksstark zu gestalten, anstatt möglichst auch das kleinste Detail zu testen. Eigentlich war das schon immer so, aber heute ist es meine oberste Priorität. Auch habe ich jetzt ein besseres Verständnis, wie ich TDD als "Denkwerkzeug" einsetzen kann, um meine Ideen klarer zu machen bevor ich sie in Code festhalte. Ich benutze mittlerweile auch System-Level-Tests, um das Grobdesign zu beeinflussen. Und schließlich habe ich begriffen, dass TDD eine tiefgehende Technik ist, wie alles bei der Programmierung. Ein paar Tage Workshop und ein Buch durchblättern ist nichts als eine kleine Kostprobe.

Gibt es Situationen, in denen du TDD nicht für den richtigen Entwicklungsansatz hältst? Und wenn ja, welche Techniken und Ansätze empfiehlst du in diesen Fällen?

Steve: Nat Pryce zitiert gerne Manny Lehmanns Softwarekategorien. Einige Arten "algorithmischer" Systeme (P-Systeme) sollten gelöst werden, indem man intensiv nachdenkt – auch wenn ich glaube, dass TDD selbst hier zur Implementierung beitragen kann. Ich arbeite jedoch meist an "unordentlichen" E-Systemen, die sich mit ihrer Umgebung verändern und keine sauberen Anforderungen haben. Dies ist der Ort, an dem TDD im weiteren Sinne besonders gut passt.

Andere Ausnahmen sind Systeme, die schnell zusammengesteckt werden um etwas zu beweisen, oder Systeme, mit denen der Programmierer ein neues Gebiet erforscht. Ich denke da u.a. an Sam Aarons live Musik-Programmierung. Außerdem gibt es eine neue Systemgeneration mit "Testen in Produktion", bei denen die Produktionsinfrastruktur so robust ist, dass die Entwickler risikolos mit dem echten System experimentieren können. Entscheidend ist dabei, dass man sich bewusst ist, was ich priorisiere und welche Risiken ich möglicherweise anhäufe. Kent Beck benutzt die Unterscheidung: Priorisierung für geringe Wartezeit (engl. low latency) versus ausdauerndem und gleichmäßigem Durchsatz (engl. sustained throughput) von neuen Funktionen.

Egal welche Techniken ich einsetze, ich halte es gewöhnlich für wertvoll innezuhalten und darüber nachzudenken, was ich als Ergebnis möchte, bevor ich mit dem Coding beginne; auch wenn ich diese Überlegungen dann nicht automatisiere.

Was ist deiner Meinung nach die Bedeutung von testgetriebener Entwicklung in der heutigen Welt - mit Lean Startups, funktionaler und nebenläufiger Programmierung, Continuous Delivery und den allgegenwärtigen Mobilgeräten?

Steve: Das ist eine ordentliche Liste, und ich bin nicht sicher, ob sie tatsächlich bereits die Mehrheit der Softwareumgebungen betrifft. Die Kurzfassung: Es scheint mir immer

> eine gute Idee zu sein, irgendeine Art von Regressionstests zu haben, insbesondere wenn ich schnell und sicher vorankommen möchte. Unter dieser Voraussetzung fällt es mir leichter, die Tests vorher zu erstellen, denn in der Praxis werde ich es danach nicht mehr tun. Es existieren einige hochentwickelte Umgebungen, in denen das nicht zutrifft, dennoch würde ich das Risiko nicht eingehen, bevor ich nicht genau verstehe, was ich aufgebe und wie ich das kompensieren kann.

Du bist ein prominenter Vertreter der "London school of TDD", welche den Einsatz von Mock-Objekten betont, um dadurch die Verträge (engl. collaboration contracts) zwischen Objekten festzulegen, anstatt lediglich das Verhalten eines isolierten Objekts zu testen. Würdest du sagen, dass diese Kollaborationstests wertvoller sind als zustandsbasiertes (engl. state-based) Testen? Was entgegnest du Kritikern, die Mock-Objekte ablehnen, weil sie nicht "die echte Integration" testen?

> **Steve:** Ich bin es mittlerweile leid, diese Diskussion mit Leuten zu führen, die unser Buch nicht gelesen haben und nicht verstehen, wovon wir wirklich reden. Ich priorisiere nicht die eine Testart grundsätzlich vor der anderen, sondern ich priorisiere die Art, die auf das zu testende Objekt am besten passt. Habe ich ein Objekt mit eigenem Verhalten und ich praktiziere "Tell, Don't Ask"-Design, dann kann ich nur die Interaktionen testen. Für andere Arten von Objekten trifft das nicht zu.
>
> Das Argument zur "echten Integration" trifft dann zu, wenn ich Mocks einsetze, um gegen eine externe Schnittstelle zu testen, was ich persönlich nie tue - wir raten auch bereits seit mehr als 10 Jahren davon ab. Für diesen Fall benutze ich Integrationstests.
>
> Der Punkt, den ich allerdings überhaupt nicht begreife, und ich sehe ihn auch bei sehr guten Entwicklern, ist die Unfähigkeit, mit vorab spezifizierten Mock-Constraints umzugehen; diese Unfähigkeit hat u.a. Werkzeuge wie Mockito hervorgebracht. Das Problem wird verschwinden, sobald Java Closures besitzt. Ich habe aufgehört dagegen zu kämpfen und nutze es als Wettbewerbsvorteil. Wenn jemand wirklich zuhören möchte, dann führe ich gerne eine zivilisierte Diskussion.

Herzlichen Dank, Steve, für die Diskussion!

Persönliches Fazit

Die Ideen hinter Manny Lehmanns *Laws of Software Evolution* waren für mich neu und können dabei helfen, die Anwendbarkeit testgetriebener Entwicklung auf ein konkretes Problem zu beurteilen. Der letzte Teil des Interviews greift einem Thema vor, dem in diesem Buch mehrere Kapitel gewidmet sind: Abhängigkeiten und wie man mit ihnen umgeht. Es lohnt sich daher sicherlich, das Interview nach der Lektüre der entsprechenden Kapitel nochmals zu lesen.

Anhänge

Anhang A: Kurze Einführung in Groovy

Groovy ist eine Programmiersprache auf der JVM, die das Ziel verfolgt, sich perfekt in des Java-Ökosystem zu integrieren, den Übergang für Java-Entwickler nahtlos zu gestalten und trotzdem so viele neue Fähigkeiten mitzubringen, dass fast alles einfacher und manches überhaupt erst möglich wird.

Bad old Java

Fangen wir mit einer einfachen Java-Klasse an und verwandeln diese Schritt für Schritt in idiomatisches Groovy.

```
1  public class HumanBeing {
2    private String name;
3
4    public String getName() {
5      return name;
6    }
7
8    public void setName(String newName) {
9      this.name = newName;
10   }
11
12   public HumanBeing(String name) {
13     this.name = name;
14   }
15
16   public String computeFromName(DoWithName code) {
17     return code.withName(name);
18   }
19
20   public static void main(String[] args) {
21     HumanBeing son = new HumanBeing("Jannek");
22     son.setName("Niklas");
23     DoWithName code = new DoWithName() {
24       @Override
25       public String withName(String name) {
26         return name + name;
```

```
27        }
28      };
29      System.out.println(son.computeFromName(code));
30    }
31
32    interface DoWithName {
33      String withName(String name);
34    }
35  }
```

Diese Klasse erlaubt die Instanziierung menschlicher Wesen mit einer einzigen *Property*, welche den Konventionen für JavaBeans Zugriffsoperationen folgt.

Erster Schritt: *.java -> *.groovy

Im ersten Schritt unserer Metamorphose benennen wir einfach die Datei `HumanBeing.java` in `HumanBeing.groovy` um und benutzen für die Kompilierung `groovyc` statt `javac`. Dabei machen wir uns die Tatsache zu nutze, dass 99% allen Java-Codes auch gültigen Groovy-Code darstellt.

Doch Vorsicht, manchmal ändert sich bei dieser Umstellung die Semantik des Programms, z.B. weil Groovys Method-Dispatching normalerweise[3] dynamisch anhand der Laufzeit-Typen der Parameter stattfindet. In unserem einfachen Beispiel ist dieser Unterschied jedoch bedeutungslos.

Zweiter Schritt: Überflüssiges

Die einfachsten Maßnahmen zur Verschlankung des Codes ist das Weglassen von Elementen, die in Groovy optional sind, und die Benutzung von masse-reduzierenden Spracheigenschaften:

- "`public`" ist der Default-Modifier und kann weggelassen werden.
- "`return`" an letzter Stelle einer Methode benötigt man nicht, wenn die letzte *Expression* (z.B. eine Variable) zurückgegeben werden soll.
- "`;`" am Zeilenende ist überflüssig.
- Normale Strings werden mit einfachen statt doppelten Hochkommata gebildet. Doppelte Hochkommata haben eine besondere Bedeutung.
- Member-Variablen ohne die *Modifier* `public`, `private` oder `protected` werden automatisch als *Properties* interpretiert. Dies bedeutet, dass Getter- und Setter-Methoden im Byte-Code generiert werden, und dass Zugriffe auf die Property automatisch auf diese Methoden umgelenkt werden.

Durch die Umsetzung dieser Vereinfachungen wird das Ergebnis bereits kürzer und – meiner Meinung nach – besser lesbar:

[3] Groovy macht das immer, es sei denn, eine Datei, eine Klasse oder eine Methode wurde explizit mit `@CompileStatic` markiert, was zu statischer Kompilierung à la Java führt – und damit auch zu Method-Dispatching anhand des statischen Compiletime-Typs.

```
class HumanBeing {
  String name

  public HumanBeing(String name) {
    this.name = name
  }

  String computeFromName(DoWithName code) {
    code.withName(name)
  }

  public static void main(String[] args) {
    HumanBeing son = new HumanBeing('Jannek')
    son.name = 'Niklas'
    DoWithName code = new DoWithName() {
      @Override
      String withName(String name) {
        name + name
      }
    }
    System.out.println(son.computeFromName(code))
  }

  interface DoWithName {
    String withName(String name)
  }
}
```

Dritter Schritt: Optionale Typen

Die Angabe von Typen für Variablen und Parameter ist in Groovy freiwillig. Aus diesem Grund benötigt Groovy ein weiteres Schlüsselwort `def`, um Variablen ohne Typ und ohne Modifier deklarieren zu können. Hier die neue Fassung – ohne explizite Typen und ohne `System.out`, da `println` als globale Funktion verfügbar ist:

```
class HumanBeing {
  def name

  public HumanBeing(name) {
    this.name = name
  }

  def computeFromName(code) {
    code.withName(name)
  }

  public static void main(args) {
    HumanBeing son = new HumanBeing('Jannek')
    son.name = 'Niklas'
    def code = new DoWithName() {
      @Override
      def withName(name) {
        name + name
      }
    }
    println(son.computeFromName(code))
  }

  interface DoWithName {
    def withName(name)
  }
}
```

Statische Typisierung

Seit Groovy 2 erfüllt die Sprache auch die Wünsche derer, die aus Sicherheits- oder Performance-Gründen nicht auf strenge Typprüfung verzichten wollen. Es existieren zwei Annotationen, die wahlweise für eine vollständige Typdeklaration oder für einzelne Methoden verwendet werden können:

- **`@TypeChecked`** behält die dynamischen Eigenschaften von Groovy bei, überprüft aber, ob alle Parameter und Variablen auch konform zu ihrer Typdeklaration verwendet werden. Diese Annotation hat keine Auswirkung auf den generierten Byte-Code.
- **`@CompileStatic`** sorgt tatsächlich für Byte-Code, der zusätzlich zur Typprüfung auch zur Generierung von statisch gebundenen Methodenaufrufen führt. Daher kann man mit Hilfe dieser Annotation Groovy-Programme schreiben, welche die gleiche Performance-Charakteristik aufweisen wie äquivalente Java-Programme. Meist bedeutet dies: Sie sind schneller.

Die statische Typprüfung bei Groovy ist um einiges rafinierter als ein Java-Compiler. Viele Typen kann Groovy statisch auflösen, die bei Java einen Cast erfordern. Beispielsweise ist

```
def son = new HumanBeing(name: 'Jannek')
println son.name
```

statisch compilierbar, da der Groovy-Compiler weiß, dass die Variable `son` eine Instanz der Klasse `HumanBeing` enthält.

Vierter Schritt: Default-Konstruktoren

Ohne dass man etwas tun müsste, erzeugt Groovy für jede Klasse einen Konstruktor, der das initiale Setzen einzelner oder aller Properties mit Hilfe von *benannten Parametern*. Damit können wir auf den `HumanBeing`-Konstruktor ganz verzichten, müssen jedoch die Instanziierung ein klein wenig verändern:

```
HumanBeing son = new HumanBeing(name: 'Jannek')
```

Auch die in Java verbreitete Form von Konstruktoren mit der Aufzählung aller Properties im Konstruktor, können wir bei Bedarf mit der Annotation `@TupleConstructor` erzeugen.

Letzte Schritte: Closures

Erst mit JDK 1.8 hat Java Lambda-Ausdrücke spendiert bekommen. Groovy besitzt dieses wichtige Konstrukt schon von Beginn an – unter dem Namen *Closures*. Closures sind nichts anderes als "Codeblöcke", die – genau wie andere Objekte auch – zugewiesen, als Berechnungsergebnis zurückgegeben und als Parameter übergeben werden können. Ausführen kann man diese Codeblöcke, indem man sie wie Funktionen aufruft, gegebenenfalls mit Parametern.

In Groovy werden Closures durch geschweifte Klammern gekennzeichnet. Hier ein kurzes Beispiel:

```
def istTeilbar = {zahl, teiler -> (zahl % teiler) == 0}
assert istTeilbar(35, 7) == true
```

Gekoppelt mit der Möglichkeit eine Closure als letzten Parameter außerhalb der Klammern zu schreiben, werden wir nicht nur das Interface `DoWithName` los, sondern können den Aufruf der `computeFromName`-Method ohne temporäre Variable vornehmen. Hier der Code nach der abermaligen Verkürzung:

```
class HumanBeing {
  def name

  def computeFromName(code) {
    code(name)
  }

  public static void main(args) {
    HumanBeing son = new HumanBeing(name: 'Jannek')
    son.name = 'Niklas'
    println(son.computeFromName() { name -> name + name } )
  }
}
```

Tatsächlich lässt sich die letzte Zeile der `main`-Methode noch weiter vereinfachen, wenn man auf den Default-Parameter **`it`** von Closures zurückgreift und ein paar unnötige Klammern weglässt:

```
    println son.computeFromName { it + it }
```

Übrigens werden Closures vom Groovy-Compiler immer dann automatisch in Java 8 Lambda Expression übersetzt, wenn der Code einen Lambda-Ausdruck erwartet. Als Groovy-Entwickler muss man daher nicht darüber nachdenken, ob an einer bestimmten Stelle eine Closure oder ein Lambd-Ausdruck erwartet wird.

Fertiger Code des Kapitels

- HumanBeing.groovy
- HumanBeingTest.groovy

Closures im GDK

Closures werden dann zu einem mächtigen Sprachmittel, wenn auch die eingesetzten Bibliotheken deren Einsatz fördern. Das Groovy Development Kit (abgekürzt *GDK*) tut genau das. So existieren für alle Collections zahlreiche Methoden, die mit Closures arbeiten, z.B.:

```
[1, 2, 3, 4].findAll {it % 2 == 0}.each {println it}
```

Doch auch an zahllosen anderen Stellen des GDK kommen Closures zum Einsatz, wie z.B. beim sicheren Zugriff auf Ressourcen oder beim Einsatz von Nebenläufigkeitsparadigmen wie Aktoren, Agenten und Dataflows.

Echte Assertions

Auf den ersten Blick könnte man vermuten, dass Groovys `assert` Schlüsselwort das gleiche tut wie Java – und in erster Näherung ist das auch richtig: Ein `assert` erhält als Parameter einen booleschen Ausdruck, der zur Laufzeit ausgewertet wird und im `false`-Falle zum Abbruch des Programms mit einem `AssertionError` führt. Assert-Ausdrücke in Groovy haben jedoch zwei grundlegende Unterschiede:

- Sie müssen nicht eingeschaltet und können auch nicht ausgeschaltet werden[4]. Dadurch wird `assert` zu einem Partner, auf den man sich verlassen kann.
- Die von einem fehlschlagenden Assert gelieferten Fehlermeldungen sind sehr detailliert und aussagekräftig, man nennt dieses Feature daher *Power Assertions*. Ein Beispiel:

  ```
  def actual = [1, 2, 3]
  def expected = [1, 2, 4]
  assert actual == expected
  ```

  ergibt die folgende Fehlermeldung:

  ```
  Assertion failed:
  assert actual == expected
         |      |  |
         |      |  [1, 2, 4]
         |      false
         [1, 2, 3]
         at MyProgramm.aMethod(MyProgramm:42)
  ```

Diese beiden Eigenschaften machen `assert` zum Assertion-Konstrukt erster Wahl; und das auch in JUnit-Testfällen, in denen man in Java ein `Assert.assertEquals` oder ähnliches einsetzen würde.

Meta-Programmierung und weitere Features

Groovy hat noch zahlreiche weitere nützliche Features; viele davon fallen in das Gebiet der Metaprogrammierung:

- Analyse und Veränderung von Programmverhalten zur Laufzeit, auch *Dynamic Groovy* genannt. Beispielsweise kann man **jeder** Klasse zusätzliche Methoden spendieren:

[4] In Java ist die Auswertung von asserts per Default ausgeschaltet und muss über die Schalter `-ea` bzw. `-enableassertions` explizit aktiviert werden. Die Folge: asserts werden in "normalen" Java-Programmen kaum eingesetzt.

```
String.metaClass.hellofy = { "Hello, $delegate!" }
assert 'Johannes'.hellofy() == 'Hello, Johannes!'
```

- AST Transformationen werden vom Compiler ausgewertet und erlauben fast beliebige Veränderung des abstrakten Syntaxbaums eines Programmes **vor dessen Ausführung**. Zahlreiche nützliche Transformationen sind bereits in Groovy eingebaut, wie z.B. `@TailRecursive`:

```
@TailRecursive
long sizeOfList(list, counter = 0) {
    if (list.size() == 0)
        return counter
    return sizeOfList(list.tail(), counter + 1)
}
// Without @TailRecursive a StackOverFlowError
// is thrown.
assert sizeOfList(1..10000) == 10000
```

Hier noch eine kleine, hochgradig unvollständige Featureliste, um dem Leser den Mund wässrig zu machen:

- *GStrings* sind Strings in doppelten Hochkommata, die durch `$` oder `${}` gekennzeichneten Code enthalten können.
- Literale Erzeugung
  - von Listen: `[1, 'hallo']` oder `[1, 2, [31, 32]]`
  - von Maps: `[a:1, b:2, 'äöü': ['ä', 'ö']]`
  - von Bereichen: `5..8` oder `'a'..'f'`
- Fähigkeit sämtliche Operatoren zu überladen
- Native Unterstützung von regulären Ausdrücken
- Ein mächtiges Builder-Konzept zum Bauen von GUIs, Markup und anderen komplexen Objekten
- Viel Unterstützung für das Schreiben interner *DSLs* (domain specific languages)

Eine kurzweilige Variante, um sich mit Groovy vertraut zu machen, ist das Projekt GroovyKoans: eine Sammlung von fehlschlagenden Testfällen, die man zum Laufen bringen soll. Viel Spaß!

Anhang C: Quellen und Literatur

[Beck 2003] Kent Beck: *Test-Driven Development: By Example. Addison-Wesley, 2003.*

[Beck 2010] Kent Beck: *Test Driven Development. Four video and screencast episodes by Kent Beck.* The Pragmatic Bookshelf.

[Evans 2030] Eric J.Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.*

[Freeman+Price 2020] Steve Freeman, Nat Price: *Growing Object-Oriented Software, Guided by Tests. Addison-Wesley, 2010.*

[Martin 2011] Robert C. Martin: *The Clean Coder: A Code of Conduct for Professional Programmers. Prentice Hall PTR, 2011å.*

[Meyer 1997] Betrand Meyer: *Object-Oriented Software Construction. Prentice Hall PTR, 2nd edition, 1997.*

[Schmeh 2013] Klaus Schmeh: *Kryptografie. Verfahren - Protokolle - Infrastrukturen. dpunkt.verlag, 5. Auflage, 2013.*

[Singh 2001] Simon Singh: *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet. Deutscher Taschenbuch Verlag, 2001.*