1 Einleitung

Testen ist wichtig. Das wissen alle Softwareentwickler, aber (fast) keiner tut's. Die Gründe für das stiefmütterliche Interesse, das Programmierer der Qualität ihrer eigenen Produkte entgegenbringen, sind vielfältig. Zum einen spielt die Ausbildung eine wichtige Rolle: Dem akademisch erzogenen Informatiker ist die Tatsache, dass Software getestet werden muss, meist nur als theoretischer Punkt im Rahmen seiner 2-semestrigen Softwaretechnik-Vorlesung untergekommen. Die Autodidakten unter den Programmieren haben in den Lehr- und Programmierbüchern meist nichts weiter gefunden als den Hinweis, dass die Programme »natürlich« auch gründlich getestet werden müssen. Nur wie und warum steht dort nie – von ein paar rühmlichen Ausnahmen wie [Gassmann00] und [Larman00] abgesehen.

So baut sich der Entwickler mit der Zeit seinen eigenen Vorrat an Vorurteilen und Gründen auf, der ihn bei seiner Abneigung gegen das Testen bekräftigt. Eine kleine Auswahl:

»Ich habe keine Zeit zum Testen«

Dieser häufig vorgebrachte Satz geht davon aus, dass Testen Zeit kostet. Glaubt man das, so gerät man in einen Teufelskreis: Je größer der Zeitdruck, desto weniger Tests. Je weniger Tests, desto unstabiler der Code. Je unstabiler der Code, desto mehr Fehlermeldungen kommen vom Kunden. Je mehr Fehlermeldungen, desto mehr Debugging-Zeit wird benötigt. Je mehr Debugging-Zeit, desto größer der Zeitdruck ...

Glaubt man jedoch, dass Tests den Code stabilisieren, dann wird aus dem Teufelskreis eine sich öffnende Spirale: Je stabiler der Code, desto weniger Debugging-Zeit wird benötigt. Je weniger Debugging-Zeit, desto mehr Zeit bleibt für Entwicklung (und Tests). Davon handelt dieses Buch.

»Testen von Software ist langweilig und stupide«

Die gängige Literatur zum Thema Softwaretesten (z.B. [Binder99] und [McGregor01]) ist eher trocken und theoretisch, was den praktisch veranlagten Programmierer verschreckt. Genauer betrachtet ist jedoch das Entdecken und damit das Vermeiden von Softwarefehlern eine ebenso anspruchsvolle und kreative Tätigkeit wie das Programmieren selbst. Mehr noch, frühzeitiges Testen kann den Programmiervorgang sogar steuern und befriedigender machen, da man mehr Vertrauen in das Ergebnis seiner Arbeit gewinnt. Davon handelt dieses Buch.

»Mein Code ist praktisch fehlerfrei, auf jeden Fall gut genug«

Gerne glauben wir Entwickler an unsere eigene intellektuelle Brillanz. Schließlich haben wir alle Bücher zum Thema gelesen, sind in alle Fallen schon mal getappt, kennen sämtliche veröffentlichte Entwurfsmuster und die Details der 363 Klassen unseres eigenen Frameworks auswendig. Und dennoch, bei einer kleinen Änderung in Klasse 276 kommen uns plötzlich Zweifel, welche Auswirkungen diese neue Codezeile auf andere Teile des Systems haben könnte. Wäre es jetzt nicht schön, anhand einer Menge bewährter und automatisierter Tests überprüfen zu können, ob wir mit der Verbesserung am einen Ende nicht versehentlich eine Funktion am anderen Ende stören? Davon handelt dieses Buch.

»Die Testabteilung testet. Die können das eh viel besser.«

Antidecomposition-Axiom

Zudem sitzen in der Testabteilung genau die Art von pedantischen und übergenauen Beamten, die man für einen Buchhalterjob wie das ständige Ausführen immer gleicher Testskripts benötigt. Schade nur, dass es in der Testtheorie ein paar massive Hinweise und Belege dafür gibt, dass die Testabteilung eben **nicht** unseren Job erledigen kann. Diese Erkenntnis führte zur Aufstellung des ***Antidecomposition-Axioms***[1]. Dieses Axiom besagt, dass das Testen eines zusammengesetzten Systems nicht ausreicht, um die Fehler seiner Komponenten aufzudecken. Der Programmierer ist daher dafür verantwortlich, dass seine Komponenten auch in isolierter Umgebung getestet werden. Darüber hinaus können frühzeitig erkannte Fehler weitaus schneller und kostengünstiger behoben werden. Auch davon handelt dieses Buch.

---

1. Robert Binders [Binder99] drei Testaxiome werden in Kapitel 7 näher beleuchtet.

1.1 Kleine Begriffslehre

Unterschiedliche Testarten

Wir sind uns also nun (hoffentlich) einig, dass ***Entwicklertests*** für eine qualitativ hochwertige Software unverzichtbar sind. Doch auch Entwickler testen unterschiedliche Dinge auf unterschiedliche Arten. So konzentrieren sich ***Performanz-*** und ***Lasttests*** auf die Erfüllung bestimmter nicht funktionaler Anforderungen wie geforderte Antwortzeiten und erwartete Nutzerzahlen. Die zentrale Testaufgabe der Programmierer sind jedoch die so genannten ***Unit Tests***, auf deutsch ***Modultests***. Dieser Begriff stammt aus der vorobjektorientierten Ära und beschreibt, dass sich die einzelnen Tests nicht auf das Gesamtsystem, sondern auf einzelne *Units* (Einheiten) des Systems konzentrieren. Heutzutage wird häufig auch von ***Komponententest*** gesprochen.

Modultests

Komponententests

Was ist eine Unit?

Seinerzeit war eine solche »Einheit« leicht als Prozedur oder Funktion zu erkennen. Bei objektorientierten Systemen kann diese »zu testende Einheit« unterschiedliche Gestalten annehmen. Die Spanne reicht von der einzelnen Methode über Klasse und Subsystem hin zum ganzen System. Meist (aber nicht immer) handelt es sich bei dieser Einheit um die »natürliche« Abstraktionseinheit objektorientierter Systeme: die Klasse bzw. um ihre instanzierte Form: das Objekt. Um das etwas holprige deutsche Wortkonstrukt »zu testende Einheit« zu vermeiden, tauchen in diesem Buch häufiger zwei Abkürzungen der angloamerikanischen Fachliteratur auf: *CUT* (Class under Test) und *OUT* (Object under Test). Der Unterschied zwischen beiden besteht in der Perspektive: Befinde ich mich mitten im Test, dann interessiert mich das Objekt an sich; spreche ich von mehreren Tests, dann ist die Klasse mein Bezugspunkt.

Class under Test (CUT)

Object under Test (OUT)

Integrationstests

Entwickler müssen sich auch mit ***Integrationstests*** herumschlagen. Dies sind Tests, die sich auf das Zusammenspiel mehrerer bereits einzeln getesteter Komponenten konzentrieren; [McGregor01] nennt sie daher auch ***Interaktionstests***. Da in objektorientierten Systemen jedoch auch jede Integration durch ein oder mehrere Objekte repräsentiert wird, ist eine scharfe Trennung zwischen Komponententests und Integrationstests häufig nicht möglich. Wo eine Unterscheidung sinnvoll ist, wird im Laufe des Buches darauf eingegangen. In der Regel sollten wir jedoch nicht allzu viele Gedanken daran verschwenden; wichtig ist das Ergebnis unserer Testbemühungen und nicht etwa die einwandfreie terminologische Klassifikation.

Interaktionstests

Statisch oder dynamisch?

Unit Tests sind das zentrale Thema dieses Buches; für eine qualitativ hochwertige und vom Kunden akzeptierte Software benötigt man jedoch noch mehr. So unterscheidet man ***statische*** und ***dynamische*** Tests; letztere finden sich als Komponententests, ***funktionale Tests***,

Akzeptanztests, *Regressionstests* und in andere Varianten wieder. Häufig werden diese von einem dedizierten Testteam oder sogar dem Kunden selbst spezifiziert und ausgeführt. Für den Großteil des Buches konzentrieren wir uns auf entwicklerseitige Tests, bis schließlich Kapitel 14 unsere Unit Tests in den Gesamtkomplex *Softwareprozess* und *Qualitätssicherung* einordnet. Wer vor Spannung platzt, darf schon mal vorblättern.

1.2 Testen in XP

Komponententests in XP

Trotz der Betonung ihrer Wichtigkeit in der Testliteratur spielten Unit Tests bislang für die meisten Entwickler eine mehr als untergeordnete Rolle. Dies änderte sich, als *Extreme Programming* (XP) die Durchführung von Komponententests zu einer zentralen Tätigkeit im XP-Entwicklungszyklus beförderte. XP (siehe [Beck00a] und [Jeffries00]) ist ein leichtgewichtiger Entwicklungsprozess, der dem Kunden die volle Macht über Richtung und Richtungswechsel eines Projektes zurückgibt. Ins Zentrum der Entwicklungstätigkeit rückt dabei das eigentliche Kodieren. Mit dieser provokanten Schwerpunktsverlagerung vergrault XP die Verfechter detaillierter und ausgefeilter Analyse- und Design-Methodiken und findet dafür bei vielen Softwareentwicklern positive Resonanz, die sich häufig durch unangemessene und zu bürokratische Vorgehensmodelle gegängelt fühlen.

Wer sich für den Gesamtkomplex Extreme Programming und seine Beziehung zu schwergewichtigeren Vorgehensmodellen interessiert, kommt um das Studium der einschlägigen Literatur bzw. der ausführlichen Web-Präsenz nicht herum (siehe Anhang D.3: *Weiterführende Lesehinweise*). An dieser Stelle soll lediglich auf einige zentrale Punkte eingegangen werden, die für das Testen, wie es in diesem Buch beschrieben wird, wichtig sind.

Kommunikation, Einfachheit, Feedback und Mut

Dies sind zentrale Werte von XP und spiegeln sich daher in jedem Stückchen Programmcode wider: Code in XP soll so geschrieben werden, dass er alle Dinge kommuniziert, die er enthält. Dies verlangt besondere Sorgfalt bei der Benennung von Klassen und Methoden. Auch können kurze Methoden mit aussagekräftigen Namen lange Programmkommentare in den meisten Fällen ersetzen und sind weniger anfällig dafür, bei späteren Programmänderungen inkonsistent zu werden.

Zusätzlich soll das Programm nur so komplex sein, wie es die *augenblickliche Funktionalität* verlangt. Insbesondere bedeutet dies einen Verzicht auf das Berücksichtigen vermuteter zukünftiger Funktionalität. XP geht nämlich davon aus, dass bei Einhaltung aller zentralen Praktiken spätere Änderungen billiger sind als das vorherige »Eindesignen« denkbarer Anforderungen, insbesondere weil sich in jedem Projekt ein Großteil der Vermutungen später als falsch herausstellt.

Für das schnelle *Feedback*, ob unser Code auch das tut, was er tun soll, dienen automatisierte Tests auf mehreren Ebenen (siehe unten). *Mut* wird vom Team immer dann verlangt, wenn Änderungen am System nötig sind. Umfangreiche Tests sorgen dafür, dass nur Mut und nicht etwa Waghalsigkeit erforderlich ist.

Pair Programming

XP fordert, dass jedes Stück Code, das in Produktion gehen soll, von zwei Entwicklern an einem Computer gemeinsam erstellt wird. Idealerweise ist dabei ein Programmierer mit seinen Gedanken sehr dicht an den Zeilen, die er gerade tippt, während der andere den größeren Zusammenhang im Auge hat. Die Rollen wechseln dabei ständig. *Pair Programming* (dt. »Programmierung in Paaren«) ist eine Art ständiger Review und sorgt für weniger Fehler, mehr Konsistenz mit den Kodierrichtlinien und der Verbreitung von Wissen über das ganze Team. Die Tests stellen sicher, dass das Paar nicht den Fokus verliert.

Programmierung in Paaren

Dem Gefühl vieler Manager, dass mit dieser Vorgehensweise Ressourcen vergeudet werden, widersprechen Studien zur Produktivität von Pair Programming (siehe [Cockburn00a]). Diese zeigen, dass ein geringfügig verkleinerter Ausstoß an Codemenge durch besseres Design und eine deutlich geringere Fehlerrate mehr als ausgeglichen wird.

Inkrementelle und iterative Entwicklung

Die Softwareentwicklung in XP findet nicht *en bloc* statt, sondern in möglichst kleinen Schritten. Das Gesamtsystem wird in *Iterationen* von 1 bis 3 Wochen Länge erstellt. Ziel jeder Iteration ist die Implementierung einer vom Kunden ausgewählten Menge kleiner »Funktionalitätshappen« (*User Stories*). Das Entwicklungsteam zerlegt diese User Stories in *Tasks*; das sind Teilaufgaben, die von einem Entwicklerpaar innerhalb weniger Tage erledigt werden können. Aber auch diese Teilaufgaben werden nicht am Stück, sondern wiederum in kleinen Schritten implementiert. Zu einem solchen Mikroschritt gehört nicht nur der Implementierungscode, sondern auch der Test, der

beweist, dass die Implementierung auch das tut, was sie soll. Ohne diesen Test gilt auch die Implementierung als nicht vorhanden.

Refactoring

Refactoring (dt. etwa »neu herstellen«) beschreibt das ständige Umstrukturieren unseres Codes hin zum einfachsten Design. Für »einfachst« gibt es folgende Kriterien – die Reihenfolge ist wichtig:

Das einfachste Design

1. Alle Unit Tests laufen.
2. Der Code kommuniziert alle seine Designkonzepte.
3. Der Code enthält keine Redundanz (= duplizierten Code).
4. Der Code enthält, unter Berücksichtigung der obigen Regeln, die geringst mögliche Anzahl an Klassen und Methoden.

XP verlangt ständiges Refactoring, insbesondere nach dem erfolgreichen Abschluss eines Tasks. Häufiges Refactoring ist ohne automatisierte Unit Tests kaum möglich, da sonst die Gefahr zu groß wird, mit dem Umbau am einen Ende funktionstüchtige Komponenten am anderen Ende zu beeinflussen. Diese Angst vor ungewollten Nebenwirkungen ist ein wesentlicher Grund dafür, dass viele Entwickler vor dem »Aufräumen« scheinbar funktionierender Komponenten zurückschrecken. Auf die Dauer entstehen dadurch die unwartbaren Systeme, die wir Entwickler alle kennen und vor deren Erweiterung und Anpassung uns graut.

Martin Fowler beschreibt in [Fowler99] ausführlich die häufigsten Refactoring-Maßnahmen, wie man ihre Notwendigkeit entdeckt[2], wie man sie Schritt für Schritt ausführt und wie Unit Tests das Refactoring erleichtern. Auch existieren einige Tools (z.B JFactor [URL:JFactor]), mit deren Hilfe manche grundlegenden Refactoring-Schritte automatisiert und damit fehlerfrei durchgeführt werden können. Für komplexere Umstrukturierungen ist die Korrektheit jedoch prinzipiell nicht beweisbar.

Testarten in XP

Extreme Programming proklamiert zwei Arten von Softwaretests: ***Akzeptanztests*** (***Acceptance Tests***) und Unit Tests. Während prinzipiell für beide Arten die gleichen Techniken und Tools zum Einsatz kommen können, unterscheiden sich deren Zweck und Verantwortlichkeiten:

---

2. Beziehungsweise wie man sie »erriecht«. Ein Anzeichen für verbesserungswürdigen Code nennt man nämlich auch »Code Smell«.

- **Unit Tests** sichern das Vertrauen des Entwicklers in seine eigene Software und die seiner Kollegen. Sie werden gleichzeitig mit dem Entwicklungscode erstellt und bei Bedarf verändert und ergänzt. Unit Tests müssen **immer** zu 100% erfolgreich laufen. »Immer« bedeutet: Bei der Integration von neuem Code ins System werden alle bislang erstellten Tests ausgeführt. Schlägt auch nur ein einziger Test fehl, muss zunächst dieser Fehler behoben werden, bevor man mit der Integration fortfährt. Dies ist in XP von besonderer Bedeutung, da die *fortlaufende Integration* (engl. Continuous Integration) die Eingliederung allen bearbeitenden Codes ins Gesamtsystem mehrmals täglich verlangt (siehe auch Kapitel 14.2, Seite 266 f.).

Unit Tests in XP

- **Akzeptanztests** dienen dem Kunden und dem Management als Maß für den Fortschritt des Gesamtprojekts. Sie werden vom Kunden spezifiziert – schließlich ist er derjenige, der dem Ergebnis der Testausführung glauben muss. Akzeptanztests spezifizieren typischerweise Funktionalität des Gesamtsystems aus Sicht der Anwender. Wichtig ist, dass vor Beginn einer Iteration der Großteil aller Testfälle für diesen Durchlauf spezifiziert wird. Der prozentuale Anteil erfolgreicher Tests wird mindest einmal pro Tag ermittelt und steht allen interessierten Parteien zur Verfügung.

  Die Umsetzung der Spezifikation in automatisiert ausführbare Testfälle übernehmen meist die Entwickler. An diesem Punkt ist aber auch der Einsatz eines dedizierten Testteams denkbar, das den Kunden bei der Spezifikation berät und die Durchführung der Tests übernimmt (vgl. [Crispin01]). Manchmal ist es möglich, Akzeptanztests auf die gleiche Weise zu automatisieren wie Unit Tests. Ab und an können kommerzielle Testwerkzeuge ein sinnvolles Anwendungsfeld finden. Häufig empfiehlt sich jedoch die Entwicklung eines kleinen Frameworks, mit dem die Spezifikationen des Kunden, die beispielsweise in Tabellenform vorliegen, direkt als Steuerdateien für das Ausführen von Tests verwendet werden können [URL:WakeAT].

Akzeptanztests in XP

Sowohl für Unit Tests als auch Akzeptanztests besteht die Forderung zur völligen Automatisierung. Die größere Anfangsinvestition verglichen mit manuellen Tests amortisiert sich bereits nach wenigen Ausführungen. Da Unit Tests unzählige Male pro Tag gestartet werden, ist ihre nicht automatisierte Verwendung in der Praxis undenkbar. Bei der Automatisierung mancher Arten von Akzeptanztests, z.B. der Benutzerschnittstelle, trifft man jedoch auf zahlreiche Schwierigkeiten. Bevor man jedoch resigniert auf manuelle Testausführung zurückfällt,

Testautomatisierung

sollte man sich überlegen, dass diese nicht nur auf Dauer teurer ist, sondern sich bei ihrer Abarbeitung und Verifikation auch leicht Fehler einschleichen. Vielleicht hilft ja in schwierigen Fällen die Literatur zum Thema Testautomatisierung weiter [Dustin99].

XP oder nicht XP?

Agile Prozesse

XP gehört zu der Gruppe der *agilen* Softwareprozesse[3]. Dies bedeutet unter anderem, dass so wenig vorgeschriebene Vorgehensschritte wie möglich, aber so viele wie nötig existieren. Für die in diesem Buch vorgestellte Art des Unit-Testens ist dabei eine Sache entscheidend: Es fehlt eine große, vorab durchgeführte, detaillierte Designphase – auch *BDUF* (Big Design Up-Front) genannt. Der detaillierte Softwareentwurf, insbesondere die Festlegung der Interfaces einzelner Klassen und ihre Beziehungen zu anderen Klassen, geschieht während des Kodierens bzw. der Testerstellung. Dieser evolutionäre Designansatz steht im Widerspruch zum Großteil der in der Literatur beschriebenen Entwurfs- und Testverfahren. Dort werden »up-front« ausgearbeitete Modelle und Spezifikationen aller Komponenten benötigt, um aus ihnen die Testfälle abzuleiten.

Evolutionäres Design

Ergänzende Praktiken

Zusätzlich gibt es in XP bestimmte Praktiken, die Unit Tests erleichtern (Pair Programming, Inkrementelle Entwicklung), absichern (Akzeptanztests) und auf ihnen aufbauen (Refactoring). XP ist daher keine Voraussetzung für Unit Testing, doch es lohnt sich für jeden Entwickler und Projektleiter darüber nachzudenken und auszuprobieren, ob nicht der eine oder andere Aspekt von XP die Testanstrengungen und damit die Qualität der Software verbessern könnte. Insbesondere die Mischung aus Unit Tests, Pair Programming und Refactoring bietet sich hier an und ist beinahe in jedes Vorgehensmodell integrierbar.

1.3 Testen Classic

Eine besondere Eigenart des XP-Vorgehens wurde bislang nur angedeutet: der *Test-First-Ansatz*. »Test-First« soll ausdrücken, dass der Test vor dem eigentlichen Implementierungscode geschrieben wird. Betrachten wir zunächst die Nachteile des klassischen, nachträglichen Testens an einer kleinen Programmieraufgabe:

> *Für die Übersetzung dieses Buches in eine Sprache unserer Wahl soll ein Wörterbuch programmiert werden. Dieses Wörterbuch,*

---

3. *Agil* hat vor einiger Zeit das Wort *leicht* (light-weighted) in Bezug auf Softwareprozesse verdrängt (vgl. [URL:AgileAlliance] und Kapitel 14.2).

in Form der Klasse Dictionary, *wird mit einer Wörterdatei initialisiert und erlaubt die Abfrage der Übersetzung eines deutschen Wortes. Mehrere Übersetzungsalternativen sollen möglich sein.*

Diese Anforderungsbeschreibung ist detailliert und kompakt genug, um die Programmierung in einer Iteration durchführen zu können. Fehlende Details, wie das genaue Format der Wörterdatei, definieren wir während der Programmierung. Folgende Schritte gehören (mindestens) zur »klassischen« Iteration: detailliertes Design, Implementierung und anschließend die Tests.

Als Design genügt uns hier ein UML-Klassendiagramm:

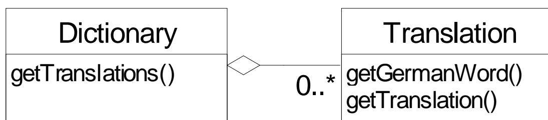

Abb. 1–1
Klassendiagramm des Wörterbuches

Unsere Implementierung der Klasse Translation besteht nur aus dem Konstruktor und zwei Getter-Methoden:

```
/**
 * Repräsentiert eine mögliche Übersetzung eines
 * deutschen Wortes
 */
public class Translation {
   private String germanWord;
   private String translation;
   public Translation(String germanWord,
                      String translation) {
      this.germanWord = germanWord;
      this.translation = translation;
   }
   public String getGermanWord() {
      return germanWord;
   }
   public String getTranslation() {
      return translation;
   }
}
```

Objekte der Klasse Dictionary erzeugen während ihrer Initialisierung Translation-Objekte und fügen diese einer internen Liste hinzu. Bei der Abfrage der Übersetzung in der Methode getTranslations() wird über diese Liste iteriert und der Ausgabestring zusammengebaut:

```
import java.io.*;
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
/**
 * Wörterbuch zur Übersetzung deutscher Wörter in eine
 * andere Sprache.
 * Das Wörterbuch wird mit einer Wörterdatei initialisiert.
 */
public class Dictionary {
   private List entries = new ArrayList();
   public Dictionary(String filename) throws IOException {
      this.initializeFromReader(new BufferedReader(
                                     new FileReader(filename)));
   }

   /**
    * Liefert die Übersetzung des deutschen Wortes.
    * Bei mehreren Alternativen werden diese
    * mit Komma aneinandergehängt.
    */
   public String getTranslations(String germanWord) {
      StringBuffer translations = new StringBuffer();
      Iterator i = entries.iterator();
      while (i.hasNext()) {
         Translation each = (Translation) i.next();
         if (each.getGermanWord().equals(germanWord)) {
            if (translations.length() > 0) {
               translations.append(", ");
            }
            translations.append(each.getTranslation());
         }
      }
      return translations.toString();
   }

   /**
    * Die zu lesende Wörterdatei besteht aus
    * 0 - n Zeilen.
    * Jede Zeile enthält einen Eintrag der Form:
    * '<deutschesWort>=<uebersetzung>'
    */
   private final void initializeFromReader(
            BufferedReader aReader)  throws IOException {
      String line = aReader.readLine();
      while (line != null) {
```

```
            int index = line.indexOf('=');
            if (index != -1) {
               String germanWord = line.substring(0, index);
               String translation = line.substring(
                                   index + 1, line.length());
               Translation entry =
                  new Translation(germanWord, translation);
               entries.add(entry);
            }
            line = aReader.readLine();
         }
      }
   }
```

So weit sieht das alles recht einfach aus, es fehlen »nur noch« die Tests. Diese können wir beispielsweise in einer separaten Klasse `DictionaryTester` ansiedeln, die in ihrer `main()`-Methode alle einzelnen Testfälle startet:

```
public class DictionaryTester {
   /**
    * Starte alle Testfälle für die Klasse Dictionary
    */
   public static void main(String[] args) {
      testfall1();
      testfall2();
      /*...*/
   }
}
```

Die Programmierung der Testfälle geschieht dann in etwa nach folgendem Schema:

1. Erzeugen einer Wörterdatei
2. Erzeugen einer Instanz der Klasse Dictionary mit dieser Datei
3. Abfrage bestimmter Übersetzungen und Überprüfen des Ergebnisses

Für den einfachen Fall einer Testdatei mit einem Wort sieht das so aus:

```
public static void testfall1() {
   String filename = "C:\\temp\\dictionary.txt";
   try {
      PrintWriter writer = new PrintWriter(
                        new FileOutputStream(filename));
      writer.println("Wort=word");
      writer.close();
      Dictionary dictionary = new Dictionary(filename);
```

```
        String translation =
                     dictionary.getTranslations("Wort");
        if (!translation.equals("word")) {
           System.out.println("Testfall 1 fehlgeschlagen." +
                       " Gefundenes Wort: " + translation);
        } else {
           System.out.println("Testfall 1 erfolgreich.");
        }
     } catch (Exception ex) {
        System.out.println("Testfall 1 fehlgeschlagen." +
                         " Unerwartete Exception.");
        System.out.println(ex.toString());
     }
  }
```

Sinnvoll wären jetzt mehrere Testfälle mit unterschiedlicher Anzahl von Einträgen in der Datei (0, 1, 2 und viele), mit identischen Einträgen, mit mehreren Übersetzungen für das gleiche Wort, Wortsuche mit groß- bzw. kleingeschriebenen Wortanfängen usw.

Dieses Vorgehen hat jedoch einige Nachteile: Jeder einzelne Testfall muss den Umweg über eine Wörterdatei gehen; das ist nicht nur umständlich, sondern kann zu diffizilen Problemen beim Erstellen und Überschreiben von Dateien führen. Das wäre vermeidbar, wenn `Translation`-Objekte auch ohne Umweg über eine Datei dem `Dictionary`-Objekt hinzugefügt werden könnten. Aber extra für den Test die Implementierung ändern und die mühsam errungene Kapselung aufgeben?

Die gleiche Frage stellt sich, wenn wir das Verhalten beim Einlesen fehlerhafter Wörterdateien testen wollen. Tritt der Fehler in der Mitte der Datei auf, so wüssten wir gerne, wie viele Übersetzungen denn bereits eingelesen wurden. Aber für diese Abfrage steht bislang keine Methode zur Verfügung. Auch haben wir uns noch nicht überlegt, wie das Wörterbuch generell bei Fehlern reagieren soll: Ignorieren? Exception werfen? Fehler ausgeben?

Hätten wir zuerst die Tests geschrieben und uns dann der Implementierung zugewandt, wären einige Probleme nicht aufgetreten:

- Über das Fehlerverhalten hätten wir uns vorher Gedanken machen müssen und eventuell fehlende Spezifikationen vorher abklären können.
- Da der Test wie jeder andere »Client« unserer Klasse behandelt wird, wären Methoden, die wir für Testzwecke benötigen, automatisch ins öffentliche Interface (`public` oder protected) gelangt.

- Bei näherer Betrachtung stellt sich die Frage, ob unsere Klasse Translation wirklich notwendig ist oder ob nicht auch die Verwendung einer HashMap genügt hätte.

In Kapitel 3 werden wir das obige Beispiel unter Verwendung des Test-First-Ansatzes neu entwickeln und dabei feststellen, dass wir zu einem anderen Design kommen, das nicht nur die Tests vereinfacht, sondern auch das Programm selbst.

1.4 »Test-First« – kleine Definition

»Test-First« (dt. »Teste zuerst«) ist eine Vorgehensweise bei der Kodierung von Softwaresystemen. Test-First ist nicht nur eine reine qualitätssichernde Tätigkeit, sondern steuert auch das Softwaredesign in Richtung Testbarkeit und Einfachheit. Folgende Punkte beschreiben das ideale Test-First-Vorgehen:

- Bevor man eine Zeile Produktionscode schreibt, entsteht ein entsprechender Test, der diesen Code motiviert.
- Es wird nur so viel Produktionscode geschrieben, wie es der Test verlangt. Mit anderen Worten: Läuft der Test, steht der Code.
- Die Entwicklung findet in kleinen Schritten statt, in denen sich Testen und Kodieren abwechseln. Eine solche »Mikro-Iteration« dauert nicht länger als 10 Minuten.
- Zum Zeitpunkt der Integration von Produktionscode ins Gesamtsystem müssen alle Unit Tests erfolgreich laufen.

Dieses kleine Regelwerk mag dem einen oder anderen Programmierer als willkürlich erscheinen und ihrer persönlichen Erfahrung widersprechen. Ziel des Buches ist es, zu vermitteln, wie Test-First-Entwicklung in der Praxis vor sich geht und dass es tatsächlich funktionieren kann. Einige Vorteile liegen auf der Hand:

- Jedes einzelne Stück Code ist getestet. Dadurch werden Änderungen, die vorhandene Funktionalität zerstören, sofort entdeckt. Dies spielt insbesondere zum Zeitpunkt der Softwareintegration eine maßgebliche Rolle.
- Die Tests dokumentieren den Code, da sie im Idealfall sowohl die normale Verwendung als auch die erwartete Reaktion in Fehlerfällen zeigen.
- Die Kürze der Mikro-Iterationen führt zu einem äußerst schnellen Feedback. In maximal zehn Minuten kann man nur wenig programmieren und daher auch nur wenig falsch machen.

- Das Design eines Programms wird maßgeblich von den Tests bestimmt. Dies führt fast immer zu einem einfacheren Design als wenn es am Reißbrett entworfen worden wäre, da für komplexe Strukturen nur selten einfache Tests geschrieben werden können. Aus diesem Grunde wird der Ansatz oft auch Test-First-*Design* genannt [URL:WikiTFD].

Ziel des Buches ist es auch, Probleme von Test-First aufzuzeigen und Hinweise zu geben, wann ein Abweichen von den Regeln akzeptabel, sinnvoll oder gar notwendig ist.

Natürliche Skepsis

Viele erfahrene Softwareentwickler, die in ihrer bisherigen Laufbahn vor allem *Design-First-Programmierung* betrieben haben und damit auch erfolgreich waren, bringen der Idee des sich schrittweise entfaltenden Entwurfs eine gesunde Skepsis entgegen. Ihnen sei empfohlen, ihre Zweifel vorübergehend zu vergessen, mit dem Ansatz für eine (nicht zu kurze) Weile zu experimentieren und erst danach ein Urteil zu fällen. Ein Phänomen lässt sich nämlich nur im Selbstversuch erfahren: Im Gegensatz zum nachträglichen Testen macht das Erstellen der Testfälle vor dem Anwendungscode tatsächlich Spaß!

1.5 Nur Java – oder auch anderen Kaffee?

Andere Programmiersprachen

Bislang war in diesem Kapitel kaum von Java die Rede. Entwickler, die andere objektorientierte Programmiersprachen verwenden und denen dieses Werk trotz des »Java« im Buchtitel in die Hände gefallen ist, stellen sich bestimmt die Frage, ob sich für sie das Weiterlesen lohnt. Unsere Antwort lautet »ja« unter folgenden Voraussetzungen:

- Sie können Java-Code lesen oder sind willens, sich die Grundbegriffe der Syntax und der Standardbibliotheken vorher oder bei Bedarf anzueignen.
- Sie sind in der Lage, von speziellen Java-Konstrukten (z.B. Interface) zu abstrahieren und diese in ihre eigene Entwicklungssprache zu übersetzen.
- Sie stören sich nicht allzu sehr an den etwa 20 Prozent dieses Buches, die wirklich nur für Java-Entwickler interessant und zutreffend sind.

In diesem Buch verwenden wir JUnit als Framework zur Testautomatisierung. JUnit ist die Java-Variante einer Familie von Unit-Testing-Tools, die jedoch auch für die meisten anderen Programmiersprachen verfügbar sind. Am Ende werden die polyglotten Nicht-Java-Entwick-

ler für das Durchhalten belohnt: In Anhang B gibt es Hinweise für Unit Tests mit anderen Programmierprachen.

1.6 Was das Buch sein möchte – und was nicht

Dieses Buch ist keine Einführung in Java oder Softwareentwicklung im Allgemeinen; Kenntnisse und Erfahrungen in beiden Bereichen werden vorausgesetzt. Es stellt auch keine systematische Einführung in das Testen objektorientierter Systeme dar; die nötige Theorie wird jedoch beleuchtet und es werden vertiefende Lesehinweise gegeben. Auch Extreme Programming wird nur insofern berührt, als es uns Hilfestellungen und Gründe für Unit Tests liefert.

Das Buch ist nur zum Teil eine Bedienungsanleitung für JUnit. Viele Probleme des praktischen Betriebs, wie Installation, Integration in die eigene Entwicklungsumgebung und andere spezielle Fragen, werden nur am Rande behandelt (siehe auch Anhang A). JUnit dient jedoch als Grundlage für die Automatisierung unserer Unit Tests (siehe Kapitel 2).

Was das Buch sein möchte, ist eine praktische Einführung ins Thema Unit Testing für Softwareentwickler. Dabei wird sowohl das grundlegende Vorgehen mittels des Test-First-Ansatzes vorgestellt als auch zahlreiche Spezialgebiete und Problemfälle behandelt. Viele der beschriebenen Techniken sind ebenso für nachträgliche Unit Tests verwendbar, manche ergeben sogar nur dort Sinn. Andere Arten von Tests, meist System- und Akzeptanztests, werden immer dann näher erläutert, wenn keine klare Abgrenzung zu Unit Tests möglich scheint. Im Idealfall soll das Buch den Entwickler beim ersten, zweiten und dritten Schritt anleiten und zum Verfolgen der zahlreichen weiterführenden Verweise motivieren.

1.7 Aufbau des Buches

Im Gegensatz zum großen Rest der Softwareentwicklungsliteratur unserer Tage wird auf ein einzelnes durchgängiges Fallbeispiel verzichtet. Dies liegt zum einen in der persönlichen Vorliebe der Autoren begründet, beim Lesen von Fachbüchern bestimmte Kapitel in wilder Reihenfolge herauszupicken; durchgängige Beispiele erschweren dabei das Nachvollziehen der Details. Zum anderen erlaubt dieser Verzicht die direkte Aufnahme authentischer Codebeispiele aus der Praxis.

Das Buch gliedert sich in zwei Hauptteile und einen Anhang. Der erste Teil – »Basistechniken« – liefert die Grundlagen für den Rest des

Buches und erfüllt seinen Zweck als Lehrbuch am besten bei chronologischer Lesefolge. Leser, die bereits mit JUnit arbeiten, können Kapitel 2 – »Automatisierung von Unit Tests« – überfliegen.

Teil 2 – »Weiterführende Themen« – enthält voneinander unabhängige Kapitel, die bei Bedarf oder Interesse zu Rate gezogen werden können. Themen sind persistente Objekte, nebenläufige und verteilte Systeme, Web-Applikationen, grafische Benutzerschnittstellen und Unit Testing im Rahmen unterschiedlicher Entwicklungsprozesse.

Als Zugabe gibt es noch einen Anhang mit JUnit-Spezifika, Tipps für Unit Tests mit anderen Programmiersprachen, einem Glossar und einem Literaturverzeichnis mit kommentierten Hinweisen auf weiterführendes Lesematerial.

1.8 Konventionen

Im laufenden Text werden neu eingeführte Begriffe kursiv geschrieben, Hervorhebungen sind **fett** gedruckt.

Das Buch besteht zu einem nicht unwesentlichen Teil aus Quellcode. Aus dem Text herausgenommene Beispiele sehen in `LetterGothic` so aus:

```
/** Quellcode Anfang
 */
public class EinBeispiel {}
```

Geänderter oder hinzugefügter Code wird zusätzlich fett gekennzeichnet. Verweise auf Code im laufenden Text werden auch in `LetterGothic` geschrieben.

Kompatibilität zu JDK 1.2 – 1.4

Die meisten Codebeispiele sind kompatibel zu JDK 1.2 – JDK 1.4. Falls eine spezielle JDK-Version benötigt wird, so findet eine Kennzeichnung am Rande und meist auch im Text statt. Die Sprache im Quelltext ist zu 99% englisch, wodurch gemischtsprachige Namen für Klassen, Methoden und Variablen vermieden werden[4].

Kodierungsrichtlinien

Um die Codebeispiele nicht unnötig aufzublasen, werden folgende Grundsätze verfolgt:

- *Import-Statements* verwenden die »Stern«-Form, sobald mehr als eine Klasse des Packages benötigt wird. Dies taugt auch als Richtlinie für Produktionscode (vgl. [Larman00]).

4. In »echten« Projekten ist Einsprachigkeit des Quellcodes oft nicht durchführbar, da es sowohl Java-Konventionen gibt, die englische Wörter verlangen, als auch Fachbegriffe, die sich nicht ohne Bedeutungsverlust ins Englische übersetzen lassen.

- Auf *Kommentare* wird verzichtet, wenn sie nicht für das Verständnis des Beispiels erforderlich sind. Insbesondere gibt es *keine JavaDoc-Kommentare*.
- Auf explizite Angabe der package-*Anweisung* wird verzichtet, wenn eine Verwechslung ausgeschlossen werden kann.

Diese Punkte dienen vor allem der Komprimierung der abgedruckten Codebeispiele. Im »echten Leben« ist es die Aufgabe jedes Teams, sich auf Kodierungsrichtlinien zu einigen – und diese dann auch zu befolgen. Wichtiger als der Inhalt der Richtlinien ist deren konsistente Verwendung.

Ungewöhnlich mag dem einen oder anderen Leser auch die explizite Verwendung von `this` bei Nachrichten an das Objekt selbst erscheinen. Diese Konvention macht jedoch den Semantikunterschied zwischen Nachrichtenverschickung und Funktionsaufruf (statische Methode) klarer.

1.9 Website zum Buch

Als Ergänzung zu diesem Buch findet man unter

`http://www.dpunkt.de/utmj/`

weitere Informationen zum Thema. Unter anderem ist dort der Quellcode aller Kapitel, eine Sammlung nützlicher Web-Links und ein ausgewählter Teil des Buches in PDF-Format verfügbar.