Ergänzungen zum Buch

Dietmar Ratz, Jens Scheffler, Detlef Seese, Jan Wiesenberger

Grundkurs Programmieren in Java

Band 1: Der Einstieg in Programmierung und Objektorientierung

Inhaltsverzeichnis

1 Strings für Fortgeschrittene 5
1.1 Reguläre Ausdrücke 5
1.2 Ein konkretes Beispiel 6
1.3 Textersetzung 8
1.4 Zusammenfassung 9

2 JUnit oder Die Kunst, fehlerfreien Code zu schreiben 11
2.1 Assertions für Fortgeschrittene 11
2.2 Unit-Tests 14
2.3 Best practices 19
2.4 Zusammenfassung 20

Literaturverzeichnis 21

Ergänzung 1

Strings für Fortgeschrittene

Wir wissen bereits, wie sich mit Hilfe der Klasse `String` grundlegende Operationen auf Zeichenketten durchführen lassen. In diesem Abschnitt werden wir uns nun mit einigen fortgeschrittenen Konzepten befassen, die seit der Version 1.4 in Java eingeführt wurden.

1.1 Reguläre Ausdrücke

Reguläre Ausruckücke stellen eine formale Sprache zur Beschreibung von Zeichenketten dar. Wer sich darunter etwas vorstellen kann, muss entweder höheres Semester Informatik oder aber ein Definitionsfanatiker sein. Lässt sich das Ganze nicht auch etwas verständlicher erklären?
Stellen wir uns einmal vor, wir kommen nach einem langen Tag nach Hause und finden eine Nachricht auf dem Anrufbeantworter vor: „Hallo, mein Name ist Herrmann Maier. Es ist meine erfreuliche Pflicht, Sie über einen Millionengewinn in der Lotterie in Kenntnis zu setzen. Bitte rufen Sie mich baldmöglichst zurück. Meine Telefonnummer ist ...“ – in diesem Moment explodiert der Anrufbeantworter.
Mit zitternden Fingern greifen wir zum Telefonbuch. Maier war der Name - wie schreibt man das doch gleich? Ist es mit „A“ oder eher mit „e“? Und wie war der Vorname noch einmal? Irgendetwas mit H, aber was?
Da wir uns nicht mehr sicher sind, wollen wir alle Möglichkeiten der Reihe nach durchgehen. Wir wissen:

- Der Nachname war Maier, wir wissen aber nicht, wie man ihn schreibt. Prinzipiell in Frage kommen also
  - Maier,
  - Meyer,
  - Mayer und

- Majer.

  Wenn man es richtig betrachtet, kennen wir eigentlich nur den ersten und die letzten zwei Buchstaben.
- Der Vorname beginnt mit einem H, aber wir können uns nicht mehr erinnern.
- wir wissen nicht, ob die entsprechende Person noch einen zweiten Vornamen hat, der im Telefonbuch geführt wird.

Glücklicherweise hat unser Telefonbuch eine CD-ROM, auf der wir nach gewissen Suchkriterien forschen können. Die Suchkriterien werden dem Computer in einem gewissen Format mitgeteilt – eben einem regulären Ausdruck. Somit ist alles klar – es handelt sich also lediglich um eine weitere Computersprache (wie Java), die wir lediglich erlernen müssen.

1.2 Ein konkretes Beispiel

Dieser Abschnitt soll keine komplette Einführung in die Welt regulärer Ausdrücke darstellen – für mehr Details empfehlen wir einen Blick in die Dokumentation der Klasse `java.util.regex.Pattern`. Wie wir anhand des folgenden Beispiels sehen werden, sind die Grundlagen allerdings nicht sonderlich schwer.
Das folgende Programm stellt eine vereinfachte Version unseres elektronischen Telefonbuchs dar:

```
1  public class Textsuche {
2
3    public static void main(String[] args) {
4      String[] namensliste = {
5        "Fritz Maier",
6        "Karl Hansen",
7        "Fred Mustermann",
8        "Horst Metzger",
9        "Hermann Meyer",
10       "Fritz H. Maurer",
11       "Hoerbi Maier",
12       "Hans Maler",
13       "Harry Mooshammer",
14       "Hurgan Malinkow"
15     };
16     String suchstring = "?????";
17     for(int i = 0; i < namensliste.length; i++) {
18       if (namensliste[i].matches(suchstring))
19         System.out.println("Gefunden: " + namensliste[i]);
20     }
21   }
22
23 }
```

In einem Feld namens `namensliste` haben wir eine Ansammlung von Namen abgespeichert – einer von ihnen ist unser Tor zu Ruhm und Reichtum. Um den

Namen zu finden, iterieren wir mit einer Schleife durch alle Feldelemente. Entspricht ein Feldeintrag unserem Suchkriterium (die Methode `matches` der Klasse `String` vergleicht einen `String` mit einem regulären Ausdruck), so geben wir den Namen auf dem Bildschirm aus.
Es verbleibt also nur noch, das Suchkriterium in der Variablen `suchstring` zu definieren. Im ersten Versuch wollem wir dem Computer lediglich mitteilen, dass im Namen ein großes M (wie Maier) vorkommen muss. Davor und danach können beliebig andere Zeichen stehen.
Wie sagen wir's dem Computer? Hierzu müssen wir wissen, dass der Punkt in einem regulärem Ausdruck für beliebige Zeichen steht. Folgt danach ein Stern, bedeutet dies „beliebig viele". Die Zeile

```
String suchstring = ".*M.*";
```

bedeutet also *beliebig viele beliebige Zeichen, dann ein großes M, danach wieder beliebig viele beliebige Zeichen*. Lassen wir unser Programm laufen, erhalten wir folgendes Suchergebnis:

Konsole

```
Gefunden: Fritz Maier
Gefunden: Fred Mustermann
Gefunden: Horst Metzger
Gefunden: Hermann Meyer
Gefunden: Fritz H. Maurer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
Gefunden: Harry Mooshammer
Gefunden: Hurgan Malinkow
```

Offensichtlich ist unser Suchmuster noch zu grob. Wir setzen deshalb noch voraus, dass der Vorname mit einem großen H beginnt:

```
String suchstring = "H.*M.*";
```

Wie wir sehen, schränkt dies die Suche schon deutlich ein:

Konsole

```
Gefunden: Horst Metzger
Gefunden: Hermann Meyer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
Gefunden: Harry Mooshammer
Gefunden: Hurgan Malinkow
```

Gehen wir nun noch davon aus, dass der Nachname (Maier) mit den Buchstaben „er" endet, also das Suchmuster

```
String suchstring = "H.*M.*er";
```

so lässt sich auch Herr Malinkow von der Suche ausschließen:

Konsole

```
Gefunden: Horst Metzger
Gefunden: Hermann Meyer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
Gefunden: Harry Mooshammer
```

Wie schränken wir unsere Namenssuche noch weiter ein? Nun, egal wie man den Namen Maier schreibt, zwischen dem *M* und dem *er* kommen immer nur zwei Buchstaben vor. In einem regulären Ausdruck können wir dies beschreiben, indem wir den Stern durch die Anzahl der Vorkommnisse (in geschweiften Klammern) ersetzen:

```
String suchstring = "H.*M.{2}er";
```

Mit dieser verfeinerten Suche lässt sich die Anzahl der Resultate so weit einschränken, dass wir sie auch leicht per Hand durchgehen können.

Konsole

```
Gefunden: Hermann Meyer
Gefunden: Hoerbi Maier
Gefunden: Hans Maler
```

1.3 Textersetzung

Wie im vorigen Kapitel bereits gesehen, unterstützt Java reguläre Ausdrücke für die Suche in Texten. Neben der Methode `match` gibt es im Paket `java.util.regex` Hilfsklassen, die auch komplexere Suchen auf Texten mit regulären Ausdrücken ermöglichen.
Doch schon mit dem Grundvorat an Hilfsmethoden in der Klassen `String` lassen sich einige erstaunliche Dinge bewerkstelligen. Die Methode `matches` ist dem Leser bereits ein Begriff. Wir wollen uns nun mit einer weiteren Möglichkeiten der regulären Ausdrücke vertraut machen: dem Ersetzen von Texten innerhalb eines Strings.
Das Ersetzen von Zeichenketten, heutzutage Standard in den einfachsten Texteditoren auf unserem Computer, war lange Zeit im Standard der Java-Sprache nicht enthalten. Ersetzen war nur auf Zeichenebene (Methode `replace`) möglich. Dies ist seit Java Version 1.4 anders geworden:

- Die Methode `replaceFirst` ersetzt *das erste* Auftreten eines regulären Ausdruckes durch einen anderen Wert. So gibt die Zeile

  ```
  System.out.println("Hund, Hemd".replaceFirst("H..d","Hand"));
  ```

  beispielsweise den Text

Konsole

```
Hand, Hemd
```

auf dem Bildschirm aus.

- Die Methode `replaceAll` ersetzt *alle* gefundenen Passagen durch einen anderen String. Die Zeile

```
System.out.println("Hund, Hemd".replaceAll("H..d","Hand"));
```

gibt also somit den Text

Konsole

```
Hand, Hand
```

auf dem Bildschirm aus.

1.4 Zusammenfassung

Reguläre Ausdrücke geben dem erfahrenen Programmierer ein mächtiges Werkzeug zur Verarbeitung von Zeichenketten in Java.[1] Neben dem Paket `java.util.regex` wurden Hilfsmethoden in die Klasse `String` aufgenommen, die Suche und Ersetzung in Texten wesentlich vereinfachen.[2] Reguläre Ausdrücke sind eine kleine Sprache an sich – wer sie verwenden will, muss sie sprechen lernen. Hat man sie jedoch einmal erlernt, lässt sich dieses Wissen auf viele andere Programmiersprachen (und auch Texteditoren wie beispielsweise **Emacs** oder **vi**) übertragen.

[1] Dies gilt auch für andere Sprachen wie Perl oder Tcl, in denen diese Ausdrücke traditionall verankert sind.

[2] Für weitere Hilfsmethoden, etwa zum Aufteilen von Text mittels der Methode `split`, sei auf die API-Dokumentation verwiesen

Ergänzung 2

JUnit oder Die Kunst, fehlerfreien Code zu schreiben

Stellen Sie sich einen Moment vor, Sie sind Teil eines Teams von Programmierern, die ein komplexes kommerzielles System erstellen. Gegen Ende des Projekts wird der verfasste Code viele Millionen Zeilen umfassen, von denen Sie selbst nicht wenige erstellt haben. Jeder von Ihnen ist für seinen Code verantwortlich und muss bei Bedarf in der Lage sein, vom Kunden gemeldete Fehler zu finden und zu eliminieren.
Um es gleich vorwegzunehmen: Dieses Kapitel wird (und kann) niemandem beibringen, keine Fehler zu machen. Softwareentwickler sind aus Fleisch und Blut und sind somit nicht vor Bugs gefeit. Allerdings gibt es gewisse Techniken, mit denen ein Programmierer dafür sorgen kann,

- dass zumindest keine offensichtlichen Programmierfehler in seinem Code stecken,
- dass er denselben Fehler nicht zweimal macht und
- dass er beim Versuch, den einen Fehler zu beheben, keine neuen Bugs in den Code einbaut[1].

Im weiteren Verlauf des Kapitels werden wir eine Bibliothek namens *JUnit* verwenden. Diese Bibliothek ist nicht Bestandteil des Standard JDK; sie kann aber kostenlos unter [5] heruntergeladen werden.

2.1 Assertions für Fortgeschrittene

Im Kapitel über Exceptions haben wir das seit Java 1.4 bestehende Schlüsselwort **`assert`** kennengelernt. Mit Hilfe dieses Konstrukts konnten wir innerhalb des

[1] Letzteres kann man natürlich nicht garantieren, aber wir können die Wahrscheinlichkeit zumindest verringern.

Programmcodes Bedingungen überprüfen, die für das Funktionieren eines Codestückes erfüllt sein mussten.
Auch das JUnit-Framework unterstützt Zusicherungen. Im Gegensatz zum **`assert`**-Schlüsselwort tut es dies allerdings durch statische Methoden in der Klasse `JUnit.framework.Assert`. Das folgende Beispielprogramm implementiert die Funktionalität von Seite 413 des Buches auf Basis von JUnit:

```
1  import junit.framework.Assert;
2  import junit.framework.AssertionFailedError;
3  import Prog1Tools.IOTools;
4
5  public class AssertionTest {
6
7    public static double kehrwert (double x) {
8      Assert.assertTrue("/ by 0", x != 0);
9      return 1 / x;
10   }
11
12   public static void main(String[] summand) {
13     double x = IOTools.readDouble("x = ");
14     try {
15       System.out.println (kehrwert(x));
16     }
17     catch(AssertionFailedError e) {
18       System.err.println(e.getMessage());
19     }
20   }
21 }
```

Beachten Sie die Änderungen:

- Anstelle des **`assert`**-Schlüsselwortes verwenden wir die Methode `Assert.assert`. Wir übergeben dieser Methode die zu überprüfende Bedingung plus einen auszugebenden Fehlertext:

  ```
  Assert.assertTrue("/ by 0", x != 0);
  ```

- Anstelle von `java.lang.AssertionError` fangen wir nun `JUnit.framework.AssertionFailedError` ab:

  ```
  catch(AssertionFailedError e) {
    System.err.println(e.getMessage());
  }
  ```

An dieser Stelle werden Sie sich wahrscheinlich zu Recht fragen: „Na und? Worin liegt der große Unterschied?“
Tatsächlich hat sich für den Programmierer bislang nicht viel geändert (außer dem Syntax). Wir haben ein von der Sprache selbst unterztütztes Schlüsselwort aus dem Programm genommen und durch einen Methodenaufruf ersetzt. Dieses bringt sowohl Vor- als auch Nachteile:

- Der Nachteil ist, dass sich die JUnit-Assertions nicht einfach abschalten lassen. Java-Assertions lassen sich schlicht und ergreifend in der virtuellen Machine

aktivieren oder deaktivieren[2]. Insbesondere in extrem rechenintensivem Code kann dies die Ausführung verlangsamen.

- Der Vorteil ist, dass JUnit-Assertions auch mit älteren Java-Versionen (also vor 1.3) funktionieren. Diese sind auch heutzutage in Projekten noch stark verbreitet – wir bleiben also abwärtskompatibel.

Diese Argumente allein rechtfertigen natürlich noch nicht den Wechsel zur Klasse `Assert`. Der Hauptvorteil dieser Klasse ist bislang allerdings noch nicht erwähnt worden: sie liefert *zusätzliche* Assertion-Methoden:

```
1  import junit.framework.Assert;
2
3  public class AssertionTest2 {
4
5    public static void main(String[] args) {
6      Assert.assertNotNull("Parameter-Arrary ist null",args);
7      // assert args != null : "Parameter-Array ist null"
8      for(int i = 0; i < args.length; i++) {
9        Assert.assertNotNull("Parameter " + i + " ist null",args[i]);
10       // assert args[i] != null : ("Parameter " + i + " ist null")
11       System.out.println(args[i]);
12     }
13   }
14 }
```

In diesem Programm geben wir den Inhalt eines übergebenen Parameter-Arrays auf dem Bildschirm aus. Damit das Programm funktioniert, nehmen wir als gegeben hin, dass das Feld und die Elemente des Feldes nicht **`null`** sind.[3] Würden wir Java-Assertions verwenden, würden wir diese Prüfung in einen boolschen Ausdruck umwandeln:

```
// assert args[i] != null : ("Parameter " + i + " ist null")
```

Wie man sieht, verringert sich die Lesbarkeit mit komplexeren Ausdrücken beträchtlich. Betrachten wir nun die Zusicherung in JUnit-Form:

```
Assert.assertNotNull("Parameter " + i + " ist null",args[i]);
```

Anhand des Methodennamens `assertNotNull` erkennen wir sofort, dass wir hier auf eine „ist nicht **`null`**"-Bedingung abprüfen. Dies erhöht die Lesbarkeit des Codes beträchtlich[4].

[2] Manche Programmierer würden übrigens argumentieren, dass es sich hierbei eher um einen Vorteil handelt. Assertions überprüfen Bedingungen im Code, die immer erfüllt sein sollten. Aus diesem Grund kann es zur Stabilität des Codes nur beitragen, sie immer angeschaltet zu lassen. Die Philosophie ist: wenn das Programm beim ersten Fehler abstürzt, kann sich der Fehler nicht weiter ins Programm fortziehen. Er kann leichter erkannt und behoben werden.

[3] Wenn Sie an dieser Stelle empört aufschreien, dies könne in einer `main`-Methode nicht vorkommen, so sind sie genau in unsere Falle getappt. Es ist korrekt, dass beim Start eines Java-Programmes ohne Argumente stets ein Feld der Länge 0 übergeben wird. Vergessen Sie aber bitte nicht, dass `main`, wie jede andere statische Methode auch, von anderen Methoden im Code direkt aufgerufen werden kann. Und in diesem Fall kann **`null`** durchaus ein übergebener Parameter sein. Für weitere interessante Beispiele zum Thema Annahme und Realität sei auf das Buch [1] bzw. [2] aus dem beigefügten Literaturverzeichnis verwiesen.

[4] Ab Java 5.0 können wir dank statischer Importe sogar das `Assert` am Anfang weglassen.

Die Klasse `Assert` beinhaltet folgende überprüfbare Zusicherungen:

- `assertTrue` prüft, ob ein boolscher Ausdruck wahr ist.
- `assertFalse` prüft, ob ein boolscher Ausdruck unwahr ist.
- `assertEquals` überprüft zwei Objekte (oder primitive Datentypen) auf Gleichheit.
- `assertSame` überprüft, ob zwei Variablen dasselbe Objekt referenzieren (oder beide **`null`**) sind.
- `assertNotSame` überprüft, ob zwei Variablen unterschiedliche Objekte referenzieren.
- `assertNull` überprüft, ob eine Variable **`null`** referenziert.
- `assertNotNull` überprüft, ob eine Variable *nicht* **`null`** referenziert.

Jede der assert-Methoden in JUnit ist überladen, sodass sie sich mit oder ohne Fehlermeldung einsetzen lässt. Die Methode `assertEquals` ist ferner so überladen, dass sie sowohl primitive Datentypen als auch Objekte verarbeiten kann.

2.2 Unit-Tests

Nehmen wir einmal an, wir haben folgende Klasse geschrieben:

```
1  public class Datenklasse {
2
3    private String name;
4
5    public String getName() {
6      return name;
7    }
8
9    public void setName(String name) {
10     this.name = name;
11   }
12
13   public String toString() {
14     return name.toString();
15   }
16
17 }
```

Die Klasse besitzt eine Instanzvariable `name`, und get/set-Methoden für den Zugriff. Ferner haben wir eine `toString`-Methode verfasst, um ein Objekt beispielsweise auf der Konsole ausgeben zu können.
Wie können wir nun sichergehen, dass der von uns verfasste Code auch funktioniert? Traditionelles Vorgehen vieler Programmierer ist es, eine `main`-Methode in den Code einzubauen:

```
public static void main(String[] args) {
  Datenklasse d = new Datenklasse();
```

```
    Assert.assertNull(d.getName());
    d.setName("test");
    Assert.assertNotNull(d.getName());
    Assert.assertEquals("test",d.getName());
  }
```

Mit anderen Worten, wir erweitern unsere Klassen um eine Testmethode. Dieses Vorgehen hat allerdings verschiedene Nachteile:

- Je komplexer eine Klasse wird, desto mehr Tests müssen durchgeführt werden. Die `main`-Methode wird äußerst lang und unübersichtlich werden.
- Die Testlogik ist in die Klasse selbst verdrahtet und somit jedermann zugänglich. Je mehr Klassen auf diese Art und Weise programmiert werden, desto mehr `main`-Methoden werden in der Anwendung herumschwirren.
- Falls die Klasse in einem größeren Kontext benutzt werden muss, z.B. in Verbindung mit einer Datenbank, kann das Vorbereiten des Tests die Klasse sogar noch weiter aufblähen.
- Das Aufrufen einer `main`-Methode ist gut und schön für einen einzelnen Test – im Rahmen eines großen Programmierprojektes ist es allerdings sehr schwer, derartige Tests zu automatisieren und auf regelmäßiger Ebene ablaufen zu lassen.

Wir wollen aus diesem Grund Abstand von Tests in einer Main-Methode nehmen und stattdessen sogenannte Unit-Tests verwenden. Unit-Tests sind Klassen, die sich von der Klasse `JUnit.framework.TestCase` ableiten. Wir betten unseren Test nun in eine solche Klasse ein:

```
1  import junit.framework.*;
2
3  public class DatenTest extends TestCase {
4
5    public DatenTest(String name) {
6      super(name);
7    }
8
9    public void testGetUndSetName() {
10     Datenklasse d = new Datenklasse();
11     Assert.assertNull(d.getName());
12     d.setName("test");
13     Assert.assertNotNull(d.getName());
14     Assert.assertEquals("test",d.getName());
15   }
16
17 }
```

Beachten Sie, dass die Klasse `DatenTest` einen Konstruktor mit einem `String` besitzt. Dies sollte für alle Testklassen der Fall sein.
Wir stellen nun sicher, dass sich die JUnit-Bibliothek in unserem Klassenpfad befindet und rufen

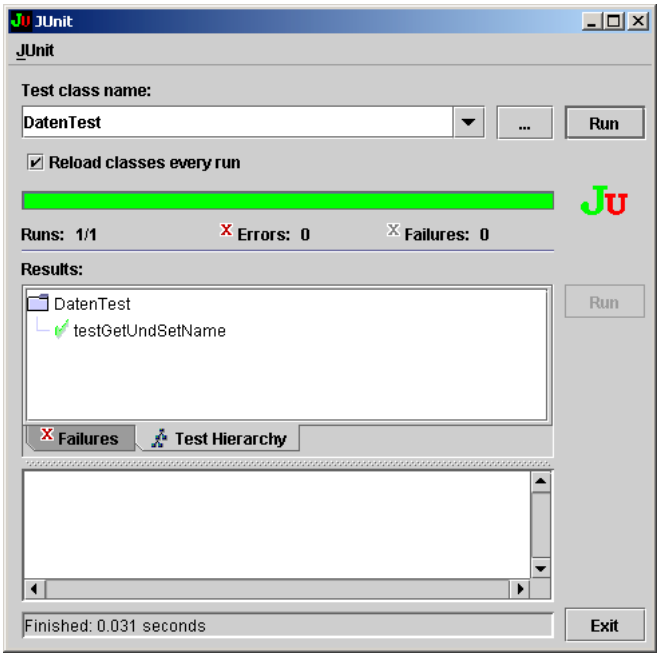

Abbildung 2.1: Der TestRunner im Einsatz

Konsole

```
java JUnit.swingui.TestRunner DatenTest
```

auf. Abbildung 2.1 stellt das Ergebnis dieses Programmaufrufs dar. Wir haben den JUnit Testrunner aufgerufen, eine von mehreren Benutzeroberflächen der Bibliothek. Der Name unserer Testklasse wurde als Parameter übergeben. Nachdem die Anwendung gestartet wurde, durchsuchte diese die Klassendefinition unseres Tests nach Testmethoden (alle Methoden, die mit dem Wort „test“ starten). Anschließend wurden diese Tests automatisch durchgeführt – das grüne Licht zeigt an, dass sie erfolgreich durchlaufen wurden.
Nehmen wir nun an, wir haben eine Beschwerde von einem unserer Kollegen erhalten. Er hat die Datenklasse verwendet, und diese hat eine `NullPointerException` in der Methode `toString` produziert. Ungläubig ewrweitern wir unsere Testklasse um eine weitere Methode:

```
public void testToString() {
  Datenklasse d = new Datenklasse();
  d.toString();
  d.setName("test");
  d.toString();
}
```

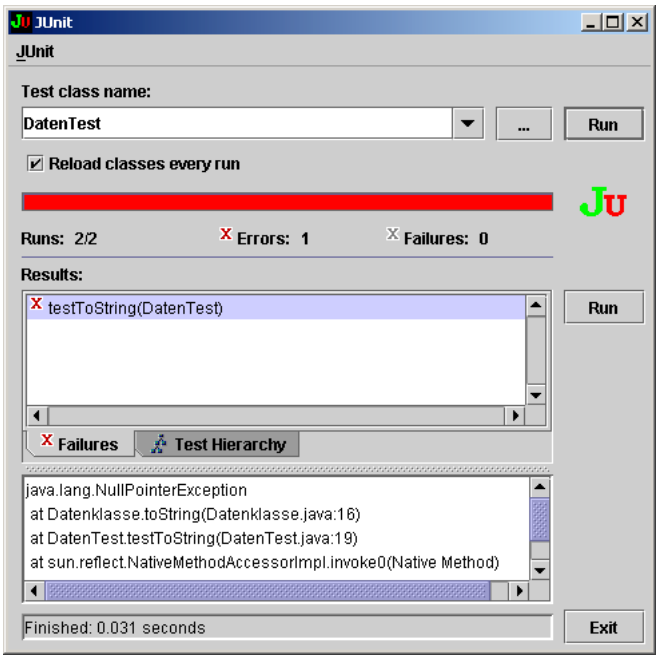

Abbildung 2.2: Der Test schlägt fehl

Wir rufen den Testrunner auf und tatsächlich – unsere neue Testmethode schlägt fehl (siehe Abbildung 2.2). Nun, dies sollte doch einfach zu reparieren sein: wir initialisierten einfach unser Feld mit irgendeinem Wert:

```
1  import junit.framework.Assert;
2
3  public class Datenklasse {
4
5    private String name = "???"; // NEU!!!
6
7    public String getName() {
8      return name;
9    }
10
11   public void setName(String name) {
12     this.name = name;
13   }
14
15   public String toString() {
16     return name.toString();
17   }
18
19 }
```

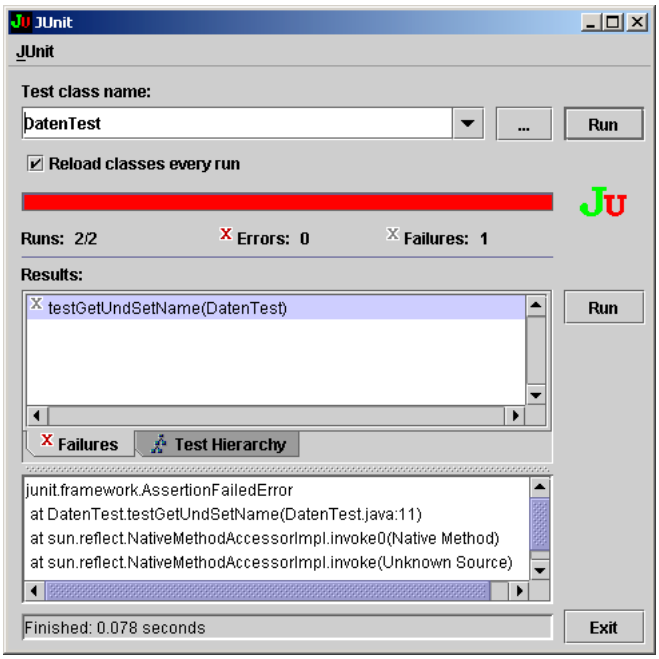

Abbildung 2.3: Ein unerwünschter Nebeneffekt

Kaum lassen wir den Test ein weiteres mal laufen (siehe Abbildung 2.3), trifft es uns wie ein Blitz: zwar funktioniert unsere Methode `testToString` jetzt reibungslos, aber dafür liefert die (bislang tadellose) Methode `testGetUndSet` auf einmal einen Assertion-Fehler!
Auch wenn dieses Beispiel stark vereinfacht erscheint, so macht es dennoch den Vorteil von Unit-Tests und Testbibliotheken klar: während wir versucht haben, ein aufgetretenes Problem zu beheben, haben wir einen neuen Bug in unsere Klasse eingebaut. Da wir jedoch bereits einen Test für eben diese Funktionalität hatten (nämlich die Methode `testGetUndSet`, wurden wir vom Computer auf dieses Problem automatisch aufmerksam gemacht). Wir wissen, dass unser Lösungsansatz offensichtlich sch;echt durchdacht war, und können nach einem besseren Lösungsweg suchen[5].
Wie sich denken lässt, ist das JUnit Framework noch weitaus vielschichtiger, als in diesem Beispiel gezeigt. So besitzt die Klasse `TestCase` noch zwei Methoden `setUp` und `tearDown`, um vor Beginn und Ende eines Tests gewisse Vorbereitungen zu treffen (z.B. die Verbindung zu einer Datenbank). Auch besitzt das Framework die Möglichkeit, meherere Testklassen zu einer `TestSuite` zusammenzubinden oder einfach automatisch nach allen Unit-Tests innerhalb des Klassenpfa-

[5]Machen Sie sich dies am besten zur Übungsaufgabe. Wie muss man die Klasse verändern, damit beide Tests funktionieren?

des zu suchen. Für dies und mehr sei hier auf die hervorragende Dokumentation auf `http://www.JUnit.org/` verwiesen.

2.3 Best practices

Die folgenden „goldenen Regeln“ beschreiben Vorgehensweisen, die die Autoren im täglichen Umgang mit Softwareentwicklung als wertvoll empfunden haben. Wir erheben natürlich keinen Anspruch auf Vollständigkeit – es gibt verschiedene „Schulen“ der Programmierkunst (siehe beispielsweise [7]), die unterschiedliche Herangehensweisen an Unit-Tests (falls überhaupt) empfehlen. Wahrscheinlich muss ein jeder seinen eigenen Weg finden...

Regel 1: *Write a little, test a little.*

Niemand schreibt wirklich gerne Tests. Andererseits gibt es kaum etwas deprimierenderes, als tagelang vor einem Stück Code zu sitzen und den Fehler zu suchen. Oftmals schreibt man ein Programm Klasse für Klasse und diese wiederum Methode für Methode. Warum nicht für jede neue Klasse eine neue Testklasse anlegen, für jede neue Methode (oder jedes neue zusammenhängende Stück Funktionalität) gleich den zugehörigen Test schreiben? Auf diese Weise kommen Fehler schnelllstmöglich ans Licht, und die Testbibliothek wächst quasi von allein. Last but not least gibt es kaum ein besseres Gefühl als die Gewissheit, dass der Code am Ende des Tages auch lauffähig sein wird.

Regel 2: *Better some test as no test.*

Viele Leute sind der Meinung, sie hätten keine Zeit für Unit-Tests. Das Projekt ist in Bälde fällig, und jede zusätzliche Zeile Code hält davon ab, Funktionalität zu implementieren.
Unglücklicherweise sind es oftmals diese Leute, die dann Nachtschicht über Nachtschicht lang nach Fehlern in ihren Sourcen suchen, und sie nicht finden. Die Zeit, die sie am Anfang gespart haben, zahlen sie am Ende doppelt und dreifach in der Testphase drauf.
Falls sich der Leser einmal in einer ähnlichen Situation befinden sollte, sei bitte auf oben stehende Regel verwiesen. Es ist immer noch besser, zumindest die kritischen Teile (jener Code, von dem wir sowieso befürchten, dass sich Fehler einschleichen könnten) mit Tests zu versehen, als auf die eigene Unfehlbarkeit zu vertrauen.

Regel 3: *Test first, then fix.*

Was ist der erste Schritt, wenn beim Entwickler ein Bug-Report eintrifft? Den Fehler suchen und beheben? Falsch! Der allererste Schritt sollte es sein, das Problem zu reproduzieren - und zwar automatisch. Wenn man einen JUnit-Test dafür schreiben kann, weiß man eines mit Gewissheit: wenn der Fehler erst behoben ist, tritt er nie wieder auf!

Regel 4: *Nightly builds.*

Diese Regel klingt vielleicht ein wenig extrem, sie wird aber in vielen grossen Programmierprojekten (von Open Source bei Apache bis hin zum Softwaregiganten Microsoft) mit großem Erfolg eingesetzt.
In vielen großen Programmierprojekten mit vielen Entwicklern wird ein Zyklus für Komplettübersetzungen des Projekts durchgeführt. Dies beinhaltet

- das Kompilieren aller Klassen (gibt es irgendwelche Übersetzungsfehler),
- das Durchführen aller existierenden Unit-Tests gegen den frisch übersetzten Code und
- das Überprüfen der Fehlermeldungen und die Benachrichtigung[6] der Verantwortlichen.

Im (anstrebenswerten) Extremfall findet dieser „Build“ jede Nacht statt, nachdem die Entwickler nach Hause gegangen sind. Der nächtliche Build ist eine hervorragende Methode, um frisch aufgetretene Unverträglichkeiten zwischen Modulen so schnell wie möglich zu erkennen. Ferner gibt es dem Team ein erhöhtes Maß an Zuversicht, wenn die nächste Deadline unaufhaltbar näher rückt.

2.4 Zusammenfassung

JUnit is eine Klassenbibliothek für das Verfassen und kontrollierte Durchführen automatischer Software-Tests. Auch wenn nicht Bestandteil des Standard-JDK, handelt es sich doch um eine der verbreitesten und beliebtesten Bibliotheken in der Java Software-Entwicklung. JUnit beinhaltet das Test-Framework plus verschiedene Umgebungen (sowohl Konsole als auch graphisch.[7] für die Ausführung der Tests.
Dieses Kapitel hat einen kurzen Einblick in die Möglichkeiten automatisierter Tests gegeben. Das Konzept derartiger Tests ist heutzutage weit verbreitet und sollte dem angehenden Entwickler kein Fremdbegriff sein.

[6] Dies kann in verschiedenen Formen und mit unterschiedlichen Konsequenzen stattfinden. Ein großes Softwarehaus beispielsweise hat es sich zu Eigen gemacht, den Verursacher mit der zukünftigen Auswertung der nächtlichen Builds zu beauftragen. Mit anderen Worten: Er muss jeden Morgen zu seiner zusätzlichen Arbeit die Auswertungen durchführen, bis ein anderer den allnächtlichen Test verfehlt und das Ruder übernimmt.

[7] Die Entwicklungsumgebung Eclipse besitzt übrigens hervorragende eingebaute Unterstützung für JUnit

Literaturverzeichnis

Bücher

[1] A. Hunt, D.Thomas: *The Pragmatic Programmer*. Pearson Education, 1999.

[2] A. Hunt, D.Thomas: *Der Pragmatische Programmierer*. Hanser, 2003.

[3] V. Massol, T. Husted: *Junit in Action*. Manning Publications, 2003.

Internet-Links

[4] *Regular Expression Tutorial.*
`http://www.regular-expressions.info/tutorial.html`

[5] *JUnit Homepage.*
`http://www.JUnit.org`

[6] J. Scheffler: *Buchempfehlungen zum Thema Software.*
`http://www.book-fan.com/listings/software.htm`

[7] *Extreme Programming.*
`http://www.extremeprogramming.org`