

HANSER

Leseprobe

Michael Johann

Ruby on Rails für JEE-Experten

Umfassender Einstieg in Rails und JEE-Integration mit JRuby

ISBN: 978-3-446-41535-5

Weitere Informationen oder Bestellungen unter

<http://www.hanser.de/978-3-446-41535-5>

sowie im Buchhandel.

## 3 Testgetriebene Entwicklung

### 3.1 Einführung

---

Ruby on Rails erlaubt sehr agile Vorgehensweisen. Agile Prozesse sind in aller Munde und in mittleren bis großen Projekten gewinnbringend eingesetzt worden. Grob zusammengefasst, stellt die agile Softwareentwicklung eine wesentlich flexiblere Variante eines Entwicklungsprozesses dar als die üblicherweise sehr langfristig und weitgehend geplanten klassischen Projektvorgehensweisen.

Bereits aus dem Bereich des XP (eXtreme Programming) sind viele Konzepte entstanden wie beispielsweise die Iterationen und Inkremente sowie die starke Integration des Kunden und seine Verantwortlichkeit im gesamten Prozess.

Neben einigen anderen Praktiken ist ein Konzept aus der agilen Welt die testgetriebene Entwicklung. Testgetrieben bedeutet hier, dass zuerst die Tests geschrieben werden und dann die eigentliche Software, die die Funktionalitäten implementiert.

Es gibt unterschiedliche sich ergänzende Testkonzepte, die ich Ihnen hier näher vorstellen möchte. Im weiteren Verlauf des Buches werden Sie auch immer wieder sehen, dass diese Tests zuerst implementiert werden und danach erst die eigentliche Funktionalität.

Folgende Testkonzepte werden in diesem Kapitel vorgestellt:

- Unittests
- Funktionale Tests
- Integrationstests
- Spezifikationstests
- User Stories
- Lasttests

### 3.1.1 Vorteile

Vielleicht ist Ihnen noch nicht klar, warum alle von Tests reden, und Sie fragen sich, was die Vorteile sind. Einige Vorteile will ich Ihnen hier kurz erläutern.

#### Qualitativ besserer Code

Wenn Sie Tests schreiben, und das auch noch vor der Implementierung, werden Sie feststellen, dass die Tests vorgeben, wann eine Funktionalität komplett abgeschlossen und funktional ist.

Voraussetzung hierfür ist sicher, dass die Sinnhaftigkeit der Tests und eine adäquate Testabdeckung gegeben sind. Weil Testläufe schnell aussagen können, ob noch Fehler existieren oder bisher lauffähige Funktionen nun mit Fehlern behaftet sind, lässt sich viel schneller beurteilen, ob noch Arbeiten zu erledigen sind. Auch bei Refaktorisierungen von vorhandenem Code sagen Tests schneller aus, ob die Software immer noch so funktioniert wie vor den Umstrukturierungen.

Statistische Auswertungen von Tests aller Art sind gerne auch im Management und beim Kunden gesehen, wobei die Deutung der Ergebnisse durchaus auch zu negativen Interpretationen führen kann. Beispielsweise kann ein Kunde eine Fehlerzahl von 300 als erschreckend hoch einstufen, während die Entwickler und die Projektleitung eine positive Einschätzung abgeben.

#### Höhere Produktivität

Wer mehr über die Fehlersituation seiner Software weiß, wird sich schneller neuen Aufgaben widmen können. Dadurch stellt sich üblicherweise ein Gefühl der Zufriedenheit ein, was bei Projekten mit Ruby on Rails besonders zu beobachten ist. Das steigert die Motivation ungemein und erhöht wiederum die Produktivität.

Aber auch hier sind Eindrücke unterschiedlich interpretierbar. Während die Entwickler zufrieden nach Hause gehen, überlegen sich die Manager, welche zusätzlichen Features man noch *mal eben* einbauen könnte, die erst nachträglich mit dem Kunden diskutiert wurden.

#### Ziele lassen sich besser erreichen

Ziele, die vorgegeben sind, lassen sich besser erreichen. Anforderungen sind leichter zu überprüfen und bieten damit eine Grundlage für die Abnahme der beauftragten Software. Die Erreichbarkeit dieser Ziele kann durch die Tests sicher besser gemessen werden als ohne, doch sollten Kunden und Projektleitung ehrlich sein und weitere Ziele nicht einfach so von der Seite *hereinreichen*.

#### Leichtere Wartbarkeit

Tests haben die Eigenschaft, einen guten Überblick über die Schnittstellen der zu testenden Software zu bieten. Damit dokumentieren Tests auch sehr gut, was die Einarbeitung neuer Mitarbeiter vereinfacht. Wartungsmitarbeiter finden sich wesentlich schneller im Code zurecht, wenn sie sehen, wie die Tests mit dem Code umgehen.

Die Syntax von Ruby ist des Weiteren ein schönes Beispiel für lesbaren Code, der an vielen Stellen auch die Dokumentation vereinfacht. Bei der Entwicklung von Spezifikationen mit rSpec beispielsweise kann in weiten Teilen auf eine separate Dokumentation verzichtet werden, da rSpec mit einer eigenen domänenspezifischen Sprache ausgestattet ist und somit die Lesbarkeit um einiges verbessert.

### 3.1.2 Unittests

Vielleicht kennen Sie aus der Java-Welt das Unittests unterstützende JUnit-Framework<sup>1</sup>. Wenn Sie noch keine Unittests geschrieben haben, werden Sie zumindest davon gehört haben. Das Vorhandensein solcher Unit-Tests ist allerdings keine Selbstverständlichkeit. Wenn ich in Projekten nach solchen Tests frage, erhalte ich leider allzu oft die Antwort, dass man diese Tests dringend bräuchte, aber leider nie das Budget dafür eingeplant wurde. Oder die Entwickler finden keinen Einstieg und wissen gar nicht, was sie testen sollen. Oft genug werden die Tests auch für die Zeit nach der Implementierung eingeplant. Dabei sind Unittests etwas, was jeder Entwickler vor Beginn der Entwicklung lokal implementieren und nutzen sollte. Damit können Änderungen im Code zeitnah nachgetestet und so die Qualität des Ergebnisses nachhaltig gesteigert werden.

Unittests haben den Zweck, die kleinste Einheit im System zu testen. Damit sind einzelne Klassen gemeint, deren Schnittstelle nach außen überprüft werden soll.

Der Grund dafür, dass die Tests vor dem eigentlichen Code implementiert werden sollten, ist, dass man sonst die Tests um den vorhandenen Code herum schreibt und nicht das testet, was die Spezifikationen vorgeben.

Treffender gesagt: man läuft Gefahr, zu testen, was man programmiert hat und wie es aktuell funktioniert. Dabei wollte man eigentlich die Einheit so testen, als würde sie leisten, was man von ihr erwartet. Alle Klarheiten beseitigt?

Wie dem auch sei, es kommt eher selten vor, dass sich ein Entwickler zuerst mit dem Testen beschäftigen darf. Zu oft sind schnelle Ergebnisse gefragt, und die Tests – ebenso wie die Dokumentation – stehen deswegen hinten an und werden vielleicht nie angefertigt.

Sie werden sehen, dass es wesentlich effektiver ist, die Tests vor der eigentlichen Programmierung zu schreiben. Daher soll diesem Ansatz auch hier Rechnung getragen werden. Ruby als Programmiersprache macht es Entwicklern übrigens leichter, Tests zu schreiben, da beispielsweise kein Compiler in der Nähe ist, der auf fehlende Definitionen hinweist.

Unittests beziehen sich, wie der Name vermuten lässt, auf eine Einheit, was in diesem Umfeld eine Klasse ist. Somit wird das Verhalten von Klassen in einem Unittest geprüft oder besser: verifiziert. Üblicherweise werden hier die Schnittstelle einer Klasse herangezogen und die Vor- und Nachbedingungen formuliert.

---

<sup>1</sup> Siehe auch <http://www.junit.org>

Ruby on Rails bietet von Haus aus schon in Projekten integrierte Unittests, die sich generieren und dann anpassen lassen. Somit steht einem schnellen Start ins Testen mit Unittests nichts mehr im Wege.

### 3.1.3 Funktionale Tests

Bei den funktionalen Tests (auch Integrationstests genannt) handelt es sich um Tests, die einzelne Funktionen einer Software testen. Dabei wird die vollständige Funktion getestet, die durchaus von mehreren Units bereitgestellt werden kann. Beispiele für funktionale Tests sind Vorgänge wie das Anlegen von Kunden oder das Übergeben von Daten an einen Transportdienstleister.

Da sich der Anspruch der funktionalen Tests von den Unittests unterscheidet, werden auch in Ruby on Rails die funktionalen Tests anders gehandhabt und in einer anderen Struktur verwaltet.

## 3.2 Test::Unit

---

Dieser Abschnitt beschreibt einführend die Möglichkeiten von Unittests und funktionalen Tests in Ruby on Rails. Den Anfang machen die Modellklassen testenden Unittests.

### 3.2.1 Unittests

Unittests ermöglichen es, sich mit der Sicherung der Qualität von Software zu befassen. Wenn Sie JUnit kennen, sind Ihnen auch die dort verwendeten Begriffe, Klassen und Konzepte bekannt.

Tests werden in Testsuiten zusammengefasst. Ein Unittest wird von `TestCase` abgeleitet, und die Namen der Testmethoden beginnen mit dem Wort `test`. Das folgende Beispiel zeigt einen Fall aus der Java-Welt.

**Listing 3.1** <http://www.pastie.org/234032>

```
import addierer.Addierer;
import junit.framework.Assert;
import junit.framework.TestCase;

public class AddiererTest extends TestCase {
    public void testAddition() {
        Integer one = new Integer(47);
        Integer two = new Integer(11);

        Integer expected = new Integer(58);

        // Funktionalität testen
        Integer result = Addierer.add(one, two);
        Assert.assertTrue(expected.equals(result));
    }

    public AddiererTest(String name) {
        super(name);
    }
}
```

Die zu testende Klasse `Addierer` stellt lediglich eine Methode `add` zur Verfügung, die zwei `Integer`-Objekte nimmt, den `int`-Wert ausliest und dann addiert. Nach der Addition wird ein neues `Integer`-Objekt erstellt und zurückgegeben. Das folgende Listing zeigt den `Addierer`.

**Listing 3.2** <http://www.pastie.org/234033>

```
package addierer;

public class Addierer {
    public static Integer add(Integer one, Integer two) {
        return new Integer(one.intValue() + two.intValue());
    }
}
```

Eine vergleichbare Situation ist mit Ruby schnell programmiert und verdeutlicht die Zusammenhänge für Ruby-Entwickler.

Zunächst ein einfacher `Addierer`.

**Listing 3.3** <http://www.pastie.org/234035>

```
class Addierer
  def self.add(one, two)
    one + two
  end
end
```

Sie sehen, dass auch hier zwei Parameter übergeben und dann addiert werden. Eine `Return`-Anweisung wird in Ruby nicht benötigt, da immer der Wert der letzten Operation zurückgegeben wird.

Die folgenden Zeilen zeigen eine passende Testklasse in Ruby.

**Listing 3.4** <http://www.pastie.org/234036>

```
require 'test/unit'
require 'addierer'

class AddiererTest < Test::Unit::TestCase
  def test_add
    one, two = 47, 11
    expected = 58

    result = Addierer.add(one, two)
    assert_equal(result, expected)
  end
end
```

Die Klasse `AddiererTest` wird von `Test::Unit::TestCase` abgeleitet und definiert eine Methode `test_add`. Diese Methode definiert zwei Variablen `one` und `two` mit den Werten 47 und 11. Eine weitere lokale Variable `expected` enthält den Vergleichswert für das zu errechnende Ergebnis (58).

Danach wird die Klassenmethode `add` der Klasse `Addierer` aufgerufen, die die beiden Parameter `one` und `two` als Übergabeparameter erhalten. Das Ergebnis der Berechnung wird dann in der lokalen Variablen `result` hinterlegt.

Mit der Anweisung `assert_equal`, wird überprüft, ob die beiden Übergabeparameter vom Wert her gleich sind. Ist dies der Fall, wird der Testlauf als fehlerfreier Durchlauf gewertet. Das Ergebnis des Testlaufs sieht dann so aus.

```
Loaded suite addierer_test
Started
.
Finished in 0.000829 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Wenn Sie nun diesen Unittest so abändern, dass ein Fehler auftritt, unterscheidet sich die Ausgabe entsprechend. Ändern Sie also beispielsweise den Wert von `expected` auf 100 ab, erhalten Sie folgende Informationen.

```
Loaded suite addierer_test
Started
F
Finished in 0.017331 seconds.

1) Failure:
test_add(AddiererTest) [addierer_test.rb:11]:
<58>_expected but was
<100>.

1 tests, 1 assertions, 1 failures, 0 errors
```

Die Ausgabe informiert Sie nun darüber, dass in der Zeile 11 ein Fehler aufgetreten ist. Es wurde der Wert 100 erwartet, aber 58 zurückgeliefert. Abgesehen davon, dass dies mathematischer Unsinn ist, sollte nur klar sein, was ein `assert_equal` bewirkt.

Mit diesen einführenden Informationen sollten Sie einen kurzen Überblick über die Schreibweise von Unittests erhalten. Damit weiterführende Konzepte für Unittests einen Sinn ergeben, folgen nun einige Hinweise zu Fixtures, YAML-Dateien, Rake und Assertions.

#### 3.2.1.1 Fixtures

Um Unittests sinnvoll ablaufen lassen zu können, werden auch einige Testdaten benötigt. Fixtures liefern solche Daten in Form von YAML-Dateien. YAML<sup>2</sup> ist ein für Menschen wesentlich besser lesbares Format als XML und bietet im Ruby on Rails-Bereich eine sehr flexible Möglichkeit, Daten auch zur Laufzeit zu generieren und beispielsweise für Testläufe zur Verfügung zu stellen. Beispielsweise sind die Konfigurationsdaten für Datenbankverbindungen einer Ruby on Rails-Anwendung in YAML beschrieben.

Die folgenden Zeilen Code zeigen für den Unittest bereitgestellte Fixtures der Klasse `Feed`, die Sie vielleicht noch aus Kapitel 1 kennen.

```
rails-experts:
  id: 1
  name: rails-experts.com
  url: http://www.rails-experts.com
  description: Deutschsprachige Seite für Ruby on Rails
  category_id: 2
```

---

<sup>2</sup> Yet another markup language

```

rubyonrails:
  id: 2
  name: RubyOnRails
  url: http://www.rubyonrails.org
  description: Die Heimat von Ruby on Rails
  category_id: 1

```

Die Fixtures für die Klasse `Feed` sollten in der Datei `test/fixtures/feeds.yml` abgelegt werden. Wie zu sehen ist, bildet YAML seine Struktur mit Hilfe von Namen und Werten (Key-Value-Paare) ab. Dieses Konzept ist allgemein aus Hash-Tabellen bekannt und ermöglicht es, Daten bequem in Hash-Objekten zu verwalten. So kann beispielsweise sehr einfach eine Instanz von `Feed` erstellt werden, ohne viel Code schreiben zu müssen. Die Einrückungen machen das Ganze besser lesbar.

In der Datei `feeds.yml` werden zwei Objekte namens `rails-experts` und `rubyonrails` definiert. Die nachfolgenden Attribute beschreiben die Instanzen näher. Die einzelnen Attributwerte mappen direkt auf die Spalten der Tabelle `feeds` in der Datenbank.

Die Fremdschlüssel der zugehörigen Kategorie sind im Attribut `category_id` angegeben. Die referenzierten Kategorien werden folgendermaßen definiert und finden sich entsprechend in einer Datei `test/fixtures/categories.yml`.

```

entwicklung:
  id: 1
  name: Entwicklung
  description: Alles rund um Softwareentwicklung

nachrichten:
  id: 2
  name: Nachrichten
  description: News, Blogs etc.

```

### YAML-Dateien einlesen

An dieser Stelle zeige ich Ihnen einige Funktionalitäten, die Ihnen den Umgang mit YAML-Dateien näher bringen und erleichtern sollen.

YAML-Dateien sind also Dateien, die Daten für Instanzen von Modellklassen enthalten. Diese Daten werden beispielsweise zur Laufzeit in Tests verwendet. Wenn Sie allerdings eigene Objekte erzeugen wollen, die Daten aus YAML-Dateien erhalten, können Sie dies sehr einfach mit einigen Zeilen Ruby-Code erledigen.

Für die nächsten Schritte sollten Sie die Dateien `feeds.yml` und `categories.yml` im Verzeichnis `test/fixtures` anpassen. Hierzu geben Sie einfach entsprechend den beiden vorherigen Listings Objektdaten in die Fixtures ein.

Starten Sie für die nächsten Versuche bitte die Rails-Console, indem Sie in der Kommandozeile im Hauptverzeichnis des Feeder-Projektes aus Kapitel 2 Folgendes eingeben.

```
$ ./script/console
```

Die Daten aus den Fixtures können genutzt werden, um daraus Instanzen von Modellklassen zu erstellen. Diesen Vorgang kann man in Tests nutzen oder auch direkt einsetzen, um Daten zu importieren.

Geben Sie nun folgende Zeile ein, und Sie erhalten ein Objekt mit den beiden Kategorien aus der Datei `categories.yml`.

```
>> model_data = YAML::load_file('test/fixtures/categories.yml')
```

Daraufhin meldet die Console Folgendes.

```
=> {"nachrichten"=>{"name"=>"Nachrichten", "id"=>2, "description"=>"News, Blogs etc."}, "unterhaltung"=>{"name"=>"Entwicklung", "id"=>1, "description"=>"Alles rund um Softwareentwicklung"}}
```

Die Darstellung zeigt ein Hash-Objekt, das es nun erlaubt, einzelne Daten für die Erstellung von `Category`-Instanzen zu extrahieren. So gibt der folgende Aufruf ein einzelnes Hash-Objekt zurück, das als Datenlieferant für eine Instanz von `Category` verwendet werden kann.

```
>> model_data['nachrichten']
```

Das Ergebnis:

```
=> {"name"=>"Nachrichten", "id"=>2, "description"=>"News, Blogs etc."}
```

Wenn das Ergebnis als Übergabeparameter für die `new`-Methode der Klasse `Category` genutzt wird, wird eine Instanz von `Category` im Speicher angelegt.

```
>> c = Category.new(model_data['nachrichten'])
```

Das Objekt `c` hat dann alle Eigenschaften einer Instanz von `Category`. Lediglich die Felder `created_at` und `updated_at` sind nicht befüllt, weil das Objekt diese Attribute erst beim Speichern in der Datenbank erhält.

Zur Überprüfung, ob dies ein noch nicht in der Datenbank enthaltenes neues Objekt ist, können Sie in der Console folgende Abfrage eingeben:

```
>> c.new_record?  
=> true
```

Es wird der Wert `true` zurückgegeben und signalisiert damit, dass das Objekt noch nicht persistent gemacht wurde.

```
=> #<Category id: nil, name: "Nachrichten", description: "News, Blogs etc.",  
created_at: nil, updated_at: nil>
```

Wenn Sie die Instanz in die Datenbank schreiben möchten, geben Sie folgende Anweisung auf der Console ein.

```
>> c.save
```

Jetzt sind die Attribute `created_at` und `updated_at` ebenfalls befüllt, was Sie leicht mit folgendem Aufruf überprüfen können.

```
>> c.inspect
```

Die Console zeigt dann die aktuellen Daten an.

```
=> #<Category id: 3, name: "Nachrichten", description: "News, Blogs etc.",  
created_at: "2008-06-23 17:25:17", updated_at: "2008-06-23 17:25:17">
```

Mit dieser Vorgehensweise können Sie Objekte sehr einfach aus YAML-Dateien importieren; Sie kennen nun die Funktionsweise von Fixtures.

Es gibt auch die Möglichkeit, dynamisch Daten in die Fixtures einzubauen, indem Sie Embedded Ruby verwenden. Um das Thema nicht zu sehr auszudehnen, sollten wir an dieser Stelle nicht weiter auf dynamische Fixture-Daten eingehen.

### 3.2.1.2 Tests mit Rake starten

Zurück zum eigentlichen Thema, dem Testen der Controller und Modelle. Mit Rake können die bereits generierten Testklassen ausgeführt werden, wobei allerdings noch nicht die gewünschte Testabdeckung erreicht ist. Die Ergebnisse werden übersichtlich angezeigt, und bei Fehlern weiß der Entwickler, wo er für Abhilfe sorgen muss.

Damit Sie einen Überblick über die verfügbaren Rake-Tasks für Testläufe erhalten, geben Sie auf der Kommandozeile im Projektverzeichnis folgende Anweisung ein.

```
$ rake -T test
```

Damit werden alle Tasks angezeigt und beschrieben, die mit dem Testen im Rails-Umfeld zu tun haben.

```
rake db:test:clone           # Recreate the test database from the current...
rake db:test:clone_structure # Recreate the test databases from the develo...
rake db:test:prepare        # Prepare the test database and load the schema
rake db:test:purge          # Empty the test database
rake test                   # Test all units and functionals
rake test:functionals       # Run tests for functionalsdb:test:prepare / ...
rake test:integration       # Run tests for integrationdb:test:prepare / ...
rake test:plugins           # Run tests for pluginsenvironment / Run the ...
rake test:recent            # Run tests for recentdb:test:prepare / Test ...
rake test:uncommitted       # Run tests for uncommitteddb:test:prepare / ...
rake test:units             # Run tests for unitsdb:test:prepare / Run th...
```

So finden sich hier Tasks zur Vorbereitung der Testdatenbank sowie zur eigentlichen Testausführung. Starten Sie Rake ohne die Angabe eines Tasks, werden alle Tests ausgeführt. Der explizite Aufruf der Tests erfolgt so:

```
$ rake test
```

Dieser Task führt alle Unittests sowie alle funktionalen Tests aus und gibt für dieses Beispielprojekt in etwa Folgendes aus.

```
(in /Users/mjohann/Documents/projects/buch/RailsBuch/Buch/projekte/Kapitel
_2/2.1/feeder)
/usr/local/bin/ruby -Ilib:test "/usr/local/lib/ruby/gems/1.8/gems/rake-
0.8.1/lib/rake/rake_test_loader.rb" "test/unit/category_test.rb"
"test/unit/feed_test.rb"
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.8.1/lib/rake/rake_test_loader
Started
..
Finished in 0.091988 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
/usr/local/bin/ruby -Ilib:test "/usr/local/lib/ruby/gems/1.8/gems/rake-
0.8.1/lib/rake/rake_test_loader.rb" "test/functional/categories_controller_test.rb"
"test/functional/feeds_controller_test.rb"
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.8.1/lib/rake/rake_test_loader
Started
.....
```

```
Finished in 1.044605 seconds.  
  
8 tests, 14 assertions, 0 failures, 0 errors  
/usr/local/bin/ruby -Ilib:test "/usr/local/lib/ruby/gems/1.8/gems/rake-  
0.8.1/lib/rake/rake_test_loader.rb"
```

Die in Fettschrift hervorgehobenen Zeilen geben die wichtigen Informationen über den Verlauf der Tests an. Neben der verbrauchten Zeit werden hier die Anzahl der Tests mit ihren Assertions sowie die Anzahl der Fehler angezeigt.

Wenn Sie sich die Unittests der beiden Modelle `Category` und `Feed` genauer ansehen, stellen Sie fest, dass hier nichts wirklich Sinnvolles getestet wird.

**Listing 3.5** <http://www.pastie.org/234049>

```
require File.dirname(__FILE__) + '/../test_helper'  
  
class CategoryTest < ActiveSupport::TestCase  
  # Replace this with your real tests.  
  def test_truth  
    assert true  
  end  
end
```

Die Kommentarzeile rät dem Entwickler dringend zur Implementierung eigener Tests, da eine einzige Testmethode, die lediglich überprüft, ob `true` dann auch wirklich `true` ist, nichts mit einem praxisnahen Test von Modellen zu tun hat.

Die folgende Testklasse zeigt einen wesentlich sinnvolleren Testansatz.

**Listing 3.6** <http://www.pastie.org/234053>

```
require File.dirname(__FILE__) + '/../test_helper'  
  
class CategoryTest < ActiveSupport::TestCase  
  fixtures :categories  
  
  def test_fixture_count  
    assert_equal 2, Category.count  
    assert_equal "Nachrichten", categories(:nachrichten).name  
    assert categories(:nachrichten).destroy  
    assert_equal 1, Category.count  
  end  
end
```

Zunächst werden hier die Fixtures für `Category`-Objekte angelegt. Diese Objekte werden dabei aus den bereits beschriebenen YAML-Dateien in Instanzen von `Category` geladen.

Die einzelnen Fixtures lassen sich nun im Test mit „Namen“ ansprechen, was in der Testmethode `test_fixture_count` geschieht. Dort sind fünf Assertions definiert, wovon die Erste die aktuelle Zahl der verfügbaren `Category`-Objekte überprüft. Mit `assert_equal` kann geprüft werden, ob zwei Werte identisch sind.

In diesem Fall enthält die Fixture-Datei `categories.yml` zwei Instanzen, so dass das Ergebnis von `Category.count` den Wert 2 ergeben sollte.

In der zweiten Assertion wird geprüft, ob der Name der Kategorie „Nachrichten“ auch wirklich `Nachrichten` lautet. Dann wird die Kategorie gelöscht und die um den Wert 1 verringerte Anzahl von `Category`-Objekten überprüft.

Der Ablauf sollte in etwa folgendes Ergebnis ausgeben.

```
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.8.1/lib/rake/rake_test_loader
Started
..
Finished in 0.14913 seconds.

2 tests, 5 assertions, 0 failures, 0 errors
```

Damit eine Fehlermeldung ausgegeben wird, können Sie nun die Anzahl der Fixtures erhöhen. Dazu geben Sie einfach eine weitere Objektbeschreibung in die Datei `categories.yml` ein.

```
musik:
  id: 3
  name: Musik
  description: Was für die Ohren
```

Wenn Sie jetzt die Tests erneut ausführen, wird berichtet, dass eine Assertion Probleme bereitet.

```
Loaded suite /usr/local/lib/ruby/gems/1.8/gems/rake-0.8.1/lib/rake/rake_test_loader
Started
F.
Finished in 0.148278 seconds.

1) Failure:
test_fixture_count(CategoryTest)
  [./test/unit/category_test.rb:7:in `test_fixture_count'
  /usr/local/lib/ruby/gems/1.8/gems/activerecord-2.0.2/lib/active_record/testing/default.rb:7:in `run']:
  <2> expected but was
  <3>.

2 tests, 2 assertions, 1 failures, 0 errors
```

Die in Fettschrift hervorgehobenen Zeilen zeigen, wo das Problem liegt. Insbesondere der Hinweis `<2> expected but was <3>` erklärt deutlich, warum dieser Test fehlgeschlagen ist.

Wenn Sie den dritten Datensatz in den Fixtures behalten möchten, können Sie die Assertions anpassen, so dass drei Objekte erwartet werden und nach dem Löschen noch zwei anstatt ein Objekt übrig bleiben.

Wenn Sie den Typ eines Fixture-Objektes prüfen möchten, können Sie eine weitere Methode einbauen.

```
def test_category_nachrichten
  @nachrichten = categories(:nachrichten)
  assert @nachrichten.is_a? Category
end
```

Damit fragen Sie die Instanzvariable `@nachrichten`, ob sie vom Typ `Category` ist. Falls diese Frage mit `true` beantwortet wird, läuft der Test durch.

Mit der richtigen Assertion wird der Code noch besser lesbar.

```
assert_instance_of Category, @nachrichten
```

### 3.2.1.3 Assertions für alle Fälle

Um Ihnen einen Überblick über die gebräuchlichsten Assertions zu geben, habe ich die folgende Tabelle erstellt. Sie erklärt zudem, welche Assertions wofür verwendet werden. Die Tabelle ist keinesfalls vollständig, sondern beschreibt die wichtigsten Assertions.

**Tabelle 3.1** Die gängigsten Assertions

Assertion	Beschreibung
<code>assert</code>	Ergebnis darf nicht <code>false</code> oder <code>nil</code> sein.
<code>assert_block</code>	Das Ergebnis eines Blocks muss <code>true</code> sein.
<code>assert_equal</code>	Prüft, ob das zweite Argument identisch mit dem ersten Argument ist. Die Reihenfolge ist wichtig, da entsprechende Fehlermeldungen sonst missverstanden werden können.
<code>assert_in_delta</code>	Prüft, ob zwei <code>Float</code> -Werte mit einer angegebenen Toleranzgrenze gleich sind.
<code>assert_instance_of</code>	Prüft, ob eine Instanz vom Typ der angegebenen Klasse ist.
<code>assert_kind_of</code>	Prüft, ob eine Instanz vom Typ der angegebenen Klasse ist.
<code>assert_match</code>	Prüft, ob ein String und ein regulärer Ausdruck zusammenpassen.
<code>assert_nil</code>	Prüft, ob ein Objekt <code>nil</code> ist.
<code>assert_no_match</code>	Das Gegenteil von <code>assert_match</code> .
<code>assert_not_equal</code>	Das Gegenteil von <code>assert_equal</code> .
<code>assert_not_nil</code>	Das Gegenteil von <code>assert_nil</code> .
<code>assert_nothing_raised</code>	Prüft, ob im Block keine Exception geworfen wurde.
<code>assert_raise</code>	Prüft, ob eine der angegebenen Exceptions geworfen wurde.
<code>assert_respond_to</code>	Prüft, ob ein Objekt eine bestimmte Methode kennt.

## 3.2.2 Funktionale Tests in Rails

In diesem Abschnitt lernen Sie die von Ruby on Rails generierten funktionalen Tests kennen. Dazu dient das Beispiel, das Ruby on Rails für das Testen des `FeedsControllers` generiert hat.

**Listing 3.7** <http://www.pastie.org/234063>

```
require File.dirname(__FILE__) + '/../test_helper'

class FeedsControllerTest < ActionController::TestCase
  def test_should_get_index
    get :index
    assert_response :success
    assert_not_nil assigns(:feeds)
  end

  def test_should_get_new
    get :new
    assert_response :success
  end
end
```

```

def test_should_create_feed
  assert_difference('Feed.count') do
    post :create, :feed => { }
  end

  assert_redirected_to feed_path(assigns(:feed))
end

def test_should_show_feed
  get :show, :id => 1
  assert_response :success
end

def test_should_get_edit
  get :edit, :id => 1
  assert_response :success
end

def test_should_update_feed
  put :update, :id => 1, :feed => { }
  assert_redirected_to feed_path(assigns(:feed))
end

def test_should_destroy_feed
  assert_difference('Feed.count', -1) do
    delete :destroy, :id => 1
  end

  assert_redirected_to feeds_path
end
end

```

Es ist zu sehen, dass bereits für jede Methode des RESTful-Controllers ein Test generiert wurde. Im Gegensatz zu den Unittests ist der funktionale Test von der Klasse `ActionController::TestCase` abgeleitet. Unittests erben von `ActiveSupport::TestCase`.

Der `TestCase` für funktionale Tests unterstützt eine wesentlich breitere Palette an Funktionalität als die `TestCase`-Klasse aus dem Unittest-Umfeld.

Betrachten wir die Methode `test_should_create_feed` näher. Dort wird versucht ein neues `Feed`-Objekt anzulegen. Das Ergebnis der Operation wird für eine Assertion verwendet, die überprüft, ob die Anzahl der `Feed`-Objekte nun um den Wert 1 erhöht wurde. Dazu wird mit `Feed.count` die aktuelle Anzahl an `Feed`-Objekten abgefragt.

Die Methode `test_should_destroy_feed` überprüft ebenfalls, ob nach dem Löschen eines `Feed`-Objektes die Anzahl der Objekte verändert wurde. Hier wird allerdings der Wert -1 als Übergabeparameter für die Assertion angegeben. Somit prüft die Assertion, ob der Wert um 1 verringert wurde.

Verändern Sie nun die Modellklasse `Feed` dahingehend, dass alle drei Parameter `name`, `url` und `description` zu Pflichtfeldern werden. Die Attribute `name` und `url` sollen zudem eindeutig sein, so dass es keine doppelten Einträge gibt. Das folgende Listing zeigt die Klasse `Feed` nach der Erweiterung.

**Listing 3.8** <http://www.pastie.org/234069>

```

class Feed < ActiveRecord::Base
  belongs_to :category
  validates_uniqueness_of :name, :url
  validates_presence_of :name, :url, :description
end

```

Diese Änderungen haben zur Folge, dass der funktionale Test `test_should_create_feed` fehlschlägt, denn die Parameter zur erfolgreichen Erstellung eines `Feed`-Objekts müssen noch übergeben werden.

Ein weiterer Hinderungsgrund für das erfolgreiche Durchlaufen der Tests ist die Tatsache, dass ein Filter vor jeder Methode versucht, den Benutzer zu authentifizieren. Kommentieren Sie daher für den Moment die folgende Zeile aus.

**Listing 3.9** <http://www.pastie.org/234070>

```
class FeedsController < ApplicationController
  # before_filter :authenticate, :except => ['index', 'show']
  before_filter :iphone_format
  # GET /feeds
```

Für den Fall, dass Sie die vollständige Spezifikation mit einem durch Authentifizierung geschützten Controller testen wollen, können Sie auch die Methode `authenticate` als Stub anlegen und so jeden beliebigen User anmelden. An dieser Stelle soll allerdings die Auskommentierung reichen.

Damit der Test beim nächsten Mal durchläuft, sollten Sie nun die Methode `test_should_create_feed` folgendermaßen anpassen.

**Listing 3.10** <http://pastie.org/234073>

```
def test_should_create_feed
  assert_difference('Feed.count') do
    post :create, :feed => { :name => "Eins",
                           :url => "http://localhost",
                           :description => "Ein Beispiel"}
  end

  assert_redirected_to feed_path(assigns(:feed))
end
```

Viele Aspekte der Unittests und der funktionalen Tests tauchen auch in anderen Testkonzepten wieder auf. Das Hauptaugenmerk dieses Kapitels gilt dem Konzept des Behavior Driven Development (BDD), weshalb wir hiermit unsere Ausführungen zu Unittests und funktionalen Tests beenden.

## 3.3 ZenTest

---

Bevor ich Ihnen die Konzepte hinter dem Behavior Driven Development näher erläutere, stelle ich Ihnen einige Hilfen für die Automatisierung von Tests vor. Die Werkzeuge vereinfachen den Prozess der testgetriebenen Entwicklung und bieten auf einen Blick wesentlich mehr Übersicht.

### 3.3.1 Automatisiertes Testen

Üblicherweise sieht die Arbeit eines agilen Softwareentwicklers folgendermaßen aus.

- Tests beschreiben
- Tests durchführen
- Fehler beheben
- Tests durchführen
- Fehler beheben
- usw.

Mit diesem Ansatz müssen immer wieder Werkzeuge benutzt und zwischen diesen gewechselt werden. Das ist viel Tipparbeit und vergeudet eine Menge Zeit. Eine geeignete Automatisierung dieses Prozesses schafft hier Abhilfe.

Der Schlüssel zu automatisierten Tests ist das Paket ZenTest, welches Sie sich zunächst installieren sollten. Hierzu gehen Sie auf die Kommandozeile und geben folgende Anweisung ein.

```
$ sudo gem install ZenTest
```

Daraufhin wird ZenTest installiert und in etwa diese Meldungen ausgegeben.

```
Updating metadata for 71 gems from http://gems.rubyforge.org
.....
complete
Successfully installed ZenTest-3.8.0
1 gem installed
Installing ri documentation for ZenTest-3.8.0...
Installing RDoc documentation for ZenTest-3.8.0...
```

ZenTest liefert das Skript `autotest` mit, das Änderungen an Dateien erkennt und bei Bedarf einen Testlauf startet.

## 3.4 Behavior Driven Development

In diesem Abschnitt geht es um das Paradigma des Behavior Driven Development, kurz: BDD. Vielen Lesern wird der Begriff „Test Driven Development“ (TDD) sicher bekannt vorkommen.

Vielleicht entwickeln Sie bisher sogar testgetrieben und kennen sich mit Unittests, funktionalen Tests und Akzeptanztests aus.

Dann wissen Sie auch, welche Vorteile die testgetriebene Entwicklung bietet.

- eine klarere Architektur;
- eine hohe Testabdeckung;
- mehr Flexibilität bei geänderten Anforderungen;
- Fokussierung auf gestellte Anforderungen.

Die Vorteile stellen sich in einigen Punkten selbstverständlich erst im fortgeschrittenen Projektverlauf ein. Dies ist oft ein Grund dafür, dass Projekte teilweise die Tests erst am Ende eines Projektes erstellen. Dies hat aber den Nachteil, dass nur der Code getestet wird,

der bereits besteht. Eine Fokussierung auf die funktionalen Spezifikationen findet so nur teilweise statt.

Generell lässt sich aus der Erfahrung, die ich in Projekten gesammelt habe, ableiten, dass der testgetriebene Ansatz das Mittel der Wahl ist.

Trotzdem hat TDD einen gravierenden Nachteil. Alles – von der Konzeption bis hin zur Implementierung der Tests – wird mit dem Wort *Test* in Zusammenhang gebracht. Unittests werden von `TestCase` abgeleitet, TestSuiten fassen Testcases zusammen, und selbst die Methoden beginnen mit dem Präfix `test`.

Dies führt nach Ansicht der Protagonisten des BDD-Ansatzes zu einer fehlgeleiteten Sichtweise auf das eigentliche Ziel, nämlich Software zu entwickeln, die sich an den Anforderungen und funktionalen Spezifikationen messen lässt.

#### **Rückblick: Object Behavior Analysis**

Mitte der neunziger Jahre, als ich mich intensiv mit der Sprache Smalltalk befasste, gab es noch keine UML zur Modellierung von Softwaresystemen, allerdings schon eine Menge Analyseverfahren zur Findung von Klassen und Methoden. Eine Methode war die Object Behavior Analysis<sup>3</sup> (OBA). Diese Methode konzentrierte sich auf Klartext-Spezifikationen von Usecases. Mit einigen klaren Regeln, wie solche Spezifikationen zu schreiben und zu lesen sind, war es sehr leicht, Klassen und deren Methoden zu „finden“.

Ein Beispiel: „Der Kunde meldet sich beim System mit seinem Benutzernamen und Passwort an. Danach kann er einen Geldbetrag überweisen.“

Mit den Methoden der OBA konnte man so die Klasse Kunde mit den Attributen Benutzername und Passwort finden. Das System ist ebenfalls eine Klasse mit den Methoden Anmelden (neudeutsch: Login) und Überweisen. Der Geldbetrag ist ein Parameter für die Methode Überweisen.

Dieses Beispiel soll Ihnen einen kleinen Vorgeschmack davon geben, wie „natürlich“ die objektorientierte Analyse mit Spezifikationen einhergeht.

BDD setzt ebenfalls auf das Beschreiben von Spezifikationen. So wie es für Unittests im Java-Umfeld JUnit gibt, steht Ihnen für die Unterstützung von BDD unter Java das Framework JBehave<sup>4</sup> zur Verfügung.

Die Konzepte von BDD haben inzwischen eine weite Verbreitung und damit immer mehr Anhänger gefunden. So ist es nur logisch, dass es für Ruby entsprechend ebenfalls Frameworks für BDD gibt. Im weiteren Verlauf dieses Kapitels werden Sie rSpec kennenlernen. Doch zuvor beschreibe ich einige Konzepte und führe Sie in die Grundlagen von BDD ein.

Die grundlegenden Begriffe von BDD sind

- Spezifikationen und
- Stories.

---

<sup>3</sup> <http://www.innolution.com/documents/oba/GETTOWHY.pdf>

<sup>4</sup> <http://www.jbehave.org>

Mit den Spezifikationen werden Objekte und deren Eigenschaften sowie ihr Verhalten (Attribute und Methoden) beschrieben. Da hier nicht einmal das Wort „Test“ auftaucht, können Sie sich leicht vorstellen, dass der Fokus bei der Entwicklung von Spezifikationen ein anderer ist, als wenn Sie Unittests beschreiben. Das Ziel ist allerdings auch bei BDD, die zu entwickelnde Software dahingehend zu überprüfen, ob alle Spezifikationen wie gewünscht umgesetzt wurden.

Die Stories beschreiben komplexe Abläufe und sind eher in den Bereich der Akzeptanztests anzusiedeln. Während bei den Spezifikationen der Fokus auf einzelnen Klassen liegt, konzentrieren sich Stories eher auf die komplette Durchführung von Szenarien oder Use-cases.

Die Spezifikationen stellen Erwartungen (Expectations) an die Objekte, die mit einer eigenen domänenspezifischen Sprache (DSL) beschrieben werden. Das Pendant bei Unittests sind hier die Assertions.

Wie findet man nun die Spezifikationen? Die Antwort sind Fragen. Gewichtige Fragen.

- In etwa könnte man sich zu Beginn die folgende Frage stellen: „Was ist das wichtigste Feature des Systems?“
- Die Antwort ist unter Umständen nicht leicht zu finden, weshalb die Frage umformuliert werden sollte: „Welches Feature fehlt aktuell?“
- Bei der Änderung von bestehenden Systemen könnte man die folgende Frage stellen: „Was wäre das größte Problem, das entsteht, wenn das aktuelle System nicht mehr vorhanden ist?“

Diese Art von Fragen haben ein besonderes Gewicht und führen oft zu einer verbesserten Wahrnehmung des wahren Wertes eines Systems.

### 3.4.1 rSpec

rSpec ist ein Framework für Ruby und Ruby on Rails, das die Konzepte der Spezifikationen und User Stories unterstützt und bereits eine hohe Verbreitung gefunden hat. Die Entwickler dieses Frameworks geben sich große Mühe, alle Aspekte von BDD zu unterstützen.

Im weiteren Verlauf dieses Kapitels werden die Grundlagen für den Einsatz von rSpec in allen weiteren Beispielprojekten vermittelt.

#### 3.4.1.1 Grundlagen und Installation neues Objekt

Zunächst einmal erläutere ich Ihnen, wo Sie rSpec bekommen und wie es installiert wird.

rSpec gibt es als Gem für Ruby und kann mit folgendem Aufruf installiert werden.

```
$ sudo gem install rspec
```

Sie sollten daraufhin in etwa folgende Ausgabe erhalten.

```
Successfully installed rspec-1.1.2
1 gem installed
Installing ri documentation for rspec-1.1.2...
Installing RDoc documentation for rspec-1.1.2...
```

Damit haben Sie nun rSpec für Ruby installiert, allerdings noch nicht für Ruby on Rails.

#### Installation mit Subversion

Für den Einsatz in Ruby on Rails-Projekten empfehlen die Entwickler von rSpec, auf die Installation mittels `gem` zu verzichten und stattdessen über Subversion die aktuellen Plugins zu installieren.

Hierzu müssen Sie im aktuellen Hauptverzeichnis Ihres Ruby on Rails-Projektes die beiden folgenden Anweisungen eingeben.

```
$ ruby script/plugin install \
http://rspec.rubyforge.org/svn/tags/CURRENT/rspec

$ ruby script/plugin install \
http://rspec.rubyforge.org/svn/tags/CURRENT/rspec_on_rails
```

Wenn Ihr Projekt ebenfalls mit Subversion versioniert wird, sollten Sie den Parameter `-x` hinzufügen.

```
$ ruby script/plugin install -x \
http://rspec.rubyforge.org/svn/tags/CURRENT/rspec

$ ruby script/plugin install -x \
http://rspec.rubyforge.org/svn/tags/CURRENT/rspec_on_rails
```

Damit werden rSpec und rSpec on Rails in das Verzeichnis `vendor/plugins` installiert, und Ihr Projekt ist nun in der Lage, mit rSpec umzugehen.

#### Installation ohne Subversion

Wenn Sie keinen Zugriff auf Subversion haben, aber dennoch mit rSpec arbeiten wollen, können Sie einen manuellen Download durchführen und die Plugins selbst installieren.

Hierzu laden Sie die aktuelle Version von folgender Webseite.

```
http://rubyforge.org/frs/?group_id=797
```

Dort laden Sie die beiden Dateien `rspec-1.1.4.tgz` und `rspec_on_rails-1.1.4.tgz` herunter und entpacken sie in das Verzeichnis `vendor/plugins` Ihres Projektes. Diesen Vorgang können Sie im folgenden Beispiel nachvollziehen.

##### 3.4.1.2 Ein Beispiel

Ausgangspunkt für das folgende Beispiel ist die letzte Version des Projektes `feeder` aus Kapitel 1. Sie können die Quellen für das Projekt unter [www.briefcasten.com](http://www.briefcasten.com) downloaden, um den aktuellen Stand des Projekts zu erhalten.

Wechseln Sie nun in das Verzeichnis `vendor/plugins`, und laden Sie die beiden Archive von der genannten Webseite.

```
$ curl -OL \
http://rubyforge.org/frs/download.php/37509/rspec-1.1.4.tgz
$ curl -OL \
http://rubyforge.org/frs/download.php/37510/rspec-rails-1.1.4.tgz
```

Danach entpacken Sie die beiden Archive:

```
$ tar xvzf rspec-1.1.4.tgz
$ tar xvzf rspec_on_rails-1.1.4.tgz
```

Die nun vorliegenden Verzeichnisse müssen noch umbenannt werden:

```
$ mv rspec-1.1.4 rspec
$ mv rspec_on_rails-1.1.4 rspec_on_rails
```

Die verbliebenen Archive können Sie aus dem Verzeichnis löschen oder zur Sicherung speichern.

Damit das Projekt mit rSpec arbeiten kann, müssen Sie im Hauptverzeichnis des Projektes Folgendes eingeben.

```
$ ruby script/generate rspec
```

Damit werden die folgenden Dateien und Verzeichnisse angelegt.

```
create spec
create spec/spec_helper.rb
create spec/spec.opts
create spec/rcov.opts
create script/spec_server
create script/spec
create stories
create stories/all.rb
create stories/helper.rb
```

Im Verzeichnis `spec` werden nun alle Spezifikationsklassen in passenden Unterverzeichnissen abgelegt, während die User Stories im gleichnamigen Verzeichnis zu finden sind.

### Ausführen von rSpec

Sie haben nun zwei Möglichkeiten, die Spezifikationen laufen zu lassen. Da rSpec einige Tasks für Rake mitbringt, können Sie folgende Anweisung eingeben.

```
$ rake spec
```

Alternativ finden Sie ein Skript vor, das Ihnen auch einige Informationen ausgibt.

```
$ ruby script/spec spec
Finished in 0.008688 seconds
0 examples, 0 failures
```

Sie sehen, dass noch keine Spezifikationen vorhanden sind und die Durchläufe daher wenig aussagekräftig sind.

### Autotest konfigurieren

Wenn Sie ZenTest noch nicht installiert haben, sollten Sie dies jetzt nachholen.

```
$ sudo gem install ZenTest
```

```
Password:
Updating metadata for 33 gems from http://gems.rubyforge.org/
.....
complete
Successfully installed ZenTest-3.10.0
1 gem installed
Installing ri documentation for ZenTest-3.10.0...
Installing RDoc documentation for ZenTest-3.10.0...
```

Ab der Version 3.8.0 von ZenTest ist es möglich, eine sehr feingranulare Konfiguration für Autotest vorzunehmen.

Folgende Reihenfolge wird beim Einlesen der Konfiguration vorgenommen.

- Autotest-Klasse (die Klasse `Autotest`)
- `~/ .autotest` (die Datei `.autotest` im Home-Verzeichnis des Benutzers)
- `.autotest` (die Datei `.autotest` im Projektverzeichnis)

Damit besteht die Möglichkeit, globale Hooks in die Konfigurationsdatei im Home-Verzeichnis einzutragen und projektspezifische Konfigurationsteile in die Datei `.autotest` im Projektverzeichnis.

Unter Mac OS X besteht zudem die Möglichkeit, das Programm Growl<sup>5</sup> zu integrieren. Growl gibt Benachrichtigungen in optischer Form aus und lässt sich leicht den eigenen Bedürfnissen anpassen.

Für die Windowsnutzer unter den Lesern gibt es Snarl<sup>6</sup>, das den gleichen Zweck erfüllt. Die folgenden Codezeilen stellen den Inhalt der Datei `.autotest` im Homeverzeichnis dar. Darin enthalten sind die notwendigen Definitionen für die Integration von Growl unter Mac OS X.

**Listing 3.11** <http://pastie.org/234112>

```
module Autotest::Growl
  AUTOTEST_IMAGE_ROOT = "~/ .autotest_images"

  def self.growl title, msg, img, pri=0, sticky=""
    system "growlnotify -n autotest --image #{img} -p #{pri} -m \
      #{msg.inspect} #{title} #{sticky}"
  end

  def self.growl_fail(output)
    growl "FAIL", "#{output}", "#{AUTOTEST_IMAGE_ROOT}/fail.png", 2
  end

  def self.growl_pass(output)
    growl "Pass", "#{output}", "#{AUTOTEST_IMAGE_ROOT}/pass.png"
  end

  Autotest.add_hook :ran_command do |at|
    results = [at.results].flatten.join("\n")

    if results.include? 'tests'
      output = \ results.slice(/(\d+)\s+tests?, \
        \s*(\d+)\s+assertions?, \
```

---

<sup>5</sup> Growl ist unter <http://growl.info/> zu finden.

<sup>6</sup> Snarl gibt es hier <http://www.fullphat.net/>

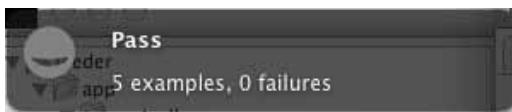
```

        \s*(\d+)\s+failures?(, \
        \s*(\d+)\s+errors)?/)
    if output
      $~[3].to_i + $~[5].to_i > 0 \
        ? growl_fail(output) : growl_pass(output)
    end
  else
    output = results.slice(/(\d+)\s+examples?, \
      \s*(\d+)\s+failures?(, \
      \s*(\d+)\s+not implemented)?/)
    if output
      $~[2].to_i > 0 ? growl_fail(output) : growl_pass(output)
    end
  end
end
end
end
end

```

Die kompletten Sourcedateien für die Grafiken usw. können Sie sich unter [www.briefcasten.com](http://www.briefcasten.com) herunterladen.

Die Integration von Growl erzeugt bei Testdurchläufen im Erfolgsfall sowie im Fehlerfall eine entsprechende Ausgabe auf dem Bildschirm. Die folgende Abbildung zeigt das Beispiel einer solchen Ausgabe.



**Abbildung 3.1**  
So sieht eine Nachricht von Growl aus.

Wenn ein Fehler gemeldet wird, erscheint ein weniger fröhliches rotes Gesicht, womit man sehr schnell über den Verlauf der Tests informiert ist. Bei Änderungen an Dateien kann sofort ein Testlauf mittels `autotest` erfolgen. Über einen Entwicklungstag verteilt, spart man auf diese Weise leicht eine halbe Stunde oder mehr Zeit ein.

Damit `autotest` weiß, wann welche Tests ausgeführt werden sollen, kann man sich nun praktischerweise im Projektverzeichnis eine weitere Datei namens `.autotest` anlegen. Die folgende Datei nutzt den Hook `:initialize`, um Ausnahmen und Dateimappings zu definieren.

**Listing 3.12** <http://pastie.org/234116>

```

Autotest.add_hook :initialize do |at|
  %w{.hg .git .svn stories tmtags Rakefile \
    nbproject Capfile README spec/spec.opts \
    spec/rcov.opts vendor/gems autotest svn-commit \
    .DS_Store }.each {
    |exception| at.add_exception(exception)
  }

  at.add_mapping(%r%^app/models/.*\.rb$%) {
    at.files_matching(%r%^spec/models/.*_spec.rb$%)
  }
end

```

Zunächst werden in einem Array alle Dateinamen aufgeführt, bei denen `autotest` getrost alle Änderungen ignorieren kann. Dazu wird über dieses Array iteriert und jeweils eine Exception hinzugefügt. Dies ist keine Exception, die irgendwann einmal geworfen wird, sondern lediglich eine Regel für das Ignorieren von Änderungen.

Im Bereich des Mappings wird mit Hilfe von regulären Ausdrücken formuliert, welche Dateien überwacht werden sollen. Wenn diese Bedingung nun erfüllt ist, werden die Tests ausgeführt, die im Block mit einem weiteren regulären Ausdruck angegeben wurden.

In diesem Beispiel sind dies alle Tests unterhalb des Verzeichnisses `spec/models`. Wenn sich also an den Klassen im Verzeichnis `app/models` etwas ändert, werden alle Tests erneut ausgeführt.

Noch sinnvoller ist es allerdings, alle Modell-Tests auszuführen, die mit den Modellen zusammenhängen, und alle Controller-Tests laufen zu lassen, die mit Controlleränderungen einhergehen usw. Dazu können die folgenden Mappings genutzt werden.

**Listing 3.13** <http://pastie.org/234117>

```
at.add_mapping(%r%^app/models/.*\.rb$%) {
  at.files_matching(%r%^spec/models/.*_spec.rb$%)
}

at.add_mapping(%r%^app/controllers/.*\.rb$%) {
  at.files_matching(%r%^spec/controllers/.*_spec.rb$%)
}

at.add_mapping(%r%^app/helpers/.*\.rb$%) {
  at.files_matching(%r%^spec/helpers/.*_spec.rb$%)
}
```

Sicher kann man die Mappings noch verbessern, so dass bei Änderungen an Modellen auch die Controller- und Helpertests durchgeführt werden. Fürs Erste soll allerdings die aktuelle Konfiguration ausreichen.

#### rSpec Modellspezifikationen für Category

Sie fragen sich sicher, wie denn wohl die Spezifikationen aussehen. Für die ersten Schritte skizziere ich daher die Spezifikationen für die Modellklasse `Category`. Sehen Sie sich zunächst folgendes Listing an.

**Listing 3.14** <http://pastie.org/234118>

```
require File.dirname(__FILE__) + '/../spec_helper'

describe Category, "from fixture :unterhaltung" do
  fixtures :categories

  before(:each) do
    @category = categories(:unterhaltung)
  end

  it "should be valid" do
    @category.should be_valid
  end

  it "should have correct name" do
    @category.name.should eql("Entwicklung")
  end

  it "should have correct description" do
    @category.description.should eql("Alles rund um Softwareentwicklung")
  end

  it "should have id that is not nil" do
```

```

    @category.id.should_not be_nil
  end
  it "should have id greater than zero" do
    @category.id.should > 0
  end

  it "should do this in the future"
end

```

Was für einen Unittest die Setup-Methode darstellt, ist in dieser Spezifikation unter rSpec der `before`-Block, in dem für jede Spezifikation eine Instanz von `Category` aus den Fixtures geladen wird.

Bei Bedarf kann für Aufräumarbeiten die Methode `after` implementiert werden. Damit ist das Pendant zu den `setup/teardown`-Methoden komplett.

Die folgenden Examples überprüfen die jeweiligen Attribute auf ihre Richtigkeit und ob sie mit den erwarteten Werten übereinstimmen. Eine Spezifikation beginnt dabei immer mit `describe` und einem Beschreibungstext.

```

describe Category, „from fixture :unterhaltung“ do
  ...
end

```

Die einzelnen Beispiele (Examples) beginnen immer mit `it` und einem nachfolgenden Text, der meistens wiederum mit dem Wort `should` beginnt.

**Listing 3.15** <http://pastie.org/234120>

```

it "should have id that is not nil" do
  @category.id.should_not be_nil
end

```

Damit lassen sich die einzelnen Spezifikationstests in ganzen Sätzen wiedergeben.

```

Category from fixture :entwicklung should do this in the future

```

So bekommen die Spezifikationstests eine wesentlich übersichtlichere Struktur als die klassischen Unittests. Ein Durchlauf der Spezifikationen ergibt beispielsweise folgende Ausgabe.

```

Category from fixture :entwicklung
- should be valid
- should have correct name
- should have correct description
- should have id that is not nil
- should have id greater than zero
- should do this in the future (PENDING: Not Yet Implemented)

Pending:
Category from fixture :entwicklung should do this in the future (Not Yet
Implemented)

Finished in 0.213723 seconds

7 examples, 0 failures, 1 pending

```

Hier wird auf einen Blick klar, dass es ein nicht implementiertes Example gibt. Der Text dafür existiert bereits, doch für eine Überprüfung fehlt der entsprechende Code, der bei Bedarf später implementiert werden kann.

Die Parameter für die Methode `should` sind sogenannte `Matcher`. Von den `Matchern` gibt es bereits eine ganze Menge, wie die folgende Tabelle zeigt.

**Tabelle 3.1** Liste der wichtigsten `Matcher`

Matcher	Beschreibung
<code>be</code>	Erwartet als Rückgabewert <code>true</code> oder <code>false</code> . Es kann jede beliebige Methode mit Boole'schem Rückgabewert an das <code>be_</code> angehängt werden. Beispielsweise ruft <code>be_older_than(4)</code> auf dem zu testenden Objekt die Methode <code>older_than?(4)</code> auf und erwartet ein <code>true</code> oder <code>false</code> .
<code>be_close</code>	Prüft einen Wert innerhalb einer gegebenen Toleranz. Beispiel: Ein Wert <code>5 +/- 0.5</code> kann also <code>4,5</code> bis <code>5,5</code> sein.
<code>change</code>	Hiermit kann die Änderung eines Attributes überwacht werden.
<code>eql</code>	Überprüft, ob zwei Werte gleich sind.
<code>equal</code>	Überprüft, ob zwei Objekte identisch sind.
<code>exist</code>	Erwartet die Methode <code>exist?</code> beim zu testenden Objekt.
<code>has</code>	Beispiel: <code>{:a =&gt; „A“}.should have_key(:a)</code>
<code>include</code>	Beispiel: <code>[1, 2, 3].should include(3)</code>
<code>raise_error</code>	Beispiel: <code>lambda {}.should_not raise_error</code>
<code>respond_to</code>	Beispiel: <code>Object.new.should respond_to(:methods)</code>

Es steht jedem Entwickler frei, ob er eigene `Matcher` entwickelt, um die Flexibilität und Lesbarkeit der eigenen Spezifikationen zu erhöhen. Die von `rSpec` mitgelieferten `Matcher` reichen allerdings schon eine Zeit lang, ohne dass man sich Gedanken über eigene `Matcher`-Implementierungen machen muss. Mit den `Matchern` aus Tabelle 3.2 können Sie nun ein wenig herumexperimentieren. Hierzu gibt Ihnen das folgende Listing einige Anregungen.

**Listing 3.16** <http://pastie.org/234124>

```
describe Category, "not from fixture" do
  before(:each) do
    @category = Category.new
  end

  it "should not be valid if name and description are not given" do
    @category.should_not be_valid
  end

  it "should change name value" do
    lambda { @category.name = "Unterhaltung" }.should change(@category, :name)
  end

  it "should respond to :name" do
    @category.should respond_to(:name)
  end

  it "should have the same name as given name" do
    @category.name = "Unterhaltung"

    @category.name.should == "Unterhaltung"
  end
end
```

```

    @category.name.should eql("Unterhaltung")
    @category.name.should_not equal("Unterhaltung")
  end
end

```

### rSpec Controller-Spezifikationen für den FeedsController

Sie haben gesehen, wie Modelle mit rSpec getestet werden können. Die dafür vorgesehenen Tests sind unter `spec/models` zu finden. Um Spezifikationen für Controller zu erstellen, sollten diese in das Unterverzeichnis `spec/controllers` gestellt werden. Damit korrespondieren die Unterverzeichnisse von rSpec mit denen der zu testenden Anwendung, was die Orientierung vereinfacht.

Um die nächsten Schritte so nachvollziehen zu können, als hätten wir die Controller-Spezifikationen vor der Entwicklung des Controllers gestartet, sollten Sie die Methoden `new`, `edit` und `create` auskommentieren.

Sie werden nun lernen, wie man mit rSpec Controller-Spezifikationen entwickelt, die dazu führen, dass Sie Controller-Code entwickeln, der die Spezifikationen erfüllt.

Wir gehen zunächst davon aus, dass eine neue Instanz von `Feed` erstellt werden soll. Hierzu würde ein Anwender die URL `http://localhost:3000/feeds/new` aufrufen. Der Controller sollte dann eine neue Instanz für die View `new.html.erb` bereitstellen.

In einem weiteren Schritt gibt der Benutzer die Daten für einen `Feed`-Eintrag ein und klickt auf den `Speichern`-Button. Das `Feed`-Objekt wird gespeichert und auf die Seite `show.html.erb` verzweigt.

Dies ist der Positivfall. Hierfür entwickeln wir nun eine Spezifikation. Im Anschluss daran wird dafür gesorgt, dass das `Speichern` fehlschlägt und zurück auf die Seite `new.html.erb` verzweigt wird. Für diese Situation schreiben wir dann eine weitere Spezifikation. Doch zunächst zurück zur ersten Spezifikation für den Fall, dass alles so verläuft wie gewünscht.

Legen Sie als Erstes bitte eine Datei namens `feeds_controller_spec.rb` im Verzeichnis `spec/controllers` an. Dann starten Sie die Spezifikation mit folgenden Zeilen.

#### Listing 3.17 <http://pastie.org/234136>

```

require File.dirname(__FILE__) + '/../spec_helper'

describe FeedsController, " called with uri /feeds/new" do
  it "should be an instance of FeedsController" do
    controller.should be_an_instance_of(FeedsController)
  end
end

```

Noch einmal zur Begriffsdefinition.

- Spezifikationen sind die mit `describe` eingeleiteten Blöcke.
- Examples sind Blöcke, die mit `it` beginnen.
- Expectations sind vergleichbar mit Assertions im Unittest.

Mit diesen Begriffen wird nun die Spezifikation entwickelt.

Die Spezifikation `FeedsController` called with uri `/feeds/new` kennt ein erstes Example, das prüft, ob die Instanz des Controllers vom Typ `FeedsController` ist. Dies ist immer der Fall, wenn nach dem `describe` direkt der Controller-Typ genannt wird.

Im Example `should be an instance of FeedsController` wird eine Expectation formuliert, die überprüft, ob der Typ `FeedsController` ist.

Damit ist das erste Example implementiert. Wenn Sie nun `autotest` im Hintergrund laufen lassen und diese erste Spezifikation speichern, wird ein Testlauf folgende Ausgabe erzeugen.

```
FeedsController called with uri /feeds/new
- should be an instance of FeedsController

Finished in 0.185817 seconds

1 example, 0 failures
```

Das war kurz und schmerzlos. Jetzt wird es etwas komplexer.

Ein neues Example soll die Seite `/feeds/new` aufrufen.

**Listing 3.18** <http://pastie.org/234138>

```
it "should create a new instance of Feed with GET to new" do
  get :new
  response.should be_success
end
```

Autotest wird nun mit folgender Ausgabe monieren, dass die Seite nicht mit Erfolg aufgerufen werden konnte.

```
FeedsController called with uri /feeds/new
- should be an instance of FeedsController
- should create a new instance of Feed with GET to new (FAILED - 1)

1)
'FeedsController called with uri /feeds/new should create a new instance of Feed
with GET to new' FAILED
expected success? to return true, got false
./spec/controllers/feeds_controller_spec.rb:14:
script/spec:4:

Finished in 0.190006 seconds

2 examples, 1 failure
```

Ein Blick in den `FeedsController` offenbart das Problem. Für die Aktion `new` wird ein authentifizierter Benutzer benötigt. Jetzt gibt es zwei Möglichkeiten, um das Problem zu lösen. Entweder, Sie kommentieren den `Before`-Filter aus, was allerdings den Controller verändert und ungewollte Seiteneffekte haben kann. Oder Sie verwenden einen Stub für die Methode `authenticate`. Die zweite Lösung ist die bessere, und daher fügen Sie einen `Before`-Block in die Spezifikation ein, die vor jedem Example aufgerufen wird.

**Listing 3.19** <http://pastie.org/234139>

```
before(:each) do
  controller.stub!(:authenticate).and_return(true)
end
```

Damit wird dem Controller vorgegaukelt, er könne mit der Methode `authenticate` den Wert `true` zurückgeben. Stubs dienen dazu, Funktionalität vorzutäuschen und gewünschte Ergebnisse zurückzuliefern. Dies ist hier sinnvoll, da auf diese Weise die Kopplung von der HTTP-Authentifizierung stattfindet und wir die Tests schließlich ganz ohne Webserver laufen lassen wollen.

Nachdem Sie den `Before`-Block eingefügt haben, laufen die beiden Examples klaglos durch.

Es ist erforderlich, dass der Browser nach dem Aufruf von `new` im Controller die Seite `new.html.erb` anzeigt. Daher kann eine weitere Expectation in das letzte Example eingebaut werden.

```
response.should render_template("feeds/new")
```

Auch diese Zeile läuft fehlerfrei durch.

Ein weiteres Example soll überprüfen, ob beim Speichern einer neuen `Feed`-Instanz Probleme auftreten. Starten Sie hier mit den folgenden Zeilen.

**Listing 3.20** <http://pastie.org/234140>

```
it "should save an instance of Feed with POST on create" do
  post :create
  response.should be_redirect
end
```

Neben dem Versuch, hier eine Aktion `create` auf dem `FeedsController` aufzurufen, erwarten wir eine Umleitung auf eine andere Seite (später `index.html.erb`), die momentan aber nicht näher spezifiziert ist. Dieses Example läuft momentan nicht durch und produziert folgende Ausgabe.

```
FeedsController called with uri /feeds/new
- should be an instance of FeedsController
- should create a new instance of Feed with GET to new
- should save an instance of Feed with POST on create (ERROR - 1)

1)
ActionController::UnknownAction in 'FeedsController called with uri /feeds/new
should save an instance of Feed with POST on create'
No action responded to create
./spec/controllers/feeds_controller_spec.rb:20:
script/spec:4:

Finished in 0.198258 seconds

3 examples, 1 failure
```

Da keine `create`-Methode im Controller existiert, kann dieses Example nicht durchlaufen. Die Farbe Violet signalisiert bei der Ausgabe übrigens, dass das Example nicht ausgeführt werden konnte. Dies ist die Vorstufe zur fehlerhaften Ausführung. Anstatt jetzt die Kommentare von der Methode `create` zu entfernen, implementieren Sie nun schrittweise immer mehr Funktionalität in diese Methode, bis das Example fehlerfrei durchläuft.

Wenn Sie jetzt lediglich den Methoden-Rumpf in den `FeedsController` einfügen, erhalten Sie eine völlig andere Fehlermeldung.

```
FeedsController:
def create
end
Ausgabe:
FeedsController called with uri /feeds/new
- should be an instance of FeedsController
- should create a new instance of Feed with GET to new
- should save an instance of Feed with POST on create (FAILED - 1)

1)
'FeedsController called with uri /feeds/new should save an instance of Feed
with POST on create' FAILED
expected redirect? to return true, got false
./spec/controllers/feeds_controller_spec.rb:20:
script/spec:4:

Finished in 0.303334 seconds

3 examples, 1 failure
```

Die Ausgabe weist darauf hin, dass in der Methode `create` kein `Redirect` stattgefunden hat, was aktuell stimmt. Fügen Sie deshalb in die Methode `create` einen `Redirect` ein.

```
def create
  redirect_to "blabla"
end
```

Damit läuft das Example wieder durch. Die `create`-Methode soll allerdings noch mehr leisten, als eine Umleitung auf irgendeine Seite durchzuführen.

Da es sich bei dem `FeedsController` um einen RESTful-Controller handelt, können URIs mit den RESTful URL-Helper-Methoden angegeben werden. Des Weiteren kann eine Instanz von `Feed` angegeben werden, um die Seite `index.html.erb` anzuzeigen.

**Listing 3.21** <http://pastie.org/234144>

```
it "should save an instance of Feed with POST on create" do
  post :create
  response.should redirect_to(feeds_url)
end
```

Ersetzen Sie an dieser Stelle den Begriff `blabla` mit `feeds_url`, und die Examples laufen durch.

```
def create
  redirect_to feeds_url
end
```

Nach dem Speichern einer Instanz von `Feed` soll eine Erfolgsmeldung in das `Flash`-Objekt geschrieben werden. Hierzu wird ein neues Example formuliert.

**Listing 3.22** <http://pastie.org/234146>

```
it "should flash a success message" do
  post :create
  flash[:notice].should == "Feed was successfully created."
end
```

Die Ausführung schlägt logischerweise fehl, da noch kein Eintrag für `:notice` im Hash-Objekt `flash` vorliegt. Daher wird im `FeedsController` Folgendes eingebaut.

**Listing 3.23** <http://pastie.org/234148>

```
def create
  flash[:notice] = "Feed was successfully created."
  redirect_to feeds_url
end
```

Jetzt werden keine Fehler mehr ausgegeben, und wir können uns einem weiteren Example widmen.

**Listing 3.24** <http://pastie.org/234149>

```
it "should save the feed" do
  @feed.should_receive(:save)
  post :create
end
```

autotest meckert sofort und gibt folgende Ausgaben auf die Console.

```
Spec::Mocks::MockExpectationError in 'FeedsController called with uri /
feeds/new should save the feed'
Mock 'NilClass' expected :save with (any args) once, but received it 0 times
.
```

NilClass soll angeblich die Nachricht :save erwarten, aber nicht erhalten haben. Das ist ein Hinweis dafür, dass die Instanzvariable @feed nicht existiert und auch kein Mock-Objekt ist.

Mock-Objekte sind künstliche Abbildungen von echten Objekten, können aber frei gehandhabt werden, ohne die Abhängigkeiten beispielsweise von ActiveRecord beachten zu müssen. Fügen Sie also die folgende Zeile an die erste Stelle im Before-Block ein.

**Listing 3.25** <http://pastie.org/234150>

```
before(:each) do
  @feed = mock("feed")
  controller.stub!(:authenticate).and_return(true)
end
```

Damit wird eine Instanzvariable @feed als Mock-Objekt angelegt. Der Name des Mock-Objektes ist feed.

Die vorhergehende Meldung behauptet zusätzlich, dass noch keine save-Methode auf dem Objekt Feed aufgerufen wurde. Daher muss im FeedsController in der Methode create entsprechend ein neues Feed-Objekt angelegt und dann gespeichert werden.

**Listing 3.26** <http://pastie.org/234152>

```
def create
  @feed = Feed.new
  @feed.save
  flash[:notice] = "Feed was successfully created."
  redirect_to feeds_url
end
```

Dies impliziert aber, dass vor save ein neues Feed-Objekt erzeugt wurde. Im Before-Block sollte daher die Methode new mit einem Stub zum Mock hinzugefügt werden.

**Listing 3.27** <http://pastie.org/234153>

```
before(:each) do
  @feed = mock("feed")
  Feed.stub!(:new).and_return(@feed)
  controller.stub!(:authenticate).and_return(true)
end
```

Damit wird die Methode `new` auf der Klasse `Feed` mit einem Stub versehen, der anstelle der eigentlichen `new`-Methode aufgerufen wird. Dabei wird dann das Mock-Objekt zurückgegeben und nicht etwa eine gültige Instanz von `Feed`.

Diese Änderung führt zu zwei Fehlern, die beide behaupten, dass das Mock-Objekt die Methode `save` nicht kennt. Daher wird der `Before`-Block entsprechend erweitert.

**Listing 3.28** <http://pastie.org/234156>

```
before(:each) do
  @feed = mock("feed")
  Feed.stub!(:new).and_return(@feed)
  @feed.stub!(:save)
  controller.stub!(:authenticate).and_return(true)
end
```

Diesmal wird der Stub direkt an das Mock-Objekt gehängt, da `save` auch nur auf der Instanz aufgerufen wird. Im Anschluss laufen alle Examples fehlerfrei durch.

Im folgenden Example sollen übergebene Parameter auf ihre Richtigkeit hin überprüft werden. Fügen Sie die folgenden Zeilen Ihrer Spezifikation hinzu.

**Listing 3.29** <http://pastie.org/234157>

```
it "should create the feed" do
  Feed.should_receive(:new).with({ "name" => "Filme", \
    "url" => "http://localhost", \
    "description" => "Alles zum Thema +
    "Filme"})

  post :create, :feed => { "name" => "Filme", \
    "url" => "http://localhost", \
    "description" => "Alles zum Thema Filme"}
end
```

Da nun die Klasse `Feed` eine Methode `new` kennen muss, die mit Hilfe der Parameter aus dem HTTP-Request aufgerufen wird, muss die Methode `create` im Controller noch angepasst werden.

**Listing 3.30** <http://pastie.org/234159>

```
def create
  @feed = Feed.new(params[:feed])
  @feed.save
  flash[:notice] = "Feed was successfully created."
  redirect_to feeds_url
end
```

Eine Fehlermeldung wird noch ausgegeben, da die Expectation das Mock-Objekt als Rückgabewert erwartet. Daher muss das letzte Example noch erweitert werden.

**Listing 3.31** <http://pastie.org/234160>

```

it "should create the feed" do
  Feed.should_receive(:new).with({"name" => "Filme", \
    "url" => "http://localhost", \
    "description" => "Alles zum Thema " +
    "Filme"}).and_return(@feed)

  post :create, :feed => {"name" => "Filme", \
    "url" => "http://localhost", \
    "description" => "Alles zum Thema Filme"}

end

```

Jetzt laufen alle Examples der Spezifikation durch. Damit ist auch der Positivfall des Speicherns einer Instanz von `Feed` durchlaufen, was sich auch im Code der Methode `create` widerspiegelt. Doch was ist, wenn das Speichern aufgrund irgendeines Problems mit der Datenbank nicht erfolgreich verlief? Für diesen Fall schreiben wir nun eine weitere Spezifikation.

**Listing 3.32** <http://pastie.org/234161>

```

describe FeedsController, " receiving /feeds/new using POST, \
but doesn't save" do

  before(:all) do
    @feed = mock("feed")
    Feed.stub!(:new).and_return(@feed)
    @feed.stub!(:save)
    controller.stub!(:authenticate).and_return(true)
  end

end

```

Im Falle eines Problems beim Speichern von `Feed`-Objekten sollte die View `new.html.erb` aufgerufen werden. Dies wird nun mit folgendem Example überprüft.

**Listing 3.33** <http://pastie.org/234162>

```

it "should render the form new.html.erb" do
  post :create
  response.should render_template(:new)
end

```

Erwartungsgemäß führt die Ausführung zu einem Fehler. Der Grund: In der `create`-Methode wird kein `Redirect` im Fehlerfall ausgeführt. Mit den folgenden Änderungen wird die `create`-Methode angepasst.

**Listing 3.34** <http://pastie.org/234163>

```

def create
  @feed = Feed.new(params[:feed])
  if @feed.save
    flash[:notice] = "Feed was successfully created."
    redirect_to feeds_url
  else
    render :action => "new"
  end
end

```

Diese Änderungen führen allerdings dazu, dass zwei Examples aus der ersten Spezifikation nicht mehr laufen und das neue Example immer noch nicht.

Abhilfe schafft hier die Modifizierung des Stubs für die `save`-Methode in der ersten Spezifikation.

**Listing 3.35** <http://pastie.org/234164>

```
before(:each) do
  @feed = mock("feed")
  Feed.stub!(:new).and_return(@feed)
  @feed.stub!(:save).and_return(true)
  controller.stub!(:authenticate).and_return(true)
end
```

Jetzt laufen die Examples aus der ersten Spezifikation wieder fehlerfrei durch. Damit müssen lediglich für die zweite Spezifikation Änderungen durchgeführt werden, damit das letzte Example ebenfalls durchläuft.

**Listing 3.36** <http://pastie.org/234165>

```
before(:each) do
  @feed = mock("feed")
  Feed.stub!(:new).and_return(@feed)
  @feed.stub!(:save).and_return(false)
  controller.stub!(:authenticate).and_return(true)
end
```

Die zweite Spezifikation sagt mit dem einzelnen Example noch nicht wirklich aus, was hier spezifiziert wurde. Daher ist es sinnvoll, ein weiteres Example einzufügen.

**Listing 3.37** <http://pastie.org/234166>

```
it "should fail when saving feed" do
  @feed.should_receive(:save).and_return(false)
  post :create
end
```

Mit diesem Example wird die Intention für diese Spezifikation deutlicher. Alles in allem sieht der Inhalt der Datei `feeds_controller_spec.rb` nun folgendermaßen aus.

**Listing 3.38** <http://pastie.org/239538>

```
require File.dirname(__FILE__) + '/../spec_helper'

describe FeedsController, " called with uri /feeds/new" do
  before(:each) do
    @feed = mock("feed")
    Feed.stub!(:new).and_return(@feed)
    @feed.stub!(:save).and_return(true)
    controller.stub!(:authenticate).and_return(true)
  end

  it "should be an instance of FeedsController" do
    controller.should be_an_instance_of(FeedsController)
  end

  it "should create a new instance of Feed with GET to new" do
    get :new
    response.should be_success
    response.should render_template("feeds/new")
  end

  it "should save an instance of Feed with POST on create" do
```

```

    post :create
      response.should redirect_to(feeds_url)
    end

    it "should flash a success message" do
      post :create
      flash[:notice].should == "Feed was successfully created."
    end

    it "should create the feed" do
      Feed.should_receive(:new).with({"name" => "Filme", \
                                     "url" => "http://localhost", \
                                     "description" => "Alles zum Thema " +
                                     "Filme"}).and_return(@feed)

      post :create, :feed => {"name" => "Filme", \
                             "url" => "http://localhost", \
                             "description" => "Alles zum Thema Filme"}

    end

    it "should save the feed" do
      @feed.should_receive(:save).and_return(true)
      post :create
    end
  end

  describe FeedsController, " receiving /feeds/new using POST, \
but doesn't save" do
    before(:each) do
      @feed = mock("feed")
      Feed.stub!(:new).and_return(@feed)
      @feed.stub!(:save).and_return(false)
      controller.stub!(:authenticate).and_return(true)
    end

    it "should fail when saving feed" do
      @feed.should_receive(:save).and_return(false)
      post :create
    end

    it "should render the form new.html.erb" do
      post :create
      response.should render_template(:new)
    end
  end
end

```

Sie haben nun einen Eindruck davon, wie Modelle und Controller getestet werden. Dabei wurde ohne die entsprechenden Views getestet. Wenn Sie die entsprechenden Views zusammen mit dem Controller testen wollen, können Sie innerhalb der Spezifikation die Methode `integrate_views` aufrufen, und schon werden die Views in die Durchläufe integriert.

Mehr zum Testen von Views finden Sie in Kapitel 7, wenn die ersten Views mit dem BDD-Ansatz entstehen.

### User Stories

Der letzte Teil dieses Abschnitts befasst sich mit den Stories aus dem BDD-Ansatz. Sie sind ein Weg, um Akzeptanztests zu formulieren sowie die vorhandenen Controller und Views über Geschäftsprozesse hinweg zu testen.

Eine besonders interessante Umsetzung des Story-Konzeptes<sup>7</sup> bietet rSpec bereits seit einigen Monaten. rSpec bietet die Möglichkeit, User Stories im Klartext zu formulieren und Daten aus diesen Formulierungen in die Abläufe zu überführen. Dieser Ansatz ist für mich so etwas wie die Weiterführung von FIT<sup>8</sup>, bei dem die einfache Bereitstellung von Testdaten im Vordergrund steht.

User Stories sind definitiv ein echtes Highlight im Bereich der Akzeptanztests. Sie lassen sich von technisch eher weniger versierten Projektmitgliedern sehr leicht formulieren und dennoch auf technischer Ebene ausführen. Dabei werden komplexe Abläufe überprüft und die Abnahmefähigkeit von Systemen sichergestellt.

BDD ist ein Weg, um von direkten Ideen in Spezifikationen zu denken und diese zu formulieren. Diese Spezifikationen und Stories sind für Fachexperten ebenso verständlich wie für Architekten und Entwickler. Wenn die Stories durchlaufen und somit die Systeme genau das tun, was sich der Kunde dabei gedacht hat, sind schneller alle Unstimmigkeiten beseitigt. Jeder kennt diese Situationen, in denen Kunden sich über falsch verstandene Anforderungen und falsch implementierte Abläufe beschweren. Stories sind der geeignete Weg, um solchen Aussagen zu entgehen.

Die Lesbarkeit der User Stories ist so gut, dass ganz nebenbei auch die Tests dokumentarischen Charakter bekommen. Grund genug, sich direkt mit der Entwicklung von User Stories zu befassen.

Eine Story beschreibt eine Anforderung, die eine Fachfunktionalität abdeckt. Dabei werden Randbedingungen formuliert, die am Ende die Akzeptanz der Funktionalität gewährleisten können.

Wenn wir nachfolgend mit Stories arbeiten, sollten Sie sich auf eine englischsprachige Darstellung einlassen, da die Schlüsselwörter der technischen Storyabläufe ebenfalls in Englisch gehalten sind.

Grundsätzlich können auch deutschsprachige Formulierungen verwendet werden, doch leidet dabei die Lesbarkeit erheblich. Eine weitere Alternative wäre die Eindeutschung von rSpec – User Stories mit Hilfe von Rubys Sprachmöglichkeiten –, doch ist das kein Thema für unser Buch.

Eine Story hat nach BDD folgende Grundstruktur.

```
Title (Eine markante einzeilige Betitelung)

Narrative:
As a <role>
I want <feature>
So what <benefit>

Acceptance Criteria (werden als Szenarien spezifiziert)

Scenario 1: Title
Given <context or initial state>
  And <more context or state>
```

---

<sup>7</sup> Siehe auch [www.behavior-driven.org](http://www.behavior-driven.org)

<sup>8</sup> [http://de.wikipedia.org/wiki/Framework\\_for\\_Integrated\\_Test](http://de.wikipedia.org/wiki/Framework_for_Integrated_Test)

```

When <event or action>
Then <outcome or result>
  And <another outcome or result>

Scenario 2: ...

```

Im Feeder-Beispiel kann das dann so aussehen.

**Listing 3.39** <http://pastie.org/234172>

```

Story: Managing Feeds

As a user
I want to manage feeds

Scenario: Adding a feed

Given a name, a url and a description for a new feed
And a number of existing feeds

When I save a feed

Then the feed should be valid
And there should be 1 more feed stored

```

Das obige Listing enthält keinerlei technische Angaben, sondern beschreibt, wie ein `Feed`-Objekt in das System gelangen soll. Sie werden im weiteren Verlauf diese textuelle Definition wieder verwenden. Daher sollten Sie sie in einer Datei namens `feed_story` im Verzeichnis `stories` abspeichern.

Mit der Implementierung in `rSpec` ist es möglich, mit wenig Aufwand diese Story in eine ablauffähige Variante zu überführen.

Legen Sie für diese Story einfach eine Datei namens `add_feed.rb` im Verzeichnis `stories/steps` an.

```

steps_for(:feeds) do
end

```

Mit diesem Block werden Schritte für die Story eingeleitet. In diesem Block starten Sie mit der ersten Kontextbeschreibung.

**Listing 3.40** <http://pastie.org/234175>

```

Given("a $name, a $url and a $description for a new feed")
do|name, url, description|
  @feed = Feed.new( :name => name, \
                   :url => url, \
                   :description => description)
end

```

In dieser Kontextbeschreibung finden Sie die Definition aus der textuellen Beschreibung wieder. Sie sehen ebenso die drei Platzhalter für die Variablenwerte `name`, `url` und `description`. `rSpec` kann beim Durchlaufen der User Stories diese Werte aus der textuellen Beschreibung übernehmen und somit variable Werte in den Ablauf einbringen. In diesem Fall werden die drei Werte für den Konstruktoraufwurf der Modellklasse `Feed` verwendet.

Fügen Sie des Weiteren folgende weitere Kontextbeschreibung ein.

**Listing 3.41** <http://pastie.org/234178>

```
Given("a number of existing feeds") do
  @count_feeds = Feed.count
end
```

Hier wird eine weitere Kontextbeschreibung aus der textuellen Beschreibung verwendet, um die aktuelle Anzahl an `Feed`-Objekten zwischenspeichern. Dies geschieht mit dem Aufruf der `count`-Methode auf der Modellklasse `Feed`.

Nachdem nun die beiden Kontextinformationen vorgegeben sind, widmen Sie sich der gewünschten Aktion.

**Listing 3.42** <http://pastie.org/234180>

```
When("I save a feed") do
  @feed.save
end
```

Mit der Aktion `save` wird das `Feed`-Objekt abgespeichert und mit den Ergebnisüberprüfungen fortgefahren.

**Listing 3.43** <http://pastie.org/234181>

```
Then("the feed should be valid") do
  @feed.should be_valid
end
```

In dieser Überprüfung des Ergebnisses sieht wieder alles wie bei den bereits bekannten Spezifikationen aus der Welt der Models und Controller aus. Hier stehen Ihnen sämtliche Expectations, die rSpec zu bieten hat, zur Verfügung.

**Listing 3.44** <http://pastie.org/234182>

```
Then("there should be $count more feed(s) stored") do |count|
  Feed.count.should == (@count_feeds + count.to_i)
end
```

Auch bei der letzten Ergebnisüberprüfung wird ein Platzhalter verwendet, um die Anzahl der nach dem Speichern vorhandenen `Feed`-Objekte zu überprüfen.

Bevor Sie nun diese Story ausführen können, müssen noch einige kleine Erweiterungen an der Datei `stories/helper.rb` vorgenommen werden.

**Listing 3.45** <http://pastie.org/234185>

```
ENV["RAILS_ENV"] = "test"
require File.expand_path(File.dirname(__FILE__) + "../config/environment")
require 'spec/rails/story_adapter'

dir = File.dirname(__FILE__)
Dir[File.expand_path("#{dir}/steps/*.rb")].uniq.each do |file|
  require file
end

def run_local_story(filename, options={})
  run File.join(File.dirname(__FILE__), filename), options
end
```

Mit diesen Änderungen werden alle Ruby-Quelldateien im Verzeichnis `stories/steps` geladen und bei Bedarf ausgeführt.

Legen Sie nun eine letzte neue Datei namens `feed_story.rb` an, um die Methode `run_local_story` aus dem oberen Listing laufen zu lassen.

**Listing 3.46** <http://pastie.org/234187>

```
require File.dirname(__FILE__) + "/helper"

with_steps_for(:feeds) do
  run_local_story "feed_story", :type => RailsStory
end
```

In dem Block wird der Methode der Name der textuellen Story-Beschreibung übergeben, die dann ausgeführt wird.

Jetzt können Sie einen Testlauf der Story starten. Hierzu geben Sie auf der Kommandozeile im Projektverzeichnis folgende Anweisung ein.

```
Running 1 scenarios
Story: Managing Feeds

  As a user
  I want to manage feeds

  Scenario: Adding a feed

    Given a name, a url and a description for a new feed
    And a number of existing feeds

    When I save a feed

    Then the feed should be valid
    And there should be 1 more feed stored

1 scenarios: 1 succeeded, 0 failed, 0 pending
```

Mit dieser Ausgabe erhalten Sie eine Menge an Informationen. Zum einen erscheint die komplette Story im Klartext. Zum anderen wird angegeben, wie viele Szenarien fehlerfrei oder fehlerhaft durchlaufen wurden.

Hier fällt auf, dass es sich bei diesem Test eher um einen Unittest handelt, da weder das Zusammenspiel von Controllern noch das Zusammenspiel von Controllern und Modellen überprüft wird.

### 3.4.2 Weitere Tools

Sie können sich nun die Frage stellen, ob es nicht leicht möglich ist, die Implementierung gegenüber den Tests so zu verändern, dass die Spezifikationen fehlerfrei durchlaufen, die Software allerdings völlig andere Dinge erledigt. Dies ist sicher eine unerwünschte Situation, die in der Praxis zu erheblichen Nachteilen führen kann.

Die Antwort auf diese Frage lautet: Ja, es ist durchaus möglich. Daher ist es wichtig, die Testabdeckung im Auge zu behalten. Je mehr Funktionalität in der Anwendung getestet wird, desto höher ist die Qualität des Codes.

Nun stellt sich nur noch die Frage, wie die Testabdeckung einfach und übersichtlich ermittelt werden kann. Hierzu gibt es das Werkzeug `rcov`, das im nächsten Abschnitt vorgestellt wird.

#### 3.4.2.1 `rcov`

`rcov` ist ein Werkzeug zur Ermittlung der Testabdeckung in funktionalen Tests und `rSpec` Stories. `rcov` wurde von Mauricio Fernandez entwickelt und kann unter <http://eigen-class.org/hiki.rb?rcov> heruntergeladen werden. Natürlich kann `rcov` auch als `gem` installiert werden. Hierzu geben Sie auf der Kommandozeile Folgendes ein.

```
$ sudo gem install rcov
```

Die nun folgende Ausgabe zeigt die erfolgreiche Installation an.

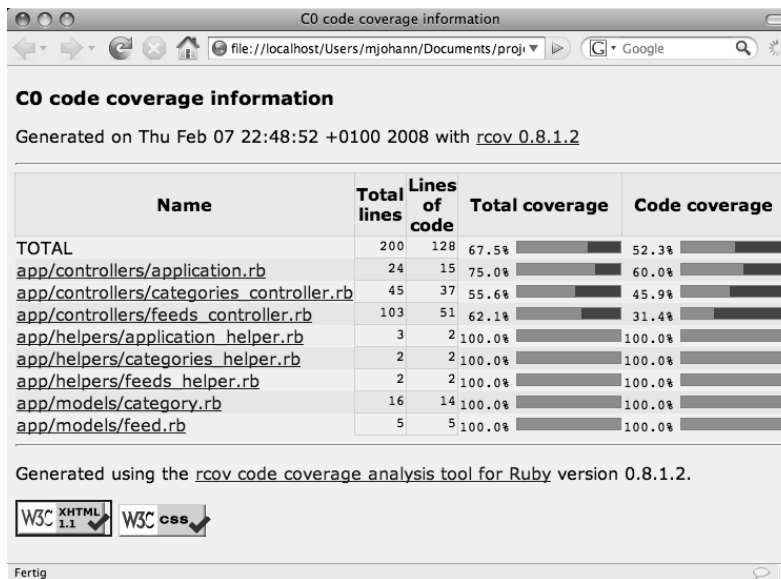
```
Building native extensions. This could take a while...
Successfully installed rcov-0.8.1.2.0
1 gem installed
Installing ri documentation for rcov-0.8.1.2.0...
Installing RDoc documentation for rcov-0.8.1.2.0...
```

Danach steht `rcov` für jedes Projekt zur Verfügung. Im Zusammenhang mit dem `feeder`-Projekt wurde durch `rSpec` folgender Rake Task hinzugefügt.

```
$ rake spec:rcov
```

Wenn Sie diesen Task in der Kommandozeile ausführen, legt `rcov` im Verzeichnis `coverage` einige HTML-Dateien an, die sämtliche Statistiken über die Testabdeckung bereithalten. Sie müssen lediglich die Datei `coverage/index.html` in Ihrem Browser öffnen.

Aktuell sieht eine `rcov`-Auswertung in etwa so aus.



**Abbildung 3.2**  
Eine Auswertung von `rcov` im Firefox

Die Links verzweigen beim Anklicken in den Quellcode der Anwendung. Dort werden die nicht von Tests abgedeckten Bereiche hervorgehoben. Sicher ist in der Praxis eine hundertprozentige Testabdeckung nicht möglich und auch nicht unbedingt notwendig. Allerdings sind achtzig Prozent schon relativ leicht zu erreichen.

`rcov` bescheinigt unserem Feeder-Projekt eine Abdeckung zwischen rund dreißig und hundert Prozent. Die Modellklassen sind beinahe ohne Code ausgestattet und daher leicht hundertprozentig zu testen. Die Controller-Klassen sehen jedoch etwas komplexer aus.

Wenn wir uns das Beispiel `CategoriesController` näher anschauen (Abbildung 3.3), stellen wir schnell fest, dass die Methoden `edit`, `update` und `destroy` nicht getestet wurden. Dies sollte nicht sein und wird im Folgenden geändert.

**C0 code coverage information**  
Generated on Thu Feb 07 22:48:52 +0100 2008 with `rcov 0.8.1.2`

Code reported as executed by Ruby looks like this...  
and this: this line is also marked as covered.  
Lines considered as run by `rcov`, but not reported by Ruby, look like this,  
and this: these lines were inferred by `rcov` (using simple heuristics).  
Finally, here's a line marked as not executed.

Name	Total lines	Lines of code	Total coverage	Code coverage
<code>app/controllers/categories_controller.rb</code>	45	37	55.6%	45.9%

```

1 class CategoriesController < ApplicationController
2   layout "feeds"
3   #before_filter :authenticate, :except => ['index', 'show', 'new']
4
5   def index
6     @categories = Category.find(:all)
7   end
8
9   def new
10    @category = Category.new
11  end
12
13  def create
14    @category = Category.new(params[:category])
15    if @category.save
16      flash[:notice] = "Kategorie #{@category.name} wurde gespeichert."
17      redirect_to :action => 'show', :id => @category
18    else
19      render :action => 'new'
20    end
21  end
22
23  def destroy
24    Category.find(params[:id]).destroy
25    redirect_to :action => 'index'
26  end
27
28  def edit
29    @category = Category.find(params[:id])
30  end
31
32  def update
33    @category = Category.find(params[:id])
34    if @category.update_attributes(params[:category])
35      flash[:notice] = "Kategorie #{@category.name} aktualisiert."
36      redirect_to :action => 'show', :id => @category
37    else
38      render :action => 'edit'
39    end
40  end
41
42  def show
43    @category = Category.find(params[:id])
44  end
45 end

```

Fertig

Abbildung 3.3 Die rot markierten Bereiche wurden bisher nicht von Tests abgedeckt.

Hierzu werden für die Spezifikationen des Controllers `CategoriesController` die folgenden Examples hinzugefügt.

**Listing 3.47** <http://pastie.org/234200>

```
describe CategoriesController, "rcov" do
  ...
end
```

In diese Spezifikation bauen wir die folgenden Examples ein.

**Listing 3.48** <http://pastie.org/234201>

```
before(:each) do
  @category = mock_model(Category)
  controller.stub!(:authenticate).and_return(true)
  @category.stub!(:new_record?).and_return(false)
  @category.stub!(:new).and_return(@category)
  Category.stub!(:find).and_return(@category)
  @category.stub!(:destroy).and_return(true)
end
```

**Listing 3.49** <http://pastie.org/234201>

```
before(:each) do
  @category = mock_model(Category)
  controller.stub!(:authenticate).and_return(true)
  @category.stub!(:new_record?).and_return(false)
  @category.stub!(:new).and_return(@category)
  Category.stub!(:find).and_return(@category)
  @category.stub!(:destroy).and_return(true)
end
```

Der Before-Filter ist identisch mit dem der ersten Spezifikation. Neu hinzu kommt nun das erste Example, das die `edit`-Action aus dem `CategoriesController` testet.

**Listing 3.50** <http://pastie.org/234202>

```
it "should edit a category with GET to /categories/1/edit" do
  get 'edit', :id => 1
  response.should be_success
  response.should render_template('categories/edit')
end
```

Hier wird zunächst mit `get` die Action `edit` aufgerufen und danach der HTTP-Returncode 200 überprüft. Eine weitere Expectation überprüft, ob im Anschluss die Seite `categories/edit.html.erb` angezeigt wird.

**Listing 3.51** <http://pastie.org/234203>

```
it "should destroy a category with DELETE" do
  delete 'destroy', :id => 1
  response.should redirect_to(categories_url)
end
```

Das obige Example prüft, ob eine Umleitung auf die `index`-View erfolgt.

**Listing 3.52** <http://pastie.org/234204>

```

it "should show a category with a given id" do
  get :show, :id => 1
  response.should be_success
  response.should render_template('categories/show')
end

```

Im obigen Example wird die `show`-Action überprüft. Hier wird eine `Category`-Instanz angefordert und überprüft, ob der Returncode 200 (success) ist. Zum Schluss wird geprüft, ob die Seite `categories/show.html.erb` angezeigt wird.

**Listing 3.53** <http://pastie.org/234205>

```

it "should not create a category with given identical parameters" do
  @category.stub!(:save).and_return(false)
  post 'create', :category => { :name => "Test1",
                               :description => "Test1" }
  response.should render_template('categories/new')
end

```

Ein weiteres Beispiel prüft gleichzeitig, ob eine Instanz von `Category` angelegt werden kann, wenn die Parameter `name` und `description` identisch sind. Laut `validate`-Methode des Modells `Category` sollte es in diesem Fall zu einem Fehler kommen. Im Fehlerfall wird auf die Seite `categories/new.html.erb` verzweigt, was hier ebenfalls überprüft wird.

**Listing 3.54** <http://pastie.org/234209>

```

it "should update a category with \
  given parameters and be successful" do
  @category.stub!(:find).with("id" => 1).and_return(@category)
  Category.should_receive(:find).and_return(@category)
  @category.should_receive(:update_attributes).and_return(true)
  @category.stub!(:name)
  post 'update', {:category => { :name => "Test23",
                                :description => "Test45"}, :id => 1}
  response.should redirect_to(category_url(@category))
end

it "should update a category with \
  given parameters and be successful" do
  @category.stub!(:find).with("id" => 1).and_return(@category)
  Category.should_receive(:find).and_return(@category)
  @category.should_receive(:update_attributes).and_return(false)
  @category.stub!(:name)
  post 'update', {:category => { :name => "Test23",
                                :description => "Test45"}, :id => 1}
  response.should render_template('categories/edit')
end

```

Diese Examples in einer zusätzlichen Spezifikation sorgen dafür, dass `rcov` uns eine 100-prozentige Testabdeckung bescheinigt (siehe auch Abbildung 3.4 auf der nächsten Seite).

`rcov` ist also ein hervorragendes Werkzeug, um viel bessere Spezifikationen zu schreiben. Auch die Problematik, die darin besteht, dass Code so modifiziert werden kann, dass die Tests immer noch durchlaufen werden, lässt sich hier in den Griff bekommen.

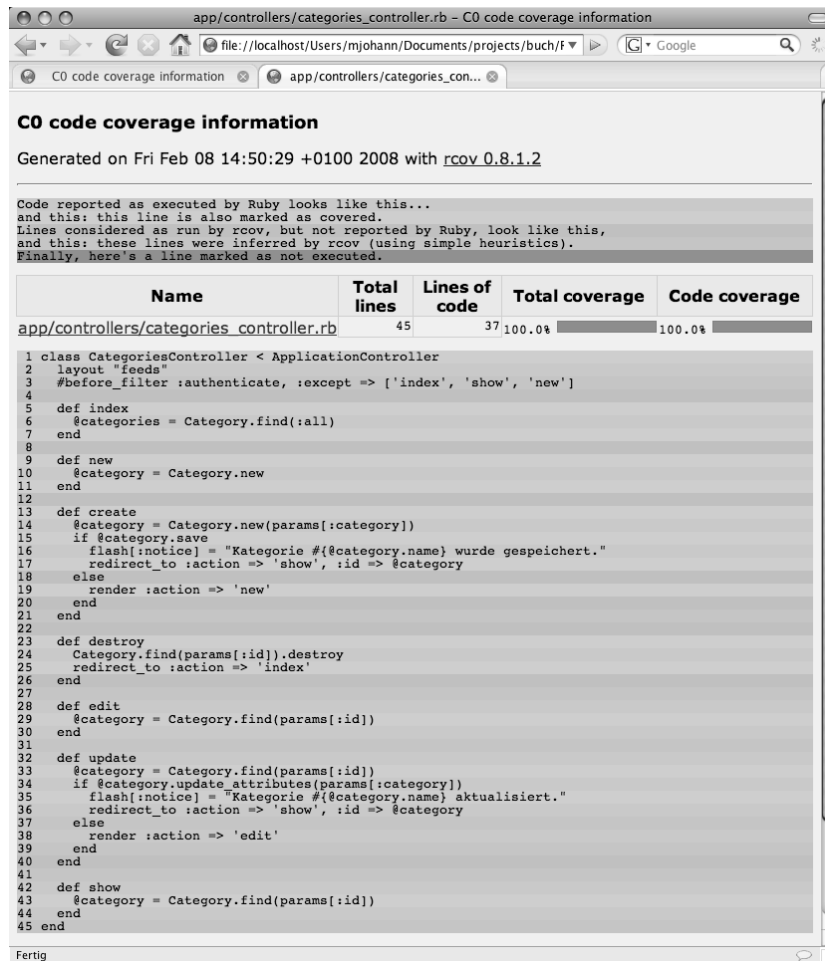


Abbildung 3.4 So sehen 100 % Testabdeckung aus.

### 3.5 Zusammenfassung

In diesem Kapitel erfuhren Sie, wie anstelle von Unittests Spezifikationen entstehen, um die Qualität der eigenen Software zu überprüfen. Hierfür wurde der Ansatz des Behavior Driven Development angewandt. Außerdem lernten Sie folgende Aspekte kennen:

- Konzepte von BDD
- rSpec – Spezifikationen, Examples und Expectations
- User Stories
- Test-Coverage mit rcov

Im nächsten Kapitel erfahren Sie mehr zum Thema Versionierung mit Mercurial und Git, den beiden verteilten Versionskontrollsystemen.