




# Produktiv programmieren

 REILLY®

Neal Ford  
Deutsche Übersetzung von Jörg Staudemeyer

# INHALTSVERZEICHNIS

ZUM GELEIT	IX
VORWORT	XI
<b>1 EINFÜHRUNG</b>	<b>1</b>
<i>Warum ein Buch über Produktivität?</i>	2
<i>Worum es in diesem Buch geht</i>	4
<i>Und wie geht es weiter?</i>	7
<b>Teil 1</b> <b>MECHANISMEN</b>	<b>9</b>
<hr/>	
<b>2 BESCHLEUNIGUNG</b>	<b>11</b>
<i>Launcher</i>	12
<i>Beschleuniger</i>	24
<i>Makros</i>	44
<i>Zusammenfassung</i>	47
<b>3 FOKUS</b>	<b>49</b>
<i>Ablenkungen beseitigen</i>	50
<i>Suche übertrumpft Navigation</i>	53
<i>Dinge suchen, die schwer zu finden sind</i>	56
<i>Verwenden Sie Rooted Views</i>	59
<i>Verwenden Sie dauerhafte Attribute</i>	61
<i>Verwenden Sie projektbezogene Verknüpfungen</i>	62
<i>Multiplizieren Sie Ihre Monitore</i>	63
<i>Teilen Sie Ihren Arbeitsbereich virtuell auf</i>	63
<i>Zusammenfassung</i>	66
<b>4 AUTOMATISIERUNG</b>	<b>67</b>
<i>Erfinden Sie das Rad nicht neu</i>	69
<i>Speichern Sie Dinge lokal</i>	70
<i>Automatisieren Sie die Interaktion mit Websites</i>	71
<i>Interagieren Sie mit RSS-Feeds</i>	72
<i>Mit Ant nicht nur Build-Prozesse steuern</i>	73

	<i>Rake für allgemeine Aufgaben verwenden</i>	75
	<i>Mit Selenium Webseiten auslesen</i>	77
	<i>Mit Bash Exceptions zählen</i>	79
	<i>Ersetze BAT durch PowerShell</i>	80
	<i>Mit Mac OS X Automator alte Downloads löschen</i>	81
	<i>Die Subversion-Befehlszeile zähmen</i>	83
	<i>Einen SQL-Splitter mit Ruby bauen</i>	84
	<i>Warum Automatisierung wichtig ist</i>	86
	<i>Rasieren Sie keine Grunzochsen</i>	89
	<i>Zusammenfassung</i>	90
<b>5</b>	<b>KANONITÄT</b>	91
	<i>DRY in der Versionskontrolle</i>	93
	<i>Verwenden Sie einen kanonischen Build-Server</i>	95
	<i>Indirektion</i>	97
	<i>Nutzen Sie die Virtualisierung</i>	106
	<i>DRY und der Impedance-Mismatch</i>	107
	<i>DRY in der Dokumentation</i>	116
	<i>Zusammenfassung</i>	124
<b>Teil 2</b>	<b>PRAXIS</b>	125
<hr/>		
<b>6</b>	<b>TESTGETRIEBENES DESIGN</b>	127
	<i>Tests entstehen lassen</i>	129
	<i>Testabdeckung</i>	137
<b>7</b>	<b>STATISCHE ANALYSE</b>	141
	<i>Bytecode-Analyse</i>	142
	<i>Quellcode-Analyse</i>	145
	<i>Metriken generieren mit Panopticode</i>	147
	<i>Analyse dynamischer Sprachen</i>	150
<b>8</b>	<b>GUTE MITBÜRGER</b>	153
	<i>Die Kapselung durchbrechen</i>	154
	<i>Konstruktoren</i>	156
	<i>Statische Methoden</i>	156
	<i>Kriminelles Verhalten</i>	162
<b>9</b>	<b>YAGNI</b>	165

<b>10</b>	ALTE PHILOSOPHEN	173
	<i>Aristoteles' essenzielle und akzidenzielle Eigenschaften</i>	174
	<i>Ockhams Skalpell</i>	176
	<i>Das Gesetz der Demeter</i>	181
	<i>Softwareüberlieferungen</i>	182
<b>11</b>	AUTORITÄTEN INFRAGE STELLEN	185
	<i>Böse Affen</i>	186
	<i>Fluent-Interfaces</i>	188
	<i>Antiobjekte</i>	190
<b>12</b>	META-PROGRAMMIERUNG	193
	<i>Java und Reflection</i>	194
	<i>Java mit Groovy testen</i>	196
	<i>Fluent-Interfaces schreiben</i>	198
	<i>Wohin führt uns Meta-Programmierung?</i>	200
<b>13</b>	COMPOSED-METHOD UND SLAP	201
	<i>Composed-Method im Einsatz</i>	202
	<i>SLAP</i>	208
<b>14</b>	POLYGLOTTES PROGRAMMIEREN	215
	<i>Wie sind wir hierher gekommen? Und was heißt »hier«?</i>	216
	<i>Wo gehen wir hin? Und wie kommen wir dort hin?</i>	220
	<i>Olas Pyramide</i>	226
<b>15</b>	PERFEKTE WERKZEUGE	229
	<i>Das Streben nach dem perfekten Editor</i>	230
	<i>Die Kandidaten</i>	234
	<i>Das richtige Werkzeug für meinen Job</i>	236
	<i>Die falschen Werkzeuge vermeiden</i>	244
<b>16</b>	SCHLUSSFOLGERUNG: DAS GESPRÄCH FORTFÜHREN	249
	ANHANG: BAUSTEINE	253
	INDEX	263



## KAPITEL ELF

**Autoritäten infrage stellen**

**GENERELL IST EINE STANDARDISIERUNG ÜBER ENTWICKLUNGSTEAMS UND -COMMUNITIES HINWEG** eine gute Sache. Sie ermöglicht den Leuten, den Code von anderen leichter zu lesen, Idiome besser zu verstehen und übertrieben idiomatische Programmierung zu vermeiden (vielleicht mit Ausnahme der Perl-Community)<sup>1</sup>.

Aber Standards blind zu befolgen, ist auch nicht besser, als gar keine Standards zu haben. Manchmal verhindern Standards sinnvolle Sonderwege. Bei allem, was Sie in der Softwareentwicklung tun, sollten Sie wissen, *warum* Sie es tun. Andernfalls werden vielleicht die bösen Affen es Sie büßen lassen.

## Böse Affen

Diese Geschichte hörte ich zum ersten Mal von Dave Thomas, der eine Keynote-Ansprache mit dem Titel *Angry Monkeys and Cargo Cults* hielt. Ich weiß nicht, ob sie wahr ist (obwohl ich ein wenig darüber geforscht habe), aber das ist eigentlich auch egal, denn sie ist eine wunderbare Illustration für das Thema, um das es hier geht.

Damals in den 1960ern (als die Wissenschaftler noch alle möglichen verrückten Sachen machen durften) führten Verhaltenswissenschaftler folgendes Experiment durch: Sie setzten fünf Affen in einen Raum, in dem sich eine Trittleiter befand und ein Bündel Bananen von der Decke hing. Die Affen bekamen schnell heraus, dass sie auf die Leiter klettern mussten, um an die Bananen zu kommen. Aber jedes Mal, wenn sich ein Affe der Leiter näherte, überschütteten die Wissenschaftler den ganzen Raum mit eiskaltem Wasser. Sie können sich vorstellen, was das zur Folge hatte: böse Affen. Nach kurzer Zeit kam kein Affe mehr in die Nähe der Leiter.

Dann ersetzten die Wissenschaftler einen der Affen durch einen anderen, der die kalten Duschen noch nicht kannte. Der ging natürlich als Erstes schnurstracks auf die Leiter zu, und alle anderen Affen fingen an, auf ihn einzuschlagen. Er wusste zwar nicht, warum er verhaßt wurde, lernte aber schnell: Bleibe fern von der Leiter. Nach und nach ersetzten die Wissenschaftler alle ursprünglich zur Gruppe gehörenden Affen, bis keiner von ihnen mehr mit kaltem Wasser übergossen worden war. Trotzdem fuhren sie fort, jeden Affen zu attackieren, der sich der Leiter zu nähern versuchte.

Und die Moral von der Geschichte? In Softwareprojekten gibt es viele Praktiken nur deshalb, weil »wir es schon immer so gemacht haben«. Oder anders gesagt: wegen der bösen Affen.

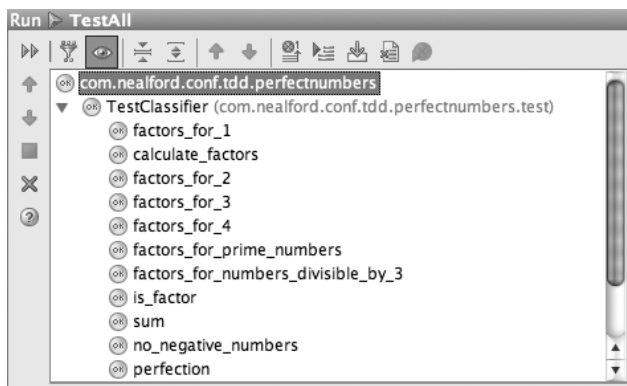
---

<sup>1</sup> Das soll ein Scherz sein. Ich mag euch, Leute! Sendet mir bitte keine E-Mails!

Das folgende Beispiel entstammt einem meiner früheren Projekte. Jeder weiß, dass in Java alle Methodennamen mit einem Kleinbuchstaben beginnen und dann als CamelCase geschrieben werden sollen, wobei die Wortgrenzen durch Großbuchstaben angezeigt werden. Für die normale Programmierung ist das in Ordnung, aber bei Namen von Tests ist es etwas anderes. Hier ist es praktisch, lange, deskriptive Namen zu verwenden, sodass man leicht erkennen kann, was getestet wird. Leider sind aber LangeCamelCaseNamenSchwerZuLesen. In jenem Projekt schlug ich daher vor, zwischen den Wörtern der Testnamen Unterstriche einzufügen, wie beispielsweise hier:

```
public void testUpdateCacheAndVerifyItemExists() {  
    }  
  
public void test_Update_cache_and_verify_item_exists() {  
    }  
}
```

Meiner Meinung nach waren die Namen mit den Unterstrichen weit besser lesbar. Die Reaktion des Entwicklungsteams auf meinen Vorschlag war sehr interessant. Einige der Entwickler begrüßten die Idee sofort, während andere mich lediglich aufgrund meiner Anregung ansahen wie die bösen Affen. Wir einigten uns schließlich darauf, diesen Benennungsstil zu verwenden (manchmal müssen technische Leiter auch wohlwollende Diktatoren sein), und stellten später fest, dass die Testnamen tatsächlich viel besser lesbar waren, insbesondere beim Betrachten langer Listen von Namen im Testrunner einer IDE (siehe Abbildung 11-1).



**Abbildung 11-1:** *Unterstrichene Testnamen sind besser lesbar*

»Weil wir es schon immer so gemacht haben« ist beim Entwickeln kein ausreichender Grund dafür, alte Gewohnheiten zu befolgen. Wenn Sie verstehen,

warum Sie es schon immer so gemacht haben, ist es in jedem Fall sinnvoll, damit fortzufahren. Aber Sie sollten stets nach deren Voraussetzungen fragen und ihre Gültigkeit prüfen.

## Fluent-Interfaces

Ein *Fluent-Interface* (fließende Schnittstelle) ist eine Art Domain-spezifische Sprache (DSL), wie sie zurzeit en vogue sind. In einem Fluent-Interface stellt man lange Codesequenzen zu Sätzen zusammen. Dies folgt der Überlegung, dass komplette Gedankengänge auch in der gesprochenen Sprache diese Form annehmen. Programmcode in diesem Stil ist leichter zu lesen, weil man genau so wie bei umgangssprachlichen Sätzen besser erkennen kann, wo ein Gedanke endet und der nächste beginnt.

Hier ein Beispiel, das auf einem meiner Projekte basiert. Wir bauten eine Anwendung, bei der es um Eisenbahnwaggons ging, und zu jedem Wagen gab es eine Marketingbeschreibung. Es gibt für Eisenbahnwaggons zahlreiche Regeln und Vorschriften, daher war es schwierig, die Testszenarien korrekt aufzustellen. Ständig mussten wir unsere Geschäftsprozessanalytiker fragen, ob wir den Waggontyp, den wir gerade testen mussten, in allen Nuancen korrekt definiert hatten. Das Folgende ist eine vereinfachte Version von dem, was wir ihnen in die Hand gaben:

```
Car car = new CarImpl();
MarketingDescription desc = new MarketingDescriptionImpl();
desc.setType("Box");
desc.setSubType("Insulated");
desc.setAttribute("length", "50.5");
desc.setAttribute("ladder", "yes");
desc.setAttribute("lining type", "cork");
car.setDescription(desc);
```

Was für einen Java-Entwickler völlig normal aussieht, war den Analytikern verhasst. »Warum gebt ihr mir Java-Programme? Sagt mir doch einfach, was ihr meint?« Jede Übersetzung birgt aber die Gefahr von Fehlern. Um das Problem zu entschärfen, führten wir ein Fluent-Interface ein, mit dem wir dieselben Informationen in einer anderen Form erfassen konnten:

```
Car car = Car.describedAs()
    .box()
    .length(50.5)
    .type(Type.INSULATED)
    .includes(Equipment.LADDER)
    .lining(Lining.CORK);
```

Dies gefiel unseren Analytikern schon deutlich besser. Wir erreichten damit, dass wir einen Großteil der fragwürdigen Redundanz loswurden, wie sie der »normale« Stil von Java-APIs erfordert. Die Implementierung war sehr einfach. Alle Property-Setter gaben `this` anstelle von `void` zurück und ermöglichten so, durch Verkettung von Methodenaufrufen Sätze zu bilden. Die Implementierung von `Car` sah etwa so aus:

```
public class Car {
    private MarketingDescription _desc;

    public Car() {
        _desc = new MarketingDescriptionImpl();
    }

    public static Car describedAs() {
        return new Car();
    }

    public Car box() {
        _desc.setType("box");
        return this;
    }

    public Car length(double length) {
        _desc.setLength(length);
        return this;
    }

    public Car type(Type type) {
        _desc.setType(type);
        return this;
    }

    public Car includes(Equipment equip) {
        _desc.setAttribute("equipment", equip.toString());
        return this;
    }

    public Car lining(Lining lining) {
        _desc.setLining(lining);
        return this;
    }
}
```

Dies ist zugleich ein Beispiel für ein DSL-Pattern, das man als Expression-Builder bezeichnet. Die Klasse `Car` verbirgt die Tatsache, dass sie intern ein `MarketingDescription`-Objekt zusammensetzt. Ein Expression-Builder kapselt interne Ausdrücke durch öffentliche Interfaces und vereinfacht so das Fluent-

Interface. Damit die Methodenaufrufe verkettet werden können, liefert jede der Mutator-Methoden von `Car` als Ergebnis `this`.

Aber warum packen wir dieses Beispiel in ein Kapitel, in dem es um die Infragestellung von Autoritäten geht? Wenn Sie ein Fluent-Interface wie `Car` schreiben, schlachten Sie damit eine heilige Kuh der Java-Gemeinde: Nun ist die Klasse `Car` keine `JavaBean` mehr. Das kommt Ihnen vielleicht nicht so wesentlich vor, aber die Java-Infrastruktur verlangt an vielen Stellen die Einhaltung dieser Spezifikation. Und wenn Sie sich die `JavaBeans`-Spezifikation etwas genauer ansehen, entdecken Sie dort noch mehr Dinge, die der Gesamtqualität Ihres Codes schaden können.

Die `JavaBeans`-Spezifikation besteht darauf, dass jedes Objekt einen Default-Konstruktor haben muss (siehe »Konstruktoren« in Kapitel 8), dabei kann ein Objekt ohne Zustand praktisch nie gültig sein. Außerdem verordnet uns die `JavaBeans`-Spezifikation die hässliche Syntax für `Java-Properties` mit `getXXX()`-Methoden als Akzessoren und `setXXX()`-Methoden für Mutatoren, wobei Letztere als `void` definiert sein müssen. Man kann verstehen, warum es diese Beschränkungen gibt (beispielsweise erleichtert der Default-Konstruktor die Serialisierung), aber niemand in der Java-Welt stellt sich die Frage, warum unbedingt jedes Objekt eine `Bean` sein muss. Man folgt einfach nur den bösen Affen und macht aus allem eine `JavaBean`.

Stellen Sie die Autoritäten infrage. Wenn Sie jedes Objekt zu einer `Bean` machen, können Sie keine Fluent-Interfaces mehr verwenden. Überlegen Sie, was Sie programmieren, verstehen Sie, wofür es gebraucht wird, und entscheiden Sie weise. »Weil wir es schon immer so gemacht haben« ist in den seltensten Fällen die richtige Antwort.

## Antiobjekte

Bisweilen kann die Autorität, die Sie infrage stellen sollten, auch Ihre eigene Neigung zu einer bestimmten Art und Weise der Problemlösung sein. Auf der OOPSLA-Konferenz von 2006 erschien ein großartiges Papier mit dem Titel *Collaborative Diffusion: Programming Anti-Objects*<sup>2</sup>. Die Autoren des Papiers stellen die Behauptung auf, dass Objekte und Objekthierarchien zwar einen exzellenten Abstraktionsmechanismus für die meisten Probleme bieten, dass aber in manchen

---

2 Download unter <http://www.cs.colorado.edu/~ralex/papers/PDF/OOPSLA06antiobjects.pdf>.

Fällen dieselben Abstraktionen Probleme komplexer machen. Hinter den »Antiobjekten« steht die Idee, dass man den wahrgenommenen Vordergrund mit dem Hintergrund des Problems vertauscht und das einfachere, weniger offensichtliche Problem löst. Was heißt hier »Vordergrund« und »Hintergrund«? Ein Beispiel macht dies deutlich. (Achtung! Wenn Sie immer noch gern *PacMan* spielen, sollten Sie die nächsten Absätze lieber nicht lesen – sie werden Ihnen für immer den Spaß verderben! Wissen hat manchmal seinen Preis.)

Denken Sie an das Konsolenspiel *PacMan*. Als es in den 1970er-Jahren herauskam, verfügte es über weniger Rechenleistung als ein billiges Mobiltelefon von heute. Trotzdem musste es ein wirklich schwieriges mathematisches Problem lösen: Wie kriegt man den Geist dazu, *PacMan* durch das Labyrinth zu jagen? Mit anderen Worten: Wie berechne ich die kürzeste Entfernung durch ein Labyrinth zu einem beweglichen Ziel? Ein nicht triviales Problem, insbesondere wenn man nur wenig Speicher und Prozessorleistung zur Verfügung hat. Die *PacMan*-Entwickler versuchten gar nicht erst, es zu lösen, sondern verfielen auf den Antiobjekt-Ansatz: Sie bauten die Intelligenz in das Labyrinth ein, nicht in den Geist.

Das Labyrinth in *PacMan* agiert wie ein Automat (ähnlich wie Conways *Game of Life*). Jeder Zelle sind ein paar einfache Regeln zugeordnet, und diese werden Zelle für Zelle von links oben nach rechts unten nacheinander ausgeführt. Jede Zelle merkt sich einen Wert für den »*PacMan*-Geruch«. Wenn *PacMan* unmittelbar auf einer Zelle sitzt, ist der *PacMan*-Geruch dort am höchsten. Hat er diese Zelle gerade verlassen, ist der *PacMan*-Geruch  $-1$ . Der Geruch nimmt bei jeder Runde etwas ab und verschwindet schließlich. Nun können die Geister ganz dumm sein: Sie schnüffeln einfach nach dem *PacMan*-Geruch, und immer wenn sie auf ihn stoßen, gehen sie zu der Zelle weiter, die den höheren Geruchswert hat.

Als »naheliegendste« Lösung des Problems würde man die Intelligenz in die Geister einbauen. Aber die viel einfachere Lösung ist, die Intelligenz in das Labyrinth zu bringen. Und genau dies ist der Antiobjekt-Ansatz: Vertauschen Sie den logischen Vordergrund mit dem logischen Hintergrund. Fallen Sie nicht der Annahme zum Opfer, eine »traditionelle« stelle immer die richtige Lösung dar. Und manchmal lässt sich ein bestimmtes Problem auch leichter in einer ganz anderen Sprache lösen. (Siehe Kapitel 14 zu den Überlegungen hinter dem Antiobjekt-Ansatz.)