5. Diskurs, Ausblick & Fazit

»You know you've achieved perfection in design,
not when you have nothing more to add,
but when you have nothing more to take away.«
—Antoine de Saint-Exupéry

5.1 Lesbarkeit als Qualitätskriterium

Lesbarkeit ist ein subjektives, also „weiches“ Kriterium für Softwarequalität. Sie lässt sich nicht quantitativ messen – objektiv wird aber jeder Softwareentwickler bestätigen können, dass lesbarer Quelltext leichter zu verstehen und zu ändern ist.

Wir haben gesehen, dass gerade das angesprochene Verändern von Quelltext nach der Auslieferung bei Web-Applikationen einen wesentlich höheren Stellenwert einnimmt als bei Desktop-Applikationen.

Zwei Aspekte der Lesbarkeit wurden im Verlauf dieser Arbeit besonders betont: das erste ist die Nähe der Programmiersprache zur Fachsprache der Problemdomäne. Dieser Anspruch kann über domänenspezifische Sprachen realisiert werden. Der andere Aspekt ist die verständliche Struktur des Gesamtsystems. Hier müssen unter anderem unnötige Wiederholdungen, hardwarenahe Programmierungen und komplizierte Algorithmen vermieden werden. Der bereits mehrfach zitierte Martin Fowler[61] umschreibt es so:

> *“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”*

Die vorgestellten Konzepte tragen wesentlich zur Produktivitätsverbesserung bei. ActiveRecord fordert zwar von Entwicklern die Einhaltung gewisser (sinnvoller) Konventionen, garantiert im Gegenzug dafür dann Erleichterungen bei der Softwareherstellung und -wartung.

Objektorientierung entspricht zwar schon sehr gut unserem Verständnis der Welt, geht aber oft nicht weit genug. In solchen Fällen sollte eine domänenspezifische (Programmier-) Sprache entwickelt werden – oder, wie im Falle von Ruby, die Programmiersprache selbst um domänenspezifische Ausdrücke erweitert werden.

Kurzfristig ist die Konzeption und Programmierung einer solchen DSL sicherlich aufwändiger und kostenintensiver als die direkte Lösung des Problems mit konventioneller Programmierung. Auf lange Sicht wird aber der Entwicklungsprozess nachhaltig erleichtert und beschleunigt: Bei entsprechenden, fachlichen Vorkenntnissen wird die Wiedereinarbeitungszeit in den Quelltext drastisch reduziert. Die DSL kann außerdem leicht Tests versehen werden, um das Vertrauen der Entwickler in die Sprache zu verbessern.

Außerdem werden Software-Reviews von Programmen in dieser DSL wesentlich vereinfacht. Allerdings muss dazu auch der Auftraggeber über seine Rolle während der Entwicklung im Klaren sein. Nur durch intensive Beteiligung des Kunden (als bester Kenner der Fachsprache) macht die Entwicklung und der Einsatz einer domänenspezifischen Sprache Sinn.

Es gibt Ansätze, auch test-[62] beziehungsweise verhaltensgetriebene[63] Entwicklung entsprechend zu unterstützen. Im Anhang B findet sich ein Beispiel zum Testen und Spezifizieren eines Kellerspeichers mit „RSpec“[64]. Auch dort wird lesbarer Code geschätzt – im Gegensatz zu konventionellen Unit-Tests, die sich eher an der verwendeten Computersprache orientieren.

Bisher war der bevorzugte Weg, qualitätiv hochwertigeren Code über sogenannte „Refactorings“[65] zu erreichen. Mit dieser Methode wird Quelltext sinnvoll umzustrukturiert, ohne die Funktionsweise zu verändern. Mit bereits lesbarem Quelltext kann diese Phase wesentlich verkürzt werden.

Lesbarkeit lässt sich aber nicht pauschal für alle Anwendungsgebiet der Informatik fordern. Im Bereich von „Embedded Software“ kommt es beispielsweise auf eine sehr bedachte Ressourcennutzung an. Dort müssen mit Assemblersprachen auch Optimierungen knapp oberhalb der Hardwareebene durchgeführt werden. Die Verwendung eines komplizierten, aber effizienten Algorithmus macht dort im Einzelfall sicher mehr Sinn.

---

[62] Test Driven Development (TDD); mehr unter [WL18]
[63] Behaviour Driven Development (BDD); mehr unter [WL19]
[64] mehr unter [WL20]; von dort stammt auch das Beispiel
[65] siehe auch [MFR] und [JKRP]

Der Lesbarkeit von Quelltext sind aber auch formale Grenzen gesetzt: nach wie vor müssen Computer mit Algorithmen programmiert werden. Die Syntax der meisten Programmiersprachen tritt dabei oft in den Vordergrund. Viele Probleme werden deshalb immer noch mit prozeduraler Programmierung gelöst und nicht, wie mit Ruby möglich, auf deklarative Weise.

5.2 Auseinandersetzung mit Ruby

Die Programmiersprache Ruby zeichnet sich durch ihre weitreichende Flexiblität aus. Die Sprache kann so erweitert werden, dass sie sich dem Problem anpasst. Damit eignet sie sich besonders gut für die Realisierung einer domänenspezifischen Sprache.

Viele Konzepte in Ruby mögen auf den ersten Blick befremdlich erscheinen (Symbols, Blocks). Die Lernkurve ist damit sicher nicht so steil wie bei der Programmiersprache PHP, die bekanntlich einen recht schnellen Einstieg in die Webprogrammierung ermöglicht. Um Ruby vollständig nutzen zu können, sollte man also bereits Erfahrungen mit Objektorientierung und Softwareentwicklung vorweisen können.

Als großer Kritikpunkt an Ruby wird immer wieder mangelhafte Performance genannt. Die Kritik ist insofern berechtigt, als dass Ruby in der Tat nicht zu den schnellsten Programmiersprachen gehört – zumindest nicht, was Laufzeitperformance angeht. In den meisten Fällen ist Ruby aber *schnell genug*.

Ruby hat gar nicht den Anspruch, möglichst schnell zu sein. Rubys Paradigma lautet, die Produktivität des Programmierers zu fördern. Wir müssen schon lange nicht mehr vorbehaltslos Machinencode von Hand optimieren, um beschränkte Computerressourcen möglichst gut auszunutzen. Vielmehr ist heute Entwicklungszeit das knappe und kostbare Gut. Ruby nimmt daher dem Programmierer im Quelltext sich wiederholende Tätigkeiten ab und stellt sie ihm als (getestete) Funktionen und Methoden zur Verfügung.

5.3 Pro & Contra Ruby on Rails

Ruby on Rails überzeugt mit einem durchdachten und konsistenten Konzept: Durch eine saubere Trennung der verschiedenen Aspekte einer Webanwendung wird gute Wartbarkeit erreicht. Die konsequente und intelligente Verwendung von Ruby sorgt für hohe Produktivität bei der Entwicklung. Dabei orientiert sich Rails eher an Erfahrungen aus der Praxis als an akademischer Lehre.

David Heinermeier Hansson beschreibt den Leistungsumfang von Rails mit den Worten[66]:

> *„Rails does, what most people do most of the time.“*

Damit drückt er zum einen aus, dass bereits ein großer Umfang der Web-Funktionalität in Ruby on Rails umgesetzt wurde, die mit anderen Sprachen und Frameworks erst noch programmiert werden muss. Der andere Aspekt dieser Aussage ist der, dass es durchaus auch Bereiche gibt, in denen Rails nicht die erste und beste Lösung ist. Für eine einfache, dynamische Webseite ist Rails sicher nicht geeignet. Und auch Applikationen, die nicht besagter Flexiblität unterliegen müssen, können unter Umständen auf anderem Wege ebenso gut programmiert werden.

Sehr kontrovers wird der auch praktische Umgang mit der Datenbank diskutiert. Zum einen entspricht die Verwendung von künstlichen Primärschlüsseln nicht dem der „reinen Lehre“. Daneben sorgt die Erzeugung der SQL-Befehle zur Laufzeit geringere Gesamtperformance des Frameworks – so die voreilige Meinung. Dem kann entgegengehalten werden, dass Ruby on Rails sehr gut skaliert, also keineswegs ungeeignet für den Einsatz bei umfangreichen Applikationen ist. Dies beweisen die Anwendungen, die sehr erfolgreich von 37signals angeboten werden. Auch andere Projekte haben ähnlich positiv von ihrem Einsatz mit Ruby on Rails berichtet.

---

[66] auch in [SARV]

Daneben wird oft vorgetragen, dass die Anzahl der fähigen Programmierer noch zu gering sei. Als richtiges Argument gegen Rails kann dies jedoch sicher nicht geltend gemacht werden. Vielmehr drückt es das große Interesse aus, Applikationen mit Ruby on Rails zu realisieren. Hier muss die schulische und akademische Ausbildung entsprechend verbessert werden – wie am Ende des Ruby-Kapitels bereits angedeutet.

Rails erfordert, im Vergleich zu PHP, schon eine gewisse Einarbeitungszeit und ist sicher auch nur für erfahrene Web-Programmierer interessant und gewinnbringend. Auf der anderen Seite grenzt sich Ruby on Rails gegenüber den typischen Java-Anwendungen durch seine hohe Flexibilität und kürzere Entwicklungszeit ab.

Das größte Manko aus Unternehmenssicht ist die mäßige Unterstützung vorhandener Infrastruktur und die mangelhafte Integration laufender Applikationen und deren Datenbankbestände. Im Umkehrschluss eignet sich Ruby on Rails besonders gut all die Umgebungen, die von Grund auf neu definiert werden können. Das von den Amerikaner genannte „Greenfield Development“ ist innerhalb von traditionellen Unternehmen sicher schwerer als bei Unternehmensneugründungen („Startups“).

Die Einführung von Rails-Applikationen ins solche traditionellen Umgebungen muss dort eher vereinzelt und auf lange Sicht geplant werden. Hier wird man Ruby on Rails erstmal für explorative Prototypen verwenden wollen.

Für Startups, deren Geschäftsmodell vor allem auf Internetnutzer zielt, ist Ruby on Rails jedoch optimal. Diese profitieren dann wohl auch am meisten von einem agilen Vorgehen in kleinen Teams: Ruby on Rails ermöglicht das zügige Erreichen von brauchbaren Ergebnissen.

Das „Agile Manifesto“[67] und die Ideen von „Extreme Programming“[68] finden sich überall in Rails wieder. Die Prägung des Frameworks spiegelt sich deshalb nicht nur im Framework wieder, sondern auch in der für diese Arbeit verwendeten Literatur.

---

[67] nachzulesen unter [WL21]
[68] weitere Informationen unter [WL22]

5.4 Ausblick

Mittlerweile integrieren sich sich zahlreiche andere Ruby-Projekte nahtlos in Rails: Der Webserver „Mongrel“[69] und das Deployment-Tool „Capistrano“[70] sind im Laufe der Zeit entstanden und stehen ebenfalls unter freien Lizenzen zur Verfügung.

Für die Zukunft bleibt auf jeden Fall die weitere Entwicklung von ActiveResource interessant: Gerade was das Thema REST und serviceorientierte Architekturen angeht, geht Ruby on Rails seinen eigenen Weg.

Ruby on Rails war von Anfang aktiv dabei, moderne AJAX-Technologie zu integrieren. Auch hier entwickelt sich im Moment sehr viel weiter und das Entwicklerteam muss sich anstrengen, um die gute Ausgangsposition des Frameworks nicht zu verlieren.

Auch die Frage, ob sich die in Ruby verankerten Konventionen nicht als Entwicklungsmuster formulieren lassen, erscheint mir eine weitere Untersuchung wert.

Die hohe Wartbarkeit von Rails-Anwendungen wird sich in ein paar (wenigen) Jahren deutlich herauskristallisieren. Im Grunde müsste man, dem agilen Gedanken folgend, die Ergebnisse dieser Arbeit mittelfristig reflektieren und in einer Art Retrospektive aufarbeiten.

69 [WL22]
70 [WL23]

5.5 Persönliches Resümee

Ich arbeite seit knapp zwei Jahren mit Ruby und Ruby on Rails: am Anfang war Rails eher eine nette Spielerei, dann entstanden ein paar private Projekte und mittlerweile ergeben sich auch geschäftliche Anwendungsfälle. Das durchdachte Konzept von Rails hat mich schnell überzeugt und die Argumente für die Verwendung der einen oder anderen (un)konventionellen Methode sind für mich sehr schlüssig. Programmieren im Web macht wieder Spaß, so wie es die Webseite von Ruby on Rails verspricht.

Die Begeisterung über Rubys klarer Syntax und die Mächtigkeit der Sprache hält nach wie vor an und war sicher auch Vorbild und Antrieb für dieser Arbeit.

Ich kann jedem empfehlen, sich Ruby on Rails anzuschauen. Nur eine Warnung sei an dieser Stelle ausgesprochen: Man läuft Gefahr, dass einen die Faszination nicht so so schnell wieder loslässt!¶