Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Ruben Schempp

Zustandsverwaltung in Web-Application-Frameworks mit Continuations

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Ruben Schempp

Zustandsverwaltung in Web-Application-Frameworks mit Continuations

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Michael Böhm
Zweitgutachter: Prof. Dr. rer. nat. Jörg Raasch

Abgegeben am 20. Oktober 2008

Ruben Schempp

Thema der Masterarbeit Zustandsverwaltung in Web-Application-Frameworks mit Continuations

Stichworte Zustand, Continuatons, Steuerung der Kontrolle, Inversion der Kontrolle, Ruby, Ruby on Rails, Seaside, Web-Application-Frameworks

Kurzzusammenfassung Diese Arbeit befasst sich mit der Zustandsverwaltung in Web-Application-Frameworks durch Continuations. Das zustandslose HTTP-Protokoll prägt die Architektur heutiger Web-Application-Frameworks. An Hand der jungen Web-Application-Frameworks Seaside (Smalltalk) und Ruby on Rails wird gezeigt, welche Rolle die automatisierte Zustandsverwaltung für Web-Anwendungen spielt. Am Beispiel von Seaside wird der Nutzen von Continuations für die Web-Entwicklung demonstriert und diskutiert. Im Vergleich zu seitenzentrierten Web-Application-Frameworks wie Ruby on Rails werden Verbesserungen für die Programmierung von Web-Anwendungen vorgestellt. Dies betrifft insbesondere die Steuerung des Kontrollflusses, die Zustandsverwaltung und das Rendern. Neben einer formalen Definition von Continuations wird dargestellt, dass eine Integration von Continuations in Ruby on Rails nicht möglich ist, da deren Implementierung in Ruby fehlerhaft ist.

Ruben Schempp

Title of the paper State Management in Web Application Frameworks with Continuations

Keywords State, Continuations, Control Flow, Inversion of Control, Ruby, Ruby on Rails, Seaside, Web Application Frameworks

Abstract This work deals with state management in Web Application Frameworks by continuations. The stateless nature of the HTTP protocol characterizes the architecture of todays Web Application Frameworks. The impact of automated state management on Web Applications is demonstrated by the emerging Web Application Frameworks Seaside (Smalltalk) and Ruby on Rails. Seaside is taken as an example to show and discuss the advantages of continuations for web development. In contrast to page-centric Web Application Frameworks like Ruby on Rails improvements for programming Web Applications are pointed out. This concerns the contol flow, state management and rendering in particular. Next to a given formal definition of continuations it is argued that an integration of continuations in Ruby on Rails is not possible because of their incorrect implementation in Ruby.

Danksagung

Mein herzlicher Dank gilt meinen Betreuern Herrn Prof. Dr. Michael Böhm und Herrn Prof. Dr. Jörg Raasch, ohne die diese Masterarbeit in dieser Form nicht möglich gewesen wäre. Die Zusammenarbeit mit Ihnen hat mir viel Freude bereitet.

Danken möchte ich auch meiner Freundin Sabine und meiner Familie, die mir mein Studium ermöglicht und mich immer unterstützt hat.

Mit den Worten aus Psalm 17,5 möchte ich meinem Gott, der mir stets treu zur Seite steht, danken:

Erhalte meinen Gang auf deinen Wegen, dass meine Tritte nicht gleiten.

Inhaltsverzeichnis

Abbildungsverzeichnis **VI**

1 Einleitung **1**
1.1 Motivation 1
1.2 Thema 2
1.3 Zielsetzung 4

2 Web-Application-Frameworks **5**
2.1 Begriffsdefinition 5
2.2 Java-Perspektive 6
2.3 Ruby on Rails 10
2.3.1 Ruby 11
2.3.2 Konzepte 14
2.3.3 Aufbau und Komponenten 16
2.3.4 Verarbeitung einer Anfrage 18
2.3.5 Zusammenfassung 20
2.4 Seaside 22
2.4.1 Konzepte und Aufbau 23
2.4.2 Ablaufsteuerung 26
2.4.3 Zustand einer Anwendung 29
2.4.4 Kombination unabhängiger Komponenten 31
2.4.5 Duplikation von Fenstern 33
2.4.6 Zusammenfassung 35

3 Analyse **37**
3.1 Programmierkonzepte 37
3.2 Continuations 38
3.3 Rückumkehrung der Steuerung 42
3.4 Rendern 45
3.5 Zusammenfassung 48

4 Umsetzbarkeit **51**
4.1 Planung 51

4.2 Continuations in Ruby 54
4.2.1 Beispiel in Ruby 58
4.3 Verwandte Arbeiten 59
4.3.1 Acts as continuable Plugin 60
4.4 Auswertung des Ergebnisses 62

5 Zusammenfassung **66**
5.1 Bewertung 66
5.2 Fazit 68
5.3 Ausblick 68

Literaturverzeichnis **70**

Glossar **76**

Abbildungsverzeichnis

1 Ruby on Rails Logo . . . 10
2 Ruby Logo . . . 11
3 Aufbau der Komponenten in Ruby on Rails . . . 16
4 Routing in Ruby on Rails . . . 18
5 Seaside Logo . . . 22
6 Squeak Logo . . . 22
7 Seaside Beispiel: Zähler . . . 25
8 Ablauffolge der Dialoge . . . 28
9 Steuerung der Kontrolle in Seaside . . . 29
10 Web-Server mit Continuations . . . 30
11 Verschiedene Zustände des Zählers . . . 31
12 Geänderter Zuständ des Zählers . . . 31
13 Mehrere Zähler in Seaside . . . 32
14 Einfacher Addierer . . . 33
15 Addierer ohne Zustand . . . 33
16 Addierer mit Zustand . . . 34

17 Ablaufsicht eines Continuation Based Web Server . . . 43
18 Gegenüberstellung . . . 50

19 Aktivitätsdiagramm . . . 52
20 Zähler in Ruby on Rails . . . 64

Verzeichnis der Codebeispiele

1 Lexical Closure in Ruby . . . 12
2 Controller in Ruby on Rails . . . 19
3 View-Template in Ruby on Rails . . . 20
4 Render-Methode in Seaside . . . 24
5 Kontrollfluss in Seaside . . . 27
6 Continuation in Scheme . . . 41
7 View-Template in Ruby on Rails . . . 46
8 View-Methode in Seaside . . . 47
9 Continuation in Ruby . . . 54
10 Wiederholte Ausführung einer Continuation . . . 55
11 Continuations und lokale Variablen . . . 56
12 For-Schleife in Ruby . . . 56
13 Each-Schleife in Ruby . . . 56
14 Continuations und Threads . . . 57
15 Kontrollfluss in Ruby . . . 58
16 Acts as Continuable Plugin . . . 60
17 Zähler mit Continuation . . . 63
18 View-Template des Zählers . . . 63

1 Einleitung

Web-Anwendungen sind im Laufe der Zeit ein wesentlicher Bestandteil des Internets geworden. Immer mehr neue Anwendungen werden als Web-Anwendung konzipiert, sodass sie flexibel und nahezu überall einsetzbar sind.

Die Entwicklung von Web-Anwendungen ist mit Schwierigkeiten verbunden. Im Vergleich zur Programmierung klassischer Desktop-Anwendungen muss die Technik des Internets und der Browser berücksichtigt werden. Das Internet bildet dabei die Kommunikationsplattform der Anwendung, während der Browser als Interaktionskomponente des Benutzers dient.

Diese Arbeit geht auf die Schwierigkeiten der Entwicklung von Web-Anwendungen am Beispiel der Web-Application-Frameworks *Ruby on Rails*[1] und *Seaside*[2] ein. Neben den relevanten Konzepten der beiden Frameworks werden Möglichkeiten zur Verbesserung der Entwicklung – insbesondere durch den Einsatz von *Continuations* – vorgestellt. Continuations sind ein mächtiges Programmierwerkzeug, das den Zustand einer Programmausführung speichert. Sie kommen seit einiger Zeit in Seaside zum Einsatz. Es wird untersucht und diskutiert, inwiefern Continuations auch in Ruby on Rails sinnvoll eingesetzt werden können. Nach [Tate, 2005a] handelt sich bei Continuations-basierten Web-Anwendungen um so genannte *Continuation Servers* bzw. *Continuation Based Web Server* .

Eine Präzisierung des Themas und der Schwerpunkte dieser Arbeit folgt in Kapitel 1.2.

1.1 Motivation

Im Rahmen der Bachelorarbeit [Schempp, 2006, Aktuelle Entwicklungstrends im Bereich von Programmiersprachen] an der Hochschule für Angewandte Wissenschaften Hamburg wurde die Programmiersprache Ruby[3] vorgestellt. Das auf ihr basierende Framework Ruby on Rails sowie Seaside wurden dort bereits erwähnt.

[1] `http://www.rubyonrails.org/`
[2] `http://www.seaside.st/`
[3] `http://www.ruby-lang.org/de/`

Der anfängliche (Medien-)Rummel (engl. Hype) um Ruby on Rails hat der Beliebtheit des Frameworks nicht geschadet. Es hat sich inzwischen in einer Marktnische etabliert und bewährt. Das Framework Ruby on Rails wirkte dabei als Killer-Applikation[4] der Sprache Ruby. Es besitzt von Haus aus keine integrierte Zustandsverwaltung.

Seaside ist ebenfalls ein junges Web-Application-Framework, das auf der Programmiersprache Smalltalk basiert. Seine Stärken liegen in der einfachen Programmierung des Kontrollflusses einer Web-Anwendung und der integrierten, automatisierten Verwaltung des Zustands.

Stellt man Ruby on Rails und Seaside gegenüber, so vertritt ersteres Framework den klassischen Ansatz der Web-Programmierung, während Seaside einen innovativeren Ansatz mitbringt. Beide Frameworks setzen mit ihren eigenen, teils sehr unterschiedlichen, Herangehensweisen neue Akzente in der produktiven Web-Entwicklung.

Laut [Queinnec, 2003] ist die Web-Entwicklung mit Continuations dem klassichen Ansatz der Programmierung von Web-Anwendungen (ohne Continuations) überlegen. Diese These wird in dieser Arbeit an Hand von Seaside, das Continuations verwendet, und Ruby on Rails, das keine Continuations einsetzt, diskutiert.

Die vielfach vorhandene Unsicherheit mit dem Umgang bzw. Einsatz von Continuations soll durch eine verständliche Erklärung ihrer Programmiertechnik ausgeräumt werden. Am Beispiel von Seaside wird gezeigt, welches Potential Continuations für die Web-Entwicklung mit sich bringen. Darüber hinaus wird geprüft, inwiefern auch Ruby on Rails von Continuations profitieren kann.

Zu einigen in dieser Arbeit vorgestellten bzw. diskutierten Aspekten gibt es in der Literatur bisher keine Antworten.

1.2 Thema

Kern dieser Arbeit ist die Verwendung von Continuations in Web-Application-Frameworks zur Verbesserung der Entwicklung und Benutzung einer Web-Anwendung. Dieses Unterkapitel führt in die Thematik ein.

Das HTTP-Protokoll stellt die Grundlage für die Kommunikation von Internet-Anwendungen zwischen Web-Server und Web-Browser dar. Die Kommunikation über HTTP ist asymmetrisch[5] und zustandslos. Das Fehlen eines Zustands in der Kommunikation war für statische

[4] Eine Killer-Applikation bezeichnet eine Anwendung, die als Katalysator eine Basistechnik – hier eine Programmiersprache – bekannt macht.

[5] HTTP ist asymmetrisch, da nur der Client Anfragen zum Server schicken kann. Der Server kann hingegen keine Nachricht (ohne eine vorhergegangene Client-Anfrage) an den Client schicken.

Webseiten gewünscht, stellt aber für die Programmierung heutiger Internet-Anwendungen im Zeitalter des Web 2.0[6] – ebenso wie die Asymmetrie – ein Hindernis dar. Web-Anwendungen benötigen einen Zustand, um eine komfortable Programmierung und Bedienung der Anwendung zu ermöglichen. Zu HTTP gibt es keine Alternative. Deshalb muss der Zustand auf eine andere Weise (aufwändig) modelliert werden. [Ducasse u. a., 2004] schreiben über diese Problematik:

„Developers have to work manually around the shortcomings of the HTTP protocol.“

Die *Umkehrung der Steuerung* (engl. *Inversion of Control*) ist bei der Entwicklung von Web-Anwendungen allgegenwärtig. Sie führt häufig zu einem unübersichtlichen Programmcode, der über viele Stellen verteilt ist, obwohl er eine logische Einheit bildet. Mit dem Einsatz von Continuations kann die Umkehrung der Steuerung wieder rückgängig gemacht werden. Dies ermöglicht eine Programmierung von Web-Anwendungen im Stil von Desktop-Anwendungen. Der Programmablauf kann – abstrakt gesehen – linear beschrieben bzw. direkt programmiert werden.

Diese Arbeit betrachtet die Auswirkungen von Continuations auf die Entwicklung von Web-Anwendungen am Beispiel des Web-Application-Frameworks Seaside. Ihm liegen einige interessante Designentscheidungen zu Grunde, die sich auf das Programmierkonzept der Anwendung auswirken. An dem Web-Application-Framework Ruby on Rails wird gezeigt – sofern dies mit einem verhältnismäßigen Aufwand möglich ist – wie sich die Programmierung von Web-Anwendungen verbessern lässt. Es wird dabei auf folgende Bereiche eingegangen:

- Programmierung bzw. Ablaufsteuerung in Ruby on Rails
- Besonderheiten der Programmierung in Seaside
- Konzeptuelle Unterschiede der Entwicklung in beiden Frameworks
  - Einsatz von Vorlagen (engl. Templates) für Webseiten mit dynamischem Inhalt
  - Trennung nach dem Model-View-Controller-Entwurfsmuster (MVC)
  - Steuerung der Anwendungskontrolle
  - Automatisierte Generierung von URLs
- Zustandsverwaltung durch Continuations
- Ablaufsteuerung und -programmierung an Hand von Beispielen
- Bedienung der Anwendung im Browser
- Continuations und deren Implementierung in Ruby

[6] Tim O’Reilly beschreibt das Web 2.0 in seinem Artikel „What Is Web 2.0“. [O’Reilly, 2005]

Abgrenzung

Diese Arbeit beschäftigt sich fast ausschließlich mit Ruby und Smalltalk und nur in sehr geringem Umfang mit dem Marktführer Java. Dies liegt daran, dass Java die technischen Voraussetzungen für die Thematik dieser Arbeit nicht oder nur ungenügend unterstützt. Java unterstützt weder *Lexical Closures* noch Continuations, sodass sich die Sprache nicht als geeignete Plattform für Continuations-basierte Frameworks anbietet.[7] Dennoch gibt es beispielsweise mit dem RIFE-Framework[8] einen Vertreter von zustandsverwaltenden Frameworks in Java, das auf einem Zustandsautomaten basiert.

Einige weitere Frameworks für die Zustandsverwaltung werden in [Tate, 2006a], [Ducasse u. a., 2004], [Ducasse u. a., 2007] und [Schmalhofer, 2007] erwähnt.

1.3 Zielsetzung

Die Arbeit führt Vor- und Nachteile der Verwendung von Continuations auf und bewertet vorhandene Risiken. In Hinblick auf Continuation Based Web Server soll die Vereinbarkeit der Konzepte von objektorientierten Web-Application-Frameworks und Continuations untersucht werden. Nachfolgend wird die Machbarkeit der Verwendung von Continuations in Ruby on Rails untersucht.

Überblick

Kapitel 2.1 beschreibt den Begriff „Web-Application-Framework“. Kapitel 2.2 führt aus der Java-Perspektive an die Thematik heran. Die Kapitel 2.3 und 2.4 stellen die Web-Application-Frameworks Ruby on Rails und Seaside vor. Diese Vorstellung beschränkt sich auf die für das Thema relevanten Aspekte. In Kapitel 3 folgt eine Analyse der beiden Web-Application-Frameworks und ihrer Programmierkonzepte. Dabei wird insbesondere auf verschiedene Aspekte der Continuations und des *Renderns* eingegangen. Diese Analyse ist notwendig, um die Grundlagen für das darauf folgende Kapitel zu schaffen. In Kapitel 4 wird die Integration von Continuations in Ruby on Rails untersucht. Kapitel 5 beinhaltet zum Abschluss der Arbeit die Zusammenfassung und das Fazit.

[7] Siehe dazu auch [Tate, 2005a] und [Schempp, 2006].
[8] `http://rifers.org/`

2 Web-Application-Frameworks

2.1 Begriffsdefinition

Ein Web-Application-Framework bildet als Rahmenwerk eine Grundlage für die Entwicklung von Software für das Internet. Es stellt vorgefertigten Programmcode für allgemeine Aufgaben zur Verfügung und lässt sich in einem bestimmten Rahmen nach den Wünschen des Entwicklers konfigurieren. Ein typisches Beispiel für ein Framework ist Active Record, das Bestandteil des Web-Application-Frameworks Ruby on Rails ist. Seine Funktion ist die Abbildung von Objekten auf die Datenbank (in beiden Richtungen).

Ein Framework ähnelt in seiner Funktionsweise einer Programmierbibliothek, unterscheidet sich aber beim Aufruf des Programmcodes wesentlich: Bei einer Bibliothek ruft der Programmierer eine Funktion auf, die beispielsweise alle Leerzeichen aus einer Zeichenkette entfernt. Ein Framework folgt oftmals dem Hollywood-Prinzip („Don't call us, we'll call you"), indem es den selbst geschriebenen Code aufruft.[9] Als Beispiel sei hier die Ausführung einer konkreten Aktion aufgeführt, die der Entwickler selbst programmiert und die (nur) vom Framework aufgerufen wird. Näheres zur dieser Thematik findet sich in Kapitel 3.3 wieder.

Web-Application-Frameworks sind aus verschiedenen Bestandteilen zusammengesetzt, deren unterschiedliche Funktionalität Bereiche der Web-Entwicklung unterstützt. Eine typische Auswahl solcher Themengebiete ist nachfolgend aufgelistet (vgl. [Wikipedia, 2008h]).

Sicherheit z.B. Benutzer-Identifikation und Absicherung gegen Missbrauch.

Datenbankanbindung Abbildung einer objekt-relationalen Datenbank durch Objekte.

URL-Abbildung Zuordnung von URLs zu den Aktionen der Anwendungs-Controller.

Templates Erzeugen dynamischer Webseiten auf der Basis von (HTML-)Vorlagen.

Caching Zwischenspeichern von bereits berechneten Ergebnissen.

AJAX-Integration [10] Problemlose Einbindung von JavaScript und JavaScript-Bibliotheken.

[9] Siehe Artikel [Fowler, 2005, Inversion of Control].

[10] AJAX: Asynchronous JavaScript and XML

2.2 Java-Perspektive

Java ist als Programmiersprache und Plattform für Web-Anwendungen weit verbreitet und behauptet sich seit einigen Jahren als Marktführer. Da der Fokus dieser Arbeit nicht auf Java und dessen Web-Application-Frameworks, sondern auf Ruby on Rails und Seaside liegt, führt dieses Kapitel aus der Java-Perspektive an die Thematik dieser Arbeit heran.

Mit der Zeit haben sich einige Lösungswege zur Erstellung von Web-Anwendungen im Java-Umfeld durchgesetzt. Eine mit Ruby on Rails erstellte Web-Anwendung besitzt eine ganz ähnliche Architektur. Die Philosophie von Ruby on Rails und die Dynamik von Ruby erlauben jedoch den Verzicht auf eine umfassende Konfiguration. Ruby on Rails stellt darüber hinaus eine Sammlung aufeinander abgestimmter Frameworks dar. Durch die Integration der Frameworks werden Informationen nur an einer einzigen Stelle der Programmkonfiguration aufgeführt, sodass Redundanzen vermieden werden. Mit Java erstellte Web-Anwendungen basieren i.d.R. auf verschiedenen und unabhängigen, nicht aufeinander abgestimmten, Frameworks, die unterschiedliche Funktionen erfüllen. Dieser Umstand erfordert einen erhöhten Konfigurationsaufwand, um die verschiedenen Frameworks zur Zusammenarbeit zu bewegen. Dabei kommt es auch vor, dass dieselben Informationen an verschiedenen Stellen auftauchen und somit Redundanzen entstehen. Das folgende Zitat verdeutlicht dies:

„While Ruby on Rails is a very new and exciting framework that has generated considerable interest in the Web development community, the core architecture follows the basic patterns found in J2EE. It's the philosophical approach to development of Web applications that sets the two frameworks apart. Rails prefers explicit code instead of configuration files, and the dynamic nature of the Ruby language generates much of the plumbing code at runtime. Most of the Rails framework has been created as a single project and application development benefits from a set of homogeneous components. In contrast, the typical J2EE stack tends to be built from best-of-breed components that are usually developed independently of one another, and XML is often used for configuration and gluing the components together." – [Rustad, 2005]

Auf die Thematik dieser Arbeit – im Speziellen auf die Implementierung einer verbesserten Anwendungssteuerung im Web-Browser – bezogen, beschreibt A. Belapurkar (siehe folgendes Zitat) die Implementierung der Navigation einer Anwendung als zusätzliche Erhöhung einer ohnehin schon hohen Komplexität des Quellcodes:

„Web development frameworks, such as Spring and Struts, allow you to handle multiple navigational paths, but they do so at the cost of increasing the complexity of an already overly complex code base." – [Belapurkar, 2004]

Defizite

„Simple things should be simple, complex things should be possible." – Alan Kay [Wikiqoute, 2008a]

Das obige Zitat von A. Kay folgt dem „Keep it Simple"-Prinzip und beschreibt einen Grundsatz, an dem sich auch Programmiersprachen messen lassen müssen. Bezogen auf Java kann festgehalten werden, dass sowohl einfache als auch komplexe Dinge umsetzbar sind, aber – besonders bei Web-Anwendungen – einfache Dinge oftmals einen relativ hohen Aufwand erfordern. Ferner ist Javas Sprachdesign nicht konsequent umgesetzt und weist einige mangelhaft umgesetzte Konzepte auf.[11]

„Java was, as Gosling says in the first Java white paper, designed for average programmers. It's a perfectly legitimate goal to design a language for average programmers. (Or for that matter for small children, like Logo.) But it is also a legitimate, and very different, goal to design a language for good programmers." – Paul Graham [Wikiqoute, 2008b]

Die Sprache Java weist einige Unzulänglichkeiten auf, die besonders im Vergleich mit Sprachen wie Ruby oder Smalltalk ins Auge fallen. Dazu gehört z.B. die *Metaprogrammierung*, die in Java nur unzureichend berücksichtigt wurde, und die geringere Ausdrucksfähigkeit. Javas Konzepte schränken einen fortgeschrittenen Programmierer in seinen Möglichkeiten ein, für einen durchschnittlichen Programmierer mögen diese Konzepte hingegen durchaus angebracht sein (siehe obiges Zitat von P. Graham). Im Folgenden werden einige der in Java unzureichend bzw. nicht umgesetzten Konzepte aufgelistet. Auf eine genauere Erläuterung dieser Punkte wird im Rahmen dieser Arbeit verzichtet. Es wird hier auf [Tate, 2005a], [Tate, 2006c] und [Schempp, 2006] verwiesen.

- Metaprogrammierung und Reflection sind nur schwer zugänglich.
- Lexical Closures sind nicht implementiert.
- Java ist eine hybride Sprache, da sie (dem Konzept der Objektorientierung entgegen) auch primitive Typen enthält.
- Das Methodenprotokoll von Javas Kern-Bibliotheken ist nicht orthogonal aufgebaut.
- Der *Feedback-Zyklus* zwischen Programmierung und Begutachtung des Ergebnisses bzw. der Auswirkungen ist in Java recht lang. Dies ist u.a. durch die Übersetzung des Codes bedingt.
- Die statische Typisierung und die damit verbundene Deklaration der Variablentypen steht dem Konzept der Objektorientierung entgegen.

[11] Siehe dazu [Tate, 2005a] und auch [Schempp, 2006].

- Die Deklaration von Variablen und Parametern und die Fehlerbehandlung erfordern einen erhöhten Zeitaufwand.
- Java-Code hat oft eine recht hohe Komplexität. Dies betrifft nach [Tate, 2006b] auch Web-Seiten (bzw. Templates), die mit JavaServer Pages (JSP) erstellt werden.
- Viele Java-Frameworks erfordern einen recht hohen Lernaufwand.

Verglichen mit den in dieser Arbeit behandelten Web-Application-Frameworks benötigen Java-basierte Web-Anwendungen i.d.R. einen erheblich größeren Code-Umfang. Daraus ergeben sich im Folgenden ein erhöhter Wartungsaufwand und somit höhere Kosten. B.Tate weist in [Tate, 2005a] darauf hin, dass der Lernaufwand für die Erstellung von Web-Anwendungen sehr hoch ist und den anderer Sprachen deutlich übersteigt. Dies hängt nicht zuletzt damit zusammen, dass die populären Java-Frameworks neben einem recht umfangreichen Funktionsumfang auch einen relativ hohen Konfigurationsaufwand mit sich bringen.

Verbesserungen

„The metaprogramming capabilities of Ruby lie so close to the surface and are quite accessible to the average Ruby programmer.“ – Jim Weirich in [Tate, 2005a]

Betrachtet man Ruby on Rails im Vergleich zu den in der Java-Welt vorherrschenden Techniken und Frameworks, so fällt auf, dass das Framework aus einem Guss konstruiert ist. Ruby on Rails erfindet das Rad nicht neu, sondern greift auf bereits bekannte Techniken und Lösungswege zurück, setzt diese aber durchgängig und in objektorientierter Weise um. Das Framework kann dabei insbesondere von der dynamischen Programmiersprache Ruby profitieren, deren Stärken nutzen und dabei Dinge umsetzen, die so in Java nicht möglich sind. Verglichen mit der innovativen Umsetzung von Seaside wirken Lösungen aus dem Java-Umfeld dagegen ein wenig veraltet.

Statistische Vergleiche belegen seit einigen Jahren, dass es beispielsweise mit Perl, Ruby und Smalltalk weitaus ausdruckskräftigere und produktivere Programmiersprachen als Java gibt. Die Frameworks Seaside und Ruby on Rails steigern darüber hinaus die Produktivität der Web-Entwicklung in durch Java unerreichbare Bereiche. B.Tate ergreift in [Tate, 2006c, From Java to Ruby] die Initiative und veranschaulicht die daraus resultierenden Folgen für die Anwendungsentwicklung aus der Sicht eines Managers:

„In short, the Java language just isn't a very productive applications language.“ – [Tate, 2005b]

Der Einsatz der in dieser Arbeit diskutierten Frameworks und Programmiersprachen wie Ruby und Smalltalk verspricht im Vergleich zu Java eine vereinfachte Entwicklung von Web-Anwendungen. Dies ist durch die flexibleren Eigenschaften der Sprache und den besseren Sprachmitteln (z.B. Metaprogrammierung, siehe Zitat von J.Weirich) begründet. Neben der Steigerung der Produktivität bringt Seaside vor allem einige gelungene und neue Ideen bzw. Konzepte mit. Prototyping und Testen werden ebenfalls unterstützt. Durch den vergleichsweise geringen Umfang an Quellcode werden die Entwicklung sowie die darauf folgende Wartung von Software-Projekten erleichtert und somit die Kosten gesenkt.

2.3 Ruby on Rails

Abbildung 1: Bahnschienen symbolisieren die Web-Entwicklung mit Ruby on Rails. [Wikipedia, 2008e]

Ruby on Rails (kurz: Rails) ist ein Open-Source-Framework für die Entwicklung von Web-Anwendungen, das auf der Programmiersprache Ruby basiert. Es wurde von David Heinemeier Hansson im Jahr 2004 als erste Beta-Version veröffentlicht und ist inzwischen bei der Version 2.1 angekommen. Das Rails Framework wurde bereits in [Schempp, 2006] vorgestellt, sodass auf eine erneute Vorstellung verzichtet und nur kurz in das Framework eingeführt wird. Die thematisch relevanten Besonderheiten der Ablaufsteuerung und des Aufbaus von Ruby on Rails werden hingegen genauer vorgestellt.

Gegenwärtiger Stand

In Anknüpfung an die in [Tate, 2005a] und [Schempp, 2006] erfolgte Beurteilung des Rails Frameworks lässt sich festhalten, dass es mittlerweile mehr als ein Dutzend Bücher über Ruby on Rails gibt.[12] Die Arbeit mit Ruby on Rails ist seit der Version 2.0 an einigen Stellen deutlich vereinfacht worden. Die Beliebtheit des Frameworks ist ungebrochen. Durch die steigende Verbreitung konnte sich eine aktive Entwicklergemeinde bilden, die in ihren Ansichten weniger voreingenommen ist als andere. Ferner sind beispielsweise mit *Aptana Studio*[13] und *IntelliJIDEA*[14] die ersten Entwicklungsumgebungen für Ruby on Rails verfügbar. Dennoch ist der nur für MacOS erhältliche Texteditor *TextMate*[15] bei den Rails-Entwicklern sehr beliebt.

[12] Umfassende Darstellungen von Ruby on Rails finden sich in [Thomas und Hannson, 2006], [Fernandez, 2008], [Morsy und Otto, 2008] und [Wirdemann und Baustert, 2007a].
[13] Aptana Studio ist ein Abkömmling der Eclipse-Plattfrom. Siehe `http://www.aptana.com/`.
[14] `http://www.jetbrains.com/idea/`
[15] `http://macromates.com/`

Das Angebot an Erweiterungen für Rails in Form von Plugins ist mittlerweile so reichhaltig, dass es unübersichtlich zu werden droht. Die Qualität der Plugins unterscheidet sich z.T. sehr. Einige wenige dieser Plugins werden bzw. wurden als feste Bestandteile in Rails integriert. Voraussetzung dafür ist jedoch, dass diese ausgereift sind und ihre Funktionalität tatsächlich von vielen Entwicklern benötigt wird.

2.3.1 Ruby

Abbildung 2: Der Rubin als Symbol für die Programmiersprache Ruby. [Wikipedia, 2008f]

Dieses Unterkapitel widmet sich Ruby, der Programmiersprache, die hinter Ruby on Rails steht.[16] Ruby ist eine objektorientierte und freie Programmiersprache. Sie wurde 1995 von dem Japaner Yukihiro Matsumoto veröffentlicht. Ruby ist dynamisch typisiert, wird interpretiert und bringt den interaktiven Interpreter *Interactive Ruby* (kurz: *irb*) mit.

Aufgrund der Vorstellung in [Schempp, 2006] wird hier auf eine detailliertere Vorstellung der Programmiersprache Ruby verzichtet. Eine ausführliche Dokumentation der Sprache findet sich in [Thomas u. a., 2005].

Ruby liegt aktuell in der Version 1.8.7 vor, die Ende Mai 2008 veröffentlicht wurde. Den Codebeispielen dieser Arbeit liegt die Version 1.8.6 zu Grunde, die sich unwesentlich von der aktuellen Version unterscheidet. Seit längerer Zeit befindet sich Ruby 2.0 in der Entwicklung. Es ist jedoch unklar, welche Funktionsmerkmale letztendlich in die Version 2.0 übernommen werden. Eine Ausschau auf mögliche Veränderungen in Ruby 2.0 bietet die Version 1.9, die jedoch nicht für den produktiven Einsatz empfohlen wird.

Kritik

Ruby folgt dem „Prinzip der geringsten Überraschung“[17], welches besagt, dass ein Programmierbefehl auch das ausführen soll, was der Benutzer – durch seine Namensgebung – von

[16] Der Rubin (siehe Abbildung 2) wurde von Rubys Entwickler Yukihiro Matsumoto in Anlehnung an die Programmiersprache Perl (dt. Perle) gewählt.

[17] Engl.: „Principle of least surprise“. Siehe dazu auch `http://c2.com/cgi/wiki?PrincipleOfLeastAstonishment`.

ihm erwartet. Diese Erwartung betrifft insbesondere die ausgeführte Aktion, die Art und Weise der Ausführung sowie den Rückgabewert, die sich alle an die Gepflogenheiten der Sprache halten sollten. In Ruby wird dieses Prinzip z.B. durch den Namensraum der freien Variablen in Lexical Closures verletzt (siehe Codebeispiel 1). Die in Zeile 2 definierte Variable *i* wird von der in Zeile 4 lokal definierten Variable *i* überschrieben. Dementsprechend hat *i* in Zeile 7 nicht den Wert *Zeichenkette*, sondern – entgegen der Erwartung – den Wert *3*.

```
1 begin
2   i = 'Zeichenkette'
3   puts i
4   [1, 2, 3].each do |i|
5     puts i
6   end
7   puts i
8 end
```

Codebeispiel 1: Lexical Closures verletzen in Ruby das Prinzip der geringsten Überraschung.

Solche und andere (i.d.R. kleine) Ungenauigkeiten oder Fehler in der Modellierung treten in Ruby relativ selten auf. In Ruby 1.9 ist das aufgeführte Fehlverhalten korrigiert worden, sodass es in zukünftigen Versionen wohl korrekt umgesetzt werden wird. Dave Thomas beschreibt dieses unerwartete Verhalten in seinem Buch [Thomas u. a., 2005, Programming Ruby] im Kapitel „Blöcke und Iteratoren“.

Einige weitere Kritikpunkte an der Implementierung und dem Sprachdesign von Ruby sind nachfolgend aufgelistet:

Internationalisierung Die Unterstützung verschiedener Sprachen, die auf unterschiedlichen Zeichensätzen beruhen, ist in Ruby mäßig umgesetzt worden. Die dafür empfehlenswerte UTF-8-Codierung (8-bit Unicode Transformation Format) des Zeichensatzes ist in Ruby nur mit zusätzlichen Bibliotheken verwendbar. Dies hängt mit der vergleichsweise geringen Verbreitung des (Mitte der neunziger Jahre standardisierten) Unicodes im ost-asiatischen Raum zusammen. In [Ediger, 2007] wird diese Thematik ausführlich behandelt.

Sprachspezifikation Ruby besitzt keine Sprachspezifikation, wie sie normalerweise für Programmiersprachen üblich ist. Die einzig verfügbare „Referenz“ ist der dem Ruby-Interpreter zu Grunde liegende C-Code.[18] An Stelle einer Sprachdefinition in C wäre eine Implementierung von Ruby in sich selbst nützlich. Der daraus entstehende kompakte Interpreter könnte als Definition der Semantik herangezogen werden. Da eine

[18] Die Sprachdefinition des Interpreters wird auch als *MRI (Matsumoto's Ruby Interpreter)* bezeichnet.

solche Spezifikation fehlt, fallen die tatsächlichen Implementierungen der Virtuellen Maschinen verschieden aus.

Geschwindigkeit Die Ausführungsgeschwindigkeit des Ruby-Interpreters ist, verglichen mit anderen Programmiersprachen, (sehr) langsam. Zudem benötigt Ruby mehr Ressourcen als andere Sprachen. Diese Problematik liegt an der Implementierung des Ruby-Interpreters und kann durch die in der Entwicklung befindlichen Virtuellen Maschinen verbessert werden.

Virtuelle Maschinen Seit einiger Zeit wird an mehreren Virtuellen Maschinen für Ruby gearbeitet. Primäres Ziel der meisten Projekte ist, die Lauffähigkeit von Ruby on Rails auf der jeweiligen Virtuellen Maschine umzusetzen. Y. Matsumoto arbeitet bei dem Design der Ruby-Version 2.0 eng mit dem Entwickler der Virtuellen Maschine *YARV* zusammen, die die offizielle Virtuelle Maschine von Ruby werden soll. Die Ruby-Version 1.9 gilt als Testplattform für die YARV-Kompatibilität. Neben YARV und *JRuby*, das Ruby auf der *Java Virtual Machine* ausführt, gibt es weitere Entwicklungsprojekte für Virtuelle Maschinen. Unter diesen stellt *MagLev* einen besonders interessanten Ansatz dar. [19]

Threads Ruby verwendet an Stelle nativer Threads so genannte *Green Threads*. Während native Threads direkt auf Threads des Betriebssystems abgebildet werden, werden Green Threads innerhalb des Ruby-Prozesses ausgeführt und verwaltet. Dies hat den Vorteil, dass Ruby Multithreading auf Betriebssystemen unterstützt, die dies von Haus aus nicht anbieten. Nachteilig wirken sich Green Threads jedoch auf heutige Mehrkernprozessoren aus, die das Ruby-Programm nur auf einem Prozessorkern ausführen können, da die Threads nicht auf das Betriebssystem abgebildet werden. Ferner übernimmt Ruby durch das intern umgesetzte Thread-Modell auch die Verwaltung bzw. das Scheduling der Threads. Dies ist problematisch, wenn ein Thread blockierend auf Eingaben bzw. Ausgaben (engl. Input/Output) wartet und somit den gesamten Ruby-Prozess blockiert. In dieser Situation kann kein (anderer) Thread weiterlaufen, da auch Rubys Threadverwaltung so lange blockiert wird, bis das blockierende Warten dieses Threads beendet wurde.

Einige dieser Punkte, z.B. die Unterstützung von Unicode und die Verwendung von nativen statt Green Threads, sollen in Ruby 2.0 unterstützt werden.

[19] *MagLev* wird von Avi Bryant und der Firma GemStone entwickelt. Es stellt eine Art verteilte Virtuelle Maschine für Ruby (on Rails) dar, die in GemStone Smalltalk implementiert wird. Dabei verwaltet Gemstone die Ruby-Objekte und ermöglicht mehreren Ruby-Prozessen den (verteilten) Zugriff auf dieselben Objekte. *MagLev* befindet sich den verfügbaren Informationen nach noch in der Entwicklungsphase. Zukünftig könnte diese Virtuelle Maschine auf Grund ihres Konzeptes und der durch die Smalltalk-Implementierung zu erwartenden höheren Geschwindigkeit das Interesse vieler Entwickler wecken.

Fazit

Ruby erfreut sich, nicht zuletzt wegen des Rails Frameworks, einer wachsenden Beliebtheit. Die Sprache setzt einige Paradigmen, so z.B. die Objektorientierung, gut um. Andere, kritikwürdige Bereiche werden in Zukunft Verbesserungen erfahren.

In [Sheehan, 2007] wird beschrieben, dass Ruby eine geeignete Sprache für den (Hochschul-)Unterricht im Fachgebiet der Betriebssysteme ist. An der Hochschule für Angewandte Wissenschaften Hamburg wird Ruby seit dem Wintersemester 2007/2008 als erste Programmiersprache unterrichtet.

Ein interessantes Themengebiet ist die voranschreitende Entwicklung der verschiedenen Interpreter für Virtuelle Maschinen, von denen inzwischen mindestens eine Hand voll existieren. Mit YARV soll Ruby offiziell eine Virtuelle Maschine erhalten und mit MagLev ist eine innovative Virtuelle Maschine mit einer zu erwartenden Geschwindigkeitssteigerung angekündigt.

Abschließend betrachtet ist Ruby eine Sprache mit wachsendem Zuspruch, die eine größere Mächtigkeit[20] und Produktivität als etwa Java besitzt (siehe [Schempp, 2006]). In Ruby ist Metaprogrammierung leicht umzusetzen, da dieses Sprachkonzept im Sprachdesign von Grund auf berücksichtigt wurde. Durch den kurzen Feedback-Zyklus wird die schnelle und produktive Programmierung von Anwendungen in Ruby gefördert.

2.3.2 Konzepte

Die Stärke von Ruby on Rails liegt in der recht einfachen und produktiven Entwicklung von Web-Anwendungen, die auf einer relationalen Datenbank basieren. Rails zeigt einen möglichen Weg für die Anwendungsentwicklung auf, der deshalb so gelungen ist, weil alle Komponenten aus einer Hand stammen und auf ein reibungsloses Zusammenspiel ausgelegt sind. Das Rails Framework lässt sich durch einige zu Grunde liegende Design-Prinzipien charakterisieren:

Convention over Configuration Konvention statt Konfiguration besagt, dass Rails von bestimmten Annahmen über die Eigenschaften eines Projekts ausgeht. Als Beispiel seien die Namenskonventionen von Active Record bei der Abbildung von Modellklassen

[20] Grundsätzlich sind alle „vollwertigen“ Programmiersprachen Turing-Äquivalent und somit in der Lage die gleichen Funktionen zu berechnen. Dies kann durch die Konstruktion einer Turing-Maschine belegt werden. Dennoch unterscheidet sich die Mächtigkeit der Sprachen in der Qualität bzw. Ausdrucksstärke ihrer Befehle. Ein Ruby-Befehl hat z.B. einen viel größeren Umfang als ein Assembler-Befehl. Vgl. [Thielecke, 1999], Kapitel 5 und [Graham, 2004], Kapitel *Appendix: Power* (S.195 ff.).

(Klassenname im Singular) auf Datenbanktabellen (Tabellenname im Plural) aufgeführt. Diese Annahmen erleichtern die Arbeit mit dem Framework ungemein, sofern man sie möglichst unverändert übernimmt. Änderungen der meisten Konventionen sind i.d.R. ohne größeren Aufwand umsetzbar.

Don't Repeat Yourself „Wiederhole dich nicht“ ist ein Leitsatz der Pragmatischen Programmierer [Hunt und Thomas, 2000].[21] Kern dieser Aussage ist es, seinen Code nur einmal – rendundanzfrei – zu schreiben. So wird vermieden an verschiedenen Stellen „das Gleiche“ auszudrücken. Dieser Ansatz wird durch das Rails Framework unterstützt, indem Metaprogrammierung eingesetzt wird. Rails erkennt beispielsweise an Hand der Tabellenstruktur der Datenbank, welche Attribute es für die Modelle zur Verfügung stellen muss. Die Attribute werden ausschließlich in der Datenbank, nicht aber im Quellcode definiert, um Redundanzen zu vermeiden. Ein weiterer Nutzen von redundanzfreiem Code ist die Vermeidung von Inkonsistenzen innerhalb des Systems, da eine „Sache“ nur an einer einzigen Stelle beschrieben wird.

MVC Pattern Das Model-View-Controller-Architekturmuster (siehe [Fowler, 2003] und [Gamma u. a., 1995]) gibt eine Aufteilung der Anwendung in die Komponenten Modellklassen (Model), Ausgabe bzw. Präsentation (View) und Steuerung (Controller) vor. Die Komponenten werden voneinander entkoppelt. Rails basiert auf diesem Architekturmuster. Es besteht aus mehreren Komponenten und gibt dem Entwickler eine nach diesem Muster angelegte Verzeichnisstruktur vor, die die Trennung unterstützt. Mehr dazu in Kapitel 2.3.3.

Templates Rails verwendet Vorlagen für die Präsentationsschicht, die zur Laufzeit durch die Daten des Modells ergänzt werden. Es kommen i.d.R. Vorlagen für HTML und JavaScript zum Einsatz.

Generatoren Rails bringt einige Generatoren mit, die (vor allem anfangs) die Arbeit mit dem Quellcode erleichtern. Sie generieren etwa Grundgerüste von Modellen und den zugehörigen Dateien und ersparen vor allem Schreibarbeit. Die so genannten *Scaffold*-Generatoren erzeugen ein grundlegendes funktionsfähiges Modell, das später nach Belieben erweitert bzw. ersetzt werden kann. Dies ist ein Punkt, durch den deutlich wird, wie Rails die *Agile Softwareentwicklung* unterstützt.

Testen *Test Driven Development (TDD)* ist standardmäßig in Rails enthalten und gut integriert. Bei der Erstellung eines Modells mit einem Generator wird automatisch ein entsprechender Unit Test angelegt. Darüber hinaus gibt es zur Verbesserung des Testens mehrere Plugins für Rails. So wird etwa das *Behaviour Driven Development (BDD)* durch so genannte *RSpecs* unterstützt. Rails folgt damit den Empfehlungen des *Extreme Programming* (siehe [Beck und Andres, 2005]).

[21] Dieser Leitsatz ist auch unter dem Titel „once and only once“ bekannt.

Migrations Der Umgang mit der Datenbank wird durch Datenbankmigrationen unterstützt. Jede Migration beschreibt dabei (inkrementell) eine Version des Datenbankschemas. Die Migration enthält die Beschreibung der Tabellen für ein oder mehrere Modelle. Rails generiert daraus die *Data Definition Language (SQL)*, um die Datenbank zu erstellen bzw. erweitern. Migrations können dabei sowohl vorwärts als auch rückwärts ausgeführt werden. Zudem können die in der Datenbank befindlichen Daten durch Migrations konvertiert werden.

2.3.3 Aufbau und Komponenten

In diesem Unterkapitel werden die einzelnen Frameworks, aus denen Ruby on Rails besteht, kurz vorgestellt.

Abbildung 3 stellt den Aufbau und das Zusammenspiel der Komponenten bzw. Sub-Frameworks dar, die das Grundgerüst von Ruby on Rails bilden. Im Folgenden werden auch Komponenten aufgeführt, die über den in der Abbildung dargestellten Rahmen hinaus gehen, aber einen wesentlichen Teil von Rails ausmachen.

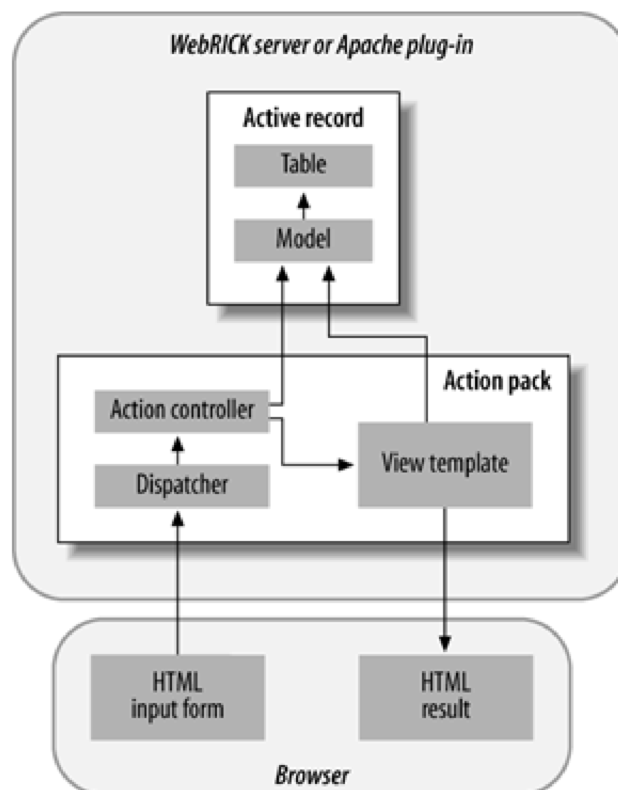

Abbildung 3: Zusammenspiel der Komponenten in Ruby on Rails. [Tate, 2005a]

Action Pack besteht aus Action Controller und Action View (in der Abbildung als „View template“ bezeichnet). Es leitet die eingehenden Anfragen an die entsprechenden Controller weiter („Dispatcher“) und verwaltet die *Sessions*. Action Controller verwaltet die Geschäftslogik und steuert den Ablauf der Anwendung. Action View ist für die Darstellung verantwortlich. Dazu gehört die Aufbereitung der vom Controller gelieferten Daten in Form von HTML, XML, JavaScript und anderen Formaten. Action View integriert die JavaScript-Framworks *Prototype*[22] und *Script.aculo.us*[23]. Neben den klassischen Templates (Ruby eingebettet in HTML) gibt es mit Haml[24], Markaby[25] und Liquid[26] weitere Domänenspezifische Sprachen. Diese werden in [Ediger, 2007, Advanced Rails] vorgestellt.

Active Record bildet die in der relationalen Datenbank gespeicherten Daten in Form von Active-Record-Objekten in die objektorientierte Welt von Ruby on Rails ab. (Die Datenbank wird in der Abbildung als „Table“ und die Objekte als „Model“ bezeichnet.) Die umgekehrte Abbildung von Objekten zur Datenbank gehört ebenso zum Funktionsumfang. Es wird dabei i.d.R. von konkreten Datenbankprodukten und deren herstellerspezifischen Eigenschaften abstrahiert. Das Framework folgt dem Active Record-Architekturmuster aus [Fowler, 2003]. Unterstützt werden grundlegende Operationen wie das Erstellen, Lesen, Ändern und Löschen von Datensätzen sowie Relationen zwischen den Datenmodellen. Dabei gibt es eine integrierte Komponente für die Validierung von Daten. Active Record erstellt selbstständig die entsprechenden SQL-Abfragen. Explizites SQL muss nicht, kann aber, geschrieben werden. Die Attribute (Instanzvariablen) einer Klasse erzeugt Active Record vollautomatisch auf Grund der Informationen der Datenbank(-definition). Neben der automatischen Konvertierung von Daten(-typen) zwischen Modell und Datenbank übernimmt Active Record auf Wunsch die Verwaltung von Zeitstempeln für die Datensätze. Ferner bietet das Framework Validierungshilfen für Objektattribute und die Integration von Callbacks an.

Active Support stellt eine Erweiterung der Standardbibliothek der Programmiersprache Ruby dar. Dabei wird Verhalten ergänzt, das für Ruby on Rails erforderlich ist und/oder die Programmierung in Ruby on Rails erleichtert.

Action Mailer unterstützt den Versand von E-Mails. Dazu können Vorlagen verwendet werden.

Active Resource übernimmt die ressourcenorientierte Verwaltung der Daten auf Grundlage

[22] http://www.prototypejs.org/
[23] http://script.aculo.us/
[24] Sehr Kompakter View-Code. Siehe http://haml.hamptoncatlin.com/
[25] Reiner Ruby-Code in der View. Siehe http://code.whytheluckystiff.net/markaby/
[26] Fokussiert auf Sicherheit. Siehe http://www.liquidmarkup.org/

der REST-Technologie[27]. Dieser Ansatz wird auch als *RESTful Rails* bezeichnet und bietet die Möglichkeit eigene Web-Services mit Ruby on Rails zu erstellen.

RailTies bindet Active Record, Action Controller und Action View in das bzw. als *Rails* Framework zusammen. Darüber hinaus übernimmt es die Konfiguration von Rails und verwaltet die Anfragebehandlung des *Dispatchers*. Ein Web-Server ist ebenfalls integriert.

2.3.4 Verarbeitung einer Anfrage

Dieses Unterkapitel führt ein einfaches Beispiel für die Bearbeitung einer Anfrage in Ruby on Rails auf. Es soll verdeutlichen, welche Komponenten dabei beteiligt sind und wie der komplette Ablauf aussieht. Besonderheiten wie z.B. die Caching-Mechanismen werden dabei nicht berücksichtigt, um die Komplexität des Beispiels niedrig zu halten.

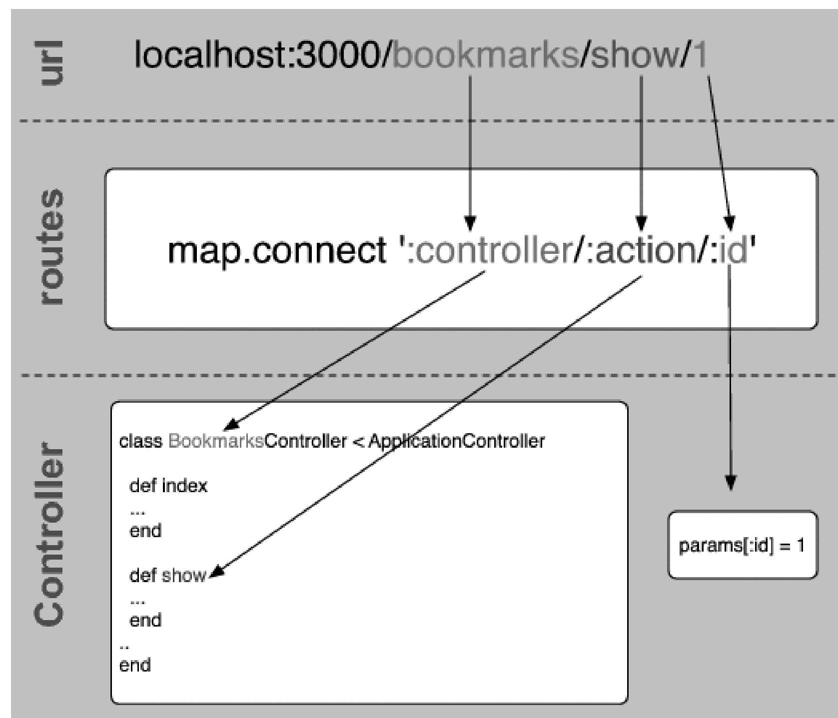

Abbildung 4: Abbildung des Routings von URLs zu Methoden. [Morsy und Otto, 2008]

Abbildung 4 zeigt die Standardkonfiguration des Routings in Rails. Die hier gezeigte URL `http://localhost:3000/bookmarks/show/1` wird vom Web-Browser angefor-

[27] Representational State Transfer. Siehe [Ediger, 2007] und [Wirdemann und Baustert, 2007b, RESTful Rails Development].

dert und von Rails entsprechend der Routingeinstellungen (*routes*) bearbeitet[28]. Der Web-Server der Rails-Anwendung ist unter dem Namen *localhost* auf dem Port *3000* zu erreichen. *bookmarks* entspricht dem Namen des Controllers. Es handelt sich also um den *Bookmarks-Controller* der Anwendung. *show* ist die entsprechende Methode bzw. Aktion. Der letzte Parameter ist die (Datenbank-)ID *1*. Die URL ruft demnach die Methode *show* des *BookmarksController* auf und übergibt ihr die ID 1 in die *params*-Struktur, die alle Parameter beinhaltet.

```
1  class BookmarksController < ApplicationController
2
3    def index
4      # ...
5    end
6
7    def show
8      id = params[:id]
9      # ...
10     if @bookmark = Bookmark.find_by_id(id)
11       render
12     else
13       render :text => "Unzulaessige␣Aktion"
14     end
15   end
16
17 end
```

Codebeispiel 2: Ein beispielhafte Controller-Methode in Ruby on Rails.

Die Methode *show* wird ausgeführt. Ihre Anwendungslogik ist in Codebeispiel 2 beispielhaft dargestellt. Dort wird zuerst der Parameter *id* ausgelesen (Zeile 8). Neben der Ausführung der Anwendungslogik ist der Controller auch dafür verantwortlich, dass die in der View benötigten Variablen zur Verfügung stehen. In diesem Beispiel soll das Lesezeichen mit der ID 1 angezeigt werden. In Zeile 10 wird über das Bookmark-Modell eine Anfrage an die Datenbank gestellt. Ist das Lesezeichen mit der ID 1 dort nicht auffindbar, so wird dem Benutzer durch seinen Web-Browser eine Textmeldung angezeigt (Zeile 13). Im Erfolgsfall (Zeile 11) wird das Template, welches den gleichen Namen wie die Methode trägt, in der View-Komponente gerendert.

[28] Siehe dazu auch Abbildung 3: Der Dispatcher führt diese Zuordnung aus.

```
1 <h1>Zeige Bookmark <%= @bookmark.id %></h1> <br />
2 <%= @bookmark.name %> <br />
3 <%= @bookmark.url %> <br />
4 <%= @bookmark.description %>
```

Codebeispiel 3: Die Template-Datei *view/bookmark/show.html.erb*.

Codebeispiel 3 zeigt die zugehörige View-Vorlage (Template). Dort wird Ruby-Code zwischen den Zeichenkombinationen <%= und %> in die HTML-Vorlage eingebettet, dessen Rückgabewerte daraufhin in die resultierende HTML-Datei aufgenommen werden. In Zeile 1 ist das die ID des Bookmarks. In den Zeilen 2 bis 4 folgen weitere Eigenschaften des Bookmark-Objekts. Die auf diesem Weg dynamisch erzeugte HTML-Datei wird über den Web-Server an den Browser des Benutzers zurückgesandt und dort dargestellt.

2.3.5 Zusammenfassung

Ruby on Rails hat sich im Wettbewerb der Web-Application-Frameworks behaupten können und stellt inzwischen eine ernsthafte Alternative zu anderen Lösungen dar. Für kleinere bis mittlere Web-Anwendungen, die eine relationale Datenbank einsetzen, ist Rails eine gute und sehr produktive Wahl. Dies bestätigen auch [Lerner, 2006] und die in [Schempp, 2008a] und [Schempp, 2008b] beschriebenen, sowie weitere praktische Erfahrungen.

Kritik

Die Konventionen bezüglich der objekt-relationalen Anbindung von Ruby on Rails an die Datenbank können zu Problemen führen, wenn auf eine bereits bestehende Datenbank zugegriffen werden soll. Sofern deren Struktur den Konventionen von Rails stark widerspricht, kann es sinnvoll sein, auf den Einsatz von Ruby on Rails zu verzichten. Bei neuen Projekten beschleunigen die Konventionen hingegen die Entwicklung.

Ein Kritikpunkt an Ruby on Rails ist der von Haus aus unzureichende Schutz gegen Angriffe aus dem Internet. Mit der Version 2 gab es zwar Verbesserungen, doch sind nicht alle Angriffsarten berücksichtigt, sodass manuelle Nacharbeit nötig ist. Ferner können nicht-triviale Abbildungen von objektorientierten Vererbungshierarchien auf Datenbanktabellen für Ruby on Rails eine Herausforderung darstellen. Hier ist ebenfalls ein wenig Nacharbeit nötig.

Das Schreiben der View-Templates in HTML mit eingebettetem Ruby-Code kann mit der Zeit zur Last fallen. Die vielen Sonderzeichen, die zu unnötig langem Code führen, sind ein Grund, warum es mit Haml und Markaby Alternativen zur HTML-Erstellung gibt. Diese Schwäche betrifft neben Rails auch andere Web-Application-Frameworks.

Über Ruby on Rails existiert bereits einige Literatur. Diese bezieht sich leider oft, wie auch ein Großteil der Internet-Dokumentation, auf die inzwischen veraltete Rails-Version 1.2.x. Dennoch ist mittlerweile ausreichend Dokumentation für die aktuellen 2.x Versionen verfügbar. Einige Änderungen der aktuellen Versionen sind so gravierend, dass sich ein aktuelles Buch zur Entwicklung empfiehlt.

Ruby on Rails ist keine umfassende Lösung für alle Arten von Web-Anwendungen. Dies war und ist aber auch nicht der Anspruch des Frameworks. Rails ermöglicht das produktive Erstellen von Web-Anwendungen auf eine angenehme Weise. Dies bestätigt die wachsende Gemeinde der Rails-Entwickler und der zunehmende Einsatz von Ruby on Rails in kommerziellen Projekten. Insbesondere der kurze Feedback-Zyklus führt zu einem nicht zu unterschätzenden (Geschwindigkeits-)Vorteil in der Anwendungsentwicklung. Eine Änderung des Quellcodes oder der Konfiguration wird i.d.R. durch simples Speichern der betroffenen Datei und ein Neuladen der Seite im Browser sichtbar.

2.4 Seaside

Abbildung 5: Seaside setzt Maßstäbe bei der objektorientierten Web-Entwicklung. [Wikipedia, 2008f]

Dieses Unterkapitel stellt das Web-Application-Framework Seaside vor. Neben Seasides wesentlichen Konzepten wird die Integration von Continuations an Hand von Beispielen erklärt. Eine Diskussion von Continuations und der Unterschiede zu klassischen Ansätzen – hier im Kontrast zu Ruby on Rails – findet sich in Kapitel 3 wieder.

Vorstellung

Seaside ist ein im Jahr 2002 von Avi Bryant entworfenes quelloffenes Web-Application-Framework. Es geht innovative und unkonventionelle Wege, die von den Konzepten klassischer Web-Application-Frameworks abweichen. In diesen individuellen Wegen liegt die Stärke des Frameworks. Seaside basiert auf dem freien Smalltalk-Dialekt *Squeak*[29]. Inzwischen gibt es auch Portierungen nach *VisualWorks*, *Dolphin* und *GemStone* Smalltalk.

Abbildung 6: Das freie Squeak-Smalltalk stellt die Basis für Seaside dar. [Wikipedia, 2008g]

In dem Buch [Tate, 2005a, Beyond Java] wird Seaside erstmals in der Literatur erwähnt und vorgestellt. Über Seaside existiert nur sehr wenig Literatur. Mit [Ducasse u. a., 2008, Dynamic Web Development with Seaside] und [Perscheid u. a., 2008, An Introduction to Seaside] gibt es seit diesem Jahr zwei recht umfassende und gute Beschreibungen von Seaside.

[29]Squeak wurde von der Smalltalk Entwickler-Gemeinde mit Unterstützung der Firma Walt Disney entwickelt und trägt das in Abbildung 6 dargestellte Logo. [Black u. a., 2008] dient als Referenz für Squeak.

Ursprünglich entwickelte Avi Bryant Seaside in Ruby, doch mit dem (recht frühzeitigen) Wechsel zu Squeak bot sich eine ausgewachsenere Entwicklungsplattform als Grundlage für die weitere Seaside-Entwicklung an. Mittlerweile wird die Entwicklung von Seaside von einer kleinen Entwicklergemeinde – vornehmlich aus dem US-amerikanischen, französisch- und deutschsprachigen Raum – getragen. Seaside wird kommerziell in verschiedenen Projekten eingesetzt.

2.4.1 Konzepte und Aufbau

Seaside vereinigt einen objektorientierten und Continuations-basierten Ansatz in einem Framework. In diesem Unterkapitel werden Seasides Konzepte und Paradigmen, angelehnt an [Ducasse u. a., 2007] und [Ducasse u. a., 2004][30] vorgestellt.

Objektorientierung und Komponenten

Das Paradigma der Objektorientierten Programmierung, welches mit Simula und Smalltalk in den sechziger und siebziger Jahren aufkam, liegt auch Squeak und Seaside zu Grunde. Eine Seaside-Anwendung besteht aus Komponenten. Jede Komponente kann als ein in sich geschlossenes System oder auch als Teil der Anwendung betrachtet werden. Seasides Komponenten sind wiederverwendbar und lassen sich zu einer Anwendung zusammensetzen. Jede Komponente kapselt – strikt der objektorientierten Denkweise folgend – ihre eigene Ablaufsteuerung und Darstellung, d.h. sie übernimmt die Aufgaben des Controllers und der View für den von ihr repräsentierten Teil der Anwendung.

Ein Beispiel für eine Komponente ist etwa ein (modaler) Dialog zur Auswahl eines Zeitraumes. Dieser Dialog weiß selbst, wie er sich darstellen muss und übernimmt die Steuerung des Anwendungsablaufs, solange er aktiv ist. Da Anwendungen nicht allein darauf basieren, dass eine Komponente die nächste aufruft, hält Seaside so genannte *Tasks* bereit.[31] Tasks bilden die Schnittstelle zwischen den einzelnen Komponenten, indem sie die Web-Anwendung (auf globaler Ebene) steuern, selbst aber keine View-Funktionalität übernehmen. Für die Darstellung sind die jeweiligen Komponenten verantwortlich, an die der Task (zeitweise) die Anwendungssteuerung abgibt.

Die Aufteilung der Anwendung nach dem MVC-Architekturmuster kann durch die Smalltalk-übliche Kategorisierung der Methoden erhalten bleiben. Die Trennung ist jedoch nicht so strikt umgesetzt wie in Ruby on Rails.

[30] „Seaside – A Multiple Control Flow Web Application Framework“ ist der erste wissenschaftliche Artikel über Seaside.

[31] Komponenten erben in Seaside von der Klasse *WAComponent* und Tasks von der Klasse *WATask*.

Da Seaside in Squeak geschrieben ist, kommen die Smalltalk-typischen Stärken ebenfalls zur Geltung. Insbesondere das Debugging ist vorbildlich umgesetzt. Falls es in der Web-Anwendung zu einem Fehler kommt, erscheint im Browser eine Fehlermeldung mit den üblichen Informationen, wie man sie auch von anderen Web-Application-Frameworks (z.B. Ruby on Rails) kennt. Durch einen Klick auf einen speziellen Link im Browser öffnet sich (in der Squeak-Umgebung) ein normaler Smalltalk-Debugger, mit dem man wie gewohnt die Objekte der Umgebung inspizieren und verändern kann. Nachdem der Fehler behoben ist, kann die Anwendung im Web-Browser (durch einen Klick auf „Fortsetzen“ im Debugger) fortgesetzt werden. Im Vergleich zu anderen Web-Application-Frameworks ist dies ein wesentlicher Vorteil bei der Entwicklung. Erwähnenswert ist, dass Änderungen am Code – wie auch in Ruby on Rails – nach dem Speichern sofort im Browser nachvollzogen werden können, ohne dass dazu mehr als ein Neuladen der Seite notwendig ist.

Aufbereitung der Darstellung (Rendern)

Wie erwähnt sind Seaside-Komponenten für ihre Darstellung im Browser selbst verantwortlich. Für diese Darstellung gibt es keine HTML-Templates im klassischen Sinne. Erzeugt wird das XHTML[32] von Seaside zur Laufzeit, wenn die Komponente durch das Framework aufgefordert wird sich zu rendern. Die dafür notwendige Methode heißt in Seaside per Konvention *renderContentOn:*. Sie wird in reinem Smalltalk geschrieben und benutzt eine objektorientierte *Domänenspezifische Sprache*, um die HTML-Strukturen zu erzeugen. Die Formatierungen werden dabei nicht durch HTML, sondern durch Cascading Stylesheets (CSS) ausgedrückt.[33]

```
1 renderContentOn: html
2         html heading: count.
3         html anchor
4                 callback: [ self increase ];
5                 with: '++'.
6         html space.
7         html anchor
8                 callback: [ self decrease ];
9                 with: '—'
```

Codebeispiel 4: Die Render-Methode 'renderContentOn:' in Seaside.

[32] XHTML bezichnet zu XML kompatibles HTML.

[33] Für die visuelle Gestaltung einer Web-Seite sollte die Formatierung in CSS umgesetzt werden. Dies hat Vorteile im Vergleich zur Vermischung der Formatierung mit HTML. Siehe dazu http://www.csszengarden.com/tr/deutsch/.

Codebeispiel 4 verdeutlicht das Rendern an einem Beispiel der *renderContentOn:*-Methode: In Zeile 1 wird als Argument ein Parameter *html* übergeben. Dieser ist vom Typ *RenderCanvas* und dient als Generator für das zu erzeugende XHTML. Im Methodenrumpf wird sämtliches HTML durch das Senden von entsprechenden Nachrichten an das *html*-Objekt erzeugt. In Zeile 2 wird die Instanzvariable *count* zur Darstellung in der Seitenüberschrift übergeben. Die Zeilen 3 bis 5 definieren einen Hyperlink mit dem Titel *++*, der bei Ausführung die Methode *increase* des eigenen Objekts *self* aufruft. Die restlichen Codezeilen funktionieren analog zu diesem Schema.

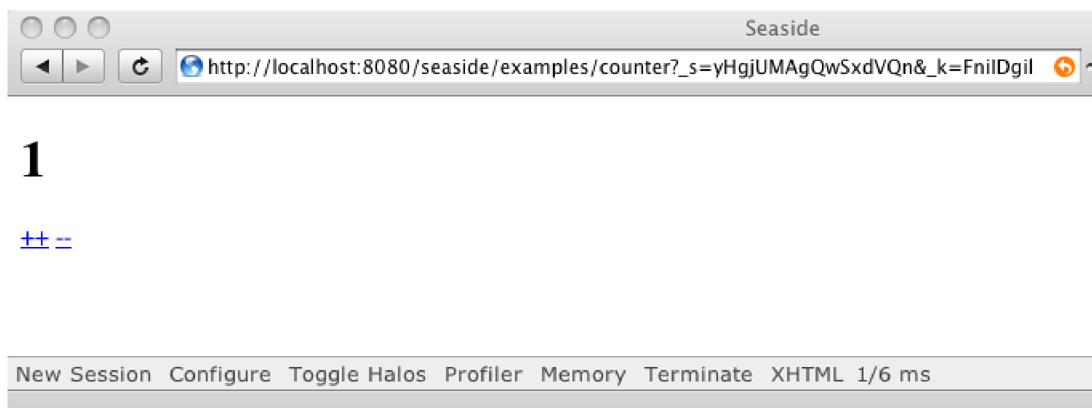

Abbildung 7: Darstellung des Seaside-Zählers.

Das Ergebnis diese Beispiels im Browser ist in Abbildung 7 dargestellt. Zu sehen ist der aktuelle Wert 1 des Zählers. Darunter befinden sich die Links zum Erhöhen und Erniedrigen des Zählers. Typisch für Seaside ist die am unteren Rand zu sehende Menüleiste, die für die Entwicklung sehr hilfreich ist. Ebenfalls typisch ist die oben im Bild sichtbare URL, die sich zusammensetzt aus dem statischen Teil `http://localhost:8080/seaside/examples/counter`, der die Anwendung bezeichnet, und dem dynamischen Teil `?_s=yHgjUMAgQwSxdVQn&_k=FnilDgil`, der die aktuelle Session beschreibt. Näheres dazu Kapitel 2.4.3.

Die Code-Blöcke *[self increase]* und *[self decrease]* aus Codebeispiel 4 werden als Callbacks bezeichnet. Sie werden von Seaside ausgeführt, sobald der Benutzer auf den zugehörigen Link klickt. Gleiches gilt auch für Buttons und weitere GUI-Elemente. Der zugehörige Code-Block wird ausgeführt und die Komponente – hier der Zähler mit einer veränderten Zahl – erneut gerendert. Dabei muss der Entwickler keinerlei Parameter übergeben. Seaside abstrahiert soweit von dem tatsächlichen Geschehen, sodass der Entwickler nur den auszuführenden Smalltalk-Code an die richtige Stelle schreiben muss.

Weiteres

Das hier vorgestellte Beispiel ist sehr einfach gehalten, um die grundlegende Funktionsweise darzustellen. Für ein komplexeres Beispiel sei auf den „Sushi Store“ aus den Seaside-Beispielen verwiesen. An Hand dieses übersichtlichen Online-Shops lässt sich ein umfassenderer Eindruck des Renderns vermitteln, der den Rahmen dieser Arbeit jedoch überschreitet.

Als Web-Application-Framework adressiert Seaside hauptsächlich den Bereich der Benutzeroberfläche und der dafür nötigen Client-Server-Kommunikation. Eine Anbindung an die Datenbank ist nicht integriert, kann aber nach Bedarf hinzugefügt werden. Erwähnenswert ist hier etwa das GLORP-Framework[34], das eine Datenbank objektrelational anbindet. Dem entgegen steht die Verwendung objektorientierter Datenbanken. Dieser Ansatz ist in GemStone Smalltalk integriert, kann aber auch mit den Frameworks GOODs und Magma (siehe [Perscheid u. a., 2008]) ergänzt werden.

Seaside bindet ebenfalls die AJAX-Funktionalität und die bekannten AJAX-Bibliotheken Script.aculo.us und Prototype mit ein. Ferner lässt es sich um die Frameworks *Magritte* und *Pier* von L. Renggli erweitern. Magritte ist ein *Meta-Description-Framework*, das an Hand einer Beschreibung Dialoge und Reports automatisiert erstellt. Magritte wird in dem Wiki-System bzw. Content-Management-System Pier verwendet und eignet sich gut zur dynamischen Erstellung von Benutzerdialogen.[35]

2.4.2 Ablaufsteuerung

Die Steuerung des Kontrollflusses wird in Seaside durch die Komponenten der Anwendung bestimmt. Hier wird ein vereinfachtes Beispiel für die Steuerung der Kontrolle in Seaside gezeigt, um die Verständlichkeit zu fördern.

Codebeispiel 5 beschreibt eine Anwendung, bei der der Anwender eine Zahl zwischen 1 und 50 erraten muss, die von der Anwendung per Zufall ausgewählt wird. Da das Beispiel die Möglichkeiten der Steuerung betonen soll, bezieht es sich nicht auf verschiedene Komponenten, sondern konzentriert sich auf die *go*-Methode des Seaside-Tasks.

[34] GLORP: „Generic Lightweight Object-Relational Persistence“. Ein Framework in Smalltalk zur Anbindung einer relationalen Datenbank. Siehe [Perscheid u. a., 2008].

[35] Mehr Details zu Magritte finden sich in [Perscheid u. a., 2008] und `http://www.lukas-renggli.ch/smalltalk/magritte`. Für Pier siehe `http://www.lukas-renggli.ch/smalltalk/pier`.

```
1  NumberGuesser>>go
2    | guess number |
3    guess := −1.
4    number := 50 atRandom.
5    count := 0.
6    self inform: 'I␣am␣thinking␣of␣a␣number␣between␣1␣and␣50.'.
7    [guess = number]
8      whileFalse: [guess := (self request: 'What␣is␣your␣guess?') asNumber.
9        count := count + 1.
10       guess < number
11         ifTrue: [self inform: 'I␣am␣thinking␣of␣a␣bigger␣number.'].
12       guess > number
13         ifTrue: [self inform: 'I␣am␣thinking␣of␣a␣smaller␣number.']].
14   self inform: 'You␣got␣it␣right!␣(after␣', count asString, '␣trys.)'
```

Codebeispiel 5: Prozedurale Sicht auf die Programmierung eines Tasks in Seaside.

Die Steuerung der Anwendung läuft folgendermaßen ab: Die Methode *go* des Tasks *NumberGuesser* (siehe Codebeispiel 5) wird durch das Framework ausgeführt, wenn ein Benutzer die dazugehörige URL aufgerufen hat. In Zeile 4 wird eine zufällige Zahl ausgewählt. Zeile 6 weist den Benutzer darauf hin, dass er diese Zahl nun erraten soll. Die Methode *inform:* steht in Seaside für einfache Nutzerinteraktionen zur Verfügung. Sie erstellt eine Webseite mit einem Text und einem *OK*-Knopf (siehe Abbildung 8, oben links). Bestätigt der Benutzer diesen Dialog, so läuft die Schleife in den Zeilen 8 bis 13 so lange, bis der Benutzer die richtige Zahl herausgefunden hat. Er wird in Zeile 8 aufgefordert eine Zahl einzugeben. Dazu wird die Seaside-Methode *request:* verwendet, die eine einfache Eingabemaske erzeugt (siehe Abbildung 8, 2. von oben links). Hat der Benutzer eine zu kleine oder zu große Zahl (Zeilen 10 bis 11 bzw. 12 bis 13) angegeben, so wird er darüber entsprechend informiert. Ist die eingegebene Zahl korrekt, so erhält er die Erfolgsmeldung aus Zeile 14, in der auch die Anzahl der benötigten Versuche ausgegeben wird. Ein möglicher Ablauf ist in Abbildung 8 dargestellt.

Codebeispiel 5 demonstriert, dass die Programmierung einer einfachen Anwendung mit Interaktion des Benutzers in Seaside nicht schwierig sein muss. Das Beispiel zeigt eine abstrahierte, prozedurale Sicht auf eine Web-Anwendung, die mit einer linearen Abfolge von Befehlen programmiert wurde. Es wird deutlich, dass der zusammengehörige Code auch an einer Stelle auftritt und nicht, wie in anderen Web-Application-Frameworks, über mehrere Stellen verteilt ist.

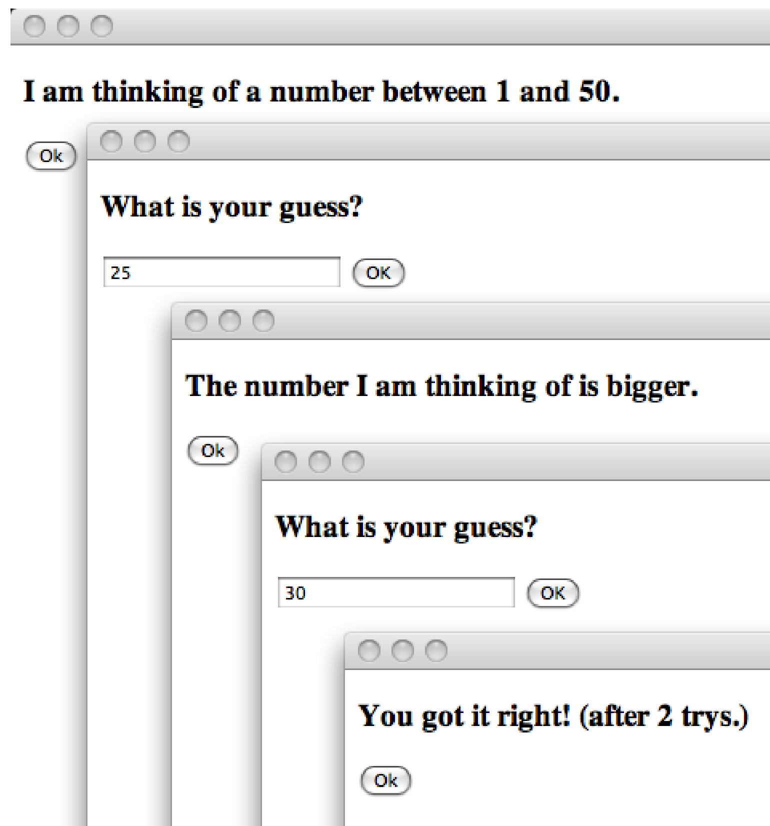

Abbildung 8: Darstellung des Ablaufs der go-Methode. Die aufeinander folgenden Dialoge sind von oben links nach unten rechts angeordnet.

call: und *answer:*

In Seaside wird die Kontrolle an eine andere Komponente übergeben, indem diese durch die Methode *call:* aufgerufen wird. In Codebeispiel 5 wird dies von den Methoden *inform:* und *request:* demonstriert, wenn diese einen Dialog aufrufen. Der Dialog bzw. die Komponente gibt die Kontrolle der Anwendung erst zurück, wenn dort die Methode *answer:* ausgeführt wird. Dabei kann *answer:* auch einen Rückgabewert übergeben. Dies geschieht beispielsweise in Zeile 8. Dort gibt der *request:*-Dialog mit der Kontrolle auch die Antwort des Benutzers an die *go*-Methode zurück.

Abbildung 9 zeigt ein Beispiel, bei dem die Übergabe der Kontrolle zwischen den Komponenten *Filter* und *Calendar* stattfindet. Der *Filter* ruft die *Calendar*-Komponente auf und übergibt ihr somit die Kontrolle. Für die Darstellung im Browser bedeutet dies, dass nun der *Calendar* sichtbar wird. Sobald im *Calendar* ein Datum ausgewählt wird (Aufruf von *select:*), gibt dieser das ausgewählte Datum und die Kontrolle an die *Filter*-Komponente zurück, die daraufhin wieder im Browser dargestellt wird.

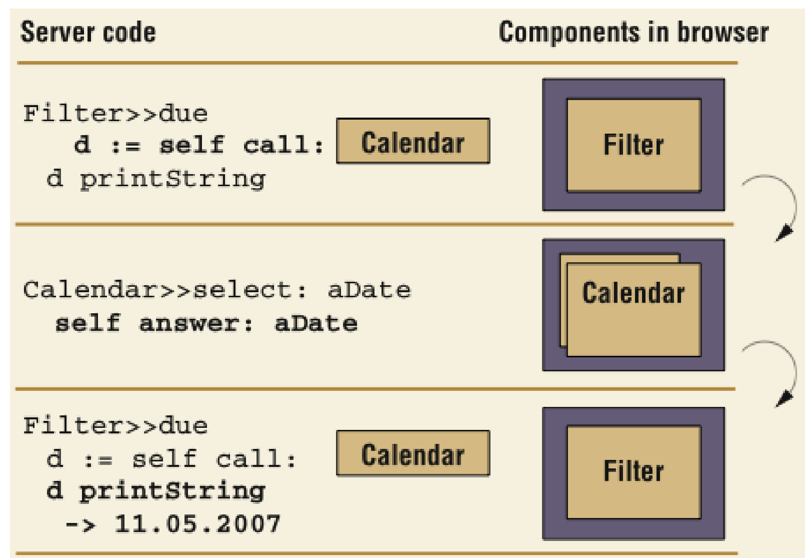

Abbildung 9: Übergabe der Steuerung mit *call:* und *answer:*. [Ducasse u. a., 2007]

Hinter dem *call:* und *answer:* Prinzip stehen die in Seaside integrierten und verborgenen Continuations[36], die für die Verwaltung des Zustands verwendet werden. Wie dieser Zustand genutzt werden kann zeigt das folgende Kapitel 2.4.3.

2.4.3 Zustand einer Anwendung

Die automatische Verwaltung des Zustands einer Seaside-Anwendung soll am Beispiel des bereits vorgestellten Zählers (engl. *Counter*, siehe Abbildung 7) erklärt werden. Der Zustand des Zählers wird sinnvollerweise als Instanzvariable oder als lokale Variable modelliert. Der Zustand bzw. Wert von lokalen Variablen wird beim Einsatz von *call:* und *answer:* durch Seaside automatisch gesichert. Sollen die Werte von Instanzvariablen gesichert werden, so muss dies explizit durch die Hook-Methode *states* angegeben werden. Diese Methode gibt als Rückgabewert die zu sichernden Variablen an das Framework weiter.

Der aktuelle Zustand der Anwendung (für einen Benutzer) wird durch den dynamischen Teil der URL referenziert. Ändert der Benutzer den Zustand der Anwendung durch eine Interaktion, so wird für diesen eine neue URL generiert. Für jede (gültige) URL hält die Anwendung einen passenden Zustand bereit, der durch die Verwendung von Continuations gesichert wird. Abbildung 10 zeigt eine schematische Darstellung der Zustände einer Anwendung (je

[36]Seaside verwendet eine eigene Implementierung von Continuations. Diese basiert auf der Pseudo-Variable *thisContext*, die den Zugriff auf den Stack erlaubt (siehe [Black u. a., 2008]). *thisContext* ist nur in wenigen Smalltalk-Dialekten implementiert, so z.B. in Squeak und VisualWorks, aber nicht in Smalltalk/X.

Session) und deren Continuations. Seaside verwendet für den dynamischen Teil der URL `?_s=XbdPyMSTsEUDgHQJ&_k=tjaIJvWL` die Parameter *_s* und *_k*. Es ordnet *_s* und *_k* als Session ID bzw. Continuation ID dem jeweiligen Zustand zu (siehe Abbildung 10). Dabei hat *_s* als Session ID eines Benutzers den Wert *XbdPyMSTsEUDgHQJ* und *_k* den Wert *tjaIJvWL* als Referenz auf eine bestimmte Continuation.

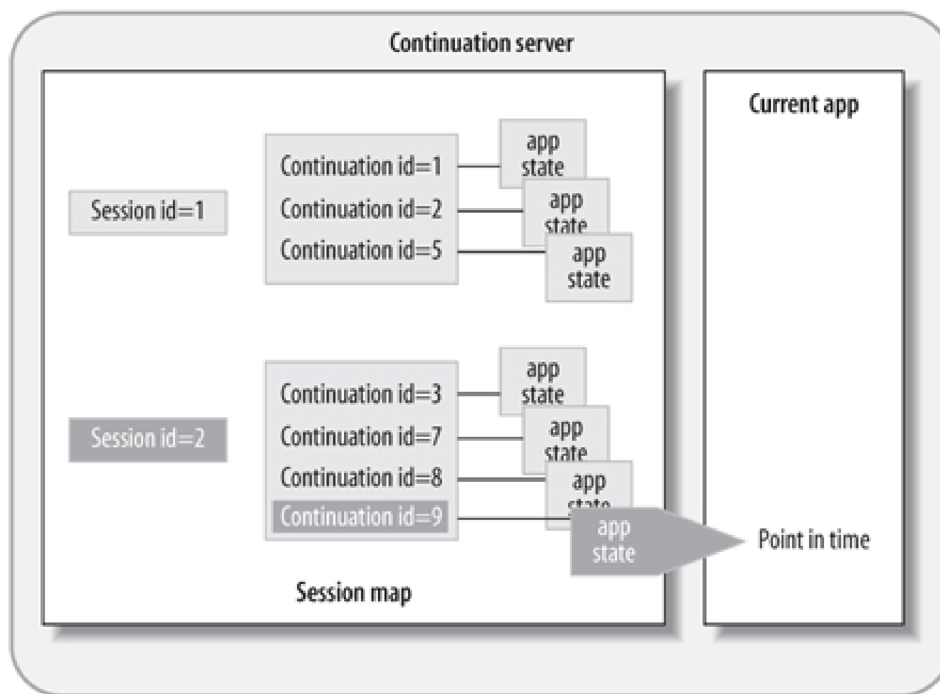

Abbildung 10: Verschiedene Continuations in Seaside. [Tate, 2005a]

Die Zuordnung von URLs zu Zuständen lässt sich ebenfalls am Beispiel des Zählers verdeutlichen. Abbildung 11 zeigt den Zustand (bzw. Wert) des Zählers zu drei verschiedenen Zeitpunkten. Die drei Zustände gehören zu derselben Session, da der Parameter *_s* gleich bleibt. *_k* besitzt hingegen je Zustand einen anderen Wert.

Dass die Anwendung den Zustand tatsächlich speichert, lässt sich nachvollziehen, wenn man eine (durch Seaside generierte) URL von anderer Stelle aufruft. Die Anwendung hat exakt den gesicherten Zustand. Die Zustandsverwaltung wird auch bei der Verwendung der Kontrollknöpfe des Browsers deutlich: Verwendet man bei dem höchsten (dritten) Zähler den Zurück-Knopf des Browsers, so wird auch der vorherige Zustand der Anwendung bzw. des Zählers wiederhergestellt.

Nach dreimaligem Zurückgehen und einmaligem Drücken von *++* erhält man so den in Abbildung 12 abgebildeten Wert 5.[37] Anwendungen ohne integrierte Zustandsverwaltung hätten

[37] Für jeden der dabei ausgeführten Zustandswechsel erzeugt Seaside eine neue URL, die den jeweiligen

an dieser Stelle den Wert 8 angenommen, da der Zustand des Zählers nicht gesichert worden wäre.

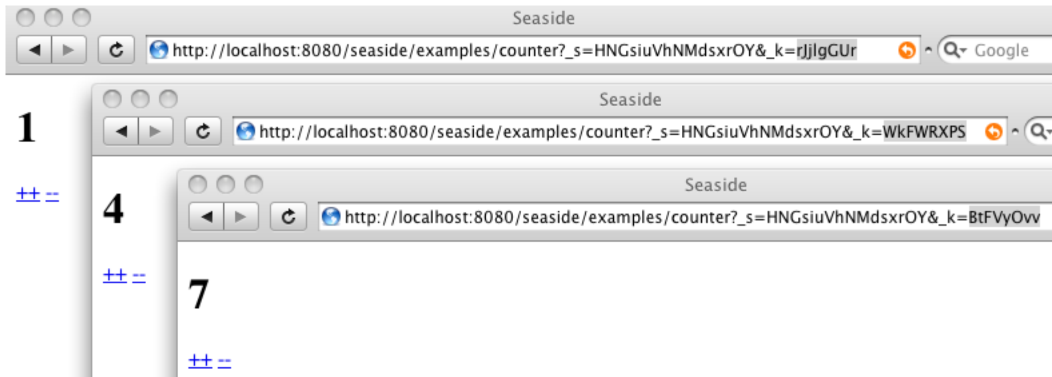

Abbildung 11: Unterschiedliche Zustände zu verschiedenen Zeitpunkten der Zähler-Anwendung.

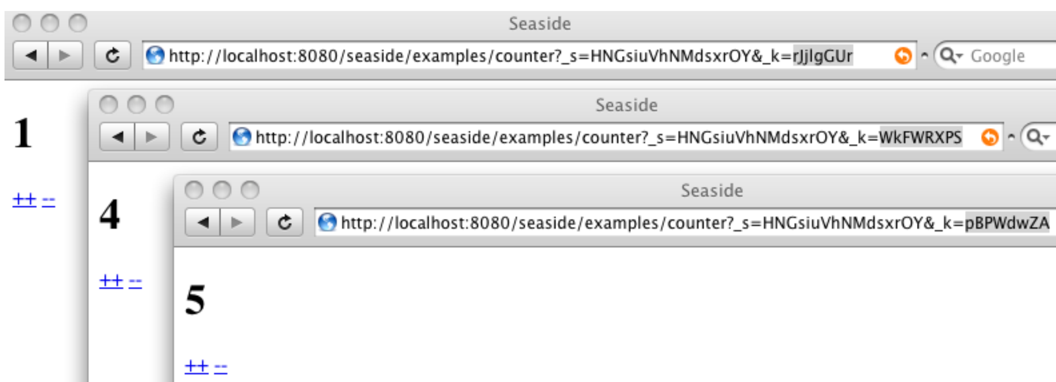

Abbildung 12: Durch Zurückgehen veränderter Zustand des höchsten Zählers.

2.4.4 Kombination unabhängiger Komponenten

In Seaside werden Anwendungen aus verschiedenen Komponenten zusammengesetzt. Es ist darüber hinaus möglich, mehrere Komponenten gleichzeitig auf einer Web-Seite darzustellen. Diese Komponenten sind voneinander unabhängig und steuern ihren eigenen Kontrollfluss. Abbildung 13 zeigt ein solches Beispiel, bei dem drei Zähler in eine *MultiCounter*-Komponente integriert sind. (In der Abbildung als *WACounter* bzw. *WAMultiCounter* bezeichnet.) Die Anordnung der Komponenten wird durch *Halos* unterstützt.

Zustand referenziert. Somit ist die Verwaltung vieler paralleler Handlungsstränge in Seaside ohne Weiteres möglich.

Der Begriff Halo bezeichnet eine Seaside-Funktion, die Rahmen um die sichtbaren Komponenten darstellt und so die (hierarchische) Struktur der Komponenten auf der Webseite verdeutlicht. Diese Funktion ist dem Entwicklungsmodus vorbehalten und stellt dort eine wertvolle Hilfe bei der Programmierung dar. Wird der Zustand einer der Komponenten verändert, ändert sich der Gesamtzustand der Anwendung. Somit wird eine neue URL mit einer neuen Continuation erstellt.

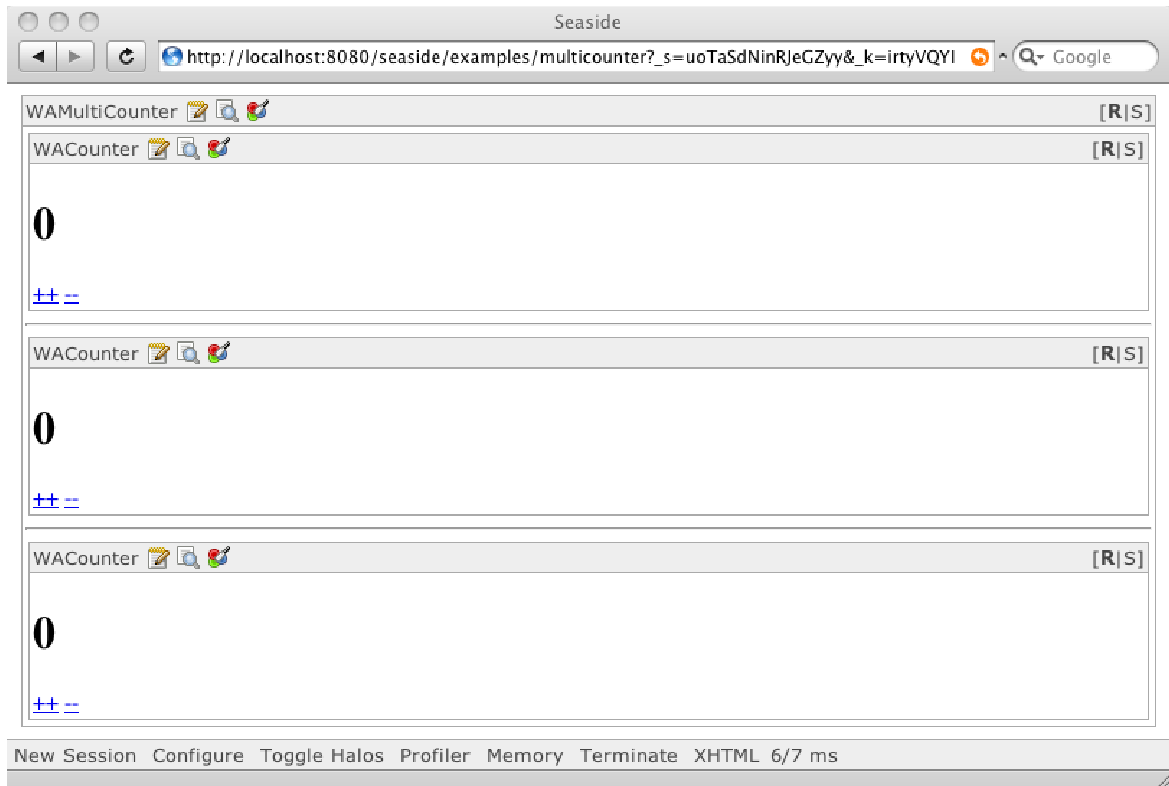

Abbildung 13: Drei Zähler eingebettet in MultiCounter Komponente.

Die Eigenschaft, mehrere Komponenten (mit unterschiedlichen Steuerungsabläufen) gleichzeitig darzustellen und (als Anwender) beliebig zwischen ihnen hin und her zu wechseln, wird oft als „Multiple Control Flow“ beschrieben. Die Besonderheit dieser Fähigkeit ist die Art der Programmierung in Seaside: Komponenten sind flexibel einsetzbar und können sowohl kombiniert als auch getrennt voneinander eingesetzt werden. Dabei kann jede Komponente für sich entwickelt werden, während die Webseite bei anderen Frameworks i.d.R. als Ganzes betrachtet und entwickelt werden muss.

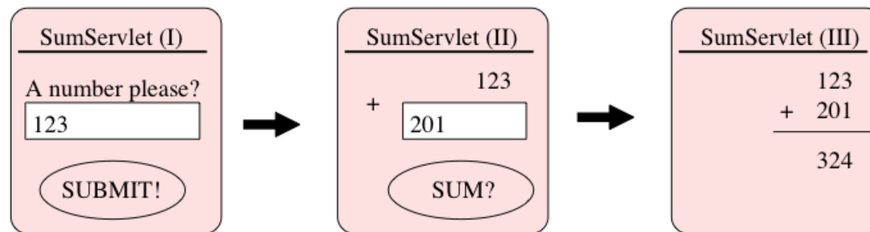

Abbildung 14: Eine einfache Addition. [Queinnec, 2003]

2.4.5 Duplikation von Fenstern

Am Beispiel eines einfachen Addierers wird in diesem Unterkapitel gezeigt, welche Rolle die Zustandsverwaltung einer Web-Anwendung spielt, wenn der Benutzer während einer Sitzung mehrere Browserfenster derselben Anwendung geöffnet hat. Abbildung 14 zeigt den Normalfall dieser Web-Anwendung. Auf der ersten Seite gibt der Anwender den ersten Summanden ein. Die zweite Seite zeigt den ersten Summanden erneut an und erwartet die Eingabe des zweiten Summanden. Das Ergebnis der Addition wird auf der dritten Seite ausgegeben. Der dargestellte (lineare) Ablauf ist erfolgreich und das Additionsergebnis korrekt.

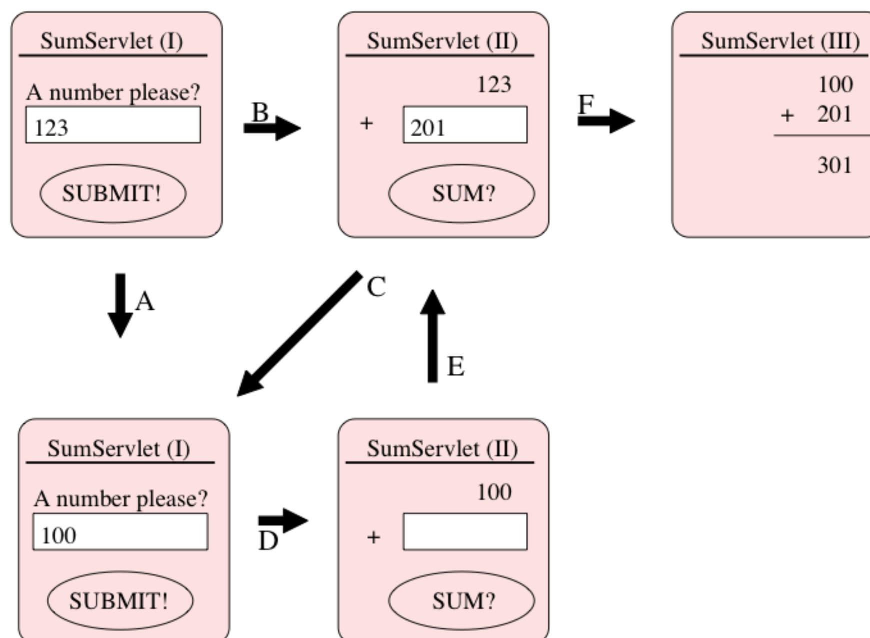

Abbildung 15: Die Addition wird mit einem falschen Wert ausgeführt. [Queinnec, 2003]

Im nächsten Fall öffnet der Anwender ein zweites Browserfenster, indem er die URL der An-

wendung dupliziert, bevor er eine Eingabe vorgenommen hat. Es ergeben sich dadurch zwei parallele Handlungslinien. (In Abbildung ist 15 die eine oben, die andere unten positioniert.) Der Anwender nimmt die Eingaben in beiden Handlungslinien entsprechend der alphabetischen Markierungen der Pfeile vor. Nach Schritt F tritt dabei ein falsches Ergebnis auf, bei dem die Addition zwar korrekt ausgeführt wurde, aber ein falscher Wert für den ersten Summanden verwendet wurde. Dieser Fehler entsteht durch die Belegung der Variablen, die von beiden Handlungslinien verwendet werden. Dabei nimmt die Variable den Wert der – über beide Handlungslinien gesehen – letzten Aktion an. Der erste Summand wird zuerst mit 123 belegt und danach mit 100 überschrieben. Dieses Überschreiben des Wertes wird dem Anwender jedoch nicht vermittelt. Er sieht weiterhin den veralteten Wert 100 und ist durch das anders lautende Ergebnis irritiert. Durch die Abfolge der beiden Handlungsstränge und die gemeinsam genutzten Variablen ist das Verhalten der Anwendung inkonsistent geworden.

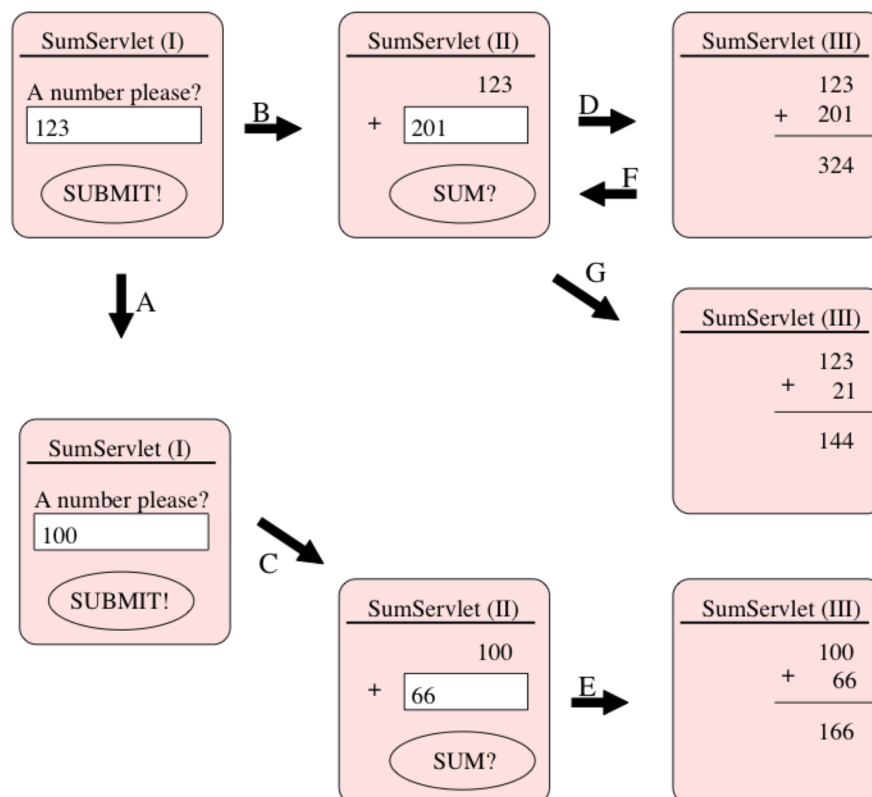

Abbildung 16: Der Addierer mit integriertem Zustand funktioniert korrekt. [Queinnec, 2003]

In Abbildung 16 wurde dieses Problem durch die Sicherung des Zustands behoben. Die Variablen sind so voneinander entkoppelt und haben, je Handlungsstrang, ihren eigenen Wert. Selbst nachdem der Benutzer in der oberen Handlungslinie das Ergebnis (D) angezeigt bekommt, kann er zurück gehen und den zweiten Summanden mit einem neuen Wert belegen

(F). Dieses Beispiel zeigt das gewünschte Verhalten der Anwendung, das (nach [Queinnec, 2003]) durch den Einsatz von Continuations erzeugt werden kann.

2.4.6 Zusammenfassung

Verglichen mit anderen Web-Application-Frameworks ist Seaside noch recht unbekannt. Mittlerweile hat sich eine kleine, aber wachsende Entwicklergruppe um das Framework gebildet. Seaside hat eindrucksvoll bewiesen, dass die Web-Entwicklung mit seinem recht ungewöhnlichen Ansatz einfach und produktiv funktioniert. Nach einer Eingewöhnungsphase ist die durchgängige und klare Linie der Umsetzung von Seaside erkennbar.

Smalltalk-Programmierer dürften sich im Umgang mit Seaside am schnellsten zurecht finden, da es im Smalltalk-Dialekt Squeak geschrieben ist. Durch die Integration in das Smalltalk-Konzept lässt sich der Smalltalk-eigene Debugger nutzen. Dies ist für die Entwicklung von Web-Anwendungen nicht selbstverständlich und von großem Vorteil. Dennoch stößt man auch an Grenzen des Image-Konzepts von Smalltalk, das z.B. für die Einbindung von externen Dateien (CSS, Bilder, etc.) wenig geeignet ist.[38]

Durch den Einsatz von Continuations benötigt Seaside deutlich mehr Speicher als andere Web-Application-Frameworks.[39] Im Gegenzug erhält man eine integrierte, automatisierte Zustandsverwaltung.[40] Mit der Referenzierung von Zuständen durch generierte URLs wird eine höhere Sicherheit erzielt. Neben diesen nicht besonders ästhetischen URLs lassen sich mit Seaside aber auch „schöne“ und aussagekräftige URLs verwenden.

Seaside ist kein Web-Application-Framework, das alle Ebenen der Web-Entwicklung mit einbindet. Seaside kümmert sich im Wesentlichen um das Benutzer-Interface einer Anwendung. Die Abbildung der Objekte auf eine Datenbank wird außen vor gelassen. Hier bieten sich verschiedene Frameworks (z.B. GLORP) aus der Smalltalk-Welt an.

Nachteilig wirkt sich hingegen die kaum vorhandene Dokumentation des Frameworks aus. Bisher gibt es einige kleinere und unvollständige Anleitungen in diversen Weblogs. Vor einigen Monaten ist mit [Perscheid u. a., 2008, An Introduction to Seaside] die erste umfassende Einführung in Seaside erschienen. Noch in diesem Jahr soll mit [Ducasse u. a., 2008, Dynamic Web Development with Seaside] eine weitere ausführliche Einführung in Seaside

[38] Die Einbindung statischer Daten sollte durch einen vorgeschalteten Web-Server (z.B. Apache) gelöst werden, da die Geschwindigkeit so enorm gesteigert wird und das Web-Application-Framework nicht durch das Laden von Dateien zusätzlich belastet wird. Siehe Kapitel „Serving Files“ in [Ducasse u. a., 2008].

[39] Dieser Speicherverbrauch wurde in den letzten Jahren durch Optimierungen deutlich gesenkt.

[40] Die Integration der Zustandsverwaltung und Callbacks mittels Continuations ist in Seaside gut gelöst. Bei näherer Betrachtung wird deutlich, dass die Umsetzung keinesfalls trivial ist. Einen detaillierten Einblick in die Tiefen der Seaside-Continuations gibt das Kapitel 11 in [Perscheid u. a., 2008].

erscheinen, an der einige der Seaside-Entwickler beteiligt sind. Ferner existeren mit [Ducasse u. a., 2004] und [Ducasse u. a., 2007] zwei wissenschaftliche Veröffentlichungen über Seaside, die einen sehr guten Überblick über das Themengebiet rund um Seaside geben.

Das komponentenbasierte Rendern mit der internen Domänenspezifischen Sprache vereinfacht die Darstellung im Vergleich zu seitenbasierten/-zentrierten Vorlagen. Seaside nimmt dem Entwickler die aufwändige und fehleranfällige Verknüpfung von Aktionen mit Web-Seiten ab. Seasides hohe Produktivität lässt sich zudem durch den Funktionsumfang von Frameworks wie Magritte und Pier weiter steigern.

Seaside ist ein wirtschaftlich erfolgreiches Web-Application-Framework. Die kleine Entwicklergemeinde hat ein so gelungenes Konzept verwirklicht, dass sogar mancher Ruby on Rails-Entwickler zu Seaside abwandert. Seasides Konzept im Vergleich zur konventionellen Web-Entwicklung wird in den folgenden Zitaten treffend beschrieben:

„Over the last few years, some best practices have come to be widely accepted in the Web development world. Share as little state as possible. Use clean, carefully chosen, and meaningful URLs. Use templates to separate your model from your presentation.“ – Avy Bryant in [Perscheid u. a., 2008]

„Seaside is a Web application framework for Smalltalk that breaks all of these rules and then some. Think of it as an experiment in tradeoffs: if you reject the conventional wisdoms of Web development, what benefits can you get in return? Quite a lot, it turns out, and this experiment has gained a large open source following, seen years of production use, and been heralded by some as the future of Web applications.“ – Avy Bryant in [Perscheid u. a., 2008]

3 Analyse

In diesem Kapitel werden die unterschiedlichen Umsetzungen der bereits vorgestellten Web-Application-Frameworks untersucht und gegenübergestellt. Diskutiert werden insbesondere die Unterschiede bei der Ablaufsteuerung durch Continuations und der Darstellung (Rendern).

Die hier vorgestellten Unterschiede zwischen den beiden Web-Application-Frameworks dienen nicht dazu, das „bessere" der beiden Frameworks ausfindig zu machen, sondern zur Verdeutlichung der verschiedenen (technischen) Konzepte. Beide Frameworks haben eine unterschiedliche Ausrichtung: Seasides Fokus liegt in der einfachen Ausdrucksweise von Arbeitsabläufen (engl. *Workflows*) komplexer Web-Anwendungen. Ruby on Rails ist hingegen ein traditionelles und vollständiges Framework für kleine bis mittelgroße Web-Anwendungen. Der direkte Vergleich beider Frameworks soll mit einem Zitat von R. Leon abschließend behandelt werden. Er schreibt über die Vorzüge von Seaside:

„However, [...], to each his own, Seaside isn't for everyone. Rails is a fine framework, if it works for you, stick with it." – [Leon, 2007]

3.1 Programmierkonzepte

Ruby on Rails ist nach dem Konzept eines klassischen bzw. traditionellen Web-Application-Frameworks aufgebaut. Die verwendete Technologie wird nach [Queinnec, 2003] als *seitenzentriert* bezeichnet. Sie basiert auf einigen Designentscheidungen, die in diesem Unterkapitel erwähnt und in den folgenden Unterkapiteln genauer erläutert werden.

Eine Web-Anwendung läuft normalerweise auf mehreren Web-Servern[41], um eine angemessene Erreichbarkeit zu gewährleisten. Die Web-Server sollen möglichst unabhängig voneinander sein, um Zustandsinformationen nicht gemeinsam zu nutzen. So können die verschiedenen Anfragen der Clients bei der Lastverteilung einem beliebigen Web-Server zur Bearbeitung zugewiesen werden. Dieser benötigt keine zusätzlichen Informationen über die Benutzersitzung. Alle notwendigen Informationen über den Zustand der Anwendung befinden sich in der Datenbank. Dieser, weitestgehend zustandslose, Ansatz skaliert gut. Setzt

[41] I.d.R. werden damit mehrere Web-Server-Prozesse auf einem (oder mehreren) Rechner(n) bezeichnet.

man hingegen Continuations ein, so müssen die Anfragen eines Clients immer zum selben Server geleitet werden, denn nur dieser hält die entsprechenden Zustandsinformationen im Speicher. Alternativ könnten diese Zustandsinformationen auch an einer zentralen Stelle (z.B. in einer Datenbank) gespeichert werden. Dazu müssten diese jedoch vollständig serialisierbar sein. Die Geschwindigkeit der Anwendung würde darunter merklich leiden.

Für die Verwaltung des Anwendungszustands verwendet Seaside Continuations. Die Technik einer Web-Anwendung steht dem Einsatz von Continuations nicht im Weg, da diese orthogonal zueinander sind. Das HTTP-Protokoll wird durch Continuations um einen Zustand erweitert. Grundsätzlich ist es daher möglich, Continuations in Web-Application-Frameworks einzusetzen. Die Vorteile einer durchgängigen und objektorientierten Umsetzung eines Continuation Based Web Server werden durch Seaside demonstriert. Die Funktionsweise von Continuations wird in Kapitel 3.2 genauer vorgestellt. Deren Auswirkungen auf die Programmierung einer Web-Anwendung werden in Kapitel 3.3 diskutiert.

Des Weiteren unterscheiden sich der seitenzentrierte und komponentenbasierte Ansatz bei der Verwendung von Vorlagen. Der seitenzentrierte Ansatz setzt Vorlagen ein, die sich – wie der Name andeutet – auf eine Web-Seite beziehen. Die Darstellungsschicht (View) wird dabei vom Modell abgegrenzt. Ruby on Rails verwendet Vorlagen und überlässt es dem Entwickler, die richtigen Verknüpfungen zwischen den Anwendungsfunktionen und den dafür zuständigen Controllern zu definieren. Dazu müssen die URLs einem bestimmten Schema genügen.[42] Seaside orientiert sich dagegen stark an seinen Komponenten. Es verwendet keine Vorlagen und verknüpft Aktionen automatisiert mit den zuständigen Controllern. Diese Thematik wird in Kapitel 3.4 vertiefend besprochen.

3.2 Continuations

Bei der Suche nach einer Antwort auf die Frage „Was ist eine Continuation?“ finden sich, besonders im Internet, viele verschiedene Antworten. Dies liegt hauptsächlich an den verschiedenen Perspektiven und Hintergründen (z.B. unterschiedliche Programmiersprachen und Erfahrungen) der Verfasser. Die in den Erklärungen aufgeführten Codebeispiele sind oft derart einfach gehalten, dass die Funktion einer Continuation nur teilweise erkennbar ist. Da zudem manche Erklärungen fehlerhaft sind, wird zunächst eine Definition aufgeführt.

[42] Dieses Schema der URLs wird durch die in der Rails-Anwendung definierten Routen vorgegeben. Es wird hier zwischen dem Standard- und dem RESTful-Routing unterschieden.

Begriffsdefinition

Eine Continuation – zu deutsch etwa „Fortsetzung“ – beschreibt den Zustand eines Programms an einer bestimmten Stelle. Aus einem anderen Blickwinkel betrachtet ist diese Momentaufnahme der „verbleibende Teil der Berechnung“ [Queinnec, 2003]. Ist eine Continuation einmal erfasst worden, kann sie beliebig oft und zu jedem Zeitpunkt aufgerufen werden.[43] Das Programm wird dann von der gesicherten bzw. erfassten Stelle aus fortgesetzt. Es macht, so gesehen, einen Zeitsprung zurück zu einem definierten Punkt in der Vergangenheit. Continuations fallen somit in den Bereich der Metaprogrammierung (siehe [Ediger, 2007]).

In [Ducasse u. a., 2007] findet sich eine technische Beschreibung wieder: Eine Continuation ist eine unveränderbare Repräsentation des (Laufzeit-)Stacks zu einem bestimmten Zeitpunkt. Diese Beschreibung kommt der tatsächlichen Umsetzung einer Continuation sehr nahe. Für die Erstellung einer Continuation müssen folgende Daten gesichert werden:

- Der Programmstack (Stack Frame) mit all seinen Variablen des aktuellen Kontexts.
- Der Programmzähler, der die Stelle angibt, an der der Programmcode beim Aufruf der Continuation fortgesetzt wird. Dies ist i.d.R. die nächste Programmzeile, nach der die Continuation erfasst wurde. Sofern die Erstellung einer Continuation ein Teilergebnis einer Berechnung ist, wird diese Berechnung beim Aufruf erneut durchgeführt. (Dabei wird aber keine neue Continuation erstellt, sondern der Parameter des Continuation-Aufrufs als Ergebnis weiterverwendet.)

Viele Programmierer assoziieren mit einer Continuation ein ihnen bereits bekanntes Programmierkonstrukt. Dies ist i.d.R. ein Trugschluss. Um Missverständnisse auszuschließen, werden hier einige den Continuations ähnliche, aber dennoch verschiedene Programmierkonstrukte erwähnt:

- Eine Continuation ist weder eine Funktion (bzw. Methode) noch eine Prozedur. Beide kehren an dem Ende ihrer Berechnung an die Ausgangsstelle zurück. Eine Continuation kehrt niemals zurück.
- Closures referenzieren den lexikalischen Stack zum Zeitpunkt ihrer Erstellung. Sie können (mehrmals) zu beliebigen Zeitpunkten ausgeführt werden, setzen das Programm aber nicht an der Stelle ihrer Definition fort, sondern kehren (wie ein Funktionsaufruf) zurück.

[43] Eine Continuation kann als *First-Class Objekt* durch eine Variable referenziert werden und so beliebig verwendet werden. Siehe auch [Haynes u. a., 1984].

- Continuations sind weder GOTOs noch Koroutinen. Sie können aber als Werkzeug der Metaprogrammierung eingesetzt werden, um verschiedene Kontrollkonstrukte zu implementieren.

Entwicklung

Über Continuations gibt es viele Veröffentlichungen, die bis in die achtziger Jahre zurückreichen. Den Anfang machten Diskussionen und Veröffentlichungen über Labels und GOTOs[44] in den sechziger und siebziger Jahren. Damals entstand auch der wohlbekannte Artikel [Dijkstra, 1968, Go To Statment Considered Harmful], in dem E.W.Dijkstra die GOTO-Anweisung als überflüssig und gefährlich beschreibt.

J.Reynolds schreibt in [Reynolds, 1993, The Discoveries of Continuations] über die Entstehung von Continuations im Zusammenhang mit dieser Diskussion. Dem erstmals in den sechziger Jahren von A. van Wijngaarden erforschten Konzept, das den „Rest“ eines Programms beschreibt, gab C. P. Wadsworth Anfang der siebziger Jahre den Namen *Continuation*. In [Thielecke, 1999, Continuations, functions and jumps] wird der Übergang von Funktionen über GOTOs zum Konzept des *Continuation Passing Style (CPS)* beschrieben. Die Implementierung von (vollständigen) Continuations durch CPS (und Closures) beschreibt P. Graham ausführlich für die Programmiersprache *Scheme* in seinem Buch [Graham, 1993, On Lisp]. Die Transformation von funktionalem Code in CPS-Schreibweise ist in [Steele, 1976] beschrieben.

Während die ursprünglichen Überlegungen in der Programmiersprache *Algol 60* (und teils in *C*) umgesetzt wurden, findet sich in der Literatur und den Veröffentlichungen ab den achtziger Jahren überwiegend die Sprache Scheme wieder. Scheme ist ein Dialekt der funktionalen Programmiersprache Lisp, die von G.J.Sussman und G.L.Steele Mitte der siebziger Jahre entwickelt wurde.[45] Scheme beinhaltete Continuations von Anfang an und behandelt Funktionen und Continuations als *First-Class Objekte* (siehe [Haynes u. a., 1984]). Die primitive Funktion für die Erstellung einer Continuation trägt in Scheme den Namen *call-with-current-continuation*, kurz *call/cc*.

Beispiel

Ein (nicht-triviales) Beispiel in Scheme soll die Funktionsweise von Continuations verdeutlichen: In Codebeispiel 6 wird eine Addition (Zeile 5) durchgeführt, während der eine Continuation erzeugt wird (Zeilen 5-10), die u.a. den Wert der lokalen Variable *i* sichert. Im ersten

[44] GOTOs (engl. von *go to*) und Sprünge (engl. *Jumps*) bezeichnen nichtlineare Sprünge im Programmablauf.

[45] Siehe dazu auch die Veröffentlichungen über das Lambda-Kalkül von Scheme in [Steele und Sussman, 1976, Lambda: The Ultimate Imperative] und [Steele, 1976, LAMBDA: The Ultimate Declarative]

Durchlauf erhält die Variable *i* den Wert 1, der bei der Erstellung der Continuation in Zeile 9 zurückgegeben wird. Aufgrund dieses Wertes wird in Zeile 18 die Continuation *cc* mit der Zahl 2 als Parameter aufgerufen. Das Programm wird nun in Zeile 5 mit der Addition von 0 (gesicherter Wert von *i*) und 2 (an die Continuation übergebener Parameter) fortgesetzt. Das Programm gibt abschließend die Erfolgsmeldung aus Zeile 15 aus. Wäre der Wert der Variable *i* bzw. der Zustand nicht mit gesichert worden, so hätte das Ergebnis der Addition 3 ergeben und das Programm in Zeile 17 über den Fehlschlag berichtet.

```
1  ;define a global variable
2  (define cc #f)
3  ;define a local variable
4  (let ((i 0))
5    (set! i (+ i (call/cc(lambda (continuation)
6                 (display "in cont\n")
7                 (set! cc continuation)
8                 ;first return of call/cc:
9                  1
10                ))))
11   (display "after set i is: ")
12   (display i)
13   (newline)
14   (if (= i 2)
15     (display "success\n")
16     (if (= i 3)
17          (display "failure\n")
18          (cc 2))))
```

Codebeispiel 6: Sichern des Zustands einer lokalen Variable i in Scheme.

Verschiedene Implementierungen

Unterschiedliche Implementierungsstrategien für Continuations werden in [Clinger u. a., 1988] und [Hieb und Dybvig, 1990] beschrieben und diskutiert. Continuations können (auf Grund ihrer Mächtigkeit) als Grundlage für verschiedene Formen von Kontrollstrukturen herangezogen werden. Ein Beispiel dafür ist die Umsetzung von Koroutinen auf der Grundlage von Continuations, wie sie in [Haynes u. a., 1984, Continuations and Coroutines] beschrieben wird. Continuations können ebenfalls für das *Exception Handling* verwendet werden.

Continuation Passing Style (CPS) beschreibt einen Programmierstil, der mit der Funktionsweise von Continuations übereinstimmt, aber sehr umständlich zu programmieren und zu lesen ist. Er kann als Grundlage für die Implementierung von Continuations herangezogen

werden (siehe [Graham, 1993]). Neben diesen unterschiedlichen Strategien gibt es auch Continuations, die von der Funktionsweise der hier vorgestellten regulären Continuations abweichen. Das Konzept der *Partial Continuations* aus [Double, 2004] beschreibt eine Continuation, die nur einen Teil der Informationen einer regulären Continuation sichert. [Hieb u. a., 1994] beschreiben mit *Subcontinuations* eine weitere Continuation-Art für nebenläufige Berechnungen.

3.3 Rückumkehrung der Steuerung

„Don't call us, we'll call you" lautet das sogenannte Hollywood-Prinzip, welches die Programmierung einer Anwendung mit einem Framework – hier: Web-Application-Framework – beschreibt. Die Steuerung des Kontrollflusses wird dabei durch den selbst geschriebenen Quellcode an das Framework abgegeben, das bestimmte Abläufe der Funktionalität übernimmt. Dabei kommt es vor, dass der eigene Code nicht das Framework, sondern das Framework bestimmte Methoden aufruft, die der Programmierer zu implementieren hat. Dieser Umstand wird mit dem Begriff „Umkehrung der Steuerung" bzw. „Inversion der Kontrolle" (engl. *Inversion of Control*) bezeichnet.[46]

Eine andere Anwendung der Inversion der Kontrolle findet sich bei der Verwendung des Beobachter-Architekturmusters[47] wieder. Dort meldet ein Stück Programmcode sein Interesse an, bei bestimmten Ereignissen benachrichtigt zu werden. Diese Benachrichtigung ist ebenfalls ein (externer) Aufruf einer eigenen Methode.

Bei der Entwicklung einer Ruby on Rails Anwendung ist es üblich, dass der eigene Code vom Web-Application-Framework aufgerufen wird. Dies geschieht beispielsweise, wenn der Benutzer der Web-Anwendung ein Formular abschickt. Die Inversion der Kontrolle führt dazu, dass sich der (zu einer Aktion) zusammengehörige Programmcode über verschiedene Stellen verteilt wiederfindet. Somit wird die Programmierung unübersichtlich und die durchgängige Entwicklung der Web-Anwendung eingeschränkt.

Durch den Einsatz von Continuations kann die Umkehrung der Steuerung rückgängig gemacht werden, da die Programmierung auf einem höheren Abstraktionsniveau stattfindet. Man spricht dabei von der Rückumkehrung der Steuerung (siehe [Queinnec, 2003, Inverting back the inversion of control]). Der Programmablauf kann quasi linear in zusammenhängendem Code beschrieben werden. Dies wurde bereits in Kapitel 2.4 durch die Befehle *call:* und *answer:* beschrieben. Seaside verwendet diese Kontrollkonstrukte, um die Anwendung zum gegebenen Zeitpunkt durch den Aufruf einer Continuation an einer bestimmten Stelle fortzusetzen. Die Programmierung wird somit wesentlich vereinfacht. Sie erinnert stark an die

[46] Siehe dazu auch [Fowler, 2005].

[47] Siehe „Observer Pattern" in [Gamma u. a., 1995].

in die Jahre gekommene Programmierung monolithischer (Terminal-)Anwendungen mit *print* und *read*, die jedoch nicht viel mit dem Zeitalter des Internets gemein haben.

Beispiel

„Whenever the Web application needs input from the user, the app saves the current continuation associated with that user. When the user responds with some information, the saved continuation is restored, and the input provided by the user is returned as the value of the continuation." – [Byrd, 2002]

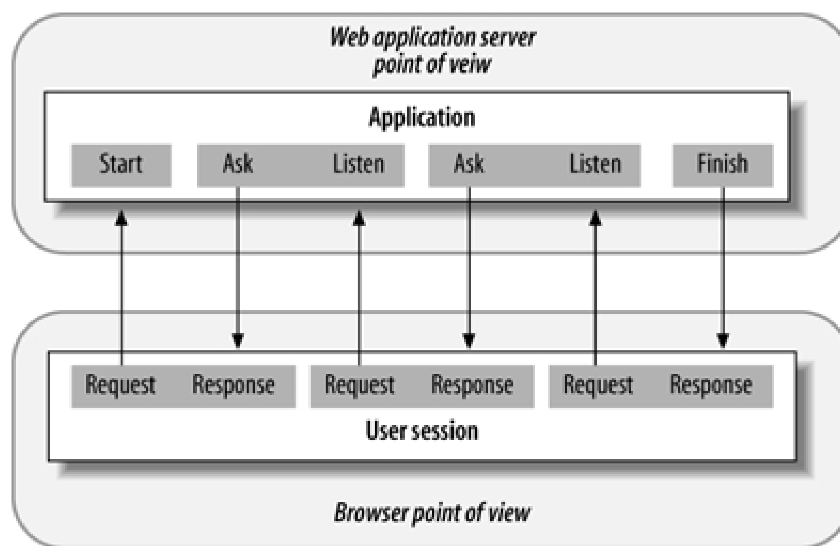

Abbildung 17: Die Ablaufsicht eines Continuation Based Web Server. [Tate, 2005a]

Abbildung 17 stellt die Beziehungen der Funktionsaufrufe einer Web-Anwendung zwischen Client und Server grafisch dar. Aus der Sicht des Browsers wird ein Aufruf (*Request*) an den Web-Server gesendet. Der Browser erhält daraufhin die dazugehörige Antwort (*Response*), die er (dem Anwender) grafisch darstellt. Dieser Vorgang findet wiederholt statt, solange der Anwender diese Anwendung nutzt.

Aus der Perspektive eines Web-Servers gibt es zwei Sichten auf den Anwendungsablauf. Dies ist zum einen die Sicht des klassischen Web-Servers (nicht in der Abbildung zu sehen) und zum anderen die in der Abbildung dargestellte Sicht des Continuation Based Web Servers. Der klassische Web-Server (z.B. der einer Ruby on Rails-Anwendung) wartet auf eine Anfrage des Clients. Trifft eine solche Anfrage ein, so wird sie von der zugehörigen Controllermethode bearbeitet und das Ergebnis an den Client zurückgesandt. Daraufhin wartet der Server auf die nächste Anfrage. Die Gesamtsicht auf die Anwendung ist also zerstückelt, da

der Anwendungsserver immer nur (für ihn zusammenhangslose) Programmstücke abarbeitet.

Ein Continuation Based Web Server wie Seaside sieht hingegen den gesamten Ablauf (der Aktion einer Komponente). Mit dem Eintreffen der ersten Anfrage startet der Web-Server die zugehörige Aktion (bzw. Methode), die solange fortläuft, bis sie an ihrem Ende angekommen ist. Abstrakt gesehen, wartet der Continuation Based Web Server dabei nicht auf Anfragen des Clients, sondern stellt selbst Anfragen (*Ask*) an diesen, wenn dies sein Programmablauf erfordert. Diese Anfrage, bei der eine Continuation gespeichert wird, stellt gleichzeitig die Antwort auf die Anfrage des Clients dar. Sendet der Client daraufhin die nächste Anfrage, so interpretiert der Continuation Based Web Server diese als Antwort auf seine vorherige Anfrage und setzt sein Programm an der vorher gespeicherten Stelle fort (*Listen*). So gesehen stellen die in der Abbildung 17 grau hinterlegten Rechtecke zusammengehörige Anfrage- und-Antwort-Paare dar. Zwischen der Anfrage und der Antwort schreitet die Anwendung auf keiner der beiden Seiten voran, es wird ausschließlich auf die Antwort gewartet.

Auswirkungen

„The client-server relationship's stateless nature requires passing the current state back and forth between browser and server, inevitably leading to undesired coupling between these parts." – [Ducasse u. a., 2007] über klassische Web-Application-Frameworks.

Der Vergleich zwischen der Programmierung von Web-Anwendungen mit klassischer Technik und der Verwendung von Continuation Based Web Servern zeigt, dass letztere die mächtigere bzw. die überlegene Technik verwenden. Obwohl der Einsatz von Continuation Based Web Servern nicht in allen Fällen empfehlenswert ist, vereinfacht der Einsatz von Continuations die Entwicklung einer Web-Anwendung wesentlich, da er eine natürliche Sicht auf den Anwendungsablauf ermöglicht.

Durch die Abbildung von URLs auf den Zustand der Anwendung bzw. an Continuations ist die Funktionalität des Backtrackings schon in die Web-Anwendung integriert. Durch Verwendung des Zurück-Knopfs im Browser wird ein vorheriger Zustand der Anwendung durch die in der Vergangenheit gespeicherte Continuation aktiviert. Dennoch ist es an dieser Stelle wichtig, das Backtracking bewusst zuzulassen bzw. zu unterbinden, da es nicht in jeder Anwendung gewünscht ist. Die Problematik beim Backtracking liegt darin, dass das Zurückgehen in der Benutzeroberfläche synchron zum Zustand des Modells (bzw. der Datenbank) geschehen muss. Bei einem Buchungssystem für Fahrkarten ist es beispielsweise nicht gewünscht, dass die Buchung durch Drücken des Zurück-Knopfs widerrufen werden kann, nachdem sie bereits bestätigt worden war. An dieser Stelle findet die *isolate:*-Methode von Seaside Verwendung. Sie unterbindet das Zurückgehen über selbstdefinierte Grenzen hinweg, sodass das eben genannte Beispiel korrekt umgesetzt werden kann.

Paul Graham erkannte die Vorteile von Continuations für Web-Anwendungen schon recht früh und setzte sie, seiner Zeit voraus, in der Lisp-basierten Web-Anwendung ViaWeb ein (siehe [Graham, 2004] und [Graham, 2001]). Die Vorteile der Continuations in Bezug auf die Ablaufsteuerung beschreibt Christian Queinnec folgendermaßen:

„Instead of seeing a web application as the cooperation of a set of pages, the web application is defined as a single program where continuations take care of sliced execution.“ – [Queinnec, 2003]

3.4 Rendern

Seaside und Ruby on Rails verfolgen bei der Darstellung unterschiedliche Konzepte. Ruby on Rails verwendet Vorlagen. Seaside setzt dagegen eine Domänenspezifische Sprache ein. Der Verzicht auf Vorlagen in Seaside ist die logische Fortsetzung des komponentenbasierten Konzepts, übertragen auf die Darstellungsschicht.

Seitenzentrierte Vorlagen in Ruby on Rails

Vorlagen tragen in Rails die Dateierweiterung *.html.erb*. Sie enthalten HTML mit eingebettetem Ruby-Code. Um die Übersichtlichkeit und Wartung der Anwendung zu vereinfachen, können so genannte *Partials* erstellt werden. Ein Partial ist ein logisch zusammengehöriger Teil einer Webseite, der innerhalb eines Templates gerendert wird.[48]

Für die Darstellung von Daten des Models in der View verwendet Ruby on Rails Instanzvariablen. Diese müssen vorher in der zugehörigen Controllermethode erzeugt werden. Für die Fortführung der Web-Anwendung durch Aktionen werden in Rails HTML-Links (oder Javscript-Code) definiert. Deren URLs müssen einem festgelegten Schema genügen und genau zu den in der Anwendung generierten Routen passen, sodass sie vom Dispatcher der richtigen Controllermethode zugeordnet weden. Über dieses Verfahren der „Verlinkung“ schreiben [Ducasse u. a., 2007]:

„Links pass control from one script to the next. This imposes a go-to hardwiring of the control flow because each page must know what comes next.“

Codebeispiel 7 soll den vorgestellten Sachverhalt verdeutlichen: In Zeile 1 findet sich normales HTML wieder. Die Zeilen 2 und 9 erzeugen durch eingebetteten Ruby-Code eine Schleife, die über jedes Objekt der durch die Instanzvariable *@items* referenzierten Liste iteriert. Um ein solches Item-Objekt anzuzeigen, wird in Zeile 5 ein entsprechender Link für

[48] Das Erstellen von Partials ist in etwa mit dem Refactoring einer zu langen Methode vergleichbar, bei dem ein Teil des Codes in neue (Unter-)Methoden verschoben bzw. herausfaktorisiert wird.

diese Aktion definiert. Die Rails-Methode *link_to* erwartet dazu (neben einer Bennenung des Links) die Angabe eines Methodennamens und einer Datenbank-ID für das betreffende Objekt. Optional kann ein Controllername übergeben werden. In Zeile 7 wird eine Eigenschaft des jeweiligen Item-Objekts dargestellt. Zeile 11 erzeugt, sofern der Benutzer an der Anwendung angemeldet ist, einen Link, um ein neues Objekt zu erstellen. Dabei muss keine ID angegeben werden.

```
1  <h1>This is a Ruby HTML-ERB example</h1>
2  <% @items.each do |item| %>
3    <tr>
4      <td>
5        <%= link_to(item.title ,:controller=>:item ,:action=>:show ,:id=>:42) %>
6      </td>
7      <td><%= item.date %>
8    </tr>
9  <% end %>
10 <p>
11   <%= link_to("Create new Item", :action => :new) if @user.logged_in? %>
12 </p>
```

Codebeispiel 7: View-Template in Ruby on Rails. Nach [Leon, 2007]

Domänenspezifische Sprache in Seaside

In Seaside ist jede Komponente selbst für ihre Darstellung verantwortlich. Dazu wird die Methode *renderContentOn:* implementiert, die in einer internen Domänenspezifischen Sprache in reinem Smalltalk geschrieben sind. Dies hat zum einen den Vorteil, dass man nicht zwischen verschiedenen Sprachen hin und her wechseln muss, zum anderen können so die normalen Smalltalk-Werzeuge – etwa für das Refactoring – verwendet werden.

Um Links für Aktionen auf einer Web-Seite zu platzieren, werden in Seaside Callbacks durch (Code-)Blöcke definiert. Das Framework generiert an dieser Stelle einen HTML-Link mit einer bestimmten URL, an die der Codeblock des Callbacks durch eine Continuation gebunden wird. Während der Entwickler normalen Smalltalk-Code schreibt, übernimmt das Framework im Hintergrund die meiste Arbeit.[49] Der im Callback befindliche Programmcode kann dabei auf die üblichen Variablen der Komponenteninstanz zugreifen. In Ruby on Rails werden an

[49] Das Framework generiert dabei nur Links, die einer gültigen Aktion zugeordnet sind. Wird ein (für den aktuellen Kontext) ungültiger Link aufgerufen, so wird diese Anfrage vom Framework verworfen. Da demnach nur die aktuell verfügbaren Aktionen gültig sind bzw. akzeptiert werden, wird die Sicherheit der Web-Anwendung verbessert. Man vergleiche dazu den routenbasierten Ansatz von Ruby on Rails, bei dem i.d.R. alle Aktionen zugänglich sind.

dieser Stelle, wie oben gezeigt, ausschließlich Objekt-IDs als Parameter übergeben. Aufgabe des Rails-Programmierers ist es nun, zunächst das zu bearbeitende Objekt neu (aus der Datenbank) zu laden. In Seaside ist dies unnötig, da hier keine IDs als Parameter, sondern Referenzen auf (im Speicher gehaltene) Objekte übergeben werden.

Analog zu Codebeispiel 7 ist das entsprechende Seaside-Gegenstück in Codebeispiel 8 dargestellt. Die beschriebenen Callbacks sind in den Zeilen 6 und 11 zu sehen.

```
1  renderContentOn: html
2    html h1: 'This␣is␣a␣Squeak␣HTML–DSL␣example'.
3    self items do: [:anItem |
4      html tableRow id: #item, anItem id;
5        with: [html anchor
6          callback: [self editItem: anItem];
7          with: anItem title]
8        tableData: recipe date].
9    html paragraph: [
10     [user logged_in] ifTrue: [html anchor
11         callback: [self editItem: Item new];
12         with: 'Create␣new␣Item']].
```

Codebeispiel 8: View-Methode mit Callbacks in Seaside. Nach [Leon, 2007]

Dieses Beispiel mag kompliziert wirken, dies liegt jedoch daran, dass es in Seaside unüblich ist, so lange Methoden zu schreiben. Normalerweise würde sich jede Komponente mit (einigen) kurzen Methoden selbst rendern, sodass die Übersichtlichkeit erhalten bleibt. Das aufgeführte Beispiel weicht jedoch von den üblichen Konventionen ab, um eine größtmögliche Ähnlichkeit zu dem vorhergehenden Beispiel in Ruby on Rails (siehe Codebeispiel 7) beizubehalten.

In Ruby on Rails ist es hingegen (auf Grund der seitenzentrierten Ausrichtung) üblich, längere Templates zu verwenden, die alle Objekte einer Web-Seite gemeinsam rendern. Wie bereits in Kapitel 2.3.5 erwähnt, gibt es für Ruby on Rails Plugins für ein alternatives Vorgehen beim Rendern. Die Plugins Haml und Markaby generieren das HTML mit einer Domänenspezifischen Sprache. Diese beiden (ebenfalls template-basierten Varianten) kommen dem Seaside-Konzept ein kleines Stück näher.

Für beide Web-Application-Frameworks gilt der allgemeine (Programmier-)Stil, dass Formatierungen für Web-Seiten nicht in HTML geschrieben, sondern mit CSS ergänzt werden sollen. Dies fördert die Übersichtlichkeit bei der Programmierung und der Darstellung in Browsern ohne CSS (bzw. ohne Layout). Ferner wird diese Trennung als guter Stil angesehen (siehe [Leon, 2007]).

3.5 Zusammenfassung

„Continuations increase application control flow abstraction.“ – [Ducasse u. a., 2004]

In diesem Kapitel wurden die unterschiedlichen Konzepte der Web-Application-Frameworks Ruby on Rails und Seaside gegenübergestellt und diskutiert. (Einen tabellarischen Vergleich der Konzepte zeigt Abbildung 18.) Neben der detaillierten Vorstellung von Continuations wurden deren Integration und Auswirkungen auf das Seaside Framework beschrieben und mit bisherigen Lösungswegen verglichen. Wesentliche Verbesserungen versprechen Continuations bei der Rückumkehrung der Kontrolle. Ferner wird das Rendern durch die Verknüpfung von Links mit Callbacks erleichtert. Nach [Byrd, 2002] erleichtern Continuations die Portierung von alten Desktop- bzw. Legacy-Anwendungen zu Web-Anwendungen, da der Kontrollfluss der Anwendung nicht grundlegend verändert werden muss.

Voraussetzungen für die Verwendung von Continuations sind ihre Unterstützung als First-Class Objekte durch die Programmiersprache. Im Vergleich zu anderen (nicht continuations-basierten) Frameworks für die Zustandsverwaltung birgt dies den Vorteil, dass die vorhandenen (Standard-)Entwicklungs-Werkzeuge weiterverwendet werden können.

Sehr hilfreich ist die Verwendung von Continuations, wenn es um die automatisierte Verwaltung von Zuständen geht. In [Queinnec, 2003] wird betont, dass die Möglichkeiten der Duplikation von Browserfenstern und die Verwendung des Zurück-Knopfs im Browser eine (logische) Konsequenz des Einsatzes von Continuations sind und somit eine nur unerhebliche Mehrarbeit erfordern. Continuations ermöglichen darüber hinaus die relativ einfache Modellierung komplexer Arbeitsabläufe und Interaktionen einer Web-Anwendung. Continuations sollten dennoch nicht nach Belieben eingesetzt werden, sondern – wie es Seaside demonstriert – in ein Framework integriert werden. Dies mindert die Gefahr der falschen Verwendung dieses mächtigen Kontrollkonstrukts der Metaprogrammierung.

Nachteilig wirkt sich die Speicherung von Continuations hingegen auf den Speicherverbrauch der Anwendung aus. Dieser steigt z.T. stark an, wobei dies eng an die Implementierung der Continuations gebunden ist. In Seaside konnte durch geschicktere Implementierungen ein Großteil des erhöhten Speicherverbrauchs wieder eingespart werden. Dennoch sollte ein höherer Speicherverbrauch bei den heute verfügbaren Speichermengen, dem günstigen Speicherpreis und den vergleichsweise hohen Stundenlöhnen nicht von der Verwendung von Continuations abschrecken. Im Vergleich zur ansteigenden Produktivität sind die Kosten des Speicherverbrauchs deshalb fast zu vernachlässigen.

Echte Alternativen zu Continuations, die einen Vorteil (in ähnlichem Umfang und bei vergleichbarem Mehraufwand) versprechen, scheint es (zum jetzigen Zeitpunkt) nicht zu geben. Zustandsmaschinen und andere Architekturmuster sind hier weniger geeignet, da sie die Thematik des Kontrollflusses außen vor lassen (siehe Kapitel 4.3).

Auf Continuations basierende Web-Application-Frameworks gliedern sich gut in die von Kent Beck in [Beck und Andres, 2005, Extreme Programming explained] vertretene „Keep it Simple“ Strategie ein. Die Rückumkehrung der Steuerung und der natürliche lineare Programmablauf machen Web-Programmierung „einfach“. Die Möglichkeit, bei Callbacks direkt mit Objekten zu arbeiten anstatt IDs manuell zu übergeben, steigert die Produktivität erneut. Durch einen höheren Grad an Abstraktion können die für Web-Anwendungen nachteiligen Eigenschaften des zustandslosen HTTP ausgeglichen werden. Die Modellierungslücke in der Zustandsverwaltung bisheriger Web-Application-Frameworks kann durch eine gelungene Integration von Continuations verkleinert bzw. geschlossen werden.

Der Einsatz von Continuations sollte dennoch gut überlegt sein und nur an notwendigen Stellen erfolgen, da sonst Speicherverbrauch, Geschwindigkeit, Verständlichkeit und Wartbarkeit der Anwendung unter deren Verwendung leiden. Paul Graham schreibt in [Graham, 2005], dass man im Wettbewerb mit Anderen die mächtigsten und leistungsstärksten Werkzeuge verwenden soll, um (gegenüber diesen Wettbewerbern) zu bestehen. Solch ein mächtiges Werkzeug sind auch Continuations. Das folgende, dieses Kapitel abschließende Zitat, beschreibt die Vorteile von Continuations in Seaside nochmals mit anderen Worten:

„The real power of the call and answer mechanism relies in the capability to build a flow of components which embed several complex user interactions. Seaside allows one to call different components one after the other, using control-statements such as loops and conditional clauses, or other non web related code in-between these calls. An important point is that passing control to another component is done with normal message sending: A method returns and the execution continues from this point – even if this is at an undefined point in the future. This prevents the goto-like definition of control flow without the possibility to return.“ – [Ducasse u. a., 2004]

	Ruby on Rails	**Seaside**
Konzepte	klassisch seitenzentriert	objektorientiert komponentenorientert
Zustand		
Continuation Based Web Server	nein	ja
Serialisierbare Continuations	nein	ja
Sichern / Continuations	eingeschränkt	ja
Verwaltung (automatisiert)	nein	ja
Rendern		
Callbacks	nein	ja
Templates	ja	nein
URL-Generierung	eingeschränkt	ja
Ablaufsteuerung		
Duplikation von Fenstern	eingeschränkt	ja
Komponentenübergreifend	begrenzt	ja
linear/natürlich	nein	ja
parallele Handlungsstränge	begrenzt	ja
Rückumkehrung der Steuerung	nein	ja
Zurück-Knopf	eingeschränkt	ja
Sonstiges		
AJAX-Integration	ja	ja
Datenbankanbindung	ja	nein
Dokumentation	gut	ausreichend
Entwicklungsunterstützung im Browser	nein	integriert
HTML-Erstellung mit DSL	optional	integriert
Internationalisierung	ausreichend	gut
Metaprogrammierung	ja	ja
Speicherverbrauch	normal	erhöht
URL-Gestaltung	schön	sicher

Abbildung 18: Gegenüberstellung der Eigenschaften von Ruby on Rails und Seaside.

4 Umsetzbarkeit

In den vorangegangenen Kapiteln wurden Ruby on Rails und die Vorteile von Continuations in Seaside vorgestellt und diskutiert. Continuations sind grundsätzlich für den Einsatz in Web-Application-Frameworks geeignet. In diesem Kapitel werden die Erfahrungen und das Vorgehen für die Verwendung von Continuations in Ruby on Rails beschrieben. Da Rails und Seaside grundlegende Unterschiede im Aufbau und durch ihre Programmiersprache aufweisen, werden für die Verwendung von Continuations in Ruby on Rails nicht die gleichen Ergebnisse wie in Seaside erwartet. Dieses Kapitel zeigt, inwiefern Ruby on Rails von Continuations profitieren kann.

Andere, neben Rails existierende Ruby-Frameworks für die Web-Entwicklung werden in Kapitel 4.3 erwähnt.

4.1 Planung

Die Verwendung von Continuations in Ruby on Rails soll hier beispielhaft gezeigt werden, aber kein fertiges oder verwendbares Produkt ergeben. Als grundlegende Voraussetzung implementiert die Programmiersprache Ruby Continuations und setzt sie darüber hinaus in der *Generator*-Bibliothek ein.

Der in dieser Arbeit verfolgte Ansatz zur Verwendung von Continuations betrifft hauptsächlich den (Anwendungs-)Controller von Ruby on Rails. Das Modell (bzw. Active Record) ist von der Umsetzung nicht betroffen. Änderungen an der View sollen nur vollzogen werden, wo sie wirklich nötig sind. Das vorhandene Template-System und das Routing (Dispatcher) sollen dabei unverändert bleiben.

Für die folgenden Überlegungen soll das aus Seaside bekannte, einfache Zähler-Beispiel herangezogen werden. Die nachfolgend beschriebenen Schritte einer (theoretischen) Umsetzung in Ruby on Rails werden in Abbildung 19 zusammenhängend dargestellt.

- Der Client stellt eine initiale Anfrage an die Zähler-Anwendung. Die zugehörige Methode des CounterControllers wird gestartet. Der Zähler wird initialisiert. Daraufhin wird das entsprechende Template gerendert, an den Client zurückgeschickt und dort dargestellt. Nach dem Rendern wird eine Continuation erstellt.

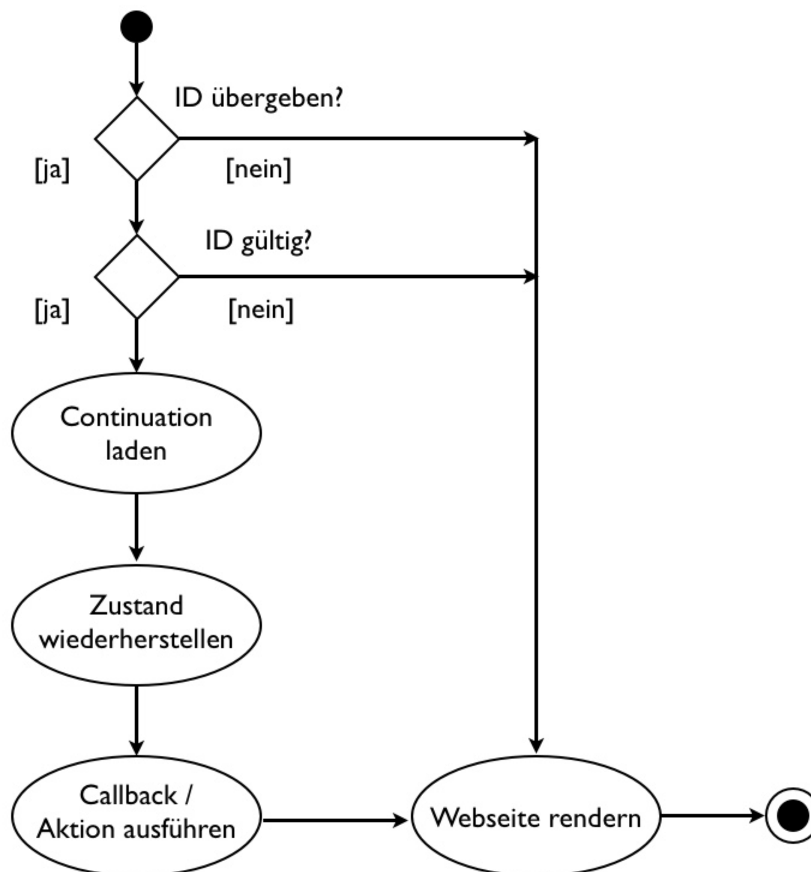

Abbildung 19: Mögliches Aktivitätsdiagramm des Zählers in Rails.

- Der Benutzer kann den Zähler durch drücken des *++*- oder *--*-Links verändern. Diese Links beinhalten verschiedene (eindeutige) IDs, die der gesicherten Continuation und der entsprechenden Aktion (*++* oder *--*) zugeordnet wurden.
- Durch die Verwendung eines Links wird eine neue Anfrage an den Controller gesendet. Dieser wertet den ID-Parameter der Anfrage aus und setzt das Programm an der durch die Continuation gespeicherten Stelle fort. Anschließend wird die Erhöhung oder Verringerung des Zählers ausgeführt, sofern (für diese ID) eine Aktion hinterlegt wurde.
- Der geänderte Zustand wird (mit neuem Zählerwert) gerendert. Dabei werden für die *++*- und *--*-Links neue (eindeutige) URLs erzeugt. Anschließend wird der Zustand in einer neuen Continuation gespeichert.
- Der Benutzer kann nun beliebig mit der Veränderung des Zählers fortfahren.

- Drückt der Benutzer den Zurück-Knopf, so wird die vorherige Seite aus dem Browser-Cache geladen und dargestellt. Die Links für die ++- und ---Aktionen zeigen nun auf die damals generierten URLs. Über die ID eines solchen Links stellt der Controller der Anwendung fest, dass es sich um einen zuvor gespeicherten Zustand handelt. Durch die Aktivierung der zugehörigen Continuation wird die Anwendung an ihrem damaligen Zustand fortgesetzt.
- Für den Fall, dass zu einer (in der URL enthaltenen) ID kein passender bzw. gültiger Zustand existiert, wird diese Anfrage verworfen und der Benutzer zur Startseite der Anwendung weitergeleitet.
- Jegliche Veränderungen des Benutzers an einem wiederhergestellten Zustand überführen die Anwendung in einen neuen Handlungsstrang, der unabhängig und parallel zu vorhandenen Handlungssträngen existiert. Dies wird dadurch gewährleistet, dass bei jedem Rendern für alle Links neue IDs erstellt werden, um die verschiedenen Handlungsstränge zu unterscheiden. Ein neuer Handlungsstrang hat dementsprechend keine Kenntnis von der „Zukunft“, aus der der Benutzer „zurückgesprungen“ ist.
- Startet ein Benutzer die Anwendung zweimal oder dupliziert das Fenster an einer bestimmten Stelle, so werden für die Links bei jedem Rendern neue URLs generiert, die in unabhängige Handlungsstränge überführen. Das Duplizieren von Fenstern und das Zurückgehen in der Zeit wird somit durch eindeutige URLs unterstützt.

Erste Versuche einer Umsetzung dieses Vorhabens zeigten, das diese Umsetzung in Ruby on Rails nicht so einfach möglich ist. Dies liegt nicht an der Technik der Continuations im Allgemeinen, sondern an der Implementierung der Continuations in Ruby. Diese Thematik wird an Hand einiger Beispiele in Kapitel 4.2 diskutiert. Ein weiterer möglicher Lösungsweg wird in Kapitel 4.3.1 vorgestellt.

4.2 Continuations in Ruby

Die Programmierung mit Continuations in Ruby funktioniert auf den ersten Blick problemlos, wenn man sich an sehr einfachen Beispielen orientiert. Sobald allerdings fortgeschrittene Beispiele ins Spiel kommen, trifft man auf einige unvorhergesehene Eigenheiten der Implementierung in Ruby, die im Folgenden diskutiert werden.

Erwartetes Verhalten und erste Überraschung

Codebeispiel 9 zeigt ein einfaches Beispiel für eine Continuation, die beim Abbruch der Schleife gesichert und als Ergebnis zurückgegeben wird (Zeilen 1 bis 6).

```
1  def loop1
2    for i in 1..6
3      puts "Value of i: #{i}"
4      callcc {|continuation| return continuation} if i == 3
5    end
6  end
7
8  # Aufrufe und Ergebnisse als Kommentare
9  cont = loop1
10 # Value of i: 1
11 # Value of i: 2
12 # Value of i: 3
13 cont.call
14 # Value of i: 4
15 # Value of i: 5
16 # Value of i: 6
17 puts cont
18 # nil
```

Codebeispiel 9: Vielfach verwendetes Beispiel für Continuations. Nach [Tate, 2006a]

Der Aufruf der Schleife steht in Zeile 9, die dazugehörige Ausgabe in den darauf folgenden Zeilen. Sobald die Variable *i* in der Schleife den Wert 3 erreicht, wird die Schleife beendet und die Continuation in der Variablen *cont* gespeichert. Beim Aufruf der Continuation in Zeile 13 wird die Schleife an der vorher gesicherten Stelle fortgesetzt.

Überraschend ist jedoch, dass die Variable *cont* nach dem einmaligen Ausführen der Continuation den Wert *nil* angenommen hat, obwohl ihr explizit kein neuer Wert zugewiesen

wurde. Ein wiederholter Aufruf ist also nicht möglich. Dies ist eine Abweichung von der (theoretischen) Definition der Continuations und dem Prinzip der geringsten Überraschung.

Interessant ist, dass diese Abweichung nur bei lokalen Variablen auftritt. Sichert man eine Continuation in einer globalen Variable, so ist sie mehrmals aufrufbar. In Ruby 1.9 wird das Sichern von Continuations in globalen Variablen allerdings verweigert.

Wiederholtes Aufrufen

Die in Codebeispiel 10 gesicherte Continuation ist mehrmals bzw. beliebig oft ausführbar. Dies hängt mit dem Rückgabewert der Funktion *loop2* zusammen, die die Continuation zu jedem Funktionsende erneut zurückgibt.

```
1  def loop2
2    cont = nil
3    for i in 1..6 do
4      puts i
5      callcc {|cc|
6        cont = cc
7        return cc
8      } if i == 3
9    end
10   return cont
11 end
12 # c = loop2
13 # c.call
14 # c.call
15 # ...
```

Codebeispiel 10: Beliebig oft ausführbare Continuation.

Zustand

In den beiden Codebeispielen 9 und 10 funktionierte die Sicherung des Zustands der Variable *i* problemlos. Codebeispiel 11 zeigt ein Programm, bei dem der Zustand der Variable *a* nicht wiederhergestellt wird. Die Variable *a* wird in Zeile 2 mit dem Wert *richtig* belegt. In Zeile 4 wird die Continuation in die Variable *cc* gesichert und durch die Rückgabe von *true* wird die Verzweigung in Zeile 5 und 6 ausgeführt. Dort wird die Variable *a* mit dem Wert *falsch* überschrieben und die eben gesicherte Continuation erneut aufgerufen. Das Programm wird

dabei in Zeile 4 fortgesetzt, wo das übergebene *false* ausgewertet wird, sodass die Verzweigung nicht erneut ausgeführt wird. In Zeile 8 wird daraufhin der Wert *falsch* ausgegeben, obwohl die lokale Variable *a* eigentlich auf den Wert *richtig* zurückgesetzt hätte werden müssen.

```
1  begin
2    a = 'richtig'
3    cc = nil
4    if callcc{|cont| cc = cont; true} then
5      a = 'falsch'
6      cc.call false
7    end
8    puts a
9  end
10 # Ergebnis:
11 # falsch
```

Codebeispiel 11: Der Zustand der lokalen Variable a wird nicht mitgesichert.

Dieses Beispiel zeigt, dass Continuations in Ruby keinen Zustand sichern bzw. (lokale) Variablen nicht wiederherstellen. Die Codebeispiele 9 und 10 müssen demnach aus einem anderen Grund funktioniert haben.

In [Thomas u. a., 2005, Programming Ruby] steht als Beschreibung einer *for ... in*-Schleife, dass diese von Ruby in eine *each*-Closure umgewandelt wird. Codebeispiel 12 wird von Ruby demnach intern in Codebeispiel 13 umgewandelt. Der wesentliche Unterschied ist die Verwendung eines Codeblocks bzw. einer Lexical Closure mit einer eigenen Variablen *song*.

```
1  for song in songlist
2    song.play
3  end
```

Codebeispiel 12: Eine einfache *for*-Schleife mit externem Iterator. [Thomas u. a., 2005]

```
1  songlist.each do |song|
2    song.play
3  end
```

Codebeispiel 13: Eine *each*-Schleife mit einem internen Iterator. [Thomas u. a., 2005]

Diese Closure-lokale Variable wird von Ruby bei Continuations als einzige gesichert und wiederhergestellt. Dies ist im Verhalten der Closure und nicht durch die Funktionalität der Continuation in Ruby bedingt. Die Ruby-interne Umwandlung erklärt somit, warum die Beispiele 9 und 10 funktionieren, während Beispiel 11 fehlschlägt.

Threads

Von den erwähnten Problemen abgesehen, stößt man in Ruby on Rails auf eine weitere Limitierung der Ruby-Continuations. Die Verwendung von Continuations in Web-Application-Frameworks ist nahezu unmöglich, da die heute eingesetzten Web-Server – so auch Ruby on Rails– mehrere Threads für die Bearbeitung von Anfragen verwenden. Dabei ist nicht festgelegt, durch welchen Thread die ankommenden Anfragen bearbeitet werden. Die Auswahl ist, wenn man so will, zufällig. Codebeispiel 14 zeigt jedoch, dass die Verwendung von Continuations in Ruby auf denselben Thread begrenzt ist, in dem sie erstellt bzw. gesichert worden ist. Führt man das Beispiel 14 aus, so erhält man eine *Continuation called across threads*-Exception.

```
1 # contiunation called accross threads
2 begin
3   cc = nil
4   callcc{|cc|}
5   t = Thread.new(cc) {|cont| cont.call }
6   t.join
7 end
```

Codebeispiel 14: Ruby-Continuations sind nur innerhalb des eigenen Threads verwendbar.

Die Bindung von Continuations an einen Thread ist eine wesentliche Einschränkung bei der Verwendung von Continuations. Sie ist durch die Implementierung in Ruby begründet und nicht durch das Konzept der Continuations beschränkt (siehe [Cunningham & Cunningham, Inc., 2006]). Diese Einschränkung entsteht dadurch, dass Continuations in Ruby mittels Threads (auf C-Ebene) implementiert sind.

Continuations in der Interactive Ruby Shell (*irb*)

Die hier aufgeführten Codebeispiele stehen entweder innerhalb einer Methodendefinition oder eines abgeschlossenen *begin ... end*-Blocks. Dies ist notwendig, damit die Beispiele in *irb* korrekt laufen. Fehlt dieser umfassende Block, würde Ruby die Codezeilen zwischen der Erzeugung und dem Aufruf der Continuation nicht erneut ausführen. Dies ist eine Eigenheit des irb-Interpreters, die sonst nicht zu beobachten ist.

Zusammenfassung

Continuations sind in Ruby vorhanden bzw. implementiert. Nach [Ediger, 2007] ist diese Implementierung sehr langsam und passt sich damit der recht gemächlichen Ausführungsgeschwindigkeit des Ruby-Interpreters an. Dennoch gibt es starke Einschränkungen bei der Verwendung der Continuations, die im Wesentlichen auf die mangelhafte Implementierung in Ruby mittels Threads und auf den fehlenden Zustand von Variablen zurückzuführen sind.

Rubys Continuations sind nicht serialisierbar und müssen so während ihrer gesamten Existenzdauer im Speicher gehalten werden. Continuations sind in Ruby nicht ohne größere Umstände für den Einsatz in Web-Application-Frameworks bzw. Continuation Based Web Server geeignet. Sie werden im Wesentlichen auf die Funktion reduziert, von einer Stelle im Code zu einer anderen zu springen, ohne die Zeit bzw. den Zustand zu verändern. So gesehen könnten Rubys Continuations als besserer Ersatz für GOTOs angesehen werden.

4.2.1 Beispiel in Ruby

Um einen Eindruck davon zu vermitteln, wie ein möglicher Programmablauf mit Continuations in Ruby on Rails aussehen könnte, wird das in Codebeispiel 5 verwendete Seaside-Programm nachfolgend adaptiert (siehe Codebeispiel 15).

```
1  guess = nil
2  number = rand(50)
3  count = 0
4  puts "I␣am␣thinking␣of␣a␣number␣between␣1␣and␣50."
5  while guess != number do
6    puts "What␣is␣your␣guess?"
7    guess = gets.to_i
8    count += 1
9    puts "I␣am␣thinking␣of␣a␣bigger␣number." if guess < number
10   puts "I␣am␣thinking␣of␣a␣smaller␣number." if guess > number
11 end
12 puts "You␣got␣it␣right!␣(after␣#{count}␣trys.)"
```

Codebeispiel 15: Steuerung des Kontrollflusses in Ruby.

In Codebeispiel 15 ist ein sehr einfaches Ruby-Programm dargestellt, das in der Konsole lauffähig ist. Dieser Code könnte nahezu unverändert in einem Ruby-basierten Continuation Based Web Server eingesetzt werden.

Dazu müssten nur die Methoden *gets* und *puts* (für Lesen und Schreiben in der Konsole) durch entsprechende Continuations-basierende Methoden des Web-Application-Frameworks ersetzt bzw. neu implementiert werden. Diese Behauptung wird durch den Vergleich des Programmcodes in Seaside und Ruby belegt. Die beiden Codebeispiele weisen – abgesehen von den Eigenheiten der Programmiersprachen – eine fast vollständige Übereinstimmung auf.

Wären Continuations in Ruby korrekt implementiert, so wäre dieses Beispiel mit einem recht geringen Anpassungsaufwand in Ruby on Rails lauffähig. Der Wechsel von der Konsole zur Web-Anwendung beeinträchtigt den Programmierer nur gering, er arbeitet wie nahezu unverändert.

4.3 Verwandte Arbeiten

Trotz der mangelhaften Umsetzung von Continuations gibt es eine Hand voll zustandsverwaltender Frameworks für die Web-Entwicklung in Ruby. Diese sind i.d.R. nicht über ein Anfangsstadium hinausgekommen, liegen seit mehr als 3 Jahren brach und werden nicht mehr weiterentwickelt. Auf diese Frameworks wird im Rahmen dieser Arbeit nicht weiter eingegangen, da sie meist schlecht bis gar nicht dokumentiert und z.T. nicht ohne Weiteres lauffähig sind. Sie werden, bis auf wenige Ausnahmen, nicht produktiv eingesetzt. Ferner ist ihr Funktionsumfang weit von dem entfernt, was Seaside mittlerweile leistet, sodass sie nahezu bedeutungslos sind. Namentlich erwähnt werden sollen an dieser Stelle *IOWA*, der Vorgänger von Seaside in Ruby, und *Borges*, das eine Portierung einer frühen Seaside 2 Version zu Ruby ist.

Für verschiedene andere Programmiersprachen gibt es Web-Application-Frameworks, die die lineare Programmierung des Kontrollflusses mehr oder weniger gut umsetzen.[50] Sie verwenden oft andere Abstraktionsmittel als Continuations, weil diese in Sprachen wie Java oder Python nicht vorhanden sind.

Alternativen zu Continuations sind auf den ersten Blick Zustandsmaschinen oder Threads. In [Clark, 2008, Advanced Rails Recipes] findet sich ein Beispiel für die Verwendung einer Zustandsmaschine in Ruby on Rails. Die Funktionalität der Zustandsmaschine wird durch einen Plugin eingebunden. Die Umsetzung dieses kleinen Beispiels ist gut gelungen. Dennoch macht dieses Beispiel deutlich, dass ein gewisser Aufwand zur Konfiguration der Zustandsmaschine notwendig ist. Die Programmierung des Ablaufs kann nicht (wie bei Continuations) linear umgesetzt werden. Allein der Zustand bzw. die Zustandsübergänge werden verwaltet.

[50] Siehe dazu [Tate, 2006a] und insbesondere [Ducasse u. a., 2004].

In [Schmalhofer, 2006] wird kurz auf die Problematik einer Lösung mit Threads eingegangen, die keine konkurrenzfähige Alternative zu sein scheint. Dort wird zudem auf das *Memento*-Architekturmuster aus [Gamma u. a., 1995, Design Patterns] verwiesen. Dieses reicht für die relativ einfache Verwaltung des Zustands eines Objektes aus. Für die teils extensive Verwendung und Steuerung des Kontrollflusses in einem Web-Application-Framework wie Seaside ist es nicht besonders gut geeignet. Eine weiterführende Untersuchung in diese Richtung über den Rahmen dieser Arbeit hinaus wäre interessant und könnte neue bzw. andere Wege eröffnen.

4.3.1 Acts as continuable Plugin

Ein aus diesem Jahr stammendes Experiment einer Integration von Continuations in Ruby on Rails ist das Plugin *Acts as continuable* von Matt Freels (siehe [Freels, 2008a] und [Freels, 2008b]). Mit diesem Plugin ist es möglich, eine Controllermethode aus Rails in mehreren Teilen abzuarbeiten: Nachdem ein Stück der Methode abgearbeitet wurde, wird gerendert. Setzt der Benutzer die Anwendung durch eine passende Aktion fort, so wird die Methode direkt nach dem Rendern fortgesetzt.

Dieses experimentelle Plugin kommt einer Integration von Continuations in Ruby on Rails einen großen Schritt näher, indem es bestimmte Umwege in Kauf nimmt. Eine Lösung, im Sinne einer wesentlichen Annäherung an Seasides Fähigkeiten, ist es jedoch nicht. Dieses Plugin soll mit seinen Möglichkeiten vorgestellt und diskutiert werden, da es die Problematik von Continuations und Ruby on Rails näher thematisiert.

```
1  class ContinuationTestController < ApplicationController
2    include ActsAsContinuable
3
4    def_continued(:index) {
5      # ...
6      render :partial => 'first'
7      continue
8      # ...
9      render :partial => 'second'
10     continue
11     # ...
12     render :text => 'finished'
13   }
14 end
```

Codebeispiel 16: Beispiel für eine Controllermethode mit Continuations in Rails.

Codebeispiel 16 zeigt beispielhaft die Controllermethode *index*, die mit *Acts as continuable* erzeugt wird: In Zeile 2 wird das Plugin eingebunden, mit Hilfe dessen *def_continued*-Methode die eigentliche *index*-Methode zur Laufzeit erstellt wird. Zu Anfang der Methode kann in Zeile 5 nahezu beliebiger Code ausgeführt werden. Anschließend wird in Zeile 6 das erste Mal gerendert und direkt danach in Zeile 7 eine Continuation erstellt. An dieser Stelle wird die Methode fortgesetzt, wenn der Benutzer auf den entsprechenden Link klickt. Dieser Ablauf wiederholt sich in den Zeilen 8 bis 10. Zum Schluss wird dann in Zeile 12 das letzte Mal gerendert bevor die Methode beendet wird.

An die HTML-Links, die die *index*-Methode fortsetzen sollen und die in den zugehörigen Partial-Templates (Rendern in Zeilen 6 und 9) enthalten sind, muss als Parameter eine ID übergeben werden. Die in der Variable *@context_id* enthaltene ID ist der Continuation zugeordnet, die die *index*-Methode an der richtigen Stelle fortsetzt. Diese ID ist der „Schlüssel" für die erzeugte Continuation, der durch das Plugin automatisch generiert wird und für die Verwendung in den Templates vorgesehen ist.

Um Continuations in Ruby on Rails verwenden zu können, behilft sich der Plugin eines Umweges. Der eigentliche Code der Methode wird als Block an einen Wrapper[51] übergeben, der diesen Code innerhalb eines eigenständigen Threads ausführt. Innerhalb dieses Threads können Continuations beliebig verwendet werden. Die *Continuation called across threads*-Exception tritt nicht auf, weil die verschiedenen Threads des Webservers immer auf den gleichen Thread zugreifen, innerhalb dessen die Continuations gültig sind. Dieser „Workaround" hat allerdings den Nachteil, dass die Continuations mit Ablauf des letzten Renderns in einer Methode nicht mehr gültig sind, da sie mit dem Ende des Threads von der Speicherverwaltung gelöscht werden und somit nicht mehr lauffähig sind. Dies beeinflusst die Nutzererfahrung negativ.

Abgesehen von der recht aufwändigen Implementierung ist die Verwendung der Continuations in dem Plugin nahezu vorbildlich gekapselt bzw. verborgen. Der Entwickler muss sich nicht um deren Verwendung kümmern, sondern folgt bei der Programmierung nur dem o.g. Schema. Diese Lösung beansprucht – von den Continuations abgesehen – mehr Speicher durch den zusätzlichen Thread pro Methode. Die Informationen über die Continuations müssen bei diesem Ansatz im Hauptspeicher liegen. Ruby on Rails muss demnach so konfiguriert werden, dass der komplette *SessionStore* im Speicher gehalten wird.

Das experimentelle Plugin beschränkt die Nutzung auf einen (aktiven) Einstiegspunkt der Anwendung, da nur ein einziger Thread verwendet wird. Durch eine kleine Codeänderung für den Einsatz mehrerer Threads lässt sich das Verhalten jedoch so ändern, dass eine Methode mehrmals zum gleichen Zeitpunkt ausgeführt wird. Seaside verwendet für dieses Verhalten hingegen unterschiedliche Sessions.

[51] Ein Wrapper kapselt und verbirgt ein Objekt. Hier kapselt er einen Codeblock und fügt diesem zusätzliche Funktionalität hinzu.

Zusammenfassung

Das Plugin *Acts as continuable* umgeht die Beschränkung der Continuations in Ruby durch den „einfachen“ Trick, die Continuations innerhalb eines neuen/eigenen Threads zu verwenden. Dieses Experiment zeigt beispielhaft, wie Continuations in Ruby on Rails verwendet werden können, weist aber noch einige ungelöste Probleme auf.[52] Der bedeutendste Nachteil ist die fehlende Speicherung des Zustands durch den Plugin bzw. die Continuations.[53] Eine Implementierung eines Zählers mit Zustand (wie in Seaside) ist somit nicht umsetzbar, ohne den Zustand des Zählers aufwändig, manuell und nicht orthogonal (etwa in einer Datenbank) zu sichern.

4.4 Auswertung des Ergebnisses

Eine gute Integration von Continuations in Ruby on Rails ist momentan nicht umsetzbar. Trotz mehrerer Integrationsversuche scheitert die Umsetzung letzendlich an der technischen Implementierung der Continuations in Ruby. Um dennoch beispielhaft zu zeigen, wie die Verwendung von Continuations in Ruby on Rails aussehen kann, wird im Folgenden eine Implementierung des Zähler-Beispiels mit dem *Acts as continuable*-Plugin vorgestellt.

Codebeispiel 17 zeigt den Anwendungscontroller mit seiner *index*-Methode. Nach der Initialisierung beginnt eine Endlosschleife, die in Zeile 9 jeweils nach dem Rendern durch eine Nutzerinteraktion fortgesetzt werden kann. In Zeile 11 wird daraufhin der übergebene Parameter *do* ausgewertet, der die eigentliche Aktion angibt. Entsprechend des Werts von *do* wird der Zähler erhöht (Zeile 12 bis 13) oder erniedrigt (Zeile 14 bis 15).

Der Zähler ist in diesem Beispiel der Einfachheit halber als bloße Zahl und nicht als Klassenobjekt dargestellt.

[52] Als Beispiel ist die Verwaltung der Lebensdauer der Continuations zu nennen. Diese ist an den (aktiven) Thread gekoppelt. Sinnvoller wäre eine zeitabhängige Lebensdauer, z.B. für wenige Minuten.

[53] Im Vergleich zu dem Rails-Plugin sichern Seasides Continuations den Zustand lokaler Variablen. Für die Sicherung des kompletten Zustands eines Objektes muss eine kurze Methode geschrieben werden, die die zu sichernden Variablen an das Seaside-Framework übergibt. Das Framework verwaltet den Zustand dann vollautomatisch.

```ruby
class ContinuationCounterController < ApplicationController
  include ActsAsContinuable

  def_continued(:index) do
    @count = 0

    loop do
      render
      continue

      case params[:do]
        when "increase"
            @count = @count + 1
        when "decrease"
            @count = @count - 1
      end
    end
  end
end
```

Codebeispiel 17: Beispiel für einen Zähler mit Continuations in Rails.

Das zu diesem Controller passende View-Template ist in Codebeispiel 18 dargestellt. Neben dem Zugriff auf die Zähler-Variable *@count* in Zeile 2 finden sich ab Zeile 3 die beiden Links für die Aktionen wieder. Diese zeigen auf die Index-Methode, übergeben ihr die *@context_id* für die entsprechende Continuation und die gewünschte Aktion mittels *do*-Parameter.

```erb
<h2>ContinuationCounter</h2>
<h1><%= @count.to_s %></h1>
<%= link_to("++", :action => :index,
            :context_id => @context_id, :do => :increase) %>
<%= link_to("--", :action => :index,
            :context_id => @context_id, :do => :decrease) %>
```

Codebeispiel 18: View-Template für den Zähler in Rails.

Abbildung 20 zeigt die Darstellung des Templates im Browser. Die URL dieser Seite ist mit den übergebenen Parametern in der Überschrift zu sehen. Unten im Bild ist der Link für die ---Aktion eingeblendet. Er unterscheidet sich von dem *++*-Link nur in seinem *do*-Parameter, nicht aber in seiner Continuation (siehe auch Codebeispiel 18).

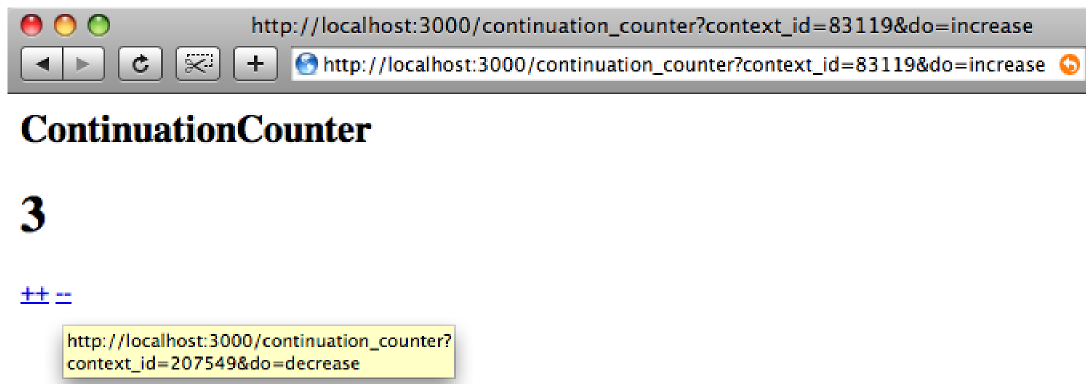

Abbildung 20: Zähler mit Continations in Rails. Die context_id ist in den Parametern der URLs zu sehen.

Bewertung

Acts as continuable verfolgt bei den Continuations weder den in Seaside verwendeten, noch den in der Planung (siehe Kapitel 4.1) vorgestellten Ansatz. Alle auf einer Seite vorhandenen Links verwenden dagegen die gleichen Continuation. Dies erzwingt die Übergabe der eigentlichen Aktion mit einem gesonderten Parameter (hier: *do*), ohne den nicht klar wäre, welche Aktion ausgeführt werden soll. Der Nachteil dieses Parameters ist jedoch, dass der Zähler beim Aktualisieren (und auch beim Duplizieren) der Seite erneut verändert wird.

Obwohl dieses Beispiel funktionsfähig ist, implementiert es keine Zustandsverwaltung, sodass das Duplizieren und Backtracking nicht die erwarteten Resultate erzeugt.[54] Für eine besser verwendbare Lösung müsste – wie in Kapitel 4.1 beschrieben – für jeden Link eine gesonderte URL (bzw. Continuation-ID) vergeben werden. Die Aktion wäre dann nicht direkt in die URL integriert und würde beim Aktualisieren der Seite nicht erneut ausgeführt werden.

Für den Fall, dass Ruby eine voll funktionsfähige Implementierung von Continuations hätte bzw. bekommen würde, wäre eine Programmierung von Web-Anwendungen im Stil von Seaside möglich. Im Fall von Ruby on Rails würde diese durch die zu Grunde liegende Philosophie von der Implementierung von Seaside abweichen und einen eigenen Weg verfolgen,

[54] Siehe dazu auch Erklärung des Zustands in Kapitel 2.4.3

bei dem aber ebenfalls von den Vorzügen der Continuations Gebrauch gemacht werden kann. Durch entsprechende Veränderungen der Architektur von Rails könnte man Seasides Eleganz ein großes Stück näher kommen, aber größtenteils an den Konzepten von Rails festhalten.

Grundsätzlich ist eine Kombination des Web-Application-Framework Ruby on Rails mit Continuations denkbar und sinnvoll, obwohl sie momentan leider nicht umsetzbar ist. Die Erfahrungen dieser Arbeit haben ebenfalls gezeigt, dass die Umsetzung von Callbacks, wie sie in Seaside existieren, nicht möglich ist. Über Umwege ließe sich eine ähnliche Funktionalität programmieren, die allerdings in der View nicht so gut integriert wäre wie Seasides Callbacks. Auf Grund der in diesem Kapitel beschriebenen Einschränkungen der Continuations wurde auf eine Implementierung der in Kapitel 4.1 beschriebenen Ablaufplanung verzichtet.

5 Zusammenfassung

In den vorangegangenen Kapiteln wurden zwei Web-Application-Frameworks vorgestellt und ihre unterschiedlichen Konzepte diskutiert. Dabei wurde insbesondere auf die Verwendung von Continuations zur Zustandsverwaltung eingegangen. Des Weiteren wurde aufgezeigt, warum die Integration von Continuations in Ruby on Rails nicht möglich ist.

In diesem Kapitel soll abschließend bewertet und zusammengefasst werden, warum Continuations ein wichtiges Werkzeug (der Metaprogrammierung) für Web-Anwendungen sind.

5.1 Bewertung

An den Beispielen von Ruby on Rails und Seaside wurden zwei verschiedene Konzepte für Web-Application-Frameworks vorgestellt. Deren unterschiedliche Umsetzungen wurden in Kapitel 3 gegenübergestellt. Bei der automatisierten Zustandsverwaltung in Seaside spielen Continuations eine große Rolle. Sie machen es möglich, die Zeit der Anwendung ähnlich wie bei einer Zeitmaschine zurückzudrehen, mehrere Sitzungen parallel zu öffnen und darüber hinaus von der Zustandslosigkeit des HTTP-Protokolls zu abstrahieren. Diese Abstraktion vereinfacht die Entwicklung von Web-Anwendungen enorm, wie einige Beispiele angedeutet haben.

Das Fachgebiet Zustandsverwaltung wird von klassischen Web-Application-Frameworks wie Ruby on Rails i.d.R. außen vor gelassen, sodass der Einsatz von Continuations orthogonal ist. Seaside demonstriert den verborgenen Einsatz von Continuations vorbildlich. Der Entwickler muss sich nahezu keine Gedanken über die Programmierung mit Continuations machen.[55]

Dem großen Nutzen durch Continuations – als mächtigem Kontrollkonstrukt der Metaprogrammierung – steht ein erhöhter Speicherverbrauch gegenüber, der für die Sicherung der verschiedenen Zustände notwendig ist. Da Speicher im Vergleich zu den Entwicklungskosten günstig ist, sollte dies niemand von der Verwendung von Continuations abhalten.

[55] Einzig die transaktionale Abarbeitung von bestimmten Aktionsfolgen sollte mit dem *isolate:*-Befehl umgesetzt werden. So können unerwünschte Änderungen an Objekten verhindert bzw. Entscheidungen unwiderruflich festgehalten werden. Des Weiteren wird der Zustand zwischen Modell und View synchron gehalten.

Auf der Suche nach gleichwertigen Ersatzlösungen für Continuations wird deutlich, wie groß deren Mächtigkeit ist und welcher Aufwand getrieben werden muss, um eine so intuitive Zustandsverwaltung und Kontrollflusssteuerung wie in Seaside zu erhalten. Dennoch kann eine falsche Verwendung von Continuations Gefahren mit sich bringen. In [Belapurkar, 2004] wird erwähnt, dass Continuations (für den Druchschnittsprogrammierer) nicht intuitiv verwendbar sind. Es besteht ebenso die Möglichkeit mit Continuations eine Art GOTO-ähnlichen (Spaghetti-)Code zu schreiben. Diese Gefahr wird durch die Integration der Continuations in Seaside gebannt. Der Programmierer beschäftigt sich nicht direkt mit Continuations, er verwaltet sie nicht manuell, sondern lässt den Zustand durch das Framework verwalten. Er kommt also nur indirekt mit Continuations in Berührung.

Der Nutzen und die Bedeutung von Continuations für eine Web-Anwendung ist abhängig von deren Ausrichtung auf ein Fachthema bzw. -gebiet. Sofern die Web-Anwendung in großen Teilen per AJAX-Funktionalität implementiert ist, wird die Verwendung von Continuations wenig sinnvoll bis unmöglich sein. Dies trifft auf bestimmte Anwendungen des Web 2.0 zu. Für eine kartengestützte, interaktive Web-Anwendung mit multimedialen Inhalten und kurzen (unvorherbestimmbaren) Aktionsabläufen ergibt der Einsatz von Continuations beispielsweise kaum Sinn. Prädestiniert für die Verwendung von Continuations sind dagegen (Geschäfts-)Anwendungen mit mittleren bis längeren und vor allem komplexen Aktionsabläufen. Darunter fallen etwa Online Shops, Ticketbuchungssysteme und Portierungen klassischer Desktopanwendungen. Sogar für manches Online-Spiel können Continuations sehr hilfreich sein. Die Stärken von Continuations liegen in der Umsetzung von komplexeren Kontrollflüssen über mehrere Elemente einer Webseite, bei Formularen (z.B. bei Registrierungen) und bei Backtracking- sowie Duplikations-Funktionalitäten. In [Queinnec, 2000] wird davon berichtet, dass Continuations für eine Lernsoftware verwendet wurden, die als Web-Anwendung auf CD ausgeliefert wurde. Für den Großteil der Web-Anwendungen dürfte ein wohldosierter und nicht übertriebener Einsatz von AJAX-Funktionalität und Continuations die richtige Empfehlung sein.

Kritik an Ruby

Diese Arbeit zeigt, dass eine Integration von Continuations in Ruby on Rails auf Grund der wenig gelungenen Implementierung der Continuations in Ruby nicht (vernünftig) umsetzbar ist. An der generellen Eignung von Continuations ändert dies jedoch nichts. Continuations sind, wie bereits erwähnt, eine große Hilfe bei der Entwicklung von zustandsbehafteten Web-Anwendungen.

Neben einigen in Kapitel 2.3.1 erwähnten Kritikpunkten an der Implementierung von Ruby ist die Programmierung der Continuations mittels Threads mangelhaft umgesetzt. Zudem ist zur Zeit noch unklar, ob Continuations in der Version 2.0 von Ruby überhaupt enthalten sein

werden. Dies hängt mit der relativ komplizierten Implementierung von Continuations für die verschiedenen in der Entwicklung befindlichen Virtuellen Maschinen für Ruby zusammen.

5.2 Fazit

Die Entwicklung von Web-Anwendungen brachte Probleme bei der Konstruktion des Anwendungsablaufs und der Zustandsverwaltung durch die Trennung zwischen Clients und Servern mit sich. Die Progammierung wird so umständlich, lästig und ähnelt oft einem GOTO-ähnlichen Programmierstil (siehe [Ducasse u. a., 2004]). Die Komponenten sind durch ihre enge Kopplung oft voneinander abhängig, sodass ihre Wiederverwendung erschwert wird. Durch die Kombination eines objektorientierten Web-Application-Frameworks mit der Mächtigkeit der Continuations können diese Probleme (durch Abstraktion) gelöst werden. Ruby on Rails ist ein Beispiel für ein gelungenes objektorientiertes Web-Application-Framework, das den konventionellen, klassischen Weg geht, während Seaside die Objektorientierung mit (verborgenen) Continuations kombiniert. Viele der dadurch erreichten Vorteile wurden in dieser Arbeit vorgestellt und diskutiert.

Continuations sind ein sinnvolles Konstrukt, um die Programmierung von Web-Anwendung wesentlich zu vereinfachen. Die Anfangs erwähnte These aus [Queinnec, 2003], welche besagt, dass der Einsatz von Continuation Based Web Servern der klassischen seitenzentrierten Web-Entwicklung überlegen ist, wurde in dieser Arbeit dargelegt und bestätigt. C. Queinnec fasst die Vorteile der Continuations zusammen und plädiert für deren Implementierung in web-orientierten Programmiersprachen:

„1. A Web application should be written as a single program in direct style and not as a series of separate pages where the state and the continuation have to be explicitly encoded. [...]
2. Continuations are a useful concept to build sophisticated interactions and web languages should offer them.“ – [Queinnec, 2003]

5.3 Ausblick

Continuations können für Web-Anwendungen von wesentlichem Nutzen sein. Eine Implementierung in den für die Web-Entwicklung verwendeten Programmiersprachen ist daher wünschenswert.

Continuation Based Web Server gab es erstmals Mitte der neunziger Jahre in Lisp. Sie werden seit wenigen Jahren produktiv eingesetzt. Mittlerweile ist Seaside der verbreitetste Continuation Based Web Server. Die Web-Programmierung mit Continuations ist noch nicht

besonders verbreitet. Dies liegt u.a. daran, dass die marktdominierenden Programmiersprachen dieses Sprachkonstrukt nicht anbieten. Teilweise sind nicht einmal Lexical Closures vollständig implementiert.

Das Gebiet der Continuation Based Web Server wächst langsam aber stetig und bietet in den nächsten Jahren noch einiges Potential. Ein besonders schnelles Wachstum ist jedoch nicht zu erwarten, da Continuation Based Web Server nicht grundsätzlich für alle Arten von Web-Anwendungen geeignet sind. Ferner bieten weit verbreitete Programmiersprachen die dazu nötigen Werkzeuge nicht an, sodass das Wachstum nur in anderen Gebieten des Marktes erfolgen kann.

Appell an Ruby-Entwickler

Für die Web-Entwicklung mit Continuations scheinen zur Zeit die Programmiersprachen Lisp bzw. Scheme und einige Smalltalk-Dialekte – darunter auch Squeak – am besten geeignet zu sein. Ruby hat hier, wie auch an anderen Stellen, noch Nachholbedarf. Mit Rubys Version 2.0 sind einige Verbesserungen der Sprache zu erwarten. Etwa die Migration von Green zu Native Threads, die die Nutzung von Mehrprozessorsystemen und den Input/Output zwischen Prozessen verbessert. Ferner wäre eine Sprachdefinition jenseits des vorhandenen C-Codes wünschenswert.

Ein wichtiger, an das o.g. Zitat von C. Queinnec angelehnter, Schritt wäre die Neu-Implementierung von Continuations in Ruby.[56] Dabei sollte unbedingt von der Verwendung von Threads abgesehen werden. Durch eine performante Implementierung von Continuations wäre Ruby eine geeignete Sprache für die Entwicklung von Continuation Based Web Servern. Von Continuations würde das beliebte Web-Application-Framework Ruby on Rails ebenfalls profitieren und könnte folglich als Continuation Based Web Server eingesetzt werden. Neben Verbesserungen bei der Zustandsverwaltung in Ruby on Rails wäre auch eine verbesserte Steuerung des Ablaufs zu erwarten.

[56] Zur Zeit wird sogar darüber spekuliert, ob Continuations überhaupt in Ruby 2.0 enthalten sein werden. Damit würde sich die Ruby- und die Rails-Entwicklergemeinde eines wichtigen und stetig an Bedeutung gewinnenden Kontrollkonstruktes berauben.

Literaturverzeichnis

[Beck und Andres 2005] BECK, Kent ; ANDRES, Cynthia: *Extreme Programming explained*. Addison Wesley, 2005. – ISBN 0-321-27865-8

[Belapurkar 2004] BELAPURKAR, Abhijit: *Crossing Borders: Web development strategies in dynamically typed languages*. Dec 2004. – URL http://www.ibm.com/developerworks/web/library/j-contin.html?S_TACT=105AGX08&S_CMP=EDU. – Zugriffsdatum: 09.06.2008

[Black u. a. 2008] BLACK, Andrew P. ; DUCASSE, Stéphane ; NIERSTRASZ, Oscar ; POLLET, Damien: *Squeak by Example*. Square Bracket Publishing, 2008. – URL http://www.iam.unibe.ch/~scg/SBE/SBE.pdf. – Zugriffsdatum: 09.06.2008. – ISBN 978-3-9523-3410-2

[Bryant 2004] BRYANT, Avi: *Where are the templates?* 2004. – URL http://www.cincomsmalltalk.com/userblogs/avi/blogView?showComments=true&entry=3257728961. – Zugriffsdatum: 06.06.2008

[Bryant und Ducasse 2004] BRYANT, Avi ; DUCASSE, Stéphane: *Controlling the Back Button in Seaside*. 2004. – URL http://www.iam.unibe.ch/~ducasse/Programmez/OnTheWeb/seaSideThree.pdf. – Zugriffsdatum: 06.06.2008

[Byrd 2002] BYRD, William E.: *Web Programming with Continuations*. November 2002. – URL http://www.double.co.nz/pdf/continuations.pdf

[Clark 2008] CLARK, Mike: *Advanced Rails Recipes*. Raleigh, NC, USA : The Pragmatic Programmers, LLC, April 2008. – ISBN 0-9787392-2-1

[Clinger u. a. 1988] CLINGER, Will ; HARTHEIMER, Anne ; OST, Eric: Implementation strategies for continuations. In: *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*. New York, NY, USA : ACM, 1988, S. 124–131. – ISBN 0-89791-273-X

[Cunningham & Cunningham, Inc. 2006] CUNNINGHAM & CUNNINGHAM, INC.: *Continuation Explanation*. 2006. – URL http://c2.com/cgi/wiki?ContinuationExplanation. – Zugriffsdatum: 09.06.2008

[Dijkstra 1968] DIJKSTRA, Edsger W.: Letters to the editor: go to statement considered harmful. In: *Commun. ACM* 11 (1968), Nr. 3, S. 147–148. – ISSN 0001-0782

[Double 2004] DOUBLE, Chris: *Partial Continuation*. 2004. – URL http://www.double.co.nz/scheme/partial-continuations/partial-continuations.html. – Zugriffsdatum: 15.06.2008

[Ducasse u. a. 2004] DUCASSE, Stéphane ; LIENHARD, Adrian ; RENGGLI, Lukas: *Seaside - A Multiple Control Flow Web Application Framework*. 2004. – URL http://www.iam.unibe.ch/~scg/Archive/Papers/Duca04eSeaside.pdf. – Zugriffsdatum: 06.06.2008

[Ducasse u. a. 2007] DUCASSE, Stéphane ; LIENHARD, Adrian ; RENGGLI, Lukas: Seaside: A Flexible Environment for Building Dynamic Web Applications. In: *IEEE Softw.* 24 (2007), Nr. 5, S. 56–63. – URL http://www.computer.org/portal/cms_docs_software/software/homepage/2007/S507/s5056.pdf. – Zugriffsdatum: 10.07.2008. – ISSN 0740-7459

[Ducasse u. a. 2008] DUCASSE, Stéphane ; RENGGLI, Lukas ; SHAFFER, David C. ; ZACCONE, Rick: *Dynamic Web Development with Seaside*. 2008. – In Vorbereitung.

[Ediger 2007] EDIGER, Brad: *Advanced Rails*. First Edition. Sebastpol, CA, USA : O'Reilly Media, Inc., December 2007. – ISBN 0-596-51032-2

[Fernandez 2008] FERNANDEZ, Obie: *The Rails Way*. Boston, MA, USA : Addison Wesley Professional/Pearson Education, 2008. – ISBN 0-321-44561-9

[Fowler 2003] FOWLER, Martin: *Patterns of Enterprise Application Architecture*. Boston, MA, USA : Pearson Education, Inc., 2003. – ISBN 0-321-12742-0

[Fowler 2005] FOWLER, Martin: *Inversion of Control*. June 2005. – URL http://martinfowler.com/bliki/InversionOfControl.html

[Freels 2008a] FREELS, Matt: *Continuations with Ruby on Rails*. 2008. – URL http://matt.freels.name/2008/4/acts-as-continuable. – Zugriffsdatum: 08.06.2008

[Freels 2008b] FREELS, Matt: *Ruby on Rails Plugin: Acts As Continuable*. 2008. – URL http://github.com/freels/acts_as_continuable. – Zugriffsdatum: 08.06.2008

[Gamma u. a. 1995] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis, IN, USA : Addison Wesley Professional/Pearson Education, 1995. – ISBN 0-210-63361-2

[Graham 1993] GRAHAM, Paul: *On Lisp, Advanced Techniques for Common Lisp*. Upper Saddle River, New Jersey : Prentice Hall, 1993. – URL http://lib.store.yahoo.net/lib/paulgraham/onlisp.pdf. – Zugriffsdatum: 04.06.2008. – ISBN 0-13-030552-9

[Graham 2001] GRAHAM, Paul: *Lisp in Web-Based Applications*. 2001. – URL http://lib.store.yahoo.net/lib/paulgraham/bbnexcerpts.txt. – Zugriffsdatum: 09.06.2008

[Graham 2004] GRAHAM, Paul: *Hackers & Painters*. First Edition. Sebastpol, CA, USA : O'Reilly Media, Inc., 2004. – ISBN 0-596-00662-4

[Graham 2005] GRAHAM, Paul: *Undergraduation*. March 2005. – URL http://www.paulgraham.com/college.html. – Zugriffsdatum: 26.07.2008

[Haynes u. a. 1984] HAYNES, Christopher T. ; FRIEDMAN, Daniel P. ; WAND, Mitchell: Continuations and coroutines. In: *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*. New York, NY, USA : ACM, 1984, S. 293–298. – ISBN 0-89791-142-3

[Hieb und Dybvig 1990] HIEB, R. ; DYBVIG, R. K.: Continuations and concurrency. In: *SIGPLAN Not.* 25 (1990), Nr. 3, S. 128–136. – ISSN 0362-1340

[Hieb u. a. 1994] HIEB, Robert ; DYBVIG, R. K. ; CLAUDE W. ANDERSON, III: Subcontinuations. In: *Lisp Symb. Comput.* 7 (1994), Nr. 1, S. 83–110. – URL http://www.cs.indiana.edu/~dyb/pubs/LaSC-7-1-pp83-110.pdf. – Zugriffsdatum: 15.06.2008. – ISSN 0892-4635

[Hunt und Thomas 2000] HUNT, Andrew ; THOMAS, David: *The Pragmatic Programmer*. Indianapolis, IN, USA : Addison Wesley Professional/Pearson Education, 2000. – ISBN 0-201-61622-X

[Leon 2007] LEON, Ramon: *Rails vs. Seaside*. 2007. – URL http://onsmalltalk.com/programming/smalltalk/rails-vs-seaside/. – Zugriffsdatum: 09.06.2008

[Lerner 2006] LERNER, Reuven M.: At the forge: assessing ruby on rails. In: *Linux J.* 2006 (2006), Nr. 142, S. 10. – ISSN 1075-3583

[Morsy und Otto 2008] MORSY, Hussein ; OTTO, Tanja: *Ruby on Rails 2*. Bonn, Germany : Galileo Computing, 2008. – URL http://www.galileocomputing.de/openbook/ruby_on_rails/. – Zugriffsdatum: 09.06.2008. – ISBN 978-3-89842-779-1

[O'Reilly 2005] O'REILLY, Tim: *What Is Web 2.0*. 2005. – URL http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html. – Zugriffsdatum: 02.07.2008

[Perscheid u. a. 2008] PERSCHEID, Michael ; TIBBE, David ; BECK, Martin ; BERGER, Stefan ; OSBURG, Peter ; EASTMAN, Jeff ; HAUPT, Michael ; HIRSCHFELD, Robert: *An Introduction to Seaside*. Software Architecture Group (Hasso-Plattner-Institut), 2008. – URL http://www.hpi.uni-potsdam.de/swa/seaside/tutorial. – Zugriffsdatum: 09.06.2008. – ISBN 978-3-00-023645-7

[Queinnec 2000] QUEINNEC, Christian: The influence of browsers on evaluators or, continuations to program web servers. In: *SIGPLAN Not.* 35 (2000), Nr. 9, S. 23–33. – ISSN 0362-1340

[Queinnec 2003] QUEINNEC, Christian: Inverting back the inversion of control or, continuations versus page-centric programming. In: *SIGPLAN Not.* 38 (2003), Nr. 2, S. 57–64. – ISSN 0362-1340

[Reynolds 1993] REYNOLDS, John C.: The discoveries of continuations. In: *Lisp Symb. Comput.* 6 (1993), Nr. 3-4, S. 233–248. – ISSN 0892-4635

[Ruby 2005] RUBY, Sam: *Continuations for Curdmudgeons*. 2005. – URL http://www.intertwingly.net/blog/2005/04/13/Continuations-for-Curmudgeons. – Zugriffsdatum: 06.06.2008

[Rustad 2005] RUSTAD, Aaron: *Crossing Borders: Ruby on Rails and J2EE: Is there room for both?* July 2005. – URL http://www.ibm.com/developerworks/web/library/wa-rubyonrails/. – Zugriffsdatum: 01.09.2008

[Schempp 2006] SCHEMPP, Ruben: *Aktuelle Entwicklungen im Bereich von Programmiersprachen*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2006. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/schempp.pdf. – Zugriffsdatum: 10.09.2006

[Schempp 2008a] SCHEMPP, Ruben: *Anwendungen II: Interaktive Karten als Rich Internet Applications*. 2008. – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master07-08-aw/schempp/bericht.pdf. – Zugriffsdatum: 01.03.2008

[Schempp 2008b] SCHEMPP, Ruben: *Projektbericht: Maps on Rails*. 2008. – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master07-08-proj/schempp/report.pdf. – Zugriffsdatum: 29.02.2008

[Schmalhofer 2006] SCHMALHOFER, Christoph: *Continuations für Webanwendungen*. Mai 2006. – URL http://www.christoph-schmalhofer.de/technik/continuations_fuer_webanwendungen.html. – Zugriffsdatum: 15.06.2008

[Schmalhofer 2007] SCHMALHOFER, Christoph: *Continuations in Spring Webflow*. November 2007. – URL http://www.christoph-schmalhofer.de/technik/spring_continuations.html. – Zugriffsdatum: 15.06.2008

[Sheehan 2007] SHEEHAN, Robert J.: Teaching operating systems with ruby. In: *SIGCSE Bull.* 39 (2007), Nr. 3, S. 38–42. – ISSN 0097-8418

[Steele 1976] STEELE, Guy L.: LAMBDA: The Ultimate Declarative. Cambridge, MA, USA : Massachusetts Institute of Technology, 1976. – Forschungsbericht. – URL ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-379.ps. – Zugriffsdatum: 17.06.2008

[Steele und Sussman 1976] STEELE, Guy L. ; SUSSMAN, Gerald J.: Lambda: The Ultimate Imperative. Cambridge, MA, USA : Massachusetts Institute of Technology, 1976. – Forschungsbericht. – URL ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-353.ps. – Zugriffsdatum: 17.06.2008

[Tate 2005a] TATE, Bruce: *Beyond Java*. First Edition. Sebastpol, CA, USA : O'Reilly Media, Inc., September 2005. – ISBN 0-596-10094-9

[Tate 2005b] TATE, Bruce: *Secrets of lightweight development success, Part 7: Java alternatives*. September 2005. – URL http://www.ibm.com/developerworks/library/os-lightweight7/

[Tate 2006a] TATE, Bruce: *Crossing Borders: Continuations, Web development, and Java Programming*. March 2006. – URL http://www-128.ibm.com/developerworks/java/library/j-cb03216/

[Tate 2006b] TATE, Bruce: *Crossing Borders: Web development strategies in dynamically typed languages*. July 2006. – URL http://www.ibm.com/developerworks/web/library/j-cb07056/index.html

[Tate 2006c] TATE, Bruce: *From Java to Ruby*. Raleigh, NC, USA : The Pragmatic Programmers, LLC, June 2006. – ISBN 0-9766940-9-3

[Thielecke 1999] THIELECKE, Hayo: Continuations, functions and jumps. In: *SIGACT News* 30 (1999), Nr. 2, S. 33–42. – ISSN 0163-5700

[Thomas u. a. 2005] THOMAS, Dave ; FOWLER, Chad ; HUNT, Andy: *Programming Ruby: The Pragmatic Programmers' Guide*. Second Edition. Raleigh, NC, USA : The Pragmatic Programmers, LLC, May 2005. – ISBN 0-9745140-5-5

[Thomas und Hannson 2006] THOMAS, Dave ; HANNSON, David H.: *Agile Web-Development with Rails*. Raleigh, NC, USA : The Pragmatic Programmers, LLC, June 2006. – ISBN 0-9766940-0-X

[Wikipedia 2008a] WIKIPEDIA: *Call-with-current-continuation*. 2008. – URL http://en.wikipedia.org/wiki/Call-with-current-continuation. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008b] WIKIPEDIA: *Continuation*. 2008. – URL http://en.wikipedia.org/wiki/Continuation. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008c] WIKIPEDIA: *Inversion of Control*. 2008. – URL http://de.wikipedia.org/wiki/Inversion_of_Control. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008d] WIKIPEDIA: *Ruby*. 2008. – URL http://de.wikipedia.org/wiki/Ruby_%28Programmiersprache%29. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008e] WIKIPEDIA: *Ruby on Rails*. 2008. – URL http://en.wikipedia.org/wiki/Ruby_on_Rails. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008f] WIKIPEDIA: *Seaside*. 2008. – URL http://en.wikipedia.org/wiki/Seaside_%28software%29. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008g] WIKIPEDIA: *Squeak*. 2008. – URL http://de.wikipedia.org/wiki/Squeak. – Zugriffsdatum: 02.07.2008

[Wikipedia 2008h] WIKIPEDIA: *Web application framework*. 2008. – URL http://en.wikipedia.org/wiki/Web_application_framework. – Zugriffsdatum: 02.07.2008

[Wikiqoute 2008a] WIKIQOUTE: *Alan Kay*. 2008. – URL http://en.wikiquote.org/wiki/Alan_Kay. – Zugriffsdatum: 05.09.2008

[Wikiqoute 2008b] WIKIQOUTE: *Lisp Programming Language*. 2008. – URL http://en.wikiquote.org/wiki/Lisp_programming_language. – Zugriffsdatum: 05.09.2008

[Wirdemann und Baustert 2007a] WIRDEMANN, Ralf ; BAUSTERT, Thomas: *Rapid Web Development mit Ruby on Rails*. 2. Auflage. München, Germany : Carl Hanser Verlag, 2007. – ISBN 978-3-466-40923-3

[Wirdemann und Baustert 2007b] WIRDEMANN, Ralf ; BAUSTERT, Thomas: *RESTful Rails Development*. 2007. – URL http://www.b-simple.de/download/restful_rails_de.pdf. – Zugriffsdatum: 06.06.2008

Glossar

Closure Siehe Lexical Closure

Continuation Ein Schnappschuss eines Programms zu einem bestimmten Zeitpunkt, an dem es jederzeit fortgesetzt werden kann.

Continuation Based Web Server / Continuation Servers Beide Begriffe bezeichnen Web-Anwendungen, deren Web Application Frameworks für die Zustandsverwaltung Continuations verwenden.

Continuation Passing Style (CPS) Aneinandergereihte Befehls- bzw. Funktionsaufrufe ohne Rückgabewert. Eine Art Vorstufe von Continuations.

Domänenspezifische Sprache (DSL) Eine i.d.R. individuell ausgerichtete Sprache, die auf ein spezielles Fachgebiet begrenzt und dort besonders ausdrucksstark ist. Es wird unterschieden zwischen internen und externen Domänenspezifischen Sprachen. Interne sind innerhalb einer Programmiersprache definiert und haben deren Syntax, externe sind eine eigenständige (kleine) Sprache und besitzen ihre eigene Syntax. (Siehe auch [Schempp, 2006].)

Feedback-Zyklus Der Feedback-Zyklus beschreibt den Zeitraum, der zwischen dem Schreiben des Codes und der Sichtung der Ergebnisse bzw. der Auswirkungen vergeht. Er gibt eine grobe Orientierung für die Entwicklungsdauer einer Anwendung in verschiedenen Programmiersprachen. Häufig besitzen dynamisch typisierte Sprachen kürzere Feedback-Zyklen, da der Kompilierungsschritt dort entfällt.

First-Classness Sagt aus, dass Programmbestandteile (z.B. Klassen, Methoden) als First-Class Objects bezeichnet werden, wenn sie ohne Einschränkungen wie normale Datenobjekte verwendet werden können. Objekte erster Klasse können z.B. als Parameter weitergereicht oder in Datenstrukturen gespeichert werden. Gegenbeispiel: Klassen und Methoden in Java.

Framework Rahmenwerk, das eine bestimmte, wiederverwendbare Funktionalität zur Entwicklung von Anwendungen bereitstellt.

Inversion der Kontrolle Übergabe der Steuerung der Kontrolle an ein Framework. Es ist i.d.R. erwünscht, dass bestimmte eigene Methoden durch das Framework aufgerufen werden. Bei Web-Application-Frameworks ist die Inversion der Kontrolle oft unerwünscht und durch das Kommunikationsprotokoll HTTP erzwungen.

Lexical Closure Ein Stück Code (auch: Codeblock) mit lexikalischer Bindung von freien Variablen an den Erzeugungskontext. Es kann gespeichert, (meist als Argument an einen Methode) weitergegeben und (zeitversetzt) ausgeführt werden.

Metaprogrammierung Eine abstrakte Programmiertechnik, die meist in dynamisch typisierten Programmiersprachen verwendet wird. Metaprogrammierung ist die 'Programmierung der Programmierung'. Sie beschreibt die Veränderung bzw. Programmierung der verfügbaren Sprachmittel und Spracheigenschaften.

Model-View-Controller (MVC) Ein Architekturmuster für Anwendungen mit grafischer Benutzeroberfläche, das das Modell (Geschäftsobjekte) von der Darstellung (View) und dem Controller (Steuerung) trennt bzw. deren Kopplungsgrad minimiert.

Rendern Die Aufbereitung für die Anzeige in der Darstellungsschicht (MVC). Aus dem Englischen von *to render* abgeleitet.

Ruby Freie, dynamisch typisierte, objektorientierte und interpretierte Skript- bzw. Programmiersprache. Erschienen im Jahr 1995. Entwickelt von Yukihiro Matsumoto.

Ruby on Rails Populäres, freies und objektorientiertes Web-Application-Framework in Ruby. Setzt traditionelle Techniken ein. Erschienen im Jahr 2005.

Seaside Freies Web-Application-Framework in Smalltalk. Wird hauptsächlich in Squeak entwickelt. Verfügt über eine integrierte Zustandsverwaltung und innovative Konzepte. Entwickelt von Avy Bryant und erschienen im Jahr 2002.

Session Benutzersitzung in einer Web-Anwendung, die Zustandsinformationen beinhaltet.

Smalltalk Dynamisch typisierte, objektorientierte Programmiersprache. Entwickelt in den siebziger Jahren am Forschungszentrum *Xerox PARC*. Integrierte Entwicklungsumgebung mit Image-Konzept und Virtueller Maschine.

Squeak Freier Smalltalk-Dialekt, in dem Seaside entwickelt wird.

Virtuelle Maschine Eine in Software implementierte abstrakte Maschine.

Web-Application-Framework Framework, das auf die Entwicklung von Web-Anwendungen spezialisiert ist.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. Oktober 2008

Ort, Datum

Unterschrift