# Monitoring von Perl-basierten Webanwendungen mittels Kieker

Bachelorarbeit

Nis Börge Wechselberg

22. April 2013

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Software Engineering

Betreut durch: Prof. Dr. Wilhelm Hasselbring
Dipl.-Inf. Peer Brauer

**Eidesstattliche Erklärung**

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

# Abstract

Die Bedeutung von Webanwendungen hat in den letzten Jahren stetig zugenommen. Um diese Anwendungen während der Laufzeit zu überwachen, muss ein Monitoringverfahren für diese Softwaresysteme etabliert werden. Das Kieker Framework stellt ein effizientes System zum Monitoring von Java-Anwendungen dar.

In dieser Arbeit wird eine Erweiterung des Frameworks zum Monitoring von Perl-Anwendungen vorgestellt. Für diese Software wird die entwickelte Klassenstruktur vorgestellt und mögliche alternative Implementierungen diskutiert. Die Anbindung der Perl-Komponenten wird mit der Kieker-Data-Bridge realisiert, welche ebenfalls hier vorgestellt wird. Weiter wird ein manuelles und ein automatisches Instrumentierungsverfahren für objektorientiertes Perl vorgestellt.

Als erste Anwendung wird dann die Software zum Profiling von Kielprints verwendet. Hierbei werden Unterschiede im Verhalten zwischen Kielprints und unmodifiziertem EPrints aufgezeigt. Es zeigt sich bei Kielprints eine signifikante Steigerung der Laufzeiten und ein massiver Anstieg der Funktionsaufrufe gegenüber EPrints. Schließlich werden Ansätze zur Architekturerkennung für diese Anwendungen skizziert.

# Inhaltsverzeichnis


**1 Einleitung** **1**

**2 Grundlagen und Technologien** **3**
2.1 Kieker Monitoring Framework . . . 3
2.2 Die Programmiersprache Perl . . . 4
2.3 EPrints & Kielprints . . . 5
2.4 Monitoring . . . 5

**3 Umsetzung der Perl-Module** **9**
3.1 Möglichkeiten der Integration . . . 9
3.1.1 Verwendung einer Perl-Java-Brücke . . . 9
3.1.2 Entwicklung einer Client/Server-Architektur . . . 10
3.2 Implementierung . . . 11
3.2.1 Kieker . . . 11
3.2.2 Kieker::Controlling . . . 11
3.2.3 Kieker::Util . . . 12
3.2.4 Kieker::Writer . . . 12
3.2.5 Kieker::Record . . . 12
3.3 Übernahme der Daten in Kieker . . . 13

**4 Instrumentierung** **15**
4.1 Statische Integration der Probes . . . 15
4.2 Integration mit Sub::WrapPackages . . . 17

**5 Testdurchführung** **19**
5.1 Systemumgebung . . . 19
5.2 Testaufbau . . . 20
5.3 Requests . . . 20
5.3.1 Request 1: Admin-Login . . . 20
5.3.2 Request 2: Eintragen eines neuen Titels . . . 20
5.3.3 Request 3: Upload der Publikation . . . 21
5.3.4 Request 4: Eingabe der Metadaten . . . 21
5.3.5 Request 5: (AJAX) Mehr Autorenfelder . . . 21

**6 Ergebnisse** **25**
6.1 Zeitliches Verhalten . . . 25
6.2 Funktionsaufrufe . . . 26
6.3 Aufrufhäufigkeiten für Funktionen . . . 27
6.4 Aufrufhäufigkeiten für Pakete . . . 28
6.5 Paketabhängigkeiten . . . 29

**7 Zusammenfassung und Ausblick** **33**
7.1 Zusammenfassung . . . 33
7.2 Ausblick . . . 34

**A Perl Dokumentation** **35**
A.1 Kieker . . . 35
A.2 Kieker::Controlling . . . 36
A.3 Kieker::Util . . . 36
A.4 Kieker::Record::OperationEntryEvent . . . 37
A.5 Kieker::Record::OperationExitEvent . . . 38
A.6 Kieker::Record::Trace . . . 38
A.7 Kieker::Writer::FileWriter . . . 39
A.8 Kieker::Writer::JMSWriter . . . 40

**B Perl Skripte zur Auswertung** **41**
B.1 Zählen von Aufrufhäugigkeiten . . . 41
B.2 Aggregieren von Funktionsaufrufen . . . 42

**Glossar** **45**

**Bibliografie** **47**

Kapitel 1

# Einleitung

*Throughout human history, we have been dependent on machines to survive. Fate, it seems, is not without a sense of irony.*
— Morpheus, *The Matrix*

Seit einigen Jahren nehmen Internetdienste eine immer wichtigere Position ein. Mit fortschreitender Zeit wachsen diese Dienste und müssen häufig mit wachsender Komplexität fertig werden. Hier müssen zum Beispiel steigende Benutzerzahlen, wachsende Datenbanken oder Ergänzungen neuer Funktionen berücksichtigt werden. Kann die Anwendung nicht mir diesen Herausforderungen umgehen, treten häufig Performanceprobleme auf und es besteht schnell Bedarf an einer Möglichkeit das Verhalten der Anwendung zur Laufzeit zu beobachten und zu analysieren.

Das Kieker Monitoring Framework wird seit einiger Zeit an der Universität Kiel entwickelt und stellt ein mächtiges System zur Überwachung und Analyse von diversen Softwaresystemen dar. Dieses Programm ist in Java programmiert und unterstützt zur Zeit, mit Java, .NET, VB6 und COBOL, nur eine eingeschränkte Anzahl von Programmiersprachen. Mit diesem Tool sollte nun versucht werden in der Publikationsdatenbank Kielprints der Universität Performance-Probleme, die im Betrieb aufgetreten sind, zu lokalisieren. Diese Anwendung besteht aus einer modifizierte Version der quelloffenen Software EPrints der University of Southampton, welche am GEOMAR | Helmholtz-Zentrum für Ozeanforschung Kiel angepasst wurde und von dem dortigen Datenmanagement-Team betrieben wird. Große Teile der Anwendung wurden in der Programmiersprache Perl geschrieben, für die zum Zeitpunkt der Arbeit noch keine Schnittstelle im Kieker Framework vorhanden war. Somit wurde es nötig diese Schnittstelle zu realisieren um anschließend das Kielprints-System untersuchen zu können.

Im Rahmen der Arbeit werden die benötigten Perl-Module für das Monitoring sowie eine Anbindung dieser Komponenten an das bestehende Kieker Framework realisiert. Hierfür wird neben den Messmethoden in Perl auch eine Brücke in die Java-Anwendung Kieker benötigt. Anschließend werden diese neuen Komponenten auf das Kielprints-System angewendet, also eine Instrumentierung der Anwendung durchgeführt. Mit diesem instrumentierten Kielprints werden dann Tests durchgeführt, um Ursachen für die Performance-Probleme aufzudecken.

Im folgenden erkläre ich in Kapitel 2 die benötigten Grundlagen und stelle die verwendeten Technologien vor. Hierauf aufbauend stelle ich dann in Kapitel 3 die von mir realisierten Softwarekomponenten vor und beschreibe anschließend die Anwendung auf das beschriebene Problem in Kapitel 4. In Kapitel 5 wird dann das Testverfahren präsentiert und die erhaltenen Ergebnisse in Kapitel 6 angegeben und analysiert. Zum Abschluss werde ich in Kapitel 7 noch eine Zusammenfassung und mögliche Weiterführungen der Arbeit präsentieren.

Kapitel 2

# Grundlagen und Technologien

*Perhaps we are asking the wrong questions.*
— Agent Brown, *The Matrix*

Im folgenden werden die verwendete Technologien vorgestellt und die benötigten Grundlagen besprochen. Dies umfasst neben einer kurzen Vorstellung des Kieker Monitoring Frameworks in Abschnitt 2.1 sowie der Programmiersprache Perl in Abschnitt 2.2 auch eine Beschreibung des EPrints-System im allgemeinen und der angepassten Variante Kielprints in Abschnitt 2.3. Schließlich wird dann in Abschnitt 2.4 eine Einführung in das Monitoring gegeben.

## 2.1. Kieker Monitoring Framework

Kieker[1] ist ein System zum Monitoring und zur Analyse des Laufzeit-Verhaltens einer Software. Es wird seit 2006 in der Arbeitsgruppe Software Engineering an der CAU Kiel entwickelt. Frühere Versionen von Kieker waren primär auf das Monitoring von Java-Anwendungen ausgerichtet, doch wurden bereits ergänzende Module entwickelt oder befinden sich zur Zeit in der Entwicklung um zum Beispiel .NET oder COBOL mit Kieker analysieren zu können. Seit 2011 wird Kieker von der SPEC Research Group als empfohlenes Tool im SPEC RG Software Verzeichnis[2] angeboten.

Wie in Abbildung 2.1 dargestellt besteht Kieker aus den zwei zentralen Komponenten Kieker.Monitoring und Kieker.Analysis. Die Monitoring-Komponente stellt Klassen und Methoden zum Instrumentieren und Überwachen der Software zur Verfügung. Hierzu werden von *Monitoring Probes* Messungen vorgenommen und hierzu *Monitoring Records* erstellt. Diese *Monitoring Records* werden dann an einen *Monitoring Writer* übergeben, der die Informationen in eine Log-Datei schreibt oder mit einem Stream an die Analysis-Komponente übergibt. Zur Instrumentierung des Systems stehen manuelle Methoden, also das statische Einfügen der *Monitoring Probes* in den Quelltext, aber auch automatische Mechanismen zur Verfügung, welche zum Beispiel mittels *AspectJ* die Programmstruktur analysieren und automatisch die benötigten Messpunkte erzeugen [Kieker Project 2013; van Hoorn u. a. 2012; 2009].

[1] `http://www.kieker-monitoring.net` (Zuletzt aufgerufen 2013-03-19)
[2] `http://research.spec.org/projects/tools.html` (Zuletzt aufgerufen 2013-03-20)

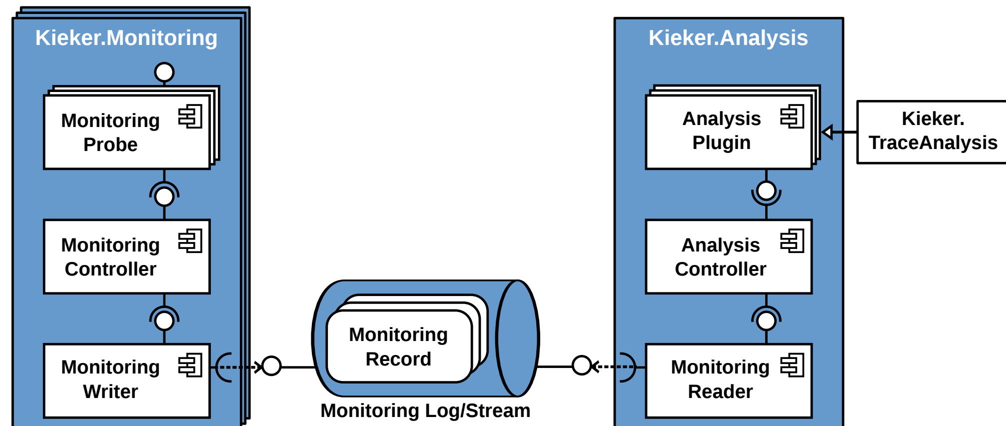


**Abbildung 2.1.** Kieker Komponentendiagramm [Kieker Project 2013]

Die Analyse der erhaltenen Daten wird mittels Kieker.Analysis durchgeführt. Diese Komponente liest die *Monitoring Records* ein und stellt einen konfigurierbaren Ablauf zum Filtern, Analysieren und Visualisieren der Daten bereit. Hierbei können zum Beispiel Sequenzdiagramme oder Abhängigkeitsbäume erzeugt werden um das Verhalten der Anwendung zu beschreiben [Kieker Project 2013; van Hoorn u. a. 2012; 2009].

Die Kieker.Monitoring-Komponente wird in dieser Arbeit mit der parallel zu dieser Arbeit neu entwickelten Kieker-Data-Bridge erweitert um die Schnittstelle zwischen den Programmiersprachen Java und Perl zu überbrücken. Die Perl-Module sowie die Kieker-Data-Bridge werden in Kapitel 3 ausführlicher erläutert.

## 2.2. Die Programmiersprache Perl

Die Programmiersprache Perl[3] ist eine imperative, plattformunabhängige, interpretierte Programmiersprache. Die erste Version wurde 1987 vorgestellt und hat seine Wurzeln primär in C und awk [*The Timeline of Perl and its Culture*]. Die aktuelle Version ist Perl 5.16, die im Mai 2012 veröffentlicht wurde. Perl ist eine interpretierte Programmiersprache, also werden die Programme nicht zu einer nativen Anwendung übersetzt sondern mit einem Interpreter ausgeführt. Der Perl-Interpreter ist für alle relevanten Betriebssysteme verfügbar und in vielen Systemen bereits integriert. Neben der Verwendung des Interpreters im Betriebssystems wird der Interpreter auch häufig in Webserver eingebettet [*mod_perl Documentation*].

In Perl werden verschiedenste Programmierparadigmen umgesetzt, allerdings ist es dem Programmierer freigestellt, welche dieser Möglichkeiten er umsetzen will. So ist zum

[3] http://www.perl.org (Zuletzt aufgerufen 2013-03-20)

Beispiel die Objektorientierung ein Teil der Sprache, allerdings nicht wie in Java erzwungen sondern mit verschiedenen Modulen optional verfügbar. Auch stellt Perl häufig mehrere Optionen für die selben Probleme bereit, so können häufig Befehle verkürzt werden oder es stehen mehrere äquivalente Befehle zur Verfügung. Dies führt allerdings sehr schnell zu schlechter Lesbarkeit der Programme, da Perl auch eine sehr freie Syntax ermöglicht. Zeilenumbrüche und Leerzeichen sind weitgehend bedeutungslos und führen zu einem sehr unterschiedlichen Code-Format und sehr persönlichen Gestaltungen der Programme [*perlsyn - Perl syntax*].

Perl verwendet keine expliziten Typen und verfügt nur über die Datenstrukturen Skalar, Array bzw. Liste sowie Hashes [*perldata - Perl data types*]. Wird mit Perl objektorientiert programmiert, so werden normale Variablen mit einem Attribut versehen, das die Klasse definiert (*blessed variables*). Diese Variablen verfügen intern über ein Hash zum Speichern ihrer Attribute und können auf Methoden der Klasse zugreifen [*perlobj - Perl object reference*].

## 2.3. EPrints & Kielprints

Das System EPrints ist eine quelloffene Webplattform zur Veröffentlichung von Publikationen, Forschungsberichten oder anderen Dokumenten. Die Plattform implementiert dabei das OAI-PMH[4] Protokoll und ist somit ein Projekt zur Umsetzung von Open Access. EPrints wurde initial von der University of Southampton entwickelt und unter der [*GNU General Public License, Version 3*] lizenziert. Die Plattform wurde in Perl implementiert und steht zur Zeit in der Version 3.3 zum Download[5] bereit.

An der Universität Kiel wird die Software Kielprints verwendet, was eine angepasste Version von EPrints, basierend auf EPrints 3.2., ist. Betrieben wird diese Plattform durch das Kieler Datenmanagement Team am GEOMAR | Helmholtz-Zentrum für Ozeanforschung Kiel. Das Deployment der Anwendung ist in Abbildung 2.2 grob skizziert.

Wie bereits angesprochen sind im laufenden Betrieb einige Performance-Probleme aufgetreten, speziell bei einigen Seiten zur Administration. Hierbei treten mitunter Wartezeiten von über 10 Sekunden auf, obwohl sich die Datenbank mit 15050 Publikationen und 1090 Autoren[6] noch in üblichen Größenordnungen bewegt.

## 2.4. Monitoring

Beim Monitoring werden während der Ausführung einer Anwendung Messdaten erzeugt und anschließend anhand dieser Messdaten das Verhalten der Anwendung analysiert. Aufgrund der Art der Datenerfassung wird diese Analyse auch dynamische Analyse genannt. Das Verfahren steht als ergänzendes Mittel zur statischen Analyse zur Verfügung,

[4] Open Archives Initiative Protocol for Metadata Harvesting
[5] `http://www.eprints.org/software/` (Zuletzt aufgerufen 2013-03-20)
[6] Stand: 2013-03-20, 12:50

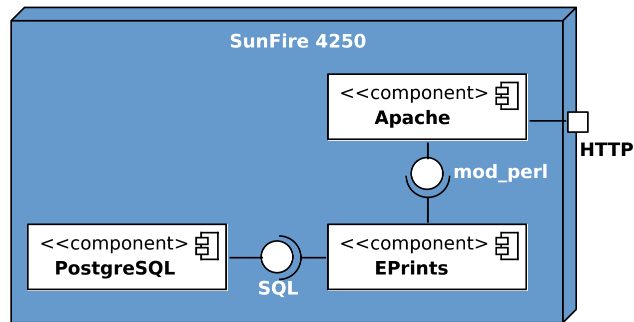


**Abbildung 2.2.** Deployment der Kielprints-Anwendung

bei welchem die Software anhand ihres Quelltextes analysiert wird. Allerdings ist die statische Analyse für umfangreichere Softwaresysteme nur schwierig durchführbar, da hier zum Beispiel alle theoretisch möglichen Programmpfade ausgewertet werden. Mit der dynamischen Analyse können die tatsächlich ausgeführten Programmpfade ermittelt werden und direkt das Verhalten der Anwendung nachvollzogen werden [vgl. auch Schmalenbach 2007]. Mit verschiedenen Techniken können dann auch bei Bedarf nur Teile der Anwendung beobachtet werden und der Fokus auf bestimmte Programmteile gelegt werden. Dies kann einerseits durch verschiedene Instrumentierungstechniken, andererseits durch Filterung der erhaltenen Daten erreicht werden.

Zur Durchführung des Monitorings wird der Quelltext des Programms mit *Monitoring Probes* versehen, welche verschiedene Ereignisse im Programmablauf protokollieren. Hierbei sind häufig Verzweigungen, Sprünge oder Funktionsaufrufe besonders von Interesse [Richter 2012, S. 38], aber auch andere Ereignisse wie Anfragen an Datenbanken können protokolliert werden.

Mit den erhaltenen Messdaten wird eine Analyse und Visualisierung durchgeführt um das Verhalten der Anwendung zu dokumentieren. Hierbei werden zum Beispiel UML Sequenzdiagramme erstellt oder das zeitliche Verhalten der Anwendung detailliert untersucht. Auch kann ein Abhängigkeitsgraph erstellt werden, der die Verschränkung der verschiedenen Komponenten zeigt, oder eine statistische Analyse durchgeführt werden [Agarwal u. a. 2004]

Eine Sonderform der dynamischen Analyse ist das *Profiling*. Hierbei wird die Analyse eingesetzt um das Verhalten eines existierenden Softwaresystems auf Funktionsebene zu erfassen. Dies wird zum Beispiel in Modernisierungsprozessen verwendet. Da Anwendungen häufig im Betrieb modifiziert und weiterentwickelt werden und hierbei mitunter die Dokumentation der Änderungen nur untergeordneten Charakter gegenüber der Funktionalität genießt, weicht häufig das dokumentierte Verhalten von dem tatsächlich beobachteten ab. Diese Differenzen können durch das *Profiling* erkannt werden und in den Modernisierungsprozess eingebunden werden [Richter 2012].

Der Schwerpunkt dieser Arbeit wird auf dem *Profiling* sowie auf dem *Performance Monitoring* liegen. Hierbei wird eine möglichst präzise Zeitmessung im Programmablauf durchgeführt und dieser Zeitcode in den *Monitoring Records* eingefügt. Somit kann nicht nur die Aufrufreihenfolge sondern auch der zeitliche Ablauf, also konkret die Dauer zur Ausführung einer Funktion und auch die Häufigkeit der Ausführung, festgestellt werden. Es können dann besonders lang und/oder häufig laufende Funktionen identifiziert werden um Ansatzpunkte für Performanceprobleme aufzuzeigen. Neben der Zeit können noch weitere Systemdaten wie CPU-Auslastung oder Speicherverbrauch protokolliert werden. Das Kieker Framework unterstützt für diesen Zweck neben Kontrollflussdaten auch *Monitoring Records* welche die Systemauslastung protokollieren oder die Systemzeit messen. Aufgrund der Erweiterbarkeit von Kieker können auch beliebige weitere *Monitoring Records* erzeugt werden.

Der Hauptunterschied zwischen *Profiling* und *Performance Monitoring* besteht darin, dass beim *Performance Monitoring* die Anwendung kontinuierlich im laufenden Betrieb untersucht wird. Beim *Profiling* werden in der Regel Testfälle ausgeführt um eine möglichst gute Abdeckung nach klassischen Testkriterien wie Pfad-, Zweig- oder Anweisungsüberdeckung zu erreichen.

Kapitel 3

# Umsetzung der Perl-Module

*Neo, sooner or later you're going to realize just as I did that there's a difference between knowing the path and walking the path.*
— Morpheus, *The Matrix*

Vor der Entwicklung von Perl-Modulen mussten zunächst einige Abwägungen zur Integrationsweise getroffen werden. Es bieten sich verschiedene Möglichkeiten zur Überbrückung der Grenze zwischen Perl und Kieker an, die einleitend in Abschnitt 3.1 diskutiert werden. Die gewählte Möglichkeit wird anschließend in Abschnitt 3.2 detaillierter vorgestellt und schließlich in Abschnitt 3.3 die Anbindung der Module an das vorhandene Kieker Framework beschrieben.

## 3.1. Möglichkeiten der Integration

Zur Verwendung des Kieker Frameworks mit neuen Programmiersprachen muss die Grenze zwischen den Sprachen überwunden werden. In früheren Arbeiten [Magedanz 2011; Richter 2012] wurden hierzu bereits Überlegungen angestellt, die zu unterschiedlichen Ergebnissen gekommen sind. Auf dieses Projekt angewendet ergeben sich primär zwei Optionen, der Einsatz einer Perl-Java-Brücke oder eine Client/Server-basierte Lösung. Die ebenfalls mögliche Neuimplementierung von Kieker in Perl wurde nicht weiter berücksichtig, da neben erheblichem Aufwand in der Portierung die Pflege zweier funktional identischer Systeme nicht anzustreben ist.

### 3.1.1. Verwendung einer Perl-Java-Brücke

Für die Anbindung von .NET Anwendungen wurde in [Magedanz 2011] eine kommerzielle .NET-Java-Bridge verwendet. Zur Zeit sind allerdings keine entsprechenden Brücken in ausreichender Stabilität für Perl verfügbar. Eine umfangreiche Implementierung, die Java-Perl Library von O'Reilly wird seit 1998 nicht mehr weiter gepflegt[Siever u. a. 1997] und mehrere Open-Source Projekte[1] haben nicht die nötige Stabilität erreicht. Das Paket Inline::Java würde einige benötigte Funktionen bereitstellen, allerdings ist auch hier, aufgrund

[1] z.B. Java::Bridge: http://blogs.perl.org/users/theorbtwo/javabridge/ (Zuletzt aufgerufen 2013-03-20)

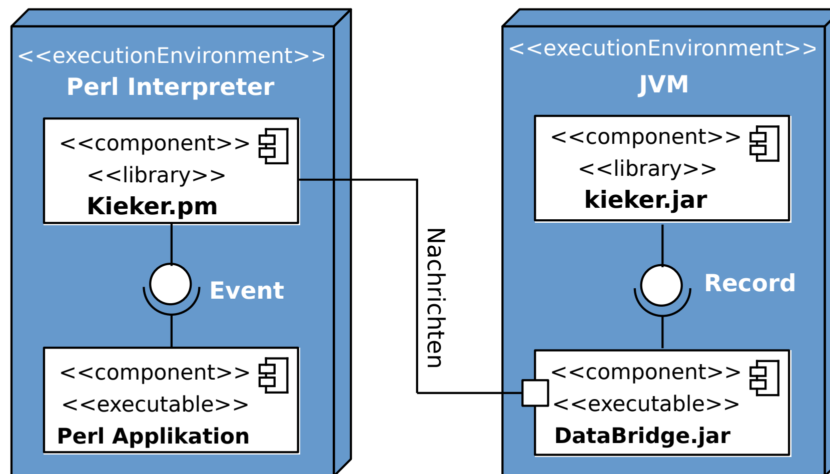


**Abbildung 3.1.** Kommunikationsmodell im Client/Server System

des hohen Alters der Bibliothek, ein problemloser Einsatz mit neueren Java Versionen zweifelhaft. Ebenso bestehen Probleme bei der Ausführung der Bibliothek im Rahmen eines Webdienstes. In Folge dieser Probleme wurde die Verwendung einer Brücke nicht weiter verfolgt.

### 3.1.2. Entwicklung einer Client/Server-Architektur

Als zweiter Ansatz ist eine Client/Server-Architektur zu verfolgen. Hierbei kommuniziert ein zu implementierender Kieker Perl-Client mit einem Java-basierten Server. Diese Prinzip ist in Abbildung 3.1 exemplarisch für diesen Fall dargestellt. Zur Kommunikation zwischen den Komponenten bieten sich verschiedene Protokolle wie TCP oder JMS an.

Es ist abzuwägen wieviele Kontrollmechanismen aus bestehenden Kieker-Komponenten übernommen werden können. Wird der bestehende Monitoring-Controller weiterverwendet müssen zusätzliche Nachrichten zur Trace-Verwaltung zwischen Client und Server versendet werden, die potentiell einen großen Einfluss auf die Laufzeit der Anwendung haben können. Andererseits steigt natürlich der Implementierungs- und Pflegeaufwand mit jedem zusätzlich neu in Perl realisierten Modul an und es können leichter Inkonsistenzen zwischen den verschiedenen Kieker-Implementierungen auftreten.

Für dieses Projekt wurde ein verhältnismäßig schwergewichtiger Perl-Client gewählt, der neben eigenen Methoden zur Zeitmessung auch die Traceverwaltung übernimmt. Die erhaltenen Daten können dann zur weiteren Aufbereitung an Kieker übergeben werden. Hierfür wird die Kieker-Data-Bridge verwendet, welche genauer in Abschnitt 3.3 beschrieben wird.

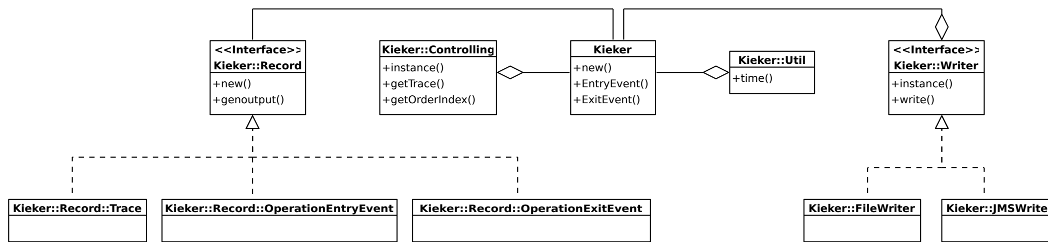


**Abbildung 3.2.** Klassendiagramm für Kieker.Perl

## 3.2. Implementierung

Für das Monitoring von Perl-Anwendungen wurde ein Perl-Modul entwickelt, welche sich um die Erzeugung von *Monitoring Records*, die Verwaltung und das Controlling sowie um die Kommunikation mit der Data-Bridge kümmert. Aufgrund der engen Zielsetzung wurde zunächst nur ein eingeschränkter Funktionsumfang des Kieker Frameworks umgesetzt. Die Module und ihre Verbindungen untereinander sind in Abbildung 3.2 zu erkennen. Die zentralen Funktionen finden sich hier in dem Modul Kieker.pm wieder. Auf die einzelnen Komponenten und die Funktionsweise wird im folgenden eingegangen. Die Dokumentation der Pakete findet sich in Anhang A.

### 3.2.1. Kieker

Das Modul Kieker stellt eine schmale Schnittstelle zur einfachen Benutzung bereit. In diesem Modul werden die Controlling- und Writing-Komponenten verwaltet und Methoden für die verschiedenen Events bereitgestellt. In diesen Methoden werden die Optionen zur Zeitmessung und zum Schreiben der *Monitoring Records* gekapselt. Somit bietet sich hier auch die Möglichkeit der Konfiguration an, um andere Writer-Module zu verwenden oder um andere Komponenten ersetzen zu können.

### 3.2.2. Kieker::Controlling

Das Controlling Modul verwaltet die Trace-Ids sowie die laufenden Zähler der Traces. Da im EPrints-Web-Kontext ein einzelner Request nur schwierig identifizierbar ist, wurde hier auf eine Notlösung zurückgegriffen und die Thread-Id des Perl-Interpreters als Trace-Id verwendet. Hierbei wird dann in der Apache-Konfiguration dafür gesorgt, dass jeder Request in einem eigenen Thread gestartet wird. Als bessere Lösung bietet es sich hier im Web-Kontext an, die angeforderte URL in Kombination mit einem Zeitstempel und eventuell einer IP-Adresse des Benutzers zu verwenden. Diese Informationen können allerdings nicht aus Umgebungsvariablen erzeugt werden, sondern erfordern z.B. ein

Apache Request Object[2], welches allerdings von EPrints blockiert wird und somit hier nicht zur Verfügung steht.

Für die Zähler der Traces wird ein Hash verwendet, in dem die aktuellen Zählerstände verwaltet werden. Wird ein Zähler zu einem unbekannten, also neu angelegten, Trace angefordert wird automatisch der entsprechende Trace-Record erzeugt und an den Writer übergeben.

### 3.2.3. Kieker::Util

Im Util Modul werden Hilfsfunktionen zur einfacheren Verwendung der Module verwaltet. Im aktuellen Status wird hier nur die Zeitmessung gekapselt. Perl stellt keine Funktionen zur Messung von Nanosekunden, sondern lediglich das Paket Time::HiRes zur Messung in einem Paar (Sekunden,Mikrosekunden) bereit. Zur Kompatibilität mit der Kieker Zeitmessung wird eine Anpassung auf ein flaches Nanosekunden-Format durchgeführt. Hierbei wird natürlich keine Steigerung der Genauigkeit erreicht sondern lediglich ein Mikrosekunden-Zeitstempel in Nanosekunden ausgegeben.

### 3.2.4. Kieker::Writer

Unter Kieker::Writer wurden zwei verschiedene Ausgabemodule erstellt. In einer frühen Phase wurde zunächst Kieker::Writer::FileWriter verwendet, der automatisch zeilenweise in eine Datei schreibt, die bei der Initialisierung des Moduls erstellt wird. Dieser FileWriter wurde primär für frühe Entwicklungen und Tests verwendet.

Der später eingesetzte Writer ist Kieker::Writer::JMSWriter. Dieses Modul baut beim Start eine Verbindung zu einem laufenden JMS-Provider auf [Zur Funktionsweise von JMS vgl. *Java Message Service Specification*]. Über diese Verbindung werden dann die *Monitoring Records* in serialisierter Form an eine Message-Queue gesendet. Zur Verbindung mit dem JMS-Provider wird das Modul Net::Stomp verwendet, das die Nachrichten in Textform überträgt. Dieses Protokoll wird noch nicht von allen JMS-Providern unterstützt, bietet aber leichte Implementierbarkeit und ist somit auch in vielen anderen Sprachen verfügbar.

### 3.2.5. Kieker::Record

Die drei zur Zeit implementierten *Monitoring Records* sind Trace, OperationEntryEvent und OperationExitEvent. Der Trace wird, wie bereits erwähnt, automatisch durch den Controller erzeugt sobald ein neuer Trace gestartet wird. Die beiden OperationEvents verfügen über die selben Attribute wie die entsprechenden Kieker Events und können nach Bedarf erzeugt werden. Zusätzlich verfügen die *Monitoring Records* noch jeweils über eine Methode zur Serialisierung der Ausgabe. Hierbei werden den Events eindeutige

[2]Dokumentation unter http://perl.apache.org/docs/2.0/api/Apache2/RequestRec.html (Zuletzt aufgerufen 2013-03-24)

Kennnummern zugewiesen, die später bei der Deserialisierung zur Identifizierung genutzt werden. Nach der Serialisierung sieht z.B. ein OperationEntryEven wie folgt aus:

```
1;1362747533540734000;6889;5;EPrints.current_repository;EPrints
```

Dieses Nachrichtenformat wurde für die weitere Verwendung mit der Kieker-Data-Bridge entworfen.

## 3.3. Übernahme der Daten in Kieker

Zur Übernahme der Monitoringdaten in das bestehende Kieker System musste eine Methode geschaffen werden die serialisierten *Monitoring Records* zu importieren. In den bisherigen Arbeiten wurde dies auf unterschiedliche Arten getan, so wurde in [Richter 2012] das Programm *Seq2Kieker* verwendet, welches sich an Stelle einer Kieker Probe in der bisherigen Struktur einbettet (vgl. dazu Abbildung 2.1) und die erzeugten Protokolldaten in Kieker *Monitoring Records* transformiert.

Damit nicht für jede neue Programmiersprache erneut eine Schnittstelle entwickelt werden muss, wurde begonnen eine einheitliche Schnittstelle bereitzustellen. Aus dem MENGES-Projekt [Goerigk u. a. 2012] heraus wurde daraufhin von Reiner Jung begonnen die Kieker-Data-Bridge zu entwickeln. Um bereits zur Entwicklungszeit mit der Data-Bridge arbeiten zu können, habe ich zunächst Teile der Data-Bridge übernommen und um die Option zur Verwendung eines JMS-Providers sowie um die Verwendung von Textnachrichten statt Binärdaten erweitert. Diese Komponenten sind dann ebenfalls in die Data-Bridge migriert worden, so dass für die späteren Tests die fertige Data-Bridge verwendet werden konnte.

Die Data-Bridge kennt verschiedene Konnektoren, die durch die *Service Connectors* realisiert werden. Zusätzliche Konnektoren für neue Kommunikationsarten können somit bei Bedarf ergänzt werden, indem ein neuer Service Connector erstellt wird. Zur Zeit kann die Bridge Daten direkt per TCP oder JMS entgegennehmen. Auch für den Verbindungsaufbau existieren verschiedene Verfahren, in denen die Kieker-Data-Bridge entweder als Server auf eingehende Verbindungen lauscht oder sich als Client an einem wartenden Monitoring Prozess anmeldet. Im JMS-Modus kann sich die Data-Bridge entweder an einem JMS-Provider anmelden oder einen eigenen JMS-Provider im eingebetteten Modus starten.

In dieser Arbeit werden die Nachrichten in Textform, wie in Abschnitt 3.2 beschrieben, übertragen. Das Format wurde möglichst einfach gewählt um eine leichte Implementierbarkeit auch für zukünftige Projekte zu gewährleisten. Die Nachrichten beginnen mit einer Id, auf die, mit Semikolons getrennt, die Attribute folgen. Sollte ein Semikolon in den Datenfeldern auftreten, muss dieses maskiert werden. Dies sollte sich aber als unkritisch erweisen, da als Nutzdaten in den *Monitoring Records* nur Zahlen oder Funktionsnamen übertragen werden, die in den meisten Programmiersprachen nur aus einem eingeschränkten

[3] Grafik freundlicherweise zur Verfügung gestellt von Reiner Jung

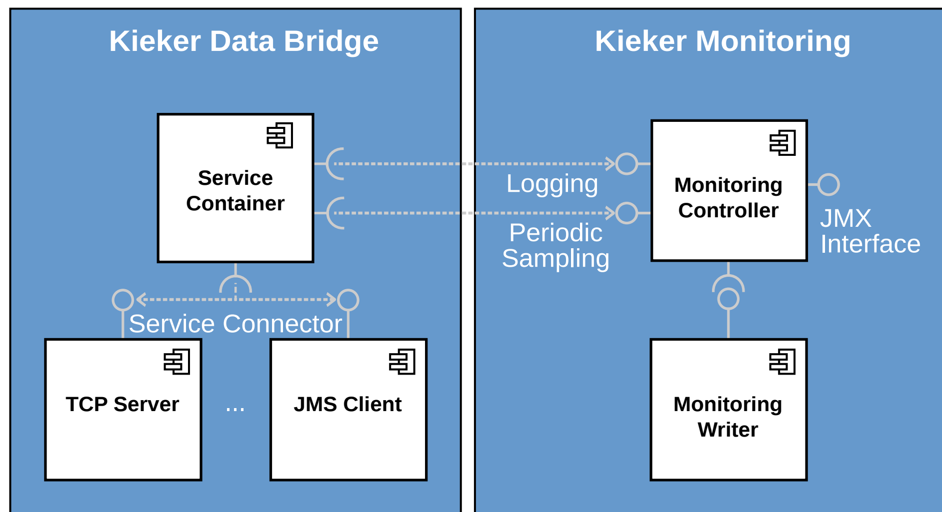


**Abbildung 3.3.** Komponentendiagramm-Kieker-Data-Bridge[3]

Zeichensatz bestehen. Das Ende der Nachricht wird mit dem betriebssystemspezifischen Zeilenende signalisiert.

Die Identifizierung der *Monitoring Records* erfolgt durch eine Mapping-Datei, in der für jeden Record Typen die eindeutige Id angegeben wird. Das von mir verwendete Mapping ist in Listing 3.1 angegeben.

```
1=kieker.common.record.flow.trace.operation.BeforeOperationEvent
2=kieker.common.record.flow.trace.operation.AfterOperationEvent
3=kieker.common.record.flow.trace.Trace
```

**Listing 3.1.** Record Mapping der Kieker-Data-Bridge

Nach der Auswahl des passenden Records werden die jeweiligen Attribute entsprechend dem TYPES Attribut der Records ausgewertet.

Neben den hier verwendeten Textnachrichten können die Nachrichten auch in einem binärem Modus übergeben werden. Außerdem kann die Data-Bridge nicht nur auf der Kommandozeile ausgeführt werden, sondern auch als Eclipse-Plugin verwendet werden. Diese Optionen wurden allerdings für diese Arbeit nicht verwendet.

Kapitel 4

# Instrumentierung

*Never send a human to do a machine's job.*
— Agent Smith, *The Matrix*

Nach der Implementierung der Perl Module mussten die *Monitoring Probes* in den zu untersuchenden Code eingebracht werden. Das EPrints System ist objektorientiert programmiert und verfügt über eine große Anzahl an verwendeten Perl Modulen. In diesen Modulen mussten alle Subroutinen mit OperationEntry- und OperationExitEvents versehen werden. Diese Instrumentierung wurde zunächst mit verschiedenen selbstentwickelten statischen Analysen versucht. Diese Versuche sind in Abschnitt 4.1 beschrieben. Da diese Versuche keine zufriedenstellenden Ergebnisse brachten wurde schließlich das CPAN-Modul Sub::WrapPackages[1] verwendet, das eine leichtgewichtige Lösung zur Erzeugung von Funktionen um eine existierende Subroutine herum darstellt. Mit diesem Paket wurde dann die Instrumentierung wie in Abschnitt 4.2 beschrieben durchgeführt.

## 4.1. Statische Integration der Probes

Zunächst wurde versucht die Pakete statisch zu analysieren und nach dieser Analyse die benötigen *Monitoring Probes* in den Quelltext einzubringen. Für OperationEntryEvents funktioniert dieses Verfahren auch zufriedenstellend, indem am Anfang jedes Moduls zunächst ein Kieker Objekt erzeugt wird, dann mittels regulärer Ausdrücke der Beginn von Funktionen erkannt wird und anschließend der folgende Code eingebracht wird.

```
sub load_source {
  $kieker->EntryEvent( 'load_source', 'EPrints::Citation' );
```

**Listing 4.1.** Eingebrachtes OperationEntryEvent

Diese Integration funktioniert weitestgehend automatisch. Um die Zuverlässigkeit zu erhöhen kann hierbei auch das Paket Perl::Tidy eingesetzt werden um Unterschiede in der Formatierung auszugleichen und die regulären Ausdrücke einfacher gestalten zu können.

[1] http://search.cpan.org/ dcantrell/Sub-WrapPackages-2.0/lib/Sub/WrapPackages.pm (Zuletzt aufgerufen 2013-03-25)

Ein größeres Problem entstand bei der Erzeugung der nötigen OperationExitEvents. Verfügt die Funktion über ein return Statement kann auch hier in einem naiven Ansatz versucht werden, wie in Listing 4.2 das ExitEvent direkt vor dem return einzufügen.

```
  $kieker->ExitEvent( 'load_source', 'EPrints::Citation' );
  return $value
}
```

**Listing 4.2.** Eingebrachtes OperationExitEvent

Hierbei ist allerdings festzustellen, dass Perl-Funktionen nicht zwingend über ein return Statement verfügen müssen. Sofern die Funktion ohne ein return Statement terminiert, wird der Wert der letzten ausgeführten Operation zurückgegeben. Wie in Listing 4.3 können somit z.B. minimale Funktionen realisiert werden, die einen Wert zurückliefern.

```
sub get_default_charset {
  "utf8"
}
```

**Listing 4.3.** Minimale Funktion ohne return

In Kombination mit verschiedenen Verzweigungen oder Schleifen macht dieses Verhalten es schwierig den Wert einer Funktion sicher zu bestimmen und die letzte ausgeführte Operation zu lokalisieren.

Weiterhin ist es möglich, dass die Argumente im return Statement über Seiteneffekte verfügen. Hierbei muss dann eine neue Variable erzeugt werden, die den Wert der Anweisung speichert, und die anschließend zurückgegeben werden kann. Diese Konstruktion ist in Listing 4.4 dargestellt.

```
  my $kieker_return = get_default_charset();
  $kieker->ExitEvent( 'load_source', 'EPrints::Citation' );
  return $kieker_return;
```

**Listing 4.4.** return mit neuer Variablenbindung

Als weiteres Problem stellt sich hierbei allerdings das Typensystem von Perl heraus. So ist es möglich, dass eine Funktion verschiedene Rückgabetypen erzeugt, je nachdem in welchem Kontext sie aufgerufen wird. Somit muss analysiert werden, ob das in Listing 4.4 erzeugte *kieker_return* einen Skalaren Typ oder eine Liste darstellen muss. Die Rückgabe von Hashes wird in Perl nicht unterstützt. Für vordefinierte Funktionen können diese Informationen noch durch ein Parsen der Dokumentation erhalten werden, jedoch ist auch hier schon ein massiver Aufwand von Nöten.

Werden nun aber Objektmethoden in dem Return statement aufgerufen, kann nicht mehr aus der Signatur erkannt werden, was für ein Rückgabetyp hier erzeugt wird. Weiterhin existiert das Konstrukt *wantarray* mit dem zur Laufzeit unterschiedliche Typen zurückgegeben werden können, je nachdem in welchem Kontext der Aufruf erfolgt. Somit scheiterte dieser Ansatz der Instrumentierung.

## 4.2. Integration mit Sub::WrapPackages

Als zweiter Ansatz wurde dann die Verwendung eines spezialisierten Paketes aus dem CPAN gewählt. Das Paket Sub::WrapPackages wurde von David Cantrell entwickelt und dient dazu, gesamte Packages oder auch spezifische Funktionen innerhalb von Packages mit Wrappern zu umschließen und so das Ausführen von beliebigem Code vor und nach der Ausführung ermöglichen.

Zur Benutzung des Paketes muss das Paket lediglich vor der Ausführung der Pakete wie folgt eingebunden werden.

```
1  use Sub::WrapPackages
2    packages => [qw(EPrints EPrints::*)],
3    pre => sub {
4      use Kieker;
5      my $kieker = Kieker->new();
6      $_[0] =~ s/::/./g;
7      $_[0] =~ /^(.*)\..*?$/;
8      $kieker->EntryEvent($_[0],$1);
9    },
10   post => sub {
11     use Kieker;
12     my $kieker = Kieker->new();
13     $_[0] =~ s/::/./g;
14     $_[0] =~ /^(.*)\..*?$/;
15     $kieker->ExitEvent($_[0],$1);
16   };
```

**Listing 4.5.** Benutzung von Sub::WrapPackages

In diesem Fall werden in Zeile 2 alle EPrints Pakete für die Instrumentierung gewählt und jeweils vor (Zeilen 3 bis 9) und nach (Zeilen 10 bis 16) der Ausführung der Funktionen Entry- bzw. ExitEvents erzeugt. In den Events werden der Funktionsname und auch der Paketname als Parameter übergeben. Hierbei ist zu beachten dass in Perl der zweifache Doppelpunkt als Trennzeichen in Signaturen verwendet wird und nicht der in Java übliche einfache Punkt. Diese Notation wird hier mit regulären Ausdrücken (Zeilen 6, 7, 13 und 14) angepasst um später besser in Kieker.Analysis ausgewertet werden zu können.

Der angegebene Code kann an verschiedenen Stellen eingebracht werden. Eine Möglichkeit wäre es, direkt in der Konfiguration des Webservers diesen Code ausführen zu lassen. Somit könnten auch Initialisierungen der Webanwendungen mit in dem Monitoring erfasst werden. Leider hat sich in einer der letzten Perl-Updates die Behandlung von definierten Konstanten verändert, was zu einem Fehler in dem Modul Sub::WrapPackages führt. Dieser Fehler verhindert die Aktivierung des Moduls bevor durch die Anwendung die Verbindung zur Datenbank aufgebaut wurde, da sonst Typenfehler bei Datenbankanfragen auftreten.

Statt dessen wurde der Code in die CGI-Dateien eingebracht, die vom Benutzer aufgerufen werden. Somit konnte selektiv bestimmt werden, welche Requests von dem Monitoring erfasst werden sollten. Zusätzlich wurden in der CGI-Datei zu Beginn und Ende des Codes Entry- und ExitEvents eingefügt, um einen äußeren Rahmen für den Trace zu bieten.

Kapitel 5

# Testdurchführung

*Remember... all I'm offering is the truth.*
*Nothing more.*

— Morpheus, *The Matrix*

Nachdem die Instrumentierung der Anwendung und die Übernahme der Daten in das Kieker Framework realisiert war, wurde die EPrints Plattform untersucht. Diese Tests wurden nicht auf dem Produktivsystem sondern auf einem lokalen Testsystem durchgeführt, das in Abschnitt 5.1 zunächst beschrieben wird. Anschließend wird in Abschnitt 5.2 der grundlegende Aufbau des Testszenarios beschrieben und in Abschnitt 5.3 die durchgeführten Requests dokumentiert.

## 5.1. Systemumgebung

Die Tests wurden nicht auf den Systemen des GEOMAR durchgeführt, sondern auf einer lokalen, extra eingerichteten virtuellen Maschine. Als Host-System wurde dabei ein MacMini aus dem Jahr 2012 verwendet, der mit einem Intel Core i5 mit 2,5 GHz und 8GB Arbeitsspeicher ausgestattet ist. Zur Virtualisierung wurde die Software VirtualBox in der Version 4.2.8 unter Mac OS X 10.8.2 verwendet. In der virtuellen Maschine wurden dem Gastsystem 4 GB Arbeitsspeicher bereitgestellt.

Als Gastsystem wurde ein Ubuntu Linux 12.04 LTS eingesetzt. In diesem System wurde versucht die Gegebenheiten des GEOMAR System zu simulieren. Hierfür wurde PostgreSQL 8.4.16, Apache 2.2.22 und Perl 5.14.2 aus den Ubuntu Paketquellen gewählt und installiert. Für den Import der vom GEOMAR gelieferten Daten mussten einigen Referenzen in der Datenbank mit Prototypen aufgelöst werden. Hierzu musste ein neues, leeres Datenbankschema und einige Tabellen angelegt werden. Diese Tabellen wurden nicht mit Daten gefüllt und sind nicht direkt auf Kielprints bezogen sondern beinhalten normalerweise Daten zu Forschungsexpeditionen des Instituts.

## 5.2. Testaufbau

In den Tests wurde ein Vergleich zwischen dem in Kiel verwendeten Kielprints und dem entsprechenden unmodifizierten EPrints 3.2 durchgeführt. Als Datenbasis wurde ein Datenexport der GEOMAR Datenbank vom 7. Februar 2013 verwendet. Kielprints wurde in der Version vom 4. März aus dem SVN-Repository des GEOMAR ausgecheckt und auf der virtuellen Maschine eingerichtet.

Beide Systeme wurden nacheinander auf der virtuellen Maschine gestartet und die selben Requests auf den Systemen ausgeführt. Hierfür wurde in beiden Systemen die Datei *eprints/cgi/users/home* wie in Abschnitt 4.2 beschrieben instrumentiert. Es wurden die Requests ausgeführt und die erhaltenen *Monitoring Records* an einen laufenden JMS-Provider gesendet. Mittels der Kieker-Data-Bridge wurden die *Monitoring Records* dem JMS-Provider entnommen und mittels der Standardkonfiguration als Trace im Dateisystem abgelegt.

## 5.3. Requests

Es wurden fünf Requests ausgewählt um einen exemplarischen Arbeitsvorgang darzustellen. Neben vier Seitenaufrufen ist hierbei auch ein Request via AJAX vorhanden, der in Kielprints besonders lange zur Ausführung benötigt. In den folgenden Abschnitten werden jeweils die genauen Requests aufgeführt und dann in Kapitel 6 die jeweils erhaltenen Daten vorgestellt.

### 5.3.1. Request 1: Admin-Login

*Request URL:* http://nbw-virtualbox/cgi/users/home

*Trace Ids:* Eprints: 6884 - Kielprints: 5821

Direkt nach dem Login wird eine Seite generiert, auf der die eingetragenen Publikationen angezeigt werden. Die Seiten unterscheiden sich primär in ihrem Aussehen, die Funktionalität ist anscheinend gleich. (Abbildung 5.1)

### 5.3.2. Request 2: Eintragen eines neuen Titels

*Request URL:* http://nbw-virtualbox/cgi/users/home
?screen=EPrint::Edit&eprintid=20282&stage=type

*Trace Ids:* Eprints: 6886 - Kielprints: 5820

Auf dieser Seite wird begonnen, wenn eine neue Publikation in der Datenbank eingetragen werden soll. Es bestehen leichte Unterschiede in den Seiten, so wird in der Kielprints-Version das zu verwendenden Archiv sowie eine Gewichtung für den Impact Factor abgefragt, während die EPrints Seite nur den Typ der Publikation erfragt. (Abbildung 5.2)

### 5.3.3. Request 3: Upload der Publikation

*Request URL:* http://nbw-virtualbox/cgi/users/home
?screen=EPrint::Edit&eprintid=20280&stage=files#t

*Trace Ids:* Eprints: 6885 - Kielprints: 5829

Es wird nach der hochzuladenden Datei gefragt. Für diesen Test wurde keine Datei hochgeladen. Als Unterschied zwischen den beiden Versionen ist zu beachten, dass die Reihenfolge der Abfragen variiert. In Kielprints erfolgt dieser Upload erst nach der Eingabe der Metadaten, in EPrints vor dieser. Funktionale Unterschiede scheinen nicht zu bestehen.

### 5.3.4. Request 4: Eingabe der Metadaten

*Request URL:* http://nbw-virtualbox/cgi/users/home
?screen=EPrint::Edit&eprintid=20282&stage=core#t

*Trace Ids:* Eprints: 6888 - Kielprints: 5824

Auf dieser Seite werden die meisten Metadaten der Publikation eingetragen, also neben Titel und Autor und Kurzbeschreibung auch Herausgeber, Erscheinungsjahr und Einstellungen zum Open-Access-Zugriff. In Kielprints können hier auch Verweise zu Expeditionen des Instituts oder Verbindungen zu Sonderforschungsbereichen angegeben werden.

### 5.3.5. Request 5: (AJAX) Mehr Autorenfelder

*Request URL:* http://nbw-virtualbox/cgi/users/home
?_internal_c15_creators_morespaces=Mehr%20Eingabefelder&[...]

*Trace Ids:* Eprints: 6889 - Kielprints: 5825

Werden für die Erfassung mehr Felder für die Autoren benötigt, so können hier zusätzliche Felder eingeblendet werden ohne die gesamte Seite neu laden zu müssen. Lediglich das Block-Element mit den Autoren wird auf dem Server neu generiert und per JavaScript in der Seite ersetzt. Die Eingabefelder verfügen über Optionen zur automatischen Vervollständigung, so dass hier nicht nur statische Eingabefelder generiert, sondern auch Daten aus der Datenbank angefordert werden müssen.

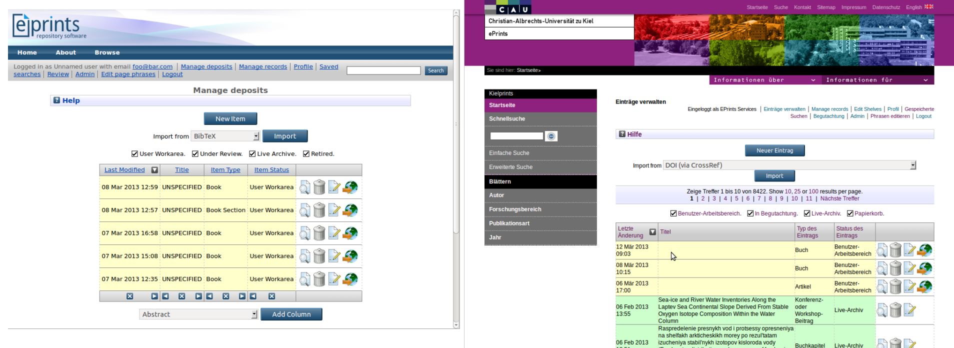

**Abbildung 5.1.** Vergleich der Seiten zu Request 1 (Links EPrints, Rechts Kielprints)

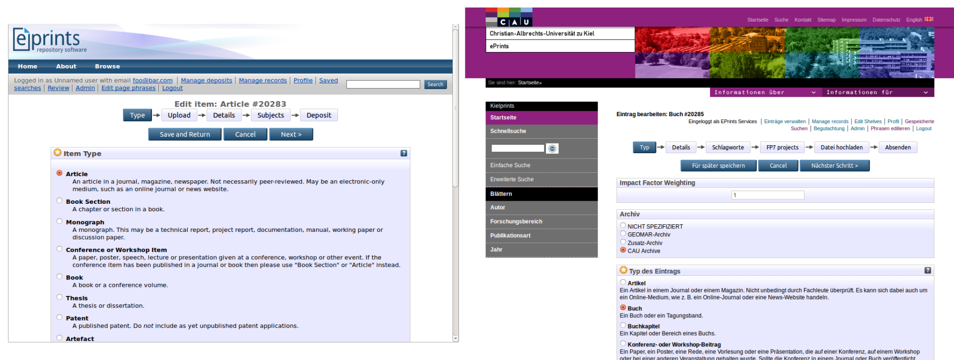

**Abbildung 5.2.** Vergleich der Seiten zu Request 2 (Links EPrints, Rechts Kielprints)

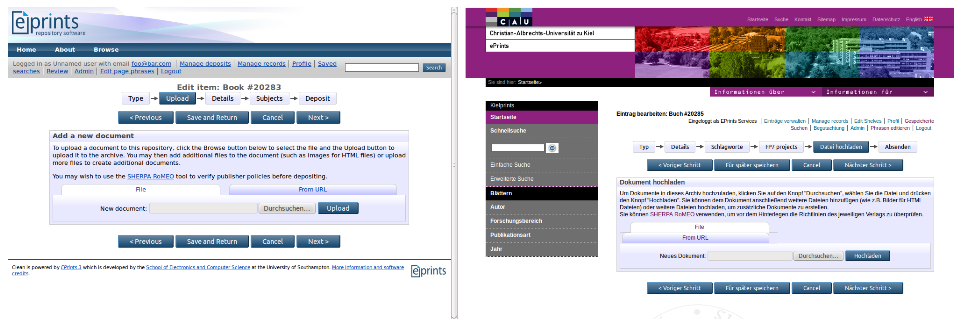

**Abbildung 5.3.** Vergleich der Seiten zu Request 3 (Links EPrints, Rechts Kielprints)

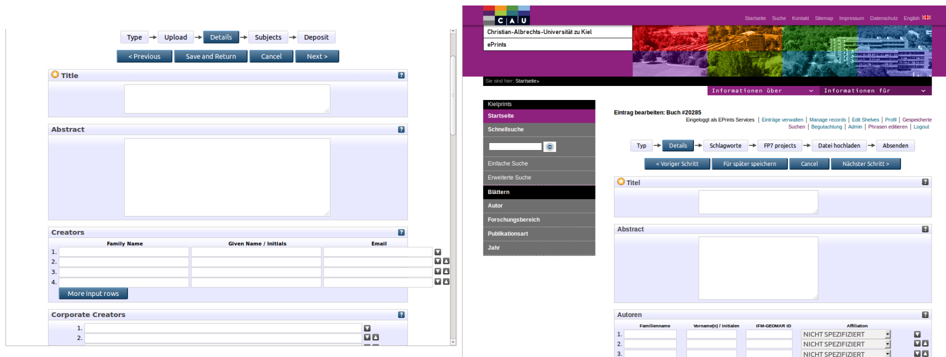


**Abbildung 5.4.** Vergleich der Seiten zu Request 4 (Links EPrints, Rechts Kielprints)

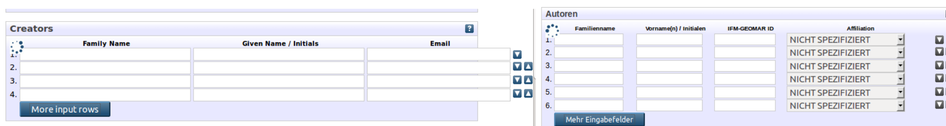


**Abbildung 5.5.** Vergleich der Seiten zu Request 5 (Links EPrints, Rechts Kielprints)

Kapitel 6

# Ergebnisse

*Welcome to the real world.*
— Morpheus, *The Matrix*

Die in Abschnitt 5.3 beschriebenen Requests wurden, nach der Verarbeitung durch die Kieker-Data-Bridge, mit dem Kommandozeilentool *kieker-analysis.sh* aufbereitet. Hierbei wurden zunächst mit der Option *--print-Execution-Traces* textuelle Beschreibungen der erstellten Traces erzeugt. Aus diesen Beschreibungen konnten dann das zeitliche Verhalten, welches in Abschnitt 6.1 erläutert wird, und die Funktionsaufrufe, welche in Abschnitt 6.2 besprochen werden, abgelesen werden.

Anschließend wurde mit einem Perl-Script (siehe Anhang B.1) die Aufrufhäufigkeit für jede Funktion gezählt. Die Ergebnisse dieser Untersuchung werden in Abschnitt 6.3 vorgestellt. Schließlich wurden die Funktionsaufrufe mit einem weiteren Perl-Script (siehe Anhang B.2) in ihre Pakete aggregiert und die Aufrufhäufigkeiten für diese bestimmt. Diese Daten werden in Abschnitt 6.4 präsentiert und diskutiert.

Zur weiteren Veranschaulichung wurden dann auf den aggregierten Daten mit der Kommandozeilenoption *--plot-Assembly-Component-Dependency-Graph* die in Abschnitt 6.5 präsentierten Abhängigkeitsgraphen erstellt, um die Verbindungen zwischen den Paketen zu visualisieren.

## 6.1. Zeitliches Verhalten

Zunächst wurde eine Zeitmessung für alle Requests ausgeführt. Die erhaltenen Daten sind in Tabelle 6.1 aufgetragen. Zunächst ist hierbei festzustellen, dass die Instrumentierung offenbar einen extremen Anstieg der Ausführungszeit verursacht hat. Als wahrscheinlichste Engstelle vermute ich hier das Versenden der Records an den JMS-Provider via TCP. Hier sollte in Zukunft auf eine asynchrone Kommunikation umgestellt werden um den Programmablauf nicht unnötig zu verzögern.

Vergleicht man die Ausführungszeiten zwischen den instrumentierten Varianten, so ist auffällig, dass in den Request 2 und 3 mit Faktor 1,1 und 0,9 kaum Abweichungen auftreten, aber in Request 4 eine Steigerung der Ausführungszeit um den Faktor 11,9 und in Request 5 sogar eine Steigung um den Faktor 48,5 auftritt. Weiter ist auffällig, dass in EPrints die Zeit für Request 5 um 70% geringer ausfällt als für Request 4, während in

**Tabelle 6.1.** Zeitliches Verhalten von Eprints und Kielprints vor und nach Instrumentierung

| Request | Nr. 1 | Nr. 2 | Nr. 3 | Nr. 4 | Nr. 5 |
|---|---|---|---|---|---|
| Eprints normal | 402 ms | 220 ms | 136 ms | 413 ms | 348 ms |
| EPrints instrumentiert | 15389 ms | 15043 ms | 18408 ms | **23430 ms** | **7066 ms** |
| Kielprints normal | 10270 ms | 227 ms | 166 ms | 13420 ms | 18890 ms |
| Kielprints instrumentiert | 28505 ms | 16623 ms | 17414 ms | **280927 ms** | **342662 ms** |
| Faktor normal | 25,5 | 1,0 | 1,2 | 32,5 | 54,3 |
| Faktor instrumentiert | 1,8 | 1,1 | 0,9 | 11,9 | 48,5 |

Kielprints hier noch eine Steigerung um 21% von Request 4 zu Request 5 festzustellen ist. Der Request 1 liegt zwischen den beiden Verhalten, indem es zwar eine Steigerung aufweißt aber diese mit dem Faktor 1,85 deutlich geringer ausfällt als in Requests 4 und 5.

Diese Verhältnisse verschieben sich noch bei den nicht-instrumentierten Versionen. Die Requests 2 und 3 weisen auch bei diesen Messungen kaum Unterschiede auf, doch bei den Request 1, 4 und 5 steigt die Ausführungszeit um Faktoren zwischen 25,5 und 54,3. Die unterschiedlichen Verhältnisse zwischen den normalen und instrumentierten Versionen lassen sich auf den großen Einfluss des Monitorings, und hier speziell das Senden der Records, zurückführen.

## 6.2. Funktionsaufrufe

Als nächstes kann die Anzahl protokollierter Funktionsaufrufe betrachtet werden. In Tabelle 6.2 sind die einzelnen Requests sowie die Summe über alle 5 Requests aufgetragen. Die Ergebnisse bieten ähnliche Folgerungen, wie bereits bei der zeitlichen Betrachtung.

**Tabelle 6.2.** Funktionsaufrufe von EPrints und Kielprints

| Request | 1 | 2 | 3 | 4 | 5 | Summe |
|---|---|---|---|---|---|---|
| Eprints | 20875 | 28590 | 33171 | 43056 | 16980 | 142672 |
| Kielprints | 31742 | 36681 | 40165 | 905580 | 934760 | 1948928 |
| Faktor | 1,5 | 1,3 | 1,2 | 21,0 | 55,1 | 13,7 |

Zwar sind die Funktionsaufrufe in den ersten drei Requests bei Kielprints höher als in EPrints, allerdings bewegen sich die beiden Systeme noch in den selben Größenordnungen. Als Abweichung zwischen dem zeitlichen Verhalten und den Funktionsaufrufen ist bei Request 3 festzustellen, dass trotz einer kürzeren Ausführungszeit eine Steigerung der Funktionsaufrufe auftritt. Bei Request 4 weißt Kielprints eine Steigerung um den Faktor 21,0 gegenüber dem unmodifizierten EPrints auf, bei Request 5 beträgt die Steigerung gar Faktor 55,1. In der Summe ergibt sich eine Steigerung um den Faktor 13,7 zwischen den Systemen.

## 6.3. Aufrufhäufigkeiten für Funktionen

Werden die Aufrufhäufigkeiten für die einzelnen Funktionen ausgewertet, zeigen sich auffällige Verschiebungen. Hierfür betrachte ich exemplarisch Request 4, für den ich in Tabelle 6.3 jeweils die 10 häufigsten Funktionen angegeben habe.

**Tabelle 6.3.** Die 10 meistgenutzten Funktionen bei Request 4

| **EPrints** | **Aufrufe** |
|---|---|
| EPrints.Script.Compiler.next_is | 5873 |
| EPrints.Repository.xml | 2668 |
| EPrints.XML.is_dom | 1690 |
| EPrints.XHTML._to_xhtml | 1345 |
| EPrints.Utils.is_set | 1230 |
| EPrints.XML.EPC.process | 1073 |
| EPrints.XML.clone_node | 968 |
| EPrints.Repository.get_repository | 943 |
| EPrints.Repository.clone_for_me | 923 |
| EPrints.XML.create_element | 842 |
| **Kielprints** | **Aufrufe** |
| EPrints.MetaField.Id.value_from_sql_row | 75008 |
| EPrints.MetaField.property | 65202 |
| EPrints.Database.quote_identifier | 52383 |
| EPrints.Utils.is_set | 44571 |
| EPrints.MetaField.get_sql_names | 37258 |
| EPrints.DataSet.field | 31053 |
| EPrints.MetaField.get_value | 30598 |
| EPrints.DataObj.get_value_raw | 30598 |
| EPrints.MetaField.get_property | 27790 |
| EPrints.XML.is_dom | 24304 |

Die am häufigsten aufgerufene Funktion in EPrints ist *EPrints.Script.Compiler.next_is*, eine Funktion die zur Verarbeitung der EPrints-internen Scriptsprache EPscript benötigt wird. Die Funktionen aus den Paketen *EPrints.XHTML* und *EPrints.XML* dienen offenbar primär der Aufbereitung des XHTML Codes der aufgerufenen Seite. Die Funktion aus dem Paket *EPrints.Repository* behandeln das zentrale Datenobjekt des EPrints-Systems, welches z.B. die Datenbankverbindung oder den aktuellen Request zum zentralen Zugriff kapselt.

Im Vergleich dazu sind die häufigsten Funktionen in Kielprints weitestgehend auf die Datenbank bezogen. Neben den direkten Datenbankfunktionen dienen die meisten Funktionen aus den *EPrints.MetaField* Paketen zur Aufbereitung der erhaltenen Daten. Besonders ist hierbei zu beachten, dass die häufigste Funktion, *EPrints.MetaField.Id.value_from_sql_row*, in der Dokumentation als *deprecated* und Relikt aus EPrints Version 2 ausgewiesen ist und

nicht mehr verwendet werden sollte. Wird eine Summe über alle 5 Requests gebildet ergibt sich ein ähnliches Bild, wie in Tabelle 6.4 zu erkennen ist.

**Tabelle 6.4.** Die 10 meistgenutzten Funktionen insgesamt

| **EPrints** | **Aufrufe** |
|---|---|
| EPrints.Script.Compiler.next_is | 17534 |
| EPrints.Repository.xml | 6428 |
| EPrints.XML.is_dom | 5457 |
| EPrints.Utils.is_set | 5361 |
| EPrints.MetaField.property | 4185 |
| EPrints.DataSet.field | 3310 |
| EPrints.Repository.get_repository | 3181 |
| EPrints.XML.EPC.process | 3011 |
| EPrints.XML.clone_node | 2682 |
| EPrints.Repository.clone_for_me | 2652 |

| **Kielprints** | **Aufrufe** |
|---|---|
| EPrints.MetaField.Id.value_from_sql_row | 155884 |
| EPrints.MetaField.property | 138063 |
| EPrints.Database.quote_identifier | 111214 |
| EPrints.Utils.is_set | 95232 |
| EPrints.MetaField.get_sql_names | 78938 |
| EPrints.DataSet.field | 66011 |
| EPrints.MetaField.get_value | 64282 |
| EPrints.DataObj.get_value_raw | 64282 |
| EPrints.MetaField.get_property | 59662 |
| EPrints.XML.is_dom | 53005 |

Analog zu Request 4 dominieren hier in Kielprints die Funktionen zur Behandlung von Datenbankanfragen, während diese in EPrints nur eine untergeordnete Rolle spielen. In EPrints wird weiterhin die Funktion *EPrints.Script.Compiler.next_is* am häufigsten ausgeführt, welche bei der Ausführung von Scripten den Typ des nächsten Objektes bestimmt.

## 6.4. Aufrufhäufigkeiten für Pakete

Mit dem in Anhang B.2 angegebenen Skript wurden die individuellen Funktionsaufrufe auf ihre Paketzugehörigkeit reduziert, um Abhängigkeiten auf Paketebene analysieren zu können. Hierbei ergeben sich, über alle Requests zusammen, die in Tabelle 6.5 angegebenen Werte.

Diese Daten spiegeln erneut die Ergebnisse der bisherigen Auswertungen. Die Datenbankanbindung ist in EPrints das neunte Paket in der Liste, während es auf den vierten Platz in Kielprints vorrückt. Das wichtigste Paket in EPrints, *EPrints.Script.Compiler*, rückt

**Tabelle 6.5.** Die 10 aktivsten Pakete

| **EPrints** | **Aufrufe** |
|---|---|
| EPrints.Script.Compiler | 30304 |
| EPrints.Repository | 25980 |
| EPrints.MetaField | 18374 |
| EPrints.XML | 12486 |
| EPrints.DataSet | 7488 |
| EPrints.Utils | 7220 |
| EPrints.XML.EPC | 5703 |
| EPrints.DataObj | 5439 |
| EPrints.Database | 3466 |
| EPrints.Script.Compiled | 3404 |

| **Kielprints** | **Aufrufe** |
|---|---|
| EPrints.MetaField | 501872 |
| EPrints.DataSet | 253493 |
| EPrints.Repository | 243699 |
| EPrints.Database | 189624 |
| EPrints.DataObj | 156391 |
| EPrints.MetaField.Id | 155978 |
| EPrints.Utils | 118036 |
| EPrints.XML | 107939 |
| EPrints.MetaField.Multilang | 45312 |
| EPrints.Script.Compiler | 39550 |

in Kielprints auf den 10. Platz. Auch ist hier zu beachten, dass kaum zusätzliche Funktionsaufrufe in diesem Paket auftreten. Erneut ist hier das Paket *EPrints.MetaField.Id* vertreten, dass seit dem Update auf Version 3 nicht mehr verwendet werden sollte.

## 6.5. Paketabhängigkeiten

Mit den gleichen Daten wie in Abschnitt 6.4 wurden die Abhängigkeiten in Kielprints sowie in EPrints mit dem Grafiktool Gephi visualisiert. Hier wurde als Layout-Algorithmus die Einstellung Fruchtermann Reingold gewählt und der Graph anschließend mit Standardeinstellungen gerendert.

Der in Abbildung 6.1 hellgrün markierte Knoten stellt das Paket EPrints.Repository dar, der hellrot markierte Knoten das Paket EPrints.PluginFactory. Die jeweils direkt mit diesen Paketen benachbarten Knoten sind in der entsprechenden gedämpften Farbe markiert. Aufgrund der hohen Anzahl Knoten und der starken Vernetzung untereinander ist die Analyse hierbei allerdings nur schwierig möglich.

Werden in den Daten funktional ähnliche Pakete zusammengefasst, also zum Bei-

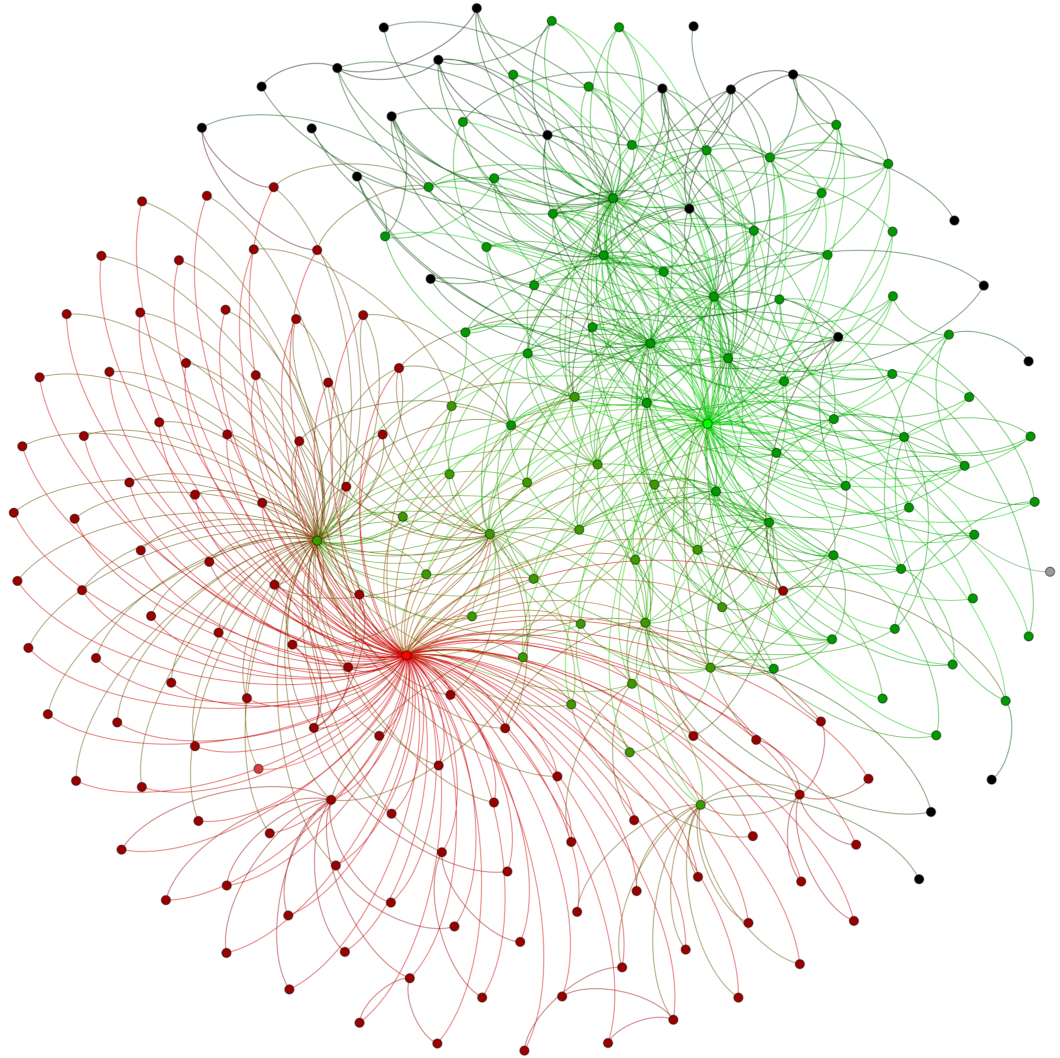

**Abbildung 6.1.** Visualisierung der Abhängigkeiten in EPrints

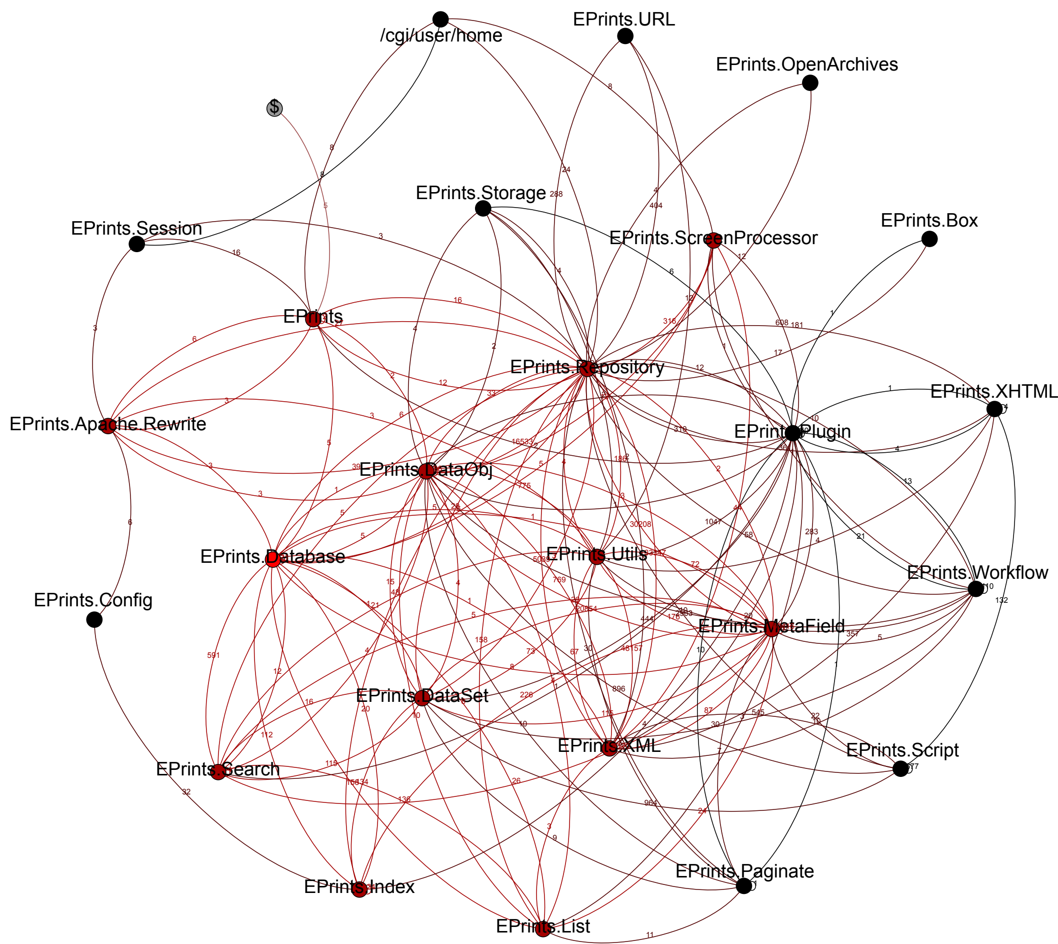


**Abbildung 6.2.** Visualisierung der Abhängigkeiten in EPrints

spiel die verschiedenen *EPrints.MetaField.** Pakete oder die Pakete *EPrints.Database.Pg* und *EPrints.Database*, so zeigen sich die Abhängigkeiten deutlicher. Ein Versuch dieser Visualisierung ist in Abbildung 6.2 dargestellt.

Es ergeben sich nur geringe Unterschiede zwischen den Abhängigkeitsgraphen von EPrints und Kielprints. Allerdings zeigt sich hier, dass auch in EPrints Probleme in der Schichtenarchitektur bestehen, denn auch hier greift zum Beispiel das Paket *EPrints.ScreenProcessor* direkt auf die Datenbank zu.

Kapitel 7

# Zusammenfassung und Ausblick

*Only human.*

— Agent Jones, *The Matrix*

Um die Arbeit abzuschließen, fasse ich zunächst in Abschnitt 7.1 das Vorgehen und die bisherigen Ergebnisse zusammen. Zum Schluss werden in Abschnitt 7.2 noch offene Fragen und mögliche Folgerungen aus dieser Arbeit präsentiert.

## 7.1. Zusammenfassung

Im Rahmen meiner Arbeit habe ich eine Software entwickelt, welche die dynamische Analyse von Perl-Programmen mittels des Kieker Frameworks ermöglicht. Hierzu wurde ein in Perl implementiertes Modul erstellt, welches die benötigten Daten im Kontext des Perl-Interpreters sammelt und diese zur weiteren Bearbeitung an die, ebenfalls neu entwickelte, Kieker-Data-Bridge überträgt. Das Modul verfügt neben den benötigten Records auch über Methoden zur Traceverwaltung, wodurch der Kommunikationsaufwand reduziert wird.

Zur Instrumentierung von bestehendem Code wurde anschließend versucht eine statische Analyse durchzuführen, was jedoch aufgrund von diversen Verhaltensweisen der Programmiersprache Perl nicht erfolgreich durchgeführt werden konnte. Statt dessen konnte ein CPAN-Paket eingesetzte werden, dass zur Erzeugung von Routinen vor und nach der Ausführung von bestehendem Code dient. Mit diesem Verfahren konnten dann sowohl die Software EPrints als auch die Variante Kielprints mit *Monitoring Probes* instrumentiert werden.

Mit diesen beiden Systemen wurde ein vergleichender Test durchgeführt um Unterschiede zwischen den Systemen zu lokalisieren. Hierbei wurde festgestellt, dass Kielprints bei selbem Datenbestand signifikant häufiger auf die Datenbank zugreift, worin ich die Ursache für die festgestellten Performanceprobleme vermute. Die Veränderungen im Verhalten deuten darauf hin, dass die Modifikationen in Kielprints einen Durchgriff durch die Schichtenarchitektur vornehmen. Auch wurde in diesem Kontext die Verwendung von veralteten Funktionen festgestellt, die auf absehbare Zeit zu einem Ausfall führen können, sollten die Funktionen in einer zukünftigen Version nicht mehr bereitgestellt werden.

Die ermittelten Daten des Vergleichstest können zur Zeit nur zur groben Lokalisierung der Probleme dienen, allerdings können die Daten auch noch detaillierter in Hinblick

auf Abhängigkeiten oder Aufrufreihenfolge analysiert werden. Die hier präsentierten Auswertungen decken in dieser Hinsicht nur einen Teil ab. Als direkte Erkenntnis lässt sich allerdings bereits feststellen, dass in Kielprints darauf geachtet werden sollte, die von EPrints vorgesehene Skriptsprache zu verwenden und direkte Zugriffe auf die Datenbank zu vermeiden.

## 7.2. Ausblick

Das im Rahmen dieser Arbeit realisierte Perl-Modul weißt zur Zeit nur einen eingeschränkten Funktionsumfang auf. Neben den implementierten drei Records unterstützt das Kieker Framework noch diverse weitere Recordtypen, die aufgrund der Zielsetzung dieser Arbeit bisher keinen Einzug in die Perl-Implementierung gefunden haben.

Als weitere Verbesserung sollte eine signifikante Verringerung des Performance-Overheads priorisiert werden. Dies kann vermutlich durch Modifikationen beim Schreiben der Records erreicht werden. In der weiteren Entwicklung sollte dann adaptives Monitoring integriert werden, um selektiver das Monitoring steuern zu können. Diese Änderungen erfordern dann auch eine Erweiterung der Kieker-Data-Bridge um einen vollständigen Rückkanal von Kieker zu dem überwachten System.

Die Analyse der ermittelten Daten kann genutzt werden, um sowohl EPrints im Allgemeinen als auch Kielprints im Speziellen zu verbessern und bisher unerkannte Schwächen in den Systemen zu lokalisieren und zu beheben. In Abstimmung mit dem GEOMAR können hier noch weitere Analysen durchgeführt werden, um die Probleme weiter eingrenzen zu können und Kielprints wieder in einen Zustand zu versetzen, der einen Regelbetrieb für die gesamte Universität ermöglicht.

Auch EPrints kann von diesen Untersuchungen profitieren. Da die Software bereits seit vielen Jahren von verschiedenen Entwicklern programmiert und modifiziert wird, steht es zu erwarten, dass die Software unerwünschtes Verhalten an den Tag legt und eine Überprüfung des theoretischen Modells gegenüber der tatsächlichen Implementierung zu einer Verbesserung der Software führen kann. Hierzu ist in dieser Arbeit im Rahmen der Abhängigkeitsanalyse schon ein erster Ansatz geboten worden.

Anhang A

# Perl Dokumentation

## A.1. Kieker

Wraps Kieker Monitoring functions

### SYNOPSIS

```
use Kieker;
my $kieker = Kieker->new();          # Creates a new monitoring instance

$kieker->EntryEvent('Foo','Bar');    # Produces a new EntryEvent

$kieker->ExitEvent('Foo','Bar');     # Produces a new ExitEvent
```

### DESCRIPTION

This module encapsulates the Kieker monitoring functions in one module. Upon creation the controlling and writing parts are loaded and the singleton instances are stored. After that Events can be constructed and written to the active writer.

### METHODS

**$object = Kieker->new()**

Creates a new Kieker object. Currently takes no parameters for configuration. Returns the object.

**$object->EntryEvent($functionName, $packageName);**

Creates and sends out a EntryEvent for a specified functionName in a specified package. Uses the current timestamp. Returns nothing.

**$object->ExitEvent($functionName, $packageName);**

Creates and sends out a ExitEvent for a specified functionName in a specified package. Uses the current timestamp. Returns nothing.

## A.2. Kieker::Controlling

Provides Controlling mechanisms to Kieker Modules

### SYNOPSIS

```
use Kieker::Controlling;
my $control = Kieker::Controlling->instance();   # Get controlling instance

my $trace = $control->getTrace();                # Get current traceID

my $orderId = $control->getOrderIndex($trace);   # Get current orderIndex
```

### DESCRIPTION

This module provides controlling methods for Kieker. The main functions are providing consistent trace and orderIDs. When new traces are created the corresponding traceEvents are created and written.

### METHODS

**$control = Kieker::Controlling->instance();**

Returns the singleton instance of Kieker::Controlling. If no instance is defined a new instance is created.

**my $orderId = $control->getOrderIndex($trace);**

Returns the current orderIndex and increments the count. If this is the first request a new traceEvent is created and written to the log.

**my $trace = $control->getTrace();**

Returns the current traceID. In this version it uses the threadID and relies on special Apache configuration.

## A.3. Kieker::Util

Kieker utility functions

### SYNOPSIS

```
use Kieker::Util;
my $time = Kieker::Util->time();   # Get current time in (pseudo)-nanoseconds
```

### DESCRIPTION

This module provides utility functions for other Kieker modules.

### METHODS

**$time = Kieker::Util->time();**

Returns current time in pseudo nanoseconds. Perl doesn't provide a real nanosecond timer but only microseconds. Output is formatted to match kieker time format. NO REAL NANOSECONDS, MICROSECOND PRECISION.

## A.4. Kieker::Record::OperationEntryEvent

Kieker Event to be produced at the start of a function call.

### SYNOPSIS

```
my $record = Kieker::Record::OperationEntryEvent->new(
                Kieker::Util->time(),
                $control->getTrace(),
                $control->getOrderIndex($control->getTrace()),
                $functionName,$packageName);
$writer->write($record->genoutput());
```

### DESCRIPTION

Generates a OperationEntryEvent. This Event should be generated at the start of each monitored function.

### METHODS

**$record = Kieker::Record::OperationEntryEvent->new(tstamp, trace, orderIndex, function, package);**

Creates a new Record. It needs a timestamp, a trace with its corresponding orderIndex, a function name and a package name.

**$string = $record->genoutput();**

Serializes the record for output. Returns the serialized form of the record. Uses the identifier "1"for the event type.

## A.5. Kieker::Record::OperationExitEvent

Kieker Event to be produced at the end of a function call.

### SYNOPSIS

```
my $record = Kieker::Record::OperationExitEvent->new(
                Kieker::Util->time(),
                $control->getTrace(),
                $control->getOrderIndex($control->getTrace()),
                $functionName,$packageName);
$writer->write($record->genoutput());
```

### DESCRIPTION

Generates a OperationExitEvent. This Event should be generated at the end of each monitored function.

### METHODS

**$record = Kieker::Record::OperationExitEvent->new(tstamp, trace, orderIndex, function, package);**

Creates a new Record. It needs a timestamp, a trace with its corresponding orderIndex, a function name and a package name.

**$string = $record->genoutput();**

Serializes the record for output. Returns the serialized form of the record. Uses the identifier "2"for the event type.

## A.6. Kieker::Record::Trace

Kieker Event indicating one trace. Gets autocreated by Kieker::Controlling.

### SYNOPSIS

```
my $record = Kieker::Record::Trace->new($traceID);

$writer->write($record->genoutput());
```

### DESCRIPTION

Generates a Trace. This Event is autocreated by Kieker::Controlling when a new trace is started.

### METHODS

**$record = Kieker::Record::Trace->new($traceID);**

Creates a new trace with the given trace ID.

**$string = $record->genoutput();**

Serializes the record for output. Returns the serialized form of the record. Uses the identifier "3"for the event type.

## A.7. Kieker::Writer::FileWriter

Provides a Kieker Writer for file output.

### SYNOPSIS

```
use Kieker::Writer::FileWriter;
my $writer = Kieker::Writer::FileWriter->instance();
                                     # Get writer instance

$writer->write($string);             # writes string to the current file
```

### DESCRIPTION

Writes Kieker records to a Logfile. Can also be used to write other strings to log files.

### METHODS

**$writer = Kieker::Writer::FileWriter->instance();**

Returns the writer singleton instance. Thsi object holds the filehandle currently wrinting to. If no instance exists, it is created and a new logfile is created using the current time for the filename.

**$writer->write(STRING);**

Writes STRING to the filehandle.

## A.8. Kieker::Writer::JMSWriter

Provides a Kieker Writer for JMS output.

### SYNOPSIS

```
use Kieker::Writer::JMSWriter;
my $writer = Kieker::Writer::JMSWriter->instance();
                                     # Get writer instance

$writer->write($string);             # writes string to JMS
```

### DESCRIPTION

Writes Kieker records to a JMS provider. Can also be used to write other strings.

### REQUIREMENTS

Uses Net::Stomp for the JMS connection.

### METHODS

**$writer = Kieker::Writer::JMSWriter->instance();**

Returns the writer singleton instance. Thsi object holds the handle currently wrinting to. If no instance exists, it is created and a connection to the JMS provider is opened. Currently no configuration options are provided and the location of the provider is static in the source. (localhost:61613)

**$writer->write(STRING);**

Writes STRING to JMS.

Anhang B

# Perl Skripte zur Auswertung

## B.1. Zählen von Aufrufhäugigkeiten

Dieses Skript liest einen von Kieker.Analysis ereugten Trace in textueller Form (erzeugt mit *–print-Execution-Trace*) und erzeugt eine CSV-Datei mit allen Funktionen und ihren Aufrufhäufigkeiten.

```
use Getopt::Long;

my $inputfile = '';
my $outputfile = 'a.out';

GetOptions ("i|input|inputfile=s"   => \$inputfile,       # string
            "o|output|outputfile=s" => \$outputfile );    # string

unless ($inputfile) {
  die("Missing␣option␣-i");
}

open(my $inFile, "<", $inputfile) || die "Can't␣open␣$inputfile:␣$!";
open(my $outFile, ">", "$outputfile.csv") || die "Can't␣open␣$outputfile:␣$!";

my %funHash = ();
my %funHashTrace = ();
my $trace;

while (<$inFile>) {
  if (/0::@\d+:(\S+)\s/) {
    $funHash{$1} += 1;
    $funHashTrace{$1} += 1;
  } elsif ($trace && /^TraceId\s(\d+)/) {
    open(my $traceOutFile, ">", "$outputfile-$trace.csv")
      || die "Can't␣open␣$outputfile-$trace.csv:␣$!";
    while (my ($key, $value) = each %funHashTrace) {
```

```
      print $traceOutFile "$key;$value\n";
    }
    $trace = $1;
    %funHashTrace = ();
  } elsif (/^TraceId\s(\d+)/) {
    $trace = $1;
    %funHashTrace = ();
  }
}

open(my $traceOutFile, ">", "$outputfile-$trace.csv")
  || die "Can't open $outputfile-$trace.csv: $!";
while (my ($key, $value) = each %funHashTrace) {
  print $traceOutFile "$key;$value\n";
}

while (my ($key, $value) = each %funHash) {
  print $outFile "$key;$value\n";
}
```

## B.2. Aggregieren von Funktionsaufrufen

Mit diesem Skript werden die Trace-Daten modifiziert. Alle Funktionsnamen werden ignoriert und durch *agg()* ersetzt. Somit existiert nur noch eine Funktion pro Paket. Die Aggregation erfolgt vor der Verarbeitung durch Kieker, damit auch graphische Auswertungen durch Kieker erzeugt werden können.

```
use File::Copy;

my $inFile;

opendir(my $dh, ".") || die "can't opendir .: $!";
my @files = grep(/\.dat$/,readdir($dh));

foreach my $d (@files) {
  copy($d,$d.".bak");
  open($inFile, "<",$d.".bak");
  open(my $outFile, ">",$d);

  while (<$inFile>) {
    s/(;EPrints.*\.).*?(;EPrints)/$1agg$2/;
```

```
    print $outFile $_;
  }
}

closedir $dh;
```

# Glossar

*A*

*Array*
: Eine geordnete Menge von Variablen eines Datentyps. 5

*B*

*blessed variable*
: Das Perl Äquivalent eines Objektes. Beim *blessing* wird die Variable mit ihrer Klasse angereichert um als Objekt verwendet werden zu können. 5

*C*

*CPAN*
: Comprehensive Perl Archive Network, ein Netzwerk zur Verteilung von Perl-Modulen. 15, 17, 33

*E*

*EPrints*
: Open-Source Plattform zur Veröffentlichung von Dokumenten, `http://www.eprints.org/software`. 5, 46

*H*

*Hash*
: Eine Menge von Schlüssel/Wert-Paaren. 5

*J*

*JMS*
: Java Message Service, ist eine Schnittstelle zur Kommunikation mit einer Nachrichten-orientierten Middleware. JMS ist Teil der Java Enterprise Edition. 10, 13

*K*

*Kieker*
: Software Framework zur Durchführung von Monitoringaufgaben, `http://www.kieker-monitoring.net`. 1, 3, 7, 9–13, 19, 33, 34

*Kielprints*
Durch das GEOMAR angepasste Version von EPrints. Eingesetzt an der Universität Kiel. iii, 1, 3, 5, 19–21, 26–29, 31, 33, 34

*M*

*Monitoring*
Überwachung des Verhalten eines Programmes während der Laufzeit. 1, 3, 5

*Monitoring Probe*
Eingefügter Messpunkt in einem Programm zur Erzeugung von Monitoring Records. 3, 6, 15, 33, 46

*Monitoring Record*
Bei der Ausführung einer Monitoring Probe erzeugter Protokoll-Eintrag einer Messung. 3, 4, 7, 11–14, 20, 46

*P*

*Profiling*
Dynamisches Monitoringverfahren zum Erkennen von Funktionsverhalten in Anwendungen. 6, 7

*S*

*Skalar*
Eine Variable, die einen einzelnen Wert, z.B. eine Zahl oder einen String, speichert. 5

# Literaturverzeichnis


[Agarwal u. a. 2004] M. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi und A. Sailer. Problem Determination Using Dependency Graphs and Run-Time Behavior Models. In: *Utility Computing*. Herausgegeben von A. Sahai und F. Wu. Band 3278. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, Seiten 171–182. URL: http://dx.doi.org/10.1007/978-3-540-30184-4_15 (besucht am 20. 03. 2013). (Siehe Seite 6)

[*GNU General Public License, Version 3*]. GNU General Public License, Version 3. Juni 2007. URL: http://www.gnu.org/licenses/gpl.html (besucht am 20. 03. 2013). (Siehe Seite 5)

[Goerigk u. a. 2012] W. Goerigk, W. Hasselbring, G. Hennings, R. Jung, H. Neustock, H. Schaefer, C. Schneider, E. Schultz, T. Stahl, R. von Hanxleden, S. Weik und S. Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In: *Software Engineering*. Herausgegeben von S. Jähnichen, A. Küpper und S. Albayrak. Band 198. LNI. GI, 2012, Seiten 119–130. URL: http://dblp.uni-trier.de/db/conf/se/se2012.html#GoerigkHHJNSSSSHWZ12 (besucht am 20. 03. 2013). (Siehe Seite 13)

[*Java Message Service Specification*]. Java Message Service Specification. März 2002. URL: http://www.oracle.com/technetwork/java/docs-136352.html (besucht am 20. 03. 2013). (Siehe Seite 12)

[Kieker Project 2013] Kieker Project. Kieker 1.7-SNAPSHOT User Guide. Software Engineering Group, Kiel University, Kiel, Germany. März 2013. URL: http://kieker-monitoring.net/documentation/ (besucht am 04. 03. 2013). (Siehe Seiten 3, 4)

[Magedanz 2011] F. Magedanz. Dynamic analysis of .NET applications for architecture-based model extraction and test generation. Diplomarbeit. Department of Computer Science, University of Kiel, Germany, Okt. 2011. URL: http://eprints.uni-kiel.de/15486/ (besucht am 20. 03. 2013). (Siehe Seite 9)

[*mod_perl Documentation*]. mod_perl Documentation. URL: http://perl.apache.org/docs/index.html (besucht am 20. 03. 2013). (Siehe Seite 4)

[*perldata - Perl data types*]. perldata - Perl data types. URL: http://perldoc.perl.org/perldata.html (besucht am 20. 03. 2013). (Siehe Seite 5)

[*perlobj - Perl object reference*]. perlobj - Perl object reference. URL: http://perldoc.perl.org/perlobj.html (besucht am 20. 03. 2013). (Siehe Seite 5)

[*perlsyn - Perl syntax*]. perlsyn - Perl syntax. URL: http://perldoc.perl.org/perlsyn.html (besucht am 20. 03. 2013). (Siehe Seite 5)

[Richter 2012] B. Richter. Dynamische Analyse von COBOL-Systemarchitekturen zum modellbasierten Testen. Diploma Thesis, University of Kiel (work in progress). Diplomarbeit. Department of Computer Science, University of Kiel, Germany, Aug. 2012. URL: http://eprints.uni-kiel.de/15489/ (besucht am 20. 03. 2013). (Siehe Seiten 6, 9 und 13)

[Schmalenbach 2007] C. Schmalenbach. Performancemanagement für serviceorientierte Java-Anwendungen: Werkzeug- und Methodenunterstützung im Spannungsfeld von Entwicklung und Betrieb. Springer, 2007. URL: http://books.google.de/books?id=IbaEtgAACAAJ (besucht am 20. 03. 2013). (Siehe Seite 6)

[Siever u. a. 1997] E. Siever, L. Wall, B. Jepson, D. Futato und N. Patwardhan. Perl Resource Kit. O'Reilly Media, Nov. 1997. URL: http://oreilly.com/catalog/prkunix/excerpt/UGtoc.html (besucht am 20. 03. 2013). (Siehe Seite 9)

[*The Timeline of Perl and its Culture*]. The Timeline of Perl and its Culture. März 2013. URL: http://history.perl.org/PerlTimeline.html (besucht am 20. 03. 2013). (Siehe Seite 4)

[Van Hoorn u. a. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey und D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technischer Bericht TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009, 27 pages. (Siehe Seiten 3, 4)

[Van Hoorn u. a. 2012] A. van Hoorn, J. Waller und W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22–25, 2012: ACM, Apr. 2012, Seiten 247–248. (Siehe Seiten 3, 4)