Modularitätsbetrachtung von Webanwendungen im Rahmen des Plat_Forms Wettbewerbs

Masterarbeit am Institut für Informatik der Freien Universität Berlin,
Arbeitsgruppe Software Engineering

für die Prüfung zum

Master of Science

von

Andreas Franz

Matrikelnummer:	4400427
Abgabedatum:	30.11.2011
Erstgutachter:	Prof. Dr. Lutz Prechelt
Zweitgutachter:	Prof. Dr. Elfriede Fehr
Betreuer:	Ulrich Stärk

Kurzfassung

Die Arbeitsgruppe Software Engineering der Freien Universität Berlin hat den Plat_Forms-Wettbewerb initiiert, um verschiedene Webentwicklungsplattformen zu analysieren. Die Untersuchung erfolgt in Bezug auf Eigenschaften, wie beispielsweise Benutzbarkeit, Funktionalität, Größe und Struktur.

Mit Kenntnis über die Eigenschaften einer Plattform kann, entsprechend der Anforderungen eines Softwareprojektes, die geeignetste ausgewählt werden.

Zu diesem Zweck werden in dieser Masterarbeit die Möglichkeiten untersucht, die Modularität der Lösungen des Wettbewerbs zu ermitteln. Ein wichtiger zu beachtender Aspekt ist ein vergleichbares Vorgehen über alle Plattformen hinweg.

Auf Basis bestehender Ansätze und Konzepte zur Modularitätsbetrachtung wird eine neue Metrik hergeleitet, die die Kopplung der Lösungen misst. Die entwickelte Metrik wird auf die vorliegenden Webanwendungen angewendet. Darüber hinaus wird das Model-View-Controller-Architekturmuster herangezogen, um weitere Aspekte der Modularität zu beleuchten. In einer abschließenden Auswertung wird gezeigt, welche Rückschlüsse sich aus den Untersuchungen ableiten lassen.

Ehrenwörtliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 30. November 2011

Andreas Franz

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Der Plat_Forms-Wettbewerb 2
1.2 Aufgabenstellung und Vorgehensweise 3
1.3 Aufbau der Arbeit 4

2 Grundlagen **5**
2.1 Modularität und Module 5
2.2 Kohäsion und Kopplung 8
2.3 Verfügbare Ansätze für die Modularitätsbetrachtung 11
2.3.1 Metriken nach Chidamber und Kemerer 11
2.3.2 Metrik nach McCabe 13
2.3.3 Beurteilung der Modularität durch Neugruppierung von Modulen 13
2.3.4 Design Structure Matrix 14
2.3.5 Das Gesetz von Demeter 15
2.4 Struktur von Webanwendungen 16
2.4.1 Frameworks 16
2.4.2 Architekturmuster 18

3 Entwurf einer Metrik zur Betrachtung der Modularität **22**
3.1 Identifikation des darzustellenden Aspekts 23
3.2 Modellierung des darzustellenden Aspekts 23
3.3 Festlegung einer Skala für die Metrik 26
3.4 Entwicklung einer Berechnungsvorschrift 27
3.5 Entwicklung der Messvorschriften für alle Größen, die in der Berechnungsvorschrift verwendet werden 27
3.5.1 Regeln für die Abstrahierung der Platzhalter in Templates . 28
3.5.2 Sonderfälle bei der Auswertung 38
3.6 Anwendung und Verbesserung der Metrik 38
3.6.1 Vorgehen bei der Anwendung der Metrik 39
3.6.2 Analyse der eingesetzten Metrik 40

4 Architekturmuster als Gütekriterium der Modularität **43**
4.1 Umsetzung der Java-Teams 44
4.1.1 Spring 44
4.1.2 abaXX 45
4.1.3 JVx 45
4.2 Umsetzung des JavaScript-Teams 46
4.3 Umsetzung der Perl-Teams 46

4.4 Umsetzung der PHP-Teams 47
4.4.1 Symfony 47
4.4.2 Zend 49
4.4.3 Flow3 49
4.5 Umsetzung der Ruby-Teams 50
4.6 Verfahren für die Ermittlung von Verstößen gegen das MVC-Muster 50

5 Auswertung **53**
5.1 Auswertung der Referenzierungstiefe 53
5.1.1 Bewertung von Team D als Sonderfall 57
5.1.2 Anmerkungen zu Team O 57
5.2 Auswertung der Model-View-Controller-Verstöße 58

6 Zusammenfassung und Ausblick **62**

Abbildungsverzeichnis **64**

Abkürzungsverzeichnis **65**

Tabellenverzeichnis **66**

Quelltextverzeichnis **67**

Literaturverzeichnis **68**

Anhang **73**
A Datenformat 73
B CD-ROM 75

1 Einleitung

Webanwendungen bieten eine Reihe von Vorteilen gegenüber Anwendungen, die speziell für ein Betriebssystem entwickelt wurden. Der Einsatz eines zentralen Servers bietet die Möglichkeit, selbst mit leistungsschwachen Clients, von unterwegs auf Daten und Dienste zuzugreifen. Weiterer Vorteil ist die Möglichkeit einer zentralen Wartung. Produktupdates erreichen alle Nutzer gleichermaßen. Das spart Zeit und Entwicklungskosten gegenüber der Bereitstellung von Updates für verschiedene, betriebssystemspezifische Anwendungen. Mit der Verlagerung von Anwendungen ins Internet geht aber auch die Gefahr einher, dass Unbefugte Zugriff auf Daten erhalten. Aus einer Studie des Web Application Security Consortium[1] geht hervor, dass sich Sicherheitslücken in nahezu jeder Webanwendung automatisiert finden lassen [Gor08]. Für ein schnelles Gegensteuern bei erkannten Sicherheitslücken, sollte eine Webanwendung einfach zu warten sein. Nicht jede Webanwendung ist offen über das Internet verfügbar. Ein Einsatz in Intranets ist ebenso denkbar, jedoch müssen auch hier Aspekte wie Sicherheit, Wartbarkeit und Wirtschaftlichkeit beachtet werden. Die Architektur einer Software spielt dabei eine wesentliche Rolle. Durch sie wird eine Software in einzelne, leichter verständliche Komponenten gegliedert. Man bezeichnet diese Gliederung auch als Modularisierung. Das Verhalten der Anwendung nach außen bleibt davon unberührt.

Bei der Zerlegung einer Software kann man nach verschiedenen Kriterien vorgehen. David Parnas [Par72] hat dazu bereits 1972, aussagekräftige Untersuchungen gemacht. Eine Möglichkeit der Modularisierung ist die *Zerlegung auf Basis*

[1] Ziel des WASC ist die methodische Identifizierung von Sicherheitslücken, mittels entwickelter Testmethodiken, in Webanwendungen.

eines Flussdiagramms. Dabei ergeben sich die einzelnen Module auf Grundlage der aufeinanderfolgenden Schritte im Programmablauf. Dieser Ansatz kann zu starken Abhängigkeiten der einzelnen Module zueinander führen. Als deutlich besseres Kriterium beschreibt Parnas das *Verbergen von Entwurfsentscheidungen* (information hiding). Demnach sollten die Entwurfsentscheidungen „verborgen" werden, die sich am wahrscheinlichsten ändern. Im Falle einer Änderung ist dann mit großer Wahrscheinlichkeit nur ein Modul betroffen.

Für Software-Entwickler ergibt sich die Fragestellung, ob durch die Auswahl der Programmierplattform die Modularität wesentlich beeinflusst wird. Dieser und weiteren Fragestellungen, bezüglich des Vergleichs von Programmiersprachen, widmet sich die Arbeitsgruppe Software Engineering, der Freien Universität Berlin. Der Plat_Forms-Wettbewerb ist ein Ansatz, diese Fragestellungen zu ergründen.

1.1 Der Plat_Forms-Wettbewerb

Mit Hilfe des Plat_Forms[2]-Wettbewerbs sollen die Charakteristiken verschiedener Programmiersprachen untersucht werden. Insbesondere geht es um deren Einfluss auf die Eigenschaften von entwickelten Webanwendungen. Zu diesen Eigenschaften zählen externe Aspekte wie Benutzbarkeit, Funktionalität, Verlässlichkeit. Ebenso interne wie beispielsweise Größe, Struktur, Flexibilität oder Modifizierbarkeit von Anwendungen. Ein besseres Verständnis dieser Eigenschaften kann bei der Auswahl einer geeigneten Plattform für die Entwicklung einer Webanwendung hilfreich sein. Durch die Auswahl hochqualifizierter Teams wird versucht, ein hohes Maß an Vergleichbarkeit zu gewährleisten. Teams zu je drei Personen hatten die Aufgabe, auf Basis identischer Anforderungen eine Webanwendung zu entwickeln [Pre07, S. 1].

Erstmals wurde der Wettbewerb im Jahr 2007 durchgeführt und ausgewertet. Je drei Teams nutzten Perl, Java, beziehungsweise PHP für die Umsetzung ihrer Lösungen. In der Auswertung (vgl. dazu [Pre07]) wurden verschiedenste Aspekte der Lösungen untersucht. Zum Beispiel wurden Benutzbarkeit, Produktgröße,

[2] http://www.plat-forms.org

Skalierbarkeit, bzw. Performanz u.ä. beleuchtet. In Bezug auf die Modularität wurde der Ansatz einer dynamischen Codeanalyse gewählt. Hierbei wurden die während der Ausführung stattfindenden Aufrufe, zwischen Anwendung und Framework, mit einem sogenannten Profiler ermittelt. Auf dieser Grundlage wurde der Zusammenhang zwischen aufrufender und aufgerufener Methoden ermittelt. Die Analyse zeigte, dass bereits große Unterschiede zwischen den Perl-Teams existierten. Es erschien deshalb unwahrscheinlich Charakteristiken zwischen den Plattformen ermitteln zu können [Pre07, S. 62].

Vom 18. bis. 19. Januar 2011 wurde der Wettbewerb erneut durchgeführt. Aus den eingegangenen Bewerbungen erfolgte die Auswahl der Teams auf Basis der angegebenen Qualifikationen. Je vier Teilnehmerteams nutzten Java, Ruby beziehungsweise PHP, außerdem drei Perl und eines JavaScript. Die Teams, bestehend aus je drei Personen, hatten die identische Aufgabenstellung – die Entwicklung einer Webanwendung für die Verwaltung von Konferenzen und deren Teilnehmer. Die Anwendung soll die Möglichkeit bieten, dass Nutzer Konferenzen anlegen. Diese Konferenzen sollen sich in Kategorien einordnen lassen. Wiederkehrende Konferenzen sollen durch das Anlegen einer Serie realisiert werden können. Ebenso soll eine Suche und eine Möglichkeit zum Durchstöbern der Konferenzen geschaffen werden. Ein Nutzer soll seine Teilnahme an Konferenzen bestätigen und dann in seinem Kalender organisieren können. Neben der Darstellung der Benutzeroberfläche im Browser sollen auch der Export bestimmter Daten (wie Kalendereinträge) und der Zugriff via RESTful Webservice-Schnittstelle realisiert werden (vgl. hierzu das Anforderungsdokument [SP11]).

1.2 Aufgabenstellung und Vorgehensweise

Mit dieser Arbeit soll eine Möglichkeit entwickelt werden, die Modularität der Lösungen des Wettbewerbs zu untersuchen. Dafür muss ein Verfahren gefunden und evaluiert werden. Dieses Verfahren muss eine vergleichbare Vorgehensweise auf alle Plattformen anwenden. Nur so können auch Vergleiche zwischen den Plattformen gezogen werden. Darüber hinaus muss das entwickelte Verfahren

allgemein anwendbar sein und sich auf andere Programmiersprachen erweitern lassen. Damit sollen die Eigenschaften bezüglich der Modularität der Plattformen aufgezeigt werden.

Zunächst werden die etablierten Konzepte und Verfahren zur Auswertung der Modularität von Anwendungen untersucht und bewertet. Daneben werden die Quelltexte der Lösungen untersucht und verwendete Frameworks näher beleuchtet. Die gewonnenen Erkenntnisse dienen dann als Ausgangspunkt für die Entwicklung des Untersuchungsverfahrens. Dieses Verfahren wird dann im Einzelnen beschrieben und auf die Lösungen angewendet. Die Resultate werden im Abschluss gegenübergestellt.

1.3 Aufbau der Arbeit

Der Fokus des zweiten Kapitels liegt auf der Schaffung eines theoretischen Grundverständnisses. Einerseits werden wichtige Begrifflichkeiten, die im Rahmen der Arbeit verwendet werden, eingeführt und erläutert. Andererseits erfolgt die Vorstellung und Bewertung verbreiteter und häufig angewendeter Verfahren zur Analyse der Modularität von Software. Gegenstand des dritten Kapitels ist der Entwurf eines Verfahrens zur Analyse der Modularität, der vorliegenden Lösungen des Wettbewerbs. Die einzelnen Schritte der Datenerhebung werden erläutert. Das vierte Kapitel beinhaltet eine Betrachtung der Struktur der Lösungen anhand des Architekturmusters Model-View-Controller (MVC). Im fünften Kapitel erfolgt dann die Anwendung der Verfahren. Mit den erhobenen Daten und Strukturbetrachtungen wird eine Auswertung vollzogen. Zum Abschluss werden die gewonnenen Erkenntnisse im sechsten Kapitel zusammenfassend dargestellt.

2 Grundlagen

In diesem Kapitel werden für die weitere Betrachtung wichtige Begriffe näher erläutert, außerdem wird eine Einführung in die verwendeten Grundkonzepte gegeben. Im Weiteren werden die etablierten Verfahren zur Modularitätsbestimmung vorgestellt.

Im Kapitel 1 auf Seite 1 wurde Modularisierung grob als Gliederung einer Software in einzelne, leichter verständliche Komponenten beschrieben. Nun werden die Begriffe Module und Modularität näher erläutert.

2.1 Modularität und Module

Baldwin und Clark [BC03] beschreiben **Modularität** allgemein als eine Eigenschaft, nach der sich Produkte und Prozesse strategisch organisieren lassen. Für eine effiziente Organisation wird das Gesamtsystem in kleinere Einheiten, so genannte Module, aufgeteilt. Das Softwaresystem bleibt in seiner Funktionalität durch die Modularisierung unberührt.

Es stellt sich die Frage, nach welchem Kriterium entschieden werden soll, was zu einem Modul zusammengefasst wird. Baldwin und Clark schlagen das gezielte Kapseln von Entwurfsentscheidungen vor. Das bedeutet: Module verbergen alle Informationen, die nur für einen modulinternen Ablauf benötigt werden. Anderen Modulen wird eine definierte Schnittstelle, mit den Operationen die das Modul bietet, zur Verfügung gestellt. Dabei verfolgt man folgende Ziele [Par72] [BC03]:

- Module sollen sich weitestgehend unabhängig voneinander entwickeln lassen
- Komplexe Produkte verständlicher machen
- Austauschbarkeit einzelner Module soll gewährleistet werden
- Entwurfsentscheidungen sollen so gekapselt werden, sodass Änderungen nur innerhalb eines Moduls vorgenommen werden müssen

Letzteres Ziel beschreibt das Prinzip des *information hiding* nach Parnas [Par72]. Ändert sich die Schnittstelle eines Moduls durch die veränderte Anforderung nicht, so muss nur dieses eine Modul modifiziert werden.

Der Begriff **Modul** wird in der Literatur aus dem Bereich des Software Engineering unterschiedlich interpretiert.

Constantine und Yourdon [YC87, S. 37] definieren ein Modul als zusammenhängende Abfolge von Programmanweisungen, die durch Begrenzungselemente getrennt sind und einen gemeinsamen Bezeichner haben. Ähnlich heißt es nach IEEE-Definition [IEE90, S. 49]: Ein Modul ist ein logisch separierbarer Teil eines Programms.

Offutt et al. [OHK93] sowie Lakhotia [Lak93] beschreiben Module aus der Sicht prozeduraler Programmiersprachen lediglich als einzelne Funktionen.

Im Kontrast dazu stehen die Definitionen von Mitchell und Mancoridis [MM06] sowie Hitz und Montazeri [HM96]. Bei ihren Untersuchungen im objektorientierten Umfeld sind Module stets einzelne Klassen.

Eine (über Klassengrenzen) weitergreifende Definition gibt Zuse [Zus98, S. 31]: Module sind eine Sammlung von Programmanweisungen mit folgenden Eigenschaften:

- Sie besitzen eine Ein- und Ausgabe von Daten
- Sie bestehen aus Funktionen

- Sie besitzen interne Daten

Demnach trifft die Definition laut Zuse auch auf Pakete, Funktionen, Mechanismen, Prozeduren, u.ä. zu.

Bei Sarkar et al. [SRK07] ist ein Modul eine „Gruppierung von Dateien, Funktionen, Datenstrukturen und Variablen zu einer (logisch) kohäsiven[3] Einheit".

Einige der Definitionen sind bewusst sehr allgemein gehalten, ohne dabei Bezug zu nehmen, welche Konsequenzen sich daraus für die Güte der Modularität ergeben. Andere Definitionen wiederum beziehen sich stark auf bestimmte Elemente, zum Beispiel aus der objektorientierten Programmierung.

Die Umsetzung objektorientierter Konzepte in den Programmiersprachen der teilnehmenden Teams ist sehr verschieden (falls überhaupt vorhanden). Grundsätzlich ist es möglich, viele der Sprachen auch für eine rein prozedurale Programmierung einzusetzen. Aus diesem Grund folgt eine für das Umfeld dieser Arbeit gültige Definition, die die wichtigsten Elemente obiger Definitionen zusammenfasst.

Definition 1 (*Modul:*)

Ein Modul umfasst ein oder mehrere Dateien (Quelltexte, Templates) mit folgenden Eigenschaften:

- *Das Modul bewerkstelligt seine Aufgaben unter Zuhilfenahme von nach außen gekapselten Daten (Blackbox-Prinzip)*
  - *Gekapselte Daten sind nur für das Modul, in dem sie definiert sind, aber nicht für andere Module verfügbar*
- *Verfügbare Operationen werden nach außen bereitgestellt, interne verborgen*

[3]siehe Abschnitt 2.2 auf der nächsten Seite

2.2 Kohäsion und Kopplung

Nachdem im Abschnitt 2.1 auf Seite 5 Modularität näher erläutert wurde, folgt nun eine Einordnung von Faktoren, mit denen sich die Eigenschaften guter Modularisierung beschreiben lassen. Die Begriffe Kopplung und Kohäsion wurden erstmals von Constantine und Yourdon [YC87, S. 85 ff.] eingeführt. Mit ihnen lassen sich die gewünschten Eigenschaften von Modulen charakterisieren.

Kohäsion beschreibt den Zusammenhang innerhalb eines Moduls. Ein Modul sollte *eine* klar abgegrenzte Zuständigkeit haben, also eine Aufgabe die es erfüllt. Demnach sollten auch alle Elemente des Moduls nur zur Erfüllung dieser Zuständigkeit dienen.

Constantine und Yourdon haben eine Einteilung in sieben Stufen entwickelt, die die Kategorisierung der Kohäsion ermöglicht [YC87, S. 111 ff.]:

Zufällige Kohäsion:
: Das Modul fasst zufällig aufeinander folgende Verarbeitungsschritte zusammen.

Logische Kohäsion:
: Die Verarbeitungsschritte haben eine logische Zusammengehörigkeit, zum Beispiel „Einlesen von Daten".

Zeitliche Kohäsion:
: Die Elemente des Moduls werden zur gleichen Zeit ausgeführt, beispielsweise zur Initialisierung.

Prozedurale Kohäsion:
: Hierbei handelt es sich um eine Zusammenfassung von Programmschritten zu Verarbeitungsblöcken. Das können zum Beispiel Schleifen sein. Häufig entstehen solche Blöcke als direkte Umsetzung von Flussdiagrammen (aus der Entwurfsphase).

Kommunikative Kohäsion:
: Alle Elemente des Moduls arbeiten auf den gleichen Eingabedaten oder erzeugen die gleichen Ausgabedaten.

Sequentielle Kohäsion:

Sind die Ausgabedaten eines Verarbeitungselements die direkten Eingabedaten für ein anderes, so sind diese Elemente sequentiell kohärent.

Funktionale Kohäsion:

Die Elemente sind funktional zusammenhängend, falls sie genau eine Problemstellung lösen.

Ein Modul gehört zu der Kategorie, die auf *alle* Elemente des Moduls zutrifft. Die Einteilung ist aufsteigend von niedriger, unerwünschter bis hin zu gewünschter Kohäsion. Anmerkend ist zu erwähnen, dass die Einteilung nicht linear ansteigend ist. Die ersten drei Stufen sollten vermieden werden, die letzten drei sind akzeptabel (nach [YC87]). Je höher der Grad der Kohäsion desto besser wurde die reale Problemstruktur abgebildet. Die Problemstruktur ergibt sich aus den gegebenen Anforderungen an eine Software. Wird je ein Teilproblem durch ein separates Modul gelöst, ist jedes Modul für sich stark zusammenhängend und somit kohäsiv.

Kopplung bezeichnet die Verbindung zwischen Modulen. Für eine möglichst unabhängige Entwicklung von Modulen sollten diese lose gekoppelt sein. Darüber hinaus ist ein Teilsystem leichter zu verstehen, zu warten und weiter zu entwickeln, als ein komplexes zusammenhängendes System als Ganzes.

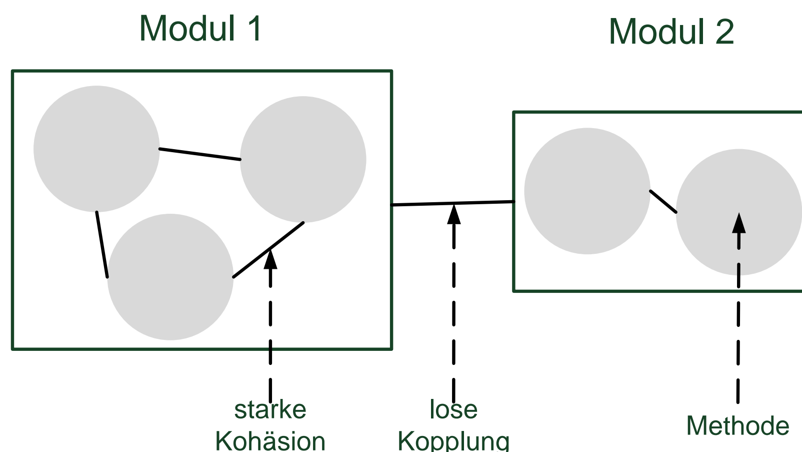


Abbildung 2–1: Zusammenhang von Kopplung und Kohäsion (Anpassung an [LL07, S. 387])

Abbildung 2–1 auf der vorherigen Seite veranschaulicht den Zusammenhang von Kopplung und Kohäsion zwischen Modulen. Schematisch repräsentieren die Vierecke die Module (1 und 2). Die enthaltenen Methoden (graue Kreise) besitzen idealerweise eine starke Kohäsion, während die Module untereinander lose gekoppelt sind.

Für die Einteilung der Kopplung schlagen Yourdon und Constantine folgende Kategorisierung vor (vgl. auch zum Folgenden [YC87, S. 87 ff.]). Die Rangfolge reicht von erwünschter, loser Kopplung hin zu unerwünscht starker Kopplung.

Datenkopplung:
Eine Kopplung über Daten besteht, wenn zwei Module mittels Parameter miteinander kommunizieren.

Kontrollkopplung:
Beeinflussen die Daten eines Moduls die interne Logik eines anderen, so spricht man von Kontrollkopplung.

Globale Kopplung:
Greifen zwei oder mehr Module auf eine global verfügbare Gruppe von Daten zu, sind diese Module global gekoppelt.

Pathologische Kopplung:
Diese Art liegt vor, wenn ein Modul auf interne Daten eines anderen Moduls zugreift.

Im Sinne einer guten Modularisierung mit gleicher Funktionalität wie ein nicht modularisiertes System ist die **Datenkopplung** die einzig notwendige Kopplung. Alle anderen Arten sind zu vermeiden, speziell die **pathologische Kopplung**. Eine derartige Verbindung liegt vor, falls ein Modul nicht über die Schnittstelle mit einem anderen Modul kommuniziert, sondern direkt auf Daten innerhalb des Moduls zugreift. Das kann möglich sein, falls die internen Daten eines Moduls nicht gegen den Zugriff von außen geschützt wurden. Zum einen wird das Prinzip des *information hiding* dadurch verletzt und zum anderen bedeutet es, dass das Modul nicht mehr als Blackbox betrachtet werden kann, da interne Implementierungsdetails über die Modulgrenze hinweg sichtbar sind.

Bei der **pathologischen Kopplung** eines Moduls an ein anderes wird eine starke Abhängigkeit erzeugt. Das Modul, das die pathologische Verbindung verwendet, verlässt sich auf die interne Implementierung des anderen. Ändert sich diese, muss auch das daran gekoppelte Modul höchst wahrscheinlich angepasst werden. Die Austauschbarkeit des Moduls wird verringert.

2.3 Verfügbare Ansätze für die Modularitätsbetrachtung

Ein Softwareprodukt anhand seiner Merkmale zu charakterisieren ermöglicht es, dieses zu bewerten und gegebenenfalls zu verbessern. Definiert man die Erfassung der Merkmale sorgfältig, wird es möglich verschiedene Produkte vergleichbar zu machen.

Den Vorgang der Datenerfassung bezeichnet man als *Messen*. Die Abbildung der Software auf ein Merkmal erfolgt durch eine *Metrik*[4] [LL07].

Für die Messung der Modularität von Anwendungen im Allgemeinen gibt es eine Reihe bestehender Ansätze. Diese werden im Folgenden vorgestellt und auf ihre Eignung hin untersucht.

2.3.1 Metriken nach Chidamber und Kemerer

Chidamber und Kemerer haben eine Sammlung von Metriken für die Bewertung objektorientierter Entwürfe entwickelt. Die Metriken haben eine weite Verbreitung und wurden auch von Anderen weiterentwickelt (vgl. auch zum Folgenden [CK94]).

Depth in inheritance tree (Tiefe im Vererbungsbaum) beschreibt den längsten Pfad von der Wurzel im Vererbungsbaum bis zur aktuellen Klasse. Diese Metrik

[4] Metrik wird nicht, wie im mathematischen Sinne, als Distanzfunktion verstanden, sondern in der umgangssprachlichen Bedeutung „Maß“ [LL07]

zielt auf den Sachverhalt ab, dass eine Klasse die tiefer im Vererbungsbaum ist, mehr Vorgänger hat. Das gängige Vererbungskonzept sieht vor, dass eine Klasse alle Attribute und Methoden seiner Oberklassen erbt, also von allen Klassen die im Vererbungsbaum auf dem Pfad zur Klasse liegen. Je mehr Methoden und Attribute es nun sind, die eine Klasse erbt, desto stärker hängt sie auch von der konkreten Implementierung ihrer Vorgänger ab. Fehler sind potentiell schwerer zu finden und Änderungen somit schlechter durchzuführen.

Den umgekehrten Sachverhalt beleuchtet die Metrik **number of children** (Anzahl der Kinder). Es wird die Anzahl, der unmittelbar im Vererbungsbaum folgenden Klassen ermittelt. Somit zeigt sich, auf wie viele Klassen die Oberklasse Einfluss hat.

Neben den Metriken zur Vererbung gibt es weitere bezüglich der Kommunikation zwischen Klassen. Dazu zählen **weighted methods per class (WMC)** (gewichtete Methoden einer Klasse) und **coupling between object classes (CBO)** (Kopplung zwischen Objektklassen). *WMC* gewichtet jede Methode einer Klasse mit einem Faktor, der die Komplexität der Methode beschreibt. Komplexe Methoden erschweren Tests und Fehlerbeseitigung. *CBO* ist die Anzahl der Klassen mit der eine Klasse gekoppelt ist. Durch eine starke Kopplung wird die Wiederverwendung und Austauschbarkeit einer Klasse reduziert, weil durch die Kopplung Abhängigkeiten erzeugt wurden.

Wird eine Methode eines Objektes einer Klasse aufgerufen kann es sein, dass weitere Methoden für die Bearbeitung aufgerufen werden. Die Zahl der Methoden, die über die eigentlich aufgerufene Methode hinausgeht, bezeichnet man als **Response for class** (Anwort einer Klasse). Konsequenzen daraus sind: Je höher die Zahl der aufgerufenen Methoden, desto schwieriger ist das Testen der Methoden. Die Fehlerbeseitigung wird ebenfalls durch die erhöhte Komplexität erschwert.

Lack of cohesion in modules (fehlende Kohäsion in Modulen) beschreibt die Anzahl der Methoden, die keine gemeinsamen Variablen benutzen, abzüglich der Methoden, die auf gemeinsame Variablen zugreifen. Diese Metrik zielt auf den Zusammenhang (Kohäsion) innerhalb eines Moduls ab. Die Kohäsion lässt

sich, wie im Abschnitt 2.2 auf Seite 8 beschrieben, in Stufen kategorisieren. Demzufolge kann ein hoher LCOM-Wert andeuten, dass die Methoden eines Moduls sehr unterschiedliche Daten verarbeiten. Demnach ist zu prüfen, ob eine Neustrukturierung der Module sinnvoll sein könnte.

Der Vergleich von Anwendungen verschiedener Programmiersprachen mit diesen Metriken kann sich als schwierig herausstellen. Dafür gilt es zunächst zu überprüfen, wie die Sprachen die objektorientierten Konzepte umsetzen. Ruby ist beispielsweise vollständig objektorientiert ausgelegt – alles ist ein Objekt (vgl. [TFH04, S. 9 ff.]). Perl hingegen bietet sowohl die Möglichkeit einer prozeduralen als auch objektorientierte Programmierung [CW00, S. 335].

2.3.2 Metrik nach McCabe

Cyclomatic complexity bezeichnet die Anzahl der Knoten und Kanten im Kontrollflussgraph. Der Graph repräsentiert alle möglichen Entscheidungen im Programmablauf. Die Metrik beruht auf der Annahme, dass ein Modul mit vielen Verzweigungen komplex ist. Ein zu komplexes Modul ist schlecht zu erfassen und Änderungen nur schwer umzusetzen [McC76].

Diese Metrik lässt sich auf Anwendungen verschiedenster Programmiersprachen gleichermaßen anwenden. Dadurch wird es möglich die Komplexität dieser zu bewerten. Problematisch ist aber, dass die Komplexität nur teilweise Aufschluss über die Kohäsion der Module geben kann. Der intermodulare Zusammenhang bleibt komplett unberücksichtigt.

2.3.3 Beurteilung der Modularität durch Neugruppierung von Modulen

Harman et al. [HHP02] schlagen eine Bewertung der Modularisierung durch einen suchbasierten Ansatz vor. Einzelne Komponenten (Prozeduren, Variablen) werden der Reihe nach Modulen zugeordnet. Die Auswahl der Komponente, die

als nächstes zugeordnet werden soll, erfolgt durch einen speziell entwickelten *Crossover-Operator*. Dieser sorgt dafür, dass Vererbungsbeziehungen erhalten bleiben. Die Bewertung, wie gut die Modularisierung ist, erfolgt mit einer Fitnessfunktion. Die Funktion basiert auf der Kohäsion, Kopplung (vgl. Abschnitt 2.2 auf Seite 8) und Granularität. Die Granularität beschreibt die Anzahl von Modulen, die ein Modul nutzt.

Die Autoren merken jedoch selbst an, dass das Verfahren stark von der Wahl der gewünschten Granularität abhängt. Diese muss nämlich selbst abgeschätzt und gewählt werden. Des Weiteren ist die Lösung nicht zwingend optimal und eindeutig.

2.3.4 Design Structure Matrix

Mit einer Design Structure Matrix (DSM) lassen sich die Abhängigkeiten von Entwurfsparametern visualisieren. Entwurfsparameter sind die Einflussfaktoren, die der Softwareentwickler beim Systementwurf festlegt. Das kann beispielsweise die Festlegung auf einen bestimmten Algorithmus, eine Datenstruktur, Technologie oder die Schnittstelle eines Moduls sein. Abhängigkeiten zwischen Entwurfsparametern entstehen, wenn die Wahl eines Parameter die Möglichkeiten der Auswahl eines anderen Parameters einschränken [Pre10]. Abbildung 2–2 zeigt ein Beispiel einer solchen Matrix.

	A	B	C
A	.		
B	x	.	x
C		x	.

Abbildung 2–2: Beispiel einer Design Structure Matrix [SGCH01]

In der ersten Spalte und Zeile werden die Entwurfsparameter eingetragen. An den Kreuzungspunkten lässt sich ablesen, ob zwei Parameter voneinander abhängen.

In Abbildung 2–2 auf der vorherigen Seite zeigt sich, dass B Informationen von A und C benötigt, um seine Aufgaben zu bewerkstelligen. Ebenso benötigt C Informationen von B. Dies wird durch ein „x“ gekennzeichnet [Epp91].

Die Matrix ermöglicht eine Bewertung, wie stark ein System gekoppelt ist. Je nachdem wie wahrscheinlich es ist, dass sich bestimmte Entwurfsparameter ändern, kann auf Basis der DSM eine (Neu-)Modularisierung angestrebt werden.

Die Verwendung einer DSM eignet sich am besten für den Einsatz während der Entwurfsphase eines Projekts. Es können dann verschiedene Entwürfe einander gegenübergestellt, und Ansatzpunkte für eine Modularisierung gefunden werden. Die Anwendung auf ein fertiges Produkt ist schwieriger. Es müssen zunächst alle Entwurfsparameter des Produkts ermittelt werden. Danach kann eine Auswertung der Abhängigkeiten erfolgen, die dann zu einer DSM führt.

2.3.5 Das Gesetz von Demeter

Im Abschnitt 2.2 auf Seite 8 wurde die Kategorisierung der Kopplung nach Yourdon und Constantine vorgestellt. Als stärkste Art der Kopplung wurde die *pathologische* Kopplung aufgezeigt.

Dass diese Kopplungsart zu vermeiden ist schildern auch Lieberherr et al. in [LHR88]. Ihre Beobachtungen aus einem Projekt namens Demeter fassen sie im *Gesetz von Demeter* (law of demeter) zusammen. Dieses besagt, dass eine Klasse C und ihre Methoden M ausschließlich mit:

- Argumenten der Methoden M (inklusive Klasse C)
- Instanzvariablen von Klasse C

kommunizieren dürfen. Neben dieser objektorientierten Formulierung, existiert auch eine allgemeinere: „Objekte[5] sollen nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren“ [LH89].

[5] Der Objektbegriff zielt hier aber nicht auf die objektorientierte Programmierung ab, sondern ist eher im Sinne von Modul zu verstehen

Hitz und Montazeri [HM96] bestätigen, dass eine Verletzung des Gesetzes von Demeter die Kopplung der Module erhöht, bzw. die Befolgung der selbigen die Kopplung reduziert. Das geschieht, weil die Anzahl der von einem Modul angesprochenen Module reduziert wird. Durch diese Vereinfachung der Struktur und der damit verbundenen Übersichtlichkeit wird auch die Anfälligkeit für Fehler während der Entwicklung reduziert.

Wie schon im Abschnitt 2.3.1 auf Seite 11 beschrieben, ist die Umsetzung der Objektorientierung in Programmiersprachen unterschiedlich interpretiert. Für eine Anwendung der Metrik müssen diese Konzepte abstrahiert werden. Dieser Abstraktionsschritt wird im Abschnitt 3.5 auf Seite 27 hergeleitet.

2.4 Struktur von Webanwendungen

Abbildung 2–3 auf der nächsten Seite zeigt die typische Architektur einer Webanwendung. Der Anwender interagiert mit dem System über einen Internetbrowser. Dieser kommuniziert mit dem Web-Server über das HTTP-Protokoll. Die Funktionsbereiche des HTTP-Servers umfassen neben Verbindungs-, Anfrage- und Sicherheitsaufgaben auch die Ressourcenverwaltung. Entsprechend wird eine Anfrage vom Ressourcenmanager ausgewertet. Entweder liefert die Webanwendung statische Dateien aus dem Dateisystem oder dynamisch erzeugte Inhalte zurück. Die Webanwendung kann hierzu (falls nötig) weitere Daten aus einer Datenbank laden. Der erzeugte Inhalt ist bei der Verwendung eines Browser üblicherweise ein HTML-Dokument, aber auch andere Ausgabeformate sind denkbar und möglich.

2.4.1 Frameworks

Bei der Entwicklung von Anwendungen gibt es eine Reihe von Funktionalitäten, die ein Entwickler immer wieder benötigt. Für solche Zwecke bietet sich die Ver-

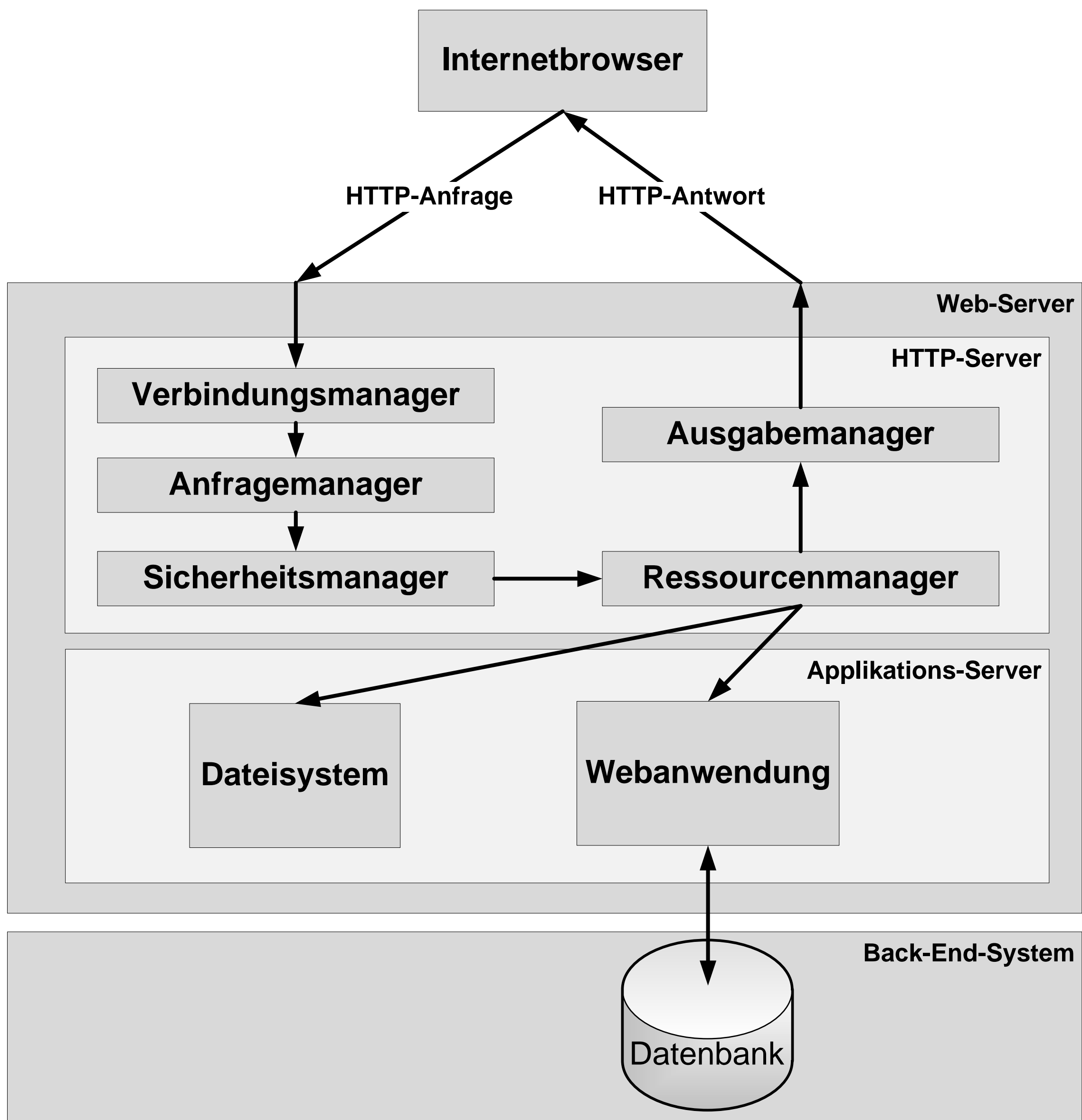


Abbildung 2–3: Funktionale Struktur eines Webservers (Anpassung an [JPMM04, Abb. 2.11., S. 23])

wendung eines Frameworks an. Ein Framework[6] gibt eine Grundstruktur vor und nimmt dem Entwickler eine Vielzahl von technischen Details ab.

Das kann beispielsweise das automatische Speichern von Änderungen in der Datenbank sein, ohne dass der Software-Entwickler datenbankspezifischen Quelltext schreiben muss. Ebenso existieren Frameworks für die Generierung eines darstellbaren Ausgabeformats der Daten auf der Grundlage von Templates. Des Weiteren kann die vorgegebene Struktur einem Architekturmuster (wie beispielsweise das MVC-Muster) folgen und dem Anwendungsentwickler eine Grundlage für die Organisation seines Quelltextes geben.

Der Entwickler nutzt die vom Framework bereitgestellten Schnittstellen und erweitert so die Funktionalität um die eigenen Anforderungen. Nach außen ist ein Framework geschlossen für Änderungen, aber offen für Erweiterungen [LR06, Kap. 3.4] – prinzipiell also eine konsequente Umsetzung einer Modularisierung.

2.4.2 Architekturmuster

Webanwendungen zeichnen sich besonders durch die Interaktion des Benutzers mit der Anwendung aus. Für die Strukturierung solcher interaktiven Systeme existieren eine Reihe von Architekturmustern. Alle für die Benutzer-Interaktion relevanten Komponenten bezeichnet man zusammenfassend als Präsentationsschicht. Eines der etablierten Architekturmuster für die Strukturierung der Präsentationsschicht ist das Model-View-Controller Architekturmuster (kurz: MVC-Muster).

Das Architekturmuster Model-View-Controller

Beim MVC-Muster erfolgt eine Aufteilung nach drei bestimmten Rollen (vgl. dazu [LR06, Kap. 8.2] und [Fow03, S. 330]).

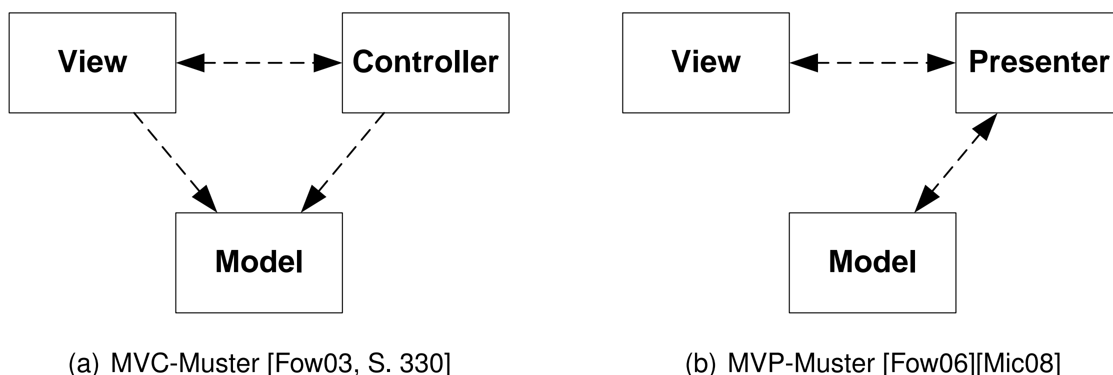
[6] engl. für Rahmenwerk/Grundgerüst

1. Das **Model** enthält die Daten, die in der View dargestellt werden sollen. Darüber hinaus reagiert es auf Aufforderungen zur Änderung der Daten und verwaltet somit den aktuellen Zustand.

2. Die **View** dient zur Darstellung der Informationen des Models. Sie ist visuelle Schnittstelle für den Benutzer. Falls möglich sollte die View keine fachliche Logik enthalten.

3. Der **Controller** schottet Eingabemechanismen von der Programmlogik ab. Er nimmt die Benutzereingaben entgegen und veranlasst das Model zur Änderung der Daten. Zum anderen sorgt er für Aktualisierungen der View, sobald sich das Model ändert.

Ursprünglich entstammt das MVC-Muster eines Konzepts, das bei der Entwicklung von Smalltalk durch Trygve Reenskaug beschrieben wurde. Das Konzept beschreibt die Idee, Benutzerinteraktionen in der Präsentationsschicht voneinander zu trennen [Ree79]. Die entstehende Aufteilung zeigt Abbildung 2–4(a) schematisch. Die Pfeilrichtungen zeigen an, welche Komponente Kenntnis über eine andere besitzt.



(a) MVC-Muster [Fow03, S. 330]

(b) MVP-Muster [Fow06][Mic08]

Abbildung 2–4: MVC- und MVP-Architekturmuster

Martin Fowler [Fow03] beschreibt, dass die Trennung von View, Controller und Model eine der „wichtigsten Heuristiken guten Software Designs“ ist. Einerseits, weil Präsentation[7] und Model unterschiedliche Belange abdecken. Andererseits kann es durchaus vorkommen, dass ein Model durch verschiedenartige oder

[7] Fowler bezeichnet View und Controller zusammen als Präsentation

mehrere Views dargestellt wird, was durch eine gute Trennung erleichtert wird. Als weiteren Punkt nennt er die bessere Testbarkeit von nicht visuellen Komponenten im Gegensatz zur Präsentation. Aufgrund der Tatsache, dass die Präsentation vom Model abhängt, aber dies umgekehrt nicht der Fall ist, kommt er zu einer Erweiterung des klassischen MVC-Architekturmusters - dem Model-View-Presenter-Architekturmuster (kurz: MVP-Muster).

Schematisch zeigt Abbildung 2–4(b) auf der vorherigen Seite den Zusammenhang der Komponenten Model, View und Presenter. Wiederum weisen die Pfeile aus, welche Komponente Kenntnis von einer anderen hat. Bei diesem Architekturmuster übernimmt die View ausschließlich die Darstellung von Informationen. Anwendungsspezifisches Verhalten wird vollständig in den Controller verlagert. Aus diesem Grund bezeichnet Fowler diese Variation auch als „Passive View“ [Fow06].

Es ergeben sich eine Reihe von Vorteilen bei der Nutzung dieses Architekturmusters [Mic08]: Ein Großteil des Codes (die View ausgenommen) lässt sich automatisiert testen. Bei Views mit gleichem Verhalten lässt sich der Presenter wiederverwenden. Außerdem wir durch die separierte Anwendungslogik der Quellcode lesbarer und ist somit besser zu warten. Weiterhin besitzt die View keine Kenntnis vom Model. Der Controller reicht nur die notwendigen Informationen an die View weiter. Das Prinzip des *information hiding* (vgl. Abschnitt 2.1 auf Seite 5) wird dabei verfolgt.

Das Front-Controller-Muster

Das Front-Controller-Muster findet Einsatz im Bereich von Webanwendungen. Zum Teil findet sich deshalb auch die Bezeichnung *Web MVC*.

Abbildung 2–5 auf der nächsten Seite zeigt die Grundkomponenten. Eingehende HTTP-Anfragen vom Client bearbeitet der Front Controller. Dieser bestimmt, welcher seitenspezifische Controller für die Anfrage zuständig ist. An diesen Controller wird die Anfrage entsprechend weitergeleitet.

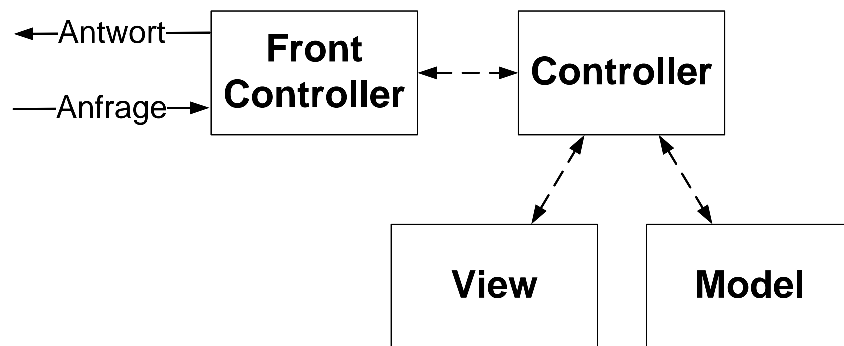


Abbildung 2–5: Das Front-Controller-Muster (Anpassung an [PZ07, Abbildung 2-2, S. 23])

Die Funktionsweise von Model, View und Controller/Presenter entspricht der des MVC-Musters.

3 Entwurf einer Metrik zur Betrachtung der Modularität

Im Abschnitt 2.3 auf Seite 11 wurden die bisherigen Ansätze zur Modularitätsbetrachtung vorgestellt. Wie zu erkennen ist, lässt sich Modularität nicht an einem einzelnen Wert festmachen. Metriken die sich aus gemessenen oder geschätzten Werten ableiten – weil man diese nicht direkt messen kann – bezeichnet man als Pseudometrik [LL07, S. 293].

Es wurde der Ansatz einer statischen Codeanalyse gewählt. Grund dafür ist das geschilderte Problem mit der dynamischen Codeanalyse bei der Auswertung des Plat_Forms-Wettbewerbs 2007 (vgl. Abschnitt 1.1 auf Seite 2 f.).

Ludewig und Lichter beschreiben eine Vorgehensweise für die Entwicklung von Pseudometriken. Die Schritte zur Ableitung einer Pseudometrik sind [LL07, S. 305 f.]:

1. Identifikation des darzustellenden Aspekts
2. Modellierung des darzustellenden Aspekts
3. Festlegung einer Skala für die Metrik
4. Entwicklung einer Berechnungsvorschrift
5. Entwicklung der Messvorschriften für alle Größen, die in der Berechnungsvorschrift verwendet werden
6. Anwendung und Verbesserung der Metrik

Die Schritte werden im Nachfolgenden näher erläutert und angewendet.

3.1 Identifikation des darzustellenden Aspekts

Für die Entwicklung einer Pseudometrik bedarf es im ersten Schritt den Aspekt zu definieren, den die Metrik widerspiegeln soll.

Der Aspekt ergibt sich aus der Aufgabenstellung dieser Arbeit. Es soll die Modularität untersucht werden. Als Datenbestand liegen die Quelltexte der Lösungen des Plat_Forms-Wettbewerbs zugrunde. Im Speziellen dient die Metrik zur Untersuchung der Kopplung zwischen Komponenten, die der Darstellung von Daten dienen (Views) und Modulen, die die Daten bereitstellen. Diese Kopplung entsteht dadurch, dass Variablen referenziert werden, deren Inhalt dargestellt werden soll. Darstellung und Bereitstellung der Daten sind nach dem MVC-Architekturmuster in unterschiedlichen Modulen angesiedelt. Die meisten Teams nutzten eine Strukturierung nach diesem Muster. Die Ausnahme davon bildet Team D Java.

3.2 Modellierung des darzustellenden Aspekts

Im zweiten Schritt zur Entwicklung der Pseudometrik erfolgt die Modellierung des Aspekts. Dadurch wird es möglich, den Aspekt ausschließlich auf Basis der relevanten Daten zu bewerten [LL07, S. 305].

Den Lösungen gemein ist die Darstellung von Daten, die durch Nutzerinteraktion verändert werden können. Diese Änderungen werden von der Anwendung gespeichert.

Die Frameworks der Lösungen bieten für diese Verarbeitungsschritte eine Organisation nach dem MVC-Muster (vgl. Abschnitt 2.4.2 auf Seite 18) an. In der View dargestellte Daten stammen aus dem Model. Das Entwurfsmuster gibt aber nur die Zuweisung bestimmter Aufgaben zu festgelegten Komponenten vor. Zusätzlich muss aber auch die Kopplung zwischen den Komponenten einbezogen werden, um den Grad der Modularität betrachten zu können.

Aus den im Abschnitt 2.3 auf Seite 11 vorgestellten Ansätzen erscheint das *Gesetz von Demeter* (siehe Seite 15) eine geeignete Grundlage dafür zu sein. Für seine Anwendung sprechen folgende Gründe:

- Die Regel lässt sich sowohl auf objektorientiert, als auch auf prozedural entwickelte Software anwenden.

- Es wird darauf abgezielt, unerwünschte Kopplungen zu identifizieren.

Wie aus Abschnitt 2.4 auf Seite 16 und Abbildung 2–3 auf Seite 17 zu erkennen, erhält der Browser stets eine HTTP-Antwort zurück.

Wurde eine Ressource angefragt, die dargestellt werden kann (z.B. ein HTML-Dokument), muss der Web-Server diese bereitstellen. Enthält die angefragte Ressource dynamisch generierten Inhalt, so sind spezielle Mechanismen für die Erzeugung dieser notwendig. Die Teams im Plat_Forms-Wettbewerb (ausgenommen Team D), verwendeten das Entwurfsmuster der *Template View* für die Erzeugung von Views. Bei Team D kam eine andere Technik zum Einsatz, die im Abschnitt 3.5 auf Seite 27 näher erläutert wird. Das Muster wird mittels der Verwendung von Templates realisiert. Ein Template besteht im Wesentlichen aus HTML-Elementen in Kombination mit Platzhaltern. Das gilt verständlicherweise nur, falls das Ausgabeformat HTML ist. Andere Formate bestehen aus einer Kombination der Platzhalter und formatspezifischer Elemente. Die Platzhalter verweisen auf Variablen. Wird eine View vom Benutzer angefragt, sorgt das Templatesystem dafür, dass die Platzhalter durch den aktuellen Inhalt der Variablen ersetzt wird. Resultat ist ein reines HTML-Dokument. Für die Ersetzung der Platzhalter muss eine Kommunikation der View mit dem Controller erfolgen. Dieser kann wiederum bei Bedarf das Model heranziehen, um diese Aufgabe zu erfüllen.

Zum besseren Verständnis, wie das *Gesetz von Demeter* darauf angewendet werden kann, folgen zwei Beispiele aus den Lösungen des Wettbewerbs.

Beispielhaft sind in Abbildung 3–1 auf der nächsten Seite Platzhalter aus Templates aufgeführt. Abbildung 3–1(a) zeigt den Zugriff auf den Namen (name) des Erstellers (creator) einer Konferenz (conference).

conference.creator.name

(a) Team F PHP

category.parent.name

(b) Team N Ruby

Abbildung 3–1: Zulässiger Variablenzugriff nach dem Gesetz von Demeter

Aus dem Blickwinkel der objektorientierten Programmierung erscheint der Zugriff mittels *conference.creator.name* am einfachsten. Eine Konferenz besitzt eine Variable mit dem Ersteller als Inhalt; dieser wiederum eine Variable mit dem Namen desselben. Bei dem Beispiel handelt es sich um eine View zur Darstellung von Konferenzdetails. Der Controller liefert das für die View benötigte Model der Konferenz zurück. Dieses Model besitzt eine Variable die den Ersteller enthält, da es sich um eine Variable der Konferenz handelt, ist der Zugriff der Regel nach akzeptabel (linker, rot umrahmter Bereich). Der Name (name) hingegen gehört zum Ersteller (rechter, gelb umrahmter Bereich). Ein solcher Zugriff bedeutet eine Verletzung des Gesetzes von Demeter. Es wird ein weiteres Modul angesprochen, das nicht zur Konferenz gehört sondern zum Ersteller. Die Problematik ist aber nicht dadurch bedingt, dass das Konferenz- auf das Ersteller-Model verweist. Vielmehr ist es kritisch, dass diese Verbindung im Template genutzt wird. Es entsteht eine Abhängigkeit der View von der internen Realisierung im Model. Nach der auf Seite 10 eingeführten Kategorisierung fällt diese Art in Kategorie der *pathologischen Kopplung*. Eine Änderung an der Verbindung im Model bedingt dann auch eine Anpassung des Templates.

Nun bedeutet nicht jede Referenzierung nach diesem Schema eine Verletzung des Gesetzes. Das verdeutlicht auch das Bespiel, welches Abbildung 3–1(b) zeigt. Hier wird auf den Name der Oberkategorie (parent.name) einer angezeigten Kategorie (category) zugegriffen. Die Oberkategorie ist wiederum gleichen Typs, wie die Kategorie selbst. Der zulässige Bereich (rote Umrahmung) reicht über die komplette Referenzierung.

Eine Verletzung des Gesetzes ist nicht immer offensichtlich. Eine Fähigkeit von vielen Templateframeworks ist die Erzeugung temporärer Namen für Variablen.

```
    [...]
93  <?php foreach ($notifications as $notification): ?>
    [...]
103 <?php echo link_to($notification->getConference()->getName()[...] ?>
    [...]
113 <?php endforeach; ?>
```

Quelltext 3.1: Verwendung temporärer Variable in PHP-Template

Der Quelltext 3.1 zeigt einen Auszug aus einem Template von Team M PHP. Das Template dient der Darstellung von Profildetails eines Nutzers. Mit *foreach* wird über die Liste von Benachrichtigungen (*$notifications*) iteriert (Zeile 93). Gleichzeitig wird eine für den Schleifendurchlauf gültige und temporäre Variable definiert ($notification). In Zeile 103 wird die zur Benachrichtigung gehörende Konferenz mit *getConference()* und deren Name mittels *getName()* referenziert. Verfolgt man den Ursprung der temporären Variable zurück, stellt man fest, dass hier der Name einer Konferenz der zu einem Benutzer gehörigen Benachrichtigung referenziert wurde. Also wurde das Gesetz gleich doppelt verletzt. Solche Verstöße sind somit deutlich schwieriger zu erkennen (vgl. [Boc00]).

Dementsprechend müssen die Zugehörigkeiten der Variablen genau untersucht werden, um festzustellen in welchen Fällen ein Verstoß vorliegt.

3.3 Festlegung einer Skala für die Metrik

Im dritten Schritt muss eine geeignete Skala gefunden werden. Der Skalentyp muss so gewählt sein, dass die Daten auch auf der Skala abgebildet werden können. Ebenso muss für einen Vergleich von Werten der Metrik die Skala hinreichend genau sein [LL07, S. 305].

Durch die zeitliche Beschränkung des Plat_Forms-Wettbewerbs auf zwei Tage (vgl. [SP11, S. 1]) kann die Auswertung nur auf unterschiedlich vollständigen Lösungen erfolgen. Es wird betrachtet, zu welchem Grad gegen das Gesetz von Demeter verstoßen wird. Der Grad der Verletzung wird als reelle Zahl auf einer

Rationalskala angegeben. Die Werte liegen zwischen 0 und 1 beziehungsweise 0% und 100%.

3.4 Entwicklung einer Berechnungsvorschrift

Der vierte Schritt dient der Herleitung der Vorschrift, mit der die gemessenen Größen auf die Metrik abgebildet werden [LL07, S. 305 ff.]. Der Grad der Verletzung definiert sich als ein Verhältnis von Variablenzugriffen. Es ist das Verhältnis von Zugriffen aus Templates mit Verstoß gegen das Gesetz von Demeter gegenüber der Gesamtzahl von Variablenzugriffen.

Somit bedeutet 0, dass keiner der Variablenzugriffe gegen die Gesetzmäßigkeit verstößt. Eine 1 bedeutet, dass alle Variablenzugriffe dagegen verstoßen.

3.5 Entwicklung der Messvorschriften für alle Größen, die in der Berechnungsvorschrift verwendet werden

Zur Anwendung der Berechnungsvorschrift müssen im fünften Schritt alle Größen bestimmt werden. Es werden zunächst alle Variablenzugriffe aus Templates ermittelt. Im Folgenden wird das Vorgehen dafür beschrieben.

Zuerst werden die verwendeten Templates identifiziert. Dafür muss wiederum festgestellt werden, welches Template-Framework verwendet wird. Diese Information findet sich zumeist in den Konfigurationsdateien der Anwendung oder im Pfad eingebundener Bibliotheken (*build path*). Gegebenenfalls ist die Dokumentation des jeweiligen Webframework heranzuziehen. Mit Kenntnis welches Template-Framework verwendet wird, können die Templates jeder Anwendung lokalisiert werden.

Als nächstes wird der Einstiegpunkt der Anwendung ermittelt, also das Template, das standardmäßig als Startseite angezeigt wird. Die Information darüber findet sich meist auch in bestimmten Konfigurationsdateien des Webservers oder der Webanwendung. Ausgehend von diesem Startpunkt werden die erreichbaren Verbindungen zu anderen Templates bestimmt. Die zwei möglichen Arten von Verbindungen sind:

1. Einbindung eines Templates in ein anderes, so genannte *partials* (also Teiltemplates)
2. Verlinkung auf ein anderes Template, durch:
    a.) Hyperlink innerhalb eines Templates
    b.) Weiterleitung durch den Controller

Mit diesem Schritt wird sichergestellt, dass auch nur die tatsächlich durch den Benutzer erreichbaren Templates (bzw. Views) untersucht werden. Als Resultat liegen alle zu untersuchenden Templates vor.

Nachfolgend erfolgt die Beschreibung, nach welchen Regeln die Platzhalter in den Templates betrachtet werden.

3.5.1 Regeln für die Abstrahierung der Platzhalter in Templates

Die nachfolgenden Regeln bieten eine Grundlage dafür die Platzhalter der verschiedenen Template-Frameworks so weit zu abstrahieren, dass sie vergleichbar werden.

Regel 1 (*Syntax*)

Für jedes Template-Framework muss die Syntax der Variablenreferenzierung aufgeschlüsselt werden.

Eine detaillierte Beschreibung der Syntax, die bei der Auswertung beachtet wurde, findet sich in Tabelle 3.1 auf der nächsten Seite bis Tabelle 3.7 auf Seite 34.

Java-Teams

JavaServer Pages (JSP)	
${Variable}	Ausdruck aus der *Expression Language*, die Variable wird ausgewertet und ausgegeben oder kann weiterverarbeitet werden (bspw. Rechenoperationen)
<%=Ausdruck%>	JSP-Ausdruck wird ausgewertet und das Ergebnis als Text ausgegeben
<jsp:Direktivenname/> oder <%@Direktivenname %>	Ermöglicht die Verwendung spezieller JSP-Direktiven (z.B. Zugriff auf eine Java Bean mittels *useBean* oder Einbinden anderer Dateien über *include*)
<% Javacode %>	Hierbei handelt es sich um ein Scriptlet, es ermöglicht das Ausführen von Javacode in JSPs
<%! Javacode %>	Innerhalb dieser Anweisung lassen sich Deklarationen von z.B. Variablen, Methoden, etc. vornehmen

Tabelle 3.1: Syntax JavaServer Pages (JSP) [Ull11, Kap. 23]

In der Tabelle 3.1 sind die für die Auswertung notwendigen Sprachelemente der JavaServer Pages aufgeführt. Verwendung fand diese Technologie bei den Java-Teams A,E und I. Schleifen zum Iterieren über Listen, sowie bedingte Anweisungen lassen sich mit der JSP Standard Tag Library [Ora11] realisieren. Darüber hinaus bietet sich die Möglichkeit, weitere Tag-Bibliotheken einzubinden oder selbst zu definieren. Mit Tag-Bibliotheken lassen sich wiederkehrende Funktionalitäten kapseln und in einem Template durch einbinden der definierten Tags verwenden.

Das JVx-Framework, eingesetzt durch Team D Java, bietet eine sehr generische Möglichkeit zur Erzeugung der Benutzeroberfläche. Die Oberfläche wird technologieunabhängig erstellt und kann ohne Veränderungen verschiedenartig dargestellt werden. Zum Beispiel sind Swing-, QT-, JSF-, SWT-, GWT-Darstellungen unterstützt [SIBb]. Das hat auch einige Auswirkungen darauf, wie die Daten für die Benutzeroberfläche bereitgestellt werden.

Eine Oberfläche zum Anzeigen und Bearbeiten von Daten aus einer Datenbank wird mittels eines *WorkScreen* realisiert. Dort werden Darstellung und Editoren festgelegt. Das kann zum Beispiel eine editierbare Tabelle sein. Die Anbindung der Datenquelle (z.B. Datenbank) erfolgt mittels eines *RemoteDataBook*s. Das Layout der Darstellungskomponenten übernimmt JVx [SIBa].

Daraus ergibt sich aber die Problematik, dass sich dieses Framework nicht nach dem vorgestellten Schema betrachten lässt.

JavaScript-Team

Mustache wurde vom JavaScript Team B als Template-Framework eingesetzt. Tabelle 3.2 auf der nächsten Seite zeigt eine Übersicht der Syntax. Prinzipiell ist Mustache nicht an die Verwendung mit einer bestimmten Programmiersprache gebunden. Ebenso kann es auch neben HTML mit beliebigen anderen Textformaten verwendet werden. Die Entwickler selbst bezeichnen Mustache als *frei von Logik* [Wan].

Tatsächlich sind Konstrukte, wie Schleifen oder bedingte Anweisungen, für den Entwickler nicht sichtbar. Vielmehr prüft Mustache eigenständig, ob eine Variable existiert. Sollte eine Variable nicht vorhanden sein wird an dieser Stelle im Template nichts ausgegeben. Gleichermaßen werden Listen automatisch erkannt und deren Größe ermittelt. Mustache iteriert selbsttätig über alle Elemente.

Mustache	
{{Variable}}	Variableninhalt wird aufgelöst
{{{Variable}}}	Variableninhalt wird ohne HTML-Escaping aufgelöst
{{&Variable }}	Gezieltes Demaskieren von HTML-Entitäten einer Variable
{{#Variable}} {{Variable}	Eingeschlossener Inhalt wird nur angezeigt, falls die Variable existiert, bzw. ihr Wert *true* ist Kann auch zum Iterieren über Listen verwendet werden
{{ˆVariable}} {{Variable}}	Stellt das umgekehrte Gegenstück des vorherigen dar Eingeschlossener Inhalt wird nur angezeigt, falls die Variable nicht existent ist, bzw. ihr Wert *false* ist
{{> partial}}	Ermöglicht das Einbinden Partials (ausgelagerte Teile von Template)
{{! Kommentar }}	Ein Kommentar

Tabelle 3.2: Syntax Mustache[Wan]

PHP-Teams

Fluid ist Bestandteil des CMS-Systems FLOW3, das vom PHP-Team F eingesetzt wurde. Fluid ist für die Verarbeitung von Templates zuständig. Tabelle 3.3 zeigt, dass Zugriffe auf Variablen mit geschweiften Klammern umschlossen werden. Damit wird hier ein anderer Ansatz verfolgt als bei den anderen PHP-Teams. Bei den übrigen Teams werden die sprachlichen Mittel von PHP selbst für die Template-Definition genutzt.

Fluid	
{Objekt}	Variablenzugriff
<f:helperName></f:helperName>	View Helper (Namensraum f=F3\Fluid\ViewHelpers), für jeden Helper existiert eigene Klasse
property=„name“	Ordnet einem Eingabefeld das Attribut eines Objekts zu (für Formulare)

Tabelle 3.3: Syntax Fluid [FLO11]

Die PHP-Teams G, L und M nutzten die Kombination aus HTML und PHP für die Definition der Templates.

Einen Unterschied gibt es beim Einbinden von Formularen. Teilnehmerteam G und L haben dafür das Zend-Framework eingesetzt. Dieses bietet die Verwendung von *Zend_Form* für Formulare. Tabelle 3.4 auf der nächsten Seite zeigt, dass mit der Methode *addElement* einzelne Formularelemente eingefügt werden können.

Team M hat Symfony für die Erzeugung von Formularen eingesetzt. Wie man der Tabelle 3.5 auf der nächsten Seite entnehmen kann, wird einen Formular mit der Funktion *setWidgets* eine Liste der Eingabefelder übergeben. Das Rendern als HTML-Formular übernimmt Symfony [Sen11, S. 159 ff.].

PHP (Zend Form)	
$Variable	Zugriff auf eine Variable
Formular: Instanz von Zend_Form $form->addElement() fügt neues Element zum Formular hinzu	

Tabelle 3.4: Syntax PHP (Zend Form) [Zen05]

PHP (symfony)	
$Variable	Zugriff auf eine Variable
$this->setWidgets(array(. . .))	Legt die Felder des Formulars fest

Tabelle 3.5: Syntax PHP (symfony) [Sen11]

Perl-Teams

Das Webentwicklungs-Framework Catalyst lässt sich durch eine Vielzahl von Plugins erweitern [She11]. Die Perl-Teams C und J nutzten das *Template Toolkit,* welches eine Reihe von Perl-Modulen zusammenfasst [War08]. Es dient ebenfalls der Erstellung und Darstellung von Templates. Wie Tabelle 3.6 zeigt, bietet das Plugin eine Reihe von Direktiven, die innerhalb von Templates genutzt werden können.

Catalyst (Template Toolkit)	
[%Variable%]	Gibt den Inhalt der Variable aus
[%Direktive%]	Template Direktive: eine Liste von Funktionen, die in Direktiven verwendet werden können findet sich unter `http://search.cpan.org/perldoc?Template`

Tabelle 3.6: Syntax Catalyst (Template Toolkit) [War]

Das von Team O verwendete Catalyst Plugin HTML::Zoom verfolgt einen etwas anderen Ansatz, als die anderen vorgestellten Template-Frameworks. Stellen im

Catalyst (HTML::Zoom)
Die Zeile <div class='username'>USERNAME</div> in einem Template ermöglicht aus der Anwendung heraus (vornehmlich aus dem Controller) mittels $zoom->select('.username')->replace_content($username); USERNAME durch den Inhalt der Variable $username zu ersetzen

Tabelle 3.7: Syntax Catalyst (HTML::Zoom) [Tro11b]

Template, an denen später dynamisch Inhalte eingefügt werden sollen, werden mit CSS-Selektoren markiert. Das Einfügen von Inhalten erfolgt aus der Anwendung heraus. Tabelle 3.7 zeigt ein Beispiel. Das Bereichs-Element (<div>) erhält eine Attributzuweisung, dass die CSS-Klasse dieses Elements *username* ist. Mittels der Select-Methode kann aus der Anwendung auf den Textinhalt des Blocks zugegriffen und dieser verändert werden.

Ruby-Teams

Die Ruby-Teams H und K nutzten das Template-Framework HAML. Größtenteils sind auch die Templates von Team N in HAML-Syntax geschrieben worden. Zusätzlich finden sich ein paar Templates in ERB-Syntax. Dieses Framework nutzte Team P für die Umsetzung aller seiner Templates.

Die erste Zeile von Tabelle 3.8 auf der nächsten Seite zeigt eine Besonderheit von HAML. Es lassen sich HTML-Elemente erzeugen. Diese Funktionalität soll den Templatecode übersichtlicher machen, weil HTML-Elemente Platzhalter nicht umschließen (so genannte Inline-Codierung). Dieses Template-Framework setzten die Ruby-Teams H, K und N ein.

ERB bietet die Möglichkeit, Ruby-Code mit beliebigen Textdokumenten zu kombinieren [THS+]. Im Wesentlichen zeigt Tabelle 3.9 auf der nächsten Seite, dass auf definierte Variablen in normaler Ruby-Syntax zugegriffen werden kann. Außerdem kann beliebiger Ruby-Code in Templates eingebettet werden.

<table>
<tr><th colspan="2">XHTML Abstraction Markup Language (HAML)</th></tr>
<tr><td>%name</td><td>Erzeugt ein HTML-Element der Form <name></name></td></tr>
<tr><td></td><td>Der Zugriff auf Instanzvariablen erfolgt mittels @Variablenname</td></tr>
<tr><td>= Ausdruck</td><td>Wertet den Ausdruck aus und gibt das Ergebnis als Text aus</td></tr>
<tr><td colspan="2">Formularzugehörigkeit wird mit: form_for @conference do |form| definiert
Das Beispiel zeigt ein Formular für ein Objekt vom Typ conference</td></tr>
<tr><td>/Kommentar oder =#Kommentar</td><td>Ein Kommentar</td></tr>
</table>

Tabelle 3.8: Syntax HAML [CWE11]

<table>
<tr><th colspan="2">Embedded Ruby (ERB)</th></tr>
<tr><td><%Rubycode%></td><td>Eingebetteter Ruby Code</td></tr>
<tr><td><%=Rubycode%></td><td>Ruby Ausdruck, wird mit Ergebnis des Ausdruck gefüllt</td></tr>
<tr><td>@variablenname</td><td>Zugriff auf Instanzvariable</td></tr>
<tr><td colspan="2">Formulare: siehe Tabelle 3.8</td></tr>
</table>

Tabelle 3.9: Syntax Embedded Ruby (ERB) [THS+]

Wie aus den Beschreibungen in Tabelle 3.1 auf Seite 29 bis Tabelle 3.7 auf Seite 34 ersichtlich ist, bieten die meisten Template-Frameworks Schleifenkonstrukte für die Ausgabe von Listen. Der auszugebende Inhalt muss dann für eine Liste nur einmal angegeben werden. Problematisch für die Betrachtung der Variablen in Schleifen ist, dass mit steigender Größe der Liste auch die Anzahl der Referenzierungen auf Variablen steigt. Dieser Sachverhalt betrifft die Lösungen aber gleichermaßen und deshalb wird die folgende Regel für Schleifen angewendet:

Regel 2 (*Variablenreferenzierung innerhalb von Schleifen*)

Variablen die innerhalb einer Schleife ausgegeben werden, werden lediglich einmal *bei der Auswertung analysiert und gezählt.*

Ein weiterer Punkt der zu beachten ist, sind lokale definierte Variablen in Templates. Zumeist dienen diese zur Vereinfachung bestimmter Abläufe und Verbesserung der Lesbarkeit. Dabei muss man zwei verschiedene Arten voneinander trennen. Zum einen solche Variablen, die ausschließlich auf Werte oder Variablen verweisen, die innerhalb des Templates gültig sind. Zum anderen lokale Variablen, die auch auf Variablen verweisen, deren Gültigkeit über Template hinausreicht. Das können zum Beispiel darzustellende Daten des Models sein (vgl. Abschnitt 3.2 auf Seite 23).

Regel 3 (*Lokal definierte Variablen in Templates*)

Ist eine Variable lokal in einem Template definiert und bezieht sich auch sonst auf keine Variable, die außerhalb des Templates gültig ist, so bleibt sie von der Auswertung unbetrachtet. Andernfalls muss die Referenz auf andere Variablen aufgelöst und untersucht werden (siehe dazu auch Abschnitt 3.2 auf Seite 23).

Bei einigen Templates sind Texte z.B. für Eingabefelder, Seitentitel oder Meldungen fest kodiert. Andere Teams haben wiederum diese Texte beispielsweise aus Ressourcendateien eingebunden.

Sind die Beschriftungen nicht festkodiert, übernimmt das Template-Framework die Ersetzung mit dem Inhalt. Ein Zugriff auf Controller oder Model findet dabei aber nicht statt, somit spielt dies auch keine entscheidende Rolle.

Eine modulübergreifende Referenzierung findet ebenfalls nicht statt. Deshalb gilt Regel 4:

Regel 4 (*Beschriftungen im Templates*)

Platzhalter in Templates die keine Variablen referenzieren, sondern durch Texte (bspw. aus Ressourcendateien) ersetzt werden, dienen der Beschriftung. Beschriftungen werden von der Betrachtung der Referenzierung ausgeschlossen.

Wie zu Beginn dieses Kapitels beschrieben, sind die Templates untereinander durch Verlinkungen verbunden, bzw. lassen sie sich als Teiltemplate in andere Templates einbinden. Aufgrund dieser Tatsache kann es sein, dass ein Template mehrfach verwendet wird. Es kann nun der Fall eintreten, dass beispielsweise Template A auf Template B verweist und B auf A. Dadurch würde bei der Anwendung des Auswertungsverfahrens eine Endlosschleife entstehen. Bei dieser Art wird jedes Templates nur einmalig betrachtet. Teiltemplates dagegen werden lediglich als einzelner Bestandteil in ein anderes Template eingefügt und werden deshalb bei jeden Vorkommen ausgewertet. Es gilt Regel 5:

Regel 5 (*Mehrfache Verwendung von Templates*)

Auch bei mehrfacher Verlinkung wird ein Template nur einmalig betrachtet. Für Teiltemplates (partials) erfolgt die Auswertung bei jeder Verwendung.

Die Auswertung der referenzierten Variablen erfolgt nun, wie im Abschnitt 3.2 auf Seite 23 beschrieben. Für jede Variable wird überprüft, ob die Kommunikation ausschließlich mit dem zuständigen Modul erfolgt oder ob die Abarbeitung über die Modulgrenze hinaus reicht. Kann die Referenzierung vom Modul selbst beantwortet werden, zählt die Referenzierungstiefe als 1. Für jedes weitere benötigte Modul erhöht sich die Tiefe.

Regel 6 (*Bestimmung der Referenzierungstiefe*)

Referenziert ein Platzhalter auf eine Variable eines Moduls ist die Referenzierungstiefe 1. Für jedes weitere Modul, das zur Auflösung der Referenzierung benötigt wird, erhöht sich der Wert um 1.

Eine Referenzierungstiefe größer als 1 bedeutet eine Verletzung des Gesetzes von Demeter.

3.5.2 Sonderfälle bei der Auswertung

Im Abschnitt 3.5 auf Seite 27 wurde bereits darauf hingewiesen, dass Team D Java eine andere Art und Weise der Erzeugung der Benutzeroberfläche verwendet hat, als die anderen Teams. Auf dieses Team wird in der Auswertung im Kapitel 5 auf Seite 53 gesondert eingegangen.

Eine weitere Besonderheit findet sich bei Team O Perl. Die Funktionsweise des Template-Framework wurde in Tabelle 3.7 auf Seite 34 beschrieben. Eine Untersuchung der Lösung, sowie ein Blog-Eintrag des Teams selbst [Tro11a] zeigen, dass die Aufgabe der Erstellung der Benutzeroberfläche teilweise offen geblieben ist. Es existieren zwar Templates jedoch findet sich kein Einstiegspunkt der Anwendung. Dieser wird jedoch benötigt, um die verwendeten Templates zu ermitteln (siehe Abschnitt 3.5 auf Seite 27). Somit kann an dieser Lösung keine Messung durchgeführt werden.

3.6 Anwendung und Verbesserung der Metrik

In den folgenden beiden Abschnitten wird die Anwendung der im Abschnitt 3.5 auf Seite 27 vorgestellten Vorgehensweise beschrieben und überprüft, ob die Anforderungen an Metriken erfüllt sind. Die Abschnitte umfassen den sechsten und letzten Schritt für die Herleitung der Pseudometrik.

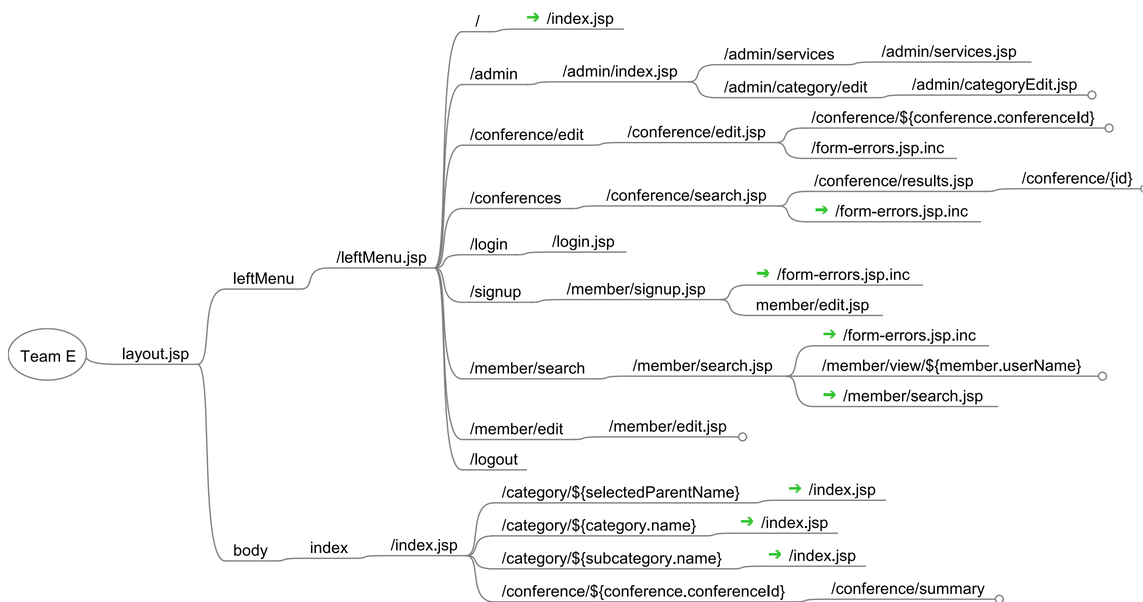


Abbildung 3–2: Auszug aus Mind-Map von Team E

3.6.1 Vorgehen bei der Anwendung der Metrik

Zunächst werden die Zusammenhänge der Templates der Teams ausgewertet und in Mind-Maps festgehalten (je eine pro Team). Dafür wird das Open-Source-Programm FreeMind[8] verwendet.

Bei einer Mind-Map handelt es sich um eine Baumstruktur. FreeMind kann diese Struktur visualisieren und ermöglicht ein einfaches Editieren. Wurzel des Baumes ist der jeweilige Teamname. Als erstes Kind-Element wird das Template des Einstiegspunkts der Anwendung eingefügt. Ausgehend von diesem Einstiegspunkt werden alle erreichbaren Verlinkungen bzw. Templates als Kind-Elemente eingefügt. Dieses Vorgehen wird dann so lange für alle Kind-Elemente wiederholt, bis alle erreichbaren Templates notiert sind. Abbildung 3–2 zeigt eine solche Struktur an einem beispielhaften Auszug von Team E Java.

Nachdem alle erreichbaren Templates ermittelt sind, erfolgt die Auswertung der Platzhalter in den Templates. Das wird nach den Vorgaben von Abschnitt 3.5.1 auf Seite 28 durchgeführt. Die Platzhalter werden jeweils als Kind-Elemente des zugehörigen Templates vermerkt. Zum Platzhalter wurde jeweils auch die Zeilennummer notiert.

[8] `http://freemind.sourceforge.net/`

Verweist eine temporäre Variable eines Templates auf einen Platzhalter, kann im Folgenden der Name der temporären Variable statt des Platzhalter verwendet werden. Damit der Ursprung eines Platzhalters auch bei Umbenennung innerhalb des Templates noch nachvollziehbar ist, wird eine solche Umbenennung ebenfalls am Platzhalter notiert (vgl. Quelltext 3.1 auf 26).

Als abschließender Schritt wird für jeden Platzhalter geprüft, welche Variable er referenziert und zu welchem Modul diese gehört. Mit diesem Wissen kann festgestellt werden, wie tief die Referenzierung reicht. Das wird ebenfalls in der Mind-Map festgehalten. Es kann außerdem bestimmt werden, ob eine Verletzung des Gesetz von Demeter vorliegt.

Das Speicherformat einer Mind-Map ist das XML-Format. Mit den vorliegen Mind-Maps wurde eine Weiterverarbeitung durchgeführt. Es wurde ein Java-Programm entwickelt, dass die XML-Struktur durchläuft und die Referenzierungen auswertet.

3.6.2 Analyse der eingesetzten Metrik

Ludewig und Lichter [LL07] empfehlen eine Reihe von Eigenschaften, die eine Metrik besitzen sollte. Metriken sollen:

1. differenziert
2. vergleichbar
3. reproduzierbar
4. verfügbar
5. relevant
6. rentabel
7. plausibel

sein [LL07, S. 281].

Im Folgenden wird untersucht, inwiefern die Eigenschaften auf die entwickelte Metrik zutreffen.

Eine *differenzierte* Metrik muss bei unterschiedlichen Merkmalsausprägungen auch unterschiedliche Werte aufweisen. Dieser Aspekt trifft zu, weil eine tiefere Referenzierungstiefe und damit verbundene stärkere Kopplung direkt als höherer Zahlenwert abgebildet werden.

Eine *Vergleichbarkeit* ist gewährleistet, wenn Werte auf genau einen Punkt einer Skala abgebildet werden. Der Skalentyp muss dafür einer Rational-, Ordinal-, oder Absolutskala entsprechen. Auch diese beiden Eigenschaften treffen zu. Das Verhältnis der Verletzungen der zulässigen Referenzierungstiefe kann genau einen Wert auf einer Rationalskala annehmen.

Reproduzierbarkeit ist dann gegeben, wenn eine wiederholte Anwendung der Metrik die gleichen Werte liefert. Durch die im Abschnitt 3.5.1 auf Seite 28 vorgegebenen Regeln ist die Metrik eindeutig. Für eine Erweiterung auf andere Plattformen müssen die Regeln auf die Plattform angewendet, bzw. angepasst werden.

Bewertungen müssen zum Zeitpunkt vorliegen, an dem sie gebraucht werden. Nur dann kann man von *Verfügbarkeit* einer Metrik sprechen. Zu diesem Punkt lässt sich sagen, dass die Metrik auf die Lösungen angewendet wurden und die Bewertungen zur nachträglichen Auswertung genutzt werden. Sie sind also zu dem Zeitpunkt verfügbar, an dem sie benötigt werden.

Die *Relevanz* ist dadurch gegeben, dass die Metrik ausgehend von einem konkreten Nutzen entwickelt wurde. Die Modularität wird anhand der Kopplung bestimmt, die durch Referenzierung von Variablen in Templates entsteht. Dieser Ansatz ist neu, dennoch wird er durch etablierte Konzepte der Modularitätsbetrachtung gestützt. Ebenso wurde auf Vergleichbarkeit und Erweiterbarkeit für verschiedene Programmiersprachen geachtet.

Eine Metrik ist *rentabel*, wenn sie zum einen relevant ist, was bereits nachgewiesen wurde. Zum anderen darf sie nicht mehr kosten, als sie nützt. Der Kosten-

aspekt wird hier vernachlässigt. Für eine Kostenminimierung müsste die Datenerhebung jedoch stärker automatisiert werden.

Mit steigender Referenzierungstiefe wird eine stärkere Kopplung erzeugt. Diese intermodulare Kopplung erzeugt mehr Abhängigkeiten zwischen Modulen der View zu denen vom Model. Aus diesem Grund ist die Modularität bei größerer Referenzierungstiefe schlechter. Die entwickelte Metrik spiegelt diesen Sachverhalt direkt wider. Dementsprechend ist der Aspekt der *Plausibilität* erfüllt.

4 Architekturmuster als Gütekriterium der Modularität

Für die effiziente und flexible Gestaltung von interaktiven Webanwendungen wurde im Abschnitt 2.4.2 auf Seite 18 bereits das MVC-Architekturmuster vorgestellt.

Durch dieses Muster wird eine Struktur vorgegeben. Diese trennt die verschiedenen Zuständigkeiten von Model, View und Controller. Falls diese Trennung konsequent vollzogen wird, sind die MVC-Komponenten nur über ihre Daten gekoppelt (vgl. Abschnitt 2.2 auf Seite 8).

Bei der Untersuchung für die Bestimmung der Referenzierungstiefe gab es eine Auffälligkeit: Einige Funktionalitäten sind nicht dort umgesetzt, wo man sie nach dem MVC-Muster erwarten würde. Es lässt sich vermuten, dass die Kohäsion dadurch negativ beeinflusst wird. Übernimmt ein Controller neben seinen Zuständigkeiten auch welche der View oder des Models, ändert sich die Kohäsion. Der Kategorisierung von Constantine Yourdon (siehe Seite 8) zufolge liegt keine *funktionale Kohäsion* mehr vor. Diese Kohäsionskategorie würde nur vorliegen, wenn ausschließlich Aufgaben des Controllers in diesem durchgeführt werden.

Wie im Abschnitt 2.2 auf Seite 8 dargelegt, trifft nur die höchste Kohäsionskategorie zu, die alle Elemente eines Moduls erfüllen. Aus diesem Grund wird beginnend von der niedrigsten Kategorie überprüft, welche der Kategorien zutrifft. Eine *zufällige Kohäsion* liegt in diesem Fall vor. Bei einer Vermischung von Zuständigkeiten von MVC-Komponenten sind die Verarbeitungsschritte auf jeden Fall aufeinander folgend. Die nächstbessere Kategorie ist die *logische Kohäsion*. Diese liegt vor, wenn die Verarbeitungsschritte des Moduls logisch zusammengehörig sind. Das

Model-View-Controller-Muster gibt aber genau eine solche logische Strukturierung vor. Daraus ergibt sich, dass in diesem Fall eine *logische Kohäsion* nicht vorliegen kann.

Jedoch ist die allgemeine Beschreibung des MVC-Musters nicht hundertprozentig fest. Vielmehr liegt ein gewisser Interpretationsspielraum der Webframeworks zu Grunde. Deshalb erfolgt zunächst eine Aufschlüsselung der MVC-Umsetzungen für die einzelnen Framework im Wettbewerb. Anschließend wird das Vorgehen für die Analyse vorgestellt. Im Abschnitt 5.2 auf Seite 58 erfolgt eine Auswertung, inwieweit von den Teams gegen das Muster verstoßen wird, um weitere Einflüsse auf Kopplung un Kohäsion zu ergründen.

4.1 Umsetzung der Java-Teams

Bei den Java-Teams finden sich drei verschiedene Frameworks. Zwei der Teams verwendeten Spring und je eines abaXX, sowie JVx.

4.1.1 Spring

Die Beantwortung einer Anfrage in Spring zeigt das Ablaufdiagramm in Abbildung 4–1 auf der nächsten Seite. Das Dispatcher-Servlet reagiert auf alle eingehenden Anfragen an die Applikation. Es ist nur einmal je Applikation vorhanden. Entsprechend dem aufgerufenen URI wird der zuständige Controller mit der Bearbeitung der Anfrage beauftragt. Dieser Controller führt die angeforderten Aktionen durch und fragt Daten beim zuständigen Model ab. Auf Basis der Anfrage stellt der Controller ein *ModelAndView*-Objekt zusammen. Das beinhaltet ein oder mehrere Model mit angefragten Daten und welche View dargestellt werden soll. Die View wird lediglich als logischer Name übergeben. Das *ModelAndView*-Objekt wird an das Dispatcher-Servlet zurückgegeben und dieses löst den logischen Namen der View (mit Hilfe des ViewResolver) in eine konkrete Instanz auf. Diese View formt dann die Daten in eine darstellbare Form um und liefert sie dem Benutzer aus [LD06]. Eingesetzt wurde Spring von Team E und I.

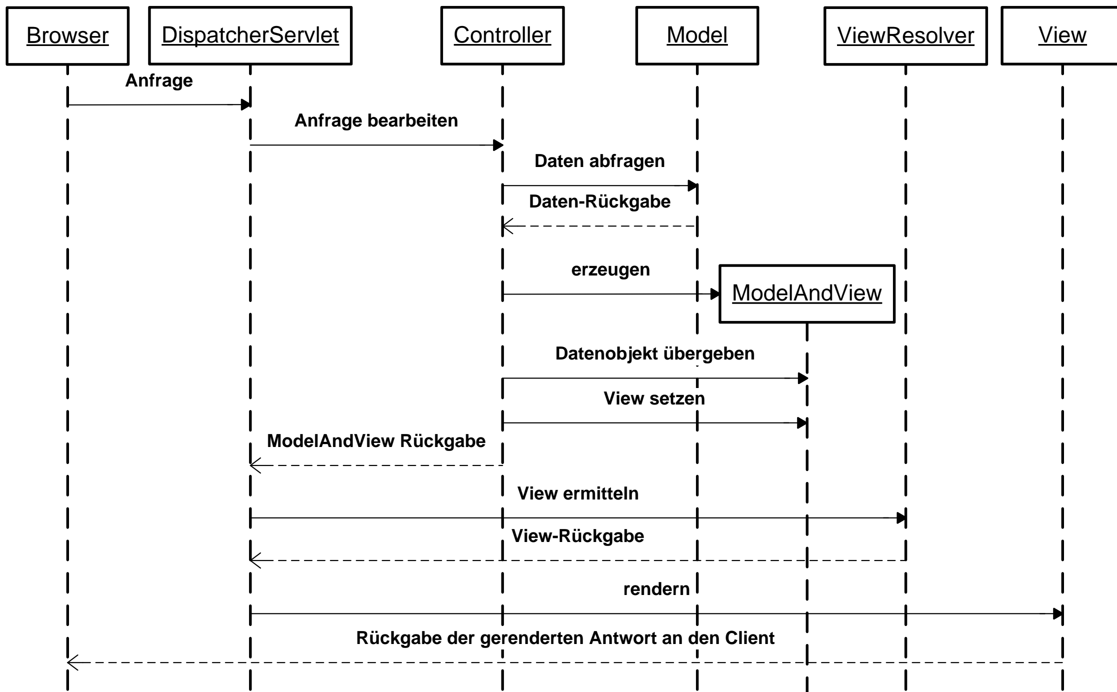


Abbildung 4–1: Anfragebearbeitung in Spring (Anpassung an [LD06, S. 64])

4.1.2 abaXX

Mit dem abaXX-Portal-Framework, das Team A verwendete, lassen sich leichtgewichtige Komponenten erzeugen. Einzelne Komponenten werden in einer XML-Konfigurationsdatei definiert (genannt parts.xml). Ein solche Komponente wird als *Part* bezeichnet. Neben dem Namen für einen *Part* beinhaltet die Konfigurationsdatei eine Zuordnung von View, Controller und Model für diesen *Part* [PP03].

Da es sich um ein kommerzielles Framework handelt, konnten keine weiteren Informationen über die genauen Zusammenhänge der MVC-Komponenten ermittelt werden.

4.1.3 JVx

Das von Team D eingesetzte JVx-Framework verfolgt einen vollständig anderen Ansatz Daten bereitzustellen und für den Benutzer editierbar zu machen.

Für Daten, die angezeigt werden sollen, wird eine Datenquelle angegeben und mit einem entsprechenden Benutzeroberflächenelement verknüpft. Das kann bei-

spielsweise eine Tabellendarstellung sein, die die Tabelle einer relationalen Datenbank abbildet und die enthaltenen Daten editierbar macht.

Durch Angabe der Datenquelle werden standardmäßig alle Spalten der Tabelle angezeigt. Das kann zielgerichtet durch Auswahl einzelner Spalten eingeschränkt werden.

Die Oberfläche wird auf Basis der ausgewählten Spalten und der gewählten Darstellungs-Technologie erzeugt [SIBa][SIBb].

4.2 Umsetzung des JavaScript-Teams

Team B JavaScript hat kein spezielles Framework eingesetzt, dass eine Organisation nach dem MVC-Muster umsetzt. Vielmehr handelt es sich um eine Zusammenstellung verschiedener Komponenten, die die jeweiligen Aufgaben von Model, View und Controller übernehmen.

Views werden unter der Verwendung des Template-Frameworks *Mustache* realisiert. *Express* ist ein Framework, das einerseits das Weiterleiten von Benutzeranfragen an den Controller bewerkstelligt, andererseits regelt es den Aufruf der zur Darstellung benötigten View.

4.3 Umsetzung der Perl-Teams

Die Perl-Teams verwendeten einheitlich das Catalyst-Framework. Das Catalyst-Framework stellt einen Front Controller bereit, der die eingehende Anfrage an eine Action im zugehörigen Controller weiterleitet. Der Controller bearbeitet die Anfrage, holt das benötigte Model und leitet dieses an eine View weiter (siehe Abbildung 4–2 auf der nächsten Seite).

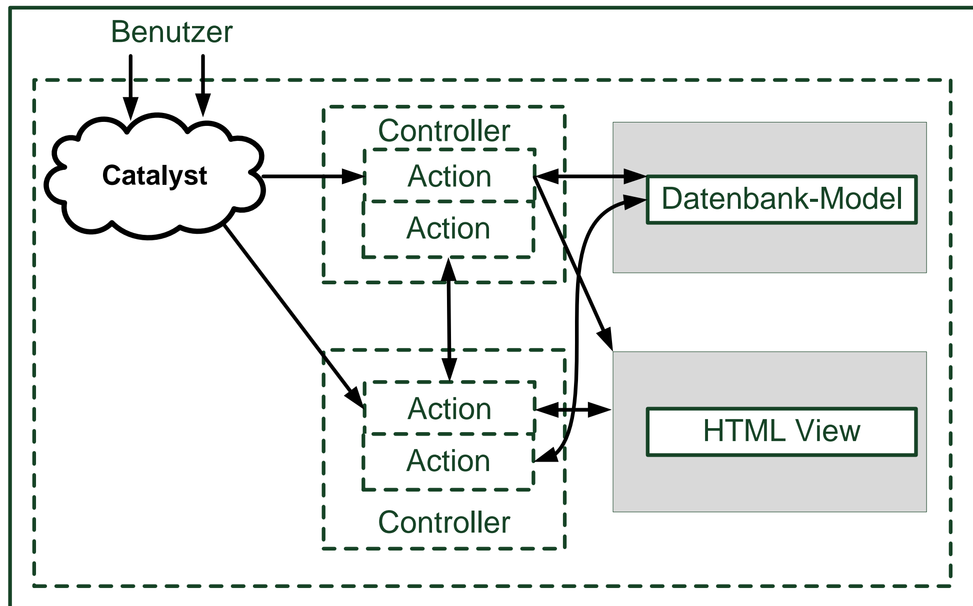


Abbildung 4–2: Anfragebearbeitung in Catalyst (Anpassung an [Joh10, S. 8])

4.4 Umsetzung der PHP-Teams

Unter den PHP-Teams finden sich drei Frameworks, die eingesetzt wurden. Diese Frameworks sind das Symfony- (Team M), das Zend- (Team G und L), sowie das Flow3-Framework (Team F).

4.4.1 Symfony

Abbildung 4–3 auf der nächsten Seite zeigt den Zusammenhang der MVC-Komponenten im Symfony-Framework. Im Symfony-Framework ist der Controller in zwei Teilbereiche gegliedert. Es existiert ein applikationsweiter Front Controller. Dieser sorgt für die Anfragebearbeitung und wählt die entsprechenden Actions aus. Diese Actions sind der zweite Teil des Controllers im Framework. In den Actions findet sich der seitenspezifische Controllercode. Dieser greift auf das Model zu, um die Anfrage zu bearbeiten. Geänderte beziehungsweise darzustellende Modeldaten werden von der Action an die View weitergegeben. Die View sorgt dann für die Darstellung der Daten. Das gerenderte Ergebnis der View erhält der Front Controller, der dieses wiederum an den Client ausliefert [PZ07, S.21 ff.].

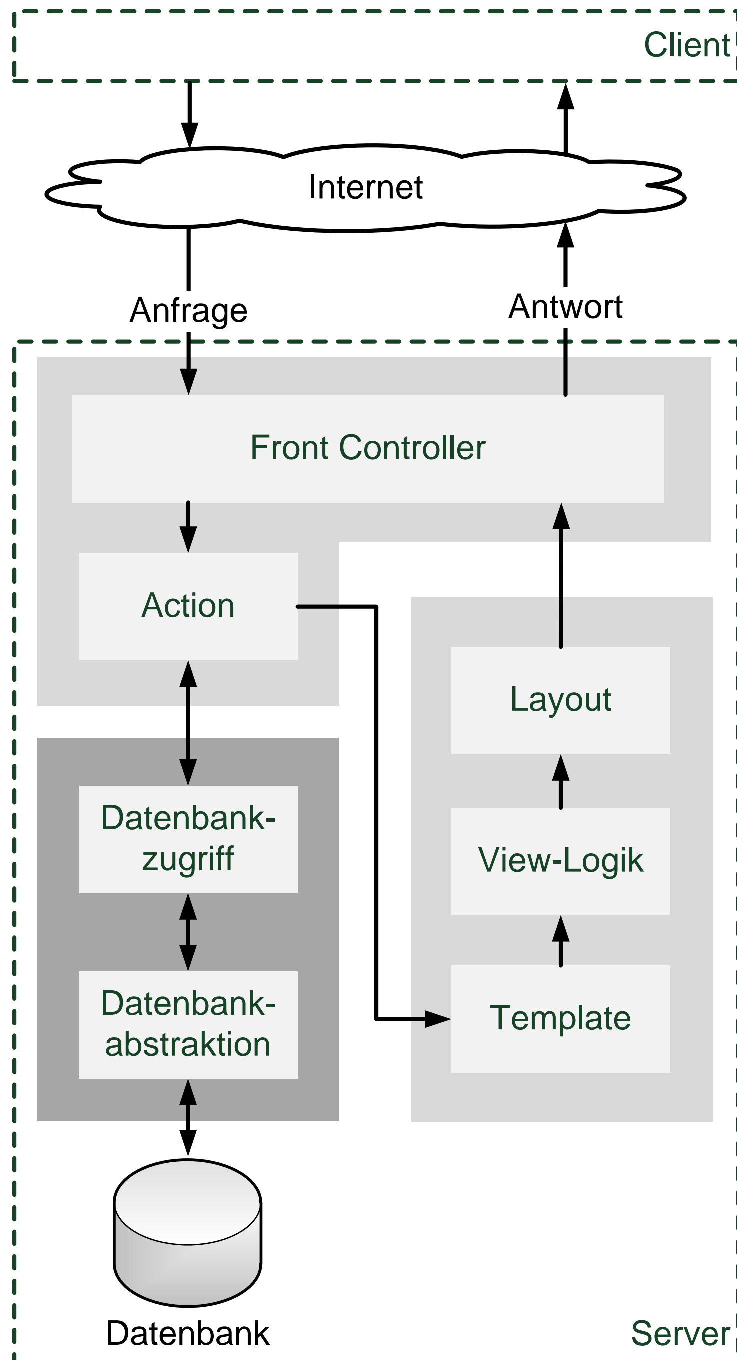


Abbildung 4–3: Ablauf in Symfony [PZ07, S. 23]

4.4.2 Zend

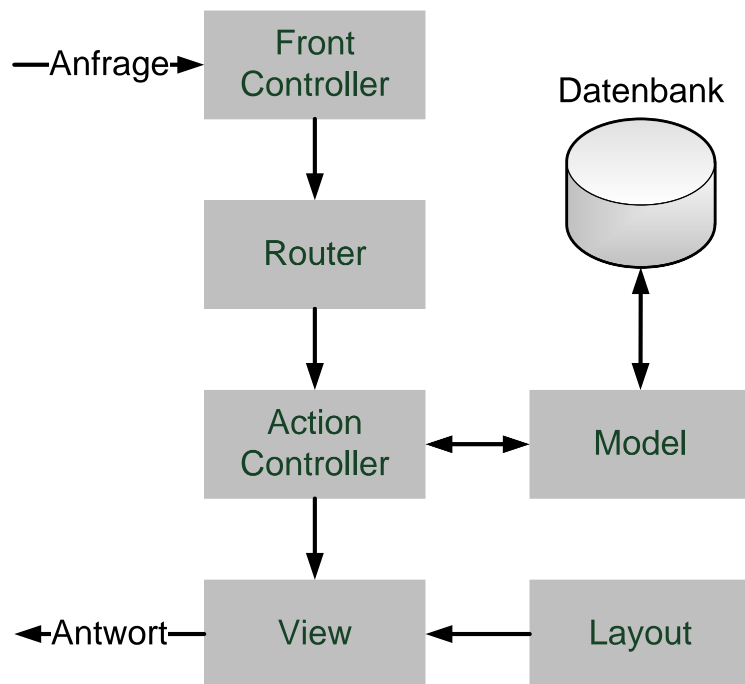


Abbildung 4–4: Anfragebearbeitung in Zend [Vas11, S. 26]

Abbildung 4–4 zeigt die Anfragebearbeitung im Zend-Framework. Der Front-Controller ermittelt aus der Anfrage-URL den Controller und dessen aufzurufende *Action*. Der entsprechende Controller erhält die Kontrolle der Anfragebearbeitung und bearbeitet die Anfrage unter Zuhilfenahme ein oder mehrerer Models. Zu den Aufgaben des Controllers gehören auch die Auswahl der View und das Bereitstellen der Daten für diese. Die View rendert die Ausgabe, die der Client als Antwort erhält [Vas11].

4.4.3 Flow3

Das Model in Flow3 ist ein normales PHP-Objekt [TYP11b]. Der Controller erhält Zugriff auf das Model über Dependency Injection [TYP11a]. Das bedeutet, bei der Erzeugung des Controllers wird das Model diesem explizit bekannt gemacht. Der Controller erhält automatisch Zugriff auf die anfragende View mittels *$this->view* [TYP11c].

4.5 Umsetzung der Ruby-Teams

Die vier Ruby-Teams nutzten das Framework Ruby on Rails für ihre Anwendungen. Auch bei Ruby on Rails dient die View ausschließlich der Präsentation der Daten des Models. Dieses wiederum ermöglicht eine Manipulation der Daten und enthält die Business-Logik der Applikation. Der Controller ist das Verbindungsstück zwischen diesen beiden Komponenten. Eingehende Anfragen werden vom Controller verarbeitet und die angeforderte *Action* ausgeführt. Gegebenenfalls wird das Model abgefragt und Daten von diesem an eine View weitergeleitet und vom Controller das Rendern von Templates angestoßen [MGL+11].

4.6 Verfahren für die Ermittlung von Verstößen gegen das MVC-Muster

Für die Ermittlung, ob ein Team gegen die MVC-Interpretation des Frameworks verstoßen hat, werden die Quelltexte von Model, View und Controller einer Durchsicht unterzogen. Eine solche manuelle Durchsicht ist erforderlich, weil die MVC-Interpretation der einzelnen Frameworks unterschiedlich ist (vgl. Abschnitt 4.1 auf Seite 44 bis Abschnitt 4.5). Eine automatisierte Betrachtung erscheint als nicht umsetzbar.

Es müssen für die Betrachtung die verwendeten MVC-Komponenten für jedes Team bekannt sein. Dafür können die Mind-Maps, die für die Metrik in Kapitel 3 auf Seite 22 erstellt wurden, herangezogen werden. Das ist möglich, weil zur Ermittlung der Referenzierungstiefe auch geprüft werden musste, welche Controller und Model für die Referenzierung verwendet wurden (vgl. Abschnitt 3.6.1 auf Seite 39).

Bei der Durchsicht wurden Auffälligkeiten identifiziert und untersucht. Eine Auflistung dieser findet sich in der folgende Tabelle 4.1. Eine entsprechende Auswertung erfolgt im Abschnitt 5.2 auf Seite 58.

Team A Java	
Auffälligkeit	Direktzugriff auf MemberManager (Model) aus View heraus
Fundort[9]	/abaxx/trunk/project/web/parts/platforms/start.jsp
Team C Perl	
Auffälligkeit	Formulare mit HTML::FormHandler erzeugt
Fundort[9]	/Cap-0.01/root/conference/create.tt /Cap-0.01/root/conference/modify.tt /Cap-0.01/root/conference/search.tt /Cap-0.01/root/category/create.tt /Cap-0.01/root/user/register.tt /Cap-0.01/root/user/modify.tt
Team D Java	
Auffälligkeit	Keine Trennung nach MVC
Team F PHP	
Auffälligkeit	HTML-Tags im Controller
Fundort[9]	/Classes/Controller/ConferenceController.php
Team H Ruby	
Auffälligkeit	Zugriff auf Methode von Model aus View heraus
Fundort[9]	/app/views/invitations/index.html.haml /app/views/invitations/new.html.haml
Team J Perl	
Auffälligkeit	Formulare mit HTML::FormHandler erzeugt
Fundort[9]	/CaP/root/src/conference/add.tt2 /CaP/root/src/users/login.tt2 /CaP/root/src/users/register.tt2 /CaP/root/src/users/status.tt2
Team L PHP	
Auffälligkeit	SQL-Abfrage aus Controller
Fundort	/platforms/application/controllers/ConferencesController.php
Team N Ruby	
Auffälligkeit	SQL-Abfrage aus Controller

Fundort[9]	/home/hk/ws/pf/platforms2011-sources/... ... app/controllers/members_controller.rb ... app/controllers/status_controller.rb ... app/controllers/admin/conferences_controller.rb
Team O Perl	
Auffälligkeit	Formulare mit HTML::FormHandler erzeugt
Fundort[9]	/CaP/lib/CaP/Form/Conference.pm /CaP/lib/CaP/Form/Member.pm /CaP/lib/CaP/Form/Search/Conference.pm /CaP/lib/CaP/Form/Search/Member.pm
Team P Ruby	
Auffälligkeit	SQL-Abfrage aus Controller
Fundort[9]	/app/controllers/home_controller.rb

Tabelle 4.1: Auffälligkeiten in Bezug auf Modularisierung nach dem MVC-Muster

[9]relativer Pfad ausgehend vom *„CaP-sources“*-Ordner

5 Auswertung

In diesem Kapitel erfolgt die Auswertung der erhobenen Daten und die Darlegung, welche Konsequenzen sich daraus ergeben. Zunächst werden die Ergebnisse für die im Kapitel 3 vorgestellten Metrik geschildert. Anschließend erfolgt die Auswertung zum Kapitel 4.

5.1 Auswertung der Referenzierungstiefe

Abbildung 5–1 auf der nächsten Seite zeigt die absolute Anzahl referenzierter Variablen (auf der Ordinatenachse). In grün (rechten Balken) dargestellt, die Variablen mit einer Referenzierungstiefe von eins. Alle Variablen mit einer Tiefe größer als eins sind durch die roten, linken Balken dargestellt.

Für eine bessere Vergleichbarkeit der Plattformen sind die Werte der Teams auf der Abszissenachse nach Plattformen sortiert. Es gilt anzumerken, dass Team D und O in dieser Darstellung fehlen. Team D aufgrund der besonderen Art der Benutzeroberflächenerzeugung und Team O mangels einer funktionsfähigen Oberfläche (vgl. Abschnitt 3.5.2 auf Seite 38).

Aus der Abbildung 5–1 auf der nächsten Seite ist ersichtlich, dass die Anzahl der referenzierten Variablen auch innerhalb der Plattformen sehr unterschiedlich ist. Das kann diverse Gründe haben. Möglicherweise hat das verwendete Template-Framework einen Einfluss darauf, wie viele Variablen benötigt werden, um die Oberfläche umzusetzen. Beispielsweise ist das beim Template-Framework Mustache der Fall. Das Framework prüft automatisch, ob referenzierte Variablen

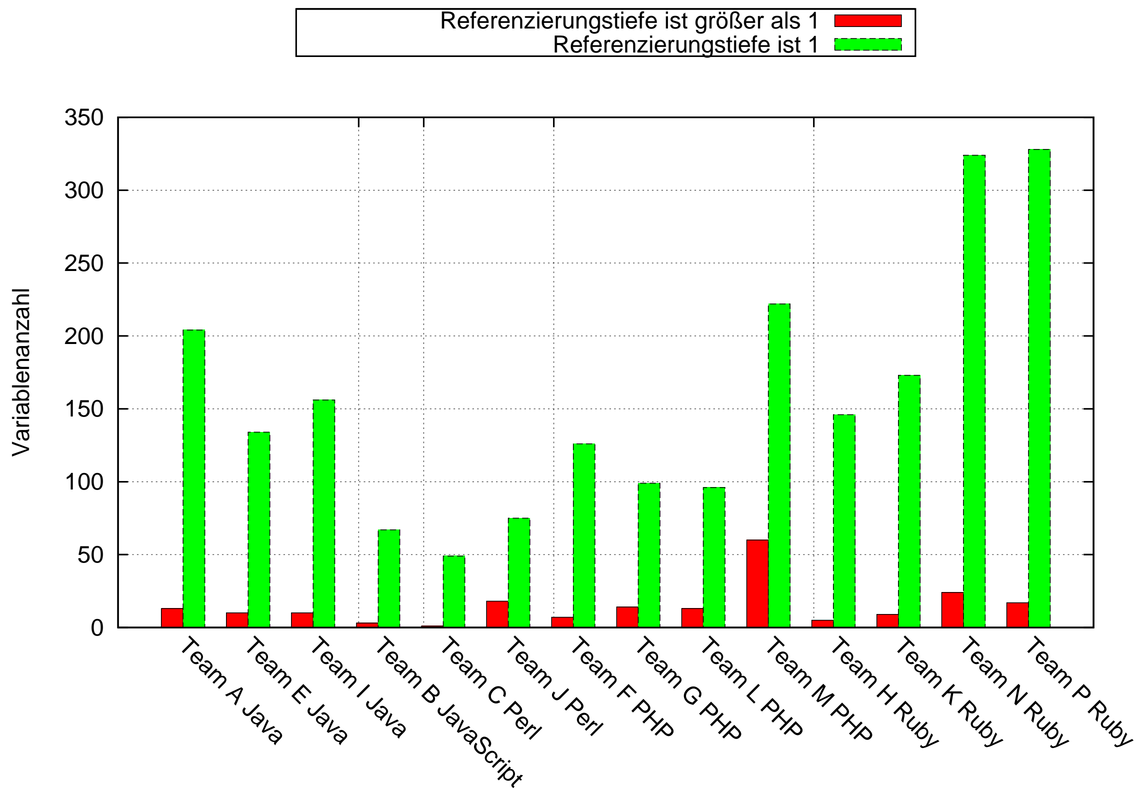


Abbildung 5–1: Referenzierungstiefe der Teams

gesetzt sind (vgl. Abschnitt 3.5.1 auf Seite 30). Es werden weniger Referenzierungen benötigt als bei Template-Frameworks, bei denen diese Aufgabe durch den Software-Entwickler realisiert werden muss. Eine weitere Möglichkeit könnte sein, dass der unterschiedlichen Zahl der Referenzierungen, die Zeitbegrenzung für den Plat_Forms-Wettbewerb von zwei Tagen zu Grunde liegt, also nicht alle Anforderungen vollständig umgesetzt wurden.

Für die weitere Betrachtung wird angenommen, dass die Art der Referenzierung davon unberührt ist. Die vorhandenen Referenzierungen spiegeln entsprechend die allgemeine Art der Referenzierungen für diese Plattform wider. Dementsprechend würden weitere Zugriffe in ähnlicher Art und Weise realisiert.

Unter der getroffenen Annahme zeigt Abbildung 5–2 auf der nächsten Seite auf der Ordinatenachse das prozentuale Verhältnis der Referenzierungen mit Tiefe größer als eins – zu denen mit Tiefe gleich eins.

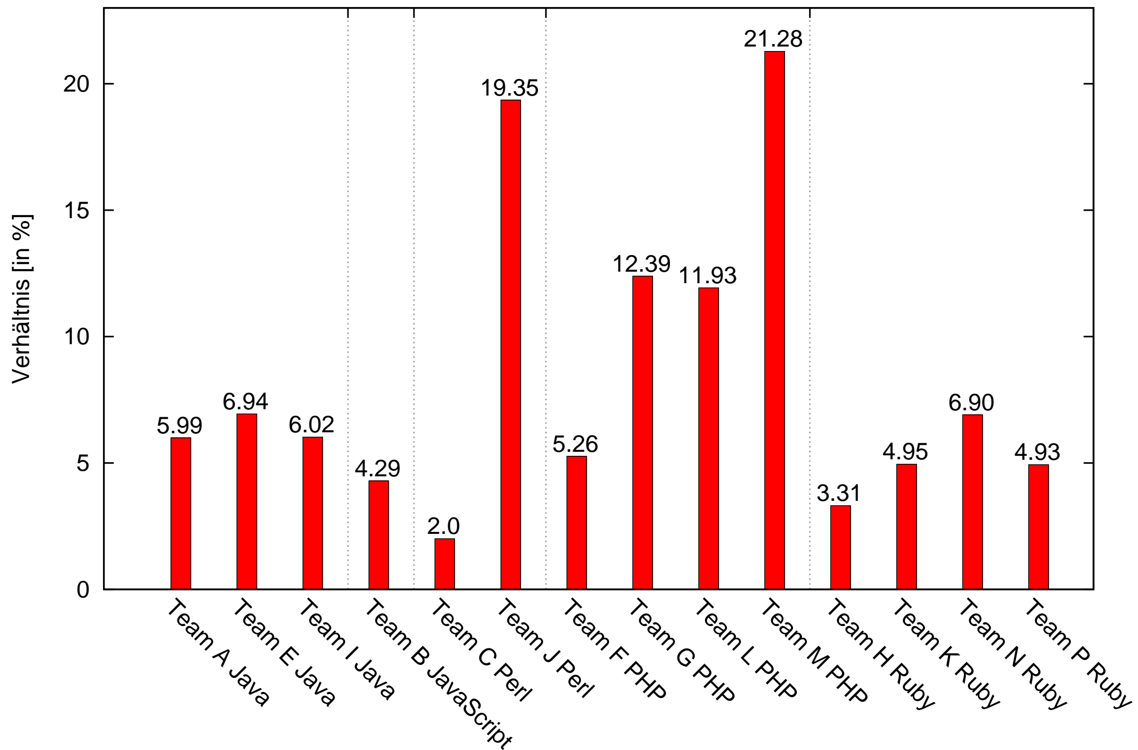


Abbildung 5–2: Verhältnis von Referenzierungen mit Verletzung des Gesetzes von Demeter, zu denen ohne Verletzung

Die Teams D und O fehlen auch in dieser Abbildung. Die Gründe sind identisch zu denen von Abbildung 5–1 auf der vorherigen Seite. Die Sortierung auf der Abszissenachse ist wiederum nach Plattformen vorgenommen worden.

Bei den Perl-Teams ist zu erkennen, dass die Verhältnisse von Team C und J sehr unterschiedlich sind. Während Team C das niedrigste Verhältnis aufweist, findet sich bei Team J das zweithöchste. Der große Unterschied zwischen Team C und Team J liegt in der Vollständigkeit begründet. Bei Team J existiert ein Template einer Statusseite für den Benutzer. Auf dieser Seite werden dem Benutzer seine befreundeten Kontakte, Kontaktanfragen und Kalendereinträge gezeigt. In diesem Template finden sich die meisten Referenzierungen mit einer Tiefe größer als 1. Team C hat ein Template mit gleichem Zweck definiert, dieses ist aber nicht in anderen Templates verlinkt. Es gehört nicht zu den für den Benutzer der Webanwendung verfügbaren Views. Es fällt aufgrund des im Abschnitt 3.6.1 auf Seite 39 festgelegten Vorgehens aus der Betrachtung heraus. Bedingt durch das

Fehlen des dritten Teams O ist es deshalb schwierig eine allgemeine Aussage für die Perl-Teams zu treffen.

Die PHP-Teams liegen im Durchschnitt über allen anderen Teams, wobei Team F ein niedriges Verhältnis hat. Unter den PHP-Teams findet sich aber auch mit Team M die Lösung mit dem höchsten Verhältnis. Ausschlaggebend für die Unterschiede der PHP-Teams könnte das verwendete Webframework sein. Team F (niedrigster Wert für PHP) nutzte FLOW3. Die beiden mittleren Werte stammen von Team G und L, die das Zend-Framework verwendeten. Team M (mit dem höchsten Verhältnis) realisierte seine Lösung mit Symfony.

Die Java-Teams weisen im Schnitt bessere und niedrigere Verhältnisse auf, als die PHP-Teams. Weiterhin ist zu erkennen, dass die Verhältnisse sehr ähnlich zueinander sind. Es existieren keine starken Schwankungen wie bei dem Perl- und PHP-Teams.

Die Lösungen der Ruby-Teams weisen ebenso ein ähnliches Verhältnis auf. Im Vergleich ist das Verhältnis der Verstöße das Niedrigste aller Plattformen.

Das JavaScript-Team B ist wegen der Einzelteilnahme zwar außer Konkurrenz, weist aber in diesem Fall ein ähnlich niedriges Verhältnis auf, wie die Ruby-Teams.

Aus diesen Erkenntnissen leiten sich folgende Kernaussagen ab:

- Eine gesicherte Modularität findet sich vor allem bei den Plattformen Ruby und Java
- Die PHP-Lösungen weisen eine deutlich stärkere Kopplung und damit verbundene schlechtere Modularität auf, als die der Java- und Ruby-Teams
- Unglücklicherweise kann für Perl keine getroffen werden, weil sich die zwei ausgewerteten Teams stark unterscheiden

5.1.1 Bewertung von Team D als Sonderfall

Im Abschnitt 3.5.1 auf Seite 28 wurde für Team D Java bereits geschildert, dass die Benutzeroberfläche durch *WorkScreen*s realisiert wird.

Das Framework verbirgt gegenüber dem Software-Entwickler die Erzeugung der Oberfläche. Clientseitig wird lediglich die Datenquelle der darzustellenden Daten angegeben. Außerdem müssen noch Layout und Editor festgelegt werden, mit dem die Daten dargestellt und editierbar werden (z.B. ein Tabelleneditor). Serverseitig wird die entsprechende Datenquelle erzeugt und beispielsweise an eine Datenbank angebunden [SIBa].

Es ist festzustellen, dass sich diese Lösung gänzlich von allen anderen im Wettbewerb unterscheidet. Es werden keinerlei Templates für die Erzeugung der Benutzeroberfläche verwendet. Layout und Zugriff auf serverseitige Daten werden im *WorkScreen* vereint. Diese Vermischung kann den Quelltext unübersichtlich machen.

Ein Punkt, der als vorteilhaft in der Dokumentation beschrieben wird, ist die automatische Erzeugung von Master-Detail-Beziehungen. Ein Beispiel dafür ist eine Auswahlliste für Kategorien (Master). Bei Auswahl einer Kategorie wird eine Auswahlliste mit Konferenzen (Detail) geladen, die zu dieser Kategorie zugeordnet sind. Für die Realisierung einer solchen Beziehung müssen die Schlüsselnamen der Datenbank angegeben werden, über die Master und Detail verknüpft sind [SIBc]. Es ist anzumerken, dass die Master-Detail-Beziehung im *WorkScreen* definiert wird. Daraus ergibt sich eine schlechtere Modularität, weil eine Kopplung an die Datenbank-Definition erzeugt wird.

Neben diesen negativen Aspekten konnten keine positiven Auswirkungen auf die Modularität (durch die Verwendung des Frameworks) festgestellt werden.

5.1.2 Anmerkungen zu Team O

Das von Team O eingesetzte Template-Framework bietet grundsätzlich eine gute Voraussetzung für eine lose Kopplung. Die View besteht nur aus HTML-Elemen-

ten, die vom Controller mit Daten „befüllt“ werden. Diese Funktionsweise verhindert Anwendungslogik in der View und somit auch Referenzierungen des Model aus der View. Sie ist also dem MVP-Muster (vgl. Abschnitt 2.4.2 auf Seite 18) zuzuordnen. Grundsätzlich könnte sich das positiv auf die Modularität auswirken, da die View vom Model entkoppelt ist.

5.2 Auswertung der Model-View-Controller-Verstöße

Aus Tabelle 4.1 auf Seite 51 geht hervor, dass es für die Java-Teams keine plattformweite Eigenschaft gibt. Der Zugriff von Team A auf ein Model direkt aus der View ist (aufgrund des einmaligen Aufkommens) als Unabsichtlichkeit zu werten.

Bei Team D ist keine Trennung nach dem MVC-Muster vorhanden. In Abschnitt 3.5.1 auf Seite 29 wurde bereits geschildert, dass die Oberfläche mit JVx mittels *WorkScreen* erzeugt wird. In der Lösung von Team D findet sich sowohl View-Code, als auch controllerspezifischer Code. Zum Einen werden die Oberflächenelemente und deren Layout definiert. Zum Anderen findet sich Applikationslogik für die Weiterleitung zu anderen Views. Verglichen zu den anderen Lösungen liegt hier eine schwächere Kohäsion vor. In der Kategorisierung, wie sie auf Seite 8 vorgestellt wurde, fällt die Lösung somit in die Kategorie der *logischen Kohäsion*. Die Verarbeitungsschritte sind logisch zum *WorkScreen* zugehörig, besitzen aber unterschiedliche Zuständigkeiten (Darstellung bzw. Applikationslogik).

Bei den PHP-Teams findet sich ebenfalls keine konsistente Plattformeigenschaft. Team F hat im Quelltext eines Controller HTML-Tags verwendet. Dabei handelt es sich um Quelltext für die Darstellung zweier HTML-Links. Der eine führt zur Suche von Konferenzen, der andere zum Formular für das Anlegen einer neuen Konferenz. Eine bessere Lösung wäre die Verwendung eines Teiltemplates gewesen, das die HTML-spezifischen Teile ausgelagert hätte.

Bei Team L PHP und Team N Ruby, sowie Team P Ruby wurden SQL-Abfragen im Controller verwendet. Diese stellen jedoch den Zugriff auf die Daten her und ge-

hören deshalb ins Model. Basierend auf einem Quelltextkommentar[10] von Team N lässt sich die These formulieren, dass dies nur eine Alternative war, die Entwicklungszeit einsparte. Somit konnten weitere Anforderungen umgesetzt werden. So lässt sich ebenfalls vermuten, dass mit mehr Entwicklungszeit eine saubere Trennung vollzogen worden wäre.

Die einzige konsistente Eigenschaft findet sich bei den Perl-Teams. Alle drei Teams verwenden *HTML::FormHandler* für die Erzeugung von Formularen. Nachfolgend soll untersucht werden, wie ausgeprägt der Einfluss auf die Modularität ist. Zunächst stellt sich die Frage, inwieweit es sich um einen Verstoß gegen das MVC-Muster handelt. Üblicherweise werden beim Catalyst-Framework Model, View und Controller in einem jeweils zugehörigen, gleichnamigen Ordner abgelegt. Ein Controller wird im Ordner „Controller" abgelegt, gleiches gilt für Model und View [Cat11]. Formulare finden sich aber im Ordner „Form" beziehungsweise „Forms" wieder. Diese Besonderheit lässt nicht sofort darauf schließen, ob ein Formular nun als Model, View, Controller oder sogar als eine Mischform einzuordnen ist. Im Folgenden werden dazu die konkreten Lösungen der Perl-Teams näher betrachtet.

```
1 use HTML::FormHandler::Moose;
2 has_field 'field_name' => (
3    type => 'FieldClass',
4    ...
5 );
```

Quelltext 5.1: Syntax für die Erzeugung eines Formulars mit HTML::FormHandler [SLD+11a]

Zunächst zur allgemeinen Vorgehensweise für das Definieren eines Formulars mit *HTML::FormHandler*: Ein Eingabelement[11] wird nach der im Quelltext 5.1 gezeigten Syntax definiert.

In Zeile 1 wird zunächst das benötigte Plugin *HTML::FormHandler::Moose* eingebunden. Dieses bietet die Möglichkeit, die in den darauffolgenden Zeilen verwen-

[10] Im Kommentar heißt es „#some crazy sql for M102:" - sinngemäß „etwas merkwürdiges SQL für die Mussanforderung 102.

[11] Das kann z.B. ein Eingabefeld, eine Auswahlliste o.ä. sein

dete Syntax einzusetzen. Mit der *has_field*-Deklaration in Zeile 2, wird ein Eingabeelement für das Formular definiert. Dieses erhält, sofern nicht explizit definiert, standardmäßig die Beschriftung „field_name“. Wie mit Zeile 3 und 4 angedeutet, folgt eine kommaseparierte Liste von Attributen für das definierte Eingabeelement. Diese Attribute legen bestimmte Eigenschaften eines Eingabeelementes fest. In [Sha11] beschreibt Shank die verfügbaren Attribute. Nachfolgend werden die von den Perl-Teams verwendeten Attribute näher erläutert und anhand ihrer Beschreibung in [Sha11] eingeordnet.

Das *type*-Attribut legt fest, um welche Art von Eingabeelement es sich handelt. Das kann beispielsweise ein Texteingabe- oder ein Auswahlfeld sein. Dieses Attribut beeinflusst einerseits die Darstellung, weil dadurch festgelegt wird, wie das Element gerendert wird. Andererseits wird es auch verwendet, um die eingegebenen Daten zu validieren.

Mit der *required*-Eigenschaft wird ein Eingabeelement als notwendig gekennzeichnet. Das bedeutet, es muss ausgefüllt beziehungsweise ausgewählt sein, damit das Formular erfolgreich verarbeitet werden kann. Somit ist dieser Teil eher der Programmlogik zuzuschreiben und sollte im Sinne guter Modularisierung von einem Controller übernommen werden.

Weiterhin bietet das *options*-Attribut die Möglichkeit, die Einträge einer Auswahlliste zu definieren. Grundsätzlich entspricht das einer View – aber es finden sich bei Team C und O entsprechende Aufrufe an das Model zur Datenabfrage. Das wiederum entspricht der Aufgabe, die üblicherweise ein Controller übernimmt.

Ebenso in Verwendung ist das *value*-Attribut, welches den Wert eines Eingabe-Elements festlegt. Ein solcher Wert wird vom Controller für die Weiterverarbeitung benötigt und gegebenenfalls ans Model übergeben. Dieses Attribut fällt also eher in den Aufgabenbereich eines Controllers. Für die View wird der Wert nicht benötigt.

Darüber hinaus verwendet Team C das *unique*-Attribut. Dieses besagt, dass ein Eingabeelement lediglich ein Mal in der Datenbank vorkommen darf. Auch diese Eigenschaft fällt in mehrere Zuständigkeitsbereiche. Einerseits kann eine Validie-

rung stattfinden, ob bereits ein Wert in der Datenbank vorhanden ist. Diese Programmlogik sollte in einem Controller definiert werden. Andererseits sollte der Benutzer des Systems durch eine Bildschirmmeldung darüber in Kenntnis gesetzt werden, dass dieser Wert bereits vergeben ist. Somit betrifft das auch die entsprechende View.

Ein weiterer Punkt der bei der Verwendung von *HTML::FormHandler* zu erwähnen ist, ist die automatische Speicherung von Formularen. Unter Verwendung von *HTML::FormHandler::TraitFor::Model::DBIC*, einer Erweiterung für Formulare, ist es möglich Formularfelder direkt in der Datenbank zu speichern. Sofern alle Daten valide sind und das Formular abgeschickt wird, erfolgt die Speicherung ohne weitere Controller-Interaktion. Allerdings gilt eine Voraussetzung: Der Elementname (vgl. Quelltext 5.1 auf Seite 59, Zeile 2) im Formular muss mit dem der Spaltendefinition beziehungsweise Datenbankzugriffsnamen oder Datenbankrelation übereinstimmen [SLD+11b]. Dadurch wird eine starke Kopplung zwischen Model und View erzeugt. Liegt beispielsweise ein Schreibfehler beim Elementnamen vor, müssen sowohl der Name im Formular als auch in der Definition des Datenbankschemas angepasst werden. Abhilfe kann die Nutzung des *label*-Attributs schaffen (vgl. [Sha11]). Darüber lässt sich die Beschriftung eines Elementes setzen. Für die Umsetzung einer Mehrsprachigkeit ist dieses Attribut ohnehin vonnöten.

Zusammenfassend lässt sich sagen, dass sich die Verwendung von *HTML::FormHandler* negativ auf die Modularität auswirken kann. Einige Möglichkeiten die sich damit bieten, erlauben eine Vermischung verschiedener Zuständigkeiten. So zeigt sich in den vorliegenden Perl-Lösungen eine Überschneidung von View und Controller, in einem Fall sogar von View und Model (siehe Erläuterungen zum *options*-Attribut, Seite 60).

6 Zusammenfassung und Ausblick

In diesem Kapitel werden die zentralen Erkenntnisse noch einmal zusammengefasst. Darüber hinaus wird ein Ausblick für zukünftige Studien gegeben.

Ziel dieser Masterarbeit war die Untersuchung der Modularität von Webanwendungen verschiedener Plattformen. Zunächst wurden Kopplung und Kohäsion als zentrale Konzepte der Modularität betrachtet. In einem weiteren Schritt wurden Verfahren der allgemeinen Modularitätsbetrachtung von Anwendungen evaluiert. Mit der anschließenden Analyse wurde überprüft, welche Verfahren durch eine Erweiterung auf die Plat_Forms-Lösungen angewendet werden können. Hierzu wurden die einzelnen Schritte zur Entwicklung einer Pseudometrik durchgeführt und dargelegt. Die entwickelte Metrik ermöglicht, die Kopplung der View-Komponenten zum Rest des Systems zu messen. Dafür wurde ein Vorgehen entwickelt, die verwendeten Templates der Views zu identifizieren. Im Folgeschritt konnten daraus die Referenzierungen von Variablen ermittelt werden. Durch die Auswertung im Abschnitt 3.6.2 auf Seite 40 wurden wichtige Eigenschaften nachgewiesen, die eine Metrik nach Ludewig und Lichter [LL07] besitzen sollte. Das bestätigt die Qualität der Metrik.

In einer Evaluation der Metrikergebnisse wurden die Erkenntnisse geschildert, die sich für die Modularität ergeben. So konnte gezeigt werden, dass die Ruby- und Java-Lösungen gegenüber denen von PHP eine geringere Kopplung aufwiesen. Durch die geringere Kopplung folgt eine bessere Modularität. Einen konkreten Vergleich für Perl zu ziehen gestaltete sich als schwierig. Einerseits gab es für diese Plattform nur drei teilnehmende Teams. Von diesen konnte ein Team aus den im Abschnitt 3.5.2 auf Seite 38 dargelegten Gründen nicht in die Untersuchung einbezogen werden. Andererseits wiesen die verbleibenden zwei Teams

deutlich unterschiedliche Werte auf. Aus diesem Grund konnte leider keine Aussage über die Plattform in Bezug zu den anderen gemacht werden.

Der zweite untersuchte Punkt ergab sich aus Erkenntnissen bei der Anwendung der Metrik. Einige Module schienen nicht stimmig nach dem MVC-Muster gegliedert zu sein. Aus diesem Grund wurden zunächst die strittigen Punkte mit einer Durchsicht der Quelltexte ermittelt. Dem wurden die spezifischen Ausprägungen des MVC-Musters, der jeweiligen Frameworks, gegenübergestellt. Anschließend wurde untersucht, ob die Unstimmigkeiten tatsächliche Verstöße gegen das Muster sind. Dabei konnte bei den Perl-Teams aufgedeckt werden, dass die Art und Weise der Formulardefinition fragwürdig ist. Durch das verwendete Perl-Plugin sind bei Formularen Model, View und Controller nicht klar abgegrenzt. Dem gegenüber stand eine sonst konsequente Trennung.

Mit den vorgestellten Untersuchungsverfahren konnten Unterschiede der Modularität verschiedener Plattformen vergleichbar aufgezeigt werden. Es finden sich aber auch Schwächen beim geschilderten Vorgehen. Eine Problematik ist der große Anteil manueller Datenerfassung. Ein solches Vorgehen ist potentiell fehlerträchtig und bedeutet einen großen Aufwand. Für zukünftige Untersuchungen erscheint es sinnvoll, den Datenerhebungsprozess zu automatisieren. Bei der Einarbeitung in die Thematik der Masterarbeit wurde auch untersucht, ob Parser existieren, mit denen man diesen Prozess durchführen könnte. Da dies jedoch nicht der Fall war, müssten eigene Parser erstellt werden. Aus der Recherche ging ANTLR als sinnvolle Möglichkeit hervor, Parser zu erzeugen.

Ein weiteres Problem, das es für zukünftige Untersuchungen zu lösen gilt, ist die unterschiedliche Vollständigkeit der Lösungen. Es ist durchaus denkbar, dass einige Ergebnisse diesem Einfluss unterliegen und deshalb nicht repräsentativ sind. Außerdem sollte eine statistische Analyse der Ergebnisse durchgeführt werden, um diese auf ihre Gültigkeit zu prüfen. Zu diesem Zweck werden die erhobenen Daten auf einem Datenträger (Anhang B) bereitgestellt. Eine Beschreibung des exportierten Datenformats findet sich in Anhang A.

Abbildungsverzeichnis

2–1 Zusammenhang von Kopplung und Kohäsion 9
2–2 Beispiel einer Design Structure Matrix 14
2–3 Funktionale Struktur eines Webserver 17
2–4 MVC- und MVP-Architekturmuster 19
2–5 Das Front-Controller-Muster 21

3–1 Zulässiger Variablenzugriff nach dem Gesetz von Demeter 25
3–2 Auszug aus Mind-Map von Team E 39

4–1 Anfragebearbeitung in Spring 45
4–2 Anfragebearbeitung in Catalyst 47
4–3 Ablauf in Symfony 48
4–4 Anfragebearbeitung in Zend 49

5–1 Referenzierungstiefe der Teams 54
5–2 Verhältnis von Referenzierungen mit Verletzung des Gesetzes von Demeter, zu denen ohne Verletzung 55

Abkürzungsverzeichnis

ANTLR Another Tool for Language Recognition
CBO Coupling between object classes
CMS Content Management System
CSS Cascading Style Sheets
DSM Design Structure Matrix
ERB Embedded Ruby
GWT Google Web Toolkit
HAML XHTML Abstraction Markup Language
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
IEEE Institute of Electrical and Electronics Engineers
JSF JavaServer Faces
JSP JavaServer Pages
JSTL JSP Standard Tag Library
LCOM Lack of cohesion in modules
MVC Model-View-Controller
MVP Model-View-Presenter
PDF Portable Document Format
PHP PHP Hypertext Preprocessor
REST Representational State Transfer
SWT Standard Widget Toolkit
SQL Structured Query Language
URI Uniform Resource Identifier
URL Uniform Resource Locator
WASC Web Application Security Consortium
WMC Weighted methods per class
XHTML Extensible Hypertext Markup Language
XML Extensible Markup Language

Tabellenverzeichnis

3.1 Syntax JavaServer Pages (JSP) 29
3.2 Syntax Mustache 31
3.3 Syntax Fluid 32
3.4 Syntax PHP (Zend Form) 33
3.5 Syntax PHP (symfony) 33
3.6 Syntax Catalyst (Template Toolkit) 33
3.7 Syntax Catalyst (HTML::Zoom) 34
3.8 Syntax HAML 35
3.9 Syntax Embedded Ruby (ERB) 35

4.1 Auffälligkeiten in Bezug auf Modularisierung nach dem MVC-Muster 52

Quelltextverzeichnis

3.1 Verwendung temporärer Variable in PHP-Template 26

5.1 Syntax für die Erzeugung eines Formulars mit HTML::FormHandler [SLD+11a] 59

Literaturverzeichnis

[BC03] BALDWIN, C. Y. und K. B. CLARK: *Managing in the modular age: architectures, networks, and organizations*, Kapitel 5, Seiten 149–171. Wiley, 2003.

[Boc00] BOCK, D.: *The paperboy, the wallet, and the law of demeter*, 2000.

[Cat11] CATALYST CONTRIBUTORS: *Catalyst::Manual::Intro*. `http://search.cpan.org/perldoc?Catalyst::Manual::Intro`, 2011. Abruf am: 17. 11. 2011.

[CK94] CHIDAMBER, S.R. und C.F. KEMERER: *A metrics suite for object oriented design*. IEEE Transactions on software engineering, 20:476–493, 1994.

[CW00] COZENS, S. und P. WAINWRIGHT: *Beginning Perl*. Wrox Press Ltd., Birmingham, UK, UK, 2000.

[CWE11] CATLIN, H., N. WEIZENBAUM und C. EPPSTEIN: *Haml (XHTML Abstraction Markup Language)*. `http://haml-lang.com/docs/yardoc/file.HAML_REFERENCE.html`, 2011. Abruf am: 16. 06. 2011.

[Epp91] EPPINGER, S.D.: *Model-based approaches to managing concurrent engineering*. Journal of Engineering Design, 2(4):283–290, 1991.

[FLO11] FLOW3 CORE TEAM: *FLOW3 The Definitive Guide*. `http://flow3.typo3.org/fileadmin/manual/nightly/TheDefinitiveGuide/PartII/View.html`, 2011. Abruf am: 02. 09. 2011.

[Fow03] FOWLER, M.: *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[Fow06] FOWLER, M.: *Passive View*. `http://www.martinfowler.com/eaaDev/PassiveScreen.html`, 2006. Abruf am: 25. 07. 2011.

[Gor08] GORDEYCHIK, S.: *Web Application Security Statistics*. `https://files.pbworks.com/download/1ANBwCRb7j/webappsec/13247070/WASS-SS-2008.pdf`, 2008. Abruf am: 16. 08. 2011.

[HHP02] HARMAN, M., R. HIERONS und M. PROCTOR: *A new representation and crossover operator for search-based optimization of software modularization*. In: *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, Seiten 1351–1358, 2002.

[HM96] HITZ, M. und B. MONTAZERI: *Chidamber and Kemerer's metrics suite: a measurement theory perspective*. Software Engineering, IEEE Transactions on Software Engineering, 22(4):267–271, 1996.

[IEE90] IEEE: *IEEE Standard Glossary of Software Engineering Terminology*. Office, 121990(1):17,22,49, 1990.

[Joh10] JOHN, ANTANO SOLAR: *Catalyst 5.8: The Perl MVC Framework*. Packt Publishing Ltd., 2010.

[JPMM04] JABLONSKI, S., I. PETROV, C. MEILER und U. MAYER: *Guide to web application and platform architectures*. Springer Professional Computing. Springer, 2004.

[Lak93] LAKHOTIA, A.: *Rule-based approach to computing module cohesion*. In: *Proceedings of the 15th international conference on Software Engineering*, Seiten 35–44. IEEE Computer Society Press, 1993.

[LD06] LADD, S. und K. DONALD: *Expert Spring MVC and Web Flows*. Apress, 2006.

[LH89] LIEBERHERR, K. und I. HOLLAND: *Formulations and Benefits of the Law of Demeter*. ACM SIGPLAN Notices, 24(3):67–78, 1989.

[LHR88] LIEBERHERR, K., I. HOLLAND und A. RIEL: *Object-oriented programming: an objective sense of style*. In: *ACM SIGPLAN Notices*, Band 23, Seiten 323–334. ACM, 1988.

[LL07] LUDEWIG, J. und H. LICHTER: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Dpunkt-Verl., 2007.

[LR06] LAHRES, B. und G. RAÝMAN: *Praxisbuch Objektorientierung*. Galileo Computing, 2006.

[McC76] MCCABE, T.J.: *A complexity measure*. IEEE Transactions on Software Engineering, 2:308–320, 1976.

[MGL+11] MILLER, J., M. GUNDERLOY, M. LINDSAAR, J. INIESTA, R. ARONDEKAR, Y. KATZ, S. MARTINEZ und P. SICHANUGRIST: *Getting Started with Rails*. http://guides.rubyonrails.org/getting_started.html, 2011. Abruf am: 16. 10. 2011.

[Mic08] MICROSOFT DEVELOPER NETWORK: *Model-View-Presenter Pattern*. http://msdn.microsoft.com/en-us/library/ff647543.aspx, 2008. Abruf am: 25. 07. 2011.

[MM06] MITCHELL, B.S. und S. MANCORIDIS: *On the automatic modularization of software systems using the bunch tool*. IEEE Transactions on Software Engineering, 32:193–208, 2006.

[OHK93] OFFUTT, A.J., M.J. HARROLD und P. KOLTE: *A software metric system for module coupling*. Journal of Systems and Software, 20(3):295–308, 1993.

[Ora11] ORACLE: *JSP Standard Tag Library*. http://www.oracle.com/technetwork/java/index-jsp-135995.html, 2011. Abruf am: 01. 09. 2011.

[Par72] PARNAS, D. L.: *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12):1053–1058, 1972.

[PP03] PRECHELT, L. und M. PETER: *Integrating a Tool into Multiple Different IDEs*. In: *3rd International Workshop on Adoption-Centric Software Engineering*, 2003.

[Pre07] PRECHELT, L.: *Plat_Forms 2007: The Web Development Platform Comparison - Evaluation and Results*. Technischer Bericht, Institut für Informatik, Freie Universität Berlin, 2007.

[Pre10] PRECHELT, L.: *Software Engineering Economics*. Vorlesungsskript: Softwareprozesse WS 2010/2011, 2010. Freie Universität Berlin, Institut für Informatik.

[PZ07] POTENCIER, F. und F. ZANINOTTO: *The Definitive Guide to symfony*. Apress, Berkely, CA, USA, 2007.

[Ree79] REENSKAUG, T.: *Models-views-controllers*. Technischer Bericht, Xerox PARC, 1979.

[Sen11] SENSIO LABS: *A Gentle Introduction to symfony*. http://www.symfony-project.org/get/pdf/gentle-introduction-1.4-en.pdf, 2011. Abruf am: 16. 06. 2011.

[SGCH01] SULLIVAN, K.J., W.G. GRISWOLD, Y. CAI und B. HALLEN: *The structure and value of modularity in software design*. In: *ACM SIGSOFT Software Engineering Notes*, Band 26, Seiten 99–108. ACM, 2001.

[Sha11] SHANK, G.: *HTML::FormHandler::Manual::Reference*. https://metacpan.org/module/HTML::FormHandler::Manual::Reference, 2011. Abruf am: 17. 11. 2011.

[She11] SHEIDLOWER, JESSE: *Catalyst::Manual::About The philosophy of Catalyst*. http://search.cpan.org/perldoc?Catalyst::Manual::About, 2011. Abruf am: 16. 07. 2011.

[SIBa] SIB VISIONS GMBH: *JVx Enterprise Application Framework - Anzeige von Daten aus einer Datenbank*. http://forum.sibvisions.com/viewtopic.php?f=5&t=23. o.J., Abruf am: 20. 10. 2011.

[SIBb] SIB VISIONS GMBH: *JVx Enterprise Application Framework - Features*. http://www.sibvisions.com/de/jvxmfeatures. o.J., Abruf am: 20. 10. 2011.

[SIBc] SIB VISIONS GMBH: *JVx Enterprise Application Framework - Master/Detail-Beziehung Datenbank*. http://forum.sibvisions.com/viewtopic.php?f=2&t=349. o.J., Abruf am: 29.10.2011.

[SLD+11a] SHANK, G., Z. LUKASIAK, T. DORAN, B. GRAF, O. KOSTYUK, F. RAGWITZ, L. A. MESA, D. THOMAS, K. ITA, J. NAPIORKOWSKI, D. MAKI, A. RODLAND, A. CLAYTON, B. BEELEY, C. HENTENYI, E. OISHI, L. W. SITU, MURRAY, N. LOGAN, V. TIMOFEEV, D. KUPERMAN, I. WELLS, A. BARKSDALE und V. MOSELEY: *HTML::FormHandler*. http://search.cpan.org/perldoc?HTML::FormHandler, 2011. Abruf am: 17. 11. 2011.

[SLD+11b] SHANK, G., Z. LUKASIAK, T. DORAN, B. GRAF, O. KOSTYUK, F. RAGWITZ, L. A. MESA, D. THOMAS, K. ITA, J. NAPIORKOWSKI, D. MAKI, A. RODLAND, A. CLAYTON, B. BEELEY, C. HENTENYI, E. OISHI, L. W. SITU, MURRAY, N. LOGAN, V. TIMOFEEV, D. KUPERMAN, I. WELLS, A. BARKSDALE und V. MOSELEY: *HTML::FormHandler::TraitFor::Model::DBIC*. http://search.cpan.org/perldoc?HTML::FormHandler::TraitFor::Model::DBIC, 2011. Abruf am: 16. 11. 2011.

[SP11] STÄRK, U. und L. PRECHELT: *Plat_Forms 2011 Task: CaP*, 2011.

[SRK07] SARKAR, S., G.M. RAMA und A.C. KAK: *API-based and information-theoretic metrics for measuring the quality of software modularization*. IEEE Transactions on Software Engineering, 33:14–32, 2007.

[TFH04] THOMAS, D., C. FOWLER und A. HUNT: *Programming Ruby: The Pragmatic Programmer's Guide*. Pragmatic Bookshelf, Second Edition Auflage, 2004. Abruf am: 04.05.2011.

[THS+] THOMAS, D., A. HUNT, G. SINCLAIR, E. HODEL, W. WEBBER, L. JOHNSON und J. E. GRAY II: *ERB - Ruby Templating*. http://ruby-doc.org/stdlib/libdoc/erb/rdoc/classes/ERB.html. o.J., Abruf am: 16.06.2011.

[Tro11a] TROUT, MATT: *Alarms, excursions, apologia and conclusions*. http://www.shadowcat.co.uk/blog/matt-s-trout/plat-forms-redux/, 2011. Abruf am: 07.09.2011.

[Tro11b] TROUT, MATT: *HTML::Zoom*. http://search.cpan.org/perldoc?HTML::Zoom, 2011. Abruf am: 30.08.2011.

[TYP11a] TYPO3 ASSOCIATION: *FLOW3 - The Definitive Guide - Part II: Getting Started - Controller*. http://flow3.typo3.org/documentation/guide/partii/controller.html, 2011. Abruf am: 25.10.2011.

[TYP11b] TYPO3 ASSOCIATION: *FLOW3 - The Definitive Guide - Part II: Getting Started - Model and Repository*. http://flow3.typo3.org/documentation/guide/partii/modelandrepository.html, 2011. Abruf am: 25.10.2011.

[TYP11c] TYPO3 ASSOCIATION: *FLOW3 - The Definitive Guide - Part II: Getting Started - View*. http://flow3.typo3.org/documentation/guide/partii/view.html, 2011. Abruf am: 25.10.2011.

[Ull11] ULLENBOOM, C.: *Java ist auch eine Insel*. Galileo Computing, Bonn, 9. aktualisierte Auflage, 2011.

[Vas11] VASWANI, V.: *Zend Framework - A Benginner's Guide*. The McGraw-Hill Companies, 2011.

[Wan] WANSTRATH, C.: *Mustache Manual: mustache – Logic-less templates*. http://mustache.github.com/mustache.5.html. o.J., Abruf am: 14.06.2011.

[War] WARDLEY, A.: *Template-Toolkit-2.22*. http://search.cpan.org/~abw/Template-Toolkit-2.22/lib/Template/Manual/Syntax.pod. o.J., Abruf am: 14.06.2011.

[War08] WARDLEY, ANDY: *Template::Toolkit Template Processing System*. http://search.cpan.org/perldoc?Template::Toolkit, 2008. Abruf am: 20.09.2011.

[YC87] YOURDON, E. und L. L. CONSTANTINE: *Structured Design - Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1987.

[Zen05] ZEND TECHNOLOGIES INC.: *Zend_Form - Programmer's Reference Guide*. http://framework.zend.com/manual/en/zend.form.quickstart.html, 2005. Abruf am: 16.06.2011.

[Zus98] ZUSE, H.: *A framework of software measurement*. de Gruyter, 1998.

Anhang

A Datenformat

Die exportierten Daten finden sich auf dem Datenträger in einer Datei mit dem Namen „export.csv". Es finden sich je Zeile sieben Spalten, diese sind jeweils durch ein Semikolon voneinander getrennt. Die Daten sind direkt aus den Mind-Maps exportiert. In der folgenden Auflistung wird die Bedeutung der einzelnen Spalten näher erläutert.

Spalte 1: Teamname

- Diese Spalte enthält den Namen des Teams im Format **'TeamA'**. Alle Teams folgen in einer alphabetischen Reihenfolge aufeinander.

Spalte 2: Name des Platzhalters

- Die zweite Spalte enthält den Namen des Platzhalters aus dem Template. Der Name wird von einfachen Anführungszeichen umschlossen.

Spalte 3: Zeilennummer

- Die Zeilennummer besagt, wo sich der Platzhalter (Spalte 2) innerhalb des Templates (Spalte 4) befindet.

Spalte 4: Templatename

- Der Templatename ist wiederum von einfachen Anführungszeichen umschlossen.

Spalte 5: Ort der Definition

- Beschreibt die Datei, in der die Variable definiert ist, die zum Platzhalter gehört. (Von einfachen Anführungszeichen umschlossen)

Spalte 6: ermittelte Referenzierungstiefe

- Eine positive ganze Zahl (größer gleich 1) die angibt, wie tief die Referenzierung reicht.

Spalte 7: eingebundenes Template

- Diese Spalte gibt an, wenn ein Template in eine anderes eingebunden wurde. Der Name des eingebundenen Templates steht in einfachen Anführungszeichen. Ist diese Spalte gesetzt, sind die Spalten 2 bis 6 leer.

B CD-ROM

Der beigefügte Datenträger enhält:

- Eine elektronische Version dieser Arbeit als PDF
- Die Mind-Maps (vgl. Abschnitt 3.6.1 auf Seite 39)
- Die aus den Mind-Maps extrahierten Daten. Das Format findet sich im Anhang A.
- Alle digital verfügbaren Quellen