FACHHOCHSCHULE VORARLBERG GMBH

Bachelorarbeit im Fachhochschul-Bachelorstudiengang Informatik

Bachelorarbeit

Automatische Anti-Pattern Korrektur für PHP

ausgeführt von

Franziskus Domig

Personenkennzeichen: 0710247027

Bearbeitung:	Dornbirn, im Juli 2010
Betreuer:	Prof. (FH) Dipl.-Inform. Thomas Feilhauer
Externer Betreuer:	Dipl.-Inf. Florian Anderiasch, Mayflower GmbH

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, 13. Juli 2010

. . . . . .

Vorwort

Diese Arbeit wäre ohne die Ideen, Hilfe und Förderung der nachfolgenden Personen und Unternehmen nicht möglich gewesen. Ich möchte mich für die Unterstützung recht herzlich bedanken.

Florian Anderiasch, Daniel Corson, Ulrich Dangel, Thomas Feilhauer, Johann-Peter Hartmann, Claudia Lange, Mayflower GmbH, Thorsten Rinne, Periklis Tsirakidis.

Abschließend möchte ich mich bei allen Personen bedanken, die in die Entwicklung der freien Software involviert waren und sind, welche in dieser Arbeit vorgestellt (lex-pass, PHP, phploc, phpunit, phpMyFAQ, PHProjekt, Wordpress) und verwendet (git, LaTeX, Subversion) wurden, bedanken.

Kurzfassung

Die Programmiersprache PHP hat sich seit der Einführung von PHP 4 im Jahre 2000 sehr stark weiterentwickelt. Zwischenzeitlich wurden durch die rasche Weiterentwicklung der Sprache viele der in PHP 4 verwendeten Konstrukte in PHP 5 ersetzt, beispielsweise wurden Exceptions[1], abstrakte Klassen, sowie Interfaces eingeführt. Zusätzlich werden Objekte bei Funktionsaufrufen sowie bei Zuweisung an eine andere Variable nicht mehr grundsätzlich kopiert (Call by Value), sondern es wird jeweils eine Referenz auf das Originalobjekt übergeben (Call by Reference) [Gutmans 04, S. 1]. Viele veraltete und mittlerweile als schlecht angesehene Konstrukte aus PHP 4 bestehen allerdings auch heute noch in verwendetem Code und führen unweigerlich zu Problemen.

Viele Unternehmen haben dadurch vor allem in den Bereichen Sicherheit, sowie Performanz zu kämpfen. Manuelles Refactoring[2] von bestehenden Programmen benötigt sehr viel Zeit und wird von vielen Unternehmen deswegen nicht durchgeführt.

Gewisse Ansätze für automatische Refactorings wurden bereits im Rahmen der Open-Source Initiative von Facebook mit dem Werkzeug lex-pass geliefert [Lex-Pass 10]. Dadurch lassen sich Transformationen am *Abstract Syntax Tree*[3] von PHP-Applikationen vornehmen. Diese Transformationen werden für lex-pass anhand von in Haskell geschriebenen Regeln durchgeführt.

Im Rahmen dieser Bachelorarbeit werden auf PHP zutreffende *Anti-Patterns* gesucht und dafür entsprechende Transformationen in Haskell geschrieben. Ziel ist es, einige einfache Anti-Patterns automatisiert zu korrigieren. Anschließend werden die erreichten Transformationen vorgestellt, sowie weitere mögliche Erweiterungen diskutiert. Zur Demonstration der erstellten Transformationen werden einige bekannte Open-Source Projekte automatisch analysiert und korrigiert.

Im Abschluss erläutert ein Fazit die Vor- und Nachteile von automatisierter Korrektur. Gleichzeitig wird auch auf sinnvolle Einsatzgebiete, sowie auf Einschränkungen hingewiesen.

[1] Ausnahme- bzw. Fehlerbehandlung

[2] Strukturverbesserung von Programm-Quelltexten

[3] AST, logisch Baumstruktur von Programmcode

Abstract

Since the first release of the programming language PHP, PHP has been rapidly developed. As a result to these huge improvements, many of the features from PHP 4 have been rewritten or replaced in PHP 5. For instance exception handling, abstract classes and interfaces have been introduced. Additionally, objects are not getting copied (call-by-value) every time they are assigned to another variable or used as function parameters, but assigned as reference (call-by-reference) [Gutmans 04, S. 1]. As a result, many old and bad constructs remain in already existing applications. This leads to several issues.

That is why many companies struggle especially with security and performance problems. Manual refactoring of existing source code is time-consuming and a reason why a lot of companies abandon such improvements.

The open-source initiative of Facebook led to the tool lex-pass which is already capable of some very basic refactoring transformations [Lex-Pass 10]. It is using the abstract syntax tree[4] to transform PHP into corrected code according to predefined rules.

This bachelor thesis detects several anti-patterns and implements some basic transformations. As a result, these anti-patterns are automatically corrected with lex-pass. Then the implemented transformations are explained in detail and some further possible corrections are discussed. For demonstration purposes, some well known open-source projects are being tested and corrected with the written transformations.

As a conclusion the advantages and disadvantages of automatic corrections are explained and the usefulness and limitations of such automatic transformations are illustrated.

[4] AST, logical tree structure of a program

Inhaltsverzeichnis

1 Einleitung **1**

2 Motivation **3**

2.1 PHP - Eine dynamische Skriptsprache 3

2.1.1 Geschichte 3

2.1.2 PHP 5 4

2.1.3 Heutige Verwendung 5

2.2 Probleme 5

2.2.1 Sicherheit 5

2.2.2 Performanz 7

3 Anti-Patterns **9**

3.1 Was sind Anti-Patterns? 9

3.2 Warum Anti-Patterns? 9

3.3 Anti-Patterns aus Sicht eines Entwicklers 10

3.3.1 Spaghetti-Code 10

3.3.2 Golden Hammer 12

3.3.3 Cut-and-Paste Programmierung 13

3.4 PHP und Web spezifische Anti-Patterns 14

3.4.1 Übermässige Verwendung von Session 14

3.4.2 Cross-Site-Scripting Anfälligkeit 15

3.4.3 Unstrukturierte Datenbank Verwendung 15

4 Korrekturen **17**

4.1 lex-pass 17

4.2 Automatische Korrekturen 18

4.2.1 Explizite Methoden-Sichtbarkeit 18

4.2.2 Zuweisbare Variablen nach Rechts 20

4.2.3 Deprecation Korrektur 22

4.3 Weitere mögliche Transformationen 26

5 Praxistest an Open-Source Projekten **27**

5.1 PHProjekt 27

5.1.1 Explizite Sichtbarkeit 28

5.1.2 Zuweisbare Variablen nach rechts 28
5.1.3 Deprecation Korrektur 28
5.2 Wordpress 28
5.2.1 Explizite Sichtbarkeit 29
5.2.2 Zuweisbare Variablen nach rechts 29
5.2.3 Deprecation Korrektur 29
5.3 phpMyFAQ 29
5.3.1 Explizite Sichtbarkeit 30
5.3.2 Zuweisbare Variablen nach rechts 30
5.3.3 Deprecation Korrektur 30

6 Fazit **31**
6.1 Welche Gründe sprechen für Korrekturen 31
6.2 Welche Anti-Pattern sollten automatisiert korrigiert werden? 31
6.3 Weitere Entwicklung 31

Literaturverzeichnis **33**

Anhang A Statistik PHProjekt **36**

Anhang B Statistik Wordpress **37**

Anhang C Statistik phpMyFAQ **38**

1 Einleitung

Während der Erstellung dieser Arbeit feierte PHP am 8. Juni 2010 den 15. Geburtstag [Heise 10]. In dieses 15 Jahren hat PHP sich sehr stark entwickelt und verändert. Es ist also an der Zeit sich die Vorzüge und auch Probleme dieser Sprache genau anzusehen.

In der Literatur findet sich häufig wie man etwas richtig macht, wie man Fehler vermeidet und welche Lösungsansätze zum Ziel führen. Das ist natürlich sehr wichtig, hilft allerdings in der Regel nur Projekten, welche erst neu entwickelt werden, beziehungsweise gerade begonnen haben. Leider wird sehr häufig vergessen, dass es bereits eine Unmenge an vorhandener Software gibt, welche noch immer verwendet wird, allerdings bereits in die Jahre gekommen ist. Selbstverständlich heißt "alter Code" nicht automatisch "schlechter Code". Erinnert man sich beispielsweise an den Dijkstra-Algorithmus [Dijkstra 59, S. 269-271], welcher auf dem alten Router im Keller implementiert ist und noch heute seinen Dienst ohne Probleme tut. Problematisch wird Code erst, wenn sich eine Programmiersprache entwickelt und neue Versionen erscheinen, welche bestehende Fehler in der Sprache an sich korrigieren. Oftmals werden dadurch bestehende Konzepte, Konstrukte und Funktionen als *deprecated* bezeichnet, was soviel heißt wie "sollte vermieden werden".

Durch die enorme Veränderung, welche PHP in den letzten Jahren erfahren hat, sowie die Verwendung von PHP Software, die bereits vor einiger Zeit programmiert wurde, gibt es ein großes Potential für Refactoring und Überarbeitung von solcher Software.

In dieser Arbeit wird speziell auf die Probleme von bestehendem Code eingegangen. Alten Code zu überarbeiten kostet Zeit und Geld. Lohnt es sich also, bestehenden Code auf Fehler zu untersuchen und zu korrigieren? Was gibt es für Werkzeuge, die dem Entwickler einiges an Arbeit abnehmen können?

Im Kapitel 2 wird auf die Geschichte von PHP und die grundsätzlichen Probleme von bestehendem Code eingegangen. Zusätzlich werden Lösungsansätze erklärt, mit deren Hilfe eine bestehende Applikation zu einer performanteren und sicheren Anwendung überarbeitet werden kann.

Das Thema Anti-Patterns wird in Kapitel 3 zunächst im Allgemeinen erläutert, um es anschließend speziell für PHP zu erklären. Das Werkzeug lex-pass wird im Rahmen dieser Arbeit zudem dahingehend erweitert, dass vorhandene Anti-Patterns

in Applikationen erkannt und automatisiert korrigiert werden können. Dazu werden Transformationen für lex-pass erstellt, welche auf dem abstrakten Syntax Baum von PHP Änderungen vornehmen und in den Quellcode zurückführen können.

Um den Prozess der Überarbeitung bestehender Software zu vereinfachen und beschleunigen, wird in Kapitel 4 mit lex-pass ein Werkzeug vorgestellt, welches bereits einige einfache Refactorings an vorhandenem Quellcode vornehmen kann.

In Kapitel 5 werden einige bekannte und weit verbreitete Open-Source Projekte getestet. Mit Hilfe der erstellten Transformationen für lex-pass sollen in diesen Projekten automatisiert Anti-Patterns gefunden und korrigiert werden. Dies soll zu einer Verbesserung der Geschwindigkeit und Sicherheit dieser Anwendungen führen.

Abschließend wird in Kapitel 6 ein Fazit zum Thema automatisierter Korrektur gegeben und der sinnvolle Einsatz dieser erläutert.

2 Motivation

2.1 PHP - Eine dynamische Skriptsprache

PHP ist eine dynamische Skriptsprache, die speziell für den Einsatz auf Webservern entwickelt wurde. Jegliche Anfrage an eine PHP-Datei auf einem entsprechend konfigurierten Webserver wird durch die *PHP-Runtime* interpretiert. Dabei wird eine passende Antwort generiert und durch den Webserver an den Client gesendet. Dies stellt die Basis für eine dynamische Webseite mit PHP dar. PHP ist verfügbar für eine breite Anzahl an Webservern, sowie für unterschiedlichste Plattformen wie Linux, Mac OS X, Windows oder Unix.

2.1.1 Geschichte

PHP ist die Nachfolgesprache einer von Rasmus Lerdorf im Jahr 1995 programmierten Skriptsammlung. Ursprünglich war PHP nichts anderes als eine Sammlung von einfachen Perlskripten, die Lerdorf zur Erfassung von Zugriffen auf seinen Online-Lebenslauf geschrieben hatte. Seine Bezeichnung war anfangs *Personal Homepage Tools*. Mit dem Hinzufügen von mehr Funktionalität im Laufe der Zeit schrieb Lerdorf eine größere Implementierung in *C*, welche auch mit Datenbanken kommunizieren konnte. Somit war die Erstellung von einfachen Web-Anwendungen möglich. Lerdorf veröffentlichte den Quellcode mit dem Namen *PHP/FI - Personal Home Page Forms Interpreter* und somit konnte jedermann die Implementierungen verwenden, verbessern und Fehler beseitigen.

In PHP/FI waren bereits einige der grundlegenden Funktionen von PHP enthalten, welche wir heute noch kennen. Es gab - wie in *Perl* - Variablen mit einem “$”-Präfix, die automatische Interpretation von Form-Variablen und in HTML eingebettete Syntax. Die Syntax selbst war ähnlich der von Perl, wenn auch viel eingeschränkter, einfacher und ziemlich inkonsistent. Beispielsweise wird für ähnliche Funktionen eine andere Reihenfolge von Parametern verwendet [Tnx 10, K. 1].

PHP/FI 2.0 wurde offiziell erst im November 1997 freigegeben. Kurz danach wurden bereits die ersten Alpha-Versionen von PHP 3.0 freigegeben [PHP Group 10a, K. 1]. Zum Zeitpunkt der Freigabe von Version 3.0, wurde PHP/FI auch offiziell in das rekursive Akronym *PHP: Hypertext Preprocessor* umbenannt [PHP Group 10a, K. 2]. Viele der heute bekannten Strukturen in PHP stammen bereits aus der Zeit von PHP 3.0.

Andi Gutmans sowie Zeev Suraski, die beiden Autoren von PHP 3.0, begannen bereits kurz nach der Freigabe von PHP 3.0 die nachfolgende Version - PHP 4 - im Kern komplett neu zu schreiben. Der PHP Code wurde sehr stark modularisiert und im Kern die neu geschriebene *Zend Engine* eingesetzt. Seit PHP 4 wird Unterstützung für eine große Anzahl an Webserver angeboten. Sprachfeatures wie HTTP-Session-Verwaltung, Output-Buffering und sichere Verarbeitung von Benutzer-Eingaben wurden in dieser Version erstmals eingeführt.

2.1.2 PHP 5

Im Jahre 2004 wurde nach langer Entwicklungszeit PHP 5 veröffentlicht. Es beinhaltet ein völlig neues Modell zur objektorientierten Programmierung. Das bis dahin verwendete Konzept der Objektorientierung war rudimentär und brachte einige Nachteile zu Ansätzen der Objektorientierung in anderen Programmiersprachen. So wurde beispielsweise beim Erstellen eines Objekts, und danach bei der Zuweisung an eine andere Variable, eine Kopie des Objekts angelegt. Selbiger Ansatz wurde auch bei Funktionsaufrufen mit Objekten als Parametern eingesetzt (Call by Value). Dieses Verhalten brachte vor allem Einbußen im Bereich Performanz mit sich. Zusätzlich war es für viele Entwickler nicht klar, dass man auf einer Kopie eines Objekts arbeitete und Änderungen keinen Einfluss auf das ursprüngliche Objekt hatten. [Gutmans 04, K. 3]

In PHP 5 änderte sich das komplette objektorientierte Modell. Es wurden Schlüsselwörter zur Bestimmung der Sichtbarkeit von Klassenmethoden sowie -attributen eingeführt (Stichwort Datenkapselung). Die Benennung von Klassenkonstruktoren wurde vereinheitlicht, statt wie bisher den Klassennamen als Konstruktor zu verwenden, wurde die magische Methode `__construct()` eingeführt. Außerdem wurde die zusätzliche Methode `__destruct()` eingeführt, welche beim Zerstören eines Objekts aufgerufen wird. PHP definiert alle Funktionsname, welche mit zwei Unterstrichen beginnen, als magisch [PHP Group 10b].

Um weitere objektorientierte Strukturen zu unterstützen, wurden außerdem Interfaces, finale Klassen und Methoden, explizites kopieren von Objekt mit der magischen `__clone()` Methode, Klassenkonstanten, statische Attribute und Methoden, sowie abstrakte Klassen und Methoden eingeführt. PHP unterstützt allerdings keine Mehrfachvererbung, Klassen können jedoch von mehreren Interfaces erben [Gutmans 04, K. 5].

2.1.3 Heutige Verwendung

Viele Unternehmen setzen heute vor allem in Webapplikationen auf PHP. Im Jahre 2007 war bereits auf über 20 Millionen Domains PHP verfügbar [PHP Group 07]. Viele große und erfolgreiche Internet-Plattformen, wie zum Beispiel Facebook [Facebook 07], Flickr [Flickr 10] oder Wikipedia [Wikipedia 10], setzen auf PHP.

2.2 Probleme

Trotz Einführung eines neuen objektorientierten Modells, einer Überarbeitung von vielen alten Konstrukten sowie ein systematisches Aufräumen im Kern von PHP 5, verwenden noch heute viele Unternehmen alten Code, welcher mit Problemen zu kämpfen hat. Vor allem im Bereich Sicherheit sowie Performanz sollte aus heutiger Sicht nachgebessert werden. Refactoring von altem Code kann allerdings einen enormen Zeitaufwand darstellen. Hier bietet sich die automatisierte Korrektur von einigen bekannten Anti-Patterns an. Facebook hat im Rahmen ihrer Open-Source-Initiative das in Haskell, einer funktionalen Programmiersprache, geschriebene Werkzeug *lex-pass* veröffentlicht. Einige entsprechende Korrekturmöglichkeiten an PHP-Code mit *lex-pass* werden in Kapitel 4 erörtert. Zuerst folgen allerdings in den nachfolgenden beiden Abschnitten einige der Hauptprobleme im Bereich Sicherheit sowie Performanz in PHP-Code.

2.2.1 Sicherheit

Viele sicherheitsrelevante Probleme im Zusammenhang mit PHP bestehen dadurch, dass Applikationen mit Benutzereingaben arbeiten. Dies ist kein besonderes Problem von PHP, sondern ein Grundsätzliches für alle Anwendungen und Webapplikationen im Besonderen. Werden solche Benutzereingaben nicht sorgfältig geprüft, sondern direkt verwendet, können dadurch einige Probleme auftreten. Beispielsweise können beim Inkludieren von zusätzlichen Daten (definiert durch Benutzereingaben), wichtige Daten ausgelesen werden, welche für den Anwender nicht sichtbare sein sollen (Stichwort Information Disclosure). Das Ausgeben von Benutzereingaben ohne das Verwenden von den PHP-Filter/Escaping-Funktionen wie `htmlentities()` oder `htmlspecialchars()` kann zu sogenannten Cross-Site-Scripting[5] Problemen führen. Ein weiteres typisches Problem betrifft das Einfügen von Benutzereingaben

[5] Eingabe von z.B. JavaScript-Code kann es zur Ausführung von Schadcode auf dem Client kommen

in eine Datenbankabfrage ohne vorherige Prüfung, beziehungsweise ohne Verwendung von Prepared-SQL-Stamements. Das heißt, das Statement enthält keine Parameterwerte sondern Variablen, welche erst nachträglich festgelegt und von Datenbank-System validiert werden. Beispiele für ungeprüfte Verwendung von Benutzereingaben sind im Codeblock 1 aufgeführt.

Codeblock 1: Ungefiltere Benutzereingaben

```
1 <?php
2 // ungefiltertes Einbinden externer Dateien
3 include dirname(__FILE__) . '/inc/' . $_GET['include'] . '.php';
4
5 // ungefilterte Ausgabe von Benutzereingaben
6 echo 'hello ' . $_GET['name'];
7
8 // ungefiltertes verwenden von Benutzereingabe in Datenbank
      Abfragen
9 $result = mysql_query('SELECT a FROM b WHERE c = ' . $_POST['c'])
      ;
```

Die in Codeblock 1 aufgezeigten ungefilterten Beispiele sind in Codeblock 2 mit entsprechenden Filtermöglichkeiten korrigiert. Hierzu gibt es natürlich verschiedene Ansätze, allerdings wurde hier lediglich eine Möglichkeit verwendet. In dem ersten Beispiel, in Codeblock 1, könnte zum Beispiel eine Usereingabe von "`../../../etc/passwd\0`" zur ungewollten Ausgabe der globalen Passwortdatei auf einem Linux/Unix System führen. Diese Informationen könnten Rückschlüsse auf das entsprechende System und deren Benutzer zulassen. Manche verwendete Benutzer-Namen deuten zum Beispiel auf installierte und verwendete (Server)-Services.

Codeblock 2: Gefiltere Benutzereingaben

```
1 <?php
2 // gefiltertes Einbinden externer Dateien
3 $includes = array('news', 'products', 'pictures');
4 $include = 'index';
5 if (in_array($_GET['include'], $includes)) {
6   $include = $_GET['include'];
7 }
8 include dirname(__FILE__) . '/inc/' . $include . '.php';
9
```

```
10 // gefilterte Ausgabe von Benutzereingaben
11 echo 'hello ' . htmlentities($_GET['name']);
12
13 // gefiltertes verwenden von Benutzereingabe in Datenbank
       Abfragen
14 $stmt = $dbh->prepare('SELECT a FROM b WHERE c = :c');
15 $stmt->bindParam(':c', $_POST['c'], PDO::PARAM_STR);
16 $result = $stmt->execute();
```

2.2.2 Performanz

Wie im nachfolgenden Codeblock 3 ersichtlich, gibt es auch Probleme im Bereich Geschwindigkeit. Diese sind, wie in dem ersten Beispiel aufgeführt, nicht immer ein Problem von PHP an sich, sondern von schlechtem Programmierstil. Das mehrmalige Ausführen einer rechenintensiven Operation innerhalb einer Schleife mit den selben Eingabeparametern - ohne Änderung dieser Parameter innerhalb der Schleife - ist unnötig und kann sehr einfach verbessert werden. Auch das Verwenden eines *Key-Value-Pairs* für eine Foreach-Schleife, in welcher allerdings nur auf Value zugegriffen wird, ist unnötig und kann auf Value reduziert werden.

Codeblock 3: Schlechte Performanz

```
1 <?php
2 // Mehrmaliges "count()" der selben Daten
3 $data = array('one' => 1, 'two' => 2, 'three' => 3);
4 for ($i = 0; $i < 10; $i++) {
5   echo 'Anzahl: ' . count($data);
6 }
7
8 // Nicht verwendeter "$key" in "foreach"-Schleife
9 foreach ($data as $key => $value) {
10   echo $value;
11 }
```

Eine Verbesserung des ersten Beispiels von Codeblock 3 ist es, die Operation `count()` vor der Schleife einmalig auszuführen und dann das Ergebnis in der Schleife mehrmals zu verwenden. Ein einfacher Test dieses Beispiels beim Durchlauf dieser Schleife führt zu einer ca. 79%igen Geschwindigkeitssteigerung der entsprechenden Schleife. Im Durchschnitt wurden 2.14 Sekunden gegenüber 0.44 Sekunden bei 10 Millionen Durchläufen benötigt.

Codeblock 4: Bessere Performanz

```php
<?php
// Einmaliges "count()"
$data = array('one' => 1, 'two' => 2, 'three' => 3);
$count = count($data);
for ($i = 0; $i < 10; $i++) {
  echo 'Anzahl: ' . $count;
}

// "foreach" ohne "$key"
foreach ($data as $value) {
  echo $value;
}
```

3 Anti-Patterns

3.1 Was sind Anti-Patterns?

Ein *Anti-Pattern* oder auch Anti-Muster ist eine "literarische Form, welche für ein Problem eine allgemein auftretende Lösung beschreibt, die entschieden negative Konsequenzen generiert" [Brown 98, S. 7]. Anti-Patterns an sich können nicht grundsätzlich als schlechte *Design-Patterns* [Gamma 94] angesehen werden, vielmehr werden darunter Fehler sowie gute Lösungsansätze im falschen Kontext durch unerfahrene Programmierer oder Manager zusammengefasst. Durch entsprechende Dokumentation wird beschrieben, wie ein Anti-Pattern erkannt wird, welche Konsequenzen dadurch entstehen sowie welcher Ansatz zum Refactoring der Lösung verwendet werden soll, um das Anti-Pattern durch eine bessere Lösung zu ersetzen. [Brown 98, S. 7]

Anti-Patterns können laut Brown von drei Perspektiven aus gesehen werden: vom Entwickler, vom Software-Architekt und vom Manager. Aus Sicht des Entwicklers beschreiben Anti-Patterns technische Probleme und Lösungen, welche dem Programmierer begegnen. Software-Architekten sehen Anti-Patterns als grundsätzlichere und allgemeinere Probleme, welche vor allem im Bereich Software-Architektur und System-Design zu finden sind. Manager sehen Anti-Patterns als übliche Probleme in Software-Prozessen und in Entwicklungs-Organisationen. [Brown 98, S. 13] In dieser Arbeit wird speziell auf die Sicht eines Software-Entwicklers eingegangen.

3.2 Warum Anti-Patterns?

Es handelt sich bei Anti-Patterns somit also um *schlechte Lösungen* für bestimmte Probleme. Anti-Patterns allerdings beinhalten immer die entsprechende Refactoring Lösung. Mit *Software Refactoring* wird im Allgemeinen die Modifizierung von Software verstanden, welche zu einer besseren Struktur führt [Brown 98, S. 68]. Durch Korrekturen am Programmcode wird bessere Wartbarkeit und Erweiterbarkeit erreicht. PHP wird hauptsächlich im Bereich der Webentwicklung, mit immer kürzeren Release-Zyklen (Stichwort Kontinuierliche Integration [Fowler 06]) eingesetzt, weshalb eine regelmäßige Wartung sowie Erweiterung eine große Rolle spielen. Schlechte und fehlerhafte Code-Teile in Software zu warten erweist sich, wie in den nachfolgenden Abschnitten erläutert, als schwieriger, als diese Teile zu besserem Code umzugestalten. Deswegen spielt das Eliminieren von Anti-Patterns in Applikationen,

welche ständig erweitert werden, eine große Rolle.

Der Prozess der kontinuierlichen Integration soll laut Fowler [Fowler 06] hauptsächlich folgende Prinzipien beachten:

- Entwickler sollen relativ oft und früh Änderungen am Source-Code in das Versionsverwaltungssystem einpflegen. Dadurch werden große Änderungen in kleinen Schritten vorgenommen, was auch zu einer besseren Nachvollziehbarkeit führt. Zumindest einmal, besser mehrmals pro Tag sollte eine Entwickler seine Änderungen einpflegen.

- Bei jeder Änderung im Versionskontrollsystem, sollte auf einem speziell dafür vorgesehenen System, das gesamte Projekt neu kompiliert (bei PHP nicht notwendig) und anschließend komplett getestet werden. Dazu werden automatisierte Funktionstests sowie Unittests eingesetzte [PHPUnit 10].

3.3 Anti-Patterns aus Sicht eines Entwicklers

In den folgenden Abschnitten 3.3.1 bis 3.3.3 werden einige Anti-Patterns aus Sicht eines Entwicklers erläutert. Dazu wird jeweils eine Erklärung zu dem entsprechenden Anti-Pattern gegeben und gleichzeitig eine mögliche Refactoring-Lösung angeboten.

3.3.1 Spaghetti-Code

Beschreibung Das Anti-Pattern Spaghetti-Code zeigt sich in Code mit keiner, beziehungsweise mit wenig Struktur. Es tritt vor allem dann auf, wenn Funktionen eine große Komplexität erreichen. Dies ist dann der Fall, wenn Funktionen relativ stark erweitert werden und zu viel *Logik* enthalten. Es führt oftmals dazu, dass der Code unverständlich wird und sogar vom ursprünglichen Entwickler nicht mehr gut gelesen werden kann. Erkennbar ist dieses Anti-Pattern zum Beispiel an fehlender Vererbungshierarchien oder keinerlei Wiederverwendung von Code. Sollte solcher Code erweitert werden, ist es oftmals “günstiger”, eine vollständig neue Lösung zu entwickeln. Typischerweise kommt es zu Spaghetti-Code, wenn Entwickler wenig Erfahrung mit dem Umgang von objektorientierten Konzepten haben, kein Mentoring bzw. schlechte Code-Reviews erhalten oder kein Design-Prozess vor der Implementierung stattgefunden hat [Brown 98, S. 119-120].

Hintergrund Spaghetti-Code ist eines der klassischen und weit verbreitetsten Anti-Patterns überhaupt. Üblicherweise kommt es vor allem in nicht objektorientierten Sprachen zum Vorschein, allerdings kann es auch in objektorientierten Sprachen vorkommen, speziell bei Entwicklern, welche wenig Erfahrung mit dem Umgang von objektorientierten Konzepten haben. [Brown 98, S. 119]

Refactoring Die beste Methode um Spaghetti-Code loszuwerden, ist "Prävention: Zuerst überlegen, dann entsprechende Aktionen planen und diese dann implementieren" [Brown 98, S. 122]. Sollte bereits Spaghetti-Code existieren, empfiehlt es sich bei jeder Erweiterung nicht im selben Stil weiter zu programmieren sondern sich Zeit zu nehmen, Refactoring am bestehenden Code vorzunehmen. Dazu sollten Klassen, Funktionen und Daten einem einheitlichen Standard nach benannt werden, welcher in Code Conventions festgelegt wird. Für PHP empfiehlt sich die Verwendung der PEAR Coding Standards [Pear 10]. Außerdem besteht für PHP mit dem, in Kapitel 4 vorgestellten Werkzeug eine Möglichkeit, Funktionen automatisch global umzubenennen. Zusätzlich können nicht verwendete private Methoden sowie Variablen aus Klassen entfernt werden. In Methoden mit ähnlicher Funktionalität, kann die Reihenfolge und Bezeichnung der Funktions-Parameter vereinheitlicht werden. Manche Code-Teile können in Funktionen ausgelagert werden, um Wiederverwendbarkeit zu erreichen (vgl. Anti-Pattern Cut-and-Paste Programmierung in Abschnitt 3.3.3). [Brown 98, S. 122]

Wenn man vor dem Start eines neuen Projekts bereits verhindern möchte, dass es zu Spaghetti-Code kommt (Prävention ist eine Option), sollte man sich laut [Brown 98, S. 123] an folgende Regeln halten:

1. **Domänenmodell** Erstellung eines konsistenten und korrekten Domänenmodells mit Hilfe von Methoden aus der objektorientierten Analyse und Design.

2. **Designmodell** Erstellung eines Designmodells, welches auf dem Domänenmodell basiert, allerdings bereits Gemeinsamkeiten von Domänenobjekten extrahiert und abstrahiert, um notwendige Objekte und Beziehungen in dem System zu finden.

3. **Decomposition** Objekte aus dem Designmodell müssen soweit zerlegt werden, dass jeder Entwickler versteht, für welche Aufgaben diese Objekte zu stehen haben.

4. **Planung** Start der Implementierung anhand eines Plans, welcher auf dem Designmodell basiert. Das Design muss nicht vollständig sein, viel wichtiger ist es, dass die Implementierung sich an einen vorgegebenen Plan hält.

3.3.2 Golden Hammer

Beschreibung Wenn ein Unternehmen beziehungsweise Entwicklungsteam maßgebliche Kompetenz in einer bestimmten Software-Lösung aufweist, wird oftmals versucht, jedes Projekt mit der selben Software-Lösung zu erstellen. Häufig tritt dies im Zusammenhang mit Datenbank-Systemen auf, gerade wenn man viel Zeit investiert hat, um sich einer bestimmten Datenbank-Lösung anzupassen. Dies führt dazu, dass man zwar im Glauben, das beste Werkzeug für eine bestimmte Aufgabe zu verwenden, eigentlich das am vertrauteste Werkzeug verwendet. Dies ist in der Regel nicht die richtige Lösung für das spezifische Problem [Brown 98, S. 111-113].

Hintergrund Eines der meist vorkommenden Anti-Patterns in Software ist das Golden-Hammer Anti-Pattern. Oftmals empfiehlt ein Drittanbieter (meist Datenbank-System Anbieter) die Verwendung seines Produkts, welches als Lösung für alle im Unternehmen anfallenden Probleme verwendet werden kann [Brown 98, S. 111].

Refactoring Vorwiegend benötigt dieses Anti-Pattern einen philosophischen Lösungsansatz, aber auch eine Änderung im Entwicklungsprozess. Ein Unternehmen sollte sich darauf festlegen offen für neue Technologien zu sein. Dies kann zum Beispiel durch den ständigen Austausch der Entwickler untereinander erreicht werden. Dazu gibt es laut [Brown 98, S. 113-114] mehrere kombinierbare Möglichkeiten, welche sowohl auf unternehmensinterne, als auch auf die globalen Technologie Aspekte ausgerichtet sein können:

1. **Diskussionsgruppen** Entwickler können sich regelmäßig zu Diskussionsgruppen treffen, um neue Technologien zu besprechen und auszuprobieren. Oftmals gibt es zu bestimmten Technologien auch *Usergroups*, welche sich in regelmäßigen Abständen treffen. Für PHP gibt es beispielsweise in Österreich die *PHP Usergroup Austria*[6] oder in Deutschland die *PHP Usergroup Germany*[7].

[6] http://phpug.at/
[7] http://www.phpug.de/

2. **Buchbesprechungsgruppen** Eine sehr effiziente Form bietet eine Buchbesprechungsgruppe, welche sich regelmäßig zur Diskussion über neu erschienene Literatur zu einem bestimmten Fachgebiet trifft.

3. **Entwicklungskonferenzen** Eine wichtige Form der Weiterbildung für Entwickler bieten Konferenzen. Diese erlauben den Austausch zwischen unterschiedlichen Drittanbietern sowie anderen Benutzern. Zusätzlich kann meist aus den unterschiedlichen Präsentationen eine Richtung festgestellt werden, in welche die aktuelle Entwicklung führt.

4. **Open-Source** Die aktive Teilnahme an Open-Source Projekten sowie der Förderung von Open-Systems führt längerfristig zu gutem Verständnis des Marktes. Dadurch werden die Werkzeuge, welche man aktiv mitentwickelt, verbessert.

All diese Möglichkeiten können sehr autonom vom Management durchgeführt werden, in dem Entwickler sich organisieren und von dem Unternehmen entsprechende Förderung bekommen. Eine weitere Möglichkeit, dieses Anti-Pattern zu vermeiden, ist es Entwickler aus verschiedenen Regionen und mit unterschiedlichem Hintergrund einzustellen [Brown 98, S. 114-115].

Golden Hammer wurde trotz des philosophischen Aspekts dieses Anti-Patterns hier vorgestellt, da es sehr häufig vorkommt und im Bereich des Software-Refactorings eine wichtige Rolle spielt. Dieses Anti-Pattern wird im Abschnitt 4.2 Automatische Korrekturen somit auch nicht weiter besprochen, da es keine Möglichkeiten gibt, dies automatisiert zu korrigieren.

3.3.3 Cut-and-Paste Programmierung

Beschreibung Cut-and-Paste Programmierung wird erkannt, durch sich wiederholende Code-Teile in einem Software Projekt. Oftmals versuchen unerfahrene Entwickler den Code von bereits erfahrenen Kollegen zu verwenden und gegebenenfalls zu erweitern, um ähnliche Probleme zu lösen. Dies führt unweigerlich zu einer unnötig großen Code-Base mit vielen Programmierzeilen mit gleicher Funktionalität. Viel gravierender ist die Tatsache, dass dadurch vor allem bestehende Fehler vielerorts in der Applikation vorkommen. Wird ein solcher Fehler gefunden und behoben, ist es meist unmöglich diesen an allen Stellen zu verbessern, wodurch die Wartungskosten enorm steigen [Brown 98, S. 134].

Hintergrund Dieses Anti-Pattern kommt sehr häufig vor. Es stellt die einfachste, aber auch schlechteste Form von *wiederverwendbarem Code* dar. Oftmals weil Unternehmen es nicht honorieren, wiederverwendbaren Code zu produzieren, da es kurzfristig als profitabler erscheint, eine schnelle Lösung anzubieten, tritt dieses Anti-Pattern in Erscheinung [Brown 98, S. 133-134].

Refactoring White-Box[8] Programmierung ist häufig die Ursache für diese Anti-Pattern. Stattdessen sollte besser auf *Black-Box*[9] Programmierung gesetzt werden. Zum Erreichen dieses Zieles benötigt es laut [Brown 98, S. 136] drei Schritte.

1. **Code Mining** Systematisches Durchsuchen von Programm-Code und Identifizieren von Software-Stellen mit selbem Code.
2. **Refactoring** Entwicklung einer Standard Version des entsprechenden Codes als Modul oder Funktion und Aufruf an allen betreffenden Stellen.
3. **Konfigurationsmanagement** Erstellung von Regeln zur Vermeidung von Cut-and-Paste Code im weiteren Entwicklungs-Prozess. Diese Regeln sollte zusätzlich zu entsprechenden Entwickler-Schulungen auch Mentoring, Code-Inspektionen sowie Revision beinhalten.

Für viele Anwendungen ist ein Refactoring meistens nur sinnvoll, wenn die entsprechenden Stellen auch in anderen oder neuen Projekten verwendet werden sollen. Eine bestehende Software zu überarbeiten um Cut-and-Paste Code loszuwerden, kann sehr zeitintensiv und aufwändig sein [Brown 98, S. 136].

3.4 PHP und Web spezifische Anti-Patterns

Für diese Arbeit sind vor allem PHP spezifische Anti-Patterns interessant. Viele davon kommen allerdings auch in Webapplikationen im Allgemeinen sehr häufig vor. Einige solcher Anti-Patterns sind in den nachfolgenden Abschnitten erläutert.

3.4.1 Übermässige Verwendung von Session

Gerade Software Entwickler mit einem starken Hintergrund im Bereich der Desktop Anwendungsentwicklung, sind es gewohnt mit dem Zustand der Applikation zu arbeiten. Eine Möglichkeit sich den Status zu “merken” bietet das Session Konzept

[8] Schnittstellen-Code liegt offen vor

[9] Schnittstellen-Code liegt nur kompiliert vor

[PHP Group 10f]. Viele unerfahrene Entwickler jedoch speichern viele Daten in der Session und somit viele Status-Informationen.

Als Refactoring empfiehlt sich hier die in Webapplikationen übliche zustandslose Programmierung zu bevorzugen (vgl. Representational State Transfer, REST [Fielding 00, S. 76]). Hierbei sollte darauf geachtet werden, dass entsprechende Funktionalität ohne einen vorgegebenen Status der Applikation arbeiten kann. Alle benötigten Informationen für die Bearbeitung einer Anfrage an den Webserver sollen jeweils von neuem (zum Beispiel aus der Datenbank) ausgelesen werden. Lediglich ein kleiner Teil an Informationen (zum Beispiel die Login-Informationen) sollte mit dem Session-Mechanismus abgedeckt werden.

3.4.2 Cross-Site-Scripting Anfälligkeit

Webapplikationen arbeiten stark mit Benutzereingaben. Werden diese Eingaben ohne entsprechenden Filter-Funktionen wieder ausgegeben, kann es dazu kommen, dass ein Angreifer Client-Seitige Skripte in Webseiten einschleusen kann, welche andere Benutzer ansehen [Grossman 06].

Mit PHP bieten sich hierfür mehre Möglichkeiten zum Refactoring dieses Anti-Patterns an. Hauptsächlich sollte auf die Filter Erweiterung (seit PHP 5.2 standardmäßig aktiviert) zurückgegriffen werden, um Benutzereingaben zu filtern und validieren [PHP Group 10g].

3.4.3 Unstrukturierte Datenbank Verwendung

Um korrektes Arbeiten mit einer Datenbank zu erreichen, sollte man auf SQL Abfragen und Kommandos innerhalb der Kontroll-Schicht verzichten. Vor allem sollte man keine SQL Abfragen inmitten des Code-Teils verwenden, der sich um die Präsentation der Webseite kümmert. Dies führt unweigerlich zu schlecht lesbarem Code (vgl. Anti-Pattern Spaghetti-Code in Abschnitt 3.3.1).

Um dieses häufig gesehene Anti-Pattern erst gar nicht aufkommen zu lassen, empfiehlt es sich für den Datenbank Zugriff ein entsprechendes Datenbank-Framework beziehungsweise ein Object-Relational-Mapper (ORM) zu verwenden. Für PHP gibt es mittlerweile zwei bekannte ORM-Projekte, Propel [Propel 10] und Doctrine [Doctrine 10]. Für den Einsatz eines ORM-Systems sprechen zusätzlich laut Karwin [Karwin 09] folgende Gründe:

- Verkürzung der Entwicklungszeit durch die Vermeidung von Wiederholungen, wie zum Beispiel dem Abbilden eines Datenbank-Resultats auf ein Objekt und umgekehrt.
- Datenzugriff wird abstrakter und portabler. Hersteller spezifischer SQL-Code wird vermieden und das verwendete Datenbank-System kann einfacher ausgetauscht werden.
- Codebausteine, welche bereits die einfachen CRUD-Operationen[10] unterstützen, können verwendet werden.

[10] Create, Read, Update, Delete

4 Korrekturen

Für Java gibt es bereits eine Möglichkeit um Anti-Patterns automatisiert korrigieren zu können. Das Open-Source Projekt *PMD* scannt Java Quelltext und findet beispielsweise mögliche Fehler wie leere `try/catch/finaly/switch` Ausdrücke. PMD ist mittlerweile in alle gängigen Java Entwicklungsumgebungen (IDE) und Entwicklungswerkzeuge (z.B. Eclipse, Netbeans, JDeveloper, BlueJ, InteliJ IDEA, Maven, Ant, etc.) integriert und führt auf Wunsch automatische Transformationen direkt beim Schreiben von Programmcode aus [PMD 10].

Für PHP existierte bisher keine Möglichkeit automatisiert Quellcode zu korrigieren. Facebook hat ein Werkzeug veröffentlicht - *lex-pass* [Lex-Pass 10] - mit welchem sich bereits einige kleine Refactorings durchführen lassen.

4.1 lex-pass

Das Werkzeug lex-pass wurde im Rahmen der Open-Source Initiative von Facebook geschrieben und veröffentlicht [Lex-Pass 10]. Dieses Werkzeug erlaubt es, automatisierte Änderungen an einer bestehenden PHP Code-Base vorzunehmen.

Durch die Verwendung der funktionalen Programmiersprache Haskell für lex-pass, lassen sich die guten Syntax-Analyse Funktionen dieser Sprache verwenden. Mit diesen Möglichkeiten lassen sich einfach Transformationen an dem abstrakten Syntax-Baum (AST) von Programmiercode vornehmen.

Ursprünglich war lex-pass lediglich dazu gedacht, einige einfache Refactorings an einer Code-Base vorzunehmen. Beispielsweise konnten bereits Funktionen im globalen Geltungsbereich umbenannt werden, oder Funktionsparameter von Funktionen global in beliebiger Reihenfolge entfernt werden.

Lex-pass wird auf der Konsole, mit Angabe der gewünschten Transformation sowie eventuellen Parametern, ausgeführt. Es werden dann die spezifizierten PHP-Dateien (bzw. die komplette Code-Base) analysiert und in den abstrakten Syntax-Baum transformiert. In diesem Schritt speichert lex-pass zu jeder PHP-Datei in einem erstellten Cache-Verzeichnis den AST. Dadurch wird jedes zukünftige Transformieren (sofern sich keine Änderungen ergeben haben) beschleunigt. Anschließend wird der AST analysiert und die entsprechende Transformation durchgeführt sowie in die original PHP-Datei als PHP-Programmcode wieder zurückgeführt und gespeichert. In Codeblock 15 wird eine Transformation, welche in Abschnitt 4.2.3 vorgestellt

wird, zur Veranschaulichung der Verwendung dargestellt.

Codeblock 5: Ausführung einer Transformation

```
1 // Transformation
2 fdomig@jupiter:~$ lex-pass kill-split
3 Checking (1/1) deprecation.php
4 - Parsing
5 - Saving
```

Im Rahmen dieser Bachelorarbeit wurde lex-pass in Zusammenarbeit mit dem Autor Daniel Corson um einige Transformationen für Anti-Patterns erweitert. Diese Transformationen sind in den nachfolgenden Abschnitten detailliert erläutert.

4.2 Automatische Korrekturen

Die in Kapitel 3 vorgestellten Anti-Patterns beschreiben vorwiegend grundsätzliche und komplexere Anti-Patterns in Programm-Code. Diese lassen sich nicht einfach und vollständig automatisiert korrigieren. Allerdings können durch kleinere Korrekturen Teile davon korrigiert werden. Vor allem das in Abschnitt 3.3.1 vorgestellte Anti-Pattern Spaghetti-Code kann zum Teil automatisiert korrigiert werden. Dennoch sind einige Refactoring-Arbeiten nachträglich und händisch durchzuführen.

4.2.1 Explizite Methoden-Sichtbarkeit

Seit PHP 5 gibt es für Klassenmethoden sowie -attribute die neu eingeführten Schlüsselwörter `public, protected, private` eine Möglichkeit zur Bestimmung der Sichtbarkeit. Um Rückwärtskompatibilität zu altem PHP 4 Code zu gewährleisten, werden alle Funktionen sowie Attribute von Klassen ohne ein explizites Sichtbarkeitsschlüsselwort, von der PHP-Runtime als öffentlich (`public`) behandelt. Dies ist eigentlich kein klassisches Anti-Pattern, bietet durch die Einfachheit der Transformation allerdings einen guten Einstieg in die Korrekturmöglichkeiten mit lex-pass.

Mit der Transformation in Codeblock 6, werden alle Methoden, welche keine Sichtbarkeitseinschränkung definiert haben, explizit als öffentliche Methode deklariert.

Codeblock 6: Explizite Sichtbarkeit Transformation

```
1 -- Prüfung ob eine Zeichenkette ein Zugriffsschlüsselwort ist
2 sAccessKeyword :: String -> Bool
```

```
3 isAccessKeyword x = x `elem` ["public", "private", "protected"]
4
5 -- Transformation von allen "Class Function Definitions" (
      Methoden)
6 -- zu öffentlichen Methoden sollte kein Zugriffsschlüsselwort
      vorhanden sein
7 makePublicExplicit :: Ast -> Transformed Ast
8 makePublicExplicit = modAll $ \ cStmt -> case cStmt of
9   CStmtFuncDef pre f -> if any (isAccessKeyword . map toLower .
        fst) pre
10     then transfNothing
11     else pure $ CStmtFuncDef (("public", [WS " "]):pre) f
12   _ -> transfNothing
```

Nach der Transformation des PHP Codes aus Codeblock 7, wird daraus der Code in Codeblock 8 erstellt. Dadurch wird es vor allem für unerfahrene PHP Entwickler einfacher, Code zu lesen, speziell auch, wenn man es aus anderen Programmiersprachen gewöhnt ist, mit einer expliziten Sichtbarkeit zu arbeiten. In vielen anderen Programmiersprachen, wie beispielsweise Java, C# oder C++ bedeutet eine fehlende Sichtbarkeitsdeklaration automatisch eine private (respektive Package/Namespace) und nicht, wie bei PHP, eine öffentliche Sichtbarkeit.

Codeblock 7: Nicht explizite Sichtbarkeit

```
1 <?php
2 class Calculator {
3   // Nicht explizit als öffentlich deklarierte Methode
4   function add($a, $b) {
5     return $a + $b;
6   }
7   // Nicht explizit als öffentlich deklarierte, statische Methode
8   static function sub($a, $b) {
9     return $a - $b;
10  }
11 }
```

Codeblock 8: Explizite Sichtbarkeit

```
1 <?php
2 class Calculator {
3     // Explizit als öffentlich deklarierte Methode
4     public function add($a, $b) {
```

```
5           return $a + $b;
6       }
7       // Explizit als öffentlich deklarierte, statische Methode
8       public static function sub($a, $b) {
9           return $a - $b;
10      }
11  }
```

4.2.2 Zuweisbare Variablen nach Rechts

Um Flüchtigkeitsfehler zu vermeiden, ist es sinnvoll, in *if*-Ausdrücken die Reihenfolge so zu setzen, dass zuweisbare Variablen auf der rechten Seite des Ausdrucks stehen. Statt *if ($a == true)* sollte *if (true == $a)* geschrieben werden. Wird nämlich aus Versehen das zweite "=" vergessen, wird der linken Seite, die rechte Seite zugewiesen, wenn der linke Bezeichner eine (zuweisbare) Variable ist. In der Regel ist dies nicht das gewünschte Resultat. Ändert man die Reihenfolge der Bezeichner (sollte der rechte Bezeichner keine Variable sein), würde bereits beim Interpretieren der PHP Datei ein Fehler ausgegeben, da es sich um eine ungültige Zuweisung handelt. Dieser Fehler würde sofort erkannt werden und kann vom Entwickler schnell korrigiert werden.

Codeblock 9 zeigt den entsprechenden Transformationscode, welcher die gewünschte Korrektur durchführen kann. Diese Transformation funktioniert ebenfalls für Größenvergleiche, bei denen der Vergleichsoperator entsprechend angepasst wird.

Achtung: Diese Transformation beinhaltet die Einschränkung, dass bereits vorhandene Zuweisungen in If-Ausdrücken (gewünscht oder fehlerhafte) nicht transformiert werden!

Codeblock 9: Zuweisbare Variablen nach Rechts Transformation

```
1 -- Prüfung auf Zuweisbarkeit der rechten Variable in einem
      Ausdruck
2 exprIsLRVal :: Expr -> Bool
3 exprIsLRVal (ExprRVal (RValLRVal _)) = True
4 exprIsLRVal _ = False
5
6 -- Prüfung auf entsprechende binäre Operationen eines Ausdrucks
7 -- mit gegebenenfalls Änderung der Reihenfolge sowie Änderung der
8 -- Größen-Vergleiche von "< "und "<=" zu ">" und ">="
9 exprLRValToRight :: Expr -> Transformed Expr
```

```
exprLRValToRight (ExprBinOp op e1 w e2)
  | op `elem` [BEQ, BNE, BID, BNI] = swapIfGood op
  | op == BLT = swapIfGood BGT
  | op == BGT = swapIfGood BLT
  | op == BLE = swapIfGood BGE
  | op == BGE = swapIfGood BLE
  | otherwise = transfNothing
  where
  swapIfGood op' = if exprIsLRVal e1 && not (exprIsLRVal e2)
    then pure $ ExprBinOp op' e2 w e1
    else transfNothing
exprLRValToRight _ = transfNothing

-- Transformation von allen "If-Block-Expressions" mit der
-- "exprLRValToRight" Funktion
assignablesGoRight :: Ast -> Transformed Ast
assignablesGoRight = modAll . modIfBlockExpr $ modWSCap2
    exprLRValToRight
```

Ein Beispiel für diese Transformation wird in Codeblock 10 (Ausgangs-Code) und Codeblock 11 (transformierter Code) dargestellt.

Codeblock 10: Zuweisbare Variablen auf der linken Seite

```
<?php
// If-Ausdruck mit zuweisbarer Variable auf der linken Seite
if ($a == true) {
  echo 'true';
}

// Größen Vergleich mit zuweisbarer Variable auf der linken Seite
if ($b <= 4711) {
  echo 'small';
}
```

Codeblock 11: Zuweisbare Variablen auf der rechten Seite

```
<?php
// If-Ausdruck mit zuweisbarer Variable auf der linken Seite
if (true == $a) {
  echo 'true';
}

```

```
// Größen Vergleich mit zuweisbarer Variable auf der linken Seite
if (4711 >= $b) {
  echo 'small';
}
```

4.2.3 Deprecation Korrektur

Der Austausch von als *deprecated* bezeichneten Funktionen mit entsprechend neuen Funktionen ist eine Transformation, welche vor allem alten Code mit neuen PHP Versionen lauffähig machen würde, ohne entsprechende *Deprecation*-Warnungen zu erzeugen. Als Beispiel wurde hier die veraltete und als deprecated bezeichnete Funktion `split()` mit der neuen Funktion `preg_split()` aus der PHP-PCRE Erweiterung [PHP PCRE 10] ausgetauscht.

In Codeblock 12 ist die entsprechende Haskell Transformation für die Umwandlung von `split()` zu `preg_split()` dargestellt.

Codeblock 12: Deprecation Transformation

```
-- Transformation von allen "split()" Aufrufen zu "preg_split()"
killSplit :: Ast -> Transformed Ast
killSplit = modAll $ \ a -> case a of
  ROnlyValFunc _c@(Right (Const [] "split")) w (Right (arg0:args)
      ) ->
    case arg0 of
      WSCap w1 (Left (ExprStrLit (StrLit s))) w2 ->
        pure . ROnlyValFunc c' w $ Right (arg0':args)
        where
        c' = Right (Const [] "preg_split")
        arg0' = WSCap w1 (strToArg s') w2
        s' = onTail (onInit $ delimify '/' '\\') s
      _ -> transfNothing
  _ -> transfNothing

-- Hinzufügen eines Delimiters für einen regulären Audruck
delimify :: (Eq a) => a -> a -> [a] -> [a]
delimify delim esc s = [delim] ++ concatMap doEsc s ++ [delim]
    where
  doEsc c = if c == delim then [esc, c] else [c]

onTail :: ([a] -> [a]) -> [a] -> [a]
onTail f (x:l) = x : f l
```

```
22 onTail _f l = l
23
24 onInit :: ([a] -> [a]) -> [a] -> [a]
25 onInit = reversify . onTail . reversify
```

Codeblock 13: Deprecated Code

```
1 <?php
2 $string = 'This is a test.';
3 $words = split(' ', $string);
4
5 foreach ($words as $word) {
6     echo $word . "\n";
7 }
```

Codeblock 14: Korrigierter Code

```
1 <?php
2 $string = 'This is a test.';
3 $words = preg_split('/ /', $string);
4
5 foreach ($words as $word) {
6     echo $word . "\n";
7 }
```

Wenn der Code aus Codeblock 13 ausgeführt wird, erzeugt PHP seit der Version 5.3.0 [PHP Group 10c] eine Deprecation Warnung, welche in Codeblock 15 ersichtlich ist.

Codeblock 15: Ausführung von deprecated Code

```
1 // Vor der Transformation
2 fdomig@jupiter:~$ php deprecation.php
3 PHP Deprecated:  Function split() is deprecated in /Users/fdomig/
      deprecation.php on line 3
4 This
5 is
6 a
7 test.
```

Eine Transformation erzeugt den in dem Codeblock 14 dargestellten Code. Wird dieser Code nun ausgeführt, führt dies, wie in Codeblock 16 ersichtlich, zu keiner

Deprecation Warnung mehr und ist somit auch mit zukünftigen PHP Versionen verwendbar.

Codeblock 16: Ausführung von korrigiertem Code

```
// Nach der Transformation
fdomig@jupiter:~$ php deprecation.php
This
is
a
test.
```

Sofern kein *Regulärer-Ausdruck* als Trennzeichen verwendet wird, kann auch die neue Funktion `explode()` verwendet werden, welche ohne die PRCR-Erweiterung auskommt und somit etwas performanter ist. Sollte es sich beim Trennzeichen um keinen Regulären Ausdruck handeln und gleichzeitig nur ein einzelnes Zeichen sein, kann `str_split()` verwendet werden. Diese Optimierung wurde zusätzlich implementiert und kann angewendet werden, nachdem die erste Transformation durchgeführt wird. In Codeblock 17 ist diese Transformation dargestellt.

Codeblock 17: Explode Transformation

```
-- Transformation: "preg_split()" zu "explode()"/"str_split()"
-- Falls das Trennzeichen kein "Regulärer Ausdruck" ist:
--  * Transformation zu "explode()" bei mehreren Zeichen
--  * Transformation zu "str_split()" bei einem Zeichen
pregSplitNonRegex :: Ast -> Transformed Ast
pregSplitNonRegex = modAll $ \ a -> case a of
  ROnlyValFunc _c@(Right (Const [] "preg_split")) w (Right (arg0:
      args)) ->
    if length args `elem` [1, 2]
      then
        case arg0 of
          WSCap w1 (Left (ExprStrLit (StrLit s))) w2 ->
            if null sRegexPost
              then
                if null sRegexUnits
                  then
                    if length args == 1
                      then pure . ROnlyValFunc cStrSplit w $
                          Right argsAlone
                      else transfNothing
                  else
```

```
20                    if any regexUnitIsMeta sRegexUnits
21                      then transfNothing
22                      else pure . ROnlyValFunc cExplode w $ Right
                             (arg0':args)
23              else transfNothing
24            where
25            (sIsDub, sUnits) = strToUnits s
26            (_, (sRegexUnits, sRegexPost)) =
27              regexUnits $ map normalizeStrUnit sUnits
28            cExplode = Right (Const [] "explode")
29            cStrSplit = Right (Const [] "str_split")
30            arg0' = WSCap w1 (strToArg s') w2
31            s' = strUnitsToStr (sIsDub, map last sRegexUnits)
32            argsAlone = onHead (onWSCap1 $ wsStartTransfer w1)
                  args
33          _ -> transfNothing
34      else transfNothing
35  _ -> transfNothing
36
37 regexUnitIsMeta :: [String] -> Bool
38 regexUnitIsMeta [c] = normedStrUnitIsRegexMeta c
39 regexUnitIsMeta ["\\\\", c] = isAlphaNum . chr $ phpOrd c
40
41 -- note that "." and "\x2E" in a PHP str both count as any-char
      for
42 -- preg stuff
43 normedStrUnitIsRegexMeta :: String -> Bool
44 normedStrUnitIsRegexMeta u = any (== chr (phpOrd u)) "|^$*+?.()[{
      "
45
46 strToArg :: String -> Either Expr b
47 strToArg = Left . ExprStrLit . StrLit
```

Weitere mögliche Transformationen für die seit PHP Version 5.3.0 [PHP Group 10d] als deprecated bezeichneten Features, wären unter anderem die Nachfolgenden.

- `call_user_method()` zu `call_user_function()`
- `ereg()`/`ereg_replace()` zu `preg_match()`/`preg_replace()`
- `mysql_escape_string()` zu `mysql_real_escape_string()`

Diese Transformationen würden vorwiegend alte PHP 4 Applikation zu lauffähigen PHP 5.3. Programmen korrigieren. Dadurch würden insbesondere alte, unperformante oder sicherheitskritische Konstrukte durch die jeweiligen neuen Funktionen ersetzt werden. Außerdem werden alle als deprecated bezeichneten Features mit der PHP Version 6 gänzlich entfernt werden und somit nicht nur zu Warnungen sondern zu einem fatalen Fehler mit Abbruch des Programms führen.

4.3 Weitere mögliche Transformationen

Für diese Arbeit wurden lediglich einige wenige und einfache Transformationen implementiert sowie vorgestellt, um zu zeigen, inwiefern es möglich ist, Veränderungen an bestehendem Quellcode vorzunehmen. Mit dem Werkzeug lex-pass können allerdings auch noch viel mehr Transformationen implementiert werden. Unglücklicherweise bestehen in PHP einige Konstrukte, welche sich zwar als schlechte Code-Teile identifizieren lassen, allerdings nicht automatisiert korrigierbar sind. Es ist beispielsweise möglich, zu analysieren, ob private Klassenmethoden beziehungsweise -attribute verwendet werden oder nicht. Jedoch können diese Teile nicht einfach entfernt werden, da aufgrund der Möglichkeit, Methoden zum Beispiel mit Hilfe von `call_user_func()` aufzurufen, nicht festgestellt werden kann, ob eine Methode wirklich verwendet wird oder nicht[11].

Eine weitere Möglichkeit wäre, *Cut-and-Paste* (vgl. Abschnitt 3.3.3) Code-Teile beziehungsweise Funktionen/Methoden zu suchen und diese in einer Funktion zusammenzufassen. Ein mögliches Werkzeug zur Erkennung von Cut-and-Paste Code ist phpcpd von Sebastian Bergmann [Bergmann 10]. Allerdings sind damit keine automatisierten Korrekturen möglich.

Zudem wäre es möglich, sehr komplexen und unübersichtlichen Code mit vielleicht unnötigen If-Ausdrücken zu finden und zu vereinfachen (vgl. Spaghetti-Code in Abschnitte 3.3.1). Gleichzeitig können beispielsweise for-Schleifen, welche eigentlich while-Schleifen sein könnten, transformiert werden.

Größere Anti-Patterns wie Gott-Klassen oder Spaghetti-Code sind zwar erkennbar, allerdings nicht automatisiert korrigierbar. Dazu gibt es bereits einige Werkzeuge wie zum Beispiel Padawan [Anderiasch 10] von Florian Anderiasch, welche solche Anti-Patterns in PHP-Code erkennen können.

[11] Vgl. Halteproblem

5 Praxistest an Open-Source Projekten

Die in Abschnitt 4.2 vorgestellten und implementierten Transformationen werden in den nachfolgenden Abschnitten an bekannten Open-Source Projekten durchgeführt. Es wurden Open-Source Projekte verwendet, da an solchen oft viele Entwickler mit sehr unterschiedlichem Hintergrund und Wissenstand arbeiten. Dadurch werden oft Coding-Standards nicht eingehalten und zudem kommt es oft zu der Erscheinung des Anti-Patterns Spaghetti-Code (vgl. Abschnitt 3.3.1).

Zusätzlich kann anhand der durchgeführten Korrekturen festgestellt werden, inwiefern die getesteten Projekte Qualitätssicherung verwenden, um zumindest diese einfachen Anti-Patterns zu vermeiden.

Ein weiter Grund, Open-Source Projekte automatisiert zu korrigieren, ist es, diese Korrekturen auch wieder in diese Projekte einfließen zu lassen und somit die Software-Qualität ein wenig zu verbessern.

Vor den Korrekturen wurde der Quell-Code des entsprechenden Projekts analysiert. Dabei wurde jeweils die Anzahl der Verzeichnisse sowie Dateien ausgewertet. Zudem wurde die Anzahl von Code-Zeilen und Kommentarzeilen, die Anzahl an Klassen und Funktionen angegeben. Außerdem wurde die zyklomatische Komplexität[12] [McCabe 76, S. 309] im Verhältnis zu der Anzahl an Programm-Code-Zeilen sowie Methoden angegeben.

5.1 PHProjekt

PHProjekt ist ein Projekt Management und Groupware Werkzeug. Es bietet entsprechende Benutzer-, Daten- und Projektverwaltung. Zudem gibt es eine Zugriffsverwaltung und Steuerung von verschiedenen Rollen der Entwickler in einem oder in mehreren Projekten. Zusätzlich wird ein Kalendermodul sowie eine einfache Zeiterfassung angeboten. Projekte beziehungsweise Module können mit Tags versehen, mit einem Suchsystem gesucht und in einem interaktiven User-Interface verändert werden [PHProjekt 10].

PHProjekt verwendet das Zend Framework [Zend Framework 10] sowie das JavaScript Framework Dojo Toolkit [Dojo 10]. Die aktuelle Version von PHProjekt ist PHProjekt 6 und diese wurde auch für die Korrekturen verwendet. Durch die Verwen-

[12] auch “McCabe-Metrik”; zum messen der Komplexität von Software; gibt die Anzahl der binären Verzweigungen + 1 an;

dung des Zend Frameworks wurde auch dieses automatisiert mit korrigiert, allerdings wurde jeweils ausgewiesen, wie viele Korrekturen auf das Zend Framework fallen. Die Statistik zu der PHProjekt Code-Base ist im Anhang im Codeblock 18 ersichtlich.

5.1.1 Explizite Sichtbarkeit

Die Transformation "Explizite Sichtbarkeit" korrigiert in PHProjekt insgesamt 57 Methoden, davon fallen 32 Korrekturen auf PHProjekt und 25 Korrekturen auf das Zend Framework. Bei den insgesamt über 31.000 Methoden von PHProjekt (inklusive dem Zend Framework), kann man hier von einem guten Resultat sprechen.

5.1.2 Zuweisbare Variablen nach rechts

Die Transformation "Zuweisbare Variablen nach rechts" korrigiert insgesamt 4043 If-Ausdrücke, davon fallen 606 Korrekturen auf PHProjekt und 3437 Korrekturen auf das Zend Framework.

5.1.3 Deprecation Korrektur

Die Transformation "Deprecation Korrektur" korrigiert an insgesamt zwei Stellen die Verwendung von `split()` zu `preg_split()`. Beide Änderungen fallen auf das Zend Framework. PHProjekt selbst verwendet diese deprecated Funktion nicht.

Mit der Optimierung, der Transformation von `preg_split()` zu `explode()` bzw. `str_split()`, werden zusätzlich 16 Korrekturen durchgeführt. Davon fällt eine auf PHProjekt und 15 auf das Zend Framework.

5.2 Wordpress

Wordpress ist ein Weblog-System, welches mittlerweile sehr häufig auch als Content Management System verwendet wird. Viele große Webseiten verwenden mittlerweile Wordpress. Über 50% aller Weblogs im Internet werden mit Wordpress betrieben [Built With 10]. Dies sind heute bereits über 200 Millionen Websites.

Wordpress bietet eine Architektur, welche es erlaubt, sehr einfach Plugins und Erweiterungen dafür zu schreiben. Es gibt eine große Anzahl an öffentlich und frei verfügbaren Plugins sowie Themes [Wordpress 10].

Die aktuelle Version von Wordpress ist Wordpress 3.0, diese wurde auch für die automatisierten Korrekturen verwendet.

Bevor Wordpress getestet werden konnte, musste der Quellcode überarbeitet und die verwendete alte Control-Statement-Syntax entfernt werden. Ein Kontroll-Block kann statt mit geschwungenen Klammern, auch mit einem Doppelpunkt sowie einem `end<Statement>` abgegrenzt werden [PHP Group 10e]. Dies führte bereits zu 932 Änderungen in 105 PHP-Dateien von Wordpress. Die Statistik zu der Wordpress Code-Base ist im Anhang im Codeblock 19 ersichtlich.

5.2.1 Explizite Sichtbarkeit

Die Transformation "Explizite Sichtbarkeit" führte zur Korrektur von 1732 Klassen Methoden welche alle bisher nicht als öffentlich deklariert wurden. Wordpress existiert seit 2003 und obwohl zu dieser Zeit bereits PHP 5 den Beta Status erreicht hatte, wurde es für PHP 4 geschrieben. Somit wurde auf Sichtbarkeits-Deklaration völlig verzichtet. Mittlerweile ist eine explizite Deklaration jedoch empfehlenswert.

5.2.2 Zuweisbare Variablen nach rechts

Die Transformation "Zuweisbare Variablen nach rechts" korrigiert im Wordpress Code insgesamt 930 If-Ausdrücke.

5.2.3 Deprecation Korrektur

Die Transformation von "Deprecated Korrekturen" korrigiert insgesamt 17 Mal `split()` zu `preg_split()`. Trotz der bereits 2009 erschienenen Version von PHP 5.3, welche diese Funktion als deprecated bezeichnet, verwendet Wordpress in der aktuellen Version 3.0 also noch relativ häufig diese Funktion.

Mit der Optimierung von `preg_spilt()` zu `explode()` beziehungsweise `str_split()` werden zusätzlich 18 Korrekturen durchgeführt.

5.3 phpMyFAQ

phpMyFAQ ist ein mehrsprachiges FAQ-System[13]. Derzeit werden 38 Sprachen unterstützt. Es ist komplett Datenbank-basiert und kann mit verschiedenen Datenbanksystemen, wie zum Beispiel MySQL, PostgreSQL, SQLite oder Oracle betrieben

[13] FAQ, Frequently Asked Questions

werden. phpMyFAQ unterstützt Benutzer- sowie Gruppenverwaltung, verwendet ein Revisions-System und bietet ein Kommentar-Modul an. Es wurde von Thorsten Rinne geschrieben und liegt aktuell in der Version 2.7 vor, welche auch für die Korrekturen verwendet wurde [PhpMyFAQ 10]. Die Statistik zu der phpMyFAQ Code-Base ist im Anhang im Codeblock 20 ersichtlich.

5.3.1 Explizite Sichtbarkeit

Die Transformation "Explizite Sichtbarkeit" führte insgesamt zu 285 Korrekturen an Klassen-Methoden. Bei insgesamt 3114 Klassen-Methoden im gesamten phpMyFAQ Code wurden somit ca. 9% aller Methoden korrigiert.

5.3.2 Zuweisbare Variablen nach rechts

Die Transformation "Zuweisbare Variablen nach rechts" korrigiert in phpMyFAQ Code insgesamt 254 If-Ausdrücke.

5.3.3 Deprecation Korrektur

In phpMyFAQ wird die deprecated Funktion `split()` nicht verwendet und somit gibt es hier keine Korrektur. Auch die optimierte Transformation von `preg_split()` zu `explode()` führt zu keiner einzigen Transformation. Es wird insgesamt 22 mal `preg_split()` mit einem regulären Ausdruck, sowie 60 mal `explode()` ohne regulären Ausdruck verwendet. In diesem Bereich ist phpMyFAQ also vorbildlich.

6 Fazit

6.1 Welche Gründe sprechen für Korrekturen

Aufgeräumter und übersichtlicher Programm-Code führt zu besserer Wartbarkeit und Wiederverwendbarkeit. Im sich schnell verändernden Bereich der Webentwicklung ist es außerordentlich wichtig, rasch neue Features implementieren zu können.

Eine automatisierte Korrektur an Software ist allerdings nicht immer sehr einfach. Speziell, wenn man komplizierte Veränderungen vornehmen möchte. Jegliche Automatisierung führt jedoch auch unweigerlich dazu, dass solchen Korrekturen nicht mehr genau nachgeprüft werden. Dadurch kann es auch zu ungewünschten Korrekturen kommen, welche man übersieht. Kleine Änderungen hingegen können automatisiert durchgeführt werden, ohne dadurch unerwünschte Resultate zu produzieren.

Ein automatisches Erkennen auch von komplexeren Anti-Patterns ist auf jeden Fall sinnvoll. Anti-Patterns sollten gesucht und von den Entwicklern geprüft sowie gegebenenfalls von Hand korrigieren werden.

6.2 Welche Anti-Pattern sollten automatisiert korrigiert werden?

Vor allem Anti-Patterns mit wenig Veränderung am Programm-Code sowie mit einem vorhersehbaren Resultat können problemlos automatisiert korrigiert werden. Alle in Abschnitt 4.2 implementierten lex-pass Transformationen für einfach zu korrigierende Anti-Patterns können mit den jeweils erläuterten Einschränkungen verwendet werden.

Automatisierung birgt immer das Risiko eines unerwünschten Resultats. Deswegen sollte nach einer automatischen Korrektur auch immer der transformierte Code überprüft werden. Keinesfalls sollten Korrekturen automatisiert durchgeführt werden, ohne die entsprechenden Resultate zu überprüfen und testen.

6.3 Weitere Entwicklung

Die Entwickler der Programmiersprache PHP werden in zukünftigen Versionen bestimmt weiterhin versuchen mehr Konsistenz zu erreichen, womit PHP für viele Unternehmen auch in Zukunft als Sprache zur Webentwicklung verwendet werden wird. Alle derzeit als deprecated bezeichneten Features, werden in der nächsten großen Version von PHP (PHP 6) vollständig entfernt und sehr wahrscheinlich werden einige der jetzigen Funktionen als deprecated bezeichnet werden. Somit spricht vieles für

die Weiterentwicklung im Bereich der Qualitätssicherung von PHP Software. Viele Werkzeuge, die in dieser Arbeit verwendet und vorgestellt wurden, werden ständig erweitert und helfen immer bessere, sicherere und performantere Software zu entwickeln.

Eine Weiterentwicklung des verwendeten Werkzeugs lex-pass und dessen Einsatz würde die Qualität von PHP-Software im Allgemeinen deutlich verbessern. Wie in Kapitel 5 aufgezeigt, ist auch der Einsatz für Open-Source Projekte empfehlenswert. Dadurch, dass lex-pass und die entwickelten Transformationen online als Open-Source verfügbar sind, ist eine Erweiterung dieses Werkzeugs möglich und wünschenswert.

Literatur

[Brown 98] W. H. Brown, R. C. Malveau, H. W. McCormick III, et al.: Anti Patterns Refactoring Software, Architectures, and Projects in Crisis. New York: John Wiley & Sons, Inc., 1998.

[Dijkstra 59] E. W. Dijkstra: Numerische Mathematik 1, 1959.

[Fielding 00] R. Fielding: Architectural Styles and the Design of Network-based Software Architectures Dissertation: University of California, Irvine, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf, besucht am 24.06.2010.

[Gamma 94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. MA: Addison-Wesley, 1994.

[McCabe 76] T. J. McCabe: A Complexity Meassure IEEE Transactions on software engineering, Vol. SE-2, No. 04, Dezember 1976. http://www.literateprogramming.com/mccabe.pdf, besucht am 03.07.2010.

Web-Referenzen

[Anderiasch 10] Florian Anderiasch: PADAWAN - PHP AST-based Detection of Anti-Patterns, Workarounds And general Nuisances http://github.com/mayflowergmbh/padawan, besucht am 07.06.2010.

[Bergmann 10] Sebastian Bergmann: phpcpd - Copy/Paste Detector for PHP http://github.com/mayflowergmbh/phpcpd, besucht am 05.07.2010.

[Built With 10] Built With: Wordpress Usage Statistics http://trends.builtwith.com/blog/WordPress, besucht am 04.07.2010.

[Doctrine 10] Doctrine: PHP Object Persistence Libraries and More http://www.doctrine-project.org/, besucht am 03.07.2010.

[Dojo 10] Dojo Toolkit: Übersicht http://framework.zend.com/, besucht am 04.07.2010.

[Facebook 07] The Facebook Blog: PHP and Facebook http://blog.facebook.com/blog.php?post=2356432130, besucht am 22.06.2010.

[Flickr 10] Flickr: Flickr and PHP http://wheel.troxo.com/wp-content/uploads/2008/01/flickr_php.pdf, besucht am 22.06.2010.

[Fowler 06] M. Fowler: Continuous Integration http://www.martinfowler.com/articles/continuousIntegration.html, besucht am 30.06.2010

[Gutmans 04] A. Gutmans: What's new in PHP 5? http://devzone.zend.com/article/1714, besucht am 02.06.2010.

[Grossman 06] J. Grossman: The origins of Cross-Site Scripting (XSS) http://jeremiahgrossman.blogspot.com/2006/07/origins-of-cross-site-scripting-xss.html, besucht am 03.07.2010.

[Heise 10] Heise Online: 15 Jahre PHP http://www.heise.de/newsticker/meldung/15-Jahre-PHP-1017276.html, besucht am 20.06.2010.

[Karwin 09] Bill Karwin: Why Should You Use an ORM? http://karwin.blogspot.com/2009/01/why-should-you-use-orm.html, besucht am 06.07.2010.

[Lex-Pass 10] Facebook Open-Source: lex-pass http://github.com/facebook/lex-pass, besucht am 04.07.2010.

[Pear 10] Pear: Coding Standards http://pear.php.net/manual/en/standards.php, besucht am 01.07.2010

[PHP Group 07] The PHP Group: Usage of PHP http://www.php.net/usage.php, besucht am 18.06.2010.

[PHP Group 10a] The PHP Group: History of PHP http://www.php.net/manual/en/history.php.php, besucht am 09.06.2010.

[PHP Group 10b] The PHP Group: Magische Methoden http://php.net/manual/en/language.oop5.magic.php, besucht am 23.06.2010.

[PHP Group 10c] The PHP Manual: Split http://at2.php.net/split, besucht am 03.7.2010.

[PHP Group 10d] The PHP Manual: Deprecated Features http://php.net/manual/de/migration53.deprecated.php, besucht am 04.07.2010.

[PHP Group 10e] The PHP Manual: Alternative syntax for control structures http://php.net/manual/en/control-structures.alternative-syntax.php, besucht am 04.07.2010.

[PHP Group 10f] The PHP Manual: Session http://at.php.net/manual/en/book.session.php, besucht am 03.07.2010.

[PHP Group 10g] The PHP Manual: Filter http://www.php.net/manual/de/book.filter.php, besucht am 05.07.2010.

[PhpMyFAQ 10] phpMyFAQ Website: Features http://www.phpmyfaq.de/features.php, besucht am 05.07.2010.

[PHP PCRE 10] PHP: Perl Compatible Regular Expressions http://at.php.net/manual/en/book.pcre.php, besucht am 04.07.2010.

[PHProjekt 10] PHProjekt Website: Features http://www.phprojekt.com/features, besucht am 04.07.2010.

[PHPUnit 10] PHPUnit: Website http://www.phpunit.de/, besucht am 04.07.2010.

[PMD 10] The PMD Website: Info http://pmd.sourceforge.net/, besucht am 29.06.2010.

[Propel 10] Propel: Smarty, easy object persistence http://www.propelorm.org/, besucht am 03.07.2010.

[Tnx 10] TNX: PHP In Contrast To Perl - Training Wheels Without The Bike http://www.tnx.nl/php.html

[Wikipedia 10] Wikipedia: Wikipedia FAQ/Technical http://en.wikipedia.org/wiki/Wikipedia:Technical_FAQ, besucht am 22.06.2010.

[Wordpress 10] Wordpress: Extend http://wordpress.org/extend/, besucht am 04.07.2010.

[Zend Framework 10] Zend Framework: Übersicht http://framework.zend.com/, besucht am 04.07.2010.

Anhang A Statistik PHProjekt

Codeblock 18: Statistik PHProjekt

```
1  Directories:                                    609
2  Files:                                         5066
3
4  Lines of Code (LOC):                        1039558
5    Cyclomatic Complexity / Lines of Code:       0.11
6  Comment Lines of Code (CLOC):                472426
7  Non-Comment Lines of Code (NCLOC):           567132
8
9  Namespaces:                                       0
10 Interfaces:                                     222
11 Classes:                                       5124
12   Abstract:                                     390 (7.61%)
13   Concrete:                                    4734 (92.39%)
14   Average Class Length (NCLOC):                 111
15 Methods:                                      33752
16   Scope:
17     Non-Static:                               31292 (92.71%)
18     Static:                                    2460 (7.29%)
19   Visibility:
20     Public:                                   26588 (78.77%)
21     Non-Public:                                7164 (21.23%)
22   Average Method Length (NCLOC):                 16
23   Cyclomatic Complexity / Number of Methods:   2.87
24
25 Anonymous Functions:                             4
26 Functions:                                       4
27
28 Constants:                                    7734
29   Global constants:                             94
30   Class constants:                            7640
```

Anhang B Statistik Wordpress

Codeblock 19: Statistik Wordpress

```
Directories:                                   22
Files:                                        292

Lines of Code (LOC):                       157804
  Cyclomatic Complexity / Lines of Code:     0.20
Comment Lines of Code (CLOC):               48917
Non-Comment Lines of Code (NCLOC):         108887

Namespaces:                                    0
Interfaces:                                    0
Classes:                                     168
  Abstract:                                    0 (0.00%)
  Concrete:                                  168 (100.00%)
  Average Class Length (NCLOC):              256
Methods:                                    1732
  Scope:
    Non-Static:                             1732 (100.00%)
    Static:                                    0 (0.00%)
  Visibility:
    Public:                                 1732 (100.00%)
    Non-Public:                                0 (0.00%)
  Average Method Length (NCLOC):              24
  Cyclomatic Complexity / Number of Methods:  5.52

Anonymous Functions:                           0
Functions:                                  2184

Constants:                                   327
  Global constants:                          327
  Class constants:                             0
```

Anhang C Statistik phpMyFAQ

Codeblock 20: Statistik PhpMyFAQ

```
Directories:                                   41
Files:                                        714

Lines of Code (LOC):                       454010
  Cyclomatic Complexity / Lines of Code:     0.04
Comment Lines of Code (CLOC):               71016
Non-Comment Lines of Code (NCLOC):         382994

Namespaces:                                    0
Interfaces:                                   16
Classes:                                     284
  Abstract:                                   18 (6.34%)
  Concrete:                                  266 (93.66%)
  Average Class Length (NCLOC):              264
Methods:                                    3114
  Scope:
    Non-Static:                             2890 (92.81%)
    Static:                                  224 (7.19%)
  Visibility:
    Public:                                 2502 (80.35%)
    Non-Public:                              612 (19.65%)
  Average Method Length (NCLOC):              24
  Cyclomatic Complexity / Number of Methods:  4.77

Anonymous Functions:                           0
Functions:                                   158

Constants:                                  1808
  Global constants:                         1598
  Class constants:                           210
```