MARTIN DUSCHEK, 67664, 16MI1-B

EVOLUTION VON CODE BEI MAJOR-RELEASES VON PROGRAMMIERSPRACHEN

[30. Oktober 2019 at 15:17 – classicthesis v4.6]

EVOLUTION VON CODE BEI MAJOR-RELEASES VON PROGRAMMIERSPRACHEN

am Beispiel der Migration zu PHP7

MARTIN DUSCHEK, 67664, 16MI1-B

I-ITWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

November 2019

[30. Oktober 2019 at 15:17 – classicthesis v4.6]

Martin Duschek, 67664, 16MI1-B: *Evolution von Code bei Major-Releases von Programmiersprachen*, am Beispiel der Migration zu PHP7, © November 2019

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

PCRE	Perl Compatible Regular Expressions
API	Application Programming Interface
PHP	PHP: Hypertext Preprocessor
ISO	International Organization for Standardization („Internationale Organisation für Normung“)
IEC	International Electrotechnical Commission („Internationale Elektrotechnische Kommission“)
php7mar	PHP 7 Migration Assistant Report
CGI	Common Gateway Interface
RFC	Request for Comments

1 EINLEITUNG

Am 03. Dezember 2015 erschien mit PHP 7.0.0 das erste Major-Release der Programmiersprache seit elf Jahren. Damit einhergehend wurde die Einstellung der Weiterentwicklung der vorhergehenden Version 5 für den 10. Januar 2019 angekündigt. Der Entwicklungsstopp führt dazu, dass Sicherheitslücken in der Implementation der alten Version nicht mehr geschlossen werden, was wiederum dazu führt, dass bereits ausgelieferte Software angreifbar wird sobald neue Lücken gefunden werden.
Derzeit setzen 79,1% der 10 Millionen meistgenutzten Webseiten PHP als serverseitige Programmiersprache ein, davon 61,5% PHP in der veralteten Version 5[1]. Diese Installationen können allesamt als unsicher eingestuft werden. Seit der letzten Veröffentlichung unter Version 5 wurden vier neue Schwachstellen veröffentlicht[2], die in unterstützten Versionen bereits geschlossen wurden.

1.1 MOTIVATION

Die Firma *tickets75*, eine unabhängige Ticketagentur, die sich auf die Vermittlung von Tickets für begehrte Veranstaltungen über den eigenen Onlineshop spezialisiert hat. Als e-Commerce-Unternehmen ist der reibungslose Betrieb der Onlinepräsenz besonders wichtig. Ebenso hat die Sicherheit von Kunden, insbesondere in Bezug auf deren Zahlungsmittel oberste Priorität. Um diese Sicherheit weiterhin gewährleisten zu können, soll der Onlineshop für die Ausführung unter PHP: Hypertext Preprocessor (PHP) optimiert werden. Der Shop basiert auf der quelloffenen e-Commerce-Plattform *Gambio*, wurde in der Vergangenheit jedoch stark angepasst, sodass eine einfache Aktualisierung des Grundsystems nicht mehr in betracht gezogen werden kann.

1.2 AUFGABENSTELLUNG

Ziel dieser Arbeit ist die Evaluation verschiedener Techniken und Technologien, die ein Upgrade der Programmiersprache in Softwareprojekten einfacher und nachhaltig gestalten oder erst in effizienter

1 W3Techs, „Usage statistics of PHP for websites“, https://w3techs.com/technologies/details/pl-php/all/all

2 CVE details, „PHP 5.6.40 Security Vulnerabilities“, https://www.cvedetails.com/vulnerability-list/vendor_id-74/product_id-128/version_id-298516/PHP-PHP-5.6.40.html

Weise ermöglichen. Dabei wird die Migration eines Onlineshops von PHP 5.6 zu PHP 7.x als praktisches Beispiel herangezogen und die verschiedenen Ansätze geprüft. Als Leitfaden dient der Internationale Standard **ISO/IEC 14764**.

1.3 AUFBAU

2 GRUNDLAGEN

2.1 SOFTWAREWARTUNG NACH ISO/IEC 14764

Die International Organization for Standardization („Internationale Organisation für Normung“) (ISO) ist ein im Jahr 1947 gegründeter Zusammenschluss internationaler Normungskommissionen, mit dem Ziel internationale Standards zu entwickeln und zu etablieren.[Intb] Die Entwicklung von Standards wird von der 1906 gegründeten Schwesterorganisation International Electrotechnical Commission („Internationale Elektrotechnische Kommission“) (IEC) übernommen, oftmals in Zusammenarbeit mit der ISO.[Inta] Aus dieser Zusammenarbeit entstandene Standards tragen die Kürzel beider Organisationen im Namen. Ein solcher Standard ist **ISO/IEC 14764** mit dem Titel **Software Engineering — Software Life Cycle Processes — Maintenance**, der erstmals im Jahr 1999 veröffentlicht wurde. **ISO/IEC 14764** normiert den Prozess der Wartung von Software bis zu deren Einstellung. Darin wird unter Anderem beschrieben, welche Schritte bei der Migration von Software zu befolgen sind, sobald diese an eine neue Umgebung angepasst werden muss. Folgende Aktionen sind durch den Ausführenden nach **ISO/IEC 14764** umzusetzen:

- Analyse der Anforderungen und Definition der Migration
- Entwicklung von Werkzeugen zur Migration
- Entwicklung der an die neue Umgebung angepassten Software
- Durchführung der Migration
- Verifikation der Migration
- Support der alten Umgebung

2.2 DIE PROGRAMMIERSPRACHE PHP

PHP ist eine Skriptsprache, welche seit 1994 entwickelt wird und seit 1995 Open-Source bereitgestellt wird. Obwohl PHP viele Einsatzzwecke abdeckt, wird es hauptsächlich dazu genutzt, dynamische Websites zu programmieren. Rasmus Lerdorf, der Erfinder von PHP, entwickelte zunächst eine Reihe von Common Gateway Interfaces (CGIs) in *C*, um die Anzahl der Besucher seiner Webseite zu erfassen. Diese CGIs wurden immer umfangreicher, wodurch sich im Laufe der Zeit eine eigenständige Programmiersprache entwickelte, die durch den **Zend-Engine** genannten Compiler interpretiert wird. PHP steht auf Platz

6 der beliebstesten Programmiersprachen weltweit[Car19] und ist die Grundlage für bekannte Projekte wie das Content Management System *Wordpress*[1] oder die e-Commerce Plattform *Magento*[2]. Die Weiterentwicklung von PHP wird von einem Team von Freiwilligen vorangetrieben. Vorschläge für neue Funktionen oder Änderungen bestehender Funktionen werden über Request for Commentss (RFCs) eingebracht, über deren Implementation in PHP das Team abstimmen kann[3].

Abbildung 2.1: Major-Releases von PHP im Zeitverlauf

2.3 VERSIONIERUNG VON SOFTWARE

Für die Benennung von Releases einer Software gibt es keinen einheitlichen Standard. Jedem Entwickler steht es frei, seine Software nach einem bestimmten Muster zu benennen. So benennt *Canonical* das Betriebssystem Ubuntu stets nach der Jahres- und Monatszahl der Veröffentlichung (bspw. erschien Ubuntu 19.10 im Oktober 2019). PHP hingegen implementiert lose die Spezifikation *Semantic Versioning 2.0.0*. Diese legt ein Muster für Versionsnummern fest, das folgendermaßen aufgebaut ist:
Die Versionsnummer folgt immer dem Muster

Major.Minor.Patch[-Pre-Release]

Mit jedem Release wird eine der Nummern inkrementiert, wobei die nachfolgenden Nummern auf „0“ zurückgesetzt werden und folgende Regeln für die Nummerierung gelten [PW]:

- **Major** wird inkrementiert, wenn inkompatible Änderungen an der API vorgenommen werden
- **Minor** wird inkrementiert, wenn abwärtskompatible Funktionalitäten eingeführt werden
- **Patch** wird inkrementiert, wenn abwärtskompatible Bugfixes implementiert werden

1 Wordpress, https://wordpress.org
2 Magento, https://magento.com
3 PHP: How to get involved, https://www.php.net/get-involved.php

- **Pre-Release** ist eine alphanumerische Zeichenkette, die frei vergeben werden kann

Von einem **Major-Release** spricht man folglich dann, wenn inkompatible Änderungen an der API stattfinden. Die Nummerierung der Releases bei PHP folgt zwar der Spezifikation, jedoch mit einer Ausnahme. So beträgt der Sprung zwischen den beiden letzten veröffentlichten Major-Releases zwei Nummern (von 5.x.x zu 7.x.x). Diese Abweichung von der Spezifikation wurde aus Gründen des Marketings beschlossen, da die Arbeit an der unveröffentlichten Version 6 im Jahr 2010 aufgrund von Schwierigkeiten in der Implementierung von Unicode abgebrochen wurde, jedoch bereits Material (Blogposts, Lehrbücher etc.) zu dieser Version im Umlauf waren und Vewirrung von Nutzern vermieden werden sollte. Ein beispielhafter Auschnitt der Versionshistorie von PHP wird in Abbildung 2.2 gezeigt, dabei werden die einzelnen Versionsschritte deutlich, wobei einzelne Versionen zur Vereinfachung ausgelassen wurden.

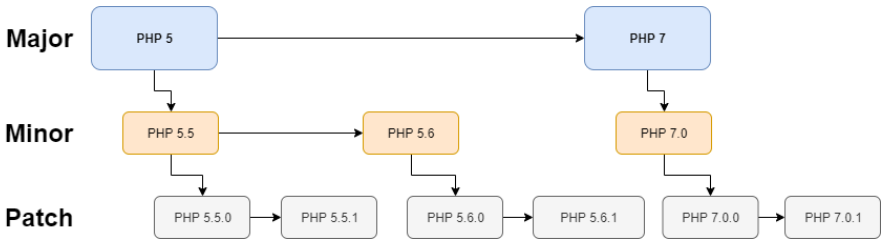

Abbildung 2.2: Grafische Darstellung der Versionierung von PHP

3 ÄNDERUNGEN DER PHP-API

Dieser Abschnitt stellt eine Auswahl der Bedingungen vor, welche die PHP-API in Version 7 gegenüber Version 5 an lauffähige Software stellt und welche neuen Mittel Entwicklern zur Verfügung gestellt werden. Die Änderungen werden in den Kontext der Weiterentwicklung der Programmiersprache gestellt, um Aussagen über die Gründe dieser zu treffen und - **ISO/IEC 14764** entsprechend - die Anforderungen an die Migration festzustellen. Die Auswahl basiert zu einem Teil auf der Auswertung des mit *php7mar* generierten Berichts. Mit Hilfe eines Python-Skripts wurden die für die Migration des Onlineshops wichtigsten Funktionen herausgefiltert. Weitere Funktionen wurden im Gespräch mit Kollegen der Firma *tickets75* als wichtig herausgearbeitet, da diese erfahrungsgemäß oft angewandt werden.

3.1 ABWÄRTSINKOMPATIBLE ÄNDERUNGEN

Änderungen in dieser Kategorie führen in älteren Versionen zu Fehlern oder unerwartetem Verhalten und sind in dieser Umgebung somit nicht lauffähig. Durch diese wird ein Wechsel der Ausführungsumgebung zwingend vorrausgesetzt.

3.1.1 *Interpretation indirekter Variablenzugriffe*

PHP bietet die Möglichkeit des indirekten Zugriffs auf Variablen. Das bedeutet, dass der Wert einer Variablen den Namen einer weiteren Variablen darstellt. Bisher war die Syntax durch mehrere Sonderfälle geregelt. Mit PHP 7 wird eine strikte Evaluierung eines solchen Audrucks von links nach rechts eingeführt, um die Nutzung dieser zu vereinheitlichen. Wie sich die einzelnen Fälle Unterscheiden ist in Tabelle 3.1 aufgeführt.

Tabelle 3.1: Vergleich der Evaluation indirekter Variablen zwischen PHP 5 und PHP 7

Ausdruck	**PHP 5**	**PHP 7**
$$foo['bar']['baz']	${$foo['bar']['baz']}	($$foo)['bar']['baz']
$foo->$bar['baz']	$foo->{$bar['baz']}	($foo->$bar)['baz']
$foo->$bar['baz']()	$foo->{$bar['baz']}()	($foo->$bar)['baz']()
Foo::$bar['baz']()	Foo::{$bar['baz']}()	(Foo::$bar)['baz']()

3.1.2 *Abfrage von Funktionsparametern mit func_get_args()*

Der Aufruf der Funktion *func_get_args()* in einer benutzerdefinierten Methode liefert ein Array mit Kopien der Argumente, mit denen die Methode aufgerufen wurde. Damit lassen sich Funktionen mit einer variablen Anzahl an Parametern realisieren. Im Gegensatz zu PHP 5 werden nun nicht mehr die Argumente zum Zeitpunkt des Aufrufs der Methode zurückgegeben, sondern deren (möglicherweise bis dahin veränderte) Wert zum Zeitpunkt des Aufrufs von *func_get_args()*. Das Beispiel 3.1 gibt unter PHP 5 den Wert „1“ aus, unter PHP 7 hingegen den Wert „2“.

Listing 3.1: Beispiel des Aufrufs von func_get_args()

```
<?php
function foo($bar) {
    bar++;
    echo(func_get_args(0));
}

foo(1);
?>
```

3.1.3 *Foreach ändert nicht mehr den Arrayzeiger*

3.1.4 *Konstruktoraufrufe per Referenz*

Aufrufe des Operators *new* können nicht mehr per Referenz an eine Variable übergeben werden. Dieses Verhalten ist seit PHP 5.3 als veraltet markiert und wurde in PHP 7 entfernt, da ein Konstruktoaufruf automatisch eine Referenz auf das Objekt zurückgibt.[PHPc]

3.1.5 *Automatische Maskierung mit magic_quotes_runtime*

Magic Quotes (dt. „Magische Anführungszeichen“) waren eine in PHP eingebaute Option, die es unerfahrenen Entwicklern erleichtern sollte, sich gegen SQL-Injections zu schützen. Dazu wurden Sonderzeichen in Strings, die aus externen Quellen stammten, automatisch mit einem Backslash maskiert. Da die Option standardmäßig aktiv war, jedoch per Konfiguration ausgeschaltet werden konnte, war es möglich, dass Code auf verschiedenen Systemen unterschiedlich reagierte und sich Entwickler in vermeintlicher Sicherheit wägten.[PHPa] Mit PHP 5.4 wurde jegliche Funktionalität entfernt, Funktionen wie *set_magic_quotes_runtime()* und *magic_quotes_runtime()* aus Gründen der Kompatibilität jedoch nur als veraltet markiert. Ein Einsatz dieser Funktionen unter PHP 7 resultiert in einem Fehler.

3.1.6 *Änderungen der Funktion list*

Die Funktion *list()* ermöglicht die Zuweisung von Variablen als wären diese ein Array. Quellcode der sich auf die bisherige Praxis verlässt, dass *list()* den letzten angegebenen Wert zuerst zuweist, kann nun nicht mehr eingesetzt werden, da die Reihenfolge der Zuweisung bei Verwendung des Operators

umgekehrt wurde. Obwohl keine klaren Gründe für die Änderung angegeben werden, liegt die Vermutung nahe, dass dadurch Verwirrungen über das Verhalten der Funktion vermindert werden sollen. Listing 3.2 würde bei der Ausführung unter PHP 5 beispielsweise „3“ als Ergebnis ausgeben. Dies entspricht nicht der erwartbaren Reihenfolge. Zudem verliert *list* die Möglichkeit Strings zu entpacken. Die Möglichkeit, leere Argumente zu übergeben wurde ebenso entfernt. Beide Optionen werden in Beispiel 3.3 gezeigt.

Listing 3.2: Beispiel der Verwendung von list()

```
<?php
list($foo[], $foo[], $foo[]) = [1,2,3];

echo($foo[0])
?>
```

Listing 3.3: Beispiel unerlaubter Verwendungen von list()

```
<?php
//Entpacken eines Strings mit list()
$foo = 'bar';
list($a[], $a[], $a[]) = $foo;

//Angabe leerer Argumente in list()
$foo = [3,2];
list(,,) = $foo;
?>
```

3.1.7 *Switch-Anweisungen mit mehreren default-Blöcken*

Switch-Anweisungen, welche mehrere default-Blöcke enthalten werden ab sofort als fehlerhafte Syntax erkannt und werfen einen Fehler. Dies war bisher nicht der Fall, allerdings wurde bei einer solchen Anweisung nur der letzte default-Block ausgewertet. Dieses Verhalten zeigt sich in Listing 3.4. Der entsprechende Codeausschnitt gibt unter PHP 5 immer Ëvaluatedäus, bei dem Versuch der Auführung unter PHP 7 wird ein Fehler geworfen. Damit wird ein Bruch der PHP-Spezifikation [PHPb] behoben.

Listing 3.4: Beispiel meherer default-Blöcke in Switch-Anweisungen

```
<?php
switch(1) {
    default:
        echo("Never evaluated");
        break;
    default:
        echo("Evaluated")
        break;
}
?>
```

3.2 VERALTETE FUNKTIONEN

Als veraltet markierte Funktionen sind in der neuen Umgebung zwar noch unterstützt, sollten aber nach Möglichkeit nicht mehr eingesetzt und schnellstmöglich durch geeignete Funktionen ersetzt werden, da sie möglicherweise in zukünftigen Versionen entfernt oder verändert werden. Werden diese Funktionen trotzdem eingesetzt, wird eine Warnung ausgegeben, die Programmierer darauf hinweisen soll, dass die Verwendung der Funktion möglicherweise gefährlich sein kann. Die Lauffähigkeit des Programms wird bis zur abschließenden Entfernung der Funktion jedoch nicht beeinflusst. [Ora04]

3.2.1 *Implizite Benennung von Konstruktoren*

Mit der Einführung der objektorientierten Programmierung in PHP 4 wurde festgelegt, dass Funktionen mit dem selben Namen wie die umschließende Klasse implizit als Konstruktor der Klasse erkannt werden. Ein Beispiel zur Implementierung eines Konstruktors nach diesem Prinzip ist in Listing 3.5 dargestellt. PHP 7 unterstützt diese Notation zwar noch, allerdings wird die, in PHP 5 eingeführte, explizite Benennung mit dem Schlüsselwort *__construct* (siehe Listing 3.6) bevorzugt. Hierdurch soll die Verwirrung darum, wann eine Funktion einen Konstruktor darstellt aufgehoben werden. [Mor14a]

Listing 3.5: Beispiel eines impliziten Konstruktors

```
<?php
class foo {
    function foo($a) {
        echo("Created instance of class 'foo'");
    }
}
?>
```

Listing 3.6: Beispiel eines expliziten Konstruktors

```
<?php
class foo {
    function __construct($a) {
        echo("Created instance of class 'foo'");
    }
}
?>
```

3.2.2 *Statische Aufrufe nicht-statischer Funktionen*

Mit dem Schlüsselwort *static* versehene Funktionen einer Klasse erlauben das Benutzen der Funktion, ohne die Instantiierung der Klasse selber. Damit steht die entsprechende Funktion nicht im Kontext eines Objekts, sondern im Kontext der entsprechenden Klasse. Im Gegensatz zu anderen objektorientierten Programmiersprachen (bspw. Java) war es in PHP bisher möglich, auch nicht-statische Methoden ohne eine Instantiierung zu verwenden. Diese Möglichkeit wurde mit PHP 7 für veraltet erklärt und sollte nicht mehr genutzt werden. Dadurch werden Programmierfehler verhindert, da der Kontext, in dem eine Funktion ausgeführt wird nun Eindeutig ist. Das Beispiel 3.7 wird eine Warnung ausgeben, dass eine nicht-statische Methode statisch aufgerufen wird.

Listing 3.7: Beispiel eines statischen Aufrufs einer nicht-satischen Funktion in PHP 7

```
<?php
class foo {
    function bar() {
        echo("'bar' is not a static function");
    }
}

foo::bar();
?>
```

3.3 GEÄNDERTE FUNKTIONEN

In diese Gruppe fallen Funktionen, deren Benutzung und/oder Verhalten geändert wurden, allerdings nicht vollständig veraltet sind. Dies bedeutet zum Beipiel, dass einzelne Funktionsparameter entfernt wurden oder andere Datentypen zurückgegeben werden.

3.3.1 *preg_replace*

Die Funktion *preg_replace()* ersetzt Teile einer Zeichenkette nach einem, als regulärem Ausdruck angegebenen, Muster. Mit *PCRE-Modifikatoren* kann die Verhaltensweise des regulären Ausdrucks gesteuert werden.

In PHP 7 wurde der Modifikator */e* entfernt, mit dem die Zeichenkette durch das Ergebnis einer Funktion ersetzt wird. Ein Beipiel ist die Umwandlung aller kleingeschriebenen Zeichen eines Strings in Großbuchstaben, dargestellt in Listing 3.8. Die Verwendung des Modifikators wird aufgrund der Maskierungsregeln für bestimmte Zeichen als sehr kompliziert beschrieben. Gleichzeitig stellt die einfache Art der Evaluierung des Ergebnisses keine Schutzmechanismen zur Verfügung, wodurch Sicherheitslücken entstehen können, sobald es einem Angreifer gelingt, ausführbaren Code in diese Funktion einzuschleusen.

Listing 3.8: Beispiel der Nutzung von preg_replace mit dem Modifikator /e

```
<?php
$uppercase = preg_replace(
    "/([a-z]*)/e",
    "strtoupper('$1')",
    $mixedCase
);
?>
```

3.4 DOPPELTE FUNKTIONSPARAMETER

Bisher war es möglich, Funktionen zu definieren, die mehrere Parameter mit dem selben Namen akzeptieren. Dabei wurde der Variablen der Wert des zuletzt definierten Parameters zugewiesen. Beispiel 3.9 verdeutlicht dieses Verhalten, indem die Ausgabe den Wert „3“ zeigt. Diese Praxis ist in PHP 7 nicht mehr möglich und führt zu einem Fehler.

Listing 3.9: Beispiel meherer gleichnamiger Funktionsparameter

```
<?php
    function foo($x, $x){
        echo($x);
    }

    foo(1,2);
?>
```

3.4.1 *setlocale*

Die Funktion *setlocale()* dient dazu, regionale Eigenheiten abzubilden. Dazu gehören zum Beispiel unterschiedliche Datumsformate oder die Formatierung von Zahlen (bspw. Trennzeichen für Dezimalzahlen). Für die Einstellung einer Region können Kategorien angegeben werden, auf die sich die Änderung auswirken soll. Ab Version 7 ist es nicht mehr möglich, die Kategorie als Zeichenkette anzugeben. Für diese Änderung ist kein Grund angegeben, allerdings liegt die Vermutung nahe, dass sich dadurch die Prüfung der Kategorie innerhalb

der Funktion vereinfachen lässt, da PHP verschiedene benannte Konstanten zur Anwendung zur Verfügung stellt. Dies lässt sich auch durch die Historie der betreffenden Funktion im Quellcode belegen, durch die ersichtlich wird, dass ein großer Teil der Überprüfung der Funktionsparameter entfernt wurde. [nik14]

3.5 NEUE FUNKTIONEN

3.5.1 *Anonyme Klassen*

Mit dem Hinzufügen von anonymen Klassen implementiert PHP ein Konzept, das bereits aus anderen Objektorientierten Sprachen, beispielsweise Java [Oraa], bekannt ist. Diese können benutzt werden, um gleichzeitig mit der Definition eine einmalig genutzte Klasse zu instanziieren, ohne eigens dafür eine neue lokale Klasse erstellen zu müssen., wie in Listing 3.10 dargestellt wird.

Listing 3.10: Beispiel der Nutzung anonymer Klassen

```
<?php
$foo = new class {
    public function bar() {
        echo "Hello World";
    }
};

$foo->bar();
?>
```

3.5.2 *preg_replace_callback_array()*

Ähnlich wie die im Abschnitt 3.3.1 beschriebene Funktion *preg_replace()* mit dem Modifikator */e*, ersetzt *preg_replace_callback_array()* Zeichenketten anhand eines Musters und einer Ersetzungsfuntion. Im eingeführten *preg_replace_callback_array()* kann nun ein assiozatives Array angegeben werden, das mehrere Muster und ihre entsprechenden Callback-Funktionen enthält. Durch die Nutzung verschiedener Ersetzungsfunktionen kann auf die Nutzung einer einzelnen, stark verzweigten Ersetzungsfuntion verzichtet werden. Dadurch wird entsprechender Quellcode lesbarer und besser wartbar (vgl. [Mar12, S. 34f]).

3.5.3 *Typdeklaration für Rückgabewerte*

Als schwach typisierte Sprache bot PHP bisher keine Möglichkeit der Deklaration von Typen für Rückgabewerte von Funktionen. Dies kann nun durch Angabe des Typs zwischen Funktionsdeklaration und dem

Code der Funktion geschehen, wie in Listing 3.11. Dadurch sollen unter anderem ungewollte Rückgabewerte verhindert werden, als auch die automatisierte Dokumentation von Funktionen vereinfacht werden. [Mor14b]

Listing 3.11: Typdeklaration für Rückgabewerte

```
<?php
public function foo(): int {
    return 42;
}
?>
```

3.6 ENTFERNTE ERWEITERUNGEN

Einige Funktionalitäten von PHP sind nicht in die Sprache selbst eingebaut, sondern werden durch externe Erweiterungen eingebunden, die jedoch standardmäßig mit PHP ausgeliefert werden. Diese stehen somit nicht unter der Verwaltung der PHP-Entwickler und werden unabhängig weiterentwickelt.

3.6.1 *mysql*

Die seit PHP 5.5 als veraltet erklärte Erweiterung *mysql* wird nicht mehr unterstützt. Dies wird mit Sicherheitsrisiken begründet. So unterstützt *mysql* beispielsweise keine **Prepared Statements**, welche einen wirksamen Schutz gegen **SQL Injections** bieten. [Orab] Zudem stehen mit *mysqli* und *PDO_MySQL* aktuellere Erweiterungen zur Verfügung.

3.6.2 *ereg*

Die Erweiterung *ereg* bietet verschiedene Funktionen für die Nutzung von **POSIX**-kompatiblen **Regulären Ausdrücken**. Die Erweiterung wurde zugungsten von *PCRE* entfernt, da diese unter anderem bessere Unterstützung von Unicode-Zeichen bietet und aktiv weiterentwickelt wird. [Pop14]

3.7 FAZIT

Die Ziele der Weiterentwicklung von PHP lassen sich folgendermaßen zusammenfassen:

- Erhöhung der Sicherheit
- Bessere Verständlichkeit des geschriebenen Quellcodes
- Höhere Ausführungsgeschwindigkeit

Die Erhöhung der Sicherheit wird hauptsächlich durch die Entfernung von Erweiterungen erreicht, die nicht weiterentwickelt werden. Entwickler werden dadurch gezwungen, diese mit Mitteln zu ersetzen, welche zum Einen auch zukünftig mit Sicherheitsupdates versorgt werden und zum Anderen Features bieten um den Quellcode zusätzlich zu härten. Auch die Erweiterung von Konzepten der Typsicherheit und die Ersetzung von als Sicherheitsrisiko geltenden Funktionen dient diesem Zweck.
Dem Ziel der besseren Verständlichkeit von Quellcode dienen beispielsweise die Vereinheitlichung von Variableninterpretation, die Abschaffung der doppelten Konzepte der Deklaration von Konstruktoren und die Anpassung der *list()*-Funktion an gewohnte Denkweisen.

4 UNTERSUCHUNG GEEIGNETER MITTEL

Wie in Kapitel 3 gezeigt, sind die Veränderungen zwischen PHP 5 und PHP 7 nicht nur sehr umfangreich, sondern erfordern auch große Eingriffe in den betroffenen Quellcode. **ISO/IEC 14764** sieht vor, die Migration zu definieren

4.1 ERKENNUNG DES ZU ÄNDERNDEN CODES

Um alten Code migrieren zu können, müssen alle Stellen gefunden werden, die in ihrer ursprünglichen Form in der neuen Umgebung nicht lauffähig sind. Dafür relevante Beispiele sind in Kapitel 3 gelistet, die gesamte Liste kann der Dokumentation entnommen werden. Die Erkennung kann je nach Umfang des Quellcodes und der verwendeten Funktionen entweder manuell oder automatisiert durchgeführt werden. Für beide Arten werden im folgenden Beispiele genannt und die jeweiligen Vor- und Nachteile diskutiert.

4.1.1 *Manuelle Erkennung*

Eine manuelle Erkennung des Codes bietet sich vor allem bei kleinen Softwareprojekten an, bei denen ein vollumfänglicher Überblick über den eingesetzten Code besteht. Hier kann durch die in typischen Editoren und Entwicklungsumgebungen integrierte Suche genutzt werden um alle Vorkommen von nicht lauffähigen Funktionen zu finden und diese anschließend einem Refactoring zu unterziehen. Besonders einfach gestaltet sich diese Methode bei entfernten Funktionen, beispielsweise die der Erweiterung *ereg*3.6.2. Diese kann der Entwickler in der Dokumentation nachschlagen und den Code auf etwaige Vorkommen prüfen. Schwierig wird die manuelle Erkennung bei Änderungen wie der Einhaltung des Standards in Switch-Anweisungen3.1.7. Hier ist eine Suche nur über umfangreiche Suchmuster (Reguläre Ausdrücke) möglich, die meist nicht trivial zu erstellen sind und viele Einzelfälle (z.B. verschachtelte Switch-Anweisungen) abdecken müssen. In diesen Fällen ist durch die manuelle Suche höchstens eine Eingrenzung des Problems möglich.

4.1.2 *Automatisierte Erkennung*

Da die zuvor besprochene manuelle Erkennung betroffenen Codes nur für einzelne Fälle oder kleine Projekte in Frage kommt, bietet sich als alternative die automatische Erkennung an, mit dem Ziel, dem

Entwickler einen vollumfänglichen Überblick der zu überarbeitenden Stellen im Code zu liefern. Im vorliegenden Fall wurde das Tool *php7mar*[1] des Entwicklers *Alexia* genutzt. *php7mar* erkennt mithilfe von **Regulären Audrücken**, **String-Matching** und **Lexikalischer Analyse** kritischen Code in Projekten und generiert daraus einen Bericht, bestehend aus Zeilenangaben, gefundenen Problemen und Lösungshinweisen. Ein Beispiel eines solchen Berichts findet sich in Listing4.1. In der Datei *GMCSS.php* werden drei Fehlerklassen gefunden: Erstens mehrere Fälle der Nutzung der veralteten Definition von Konstruktoren, zweitens einige Vorkommen der Entfernten Erweiterung *mysql*, sowie drittens ein indirekter Variablenzugriff, dessen Aussage unter PHP 7 möglicherweise eine andere ist (vgl. Kapitel 3.1.1). Insbesondere der erste Fall zeigt die Überlegenheit eines Analysetools, da solche Fehler nur schwer mit einer trivialen Suche zu finden sind. Trotzdem ist auch dieses Werkzeug nicht vollständig. Beispielsweise werden Fehler, die die Funktion *preg_replace* 3.3.1 betreffen, nicht gefunden. Diese Funktion wurde zum Zwecke der vollständigen Analyse nachgepflegt und auf *Github* veröffentlicht.[2]

Listing 4.1: Beispiel eines generierten Berichts mit *php7mar*

```
#### C:\Users\Nutzer\Documents\GitHub\gambio_tickets75\
    StyleEdit\classes\GMCSS.php
* oldClassConstructors
* Line 55:  'function GMCSS($p_css_file, $p_type='archive
    ')'
* Line 384: 'function GMCSSImport($p_css_file = false,
    $p_import_mode = '')'
* Line 791: 'function GMCSSExport($p_css_file)'
* Line 912: 'function GMCSSUpload($p_files, $p_type)'
* Line 982: 'function GMCSSArchive()'
* deprecatedFunctions
* Line 302: '$t_css_query = mysql_query('
* Line 311: 'if((int)mysql_num_rows($t_css_query) > 0)'
* Line 313: '$t_row_styles = mysql_fetch_array(
    $t_css_query, MYSQL_ASSOC);'
* Line 316: '$t_css_query = mysql_query('
* Line 325: 'if((int)mysql_num_rows($t_css_query) > 0)'
* variableInterpolation
* Line 359: 'global $$shippingModule; //notice $$'
```

4.2 REFACTORING

Refactoring beschreibt die Technik, bestehenden Code in seiner Struktur so zu verändern, dass dieser änderbar bleibt und wichtige Bestandteile leicht indentifizierbar bleiben, ohne jedoch den eigentlichen Sinn des Programms zu verändern. [Fow99] Bei der Migration von

1 Alexia. php7mar. URL: https://github.com/Alexia/php7mar
2 Martin Duschek. php7mar, URL: https://github.com/maddin333/php7mar

Software muss dem Refactoring ein hoher Stellenwert zugemessen werden, da große Bestandteile des Codes ausgetauscht und verändert werden, was nach Lehman [Leh80, S. 1060-1076] zur Degeneration von Software führt. Die Schwierigkeit des Refactorings liegt häufig im zeitaufwändigen Verstehen des vorliegenden Codes. Diese Zeit muss jedoch für die Migration der Software ohnehin aufgewendet werden, womit sich ein Refactoring hier durchaus anbietet. Die Ziele des Refatorings können durch verschiedenste Techniken erreicht werden, beispielsweise die Zerlegung von Funktionen in kleinere, bessere verständliche Funktionen oder die Auslagerung von immer wieder genutztem Code in eine eigene Funktion.

4.2.1 *Extrahieren*

Bei der Extraktion von Funktionen sollen häufig genutzte Programmteile in eine eigene, wiederverwendbare Funktion ausgelagert werden. Dazu wird die neue Funktion angelegt,

4.2.2 *Zerlegung von Funktionen*

4.2.3 *Unit-Tests*

4.3 LAUFFÄHIGKEIT HISTORISCHEN CODES

Der Standard ISO/IEC 14764 sieht auch nach der erfolgten Migration eine Unterstützung der alten Umgebung vor. Dies ist natürlich vor allem bei Produkten sinnvoll, die von Dritten eingesetzt werden, die dadurch vor nicht erwartetem Verhalten der Software geschützt werden. Allerdings ist eine Unterstützung der alten Umgebung auch bei unternehmenseigener Software sinnvoll. Beispielsweise lassen sich Fehler im Programm zurückverfolgen, Änderungen nachvollziehen und sichergestellt werden, dass im Falle eines Fehlers bei der Migration eine lauffähige Version zur Verfügung steht. Daraus ergeben sich jedoch einige Probleme. So muss nicht nur der Quellcode von alten Versionen eines Programms zur Verfügung stehen, sondern auch die verschiedenen Umgebungen um die Software ausführen zu können.

4.3.1 *Versionsverwaltung*

Grundlage für die Unterstützung alter Versionen einer Software ist, dass diese Versionen in ihrem ausgelieferten Zustand vorhanden sind und Änderungen, die seitdem vorgenommen wurden protokolliert und nachvollziehbar sind. Ein einfacher Ansatz dazu wäre beispielsweise die Ablage des Programmcodes in einem eindeutig benannten Ordner, nachdem eine Änderung durchgeführt wurde, sowie der Ein-

satz eines Programms wie beispielsweise **Diff** [3] zur Kenntlichmachung von Änderungen zwischen zwei Dateien. Diese Vorgehensweise kann zwar für kleinere Projekte genügen, ist jedoch für große Projekte mit mehreren hundert einzelnen Dateien und beliebig vielen bearbeitenden Personen nicht geeignet. Dieses Problem kann durch den Einsatz einer dedizierten Software zur Versionsverwaltung gelöst werden. Diese Software beinhaltet den gesamten Quellcode des Projekts in einem **Repository**, stellt diesen den Nutzern bereit und protokolliert jede Änderung, sogenannte **Commits**. Durch diese Protokollierung kann der Quellcode in jeden beliebigen früheren Zustand zurückversetzt werden, und somit auch weiter gewartet werden. Bekannte Programme zur Versionsverwaltung mit den genannten Funktionen sind zum Beispiel *Git*[4] oder *Mercurial*[5]

4.3.2 *Ausführungsumgebung*

Zu jeder historischen Version der Software muss die passende Ausführungsumgebung zur Verfügung stehen. Eine triviale Lösung ist die lokale Installation verschiedener Umgebungen auf dem relevanten System. Einen weiteren Lösungsansatz bietet die Virtualisierung der Ausführungsumgebung mittels Containern. Die Vor- und Nachteile beider Ansätze werden im folgenden Diskutiert.

4.3.2.1 *Lokale Ausführungsumgebung*

Der triviale Lösungsansatz, verschiedene Ausführungsumgebungen bereitzustellen ist, alle benötigten Versionen lokal auf dem relevanten System (beispielsweise dem Computer des Entwicklers) zu installieren. Dies hat den Vorteil, dass keine zusätzliche Software benötigt wird und die volle Geschwindigkeit des Systems zur Ausführung bereitsteht. Allerdings hat diese Variante den Nachteil, dass der Nutzer (im Besipiel der Entwickler) selbst dafür verantwortlich ist, die passende Ausführungsumgebung bereitzustellen und sicherstellen muss, dass diese richtig konfiguriert sind.

4.3.2.2 *Continuous Integration mittels Containern*

Container sind eine Art der Virtualisierung von Betriebssystemen, die schon seit einigen Jahren, beispielsweise seit 2008 durch *LXC*[6] zum Einsatz kommt. Container unterscheiden sich von herkömmlichen virtuellen Maschinen darin, dass diese sich den Kernel des Host-Betriebssystems teilen und dadurch deutlich leichtgewichtiger und portabler sind [Sch14]. Docker, eine populäre Software zur Con-

3 GNU Diffutils, URL: http://www.gnu.org/software/diffutils/
4 Git, https://git-scm.com/
5 Mercurial, URL: https://www.mercurial-scm.org/
6 Linux Containers, https://linuxcontainers.org/

tainervirtualisierung, bietet neben der Unterstützung von Windows als Host- Betriebssystem auch eine Plattform zum Austausch von fertig konfigurierten Conatainern an und stellt zudem einfache Mechanismen zur Anpassung bereit. So können Container sehr einfach über ein sogenanntes *Dockerfile* konfiguriert werden. Dieses enthält alle Informationen über den Container, beispielsweise das gewünschte Betriebssystem und die installierten Applikationen mitsamt deren Konfigurationen [And15]. Diese Datei kann im Repository platziert werden, wodurch jeder Entwickler eine exakt gleich konfigurierte Ausführungsumgebung für die Software starten kann. Durch die Konfiguration über das *Dockerfile*, eine einfache Textdatei, und deren Speicherung im Repository ergibt sich auch der Vorteil, dass für jede Version der Software eine angepasste Ausführungsumgebung bereitgestellt werden kann. Nachteilig ist bei dieser Variante der Bereitstellung der Ausführungsumgebung, dass die initiale Konfiguration des Containers nicht nur die Konfiguration der Ausführungsumgebung (etwa des Webservers) betrifft, sondern auch die des Betriebssystems sowie der Virtualisierungssoftware.

5 MIGRATION DES TICKETS75 ONLINESHOPS

Dieses Kapitel beschreibt die Migration des Onlineshops der Firma *tickets75* von PHP 5.6 zu PHP 7.3, sowie die genutzten Werkzeuge und Techniken. Die Codebasis für das Projekt umfasst 5732 einzelne PHP-Dateien, bestehend aus 596.198 Zeilen Code. Diese Menge verdeutlicht, dass entsprechende Werkzeuge zur Automatisierung nötig sind, da kein Überblick über die Gesamtheit der Codebasis bestehen kann.

5.1 ANFORDERUNGSANALYSE

Die Anforderungsanalyse ist der wichtigste Schritt zur Vorbereitung der Migration. Durch sie kann eine Abschätzung getroffen werden, wie schwer und welche Teile des Codes von der Migration betroffen sind. Dies ist vor allem wichtig, hinsichtlich der Wirtschaftlichkeit einer Migration. Ist der Anteil zu migrierenden Codes zu hoch, kann es sinnvoll sein, eine Software komlett neu zu schreiben. Im vorliegenden Fall wurde die Analyse mit dem Programm *php7mar* durchgeführt. Die Ergebnisse, dargestellt in Tabelle 5.1, zeigen, dass zwar über 10% der Dateien unter PHP 7 Fehler enthalten, gemessen an den betroffenen Codezeilen aber nur ein kleiner Teil (0,24%) des Codes migiriert werden muss. Bei dieser Zahl ist zu beachten, dass Zeilen mit mehreren Eine granularere Analyse erlaubt das angehängte Python-Skript, welches die tatsächlich genutzten Funktionen aus dem Report herausfiltert. Die Ergebnisse in Tabelle 5.2 zeigen, dass die Datenbankverbindung mittels der Erweiterung *mysql* der größte Schwerpunkt der Migration darstellt. Ähnlich große Verbreitung haben implizite Konstruktoren. Die Fehlerklassen *Arraywert per Referenz* sowie *foreach per Referenz* wird von *php7mar* zwar ausgewiesen, stehen jedoch nur bedingt in Zusammenhang mit der Migration zu PHP 7. Diese Konstrukte sind keine Fehler, werden aufgrund ihrer Implikationen (beispielsweise des empfohlenen Zurücksetzen des Arrayzeigers nach der Benutzung von foreach mit Referenz) von einigen Entwicklern nicht zur Benutzung empfohlen[1].

5.2 ENTWICKLUNG VON WERKZEUGEN ZUR DURCHFÜHRUNG DER MIGRATION

Für die Durchführung der Migration, im speziellen die Anpassung der Software an die neue Umgebung, wurde ein Docker-Container erstellt,

1 Johannes Schlüter, 'References and foreach', https://schlueters.de/blog/archives/141-references-and-foreach.html (besucht am 30.10.2019)

Tabelle 5.1: Anteil zu migrierender Codeteile an der gesamten Codebasis

	Gesamt	Betroffen	Anteil
Dateien	5732	690	12,04%
Codezeilen	596198	1431	0,24%

Tabelle 5.2: Vorkommen zu migrierender Funktionen in der Codebasis

	Anzahl der Vorkommen
mysql	722
Implizite Konstrukoren	453
foreach per Referenz	21
Magic Quotes	20
Indirekte Variablenzugriffe	20
Konstruktoraufruf per Referenz	6
preg_replace mit Option /e	5
Doppelte Funktionsparameter	3
Arraywert per Referenz	1

welcher eine einheitliche Konfiguration der Auführungsumgebung bereitstellt. Dazu wurde auf zwei vorgefertigte Images zurückgegriffen, eine PHP-Installation in Version 7.2 und eine *MySQL*-Datenbank um eine lokale Kopie der Datenbank für Tests bereitzustellen. Zudem wurde zur Unterstützung der Entwicklung die Erweiterung *Xdebug*[2] installiert und konfiguriert, wodurch Entwicklern auch bei einer serverseitigen Ausführung des Codes eine Schnittstelle für den Debugger der Entwicklungsumgebung bereitgestellt wird.

5.3 ENTWICKLUNG DER AN DIE NEUE UMGEBUNG ANGEPASSTEN SOFTWARE

5.3.1 *Ersetzen der Erweiterung mysql*

Da die Erweiterung *mysql* für Datenbankverbindungen schon seit PHP 5.5 veraltet ist, stehen Entwicklern seitdem auch Alternativen zur Verfügung. Mit *mysqli* und *PDO_MySQL* werden zwei Application Programming Interfaces (APIs) bereitgestellt, deren Funktionalitäten weit über die der alten Erweiterung hinausgehen. Die Entscheidung zwischen beiden Möglichkeiten fiel aufgrund zweier Faktoren auf *mysqli*: Zum einen die syntaktische Ähnlichkeit zwischen *mysql* und *mysqli*, da bei vielen Funktionen nur der *mysql*-Präfix geändert, sowie eine explizite Datenbankverbindung angegeben werden muss. Zum anderen bietet *mysqli* sowohl ein objektorientiertes, als auh ein

2 Xdebug, https://xdebug.org/

prozedurales Interface, *PDO_MySQL* ist dagegen nur objektorientiert nutzbar. Da *mysql* rein prozedural genutzt werden konnte, bietet es sich an, dies voerst beizubehalten und bei Bedarf auf einen objektorientierten Ansatz zu wechseln.

5.3.2 *Ersetzen impliziter Konstruktoren*

Die Ersetzung impliziter Konstruktoren (vgl. Kapitel 3.2.1 im Quellcode stellt sich im Grunde trivial dar, da der Funktionsname des Konstruktors direkt durch das Schlüsselwort *__construct* ersetzt werden kann. Dadurch bleiben alle Konstruktoraufrufe durch *new* funktionsfähig. Allerdings ist dabei zu beachten, dass die betreffende Konstruktorfunktion auch direkt über ihren Namen aufgerufen werden kann. Deshalb ist es ratsam, die ursprüngliche Konstruktorfunktion unverändert zu lassen und diese über den neuen Konstruktor aufzurufen. Funktionsparameter müssen entsprechend vom neuen Konstruktor an die Funktion übergeben werden. Das Beispiel 5.1 zeigt diesen Workaround, der zu maximal möglicher Kompatibilität führt, ohne jeden Aufruf der Funktion zu überprüfen. Dabei wurde der ursprüngliche Konstruktor unverändert belassen, die Funktion *__construct* wurde hinzugefügt. Diese nimmt den Parameter *$order_id* entgegen und ruft die Funktion *order* mit diesem auf. Dadurch bleibt *order* innerhalb der Klasse aufrufbar.

Listing 5.1: Beispiel der Ersetzung impliziter Konstruktoren

```
<?php
class order {
    function __construct($order_id) {
        $this->$order($order_id);
    }

    function order($order_id) {
        print($order_id);
    }
}
?>
```

5.3.3

5.3.4 *Entfernen aller Aufrufe von Magic Quotes*

Wie in Kapitel 3.1.5 ausgeführt, haben Aufrufe der Funktionen *magic_quotes_runtime* sowie *set_magic_quotes_runtime* seit PHP 5.4 keinen Effekt mehr. Daher können diese bei einer Migration von PHP 5.6 gefahrlos entfernt werden. Trotzdem ist darauf zu achten, dass Inhalte aus externen Quellen mit angemessenen Funktionen maskiert werden, um Sicherheitsrisiken zu vermeiden. Die Erweiterung *mysqli* bietet

dafür beispielsweise die Funktion *mysqli_real_escape_string*, welche Sonderzeichen unter Berücksichtigung des eingestellten Zeichensatzes maskiert.

5.3.5 *Korrektur indirekter Variablenzugriffe*

Listing 5.2: Anpassung indirekter Variablenzugriffe

```
<?php
if (isset ($$_SESSION['payment']->form_action_url)) {
    $form_action_url = $$_SESSION['payment']->
        form_action_url;
}

if (isset (${$_SESSION}['payment']->form_action_url)) {
    $form_action_url = ${$_SESSION}['payment']->
        form_action_url;
}
?>
```

5.3.6 *Ersetzen von Konstruktoraufrufen per Referenz*

Die Ersetzung von Konstruktoraufrufen, die per Referenz zugewiesen werden ist trivial. Es genügt, bei allen Vorkommen die Referenz zu entfernen und durch einen Zuweisungsoperator zu ersetzen, da der Operator *new* seit PHP 5 automatisch eine Referenz zurückgibt. Die Ersetzung wird im Beispiel 5.3 deutlich. Der auf den Konstruktoraufruf folgende Code kann unverändert mit dem Objekt arbeiten.

Listing 5.3: Beispiel der Ersetzung von Konstruktoraufrufen per Referenz

```
<?php
$rv =& new $class($this); //alter Konstruktoraufruf per
    Refrenz
$rv = new $class($this); //neuer Konstruktoraufruf ohne
    Referenz

if(is_subclass_of($rv, 'IclearBase')) {
    $rv->icError =& $this->getObject('IclearError', true)
        ;
    }
?>
```

5.3.7 *Ersetzen von preg_replace mit Option /e*

Die gleiche Funktionsweise wie *preg_replace* mit der Option */e* bietet PHP seit der Version 4 mit der Funktion *preg_replace_callback*. Diese Funktion erwartet neben dem regulären Ausdruck und dem zu prüfenden String eine Callback-Funktion, welche die Treffer des regulären

Ausdrucks verarbeitet und eine Ersetzung zurückgibt. Beispiel 5.4 zeigt, wie *preg_replace_callback* die Funktion *preg_replace* unter Verwendung einer anonymen Funktion ersetzen kann. Die erste Zeile zeigt die ursprüngliche Variante, darunter ist die Ersetzung zu sehen. Diese definiert eine anonyme Funktion, die wiederrum die Funktion *removeEvilAttributes* aufruft. Diese Verkettung geschieht, um maximale Kompatibilität zu gewährleisten, da *removeEvilAttributes* einen String als Argument erwartet, beim direkten Aufruf über *preg_relace_callback* ein Array übergeben bekommen würde. Um die Funktion nicht ändern zu müssen und somit möglicherweise Inkompatibilitäten zu schaffen, wurde dieser Workaround gewählt.

Listing 5.4: Beispiel der Nutzung von preg_replace_callback

```
<?php
preg_replace('/<(.*?)>/ie', "'<'.removeEvilAttributes
    ('\\1').'>'", $source);

preg_replace_callback(
    '/<(.*?)>/i',
function($matches){
    return '<'.removeEvilAttributes($matches[1]).'>';
    },
    $source);
?>
```

5.3.8 *Entfernen doppelter Funktionsparameter*

Wie in Kapitel 3.4 beschrieben, führt die Angabe mehrerer Funktionsparameter mit dem selben Namen unter PHP 7 zu einem Fehler. Um diesen zu korrigieren reicht es nicht aus, die überflüssigen Parameter zu löschen. Vielmehr müssen alle Aufrufe der Funktion überprüft werden, um entscheiden zu können, welches Vorgehen sinnvoll ist. Sind die Parameter wie in Aufruf 1 im Beispiel 5.5 mit der gleichen Variable belegt, genügt das Löschen eines Parameters in der Funktion und im Aufruf. Sind die Parameter jedoch unterschiedlich belegt, darf, um die Funktionalität zu erhalten, nur der letzte Parameter bestehen bleiben, da dessen Wert innerhalb der Funktion zum Tragen kommt.

Listing 5.5: Beispiel der Nutzung von preg_replace_callback

```
<?php
function foo($bar, $bar) {
    echo($bar);
    }

    //Aufruf 1
    foo($x,$x);
    //Aufruf 2
    foo($x,$y);
```

```
?>
```

5.4 DURCHFÜHRUNG DER MIGRATION

Für die Ausführung der im vorangegangenen Kapitel entwickelten Software wird eine neue Ausführungsumgebung mit PHP 7 benötigt. Dazu wurden zwei neue Server angemietet, um die alte Infrastruktur nachzubilden und beide Systeme für Tests an der neuen Umgebung parallel zueinander zu betreiben. Auf den neuen Servern, die identische Konfigurationen zu den bisher zum Einsatz gekommenen Geräten aufweisen, wurde PHP 7.2 als Ausführungsumgebung installiert. Als Software für den Webserver wird Apache[3] eingesetzt. Nach erfolgreichen Tests konnte die alte Infrastruktur durch die neue ersetzt werden, die migrierte Software also produktiv eingesetzt werden. Durch den vorgeschalteten Load-Balancer konnte jeweils ein Server aus dem Produktivbetrieb entfernt werden und durch den neuen ersetz werden. Dadurch konnte die Webseite mit geringer Ausfallzeit migiriert werden und im Falle eines Fehlers hätte schnell auf die alte Infrastruktur zurückgegriffen werden können.

5.5 VERIFIKATION DER MIGRATION

Zur Verifikation der Migration wurden zum einen eigene Tests durchgeführt, vor allem verschiedene Testkäufe um die Funktion für Kunden des Onlineshops zu verifizieren, sowie administrative Tätigkeiten wie die Anpassung der Startseite und das Hinzufügen von Produkten zum Katalog. Zum Anderen wurde die Funktion der Webseite durch das Monitoring-Tool *New Relic*[4] überwacht. Dieses Werkzeug überwacht die Performance der Applikation und meldet Fehler in der Ausführung.

5.6 SUPPORT DER ALTEN UMGEBUNG

Diese Anforderung im Standard **ISO/IEC 14764** betrifft vor allem Software, die an Kunden ausgeliefert wird und dort möglicherweise auch nach der Migration zum Einsatz kommt und weiter gepflegt werden soll. Für den vorliegenden Fall ist dies nur relevant, falls signifikante Fehler in der migrierten Software auftreten, die einen weiteren Einsatz unmöglich machen und den erneuten Einsatz der alten Software und Ausführungsumgebung erfordern. Zu diesem Zweck wurde für eine Übergangszeit von einem Monat die alte Infrastruktur weiter in Betrieb gehalten, bevor diese abgeschaltet wurde. Der ursprüngliche Code, der nicht unter PHP 7 lauffähig ist, steht mitsamt einer lauffä-

3 Apache HTTP Server Project, https://httpd.apache.org/
4 New Relic, https://newrelic.de

higen Ausführungsumgebung, wie in Kapitel 4.3 beschrieben, weiter über die Versionsverwaltung zur Verfügung.

6 AUSWERTUNG

Die vorangegangenen Kapitel zeigen, dass die Migration einer komplexen Software keineswegs trivial ist. Gleichzeitig vereinfachen verschiedene Technologien die sowohl die Arbeit, als auch die Beachtung des Standards **ISO/IEC 14764**. Die Anforderungsanalyse in Kapitel 5.1 zeigt, dass eine manuelle Erkennung des zu ändernden Codes aufgrund der Menge nahezu unmöglich ist und nur mit technischen Hilfsmitteln wie dem in Kapitel 4.1.2 vorgestellten *php7mar* zu überblicken ist. Dazu ist jedoch auch zu erwähnen, dass diese Hilfsmittel unter Umständen fehlerhaft sind oder nicht alle Bereiche der Migration abdecken und dementsprechend erst (weiter-) entwickelt werden müssen. Die in **ISO/IEC 14764** geforderte Unterstützung des historischen Codes kann zwar auf klassischem Wege gelöst werden, stellt sich jedoch, wie in Kapitel 4.3 dargelegt, für umfangreiche Projekte nur unter Verwendung von Virtualisierung als sinnvoll dar. Die in Kapitel 5.3 gezeigten Änderungen an der Software zeigen, wie wichtig die aufgezeigten Techniken des Refactorings für die Migration einer Software sind. Zudem zeigen diese Änderungen, dass zwei der in Kapitel 3 herausgearbeiteteten Ziele des neuen Major-Release von PHP erreicht wurden:
So zeigen beispielsweise die Kapitel 5.3.5 und 5.3.2, dass Sonderfälle reduziert wurden und Sprachkonstrukte in PHP vereinheitlicht wurden. Das Ziel der erhöhten Sicherheit von Software wird über die Entfernung veralteter Erweiterungen (vgl. Kapitel 5.3.1) sowie potentiell gefährlicher Optionen (vgl. Kapitel 5.3.7) erreicht. Das letzte Ziel, die Erhöhung der Ausführungsgeschwindigkeit, lässt sich über das in Kapitel 5.5 vorgestellte Tool *New Relic* überprüfen. Eine Auswertung der 24 Stunden vor, sowie nach der Migration (dargstellt in Abbildung 6.1) zeigt, dass die durchschnittliche Ausführungszeit von Anfragen an den Onlineshop von 66,05ms auf 43,38ms und somit um ca. 34% zurückgegangen ist.

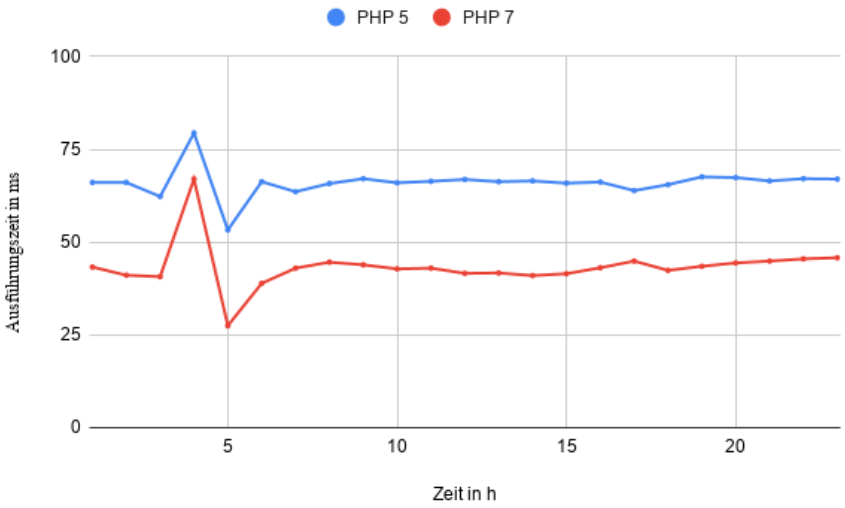

Abbildung 6.1: Ausführungszeiten von Anfragen vor und nach der Migration

7

SCHLUSSBETRACHTUNGEN

[And15] C. Anderson. „Docker [Software engineering]“. In: *IEEE Software* 32.3 (Mai 2015), S. 102–c3. ISSN: 0740-7459. DOI: 10.1109/MS.2015.62.

[Car19] Pierre Carbonnelle. *PYPL PopularitY of Programming Language index*. PYPL PopularitY of Programming Language index. Okt. 2019. URL: http://pypl.github.io/PYPL.html (besucht am 22. 10. 2019).

[Fow99] Martin Fowler. „Refactoring - Improving the Design of Existing Code“. In: (1999). Unter Mitarb. von Kent Beck und John Brant, S. 337.

[Inta] International Electrotechnical Commission. *IEC - About the IEC*. About the IEC. URL: https://www.iec.ch/about/?ref=menu (besucht am 05. 10. 2019).

[Intb] International Organization for Standardization: *About ISO*. About ISO. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/home/about-us.html (besucht am 05. 10. 2019).

[Leh80] Meir M. Lehman. *Programs, Life Cycles, and Laws of Software Evolution*. 1980.

[Mar12] Robert C. Martin. *Clean code: a handbook of agile software craftsmanship /*. [Repr.] Robert C. Martin series. Upper Saddle River, NJ: : Prentice Hall, 2012. xxix+431. ISBN: 978-0-13-235088-4.

[Mor14a] Levi Morrison. *PHP: rfc:remove_php4_constructors*. 17. Nov. 2014. URL: https://wiki.php.net/rfc/remove_php4_constructors (besucht am 30. 09. 2019).

[Mor14b] Levi Morrison. *PHP: rfc:return_types*. 20. März 2014. URL: https://wiki.php.net/rfc/return_types (besucht am 02. 10. 2019).

[Oraa] Oracle. *Anonymous Classes (The Java™ Tutorials > Learning the Java Language > Classes and Objects)*. Anonymous Classes (The Java™ Tutorials). URL: https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html (besucht am 02. 10. 2019).

[Orab] Oracle. *MySQL :: MySQL 8.0 Reference Manual :: 13.5 Prepared SQL Statement Syntax*. URL: https://dev.mysql.com/doc/refman/8.0/en/sql-syntax-prepared-statements.html (besucht am 02. 10. 2019).

[Ora04] Oracle. *How and When to Deprecate APIs*. 2004. URL: https://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html (besucht am 30. 09. 2019).

[PHPa] PHP Group. *PHP: History of PHP - Manual*. History of PHP. URL: https://www.php.net/manual/en/history.php.php (besucht am 21. 10. 2019).

[PHPb] PHP Group. *PHP Language Specification*. GitHub. Unter Mitarb. von nikic, smalyshev, zhujinxuan und mousetraps. URL: https://github.com/php/php-langspec/blob/master/spec/11-statements.md#the-switch-statement (besucht am 04. 10. 2019).

[PHPc] PHP Group. *PHP: Objects and references - Manual*. Objects and references. URL: https://www.php.net/manual/en/language.oop5.references.php (besucht am 29. 10. 2019).

[Pop14] Nikita Popov. *PHP: rfc:remove_deprecated_functionality_in_php7*. 11. Sep. 2014. URL: https://wiki.php.net/rfc/remove_deprecated_functionality_in_php7 (besucht am 03. 10. 2019).

[PW] Tom Preston-Werner. *Semantic Versioning 2.0.0*. Semantic Versioning. URL: https://semver.org/ (besucht am 21. 10. 2019).

[Sch14] Thijs Scheepers. „Virtualization and Containerization of Application Infrastructure: A Comparison“. In: (2014), S. 7.

[nik14] nikic. *Remove string category support in setlocale()*. GitHub. 10. Sep. 2014. URL: https://github.com/php/php-src/commit/4c115b6b71e31a289d84f72f8664943497b9ee31#diff-b31234a9f5a03a328b60d0453988140f (besucht am 01. 10. 2019).