

sebastian BERGMANN
stefan PRIEBSCH



SOFTWAREQUALITÄT IN PHP PROJEKTEN

Digg // eZ Components // studiVZ // swoodo //
symfony // TYPO3 // Zend Framework



HANSER

Bergmann, Priebisch



**Softwarequalität
in PHP-Projekten**



Bleiben Sie einfach auf dem Laufenden:
www.hanser.de/newsletter

Sofort anmelden und Monat für Monat
die neuesten Infos und Updates erhalten.

Sebastian Bergmann
Stefan Pribsch



Softwarequalität in PHP-Projekten

HANSER

Die Autoren:

Sebastian Bergmann & Stefan Priebsch, thePHP.cc, Wolfratshausen

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht. Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im
Internet über <http://dnb.d-nb.de> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2010 Carl Hanser Verlag München
Gesamtlektorat: Fernando Schneider
Sprachlektorat: Sandra Gottmann, Münster-Nienberge
Herstellung: Thomas Gerhardy
Coverconcept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisierung: Stephan Rönigk
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-41923-0

www.hanser.de/computer

Inhaltsverzeichnis

I Grundlagen	1
1 Software-Qualität	3
1.1 Was ist Software-Qualität?	3
1.1.1 Externe Qualität	4
1.1.2 Interne Qualität	5
1.2 Technical Debt	6
1.3 Konstruktive Qualitätssicherung	8
1.4 Sauberer Code	9
1.4.1 Explizite und minimale Abhängigkeiten	9
1.4.2 Klare Verantwortlichkeiten	10
1.4.3 Keine Duplikation	10
1.4.4 Kurze Methoden mit wenigen Ausführungszweigen	10
1.5 Software-Metriken	10
1.5.1 Zyklomatische Komplexität und NPath-Komplexität	11
1.5.2 Change Risk Anti-Patterns (CRAP) Index	11
1.5.3 Non-Mockable Total Recursive Cyclomatic Complexity	12
1.5.4 Global Mutable State	12
1.5.5 Kohäsion und Kopplung	13
1.6 Werkzeuge	13
1.7 Fazit	16
2 Testen von Software	17
2.1 Black-Box- und White-Box-Tests	17
2.2 Wie viele Tests braucht man?	18
2.3 Systemtests	20
2.3.1 Testen im Browser	20

2.3.2	Automatisierte Tests	21
2.3.3	Testisolation	23
2.3.4	Akzeptanztests	24
2.3.5	Grenzen von Systemtests	25
2.4	Unit-Tests	25
2.4.1	Rückgabewerte	28
2.4.2	Abhängigkeiten	30
2.4.3	Seiteneffekte	31
2.5	Praxisbeispiel	32
2.5.1	Den zu testenden Code analysieren	36
2.5.2	Eine Testumgebung aufbauen	37
2.5.3	Globale Abhängigkeiten vermeiden	40
2.5.4	Unabhängig von Datenquellen testen	41
2.5.5	Asynchrone Vorgänge testen	49
2.5.6	Änderungen in der Datenbank speichern	54
2.5.7	Nicht vorhersagbare Ergebnisse	56
2.5.8	Eingabedaten kapseln	59
2.5.9	Weiterführende Überlegungen	61
2.6	Fazit	62

II Best Practices 63

3	TYPO3: die agile Zukunft eines schwergewichtigen Projekts	65
3.1	Einführung	65
3.1.1	Die Geschichte von TYPO3 – 13 Jahre in 13 Absätzen	65
3.1.2	Den Neuanfang wagen!	67
3.1.3	Unsere Erfahrungen mit dem Testen	68
3.2	Grundsätze und Techniken	69
3.2.1	Bittersüße Elefantenstückchen	70
3.2.2	Testgetriebene Entwicklung	71
3.2.3	Tests als Dokumentation	72
3.2.4	Kontinuierliche Integration	73
3.2.5	Sauberer Code	74
3.2.6	Refaktorisierung	76
3.2.7	Programmierrichtlinien	77

3.2.8	Domänengetriebenes Design	78
3.3	Vorgehen bei der Entwicklung	79
3.3.1	Neuen Code entwickeln	79
3.3.2	Code erweitern und ändern	80
3.3.3	Code optimieren	81
3.3.4	Fehler finden und beheben	83
3.3.5	Alten Code fachgerecht entsorgen	83
3.4	Testrezepte	84
3.4.1	Ungewollt funktionale Unit-Tests	84
3.4.2	Zugriffe auf das Dateisystem	85
3.4.3	Konstruktoren in Interfaces	86
3.4.4	Abstrakte Klassen testen	87
3.4.5	Testen von geschützten Methoden	88
3.4.6	Verwendung von Callbacks	91
3.5	Auf in die Zukunft	92
4	Bad Practices in Unit-Tests	95
4.1	Einführung	95
4.2	Warum guter Testcode wichtig ist	95
4.3	Bad Practices und Test-Smells	96
4.3.1	Duplizierter Testcode	97
4.3.2	Zusicherungsroulette und begierige Tests	99
4.3.3	Fragile Tests	102
4.3.4	Obskure Tests	104
4.3.5	Lügende Tests	112
4.3.6	Langsame Tests	113
4.3.7	Konditionale Logik in Tests	114
4.3.8	Selbstvalidierende Tests	116
4.3.9	Websurfende Tests	117
4.3.10	Mock-Overkill	119
4.3.11	Skip-Epidemie	120
4.4	Fazit	121
5	Qualitätssicherung bei Digg	123
5.1	Die Ausgangssituation	123
5.1.1	Unsere Probleme	123

5.1.2	Code-Altlasten	124
5.1.3	Wie lösen wir unsere Probleme?	126
5.1.4	Ein Test-Framework wählen	128
5.1.5	Mit einem Experten arbeiten	128
5.2	Das Team trainieren	129
5.3	Testbaren Code schreiben	133
5.3.1	Statische Methoden vermeiden	133
5.3.2	Dependency Injection	136
5.4	Mock-Objekte	137
5.4.1	Überblick	137
5.4.2	Datenbank	137
5.4.3	Lose gekoppelte Abhängigkeiten	138
5.4.4	Beobachter für klasseninternes Verhalten	139
5.4.5	Memcache	141
5.4.6	Mocken einer serviceorientierten Architektur	142
5.5	Der Qualitätssicherungsprozess bei Digg	147
5.5.1	Testen	147
5.5.2	Vorteile	149
5.5.3	Herausforderungen	151
5.6	Fazit	152

III Server und Services 153

6	Testen von serviceorientierten APIs	155
6.1	Die Probleme	157
6.2	API-Zugangskennungen	158
6.3	API-Beschränkungen	163
6.4	Service-Protokolle offline testen	164
6.5	Konkrete Services offline testen	169
6.6	Fazit	175
7	Wie man einen WebDAV-Server testet	177
7.1	Über die eZ WebDAV-Komponente	177
7.1.1	WebDAV	177
7.1.2	Architektur	180

7.2	Herausforderungen bei der Entwicklung	182
7.2.1	Anforderungsanalyse	182
7.2.2	TDD nach RFC	183
7.2.3	Den Server testen	184
7.3	Automatisierte Akzeptanztests mit PHPUnit	186
7.3.1	Test-Trails aufzeichnen	188
7.3.2	Das Testrezept	190
7.3.3	Integration mit PHPUnit	192
7.4	Fazit	201
 IV Architektur		 203
8	Testen von Symfony und Symfony-Projekten	205
8.1	Einführung	205
8.2	Ein Framework testen	206
8.2.1	Der Release-Management-Prozess von Symfony	206
8.2.2	Verhältnis von Testcode und getestetem Code	208
8.2.3	Die Ausführung der Testsuite muss schnell sein	208
8.2.4	Gesammelte Erfahrungen	209
8.3	Testen von Webanwendungen	215
8.3.1	Die Hemmschwelle für das Testen abbauen	215
8.3.2	Unit-Tests	216
8.3.3	Funktionale Tests	222
8.4	Fazit	227
9	Testen von Grafikausgaben	229
9.1	Einführung	229
9.2	Entwicklungsphilosophie	230
9.3	Die ezcGraph-Komponente	230
9.3.1	Architektur	232
9.3.2	Anforderungen an die Tests	233
9.4	Ausgabetreiber durch Mock-Objekt ersetzen	234
9.4.1	Mehrfache Erwartungen	236
9.4.2	Structs	238
9.4.3	Generierung der Erwartungen	238

9.4.4	Zusammenfassung	239
9.5	Binäre Ausgaben testen	239
9.5.1	Die Ausgabetreiber	240
9.5.2	Generierung der Erwartungen	241
9.5.3	SVG	241
9.5.4	Bitmap-Erzeugung	243
9.5.5	Flash	246
9.6	Fazit	249
10	Testen von Datenbank-Interaktionen	251
10.1	Einführung	251
10.2	Pro und Kontra	252
10.2.1	Was gegen Datenbanktests spricht	252
10.2.2	Warum wir Datenbanktests schreiben sollten	253
10.3	Was wir testen sollten	254
10.4	Datenbanktests schreiben	256
10.4.1	Die Datenbankverbindung mocken	256
10.4.2	Die Datenbankeerweiterung von PHPUnit	257
10.4.3	Die Klasse für Datenbanktestfälle	258
10.4.4	Die Verbindung zur Testdatenbank aufbauen	259
10.4.5	Datenbestände erzeugen	263
10.4.6	Operationen auf den Daten	280
10.4.7	Tests schreiben	283
10.4.8	Den Datenbanktester benutzen	292
10.5	Testgetriebene Entwicklung und Datenbanktests	294
10.6	Datenbanktests als Regressionstests	295
10.6.1	Probleme mit den Daten testen	296
10.6.2	Probleme testen, die durch Daten sichtbar werden	297
10.7	Zusammenfassung	298
V	QA im Großen	299
11	Qualitätssicherung bei studiVZ	301
11.1	Einführung	301
11.2	Akzeptanztests	303

11.3 Selenium	305
11.3.1 Die Selenium-Erweiterung von PHPUnit	307
11.4 Technisches Setup von studiVZ	308
11.4.1 Codeumgebung	308
11.4.2 Testumgebung	309
11.5 Best Practices	310
11.5.1 Jugendsünden	310
11.5.2 Strategiewechsel	313
11.6 Eine DSL muss her	326
11.6.1 Interne DSL	327
11.6.2 Testing_SeleniumDSL 1.0	327
11.6.3 Testing_SeleniumDSL 2.0 – ein Entwurf	330
11.7 Fazit	331
12 Kontinuierliche Integration	335
12.1 Einführung	335
12.1.1 Kontinuierliche Integration	337
12.1.2 Statische Analyse	339
12.2 Installation und Inbetriebnahme	353
12.3 Konfiguration	353
12.3.1 Statische Tests	356
12.3.2 Dynamische Tests	363
12.3.3 Reporting	363
12.3.4 Deliverables erzeugen	365
12.4 Betrieb	366
12.5 Weiterführende Themen	367
12.5.1 Continuous Deployment	367
12.5.2 Einen Reverse Proxy nutzen	369
12.5.3 Kontinuierliche Integration und agile Paradigmen	369
12.6 Fazit	370
13 swoodoo – eine wahrhaft agile Geschichte	373
13.1 Einführung	373
13.2 Evolution: Nur die Starken überleben	374
13.3 Wie wir die "eXtreme Seite" erreichten	379
13.3.1 Kontinuierliche Integration	380

13.3.2	Testgetriebene Entwicklung	381
13.3.3	Tägliche Standup-Meetings	382
13.4	Und wo wir schon einmal dabei sind	383
13.4.1	User Storys und Story Points	384
13.4.2	Velocity	385
13.4.3	Iterationsplanung	386
13.4.4	Programmieren in Paaren	386
13.4.5	Kollektives Eigentum	388
13.4.6	Offenheit für Änderungen	389
13.4.7	Überstunden	391
13.5	Die Kunst der Evolution	391
13.6	KISS und YAGNI – zwei Seiten einer Medaille	397
13.7	Evolutionstheorie und Fazit	398
VI	Nichtfunktionale Aspekte	401
14	Gebrauchstauglichkeit	403
14.1	Einführung	403
14.2	Anything goes – aber zu welchem Preis?	405
14.3	Designaspekte	407
14.3.1	Barrierefreiheit	407
14.3.2	Lesbarkeit	407
14.3.3	Label für Formularelemente	408
14.3.4	Tastaturbedienbare Webseite	408
14.3.5	Gute Farbkontraste	410
14.3.6	Logo zur Startseite verlinken	410
14.3.7	Alternativtexte für Bilder	410
14.3.8	Hintergrundbild mit Hintergrundfarbe	410
14.3.9	Druckversion nicht vergessen	411
14.3.10	Erkennbare Links	411
14.3.11	Gute Bookmarks	411
14.3.12	Keine Frames	412
14.3.13	Skalierbare Schrift	412
14.4	Technische Aspekte	412
14.4.1	Performanz	412

14.4.2	JavaScript	414
14.5	Benutzerführung	415
14.5.1	Der Mythos des Falzes	416
14.5.2	Feedback bei Interaktionen	417
14.5.3	Navigation	417
14.5.4	Popups und andere Störenfriede	418
14.5.5	Gewohnheiten bedienen, Erwartungen nicht enttäuschen	419
14.5.6	Fehlertoleranz und Feedback	419
14.6	Testen der Usability	420
14.7	Fazit	421
15	Performanz	423
15.1	Einführung	423
15.1.1	Werkzeuge	424
15.1.2	Umgebungsbezogene Gesichtspunkte	425
15.2	Lasttests	427
15.2.1	Apache Bench	428
15.2.2	Pylot	430
15.2.3	Weitere Werkzeuge für Lasttests	432
15.3	Profiling	432
15.3.1	Callgrind	434
15.3.2	APD	439
15.3.3	Xdebug	440
15.3.4	XHProf	442
15.3.5	OProfile	443
15.4	Systemmetriken	445
15.4.1	strace	445
15.4.2	Sysstat	446
15.4.3	Lösungen im Eigenbau	448
15.5	Übliche Fallstricke	449
15.5.1	Entwicklungsumgebung gegen Produktivumgebung	449
15.5.2	CPU-Zeit	450
15.5.3	Mikro-Optimierungen	450
15.5.4	PHP als <i>Glue Language</i>	451
15.5.5	Priorisierung von Optimierungen	451
15.6	Fazit	452

16 Sicherheit	455
16.1 Was ist eigentlich Sicherheit?	455
16.2 Secure by Design	456
16.2.1 Der Betrieb	456
16.2.2 Physikalischer Zugang	458
16.2.3 Software-Entwicklung	458
16.3 Was kostet Sicherheit?	462
16.4 Die häufigsten Probleme	463
16.4.1 A10 – Transportschicht	463
16.4.2 A9 – Kryptografie	464
16.4.3 A8 – Weiterleitungen	465
16.4.4 A7 – Zugriffsschutz	466
16.4.5 A6 – Konfiguration	467
16.4.6 A5 – Cross Site Request Forgery (CSRF/XSRF)	468
16.4.7 A4 – Direkte Zugriffe	468
16.4.8 A3 – Authentifizierung und Session-Management	469
16.4.9 A2 – Cross Site Scripting (XSS)	470
16.4.10 A1 – Injections	472
16.5 Fazit	473
VII Schlussbetrachtungen	475
Stichwortverzeichnis	489

Geleitwort

Die Entwicklung qualitativ hochwertiger Software sowie das Sicherstellen der Software-Qualität sind keine neuartigen Konzepte, und kaum jemand wird widersprechen, dass diese Konzepte für die Softwareentwicklung von enormer Bedeutung sind. Ich hatte das Privileg, über viele Jahre wirklich missionskritische Software zu entwickeln. Ich meine, die Art von Software, von der Menschenleben abhängen.

Während dieser Zeit habe ich eine Menge darüber gelernt, wie man einen Prozess für die Qualitätssicherung zu Beginn eines Projektes einführt und bis zum unternehmenskritischen Einsatz in Produktion vorantreibt. Das Gestalten eines Entwicklungsprozesses, der zu qualitativ hochwertiger Software führt, ist nicht trivial und erfordert nicht nur Unterstützung durch das Management, sondern auch dessen eigenes Engagement. Dies hat Einfluss auf die Aufbauorganisation des Unternehmens sowie seine Mitarbeiter, Systeme und Prozesse.

Meiner Meinung nach stellen die Probleme, die sich aus der großen Reichweite sowie aus der hohen Entwicklungsgeschwindigkeit des Internets ergeben, die Probleme in den Schatten, denen ich mich bei der Entwicklung der oben erwähnten missionskritischen Software stellen musste. Während viele dieser neuen Softwaresysteme „nur“ unternehmenskritisch sind, sind sie in Wahrheit nicht weniger wichtig und müssen zusätzliche Schwierigkeiten wie beispielsweise Internationalisierung, Schutz vor bekannten und neuen Angriffen im Web sowie die Arbeit in verteilten Teams und mit immer kürzeren Releasezyklen bewältigen.

Im E-Commerce-Bereich schlägt sich ein Ausfall der Anwendung direkt in einem Ausfall von Einnahmen nieder; daher ist dort der Bedarf der Software-Qualität noch größer. Besonderes Augenmerk liegt hierbei auf Compliance sowie der Möglichkeit, eventuelle Fehler schnell beheben und diese Fehlerbehebung sofort testen und umgehend bereitstellen zu können. Die Anwendung muss nicht nur „online“, sondern tatsächlich in der Lage sein, Transaktionen in Echtzeit zu verarbeiten.

Die wachsende Bedeutung der *User Experience* führt außerdem dazu, dass die durch die Benutzer der Anwendung wahrgenommene Qualität unternehmenskritisch wird: die Software muss nicht nur korrekt funktionieren, sondern auch den Erwartungen der Benutzer genügen. Ist dies nicht der Fall, müssen die ent-

sprechenden Änderungen in kürzester Zeit so umsetzbar sein, dass die Software-Qualität nicht beeinträchtigt wird. Sowohl die Prozesse für die Entwicklung und Qualitätssicherung der Software als auch die eingesetzten Systeme müssen diese schnellen Entwicklungen unterstützen.

Diese Herausforderungen haben zu signifikanten Änderungen im Bereich der Qualitätssicherung geführt, besonders im Vergleich dazu, wie früher auftragsentscheidende Software entwickelt wurde. Die Softwareentwicklung hat in den letzten Jahren große Fortschritte gemacht. Best Practices wurden etabliert und das Bewusstsein für Software-Qualität wurde gestärkt. Zu den wichtigsten Fortschritten gehört hierbei die Erkenntnis, dass die Entwickler in die Qualitätssicherung einbezogen werden müssen und die entsprechende Verantwortung nicht alleine bei der QA-Abteilung liegen darf. Die Methodik der kontinuierlichen Integration entschärft eines der größten Probleme – und einen der größten Flaschenhälse – bei der Entwicklung von qualitativ hochwertiger Software: die Integrationsphase. Der strategische Fokus auf automatisiertes Testen ermöglicht es, Fehlerbehebungen schneller ausliefern zu können. Hierdurch können Dienstgütevereinbarungen (englisch: *Service Level Agreements*) nicht nur eingehalten, sondern meist auch übererfüllt werden, was zu einer höheren Kundenzufriedenheit führt.

Dieses Buch behandelt die verschiedenen Disziplinen der Qualitätssicherung und ihre Auswirkungen auf Menschen, Systeme, Prozesse sowie Werkzeuge. Hierbei liegt der Fokus auf der praktischen Anwendung in PHP-basierten Projekten. Die in diesem Buch zusammengetragenen Fallstudien sind von unschätzbarem Wert. Sie erlauben das Lernen von anderen Entwicklerteams, beispielsweise wie diese Best Practices und Werkzeuge „in der richtigen Welt“ einsetzen. Die Autoren verfügen über eine beispiellose Mischung aus theoretischem Hintergrundwissen und praktischer Erfahrung aus dem Alltag großer Projekte, die mit PHP realisiert wurden. Darüber hinaus leisten sie durch die Entwicklung von Werkzeugen für die Qualitätssicherung einen entscheidenden Beitrag für das PHP-Ökosystem. Ich kann mir keine besseren Autoren für ein solches Buch vorstellen.

Ich bin mir sicher, dass Ihnen dieses Buch dabei helfen wird, die Qualität Ihrer Projekte zu steigern, sodass sowohl Ihr Entwicklerteam als auch das Management auf die entwickelte Software stolz sein können.

Andi Gutmans, CEO Zend Technologies Ltd.

Vorwort

*„Experience: that most brutal of teachers.
But you learn, my God do you learn.“ — C.S. Lewis*

Über dieses Buch

Dem TIOBE Programming Community Index zufolge ist PHP die populärste Programmiersprache nach C/C++ und Java [TIOBE Mai 2010]. Gartner geht davon aus, dass viele neue, geschäftskritische Projekte von Unternehmen in einer dynamischen Sprache realisiert werden, und sieht PHP als derzeit stärksten Vertreter dieser Art von Programmiersprache [Gartner 2008]. PHP war von Anfang an für die Webprogrammierung konzipiert und dürfte um die Jahrtausendwende einer der wesentlichen Motoren des Dotcom-Booms gewesen sein.

Mittlerweile ist PHP zu einer Mehrzweckprogrammiersprache gereift und unterstützt sowohl die prozedurale als auch die objektorientierte Programmierung. Waren in der Vergangenheit Themen wie Performanz, Skalierbarkeit und Sicherheit Dauerbrenner in der PHP-Community, ist in den letzten Jahren auch Architektur und Qualität vermehrt Aufmerksamkeit zuteil geworden. In der Beratungspraxis zeigt sich, dass immer mehr Unternehmen ihre PHP-basierte Software modernisieren und ihren Entwicklungsprozess meist nach agilen Werten neu gestalten möchten. Eine Modernisierung der Codebasis ist meist entweder durch eine Migration von PHP 4 auf PHP 5 oder, um die Entwicklung zu standardisieren, durch die Einführung eines Frameworks motiviert.

Vor diesem Hintergrund verwundert es nicht, dass es heute eine Fülle an PHP-Frameworks gibt, die beim Lösen wiederkehrender Anwendungsfälle und bei der Standardisierung der Anwendungsentwicklung helfen wollen. Dynamische und statische Testverfahren sowie automatisierte Builds und kontinuierliche Integration sind für viele PHP-Entwickler längst keine Fremdwörter mehr. Aus PHP-Programmierung ist, gerade in unternehmenskritischen Projekten, Software-Engineering mit PHP geworden.

Ist dies ein PHP-Buch?

Anhand von Beispielen aus der PHP-Welt vermittelt dieses Buch die Planung, Durchführung und Automation von Tests für die unterschiedlichen Software-schichten, die Messung von Software-Qualität mithilfe von Software-Metriken sowie den Einsatz geeigneter Methoden wie beispielsweise kontinuierlicher Integration. Wir gehen davon aus, dass unsere Leser entweder erfahrene PHP-Entwickler sind, die sich für die Qualitätssicherung in PHP-Projekten interessieren, oder Entwickler, die mit einer anderen Programmiersprache so weit vertraut sind, dass sie den Beispielen folgen können.

Dieses Buch kann und will keine Einführung in die (objektorientierte) Programmierung mit PHP 5 sein. Und obwohl sich viele Unternehmen im Rahmen der Migration von PHP 4 nach PHP 5 zum ersten Mal intensiver mit Qualitätssicherung in einem PHP-Projekt auseinandersetzen, so kann auch die Migration von PHP-Umgebungen und -Anwendungen in diesem Buch nicht behandelt werden. Für diese beiden Themen sei auf [Bergmann 2005] und [Pribsch 2008] verwiesen.

Neben den Entwicklern müssen sich auch Projektleiter und Qualitätsbeauftragte mit dem Thema Software-Qualität befassen. Wir hoffen, dass dieses Buch das gegenseitige Verständnis zwischen den verschiedenen an Software-Projekten beteiligten Gruppen fördert und allen Lesern eine Motivation bietet, die interne Qualität (siehe Abschnitt 1.1.2) ihres Codes zu verbessern.

Aufbau des Buches

Der Idee folgend, dass man am besten durch Erfahrung – auch und gerade aus den Erfahrungen anderer – lernt, stellt dieses Buch Fallstudien zusammen, die einen Blick hinter die Kulissen bekannter Firmen und Projekte erlauben und wertvolle Praxiserfahrungen vermitteln.

Der erste Abschnitt, *Grundlagen*, erklärt, was wir unter der Qualität von Software verstehen und wie man die unterschiedlichen Schichten einer Software testen kann.

Der Abschnitt *Best Practices* zeigt erprobte Vorgehensweisen, unter anderem in Bezug auf das Schreiben von Unit-Tests, und wie diese von den Entwicklern von Digg Inc. und des TYPO3-Projektes umgesetzt werden. Das Kapitel *Bad Practices* greift dieselbe Thematik aus der anderen Richtung auf und zeigt die Fallstricke, auf die man beim Schreiben von Unit-Tests achten sollte.

Im Abschnitt *Server und Services* wird das Testen von serviceorientierten APIs und Server-Komponenten behandelt.

Der Abschnitt *Architektur* zeigt am Beispiel von Symfony, wie sowohl ein Framework selbst als auch die auf dessen Basis entwickelten Anwendungen getestet werden können. Am Beispiel der Graph-Komponente aus den eZ Components wird erklärt, wie eine gute Architektur aus lose gekoppelten Objekten selbst das Testen von binären Ausgabedaten wie Grafiken möglich macht. Das *Testen von*

Datenbank-Interaktionen ist ein Thema, das mehrere Schichten der Architektur einer Anwendung betrifft, und daher ein eigenes Kapitel wert.

Im Abschnitt *QA im Großen* berichten die Entwickler von studiVZ und Swoodoo von ihren Erfahrungen mit der Qualitätssicherung in großen Projekten und Teams. Das Kapitel *Kontinuierliche Integration* schlägt einen Bogen von den dynamischen zu den statischen Testverfahren und zeigt, wie die vielen verschiedenen Werkzeuge der Qualitätssicherung effektiv zusammen eingesetzt werden können.

Der letzte Abschnitt, *Nichtfunktionale Aspekte*, rundet das Buch mit einer Betrachtung der Qualitätsaspekte *Gebrauchstauglichkeit*, *Performanz* und *Sicherheit* ab.

Vorstellung der Autoren

Sebastian Bergmann, thePHP.cc

Diplom-Informatiker Sebastian Bergmann ist ein Pionier der Qualitätssicherung in PHP-Projekten. Sein Test-Framework PHPUnit ist ein De-facto-Standard. Er ist aktiv an der Entwicklung von PHP beteiligt und Schöpfer verschiedener Entwicklungswerkzeuge. Sebastian Bergmann ist ein international nachgefragter Experte. Als Autor gibt er seine langjährige Erfahrung in Büchern und Fachartikeln weiter und hält Vorträge auf Fachkonferenzen rund um die Welt.

Stefan Pribsch, thePHP.cc

Diplom-Informatiker Stefan Pribsch ist Spezialist für die Entwicklung PHP-basierter Software. Seine Erfahrung bringt er in Entwicklungswerkzeuge ein. Als Experte für objektorientierte Programmierung, Software-Architektur und Frameworks spricht er regelmäßig auf internationalen IT-Konferenzen. Stefan Pribsch ist Autor zahlreicher Bücher und Fachartikel über sämtliche Aspekte des Software-Lebenszyklus.

Robert Lemke, TYPO3 Association und Karsten Dambekalns, TYPO3 Association

Robert Lemke ist Gründungsmitglied der TYPO3 Association und leitet die Entwicklung des Rewrites von TYPO3 sowie des Frameworks FLOW3. Er hat eine besondere Vorliebe für agile Entwicklungsmethoden und es sich zum Ziel gesetzt, neue Ansätze wie Domain-Driven Design oder aspektorientiertes Programmieren in der PHP-Welt zu etablieren. Robert lebt in Lübeck, zusammen mit seiner Frau Heike, seiner Tochter Smilla und Vibiemme, ihrer Espressomaschine.

Karsten Dambekalns programmiert seit 1999 in PHP und entdeckte 2002 die immensen Möglichkeiten von TYPO3. Er ist heute einer der Kernentwickler von TYPO3 und FLOW3 sowie Mitglied im Steering Committee der TYPO3 Association.

Nach der Gründung einer eigenen Firma im Jahre 2000 steht Karsten Dambekalns seit 2008 wieder als Freelancer voll im Dienst der TYPO3-Entwicklung. Außerdem ist er Autor, spricht auf Konferenzen und verbringt den Großteil seiner Freizeit mit seiner Frau und seinen drei Kindern.

In ihrem Kapitel *TYPO3: die agile Zukunft eines schwergewichtigen Projekts* stellen Robert Lemke und Karsten Dambekalns Grundsätze und Techniken vor, mit denen das TYPO3-Projekt die Software-Qualität nachhaltig verbessern konnte.

Benjamin Eberlei, direkt:effekt GmbH

Benjamin Eberlei ist Softwareentwickler bei der direkt:effekt GmbH. In seiner Freizeit pflegt und entwickelt er Komponenten für das Zend-Framework sowie einige kleinere Open-Source-Projekte.

In seinem Kapitel *Bad Practices in Unit-Tests* zeigt Benjamin Eberlei, welche Fehler man beim Schreiben von Tests vermeiden sollte, um den größtmöglichen Nutzen aus dem Testen von Software ziehen zu können.

Matthew Weier O'Phinney, Zend Technologies Ltd.

Matthew Weier O'Phinney arbeitet als Project Lead für das Zend Framework. Zu seinen Aufgaben zählen das Release-Management ebenso wie die Implementierung und Verbreitung von Best Practices sowie die Kommunikation mit der Community. Matthew ist ein aktiver Befürworter von Open-Source-Software und PHP. Seit 2000 programmiert er in PHP und anderen Sprachen. Für Magazine wie PHP Architect und in seinem eigenen Blog¹ sowie in der Zend DevZone² schreibt er über aktuelle Themen.

In seinem Kapitel *Testen von serviceorientierten APIs* geht Matthew Weier O'Phinney auf die besonderen Herausforderungen beim Testen von Webdiensten ein und präsentiert Ansätze und Lösungen, die sich im Zend Framework-Projekt bewährt haben.

Tobias Schlitt, Qafoo GmbH

Tobias Schlitt ist ausgebildeter Fachinformatiker und Diplom-Informatiker. Er beschäftigt sich seit 1999 mit der Entwicklung von Webanwendungen auf Basis von PHP und war, nach mehrjähriger Aktivität im PEAR-Projekt, maßgeblich an Architektur und Entwicklung der eZ Components beteiligt. Daneben beteiligt er sich dauerhaft an verschiedenen Open-Source-Projekten rund um PHP. Als anerkannter Experte ist er beratend in den Bereichen Software-Architektur und Qualitätssicherung tätig. Tobias Schlitt gründete Mitte 2010 zusammen mit Manuel Pichler und Kore Nordmann, beide ebenfalls als Autoren in diesem Buch vertre-

¹ <http://weierophinney.net/matthew/>

² <http://devzone.zend.com/>

ten, die Qafoo GmbH, welche Experten-Consulting, Training und Support rund um die Entwicklung von hoch-qualitativem PHP-Code und Qualitätssicherung in Software-Projekten anbietet.

In seinem Kapitel *Wie man einen WebDAV-Server testet* zeigt Tobias Schlitt, dass man beim Testen manchmal unkonventionelle Wege gehen muss, um seine Ziele erreichen zu können.

Fabien Potencier, Sensio Labs

Fabien Potencier¹ entdeckte das Web 1994, zu einer Zeit, als das Verbinden mit dem Internet noch von kreischenden Tönen eines Modems begleitet wurde. Als passionierter Entwickler begann er sofort mit der Entwicklung von Webseiten mit Perl. Mit der Veröffentlichung von PHP 5 legte er seinen Fokus auf PHP und startete 2004 das Symfony Framework², um in seiner Firma die Mächtigkeit von PHP für Kunden voll nutzen zu können. Fabien ist ein "Serienunternehmer", der 1998 neben anderen Firmen auch Sensio, einen auf Webtechnologien und Internet-Marketing spezialisierten Dienstleister, gegründet hat. Ferner ist er der Schöpfer mehrerer Open-Source-Projekte, Autor, Blogger und Referent auf internationalen Konferenzen sowie stolzer Vater von zwei wundervollen Kindern.

In seinem Kapitel *Testen von Symfony und Symfony-Projekten* berichtet Fabien Potencier von seinen Erfahrungen aus dem Symfony-Projekt und zeigt unter anderem, wie das Testen von Symfony zu besseren Programmierschnittstellen geführt hat.

Kore Nordmann, Qafoo GmbH

Kore Nordmann entwickelt, leitet und plant seit mehreren Jahren verschiedene PHP-basierte Open-Source-Projekte. Im Sommer 2010 hat er zusammen mit Manuel Pichler und Tobias Schlitt die Qafoo GmbH gegründet und steht darüber Unternehmen zur Verfügung, um die Architektur und Qualität von Software-Projekten zu verbessern. Seine Expertise teilt er regelmäßig auf verschiedenen Konferenzen sowie in Artikeln und Büchern mit. Neben der Entwicklung der eZ Components leitet er die Entwicklung von Arbit³, einer neuen Software für die Verwaltung von Software-Projekten, inklusive der Integration von Werkzeugen für die Qualitätssicherung.

In seinem Kapitel *Testen von Grafikausgaben* beschreibt Kore Nordmann, wie es mit einer guten Architektur und dem Einsatz von Mock-Objekten möglich ist, selbst eine Komponente, die binäre Grafikausgaben erzeugt, umfassend zu testen.

¹ <http://fabien.potencier.org/>

² <http://www.symfony-project.org/>

³ <http://arbitracker.org/>

Michael Lively Jr, SellingSource LLC.

Michael Lively arbeitet seit 2001 mit PHP und bringt sich seit 2005 in der PHP Testing Community ein. Er ist der Schöpfer der Erweiterung für Datenbanktests in PHPUnit, zu dem er auch weitere Patches beigetragen hat. Michael Lively arbeitet als Lead Developer und Application Architect für die SellingSource LLC mit Sitz in Las Vegas. Zu seinem Arbeitsbereich gehört die Entwicklung einer mit PHP realisierten Enterprise-Plattform für Kreditmanagement, die von Hunderten Maklern für Millionen von Kunden verwendet wird.

In seinem Kapitel *Testen von Datenbank-Interaktionen* dokumentiert Michael Lively Jr die Funktionalität der Datenbankerweiterung von PHPUnit und zeigt, wie dieses mächtige Werkzeug effektiv eingesetzt werden kann.

Christiane Philipps, DailyDeal GmbH, und Max Horváth, Vodafone GmbH

Max Horváth ist Lead Software Engineer bei Vodafone Internet Services und beschäftigt sich seit zehn Jahren mit Webentwicklung. In dieser Zeit hat er mit Unternehmen, Entwicklern und Designern zusammengearbeitet, um Webprojekte für große und kleine Unternehmen umzusetzen. Während der Arbeit an der Fallstudie für dieses Buch war er Team Lead Mobile Development bei VZnet Netzwerke.

Christiane Philipps ist CTO bei der DailyDeal GmbH. Die Fachinformatikerin arbeitet seit 2000 als Consultant und in Festanstellung für und mit Webunternehmen. Ihr Herz schlägt besonders für Agile Testing und Agile Leadership, Themen, über die sie auch in ihrem Blog¹ regelmäßig schreibt. Von Frühjahr 2008 bis Herbst 2009 arbeitete sie bei VZnet Netzwerke, davon ein Jahr lang als Leiterin des Bereichs Quality Assurance & Deployment.

In ihrem Kapitel *Qualitätssicherung bei studiVZ* berichten Christiane Philipps und Max Horváth, wie sie PHPUnit und Selenium RC erfolgreich in einem der größten sozialen Netzwerke Europas eingeführt haben.

Manuel Pichler, Qafoo GmbH, und Sebastian Nohn, Ligatus GmbH

Manuel Pichler kam erstmals 1999 mit der Programmiersprache PHP in Kontakt und ist ihr bis heute treu geblieben. Während seines Studiums sammelte er erste Erfahrungen im Bereich Qualitätssicherung. Im Anschluss an das Studium arbeitete er als Software-Architekt und entwickelte im Rahmen dieser Tätigkeit verschiedene Erweiterungen für CruiseControl, die dann 2007 die Grundlage für phpUnderControl bildeten. Neben diesem Projekt ist er auch der Schöpfer von

¹ <http://agile-qa.de/>

PHP.Depend¹ und PHPMD², Werkzeugen zur statischen Analyse von PHP-Code. Im Sommer 2010 hat er zusammen mit Kore Nordmann und Tobias Schlitt die Qa-foo GmbH gegründet und bietet darüber Support und Schulungen rund um das Themengebiet Qualitätssicherung an.

Sebastian Nohn beschäftigt sich seit 1996 mit dynamischen Websites und seit 2002 mit Qualitätssicherung im kommerziellen und Open-Source-Bereich. Er war einer der ersten, der CruiseControl für die Nutzung in PHP-Projekten adaptierte, und gab den Anstoß für die Entwicklung von phpUnderControl. Zurzeit ist er bei der Ligatus GmbH, einem der führenden Performance-Marketing-Anbieter beschäftigt, wo er den Bereich Qualitätssicherung aufbaute und die Verantwortung als Teamleiter für die Bereiche Qualitätssicherung und Infrastruktur trägt. Sebastian Nohn ist Wirtschaftsinformatiker und schreibt in unregelmäßigen Abständen in seinem Weblog³ über IT-Themen.

In ihrem Kapitel *Kontinuierliche Integration* berichten Manuel Pichler und Sebastian Nohn, wie kontinuierliche Integration, nachträglich eingeführte Unit-Tests, Software-Metriken und weitere statische Testverfahren dazu beigetragen haben, die Qualität einer Legacy-Applikation deutlich zu erhöhen.

Lars Jankowsky, swoodoo AG

Lars Jankowsky ist CTO der Swoodoo AG und verantwortlich für den PHP-basierten Flug- und Hotel-Preisvergleich. Seit mehr als 15 Jahren entwickelt er Webanwendungen und nutzt PHP seit den frühen Anfängen. Neben der Entwicklung von Software ist die Leitung von eXtreme Programming-Teams eine seiner Leidenschaften.

In seinem Kapitel *swoodoo – eine wahrhaft agile Geschichte* zeigt Lars Jankowsky, wie agile Methoden und eine serviceorientierte Architektur die sanfte und kontinuierliche Evolution einer Anwendung ermöglicht haben.

Jens Grochtdreis

Jens Grochtdreis⁴ ist freier Webentwickler und Berater. Er ist dabei auf moderne Frontend-Entwicklung und Barrierefreiheit spezialisiert. Vor seiner Selbstständigkeit arbeitete er zehn Jahre in Agenturen, unter anderem an Projekten für eine sehr große deutsche Bank und einen großen Telekommunikationsanbieter. Jens gründete 2005 die Webkrauts⁵, um für ein besseres Medium zu streiten. Wenn er nicht gerade bloggt, twittert, surft oder codet, dann entspannt er sich bei Comics, Krimis, hört Blues oder kocht.

¹ <http://pdepend.org/>

² <http://phpmd.org/>

³ <http://nohn.net/blog/>

⁴ <http://grochtdreis.de>

⁵ <http://webkrauts.de/>

In seinem Kapitel *Gebrauchstauglichkeit* zeigt Jens Grochtdreis, wie einfach nutzbare und verständliche Webseiten entwickelt werden können und wie die Gebrauchstauglichkeit getestet werden kann.

Brian Shire

Brian Shire entdeckte das Programmieren im Alter von acht Jahren auf einem Apple IIe. Wenn er keine Spiele spielte, lernte er die Programmiersprache BASIC. Während der Arbeit an der Fallstudie für dieses Buch arbeitete er bei Facebook Inc., wo er für die Skalierung der PHP-Infrastruktur verantwortlich war. In seinen vier Jahre bei Facebook wuchs die Plattform von 5 Millionen Nutzern auf 175 Millionen Nutzer. In dieser Zeit wurde Brian zu einem Kernentwickler von APC, einem Bytecode- und Daten-Cache für PHP. Er trug außerdem zur Entwicklung des PHP-Interpreters sowie verschiedenen Erweiterungen in PECL bei. Brian teilt seine Erfahrung und sein Wissen als Referent auf internationalen Konferenzen und in seinem Blog¹. Zurzeit lebt er in San Francisco.

In seinem Kapitel *Performanz* motiviert Brian Shire das Testen der Performanz von Webanwendungen und führt in die wichtigsten Werkzeuge und Methoden dafür ein.

Arne Blankerts, thePHP.cc

Arne Blankerts hat langjährige Erfahrung als IT-Entwicklungsleiter. Seine Software fCMS nutzt innovativ XML-Technologien und ist Basis unternehmenskritischer Anwendungen internationaler Konzerne. Er ist aktiv an der Dokumentation von PHP beteiligt. Arne Blankerts ist Experte für IT-Sicherheit und schreibt darüber eine Kolumne in einem Fachmagazin. Er ist gefragter Referent auf internationalen IT-Konferenzen, Buchautor und veröffentlicht Fachartikel.

In seinem Kapitel *Sicherheit* zeigt Arne Blankerts, wie einfach das Schreiben grundsätzlich sicherer Anwendungen ist, wenn man die gängigen Angriffsvektoren kennt und einige wichtige Regeln beachtet.

¹ <http://tekrat.com>

Stichwortverzeichnis

- Äquivalenzklasse, 18
- Überwachung, 455

- Akzeptanztest, 4, 24, 147, 303, 333
- Allgegenwärtige Sprache, 75, 78
- Alternative PHP Cache (APC), 141, 434
- Alternative PHP Debugger (APD), 434, 439
 - apd.set_pprof_trace(), 439
 - pprofp, 439
- Apache Bench, 424, 428
- Apache HTTP Server, 428, 433, 436
- Arbit, 16
- Aufwärmphase, 436
- Ausführungspfad, 25

- Backup, 458
- Barrierefreiheit, 407
- Benchmark, 428
- Build
 - automatisierter, 338, 342
 - kontinuierlicher, 338
 - Management, 338
 - täglicher, 338
- Bytekit, 15

- Caching, 141
 - Cache Locking, 427
 - Cache Miss, 426
 - Cache Priming, 426
 - Cache Warming, 426
- Call Graph, 434, 438
- Callee, 438

- Caller, 438
- Callgrind, 434
- Capability Maturity Model Integration (CMMI), 8
- Carica CacheGrind, 441
- Change Risk Anti-Patterns (CRAP) Index, 11
- Code Smell
 - Duplizierter Code, 10, 76, 97, 340
 - Große Klasse, 76
 - Lange Methode, 76
- Code-Altlasten, 124, 360
- Code-Coverage, 11, 24, 80, 208, 220, 256
 - Pfadabdeckung, 24
 - Statement Coverage, 24
- Code-Review, 72, 342
- Codeduplikat, 339
- Coding Standard, 14, 77
- Collective Code Ownership, *siehe* Kollektives Eigentum
- CPU-Metrik, 424
- CPU-Zeit, 424
- Cross Site Request Forgery, 468
- Cross Site Scripting, 470
- CruiseControl, 15

- Data Set, 263
- Datenbanktest, 251
- Datenzugriffsschicht, 142
- DBUnit, 257
- Debug Build, 426
- Debugger, 439, 440

- Debugging-Symbole, 435
- Dependency Injection, 9, 12, 30, 136, 147, 211
- Dialog, 403
- Domänengetriebenes Design, 78
- Domain-Driven Design, *siehe* Domänengetriebenes Design
- Entwurfsmuster
 - Acceptor, 138
 - Chain of Responsibility, 141
 - Data Transfer Object, 345
 - Decorator, 264, 272
 - Multiton, 209
 - Observer, 139, 214
 - Proxy, 143
 - Repository, 113
 - Singleton, 209
 - Table Data Gateway, 42
 - Visitor, 138
- eXtreme Programming, 379
- Fehlbedienung, 419
- Fehlerbehandlung, 460
- Fehlertoleranz, 419
- Fixture, 97
- Flaschenhals, 252, 423
- Flood, 432
- Fluent Interface, 224
- Funktion
 - aufgerufene, 438
 - Aufrufer, 438
- Funktionalität, 4
- FURPS, 3
- GCC, 435
- Gebrauchstauglichkeit, 4, 403
- Gleitpunktzahl, 235
- Global State, 12, 105
- gprof, 443
- Happy Path, 18
- Hardening, 455
- Heatmap, 421
- HipHop, 337
- HTTPPerf, 432
- HTTPLoad, 424
- Hudson, 15, 73, 74
- Instrumentierung, 432, 437
- Integrationstest, 151
- Iteration, 385
- JavaScript, 414
- JMeter, 432
- KCachegrind, 81, 437, 439
- Keep It Simple, Stupid (KISS), 381
- Klickdummy, 404, 421
- Kohäsion, 13
- Konfiguration, 337
- Kontinuierliche Integration, 149, 157, 209, 307, 380
- Kopplung, 13, 459
- Lasttest, 424, 427
- Lime, 216
- Lines of Code, 342
 - Comment, 343
 - Executable, 343
 - Non Comment, 343
- Lock, 434
- Logfile, 421
- MacCallGrind, 441
- Memcache, 141, 425
- meminfo, 425
- Migration, 253
- Mock-Objekt, 9, 12, 47, 84, 137, 234, 251, 256
- Model-View-Controller (MVC), 376
- MySQL, 142, 253
- Navigation, 403
- Nebenläufigkeit, 428
- Non-Mockable Total Recursive Cyclo-matic Complexity, 12
- NPath-Komplexität, 11, 25
- OProfile, 434, 443
- Optimierung, 423
- OWASP, 463

- Pair Programming, *siehe* Programmieren in Paaren
- Partitionierung, 142, 378
- PEAR, 127
- Performanz, 412, 423
- Performanztest, 423, 426
- Persistenz, 251
- php.ini, 439
- PHP_CodeBrowser, 15
- PHP_CodeSniffer, 14, 74
- PHP_Depend, 14
- phpcpd, 14
- phpdcd, 14
- phploc, 13
- phpmd, 14
- PHPT, 128
- phpUnderControl, 15
- PHPUnit, 13, 79, 128, 257
 - Selenium-Erweiterung, 308
- Planning Game, 383
- Port
 - privilegierter, 433
- Profiler, 424, 426, 432, 439, 440, 442, 443
- Profiling, 424, 432
- Propel, 337
- Pylot, 424, 430
- Qualität
 - externe, 4
 - interne, 5
- Qualitätsmanagement, 5
- Qualitätssicherung
 - konstruktive, 8
- Reaktionsfreudigkeit, 4
- Rechteverwaltung, 455
- Redirect, 171
- Refaktorisierung, 5, 76, 80, 215, 340, 374
- Regression, 29
- Regressionstest, 295
- Remote Procedure Call (RPC), 156
- Representational State Transfer (REST), 156
- Response Time, 430
- Root-Server, 457
- Runkit, 125
- sar, 425
- Scrum, 74, 147, 304, 382
- Secure by Design, 456
- Security by Obscurity, 458
- Seiteneffekt, 31
- Selenium, 22, 222, 302, 376
- Selenium Grid, 307
- Selenium IDE, 306
- Selenium RC, 150, 306
- Separation of Concerns, 376, 459
- Server API (SAPI), 433
- Serviceorientierte Architektur (SOA), 142, 378
- Sharding, 142, 378
- Shared Hosting, 457
- Shared Memory, 434
- Sicherheit, 4, 455
- Siege, 432
- Single Responsibility Principle, 10, 31
- sismo, 209
- SOAP, 156
- Software Process Improvement and Capability Determination (SPICE), 8
- Software-Artefakt, 342
- Software-Metrik, 10, 341
- Software-Qualitätsmodell, 3
- Spaghetti-Code, 124
- Spike Solution, 384
- Sprint, 147
- Standup-Meeting, 382
- Story Point, 384
- Stub, 47, 84, 251
- Suhosin, 457
- Symfony, 337
- Systemaufruf, 436
- Systemmetrik, 424
- Systemmetriken, 428
- Systemtest, 20
- Technical Debt, 6
- Test
 - Black-Box, 17, 124, 303
 - Browserkompatibilitäts, 333

- Capture&Replay, 306
- End-to-End, 302
- End-to-Test, 4
- Integration, 333
- White-Box, 17
- Test-First Programmierung, 8
- Test-Smell, 97
 - Begieriger Test, 99
 - Duplizierter Testcode, 97
 - Fragiler Test, 22, 102, 313, 316, 318
 - Indirekter Test, 108
 - Konditionale Testlogik, 114
 - Lügender Test, 112
 - Langsamer Test, 113
 - Mock-Overkill, 119
 - Obskurer Test, 104
 - Selbstvalidierender Test, 116
 - Skip-Epidemie, 120
 - Undurchsichtiger Testname, 109
 - Websurfender Test, 117
 - Zusicherungsroulette, 99
- Testautomatisierung, 325
- Testbarkeit, 10, 31, 136, 143
- Testdatenbank, 280, 298
- Testgetriebene Entwicklung, 8, 27, 71, 131, 381
- Testinventar, 24, 224, 252, 263, 280, 295, 298, 313
- Testisolation, 23
- Testplan, 21
- Testumgebung, 425
- Throughput, 430
- Timeboxing, 151
- Toleranzintervall, 235
- top, 425
- Trait, 139
- Ubiquitous Language, *siehe* Allge-
genwärtige Sprache
- Unit-Test, 25
- Usability, 403
- User Story, 384
- User-Test, 420
- Valgrind, 434, 443
- Velocity, 151, 385
- Verfügbarkeit, 4
- Versionsmanagement, 338
- vfsStream, 85
- Virtualisierung, 457
- Wasserfallmodell, 150
- WebDAV, 177
- Webgrind, 81, 441
- Webservice, *siehe* Webdienst
- White-Box-Test, 25
- Xdebug, 80, 434, 439, 440
 - xdebug.profiler.enable, 440
 - xdebug.profiler.enable.trigger, 440
 - xdebug.profiler.output_dir, 440
- XHProf, 82, 434, 442
 - xhprof.disable(), 442
 - xhprof.enable(), 442
 - xhprof.html, 442
 - xhprof.output_dir, 442
 - xhprof.lib, 442
- XML-RPC, 156
- You Ain't Gonna Need It (YAGNI), 381
- YSlow, 413
- Zend Extension, 439
- Zusicherung, 29
- Zuverlässigkeit, 4
- Zyklomatische Komplexität, 11, 342