

IT kompakt

Pascal Moll
Daniel Sonnet

Softwaretesting kompakt

Grundlagen von Tests und
Testautomatisierung mit Java



Springer Vieweg

IT kompakt

Die Bücher der Reihe „IT kompakt“ zu wichtigen Konzepten und Technologien der IT:

- ermöglichen einen raschen Einstieg,
- bieten einen fundierten Überblick,
- eignen sich für Selbststudium und Lehre,
- sind praxisorientiert, aktuell und immer ihren Preis wert.

Pascal Moll · Daniel Sonnet

Softwaretesting kompakt

Grundlagen von Tests und
Testautomatisierung mit Java



Springer Vieweg

Pascal Moll
Karben, Deutschland

Daniel Sonnet
Hochschule Fresenius Hamburg
Hamburg, Deutschland

ISSN 2195-3651

IT kompakt

ISBN 978-3-658-46104-1

<https://doi.org/10.1007/978-3-658-46105-8>

ISSN 2195-366X (electronic)

ISBN 978-3-658-46105-8 (eBook)

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <https://portal.dnb.de> abrufbar.

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2025

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jede Person benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des/der jeweiligen Zeicheninhaber*in sind zu beachten.

Der Verlag, die Autor*innen und die Herausgeber*innen gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autor*innen oder die Herausgeber*innen übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung/Lektorat: David Imgrund

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Danksagung

Mein besonderer Dank gilt **Alex Schuler** fürs sorgfältige Korrekturlesen, Fehlerfinden und den entscheidenden Anstoß, dieses Werk zu verfassen. Ohne dich hätte ich dieses Buch vermutlich noch immer nicht geschrieben.

Ebenso möchte ich mich bei **Moritz Wittig** fürs ausführliche Korrekturlesen und Fehlerfinden bedanken. Deine wertvollen Vorschläge haben dieses Buch besser gemacht und du hast mit den gefundenen Bugs in Screenshots und Texten die Qualität gesteigert.

Herzliches Dankeschön auch an **Patrick Eschenbach** und **Stefan Ludwig**, die ständig zum Korrekturlesen und Fehlerfinden für mich da waren.

Besonders möchte ich meiner Frau **Julia Beck** und meinen Kindern **Zoe Moll** und **Jan Moll** danken. Ohne eure Unterstützung wäre es mir nicht möglich gewesen, dieses Werk zu vollenden.

Inhaltsverzeichnis

1	Grundlagen des Softwaretestings	1
	Einleitung	1
	Ausgangslage – Das Projekt	2
	Folgen der fehlenden Tests	3
	Die Aufgaben eines Testers	5
	Grundlagen der Testarten	6
	Testpyramide	7
	Die Realität ist eine Eistüte	10
	Testaufbau – Vorbedingung, Ausführung, Nachbedingung	11
	Box-Testing	13
	Kommunikation und der Umgang mit gefundenen Fehlern	17
	Takeaways	18
2	Java-Grundlagen für Softwaretesting	19
	Die Java Virtual Machine	20
	Installation	21
	Programmerzeugung und Ausführung	22
	Java-Programmierung einer Fahrzeugverwaltung von Anfang an	23
3	Welcome to Apache Maven	59
	Was ist Maven?	60
	Setup	60
	POM – das Project Object Model	61
	Der Maven-Lebenszyklus	65

Maven-Proxy-Konfiguration	67
Verwendung der Datenbank-Dependency	68
Takeaways	71
4 Grundlagen der Testautomatisierung	73
Was ist Testautomatisierung	74
Funktionale und nichtfunktionale Tests	78
Testframeworks	79
Weboberflächentests	87
Deskoptests	89
Record & Replay	90
Strukturierte Skripte	91
Schlüsselwortgetriebene Skripte	92
Allgemeine Testautomatisierungsarchitektur	92
Die SOLID-Prinzipien	93
Allgemeine Testautomatisierungsarchitektur	95
Entwurfsmuster	98
Takeaways	102
5 Nachhaltigkeitsaspekte des Softwaretestings	105
Problemstellung	105
Green in Softwaretests	107
Takeaways	116
Literatur	117
6 Mocking und API-Testing	119
Was ist Mocking?	120
Wann eignet sich Mocking besonders gut?	120
Die verschiedenen Mock-Objekte	122
Vor- und Nachteile	123
WireMock	124
Takeaways	132
7 Oberflächen und deren automatisierte Tests mit Selenium	133
Funktionsweise	134
Selenium IDE	134
Selenium Grid	135
Eine Oberfläche für die Fahrzeugverwaltung	135
Selenium WebDriver	137
Takeaways	165

8	Behaviour Driven Development mit Cucumber	167
	Behaviour Driven Development	168
	Cucumber und Gherkin.	168
	Vor- und Nachteile von Behaviour Driven	
	Development.	180
	Good Practices	181
	Takeaways.	182
9	Exploratives Testen	185
	Wann sich exploratives Testen gut eignet	186
	Die Vorbereitung	187
	Die Nachbereitung	189
	Verschiedene Durchführungsarten einer	
	explorativen Testsession	189
	Durchführung des explorativen Testprozesses	191
	Vorteile	193
	Nachteile	194
	Good Practices	195
	Takeaways.	196
10	Zusammenfassung, wie geht es weiter	
	und Tipps	199
	Was nicht behandelt wurde und wie es	
	weitergehen könnte	201
	Tipps und Tricks	202
	Schlusswort	204
	Benutzerdefiniertes Verzeichnis	205

Abbildungsverzeichnis

Abb. 1.1	Testpyramide (Quelle: eigene Darstellung)	10
Abb. 1.2	Beispiel einer Zweigüberdeckung. (Quelle: Eigene Darstellung)	14
Abb. 2.1	Konsole (eigener Screenshot)	22
Abb. 2.2	Definition Class Auto (eigene Darstellung)	26
Abb. 2.3	Class Auto mit Variablen (eigene Darstellung)	26
Abb. 2.4	Class Auto mit gesetzten Variablen (eigene Darstellung)	27
Abb. 2.5	Main Methode (eigene Darstellung)	27
Abb. 2.6	Instanziierung der Class Auto (eigene Darstellung)	28
Abb. 2.7	Konstruktoren (eigene Darstellung)	29
Abb. 2.8	Aufruf verschiedener Konstruktoren (eigene Darstellung)	29
Abb. 2.9	Beispiele für Methoden (eigene Darstellung)	30
Abb. 2.10	Verwendung von Methoden und Konsolenausgabe (eigene Darstellung)	31
Abb. 2.11	If-Abfrage (eigene Darstellung)	34
Abb. 2.12	If-else (eigene Darstellung)	34
Abb. 2.13	Equals-Methode (eigene Darstellung)	35
Abb. 2.14	Switch Case (eigene Darstellung)	35
Abb. 2.15	Erstellen von Autos (eigene Darstellung)	36
Abb. 2.16	Konsoleneingaben einlesen (eigene Darstellung)	36

Abb. 2.17	If-Abfragen mit verschiedenen Operatoren (eigene Darstellung)	39
Abb. 2.18	Präinkrement (eigene Darstellung)	39
Abb. 2.19	Postinkrement (eigene Darstellung)	39
Abb. 2.20	Zuweisungsoperator mit Arithmetik (eigene Darstellung)	40
Abb. 2.21	Class Extends (eigene Darstellung)	41
Abb. 2.22	Vererbung von Auto-Konstruktor (eigene Darstellung)	41
Abb. 2.23	Vererbte Methode überschreiben (eigene Darstellung)	42
Abb. 2.24	Verwendung der vererbten Klasse (eigene Darstellung)	42
Abb. 2.25	Methodenerweiterung durch super (eigene Darstellung)	43
Abb. 2.26	Ergebnis Methodenerweiterung (eigene Darstellung)	43
Abb. 2.27	Optimierter Konstruktor Klasse Motorrad (eigene Darstellung)	43
Abb. 2.28	Interface IVersicherung (eigene Darstellung)	44
Abb. 2.29	Interface-Einbindung (eigene Darstellung)	45
Abb. 2.30	Verwendung von Interfaces (eigene Darstellung)	45
Abb. 2.31	Arrays (eigene Darstellung)	46
Abb. 2.32	Alternativzuweisung (eigene Darstellung)	47
Abb. 2.33	Zweidimensionales Array (eigene Darstellung)	47
Abb. 2.34	ArrayList Autos (eigene Darstellung)	48
Abb. 2.35	HashMap Autos (eigene Darstellung)	49
Abb. 2.36	Beispiel von values() (eigene Darstellung)	49
Abb. 2.37	Anwendung der HashMap (eigene Darstellung)	50
Abb. 2.38	HashMap auto ausgeben (eigene Darstellung)	51
Abb. 2.39	Anwendung im Hauptmenü (eigene Darstellung)	52
Abb. 2.40	Do Schritt 1 (eigene Darstellung)	52

Abb. 2.41	While Schritt 2 (eigene Darstellung)	53
Abb. 2.42	While mit Iterator (eigene Darstellung)	54
Abb. 2.43	Try Catch (eigene Darstellung)	55
Abb. 2.44	Eigene Exception (eigene Darstellung)	56
Abb. 2.45	Setze Hoechstgeschwindigkeit mit eigener Exception (eigene Darstellung)	57
Abb. 3.1	Metadaten einer POM (eigene Darstellung)	62
Abb. 3.2	Dependency einbinden (eigene Darstellung)	63
Abb. 3.3	Plugins und Skripte (eigene Darstellung)	64
Abb. 3.4	Alternative Repositories in einer POM (eigene Übersetzung)	64
Abb. 3.5	Maven-Lebenszyklus (eigene Darstellung)	65
Abb. 3.6	Maven Install in Eclipse (eigene Darstellung)	68
Abb. 3.7	Datenbank-Controller-Konstruktor (eigene Darstellung)	68
Abb. 3.8	Verwendung des Datenbank-Controller (eigene Darstellung)	69
Abb. 3.9	Selektion und Update der Datenbank (eigene Darstellung)	69
Abb. 3.10	Auslesen einer Spalte im ResultSet (eigene Darstellung)	70
Abb. 3.11	Speichern der Daten (eigene Darstellung)	70
Abb. 3.12	Daten aus Datenbank laden (eigene Darstellung)	71
Abb. 4.1	Ersten Test erzeugen mit @Test. (Eigene Darstellung)	80
Abb. 4.2	Beispiel von Variablen mit Wert null. (Eigene Darstellung)	82
Abb. 4.3	NullPointerException. (Eigene Darstellung)	82
Abb. 4.4	NullPointerException vermeiden. (Eigene Darstellung)	83
Abb. 4.5	Handling von erwarteten Exceptions. (Eigene Darstellung)	84

Abb. 4.6	Beispiel Parametrisierung JUnit. (Eigene Darstellung)	85
Abb. 4.7	Daten-Provider für komplexe Datentypen. (Eigene Darstellung)	86
Abb. 4.8	Parameterized Test. (Eigene Darstellung)	86
Abb. 4.9	BeforeEach. (Eigene Darstellung)	87
Abb. 4.10	Annotationen und Webbtests. (Eigene Darstellung)	88
Abb. 4.11	Beispiel einer ObjectMap. (Eigene Darstellung)	90
Abb. 4.12	Beispiel für generierten Code. (Eigene Darstellung)	90
Abb. 4.13	Beispieltest von Motorrad (Auszug; eigene Darstellung)	96
Abb. 4.14	Factory Pattern der Fahrzeuge. (Eigene Darstellung)	99
Abb. 4.15	IFactoryFahrzeuge. (Eigene Darstellung)	99
Abb. 4.16	FactoryAuto. (Eigene Darstellung)	99
Abb. 4.17	FahrzeugFactory. (Eigene Darstellung)	100
Abb. 4.18	Beispiel instanziierung neues Auto-Objekt. (Eigene Darstellung)	100
Abb. 4.19	Strategy Pattern Versicherungen. (Eigene Darstellung)	101
Abb. 4.20	VersicherungStrategy Class. (Eigene Darstellung)	101
Abb. 4.21	Anwendung Strategy Pattern mit Versicherung. (Eigene Darstellung)	102
Abb. 6.1	Servermanagement von WireMock mit JUnit 5 (eigene Darstellung)	126
Abb. 6.2	WireMock-Objekterzeugung (eigene Darstellung)	126
Abb. 6.3	Beispiel einer JSON-Antwort (eigene Darstellung)	127
Abb. 6.4	Testcode für WireMock-Testcase (eigene Darstellung)	128
Abb. 6.5	Mock-Test mit Parametern (eigene Darstellung)	128
Abb. 6.6	Beispiel von Verify (eigene Darstellung)	129

Abb. 6.7	Mocking auf Error Code 500 (eigene Darstellung)	131
Abb. 6.8	Timeout-Test mit WireMock (eigene Darstellung)	131
Abb. 7.1	Beispiel Javacode in HTML mit Java Server Pages. (Eigene Darstellung)	136
Abb. 7.2	Fahrzeug-Anlegen in der Fahrzeugverwaltung. (Eigene Darstellung)	137
Abb. 7.3	Anlegen eines Fahrzeugs erfolgreich. (Eigene Darstellung)	138
Abb. 7.4	Laden eines neuen Drivers mit System- Properties. (Eigene Darstellung)	140
Abb. 7.5	Beispiel eines Elements mit ID Tag. (Eigene Darstellung)	141
Abb. 7.6	Beispiel für einen Link, eingebettet im div-Container. (Eigene Darstellung)	143
Abb. 7.7	Beispiel für HTML Dokument. (Eigene Darstellung)	145
Abb. 7.8	Startpunkt ist das Feld mit der ID „marke“. (Eigene Darstellung)	146
Abb. 7.9	Beispiel von Entwicklertools in Firefox. (Eigene Darstellung)	148
Abb. 7.10	Ergebnis von SendKeys. (Eigene Darstellung)	148
Abb. 7.11	Erster Selenium-Test. (Eigene Darstellung)	150
Abb. 7.12	Dropdown-Feld mit Selenium bearbeiten. (Eigene Darstellung)	151
Abb. 7.13	Test mit PageObject Pattern. (Eigene Darstellung)	152
Abb. 7.14	PageFactory für PageObjects. (Eigene Darstellung)	153
Abb. 7.15	Element über @FindBy nutzen. (Eigene Darstellung)	154
Abb. 7.16	Laden eines Firefox-Drivers als Iphone 13 Pro Max. (Eigene Darstellung)	157
Abb. 7.17	Explizites Warten Beispiel. (Eigene Darstellung)	159

Abb. 7.18	Selenium und die Verwendung von Javascript. (Eigene Darstellung)	161
Abb. 7.19	Cookie-Test. (Eigene Darstellung)	162
Abb. 7.20	Screenshot Erstellen einer Methode. (Eigene Darstellung)	163
Abb. 7.21	Beispiel Methodenaufruf für Screenshots. (Eigene Darstellung)	164
Abb. 8.1	Erstes Cucumber-Feature. (Eigene Darstellung)	169
Abb. 8.2	Erstes Step File. (Eigene Darstellung)	170
Abb. 8.3	Tag-Beispiele. (Eigene Darstellung)	172
Abb. 8.4	Beispiel für eine Runner Class. (Eigene Darstellung)	173
Abb. 8.5	Beispiel Hooks in Cucumber. (Eigene Darstellung)	174
Abb. 8.6	Mit Hooks auf Tests reagieren. (Eigene Darstellung)	175
Abb. 8.7	Beispiel für Variablen in Featuredateien. (Eigene Darstellung)	175
Abb. 8.8	Step-Datei mit Variable. (Eigene Darstellung)	176
Abb. 8.9	Background Keyword in Cucumber. (Eigene Darstellung)	176
Abb. 8.10	Beispiel einer Datentabelle. (Eigene Darstellung)	177
Abb. 8.11	Verwendung von einfachen Datentabellen in Java. (Eigene Darstellung)	178
Abb. 8.12	Szenario-Outline Beispiel. (Eigene Darstellung)	179
Abb. 8.13	Kombination von Datentabellen und Example-Tabelle. (Eigene Darstellung)	179

Tabellenverzeichnis

Tab. 1.1	Beispiel Grenzwertanalyse	16
Tab. 2.1	Primitive Datentypen in Java.	32
Tab. 2.2	Boolsche Vergleichsoperatoren	37



Grundlagen des Softwaretestings

1

„Etwas zu testen bedeutet, eine Annahme kritisch auf Wahrheit zu prüfen“

Eigene Definition von Testing

Inhaltsverzeichnis

Einleitung	1
Ausgangslage – Das Projekt	2
Folgen der fehlenden Tests.	3
Die Aufgaben eines Testers	5
Grundlagen der Testarten	6
Testpyramide	7
Die Realität ist eine Eistüte	10
Testaufbau – Vorbedingung, Ausführung, Nachbedingung	11
Box-Testing	13
Kommunikation und der Umgang mit gefundenen Fehlern	17
Takeaways	18

Einleitung

Vor mehr als einem Jahrzehnt startete ich mein erstes Projekt als Softwaretester. Zu diesem Zeitpunkt hatte ich nahezu keine Erfahrungen in diesem Bereich. Klar war das Thema im Studium prä-

sent, aber die verschiedenen Testarten oder die Erstellung einer Teststrategie waren kein Bestandteil der Vorlesung. Überhaupt fehlte mir der erste Einstieg in die Thematik. Ich begann daher bei null und musste mir sämtliches Wissen zunächst aneignen.

Für das Projekt eines großen Kunden war es notwendig, auf Oberflächentests zurückzugreifen; dies war daher auch mein Fokus. Würde ich allerdings noch mal beginnen, wäre der erste Schritt, die Grundlagen kennenzulernen. Diese Grundlagen sind wichtig, um Fehler zu vermeiden, die richtige Testart für die richtige Problemstellung zu nutzen und ein gutes Fundament für die anstehenden Inhalte zu schaffen. Dieser Einstieg hilft auch dabei, komplexe Sachverhalte leicht zu verstehen, sodass aufbauendes Wissen schneller und leichter möglich ist.

In diesem Buch werde ich daher zuerst besagte Grundlagen des Softwaretestens erläutern, gängige Testarten vorstellen und die Programmiersprache Java aus Testsicht vermitteln. Es sind keine Vorkenntnisse notwendig, um den Erklärungen zu folgen. Die weiteren Kapitel umfassen dann Testautomatisierung und Testframeworks, Mocking und Oberflächentests. Den Abschluss bilden Behaviour Driven Development und exploratives Testen.

Ausgangslage – Das Projekt

Ein neues System einführen ohne zu testen? Was soll schon schiefgehen...

Vor langer Zeit war ich als Entwickler daran beteiligt, ein neues System für eine Onlinehandelsplattform aufzubauen. Grundlegend sollte ein Onlineshop eingeführt werden, der verschiedenste Schnittstellen zu Ebay, Amazon und diversen Zahlungsdienstleistern bietet. Das ganze Rechnungswesen wurde über SAP abgewickelt. Da der Onlineshop zur damaligen Zeit zwar eingekauft wurde, aber verschiedenste Features nicht mitlieferte, wurden diese von Entwicklern nach implementiert.

Beispielsweise das Gutscheinsystem erhielt zahlreiche neue Funktionen. All das erfolgte ohne Teststrategie und ohne ausreichende Tests.

Dieses Fehlen hatte weitreichende Folgen.

Folgen der fehlenden Tests

Die Angst vor Codeoptimierungen

Während der Entwicklung neuer Features wird vorhandener Programmcode oft verändert, so zum Beispiel bei der erwähnten Gutscheinfunktion. Eine Erweiterung sollte das Eingeben von mehreren Gutscheinen auf einmal ermöglichen. Der vorhandene Code wurde angepasst und dabei entstanden zahlreiche neue Zeilen mit mehr Funktionsinhalten. Nach der Featureentwicklung fanden aber keine Optimierung und keine Anpassungen mehr statt. Der Kunde klickte die Oberflächen durch und nahm die Funktion ab. Danach wurde dieses Feature nicht mehr verändert und der Code blieb in seiner Form bestehen mit allen unnötigen Programmzeilen und so chaotisch wie bisher. Schon eine Woche später fehlte jegliches Verständnis über die vorhandene Programmlogik.

Eigentlich entwickelt Programmcode sich stetig weiter, wird aufgeräumt und verbessert. Dieses sogenannte Refactoring findet immer wieder statt und dabei schleichen sich hin und wieder auch Fehler ein. Bei vorhandenen Tests fallen diese Bugs schnell und zeitnah auf, werden behoben und das Refactoring wird abgeschlossen. Sind keine Tests vorhanden, bleibt immer ein Risiko bestehen und eine damit verbundene Unsicherheit. Niemand möchte für Fehler verantwortlich sein und dadurch werden Codeoptimierungen wenig bis gar nicht mehr durchgeführt.

Fehler frühzeitig erkennen und beheben

Wird bei einer Abfragebedingung nur ein Größerzeichen statt einem Größer-gleich verwendet, liefert die Funktion schon ganz andere Ergebnisse. Solche Probleme bleiben ohne Tests zunächst unerkannt, da sie erst bei der genaueren Ergebnisbetrachtung auffallen. So war es auch beim Shop. Besagter Fehler trat auf und schon war der Minimalwert für die Anwendung des Gutscheins plötzlich falsch.

Für einen Gutschein, der einen Mindesteinlösewert von 10,00 oder mehr Euro hatte, sollte ein Rabatt von 10 % gewährt werden. Hat die einkaufende Person einen Warenkorbwert von 10,01 € geht die Funktion durch, bei 10,00 € erscheint eine Fehlermeldung. Ein kleiner Fehler, der zunächst unerkannt blieb, aber einige Einkäufe abbrechen ließ und somit für Umsatzverluste sorgte. Während des Entwickelns trat einfach kein gezielter Fall auf, bei dem der Gesamtwarenwert exakt 10 € betrug, alle waren weit drüber.

Nachdem die Meldung vom Kunden kam, musste zunächst der Fall rekonstruiert werden. Zunächst war nicht klar, wo genau die Fehlerursache lag. Erst nach längerer Analyse fiel das fehlende Gleichheitszeichen auf.

Menschen machen Fehler, so auch beim Softwareentwickeln. Aber diese Fehler werden erst zum Problem, wenn sie unerkannt bleiben. Tests in verschiedenen Varianten, mit verschiedensten Parametern helfen dabei, diese Defekte auch schon beim Entwickeln früh zu finden. Dadurch wird eine direkte Behebung auch gleich möglich und spart damit Zeit und Geld.

Kundenzufriedenheit und Firmenimage

Was passiert, wenn knapp hundert Personen gleichzeitig einen Shop besuchen und darauf einkaufen? Wie verhält es sich, wenn diese Zahl plötzlich auf tausend steigt?

Durch eine Marketingaktion mit verschiedenen Rabatten auf limitierte Sportschuhe wurden zahlreiche Besucher auf den Shop gelockt. Die Aktion war sehr erfolgreich, aus Serversicht zu erfolgreich. Um 24:00 Uhr nachts wurden die Gutscheine freigeschaltet und somit begann auch der Ansturm auf den Shop.

Die gleichzeitige Besucherzahl war für das Shopsystem ein großes Problem. Im Hintergrund fungierte eine MySQL-Datenbank als Datenhaltungsschicht, während verschiedene Schnittstellen mit SAP und weiteren Systemen kommunizierten. Irgendwann war der Webserver überlastet. Die Performance des Systems ging immer weiter zurück, bis schließlich der ganze Server zusammenbrach. Welche Komponente der Auslöser war, blieb

zunächst unerkannt und benötigte einige Zeit der Nachforschung. Nach zahlreichen Telefonaten mit dem Serverprovider und diverser Ermittlungsarbeit konnte die Datenbank identifiziert werden. Zwar wurden notwendige Einstellungen rasch umgesetzt, dennoch deutlich zu spät für die teure Marketingkampagne.

Entsprechende Kritik hagelte es auch von Shopbesuchern, die ihre Bestellungen nicht abschließen konnten. Kunden behalten solche Situationen im Gedächtnis und verbreiten sie weiter. Dies kann schnell zu Reputationsschäden führen. Unbekannt bleibt, wie viel potenzielle Kunden dadurch zur Konkurrenz wechselten.

Mit meinem heutigen Wissen und der Perspektive eines Testers würde ich dieses Projekt komplett anders angehen.

Die Aufgaben eines Testers

Eine der wichtigsten Aufgaben eines Softwaretesters, einer Softwaretesterin ist, die Qualität einer Software dauerhaft und nachhaltig zu verbessern. Dies gelingt durch die Entwicklung und Umsetzung einer Softwareteststrategie. Dabei werden Testkonzepte, Vorgehen und Methoden festgelegt. Meistens ist hierfür ein Testmanager verantwortlich. Außerdem gehören die Auswahl und Entwicklung von Testtools sowie Testframeworks in den Aufgabenbereich von Testern. Gerade Eigenentwicklungen sollten dabei wie ein Softwareprodukt behandelt werden, einer Architektur folgen und grundlegende Designvorgaben erfüllen. Häufig fällt dieser Aufgabenbereich in die Rolle des Testautomatisierers. Wichtig ist bei der Erstellung von Frameworks und Tools, dass dabei Flexibilität eingehalten und Komplexität minimiert wird. Mit genau diesen Themen beschäftigt sich das Kap. 4.

Zudem existiert die Rolle des Testanalysten. Dabei stehen Analyse, Planung und Erstellung von Tests im Vordergrund. Vorgaben werden analysiert und Testfälle mit unterschiedlichen Testmöglichkeiten sowie Testtypen hergeleitet. Diese können manuell oder automatisiert durchgeführt werden. Um eine Auswahl über die richtigen Testarten für die richtige Situation treffen zu können, werden diese grundlegend vorgestellt.