Xpert.press

Johannes Hubertz

Softwaretests mit Python

Springer Vieweg

Xpert.press

Weitere Bände in dieser Reihe
http://www.springer.com/series/4393

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Johannes Hubertz

Softwaretests mit Python

Springer Vieweg

Johannes Hubertz
Köln, Deutschland

ISSN 1439-5428
Xpert.press
ISBN 978-3-662-48602-3 ISBN 978-3-662-48603-0 (eBook)
DOI 10.1007/978-3-662-48603-0

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

Springer Vieweg

Satz: Johannes Hubertz mit LaTeXe

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Berlin Heidelberg ist Teil der Fachverlagsgruppe Springer Science+Business Media
(www.springer.com)

Vorwort

Die Idee zu diesem Buch entstand in einem Gespräch mit einem Freund. Nach erfolgreichem Abschluss einer umfangreichen, mehrmonatigen Programmiertätigkeit mit Python in der Qualitätssicherung eines Datenbankherstellers sollten die dabei gewonnenen Erfahrungen dokumentiert werden. Ziel war die Erweiterung des vorhandenen Testframeworks. Ein Nose-Plugin mit mehreren zehntausend Zeilen, welches das Produkt des Hauses auf Herz und Nieren automatisiert und kontinuierlich prüfen sollte, benötigte weitere Funktionalität. Zwei Jahre zuvor entstand ein Tutorial für die deutsche Python Konferenz 2013 mit dem Thema „Unittests für Einsteiger“. Eigene Entwicklungen mit Python und dem Ziel, Paketfilter für Linux performanter zu generieren und Latenzen zu minimieren, waren fertig und auf dem Weg in die Debian GNU/Linux Distribution.

Da das Thema IT-Sicherheit in vielen Facetten seit über 20 Jahren Beruf und Berufung meines Arbeitslebens ist, bildet verlässliche Software dabei die wichtigste Grundlage des Erfolges. Zuverlässige Software benötigt im gesamten Lebenszyklus vom Entstehen über den Betrieb bis zum Tag der Außerbetriebnahme auf Verlässlichkeit ausgerichtete Kenntnisse und Konzepte. Softwaretests, als kontinuierlicher Qualitätssicherungs- und gleichwohl IT-Sicherheits-Prozess, gewinnen dabei zunehmend an Bedeutung im Tagesgeschäft.

Die gesammelten Erfahrungen kommen dem Buch zugute. Es soll gestandenen Programmieren wie auch Anfängern Freude und Begeisterung fürs Testen von Software vermitteln, so dass sie auch Überzeugungstäter werden. Testgetriebene Entwicklung kann manche Fallstricke herkömmlicher Entwicklungsmodelle vermeiden. Zuverlässigkeit spielt in kritischen Infrastrukturen und hochverfügbaren Rechenzentren eine entscheidende Rolle und stiftet ganz automatisch auch anderweitig vielfachen Mehrwert. Nicht nur Entwickler, Systemadministratoren und Qualitätsmanager können von Softwaretests profitieren, sondern alle, die sich mit der Entstehung und dem Betrieb von Software beschäftigen, selbst wenn sie nur gelegentlich programmieren.

Johannes Hubertz — Köln, im Oktober 2015

Inhalt

1 Einleitung

> Als wir die Richtung endgültig aus den Augen verloren hatten, verdoppelten wir unsere Anstrengungen.
>
> *(Mark Twain, nach einem indischen Sprichwort)*

Überblick

Eine kurze Übersicht sei als Anfang erlaubt, um einen roten Faden und schnelle Orientierung im Buch zu geben.

Die Historie der Softwareentwicklung und des einhergehenden Qualitätsbewusstseins bilden den Einstieg ins Thema. Verschiedene Bezeichnungen werden eingeführt und die zugrunde liegende abstrakte Methodik zum Entwurf von Tests vorgestellt. Anschließend werden die praktischen Aspekte rund um das Aufsetzen eines Softwareprojektes kurz beleuchtet. Einige Hilfsmittel und Methoden werden benannt, die das Leben der Entwickler und Anwender erleichtern sollen. Die Dokumentation spielt im Zusammenhang mit der Qualität von Software eine immer wichtigere Rolle. Damit sind die Grundlagen einer erfolgreichen Projektarbeit gelegt. Dieses und die folgenden Kapitel schließen jeweils mit einem Interview ab. Die Fragen an freundliche Menschen aus dem Umfeld von Python wurden gerne aus der persönlichen Perspektive heraus beantwortet. Zum einen lockert dies den doch etwas trockenen Stoff auf. Andererseits treten so verschiedene Aspekte zutage, um als Anregung zu dienen. Die Antworten spiegeln nicht zwingend die Meinung des Autors wieder, sondern stets die der Interviewpartner. Übersetzung und Anpassungen wurden nach bestem Wissen und Gewissen sinnerhaltend vorgenommen.

J. Hubertz, *Softwaretests mit Python,* Xpert.press,
DOI 10.1007/978-3-662-48603-0_1, © Springer-Verlag Berlin Heidelberg 2016

Im zweiten Kapitel stellen einfache Testverfahren mit dem *Doctest-Modul* innerhalb des Produktionscodes den Einstieg ins Testen dar. Sie eignen sich insbesondere, weil sie untrennbar mit dem Produkt verbunden sind und so zum Verständnis eben dessen beitragen können. Neben dem Verfahren werden verschiedene Tricks und Kniffe vorgestellt.

Das dritte Kapitel ist dem *Unittest*-Modul gewidmet. Einige Besonderheiten werden am Beispiel gezeigt. Davon werden einige bereits mit dem Aufruf durch das Hilfsprogramm *Nose* ausgeführt, weil es die Benutzung auf der Kommandozeile erheblich vereinfacht. Da dieses Hilfsmittel deutlich mehr vermag, als nur Unittests einzusammeln und auszuführen, stellt das vierte Kapitel die anderen Vorzüge bis hin zur Erweiterbarkeit mit der Plugin-Schnittstelle dar.

Ein weiteres Hilfsmittel namens *pytest*, aus dem Nose vor einigen Jahren durch eine Abspaltung hervorging, wurde im Laufe der letzten Jahre stark weiterentwickelt und füllt das fünfte Kapitel. Die vielfältigen Möglichkeiten können jedoch nur angedeutet werden. Eine vollständige Beschreibung würde ein ganzes Buch erfordern, die Vorstellung eines installierbaren Plugins schließt mit den dazugehörigen Tests ab.

Fingerübungen in den Kapiteln zu Unittests und pytest ergänzen den Stoff um praxisnahe Aspekte. Die gezeigten Programmteile sollen dem Verständnis dienen, wie und wozu die Tests nützlich sind und wie diese mit pytest einfacher und eleganter funktionieren.

Das sechste Kapitel stellt das Hilfsprogramm *tox* vor. Es dient dazu, beliebige Tests mit verschiedenen Python-Versionen jeweils in eigens dazu erstellten virtuellen Umgebungen auszuführen. Auch die Einhaltung der PEP8-Konformität lässt sich damit prüfen. Im Sinne der Qualitätssicherung dient es vorwiegend der automatisierten Durchführung jeweils aller Tests.

Im ersten Teil des siebten Kapitels werden Tests zu einer grafischen Benutzeroberflächen angerissen. Wegen wachsender Bedeutung wird danach umfassender auf Software fürs Web am Beispiel von *Django* eingegangen. Der Beginn einer testgetriebenen Entwicklung zeigt, wie und mit welchen Testwerkzeugen die serverseitige Software zuverlässig erstellt werden kann. Dazu werden Unittest und funktionaler Test in der Kombination dargestellt.

Das achte und letzte Kapitel stellt zwei voneinander unabhängige, umfangreiche Python-Projekte unter freien Lizenzen vor, *SaltStack* und *OpenStack keystone,* welches nur einen Bruchteil des fast ausschließlich in Python ge-

schriebenen OpenStack-Universums ausmacht. Beide Projekte mit vielen tausend Codezeilen verfügen neben dem Produktionscode über tausende Testfälle, die mit eindrucksvollen Zahlen belegt werden. Das letzte Interview mit einem der OpenStack-Entwickler macht deutlich, dass in sehr großen Projekten mit vielen Entwicklern nichts anderes als testgetriebene Entwicklung vorstellbar ist. Dies gilt sogar dann, wenn die Teams klein bleiben.

1.1 Die Softwarekrise

Schon in den 1960er Jahren war das Wort Softwarekrise in der Branche gefürchtet, glaubte man doch um die Ursachen zu wissen. Zunehmende Komplexität und Rechenkapazität führten zu stets komplizierter werdenden Programmen, die über Jahre, teils sogar über Jahrzehnte, gepflegt und dabei unbeherrschbar wurden. Fehlende Qualitätsstandards und mangelnde Dokumentation wurden vielerorts zum Problem. Dies wurde insbesondere dann deutlich, wenn Mitarbeiter die Aufgaben wechselten, lange nicht an einem Programm gearbeitet wurde oder Nachfolger eingearbeitet werden mussten. Die Probleme eskalierten, Auswege wurden und werden bis heute gesucht. Laut Wikipedia [Wik15a] wurde bereits 1968 der Begriff *Software Engineering* auf einer NATO-Tagung geprägt. In den „Communications of the ACM (Mitteilungen der Association for Computing Machinery)“ findet sich 1972 der Begriff in der Dankesrede von Edsger W. Dijkstra zur Verleihung des Turing Award, veröffentlicht als „Der bescheidene Programmierer“, im Original und als deutsche Übersetzung ebenda:

> Zitat:
> Die Hauptursache für die Softwarekrise liegt darin begründet, dass die Maschinen um einige Größenordnungen mächtiger geworden sind! Um es ziemlich einfach auszudrücken: Solange es keine Maschinen gab, war Programmierung kein existierendes Problem; als wir ein paar schwache Computer hatten, wurde Programmierung zu einem geringen Problem, und nun, da wir gigantische Computer haben, ist die Programmierung ein ebenso gigantisches Problem.
> Edsger Dijkstra: The Humble Programmer

Der Trend, sowohl die Komplexität der Software als auch die Leistungsfähigkeit der verwendeten Computer zu steigern, hält bis heute an. Ross

Anderson ist eine Ikone im Bereich Sicherheit und Zuverlässigkeit von IT-Systemen. Er schreibt 2008: [And08] „Der vermutlich größte Fortschritt des Software Engineerings ist die Technologie von Softwaretests in den 1990er Jahren."

Vor diesem Fortschritt waren Regressionstests üblich, das heißt die Ausgaben eines Testlaufs einer neueren Version werden mit den Ausgaben einer älteren Version bei gleichen Eingaben verglichen. Die Software wurde dabei komplexer und nur durch fortwährende Tests zuverlässiger. Speziell bei Software für Sicherheit, zum Beispiel in Steuerungen, die auch immer komplexer und umfangreicher wurden, war dieses nur unter wachsenden Aufwänden zu erreichen. So etablierte sich bald, Teams für die beiden Aufgaben Test- und Wirksoftware zu bilden.

Verschiedene Methoden wurden im Laufe der Zeit propagiert und in der Breite angewandt. Eine ultimative Methode, die alle Probleme gleichermaßen löst, lässt noch auf sich warten. In den letzten Jahrzehnten wurden vorgestellt (entnommen aus *Basiswissen Sichere Software* [Pau11]):

- Das *Wasserfallmodell* ist das klassische Entwicklungsmodell mit einer streng sequenziellen Bearbeitung: Anforderungserfassung, Entwurf, Programmierung, Test und Einrichtung.
- Das *V-Modell* (und die Weiterentwicklung *V-Modell XT*) ist eine Abwandlung des Wasserfallmodells. Anforderungen werden zu Beginn festgelegt. Nachträgliche Änderungen sind aus Prinzip nicht vorgesehen.
- *Prototyp*-getriebene Entwicklungsmodelle, wie zum Beispiel „Rapid Application Development" entwickelt in Folgen von Prototypen, deren jeweilige Anforderungen immer kurz vor der Entwicklung des Prototyps festgelegt werden. Ganzheitliche Betrachtungsweise des Produktes ist aus Prinzip nicht erwünscht.
- *Spiralmodell* entwickelt inkrementell in Schleifen durch verbleibende Projektrisiken.
- *Stark strukturierte Methoden,* zum Beispiel „Model-Driven Development" oder „Rational Unified Process" von IBM machen alle Entwicklungsschritte nachvollziehbar auf Kosten einer streng sequenziellen Vorgehensweise. Methoden zum Nachweis der Korrektheit und Sicherheit machen diese Methoden noch starrer als das Wasserfallmodell.

- *Agile Methoden* bilden einen Gegensatz zu allen oben genannten, weil sie größten Wert auf Flexibilität legen. *Scrum, Extreme Programming* und *Test-Driven Development* lassen die Qualität der entwickelten Funktionalitäten schnell nachvollziehen.

Alle diese Methoden verfolgen das Ziel, Qualität und Stabilität des entwickelten Produktes zu steigern wie auch Entwicklungskosten zu minimieren. Unabhängig von der Methode gilt: Je später in der Entwicklungsphase von Software ein Fehler erkannt wird, um so teurer ist seine Behebung, da die Anzahl der Codezeilen mit der Komplexität einhergeht und so den Korrekturaufwand steigern. Insofern ist es naheliegend, mit automatisierten Tests möglichst früh, am Besten gleich zu Anfang, zu beginnen. Genau das ist die Idee hinter dem Begriff „Test-Driven Development“, oft abgekürzt als *TDD*.

1.2 Motivation für Softwaretests

Die Entwicklung eines Softwareproduktes ist aufwendig und stets fehlerbehaftet. Insbesondere ist nur in wenigen, modernen Programmiersprachen eine Beweisführung auf Fehlerfreiheit möglich. Daher ist nicht selten der gute Glaube an die Produktqualität reine Illusion. Dennoch wird der gewissenhafte Softwareentwickler dafür Sorge tragen, dass seine Programme möglichst keine oder jedenfalls wenig Fehler enthalten. Um Fehler aufzudecken, eignet sich Software, die genau zu diesem Zweck entwickelt wird. Wenn diese bereits vor der Fertigstellung des eigentlichen Programms schon die richtigen und falschen Ergebnisse automatisch prüfen kann, also die Rahmenbedingungen feststehen, dann ist die Entwicklung der gewünschten Funktion eben erst genau dann fertig, wenn *alle* Ergebnisse geprüft und für richtig befunden sind. Das führt zur Unterscheidung zwischen Produktions- und Testcode. Letzterer dient dazu, Ersteren auf korrektes Verhalten zu untersuchen. Beide Arten können in verschiedenen Dateien gepflegt werden. Ebenso ist es möglich, den jeweiligen Testcode in einer Datei zusammen mit dem Produktionscode aufzubewahren. Bei größeren Projekten mag auch die Teilung in verschiedene Verzeichnisse sinnvoll sein. Dies wird später näher beleuchtet.

Doch auch nach einer Auslieferung von Software können Fehler entdeckt werden, im ungünstigsten Fall durch den Kunden. Aus jedem vom Entwickler akzeptierten Fehlerreport lässt sich ein Testfall konstruieren, und zu-