O'REILLY®

Architektur-patterns mit Python

Test-Driven Development, Domain-Driven Design und Event-Driven Microservices praktisch umgesetzt

Harry J. W. Percival
& Bob Gregory
Übersetzung von Thomas Demmig

Architekturpatterns mit Python

Test-Driven Development, Domain-Driven Design und Event-Driven Microservices praktisch umgesetzt

Harry Percival, Bob Gregory

Deutsche Übersetzung von Thomas Demmig

O'REILLY®

Harry Percival, Bob Gregory

Lektorat: Ariane Hesse
Übersetzung: Thomas Demmig
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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.

ISBN:
Print 978-3-96009-165-3
PDF 978-3-96010-572-5
ePub 978-3-96010-573-2
mobi 978-3-96010-574-9

1. Auflage 2021

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition *Architecture Patterns with Python*
ISBN 9781492052203 © 2020 Harry Percival and Bob Gregory.

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«.
O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen.
PEFC™
PEFC/04-31-0810
www.pefc.de

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

5 4 3 2 1 0

Inhalt

Vorwort

Sie fragen sich vielleicht, wer wir sind und warum wir dieses Buch geschrieben haben.

Am Ende von Harrys letztem Buch *Test-Driven Development with Python* (O'Reilly, *http://www.obeythetestinggoat.com/*) hat er ein paar Fragen rund um Architektur gestellt – zum Beispiel, auf welchem Weg Sie Ihre Anwendung am besten so strukturieren können, dass sie sich leicht testen lässt. Genauer gesagt, geht es darum, wie Ihre zentrale Businesslogik durch Unit Tests abgedeckt werden kann und wie Sie die Menge an erforderlichen Integrations- und End-to-End-Tests minimieren können. Er verwies wolkig auf »hexagonale Architektur«, »Ports und Adapter« sowie »funktionaler Kern, imperative Shell«, aber er musste ehrlich gesagt zugeben, dass er all das nicht so richtig verstanden oder ernsthaft eingesetzt hatte.

Aber wie es der Zufall so will, traf er auf Bob, der Antworten auf all diese Fragen hatte.

Bob war Softwarearchitekt geworden, weil das kein anderer in seinem Team machen wollte. Es stellte sich heraus, dass er das nicht wirklich gut konnte, aber er wiederum traf glücklicherweise auf Ian Cooper, der ihm neue Wege zeigte, Code zu schreiben und darüber nachzudenken.

Komplexität managen, Businessprobleme lösen

Wir arbeiten beide für MADE.com, ein europäisches E-Commerce-Unternehmen, das Möbel über das Internet verkauft. Dort wenden wir die Techniken aus diesem Buch an, um verteilte Systeme aufzubauen, die Businessprobleme aus der Realität modellieren. Unsere Beispieldomäne ist das erste System, das Bob für MADE geschaffen hat, und dieses Buch ist der Versuch, all das aufzuschreiben, was wir neuen Programmiererinnen und Programmierern beibringen müssen, wenn sie in eines unserer Teams kommen.

MADE.com agiert mit einer globalen Lieferkette aus Frachtpartnern und Herstellern. Um die Kosten niedrig zu halten, versuchen wir, die Bestände in unseren Lagern so zu optimieren, dass Waren nicht lange herumliegen und Staub ansetzen.

Idealerweise wird das Sofa, das Sie kaufen wollen, an genau dem Tag im Hafen eintreffen, an dem Sie sich zum Kauf entscheiden, und wir werden es direkt zu Ihnen liefern, ohne es überhaupt einlagern zu müssen.

Dieses Timing richtig hinzubekommen, ist ein überaus kniffliger Balanceakt, wenn die Produkte drei Monate benötigen, bis sie mit dem Containerschiff eintreffen. Auf dem Weg gehen Dinge kaputt, oder es gibt einen Wasserschaden, Stürme können zu unerwarteten Verzögerungen führen, Logistikpartner gehen nicht gut mit den Waren um, Papiere gehen verloren, Kunden ändern ihre Meinung und passen ihre Bestellung an und so weiter.

Wir lösen diese Probleme, indem wir intelligente Software bauen, die die Aktionen aus der realen Welt repräsentieren, sodass wir so viel wie möglich automatisieren können.

Warum Python?

Wenn Sie dieses Buch lesen, müssen wir Sie wahrscheinlich nicht davon überzeugen, dass Python großartig ist, daher ist die eigentliche Frage: »Warum braucht die Python-Community solch ein Buch?« Die Antwort liegt in der Beliebtheit und dem Alter von Python: Auch wenn sie die vermutlich weltweit am schnellsten wachsende Programmiersprache ist und sich in die Spitze der Tabellen vorarbeitet, kommt sie erst jetzt langsam in den Bereich der Probleme, mit denen sich die C#- und die Java-Welt seit Jahren beschäftigen. Start-ups werden zu ernsthaften Firmen, Webanwendungen und geskriptete Automatisierungshelferlein werden zu (im Flüsterton) *Enterprisesoftware*.

In der Welt von Python zitieren wir häufig das Zen von Python: »Es sollte einen – und möglichst genau einen – offensichtlichen Weg geben, etwas zu tun.«[1] Leider ist mit wachsender Projektgröße der offensichtlichste Weg nicht immer der Weg, der Ihnen dabei hilft, die Komplexität und die sich wandelnden Anforderungen im Griff zu behalten.

Keine der Techniken und Patterns, die wir in diesem Buch besprechen, ist insgesamt neu, aber sie sind es größtenteils für die Python-Welt. Und dieses Buch ist kein Ersatz für Klassiker wie Eric Evans *Domain-Driven Design* oder Martin Fowlers *Patterns of Enterprise Application Architecture* (beide bei Addison-Wesley Professional veröffentlicht) – im Gegenteil, wir beziehen uns oft darauf und raten Ihnen, sie ebenfalls zu lesen.

Aber all die klassischen Codebeispiele in der Literatur sind doch meist in Java oder C++/C# geschrieben, und wenn Sie eher eine Python-Person sind und schon lange nicht mehr (oder gar noch nie) eine dieser Sprachen genutzt haben, können diese Codebeispiele doch ziemlich – wie soll man sagen – herausfordernd sein. Es gibt einen Grund dafür, dass die Beispiele in der neuesten Auflage eines weiteren Klassikers – Fowlers *Refactoring* (Addison-Wesley Professional) – in JavaScript geschrieben sind.

1 `python -c "import this"`

TDD, DDD und eventgesteuerte Architektur

In der Reihenfolge ihrer Bekanntheit gibt es drei Werkzeuge, um die Komplexität im Griff zu behalten:

1. *Test-Driven Development* (TDD) hilft uns dabei, Code zu erstellen, der korrekt ist, und es ermöglicht uns, Features zu refaktorieren oder neu hinzuzufügen, ohne dass wir Angst vor Regressionen haben müssen. Aber es kann sehr schwer sein, das Beste aus unseren Tests herauszuholen: Wie stellen wir sicher, dass sie so schnell wie möglich laufen? Dass wir so viel Abdeckung und Feedback wie möglich aus schnellen, unabhängigen Unit Tests bekommen und so wenig langsamere, anfällige End-to-End-Tests wie möglich einsetzen müssen?
2. *Domain-Driven Development* (DDD) hilft uns dabei, unsere Arbeit darauf zu konzentrieren, ein gutes Modell der Businessdomäne zu erstellen. Aber wie schaffen wir es, dass unsere Modelle nicht mit Infrastrukturbedenken überladen werden und sich nur schwer ändern lassen?
3. Lose gekoppelte (Micro-)Services, die über Nachrichten miteinander kommunizieren (manchmal als *reaktive Microservices* bezeichnet), sind eine bewährte Antwort auf den Umgang mit Komplexität über mehrere Anwendungen oder Businessdomänen hinweg. Aber es ist nicht immer offensichtlich, wie Sie sie mit den bekannten Tools der Python-Welt – Flask, Django, Celery und so weiter – aufeinander abstimmen können.

Machen Sie sich keine Sorgen, wenn Sie nicht mit Microservices arbeiten (oder auch gar kein Interesse daran haben). Der größte Teil der hier besprochenen Patterns – auch viele aus der eventgesteuerten Architektur – lässt sich ebenfalls wunderbar in einer monolithischen Architektur anwenden.

Wir haben uns mit diesem Buch zum Ziel gesetzt, eine Reihe klassischer Architektur-Patterns vorzustellen und zu zeigen, wie sie TDD, DDD und eventgesteuerte Services unterstützen. Wir hoffen, dass es als Referenz für ihre Implementierung auf pythoneske Art und Weise dient und dass die Menschen es als ersten Schritt für weitere Untersuchungen auf diesem Gebiet nutzen können.

Wer dieses Buch lesen sollte

Es gibt ein paar Dinge, die wir von unserem Publikum annehmen:

- Sie hatten bereits mit ein paar halbwegs komplexen Python-Anwendungen zu tun.
- Sie haben schon die Schmerzen erlebt, die entstehen, wenn man versucht, die Komplexität im Griff zu behalten.
- Sie müssen nicht unbedingt etwas über DDD oder eines der klassischen Architektur-Patterns wissen.

Wir strukturieren unsere Erkundung der Architektur-Patterns rund um eine Beispiel-App, die wir Kapitel für Kapitel aufbauen. In unserem Hauptjob verwenden wir TDD, daher tendieren wir dazu, erst Listings mit Tests zu zeigen, auf die dann Implementierungen folgen. Sind Sie nicht damit vertraut, erst Tests, dann Implementierungen zu schreiben, mag das zu Beginn ein wenig seltsam erscheinen, aber wir hoffen, dass Sie sich schnell daran gewöhnen werden, »verwendeten« Code zu sehen, bevor Sie erfahren, wie er im Inneren aufgebaut ist.

Wir nutzen ein paar Python-Frameworks und -Technologien, unter anderem Flask, SQLAlchemy und pytest, dazu Docker und Redis. Sind Sie damit schon vertraut, ist das schön, aber unserer Meinung nach nicht unbedingt erforderlich. Eines unserer Hauptziele bei diesem Buch besteht darin, eine Architektur aufzubauen, für die spezifische Technologieentscheidungen zu unwichtigeren Implementierungsdetails werden.

Was lernen Sie in diesem Buch?

Das Buch ist in zwei Teile unterteilt – hier ein Überblick über die Themen, die wir behandeln, und die Kapitel, in denen Sie sie finden werden.

Teil I: Eine Architektur aufbauen, die Domänenmodellierung unterstützt

Domänenmodellierung und DDD (Kapitel 1 und 7)
: Wir alle haben mehr oder weniger die Lektion gelernt, dass sich komplexe Businessprobleme im Code widerspiegeln müssen – in Form eines Modells oder einer Domäne. Aber warum scheint es immer so schwierig zu sein, das umzusetzen, ohne sich in Infrastrukturbedenken, unseren Web-Frameworks oder in sonst etwas zu verheddern? Im ersten Kapitel werden wir einen allgemeinen Überblick über *Domänenmodellierung* und *DDD* geben und zeigen, wie Sie mit einem Modell ohne externe Abhängigkeiten und schnelle Unit Tests loslegen können. Später kehren wir zu DDD-Patterns zurück, um zu zeigen, wie Sie die richtigen Aggregate wählen und wie diese Wahl mit Fragen zur Datenintegrität im Zusammenhang steht.

Repository-, Service-Layer- und Unit-of-Work-Patterns (Kapitel 2, 4 und 5)
: In diesen drei Kapiteln stellen wir drei eng miteinander verbundene und sich gegenseitig unterstützende Patterns vor, die uns dabei helfen, das Modell frei von zusätzlichen Abhängigkeiten zu halten. Wir bauen eine Abstraktionsschicht für einen persistenten Storage auf und erstellen einen Service Layer, um die Zugangspunkte für unser System zu definieren und die wichtigsten Use Cases abzubilden. Wir zeigen, wie es diese Schicht einfach macht, schlanke Zugangspunkte zu unserem System zu schaffen – sei es eine Flask-API oder ein CLI.

Gedanken zum Testen und zu Abstraktionen (Kapitel 3 und 6)
: Nachdem wir die erste Abstraktion vorgestellt haben (das Repository-Pattern), nutzen wir die Gelegenheit zu einer allgemeinen Diskussion über das Auswählen von Abstraktionen und ihre Rolle bei der Entscheidung zum Koppeln unserer Software. Nachdem wir das Service-Layer-Pattern vorgestellt haben, reden wir ein bisschen über das Erschaffen einer Testpyramide und das Schreiben von Unit Tests als größtmögliche Abstraktion.

Teil II: Eventgesteuerte Architektur

Eventgesteuerte Architektur (Event-Driven Architecture, Kapitel 8 bis 11)
: Wir stellen drei sich gegenseitig unterstützende Patterns vor: Domain Events, Message Bus und Handler. *Domain Events* sind eine Umsetzung der Idee, dass bestimmte Interaktionen mit einem System Auslöser für andere sind. Wir nutzen einen *Message Bus*, um damit Aktionen Events auslösen und zugehörige *Handler* aufrufen zu lassen. Dann kümmern wir uns darum, wie Events als Patterns für die Integration zwischen Services in einer Microservices-Architektur zum Einsatz kommen können. Schließlich unterscheiden wir zwischen *Befehlen* und *Events*. Unsere Anwendung ist nun im Grunde ein System, das Nachrichten verarbeitet.

Command-Query Responsibility Segragation (Kapitel 12)
: Wir stellen ein Beispiel für eine Command-Query Responsibility Segregation vor – mit und ohne Events.

Dependency Injection (Kapitel 13)
: Wir räumen unsere expliziten und impliziten Abhängigkeiten auf und implementieren ein einfaches Dependency Injection Framework.

Zusätzliche Inhalte

Wie es weitergeht (Epilog)
: Das Implementieren von Architektur-Patterns sieht immer einfach aus, wenn Sie es an einem einfachen Beispiel vorgestellt bekommen und ohne vorhandenen Code starten, aber viele werden sich sicherlich fragen, wie sie diese Prinzipien auf bestehende Software anwenden. Wir geben im Epilog ein paar Hinweise auf weiteres Material und nennen Ihnen einige Links dazu.

Beispielcode und Programmieren

Sie lesen gerade ein Buch, aber vermutlich sind Sie auch unserer Meinung, dass man am besten etwas über das Programmieren lernt, wenn man programmiert. Das meiste von dem, was wir wissen, haben wir durch die Zusammenarbeit mit anderen gelernt, durch das gemeinsame Schreiben von Code und durch Learning by Doing – und wir würden diese Erfahrung in diesem Buch für Sie gern wiederholbar machen.

Daher haben wir das Buch rund um ein einzelnes Beispielprojekt aufgebaut (auch wenn wir manchmal auf andere Beispiele zurückgreifen). Dieses Projekt wächst mit jedem Kapitel – so als würden wir uns zusammensetzen und Ihnen erklären, was wir warum in jedem Schritt tun.

Aber um mit diesen Patterns wirklich vertraut zu werden, müssen Sie sich selbst die Hände am Code schmutzig machen und ein Gefühl dafür bekommen, wie er funktioniert. Sie finden ihn vollständig auf GitHub – jedes Kapitel hat dort seinen eigenen Branch. Eine Liste dieser Branches gibt es ebenfalls auf GitHub unter *https://github.com/cosmicpython/code/branches/all*.

Dies sind drei Möglichkeiten, mit dem Code zum Buch zu arbeiten:

- Erstellen Sie Ihr eigenes Repository und versuchen Sie, die App genauso wie wir aufzubauen, indem Sie den Beispielen aus den Listings im Buch folgen und sich gelegentlich Hinweise durch einen Blick in unser Repository holen. Ein Wort der Warnung sei aber angebracht: Haben Sie schon Harrys vorheriges Buch gelesen und mit dessen Code gearbeitet, werden Sie feststellen, dass Sie dieses Mal mehr selbst herausfinden müssen – eventuell müssen Sie deutlich mehr auf die funktionierenden Versionen in GitHub zurückgreifen.
- Versuchen Sie, Kapitel für Kapitel jedes Pattern auf Ihr eigenes Projekt (möglichst ein kleines oder ein Spielprojekt) anzuwenden und es für Ihren Anwendungsfall möglichst gut einzusetzen. Das Risiko ist hier deutlich höher (ebenso der Aufwand!), aber die Ergebnisse sind es wert. Es kann anstrengend sein, die Dinge an die Besonderheiten Ihres Projekts anzupassen, aber andererseits lernen Sie so vermutlich am meisten.
- Ist Ihnen das zu viel Aufwand, finden Sie in jedem Kapitel eine Übung mit einem Verweis auf GitHub, wo Sie teilweise fertiggestellten Code für das Kapitel herunterladen und die fehlenden Teile ergänzen können.

Vor allem wenn Sie einige der Patterns auf Ihre eigenen Projekte anwenden wollen, ist das Durcharbeiten eines einfachen Beispiels ein sicherer Weg, um Praxiserfahrung zu sammeln.

Führen Sie beim Lesen eines Kapitels zumindest ein `git checkout` aus. Wenn Sie sich den Code einer tatsächlich funktionierenden App anschauen können, beantwortet das schon viele Fragen und sorgt für deutlich mehr Realitätsnähe. Anweisungen dazu finden Sie am Beginn jedes Kapitels.

Lizenz

Der Code (und die englischsprachige Online-Version des Buchs) steht unter einer Creative Commons CC BY-NC-ND-Lizenz, was bedeutet, dass Sie ihn frei kopieren und mit anderen zu nicht-kommerziellen Zwecken teilen können, solange Sie die Quelle angeben. Wenn Sie Inhalte aus diesem Buch wiederverwenden möch-

ten und Bedenken bezüglich der Lizenz haben, kontaktieren Sie O'Reilly unter *permissions@oreilly.com*. Die Druckausgabe ist anders lizenziert; bitte lesen Sie die Copyright-Seite.

In diesem Buch genutzte Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch genutzt:

Kursiv
: Für neue Begriffe, URLs, E-Mail-Adressen, Dateinamen und Dateierweiterungen.

`Nichtproportionalschrift`
: Für Programm-Listings, aber auch für Codefragmente in Absätzen, wie zum Beispiel Variablen- oder Funktionsnamen, Datenbanken, Datentypen, Umgebungsvariablen, Anweisungen und Schlüsselwörter.

`Fette Nichtproportionalschrift`
: Für Befehle und anderen Text, der genau so vom Benutzer eingegeben werden sollte.

`Kursive Nichtproportionalschrift`
: Für Text, der vom Benutzer durch eigene Werte ersetzt werden sollte.

Dieses Symbol steht für einen Tipp oder Vorschlag.

Dieses Symbol steht für eine allgemeine Anmerkung.

Dieses Symbol steht für eine Warnung oder Vorsichtsmaßnahme.

Danksagung

An unsere Technologiegutachter David Seddon, Ed Jung und Hynek Schlawack: Wir haben euch wirklich nicht verdient. Ihr seid alle unglaublich engagiert, gewissenhaft und gründlich. Jeder von euch ist unfassbar klug, und eure unterschiedlichen Sichtweisen waren für die anderen gleichzeitig nützlich und ergänzend. Wir danken euch von ganzem Herzen.

Ein riesiger Dank geht auch an die Leserinnen und Leser des Early Release für ihre Anmerkungen und Vorschläge: Ian Cooper, Abdullah Ariff, Jonathan Meier, Gil Gonçalves, Matthieu Choplin, Ben Judson, James Gregory, Łukasz Lechowicz, Clinton Roy, Vitorino Araújo, Susan Goodbody, Josh Harwood, Daniel Butler, Liu Haibin, Jimmy Davies, Ignacio Vergara Kausel, Gaia Canestrani, Renne Rocha, pedroabi, Ashia Zawaduk, Jostein Leira, Brandon Rhodes und viele mehr – wir bitten um Verzeihung, wenn wir jemanden vergessen haben.

Einen Super-mega-Dank an unseren Lektor Corbin Collins für sein freundliches Drängeln und für sein fortlaufendes Eintreten für die Leserinnen und Leser. Ein genauso großer Dank geht an das Produktionsteam Katherine Tozer, Sharon Wilkey, Ellen Troutman-Zaig und Rebecca Demarest für das Engagement, die Professionalität und das Auge fürs Detail. Dieses Buch hat sich dadurch unbeschreiblich verbessert.

Alle verbleibenden Fehler in diesem Buch gehen natürlich trotzdem auf unsere Kappe.

Einleitung

Warum schlagen unsere Designs fehl?

Was kommt Ihnen in den Sinn, wenn Sie das Wort »Chaos« hören? Vielleicht denken Sie an einen hektischen Börsensaal oder an Ihre Küche am Morgen – alles ist durcheinander und unordentlich. Wenn Sie dagegen an das Wort »Ordnung« denken, stellen Sie sich vielleicht einen leeren und ruhigen Raum vor. Für die Wissenschaft wird *Chaos* allerdings durch Homogenität (Gleichheit) charakterisiert, während sich *Ordnung* durch Komplexität (Unterschiedlichkeit) auszeichnet.

So ist beispielsweise ein gepflegter Garten ein sehr geordnetes System. Gärtner definieren Grenzen durch Pfade und Zäune, und sie stecken Blumen- oder Gemüsebeete ab. Mit der Zeit entwickelt sich der Garten weiter, er wird reichhaltiger und ist dichter bewachsen, aber ohne sorgfältige Arbeiten wird er zu einer Wildnis werden. Gräser werden andere Pflanzen überwuchern und die Wege bedecken, bis schließlich alle Teile gleich aussehen – wild und unkontrolliert.

Softwaresysteme tendieren ebenfalls zum Chaos. Beginnen wir damit, ein neues System zu bauen, haben wir großartige Ideen dazu, dass unser Code sauber und ordentlich werden wird, aber mit der Zeit stellen wir fest, dass sich Müll und seltsame Grenzfälle ansammeln, was sich schließlich zu einem verwirrenden Morast aus Managerklassen und Hilfsmodulen ausbildet. Wir merken, dass unsere sorgsam in Schichten erstellte Architektur wie ein zu feuchtes Trifle in sich zusammengefallen ist. Chaotische Softwaresysteme zeichnen sich durch eine Gleichheit von Funktionen aus: API-Handler, die Domänenwissen besitzen, E-Mails verschicken und Logging durchführen, »Businesslogik«-Klassen, die keine Berechnungen durchführen, sondern I/O vornehmen – und alles ist mit allem gekoppelt, sodass ein Ändern eines Teils des Systems sehr gefährlich wird. Das geschieht so häufig, dass man in der Softwareentwicklung einen eigenen Begriff für Chaos hat: das Antipattern *Big Ball of Mud* (siehe Abbildung T-1).

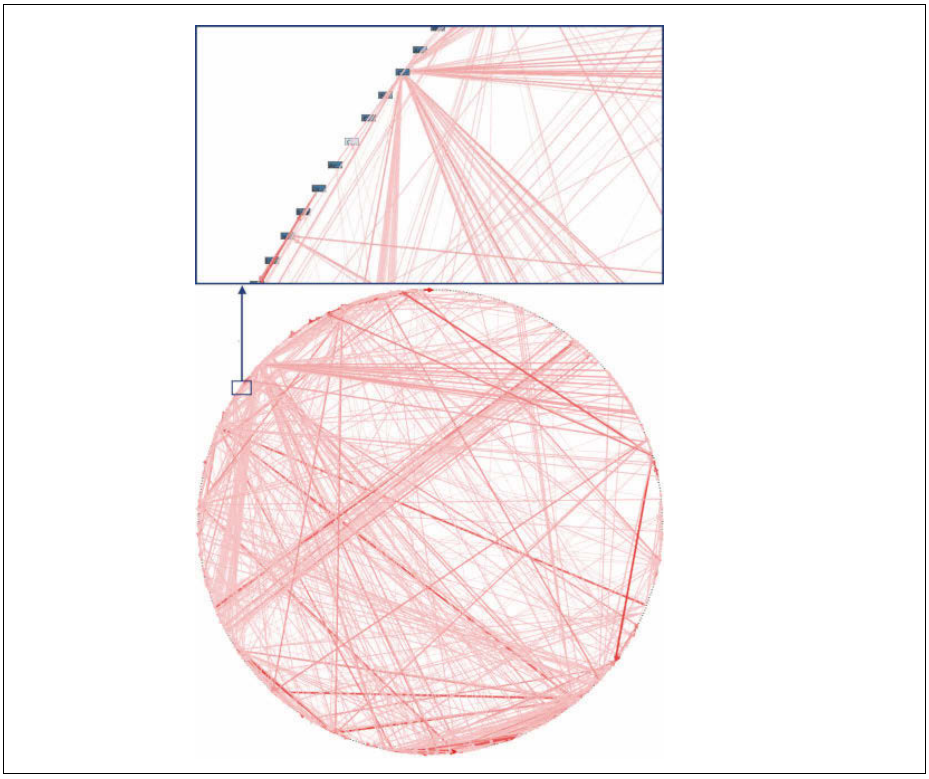

Abbildung T-1: Ein Abhängigkeitsdiagramm aus dem echten Leben (Quelle: »Enterprise Dependency: Big Ball of Yarn« von Alex Papadimoulis, https://oreil.ly/dbGTW)

Ein Big Ball of Mud ist für Software genauso der natürliche Zustand, wie Wildnis der natürliche Zustand Ihres Gartens ist. Es bedarf Energie und Orientierung, um den Kollaps zu verhindern.

Zum Glück sind die Techniken, mit denen sich ein Big Ball of Mud vermeiden lässt, gar nicht so komplex.

Kapselung und Abstraktionen

Kapselung und Abstraktion sind Werkzeuge, auf die wir beim Programmieren alle instinktiv zurückgreifen, auch wenn wir nicht immer genau diese Begriffe verwenden. Schauen wir sie uns ein wenig genauer an, da sie im Buch immer wieder vorkommen.

Der Begriff *Kapselung* beschreibt zwei eng miteinander verbundene Ideen: das Vereinfachen von Verhalten und das Verbergen von Daten. Hier soll es um Ersteres ge-

hen. Wir kapseln Verhalten, indem wir eine Aufgabe identifizieren, die in unserem Code erledigt werden muss, und diese Aufgabe an ein wohldefiniertes Objekt oder eine Funktion geben. Dieses Objekt oder diese Funktion nennen wir dann eine *Abstraktion*.

Schauen Sie sich die folgenden zwei Python-Codeabschnitte an:

Mit urllib suchen

```
import json
from urllib.request import urlopen
from urllib.parse import urlencode

params = dict(q='Sausages', format='json')
handle = urlopen('http://api.duckduckgo.com' + '?' + urlencode(params))
raw_text = handle.read().decode('utf8')
parsed = json.loads(raw_text)

results = parsed['RelatedTopics']
for r in results:
    if 'Text' in r:
        print(r['FirstURL'] + ' - ' + r['Text'])
```

Mit requests suchen

```
import requests

params = dict(q='Sausages', format='json')
parsed = requests.get('http://api.duckduckgo.com/', params=params).json()

results = parsed['RelatedTopics']
for r in results:
    if 'Text' in r:
        print(r['FirstURL'] + ' - ' + r['Text'])
```

In beiden Codeabschnitten wird die gleiche Aufgabe erledigt: Sie schicken Formcodierte Werte an eine URL, um die API einer Suchmaschine zu nutzen. Aber der zweite Code ist einfacher zu lesen und zu verstehen, weil er auf einer höheren Abstraktionsebene operiert.

Wir können das sogar weitertreiben, indem wir die umzusetzende Aufgabe identifizieren, mit einem Namen versehen und eine noch höhere Abstraktion nutzen, um das Ganze noch expliziter zu machen:

Mit dem Modul duckduckgo suchen

```
import duckduckgo
for r in duckduckgo.query('Sausages').results:
    print(r.url + ' - ' + r.text)
```

Das Kapseln von Verhalten durch den Einsatz von Abstraktionen ist ein leistungsfähiges Werkzeug, mit dem Code ausdrucksstärker, testbarer und leichter wartbar gemacht wird.

In der objektorientierten (OO-)Literatur wird eine der klassischen Charakterisierungen dieses Vorgehens als *Responsible-Driven Design* bezeichnet (*http://www.wirfs-brock.com/Design.html*) – es nutzt die Begriffe *Rollen (Roles)* und *Verantwortlichkeiten (Responsibilities)* statt *Aufgaben (Tasks)*. Entscheidend ist dabei, den Code anhand seines Verhaltens zu betrachten und nicht anhand der Daten oder Algorithmen.[1]

Abstraktionen und ABCs

In einer klassischen OO-Sprache wie Java oder C# nutzen Sie vielleicht eine abstrakte Basisklasse (*Abstract Base Class*, ABC) oder ein Interface, um eine Abstraktion zu definieren. In Python können Sie ABCs nutzen (was wir tatsächlich manchmal tun), aber Sie können sich auch ganz aufs Duck Typing verlassen.

Die Abstraktion kann einfach »die öffentliche API des eingesetzten Dings« bedeuten – zum Beispiel ein Funktionsname plus ein paar Argumente.

Bei einem Großteil der Patterns in diesem Buch gehört auch das Auswählen einer Abstraktion dazu, daher werden Sie in jedem Kapitel viele Beispiele finden. Zudem geht es in Kapitel 3 speziell um ein paar allgemeine Heuristiken zur Auswahl von Abstraktionen.

Layering

Kapseln und Abstrahieren helfen uns dabei, Details zu verbergen und die Konsistenz unserer Daten zu schützen, aber wir müssen auch auf die Interaktionen zwischen unseren Objekten und Funktionen achten. Nutzt eine Funktion, ein Modul oder ein Objekt ein anderes, sagen wir, dass das eine vom anderen *abhängt*. Diese Abhängigkeiten bilden eine Art von Netzwerk oder einen Graphen.

In einem Big Ball of Mud sind die Abhängigkeiten außer Kontrolle geraten (wie Sie in Abbildung T-1 gesehen haben). Es wird schwierig, einen Knoten dieses Graphen zu ändern, weil er potenziell viele andere Teile des Systems beeinflusst. Eine Architektur in Schichten ist eine Möglichkeit, dieses Problem anzugehen. Dabei teilen wir unseren Code in getrennte Kategorien oder Rollen auf und führen Regeln dazu ein, welche Codekategorien sich jeweils aufrufen können.

Eines der am häufigsten anzutreffenden Beispiele dafür ist die Drei-Schichten-Architektur, die in Abbildung T-2 dargestellt wird.

1 Wenn Sie einmal mit Class-Responsibility-Collaborator-(CRC-)Karten zu tun haben – da geht es um das Gleiche: Indem Sie über *Verantwortlichkeiten* nachdenken, hilft Ihnen das dabei, zu entscheiden, wie Sie etwas aufteilen.

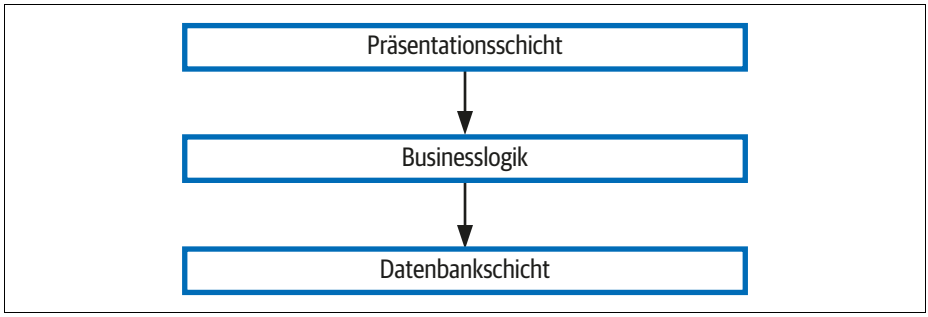

Abbildung T-2: Schichtenarchitektur

Die Schichtenarchitektur ist das vielleicht am häufigsten eingesetzte Pattern für das Erstellen von Businesssoftware. In diesem Modell haben wir Komponenten für die Benutzeroberfläche, bei denen es sich um eine Webseite, eine API oder eine Befehlszeile handeln kann. Diese UI-Komponenten kommunizieren mit einer Businesslogikschicht, die unsere Businessregeln und Workflows enthält. Und schließlich haben wir eine Datenbankschicht, die für das Speichern und Bereitstellen von Daten verantwortlich ist.

Im Rest dieses Buchs werden wir dieses Modell systematisch auseinandernehmen, indem wir ein einfaches Prinzip befolgen.

Das Dependency-Inversion-Prinzip

Vielleicht sind Sie schon mit dem *Dependency-Inversion-Prinzip* (DIP) vertraut, denn dabei handelt es sich um das *D* in SOLID[2].

Leider können wir das DIP nicht durch drei kurze Codeabschnitte darstellen, wie wir das für das Kapseln getan haben. Aber der gesamte Teil I ist mehr oder weniger ein ausgearbeitetes Beispiel für das Implementieren des DIP in einer Anwendung, daher werden Sie dort konkrete Beispiele finden.

Bis dahin können wir uns zumindest über die formale Definition des DIP unterhalten:

1. Module auf höherer Ebene sollten nicht von Modulen auf niedrigerer Ebene abhängen. Beide sollten von Abstraktionen abhängen.
2. Abstraktionen sollten nicht von Details abhängen. Stattdessen sollten Details von Abstraktionen abhängen.

Aber was heißt das? Schauen wir es uns im Einzelnen an.

2 SOLID ist ein Akronym für die fünf Prinzipien objektorientierten Designs von Robert C. Martin: Single Responsibility, Open for Extension but Closed for Modification, Liskov Substituion, Interface Segragation und Dependency Inversion. Siehe »S.O.L.I.D.: The First 5 Principles of Object-Oriented Design« (*https://oreil.ly/UFM7U*) von Samuel Oloruntoba.

Module auf höherer Ebene sind der Code, um den sich Ihre Organisation wirklich kümmert. Vielleicht arbeiten Sie für ein Pharmaunternehmen, und Ihre Module auf höherer Ebene kümmern sich um Patienten und Studien. Oder Sie arbeiten für eine Bank, und Ihre High-Level-Module managen Aktien und Währungen. Die High-Level-Module eines Softwaresystems sind die Funktionen, Klassen und Pakete, die sich mit unseren Konzepten aus der realen Welt befassen.

Im Gegensatz dazu sind die *Module auf niedrigerer Ebene* der Code, um den sich Ihre Organisation nicht kümmert. Es ist unwahrscheinlich, dass Ihre Personalabteilung für Dateisysteme oder Netzwerk-Sockets brennt. Und Sie werden auch nicht so oft mit Ihrer Finanzabteilung über SMTP, HTTP oder AMQP sprechen. Unsere nicht technischen Stakeholder interessieren sich nicht für diese Low-Level-Konzepte, oder sie sind für sie nicht relevant. Sie kümmern sich nur darum, ob die High-Level-Konzepte funktionieren. Wenn die Gehaltsabrechnung pünktlich ist, wird sich Ihr Business nicht dafür interessieren, ob das ein Cron-Job oder eine transiente Funktion ist, die auf Kubernetes läuft.

Hängt ab von heißt nicht unbedingt *importiert* oder *ruft auf*, sondern ist eher eine allgemeinere Idee davon, dass ein Modul von einem anderen *weiß* oder es *benötigt*.

Und wir haben schon *Abstraktionen* erwähnt: Sie vereinfachen Schnittstellen, die Verhalten kapseln – so wie unser `duckduckgo`-Modul die API einer Suchmaschine kapselt.

> In der Informatik lassen sich alle Probleme lösen, indem man noch eine Indirektionsschicht hinzufügt.
>
> – David Wheeler

Der erste Teil des DIP sagt also, dass unser Businesscode nicht von technischen Details abhängen sollte, sondern beide stattdessen Abstraktionen einsetzen sollten.

Warum? Nun, ganz allgemein, weil wir sie unabhängig voneinander ändern können wollen. Module auf höherer Ebene sollten sich als Reaktion auf Businessanforderungen leicht anpassen lassen. Module auf niedrigerer Ebene (Details) lassen sich in der Praxis häufig nur schwerer ändern: Denken Sie an das Refaktorieren eines Funktionsnamens im Gegensatz zum Definieren, Testen und Deployen einer Datenbankmigration, um einen Spaltennamen anzupassen. Wir wollen nicht, dass Änderungen an der Businesslogik verzögert werden, weil sie eng mit Infrastrukturdetails auf niedriger Ebene verzahnt ist. Aber genauso ist es wichtig, Ihre Infrastrukturdetails ändern zu *können*, wenn das notwendig ist (zum Beispiel weil Sie eine Datenbank sharden müssen), ohne Änderungen an Ihrer Businessschicht vornehmen zu müssen. Durch das Hinzufügen einer Abstraktion zwischen beidem (die berühmte zusätzliche Indirektionsschicht) können sich beide unabhängig(er) voneinander ändern.

Der zweite Teil ist sogar noch mysteriöser. »Abstraktionen sollten nicht von Details abhängen« scheint klar zu sein, aber »Details sollten von Abstraktionen abhängen« lässt sich nur schwer vorstellen. Wie können wir eine Abstraktion haben,

die nicht von den Details abhängt, die sie abstrahiert? Wenn wir bei Kapitel 4 angelangt sind, werden wir ein konkretes Beispiel haben, das das ein bisschen klarer machen sollte.

Ein Platz für all unsere Businesslogik: das Domänenmodell

Aber bevor wir unsere Drei-Schichten-Architektur auseinandernehmen können, müssen wir uns eingehender über die mittlere Schicht unterhalten – die High-Level-Module oder die Businesslogik. Einer der am häufigsten anzutreffenden Gründe für ein nicht funktionierendes Design ist, dass sich die Businesslogik über die Schichten der Anwendung verteilt und sie damit nur schwer zu identifizieren, verstehen und zu ändern ist.

In Kapitel 1 sehen Sie, wie Sie eine Businessschicht mit einem *Domain-Model*-Pattern bauen. Die restlichen Patterns in Teil I zeigen, wie wir das Domain Model leicht anpassbar und frei von Low-Level-Dingen halten, indem wir die richtigen Abstraktionen wählen und immer wieder das DIP anwenden.

TEIL I

Eine Architektur aufbauen, die Domänenmodellierung unterstützt

> Die meisten Entwickler haben noch nie ein Domänenmodell gesehen, sondern höchstens ein Datenmodell.
>
> – Cyrille Martraire, DDD EU 2017

Die meisten Menschen aus der Entwicklung, mit denen wir über Architektur sprechen, haben irgendwie das Gefühl, dass es besser gehen könnte. Sie versuchen häufig, ein System zu retten, das irgendwo in die falsche Richtung gelaufen ist, und wollen den Ball of Mud wieder mit etwas Struktur versehen. Sie wissen, dass ihre Businesslogik nicht über die ganze Anwendung verteilt sein sollte, aber sie haben keine Idee, wie sie das lösen sollen.

Wir haben festgestellt, dass viele Entwicklerinnen und Entwickler, die gebeten werden, ein neues System zu designen, sofort damit beginnen, ein Datenbankschema aufzubauen, und das Objektmodell erst danach betrachten. Bereits damit fängt das Projekt an, in die falsche Richtung zu laufen, denn stattdessen *sollte das Verhalten an erster Stelle stehen und Grundlage für unsere Storage-Anforderungen sein* – schließlich kümmern sich unsere Kundinnen und Kunden nicht um das Datenmodell. Ihnen ist nur wichtig, was das System tut – ansonsten würden sie einfach eine Tabellenkalkulation verwenden.

Der erste Teil des Buchs kümmert sich darum, wie man mithilfe von TDD (*Test-Driven Development*) ein umfassendes Objektmodell aufbaut (in Kapitel 1), anschließend erläutern wir Ihnen, wie Sie dieses Modell von technischen Aspekten getrennt halten können. Wir zeigen Ihnen, wie Sie Code erstellen, der sich nicht um Persistenz kümmert, und wie Sie stabile APIs rund um unsere Domäne schaffen, sodass wir aggressiv refaktorieren können.

Dazu stellen wir vier zentrale Design-Patterns vor:

- Das Repository-Pattern (Kapitel 2) – eine Abstraktion über die Idee von persistentem Storage.
- Das Service-Layer-Pattern (Kapitel 4), mit dem klar definiert wird, wo die Grenzen unserer Use Cases liegen.

- Das Unit-of-Work-Pattern (Kapitel 6), um atomare Operationen zu ermöglichen.
- Das Aggregate-Pattern (Kapitel 7), um die Integrität unserer Daten sicherzustellen.

Würden Sie sich gern ein Bild dessen machen, was da vor uns liegt, schauen Sie sich Abbildung I-1 an. Es sollte Sie aber nicht beunruhigen, wenn Sie davon noch nichts verstehen! Wir stellen in diesem Teil des Buchs Schritt für Schritt jeden Block aus dieser Abbildung vor.

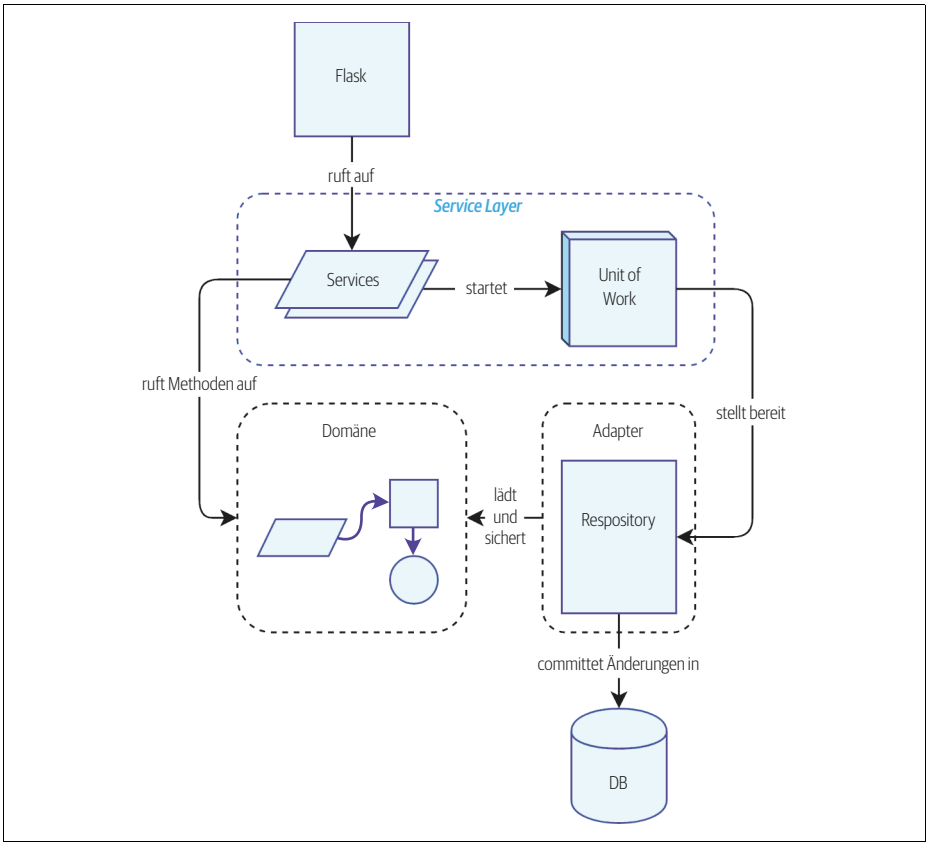

Abbildung I-1: Ein Komponentendiagramm für unsere Anwendung am Ende von Teil I

Wir nehmen uns zudem ein wenig Zeit, um in Kapitel 3 über Kopplung und Abstraktionen zu sprechen, und illustrieren das mit einem einfachen Beispiel, das zeigt, wie und warum wir unsere Abstraktionen gewählt haben.

Drei Anhänge gehen noch tiefer auf einzelne Aspekte der Inhalte von Teil I ein:

- In Anhang B ist die Infrastruktur für unseren Beispielcode zusammengestellt: wie wir die Docker-Images bauen und ausführen, wo wir Konfigurationsinformationen verwalten und wie wir verschiedene Testtypen ausführen.
- In Anhang C finden Sie eine Art »Proof of Concept«, der zeigt, wie einfach es ist, unsere gesamte Infrastruktur auszutauschen – die Flask-API, das ORM und Postgres –, um ein komplett anderes I/O-Modell zu erhalten, das ein CLI und CSVs nutzt.
- Anhang D kann für Sie schließlich interessant sein, wenn Sie sich fragen, wie diese Patterns aussehen würden, wenn statt Flask und SQLAlchemy nun Django zum Einsatz käme.