

Vorwort	11
Einleitung	19

Teil I Eine Architektur aufbauen, die Domänenmodellierung unterstützt

1 Domänenmodellierung	31
Was ist ein Domänenmodell?	31
Die Domänensprache untersuchen	34
Unit Testing für Domänenmodelle	36
Dataclasses sind großartig für Value Objects	41
Value Objects und Entitäten.	42
Nicht alles muss ein Objekt sein: eine Domänenservice-Funktion	44
Pythons magische Methoden lassen uns unsere Modelle mit idiomatischem Python nutzen	45
Auch Exceptions können Domänenkonzepte ausdrücken.	46
2 Repository-Pattern	49
Unser Domänenmodell persistieren	50
Etwas Pseudocode: Was werden wir brauchen?	50
DIP auf den Datenzugriff anwenden	51
Erinnerung: unser Modell	52
Der »normale« ORM-Weg: Das Modell hängt vom ORM ab	53
Die Abhängigkeit umkehren: ORM hängt vom Modell ab.	54
Das Repository-Pattern	57
Das Repository im Abstrakten	58
Vor- und Nachteile	59
Es ist nicht einfach, ein Fake-Repository für Tests zu erstellen!	63
Was ist in Python ein Port und was ein Adapter?	63
Zusammenfassung.	64

3	Ein kleiner Exkurs zu Kopplung und Abstraktionen	67
	Das Abstrahieren eines Status verbessert die Testbarkeit	68
	Die richtige(n) Abstraktion(en) wählen	71
	Unsere gewählten Abstraktionen implementieren	73
	Edge-to-Edge-Tests mit Fakes und Dependency Injection	75
	Warum nicht einfach herauspatchen?	77
	Zusammenfassung	79
4	Unser erster Use Case: Flask-API und Serviceschicht	81
	Unsere Anwendung mit der echten Welt verbinden	83
	Ein erster End-to-End-Test	83
	Die direkte Implementierung	84
	Fehlerbedingungen, die Datenbank-Checks erfordern	86
	Einführen eines Service Layer und Einsatz von FakeRepository für die Unit Tests	87
	Eine typische Servicefunktion	89
	Warum wird alles als Service bezeichnet?	92
	Dinge in Ordnern ablegen, um zu sehen, wohin sie gehören	92
	Zusammenfassung	94
	Das DIP in Aktion	94
5	TDD hoch- und niedertourig	97
	Wie sieht unsere Testpyramide aus?	98
	Sollten Tests der Domänenschicht in den Service Layer verschoben werden?	98
	Entscheiden, was für Tests wir schreiben	99
	Hoch- und niedertourig	100
	Tests für den Service Layer vollständig von der Domäne entkoppeln	101
	Linderung: alle Domänenabhängigkeiten in Fixture-Funktionen unterbringen	102
	Einen fehlenden Service hinzufügen	102
	Die Verbesserung in die E2E-Tests bringen	103
	Zusammenfassung	105
6	Unit-of-Work-Pattern	107
	Die Unit of Work arbeitet mit dem Repository zusammen	109
	Eine UoW über Integrationstests voranbringen	110
	Unit of Work und ihr Context Manager	111
	Die echte Unit of Work nutzt SQLAlchemy-Sessions	111
	Fake Unit of Work zum Testen	112

Die UoW im Service Layer einsetzen	114
Explizite Tests für das Commit/Rollback-Verhalten	114
Explizite versus implizite Commits	115
Beispiele: mit UoW mehrere Operationen in einer atomaren Einheit gruppieren	116
Beispiel 1: Neuzuteilung von Aufträgen	116
Beispiel 2: Chargengröße ändern	117
Die Integrationstests aufräumen	117
Zusammenfassung	118
7 Aggregate und Konsistenzgrenzen	121
Warum nehmen wir nicht einfach eine Tabellenkalkulation?	122
Invarianten, Constraints und Konsistenz	122
Invarianten, Concurrency und Sperren	123
Was ist ein Aggregat?	124
Ein Aggregat wählen	125
Ein Aggregat = ein Repository	128
Und was ist mit der Performance?	129
Optimistische Concurrency mit Versionsnummern	130
Optionen für Versionsnummern implementieren	132
Unsere Regeln zur Datenintegrität testen	134
Concurrency-Regeln durch den Einsatz von Isolation Level für Datenbanktransaktionen sicherstellen	135
Beispiel zur pessimistischen Concurrency-Steuerung: SELECT FOR UPDATE	135
Zusammenfassung	136
Teil I – Zusammenfassung	137

Teil II Eventgesteuerte Architektur

8 Events und der Message Bus	143
Vermeiden Sie ein Chaos	144
Zuerst einmal vermeiden wir ein Chaos in unseren Webcontrollern	145
Unser Modell soll auch nicht chaotisch werden	145
Vielleicht im Service Layer?	146
Single Responsibility Principle	146
Alles einsteigen in den Message Bus!	147
Das Modell zeichnet Events auf	147
Events sind einfache Dataclasses	147
Das Modell wirft Events	148
Der Message Bus bildet Events auf Handler ab	149

Option 1: Der Service Layer übernimmt Events aus dem Modell und gibt sie an den Message Bus weiter	150
Option 2: Der Service Layer wirft seine eigenen Events	151
Option 3: Die UoW gibt Events an den Message Bus	152
Zusammenfassung	156
9 Ab ins Getümmel mit dem Message Bus	159
Eine neue Anforderung bringt uns zu einer neuen Architektur	160
Stellen wir uns eine Architekturänderung vor: Alles wird ein Event-Handler sein.	161
Servicefunktionen in Message-Handler refaktorisieren.	163
Der Message Bus sammelt jetzt Events von der UoW ein	165
Die Tests sind ebenfalls alle anhand von Events geschrieben	166
Ein vorübergehender hässlicher Hack: Der Message Bus muss Ergebnisse zurückgeben.	167
Unsere API für die Arbeit mit Events anpassen	167
Unsere neue Anforderung implementieren.	168
Unser neues Event	169
Test-Drive für einen neuen Handler	169
Implementierung	170
Eine neue Methode im Domänenmodell	171
Optional: isolierte Unit Tests für Event-Handler mit einem Fake-Message-Bus.	172
Zusammenfassung	174
Was haben wir erreicht?.	175
Warum haben wir das erreicht?.	175
10 Befehle und Befehls-Handler	177
Befehle und Events	177
Unterschiede beim Exception Handling.	179
Events, Befehle und Fehlerbehandlung.	181
Synchrones Wiederherstellen aus Fehlersituationen	184
Zusammenfassung	186
11 Eventgesteuerte Architektur: Events zum Integrieren von Microservices	187
Distributed Ball of Mud und Denken in Nomen	188
Fehlerbehandlung in verteilten Systemen.	191
Die Alternative: temporales Entkoppeln durch asynchrone Nachrichten	192
Einen Redis Pub/Sub Channel zur Integration verwenden.	193

Mit einem End-to-End-Test alles überprüfen	194
Redis ist ein weiterer schlanker Adapter für unseren Message Bus	195
Unser neues Event in die Außenwelt	196
Interne und externe Events	197
Zusammenfassung	197
12 Command-Query Responsibility Segregation (CQRS)	199
Domänenmodelle sind zum Schreiben da	200
Die meisten Kundinnen und Kunden werden Ihre Möbel nicht kaufen	201
Post/Redirect/Get und CQS	203
Ruhe bewahren!	205
CQRS-Views testen	205
»Offensichtliche« Alternative 1: Das bestehende Repository verwenden	206
Ihr Domänenmodell ist nicht für Leseoperationen optimiert	207
»Offensichtliche« Alternative 2: Verwenden des ORM	208
SELECT N+1 und andere Performanceüberlegungen	208
Ziehen wir die Reißleine	209
Eine Tabelle im Lesemodell mit einem Event-Handler aktualisieren	210
Es ist einfach, die Implementierung unseres Lesemodells zu verändern	212
Zusammenfassung	214
13 Dependency Injection (und Bootstrapping)	215
Implizite und explizite Abhängigkeiten	217
Sind explizite Abhängigkeiten nicht total schräg und javaesk?	218
Handler vorbereiten: manuelles DI mit Closures und Partials	220
Eine Alternative mit Klassen	222
Ein Bootstrap-Skript	223
Der Message Bus bekommt die Handler zur Laufzeit	225
Bootstrap in unseren Einstiegspunkten verwenden	227
DI in unseren Tests initialisieren	227
Einen Adapter »sauber« bauen: ein größeres Beispiel	229
Abstrakte und konkrete Implementierungen definieren	229
Eine Fake-Version für die Tests erstellen	230
Wie führen wir einen Integrationstest durch?	231
Zusammenfassung	232

Epilog	235
Anhang A Übersichtsdiagramm und -tabelle	253
Anhang B Eine Template-Projektstruktur	255
Anhang C Austauschen der Infrastruktur: alles mit CSVs	263
Anhang D Repository- und Unit-of-Work-Pattern mit Django	269
Anhang E Validierung	279
Index	289