KAPITEL 2

Repository-Pattern

Es ist nun an der Zeit, unser Versprechen wahr zu machen und das Dependency-Inversion-Prinzip zum Entkoppeln unserer Logistik von Infrastrukturdingen einzusetzen.

Wir werden das *Repository*-Pattern vorstellen, bei dem es sich um eine vereinfachende Abstraktion eines Daten-Storage handelt, mit dem wir unsere Modellschicht von der Datenschicht entkoppeln können. Wir werden ein konkretes Beispiel dazu zeigen, mit dessen Hilfe Sie sehen, wie diese vereinfachende Abstraktion unser System testbarer macht, indem es die Komplexitäten der Datenbank verbirgt.

In Abbildung 2-1 sehen Sie eine kleine Vorschau auf das, was wir bauen werden: ein `Repository`-Objekt, das zwischen unserem Domänenmodell und der Datenbank sitzt.

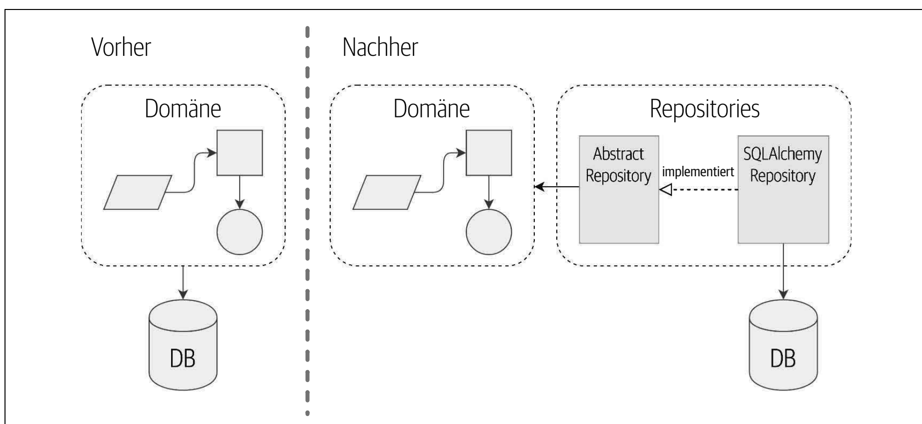
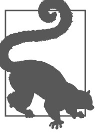


Abbildung 2-1: Vor und nach dem Repository-Pattern



Den Code für dieses Kapitel finden Sie im Branch chapter_02_repository auf GitHub unter *https://oreil.ly/6STDu*.

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_02_repository
# Oder führen Sie, um selbst zu programmieren, einen
# Check-out für das vorige Kapitel durch:
git checkout chapter_01_domain_model
```

Unser Domänenmodell persistieren

In Kapitel 1 haben wir ein simples Domänenmodell gebaut, das Aufträge Waren-Chargen zuteilen kann. Es ist einfach, Tests für diesen Code zu schreiben, weil es keine Abhängigkeiten oder einzurichtende Infrastruktur dafür gibt. Müssten wir eine Datenbank oder API betreiben und Testdaten erzeugen, wäre das viel schwieriger zu erstellen und zu warten.

Leider werden wir irgendwann unser perfektes kleines Modell in die Hände der Anwenderinnen und Anwender geben und mit der Realität aus Spreadsheets, Webbrowsern und Race Conditions konfrontieren müssen. In den nächsten paar Kapiteln werden wir uns anschauen, wie wir unser idealisiertes Domänenmodell mit externen Status verbinden können.

Wir gehen davon aus, dass wir agil arbeiten, daher liegt unsere Priorität darin, so schnell wie möglich ein Minimum Viable Product zu realisieren. In unserem Fall wird das eine Web-API sein. In einem echten Projekt würden Sie vielleicht direkt mit ein paar End-to-End-Tests starten und ein Web-Framework einsetzen, mit dem Sie alles von Kopf bis Fuß testen könnten.

Aber wir wissen, dass wir auf jeden Fall irgendeine Form von persistentem Storage brauchen. Und weil das hier ein Lehrbuch ist, können wir es uns erlauben, ein wenig mehr von unten nach oben vorzugehen und zunächst über Storage und Datenbanken nachzudenken.

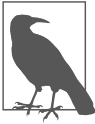
Etwas Pseudocode: Was werden wir brauchen?

Wenn wir unseren ersten API-Endpunkt bauen, wissen wir, dass wir auch Code haben werden, der mehr oder weniger wie der folgende aussieht:

Wie unser erster API-Endpunkt aussehen wird

```
@flask.route.gubbins
def allocate_endpoint():
    # Auftragszeile aus Request auslesen
    line = OrderLine(request.params, ...)
    # alle Batches aus der DB laden
    batches = ...
    # unseren Domänenservice aufrufen
    allocate(line, batches)
```

```
    # dann die Zuteilung irgendwo in der DB speichern
    return 201
```



Wir haben Flask genutzt, weil es schlank ist, aber Sie müssen es nicht einsetzen, um dieses Buch zu verstehen. Tatsächlich werden wir Ihnen zeigen, wie Sie die Wahl Ihres Frameworks zu einem nebensächlichen Detail machen.

Wir werden eine Möglichkeit brauchen, Batch-Informationen aus der Datenbank zu holen und unsere Domänenmodell-Objekte damit zu instanziieren, und wir werden sie auch wieder zurück in die Datenbank schreiben können müssen.

Was? Ach so, »gubbins« ist ein britisches Wort für »Kram«. Sie können es einfach ignorieren. Schließlich ist es Pseudocode, okay?

DIP auf den Datenzugriff anwenden

Wie in der Einleitung erwähnt, handelt es sich bei einer geschichteten Architektur um einen verbreiteten Ansatz zum Strukturieren eines Systems mit einer Benutzeroberfläche, einer Logik und einer Datenbank (siehe Abbildung 2-2).

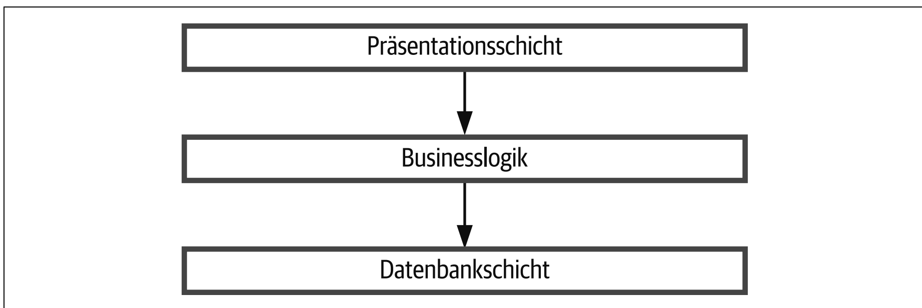


Abbildung 2-2: Schichtarchitektur

Die Model-View-Template-Struktur von Django steht damit in einem engen Zusammenhang – und ebenso *Model-View-Controller* (MVC). Jedes Mal ist es das Ziel, die Schichten getrennt zu halten (was eine gute Sache ist) und jede Schicht nur von der direkt darunter abhängig sein zu lassen.

Aber wir wollen, dass unser Domänenmodell *gar keine Abhängigkeiten besitzt*.[1] Wir wollen nicht, dass Infrastrukturaspekte in unser Domänenmodell sickern und unsere Unit Tests oder unsere Änderungsmöglichkeiten verlangsamen.

1 Ich gehe davon aus, dass wir »keine zustandsbehafteten Abhängigkeiten« meinen. Eine Abhängigkeit von einer Hilfsbibliothek ist in Ordnung, eine Abhängigkeit von einem ORM oder einem Web-Framework nicht.

Stattdessen stellen wir uns unser Modell als »innen liegend« vor, in das die Abhängigkeiten hineinfließen – das wird manchmal als *Zwiebelarchitektur* bezeichnet (siehe Abbildung 2-3).

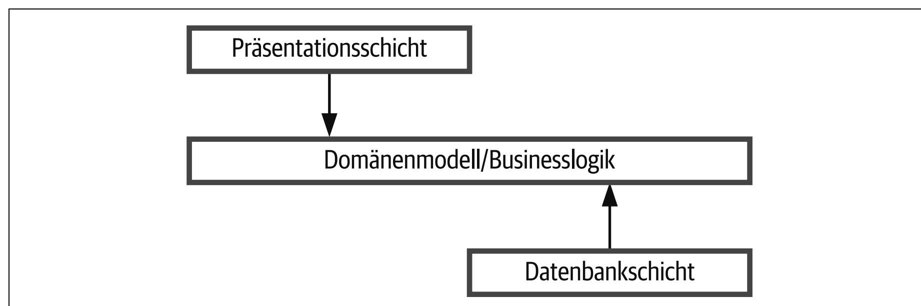


Abbildung 2-3: Zwiebelarchitektur

Geht es hier um Ports und Adapter?

Wenn Sie über Architektur-Patterns lesen, stellen Sie sich vielleicht Fragen wie die folgenden:

> Geht es hier um Ports und Adapter? Oder ist das hexagonale Architektur? Ist es das Gleiche wie Zwiebelarchitektur? Was ist ein Port und was ein Adapter? Warum haben die Leute so viele Begriffe für das gleiche Ding?

Auch wenn sich manche Menschen gern an den kleinen Unterschieden aufhängen, sind es alles mehr oder weniger nur unterschiedliche Namen für das Gleiche, und sie alle lassen sich auf das *Dependency Inversion Principle* zurückführen: High-Level-Module (die Domäne) sollten nicht von Low-Level-Modulen (der Infrastruktur) abhängen.[2]

Wir werden weiter unten im Buch in *Von Abstraktionen abhängen* auf Seite 90 noch auf die Details rund um die Abhängigkeit von Abstraktionen und ein pythoneskes Äquivalent von Schnittstellen eingehen. Werfen Sie auch einen Blick auf *Was ist in Python ein Port und was ein Adapter?* auf Seite 63.

Erinnerung: unser Modell

Erinnern wir uns noch mal an unser Domänenmodell (siehe Abbildung 2-4): Eine Zuteilung beziehungsweise Allocation ist das Konzept der Verbindung einer `OrderLine` mit einem `Batch`. Wir speichern diese Zuteilungen als eine Collection in unserem `Batch`-Objekt.

2 Mark Seeman hat zu diesem Thema unter *https://oreil.ly/LpFS9* einen ausgezeichneten Blogpost geschrieben.

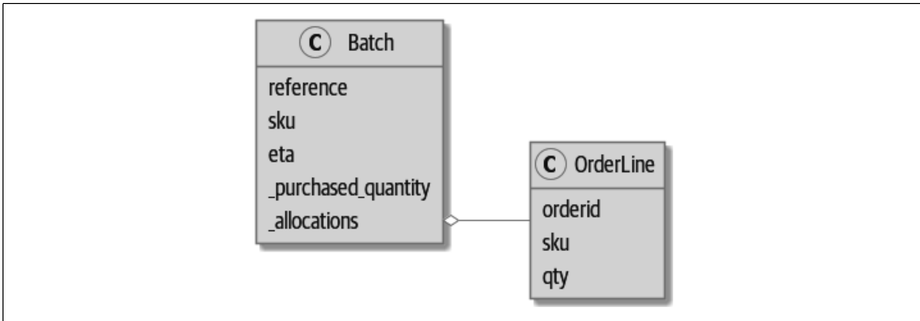


Abbildung 2-4: Unser Modell

Schauen wir uns an, wie wir das in eine relationale Datenbank übersetzen können.

Der »normale« ORM-Weg: Das Modell hängt vom ORM ab

Heutzutage ist es unwahrscheinlich geworden, dass Ihre Teammitglieder ihre SQL-Abfrage liebevoll per Hand formen. Stattdessen werden Sie mit ziemlicher Sicherheit ein Framework nutzen, um aus Ihren Modellobjekten SQL-Code zu erzeugen.

Diese Frameworks werden als *objektrelationale Mapper* (ORMs) bezeichnet, weil sie dazu dienen, die Lücke zwischen der Welt der Objekte und der Domänenmodellierung auf der einen Seite und der Welt der Datenbanken und der relationalen Algebra auf der anderen zu überbrücken.

Eines der wichtigsten Dinge, die wir durch einen ORM erhalten, ist die *Persistenzignoranz*: Dabei handelt es sich um die Idee, dass unser schickes Domänenmodell nichts darüber wissen muss, wie Daten geladen oder persistiert werden. Das hilft uns dabei, unsere Domäne frei von direkten Abhängigkeiten zu spezifischen Datenbanktechnologien zu halten.[3]

Folgen Sie aber dem typischen SQLAlchemy-Tutorial, werden Sie bei so etwas wie dem folgenden landen:

»Deklarative« Syntax von SQLAlchemy, Modell hängt von ORM ab. (orm.py)

```
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Order(Base):
    id = Column(Integer, primary_key=True)
```

3 In diesem Sinne ist der Einsatz eines ORM bereits ein Beispiel des DIP. Statt von hartcodiertem SQL hängen wir nun von einer Abstraktion ab – dem ORM. Aber das reicht uns nicht – nicht in diesem Buch!

```
class OrderLine(Base):
    id = Column(Integer, primary_key=True)
    sku = Column(String(250))
    qty = Integer(String(250))
    order_id = Column(Integer, ForeignKey('order.id'))
    order = relationship(Order)

class Allocation(Base):
    ...
```

Sie müssen SQLAlchemy nicht weiter kennen, um auf einen Blick zu sehen, dass unser sauberes Modell nun voll von Abhängigkeiten zum ORM ist und zudem anfängt, ganz furchtbar auszusehen. Können wir guten Gewissens sagen, dass dieses Modell unabhängig von der Datenbank ist? Wie kann es von Storage-Aspekten getrennt sein, wenn die Eigenschaften unseres Modells direkt mit Datenbankspalten gekoppelt sind?

Djangos ORM ist im Grunde das Gleiche, nur restriktiver

Sind Sie mit Django vertrauter, würde der obige »deklarative« SQLAlchemy-Codeabschnitt eher so aussehen:

ORM-Beispiel mit Django

```
class Order(models.Model):
    pass

class OrderLine(models.Model):
    sku = models.CharField(max_length=255)
    qty = models.IntegerField()
    order = models.ForeignKey(Order)

class Allocation(models.Model):
    ...
```

Es läuft aber auch hier auf das Gleiche hinaus: Unsere Modellklassen erben direkt von ORM-Klassen, damit hängt unser Modell vom ORM ab. Wir wollen aber, dass es andersherum ist.

Django bietet kein Äquivalent für den klassischen Mapper von SQLAlchemy, aber in Anhang D finden Sie Beispiele dafür, wie sich Dependency Inversion und das Repository-Pattern mit Django anwenden lassen.

Die Abhängigkeit umkehren: ORM hängt vom Modell ab

Nun, dankenswerterweise ist das nicht die einzige Möglichkeit, SQLAlchemy einzusetzen. Alternativ können Sie Ihr Schema auch getrennt erstellen und dann einen expliziten *Mapper* definieren, mit dem zwischen dem Schema und unserem Domänenmodell hin- und herkonvertiert wird – SQLAlchemy nennt das klassisches Mapping (*https://oreil.ly/ZucTG*):

Explizites ORM-Mapping mit Table-Objekten in SQLAlchemy (orm.py)

```
from sqlalchemy.orm import mapper, relationship

import model  ❶


metadata = MetaData()

order_lines = Table(  ❷
    'order_lines', metadata,
    Column('id', Integer, primary_key=True, autoincrement=True),
    Column('sku', String(255)),
    Column('qty', Integer, nullable=False),
    Column('orderid', String(255)),
)

...

def start_mappers():
    lines_mapper = mapper(model.OrderLine, order_lines)  ❸
```

❶ Das ORM importiert das Domänenmodell (oder »hängt ab von« oder »kennt«) und nicht umgekehrt.

❷ Wir definieren unsere Datenbanktabellen und -spalten mithilfe der SQLAlchemy-Abstraktionen.[4]

❸ Rufen wir die Funktion `mapper` auf, entwickelt SQLAlchemy seine Magie und bindet unsere Domänenmodell-Klassen an die diversen von uns definierten Tabellen.

Im Endergebnis können wir mit dem Aufruf von `start_mappers` Instanzen des Domänenmodells einfach aus der Datenbank laden und wieder dorthin zurückschreiben. Aber wenn wir diese Funktion niemals aufrufen, verbleiben unsere Domänenmodell-Klassen in seliger Ungewissheit über die Datenbank.

So bekommen wir alle Vorteile von SQLAlchemy – unter anderem die Möglichkeit, `alembic` für die Migrationen einzusetzen, und die Fähigkeit, mit unseren Domänenklassen transparent Abfragen auszuführen (wie wir noch sehen werden).

Probieren Sie erstmalig, Ihre ORM-Konfiguration aufzubauen, kann es hilfreich sein, Tests dafür zu schreiben, wie zum Beispiel:

Das ORM direkt testen – Wegwerftests (test_orm.py)

```
def test_orderline_mapper_can_load_lines(session):  ❶
    session.execute(
        'INSERT INTO order_lines (orderid, sku, qty) VALUES '
        '("order1", "RED-CHAIR", 12),'
        '("order1", "RED-TABLE", 13),'
```

4 Selbst in Projekten, in denen kein ORM zum Einsatz kommt, nutzen wir SQLAlchemy oft zusammen mit Alembic, um Schemata in Python deklarativ zu erstellen und Migrationen, Verbindungen und Sessions zu managen.

```
        '("order2", "BLUE-LIPSTICK", 14)'
    )
    expected = [
        model.OrderLine("order1", "RED-CHAIR", 12),
        model.OrderLine("order1", "RED-TABLE", 13),
        model.OrderLine("order2", "BLUE-LIPSTICK", 14),
    ]
    assert session.query(model.OrderLine).all() == expected


def test_orderline_mapper_can_save_lines(session):
    new_line = model.OrderLine("order1", "DECORATIVE-WIDGET", 12)
    session.add(new_line)
    session.commit()

    rows = list(session.execute('SELECT orderid, sku, qty FROM "order_lines"'))
    assert rows == [("order1", "DECORATIVE-WIDGET", 12)]
```

❶ Wenn Sie pytest noch nie verwendet haben, erfordert das Argument `session` ein wenig Erläuterung. Sie müssen sich im Rahmen dieses Buchs nicht mit den Details von pytest oder seinen Fixtures auseinandersetzen, aber die Kurzform ist, dass Sie gemeinsam Abhängigkeiten für Ihre Tests als »Fixtures« definieren können, die pytest dann in die Tests injiziert, die sie benötigen, indem es sich die Funktionsargumente anschaut. In diesem Fall ist es eine Datenbanksession von SQLAlchemy.

Sie werden diese Tests vermutlich nicht lange benötigen – wie Sie gleich sehen, ist es nach dem Invertieren der Abhängigkeit zwischen ORM und Domänenmodell nur noch ein kleiner zusätzlicher Schritt, eine weitere Abstraktion namens Repository-Pattern zu implementieren, für die sich leichter Tests schreiben lassen und die eine einfache Schnittstelle zum Faken bietet.

Aber wir haben schon unser Ziel erreicht, die klassische Abhängigkeit umzukehren: Das Domänenmodell bleibt »sauber« und frei von Infrastrukturaspekten. Wir könnten SQLAlchemy wegnehmen und ein anderes ORM einsetzen – oder ein ganz anderes Persistenzsystem –, und trotzdem müsste sich das Domänenmodell nicht ändern.

Abhängig davon, was Sie in Ihrem Domänenmodell tun – und insbesondere, wenn Sie sich weit vom OO-Paradigma entfernen –, werden Sie es zunehmend schwieriger finden, das ORM genau das Verhalten erzeugen zu lassen, das Sie benötigen, und eventuell müssen Sie Ihr Domänenmodell anpassen.[5] Wie so oft bei architektonischen Entscheidungen werden Sie Kompromisse eingehen müssen. Das Zen von Python sagt es schon: »Praktikabilität schlägt Reinheit!«

5 Ein großer Dank geht an die außerordentlich hilfsbereiten Maintainer von SQLAlchemy und speziell an Mike Bayer.

Unser API-Endpunkt sieht nun vielleicht wie folgt aus, und er wird schon gut funktionieren:

SQLAlchemy direkt im API-Endpunkt einsetzen

```
@flask.route.gubbins
def allocate_endpoint():
    session = start_session()

    # Auftragszeile aus Request auslesen
    line = OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

    # alle Batches aus der DB laden
    batches = session.query(Batch).all()

    # unseren Domänenservice aufrufen
    allocate(line, batches)

    # die Zuteilung zurück in die DB schreiben
    session.commit()

    return 201
```

Das Repository-Pattern

Das *Repository*-Pattern ist eine Abstraktion über persistenten Storage. Es verbirgt die langweiligen Details des Datenzugriffs, indem es vorgibt, dass sich all unsere Daten im Speicher befinden.

Hätten wir in unseren Laptops unbegrenzt viel Speicher, bräuchten wir all diese schwerfälligen Datenbanken nicht. Stattdessen könnten wir einfach unsere Objekte dort einsetzen, wo wir das wollten. Wie würde das aussehen?

Sie müssen Ihre Daten irgendwo herbekommen.

```
import all_my_data

def create_a_batch():
    batch = Batch(...)
    all_my_data.batches.add(batch)

def modify_a_batch(batch_id, new_quantity):
    batch = all_my_data.batches.get(batch_id)
    batch.change_initial_quantity(new_quantity)
```

Auch wenn sich unsere Objekte im Speicher befinden, müssen wir sie *irgendwo* ablegen, damit wir sie auch wiederfinden können. Bei unseren In-Memory-Daten könnten wir neue Objekte hinzufügen – so wie bei einer Liste oder einem Set. Weil

sich die Objekte im Speicher befinden, müssten wir nie eine Methode `.save()` aufrufen – wir würden einfach das für uns interessante Objekt holen und es direkt im Speicher verändern.

Das Repository im Abstrakten

Das einfachste Repository hat nur zwei Methoden: `add()`, um ein neues Element in das Repository zu stecken, und `get()`, um ein vorher hinzugefügtes Element zurückzugeben.[6] Wir werden in unserer Domänen- und der Serviceschicht streng bei nur diesen Methoden bleiben, um auf Daten zuzugreifen. Diese selbst auferlegte Einfachheit hält uns davon ab, unser Domänenmodell mit der Datenbank zu koppeln.

Abstrakte Basisklassen, Duck Typing und Protokolle

Wir verwenden die abstrakten Basisklassen in diesem Buch aus didaktischen Gründen: Wir hoffen, dass sie uns dabei helfen, zu erläutern, wie die Schnittstelle der Repository-Abstraktion aussieht.

In der Realität haben wir manches Mal ABCs aus unserem produktiven Code gelöscht, weil Python es uns zu leicht macht, sie zu ignorieren, sodass sie schließlich nicht mehr gewartet wurden und schlimmstenfalls sogar in die Irre führten. Wir nutzen stattdessen oft das Duck Typing in Python, um Abstraktionen zu ermöglichen. Für Pythonistas ist ein Repository jedes beliebige Objekt mit den Methoden `add(`*thing*`)` und `get(`*id*`)`.

Eine interessante Alternative sind die PEP-544-Protokolle (*https://oreil.ly/q9EPC*). Durch diese erhalten Sie Typisierung ohne Vererbungsmöglichkeiten, was Fans von »Komposition vor Vererbung« ganz besonders mögen werden.

So könnte eine abstrakte Basisklasse (ABC) für unser Repository aussehen:

Das einfachste Repository (repository.py)

```
class AbstractRepository(abc.ABC):

    @abc.abstractmethod  ❶
    def add(self, batch: model.Batch):
        raise NotImplementedError  ❷

    @abc.abstractmethod
    def get(self, reference) -> model.Batch:
        raise NotImplementedError
```

6 Sie fragen sich vielleicht: »Was ist mit `list`, `delete` oder `update`?« Nun, in einer idealen Welt bearbeiten wir unsere Modellobjekte immer eines nach dem anderen, und ein Löschen geschieht im Allgemeinen über ein Soft-Delete, also `batch.cancel()`. Um das Aktualisieren kümmern wir uns schließlich im Unit-of-Work-Pattern, wie Sie in Kapitel 6 sehen werden.

❶ Python-Tipp: `@abc.abstractmethod` ist eines der Hilfsmittel, dank dem ABCs in Python »funktionieren«. Sie können in Python keine Klasse instanziieren, die nicht alle `abstractmethods`-Methoden der übergeordneten Klasse implementiert.[7]

❷ `raise NotImplementedError` ist zwar nett, aber weder hinreichend noch erforderlich. Tatsächlich können Ihre abstrakten Methoden ein eigenes Verhalten besitzen, das Unterklassen aufrufen kann, wenn Sie das wirklich wollen.

Vor- und Nachteile

> Sie kennen den Spruch, dass Ökonomen von allem den Preis, aber von nichts den Wert kennen? Nun, Menschen, die programmieren, kennen von allem die Vorteile, aber von nichts die Nachteile.
>
> – Rich Hickey

Immer wenn wir ein Architektur-Pattern in diesem Buch vorstellen, fragen wir uns: »Was bekommen wir damit? Und was kostet es uns?«

Normalerweise führen wir mindestens eine zusätzliche Abstraktionsschicht ein, und auch wenn wir hoffen, dass sich damit die Gesamtkomplexität verringert, erhöhen wir die lokale Komplexität durchaus. Das bringt Kosten mit sich – in Bezug auf die reine Menge an beweglichen Teilen und Fragen der Wartung.

Das Repository-Pattern ist aber vermutlich die einfachste Wahl in diesem Buch, wenn Sie sowieso schon den Weg von DDD und Dependency Inversion eingeschlagen haben. Soweit unser Code betroffen ist, tauschen wir tatsächlich nur die SQL-Alchemy-Abstraktion (`session.query(Batch)`) gegen eine andere aus, die wir entworfen haben (`batches_repo.get`).

Wir werden jedes Mal, wenn wir ein neues Domänenobjekt hinzufügen, das wir abfragen wollen, unsere Repository-Klasse um ein paar Zeilen Code ergänzen müssen, aber dafür erhalten wir eine einfache Abstraktion für unsere Storage-Schicht, die wir kontrollieren. Das Repository-Pattern wird es uns erleichtern, grundlegende Änderungen an der Art und Weise vornehmen zu können, wie wir Dinge abspeichern (siehe Anhang C), und wie Sie sehen werden, lassen sich damit leicht Fakes für Unit Tests erstellen.

Zudem ist das Repository-Pattern in der DDD-Welt so verbreitet, dass auch Menschen, mit denen Sie zusammenarbeiten und die von Java oder C# zu Python kommen, damit vertraut sein sollten. In Abbildung 2-5 ist das Pattern dargestellt.

7 Um wirklich alle Vorteile von ABCs ausnutzen zu können (sofern es sie denn gibt), nutzen Sie Hilfsmittel wie `pylint` oder `mypy`.

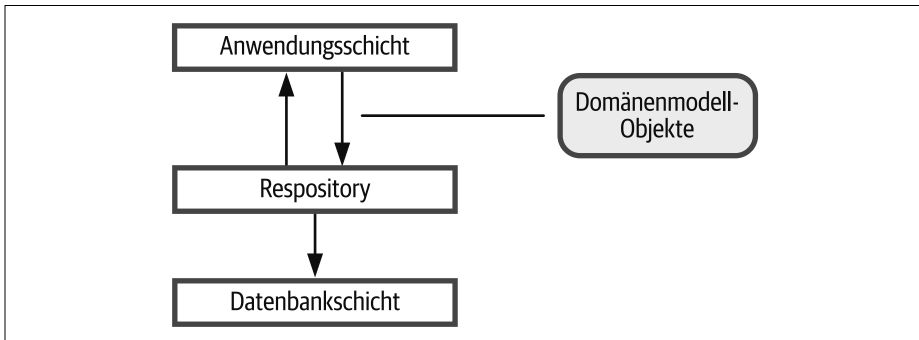
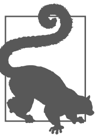


Abbildung 2-5: Repository-Pattern

Wie immer beginnen wir mit einem Test. Dieser würde vermutlich als Integrationstest klassifiziert werden, da wir prüfen, ob unser Code (das Repository) korrekt mit der Datenbank verbunden ist – daher werden die Tests eher aus einer Mischung aus SQL mit Aufrufen und Assertions unseres eigenen Codes bestehen.



Anders als die weiter oben erwähnten ORM-Tests sind diese Tests gute Kandidaten für einen längeren Aufenthalt in Ihrer Codebasis – insbesondere, wenn Teile Ihres Domänenmodells dafür sorgen, dass das objektrelationale Mapping nicht trivial ist.

Repository-Test zum Sichern eines Objekts (test_repository.py)

```
def test_repository_can_save_a_batch(session):
    batch = model.Batch("batch1", "RUSTY-SOAPDISH", 100, eta=None)

    repo = repository.SqlAlchemyRepository(session)
    repo.add(batch)  ❶
    session.commit()  ❷

    rows = list(session.execute(
        'SELECT reference, sku, _purchased_quantity, eta FROM "batches"'  ❸
    ))
    assert rows == [("batch1", "RUSTY-SOAPDISH", 100, None)]
```

❶ `repo.add()` ist die zu testende Methode.

❷ Wir halten das `.commit()` außerhalb des Repository und legen es in die Verantwortung des aufrufenden Codes. Es gibt gute Gründe für und gegen diese Entscheidung – manche davon werden klarer werden, wenn wir Kapitel 6 erreichen.

❸ Wir verwenden reines SQL, um zu prüfen, ob die richtigen Daten gesichert wurden.

Beim nächsten Test geht es darum, Batches und Zuteilungen zu lesen, daher ist er komplexer:

Repository-Test zum Lesen eines komplexen Objekts (test_repository.py)

```
def insert_order_line(session):
    session.execute(  ❶
        'INSERT INTO order_lines (orderid, sku, qty)'
        ' VALUES ("order1", "GENERIC-SOFA", 12)'
    )
    [[orderline_id]] = session.execute(
        'SELECT id FROM order_lines WHERE orderid=:orderid AND sku=:sku',
        dict(orderid="order1", sku="GENERIC-SOFA")
    )
    return orderline_id

def insert_batch(session, batch_id):  ❷
    ...

def test_repository_can_retrieve_a_batch_with_allocations(session):
    orderline_id = insert_order_line(session)
    batch1_id = insert_batch(session, "batch1")
    insert_batch(session, "batch2")
    insert_allocation(session, orderline_id, batch1_id)  ❸

    repo = repository.SqlAlchemyRepository(session)
    retrieved = repo.get("batch1")

    expected = model.Batch("batch1", "GENERIC-SOFA", 100, eta=None)
    assert retrieved == expected  # Batch.__eq__ vergleicht
                                  # nur Referenz  ❸
    assert retrieved.sku == expected.sku  ❹
    assert retrieved._purchased_quantity == expected._purchased_quantity
    assert retrieved._allocations == {  ❹
        model.OrderLine("order1", "GENERIC-SOFA", 12),
    }
```

❶ Das testet das Lesen, daher sorgt der SQL-Code dafür, dass Daten von `repo.get()` gelesen werden können.

❷ Wir ersparen Ihnen hier die Details von `insert_batch` und `insert_allocation` – damit werden ein paar Batches erstellt und für den für uns interessanten Batch auch eine Auftragszeile, die ihm zugeordnet ist.

❸ Hier prüfen wir das Ganze. Das erste `assert ==` testet, ob die Typen übereinstimmen und die Referenz die gleiche ist (weil – wie Sie sich erinnern – `Batch` eine Entität ist und wir ein eigenes `eq` dafür haben).

❹ Wir prüfen auch explizit auf die wichtigsten Attribute, unter anderem `._allocations`, bei dem es sich um ein Python-Set mit Value Objects vom Typ `OrderLine` handelt.

Sie müssen selbst entscheiden, ob Sie gewissenhaft Tests für jedes Modell schreiben wollen. Haben Sie für eine Klasse Tests zum Erstellen/Verändern/Sichern geschrieben, reicht es Ihnen vielleicht, für die anderen nur minimale Round-Trip-

Tests zu erstellen – oder auch gar keine Tests, wenn alle einem gleichen Muster folgen. In unserem Fall ist die ORM-Konfiguration, die das `._allocations`-Set einrichtet, ein wenig komplex, daher lohnt hier ein eigener Test.

Sie erhalten schließlich so etwas:

Ein typisches Repository

```
class SqlAlchemyRepository(AbstractRepository):

    def __init__(self, session):
        self.session = session

    def add(self, batch):
        self.session.add(batch)

    def get(self, reference):
        return self.session.query(model.Batch).filter_by(reference=reference).one()

    def list(self):
        return self.session.query(model.Batch).all()
```

Und jetzt kann unser Flask-Endpunkt so aussehen:

Unser Repository direkt in unserem API-Endpunkt einsetzen

```
@flask.route.gubbins
def allocate_endpoint():
    batches = SqlAlchemyRepository.list()
    lines = [
        OrderLine(l['orderid'], l['sku'], l['qty'])
         for l in request.params...
    ]
    allocate(lines, batches)
    session.commit()
    return 201
```

Übung

Neulich auf einer DDD-Konferenz sagte uns ein Freund: »Ich habe seit zehn Jahren kein ORM eingesetzt.« Das Repository-Pattern und ein ORM agieren beide als Abstraktion vor purem SQL, daher ist es nicht wirklich erforderlich, das eine und das andere zu verwenden. Warum sollten wir nicht versuchen, unser Repository ohne einen ORM zu implementieren? Den Code dazu finden Sie auf GitHub unter *https://github.com/cosmicpython/code/tree/chapter_02_repository_exercise*.

Wir haben die Repository-Tests stehen gelassen, aber es liegt an Ihnen, herauszufinden, wie der SQL-Code auszusehen hat. Vielleicht wird es schwerer sein, als Sie denken – vielleicht auch einfacher. Aber das Schöne ist, dass sich der Rest Ihrer Anwendung einfach nicht darum schert.

Es ist nicht einfach, ein Fake-Repository für Tests zu erstellen!

Das ist einer der größten Vorteile des Repository-Patterns:

Ein einfaches Fake-Repository mithilfe eines Sets (repository.py)

```
class FakeRepository(AbstractRepository):

    def __init__(self, batches):
        self._batches = set(batches)

    def add(self, batch):
        self._batches.add(batch)

    def get(self, reference):
        return next(b for b in self._batches if b.reference == reference)

    def list(self):
        return list(self._batches)
```

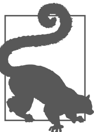
Weil es sich um einen einfachen Wrapper um ein `set` handelt, bestehen alle Methoden aus Einzeilern.

Der Einsatz eines Fake-Repository in Tests ist wirklich einfach, und wir haben eine simple Abstraktion, die sich problemlos verwenden lässt:

Beispieleinsatz eines Fake-Repository (test_api.py)

```
fake_repo = FakeRepository([batch1, batch2, batch3])
```

Das werden Sie im nächsten Kapitel auch im Einsatz erleben.



Das Erstellen von Fakes für Ihre Abstraktionen ist eine ausgezeichnete Möglichkeit, Design-Feedback zu erhalten: Ist es schwer, Fakes zu erzeugen, ist die Abstraktion vermutlich zu kompliziert.

Was ist in Python ein Port und was ein Adapter?

Wir wollen hier nicht zu sehr auf den Begrifflichkeiten herumreiten, weil wir uns vor allem auf die Dependency Inversion konzentrieren möchten und die von Ihnen genutzte Technik gar nicht so entscheidend ist. Zudem ist uns bewusst, dass verschiedene Personen auch leicht unterschiedliche Definitionen nutzen.

Ports und Adapter kommen aus der OO-Welt, und die von uns verwendete Definition ist, dass es sich bei einem *Port* um die *Schnittstelle* beziehungsweise das *Interface* zwischen unserer Anwendung und dem handelt, was wir wegabstrahieren wollen, während der *Adapter* die *Implementierung* hinter dieser Schnittstelle oder Abstraktion ist.

Nun bietet Python nicht direkt Interfaces an, daher ist es meist einfach, einen Adapter zu identifizieren, während die Definition des Ports kniffliger sein kann. Nutzen Sie eine abstrakte Basisklasse, ist das der Port. Wenn nicht, handelt es sich beim Port einfach um den Duck Type, dem Ihre Adapter entsprechen und den Ihre eigentliche Anwendung erwartet – die Funktions- und Methodennamen sowie die Namen und Typen der entsprechenden Argumente.

Konkret ist in diesem Kapitel `AbstractRepository` der Port, während es sich bei `SQLAlchemyRepository` und `FakeRepository` um die Adapter handelt.

Zusammenfassung

Mit dem Zitat von Rich Hickey im Hinterkopf fassen wir in jedem Kapitel die Vor- und Nachteile jedes architektonischen Patterns zusammen, das wir vorstellen. Lassen Sie uns klarstellen: Wir sagen nicht, dass jede einzelne Anwendung so gebaut werden muss – nur ist manchmal die Komplexität der App und der Domäne so groß, dass es sich lohnt, die Zeit und den Aufwand für das Hinzufügen dieser zusätzlichen Indirektionsschichten auf sich zu nehmen.

Mit diesen Worten vorweg zeigt Tabelle 2-1 nun ein paar der Vor- und Nachteile des Repository-Patterns und unseres persistenzignoranten Modells.

Tabelle 2-1: Vor- und Nachteile des Repository-Patterns und der Persistenzignoranz

Vorteile	Nachteile
Wir haben eine einfache Schnittstelle zwischen dem persistenten Storage und unserem Domänenmodell. Man kann leicht eine Fake-Version des Repository für Unit Tests erstellen oder unterschiedliche Storage-Lösungen austauschen, weil wir das Modell vollständig von Infrastrukturaspekten entkoppelt haben. Es hilft uns, das Domänenmodell zu schreiben, bevor wir über Persistenz nachdenken, weil wir uns so auf das eigentliche Businessproblem konzentrieren können. Wollen wir unseren Ansatz einmal komplett ändern, können wir das in unserem Modell tun und müssen uns erst später Gedanken über Fremdschlüssel oder Migrationen machen. Unser Datenbankschema ist wirklich einfach, weil wir die vollständige Kontrolle darüber haben, wie wir unsere Objekte auf Tabellen abbilden.	Ein ORM sorgt immer schon für ein gewisses Entkoppeln. Es kann schwierig sein, Fremdschlüssel zu ändern, aber es sollte ziemlich einfach sein, von MySQL zu Postgres zu wechseln, wenn Sie das irgendwann einmal müssen. Das händische Warten von ORM-Mappings erfordert zusätzlichen Aufwand und extra Code. Jede zusätzliche Indirektionsschicht lässt die Wartungskosten steigen und sorgt für einen »WTF-Faktor« bei Python-Programmierinnen und -Programmierern, die das Repository-Pattern noch nie gesehen haben.

In Abbildung 2-6 sehen Sie die zentrale Aussage: Ja, für einfache Fälle sorgt ein entkoppeltes Domänenmodell für mehr Arbeit als ein einfaches ORM/ActiveRecord-Pattern.[8]

8 Das Diagramm wurde inspiriert vom Post »Global Complexity, Local Simplicity« von Rob Vens (*https://oreil.ly/fQXkP*).

Wenn Ihre App nicht mehr als ein einfacher CRUD-Wrapper um eine Datenbank ist, brauchen Sie auch kein Domänenmodell und kein Repository.

Aber je komplexer die Domäne ist, desto mehr macht es sich bezahlt, wenn Sie sich von Infrastrukturaspekten unabhängig machen, sobald es um das einfache Umsetzen von Änderungen geht.

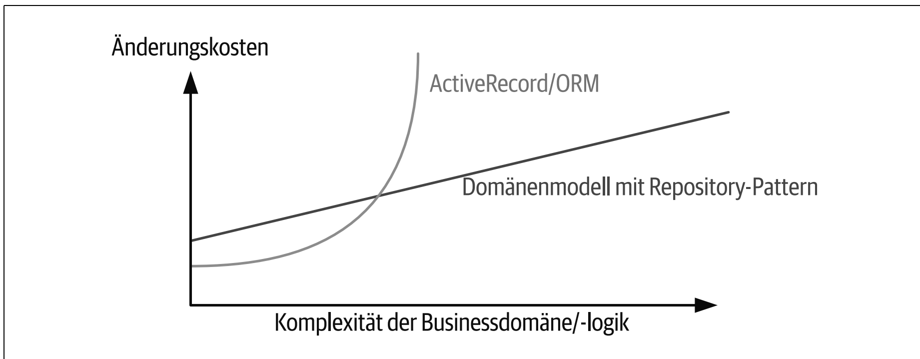


Abbildung 2-6: Vor- und Nachteile des Domänenmodells als Diagramm

Unser Beispielcode ist nicht komplex genug, um mehr als eine Andeutung dazu zu geben, wie das Diagramm im rechten Bereich aussehen könnte, aber die Hinweise darauf sind vorhanden. Stellen Sie sich beispielsweise vor, dass wir uns eines Tages dafür entscheiden würden, die Zuteilungen im `OrderLine`- statt im `Batch`-Objekt zu verwalten: Nutzen wir Django, müssten wir zum Beispiel erst die Datenbankmigration definieren und durchdenken, bevor wir irgendwelche Tests laufen lassen könnten. Aber so können wir – weil unser Modell nur aus den guten alten Python-Objekten besteht – ein `set()` als neues Attribut ändern und müssen erst später über die Datenbank nachdenken.

Zusammenfassung Repository-Pattern

Dependency-Inversion auf Ihr ORM anwenden
: Unser Domänenmodell sollte frei von Infrastukturaspekten sein, daher sollte Ihr ORM Ihr Modell importieren und nicht umgekehrt.

Das Repository-Pattern ist eine einfache Abstraktion für permanenten Storage
: Das Repository vermittelt Ihnen die Illusion einer Collection aus In-Memory-Objekten. Es erleichtert das Erstellen eines `FakeRepository` zum Testen und das Austauschen grundlegender Details Ihrer Infrastruktur, ohne Ihre Kernanwendung auseinanderzureißen. In Anhang C finden Sie ein Beispiel dazu.

Sie fragen sich vielleicht, wie wir diese Repositories instanziieren – egal ob echt oder gefakt? Wie wird unsere Flask-App aussehen? Darüber erfahren Sie mehr im spannenden Kapitel 4 zum Service-Layer-Pattern.

Aber zuerst noch ein kleiner Exkurs.