O'REILLY®

# Architektura aplikacji w Pythonie

## TDD, DDD i rozwój mikrousług reaktywnych

Harry Percival
Bob Gregory

Helion

Tytuł oryginału: Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-7126-2



Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/arappy*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*ftp://ftp.helion.pl/przyklady/arappy.zip*

Printed in Poland.

# Spis treści

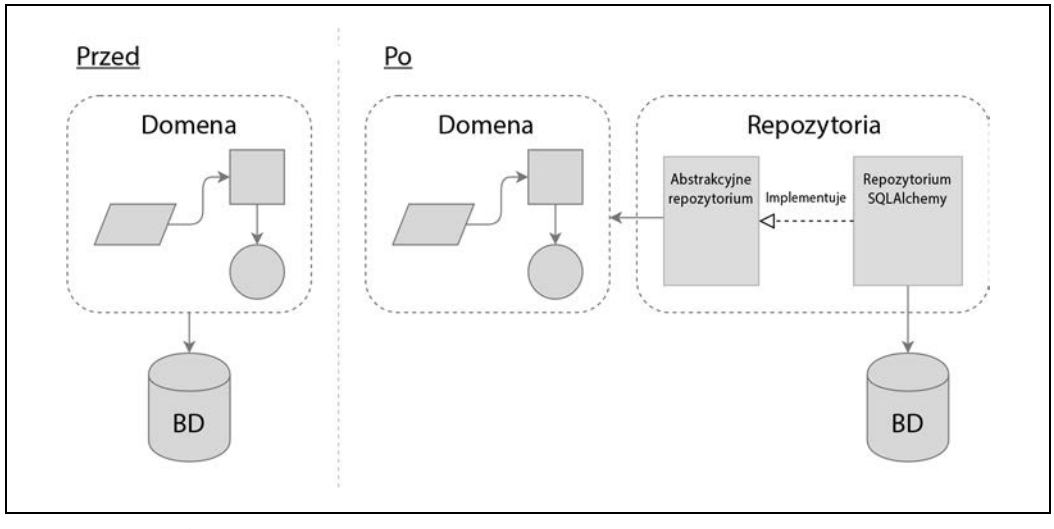
ROZDZIAŁ 2.

# Wzorzec Repozytorium

Czas spełnić obietnicę, że użyjemy zasady odwrócenia zależności w celu rozłączenia naszej logiki podstawowej od spraw dotyczących infrastruktury.

W tym rozdziale opisujemy wzorzec **Repozytorium** (ang. *Repository*), który upraszcza abstrakcję nad magazynem danych, umożliwiając oddzielenie warstwy modelu od warstwy danych. Pokażemy konkretny przykład, jak taka uproszczona abstrakcja ułatwia testowanie naszego systemu dzięki temu, że ukrywa kwestie związane z obsługą bazy danych.

Na rysunku 2.1 znajduje się schemat pokazujący, co chcemy zbudować — obiekt `Repository` osadzony między naszym modelem domeny i bazą danych.




***Rysunek 2.1.*** *Przed zastosowaniem wzorca Repozytorium i po jego użyciu*

Kod źródłowy do tego rozdziału znajduje się w archiwum na serwerze FTP (*ftp://ftp.helion.pl/przyklady/arappy.zip*), w folderze *r02*.

# Zapisywanie modelu domeny

W rozdziale 1. zbudowaliśmy prosty model domeny mogący alokować zamówienia w partiach produktów dostępnych na stanie. Napisanie testów do tego kodu nie sprawia nam trudności, ponieważ nie jest on od niczego zależny i nie wymaga konfiguracji żadnej infrastruktury. Gdybyśmy użyli bazy danych lub interfejsu API i utworzyli dane testowe, nasze testy byłyby trudniejsze do napisania i utrzymania.

Niestety wcześniej czy później będziemy musieli oddać nasz idealny mały model w ręce użytkowników i narazić go na trudy prawdziwego świata zdominowanego przez arkusze kalkulacyjne, przeglądarki internetowe i wyścigi do zasobów. W kilku kolejnych rozdziałach zajmujemy się sposobami połączenia naszego wyidealizowanego modelu domeny ze stanem zewnętrznym.

Zamierzamy pracować przy użyciu technik zwinnych, a więc naszym priorytetem będzie uzyskanie minimalnej funkcjonalności produktu w jak najkrótszym czasie. W tym przypadku będzie to sieciowy interfejs API. W prawdziwym projekcie może od razu przeszlibyśmy do kompleksowych testów i zaczęlibyśmy podłączać framework sieciowy, testując wszystko od zewnętrz.

Wiemy jednak, że bez względu na wszystko, w końcu będziemy potrzebowali jakiegoś magazynu danych, a ponieważ to jest podręcznik, możemy sobie pozwolić na wybór bardziej tradycyjnej metody działania od podstaw i zacząć już teraz zastanawiać się nad magazynowaniem i bazami danych.

# Trochę pseudokodu — czego będziemy potrzebować?

W trakcie budowy pierwszego punktu końcowego API wiemy, że będziemy musieli napisać kod wyglądający mniej więcej tak, jak poniższy.

*Tak będzie wyglądał nasz pierwszy punkt końcowy API*

```
@flask.route.gubbins
def allocate_endpoint():
    # Pobranie pozycji zamówienia z żądania
    line = OrderLine(request.params, ...)
    # Załadowanie wszystkich partii z BD
    batches = ...
    # Wywołanie usługi domeny
    allocate(line, batches)
    # Zapisanie alokacji w bazie danych
    return 201
```

Używamy biblioteki Flask, ponieważ jest lekka, ale nie musisz być jej użytkownikiem, aby zrozumieć treść tej książki. W istocie pokażemy Ci, jak sprawić, aby wybór frameworku był tylko drobnym detalem.

Będziemy potrzebować sposobu na pobieranie informacji o partii z bazy danych i utworzenie z nich obiektów naszego modelu domeny oraz będziemy potrzebować sposobu na zapisanie ich z powrotem w bazie danych.

# Zastosowanie zasady odwrócenia zależności do dostępu do danych

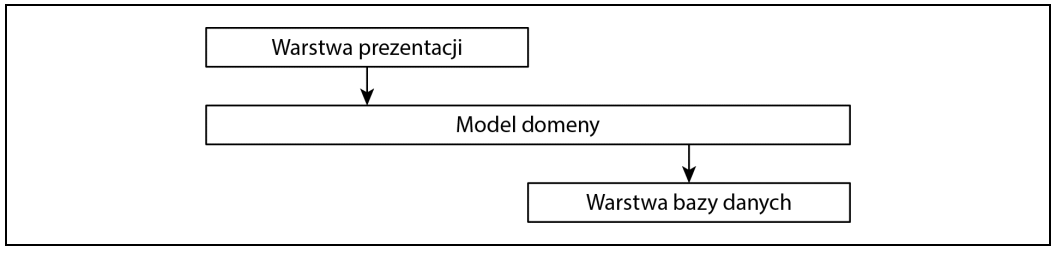
Jak wspomnieliśmy we wstępie, architektura warstwowa to często stosowany sposób konstrukcji systemu, który ma interfejs użytkownika, logikę i bazę danych (rysunek 2.2).


***Rysunek 2.2.*** *Architektura warstwowa*

Bardzo podobna jest struktura *Model-View-Template* (model-widok-szablon) systemu Django, a także architektura MVC (ang. *Model-View-Controller* — model-widok-kontroler). W każdym przypadku chodzi o rozdzielenie warstw (co jest dobrym pomysłem) i sprawienie, aby każda z nich zależała tylko od tej, która znajduje się pod nią.

My jednak chcemy, aby nasz model domeny nie miał *w ogóle żadnych zależności*[1]. Nie chcemy, aby sprawy związane z infrastrukturą wpływały na nasz model domeny i spowalniały nasze testy jednostkowe lub utrudniały nam wprowadzanie zmian.

Dlatego, jak zapowiedzieliśmy we wstępie, o naszym modelu będziemy myśleć tak, jakby był „wewnątrz", a zależności będą wpływały do jego wnętrza. Czasami takie podejście jest nazywane **architekturą cebulową** (ang. *onion architecture*) — rysunek 2.3.




***Rysunek 2.3.*** *Architektura cebulowa*

# Przypomnienie — nasz model

Przypomnijmy sobie nasz model domeny (rysunek 2.4) — alokacja to działanie polegające na połączeniu `OrderLine` z `Batch`. Alokacje zapisujemy jako kolekcje w naszym obiekcie `Batch`.

[1] Mamy raczej na myśli: „żadnych stanowych zależności". Zależność od biblioteki pomocniczej jest w porządku, w przeciwieństwie do zależności od ORM lub frameworku sieciowego.

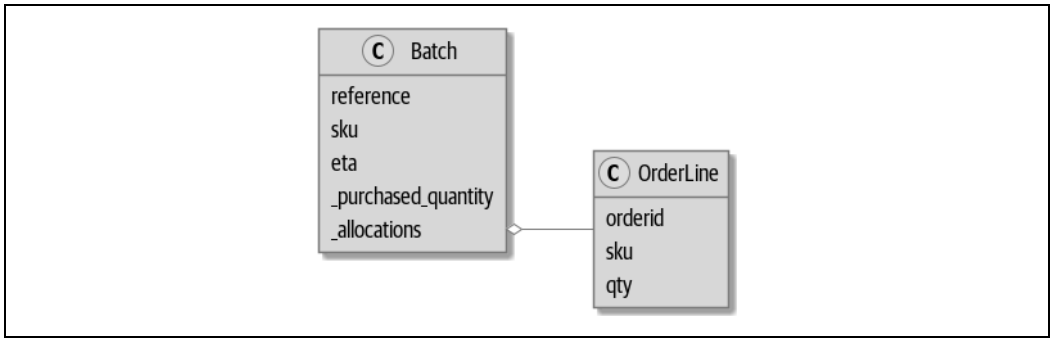
## Czy to są porty i adaptery?

Osoby czytające o wzorcach architekturalnych czasami zadają sobie następujące pytania:

> *Czy to porty i adaptery? Czy to architektura sześciokątna? Czy to jest to samo, co architektura cebulowa? A co z czystą architekturą? Czym jest port, a czym jest adapter? Dlaczego ludzie wymyślają tyle różnych nazw dla jednej rzeczy?*

Choć niektórzy lubią rozwodzić się nad subtelnymi różnicami, wszystkie wymienione nazwy dotyczą tego samego, czyli sprowadzają się do zasady odwrócenia zależności: moduły wysokiego poziomu (domena) nie powinny zależeć od modułów niskiego poziomu (infrastruktury)[2].

W dalszych rozdziałach przyjrzymy się dokładniej kwestii „zależności od abstrakcji" i zastanowimy się, czy w Pythonie istnieje odpowiednik interfejsów. Zobacz także podrozdział „Czym są porty i adaptery w Pythonie" w dalszej części rozdziału.




***Rysunek 2.4.*** *Nasz model*

Zobaczmy, jak można to przełożyć na relacyjną bazę danych.

## „Normalny" sposób — model zależy od ORM

Jest mało prawdopodobne, aby ktokolwiek w tych czasach ręcznie wykonywał własne zapytania SQL. Praktycznie każdy używa jakiegoś frameworku generującego kod SQL na podstawie obiektów modelu.

Frameworki tego typu nazywają się **maperami obiektowo-relacyjnymi** (ang. *object-relational mappers* — ORM), ponieważ ich przeznaczeniem jest eliminacja luki koncepcyjnej między światem obiektów i modelowania domeny a światem baz danych i algebry relacyjnej.

Największą korzyścią z używania ORM jest możliwość *zignorowania spraw związanych z przechowywaniem*, co sprawia, że nie musimy nic wiedzieć o sposobie ładowania lub zapisywania danych.

[2] Mark Seemann napisał na ten temat doskonały wpis na blogu (*https://oreil.ly/LpFS9*).

To umożliwia nam utrzymanie naszej domeny wolnej od bezpośrednich zależności od konkretnych technologii zarządzania bazami danych[3].

Jednak po lekturze typowego kursu na temat SQLAlchemy stworzysz coś w poniższym stylu.

*Deklaratywna składnia SQLAlchemy, model zależy od ORM (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)

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):
    ...
```

Nie trzeba znać SQLAlchemy, aby dostrzec, że teraz nasz nieskazitelny model jest pełen zależności od ORM i zaczyna straszyć swoim wyglądem. Czy naprawdę można powiedzieć, że ten model ignoruje sprawy związane z obsługą bazy danych? Jak może być oddzielony od kwestii przechowywania, kiedy własności naszego modelu są wprost powiązane z kolumnami bazy danych?

## Odwrócenie zależności — ORM zależy od modelu

Na szczęście to niejedyny sposób, w jaki można używać SQLAlchemy. Alternatywą jest osobne zdefiniowanie schematu i *mapera* określającego sposób konwersji między tym schematem a naszym modelem domeny, co w SQLAlchemy nazywa się mapowaniem klasycznym (*https://oreil.ly/ZucTG*):

*Bezpośrednie mapowanie ORM przy użyciu obiektów Table 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)),
```

[3] W tym sensie używanie ORM jest przykładem zastosowania zasady odwrócenia zależności — zamiast zależności od sztywnego kodu SQL, tworzymy zależność od abstrakcji, czyli ORM. To jednak dla nas za mało, przynajmniej w tej książce!

## ORM Django jest praktycznie taki sam, tylko bardziej restrykcyjny

Dla osób przyzwyczajonych do Django powyższy „deklaratywny” fragment kodu SQLAlchemy odpowiada takiemu:

*Przykładowy ORM 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):
    ...
```

Sedno jest identyczne — klasy naszego modelu dziedziczą bezpośrednio z klas ORM, więc nasz model zależy od ORM. Nam jednak zależy, aby było odwrotnie.

W Django nie ma odpowiednika klasycznego mapera SQLAlchemy, ale w dodatku D pokazaliśmy przykłady, jak zastosować odwrócenie zależności i wzorzec Repozytorium w Django.

```
)

...

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

❶ ORM importuje (albo „zależy” lub „wie o”) model domeny, nie odwrotnie.

❷ Definiujemy tabele i kolumny naszej bazy danych przy użyciu abstrakcji SQLAlchemy[4].

❸ Gdy wywołujemy funkcję `mapper`, SQLAlchemy wiąże klasy naszego modelu domeny z różnymi zdefiniowanymi przez nas tabelami.

W efekcie tego wszystkiego, jeśli wywołamy funkcję `start_mappers`, będziemy mogli bezproblemowo załadować i zapisać egzemplarze modelu domeny z i do bazy danych. A jeśli nigdy nie wywołamy tej funkcji, nasze klasy modelu domeny pozostaną w błogiej nieświadomości istnienia bazy danych.

W ten sposób korzystamy z wszystkich zalet SQLAlchemy, w tym z możliwości używania `alembic` do migracji, i jednocześnie możemy wysyłać transparentne zapytania za pomocą naszych klas domeny, o czym wkrótce się przekonasz.

Przy pierwszej próbie budowy konfiguracji ORM dobrze jest napisać testy, jak w poniższym przykładzie.

[4] Nawet w projektach, w których nie używamy ORM, często używamy SQLAlchemy razem z Alembic w celu deklaratywnego tworzenia schematów w Pythonie oraz zarządzania migracjami, połączeniami i sesjami.

*Bezpośrednie testowanie ORM (testy do wyrzucenia) (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),'
        '("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 Wyjaśnienie dla czytelników nieznających pytest, do czego służy argument `session` w tym teście. Czytając tę książkę, nie musisz przejmować się szczegółami dotyczącymi frameworku pytest ani jego narzędziami, ale warto wiedzieć, że wspólne zależności dla swoich testów można definiować jako konfiguracje testów (ang. *fixtures*). Wówczas pytest wstrzyknie je do testów, które ich potrzebują, analizując ich argumenty funkcji. W tym przypadku jest to sesja bazy danych SQLAlchemy.

Te testy prawdopodobnie długo nie będą Ci się przydawać — jak się wkrótce przekonasz, po odwróceniu zależności ORM i modelu domeny pozostanie już tylko mały krok do zaimplementowania kolejnej abstrakcji, zwanej wzorcem Repozytorium, dla której pisanie testów będzie łatwiejsze i która zapewni prosty interfejs do imitowania dalszych testów.

Już teraz jednak osiągnęliśmy nasz cel odwrócenia tradycyjnych zależności — nasz model domeny pozostaje „czysty" i wolny od spraw infrastrukturalnych. Możemy wyrzucić SQLAlchemy i użyć innego ORM albo całkiem innego systemu przechowywania danych, a model domeny nie będzie wymagał w związku z tym żadnych zmian.

W zależności od tego, jakie funkcje pełni model domeny, a szczególnie jeśli odejdziemy daleko od paradygmatu obiektowego, zmuszenie ORM do dokładnie takiego zachowania, na jakim nam zależy, może być coraz trudniejsze i możemy być zmuszeni do wprowadzenia zmian w modelu domeny[5]. Jak to często bywa przy podejmowaniu decyzji architekturalnych, trzeba będzie pójść na kompromis. Filozofia Pythona przecież głosi: „Praktyczność ponad czystość!".

W tym momencie nasz punkt końcowy API może wyglądać mniej więcej tak jak poniżej i moglibyśmy sprawić, aby działał poprawnie:

[5] Szacunek dla niesamowicie pomocnych osób zajmujących się utrzymaniem SQLAlchemy, a w szczególności dla Mike'a Bayera.

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

    # Pobranie pozycji zamówienia z żądania
    line = OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

    # Załadowanie wszystkich partii z bazy danych
    batches = session.query(Batch).all()

    # Wywołanie naszej usługi domeny
    allocate(line, batches)

    # Zapisanie alokacji z powrotem w bazie danych
    session.commit()

    return 201
```

# Wprowadzenie do wzorca Repozytorium

Wzorzec Repozytorium jest abstrakcją magazynu danych. Ukrywa mało istotne szczegóły dostępu do danych, udając, że wszystkie nasze dane znajdują się w pamięci.

Gdybyśmy mieli nieskończoną ilość pamięci w naszych laptopach, nie potrzebowalibyśmy żadnych nieporęcznych baz danych. Zamiast tego moglibyśmy w każdej chwili korzystać z naszych obiektów. Jak to by wyglądało?

*Musimy skądś pobierać dane*

```
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)
```

Choć nasze obiekty znajdują się w pamięci, musimy **gdzieś** je zapisać, aby móc skorzystać z nich także później. Do danych przechowywanych w pamięci moglibyśmy dodawać nowe obiekty, takie jak listy czy zbiory. Jako że obiekty te znajdują się w pamięci, nigdy nie musimy wywoływać metody `.save()`. Pobieramy tylko ten obiekt, który nas interesuje, i modyfikujemy go w pamięci.

## Abstrakcja repozytorium

Najprostsze repozytorium ma tylko dwie metody: `add()`, służącą do dodawania nowych elementów, i `get()`, służącą do pobierania wcześniej dodanych elementów[6]. Trzymamy się sztywno tych metod dostępu do danych w naszej domenie i warstwie usług. Taka prostota z wyboru powstrzymuje nas przed powiązaniem modelu domeny z bazą danych.

Poniżej pokazujemy, jak wyglądałaby nasza abstrakcyjna klasa bazowa (ang. *abstract base class* — ABC).

*Najprostsze możliwe repozytorium (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
```

❶ Wskazówka na temat korzystania z języka Python: `@abc.abstractmethod` sprawia, że abstrakcyjne klasy bazowe w Pythonie mogą „działać". Python nie pozwoli na utworzenie egzemplarza klasy, która nie implementuje wszystkich metod abstrakcyjnych (`abstractmethod`) klasy nadrzędnej[7].

❷ `raise NotImplementedError` to ogólnie dobre rozwiązanie, ale nie jest ani wystarczające, ani niezbędne. W praktyce w metodach abstrakcyjnych można zaimplementować prawdziwe zachowania, do których mogą odwoływać się podklasy.

### Abstrakcyjne klasy bazowe, kacze typizowanie i protokoły

W tej książce używamy abstrakcyjnych klas bazowych tylko w celach dydaktycznych — w nadziei, że pomogą nam wyjaśnić, czym jest interfejs abstrakcji repozytorium.

W rzeczywistej pracy zdarzało nam się usuwać takie klasy z kodu produkcyjnego, ponieważ Pythonowi zbyt łatwo przychodzi ich ignorowanie, przez co pozostają bez konserwacji, a w najgorszym przypadku nawet wprowadzają w błąd. W praktyce często polegamy po prostu na kaczym typizowaniu Pythona, aby korzystać z abstrakcji. Dla programisty tego języka repozytorium jest *każdym* obiektem mającym metody `add(coś)` i `get(id)`.

Alternatywą, którą warto się zainteresować, są protokoły PEP 544 (*https://oreil.ly/q9EPC*). Dają one typizowanie bez możliwości dziedziczenia, które szczególnie spodoba się miłośnikom zasady „kompozycja ponad dziedziczenie".

[6] Może sobie myślisz — „A co z `list`, `delete` i `update`?". Jednak w idealnym świecie obiekty modelu modyfikujemy pojedynczo, usuwamy w sposób miękki, tzn. za pomocą metody `batch.cancel()`, a aktualizujemy dane przy użyciu wzorca Jednostka Pracy opisanego w rozdziale 6.

[7] Aby naprawdę odnieść korzyści z używania abstrakcyjnych klas bazowych (jeśli jakieś są), korzystaj z narzędzi pomocniczych, takich jak `pylint` i `mypy`.

## Gdzie tkwi haczyk

> Mówi się, że ekonomiści znają cenę wszystkiego, ale nie znają wartości niczego. Natomiast programiści znają zalety wszystkiego, ale nie znają żadnych kompromisów.
>
> — Rich Hickey

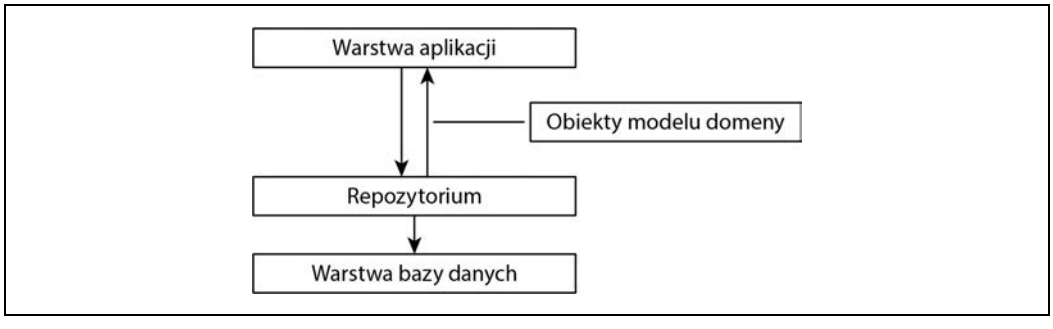
Na początku opisu każdego wzorca w tej książce zadajemy pytania: „Co dzięki niemu zyskamy? Co będziemy musieli dać w zamian?".

Zazwyczaj najniższą ceną jest wprowadzenie przynajmniej dodatkowej warstwy abstrakcji i choć możemy mieć nadzieję, że ogólny poziom złożoności się zmniejszy, lokalnie zostanie zwiększony. Poza tym poniesiemy koszt związany z przenoszeniem części programu i jego utrzymaniem.

Wzorzec Repozytorium jest jednak prawdopodobnie najprostszy z wszystkich opisanych w tej książce, jeśli już zaczynałeś planować wdrażanie DDD i odwracanie zależności. Jeśli chodzi o nasz kod, to po prostu zamienimy abstrakcję SQLAlchemy (`session.query(Batch)`) na inną (`batches_repo.get`), którą sami zaprojektujemy.

Przy każdym dodawaniu nowego obiektu domeny, który będziemy chcieli móc pobierać, będziemy musieli napisać kilka wierszy kodu w naszej klasie repozytorium, ale w zamian otrzymamy prostą abstrakcję naszej warstwy magazynowej, którą będziemy kontrolować. Wzorzec Repozytorium ułatwi wprowadzanie fundamentalnych zmian w sposobie przechowywania danych (zobacz dodatek C) i — jak się wkrótce przekonasz — pozwala na łatwą imitację w celu przeprowadzania testów jednostkowych.

Dodatkowo wzorzec Repozytorium jest tak powszechny w świecie DDD, że jeśli napotkasz programistów, którzy przeszli na Pythona z Javy lub C#, to prawdopodobnie będą go znać. Na rysunku 2.5 przedstawiono schemat tego wzorca.




***Rysunek 2.5.*** *Wzorzec Repozytorium*

Jak zawsze, zaczynamy od testu. Ten można byłoby zaklasyfikować jako test integracyjny, ponieważ za jego pomocą sprawdzamy, czy nasz kod (repozytorium) jest prawidłowo zintegrowany z bazą danych. Dlatego w kodzie są pomieszane zwykłe zapytania SQL z wywołaniami i asercjami dotyczącymi naszego własnego kodu.

W odróżnieniu od wcześniejszych testów ORM, te testy mają szansę zostać w naszej bazie kodu na dłużej, szczególnie jeśli jakiekolwiek części modelu domeny sprawiają, że mapowanie obiektowo-relacyjne nie jest proste.

*Test repozytorium do zapisywania obiektu (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) (1)
    session.commit() (2)

    rows = list(session.execute(
        'SELECT reference, sku, _purchased_quantity, eta FROM "batches"' (3)
    ))

    assert rows == [("batch1", "RUSTY-SOAPDISH", 100, None)]
```

(1) `repo.add()` to metoda, którą testujemy.

(2) Wywołanie `.commit()` umieściliśmy poza repozytorium i pozostawiliśmy je pod odpowiedzialnością wywołującego. To rozwiązanie ma zalety i wady. Niektóre z naszych powodów staną się jasne, gdy dojdziemy do rozdziału 6.

(3) Za pomocą zapytań SQL sprawdzamy, czy zostały zapisane odpowiednie dane.

Następny test dotyczy pobierania partii i alokacji, więc jest bardziej skomplikowany.

*Test repozytorium dotyczący pobierania skomplikowanego obiektu (test_repository.py)*

```
def insert_order_line(session):
    session.execute( (1)
        '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): (2)
    ...

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) (3)

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

    expected = model.Batch("batch1", "GENERIC-SOFA", 100, eta=None)
    assert retrieved == expected # Batch.__eq__ tylo porównuje referencję (3)
    assert retrieved.sku == expected.sku (4)
```

```
    assert retrieved._purchased_quantity == expected._purchased_quantity
    assert retrieved._allocations == { ❹
        model.OrderLine("order1", "GENERIC-SOFA", 12),
}
```

❶ To jest test odczytu, więc kod SQL przygotowuje dane do odczytania przez `repo.get()`.

❷ Oszczędzimy sobie rozwodzenia się nad `insert_batch` i `insert_allocation`. Chodzi o utworzenie paru partii i — w tej, która nas interesuje — alokowanie istniejącej pozycji zamówienia.

❸ To jest to, co sprawdzamy. Pierwsza asercja sprawdza, czy typy pasują oraz czy referencja jest taka sama (ponieważ, jak pamiętamy, `Batch` jest jednostką i mamy dla niej własną metodę `eq`).

❹ Sprawdzamy także najważniejsze atrybuty, w tym `._allocations`, który jest zbiorem obiektów wartości `OrderLine`.

To, czy należy napisać testy dla każdego modelu, jest kwestią otwartą. Kiedy napisze się testy tworzenia, modyfikowania i zapisywania dla jednej klasy, w przypadku pozostałych można poprzestać na minimalnym teście, a nawet zupełnie zaniechać testowania, jeśli wszystkie są zbudowane według jednego wzorca. W naszym przypadku konfiguracja ORM tworząca zbiór `._allocations` jest dość skomplikowana, więc zasługiwała na specjalny test.

Teraz mamy coś takiego:

*Typowe repozytorium (repository.py)*

```
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()
```

Punkt końcowy Flask może teraz wyglądać następująco:

*Użycie naszego repozytorium bezpośrednio w punkcie końcowym API*

```
@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
```

## Ćwiczenie dla czytelnika

Kiedyś na konferencji poświęconej tematyce DDD wpadliśmy na znajomego, który powiedział, że nie używał ORM już od 10 lat. Zarówno wzorzec Repozytorium, jak i ORM to abstrakcje czystego kodu SQL, więc używanie obu nie jest konieczne. Może w takim razie zaimplementujemy nasze repozytorium bez użycia ORM? Kod znajdziesz w archiwum na serwerze FTP (folder *r02_ćwiczenie*).

Są tam testy repozytorium, ale napisanie zapytań SQL pozostawiliśmy Tobie. To może być trudniejsze albo łatwiejsze, niż myślisz. Jak by nie było, najlepsze jest to, że dla reszty naszej aplikacji to bez znaczenia.

# Budowa imitacji repozytorium na potrzeby testów nie jest łatwa

Oto jedna z największych korzyści użycia wzorca Repozytorium:

*Prosta imitacja repozytorium wykorzystująca zbiór (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)
```

To proste opakowanie zbioru, więc wszystkie metody są jednowierszowe.

Użycie imitacji repozytorium w testach jest naprawdę łatwe i mamy prostą abstrakcję, która jest łatwa w użyciu i nie sprawia kłopotów w opisie.

*Przykład użycia imitacji repozytorium (test_api.py)*

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

W następnym rozdziale pokazujemy przykład użycia tej imitacji w praktyce.

Budowa imitacji dla abstrakcji jest doskonałym sposobem na sprawdzenie jakości projektu — jeśli imitacja jest trudna do napisania, to znaczy, że poziom skomplikowania abstrakcji może być zbyt duży.

# Czym są porty i adaptery w Pythonie

Nie zamierzamy nadmiernie zagłębiać się w zawiłości terminologiczne, ponieważ naszym podstawowym celem jest koncentracja na odwróceniu zależności, a specyficzne właściwości wykorzystywanej techniki nie są aż tak istotne. Mamy jednak świadomość, że ludzie używają odrobinę różniących się definicji.

Porty i adaptery pochodzą ze świata programowania obiektowego. My posługujemy się definicją głoszącą, że *port* jest *interfejsem* między naszą aplikacją i tym, co chcemy wyabstrahować, natomiast *adapter* to *implementacja* stojąca za tym interfejsem lub tą abstrakcją.

W języku Python nie ma interfejsów w ścisłym znaczeniu, w związku z czym, choć łatwo można zidentyfikować adapter, definicja portu może być trudniejsza. Jeśli używamy abstrakcyjnej klasy bazowej, to ona jest portem. Jeśli nie, to port jest kaczym typem, z którym zgadzają się Twoje adaptery i którego oczekuje Twoja podstawowa aplikacja — nazwy funkcji i metod oraz nazwy ich argumentów i typy.

Konkretnie mówiąc, w tym rozdziale `AbstractRepository` jest portem, a `SqlAlchemyRepository` i `Fake` ↪`Repository` są adapterami.

# Podsumowanie

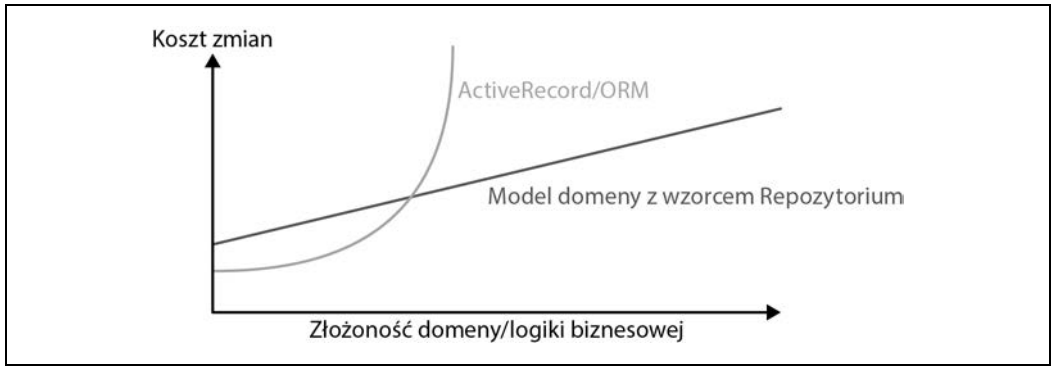
Pamiętając o przytoczonych słowach Richa Hickeya, w każdym rozdziale przedstawiamy podsumowanie zalet i kosztów każdego opisanego wzorca architekturalnego. Chcemy podkreślić, że nie każda aplikacja musi być zbudowana w ten sposób. Tylko czasami aplikacja i domena są na tyle skomplikowane, że warto zainwestować czas i wysiłek w dodanie kolejnych warstw abstrakcji.

W tabeli 2.1 znajduje się wykaz zalet i wad wzorca Repozytorium i naszego modelu niezależnego od sposobu przechowywania danych.

***Tabela 2.1.*** *Wzorzec Repozytorium i niezależność od sposobu przechowywania danych — kompromisy*

| Zalety | Wady |
|---|---|
| • Mamy prosty interfejs między magazynem danych i naszym modelem domeny.<br>• Łatwo możemy utworzyć imitację repozytorium na potrzeby testów jednostkowych i zmienić sposób przechowywania danych, ponieważ model jest całkowicie oddzielony od spraw infrastrukturalnych.<br>• Napisanie modelu domeny przed rozważeniem kwestii przechowywania danych pomaga nam skupić się na bieżącym problemie. Jeśli kiedyś zdecydujemy się radykalnie zmienić nasze podejście, możemy to zrobić w modelu, a kwestie związane z kluczami obcymi czy migracjami możemy odłożyć na później.<br>• Nasz schemat bazy danych jest bardzo prosty, ponieważ w pełni kontrolujemy sposób mapowania obiektów na tabele. | • Sam ORM także daje pewien stopień niezależności. Zmiana kluczy obcych może być trudna, ale już zmiana bazy danych MySQL na Postgres w razie potrzeby nie powinna sprawić kłopotów.<br>• Ręczne utrzymywanie mapowań ORM wymaga dodatkowej pracy i kodu.<br>• Każda dodatkowa warstwa abstrakcji zwiększa koszty utrzymania i wprawia w osłupienie programistów Pythona, którzy nigdy wcześniej nie mieli do czynienia z wzorcem Repozytorium. |

Na rysunku 2.6 jest przedstawiona prosta teza — tak, w prostych przypadkach niezależny model domeny jest trudniejszy do zaimplementowania niż prosty wzorzec ORM/ActiveRecord[8].




***Rysunek 2.6.*** *Koszty modelu domeny przedstawione na wykresie*

Jeśli Twoja aplikacja jest prostym opakowaniem CRUD (ang. *create-read-update-delete*) bazy danych, to nie potrzebujesz modelu domeny ani repozytorium.

Jednak im bardziej jest skomplikowana domena, tym większe korzyści da inwestycja w uwolnienie się od spraw infrastrukturalnych pod względem łatwości wprowadzania zmian.

Nasz kod nie jest wystarczająco skomplikowany, aby dać nam przynajmniej przedsmak tego, co widać po prawej stronie wykresu, ale możemy zauważyć pewne korzyści. Wyobraź sobie na przykład, że pewnego dnia postanawiamy przenieść alokacje z obiektu `Batch` do `OrderLine`. Gdybyśmy używali Django, musielibyśmy zdefiniować i przemyśleć migrację bazy danych, aby móc wykonać jakiekolwiek testy. Dzięki temu, że nasz model jest zwykłym obiektem Pythona, możemy zmienić `set()` na nowy atrybut, a konieczność przemyślenia struktury bazy danych możemy odłożyć na później.

## Podsumowanie wiadomości o wzorcu Repozytorium

**Zastosuj odwrócenie zależności do swojego ORM**
: Nasz model domeny powinien być wolny od kwestii związanych z infrastrukturą, a więc to ORM powinien importować model, nie odwrotnie.

**Wzorzec Repozytorium jest prostą abstrakcją magazynu danych**
: Repozytorium daje iluzję, że kolekcja obiektów znajduje się w pamięci. Ułatwia utworzenie imitacji repozytorium do testowania i pozwala na wymianę podstawowych elementów infrastruktury bez powodowania problemów z działaniem aplikacji. W dodatku C zamieściliśmy przykład.

[8] Schemat inspirowany artykułem *Global Complexity, Local Simplicity* (*https://oreil.ly/fQXkP*) Roba Vensa.

Pewnie się zastanawiasz, jak tworzyć obiekty tych repozytoriów, czy to prawdziwych, czy imitacji. Jak będzie wyglądała nasza aplikacja Flask? Tego dowiesz się w rozdziale, w którym opisujemy wzorzec Warstwa usług.

Przedtem jednak czeka Cię jeszcze krótka dygresja.

# Architektura nowoczesnych aplikacji w Pythonie

Python zyskuje coraz większą popularność i jest wykorzystywany do tworzenia różnych aplikacji, jednak projektowanie dużych, niezawodnych systemów w tym języku bywa wyzwaniem. Rozwijanie złożonych systemów o wysokiej jakości wymaga zastosowania odpowiedniej architektury. Trudno w Pythonie stosować takie wysokopoziomowe wzorce projektowe jak architektura sześciokątna, architektura oparta na zdarzeniach czy wzorce zalecane w projektowaniu dziedzinowym (DDD). Sytuacji nie poprawia również to, że klasyczna literatura dotycząca metod zarządzania złożonością aplikacji zawiera przykłady kodu napisanego w Javie lub C#.

Ten praktyczny przewodnik przybliży projektantom sprawdzone wzorce architektury, które ułatwiają zapanowanie nad złożonością aplikacji i pozwalają wykorzystać zestawy testów. Prezentację poszczególnych wzorców oparto na przykładowej, stopniowo rozbudowywanej aplikacji. Podejście to pozwoliło na pokazanie zalet metodyki TDD. W rozdziałach poświęconych modelowaniu dziedzinowemu zwrócono uwagę na unikanie zależności zewnętrznych przy zapewnieniu integralności danych. Wśród ciekawszych koncepcji warto wskazać wykorzystywanie zdarzeń w roli wzorca integracji usług w architekturze mikrousługowej. Zaprezentowano również praktyczne strony stosowania kilku frameworków i technologii Pythona, między innymi Flask, SQLAlchemy, pytest, Docker i Redis.

## W tej książce między innymi:

- modelowanie dziedzinowe i stosowanie wzorców DDD
- jednostki, obiekty wartości i agregaty w architekturze domenowej
- tworzenie modeli bez zbędnych zależności
- zdarzenia, polecenia i szyna wiadomości
- wzorce architektury zdarzeniowej i mikrousług reaktywnych

**Harry Percival** był konsultantem specjalizującym się w zarządzaniu, później brał udział w pracach nad arkuszem kalkulacyjnym Resolver One. Pracował w PythonAnywhere LLP i promował metodykę TDD na konferencjach, warsztatach i innych wydarzeniach na całym świecie. Obecnie pracuje w MADE.com.

**Bob Gregory** jest architektem oprogramowania w MADE.com. Od ponad dekady zajmuje się systemami sterowanymi zdarzeniami i architekturą domenową.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl



KOD KORZYŚCI
*Sięgnij po więcej!*

ISBN 978-83-283-7126-2

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 67,00 zł