Software Architecture and Design Patterns

Course information (Spring 2021)

This full-semester course introduces important architectural patterns for making software easier to build, test and maintain.

Architecture: carefully designed structure of something.

Required textbooks: None.

Recommended textbooks:

- Harry Percival and Bob Gregory. Architecture Patterns with Python. O'Reilly Media; 1st edition (March 31, 2020).

- Dusty Phillips. Python 3 Object Oriented Programming. Packt Publishing; 3rd edition (October 30, 2018).

A significant portion of the lecture notes is built on the above two books.

Recommended reference books:

- Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

- Martin Fowler. Patterns of Enterprise Application Architecture. Addison Wesley, 2002.

Grading policy:

Component	**Weight**
Quizzes	10
Labs	20
Course project	20
Final exam	50

Software freely available for this course:

- Python 3.9.2 interpreter:

  https://www.python.org/downloads/release/python-392

  [Scroll to the bottom to find an installer for your operating system.]

- Wing IDE 101: http://wingware.com/downloads/wing-101
- Flask: https://pypi.org/project/Flask
- SQLAlchemy: https://pypi.org/project/SQLAlchemy

What Is Wrong?

Big ball of mud.

Big ball of yarn.

Figure 1. A real-life dependency diagram

Both refer to a software that lacks a perceivable architecture.

Caution: with a circular layout, how messy the graph looks depends on how the nodes are arranged. So you cannot just say the software lacks architecture if it looks like a big ball of mud with the circle layout.

Online demo tool: https://dreampuf.github.io/GraphvizOnline

```
digraph {
    1->2;
    2->3;
    1->3;
    3->4;
}
```

Analogy: the garden runs wild without maintenance.

Chaotic software application – everything is coupled to everything else and changing any part is dangerous.

A common problem: business logic spread over all places.

A common practice: build a database schema first. Object model? Never thought about that.

Why Architecture?

MANAGE COMPLEXITY.

Structure the application to facilitate testing (unit, integration, end-to-end).

It needs efforts.

Layered design/archictecture (presentation layer – business logic – database layer). See Figure 5-2 Typical Application Layers. **Do not cross layers**. Like our government structure?

Levels of abstractions. (Note: this abstraction is different the abstraction we talked in the OOAD course. In this course, we call an object or function an abstraction.)

Different names: Hexagonal Architecture, Ports-and-Adapters, Onion Architecture, Clean Architecture.

Figure 5-7 Onion view of clean architecture.

Figure 3. Onion architecture in Chapter 2 of the textbook.

The OAPS Class Diagram from LiuSiyuan et al.

Undesirable Things during Software Construction

- Complexity.
- Big project size.
- Evolving requirements.
- Regression.
- High coupling.

- High cost in maintaining tests.
- Slow speed while testing the UI layer or the real data layer.
- Requiring a test database.

Encapsulation and Abstractions

Encapsulating **behavior** using abstractions.

Think on the level of behavior, not on the level of data or algorithm. BDD: *behavior should come first and drive our storage requirements.*.

How to understand abstraction? Public API. Interface. Function. Method.

Responsibility: search *sausages* from duckduckgo.com.

A low-level abstraction

```
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'])
```

A medium-level abstraction

```
import requests
```

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

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

A high-level abstraction

```
import duckduckpy
for r in duckduckpy.query('Sausages').related_topics:
    print(r.first_url, ' - ', r.text)
```

Which one is easier to understand?

Layering

When do we say one *depends on* the other?

Depends: uses, needs, knows, calls, imports.

Dependency graph/network.

Figure 2. Layered architecture

Things in Presentation Layer should not use things in Database Layer directly.

Domain Modeling

For business logic. Seperate business logic from infrastructure and from user-interface logic.

Domain model: a mental map that business owners have of their business, expressed in domain-specific terminologies. Describe the business core (most valuable and likely to change).

Domain model: another name for *business logic layer*.

Why? Keep the model decoupled from technical concerns.

Figure 1. A component diagram for our app at the end of

Building an Architecture to Support Domain Modeling

Important concepts:

- Value Object: uniquely identified by the data it holds (not by a long-lived ID).

```
@dataclass(frozen=True)
class OrderLine:
    orderid: OrderReference
    sku: ProductReference
    qty: Quantity
```

- Entity: uniquely identified by a unique ID.

- Aggregate.

- Domain Service.

  What is **domain**? *The problem (we are going to solve).*

  Example: an online furniture retailer (purchasing, product design, logistics, delivery, etc).

  Business jargons (domain language): source, goods, SKU (stock-keeping unit), supplier, stock, out of stock, warehouse, inventory, lead time, ETA, batch, order, order reference, order line, quantity, allocate, deliver, ship, shipment tracking, dispatch, etc. Check MADE.com.

  What is **model**? A map of a process that captures useful property.

Data model: database, database schema.

Decouple it (the model) from technical concerns (or infrastructure concerns).

Related topic: DDD.

Tips:

1. Ask the domain expert for glossary and concrete examples.

2. Start with a minimal version of the domain model.

3. Get a minimum viable product (MVP) as quickly as possible.

4. Add new classes instead of changing existing ones. Reason: won't affect existing code.

5. Do not share a database between two applications.

** Talk over Some Notes on Allocation.

Task: allocate order lines to batches.

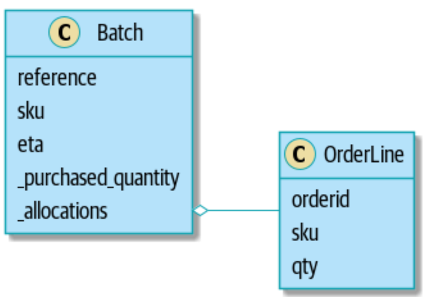
Our domain model expressed in Python:

```
from dataclasses import dataclass
from typing import Optional
```

```
from datetime import date

@dataclass(frozen=True)  #(1) (2)
class OrderLine:
    orderid: str
    sku: str
    qty: int


class Batch:
    def __init__(self, ref: str, sku: str, qty: int, eta: Optional[dat
#(2)
        self.reference = ref
        self.sku = sku
        self.eta = eta
        self.available_quantity = qty

    def allocate(self, line: OrderLine):
```

```python
        self.available_quantity -= line.qty  #(3)

    def can_allocate(self, line: OrderLine) -> bool:
        return self.sku == line.sku and self.available_quantity >= lin
```

Our model in a graphic form:



Real-world business has lots of edge cases:

- Deliver on a specific date.

- Deliver from a warehourse in a different region than the customer's location.

The EnglishPal story: the web UI was built after the domain model had been built. In fact, I never thought I would build a web application for EnglishPal.

MADE.com

Figure 2. Context diagram for the allocation service

- A global supply chain.

- Freight partners.

- Manufacturers.

- **Minimum storage**. Treat goods on the road as our stock.

Unit Testing Domain Models

Use unit test to describe the *behavior* of the system.

Think about how to use it before implementing it.

The test describes the *behavior* of the system from an interface point of view.

```
def test_allocating_to_a_batch_reduces_the_available_quantity():

    batch = Batch("batch-001", "SMALL-TABLE", qty=20, eta=date.today()
    line = OrderLine("order-ref", "SMALL-TABLE", 2)

    batch.allocate(line)
```

```
    assert batch.available_quantity == 18


### More tests ###
def make_batch_and_line(sku, batch_qty, line_qty):
    return (
        Batch("batch-001", sku, batch_qty, eta=date.today()),
        OrderLine("order-123", sku, line_qty),
    )

def test_can_allocate_if_available_greater_than_required():
    large_batch, small_line = make_batch_and_line("ELEGANT-LAMP", 20,
    assert large_batch.can_allocate(small_line)

def test_cannot_allocate_if_available_smaller_than_required():
    small_batch, large_line = make_batch_and_line("ELEGANT-LAMP", 2, 2
    assert small_batch.can_allocate(large_line) is False
```

```
def test_can_allocate_if_available_equal_to_required():
    batch, line = make_batch_and_line("ELEGANT-LAMP", 2, 2)
    assert batch.can_allocate(line)

def test_cannot_allocate_if_skus_do_not_match():
    batch = Batch("batch-001", "UNCOMFORTABLE-CHAIR", 100, eta=None)
    different_sku_line = OrderLine("order-123", "EXPENSIVE-TOASTER", 1
    assert batch.can_allocate(different_sku_line) is False
```

- allocate
- can_allocate
- deallocate

```
def test_can_only_deallocate_allocated_lines():
    batch, unallocated_line = make_batch_and_line("DECORATIVE-TRINKET"
    batch.deallocate(unallocated_line)
    assert batch.available_quantity == 20
```

Updated model:

```
class Batch:
    def __init__(self, ref: str, sku: str, qty: int, eta: Optional[dat
        self.reference = ref
        self.sku = sku
        self.eta = eta
        self._purchased_quantity = qty
        self._allocations = set()  # type: Set[OrderLine]

    def allocate(self, line: OrderLine):
        if self.can_allocate(line):
            self._allocations.add(line)
```

```
    def deallocate(self, line: OrderLine):
        if line in self._allocations:
            self._allocations.remove(line)

    @property
    def allocated_quantity(self) -> int:
        return sum(line.qty for line in self._allocations)

    @property
    def available_quantity(self) -> int:
        return self._purchased_quantity - self.allocated_quantity

    def can_allocate(self, line: OrderLine) -> bool:
        return self.sku == line.sku and self.available_quantity >= lin
```

Can the above model pass the following test?

```python
def test_allocation_is_idempotent():
    batch, line = make_batch_and_line("ANGULAR-DESK", 20, 2)
    batch.allocate(line)
    batch.allocate(line)
    assert batch.available_quantity == 18
```

Easy to test as no database is involved.

Value Objects

For something that has data but no ID. For example, order line, name, and money.

Value Object pattern. Value objects have *value equality*.

Orer line:

```
@dataclass(frozen=True)
class OrderLine:
    orderid: OrderReference
    sku: ProductReference
    qty: Quantity
```

Name and money:

```
from dataclasses import dataclass
from typing import NamedTuple
from collections import namedtuple

@dataclass(frozen=True)
class Name:
    first_name: str
    surname: str

class Money(NamedTuple):
    currency: str
    value: int

Line = namedtuple('Line', ['sku', 'qty'])

def test_equality():
    assert Money('gbp', 10) == Money('gbp', 10)
```

```
    assert Name('Harry', 'Percival') != Name('Bob', 'Gregory')
    assert Line('RED-CHAIR', 5) == Line('RED-CHAIR', 5)

def test_name_equality():
    assert Name("Harry", "Percival") != Name("Barry", "Percival")
```

Math for value objects:

```
fiver = Money('gbp', 5)
tenner = Money('gbp', 10)

def can_add_money_values_for_the_same_currency():
    assert fiver + fiver == tenner

def can_subtract_money_values():
    assert tenner - fiver == fiver

def adding_different_currencies_fails():
```

```
    with pytest.raises(ValueError):
        Money('usd', 10) + Money('gbp', 10)

def can_multiply_money_by_a_number():
    assert fiver * 5 == Money('gbp', 25)

def multiplying_two_money_values_is_an_error():
    with pytest.raises(TypeError):
        tenner * fiver
```

Entities

For something with a permanent ID (that is identified by a reference). For example, a person (who has a permanent identity).

```
class Person:

    def __init__(self, name: Name):
        self.name = name


def test_barry_is_harry():
    harry = Person(Name("Harry", "Percival"))
    barry = harry
```

```python
    barry.name = Name("Barry", "Percival")

    assert harry is barry and barry is harry
```

Entities have *identity equality*. Unlike value objects, an entitie's attributes can change without making it a different entity (as long as its identity keeps the same).

Define equality operator (__eq__) on entities:

```python
class Batch:
    ...

    def __eq__(self, other):
        if not isinstance(other, Batch):
            return False
```

```
        return other.reference == self.reference

    def __hash__(self):
        return hash(self.reference)
```

Domain Service

It is just a function.

`allocate()`: A service that allocates an order line, given a list of batches.

Testing script:

```
def test_prefers_current_stock_batches_to_shipments():
    in_stock_batch = Batch("in-stock-batch", "RETRO-CLOCK", 100, eta=N
    shipment_batch = Batch("shipment-batch", "RETRO-CLOCK", 100, eta=t
    line = OrderLine("oref", "RETRO-CLOCK", 10)

    allocate(line, [in_stock_batch, shipment_batch])
```

```
    assert in_stock_batch.available_quantity == 90
    assert shipment_batch.available_quantity == 100


def test_prefers_earlier_batches():
    earliest = Batch("speedy-batch", "MINIMALIST-SPOON", 100, eta=toda
    medium = Batch("normal-batch", "MINIMALIST-SPOON", 100, eta=tomorr
    latest = Batch("slow-batch", "MINIMALIST-SPOON", 100, eta=later)
    line = OrderLine("order1", "MINIMALIST-SPOON", 10)

    allocate(line, [medium, earliest, latest])

    assert earliest.available_quantity == 90
    assert medium.available_quantity == 100
    assert latest.available_quantity == 100
```

```python
def test_returns_allocated_batch_ref():
    in_stock_batch = Batch("in-stock-batch-ref", "HIGHBROW-POSTER", 10
    shipment_batch = Batch("shipment-batch-ref", "HIGHBROW-POSTER", 10
    line = OrderLine("oref", "HIGHBROW-POSTER", 10)
    allocation = allocate(line, [in_stock_batch, shipment_batch])
    assert allocation == in_stock_batch.reference
```

Model:

```python
def allocate(line: OrderLine, batches: List[Batch]) -> str:
    batch = next(b for b in sorted(batches) if b.can_allocate(line))
    batch.allocate(line)
    return batch.reference
```

For the `sorted()` method to work, we need to define an order function `__gt__`:

```python
class Batch:
```

```python
    ...

    def __gt__(self, other):
        if self.eta is None:
            return False
        if other.eta is None:
            return True
        return self.eta > other.eta
```

Tips:

- Instead of FooManager, use manage_foo().
- Instead of BarBuilder, use build_bar().
- Instead of BazFactory, use get_baz().

Use Exceptions to Express Domain Concepts

A domain concept: out of stock.

```
def test_raises_out_of_stock_exception_if_cannot_allocate():
    batch = Batch("batch1", "SMALL-FORK", 10, eta=today)
    allocate(OrderLine("order1", "SMALL-FORK", 10), [batch])

    with pytest.raises(OutOfStock, match="SMALL-FORK"):
        allocate(OrderLine("order2", "SMALL-FORK", 1), [batch])
```

Define the exception class:

```
class OutOfStock(Exception):
    pass
```

```python
def allocate(line: OrderLine, batches: List[Batch]) -> str:
    try:
        batch = next(
        ...
    except StopIteration:
        raise OutOfStock(f"Out of stock for sku {line.sku}")
```

The Domail Model in A Graphical Form

Figure 4. Our domain model at the end of the chapter

What do we lack? A database!

Business Logic

Ball-of-mud Problem: business logic spreads all over the place. "Business logic" classes that perform no calculations but do perform I/O.

Customers do not care about how the data is stored (MySQL, SQLite, Redis, etc).

What do they care?

Domain model includes business logic.

Hiding Implementation Details

Wrong: build a database schema first for whatever software application.

Correct: behavior should come first and drive our storage requirements.

Public contract.

Architectural Patterns

- Repository pattern - for persistent storage.
- Service layer - for use cases.
- Unit of Work pattern - for atomic operations.

Repository Patterns

We don't want our domain model to depend on infrastructure in afraid of slowing down unit tests or making changes hard.

An abstraction over persistent storage (`add()` and `get()`).

A repository pattern pretends that all data is *in memory*.

Usage example:

```
import all_my_data

def create_a_batch():
    batch = Batch(...)
```

```python
    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)
```

Abstract Repository (allowing us to use `repo.get()` instead of `session.query()`):

```python
class AbstractRepository(abc.ABC):
    @abc.abstractmethod  #(1)
    def add(self, batch: model.Batch):
        raise NotImplementedError  #(2)

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

A repository object sits between our domain model and the database. Check Figure 5. Repository pattern in Chapter 2.

Avoid using raw SQL in the domain model.

Check Figure 1. Before and after the Repository pattern in Chapter 2.

- Retrieve batch info from database.

- Save batch info to database.

Check Figure 3 (onion architecture) in Chapter 2.

Dependency Inversion Principle (DIP): high-level modules (the domain) should not depend on low-level ones (the infrastructure).

DIP

Why? Decouple core logic from infrastructural concerns.

What?

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend on details. Instead, details should depend on abstractions.

Low-level things: file systems, sockets, SMTP, IMAP, HTTP, cron job, Kubernetes, etc.

How? Depending on abstractions. Interfaces. Let infrastructure depend on domain.

```
def allocate(line: OrderLine, repo: AbstractRepository, session) -> st
```

We can use `SqlAlchemyRepository` or `CsvRepository`.

Check Figure 4-1. Direct denpendency graph and Figure 4-2. Inverted dependency graph for illustration.

Benfits: allowing high-level modules to change more independently of low-level modules, vice versa.

Persisting the Domain Model

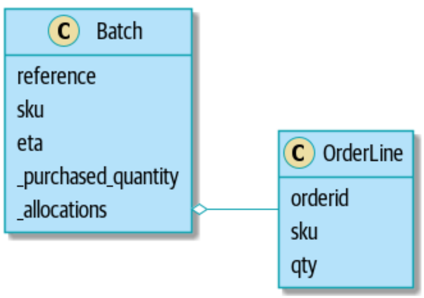
Need a database.

1. Retrieve batch information from database.
2. Instantiate domain objects.

```
@flask.route.gubbins  # gubbins means stuff
def allocate_endpoint():
    # extract order line from request
    line = OrderLine(request.params, ...)
    # load all batches from the DB
    batches = ...
```

```python
    # call our domain service
    allocate(line, batches)
    # then save the allocation back to the database somehow
    return 201
```

Translate the model to a relational database.



Model Depends on ORM

ORM - object-relational mapper

O - the world of objects and domain modeling.

R - the world of databases and relational algebra.

Benefit of ORM: **persistence ignorance**.

Heavily depend on SQLAlchemy's ORM now:

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

```python
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):
    ...
```

Is the above model indenpendent (ignorant) of database?

No. It is coupled to `Column`!

Who depends on whom? Look at who uses (imports) whom.

ORM Depends on Model

Principle: **Persistence Ignorance** (PI). Ignore particular database technologies.

Purpose: make the domain model stay "pure".

Explicit ORM mapping with SQLAlchemy Table objects.

Note the statement `import model`.

```
# orm.py
from sqlalchemy.orm import mapper, relationship

import model  #(1)
```

```
metadata = MetaData()

order_lines = Table(  #(2)
    "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)  #(3)
```

Result: model (model.py) stays "pure". Let ORM (orm.py) do ugly things.

Testing The ORM

`session`: a SQLAlchemy database session.

```
def test_orderline_mapper_can_load_lines(session):   #(1)
    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_
    assert rows == [("order1", "DECORATIVE-WIDGET", 12)]
```

Testing The Repository

We use a more specific repository here: `SqlAlchemyRepository`.

From `session.query(Batch)` (the SQLAlchemy language) to `batches_repo.get` (the repository pattern).

Test `add` - saving an object

```
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 = session.execute(  #(3)
        'SELECT reference, sku, _purchased_quantity, eta FROM "batches
    )
    assert list(rows) == [("batch1", "RUSTY-SOAPDISH", 100, None)]
```

Test get - retrieving a complex object.

```
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=:sk
        dict(orderid="order1", sku="GENERIC-SOFA"),
    )
    return orderline_id
```

```python
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)  #(2)

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

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

```
    assert retrieved._purchased_quantity == expected._purchased_quanti
    assert retrieved._allocations == {  #(4)
        model.OrderLine("order1", "GENERIC-SOFA", 12),
    }
```

SqlAlchemyRepository:

```
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=ref

    def list(self):
```

```
        return self.session.query(model.Batch).all()
```

Updated API Endpoint

Not using Repository Pattern:

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

    # extract order line from request
    line = OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

    # load all batches from the DB
```

```python
    batches = session.query(Batch).all()

    # call our domain service
    allocate(line, batches)

    # save the allocation back to the database
    session.commit()

    return 201
```

Using Repository Pattern:

```python
@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
```

Coupling and Abstractions

Problem: high-level code is coupled to low-level details.

Tricks:

- Use simple abstractions to hide mess.
- Separate *what* we want to do from *how* to do it.
- Make interface between business logic and messy I/O.

Coupling: changing component A may break component B.

Locally, high coupling is not a bad thing – this means high cohesion.

Globally, coupling is bad.

Figure 1. Lots of coupling in Chapter 3.

Figure 2. Less coupling in Chapter 3.

Example: directory sync.

Imperative Shell, Functional Core.

The Service Layer

This layer deals with *orchestration logic*.

What does the *Service Layer* pattern look like?

How is it different from *entrypoint*, i.e., Flask API endpoint?

How is it different from *domain model*, i.e., business logic?

Flask API talks to the service layer; the service layers talks to the domain model.

Don't forget our task: allocate an **order line** to a **batch**.

BTW, the authors faithfully use TDD (test code is even more than non-test code):

```
 476 Apr 10 19:47 config.py
2.9K Apr 10 19:47 conftest.py
 794 Apr 10 19:47 flask_app.py
2.0K Apr  7 10:02 model.py
1.3K Apr  7 10:02 orm.py
 650 Apr  7 10:02 repository.py
 541 Apr 10 19:47 services.py
1.8K Apr  7 10:02 test_allocate.py
1.5K Apr 10 19:47 test_api.py
1.9K Apr  7 10:02 test_batches.py
3.0K Apr  7 10:02 test_orm.py
2.2K Apr  7 10:02 test_repository.py
1.4K Apr 10 19:47 test_services.py
```

Test code: 11.8 KB

Functional code: 8.7KB

Fast tests (e.g., unit tests) and slow tests (e.g., E2E tests).

Figure 2. The service layer will become the main way into our app.

What we have now:

- Domain model.
- Domain service (`allocate`).
- Repsitory interface.

```
@pytest.mark.usefixtures("restart_api")
def test_api_returns_allocation(add_stock):
    sku, othersku = random_sku(), random_sku("other")  #(1)
    earlybatch = random_batchref(1)
```

```python
laterbatch = random_batchref(2)
otherbatch = random_batchref(3)
add_stock(  #(2)
    [
        (laterbatch, sku, 100, "2011-01-02"),
        (earlybatch, sku, 100, "2011-01-01"),
        (otherbatch, othersku, 100, None),
    ]
)
data = {"orderid": random_orderid(), "sku": sku, "qty": 3}
url = config.get_api_url()  #(3)

r = requests.post(f"{url}/allocate", json=data)

assert r.status_code == 201
assert r.json()["batchref"] == earlybatch
```

```
@pytest.mark.usefixtures("restart_api")
def test_allocations_are_persisted(add_stock):
    sku = random_sku()
    batch1, batch2 = random_batchref(1), random_batchref(2)
    order1, order2 = random_orderid(1), random_orderid(2)
    add_stock(
        [(batch1, sku, 10, "2011-01-01"), (batch2, sku, 10, "2011-01-0
    )
    line1 = {"orderid": order1, "sku": sku, "qty": 10}
    line2 = {"orderid": order2, "sku": sku, "qty": 10}
    url = config.get_api_url()

    # first order uses up all stock in batch 1
    r = requests.post(f"{url}/allocate", json=line1)
    assert r.status_code == 201
    assert r.json()["batchref"] == batch1
```

```python
    # second order should go to batch 2
    r = requests.post(f"{url}/allocate", json=line2)
    assert r.status_code == 201
    assert r.json()["batchref"] == batch2




@pytest.mark.usefixtures("restart_api")
def test_400_message_for_out_of_stock(add_stock):   #(1)
    sku, smalL_batch, large_order = random_sku(), random_batchref(), r
    add_stock(
        [(smalL_batch, sku, 10, "2011-01-01"),]
    )
    data = {"orderid": large_order, "sku": sku, "qty": 20}
    url = config.get_api_url()
    r = requests.post(f"{url}/allocate", json=data)
    assert r.status_code == 400
```

```
    assert r.json()["message"] == f"Out of stock for sku {sku}"



@pytest.mark.usefixtures("restart_api")
def test_400_message_for_invalid_sku():  #(2)
    unknown_sku, orderid = random_sku(), random_orderid()
    data = {"orderid": orderid, "sku": unknown_sku, "qty": 20}
    url = config.get_api_url()
    r = requests.post(f"{url}/allocate", json=data)
    assert r.status_code == 400
    assert r.json()["message"] == f"Invalid sku {unknown_sku}"
```

@pytest.mark.usefixtures("restart_api"): call `restart_api` in `conftest.py` before carrying out the test.

```
from flask import Flask, request
from sqlalchemy import create_engine
```

```python
from sqlalchemy.orm import sessionmaker

import config
import model
import orm
import repository


orm.start_mappers()
get_session = sessionmaker(bind=create_engine(config.get_postgres_uri(
app = Flask(__name__)


def is_valid_sku(sku, batches):
    return sku in {b.sku for b in batches}


@app.route("/allocate", methods=["POST"])
```

```python
def allocate_endpoint():
    session = get_session()
    batches = repository.SqlAlchemyRepository(session).list()
    line = model.OrderLine(
        request.json["orderid"], request.json["sku"], request.json["qt
    )

    if not is_valid_sku(line.sku, batches):
        return {"message": f"Invalid sku {line.sku}"}, 400

    try:
        batchref = model.allocate(line, batches)
    except model.OutOfStock as e:
        return {"message": str(e)}, 400

    session.commit()
    return {"batchref": batchref}, 201
```

Consequence: inverted test pyramid (ice-cream cone model).

Solution: introduce a service layer and put *orchestration logic* in this layer.

Steps:

- Fetch some objects from the repository.
- Make some checks.
- Call a domain service.
- Save/update any state that has been changed.

```python
class InvalidSku(Exception):
    pass


def is_valid_sku(sku, batches):
    return sku in {b.sku for b in batches}


def allocate(line: OrderLine, repo: AbstractRepository, session) -> st
    batches = repo.list()  #(1)
    if not is_valid_sku(line.sku, batches):  #(2)
        raise InvalidSku(f"Invalid sku {line.sku}")
    batchref = model.allocate(line, batches)  #(3)
    session.commit()  #(4)
    return batchref
```

Two awkwardness in the service layer: (1) Coupled to

`OrderLine`. (2) Coupled to `session`.

Depend on Abstractions: `def allocate(line: OrderLine, repo: AbstractRepository, session)`

Let the Flask API endpoint do "web stuff" only.

```
@app.route("/allocate", methods=["POST"])
def allocate_endpoint():
    session = get_session()  #(1)
    repo = repository.SqlAlchemyRepository(session)  #(1)
    line = model.OrderLine(
        request.json["orderid"], request.json["sku"], request.json["qt
#(2)
    )

    try:
        batchref = services.allocate(line, repo, session)  #(2)
```

```
    except (model.OutOfStock, services.InvalidSku) as e:
        return {"message": str(e)}, 400  (3)

    return {"batchref": batchref}, 201  (3)
```

E2E tests: happy path and unhappy path.

```
@pytest.mark.usefixtures("restart_api")
def test_happy_path_returns_201_and_allocated_batch(add_stock):
    sku, othersku = random_sku(), random_sku("other")
    earlybatch = random_batchref(1)
    laterbatch = random_batchref(2)
    otherbatch = random_batchref(3)
    add_stock(
        [
            (laterbatch, sku, 100, "2011-01-02"),
            (earlybatch, sku, 100, "2011-01-01"),
            (otherbatch, othersku, 100, None),
```

```python
        ]
    )
    data = {"orderid": random_orderid(), "sku": sku, "qty": 3}
    url = config.get_api_url()

    r = requests.post(f"{url}/allocate", json=data)

    assert r.status_code == 201
    assert r.json()["batchref"] == earlybatch


@pytest.mark.usefixtures("restart_api")
def test_unhappy_path_returns_400_and_error_message():
    unknown_sku, orderid = random_sku(), random_orderid()
    data = {"orderid": orderid, "sku": unknown_sku, "qty": 20}
    url = config.get_api_url()
    r = requests.post(f"{url}/allocate", json=data)
    assert r.status_code == 400
```

```
    assert r.json()["message"] == f"Invalid sku {unknown_sku}"
```

Service layer vs. domain service. For example, where should we put the responsiblity of *Calculating Tax*?

Creating and Organizing Folders to Show Architecture

Check Putting Things in Folders to See Where It All Belongs.

Unit of Work Pattern

UoW (pronounced *you-wow*): Unit of Work

Without UoW:

Check Figure 1. Without UoW: API talks directly to three layers

- The Flask API talks directly to the database layer to start a session.
- The Flask API talks to the repository layer to initialize

SQLAlchemyRepository.

- The Flask API talks to the service layer to ask it to allocate.

With UoW:

Check Figure 2. With UoW: UoW now manages database state

Benefit: Decouple service layer from the data layer.

- The Flask API initializes a unit of work.

- The Flask API invokes a service.

Repository as a collaborator class for UnitOfWork.

service_layer/services.py:

```
def allocate(
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:  #(1)
        batches = uow.batches.list()  #(2)
        ...
        batchref = model.allocate(line, batches)
        uow.commit()  #(3)
```

Use `uow` as a context manager (see this tutorial for more explanation on Context Managers).

`uow.batches` gives us a repo.

integration/test_uow.py

```
def test_uow_can_retrieve_a_batch_and_allocate_to_it(session_factory):
    session = session_factory()
    insert_batch(session, "batch1", "HIPSTER-WORKBENCH", 100, None)
    session.commit()

    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)  #(1)
    with uow:
        batch = uow.batches.get(reference="batch1")  #(2)
        line = model.OrderLine("o1", "HIPSTER-WORKBENCH", 10)
        batch.allocate(line)
        uow.commit()  #(3)

    batchref = get_allocated_batch_ref(session, "o1", "HIPSTER-WORKBEN
    assert batchref == "batch1"
```

Now define UoW

```
class AbstractUnitOfWork(abc.ABC):
    batches: repository.AbstractRepository  #(1)

    def __exit__(self, *args):  #(2)
        self.rollback()  #(4)

    @abc.abstractmethod
    def commit(self):  #(3)
        raise NotImplementedError

    @abc.abstractmethod
    def rollback(self):  #(4)
        raise NotImplementedError
```

```python
DEFAULT_SESSION_FACTORY = sessionmaker(  #(1)
    bind=create_engine(
        config.get_postgres_uri(),
    )
)


class SqlAlchemyUnitOfWork(AbstractUnitOfWork):
    def __init__(self, session_factory=DEFAULT_SESSION_FACTORY):
        self.session_factory = session_factory  #(1)

    def __enter__(self):
        self.session = self.session_factory()  # type: Session
#(2)
        self.batches = repository.SqlAlchemyRepository(self.session)
#(2)
        return super().__enter__()
```

```
    def __exit__(self, *args):
        super().__exit__(*args)
        self.session.close()  #(3)

    def commit(self):  #(4)
        self.session.commit()

    def rollback(self):  #(4)
        self.session.rollback()
```

Use the UoW in the service layer

```
def add_batch(
    ref: str, sku: str, qty: int, eta: Optional[date],
    uow: unit_of_work.AbstractUnitOfWork,  #(1)
):
    with uow:
```

```
        uow.batches.add(model.Batch(ref, sku, qty, eta))
        uow.commit()


def allocate(
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,  #(1)
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        batches = uow.batches.list()
        if not is_valid_sku(line.sku, batches):
            raise InvalidSku(f"Invalid sku {line.sku}")
        batchref = model.allocate(line, batches)
        uow.commit()
    return batchref
```

Aggregate Pattern

A collection of entities (domain objects) treated a unit for data changes. Example: a shopping cart (encapsulating many items).

An *aggregate* provides a single Consistency Boundary. A shopping cart provides a consistency bounary. We do not change the shopping carts of several customers at the same time. Cart is the accesible root entity, while other entities are not directly accessible from outside.

Example: Aggregates with multiple or single entities

Without any design patterns: CSV over SMTP / spreadsheet over email. Low initial complexity. Does not scale well.

Invariants – things that must be true after we finish an operation.

Constraints – double-booking is not allowed.

Business rules for order line allocation: 1 and 2.

Concurrency and locks: allocating 360,000 order lines per hour. How many order lines per second? Too slow if we lock the `batches` table while allocating each order line.

Not OK: allocate multiple order lines for the same product

DEADLY-SPOON at the same time.

OK: allocate multiple order lines for *different* products at the same time.

Bounded Context:

- `Product(sku, batches)` for allocation service.
- `Product(sku, description, price, image_url)` for ecommerce.

Define the aggregate Product:

```python
class Product:
    def __init__(self, sku: str, batches: List[Batch]):
        self.sku = sku  #(1)
        self.batches = batches  #(2)

    def allocate(self, line: OrderLine) -> str:  #(3)
        try:
            batch = next(b for b in sorted(self.batches) if b.can_allo
            batch.allocate(line)
            return batch.reference
        except StopIteration:
            raise OutOfStock(f"Out of stock for sku {line.sku}")
```

What has changed in the repository? The get method.

```python
class SqlAlchemyRepository(AbstractRepository):
    def __init__(self, session):
        self.session = session
```

```
    def add(self, product):
        self.session.add(product)

    def get(self, sku):
        return self.session.query(model.Product).filter_by(sku=sku).fi
```

How does unit of work look like now?

```
class SqlAlchemyUnitOfWork(AbstractUnitOfWork):
    def __init__(self, session_factory=DEFAULT_SESSION_FACTORY):
        self.session_factory = session_factory

    def __enter__(self):
        self.session = self.session_factory()  # type: Session
        self.products = repository.SqlAlchemyRepository(self.session)
        return super().__enter__()
```



```
    def __exit__(self, *args):
        super().__exit__(*args)
        self.session.close()

    def commit(self):
        self.session.commit()

    def rollback(self):
        self.session.rollback()
```

How does service `allocate` look like now?

```
def add_batch(
    ref: str, sku: str, qty: int, eta: Optional[date],
    uow: unit_of_work.AbstractUnitOfWork,
):
    with uow:
        product = uow.products.get(sku=sku)
```

```
        if product is None:
            product = model.Product(sku, batches=[])
            uow.products.add(product)
        product.batches.append(model.Batch(ref, sku, qty, eta))
        uow.commit()


def allocate(
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f"Invalid sku {line.sku}")
        batchref = product.allocate(line)
        uow.commit()
```

```
return batchref
```

Domain Events Pattern & Message Bus Pattern

Task: send out an email alert when allocation is failed due to out of stock.

Possible solutions:

- Dump the email sending stuff in `flask_app.py`, the HTTP layer (the endpoint). Quick and dirty. See function `send_mail`.

```
@app.route("/allocate", methods=["POST"])
```

```python
def allocate_endpoint():
    line = model.OrderLine(
        request.json["orderid"],
        request.json["sku"],
        request.json["qty"],
    )
    try:
        uow = unit_of_work.SqlAlchemyUnitOfWork()
        batchref = services.allocate(line, uow)
    except (model.OutOfStock, services.InvalidSku) as e:
        send_mail(
            "out of stock",
            "stock_admin@made.com",
            f"{line.orderid} - {line.sku}"
        )
        return {"message": str(e)}, 400

    return {"batchref": batchref}, 201
```

Question: Should the HTTP layer be responsible for sending out alert email messages? This may violate the Thin Web Controller Principle, as well violate Single Responsibility Principle (**SRP**). Also, how to do unit testing?

- Move the email sending stuff to the domain layer, i.e., `model.py`.

  Question: Is the domain layer for sending email, or for allocating an order line?

- Move the email sending stuff to the service layer, i.e., `services.py`.

```
def allocate(
```

```python
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f"Invalid sku {line.sku}")
        try:
            batchref = product.allocate(line)
            uow.commit()
            return batchref
        except model.OutOfStock:
            email.send_mail("stock@made.com", f"Out of stock for {li
            raise
```

`allocate` or `allocate_and_send_mail_if_out_of_stock`?

Rule of thumb: one function for one thing.

Where does the responsibility of sending alert belong? Message Bus!

- Make an Event class for the `Out of Stock` event. Check `domain/events.py`.

```
from dataclasses import dataclass


class Event:  #(1)
    pass


@dataclass
class OutOfStock(Event):  #(2)
```

```
    sku: str
```

An event becomes a value object. Do you still remember that `OutOfStock` is an exception class in the previous chapters?

- Add an attribute called `events` in class `Product`.
- Append this event object in `events` – raise the event.

Write test before modifying `model.py` and other files.

```
def test_records_out_of_stock_event_if_cannot_allocate():
    batch = Batch("batch1", "SMALL-FORK", 10, eta=today)
    product = Product(sku="SMALL-FORK", batches=[batch])
    product.allocate(OrderLine("order1", "SMALL-FORK", 10))
```

```
    allocation = product.allocate(OrderLine("order2", "SMALL-FORK", 1)
    assert product.events[-1] == events.OutOfStock(sku="SMALL-FORK")
#(1)
    assert allocation is None
```

Note: `product.events[-1]`.

Now let's update `model.py`.

```
from . import events


class Product:
    def __init__(self, sku: str, batches: List[Batch], version_number:
        self.sku = sku
        self.batches = batches
        self.version_number = version_number
        self.events = []  # type: List[events.Event]
```

```
    def allocate(self, line: OrderLine) -> str:
        try:
            batch = next(b for b in sorted(self.batches) if b.can_allo
            batch.allocate(line)
            self.version_number += 1
            return batch.reference
        except StopIteration:
            self.events.append(events.OutOfStock(line.sku))
            return None
```

Note: we no longer raise the `OutOfStock` exception now.

Message Bus

Purpose: to provide a unified way of invoking use cases from the endpoint.

Check Figure 1. Events flowing through the system in Chapter 8 of the textbook.

Method: mapping an event to a list of handlers (function names).

It is a dictionary.

A message bus has an important function called `handle`.

```python
HANDLERS = {
    events.OutOfStock: [send_out_of_stock_notification],
}  # type: Dict[Type[events.Event], List[Call


def handle(event: events.Event):
    for handler in HANDLERS[type(event)]:
        handler(event)


def send_out_of_stock_notification(event: events.OutOfStock):
    email.send_mail(
        "stock@made.com",
        f"Out of stock for {event.sku}",
    )
```

`HANDLERS` is a dictionary. Its key is an event type, and its value is a list of functions that will do something for that type of event.

In which folder should we put `messagebus.py`?

Where will the function `handle` be called?

Let Service Layer Publish Events to the Message Bus

services.py:

```
from . import messagebus
...

def allocate(
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
```

```
            raise InvalidSku(f"Invalid sku {line.sku}")
        try:  #(1)
            batchref = product.allocate(line)
            uow.commit()
            return batchref
        finally:  #(1)
            messagebus.handle(product.events)  #(2)
```

Let Service Layer Raise Events

`services.py`:

```
def allocate(
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f"Invalid sku {line.sku}")
        batchref = product.allocate(line)
        uow.commit() #(1)
```

```python
        if batchref is None:
            messagebus.handle(events.OutOfStock(line.sku))
        return batchref
```

Let UoW Publish Events to the Message Bus

... and message bus handles them.

```
from . import messagebus


class AbstractUnitOfWork(abc.ABC):
    products: repository.AbstractRepository

    def __enter__(self) -> AbstractUnitOfWork:
        return self

    def __exit__(self, *args):
        self.rollback()
```

```python
    def commit(self):
        self._commit()
        self.publish_events()

    def publish_events(self):
        for product in self.products.seen:
            while product.events:
                event = product.events.pop(0)
                messagebus.handle(event)

    @abc.abstractmethod
    def _commit(self):
        raise NotImplementedError

    @abc.abstractmethod
    def rollback(self):
        raise NotImplementedError
```

Note: `products` is a repository object. `product` is a `Product` object which contains a list of batches.

Question: how to understand the `seen` in `repostory.py`?

Magic. Most complex design as the authors said!

Event Storming

Treat events as first-class citizens.

- BatchCreated
- BatchQuantityChanged
- AllocationRequired
- OutOfStock

Each of the above events can be denoted as a `dataclass`.

How does the endpoint `flask_app.py` look like now?

```
# add batch
event = events.BatchCreated(
               request.json["ref"],
               request.json["sku"],
               request.json["qty"], eta
        )

messagebus.handle(event, unit_of_work.SqlAlchemyUnitOfWork())


# allocate order line
event = events.AllocationRequired(
```

```
                request.json["orderid"],
                request.json["sku"],
                request.json["qty"]
            )

    messagebus.handle(event, unit_of_work.SqlAlchemyUnitOfWork())
```

From Services to Event Handlers

Convert use-case functions to event handlers.

`services.py` is replaced by `messagebus.py`.

```
def handle(
    event: events.Event,
    uow: unit_of_work.AbstractUnitOfWork,
):
    results = []
    queue = [event]
    while queue:
        event = queue.pop(0)
        for handler in HANDLERS[type(event)]:
            results.append(handler(event, uow=uow))
```

```python
            queue.extend(uow.collect_new_events())
    return results


HANDLERS = {
    events.BatchCreated: [handlers.add_batch],
    events.BatchQuantityChanged: [handlers.change_batch_quantity],
    events.AllocationRequired: [handlers.allocate],
    events.OutOfStock: [handlers.send_out_of_stock_notification],
}  # type: Dict[Type[events.Event], List[Callable]]
```

Question: what does `uow.collect_new_events()` do?

Handle a series of old and new events using `handler`.

Use a handler to handle an incoming event. This handler could make another event, which is to be handled by another handler.

A possibility of circular dependency?

Example: the event `BatchQuantityChanged` is handled by `change_batch_quantity`, which could in turn make an event called `AllocationRequired`.

`change_batch_quantity` in `model.py`.

```
class Product:
    def __init__(self, sku: str, batches: List[Batch], version_number: int = 0):
        self.sku = sku
        self.batches = batches
        self.version_number = version_number
        self.events = []  # type: List[events.Event]
```

```
def allocate(self, line: OrderLine) -> str:
    try:
        batch = next(b for b in sorted(self.batches) if b.can_allocate(line)
        batch.allocate(line)
        self.version_number += 1
        return batch.reference
    except StopIteration:
        self.events.append(events.OutOfStock(line.sku))
        return None

def change_batch_quantity(self, ref: str, qty: int):
    batch = next(b for b in self.batches if b.reference == ref)
    batch._purchased_quantity = qty
    while batch.available_quantity < 0:
        line = batch.deallocate_one()
        self.events.append(  ##########################################
            events.AllocationRequired(line.orderid, line.sku, line.qty)
        )
```

Message Processor

Now the service layer becomes an **event processor**.

Before: uow pushes events onto the message bus.

Now: the message bus pulls events from the uow.

Commands and Command Handlers

BatchCreated versus CreateBatch.

BatchCreated - event (broadcast knowledge, for notification).

CreateBatch - command (capture intent).

Examples:

```
class Command:
    pass
```

```
@dataclass
class CreateBatch(Command):  #(2)
    ref: str
    sku: str
    qty: int
    eta: Optional[date] = None


@dataclass
class Allocate(Command):  #(1)
    orderid: str
    sku: str
    qty: int


@dataclass
class ChangeBatchQuantity(Command):  #(3)
```

```
    ref: str
    qty: int
```

Separate two kinds of handlers, `EVENT_HANDLERS` and `COMMAND_HANDLERS`:

```
EVENT_HANDLERS = {
    events.OutOfStock: [handlers.send_out_of_stock_notification],
}  # type: Dict[Type[events.Event], List[Callable]]


COMMAND_HANDLERS = {
    commands.Allocate: handlers.allocate,
    commands.CreateBatch: handlers.add_batch,
    commands.ChangeBatchQuantity: handlers.change_batch_quantity,
}  # type: Dict[Type[commands.Command], Callable]
```

The message bus should be adpated with the commands.

We should keep the important function `handle`. Now we use the name `message` instead of `event`.

```
Message = Union[commands.Command, events.Event]


def handle(  #(1)
    message: Message,
    uow: unit_of_work.AbstractUnitOfWork,
):
    results = []
    queue = [message]
    while queue:
        message = queue.pop(0)
        if isinstance(message, events.Event):
            handle_event(message, queue, uow)  #(2)
        elif isinstance(message, commands.Command):
            cmd_result = handle_command(message, queue, uow)  #(2)
```

```
            results.append(cmd_result)
        else:
            raise Exception(f"{message} was not an Event or Command")
    return results
```


Test-driven Development

It forces writing tests before writing code.

It helps understand requirements better.

A good reading: Yamaura, Tsuneo. "How to Design Practical Test Cases." IEEE Software (November/December 1998): 30-36.

It ensures that code continues working after we make changes.

Note that changes to one part of the code can inadvertently break other parts.