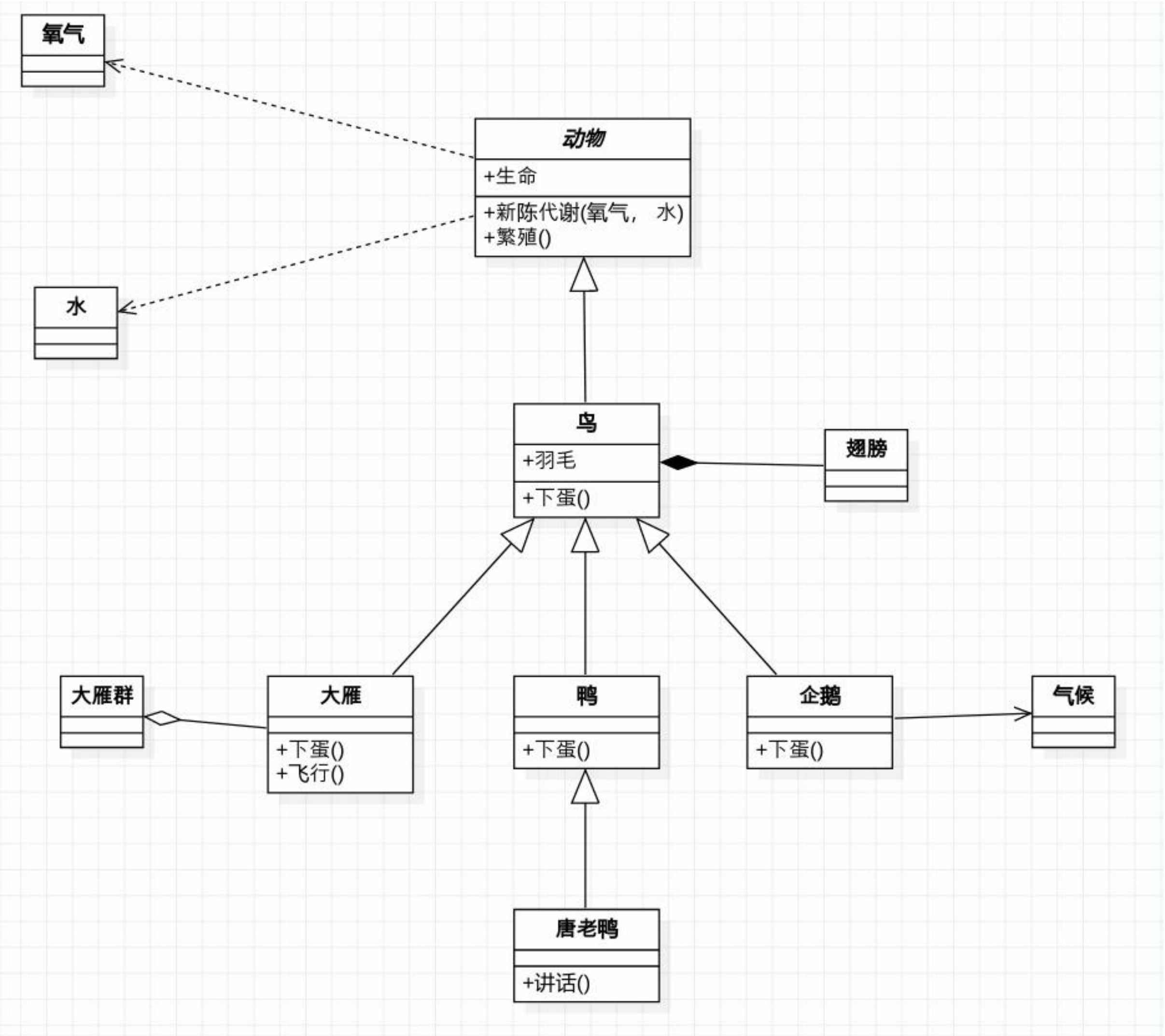
Onion model in Python

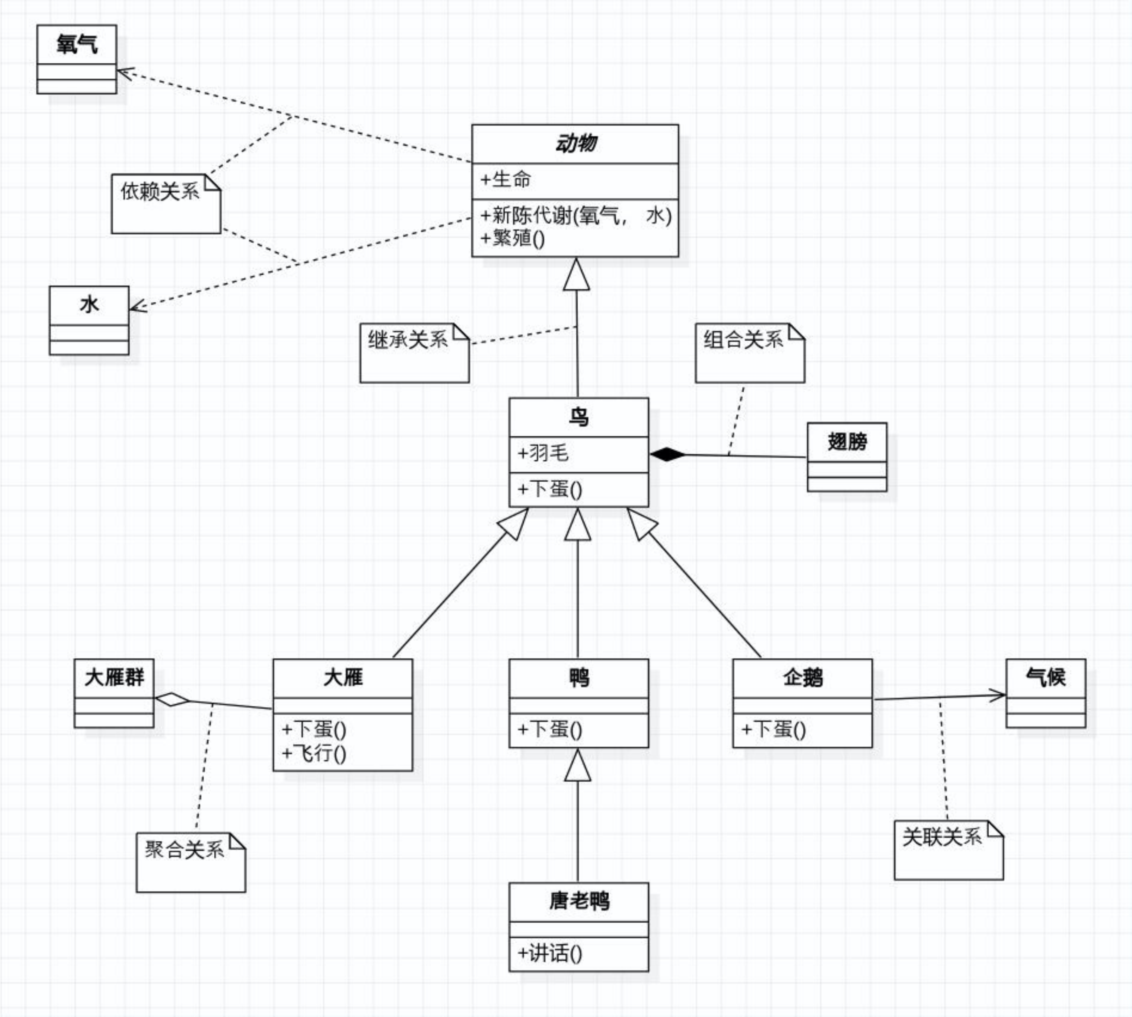
Suiyun Yang

2020.12.3

Pre-learning



Class graph


氧气
动物
+生命
+新陈代谢(氧气， 水)
+繁殖()
依赖关系
水
继承关系
组合关系
鸟
+羽毛
+下蛋()
翅膀
大雁群
大雁
+下蛋()
+飞行()
鸭
+下蛋()
企鹅
+下蛋()
气候
聚合关系
关联关系
唐老鸭
+讲话()

简单案例

- 用户可以通过银行网页转账给另一个账号，支持跨币种转账。
- 同时因为监管和对账需求，需要记录本次转账活动。

https://mp.weixin.qq.com/s/MU1rqpQ1aA1p7OtXqVVwxQ

```
public class TransferController {
   private TransferService transferService;

   public Result<Boolean> transfer(String targetAccountNumber, BigDecimal amount, HttpSession session) {
      Long userId = (Long) session.getAttribute("userId");
      return transferService.transfer(userId, targetAccountNumber, amount, "CNY");
   }
}

public class TransferServiceImpl implements TransferService {
   private static final String TOPIC_AUDIT_LOG = "TOPIC_AUDIT_LOG";
   private AccountMapper accountDAO;
   private KafkaTemplate<String, String> kafkaTemplate;
   private YahooForexService yahooForex;

   @Override
   public Result<Boolean> transfer(Long sourceUserId, String targetAccountNumber, BigDecimal targetAmount, String targetCurrency) {
      // 1. 从数据库读取数据，忽略所有校验逻辑如账号是否存在等
      AccountDO sourceAccountDO = accountDAO.selectByUserId(sourceUserId);
      AccountDO targetAccountDO = accountDAO.selectByAccountNumber(targetAccountNumber);

      // 2. 业务参数校验
      if (!targetAccountDO.getCurrency().equals(targetCurrency)) {
         throw new InvalidCurrencyException();
      }

      // 3. 获取外部数据，并且包含一定的业务逻辑
      // exchange rate = 1 source currency = X target currency
      BigDecimal exchangeRate = BigDecimal.ONE;
      if (sourceAccountDO.getCurrency().equals(targetCurrency)) {
         exchangeRate = yahooForex.getExchangeRate(sourceAccountDO.getCurrency(), targetCurrency);
      }
      BigDecimal sourceAmount = targetAmount.divide(exchangeRate, RoundingMode.DOWN);

      // 4. 业务参数校验
      if (sourceAccountDO.getAvailable().compareTo(sourceAmount) < 0) {
         throw new InsufficientFundsException();
      }
      if (sourceAccountDO.getDailyLimit().compareTo(sourceAmount) < 0) {
         throw new DailyLimitExceededException();
      }

      // 5. 计算新值，并且更新字段
      BigDecimal newSource = sourceAccountDO.getAvailable().subtract(sourceAmount);
      BigDecimal newTarget = targetAccountDO.getAvailable().add(targetAmount);
      sourceAccountDO.setAvailable(newSource);
      targetAccountDO.setAvailable(newTarget);

      // 6. 更新到数据库
      accountDAO.update(sourceAccountDO);
      accountDAO.update(targetAccountDO);

      // 7. 发送审计消息
      String message = sourceUserId + "," + targetAccountNumber + "," + targetAmount + "," + targetCurrency;
      kafkaTemplate.send(TOPIC_AUDIT_LOG, message);

      return Result.success(true);
   }
}
```

1、从MySql数据库中找到转出和转入的账户，选择用 MyBatis 的 mapper 实现 DAO；

2、从 Yahoo（或其他渠道）提供的汇率服务获取转账的汇率信息（底层是 http 开放接口）；

3、计算需要转出的金额，确保账户有足够余额，并且没超出每日转账上限；

4、实现转入和转出操作，扣除手续费，保存数据库；

5、发送 Kafka 审计消息，以便审计和对账用；

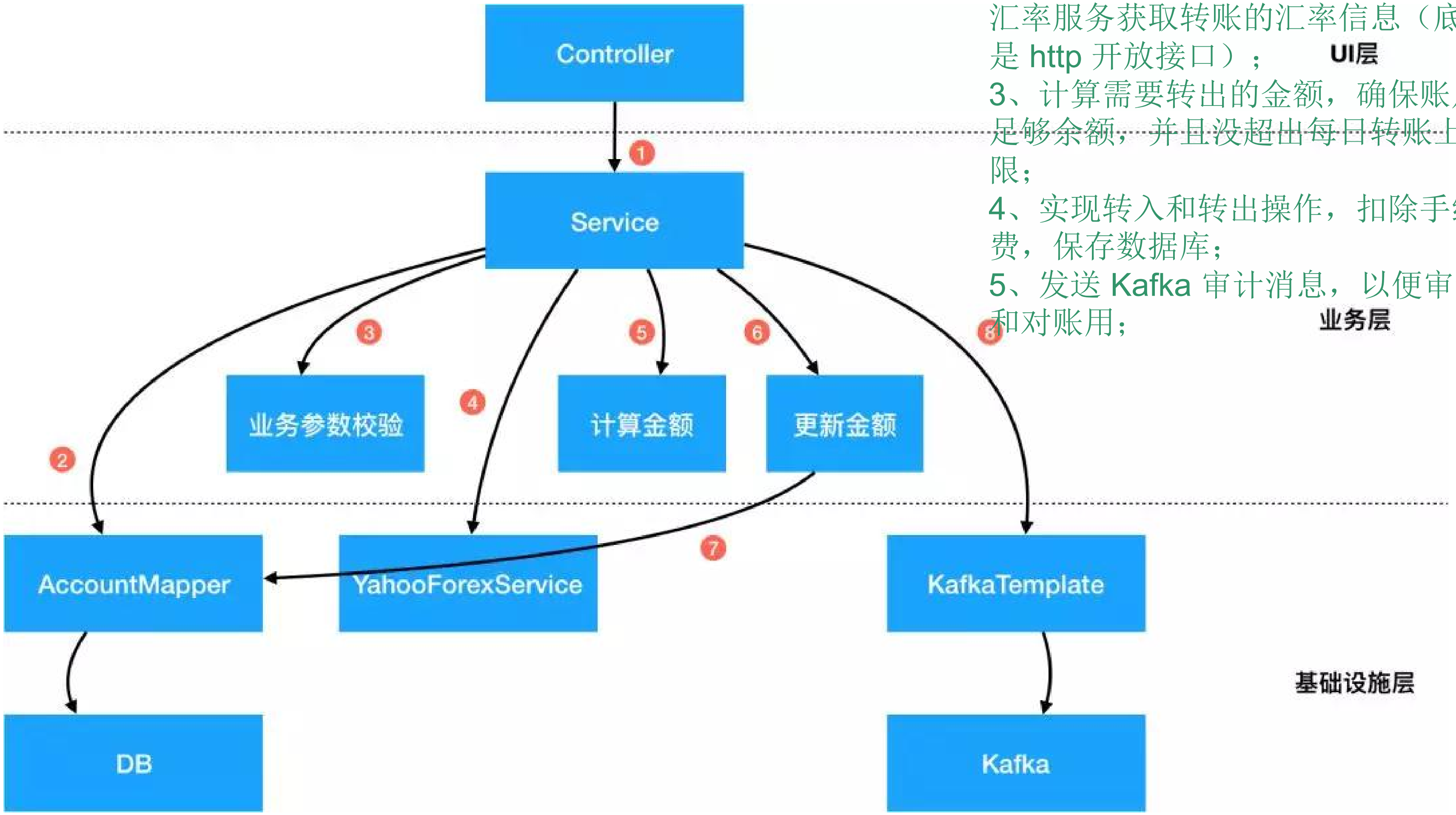
1、从MySql数据库中找到转出和转入的账户，选择用 MyBatis 的 mapper 实现 DAO；

2、从 Yahoo（或其他渠道）提供的汇率服务获取转账的汇率信息（底层是 http 开放接口）；

3、计算需要转出的金额，确保账户有足够余额，并且没超出每日转账上限；

4、实现转入和转出操作，扣除手续费，保存数据库；

5、发送 Kafka 审计消息，以便审计和对账用；



问题1-可维护性能差

可维护性 = 当依赖变化时，有多少代码需要随之改变

参考以上的案例代码，事务脚本类的代码很难维护因为以下几点：

- 数据结构的不稳定性：AccountDO类是一个纯数据结构，映射了数据库中的一个表。这里的问题是数据库的表结构和设计是应用的外部依赖，长远来看都有可能会改变，比如数据库要做Sharding，或者换一个表设计，或者改变字段名。
- 依赖库的升级：AccountMapper依赖MyBatis的实现，如果MyBatis未来升级版本，可能会造成用法的不同（可以参考iBatis升级到基于注解的MyBatis的迁移成本）。同样的，如果未来换一个ORM体系，迁移成本也是巨大的。
- 第三方服务依赖的不确定性：第三方服务，比如Yahoo的汇率服务未来很有可能会有变化：轻则API签名变化，重则服务不可用需要寻找其他可替代的服务。在这些情况下改造和迁移成本都是巨大的。同时，外部依赖的兜底、限流、熔断等方案都需要随之改变。
- 第三方服务API的接口变化：YahooForexService.getExchangeRate返回的结果是小数点还是百分比？入参是（source, target）还是（target, source）？谁能保证未来接口不会改变？如果改变了，核心的金额计算逻辑必须跟着改，否则会造成资损。
- 中间件更换：今天我们用Kafka发消息，明天如果要上阿里云用RocketMQ该怎么办？后天如果消息的序列化方式从String改为Binary该怎么办？如果需要消息分片该怎么改？

问题2-可拓展性差

可扩展性 = 做新需求或改逻辑时，需要新增/修改多少代码

参考以上的代码，如果今天需要增加一个跨行转账的能力，你会发现基本上需要重新开发，基本上没有任何的可复用性：

- 数据来源被固定、数据格式不兼容：原有的AccountDO是从本地获取的，而跨行转账的数据可能需要从一个第三方服务获取，而服务之间数据格式不太可能是兼容的，导致从数据校验、数据读写、到异常处理、金额计算等逻辑都要重写。
- 业务逻辑无法复用：数据格式不兼容的问题会导致核心业务逻辑无法复用。每个用例都是特殊逻辑的后果是最终会造成大量的if-else语句，而这种分支多的逻辑会让分析代码非常困难，容易错过边界情况，造成bug。
- 逻辑和数据存储的相互依赖：当业务逻辑增加变得越来越复杂时，新加入的逻辑很有可能需要对数据库schema或消息格式做变更。而变更了数据格式后会导致原有的其他逻辑需要一起跟着动。在最极端的场景下，一个新功能的增加会导致所有原有功能的重构，成本巨大。

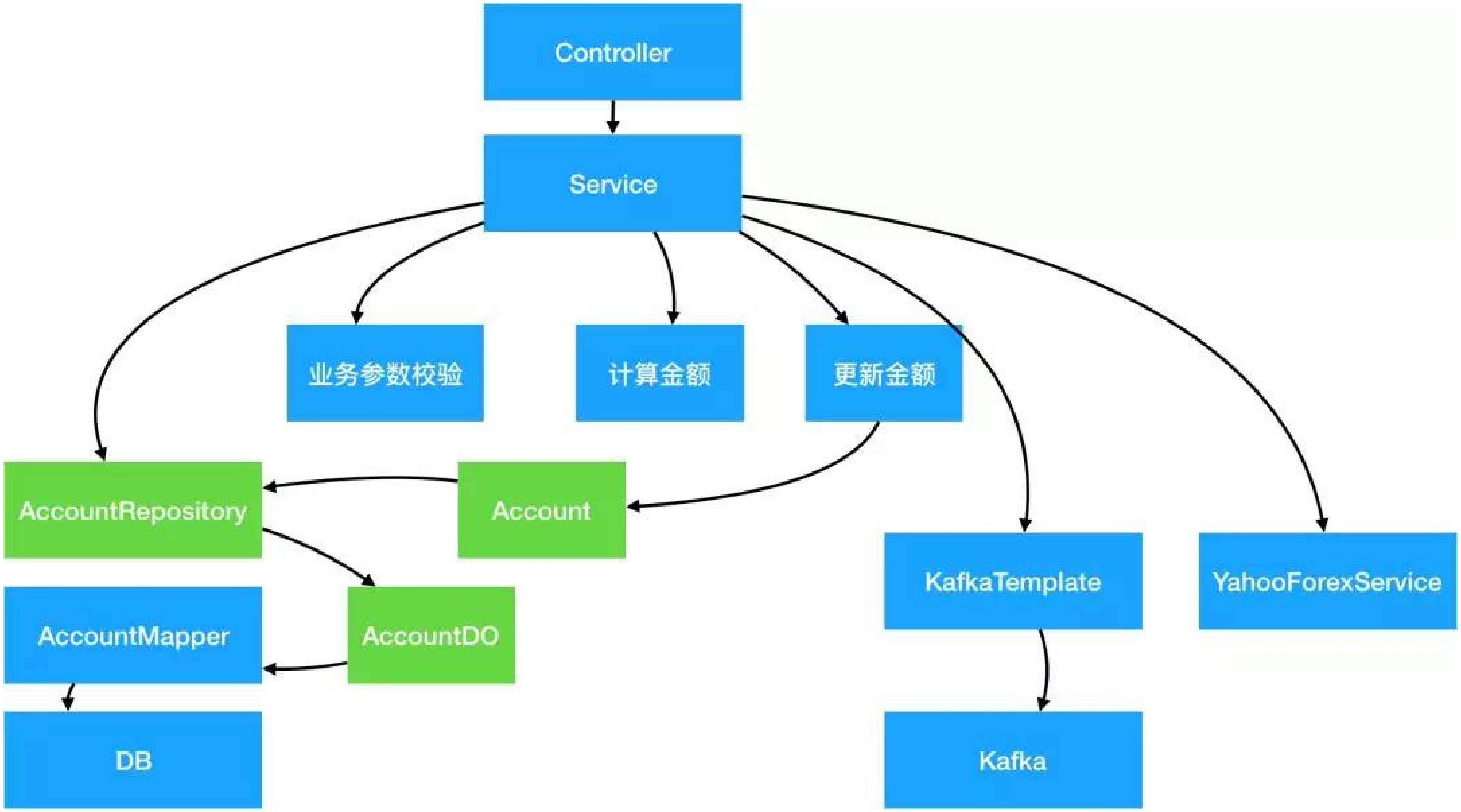
问题3-可测试性能差

可测试性 = 运行每个测试用例所花费的时间 * 每个需求所需要增加的测试用例数量

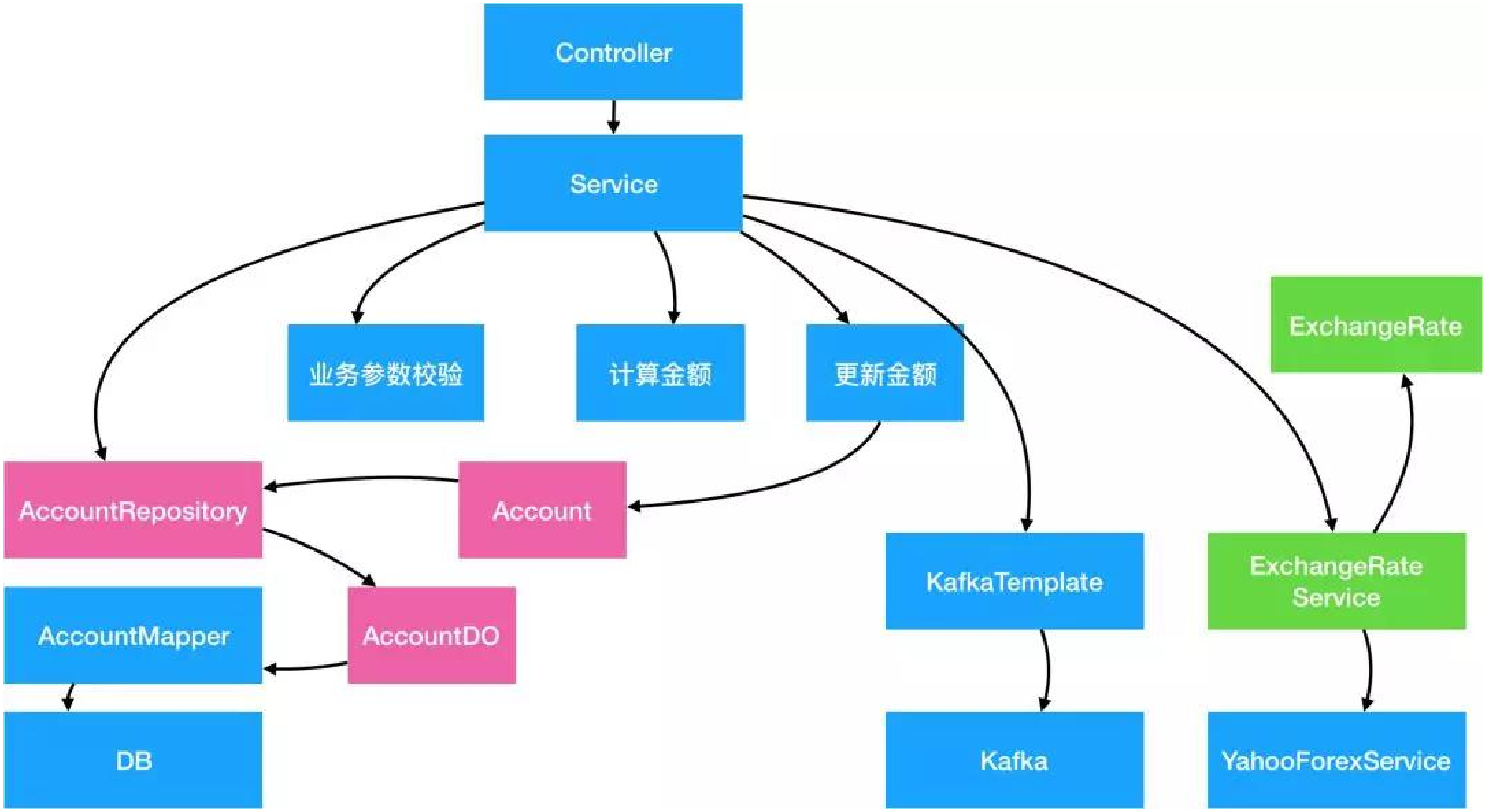
参考以上的一段代码，这种代码有极低的可测试性：

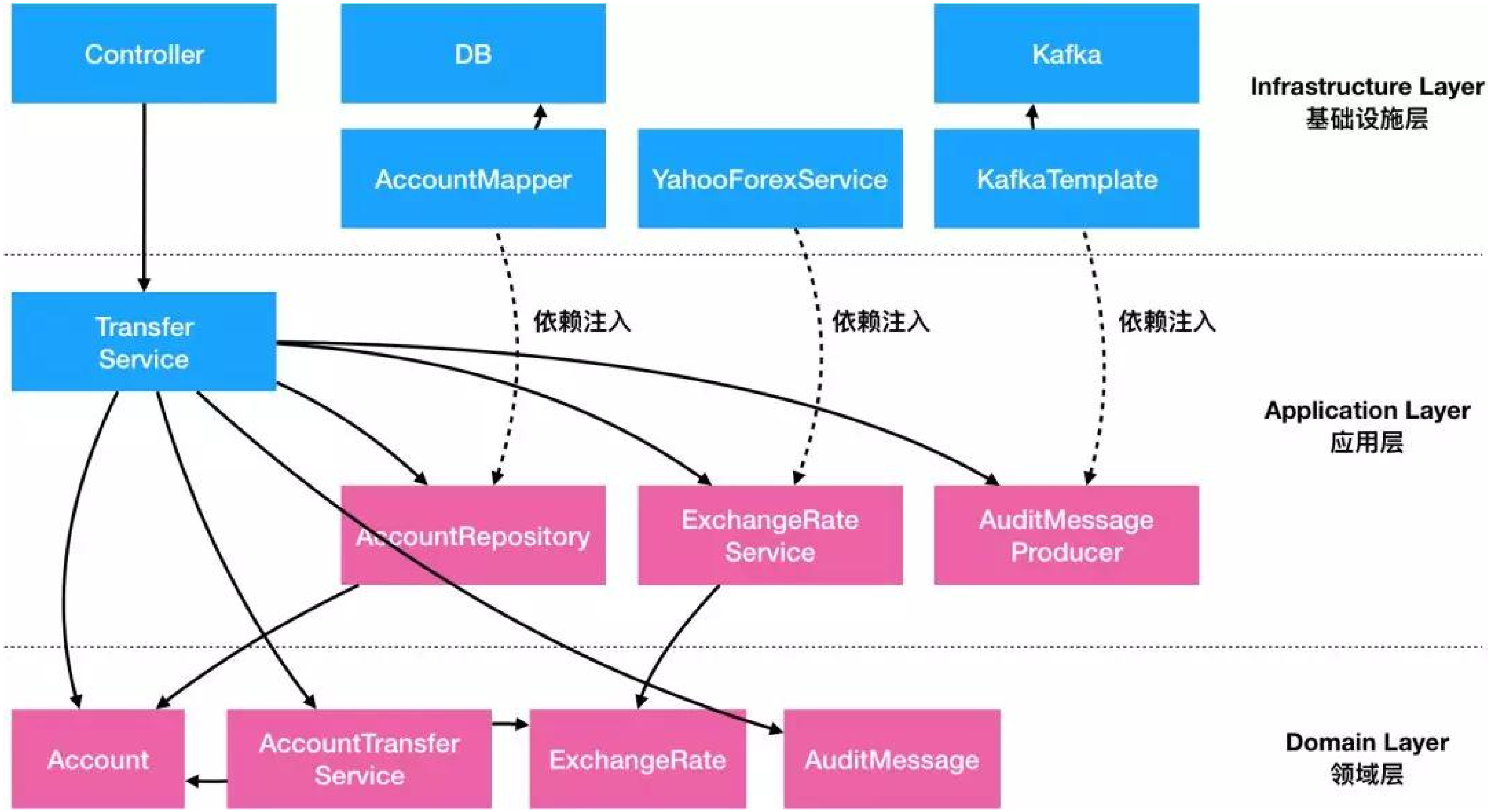
- 设施搭建困难：当代码中强依赖了数据库、第三方服务、中间件等外部依赖之后，想要完整跑通一个测试用例需要确保所有依赖都能跑起来，这个在项目早期是及其困难的。在项目后期也会由于各种系统的不稳定性而导致测试无法通过。
- 运行耗时长：大多数的外部依赖调用都是I/O密集型，如跨网络调用、磁盘调用等，而这种I/O调用在测试时需要耗时很久。另一个经常依赖的是笨重的框架如Spring，启动Spring容器通常需要很久。当一个测试用例需要花超过10秒钟才能跑通时，绝大部分开发都不会很频繁的测试。
- 耦合度高：假如一段脚本中有A、B、C三个子步骤，而每个步骤有N个可能的状态，当多个子步骤耦合度高时，为了完整覆盖所有用例，最多需要有N * N * N个测试用例。当耦合的子步骤越多时，需要的测试用例呈指数级增长。

抽象数据存储层

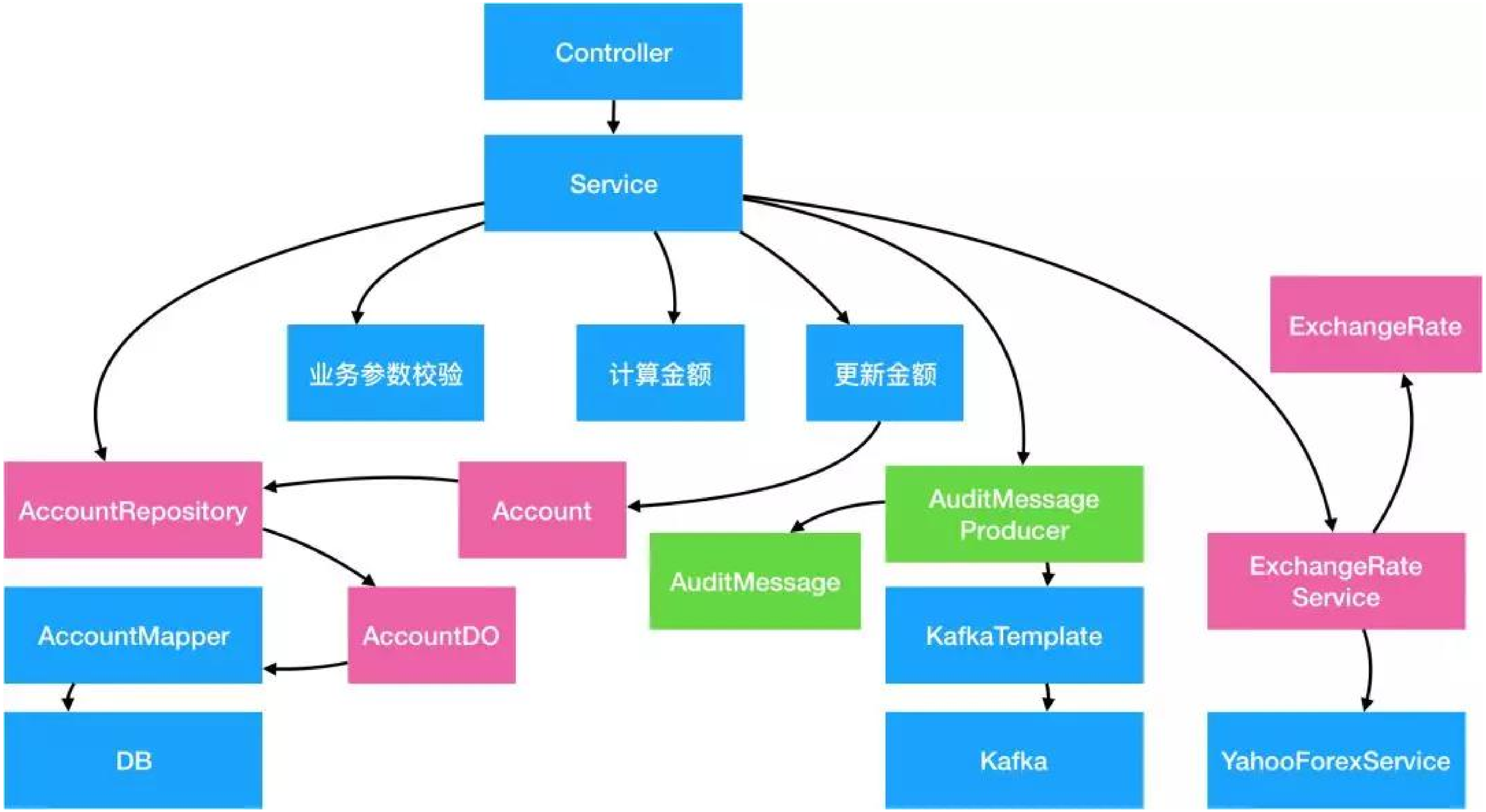

Controller
Service
业务参数校验
计算金额
更新金额
AccountRepository
Account
AccountDO
AccountMapper
DB
KafkaTemplate
Kafka
YahooForexService

抽象第三方服务

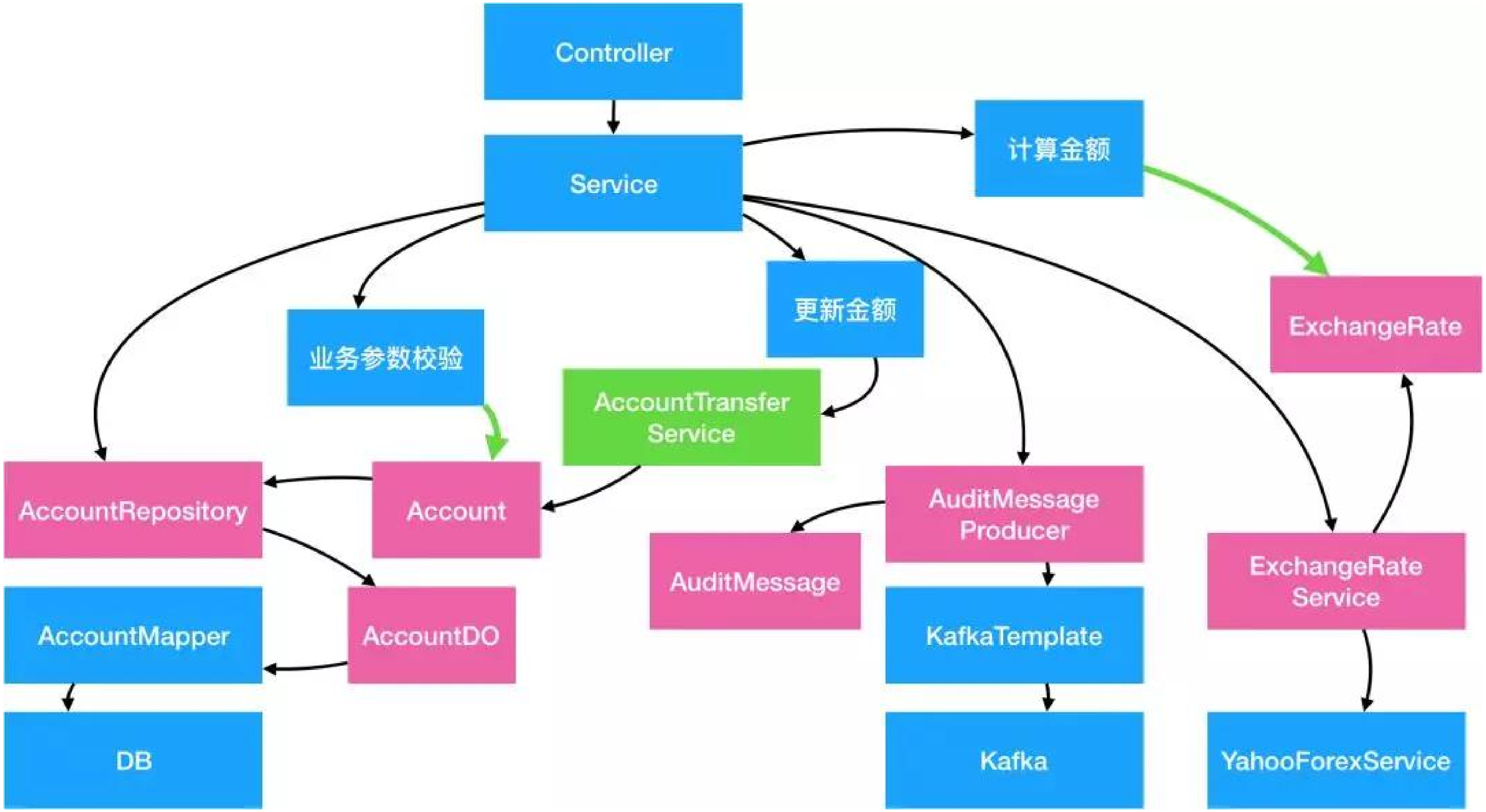

Controller
Service
业务参数校验
计算金额
更新金额
ExchangeRate
AccountRepository
Account
KafkaTemplate
ExchangeRate Service
AccountMapper
AccountDO
DB
Kafka
YahooForexService


Controller
DB
Kafka
Infrastructure Layer
基础设施层
AccountMapper
YahooForexService
KafkaTemplate
依赖注入
依赖注入
依赖注入
Transfer Service
Application Layer
应用层
AccountRepository
ExchangeRate Service
AuditMessage Producer
Account
AccountTransfer Service
ExchangeRate
AuditMessage
Domain Layer
领域层

抽象中间件

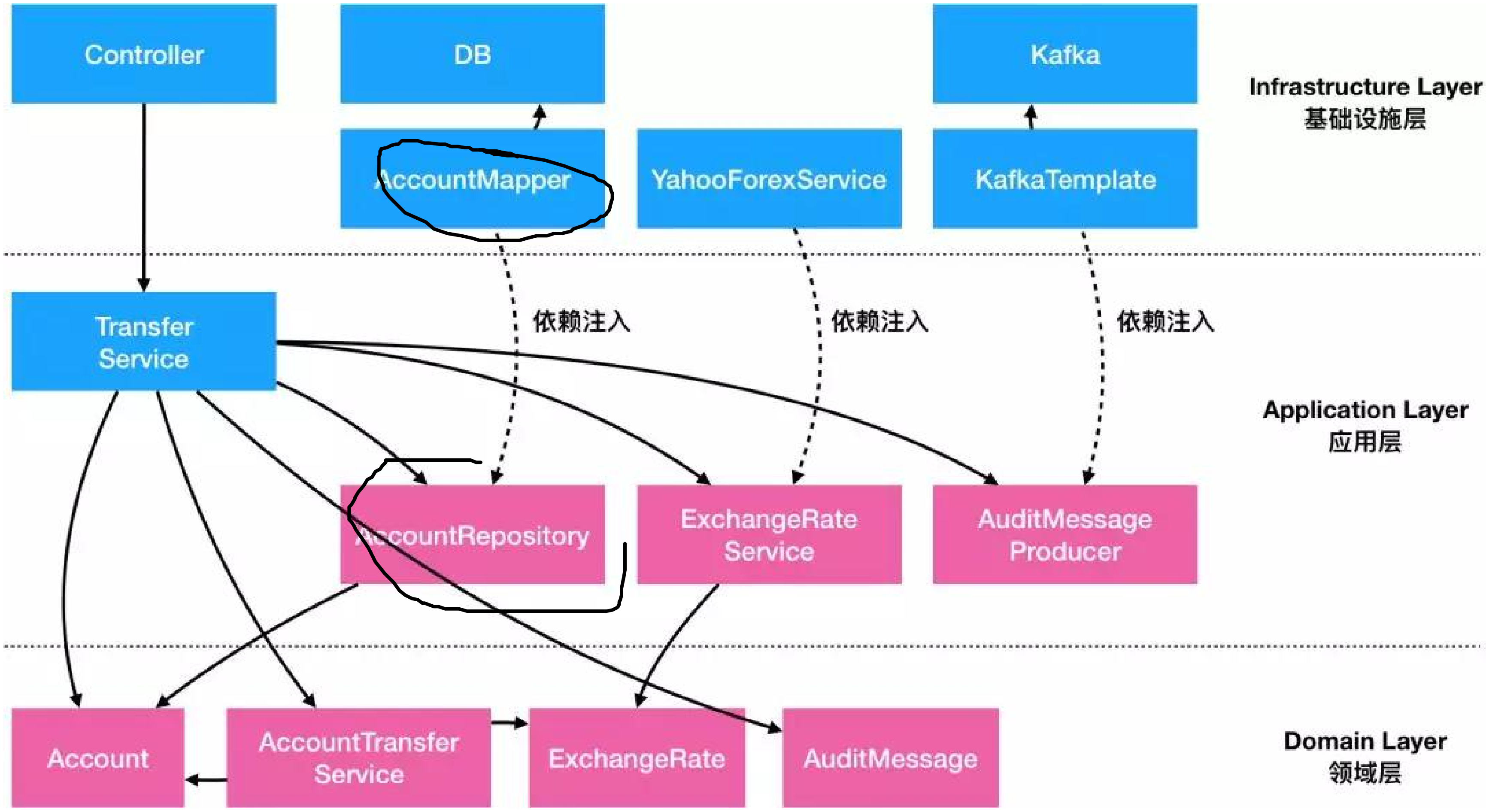

Controller
Service
业务参数校验
计算金额
更新金额
ExchangeRate
AccountRepository
Account
AuditMessage Producer
AuditMessage
ExchangeRate Service
AccountMapper
AccountDO
KafkaTemplate
DB
Kafka
YahooForexService

封装业务逻辑


Controller
Service
计算金额
ExchangeRate
业务参数校验
更新金额
AccountTransfer Service
AccountRepository
Account
AuditMessage Producer
AuditMessage
ExchangeRate Service
AccountMapper
AccountDO
KafkaTemplate
DB
Kafka
YahooForexService

重构后

```
public class TransferServiceImplNew implements TransferService {

    private AccountRepository accountRepository;
    private AuditMessageProducer auditMessageProducer;
    private ExchangeRateService exchangeRateService;
    private AccountTransferService accountTransferService;

    @Override
    public Result<Boolean> transfer(Long sourceUserId, String targetAccountNumber, BigDecimal targetAmount, String
targetCurrency) {
        // 参数校验
        Money targetMoney = new Money(targetAmount, new Currency(targetCurrency));

        // 读数据
        Account sourceAccount = accountRepository.find(new UserId(sourceUserId));
        Account targetAccount = accountRepository.find(new AccountNumber(targetAccountNumber));
        ExchangeRate exchangeRate = exchangeRateService.getExchangeRate(sourceAccount.getCurrency(),
targetMoney.getCurrency());

        // 业务逻辑
        accountTransferService.transfer(sourceAccount, targetAccount, targetMoney, exchangeRate);

        // 保存数据
        accountRepository.save(sourceAccount);
        accountRepository.save(targetAccount);

        // 发送审计消息
        AuditMessage message = new AuditMessage(sourceAccount, targetAccount, targetMoney);
        auditMessageProducer.send(message);

        return Result.success(true);
    }
}
```

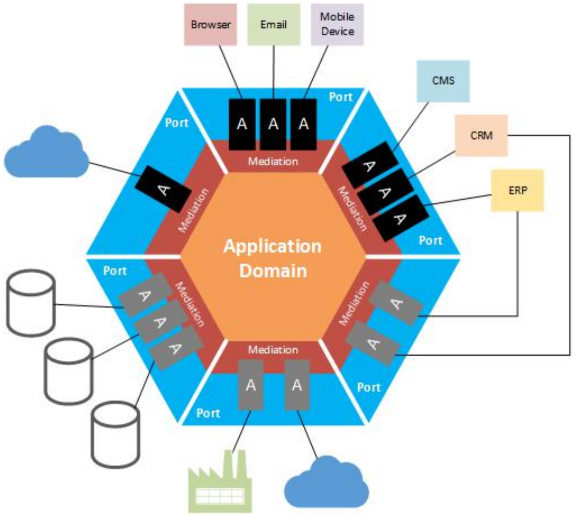


- 业务逻辑清晰，数据存储和业务逻辑完全分隔。
- Entity、Domain Primitive、Domain Service都是独立的对象，没有任何外部依赖，但是却包含了所有核心业务逻辑，可以单独完整测试。
- 原有的TransferService不再包括任何计算逻辑，仅仅作为组件编排，所有逻辑均delegate到其他组件。这种仅包含Orchestration（编排）的服务叫做Application Service（应用服务）。

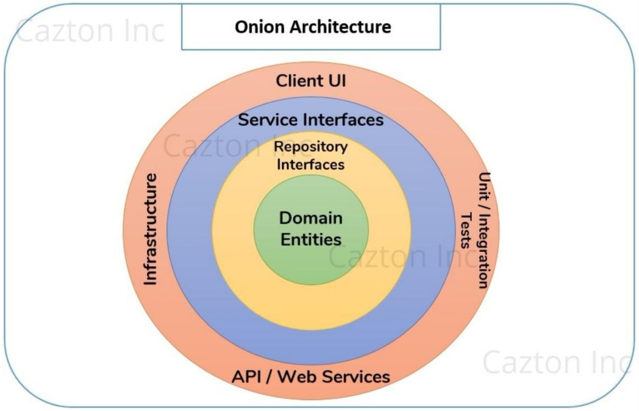
What is good architecture

- 独立于框架：架构不应该依赖某个外部的库或框架，不应该被框架的结构所束缚。
- 独立于UI：前台展示的样式可能会随时发生变化（今天可能是网页、明天可能变成console、后天是独立app），但是底层架构不应该随之而变化。
- 独立于底层数据源：无论今天你用MySQL、Oracle还是MongoDB、CouchDB，甚至使用文件系统，软件架构不应该因为不同的底层数据储存方式而产生巨大改变。
- 独立于外部依赖：无论外部依赖如何变更、升级，业务的核心逻辑不应该随之而大幅变化。
- 可测试：无论外部依赖了什么数据库、硬件、UI或者服务，业务的逻辑应该都能够快速被验证正确性。

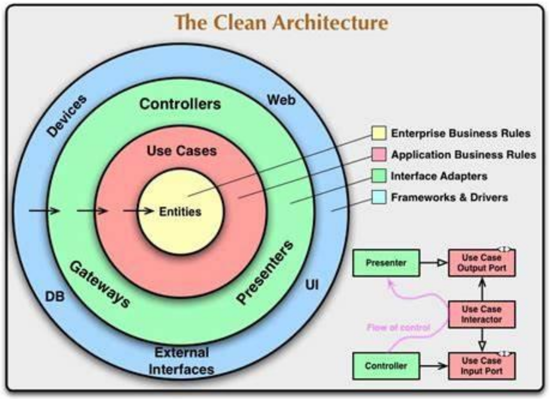
Architecture pattern



2005 Hex

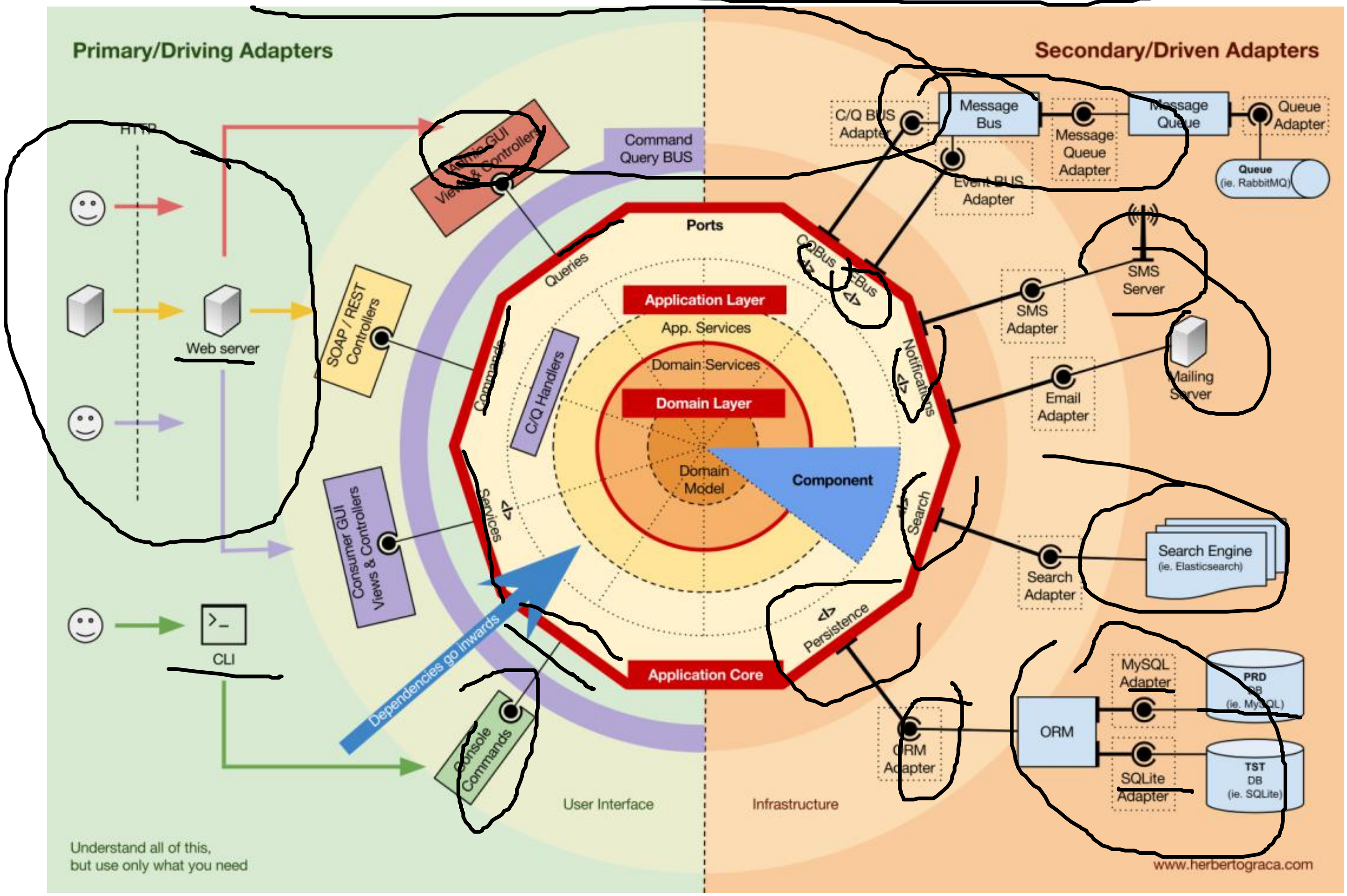


2008年 Jeffery Palermo的Onion Architecture



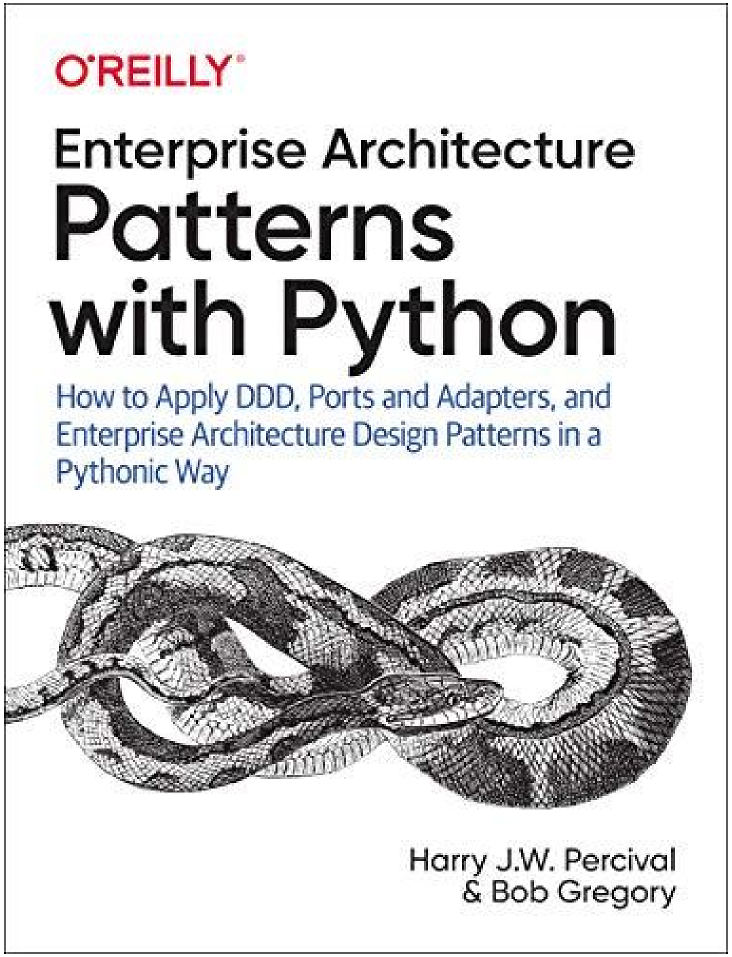
2017年 Robert Martin 的Clean Architecture

Explicit Architecture



https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/

Architecture Patterns with Python



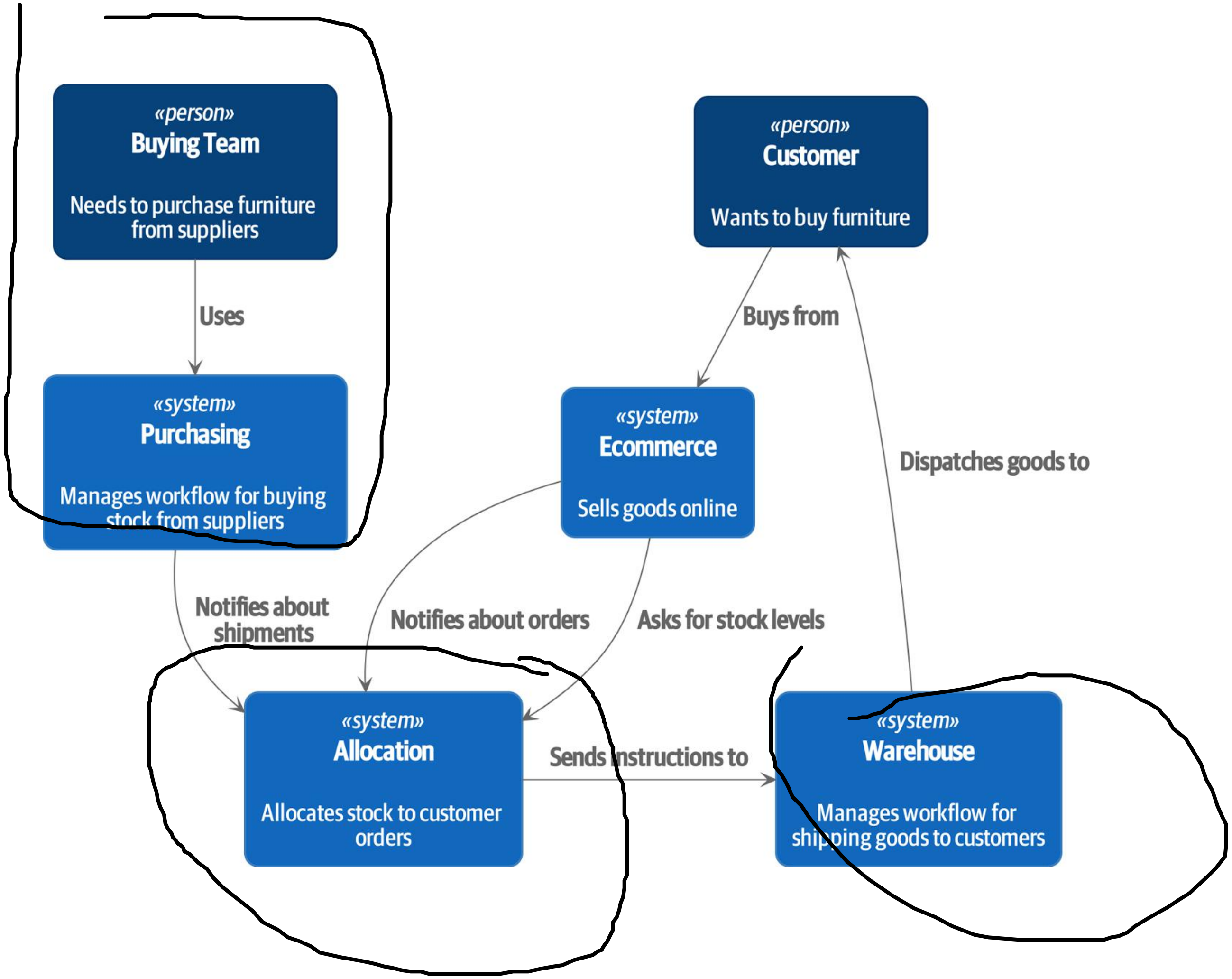
O'REILLY

Enterprise Architecture
Patterns with Python

How to Apply DDD, Ports and Adapters, and Enterprise Architecture Design Patterns in a Pythonic Way

Harry J.W. Percival
& Bob Gregory

Context for allocation

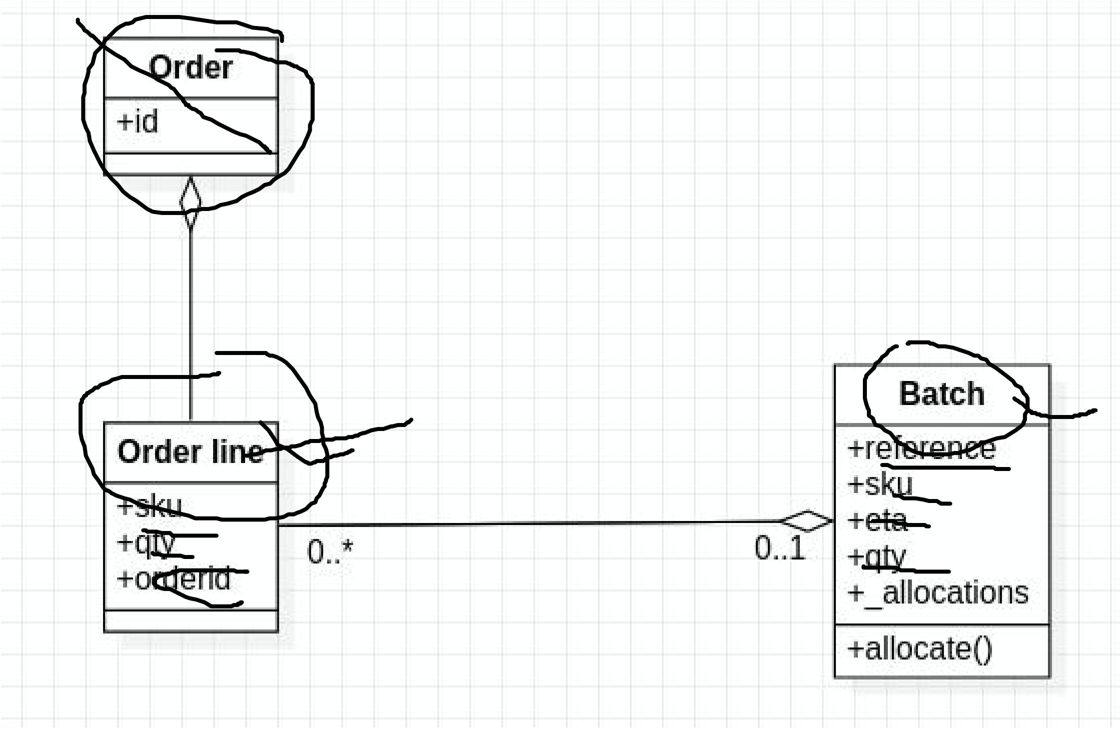


About allocation

- Customers place orders. An order is identified by an order reference and comprises multiple order lines, where each line has a SKU and a quantity. For example:
  - ➢ 10 units of RED-CHAIR
- The purchasing department orders small batches of stock. A batch of stock has a unique ID called a reference, a SKU, and a quantity.
- **We need to allocate order lines to batches**. When we've allocated an order line to a batch, we will send stock from that specific batch to the customer's delivery address. 1 When we allocate x units of stock to a batch, the available quantity is reduced by x.
  - ➢ 2 We can't allocate to a batch if the available quantity is less than the quantity of the order line.
  - ➢ 3 We can't allocate the same line twice.
- Batches have an ETA(Estimated Time of Arrival) if they are currently shipping, or they may be in warehouse stock. We allocate to warehouse stock in preference to shipment batches. 4 We allocate to shipment batches in order of which has the earliest ETA. ETA=None when it is in stock. Or, it is shipping.

Sprint 1-1

- **We need to allocate order lines to batches**. When we've allocated an order line to a batch, we will send stock from that specific batch to the customer's delivery address. When we allocate x units of stock to a batch, the available quantity is reduced by x.
  - ➢ We can't allocate to a batch if the available quantity is less than the quantity of the order line.
  - ➢ We can't allocate the same line twice.

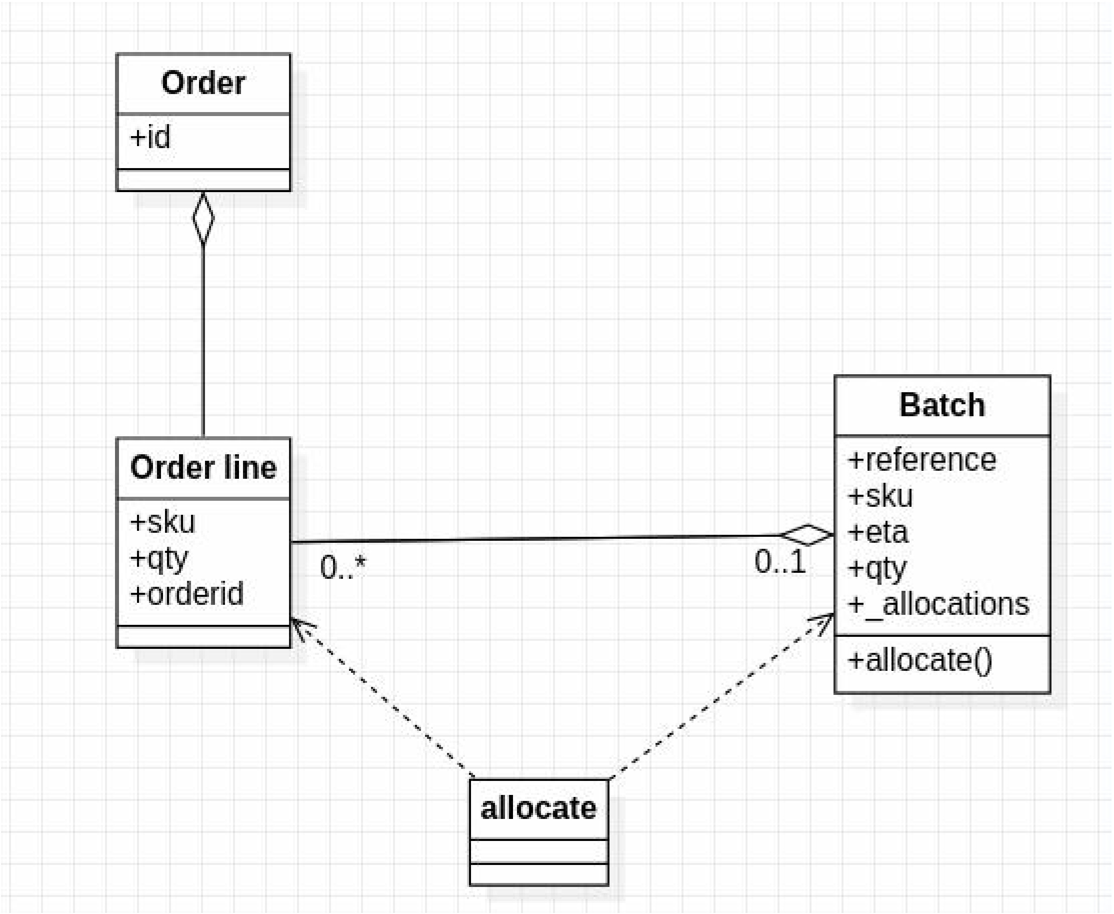

Order
+id
Order line
+sku
+qty
+orderid
0..*
0..1
Batch
+reference
+sku
+eta
+qty
+_allocations
+allocate()

Value object, Entity

- Value object: any object that is identified only by its data and doesn't have a long-lived identity.
  - ✓ For value objects, the hash should be based on all the value attributes, and we should ensure that the objects are immutable.
  - ✓ dataclass is sutiable model for it, and @frozen=True
- Entity: describe a domain object that has long-lived identity.
  - ✓ Entities, unlike values, have identity equality. We can change their values, and they are still recognizably the same thing.
  - ✓ If for some reason you decide you really do want to use set or dict operations with entities, the hash should be based on the attribute(s), such as .reference, that defines the entity's unique identity over time. You should also try to somehow make that attribute read-only.

Sprint1-2

- Batches have an ETA(Estimated Time of Arrival) if they are currently shipping, or they may be in warehouse stock. We allocate to warehouse stock in preference to shipment batches. We allocate to shipment batches in order of which has the earliest ETA. ETA=None when it is in stock. Or, it is shipping.
  - ✓ prefer current stock than shiping
  - ✓ prefer earlier


Order
+id
Order line
+sku
+qty
+orderid
0..*
0..1
Batch
+reference
+sku
+eta
+qty
+_allocations
+allocate()
allocate

Domain Service Function

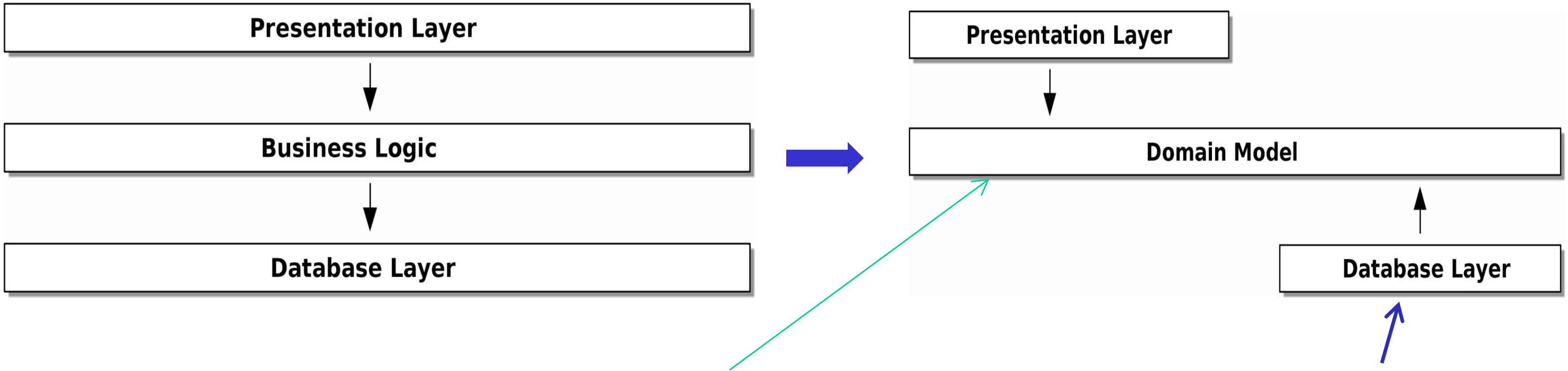
Domain Service operations that don’t have a natural home in an entity or value object

Sprint2

- Persist our domain model

Sprint2-1

- We don’t want infrastructure concerns bleeding over into our domain model and slowing our unit tests or our ability to make changes.
  - ➢ Build orm model
  - ➢ map orm model with domain model



Presentation Layer

Business Logic

Database Layer

Presentation Layer

Domain Model

Database Layer

Only deal with business logic

Deal with storage, offer different storage solution for domain model

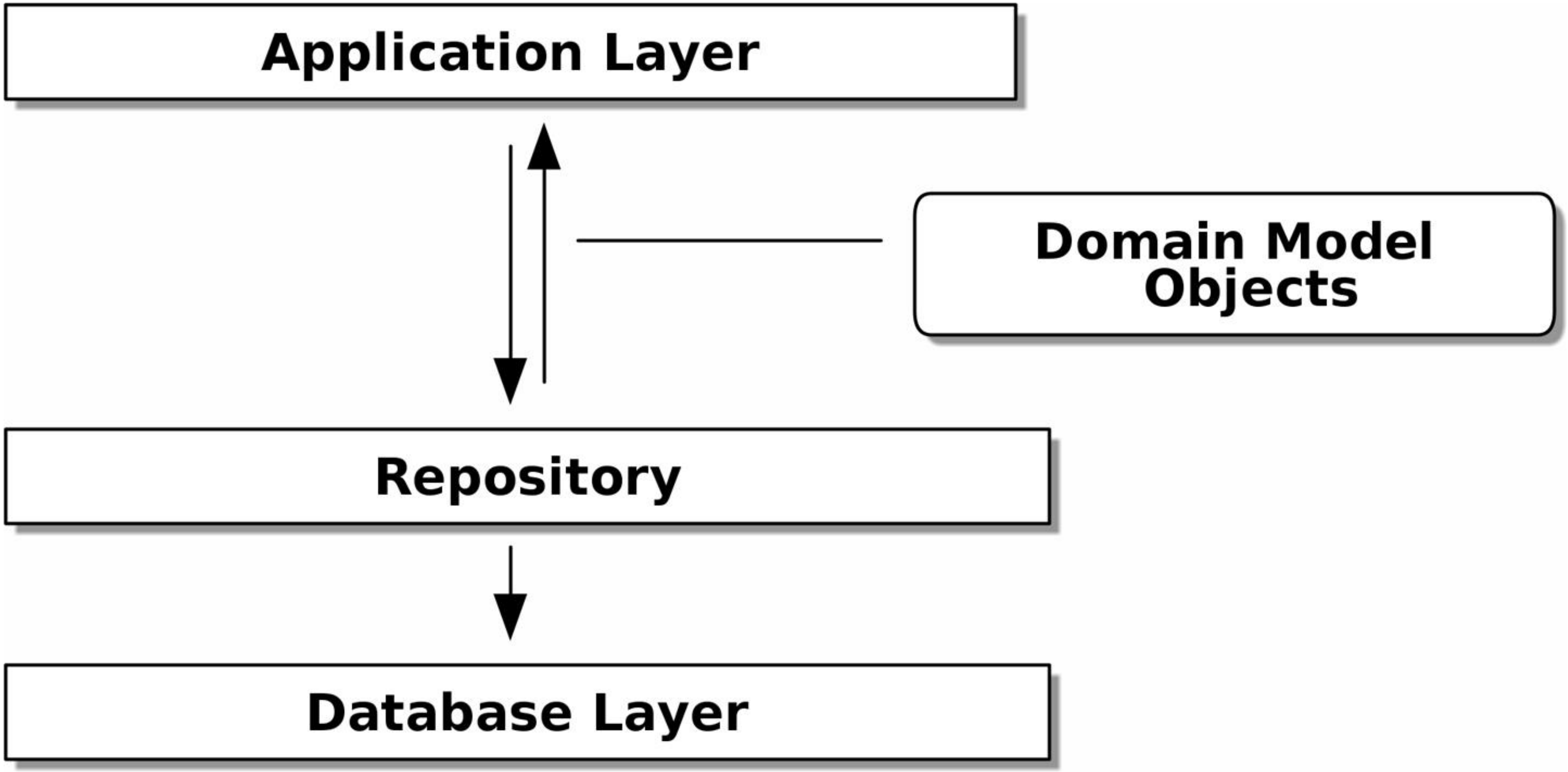
Sprit2-2

- Hide details of data access

Usage of sprint2-1 as follows:

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

    # call our domain service
    allocate(line, batches)

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

    return 201
```

Process data model directly, and application layer depend on orm



Application Layer

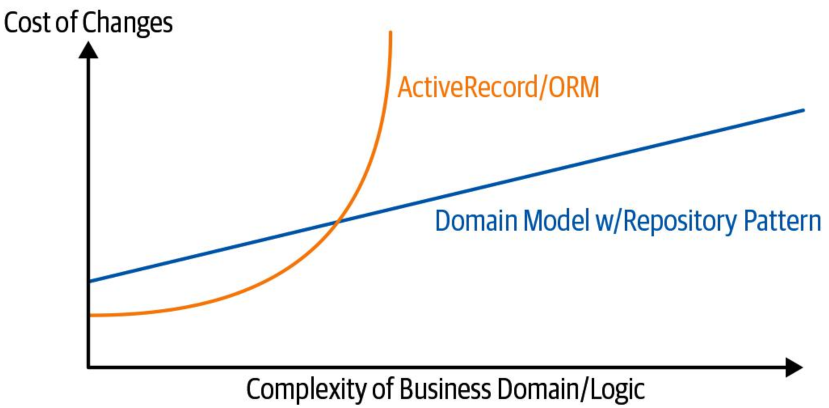
Domain Model Objects

Repository

Database Layer

Pros and Cons

Pros	Cons
We have a simple interface between persistent storage and our domain model.	An ORM already buys you some decoupling. Changing foreign keys might be hard, but it should be pretty easy to swap between MySQL and Postgres if you ever need to.
It’s easy to make a fake version of the repository for unit testing, or to swap out different storage solutions, because we’ve fully decoupled the model from infrastructure concerns.	Maintaining ORM mappings by hand requires extra work and extra code.
Writing the domain model before thinking about persistence helps us focus on the business problem at hand. If we ever want to radically change our approach, we can do that in our model, without needing to worry about foreign keys or migrations until later.	Any extra layer of indirection always increases maintenance costs and adds a "WTF factor" for Python programmers who’ve never seen the Repository pattern before.
Our database schema is really simple because we have complete control over how we map our objects to tables.	



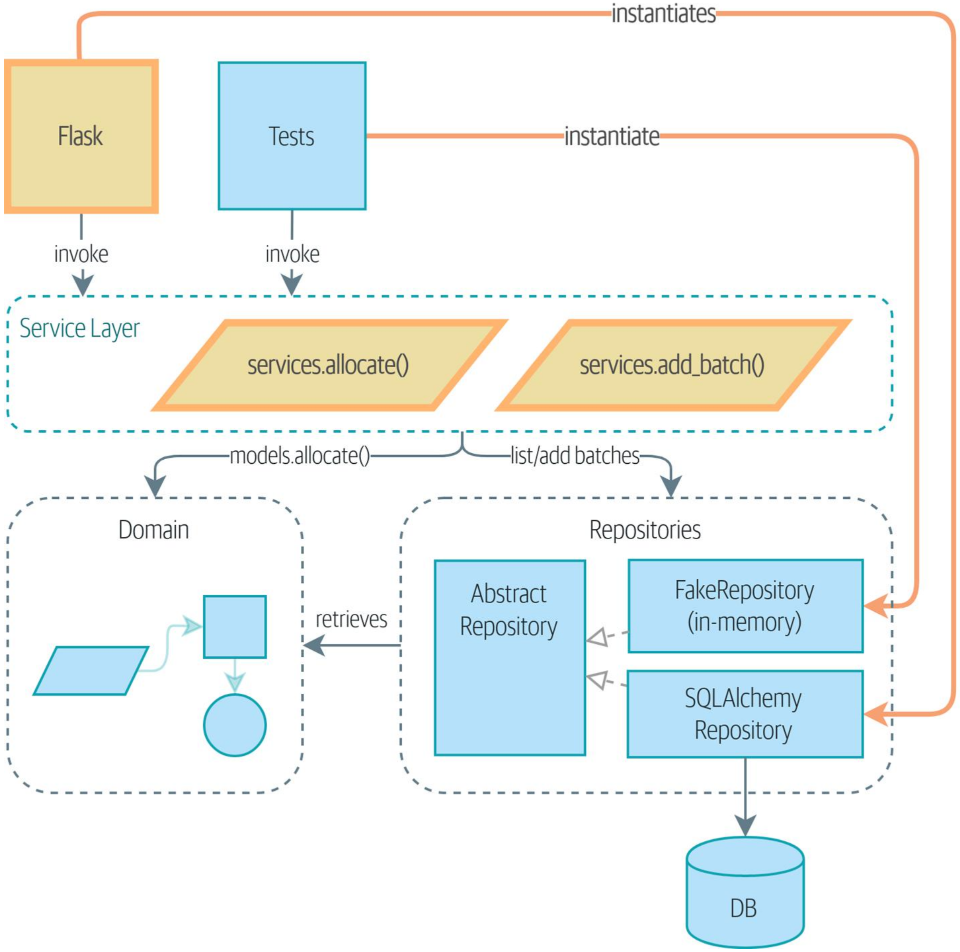
Sprint3-1

- Use Flask to put an API endpoint in front of our allocate domain service.

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

@app.route("/allocate", methods=['POST'])
def allocate_endpoint():
    session = get_session()
    batches = repository.SqlAlchemyRepository(session).list()
    line = model.OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

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

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

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

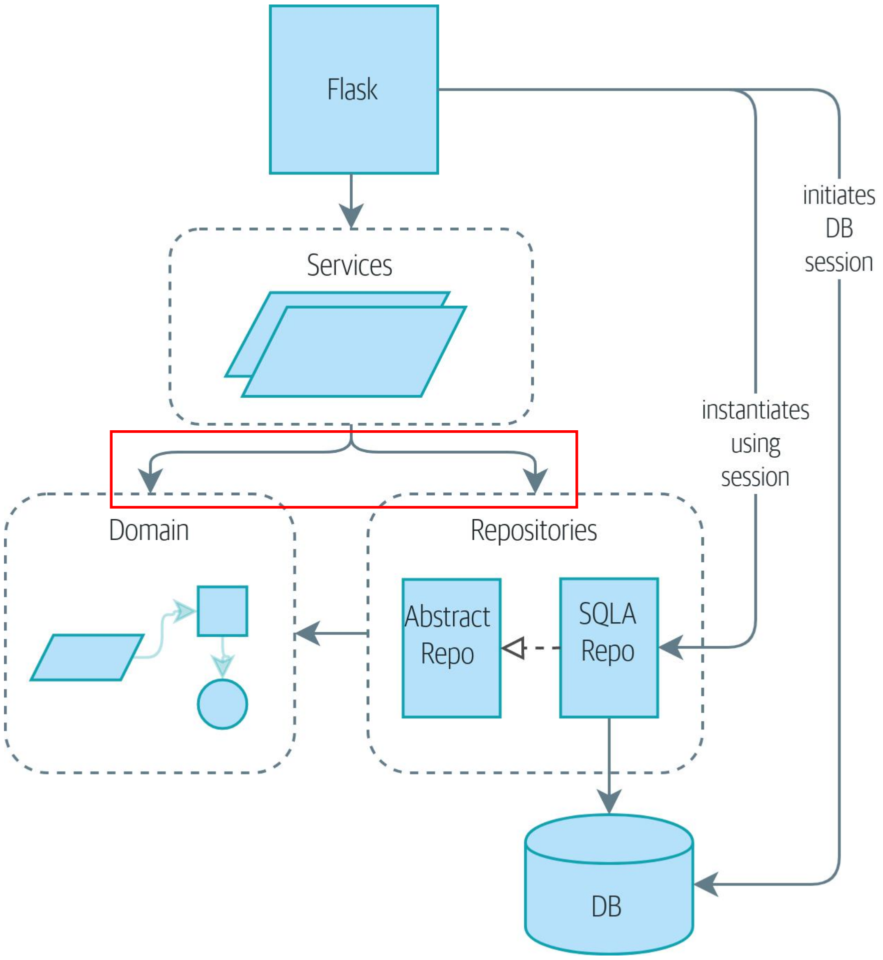
1. Too many responsibility: fetching stuff out of our repository, validating our input against database state, handling errors, and committing in the happy path
2. E2E test would be created for each code branches

Sprint3-2 Add service layer

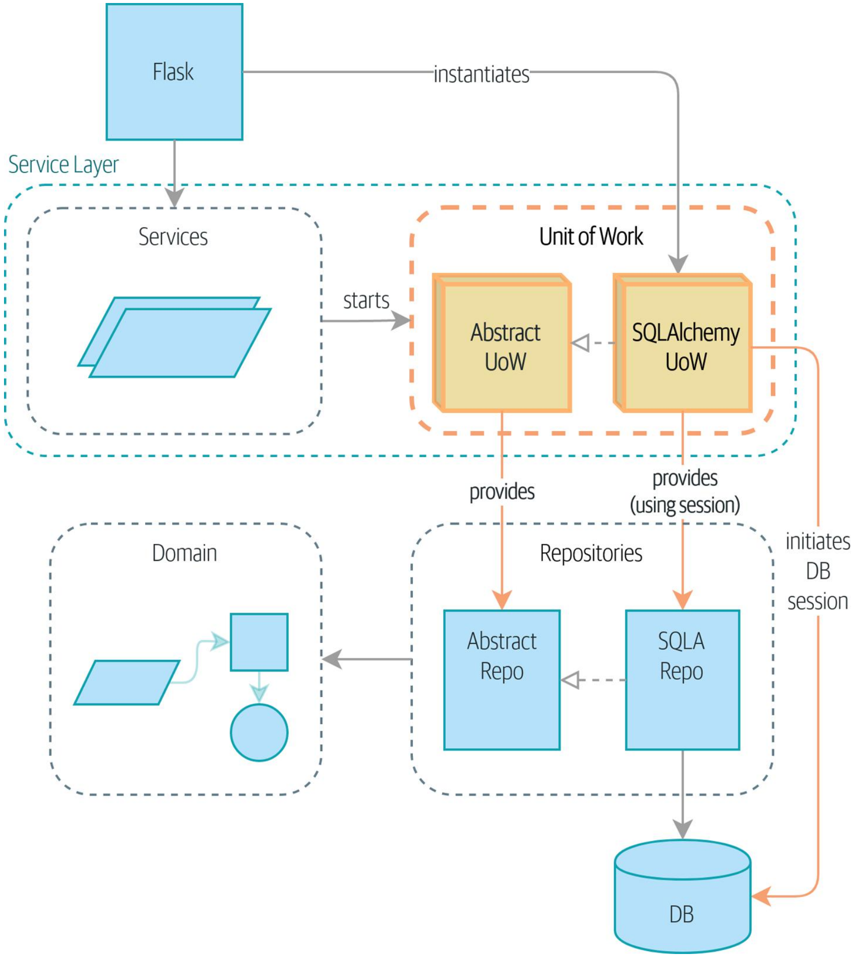


Pros	Cons
We have a single place to capture all the **use cases** for our application.	If your app is **purely** a web app, your controllers/view functions can be the single place to capture all the use cases.
We’ve placed our clever domain logic behind an API, which leaves us free to refactor.	It’s yet another layer of abstraction.
We have cleanly separated "stuff that talks HTTP" from "stuff that talks allocation."	Putting too much logic into the service layer can lead to the Anemic Domainanti-pattern. It’s better to introduce this layer after you spot orchestration logic creeping into your controllers.
When combined with the Repository pattern and FakeRepository, we have a nice way of writing tests at a higher level than the domain layer; we can test more of our workflow without needing to use integration tests (read on to TDD in High Gear and Low Gear for more elaboration on this).	You can get a lot of the benefits that come from having rich domain models by simply pushing logic out of your controllers and down to the model layer, without needing to add an extra layer in between (aka "fat models, thin controllers").

Sprint4



Fully decouple our service layer from the data layer

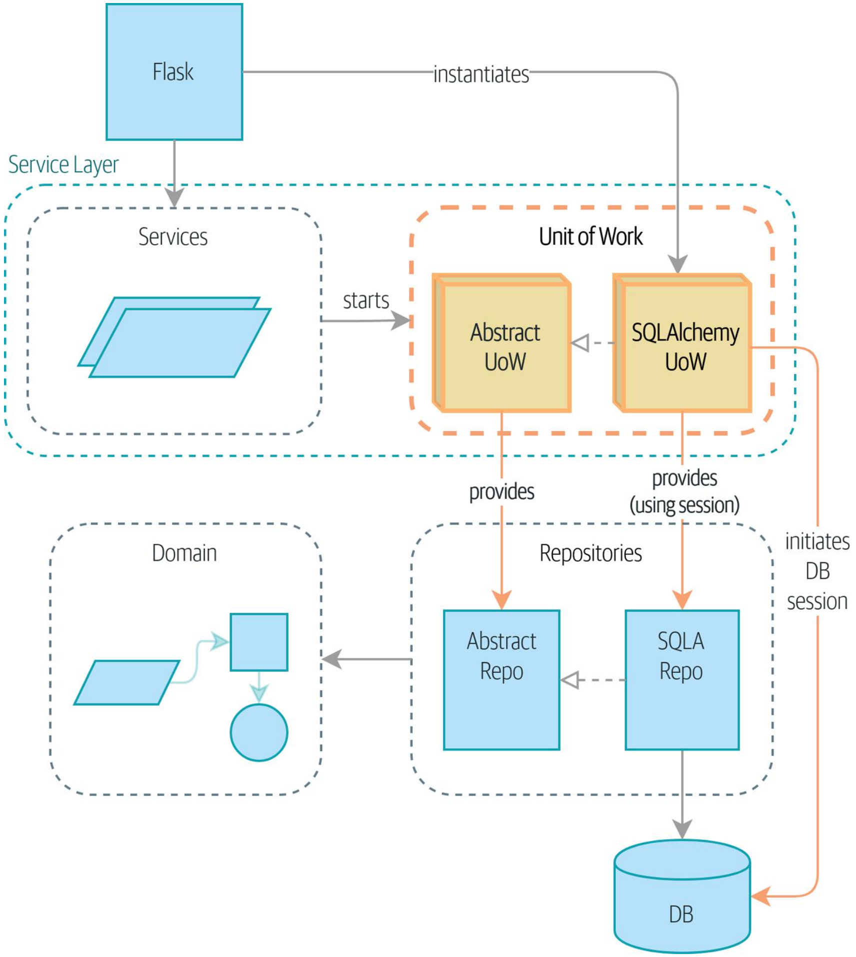


- The Flask API now does only two things: it initializes a unit of work, and it invokes a service.
- The service collaborates with the UoW (we like to think of the UoW as being part of the service layer), but neither the service function itself nor Flask now needs to talk directly to the database.

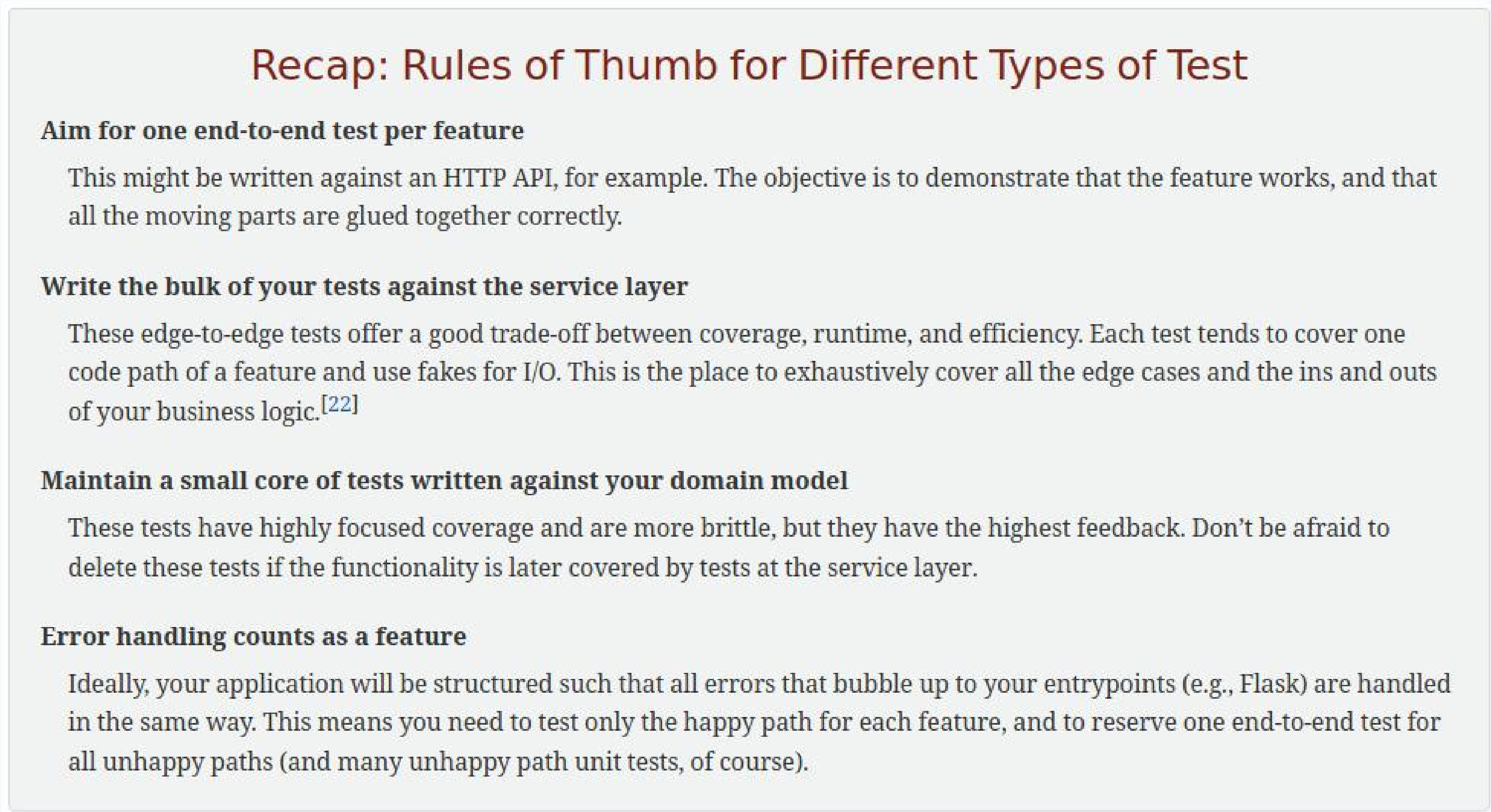
Pros	Cons
We have a nice abstraction over the concept of atomic operations, and the context manager makes it easy to see, visually, what blocks of code are grouped together atomically.	Your ORM probably already has some perfectly good abstractions around atomicity. SQLAlchemy even has context managers. You can go a long way just passing a session around.
We have explicit control over when a transaction starts and finishes, and our application fails in a way that is safe by default. We never have to worry that an operation is partially committed.	We’ve made it look easy, but you have to think quite carefully about things like rollbacks, multithreading, and nested transactions. Perhaps just sticking to what Django or Flask-SQLAlchemy gives you will keep your life simpler.
It’s a nice place to put all your repositories so client code can access them.	
As you’ll see in later chapters, atomicity isn’t only about transactions; it can help us work with events and the message bus.	

Question

1. What’s the responsibilities of repository, Unit of Work, Domain, services?
2. Pros and cons for each layer?
3. How to add a redis storage?



TDD



Recap: Rules of Thumb for Different Types of Test

Aim for one end-to-end test per feature

This might be written against an HTTP API, for example. The objective is to demonstrate that the feature works, and that all the moving parts are glued together correctly.

Write the bulk of your tests against the service layer

These edge-to-edge tests offer a good trade-off between coverage, runtime, and efficiency. Each test tends to cover one code path of a feature and use fakes for I/O. This is the place to exhaustively cover all the edge cases and the ins and outs of your business logic.[22]

Maintain a small core of tests written against your domain model

These tests have highly focused coverage and are more brittle, but they have the highest feedback. Don't be afraid to delete these tests if the functionality is later covered by tests at the service layer.

Error handling counts as a feature

Ideally, your application will be structured such that all errors that bubble up to your entrypoints (e.g., Flask) are handled in the same way. This means you need to test only the happy path for each feature, and to reserve one end-to-end test for all unhappy paths (and many unhappy path unit tests, of course).