...к ситуации на рис. 9.2, где все идет через шину сообщений и приложение было фундаментально преобразовано в процессор сообщений.

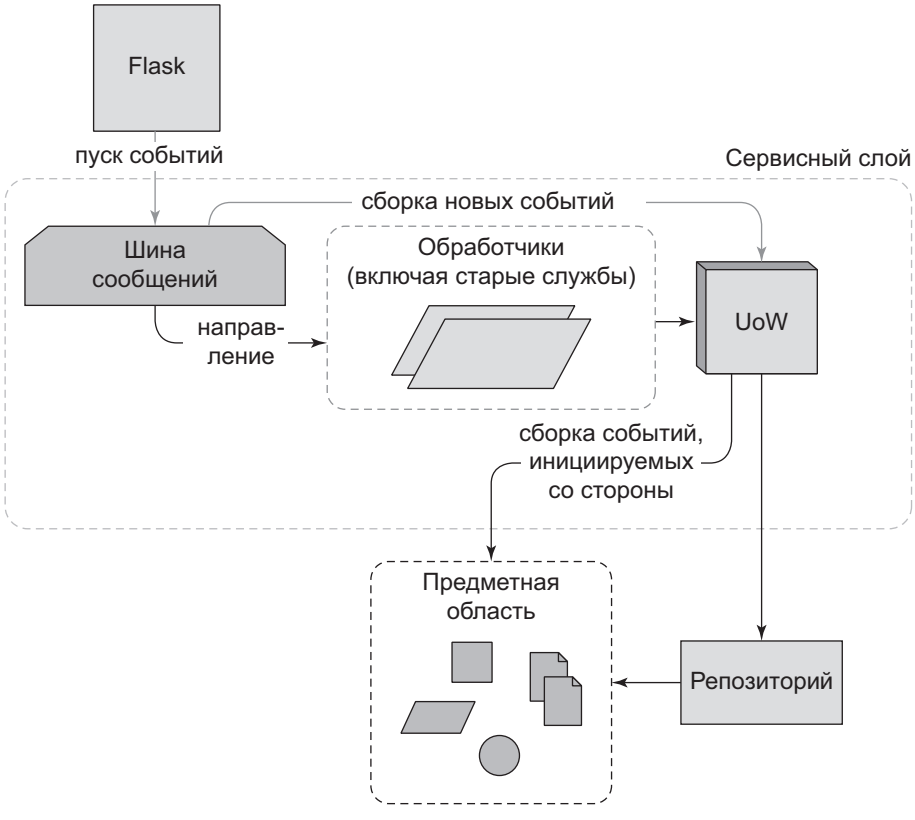

Рис. 9.2. Шина сообщений теперь является главной точкой входа в сервисный слой

Код для этой главы находится в ветке chapter_09_all_messagebus на GitHub[1]:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_09_all_messagebus
# или, если пишете код по ходу чтения, возьмите за основу
# материал из предыдущей главы:
git checkout chapter_08_events_and_message_bus
```

[1] См. https://oreil.ly/oKNkn

Новое требование приводит к новой архитектуре

Рич Хикки говорит о «ситуативном» (situated) ПО, имея в виду, что оно работает в течение длительных периодов, управляя реальным процессом. Примеры включают системы управления складами, логистические планировщики и системы расчета заработной платы.

Такое ПО сложно написать, потому что в реальном мире физических объектов и ненадежных людей внезапные вещи происходят постоянно. Например:

- во время инвентаризации мы обнаруживаем, что три артикула МАТРАС-ПРУЖИННЫЙ, `SPRINGY-MATTRESS`, промокли из-за протекающей на складе крыши;
- партия артикула ВИЛКА-НАДЕЖНАЯ, `RELIABLE-FORK`, не имеет необходимой документации и простаивает на таможне уже несколько недель. Три штуки артикула ВИЛКА-НАДЕЖНАЯ впоследствии не выдерживают испытания на безопасность и ломаются;
- глобальный дефицит блесток означает, что мы не можем произвести следующую партию артикула ШКАФ-СВЕРКАЮЩИЙ, `SPARKLY-BOOKCASE`.

В таких ситуациях мы узнаем о необходимости изменения размера партий товара, когда они уже находятся в системе. Возможно, кто-то ошибся номером в декларации или несколько диванов выпали из грузовика. После разговора со стейкхолдером[1] мы моделируем ситуацию, как показано на рис. 9.3.

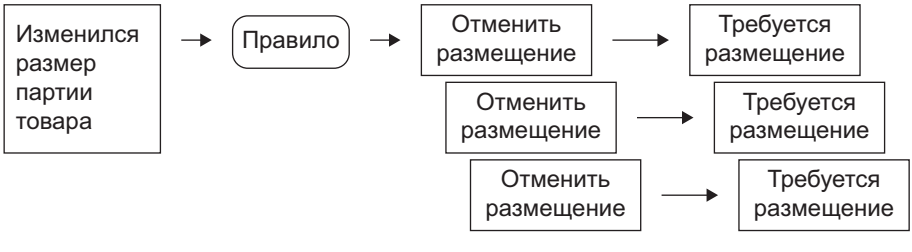

Рис. 9.3. Изменение размера партии товара означает отмену размещения и повторное размещение заказа

[1] Моделирование на основе событий настолько популярно, что для облегчения сбора требований на основе событий и разработки модели предметной области была разработана методика под названием *«событийный штурм»* (event storming).

Событие, которое мы назовем `BatchQuantityChanged`, должно привести к изменению размера партии, но не только. Мы также выполняем *бизнес-требование*: если в обновленной партии меньше товара, чем было заказано, то нужно *отменить* размещенные в ней заказы. Тогда каждый из этих заказов нужно разместить заново. Новое размещение можно захватывать как событие `AllocationRequired`.

Возможно, вы уже видите, что шина внутренних сообщений и события могли бы с этим помочь. Мы могли бы задать службу `change_batch_quanity`, которая знает, как корректировать размер партий товара, а также как отменять размещение заказа на лишние товарные позиции. Каждая такая отмена может инициировать событие `AllocationRequired`, которое будет перенаправляться в службу `allocate` в отдельных транзакциях. И снова обращаем внимание на то, что шина сообщений помогает обеспечивать соблюдение принципа единственной обязанности, что позволяет принимать взвешенные решения насчет транзакций и целостности данных.

Воображаемое изменение архитектуры: все будет обработчиком событий

Но прежде, чем приступить, подумайте о том, куда мы движемся. Через систему протекает два вида потоков:

- API-вызовы, которые обрабатываются функцией сервисного слоя.
- Внутренние события (которые могут быть инициированы как побочный эффект функции сервисного слоя) и их обработчики (которые, в свою очередь, вызывают функции сервисного слоя).

Разве не проще сделать все обработчиком событий? Если мы переосмыслим вызовы API как захват событий, то функции сервисного слоя также могут быть обработчиками событий, и тогда больше не нужно различать обработчики внутренних и внешних событий.

- `services.allocate()` может быть обработчиком события `AllocationRequired` и может порождать события `Allocated` на выходе.
- `services.add_batch()` может быть обработчиком события `BatchCreated`[1].

[1] Если вы немного читали о событийно-управляемой архитектуре, то, наверное, думаете, что некоторые из этих событий больше похожи на команды! Потерпите немного! Мы пытаемся вводить по одной концепции за раз. В следующей главе рассмотрим различие между командами и событиями.

Новое требование будет соответствовать той же схеме.

- Событие `BatchQuantityChanged` может активизировать обработчик `change_batch_quantity()`.
- Новые события `AllocationRequired`, которые он может инициировать, могут передаваться дальше в `services.allocate()`, поэтому нет никакой концептуальной разницы между совершенно новым размещением, поступающим из API, и повторным размещением, которое внутренне инициируется отменой размещения.

Звучит как-то чересчур, да? Давайте разбираться. Мы будем следовать рабочему процессу подготовительного рефакторинга[1], то есть «упрощать внесение изменений и потом вносить простые изменения».

1. Сделаем рефакторинг сервисного слоя в обработчики событий. Просто привыкните к идее, будто события описывают входы в систему. В частности, прежняя функция `services.allocate()` станет обработчиком для события `AllocationRequired`.
2. Создаем сквозной тест, который помещает события `BatchQuantityChanged` в систему и ищет выходящие события `Allocated`.
3. Реализация концептуально очень простая: это будет новый обработчик для событий `BatchQuantityChanged`, реализация которого будет порождать события `AllocationRequired`, которые, в свою очередь, будут обрабатываться точно таким же используемым в API обработчиком размещений.

По пути мы сделаем небольшую настройку шины сообщений и паттерна UoW, передав обязанность размещения новых событий в шине сообщений в саму шину сообщений.

Рефакторинг функций служб для обработчиков сообщений

Начнем с определения двух событий, которые захватывают текущие входы в API, — `AllocationRequired` и `BatchCreated`.

[1] См. https://oreil.ly/W3RZM

События BatchCreated и AllocationRequired (src/allocation/domain/events.py)

```
@dataclass
class BatchCreated(Event):
    ref: str
    sku: str
    qty: int
    eta: Optional[date] = None

...

@dataclass
class AllocationRequired(Event):
    orderid: str
    sku: str
    qty: int
```

Затем переименуем services.py в handlers.py, добавим прежний обработчик сообщений для `send_out_of_stock_notification` и, самое главное, поменяем все обработчики так, чтобы они имели одинаковые данные на входе, событие и UoW.

Обработчики и службы — это одно и то же (src/allocation/service_layer/handlers.py)

```
def add_batch(
        event: events.BatchCreated, uow: unit_of_work.AbstractUnitOfWork
):
    with uow:
        product = uow.products.get(sku=event.sku)
        ...

def allocate(
        event: events.AllocationRequired,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(event.orderid, event.sku, event.qty)
    ...

def send_out_of_stock_notification(
        event: events.OutOfStock, uow: unit_of_work.AbstractUnitOfWork,
):
    email.send(
        'stock@made.com',
        f'Артикула {event.sku} нет в наличии',
    )
```

ОТ ОБЪЕКТОВ ПРЕДМЕТНОЙ ОБЛАСТИ ЧЕРЕЗ ОДЕРЖИМОСТЬ ПРИМИТИВАМИ К СОБЫТИЯМ В КАЧЕСТВЕ ИНТЕРФЕЙСА

Некоторые из вас, возможно, помнят раздел «Устранение связей в тестах сервисного слоя с предметной областью» на с. 113, в котором мы поменяли API сервисного слоя с точки зрения объектов предметной области на примитивы. Теперь же мы возвращаемся назад, но к другим объектам. Что это дает?

В объектно-ориентированных кругах говорят об одержимости примитивами как об антипаттерне: избегать примитивов в публичных API и вместо этого, как бы они выразились, обертывать их собственными классами-значениями. В мире Python многие люди относятся к этому весьма скептически. При бездумном применении такой подход лишь все усложняет. Конечно, мы таким не занимаемся.

Переход от объектов предметной области к примитивам помог устранить связи: клиентский код больше не связан непосредственно с предметной областью, поэтому сервисный слой может представлять API, который не изменится, даже если мы поменяем что-нибудь в модели, и наоборот.

Значит, это шаг назад? Ну, ключевые объекты модели предметной области по-прежнему могут меняться, зато внешний мир мы привязали к классам событий. Да, это тоже часть предметной области, но мы надеемся, что классы не придется менять очень уж часто, так что они выглядят подходящими кандидатами для связывания.

А какая нам выгода? Теперь при вызове варианта использования в приложении больше не нужно запоминать конкретную комбинацию примитивов. Держать в уме нужно лишь один класс события, который представляет собой вход в приложение. В концептуальном плане это удобно. Кроме того, как вы увидите в приложении Д в конце книги, указанные событийные классы могут быть хорошим местом для проверки входных данных.

Это изменение, возможно, будет нагляднее в таком виде[1]:

Переход от служб к обработчикам (src/allocation/service_layer/handlers.py)

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

[1] Минус в начале строки означает удаление кода, плюс — добавление. — *Примеч. ред.*

```
def allocate(
-       orderid: str, sku: str, qty: int,
-       uow: unit_of_work.AbstractUnitOfWork
+       event: events.AllocationRequired, uow:
        unit_of_work.AbstractUnitOfWork
) -> str:
-   line = OrderLine(orderid, sku, qty)
+   line = OrderLine(event.orderid, event.sku, event.qty)
    ...

+
+def send_out_of_stock_notification(
+        event: events.OutOfStock, uow: unit_of_work.AbstractUnitOfWork,
+):
+   email.send(
    ...
```

Попутно мы сделали API сервисного слоя более структурированным и последовательным. Раньше это была россыпь примитивов, теперь же в нем используются четко определенные объекты (см. врезку выше).

Шина сообщений теперь собирает события из UoW

Теперь обработчикам событий нужен UoW. Кроме того, поскольку шина сообщений занимает уже центральное место в приложении, имеет смысл возложить на нее обязанность по сбору и обработке новых событий явным образом. До сих пор существовала некоторая циклическая зависимость между UoW и шиной сообщений, так что это сделает ее односторонней.

Обработчик принимает UoW и управляет очередью (src/allocation/service_layer/messagebus.py)

```
def handle(event: events.Event, uow: unit_of_work.AbstractUnitOfWork): ❶
    queue = [event] ❷
    while queue:
        event = queue.pop(0) ❸
        for handler in HANDLERS[type(event)]: ❸
            handler(event, uow=uow) ❹
            queue.extend(uow.collect_new_events()) ❺
```

❶ Теперь UoW проходит через шину сообщений всякий раз, когда запускается.

❷ Когда мы начинаем обрабатывать первое событие, мы запускаем очередь.

❸ Мы извлекаем события из начала очереди и активизируем их обработчики (словарь HANDLERS не изменился; он по-прежнему попарно сопоставляет типы событий с функциями-обработчиками).

❹ Шина сообщений передает UoW дальше каждому обработчику.

❺ После завершения работы каждого обработчика мы собираем все новые сгенерированные события и добавляем их в очередь.

В модуле unit_of_work.py publish_events() становится менее активным методом collect_new_events():

UoW больше не помещает события непосредственно в шину сообщений (src/allocation/service_layer/unit_of_work.py)

```
-from . import messagebus ❶
-

 class AbstractUnitOfWork(abc.ABC):
@@ -23,13 +21,11 @@ class AbstractUnitOfWork(abc.ABC):

     def commit(self):
         self._commit()
-        self.publish_events() ❷

-    def publish_events(self):
+    def collect_new_events(self):
         for product in self.products.seen:
             while product.events:
-                event = product.events.pop(0)
-                messagebus.handle(event)
+                yield product.events.pop(0) ❸
```

❶ Модуль unit_of_work теперь больше не зависит от messagebus.

❷ Мы больше не публикуем события, publish_events, автоматически при фиксации. Вместо этого шина сообщений отслеживает очередь событий.

❸ И UoW больше не помещает события в шину сообщений, он просто делает их доступными.

Все тесты тоже написаны с помощью событий

Тесты теперь работают, создавая события и помещая их в шину сообщений вместо того, чтобы активизировать функции сервисного слоя напрямую.

В тестах обработчиков используются события (tests/unit/test_handlers.py)

```
class TestAddBatch:

    def test_for_new_product(self):
        uow = FakeUnitOfWork()
-       services.add_batch("b1", "CRUNCHY-ARMCHAIR", 100, None, uow)
+       messagebus.handle(
+           events.BatchCreated("b1", "CRUNCHY-ARMCHAIR", 100, None), uow
+       )
        assert uow.products.get("CRUNCHY-ARMCHAIR") is not None
        assert uow.committed

...

class TestAllocate:

    def test_returns_allocation(self):
        uow = FakeUnitOfWork()
-       services.add_batch("batch1", "COMPLICATED-LAMP", 100, None, uow)
-       result = services.allocate("o1", "COMPLICATED-LAMP", 10, uow)
+       messagebus.handle(
+           events.BatchCreated("batch1", "COMPLICATED-LAMP", 100,
            None), uow
+       )
+       result = messagebus.handle(
+           events.AllocationRequired("o1", "COMPLICATED-LAMP", 10), uow
+       )
        assert result == "batch1"
```

Уродливый костыль: шине сообщений приходится возвращать результаты

API и сервисный слой теперь хотят знать ссылку на размещенную партию товара, когда вызывают обработчик `allocate()`. Это означает, что нужно вставить костыль в шину сообщений, чтобы она смогла возвращать события.

Шина сообщений возвращает результаты (src/allocation/service_layer/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)]:
-           handler(event, uow=uow)
+           results.append(handler(event, uow=uow))
            queue.extend(uow.collect_new_events())
+   return results
```

Способ некрасивый, потому что в системе мы смешиваем обязанности по чтению и записи. Пофиксим все в главе 12.

Изменение API для работы с событиями

Замена в коде Flask на шину сообщений (src/allocation/entrypoints/flask_app.py)

```
@app.route("/allocate", methods=['POST'])
def allocate_endpoint():
    try:
-       batchref = services.allocate(
-           request.json['orderid'], ❶
-           request.json['sku'],
-           request.json['qty'],
-           unit_of_work.SqlAlchemyUnitOfWork(),
+       event = events.AllocationRequired( ❷
+           request.json['orderid'], request.json['sku'],
            request.json['qty'],
        )
+       results = messagebus.handle(event,
                    unit_of_work.SqlAlchemyUnitOfWork()) ❸
+       batchref = results.pop(0)
    except InvalidSku as e:
```

❶ Вместо того чтобы вызывать сервисный слой со множеством примитивов, извлеченных из JSON-запроса...

❷ ...создаем экземпляр события.

❸ Затем передаем его в шину сообщений.

Ну вот, мы снова вернулись к абсолютно функциональному приложению, но теперь оно полностью управляется событиями.

- То, что раньше было функциями сервисного слоя, теперь является обработчиками событий.
- Это делает их такими же, как и функции, которые мы активизируем для обработки внутренних событий, инициированных моделью предметной области.
- Мы используем события в качестве специальной структуры для сбора данных на входе в систему, а также для передачи внутренних рабочих пакетов.

- Теперь самое подходящее описание для нашего приложения — процессор сообщений или процессор событий, если угодно. Поговорим об этом различии в следующей главе.

Реализация нового требования

С фазой рефакторинга разобрались. Теперь посмотрим, действительно ли мы «упростили внесение изменений». Реализуем новое требование, показанное на рис. 9.4: на входе будем получать новые события `BatchQuantityChanged` и передавать их обработчику, который будет порождать события `AllocationRequired`, а те, в свою очередь, будут возвращаться к существующему обработчику для повторного размещения.

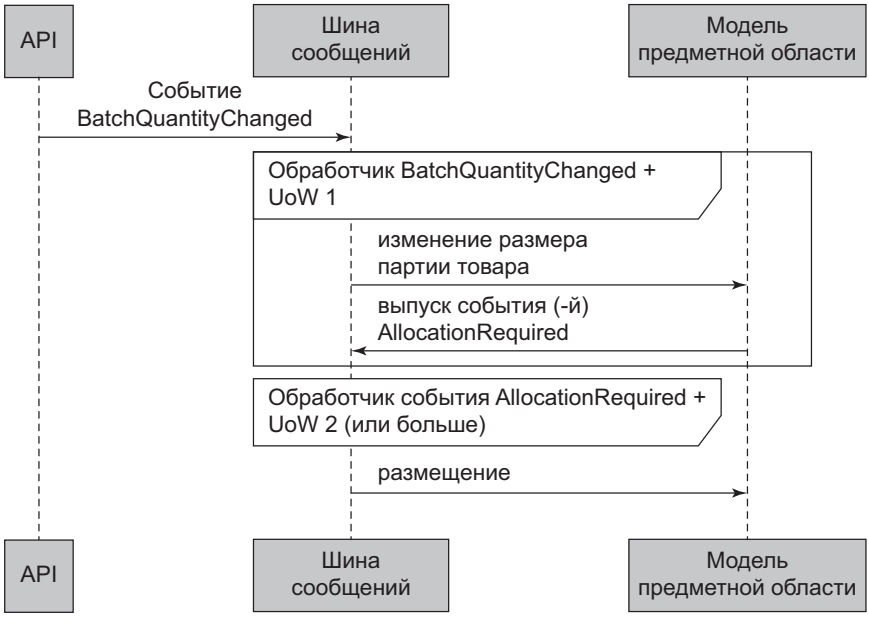

Рис. 9.4. Схема последовательности для потока повторного размещения заказа

Когда вы разделяете такие вещи на два UoW, у вас будут проходить две транзакции базы данных, и таким образом вы навлекаете на себя проблемы целостности данных: может возникнуть ситуация, в которой первая транзакция завершилась, а вторая — нет. Стоит подумать, насколько это приемлемо, нужно ли отслеживать такие вещи и что с ними делать. Подробности см. в разделе «Выстрел в ногу» на с. 287.

Новое событие

Событие, которое сообщает нам о том, что размер партии товара изменился, очень простое; ему нужна только ссылка на партию и ее новый размер.

Новое событие (src/allocation/domain/events.py)

```
@dataclass
class BatchQuantityChanged(Event):
    ref: str
    qty: int
```

Тест-драйв нового обработчика

Вспоминая урок из главы 4, мы можем «ехать на повышенной передаче» и писать юнит-тесты на максимально возможном уровне абстракции с помощью событий. Вот как они могут выглядеть:

Тесты обработчика для change_batch_quantity (tests/unit/test_handlers.py)

```
class TestChangeBatchQuantity:

    def test_changes_available_quantity(self):
        uow = FakeUnitOfWork()
        messagebus.handle(
            events.BatchCreated("batch1", "ADORABLE-SETTEE", 100,
            None), uow
        )
         [batch] = uow.products.get(sku="ADORABLE-SETTEE").batches
        assert batch.available_quantity == 100 ❶

        messagebus.handle(events.BatchQuantityChanged("batch1", 50), uow)

        assert batch.available_quantity == 50 ❶

    def test_reallocates_if_necessary(self):
        uow = FakeUnitOfWork()
        event_history = [
            events.BatchCreated("batch1", "INDIFFERENT-TABLE", 50, None),
            events.BatchCreated("batch2", "INDIFFERENT-TABLE", 50,
            date.today()),
            events.AllocationRequired("order1", "INDIFFERENT-TABLE", 20),
            events.AllocationRequired("order2", "INDIFFERENT-TABLE", 20),
        ]
        for e in event_history:
```

```
        messagebus.handle(e, uow)
    [batch1, batch2] = uow.products.get(
    sku="INDIFFERENT-TABLE").batches
    assert batch1.available_quantity == 10
    assert batch2.available_quantity == 50

    messagebus.handle(events.BatchQuantityChanged("batch1", 25), uow)

    # размещение заказа order1 или order2 будет отменено, и у нас
    # будет 25 - 20
    assert batch1.available_quantity == 5 ❷
    # и 20 будет повторно размещено в следующей партии
    assert batch2.available_quantity == 30 ❷
```

❶ Простой случай реализуется тривиально — просто изменяем размер партии.

❷ Если попытаться уменьшить размер партии, но при этом был размещен более крупный заказ, то придется отменить размещение по крайней мере одного заказа, чтобы затем повторно разместить его в новой партии товара.

Реализация

Новый обработчик очень прост.

Обработчик делегирует обязанности в слой модели (src/allocation/service_layer/handlers.py)

```
def change_batch_quantity(
        event: events.BatchQuantityChanged, uow:
        unit_of_work.AbstractUnitOfWork
):
    with uow:
        product = uow.products.get_by_batchref(batchref=event.ref)
        product.change_batch_quantity(ref=event.ref, qty=event.qty)
        uow.commit()
```

Мы понимаем, что понадобится новый тип запроса в репозиторий.

Новый тип запроса в репозиторий (src/allocation/adapters/repository.py)

```
class AbstractRepository(abc.ABC):
    ...

    def get(self, sku) -> model.Product:
        ...
```

```
    def get_by_batchref(self, batchref) -> model.Product:
        product = self._get_by_batchref(batchref)
        if product:
            self.seen.add(product)
        return product

    @abc.abstractmethod
    def _add(self, product: model.Product):
        raise NotImplementedError

    @abc.abstractmethod
    def _get(self, sku) -> model.Product:
        raise NotImplementedError

    @abc.abstractmethod
    def _get_by_batchref(self, batchref) -> model.Product:
        raise NotImplementedError
    ...

class SqlAlchemyRepository(AbstractRepository):
    ...

    def _get(self, sku):
        return self.session.query(model.Product).filter_
               by(sku=sku).first()

    def _get_by_batchref(self, batchref):
        return self.session.query(model.Product).join
               (model.Batch).filter(
            orm.batches.c.reference == batchref,
        ).first()
```

И в поддельный репозиторий, `FakeRepository`, тоже.

Обновляем поддельный репозиторий (tests/unit/test_handlers.py)

```
class FakeRepository(repository.AbstractRepository):
    ...

    def _get(self, sku):
        return next((p for p in self._products if p.sku == sku), None)

    def _get_by_batchref(self, batchref):
        return next((
            p for p in self._products for b in p.batches
            if b.reference == batchref
        ), None)
```

Мы добавляем запрос в репозиторий, чтобы упростить реализацию этого варианта использования. До тех пор пока запрос возвращает один-единственный агрегат, никакие правила не нарушаются. Если же вы пишете сложные запросы к своим репозиториям, то подумайте об изменении дизайна. Такие методы, как «получить наиболее популярный продукт», `get_most_popular_products`, или «найти продукты по идентификатору заказа», `find_products_by_order_id`, помогут вам найти правильное решение. В главе 11 и эпилоге будет несколько советов по управлению сложными запросами.

Новый метод в модели предметной области

Мы добавляем в модель новый метод, который меняет размер партии и размещение в одной строке кода и публикует новое событие. Изменим и прежнюю функцию размещения с учетом публикации события.

Модель улучшается с учетом нового требования (src/allocation/domain/model.py)

```
class Product:
    ...

    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)
            )
...

class Batch:
    ...
    def deallocate_one(self) -> OrderLine:
         return self._allocations.pop()
```

Подключаем новый обработчик.

Шина сообщений растет (src/allocation/service_layer/messagebus.py)

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

И вот новое требование теперь полностью реализовано.