Лабораторная работа №5
«Доменная модель»

Цель работы

Познакомиться с тактическими шаблонами предметно-ориентированного проектирования

Задание для выполнения

Реализуйте не менее двух агрегатов, сущностей, объектов значений домена Вашей гексагональной архитектуры из предыдущей работы, а также репозитории и не менее двух доменных событий для агрегатов.

Один агрегат должен включать в себя сущность-корень агрегата, список дочерних сущностей, каждая сущность содержать наряду с обычными свойствами ещё и объекты-значения.

Идентификаторы сущностей должны генерироваться: для корней агрегатов – репозиторием, для внутренних сущностей – самим корнем агрегата.

Проверку работы агрегатов организовать путём тестирования (ЛР №4).

Теоретические сведения

Как и в предыдущей работе, теории здесь чуть больше, чем нужно только для доменного уровня, затрагиваются также и уровни приложения и инфраструктуры.

В DDD каждый сервис имеет собственную доменную модель, что позволяет избежать проблем с единой доменной моделью, которая охватывает все приложение. Поддомены и связанная с ними концепция изолированного контекста — это два стратегических шаблона DDD.

В DDD есть также тактические шаблоны, которые служат строительными блоками для доменных моделей. Каждый шаблон представляет собой роль, которую класс играет в доменной модели, и описывает характеристики этого класса. Разработчики широко применяют следующие строительные блоки.

Сущность — объект, обладающий устойчивой идентичностью. Две сущности, чьи атрибуты имеют одинаковые значения, — это все равно разные объекты.

Объект значений — объект, представляющий собой набор значений. Два объекта значений с одинаковыми атрибутами взаимозаменяемы.

Примером таких объектов может служить класс Money, который состоит из валюты и суммы.

Фабрика — объект или метод, реализующий логику создания объектов, которую ввиду ее сложности не следует размещать прямо в конструкторе. Фабрика также может скрывать конкретные классы, экземпляры которых создает. Она реализуется в виде статического метода или класса.

Репозиторий — объект, предоставляющий доступ к постоянным сущностям и инкапсулирующий механизм доступа к базе данных.

Сервис — объект, реализующий бизнес-логику, которой не место внутри сущности или объекта значений.

Многие разработчики используют эти строительные блоки. Некоторые из них поддерживаются такими фреймворками, как JPA и Spring. Но есть еще одна концепция, которую обычно игнорируют все, за исключением истинных ценителей DDD. Речь идет об агрегатах. Несмотря на свою непопулярность, этот строительный блок чрезвычайно полезен при разработке микросервисов [1].

Агрегат

Представьте, к примеру, что вы хотите выполнить с бизнес-объектом Order операцию, такую как загрузка или удаление. Что именно это означает? Какова область применения этой операции? Загрузить или удалить объект Order — не проблема. Но в реальности заказ не ограничивается одним лишь этим объектом. Вам придется иметь дело с позициями заказа, информацией о платеже и т. д. На рисунке 1 границы доменного объекта оставлены на усмотрение разработчика:

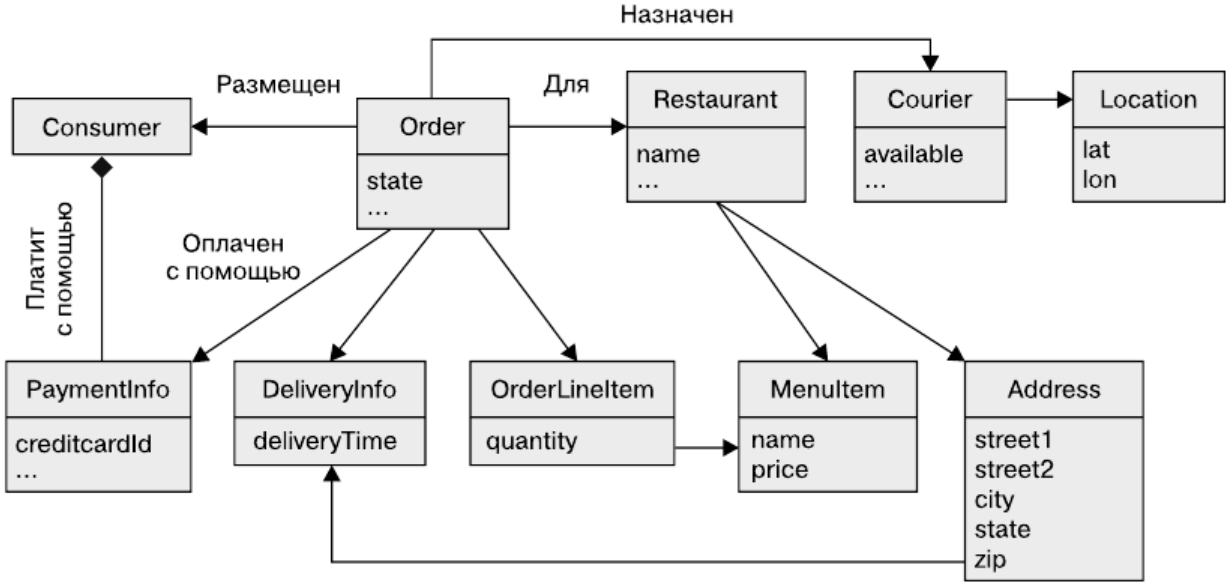

Рисунок 1 – Традиционная доменная модель

Помимо концептуальной неопределенности, отсутствие четких границ вызывает проблемы при обновлении бизнес-объекта. Типичный бизнес-объект

имеет *инварианты* — бизнес-правила, которые всегда должны соблюдаться. Например, для заказа есть минимальная сумма.

Агрегат — это кластер доменных объектов, с которыми можно обращаться как с единым целым. Он состоит из корневой сущности и иногда одной или нескольких сущностей и объектов значений. Многие бизнес-объекты моделируются в виде агрегатов:

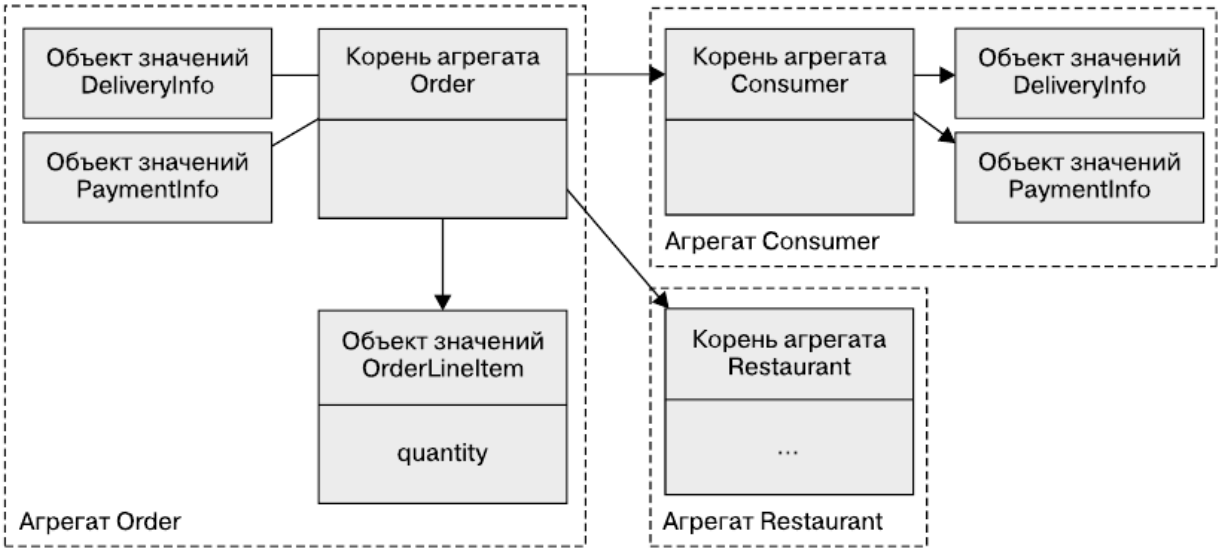

Рисунок 2 – Чёткие границы при структуризации доменной модели в виде набора агрегатов

Агрегаты разбивают доменную модель на блоки, в которых легче разобраться по отдельности. Они также проясняют область применения операций, таких как загрузка, обновление и удаление. Эти операции распространяются на весь агрегат, а не на какие-то его части. Агрегат часто загружается из базы данных целиком, что позволяет избежать любых проблем с ленивой загрузкой. Вместе с агрегатом из базы данных удаляются все его объекты.

Агрегаты — это границы согласованности

Обновление целого агрегата, а не отдельных его частей решает проблемы с согласованностью, подобных описанным в предыдущем примере. Операции обновления вызываются для корня агрегата, который обеспечивает соблюдение инвариантов.

Кроме того, чтобы поддерживать конкурентность, корень агрегата блокируется с помощью, скажем, номера версии или блокировки уровня базы данных. Например, вместо непосредственного обновления количества единиц для определенных позиций клиент должен вызвать метод из корня агрегата Order, который следит за соблюдением таких инвариантов, как минимальная сумма заказа. Однако следует упомянуть, что этот подход не требует обновления всего агрегата в базе данных. Приложение может, к примеру, обновить поля, относящиеся к заказу Order и измененному объекту OrderLineItem.

Главное — определить агрегаты

В DDD ключевым аспектом проектирования доменной модели является определение агрегатов, их границ и корней. Детали внутренней структуры агрегатов вторичны. Однако преимущества этого подхода далеко не ограничены разделением доменной модели на модули. Причина этого в том, что агрегаты обязаны придерживаться определенных правил.

Правила для агрегатов

DDD требует, чтобы агрегаты подчинялись набору правил. Это делает агрегат автономной единицей, способной обеспечивать соблюдение инвариантов. Рассмотрим эти правила.

Правило 1. Ссылайтесь только на корень агрегата

Предыдущий пример проиллюстрировал риски прямого обновления объекта OrderLineItems. Первое правило призвано устранить эту проблему. Оно требует, чтобы корневая сущность была единственной частью агрегата, на которую могут ссылаться внешние классы. Для обновления агрегата клиенту необходимо вызвать метод из его корня.

Например, сервис использует репозиторий, чтобы загрузить агрегат из базы данных и получить ссылку на его корень. С помощью метода, вызываемого из корня, он обновляет агрегат. Это правило гарантирует, что агрегат способен обеспечивать соблюдение своих инвариантов.

Правило 2. Межагрегатные ссылки должны применять первичные ключи

Правило состоит в том, что агрегаты ссылаются друг на друга по уникальному значению, например по первичному ключу, а не по объектным ссылкам. На рисунке 3 показано, как заказ ссылается на своего заказчика с помощью consumerId, а не ссылки на объект Consumer. Аналогичным образом заказ ссылается на ресторан с использованием restaurantId.

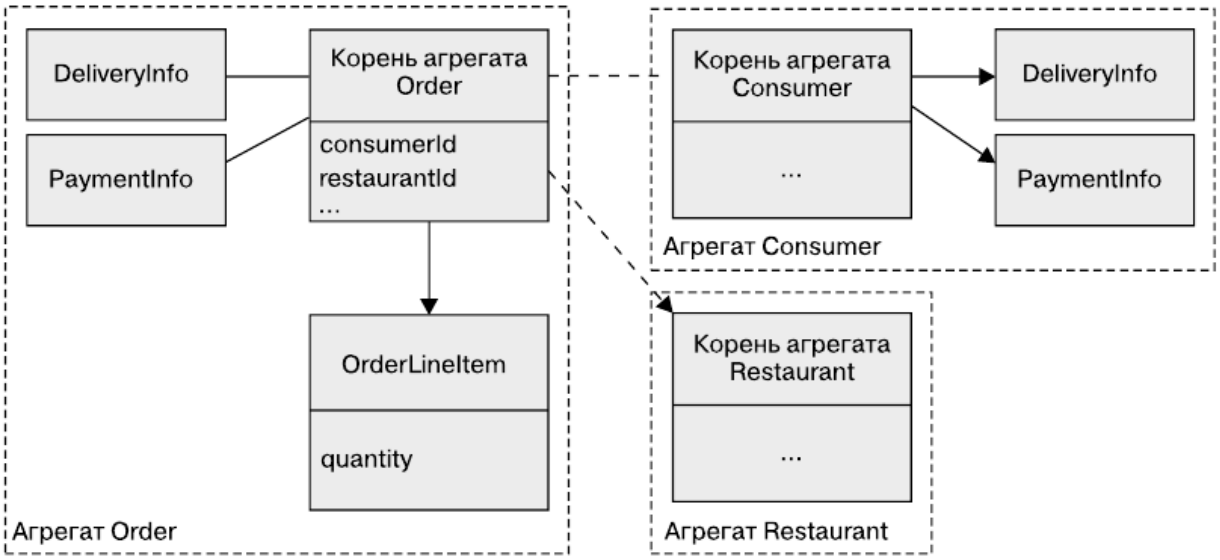

Рисунок 3 – Ссылки на агрегаты – по ключам (идентификаторам)

Этот подход заметно отличается от традиционного моделирования объектов, с точки зрения которого использование внешних ключей в доменной модели — плохое архитектурное решение. Данный метод имеет ряд преимуществ. Отказ от объектных ссылок в пользу уникальных идентификаторов означает, что агрегаты слабо связаны между собой. Это позволяет четко определить границы между ними и избежать случайного обновления не того агрегата. К тому же нам не нужно беспокоиться об объектных ссылках, которые выходят за пределы одного сервиса.

Правило 3. Одна транзакция создает или обновляет один агрегат

Еще одно правило, которому должны подчиняться агрегаты, состоит в том, что транзакция может создать или обновить только один агрегат. Когда я впервые прочитал о нем много лет назад, оно показалось мне бессмысленным! *(здесь и в других местах прямая речь от Криса Ричардсона, автора [1]).*

В то время я занимался разработкой традиционных монолитных приложений с использованием СУРБД, поэтому в моем случае транзакции могли обновлять множественные агрегаты. Но в наши дни это ограничение идеально подходит для микросервисной архитектуры. Оно гарантирует, что транзакция не выйдет за пределы сервиса. А также хорошо согласуется с ограниченной транзакционной моделью большинства баз данных NoSQL.

Это правило усложняет реализацию операций, которым нужно создавать или обновлять несколько агрегатов. Но это явно одна из тех проблем, для решения которых предназначены повествования (в последующих лабораторных работах). Каждый этап повествования создает или обновляет ровно один агрегат. На рисунке 4 показано, как это работает.

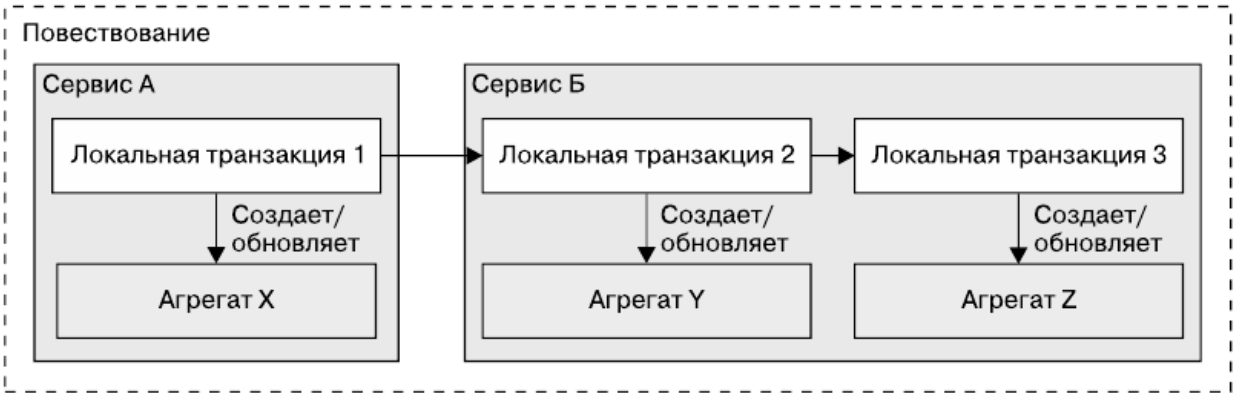

Рисунок 4 – Пример повествования

В этом примере повествование состоит из трех транзакций. Первая транзакция обновляет агрегат X в сервисе A. Две другие происходят в сервисе B: одна транзакция обновляет агрегат Y, а другая — агрегат Z.

Есть и альтернативный вариант. Чтобы обеспечить согласованность между несколькими агрегатами внутри одного сервиса, мы могли бы «сжульничать» и обновить все эти агрегаты в рамках одной транзакции. Например, одной транзакции могло бы быть достаточно для обновления агрегатов Y и Z в сервисе B. Но это возможно только в СУРБД с развитой транзакционной моделью. Если вы применяете базу данных NoSQL, которая поддерживает только простые транзакции, у вас нет другого варианта, кроме как использовать повествования.

Но так ли это? Оказывается, границы между агрегатами не высечены в камне. При разработке доменной модели вы сами можете определить, где они должны проходить. Но, помня о том, как колониальные державы чертили государственные границы в прошлом веке, вы должны подходить к этому осторожно.

Объекты значений v Сущности

Типичный пример объекта значений – класс Money, который включает в себя равно количество, так и наименование валюты. Другой пример – адрес, который включает в себя индекс, город, улицу, дом, квартиру.

```
class Money
{
    private $amount;
    private $currency;

    public function __construct($anAmount, Currency $aCurrency)
    {
        $this->setAmount($anAmount);
        $this->setCurrency($aCurrency);
    }

    private function setAmount($anAmount)
    {
        $this->amount = (int) $anAmount;
    }
```

```
    private function setCurrency(Currency $aCurrency)
    {
        $this->currency = $aCurrency;
    }

    public function amount()
    {
        return $this->amount;
    }

    public function currency()
    {
        return $this->currency;
    }
}
```

В отличие от объектов значений – сущности имеют идентификатор, однозначно определяющий данную сущность.

«10 долларов» - это объект значений класса Money,

«10-долларовая купюра» - это сущность класса Купюра, имеющая свойство Номинал типа Money, а также идентификатор НомерКупюры, отличающий эти 10 долларов от других 10 долларов.

Генерация идентичности сущности

В классических подходах для генерации идентификатора сущностей часто полагаются на механизм хранения – базу данных. Например, на поле с атрибутом AUTO_INCREMENT в БД MySQL.

```
CREATE TABLE `orders` (
    `id` int(11) NOT NULL auto_increment,
    `amount` decimal (10,5) NOT NULL,
    `first_name` varchar(100) NOT NULL,
    `last_name` varchar(100) NOT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

В этом случае модель получит идентификатор только после сохранения в БД, что очень неудобно, поскольку он может быть необходим в процессе обработки (например, для публикации в событии, чтобы потребители событий знали, в каком именно агрегате они произошли).

Среди других подходов можно выделить генерацию идентификатора клиентом, суррогатную идентичность и пр. Мы же будем рассматривать следующий подход: идентификатор корня агрегата генерируется приложением, реализацией специального метода репозитория `nextIdentity()`:

```
interface OrderRepositoryInterface
{
    public function nextIdentity();
/// ...
}
```

Это позволит в зависимости от реализации данного репозитория на инфраструктурном уровне релизовать как генерацию в БД MySQL, так и, например, генерацию UUID, или фейковых идентификаторов для тестирования.

Идентификатор, единожды сгенерированный, является неизменным (immutable). Для максимальной степени обеспечения данного правила идентификаторы представляются не простыми свойствами ординарного типа, которые могут быть просто изменены, а объектами значений:

```
abstract class Id
{
    /**
     * @var string The identifier
     */
    private $id;

    public function __construct(string $anId)
    {
        $this->id = $anId;
    }
    public function equals(ValueObjectInterface $object): bool
    {
        /** @noinspection PhpNonStrictObjectEqualityInspection */
        /** @noinspection TypeUnsafeComparisonInspection */
        return $this == $object;
    }

    /**
     * @return mixed
     */
    public function getValue()
    {
        return $this->id;
    }
}
```

На доменном уровне в модели Payment:

```
class PaymentId extends Id
{
}

interface PaymentRepositoryInterface
{
    /**
     * @return PaymentId
     */
    public function nextIdentity();

    // ...
}
```

На инфраструктурном уровне реализовываем репозиторий:

```
class PaymentRepository implements PaymentRepositoryInterface
{
    /**
```

```
     * @return PaymentId
     */
    public function nextIdentity()
    {
        /*
        return new PaymentId(
            RedisMysqlIdGenerator::generateIdentity(Tables::TABLE_PAYMENT)
        );
        */
        return new PaymentId(uniqid());
    }
}
```

Теперь, при создании нового агрегата в сервисе уровня приложения мы сразу генерируем для него новый идентификатор:

```
$payment = Payment::create($this->paymentRepository->nextIdentity(), ...);
```

А загружая агрегат для в обработчике команды (также на уровне приложения) новый идентификатор создавать не нужно, а предстоит работать с уже существующим, значение которого было передано в команде:

```
$payment = $this->paymentRepository->get(new PaymentId($command->getPaymentId()));
```

Ок, понятно, где и как генерировать идентификаторы для корня агрегата, но как быть с сущностями, входящими внутрь агрегата?

```
class SellerList
{
    /** @var ListId */
    private $id;

    /** @var Item[] */
    private $listItems;

    /**
     * SellerList constructor.
     * @param ListId $id
     */
    public function __construct(
        ListId $id
    )
    {
        $this->id = $id;
        $this->listItemIds = [];
    }
}
```

В этом случае важно, что доступ к данным сущностям напрямую отсутствует, а производится только через корень агрегата. А значит. уникальность данных сущностей в рамках системы не важна, а необходимо её поддерживать только в рамках данного агрегата (любым способом):

```
class SellerList
{
    /** @var ListId */
    private $id;

    /** @var Item[] */
    private $listItems;

    /**
     * SellerList constructor.
     * @param ListId $id
     */
    public function __construct(
        ListId $id
    )
    {
        $this->id = $id;
        $this->listItemIds = [];
    }

    public function getListItems(): array
    {
        return $this->listItemIds;
    }

    public function removeItem(Item $item)
    {
        unset($this->listItems[$item->getId()->getValue()]);
    }

    public function addItem(string $name)
    {
        $id = count($this->listItems);
        $this->listItems[$id] = new ListItem(new ItemId($id), $name);
    }
}
```

Публикация доменных событий

В контексте DDD доменное событие — это нечто произошедшее с агрегатом. В доменной модели оно имеет вид класса. Событие обычно представляет изменение состояния. Возьмем, к примеру, агрегат Order в приложении FTGO. В число событий, которые меняют его состояние, входят Order Created, Order Cancelled, Order Shipped и т. д. При наличии заинтересованных потребителей агрегат Order может публиковать одно из этих событий в момент изменения своего состояния.

Полезность доменных событий связана с тем, что другие стороны взаимодействия (пользователи, внешние приложения или другие компоненты

внутри того же приложения) часто заинтересованы в информации об изменениях в состоянии агрегата.

Доменное событие — это класс с именем на основе страдательного причастия прошедшего времени. Он содержит свойства, которые выразительно передают это событие. Каждое свойство представляет собой либо простое значение, либо объект. Например, класс события OrderCreated содержит свойство orderId.

У доменного события обычно есть метаданные, такие как его идентификатор и временная метка. Оно может нести в себе идентификатор пользователя, который сделал изменение, поскольку это полезно для аудита. Метаданные могут быть частью объекта события — возможно, определенные в родительском классе. Или же они могут находиться внутри обертки вокруг объекта события. Идентификатор агрегата, который сгенерировал событие, тоже может быть не его непосредственным свойством, а частью обертки [1].

```php
Namespace My\Ddd\Domain\Event\DomainEventInterface;

interface DomainEventInterface
{
    /**
     * @return DateTimeInterface
     */
    public function getOccurredOn();
}
```

```php
namespace My\Payment\Domain\Model\Payment\Event;

use My\Ddd\Domain\Event\DomainEventInterface;

class Created implements DomainEventInterface
{
    /**
     * @var string
     */
    protected $providerCode;

    /** @var string */
    private $paymentId;
    /**
     * @var \DateTimeImmutable
     */
    private $occurredOn;

    /**
     * Created constructor.
     * @param string $paymentId
     * @param string $providerCode
     */
    public function __construct(string $paymentId, string $providerCode)
    {
        $this->paymentId = $paymentId;
        $this->providerCode = $providerCode;
        $this->occurredOn = new \DateTimeImmutable();
```

```
    }

    public function getPaymentId(): string
    {
        return $this->paymentId;
    }

    public function getProviderCode(): string
    {
        return $this->providerCode;
    }

    public function getOccurredOn(): \DateTimeImmutable
    {
        return $this->occurredOn;
    }
}
```

Генерация событий

Исключительно агрегат, инкапсулирующий в себе критическую бизнес-логику, знает, когда с ним что-либо произошло. Поэтому создаются объекты событий внутри агрегата в процессе выполнения любых действий.

```
class Payment implements AggregateRootInterface
{
    use AggregateTrait;

    public function paid()
    {
        $this->status = Status::PAID;
        $this->recordEvent(
            new Paid(
                $this->getId()->getValue(),
                $this->getAmount()
            )
        );
    }
}
```

Публикация событий

Для того, чтобы не переплетать бизнес-логику агрегата и инфраструктурную часть механики публикации событий, этот процесс ложится на плечи сервиса уровня приложения, который с использованием механизмов внедрения зависимостей (реализация DIP) фактически публикует события адаптерами инфраструктурного уровня, например, в очередь сообщений.

Надёжность публикации событий

Работая с локальными транзакциями необходимо и события публиковать в рамках транзакций. Один из подходов: все события

накапливаются в агрегате, а извлекаются из него и публикуются только в момент фиксации (commit) транзакции.

Что это даёт? Если в какой-то момент что-то пошло не так, и транзакция откачена (rollback), то агрегат не изменился, и ни одно из событий, которые уже произошли в агрегате к тому моменту, не должно быть обработано другими сервисами.

```php
namespace My\Ddd\Domain\Traits;

use My\Ddd\Domain\Event\DomainEventInterface;

trait AggregateTrait
{
    /**
     * @var DomainEventTrait[]
     */
    private $events = [];

    /**
     * @param DomainEventTrait $event
     */
    public function recordEvent(DomainEventInterface $event)
    {
        $this->events[] = $event;
    }

    /**
     * @return DomainEventTrait[]
     */
    public function releaseEvents(): array
    {
        $events       = $this->events;
        $this->events = [];

        return $events;
    }
}
```

Обработчик команды, которая оплачивает платёж (сервис уровня приложения):

```php
namespace My\Payment\Application\Command\PaidPayment;

use My\Ddd\Application\Service\TransactionInterface;
use My\Ddd\Domain\Event\EventDispatcherInterface;
use My\Payment\Domain\Model\Payment\PaymentId;
use My\Payment\Domain\Model\Payment\PaymentRepositoryInterface;

class PaidPaymentHandler implements PaidPaymentHandlerInterface
{

    /**
     * @var PaymentRepositoryInterface
     */
    private $paymentRepository;
```

```
    /**
     * @var TransactionInterface
     */
    private $transaction;
    /**
     * @var EventDispatcherInterface
     */
    private $eventDispatcher;

    public function __construct(
        PaymentRepositoryInterface $paymentRepository,
        TransactionInterface $transaction,
        EventDispatcherInterface $eventDispatcher
    )
    {
        $this->paymentRepository = $paymentRepository;
        $this->transaction       = $transaction;
        $this->eventDispatcher   = $eventDispatcher;
    }

    /**
     * @param PaidPaymentCommand $command
     * @return mixed|void
     * @throws \Exception
     */
    public function handle($command)
    {
        $payment = $this->paymentRepository->get(
            new PaymentId($command->getPaymentId())
        );

        $payment->paid();

        $this->transaction->execute(
            function () use ($payment) {
                $this->paymentRepository->save($payment);
                $this->eventDispatcher->dispatch($payment->releaseEvents());
            }
        );
    }
}
```

Здесь интерфейс транзакции инвертирует зависимость от инфраструктурного уровня, где транзакции реально реализованы при помощи ORM, а действия с БД завернуты в callable, реализуя концепцию *Unit of Work* во избежание прерывания действия посередине (критическая секция):

```
namespace My\Ddd\Infrastructure\Application\Service;

use My\Ddd\Application\Service\TransactionInterface;

class Transaction implements TransactionInterface
{
    /**
     * @param callable $callable
     * @return mixed Returns $callable's execution result
     * @throws \Throwable
```

```
     */
    public function execute(callable $callable)
    {
        $transaction = new DbTransaction(); // Use your ORM here!

        $result = $callable();
        $transaction->commit();

        return $result;
    }
}
```

В свою очередь, диспетчер событий реализует одно действие – диспетчеризация:

```
namespace My\Ddd\Domain\Event;

interface EventDispatcherInterface
{
    /**
     * @param DomainEventInterface[] $events
     */
    public function dispatch(array $events);
}
```

Реализация может, например, использовать БД для сохранения событий и commitCallback для выбрасывания их в брокер сообщений после фиксации транзакции. Или напрямую обращаться к брокеру. Или использовать любой другой механизм, как из фреймворка, так и ваш собственный.

Гидрация объектов

В DDD программа реализует действия именно так, как они называются – в соответствии с бизнес-логикой и единым языком. Создание объекта – не исключение: это означает именно *создание нового объекта определённого типа*.

И если для объектов значений вызов `new Money(10, 'USD')` всегда будет приводить к одному и тому же результату – созданию объекта значения «10 долларов», то для сущностей, обладающих идентичностью каждый вызов конструктора может сопровождаться генерацией нового идентификатора. Этого можно избежать, передавая существующие значения, но вторую проблему это не решит: обычно создание объекта прямо в конструкторе порождает событие типа `Created`, обработка которого запускает определённые действия бизнес-логики в других частях системы.

Как же тогда быть при загрузке уже существующих агрегатов из БД репозиторием? Ведь реализация метода `get($id)` в репозитории теперь не может воспользоваться конструктором корня агрегата, чтобы не породить событие `Created`, не соответствующее действительности (это *загрузка существующего*, а не *создание нового*).

Для этого на помощь приходит подход, называемый **гидрация**: создание объекта путём настройки его текущего состояния (т.е. задание значений свойств) *без вызова конструктора*, исключительно по рефлексии. А если не было вызова конструктора, то и лишнее событие `Created` порождено не было.

Вы можете использовать, например, вот этот гидратор: https://github.com/samdark/hydrator

Список использованных источников

1. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. – 544 с.: ил. – (Серия «Библиотека программиста»).
2. Buenosvinos, C. et al. Domain-Driven Design in PHP. – Packt, 2017. – 387 p.