Гексагональная архитектура

26 окт 2015 в 11:15

На недавнем Laracon NYC я читал доклад о гексагональной архитектуре. Несмотря на то, что я получил позитивную реакцию слушателей, мне кажется, что остались люди, которые хотели бы получить чуть более полное представление о том, что это такое. Разумеется, с примерами. Это моя попытка расширить тот доклад.

1. Видео с доклада
2. Слайды

По моему мнению, данная архитектура является отличным примером того, как должна строиться структура приложения. Более того, когда я писал свои проекты на Laravel, я, даже не зная этого, частенько использовал идеи, заложенные в основе гексагональной архитектуры.

Являясь одним из вариантов слоеной архитектуры, гексагональная подразумевает разделение приложения на отдельные концептуальные слои, имеющие разную зону ответственности, а также регламентирует то, каким образом они связаны друг с другом. Разбираясь с этим типом архитектуры, мы также можем понять как, зачем, и почему при проектировании приложения используются интерфейсы.

Гексагональная архитектура, ни в коем случае **не новый** подход к разработке с применением фреймворков. Напротив, это всего лишь

обобщение «лучших практик» — практик новых и старых. Я обернул эти слова в кавычки, чтобы люди не воспринимали их совсем буквально. Лучшие практики, которые работают для меня, могут не работать для вас — все зависит от задачи и преследуемых целей.

Этот тип архитектуры придерживается классических идей, к которой приходят разработчики при проектировании приложений: отделение кода приложения от фреймворка. Пусть наше приложение формируется само по себе, а не на базе фреймворка, используя последний только как инструмент для решения каких-то задач нашего приложения.

Введение

Термин “Гексагональная архитектура” был введен (насколько мне известно) Алистером Коберном. Он отлично описал основные принципы этой архитектуры на своем сайте.

Основное назначение этой архитектуры:

> Позволяет взаимодействовать с приложением как пользователю, так и программам, автоматическим тестам, скриптам пакетной обработки. Также позволяет разрабатывать и тестировать приложение без каких-либо дополнительных устройств или баз данных.

Почему Гексагон

Несмотря на то, что архитектура называется гексагональной, что должно указывать на фигуру с определенным количеством граней, основной мыслью все же является то, что граней у этой фигуры много. Каждая грань представляет собой «порт» доступа к нашему приложению или же его связь с внешним миром.

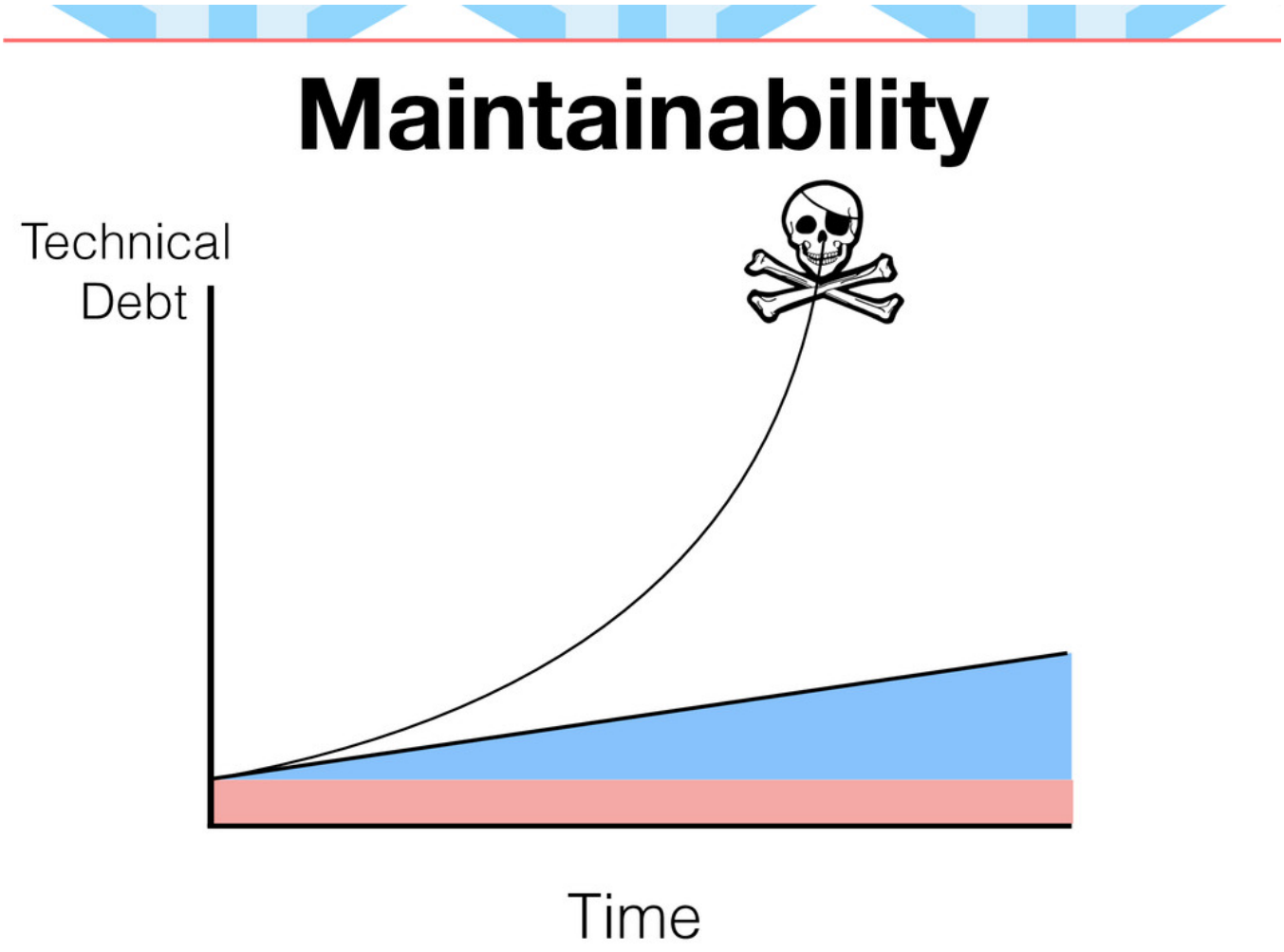
Порт может быть представлен как какой-либо проводник входящих запросов (или данных) к нашему приложению. Например, через HTTP-порт (запросы браузера, API) приходят HTTP запросы, которые конвертируются в команды для приложения. Похожим образом могут действовать различные менеджеры очередей или что угодно взаимодействующее с приложением по протоколу пересылки сообщений (например AMQP). Все это является лишь портами через которые мы можем попросить приложение сделать какие-то действия. Эти грани составляют множество «входящих портов». Другие порты могут быть использованы для доступа к данным со стороны приложения, например порт базы данных.

Архитектура

Почему мы вообще заговорили об архитектуре? А все потому, что мы хотим, чтобы наше приложение имело два показателя:

- высокая поддерживаемость
- низкий уровень технического долга

Собственно, оба эти показателя выражают одно и то же, мы хотим, чтобы с приложением было удобно работать, а также оно должно позволять быстро вносить изменения в будущем.



Поддерживаемость

Поддерживаемость определяется отсутствием (или минимизацией) технического долга. Поддерживаемым будет то приложение, при изменении которого мы будем получать минимально возможный

уровень технического долга.

Поддерживаемость это концепция, рассчитанная на большие промежутки времени. На ранних стадиях разработки с приложением легко работать — оно еще не до конца сформировано и разработчики формируют его исходя из своих первоначальных решений. На этом этапе добавление нового функционала или библиотек происходит легко и быстро.

Однако с течением времени, работать с приложением становится все сложней. Добавление новой функциональности может конфликтовать с уже имеющейся. Баги могут свидетельствовать о более общей проблеме, решение которой потребует больших изменений в коде (а также упрощения сложных участков кода).

Закладывая хорошую архитектуру на ранних этапах мы можем предотвратить эти проблемы.

Так все-таки, какого типа поддерживаемость мы хотим получить? Какие метрики могут сказать нам что мы имеем высокую поддерживаемость нашего приложения?

1. Изменения в одной части приложения должно затрагивать как можно меньше других частей;
2. Добавление нового функционала не должно требовать каких-то масштабных изменений в коде;
3. Организация нового интерфейса для взаимодействия с приложением должно приводить к минимально возможным

изменениям приложения;
4. Отладка должна включать как можно меньше временных решений и хаков;
5. Тестирование должно происходить относительно легко;
6. Так как не существует совершенного кода, слово «должно» стоит воспринимать как то, к чему нужно стремиться но не более. Мы стараемся сделать наше приложение более простым в поддержке, попытки же сделать его «идеальным» приведут к потерям времени и ментальной энергии.

> Если вы думаете что свернули с “правильного пути” при решении задачи, просто добейте ее до конца. Вернитесь к ней позже, или же оставьте в состоянии “работает не трогай”. В мире не существует приложений с идеально написанным исходным кодом.

Технический долг

Технический долг, это долг, который нам приходится выплачивать за наши (плохие) решения. Причем выплачиваем этот долг мы временем и чувством разочарования.

> Все приложения включают в себя начальный уровень технического долга. Мы вынуждены работать в рамках и с учетом ограничений выбранных нами механизмов хранения данных, языков программирования, фреймворков, команд и организаций!

Плохие архитектурные решения принятые на ранних этапах разработки в совокупности выльются нам все большими и большими проблемами.

Каждое плохое решение как правило приводит к костылям и хакам. Причем то, что решение плохое, не всегда очевидно — мы можем просто сделать класс, который «делает слишком много», или случайно смешаем решения нескольких проблем.

Незначительные, но плохие решения, принятые во время разработки, также могут привести к проблемам. К счастью, обычно это не приводит к масштабным проблемам, к которым могут привести ошибки проектирования на ранних этапах проектирования. Хороший фундамент уменьшает скорость роста технического долга!

Итак, мы хотим минимизировать количество плохих решений, особенно на ранних стадиях проекта.

> Мы обсуждаем архитектуру для того, чтобы мы могли сосредоточиться на повышении поддерживаемости и уменьшении технического долга.

Как мы можем сделать наше приложение поддерживаемым?

Мы упрощаем внесение изменений в приложение.

Что мы можем сделать чтобы упростить внесение изменений в приложение? Мы можем определить те части приложения, которые могут измениться и отделить их от того, что останется неизменным.

Мы вернемся к этому утверждению еще пару раз.

Интерфейсы и реализация

Давайте ненадолго отвлечемся и обсудим кое-что простое (как это может показаться) из мира ООП: Интерфейсы.

> Не все языки (например JavaScript, Python и Ruby) имеют явные интерфейсы, однако с концептуальной точки зрения цели, которые преследует использование оных могут быть легко достигнуты в этих языках.

Вы можете понимать под интерфейсом некий контракт, который регламентирует что нужно приложению. Если приложение может или должно содержать несколько реализаций, то мы можем применить интерфейсы.

Другими словами мы используем интерфейсы всякий раз, когда мы планируем несколько *реализаций* одного *интерфейса*.

Например если у нашего приложения имеется возможность отправлять уведомления пользователю, то мы можем сделать SES

нотификатор, использующий Amazon SES, Mandrill нотификатор, использующий Mandrill, или же другие реализации использующие различные сервисы для отправки почты.

> Интерфейс гарантирует, что конкретные методы доступны для использования в нашем приложении, вне зависимости от реализации, лежащей в основе.

Например интерфейс нашего нотификатора может выглядеть так:

```
interface Notifier {

    public function notify(Message $message);
}
```

Мы знаем, что любая реализация этого интерфейса **должна** содержать метод `notify`. Это позволяет нам определить зависимость от этого интерфейса в других частях нашего приложения.

Приложению без разницы, какую реализацию оно использует. Ему важно, что у нас есть метод `notify` и мы можем его использовать.

```
class SomeClass {

    public function __construct(Notifier $notifier)
    {
        $this->notifier = $notifier;
    }
```

```
    public function doStuff()
    {
        $to = 'some@email.com';
        $body = 'This is a message';
        $message = new Message($to, $body);

        $this->notifier->notify($message);
    }
}
```

В этом примере реализация класса `SomeClass` не зависит от конкретной реализации, а только требует наличие подкласса `Notifier`. Это позволяет нам использовать SES, Mandrill или любую другую реализацию.

Таким образом, интерфейсы являются удобным инструментом для увеличения поддерживаемости вашего приложения. Интерфейсы позволяют нам легко менять способ отправки уведомлений — для этого нам просто нужно добавить еще одну реализацию и все.

```
class SesNotifier implements Notifier {

    public function __construct(SesClient $client)
    {
        $this->client = $client;
    }

    public function notify(Message $message)
    {
        $this->client->send([
            'to' => $message->to,
            'body' => $message->body]);
```

```
    }
}
```

В примере выше мы использовали реализацию, использующую сервис от Amazon, называющийся Simple Email Service (SES). Но что если мы все-таки захотим использовать Mandrill для отправки нотификаций? А если мы хотим отправлять уведомления по SMS через Twilio?

Как мы уже показали ранее, мы можем легко добавить еще одну реализацию и переключаться между ними по нашей прихоти.

```
// Используя SES Notifier
$sesNotifier = new SesNotifier(...);
$someClass = new SomeClass($sesNotifier);

// Или мы можем использовать MandrillNotifier

$mandrillNotifier = new MandrillNotifier(...);
$someClass = new SomeClass($mandrillNotifier);

// Это все равно будет работать вне зависимости от реализации, которую мы исп
$someClass->doStuff();
```

По такому же принципу наши фреймворки используют интерфейсы. По сути фреймворки представляют для нас пользу именно потому, что могут содержать столько реализаций, сколько разработчикам необходимо. Например, различные SQL сервера, системы отправки email-ов, драйверы для кеширования и другие сервисы.

Фреймворки используют интерфейсы потому, что это позволяет увеличить поддерживаемость фреймворков — проще добавлять новые фичи, проще для нас расширять фреймворк в зависимости от наших нужд.

Использование интерфейсов помогает нам правильно **инкапсулировать изменения**. Мы можем просто добавить ту реализацию, которая нам сейчас нужна!

Идем дальше

А что если мы вдруг захотим добавить дополнительную функциональность в рамках отдельной реализации (а может и всем)? Например у нас может возникнуть необходимость логировать работу реализации нашего SES нотификатора, например для отладки какой-то проблемы, которая у нас периодически возникает.

Наиболее очевидный способ, конечно, это поправить код прямо в реализации:

```
class SesNotifier implements Notifier {

    public function __construct(SesClient $client, Logger $logger)
    {
        $this->logger = $logger;
        $this->client = $client;
    }
```

```
    public function notify(Message $message)
    {
        $this->logger->logMessage($message);
        $this->client->send([...]);
    }
}
```

Использовать логирование прямо в конкретной реализации может быть нормальным решением, но теперь эта реализация занимается двумя вещами вместо одной — мы смешиваем обязанности. Более того, что если нам нужно добавить логирование во все реализации нашего нотификатора? В итоге мы получим похожий код для каждой реализации, что противоречит принципу DRY. Если нам нужно поменять способ логирования, то нам придется изменить код всех реализаций. Есть ли более простой способ добавить эту функциональность так, чтобы сохранить код поддерживаемым? Да!

> Узнаете ли вы какие-либо из принципов SOLID, которые затрагивает данная глава?

Чтобы сделать код чище, мы можем воспользоваться одним из моих любимых шаблонов проектирования — Декоратор. Данный шаблон использует интерфейсы для того чтобы «обернуть» нашу реализацию для добавление новой функциональности. Давайте рассмотрим пример.

```
// Добавляем логирование при помощи декорации
class NotifierLogger implements Notifier {

    public function __construct(Notifier $next, Logger $logger)
```

```
    {
        $this->next = $next;
        $this->logger = $logger;
    }

    public function notify(Message $message)
    {
        $this->logger->logMessage($message);
        return $this->next->notify($message);
    }
}
```

Так же как и другие наши реализации нотификатора, класс `NotifierLogger` реализует интерфейс `Notifier`. Однако, как вы возможно заметили, он и не думает слать кому-то уведомления. Вместо этого, он принимает в конструкторе другую реализацию интерфейса Notifier. Когда мы просим нотификатор кого-то о чем-то уведомить, наш `NotifierLogger` добавит в лог данные сообщения и затем попросит передать уведомление настоящую реализацию нашего нотификатора.

Как вы видите, наш декоратор логирует сообщения и передает их далее в нотификатор, чтобы тот на самом деле что-нибудь послал. Вы также можете развернуть порядок так, что логирование будет выполнено после отправки сообщения. Таким образом мы сможем добавить в лог не только сообщение, которое мы собирались отправить, но и результат отправки.

Итак, `NotifierLogger` декорирует наш нотификатор, привнося функционал логирования.

Удобство тут заключается в том, что нашему клиентскому коду (в нашем случае `SomeClass` из примера выше) нет дела до того, что мы передали ему декорированный объект. Декоратор реализует тот же интерфейс, от которого зависит наш `SomeClass`.

Также мы можем объединить несколько декораторов в одну цепочку. Например, мы можем обернуть email нотификатор реализацией для работы с SMS, чтобы дублировать сообщения. В этом случае мы добавляем дополнительную функциональность (SMS) поверх отправки email-ов.

Как показывает пример с логгером, мы не ограничены добавлением конкретных реализаций нотификатора. Например, мы можем организовать обновление базы данных, или сбор каких-то метрик. Возможности безграничны!

Теперь у нас есть возможность добавлять дополнительное поведение, при этом сохраняя зону ответственности каждого класса не перегружая его лишней работой. Также мы можем свободно добавлять дополнительные реализации функционала, что дает нам дополнительную гибкость. Менять такой код становится намного проще!

> Декоратор является лишь одним из многих шаблонов проектирования, который использует интерфейсы для инкапсуляции изменений. На самом деле, почти все классические шаблоны проектирования используют интерфейсы.

> Более того, почти все шаблоны проектирования призваны упростить внесение изменений. И это не совпадение. Изучение шаблонов проектирования (а так же где и когда их применять) является довольно важным шагом на пути к принятию хороших архитектурных решений. В качестве подспорья для изучения шаблонов проектирования я рекомендую книгу Head First Design Patterns.

Повторю: Интерфейсы это основной способ инкапсуляции изменений. Мы можем добавить функционал посредствам создания новой реализации, или же мы можем добавить поведение в существующую реализацию. И все это не будет затрагивать весь остальной код!

Обеспечив хорошую инкапсуляцию, функциональность может быть проще подвергнута изменениям. Упрощение изменения кода повышает поддерживаемость приложения (их проще менять) и уменьшает технический долг (мы инвестируем меньше времени, для того чтобы внести изменения).

Пожалуй про интерфейсы достаточно. Надеюсь это помогло прояснить некоторым зачем нужны интерфейсы, а также распробовать некоторые шаблоны проектирования, которые мы использовали, чтобы повысить поддерживаемость приложения.

Порты и Адаптеры

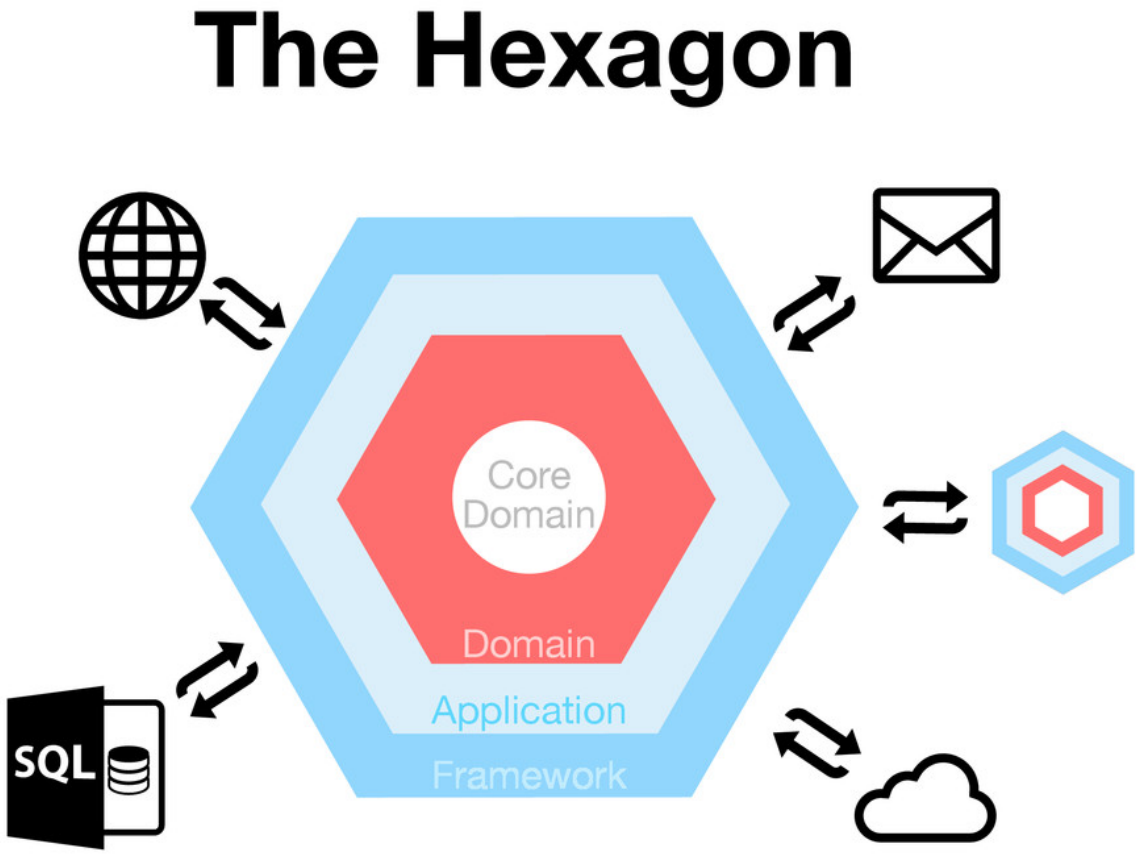
Что ж, наконец-то мы можем приступить к знакомству с гексагональной архитектурой.

Гексагональная архитектура, это слоеная архитектура, также иногда называемая архитектурой портов и адаптеров. Называют ее так потому, что в рамках этой архитектуры имеется концепция различных портов, которые могут быть использованы (адаптированы) для использования с другими слоями.

Например, наш фреймворк использует «порты» для работы с SQL под любое количество различных SQL серверов, с которыми может работать наше приложение. Точно так же мы можем реализовать интерфейсы для каких-то ключевых вещей, и реализовать их в других слоях приложения. Это позволяет нам иметь несколько реализаций этих интерфейсов в зависимости от наших потребностей или для нужд тестирования. Интерфейсы будут нашим главным средством для снижения связанности кода между слоями.

> Для частей приложения, которые могут меняться, создайте интерфейс. Таким образом мы инкапсулируем изменения. Мы можем создать новую реализацию или добавить дополнительный функционал поверх существующего.

Перед тем как познакомиться поближе с концепцией портов и адаптеров, давайте рассмотрим слои, имеющиеся в рамках Гексагональной Архитектуры.



Слои: Код

Как я уже сказал, в рамках гексагональной архитектуры, приложение разбивается на несколько слоев.

Целью такого представления приложения в виде отдельных слоев, является возможность разделить приложение на разные области ответственности.

Код в пределах слоев (и на их границах) должен описывать то, как должно происходить их взаимодействие. Так как слои действуют как

порты и адаптеры для других слоев, окружающих их, важно иметь правила взаимодействия между ними.

Слои взаимодействуют друг с другом используя интерфейсы (порты) и реализации этих интерфейсов (адаптеры).

Каждый слой состоит из двух элементов:

Под **кодом** тут мы понимаем именно то, что вы могли подумать. Это просто какой-то код, при помощи которого мы делаем какие-то вещи. Довольно часто этот код выступает только как адаптер для других слоев, но это может быть и просто какой-то полезный код (бизнес логика, какие-то сервисы).

Каждый слой также имеет **границу**, отделяющую его от соседних слоев. На границе мы можем найти наши «порты». Как мы уже говорили, порты являются ни чем иным, как интерфейсами, которые указывают другим слоям на то, каким образом будет происходить взаимодействие. Мы рассмотрим вопрос взаимодействия слоев чуть позже, для начала нам надо познакомиться с имеющимися у нас слоями.

Слой предметной области (Domain Layer)

В самом центре нашего приложения расположен слой предметной области. Этот слой содержит в себе реализацию бизнес логики и

определяет, как внешние слои могут с ней взаимодействовать.

Бизнес логика это сердце нашего приложения. Ее можно описать словом “устав” — правила, которым должен подчиняться ваш код.

Слой предметной области, и бизнес логика, реализованная в нем, определяют поведение и ограничения вашего приложения. Это то, что отличает ваше приложение от других, то что придает приложению ценность.

Если ваше приложение содержит сложную бизнес логику, различные варианты поведения, то у вас получится богатый слой предметной области. Если же ваше приложение является небольшой надстройкой над базой данных (а многие приложения и являются таковыми), то этот слой будет «тоньше».

В добавок к бизнес логике (ядро предметной области или core domain), в слое предметной области часто можно встретить дополнительную логику, например события предметной области (domain events, события, которые выбрасываются в ключевых точках бизнес логики) и “сценарии использования” или use-cases (определение того, что должно делать наше приложение).

Что содержит слой предметной области, является темой для целой книги (или серии книг) — особенно если вы интересуетесь предметно-ориентированным проектированием (DDD). Эта методология намного более детально описывает способы, при которых мы можем реализовать приложение как можно ближе к

терминам предметной области, а стало быть бизнес процессам, которые мы хотим перенести в код.

Рассмотрим небольшой пример “основной” логики из ядра предметной области:

```
<?php  namespace Hex\Tickets;

class Ticket extends Model {

   public function assignStaffer(Staffer $staffer)
   {
       if( ! $staffer->categories->contains( $this->category ) )
       {
           throw new DomainException("Staffer can't be assigned to ".$this->c
       }

       $this->staffer()->associate($staffer); // Set Relationship

       return $this;
   }

   public function setCategory(Category $category)
   {
       if( $this->staffer instanceof Staffer &&
           ! $this->staffer->categories->contains( $category ) )
       {
           // открепляем сотрудника, в случае
           // если он не может быть закреплен за заявкой из этой категории
           $this->staffer = null;
       }

       $this->category()->associate($category); // устанавливаем отношения об

       return $this;
   }
```

```
}
```

В примере выше мы можем увидеть **ограничение** в методе `assignStaffer`. Если сотрудник (Stuffer) не закреплен за той же категорией (Category), что и наша заявка (Ticket), мы бросаем исключение.

Также мы можем видеть определенное **поведение**. В случае если нам нужно поменять категорию заявки, за которой уже закреплен какой-то сотрудник, мы снова пытаемся закрепить ее за ним. Если это не выходит, то мы просто открепляем завяку от сотрудника. Мы не бросаем исключение, вместо этого мы даем возможность закрепить за заявкой другого сотрудника при смене категории.

Мы рассмотрели оба примера выполнения бизнес-логики. В одном сценарии, мы добавили ограничение, в случае не соблюдения которого мы бросаем исключение. В другом — предоставили определенное поведение — как только мы поменяем категорию, мы должны оставить возможность закрепить заявку за тем сотрудником, который может обрабатывать заявки из новой категории.

Как мы уже говорили, слой предметной области может также содержать вспомогательную бизнес логику:

```
class RegisterUserCommand {
```

```
    protected $email;

    protected $password;

    public function __construct($email, $password)
    {
        // сеттеры для email/password
    }

    public function getEmail() { ... } // возвращаем $email

    public function getPassword() { ... }   // возвращаем $password
}

class UserCreatedEvent {

    public function __construct(User $user) { ... }
}
```

Выше мы имеем вспомогательную (но очень важную) логику предметной области. Первый класс является своего рода командой (сценарий использования в терминологии UML), который определяет каким образом наше приложение может быть использовано. В нашем случае данная команда просто берет необходимые данные для того чтобы зарегистрировать нового пользователя. Как именно? Мы вернемся к этому чуть позже.

Второй класс, это пример события предметной области (Domain Event), которое наше приложение может пустить на обработку после того, как создаст пользователя. Важно отметить, что те вещи, которые могут произойти в рамках нашей предметной области, относятся к слою предметной области. Это не системные события, вроде `pre-dispatch` хуков, которые часто используются

фреймворками для расширения их возможностей.

Слой приложения (Application Layer)

Сразу за слоем предметной области сидит наш слой приложений. Этот слой занимается исключительно оркестрацией действий, производимых над сущностями из слоя предметной области. Также этот слой является адаптером запросов из слоя фреймворка и отделяет его от слоя предметной области.

Например этот слой может содержать класс-обработчик, который выполняет какой-то юз-кейс. Этот класс-обработчик принимает входящие данные, пришедшие к нему из слоя фреймворка, и будет выполнять над ними какие-то действия, которые требуются для выполнения нашего юз-кейса.

Также он может отправлять на обработку события (domain events), которые произошли в слое предметной области.

Это внешний слой кода, составляющего наше приложение.

Конечно же, вы могли заметить что снаружи слоя приложения есть еще “слой фреймворка”. Этот слой содержит вспомогательный код нашего приложения (возможно, принимающий HTTP запросы, или отправляющий email-ы), но он **не** является самим приложением.

Слой фреймворка (Framework Layer)

Наше приложение укутано в слой фреймворка (его также называют инфраструктурным слоем, infrastructure layer). Как уже было сказано выше, этот слой содержит код, который использует ваше приложение, но при этом он не является сам по себе частью приложения. Обычно этот слой представлен вашим фреймворком, но конечно же может включать в себя любые сторонние библиотеки, SDK и любой другой код. Вспомните все библиотеки, которые вы подключили через composer (предположим что мы все же пишем на PHP). Они не являются частью вашего фреймворка, но все же они объединены в один слой. Весь этот слой необходим лишь для одного — выполнять различные задачи для удовлетворения потребностей нашего приложения.

В этом слое реализуются сервисы, интерфейсы которых объявлены во внутренних слоях. Например, тут может быть реализован интерфейс `Notificator` для отправки уведомлений по email или через SMS. Наше приложение просто хочет отправить уведомления пользователям, ему не нужно знать как именно это происходит (через email или через SMS или еще каким-то образом).

```
class SesEmailNotifier implements Notifier {

    public function __construct(SesClient $client) { ... }

    public function notify(Message $message)
    {
        $this->client->sendEmail([ ... ]); // Send email with SES particulars
```

```
    }
}
```

Еще одним примером является диспетчер событий (*event dispatcher*). В слое уровня фреймворка может быть реализован интерфейс оного, объявленный в слое приложения. Опять же, приложению может понадобиться обрабатывать какие-либо события, но зачем писать свой диспетчер событий? Скорее всего ваш фреймворк уже имеет готовую реализацию. Ну или в крайнем случае мы можем подключить готовую библиотеку, и использовать ее как основу для реализации нашего интерфейса диспетчера событий.

Наш слой уровня фреймворка также может выступать в роли адаптера HTTP запросов к нашему приложению. Например, мы можем возложить на его плечи прием HTTP запросов, сбор входящих данных и маршутизацию запроса/данных к соответствующему контроллеру. Далее контроллер уже может запустить конкретный сценарий использования из слоя уровня приложения (вместо того, чтобы обработать все прямо в контроллере).

Таким образом данный слой отделяет наше приложение от всех внешних запросов (например, сделанных из браузера). Именно этот слой является адаптером запросов приложения.

Взаимодействие слоев: границы

Теперь, когда мы разобрались с тем, что находится в том или ном слое, давайте поговорим о том, как они взаимодействуют друг с другом.

Как уже было сказано выше, каждый слой регламентирует то, каким образом другим слоям можно с ним взаимодействовать. Если конкретнее, то каждый слой ответственен за определение того, каким образом с ним будет взаимодействовать каждый следующий внешний слой.

Инструментом для этого нам послужат интерфейсы. **На границе каждого слоя мы найдем интерфейсы**. Эти интерфейсы являются портами для следующих слоев, в которых будут реализованы адаптеры.

Мы уже рассмотрели как это приблизительно работает в примерах с нотификаторами и диспетчером событий.

Например, слой приложения будет содержать реализацию интерфейсов (адаптеры к портам), объявленных в слое предметной области. Также этот слой будет содержать код для других вещей, специфичных для нашего приложения.

Давайте пройдем через границы каждого слоя, чтобы разобраться как они связаны друг с другом.

Слой предметной области

На границе слоя предметной области мы найдем средства, предоставляемые им для общения внешнему слою (слою приложения).

Например, наш слой предметной области может содержать команды (сценарии использования). Выше мы уже видели пример команды `RegisterUserCommand` (регистрация пользователя). Эта команда довольно простая, мы можем воспринимать ее как простой DTO (Data Transfer Object).

Итак, слой предметной области определяет сценарии использования, но на этом все. Его задачей является сказать внешнему слою “как ты можешь меня использовать”. Ведь это забота слоя уровня приложения управлять бизнес логикой для реализации какой-то задачи. Итак, у нас есть способ общения через границы. Слой предметной области объявляет каким образом он может быть использован, а уже слой приложения использует эти определения (как составляющую) для выполнения объявленного сценария использования.

Однако нашему слою приложения еще нужно рассказать, каким образом он может обработать команду для регистрации пользователя. Так как для этого этим слоям надо общаться, “на границе” слоя предметной области мы разместим следующий интерфейс:

```
interface CommandBus {

    public function execute($command);
}
```

При помощи этого интерфейса мы рассказали слою приложения, как “выполнять” команды используя шину команд (`CommandBus`). Шина команд устроена весьма просто — она просто должна содержать метод `execute`, в котором будет содержаться реализация обработки команды.

Итак, мы добавили интерфейс `CommandBus` в наш слой предметной области, и теперь мы сможем реализовать его в слое приложения. Повторю, интерфейс это порт, а его реализация это адаптер к этому порту.

Итак, что же мы сделали:

- Мы узнали что наши команды могут обрабатываться различными способами
- Из предыдущего пункта следует, что у нас может быть несколько реализаций шины команд
- Как мы выяснили ранее, слой уровня приложения решает, как будет использован наш слой предметной области, а следовательно реализацию (или же множество реализаций) шины команд логично разместить там (поскольку именно там определяется как мы будем обрабатывать эти команды, определенные в слое предметной области).

- Мы отделили слой предметной области от слоя приложения при помощи интерфейсов. Наша шина команд имеет определенный способ запуска команд на выполнение.

Слой приложения

Итак, в данном слое мы должны реализовать шину команд. Находиться она будет в центре данного слоя, как и другие реализации (адаптеры) к другим слоям. Нас же сейчас интересуют границы слоев, так что мы вернемся к ней чуть позже.

Слою приложения также нужно как-то взаимодействовать с внешними слоями. Например, как быть, если нам понадобится отправлять уведомления при обработке команд? Как уже говорилось ранее, все что для этого нужно есть в слое уровня фреймворка. Ваш фреймворк скорее всего уже содержит инструменты для отправки email-ов, или же мы можем подключить библиотеку для отправки SMS. Именно в слое фреймворка удобнее всего разместить реализацию сервисов по отправке уведомлений.

Итак, у нас проглядывается связь между двумя слоями. Как вы думаете что мы сделаем? Правильно! Добавим еще один интерфейс:

```
interface Notifier {

    public function notify(Message $message);
}
```

Слой приложения определяет как он будет взаимодействовать со слоем фреймворка. Даже больше, он определяет то, как будет использоваться слой фреймворка, при этом не выстраивая прямой зависимости от него. Интерфейсы (порты) и их реализации (адаптеры) позволяют нам с легкостью сменить один адаптер на другой. Таким образом мы не привязываемся намертво к слою фреймворка.

Интерфейсы определенные “на границе” слоя приложения регламентируют как он будет взаимодействовать со слоем фреймворка.

> Это только идея, не стоит воспринимать ее как какое-то конкретное правило. Если у вас возникнет вопрос “а что если мне понадобится использовать стороннюю библиотека из слоя фреймворка в слое предметной области”, то ничего страшного.
>
> Если так надо, то просто добавьте интерфейс и реализуйте его где надо и используя ту библиотеку, которую требуется использовать. Наша цель все же заставить наш код работать. Мы недалеко уедем, если мы будем переживать всякий раз как нарушаем “правила” придуманные каким-то чуваком или чувихой в интернете.
>
> Основной идеей тут служит разделение обязанностей (подсказка: когда мы определяем интерфейс, мы делаем именно это), так что мы сможем сменить/поменять функциональную составляющую позже. Не стоит воспринимать написанное здесь как какие-то

> догматы. Не существует “не правильных” подходов. Повторюсь: нет никаких неправильных подходов. Есть только различные варианты того, как можно прострелить себе ногу.
>
> Если вы все еще переживаете по поводу использования сторонних библиотек в слое предметной области, то можете почитать об этом подробнее по приведенной ссылке.

Слой фреймворка

Пока-что мы рассмотрели границу слоев предметной области и приложения. Оба этих слоя взаимодействуют со слоями, которые находятся под нашим контролем. Слой предметной области взаимодействует со слоем приложения. Слой приложения — со слоем фреймворка. А с кем взаимодействует слой фреймворка?

С внешним миром, конечно! С миром, заполненным различными протоколами. Как правило это какие-то протоколы, основанные на TCP (такие как HTTP/HTTPS). Определенно слой фреймворка содержит много кода (все библиотеки которые мы используем). Не стоит забывать про код написанный нами, например контроллеры и реализации интерфейсов объявленных во внутренних слоях.

Что именно находится на границе между слоем фреймворка и внешним миром? Ну конечно же интерфейсы (и реализации этих интерфейсов)!

Большинство фреймворков уже содержат код, который отвечает за общение с внешним миром. Например, реализация обработки HTTP запросов, различные реализации SQL драйверов, транспорты для отправки email и т.д.

К счастью, по большому счету нам нет нужды заботиться об организации границы между этим слоем и внешним миром. Это обязанность нашего фреймворка, чьи авторы великодушно позаботились об этом.

По сути это основная цель фреймворков. Они предоставляют нам средства для взаимодействия с внешним миром, при этом снимая с нас необходимость снова и снова писать весь этот код.

В большинстве случаем нам не нужно добавлять что-то на границе слоя фреймворка, однако такие случаи бывают. Например, если мы делаем API для нашего приложения, у нас добавится забот на уровне HTTP. Обычно это реализация CORS, HTTP кеширование, HATEOAS и другие специфические для обработки HTTP запросов проблемы. Для нашего приложения все эти вещи могут быть важными, но не для слоя предметной области или даже слоя приложения.

Сценарии использования/Команды

Ранее в статье я упомянул “сценарии использования” и “команды”. Пришло время разобраться с ними.

В рамках гексагональной архитектуры рассматривается не только взаимодействие слоев на микро уровне (интерфейсы, реализация адаптеров к портам). В ней также рассматривается концепция границы приложения на макро-уровне.

> Эта граница отделяет все наше приложения от всего остального (и от фреймворка и от взаимодействия с внешним миром).

Мы можем строго регламентировать то, каким образом внешний мир сможет взаимодействовать с приложением. Для этого мы должны создать “сценарии использования” (можно также назвать их “командами”). Обычно это какие-то классы, чьи имена содержат описание действий, которые может совершить приложение. Например, наша команда `RegisterUserCommand` определяет, что приложение может зарегистрировать пользователя. Команда `UpdateBillingCommand`, судя по названию, определяет возможность обновить информацию по платежам пользователя.

> Сценарий использования (команда), это определение того, как мы можем использовать наше приложение.

Определяя сценарий использования мы можем наблюдать интересные побочные эффекты. Например, мы можем увидеть, каким образом приложение “хочет”, чтобы мы с ним взаимодействовали. Он строго следует бизнес логике, которая заложена нашим приложением. Сценарии использования также полезны для добавления ясности между членами команды разработчиков. Мы можем планировать эти сценарии наперед, или

добавлять их по необходимости, но добавить какую-то странную логику вне наших сценариев использования становится уже сложнее. Мы сразу увидим что эта логика плохо соотносится с бизнес логикой приложения.

Как мы определяем сценарий использования?

Мы уже видели некоторые примеры — для этого мы создавали какой-то объект, представляющий конкретный сценарий использования. Назовем такой объект “командой” (Command). Эти команды затем могут быть обработаны “шиной команд” (Command Bus) нашего приложения, которая попросит конкретный “обработчик” (Handler). Обработчик в свою очередь уже будет заниматься оркестрацией процесса выполнения сценария использования.

Итак, для обработки команд у нас служат три действующих лица:

- Команда
- Шина команд
- Обработчик

Задачей шины команд является принять команду на выполнение в методе `execute`. Затем шина команд должна, за счет какой-то внутренней логики, найти и инициализировать обработчик нашей команды. И наконец, мы вызываем метод `handle` обработчика, в

котором происходит управление обработкой команды.

```
class SimpleCommandBus implements CommandBus {

    // остальные методы убраны для краткости

    public function execute($command)
    {
        return $this->resolveHandler($command)->handle($command);
    }
}
```

Следует заметить, что мы помещаем в обработчике координирующую логику, которую вы можете часто видеть в контроллерах. Это хорошо, таким образом мы отделяем приложение от слоя фреймворка, что и позволяет нам получить большую защиту от изменений в оном (упростить поддержку приложения). Также это позволяет нам запускать один и тот же код в разных контекстах (CLI, API-вызовы и т.д.).

Возможность переиспользовать логику в разных контекстах (например web, http api, cli, воркеры очередей) и является основным преимуществом использования сценариев использования.

К примеру, код, реализующий регистрацию пользователя через WEB, HTTP или CLI интерфейсы может быть с большего одинаковым.

```
public function handleSomeRequest()
```

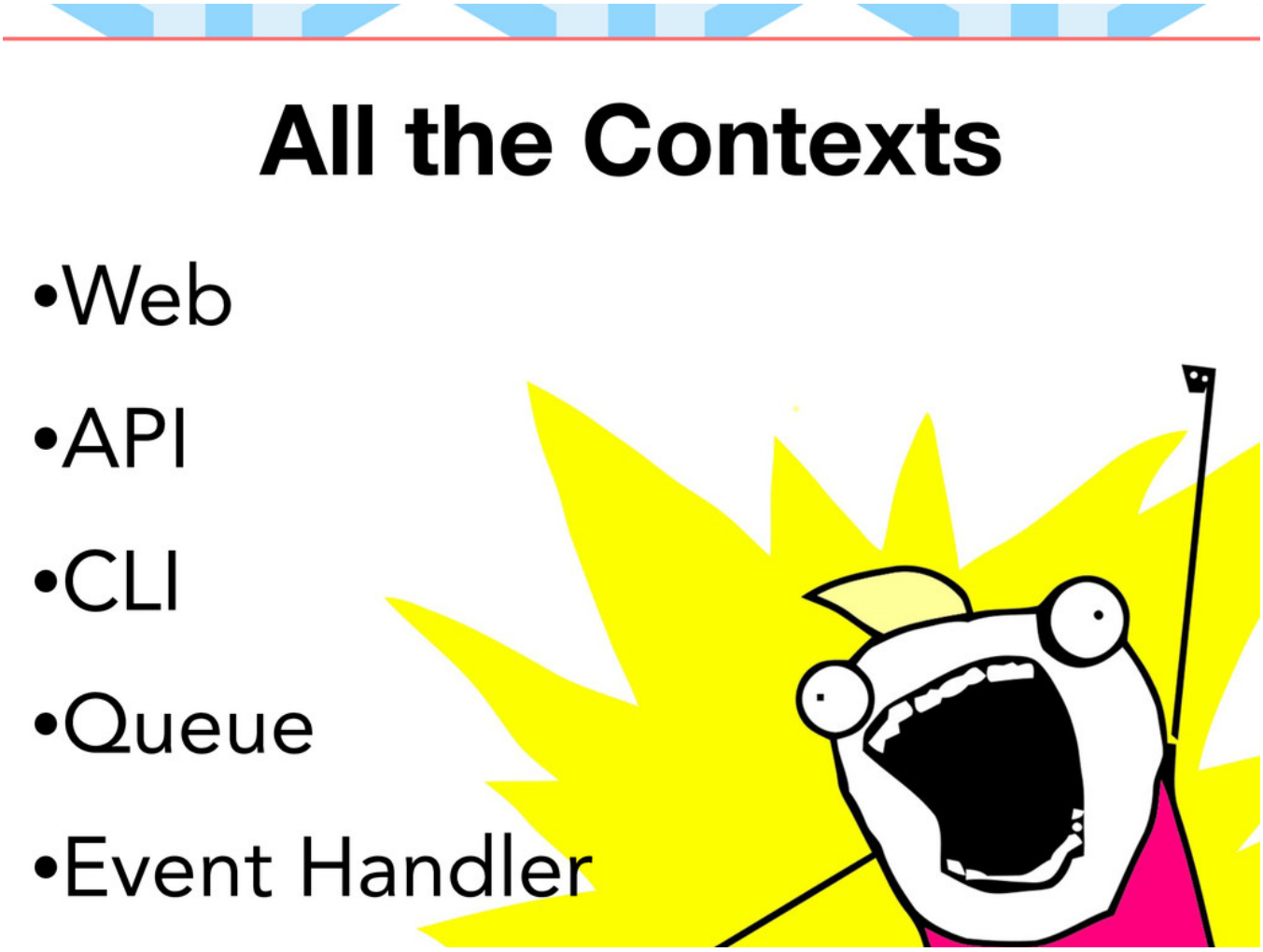
```
{
    try {
        $registerUserCommand = new RegisterUserCommand(
            $this->request->username, $this->request->email, $this->request->
        );

        $result = $this->commandBus->execute($registerUserCommand);

        return Redirect::to('/account')->with([ 'message' => 'success' ]);
    } catch( \Exception $e )
    {
        return Redirect::to('/user/add')->with( [ 'message' => $e->getMessage
    }
}
```

Единственным отличием между различными контекстами является то, каким образом пользователь вводит данные и передает их в команду, а также способ обработки ошибок. По большому счету, все это забота фреймворка. Нашему приложению не важно, используют ли его через WEB-интерфейс, через HTTP API или еще как-то.

Резюмируя, именно так проявляется потенциал сценариев использования. Мы можем использовать их в любом контексте, в котором может быть использовано наше приложения (HTTP, CLI, API, AMQP или другой протокол очереди сообщений)! В добавок к этому мы создали прочную границу между фреймворком и нашим приложением. По сути наше приложение может быть использовано отдельно от нашего фреймворка (что положительно сказывается при тестировании).

У нас все еще может быть потребность в фреймворке для

реализации каких-то задач приложения. Например, валидация, маршрутизация событий, доставка электронной почты и другие задачи, которые решает для нас фреймворк. Граница приложения построенная на сценариях использования, это лишь один из аспектов гексагональной архитектуры.



Сценарии использования служат для большего разделения приложения от фреймворка. Это дает нам определенную защиту от изменений в фреймворке (при обновлении и т.д.) а так же, что не маловажно, упрощает тестирование.

Если забираться в крайности, вы в принципе можете сменить

фреймворк без необходимости полностью переписывать приложение. Однако, я считаю что это **самый крайний случай который только можно придумать**. Это не реалистичный сценарий, как минимум потому что в этом мало смысла. Все что нам нужно, так это удобнее работать с приложением а не потворствовать каким-то метрикам связанным с редкими случаями.

Примеры команд, обработчика команд и шины команд

Для начала, нашему приложению нужны команды. Затем, приложению понадобится шина, которая будет выполнять команды. И наконец, нам необходим обработчик, который будет заниматься оркестрацией процесса выполнения команды.

Сами по себе команды это довольно простая штука. Они просто должны быть. Целью их существования является определение того, как может быть использовано наше приложение. Данные, которые они требуют, говорят нам о том, какие данные нужны, чтобы совершить какое-то действие. Так что нет необходимости создавать интерфейс команды. Это просто название (описание того что мы хотим сделать) и DTO (Data Transfer Object).

```
class RegisterUserCommand {

    public function __construct($username, $email, $password)
    {
        // устанавливаем данные тут
    }
```

```
    // геттеры здесь
}
```

Итак, наша команда для регистрации нового пользователя довольно простая. С ее помощью мы определили один из способов использования нашего приложения.

Наш обработчик будет чуть посложнее. Они связаны с командами, так как для их работы нужны данные, которые содержит команда. В этом месте у нас наблюдается тесное связывание. Изменения в бизнес логике могут привести к изменению обработчика, и как следствие, к изменению команды. И с этим все “ОК”, так как и обработчики команд, и команды завязаны на бизнес логику.

Если команды это простые DTO (содержащий различные данные), обработчики содержат в себе поведение, которое использует шина команд. Обработчики, являясь частью слоя приложения, руководят использованием сущностей из слоя предметной области с целью выполнения команд.

Для того чтобы наша шина команд имела возможность выполнить любую команду, у нее должна быть возможность вызвать любой обработчик. По этому мы должны реализовать интерфейс обработчика, чтобы нашей шине команд всегда было с кем работать.

```
interface Handler {
```

```
    public function handle($command);
}
```

Обработчик должен иметь метод `handle`, но выбор команды, которую он обрабатывает, мы оставляем на совесть конкретной реализации этого интерфейса. Давайте посмотрим на то, как может выглядеть наш обработчик команды `RegisterUserCommand`:

```
class RegisterUserHandler {

    public function handle($command)
    {
        $user = new User;
        $user->username = $command->username;
        $user->email = $command->email;
        $user->password = $this->auth->hash($command->password);

        $user->save();

        $this->dispatcher->dispatch( $user->flushEvents() );

        // Подумайте о том, чтобы также вернуть DTO,
        // вместо того, чтобы возвращать класс содержащий поведение
        // Таким образом наш слой "представления" (вне зависимости от контекс
        // не сможет случайно повлиять на наше приложения.
        // Он просто будет читать данные из результата
        return $user->toArray();
    }
}
```

Здесь мы видим, как наш обработчик руководит использованием сущностей предметной области, включая то, как мы присваиваем

данные, сохраняем их и отправляем события на обработку (если конечно в наших сущностях произошли какие-то события).

> Также как в нашем примере с интерфейсами, где мы при помощи декоратора добавляли дополнительное поведение в наш сервис уведомлений, давайте подумаем, какое дополнительное поведение мы можем прикрутить к нашей шине команд или к обработчику.

Напоследок, давайте рассмотрим устройство пожалуй самого интересного участника — шину команд (Command Bus).

У нашей шины команд может быть множество реализаций. Например, мы можем реализовать синхронную шину команд (запускающую команды по мере поступления). Или мы можем организовать пакетную обработку команд, складывая их в очередь и выполняя их скопом, когда их наберется достаточно. А может быть мы вообще захотим организовать асинхронную шину команд, которая будет публиковать наши команды в менеджере очередей для их дальнейшего выполнения вне текущего запроса пользователя.

Поскольку реализаций может быть множество, давайте создадим интерфейс нашей шины команд:

```
interface CommandBus {

    public function execute($command);
}
```

Мы уже рассматривали простейшую реализацию этого интерфейса. Рассмотрим более детально:

```
class SimpleCommandBus implements CommandBus {

    public function __construct(Container $container, CommandInflector $infle
    {
        $this->container = $container;
        $this->inflector = $inflector;
    }

    public function execute($command)
    {
        return $this->resolveHandler($command)->handle($command);
    }

    public function resolveHandler($command)
    {
        return $this->container->make( $this->inflector->getHandlerClass($com
    }
}
```

У нас есть компонент `CommandInflector`, который может использовать любую стратегию, связывающую обработчик и конкретный класс команды. Например, можно просто вернуть `str_replace('Command', 'Handler', get_class($command));`. Все что требуется для реализации такого подхода, так это хранить все обработчики и команды придерживаясь конкретной структуры директорий (если вы используете PSR-совместимую автозагрузку). Как вы реализуете это связывание, решать вам. Все зависит только от вас и нужд вашего проекта.

Какие еще реализации шины команды, помимо приведенной выше “простой”, мы можем захотеть использовать? Ну давайте для начала переименуем простую в синхронную шину команд (`SynchronousCommandBus`), так как именно это она и делает. Выполняет команды синхронно, по мере поступления. Из этого следует, что мы можем захотеть реализовать асинхронную шину команд (`AsynchronousCommandBus`). Как мы уже говорили выше, эта реализация будет добавлять команды в очередь сообщений на выполнение обработчиками по мере доступности, вне текущего запроса пользователя.

Помимо различных реализаций шины команд, мы также можем добавить больше декораторов для уже существующих реализаций. Например, я нахожу полезным обернуть шину команд декоратором с целью валидации входящих данных перед их обработкой.

```
class ValidationCommandBus implements CommandBus {

    public function __construct(CommandBus $bus, Container $container, Comman

    public function execute($command)
    {
        $this->validate($command);
        return $this->bus->execute($command);
    }

    public function validate($command)
    {
        $validator = $this->container->make($this->inflector->getValidatorCla
        $validator->validate($command); // бросить исключение в случае невали
    }
}
```

Здесь `ValidationCommandBus` является декоратором. Его задача — провалидировать команду и затем отправить ее следующей шине на обработку, если все хорошо. Следующей шиной может также быть декоратор (может для логирования? Или например аудита действий), или же настоящая реализация, которая уже займется непосредственно обработкой.

Итак, имея возможность комбинировать различные варианты шин команд, а также учитывая возможность добавление дополнительного поведения поверх нее, мы получаем довольно гибкий инструмент для обработки команд нашего приложения (сценариев использования)!

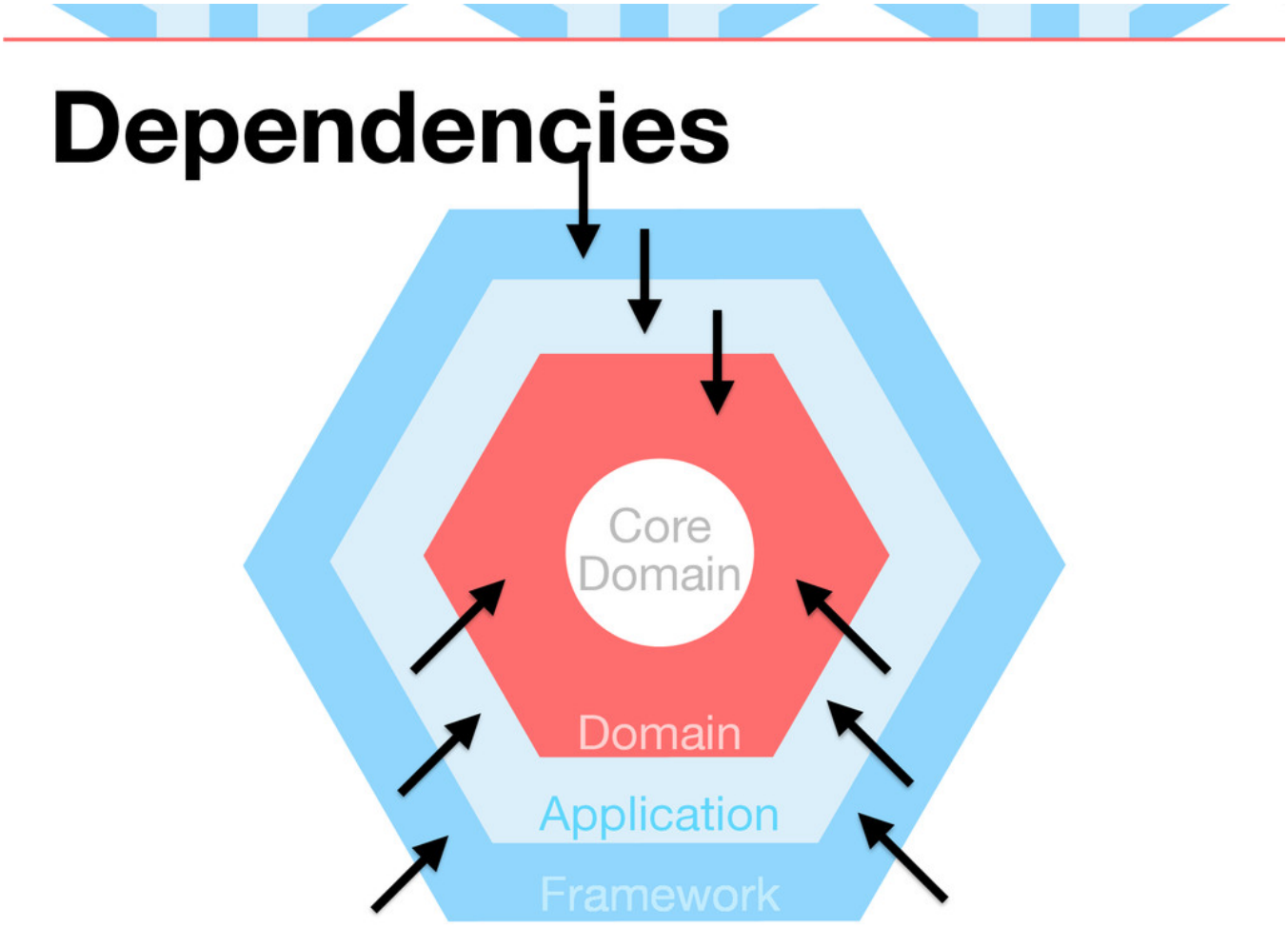
И все эти вещи максимально изолированы от внешних слоев нашего приложения. Наш слой фреймворка (и далее) никак не диктует, как мы можем использовать приложение. Напротив, именно приложение диктует то, как им можно воспользоваться.

Зависимости

Пока-что мы только слегка затронули понятие зависимостей. В рамках гексагональной архитектуры мы придерживаемся одностороннего потока зависимостей: снаружи внутрь. Слой предметной области (центральный слой) не должен зависеть от наружных слоев. Слой уровня приложения должен зависеть от слоя предметной области, но не от слоя фреймворка. Слой фреймворка

должен зависеть от слоя уровня приложения, но не от внешних факторов.

Ранее мы говорили, что интерфейсы это основной механизм инкапсуляции изменений. Они позволяют нам регламентировать, как слои будут общаться друг с другом, при этом избегая излишней связанности между ними. Если мы посмотрим на зависимости, то по сути придем к той же идее. Давайте разберемся.



Зависимости: идем внутрь

Зависимости проще представить когда мы прокидываем данные/логику “внутрь”. Когда на сервер приходит HTTP запрос, у нас должен быть какой-то код, который обработает его, иначе ничего не произойдет. Следовательно, внешний HTTP запрос зависит от нашего слоя фреймворка, который нужен для интерпретации запроса. Если фреймворку удалось интерпретировать запрос и определить маршрут к контроллеру, которому он соответствует, то этому контроллеру нужно что-то чтобы выполнить действия. Без слоя уровня приложения ему нечего делать. Слой уровня фреймворка зависит от слоя уровня приложения. Чтобы слою уровня приложения было кем руководить, ему нужен слой предметной области. Он зависит от него для того, чтобы иметь возможность выполнить требуемое действие. Слой предметной области же, зависит (в основном) от поведения и ограничений, определенных в нем же.

Когда мы пытаемся проследить путь запроса, совершенного извне, внутрь нашего приложения, то зависимости проглядываются весьма явно. Внешние слои зависят от внутренних, но они могут полностью игнорировать существование более глубоких слоев приложения. Все что слой должен знать, это какой метод вызвать и какие данные туда передать. Детали реализации безопасно инкапсулированы там где нужно. То как мы использовали интерфейсы хорошо это показывает.

Зависимости: выходим наружу (DI)

Когда мы выходим наружу, все становится чуточку сложнее. Давайте разберемся, что делает наше приложение в ответ на какой-то запрос, а именно как происходит обработка запроса и формирование ответа. Приведу примеры:

Нашему слою предметной области скорее всего нужен будет доступ к базе данных, чтобы загрузить оттуда сущности. Следовательно слой предметной области зависит от какого-то хранилища данных.

Слой уровня приложения после завершения какой-то задачи должен отправить уведомление пользователю. Если мы используем email-ы для доставки уведомлений, и при этом для используем AWS SES, то мы можем сказать, что наш слой уровня приложения имеет зависимость от SES для организации доставки уведомлений.

Как вы могли заметить, в нашей концепции прослеживается зависимость внутренних слоев от того, что содержится во внешних! Как бы нам инвертировать направление зависимостей?

Я специально использовал слово “инвертировать”. Это должно было намекнуть вам на принцип **Dependency Inversion** или принцип иверсии зависимостей. Буква **D** в SOLID. И для этого нам опять на помощь приходят интерфейсы.

Воспользуемся интерфейсами для инверсии зависимостей. С их помощью мы можем регламентировать, как будут взаимодействовать наши слои. Причем нам совсем не интересно как будут реализованы эти интерфейсы в других слоях. Таким образом конкретный слой диктует внешним слоям что он от них хочет, не завися от реализации. Это и есть инверсия зависимостей.

Наш слой предметной области может объявить интерфейс репозитория наших сущностей. Этот интерфейс затем будет реализован в одном из внешних слоев (скорее всего в слое уровня фреймворка). Однако за счет того, что мы объявили интерфейс, мы отделили нашу бизнес логику от конкретного способа хранения данных. Это дает нам возможность поменять слой хранения наших моделей при тестировании, или же сменить его если того будет требовать необходимость (будет здорово, если наше приложение будет нуждаться в гибком масштабировании).

С нотификатором (сервисом уведомлений) схожая ситуация. Наш слой уровня приложения знает только то, что ему нужно что-то для отправки уведомления. Для этого мы и сделали интерфейс `Notifier`. Приложению не нужно знать, каким образом мы будем отправлять эти уведомления. Оно только определяет каким образом с ним взаимодействовать. Таким образом, наш интерфейс `Notifier` будет объявлен в слое уровня приложения и реализован в слое уровня фреймворка. За счет этого направление зависимости изменилось, мы инвертировали его. Мы сказали внешним слоям как мы собираемся их использовать. Поскольку мы можем сменить реализацию интерфейсов когда заходим, мы можем говорить о том, что наши слои разделены.

Итак, подведем итог. Мы используем интерфейсы для обозначения направления потока логики внутрь и наружу. Мы использовали инверсию зависимостей для того, чтобы сохранить одинаковое направление зависимостей. Мы отделили внутренние слои от внешних, и при этом продолжаем ими пользоваться!

Заключение

Мы покрыли довольно много материала! Эта статья является результатом довольно объемного исследования архитектуры кода. Дабы не заострять внимание на конкретике, многие вещи были обобщены. Мы все же имеем дело с концепцией, а не с правилами в духе “делайте только так!”.

Подводя итог, гексагональная архитектура описывает “хорошие” практики написания кода. Это не какой-то конкретный подход к написанию кода приложений. Он хорошо подходит как для людей, использующих конкретный фреймворк, так и для людей, предпочитающих отдельные библиотеки фреймворкам.

Гексагональная архитектура позволяет по новому взглянуть на все те же старые принципы, с которыми знакомятся разработчики изучая правила построения архитектуры приложений.

Просьба высылать все комментарии по оформлению перевода (а-ля грамматика, пунктуация, не нравится построение предложения) в личку. Буду премного благодарен.