Лабораторная работа №3 «Гексагональная архитектура»

Цель работы

Познакомиться с гексагональной архитектурой и проектированием систем на её основе.

Задание для выполнения

Определите структуру вашего приложения (ЛР №№1–2) в соответствии с архитектурным стилем «гексагональная архитектура».

Разработайте файловую структуру для всех уровней, примерно представьте, какие сущности, репозитории, сервисы вам понадобятся, где необходима инверсия зависимостей и т.д.

Конкретную реализацию на каждом уровне вы будете делать на последующих работах.

Теоретические сведения

Теоретические сведения из данной работы пригодятся Вам и при выполнении последующих работ.

С распространением предметно-ориентированного проектирования (DDD), доменно-центрические архитектуры становятся всё более популярными. Поэтому **гексагональная архитектура (ГА)**, также известная как **порты и адаптеры**, переоткрывается PHP-сообществом. Изобретенная в 2005 году Алистером Кокберном, одним из авторов «Манифеста Agile», ГА позволяет приложению быть равно управляемым пользователями, программами, автоматизированными или пакетными скриптами, и разрабатываться и тестироваться изолированно от реальной инфраструктуры запуска (устройств, БД). В результате получается агностическая инфраструктура веб-приложения, которое легко тестировать, писать и сопровождать [2].

В большинстве блогов и книг вы найдете концентрические круги, представляющие разные слои ПО. Как объясняет Роберт Мартин в «Чистой архитектуре» [1], внешний круг – где живет ваша инфраструктура. Внутренний – где живут сущности. А работать данную архитектуру заставляет **Правило Зависимостей**. Это правило говорит, чтозависимости могут указывать только внутрь. Ничто во внутренних кругах не должно знать ни о чем из внешних (см. рисунок 1).

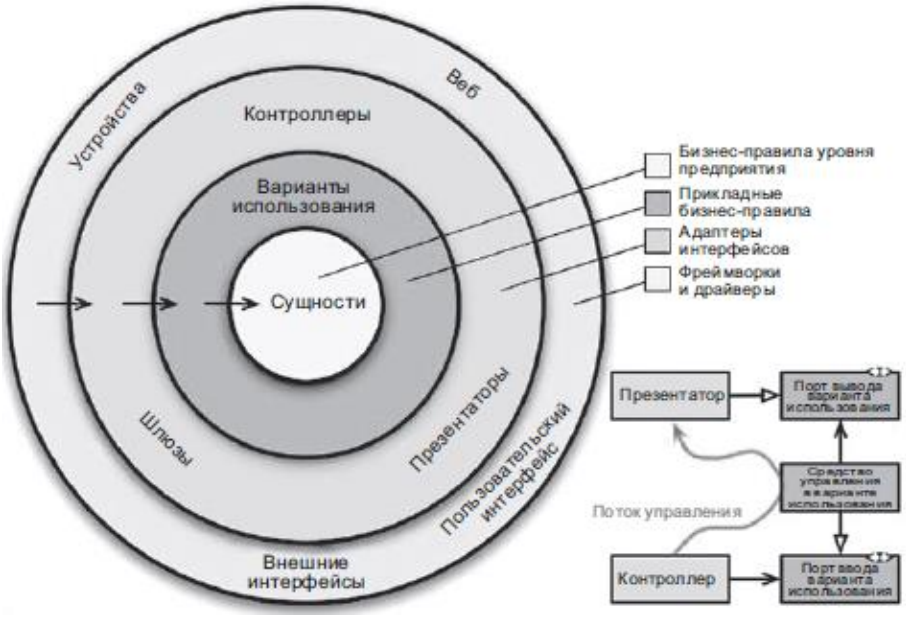

Рисунок 1 – Чистая архитектура [1]

Гексагональная архитектура, или Порты и Адаптеры, развивает идею чистой архитектуры, представляя архитектуру в виде концентрических гексагонов (см. Рисунок 2).

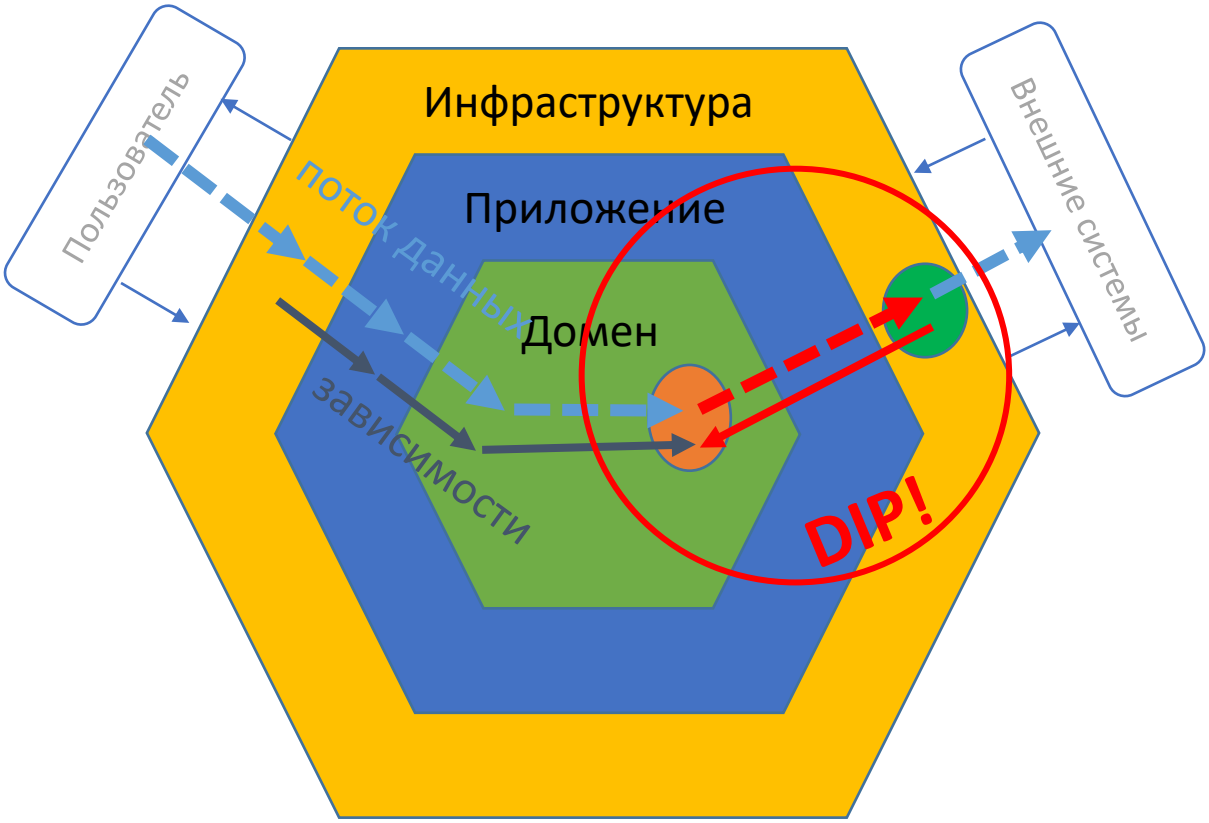

Рисунок 2 – Гексагональная архитектура

Бизнес-логика реализуется уровнями **домена** (критические бизнес-правила) и **приложения** (варианты использования). **Инфраструктура** содержит адаптеры для портов взаимодействия с внешним миром (см. рисунок 3).

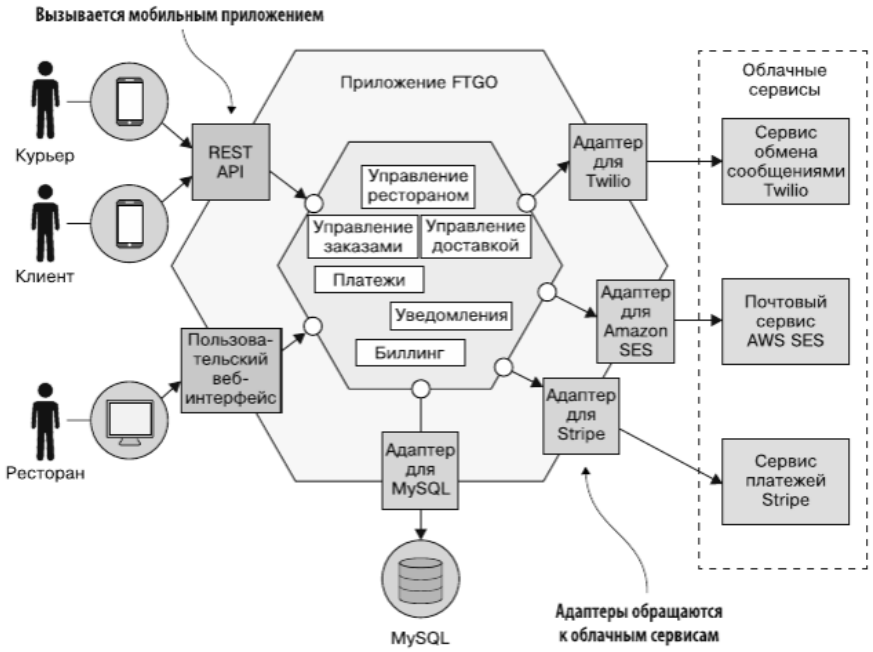

Рисунок 3 – Пример гексагональной архитектуры [3]

Давайте посмотрим, как это применить к реальным PHP приложениям.

Представим, что ваша компания проектирует систему *Idy*. Пользователи добавляют и оценивают идеи, так что наиболее интересные из них могут быть реализованы компанией. Понедельник, утро, начался новый спринт, и вы проводите ревью некоторых пользовательских историй с командой и владельцем продукта. **Как неавторизованный пользователь, я хочу оценивать идею и автор должен быть уведомлен по почте,** ведь это реально важно, не так ли?

Первый подход

Будучи хорошим разработчиком, вы решаете разделять и властвовать – и начинаете с первой части, *я хочу оценивать идею*. И уже потом подумать об *автор должен быть уведомлен по почте*. Звучит, как неплохой план.

В терминах бизнес-правил, оценка идеи – это нахождение идеи по идентификатору в репозитории идей, где хранятся все идеи, добавить оценку, пересчитать средний рейтинг и сохранить идею обратно. Если идея не существует или репозиторий недоступен необходимо выбросить исключение, чтобы показать сообщение об ошибке, перенаправить пользователя или сделать что-то ещё, что требует бизнес.

Для того, чтобы *выполнить* этот *Вариант Использования (Use Case)*, необходим всего лишь идентификатор идеи и рейтинг от пользователя. Два целых числа из запроса пользователя.

Для построения веб-приложения компания всё ещё использует Zend Framework 1. Как и в большинстве компаний, некоторые части приложения могут быть новее, более SOLID, а прочие – большой ком грязи. В любом случае, вы знаете, что неважно на самом деле, какой фреймворк используется, важно только, как писать чистый код, поддержка которого будет стоить меньше для вашей компании.

Вы пытаетесь применить некоторые принципы Agile (ну те, которые запомнились с последней конференции), например, «**Make it work. Make it right. Make it fast**». После некоторой работы получен следующий код:

Листинг 1 – Сценарий транзакции для обновления рейтинга

```
class IdeaController extends Zend_Controller_Action
{
      public function rateAction()
      {
            // Getting parameters from the request
            $ideaId = $this->request->getParam('id');
            $rating = $this->request->getParam('rating');

            // Building database connection
            $db = new Zend_Db_Adapter_Pdo_Mysql([
               'host'       => 'localhost',
               'username' => 'idy',
               'password' => '',
               'dbname'     => 'idy'
            ]);

            // Finding the idea in the database
            $sql = 'SELECT * FROM ideas WHERE idea_id = ?';
            $row = $db->fetchRow($sql, $ideaId);
            if (!$row) {
                  throw new Exception('Idea does not exist');
            }

            // Building the idea from the database
            $idea = new Idea();
            $idea->setId($row['id']);
            $idea->setTitle($row['title']);
            $idea->setDescription($row['description']);
            $idea->setRating($row['rating']);
            $idea->setVotes($row['votes']);
            $idea->setAuthor($row['email']);

            // Add user rating
            $idea->addRating($rating);

            // Update the idea and save it to the database
            $data = [
                  'votes' => $idea->getVotes(),
                  'rating' => $idea->getRating()
            ];

            $where['idea_id = ?'] = $ideaId;
            $db->update('ideas', $data, $where);

            // Redirect to view idea page
            $this->redirect('/idea/' . $ideaId);
      }
}
```

Понятно, что подумает читатель: *Да кто же обращается к данным непосредственно из контроллера? Это же пример из 90х!,* да, да , вы правы. Если вы используете фреймворк, то, наверняка, используете и ORM. Возможно, самописную, иди одну из существующих на рынке, типа Doctrine, Eloquent, Zend, и т.д. Если так, то вы на один шаг дальше тех, у кого есть какой-то объект соединения с БД, но они не видят дальше своего носа.

Для новичков: Листинг 1 просто работает. В любом случае, если присмотреться к контроллеру, вы увидите больше, чем бизнес-правила, вы также увидите, как ваш фреймворк маршрутизирует запрос к вашим бизнес-правилам, ссылки на БД и как с ней соединиться. Вблизи вы видите ссылки на вашу **инфраструктуру**.

Инфраструктура – это детали, которые позволяют работать бизнес-правилам. Очевидно, необходимы механизмы доступа к подобным деталям (API, веб, консольные приложения, и т.д.) и необходимо какое-то физическое место для хранения наших идей (память, БД, NoSQL, и т.д.). В любом случае, нам нужно иметь возможность заменить любую из этих частей другой, которая ведет себя так же, но с другой реализацией. Может, начнем с доступа к БД? Все эти соединения `Zend_DB_Adapter` (или прямые команды MySQL в вашем случае) просятся в некий объект, который инкапсулирует выборку и сохранение объектов Idea. Они просто просятся быть **Репозиторием**.

Репозитории и «ребро хранения»

Независимо от того, изменяются бизнес-правила или ифраструктура, мы редактируем один и тот же кусок кода. Поверьте, в «суровом энтерпрайзе» вам совсем не захочется, чтобы много людей трогало один и тот же кусок кода по различным причинам. Старайтесь, чтобы ваши функции делали одну и только одну вещь, чтобы снизить вероятность, что вокруг будут ошиваться другие люди с тем же куском кода. Узнать об этом больше можно, глянув на **Принцип Единственной Ответственности (SRP)**, например, тут: http://www.objectmentor.com/resources/articles/srp.pdf.

Листинг 1 – как раз этот случай. Если мы захотим перенести данные в Redis, или добавить фичу уведомления автора, всё равно придется обновить метод `rateAction`. Слишком высока вероятность затронуть одни и те же строки `rateAction` несколькими разработчиками одновременно. Код Листинга 1 очень хрупок. И если вы с командой придерживаетесь принципа *Работает – не трожь!*, то с SRP у вас явно не срослось.

Итак, нам нужно разбить наш код и инкапсулировать ответственность за выборку и хранение идей в другом объекте. Наилучший путь, как описано ранее, использовать для этого Репозиторий. Вызов принят! Давайте посмотрим на результат:

Листинг 2 – Выделение репозитория из контроллера

```
class IdeaController extends Zend_Controller_Action
{
```

```
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');

        $ideaRepository = new IdeaRepository();
        $idea = $ideaRepository->find($ideaId);
        if (!$idea) {
            throw new Exception('Idea does not exist');
        }

        $idea->addRating($rating);

        $ideaRepository->update($idea);

        $this->redirect('/idea/' . $ideaId);
    }
}

class IdeaRepository
{
    private $client;

    public function __construct()
    {
        $this->client = new Zend_Db_Adapter_Pdo_Mysql([
            'host' => 'localhost',
            'username' => 'idy',
            'password' => '',
            'dbname' => 'idy'
        ]);
    }

    public function find($id)
    {
        $sql = 'SELECT * FROM ideas WHERE idea_id = ?';

        $row = $this->client->fetchRow($sql, $id);
        if (!$row) {
            return null;
        }

        $idea = new Idea();
        $idea->setId($row['id']);
        $idea->setTitle($row['title']);
        $idea->setDescription($row['description']);
        $idea->setRating($row['rating']);
        $idea->setVotes($row['votes']);
        $idea->setAuthor($row['email']);

        return $idea;
    }

    public function update(Idea $idea)
    {
        $data = [
            'title' => $idea->getTitle(),
            'description' => $idea->getDescription(),
            'rating' => $idea->getRating(),
            'votes' => $idea->getVotes(),
            'email' => $idea->getAuthor(),
        ];

        $where = ['idea_id = ?' => $idea->getId()];
        $this->client->update('ideas', $data, $where);
    }
}
```

Результат уже приятнее. Метод `rateAction` контроллера `IdeaController` более понятен. При чтении видны бизнес-правила. `IdeaRepository` – **бизнес-концепт**. Когда говорим с людьми от бизнеса, они понимают, что такое `IdeaRepository`: это место, куда я кладу идеи и откуда их забираю.

Репозиторий *служит посредником между доменом и слоем данных используя интерфейс, схожий с коллекцией, для доступа к доменным объектам*, так нам говорит каталог паттернов Мартина Фаулера.

Если вы уже используете ORM такую, как Doctrine, то ваши текущие репозитории наследуются от `EntityRepository`. Если вам нужен какой-то из них, вы просите Doctrine `EntityManager` сделать всю работу. Результирующий код будет почти таким же, с дополнительным доступом к `EntityManager` в контроллере для получения `IdeaRepository`.

На данном этапе, мы можем видеть ландшафт одного из ребер нашего гексагона, ребро *хранения*. В любом случае, эта сторона еще недостаточно проработана, поскольку остается зависимость между `IdeaRepository` и тем, как он реализован.

Для того, чтобы эффективно разделить границы приложения и инфраструктурные границы, необходим дополнительный шаг. Нам нужно явно отделить поведение от реализации используя интерфейсы.

Разделение бизнеса и хранилища

Вы когда-нибудь сталкивались с ситуацией, когда начинаете говорить с Владельцем Продукта, Бизнес-аналитиком или Проектным Менеджером о проблемах в БД? Можете вспомнить (или представить) их лица, когда описываете выборку или сохранение объекта? Да они вообще не понимают, о чем вы!

Правда в том, что им наплевать на БД, но это нормально. Вы решите хранить идеи в MySQL, Redis или SQLite – это ваша проблема, но не их. Запомните, с точки зрения бизнеса, **ваша инфраструктура- это деталь**. Бизнес-правила не изменятся от того, что вы пользуетесь Symfony или Zend Framework, MySQL или PostgreSQL, REST или SOAP, и т.д.

Вот почему так важно отделить наш `IdeaRepository` от его реализации. Простейший путь – использовать соответствующий интерфейс. Как можно этого достичь? Посмотрим:

Листинг 3 – Применение интерфейса репозитория

```
class IdeaController extends Zend_Controller_Action
{
      public function rateAction()
      {
            $ideaId = $this->request->getParam('id');
            $rating = $this->request->getParam('rating');

            $ideaRepository = new MySQLIdeaRepository();
            $idea = $ideaRepository->find($ideaId);
```

```
            if (!$idea) {
                throw new Exception('Idea does not exist');
            }

            $idea->addRating($rating);

            $ideaRepository->update($idea);

            $this->redirect('/idea/' . $ideaId);
        }
}

interface IdeaRepository
{
        /**
        * @param int $id
        * @return null|Idea
        */
        public function find($id);

        /**
        * @param Idea $idea
        */
        public function update(Idea $idea);
}

class MySQLIdeaRepository implements IdeaRepository
{
        // ...
}
```

Просто, не правда ли? Мы вынесли поведение `IdeaRepository` в интерфейс, переименовав `IdeaRepository` в `MySQLIdeaRepository` и обновив `rateAction` так, чтобы использовался `MySQLIdeaRepository`. Но в чем выгода?

Мы теперь можем заменять репозиторий, используемый в контроллере, на любой другой, реализующий тот же интерфейс. Что ж, попробуем другую реализацию.

Перенесем хранилище в Redis

В процессе спринта и после обсуждение вы поняли, что использование стратегии NoSQL поможет увеличить эффективность фичи. Redis – один из наших лучших друзей. Поехали:

Листинг 4 – Замена репозитория на другой, реализующий тот же интерфейс

```
class IdeaController extends Zend_Controller_Action
{
        public function rateAction()
        {
            $ideaId = $this->request->getParam('id');
            $rating = $this->request->getParam('rating');

            $ideaRepository = new RedisIdeaRepository();
            $idea = $ideaRepository->find($ideaId);
            if (!$idea) {
                throw new Exception('Idea does not exist');
            }

            $idea->addRating($rating);
```

```
            $ideaRepository->update($idea);

            $this->redirect('/idea/' . $ideaId);
        }
}

interface IdeaRepository
{
        /**
        * @param int $id
        * @return null|Idea
        */
        public function find($id);

        /**
        * @param Idea $idea
        */
        public function update(Idea $idea);
}

class RedisIdeaRepository implements IdeaRepository
{
        private $client;

        public function __construct()
        {
            $this->client = new Predis\Client();
        }

        public function find($id)
        {
            $idea = $this->client->get($this->getKey($id));
            if (!$idea) {
                return null;
            }

            return unserialize($idea);
        }

        public function update(Idea $idea)
        {
            $this->client->set(
                $this->getKey($idea->getId()),
                serialize($idea)
            );
        }

        private function getKey($id)
        {
            return 'idea:' . $id;
        }
}
```

И вновь просто. Вы создали `RedisIdeaRepository`, который реализует интерфейс `IdeaRepository` и решили использовать Predis как менеджер соединения. Код выглядит меньшим, простейшим и быстрейшим. Но что насчет контроллера? Он остался таким же, поменялось только указание, какой репозиторий использовать, т.е. одна строчка кода.

В качестве упражнения для читателя попробуйте создать `IdeaRepository` для SQLite, для файла или in-memory реализацию, использующую массивы. Дополнительные баллы, если подумаете, как ORM репозитории соотносятся с доменными репозиториями и как `@аннотации` ORM подходят такой архитектуре.

Разделение бизнеса и веб-фреймворка

Мы уже увидели, как просто изменить одну стратегию хранения на другую. В любом случае, хранение – не единственное ребро гексагона. Что насчет пользовательского взаимодействия с приложением?

Ваш CTO решил, что приложение должно мигрировать на Symfony2, так что разрабатывая новые фичи с использованием ZF1 необходимо учитывать простоту предстоящей миграции.

Листинг 5 – Выделение варианта использования

```
class IdeaController extends Zend_Controller_Action
{
      public function rateAction()
      {
            $ideaId = $this->request->getParam('id');
            $rating = $this->request->getParam('rating');

            $ideaRepository = new RedisIdeaRepository();
            $useCase = new RateIdeaUseCase($ideaRepository);
            $response = $useCase->execute($ideaId, $rating);

            $this->redirect('/idea/' . $ideaId);
      }
}

interface IdeaRepository
{
      // ...
}

class RateIdeaUseCase
{
      private $ideaRepository;

      public function __construct(IdeaRepository $ideaRepository)
      {
            $this->ideaRepository = $ideaRepository;
      }

      public function execute($ideaId, $rating)
      {
            try {
                  $idea = $this->ideaRepository->find($ideaId);
            } catch(Exception $e) {
                  throw new RepositoryNotAvailableException();
            }

            if (!$idea) {
                  throw new IdeaDoesNotExistException();
            }
            try {
                  $idea->addRating($rating);
                  $this->ideaRepository->update($idea);
            } catch(Exception $e) {
                  throw new RepositoryNotAvailableException();
            }

            return $idea;
      }
}
```

Давайте оценим изменения. Наш контроллер теперь вовсе не содержит бизнес-логики. Мы перенесли всю логику внутрь нового объекта `RateIdeaUseCase`, который её инкапсулирует. Этот объект также известен как Контроллер, Интерактор, или **Сервис Приложения (Application Service)**.

Вся магия происходит в методе execute. Все зависимости, такие как `RedisIdeaRepository`, передаются как аргумент в конструктор. Все ссылки на `IdeaRepository` внутри Варианта Использования указывают на интерфейс, но не на конкретную реализацию.

Это реально круто. Если вы посмотрите внутрь `RateIdeaUseCase`, то увидите, что там ничто не говорит о MySQL или Zend Framework. Нет ссылок, нет экземпляров, нет аннотаций, ничего. Ваша инфраструктура не важна, всё говорит только о бизнес-логике.

Дополнительно, мы немного подкорректировали выбрасываемые исключения. Бизнес-процессы также имеют исключения. `NotAvailableRepository` и `IdeaDoesNotExist` – два из них. Основываясь на том, которое из них выброшено, мы можем по-разному реагировать на границе фреймворка.

Иногда количество параметров, передаваемых Варианту Использования слишком велико. С целью организовать их достаточно часто применяют **Объект Передачи Данных (Data Transfer Object, DTO)** и передают все параметры вместе:

Листинг 6 – Использование DTO

```
class IdeaController extends Zend_Controller_Action
{
	public function rateAction()
	{
		$ideaId = $this->request->getParam('id');
		$rating = $this->request->getParam('rating');

		$ideaRepository = new RedisIdeaRepository();
		$useCase = new RateIdeaUseCase($ideaRepository);
		$response = $useCase->execute(
			new RateIdeaRequest($ideaId, $rating)
		);

		$this->redirect('/idea/' . $response->idea->getId());
	}
}

class RateIdeaRequest
{
	public $ideaId;
	public $rating;

	public function __construct($ideaId, $rating)
	{
		$this->ideaId = $ideaId;
		$this->rating = $rating;
	}
}

class RateIdeaResponse
{
	public $idea;
```

```
        public function __construct(Idea $idea)
        {
                $this->idea = $idea;
        }
}

class RateIdeaUseCase
{
        // ...
        public function execute($request)
        {
                $ideaId = $request->ideaId;
                $rating = $request->rating;
                // ...
                return new RateIdeaResponse($idea);
        }
}
```

Основные изменения во введении двух новых объектов – Request и Response. Они не обязательны, Вариант Использования может и не иметь запроса или ответа. Другая важная деталь – как строится этот запрос. В данном случае мы строим запрос из параметров объекта запроса ZF.

Ок, но постойте, в чем же реальная выгода? Легче поменять один фреймворк на другой, или вызвать наш Вариант Использования из другого *механизма доставки*. Давайте посмотрим с этой стороны.

Оценка идеи по API

В течение дня Владелец Продукта приходит к вам и говорит: *кстати, пользователь должен иметь возможность оценить идею, используя наше мобильное приложение. Мне кажется, нам нужно будет обновить API, сможешь сделать это в текущем спринте?*

Без проблем! И бизнес впечатлён вашей обязательностью.

Согласно Роберту Мартину: *Всемирная паутина (или Веб) — это механизм доставки, устройство ввода/вывода. ... Архитектура системы должна быть максимально нейтральной к механизмам доставки услуг. У вас должна иметься возможность реализовать систему в форме консольного приложения, веб-приложения, толстого клиента или даже веб-службы без чрезмерного усложнения или изменения основной архитектуры.* [1, стр. 199]

Ваше нынешнее API построено с использованием Silex, микрофреймворка на базе Symfony2 Components. Перейдем к Листингу 7:

Листинг 7 – API

```
require_once __DIR__.'/../vendor/autoload.php';

$app = new \Silex\Application();

// ... more routes

$app->get(
    '/api/rate/idea/{ideaId}/rating/{rating}',
```

```
    function($ideaId, $rating) use ($app) {
        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );


        return $app->json($response->idea);
    }
);


$app->run();
```

Есть что-то знакомое? Есть код, виденный ранее? Подскажу:

```
$ideaRepository = new RedisIdeaRepository();
$useCase = new RateIdeaUseCase($ideaRepository);
$response = $useCase->execute(
     new RateIdeaRequest($ideaId, $rating)
);
```

Чувак, я помню эти три строчки! Они выглядят точно, как в веб-приложении. И это верно, поскольку Вариант Использования инкапсулирует бизнес-правила, которые необходимы для подготовки запроса, получения ответа и соответствующих действий.

Мы предоставляем нашим пользователям другой путь оценки идеи; другой *механизм доставки*. Основная разница в том, где создается RateIdeaRequest. В первом примере он создан из запроса ZF, а теперь из запроса Silex, используя параметры пути.

Консольное приложения для оценки

Иногда Вариант Использования должен быть вызван из задачи Cron или командной строки. Например, для пакетной обработки или тестирования. Тестируя фичу через веб или API вы понимаете, что было бы неплохо иметь и командную строку, чтобы не ходить в браузер.

Если вы используете shell-скрипты, я бы порекомендовал обратить внимание на компонент Symfony Console. Тогда код выглядел бы так:

Листинг 8 – Консольное приложение

```
namespace Idy\Console\Command;


use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;


class VoteIdeaCommand extends Command
{
    protected function configure()
    {
        $this
            ->setName('idea:rate')
```

```
            ->setDescription('Rate an idea')
            ->addArgument('id', InputArgument::REQUIRED)
            ->addArgument('rating', InputArgument::REQUIRED)
        ;
    }


    protected function execute(
        InputInterface $input,
        OutputInterface $output
    )
    {
        $ideaId = $input->getArgument('id');
        $rating = $input->getArgument('rating');


        $ideaRepository = new RedisIdeaRepository();
        $useCase = new RateIdeaUseCase($ideaRepository);
        $response = $useCase->execute(
            new RateIdeaRequest($ideaId, $rating)
        );


        $output->writeln('Done!');
    }
}
```

Опять те же три строчки! Как и ранее, Вариант Использования и его бизнес-логика остаются нетронутыми, мы просто добавляем ещё один *механизм доставки*. Поздравляем, вы открыли для себя пользовательское ребро гексагона.

Ещё много нужно сделать. Вы наверное слышали, что реальные разрабы делают TDD. Мы уже начали нашу историю, так что нам нужно соответствовать и далее.

… Про тестирование – в следующей работе.

Apргх, Как много зависимостей!

Это нормально, что я должен создавать столько зависимостей вручную (*в том числе для нужд тестирования, – прим. переводчика)*? Нет. Обычно используют компонент внедрения зависимостей (dependency injection) или Service Container с подобными способностями. Опять же, на помощь приходит Symfony, во всяком случае можете глянуть на PHP-DI 4.

Давайте глянем на результат в Листинге 14 после применения Symfone Service Container к нашему приложению:

Листинг 14 – Внедрение зависимостей

```
class IdeaController extends ContainerAwareController
{
    public function rateAction()
    {
        $ideaId = $this->request->getParam('id');
        $rating = $this->request->getParam('rating');


        $useCase = $this->get('rate_idea_use_case');
        $response = $useCase->execute(
```

```
            new RateIdeaRequest($ideaId, $rating)
        );


        $this->redirect('/idea/'.$response->idea->getId());
    }
}
```

Контроллер был изменен, чтобы получить доступ к контейнеру, вот почему он наследуется от другого базового контроллера `ContainerAwareController`, у которого есть метод `get` для получения каждого из сервисов контейнера:

Листинг 15 – Конфигурация Service Container

```
<?xml version="1.0" ?>
<container xmlns="http://symfony.com/schema/dic/services"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://symfony.com/schema/dic/services
           http://symfony.com/schema/dic/services/services-1.0.xsd">
    <services>
        <service
            id="rate_idea_use_case"
            class="RateIdeaUseCase">
            <argument type="service" id="idea_repository" />
        </service>


        <service
            id="idea_repository"
            class="RedisIdeaRepository">
            <argument type="service">
                <service class="Predis\Client" />
            </argument>
        </service>
    </services>
</container>
```

Листинг 15 достаточно понятен, но если вам нужно больше информации: https://symfony.com/doc/current/service_container.html

Доменные сервисы и край гексагона для уведомлений

О чём мы забыли? *Автор должен быть уведомлен по почте,* точно! Посмотрим, как стоит изменить Вариант Использования, чтобы это делалось:

Листинг 16 – Уведомления в сервисе

```
class RateIdeaUseCase
{
    /**
     * @var IdeaRepository
     */
    private $ideaRepository;


    /**
     * @var AuthorNotifier
     */
```

```
    private $authorNotifier;


    public function __construct(
        IdeaRepository $ideaRepository,
        AuthorNotifier $authorNotifier
    )
    {
        $this->ideaRepository = $ideaRepository;
        $this->authorNotifier = $authorNotifier;
    }


    public function execute(RateIdeaRequest $request)
    {
        $ideaId = $request->ideaId;
        $rating = $request->rating;


        try {
            $idea = $this->ideaRepository->find($ideaId);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }


        if (!$idea) {
            throw new IdeaDoesNotExistException();
        }


        try {
            $idea->addRating($rating);
            $this->ideaRepository->update($idea);
        } catch(Exception $e) {
            throw new RepositoryNotAvailableException();
        }


        try {
            $this->authorNotifier->notify(
                $idea->getAuthor()
            );
        } catch(Exception $e) {
            throw new NotificationNotSentException();
        }


        return $idea;
    }
}
```

Как вы заметили, мы добавили новый параметр для передачи сервиса `AuthorNotifier`, который будет отправлять письма автору. Это **Порт** в наименовании *Порты и Адаптеры*. Мы также модифицировали бизнес-правила метода `execute`.

Репозитории – не единственные объекты, которые могут иметь доступ к вашей инфраструктуре и должны быть отделены, используя интерфейсы или абстрактные классы. Доменные сервисы так же. Когда есть поведение, не чисто принадлежащее одной сущности вашего домена, тогда нужно создавать доменный сервис. Типичный паттерн – создать абстрактный доменный сервис, у которого есть конкретная имплементация и некоторые абстрактные методы, которые будет реализовывать **Адаптер**.

Подведем итоги

Чтобы иметь *чистую архитектуру*, которая помогает писать и тестировать программы, мы можем использовать Гексагональную Архитектуру. Для достижения этого мы инкапсулируем пользовательские бизнес-правила внутри объектов Вариантов Использования. Мы строим запрос к Варианту Использования из запроса нашего фреймворка, инстанциируем Вариант Использования и все зависимости, после чего выполняем. Мы получаем ответ и действуем соответственно ему. Если наш фреймворк умеет внедрять зависимости, тогда код значительно упрощается.

Подобный же Вариант Использования может быть использован для других *механизмов доставки* для того, чтобы пользователи получили доступ к фичам из разных клиентов (веб, API, консоль, и т.д.) [2]

Список использованных источников

1. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2018. – 352 с. : ил. – (Серия «Библиотека программиста»).
2. Buenosvinos, C. Hexagonal Architecture with PHP // php[architect]. – July, 2014. – *Перевод с англ. Кочурко, П.А.*
3. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. – 544 с.: ил. – (Серия «Библиотека программиста»).