|  |
| --- |
|  |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** |
| Институт информационных технологий (ИТ) |
| Кафедра инструментального и прикладного программного обеспечения (ИППО) |

|  |  |
| --- | --- |
| **ОТЧЕТ**  **ПО ЛАБОРАТОРНОЙ РАБОТЕ №7** | |
| **по дисциплине** |  |
| **«Разработка серверных частей интернет ресурсов»** | |
| Выполнил студент группы ИКБО-30-20 | Бессмертных И.В. |
| Принял ассистент | Благирев М.М. |

|  |  |  |
| --- | --- | --- |
| Практические работы выполнены | « » 2022г. |  |
| «Зачтено» | « » 2022 г. |  |

Москва 2022

СОДЕРЖАНИЕ

[1. Цель работы 3](#_Toc121324427)

[2. Ход работы 4](#_Toc121324428)

[ЗАКЛЮЧЕНИЕ 6](#_Toc121324429)

[Ответы на вопросы к практической работе 7](#_Toc121324430)

[Ссылка на удаленный репозиторий проекта 19](#_Toc121324431)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 20](#_Toc121324432)

1. Цель работы

В рамках данной практической работы предполагается провести рефакторинг информационной системы, созданной в практических работах 1-6. Рефакторинг - это процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и четкости.

Рефакторинг системы следует проводить за счет перевода с процедурой парадигмы на ООП парадигму с переносом на определенную архитектуру.

Также на внешний слой архитектуры, если будет использоваться DDD или на слой интерфейсов, если будет использоваться чистая архитектура, внедрить какой-либо паттерн проектирования. Например, MVC или MVP.

1. Ход работы

В процессе рефакторинга кода, я смог структурировать основной функционал и преобразовать его так, чтобы он соответствовал правилам ООП, а также переделал данный проект под паттерн проектирования MVC (Рисунок 2.1)

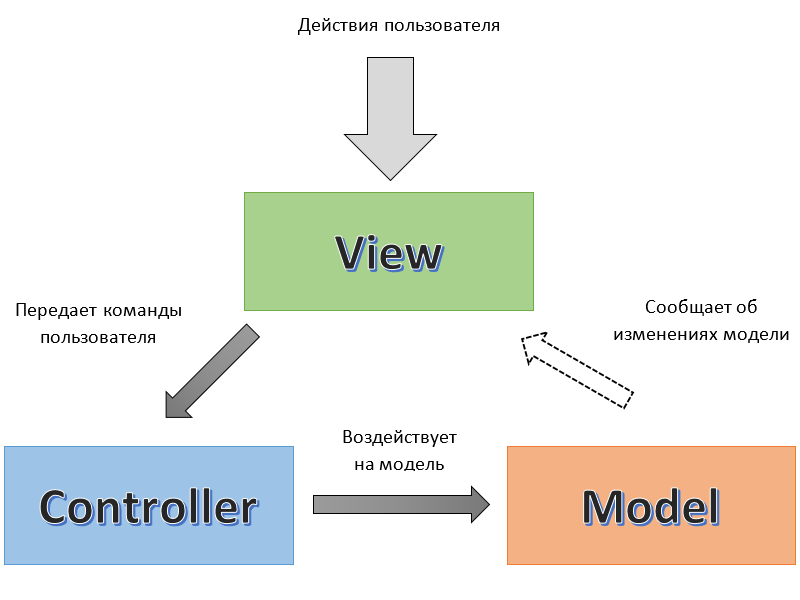


Рисунок 2.1 – MVC

Далее были сформированы директории в которых храняться Рисунок (2.2):

* контроллеры
* модели
* сервисы
* папка для загрузок
* представления

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2.2 – модели контроллеры представления и остальные директории

ЗАКЛЮЧЕНИЕ

Была создана конфигурация серверного программного обеспечения на наборе LAMP и проверена ее работоспособность. Данная конфигурация будет использоваться для выполнения следующих практических работ по данной дисциплине и для выполнения курсового проектирования.

Ответы на вопросы к практической работе

1. **Назовите основные признаки ООП.**

Абстракция – это способ придания объектам определённых характеристик, которые отличают его от всех остальных объектов.

Инкапсуляция – один из принципов ООП, позволяющее объединить данные и методы, работающие с ними, в классе.

Наследование – один из принципов ООП, позволяющий описать новый класс на основе уже существующего с частично или полностью заимствованной функциональностью.

Полиморфизм – один из принципов ООП, позволяющий использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

1. **Опишите как определить класс в PHP.**

Для примера представлен класс машины.

<?php  
class Car {  
    public $brand;  
    public $color;  
    public function \_\_construct($brand, $color = "blue")  
    {  
        $this->brand = $brand;  
        $this->color = $color;  
    }  
    public function getBrand()  
    {  
        return $this->brand;  
    }  
    public function getColor()  
    {  
        return $this->color;  
    }  
}  
?>

1. **Как создать экземпляр класса в PHP.**

Создадим экземпляр класса машины. Ниже представлен пример.

$car = new Car(“Mercedes”, “black”);

1. **Опишите механизм наследования в PHP.**

Ниже приведён примера наследования в PHP.

<?php  
class A {  
        // code here  
}  
class B extends A {  
        // code here  
}  
class C extends B {  
        // code here  
}  
?>

1. **Опишите правила совместимости сигнатур.**

При переопределении метода его сигнатура должна быть совместима с родительским методом. В противном случае выдаётся фатальная ошибка или, до PHP 8.0.0, генерируется ошибка уровня E\_WARNING. Сигнатура является совместимой, если она соответствует правилам [контравариантности](https://www.php.net/manual/ru/language.oop5.variance.php), делает обязательный параметр необязательным и если какие-либо новые параметры являются необязательными. Это известно как принцип подстановки Барбары Лисков или сокращённо LSP. Правила совместимости не распространяются на [конструктор](https://www.php.net/manual/ru/language.oop5.decon.php#language.oop5.decon.constructor) и сигнатуру private методов, они не будут выдавать фатальную ошибку в случае несоответствия сигнатуры. Ниже представлен пример.

<?php  
class Base  
{  
    public function foo(int $a) {  
        echo "Допустимо\n";  
    }  
}  
class Extend1 extends Base  
{  
    function foo(int $a = 5)  
    {  
        parent::foo($a);  
    }  
}  
class Extend2 extends Base  
{  
    function foo(int $a, $b = 5)  
    {  
        parent::foo($a);  
    }  
}  
$extended1 = new Extend1();  
$extended1->foo();  
$extended2 = new Extend2();  
$extended2->foo(1);

1. **Опишите методы и свойства Nullsafe.**

Начиная с PHP 8.0.0, к свойствам и методам можно также обращаться с помощью оператора "nullsafe": ?->. Оператор nullsafe работает так же, как доступ к свойству или методу, как указано выше, за исключением того, что если разыменование объекта выдаёт null, то будет возвращён null, а не выброшено исключение. Если разыменование является частью цепочки, остальная часть цепочки пропускается.

Аналогично заключению каждого обращения в is\_null(), но более компактный. Ниже представлен пример.

<?php  
// Начиная с PHP 8.0.0, эта строка:  
$result = $repository?->getUser(5)?->name;  
// Эквивалентна следующему блоку кода:  
if (is\_null($repository)) {  
    $result = null;  
} else {  
    $user = $repository->getUser(5);  
    if (is\_null($user)) {  
        $result = null;  
    } else {  
        $result = $user->name;  
    }  
}  
?>

1. **Опишите понятие автоматическая загрузка классов.**

Большинство разработчиков объектно-ориентированных приложений используют такое соглашение именования файлов, в котором каждый класс хранится в отдельно созданном для него файле. Одна из самых больших неприятностей - необходимость писать в начале каждого скрипта длинный список подгружаемых файлов (по одному для каждого класса).

Функция spl\_autoload\_register() позволяет зарегистрировать необходимое количество автозагрузчиков для автоматической загрузки классов и интерфейсов, если они в настоящее время не определены. Регистрируя автозагрузчики, PHP получает последний шанс для интерпретатора загрузить класс прежде, чем он закончит выполнение скрипта с ошибкой. Ниже представлен пример автоматической загрузки.

<?php  
spl\_autoload\_register(function ($class\_name) {  
    include $class\_name . '.php';  
});  
$obj = new MyClass1();  
$obj2 = new MyClass2();  
?>

1. **Опишите конструкторы и деструкторы в PHP.**

PHP позволяет объявлять методы-конструкторы. Классы, в которых объявлен метод-конструктор, будут вызывать этот метод при каждом создании нового объекта, так что это может оказаться полезным, например, для инициализации какого-либо состояния объекта перед его использованием. Конструктор задаётся следующим образом.

\_\_construct([mixed](https://www.php.net/manual/ru/language.types.declarations.php#language.types.declarations.mixed) ...$values = ""): void

PHP предоставляет концепцию деструктора, аналогичную с той, которая применяется в других ОО-языках, таких как C++. Деструктор будет вызван при освобождении всех ссылок на определённый объект или при завершении скрипта (порядок выполнения деструкторов не гарантируется).

\_\_destruct(): void

1. **Опишите понятие области видимости и модификаторы доступа в PHP.**

Область видимости свойства, метода или константы (начиная c PHP 7.1.0) может быть определена путём использования следующих ключевых слов в объявлении: public, protected или private. Доступ к свойствам и методам класса, объявленным как public (общедоступный), разрешён отовсюду. Модификатор protected (защищённый) разрешает доступ самому классу, наследующим его классам и родительским классам. Модификатор private (закрытый) ограничивает область видимости так, что только класс, где объявлен сам элемент, имеет к нему доступ.

1. **Опишите оператор разрешения области видимости.**

Оператор разрешения области видимости (также называемый "Paamayim Nekudotayim") или просто "двойное двоеточие" — это лексема, позволяющая обращаться к статическим свойствам, константам и переопределённым свойствам или методам класса.

При обращении к этим элементам извне класса, необходимо использовать имя этого класса.

1. **Опишите абстрактные классы и методы в PHP.**

PHP поддерживает определение абстрактных классов и методов. На основе абстрактного класса нельзя создавать объекты, и любой класс, содержащий хотя бы один абстрактный метод, должен быть определён как абстрактный. Методы, объявленные абстрактными, несут, по существу, лишь описательный смысл и не могут включать реализацию.

При наследовании от абстрактного класса, все методы, помеченные абстрактными в родительском классе, должны быть определены в дочернем классе и следовать обычным правилам наследования и совместимости сигнатуры.

Ниже представлен пример наследования от абстрактного класса на PHP.

<?php  
abstract class AbstractClass  
{  
   /\* Данный метод должен быть определён в дочернем классе \*/  
    abstract protected function getValue();  
    abstract protected function prefixValue($prefix);  
  
   /\* Общий метод \*/  
    public function printOut() {  
        print $this->getValue() . "\n";  
    }  
}  
class ConcreteClass1 extends AbstractClass  
{  
    protected function getValue() {  
        return "ConcreteClass1";  
    }  
    public function prefixValue($prefix) {  
        return "{$prefix}ConcreteClass1";  
    }  
}

?>

1. **Опишите интерфейсы в PHP.**

Интерфейсы объектов позволяют создавать код, который указывает, какие методы должен реализовать класс, без необходимости определять, как именно они должны быть реализованы. Интерфейсы разделяют пространство имён с классами и трейтами, поэтому они не могут называться одинаково.

Ниже приведён пример интерфейса и его имплементации.

<?php  
// Объявим интерфейс 'Template'  
interface Template  
{  
    public function setVariable($name, $var);  
    public function getHtml($template);  
}  
  
// Реализация интерфейса  
// Это будет работать  
class WorkingTemplate implements Template  
{  
    private $vars = [];  
    public function setVariable($name, $var)  
    {  
        $this->vars[$name] = $var;  
    }  
    public function getHtml($template)  
    {  
        foreach($this->vars as $name => $value) {  
            $template = str\_replace('{' . $name . '}', $value, $template);  
        }  
        return $template;  
    }  
}

?>

1. **Что такое трейт и как это используется?**

Трейт — это механизм обеспечения повторного использования кода в языках с поддержкой только одиночного наследования, таких как PHP. Трейт предназначен для уменьшения некоторых ограничений одиночного наследования, позволяя разработчику повторно использовать наборы методов свободно, в нескольких независимых классах и реализованных с использованием разных архитектур построения классов. Семантика комбинации трейтов и классов определена таким образом, чтобы снизить уровень сложности, а также избежать типичных проблем, связанных с множественным наследованием и смешиванием (mixins).

1. **Что такое магические методы? Приведите примеры.**

Магические методы — это специальные методы, которые переопределяют действие PHP по умолчанию, когда над объектом выполняются определённые действия.

Например,\_\_construct(), \_\_destruct(), \_\_call(), \_\_callStatic(), \_\_get(), \_\_set(), \_\_isset(), \_\_unset(), \_\_sleep(), \_\_wakeup(), \_\_serialize(), \_\_unserialize(), \_\_toString(), \_\_invoke(), \_\_set\_state(), \_\_clone() и \_\_debugInfo().

1. **Что такое позднее статическое связывание?**

PHP реализует функцию, называемую позднее статическое связывание, которая может быть использована для того, чтобы получить ссылку на вызываемый класс в контексте статического наследования.

Если говорить более точно, позднее статическое связывание сохраняет имя класса указанного в последнем "неперенаправленном вызове". В случае статических вызовов это явно указанный класс (обычно слева от оператора [::](https://www.php.net/manual/ru/language.oop5.paamayim-nekudotayim.php)); в случае не статических вызовов это класс объекта. "Перенаправленный вызов" - это статический вызов, начинающийся с self::, parent::, static::, или, если двигаться вверх по иерархии классов, forward\_static\_call(). Функция get\_called\_class() может быть использована для получения строки с именем вызванного класса, а static:: представляет её область действия.

Само название "позднее статическое связывание" отражает в себе внутреннюю реализацию этой особенности. "Позднее связывание" отражает тот факт, что обращения через static:: не будут вычисляться по отношению к классу, в котором вызываемый метод определён, а будут вычисляться на основе информации в ходе исполнения. Также эта особенность была названа "статическое связывание" потому, что она может быть использована (но не обязательно) в статических методах.

1. **Что такое ковариантность и контравариантность?**

Ковариантность – это сохранение иерархии наследования исходных типов в производных типах в том же порядке. Ниже приведён пример ковариантности на языке PHP.

<?php  
abstract class Animal  
{  
    protected string $name;  
    public function \_\_construct(string $name)  
    {  
        $this->name = $name;  
    }  
    abstract public function speak();  
}  
class Dog extends Animal  
{  
    public function speak()  
    {  
        echo $this->name . " лает";  
    }  
}  
class Cat extends Animal  
{  
    public function speak()  
    {  
        echo $this->name . " мяукает";  
    }  
}

interface AnimalShelter  
{  
    public function adopt(string $name): Animal;  
}  
class CatShelter implements AnimalShelter  
{  
    public function adopt(string $name): Cat // Возвращаем класс Cat вместо Animal  
    {  
        return new Cat($name);  
    }  
}  
class DogShelter implements AnimalShelter  
{

// Возвращаем класс Dog вместо Animal  
    public function adopt(string $name): Dog   
    {  
        return new Dog($name);  
    }  
}

?>

Контравариантность – это обращение иерархии исходных типов на противоположную в производных типах. Ниже приведён пример контравариантности на языке PHP.

<?php  
class Food {}  
class AnimalFood extends Food {}  
abstract class Animal  
{  
    protected string $name;  
    public function \_\_construct(string $name)  
    {  
        $this->name = $name;  
    }  
    public function eat(AnimalFood $food)  
    {  
        echo $this->name . " ест " . get\_class($food);  
    }  
}

class Dog extends Animal  
{  
    public function eat(Food $food) {  
        echo $this->name . " ест " . get\_class($food);  
    }  
}

?>

1. **Опишите понятие чистой архитектуры.**

Понятие чистой архитектуры пошло из одноименной статьи Роберта Мартина 2012 года. Оно включает в себя несколько принципов:

* Независимость от фреймворков. [Архитектура](https://apptractor.ru/info/media/appscast-3-arhitektura-arhitektura-i-eshhe-nemnogo-arhitektury.html) не должна полагаться на существование какой-либо библиотеки. Так вы сможете использовать фреймворки как инструменты, а не пытаться загнать свою систему в их ограничения;
* Тестируемость. Бизнес-логика должна быть тестируемой без любых внешних элементов вроде интерфейса, базы данных, сервера или любого другого элемента;
* Независимость от интерфейса. Интерфейс должен легко изменяться и не требовать изменения остальной системы. Например, веб-интерфейс должен заменяться на интерфейс консоли без необходимости изменения бизнес-логики;
* Независимость от базы данных. Ваша бизнес-логика не должна быть привязана и к конкретным базам данных;
* Независимость от любого внешнего агента. Ваша бизнес-логика не должна знать вообще ничего о внешнем мире.

1. **Сформулируйте правило зависимостей.**

Зависимости в исходном коде могут указывать только во внутрь. Ничто из внутреннего круга не может знать что-либо о внешнем круге, ничто из внутреннего круга не может указывать на внешний круг. Это касается функций, классов, переменных и т. д. Более того, структуры данных, используемых во внешнем круге, не должны быть использованы во внутреннем круге, особенно если эти структуры генерируются фреймворком во внешнем круге. Мы не используем ничего из внешнего круга, чтобы могло повлиять на внутренний.

1. **Чем определяются сущности, чем они могут быть?**

Сущность (entity) – это объект, который может быть идентифицирован неким способом, отличающим его от других объектов. Примеры: конкретный человек, предприятие, событие и т.д.

Сущности определяются бизнес-правилами предприятия. Сущность может быть объектом с методами или она может представлять собой набор структур данных и функций. Не имеет значения как долго сущность может быть использована в разных приложениях.

Если же вы пишете просто одиночное приложение, в этом случае сущностями являются бизнес-объекты этого приложения. Они инкапсулируют наиболее общие высокоуровневые правила. Наименее вероятно, что они изменятся при каких-либо внешних изменениях. Например, они не должны быть затронуты при изменении навигации по страницам или правил безопасности. Внешние изменения не должны влиять на слой сущностей.

1. **Что такое слой сценариев?**

В данном слое реализуется специфика бизнес-правил. Он инкапсулирует и реализует все случаи использования системы. Эти сценарии реализуют поток данных в и из слоя сущностей для реализации бизнес-правил.

Мы не ожидаем изменения в этом слое, влияющих на сущности. Мы также не ожидаем, что этот слой может быть затронут внешними изменениями, таких как базы данных, пользовательским интерфейсом или фреймворком. Этот слой изолирован от таких проблем.

Мы, однако, ожидаем, что изменения в работе приложения повлияет на Cценарии. Если будут какие-либо изменения в поведении приложения, то они несомненно затронут код в данном слое.

1. **Что такое DTO?**

Data Transfer Object (DTO) — один из шаблонов проектирования, используется для передачи данных между подсистемами приложения.

1. **Что является деталью в рамках чистой архитектуры?**

Это части структуры, которые позволяют миру взаимодействовать с бизнес-правилами. Например: веб, фреймворк.

1. **Опишите принципы организации компонентов.**

Под понятием компонентов принято понимать единицы развертывания. Например, это может быть библиотека или исполняемый файл, в Java таковым назвать можно jar-файлы. Принципы организации компонентов в разработки ПО помогают сгруппировать классы в компоненты и сделать их более структурируемыми и управляемыми. Принципы разделяются на две группы: связность (component cohesion) (какие классы стоит поместить?) и сочетаемость (component coupling) компонентов (как должны взаимодействовать друг с другом?).

Три принципа связности компонентов:

* Принцип эквивалентности повторного использования и выпусков. Единица повторного использования есть единица выпуска. Этот принцип требует, чтобы компоненты проходили процесс выпуска и получали версии. В один компонент должны объединяться классы c одной целью, которая будет выражена в очередном выпуске. По документации же пользователи и разработчики должны понимать, нужно ли им переходить на новую версию.
* Принцип согласованного изменения. В один компонент должны объединяться классы, изменяющиеся по одним причинам. Идея заключения вместе сущностей, которые могут изменяться в одно и то же время и по одним причинам, является одной из ключевых идей архитектуры ПО, поэтому она проходит красной нитью по всем структурным уровням.
* Принцип совместного повторного использования. Не вынуждайте пользователей компонента зависеть от того, чего им не требуется. Главная мысль этого принципа в объединении в компоненты тех классов, которые имеют множественные зависимости друг от друга. Кроме того, нужно избегать слабых зависимостей от других компонентов — даже зависимость от одного редко используемого класса наверняка потребует повторной компиляции и тестирования всего компонента в случае изменений в зависимом.

Теперь перейдем ко второй группе принципов. Сочетаемость компонентов:

* Принцип ацикличности зависимостей. Нельзя допускать циклов в графе зависимостей. Это позволяет разбить проект на компоненты, которые будут выпускаться независимо. При возникновении цикла между зависимостями для тестирования и выпуска новой версии придется отладить и подготовить все компоненты, входящие в цикл. Они превращаются в один большой компонент.
* Принцип устойчивых зависимостей. Зависимости должны быть направлены в сторону устойчивых компонентов. Нужно использовать ссылки на компоненты, которые будут редко меняться и избегать зависимостей от изменчивых компонентов. Это добавит гибкость в разработке, так как всегда нужны компоненты, которые можно легко изменять. Если же создать большое число зависимостей от такого компонента, эта возможность легких изменений испарится.
* Принцип устойчивости абстракций. Этот принцип проводит связь между устойчивыми и абстрактными компонентами. Компоненты, которые содержат интерфейс для высокоуровневой бизнес-логики, должны быть абстрактными и почти не меняться. Реализации же этого интерфейса должны быть неустойчивыми — избегайте множественных зависимостей от таких компонентов.

1. **Опишите принципы дизайна архитектуры.**

Под принципами дизайна архитектуры понимаются SOLID принципы. Эта аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion. В переводе на русский: принципы единственной ответственности, открытости / закрытости, подстановки Барбары Лисков, разделения интерфейса и инверсии зависимостей).

* Принцип единственной обязанности / ответственности (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.
* Принцип открытости / закрытости (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.
* Принцип подстановки Барбары Лисков (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом».
* Принцип разделения интерфейса (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.
* Принцип инверсии зависимостей (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

1. **Опишите понятие DDD (Domain Driven Design, предметно-ориентированное проектирование).**

Предметно-ориентированное проектирование — это набор принципов и схем, направленных на создание оптимальных систем объектов. Сводится к созданию программных абстракций, которые называются моделями предметных областей. В эти модели входит бизнес-логика, устанавливающая связь между реальными условиями области применения продукта и кодом.

1. **Что такое ограниченный контекст (Bounded Context)?**

Ограниченный контекст (bounded context)– это граница, которая окружает ту или иную модель. Это держит знания внутри соответствующей границы, в то же время игнорируя помехи от внешнего мира. Это выгодно по ряду причин. Во-первых, вы можете моделировать различные аспекты проблемы, не контактируя с другими частями бизнеса. Используя предыдущий пример, объект Product в рамках складской системы должен быть связан с помощью методов и свойств этой единой системы, а не какой-либо другой коммерческой фирмы, что случается, когда пытаются соответствовать объекту под названием Product. Во-вторых, терминология в ограниченном контексте (Bounded Context) может иметь одно, четкое определение, что точно описывает конкретную проблему. Различные отделы по всей компании, как правило, имеют немного разные идеи и определения аналогичных условий, это часто может сорвать проект из-за отсутствия ясности и понимания, если сроки являются неоднозначными.

1. **Что такое Ubiquitous Language (Единый язык)?**

Этот коллективный язык терминов называется - единый язык. (Ubiquitous Language). Это один из основных и самых важных шаблонов предметного-ориентированного проектирования. Это не бизнес-жаргон, навязанный разработчикам, а настоящий язык, созданный целостной командой – экспертами в предметной области, разработчиками, бизнес-аналитиками и всеми, кто вовлечен в создание системы. Роль в команде не столь существенна, поскольку каждый член команды использует для описания проекта единый язык.

1. **Что такое Смысловое ядро (Core domain)?**

Смысловое ядро – это подобласть, имеющая первостепенное значение для организации. Со стратегической точки зрения бизнес должен выделяться своим смысловым ядром. Большинство DDD проектов сосредоточены именно на смысловом ядре.

1. **Что такое Предметная область (Domain)?**

Множество понятий и объектов, рассматриваемых в пределах отдельного рассуждения, исследования или научной теории. Включает объекты, изучаемые теорией, а также свойства, отношения и функции, которые принимаются во внимание в теории. В анализе данных в качестве предметной области может выступать компания, в интересах которой реализуется аналитический проект, внешнее окружение, сегмент рынка и т. д. Это понятие играет большую роль в анализе данных, поскольку используемые там подходы и методы оперируют объектами и терминами предметной области и, следовательно, зависят от нее. В хранилищах данных, которые являются предметно ориентированными, под предметной областью понимают устойчивую связь между именами, понятиями и объектами внешнего мира, не зависящую от самой информационной системы и круга ее пользователей. Введение в рассмотрение понятия предметной области ограничивает и делает обозримым пространство информационного поиска в хранилище данных и позволяет выполнять за конечное время даже сложные нерегламентированные запросы.

1. **Что такое пространство задач и пространство решений?**

Пространство задач и пространство решений в архитектуре подразумевает под собой набор определённых задач и их решений, в зависимости от которых будет зависеть построение целой архитектуры информационной системы/приложения.

Ссылка на удаленный репозиторий проекта

Полный код проекта можно найти по ссылке:

https://github.com/Ivanbessmertnykh/Data-Servers

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. «Конспект лекции №1»: [Электронный ресурс]. URL: <https://online-edu.mirea.ru/mod/resource/view.php?id=403993> (дата обращения 08.09.2022)
2. «Руководство по Docker Compose»: [Электронный ресурс]. URL: <https://habr.com/ru/company/ruvds/blog/450312/> (дата обращения 08.09.2022)
3. «Docker Hub»: [Электронный ресурс]. URL: <https://hub.docker.com/> (дата обращения 08.09.2022)