Вопросы для подготовки к зачету

ППО сезон 2021

**Lekci 1**

1. **Что такое архитектура программного обеспечения и какие задачи она решает?**

* Архитектура программного обеспечения — совокупность важнейших решений об организации программной системы.
  + выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов
  + соединение выбранных элементов структуры и поведения во всё более крупные системы
  + общий архитектурный стиль
* Цель – уменьшение трудозатрат на создание и сопровождение

системы

* Задачи
  + Поддержание жизненного цикла системы
  + Легкость освоения
  + Простота разработки сопровождения и развертывания
  + Минимизация затрат на проект
  + Максимизация продуктивности программистов

1. **Важность “простоты изменений” в программных архитектурах**

* Изменение требований
* Исправление ошибок

1. **Существующие парадигмы программирования: виды, история развития, достоинства и недостатки, сферы применения**

* Структурное программирование
* Объектно-ориентированное программирование
* Функциональное программирование

1. **Структурное программирование. Принципы структурного программирования. Достоинства и недостатки.**

* 1968, принципы Дейкстры
* Структурное программирование накладывает ограничение на прямую передачу управления

1. **Объектно-ориентированное программирование. Принципы объектно-ориентированного программирования. Достоинства и недостатки.**

* Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления

1. **Функциональное программирование. Идеи функционального подхода. Достоинства и недостатки.**

* Функциональное программирование накладывает ограничение на рисваивание

**Lekci 2**

1. Паттерны объектно-ориентированного программирования.

паттерн — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

1. **Принципы SOLID**

* SRP: Single Responsibility Principle

Принцип единственной ответственности

* OCP: Open-Closed Principle

Принцип открытости/закрытости

* LSP: Liskov Substitution Principle

Принцип подстановки Барбары Лисков

* ISP: Interface Segregation Principle

Принцип разделения интерфейсов

* DIP: Dependency Inversion Principle

Принцип инверсии зависимости

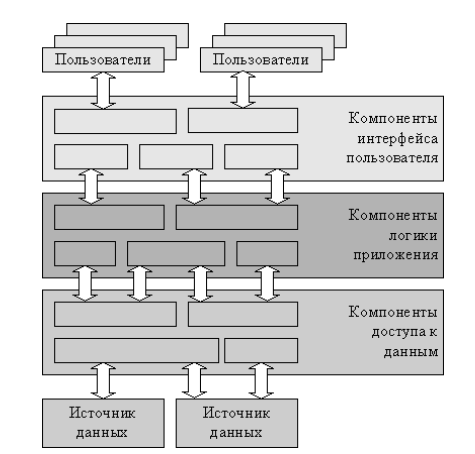
1. **Инверсия управления и внедрение зависимости**

* Инверсия управления
  + IoC - архитектурное решение интеграции, упрощающее расширение возможностей системы, при котором поток управления программы контролируется фреймворком
  + Критика:
    - логика взаимодействия программы разбросана
    - поток управления задан неявно
* Внедрение зависимости (англ. Dependency injection, DI) — процесс предоставления внешней зависимости программному компоненту.
  + В соответствии с SRP объект отдаёт заботу о построении требуемых ему зависимостей внешнему общему механизму

1. **Понятие программного компонента**

Программный компонент — программная часть системы компонент программного обеспечения

Программная компонента – это единица программного обеспечения, исполняемая на одном компьютере в пределах одного процесса, и предоставляющая некоторый набор сервисов, которые используются через ее внешний интерфейс другими компонентами, как выполняющимися на этом же компьютере, так и на удаленных компьютерах.



**LEKCI 3**

1. **Принципы связности компонентов**

Связность компонентов

• REP: Reuse/Release Equivalence Principle

(Принцип эквивалентности повторного использования и выпусков)

• CCP: Common Closure Principle (Принцип согласованного изменения)

• CRP: Common Reuse Principle

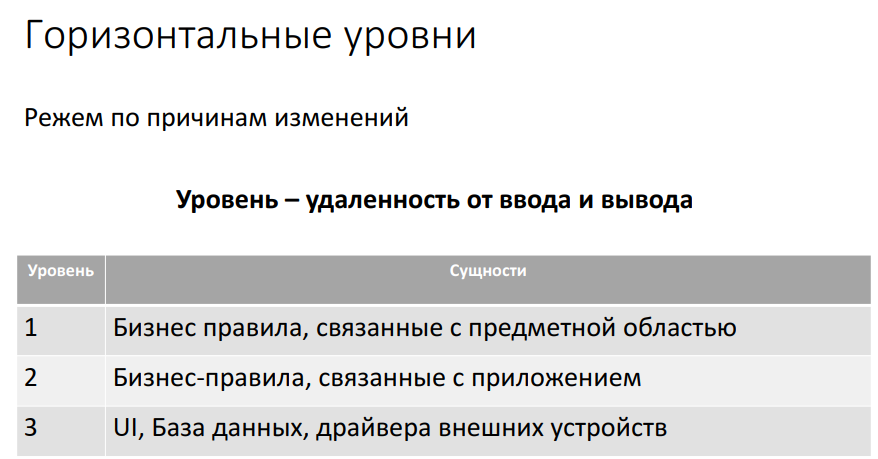
(Принцип совместного повторного использования)

1. **Принципы сочетаемости компонентов**

* Сочетаемость компонентов
  + Принцип ацикличности зависимостей
  + Принцип устойчивых зависимостей
  + Принцип устойчивости абстракций

**Lekci** 4

1. Цель и задачи хорошей архитектуры. Задачи планирования и риск-менеджмента в проектировании архитектур информационных систем
2. **Горизонтальные уровни, вертикальные срезы и границы в архитектуре информационных систем**



* Вертикальные узкие срезы
  + Режем по меняющимся и появляющимся вариантам использования
  + Срез «варианта использования»
    - - часть UI
    - - часть бизнес логики приложения
    - - часть бизнес логики предметной области
    - - часть базы данных
* Жесткость границ – вопрос выбора архитектора

1. **Чистая архитектура**

• Не зависят от фреймворков (Independent of Frameworks): Ваше приложение не должно зависить от фреймворка что вы используете.

• Тестируемые (Testable): Ваше приложение и бизнес логика должны тестироваться без всяких зависимостей от пользовательского интерфейса, базы данных или веб-API.• Не зависят от графического интерфейса (Independent of UI): Пользовательский интерфейс вашего приложения должен контролировать вашу бизнес-логику, но он не должен контролировать то как структурирован ваш поток данных.

• Не зависят от базы данных (Independent of Database): Ваше приложение не должно быть спроектированно под конкретный тип базы данных которую вы используете. Вашу бизнес-логику не должно волновать то как и где хранятся данные в БД или в памяти.

• Независимость от любого внешнего агента (Independent of any external agency): Правила Вашей бизнес-логики должны заботиться только о своих задачах и ни о чем больше что может быть в Вашем приложении.

• Явная зависимость от назначения

1. **Архитектурные паттерны прикладных приложений (MV\*)**

**MVP: Model-View-Presenter**

Признаки Presenter:

• Двухсторонняя коммуникация с View;

• View взаимодействует напрямую с Presenter, путем вызова

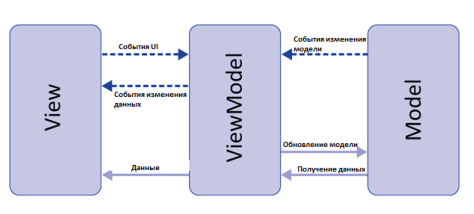
соответствующих функций или событий экземпляра Presenter;

• Presenter взаимодействует с View путем использования специального

интерфейса, реализованного View;

• Один экземпляр Presenter связан с одним View.

**MVVM: Model-View-View Model**

****

Признаки View-модели:

• Двухсторонняя коммуникация с представлением;

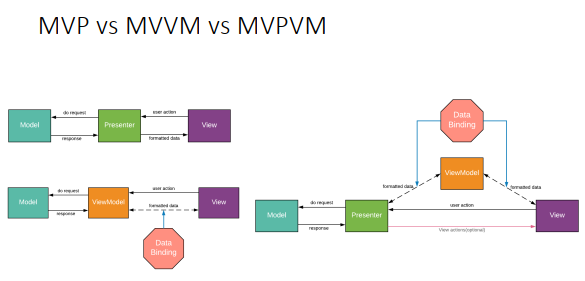
• View-модель — это абстракция View. Обычно означает, что свойства View совпадают со свойствами View-модели / модели

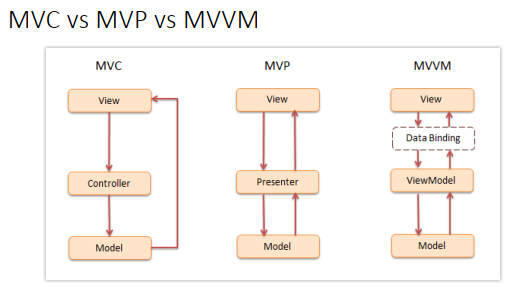
• View-модель не имеет ссылки на интерфейс представления (IView). Изменение

состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings).

• Один экземпляр View-модели связан с одним View.

Пример использования – WPF





1. **Сервис-ориентированные и микросервисные архитектуры**

**//TODO**

**Lekci 5**

1. **Жизненный цикл разработки ПО: модели.**

* Жизненный цикл разработки
  + • Водопад
  + • V-модель
  + • Итеративная разработка
  + • Agile

**«V-Model»**

Это усовершенствованная каскадная модель, в которой заказчик с

командой программистов одновременно составляют требования к

системе и описывают, как будут тестировать её на каждом этапе



Когда использовать V-модель?

• Если требуется тщательное тестирование продукта, то V-модель

оправдает заложенную в себя идею: validation and verification.

• Для малых и средних проектов, где требования четко определены

и фиксированы.

• В условиях доступности инженеров необходимой квалификации,

особенно тестировщиков.

**Итеративная модель**

Итерационная модель жизненного цикла не требует для начала полной спецификации требований. Вместо этого, создание начинается с реализации части функционала, становящейся базой для определения дальнейших требований. Этот процесс повторяется. Версия может быть неидеальна, главное, чтобы она работала.

Когда оптимально использовать итеративную модель?

• Требования к конечной системе заранее четко определены и

понятны.

• Проект большой или очень большой.

• Основная задача должна быть определена, но детали реализации

могут эволюционировать с течением времени.



**Спиральная модель**

предполагает 4 этапа для каждого витка:

1. планирование;

2. анализ рисков;

3. конструирование;

4. оценка результата и при удовлетворительном качестве переход к

новому витку.

Эта модель не подойдет для малых проектов, она резонна для сложных

и дорогих.



1. **Этап анализа в разработке ПО.**

Выделение и формализация автоматизируемых бизнеспроцессов

• Выделение Акторов

• Построение Use Case

• Формализация User Story

• Проектирование сущностей системы

1. **Этап проектирования в разработке ПО**

• Формализация архитектурной концепции

• Верхнеуровневое описание архитектуры на уровне слоев и срезов

• Декомпозиция архитектуры до уровня компонентов

• Проектирование компонентов

1. **Декомпозиция и оценка сложности задач.**

//TODO

1. **Проектная команда: роли и ответственность**

Project manager

• Tech Lead

• Team Lead

• Architect

• Expert

• Senior

• Middle

• Junior

• Intern

• Product manager

1. **Системы контроля версий и их применение в разработке ПО. Модели использования систем контроля версий. Код-ревью.**

Git

Github

Gitlab

Bitbucket

1. **Базовый (стандартный) процесс разработки продукта**

Идея

• ТЗ на MVP

• Верхнеуровневый архитектурный тех. проект, проект на MVP

• Итерации, в каждой – рабочая версия

• MVP

• Набор ТЗ/Одно ТЗ

• Детализированное проектирование архитектуры системы

• Итерации, в каждой – сохраняем работоспособность системы

• Версия системы 1.0

**Lekci 6**

1. **Подходы к процессу разработки (\*DD)**

• Разработка через тестирование (TDD - Test DD)

• Разработка через пользовательские сценарии (BDD - Behavior DD)

• Разработка на основе типов (TDD — Type DD)

• Разработка на основе features (FDD — Features DD)

• Разработка на основе модели (MDD – Model DD)

• Разработка на основе паники (PDD — Panic DD)

1. **DDD**

Предметно-ориентированное проектирование

(Domain Driven Design, DDD)

это набор принципов и схем, направленных на создание оптимальных систем объектов. Процесс разработки 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. **Рефакторинг, оптимизация и исправление ошибок в информационных системах**

Рефакторинг

Причины:

• Модификация

• Ошибки

• Проблемы с разработкой

Подход:

• Небольшие, эквивалентные преобразования

• TDD

Оптимизация

Программисты тратят огромное количество времени,

размышляя и беспокоясь о некритичных местах кода, и пытаются

оптимизировать их, что исключительно негативно сказывается на

последующей отладке и поддержке. Мы должны вообще забыть

об оптимизации в, скажем, 97% случаев; более того, поспешная

оптимизация является корнем всех зол. И напротив, мы должны

уделить все внимание оставшимся 3%.

Исправление ошибок

• Написание теста на новую ошибку

• Исправление ошибки

• Надежда на светлое будущее

1. **CI/CD. Развертывание. Контейнеризация.**

CI

• Сборка каждой ветки по каждому коммиту

• Прогонка Unit тестов в каждой ветке по каждому коммиту

• Прогонка интеграционных и системных тестов – по запросу /

автоматом в develop

CD

• Сборка релизов

• Доставка (развертывание релизов) на разных средах

• Dev

• Feature - среды

• Staging

• Production

1. **Задачи архитектора ПО**

Архитектор ПО

• Анализ всех требований

• Построение четкой ментальной картины предметной области и

требуемой функциональности

• Формализация предметной области

• Выбор архитектурного подхода

• Верхнеуровневая формализация архитектуры

• Постепенная детализация и декомпозиция до уровня программных

компонент

• Постоянный контроль соблюдения выбранного подхода

• Постоянный поиск решений «новых вызовов» - проблем и требований

1. **Язык UML для задач проектирования архитектуры ПО**

**//TODO**

1. **Сущности системы. Понятия модели базы данных и DTO.**

**//TODO**