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

Архитектура отражает важные проектные решения по формированию системы, где важность определяется стоимостью изменений

Самонадеянность, управляющая перепроектированием, приведет к тому же беспорядку что и прежде

Естественная стратегия:
как можно дольше иметь как можно больше вариантов

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

Парадигмы программирования
• Структурное программирование
• Объектно-ориентированное программирование
• Функциональное программирование

Структурное программирование накладывает ограничение на прямую передачу управления (goto)

Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления

Функциональное программирование накладывает ограничение на присваивание

Уровни проектирования
• Уровень функций и методов
• Уровень классов
• Уровень организации компонентов • Архитектурный уровень

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.

ПК - Это единица развертывания (DLL GEM JAR NPM EXE).
• Независимое развертывание
• Независимая разработка

Развёртывание программного обеспечения— это все действия, которые делают программную систему готовой к использованию.

Из чего состоит программный компонент?

Из хорошо спроектированных программных структур среднего уровня.

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

Принцип единственной ответственности
Класс должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой. Модуль должен иметь одну и только одну причину для изменения

Несоблюдение:
• Проблема модификации общих частей •
• Проблема слияния изменений

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
    saveAnimal(a: Animal) { }
}
```

Он же решает две, занимаясь работой с хранилищем данных в методе `saveAnimal` и манипулируя свойствами объекта в конструкторе и в методе `getAnimalName`.
Если изменится порядок работы с хранилищем данных, используемым приложением, то придётся вносить изменения во все классы, работающие с хранилищем.

Принцип открытости-закрытости
Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

Цель: легкая расширяемость и безопасность от влияния изменений
Упорядочивание в иерархию, защищающую компоненты уровнем

выше от изменения в компонентах уровнем ниже

Принцип подстановки Барбары Лисков
Необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов.
Цель этого принципа заключаются в том, чтобы классы-наследники могли бы использоваться вместо родительских классов, от которых они образованы, не нарушая работу программы. Если оказывается, что в коде проверяется тип класса, значит принцип подстановки нарушается.

Простое нарушение совместимости может вызвать загрязнение архитектуры системы значительным количеством дополнительных механизмов.

Принцип разделения интерфейса
Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют. Зависимости, несущие лишний груз ненужных и неиспользуемых особенностей, могут стать причиной неожиданных проблем.
Этот принцип направлен на устранение недостатков, связанных с реализацией больших интерфейсов.

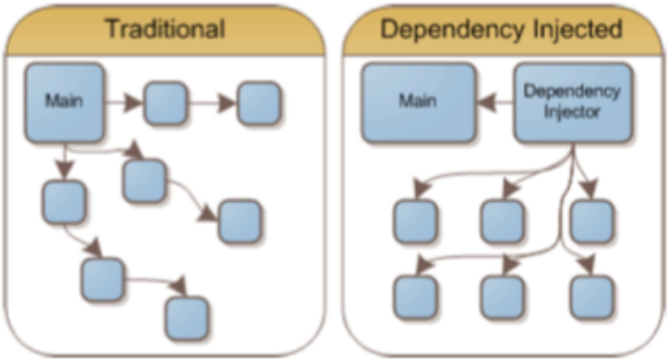
Принцип инверсии зависимостей
Объектом зависимости должна быть абстракция, а не что-то конкретное.

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

В процессе разработки программного обеспечения существует момент, когда функционал приложения перестаёт помещаться в рамках одного модуля. Когда это происходит, нам приходится решать проблему зависимостей модулей. В результате, например, может оказаться так, что высокоуровневые компоненты зависят от низкоуровневых компонентов.

Инверсия управления IoC
В обычной программе программист сам решает, в какой последовательности делать вызовы процедур. Но, если используется фреймворк, программист может разместить свой код в определенных

точках (используя другие затем «главную фреймворка, обеспечит и вызовет



выполнения callback или механизмы), запустить функцию» которая все выполнение код программиста тогда, когда это будет необходимо. Как следствие, происходит утеря контроля над выполнением кода — это и называется **инверсией управления** (фреймворк управляет кодом программиста, а не программист управляет фреймворком).

Критика:
- логика взаимодействия программы разбросана по отдельным обработчикам событий или классам;
- поток управления задан неявно и использует общее состояние (shared state) обработчиков событий.

Внедрение зависимости (англ. *Dependency injection*, DI) — процесс предоставления внешней зависимости программному компоненту. Является специфичной формой «инверсии управления» (англ. *Inversion of control*, IoC), когда она применяется к управлению зависимостями. В полном соответствии с принципом единственной ответственности объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

История
- Неперемещаемые библиотеки
- Перемещаемые библиотеки (связывающий загрузчик)
- Компоновщик (редактор связей) + загрузчик

Компоновщик (или редактор связей) предназначен для связывания между собой объектных файлов, порождаемых компилятором, а также файлов библиотек, входящих в состав системы программирования.

Загрузочный модуль – программный модуль, пригодный для загрузки и выполнения, получаемый из объектного модуля при редактировании связей и представляющий собой программу в виде последовательности машинных команд.

Программные компоненты – динамически связываемые файлы, которые можно подключать во время выполнения (связывающий загрузчик)

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)

REP: *Reuse/Release Equivalence Principle – Принцип эквивалентности повторного использования и выпусков.*

С точки зрения архитектуры и дизайна этот принцип означает, что классы и модули, составляющие компонент, должны принадлежать связной группе. Компонент не может просто включать случайную смесь классов и модулей; должна быть какая-то тема или цель, общая для всех модулей.

Классы и модули, объединяемые в компонент, должны выпускаться вместе.
«Выпуск»:
• Номер версии
• Описание новой версии
• Change Log (Лог изменений)
Единица повторного использования == Единица выпуска

CCP: *Common Closure Principle – принцип согласованного изменения.*
Развитие принципов «единственной ответственности» (SRP) и «открытости/закрытости» (OCP) из SOLID
Изменение требований => изменение МИНИМАЛЬНОГО количества компонентов

В один компонент должны включаться классы, изменяющиеся по одним и тем же причинам и в одно и то же время. В разные компоненты должны включаться классы, изменяющиеся в разное время и по разным причинам.

(Для большинства приложений простота сопровождения важнее возможности повторного использования)

CRP: *Common Reuse Principle – принцип совместного повторного использования.*
Этот принцип имеет схожесть с ISP (interface segregation).
Не вынуждайте пользователей компонента зависеть от того, чего им не требуется. Принцип указывает, что классы не имеющие тесной связи, не должны включаться в один компонент.

Принципы эквивалентности повторного использования (REP) и согласованного изменения (CCP) являются включительными: оба стремятся сделать компоненты как можно крупнее. Принцип повторного использования (CRP) – исключительный, стремящийся сделать компоненты как можно мельче.

Сочетаемость компонентов
- Принцип ацикличности зависимостей
- Принцип устойчивых зависимостей
- Принцип устойчивости абстракций

ADP: *Acyclic Dependencies Principle — принцип ацикличности зависимостей.*
Циклы в графе зависимостей компонентов недопустимы!
Отдельные компоненты – отдельные разработчики/команды Появление цикла – появление одного БОЛЬШОГО компонента

Разрыв цикла:
1. Применить принцип DIP

2. Создать новые компонент, от которого зависят проблемные

Проектирование сверху вниз?

- Граф зависимостей формируется для защиты стабильных и ценных компонентов от влияния изменчивых компонентов

По мере развития приложения мы начинаем беспокоиться о создании элементов многократного пользования. На этом этапе на состав компонентов начинает влиять принцип совместного повторного использования (CRP). Наконец, с появлением циклов мы начинаем применять принцип ацикличности зависимостей (ADP), в результате начинает изменяться и разрастаться граф зависимостей компонентов.

Stable Dependencies Principle (SDP) — Принцип устойчивых зависимостей Зависимости должны быть направлены в сторону устойчивости.

SDP говорит, что от модулей, которые изначально спроектированы так, чтобы их легко можно было изменять, не должны зависеть модули, изменение которых затруднено.

Метрика неустойчивости = выходы / (входы + выходы)
Метрика неустойчивости компонента должна быть выше метрик неустойчивости компонентов, от которых он зависит

Stable Abstractions Principle (SAP) — Принцип устойчивых абстракций Пакет должен быть столь же абстрактным, сколь и устойчивым

Он говорит, что устойчивый компонент должен быть также и абстрактным, чтобы устойчивость не препятствовала его расширению. То есть стабильный компонент должен состоять из интерфейсов и абстрактных классов, чтобы его легко было расширять. Устойчивые компоненты, доступные для расширения, обладают достаточной гибкостью, чтобы не накладывать чрезмерные ограничения на архитектуру.

Зависимость разных аспектов проекта от архитектуры

Сверхсильная зависимость

- Сопровождение

(После написания сопровождать программу тоже, как правило, приходится людям, не участвовавшем в ее разработке. Поэтому хорошая архитектура должна давать возможность относительно легко и быстро разобраться в системе новым людям.)

Сильная зависимость

- Разработка, развертывание

(Архитектура должна позволять распараллелить процесс разработки, так чтобы множество людей могли работать над программой одновременно.)
Слабая зависимость
• Эффективность(Практически любые проблемы эффективности можно решить вводом в систему нового аппаратного обеспечения без существенного влияния на ее архитектуру)
Почти нулевая зависимость
• Функциональность

Горизонтальные уровни

Режем по причинам изменений
Уровень – удаленность от ввода и вывода
• UI
• Бизнес-правила, связанные с приложением
• Бизнес правила, связанные с предметной областью
• База данных

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

Тонкости дублирования

• Устранение истинного дублирования
• Похожие сущности и алгоритмы в разных уровнях и срезах – это нормально

Режимы разделения

• Уровень исходного кода
• Уровень развертывания
• Уровень локального независимого выполнения
• Уровень служб

Вернемся к режимам. Существует много разных способов деления на уровни и варианты использования. Деление можно выполнить на уровне исходного кода, на уровне двоичного кода (развертывания) и на уровне единиц выполнения (служб).

- **Уровень исходного кода**. Мы можем так настроить зависимости между модулями с исходным кодом, чтобы изменения в одном модуле не вынуждали производить изменения в других (например, Ruby Gems).

  При использовании этого режима разделения все компоненты выполняются в общем адресном пространстве и взаимодействуют, просто вызывая функции друг друга. То есть имеется единственный выполняемый файл, загружаемый в память компьютера. Люди часто называют это монолитной структурой.

- **Уровень развертывания**. Мы можем так настроить зависимости между единицами развертывания, jar-файлами или динамически загружаемыми библиотеками, чтобы изменения в исходном коде в одном модуле не вынуждали производить повторную сборку и развертывание других.

  Многие компоненты могут находиться в общем адресном пространстве и взаимодействовать, вызывая функции друг друга. Другие компоненты могут выполняться в других процессах на той же машине и взаимодействовать посредством механизмов межпроцессных взаимодействий, сокетов или разделяемой памяти. Важно отметить, что в этом случае разделенные компоненты находятся в независимых единицах развертывания, таких как jar-, gem-файлы или динамически загружаемые библиотеки.

- **Уровень служб**. Мы можем ограничить зависимости до уровня структур данных и взаимодействовать, обмениваясь исключительно сетевыми пакетами, чтобы каждая единица выполнения была по-настоящему независимой от изменений в исходном и двоичном коде в других (как, например, службы и микрослужбы).

Разработка архитектуры – искусство проведения разделяющих линий–

Отделять линиями нужно все, что не имеет значения. Графический интерфейс не имеет значения для бизнес-правил, поэтому между ними нужно провести границу. База данных не имеет значения для графического интерфейса, поэтому между ними нужно провести границу. База данных не имеет значения для бизнес-правил, поэтому между ними нужно провести границу.

границ. Границы отделяют программные элементы друг от друга и избавляют их от необходимости знать, что находится по ту сторону .
Жесткость границ – вопрос выбора архитектора
Архитектура плагинов:
• Независимые высоукоровневые компоненты • Граница
• Зависимые низкоуровневые компоненты

Что такое 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**

Основные сущности: Это простые модели данных, которые по существу необходимы для представления нашей основной логики, построения потока данных и обеспечения работы нашего бизнес-правила.

Случаи использования (Usecases): Они построены на основе основных сущностей и реализуют всю бизнес-логику приложения.

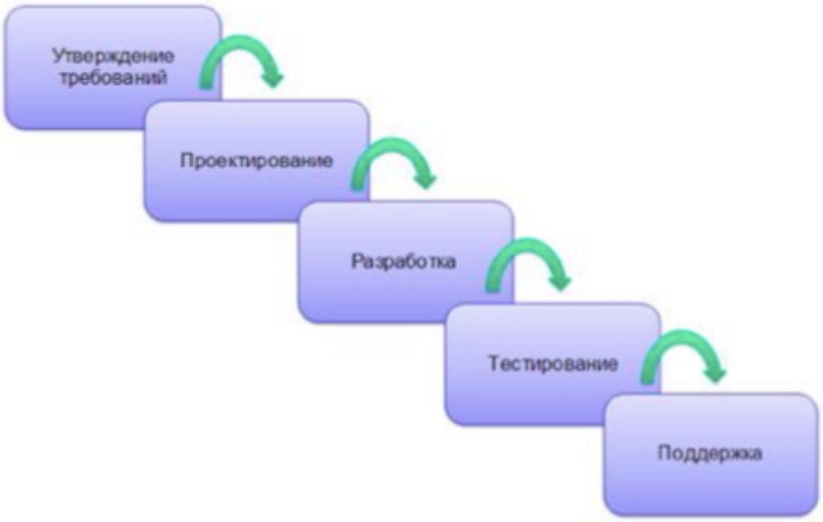
Правила зависимости (Dependency rule):Каждый уровень должен иметь доступ только к нижестоящему уровню. Так что уровень usecase должен использовать только entities которые определены в уровне

entity, и контроллер должен использовать только usecase-ы из уровня usecase который находится ниже.

Микросервисная и SOA архитектура – спасение?
- Это не архитектура
- Это не спасение

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

Водопад
Одна из самых старых, подразумевает последовательное прохождение стадий, каждая из которых должна завершиться полностью до начала следующей. В модели Waterfall легко управлять проектом. Благодаря её жесткости, разработка проходит быстро, стоимость и срок заранее определены. Но это палка о двух концах. Каскадная модель будет давать отличный результат только в проектах с четко и заранее определенными требованиями и способами их реализации. Нет возможности сделать шаг назад, тестирование начинается только после того, как разработка завершена или почти завершена.



Когда использовать каскадную методологию?

- Только тогда, когда требования известны, понятны и зафиксированы. Противоречивых требований не имеется.

- Нет проблем с доступностью программистов нужной квалификации.
- В относительно небольших проектах.

«**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.

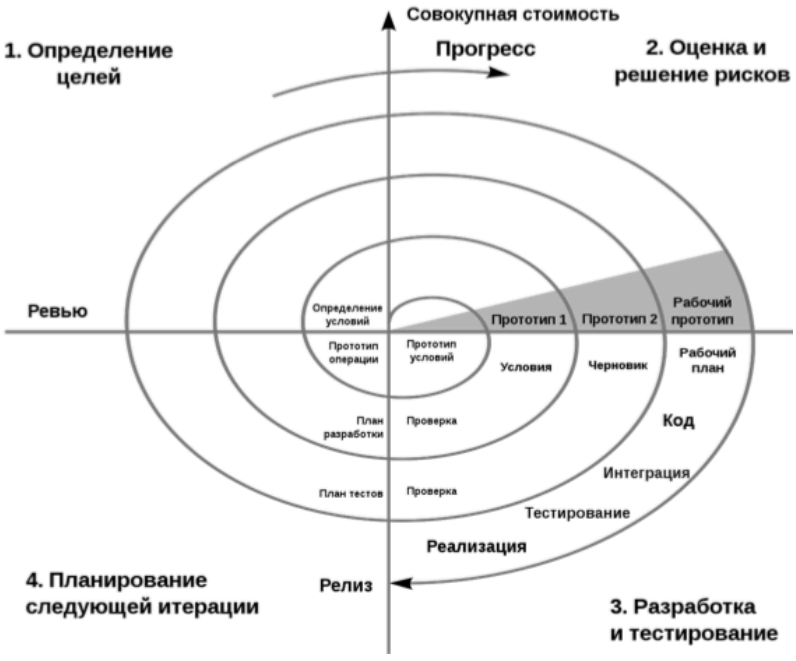
Когда оптимально использовать итеративную модель?

- 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.



Agile
Основные идеи:
• люди и взаимодействие важнее процессов и инструментов;
• работающий продукт важнее исчерпывающей документации;
• сотрудничество с заказчиком важнее согласования условий контракта;
• готовность к изменениям важнее следования первоначальному плану.

Agile принципы
• Ранняя и бесперебойная поставка ПО
• Готовность к изменениям требований
• Частая поставка рабочего ПО
• Тесное общение с заказчиком
• Мотивированность
• Личный разговор как средство коммуникации
• Работающее ПО
• Постоянный темп
• Постоянное улучшение технического мастерства
• Простота
• Самоорганизованная команда
• Постоянная адаптация к изменяющимся обстоятельствам

Agile: XP
• Короткий цикл обратной связи (Fine-scale feedback)
• Разработка через тестирование (Test-driven development)
• Игра в планирование (Planning game)
• Заказчик всегда рядом (Whole team, Onsite customer)
• Парное программирование (Pair programming)
• Непрерывный, а не пакетный процесс
• Непрерывная интеграция (Continuous integration)
• Рефакторинг (Design improvement, Refactoring)
• Частые небольшие релизы (Small releases)
• Понимание, разделяемое всеми
• Простота проектирования (Simple design)
• Метафора системы
• Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)
• Стандарт оформления кода (Coding standard or Coding conventions)
• Социальная защищённость программиста (Programmer welfare)
• 40-часовая рабочая неделя (Sustainable pace, Forty-hour week)

Scrum – это «подход структуры». Над каждым проектом работает универсальная команда специалистов, к которой присоединяется еще два человека: владелец продукта и scrum-мастер. Первый соединяет

команду с заказчиком и следит за развитием проекта; это не формальный руководитель команды, а скорее куратор. Второй помогает первому организовать бизнес-процесс: проводит общие собрания, решает бытовые проблемы, мотивирует команду и следит за соблюдением scrum-подхода.

ТЗ

Документ, регламентирующий требования, предполагаемый вид, устройство и процесс разработки продукта, условия приемки.

- Цели
- Задачи
- Требования (по категориям) - Условия приемки
- Сроки
- Бюджет

Что такое Покер планирования (Planning Poker, Scrum poker)?

Покер планирование — это гибкая техника, которая позволяет на основе коллегиальности (консенсуса) четко оценить сложность и объем задач, которые предстоит решить в ходе создания программного продукта. При этом к оценке привлекают всех: программистов, команду тестеров, инженеров баз данных, аналитиков, дизайнеров и всех других сотрудников, участвующих в проекте.

Scrum-практики

- Итерации с выпуском версий
- Стендапы
- Бэклог(полный список всех требований)
- Высокая вовлеченность в проект

Подходы к процессу разработки (* Driven Development)

- 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)

TDD — Test Driven Development

TDD — это методология разработки ПО, которая основывается на повторении коротких циклов разработки: изначально пишется тест, покрывающий желаемое изменение, затем пишется программный код, который реализует желаемое поведение системы и позволит пройти написанный тест. Затем проводится рефакторинг написанного кода с постоянной проверкой прохождения тестов.

BDD — Behaviour Driven Development

TDD — это больше о программировании и тестировании на уровне технической реализации продукта, когда тесты создают сами

разработчики. BDD предполагает описание тестировщиком или аналитиком пользовательских сценариев на естественном языке — если можно так выразиться, на языке бизнеса.

TDD — Type Driven Development

При разработке на основе типов ваши типы данных и сигнатуры типов являются спецификацией программы. Типы также служат формой документации, которая гарантированно обновляется.

Типы представляют из себя небольшие контрольные точки, благодаря которым, мы получаем множество мини-тестов по всему нашему приложению. Причем затраты на создание типов минимальны и актуализировать их не требуется, так как они являются частью кодовой базы.

DDD — Domain Driven Design

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.

FDD — Features Driven Development

Основной целью данной методологии является разработка реального, работающего программного обеспечения систематически, в поставленные сроки.

Как и остальные адаптивные методологии, она делает основной упор на коротких итерациях, каждая из которых служит для проработки определенной части функциональности системы.

MDD — Model Driven Development

Если говорить проще, то вся суть разработки сводится к построению необходимых диаграмм, из которых впоследствии мы генерируем рабочий код проекта.

PDD — Panic Driven Development

Новые задачи приоритетнее старых. Пишите столько кода, сколько нужно, чтобы решить проблему. Тесты должны писаться в конце.

Рефакторинг

Причины:

• Модификация
• Ошибки
• Проблемы с разработкой

Подход:

• Небольшие, эквивалентные преобразования
• TDD

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

Архитектура: вчера, сегодня, завтра
• Ничего не меняется
• Новые вызовы – новые решения
• Новые возможности – новые решения

SOA (Service Oriented Architecture) - концепция сервис-ориентированной архитектуры, предназначенная для решения вопросов интеграции информационной инфраструктуры компании за счет построения архитектуры, позволяющей интегрировать с максимальной гибкостью разнородные приложения.
Сервис-ориентированная архитектура строится за счет проектирования и разработки сервисов и средств их подключения. Сервис представляет собой определенную работу или бизнес-функцию, предназначенную для обеспечения согласованной работы приложений.

SOA
• Web-службы
• Тренд к «микро»
• HTTP + REST
• Независимые, переиспользуемые компоненты
• Docker-way

API расшифровывается как «Application Programming Interface» (интерфейс программирования приложений, программный интерфейс приложения) — описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой.
API включает в себя:

- саму операцию, которую мы можем выполнить,
- данные, которые поступают на вход,

- данные, которые оказываются на выходе (контент данных или сообщение об ошибке).

• API библиотек
• API ОС
• API прикладного ПО
• API сервиса

API сервиса
Если сервис – значит взаимодействие через сеть
• Прикладной протокол поверх TCP/UDP
• XML RPC (XML на HTTP 1.1)
• SOAP (XML на TCP/FTP/SMTP/HTTP)
• REST (JSON на HTTP 1.1)
• JSON-RPC (JSON на HTTP 1.1)
• GRPC (Protobuf на HTTP2)
• WebSocket

REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

REST
1. Модель клиент-сервер
2. Отсутствие состояния
3. Кэширование
4. Единообразие интерфейса
• Идентификация ресурсов
• Манипуляция ресурсами через представление
• «Самоописываемые» сообщения
5. Слои
6. Код по требованию

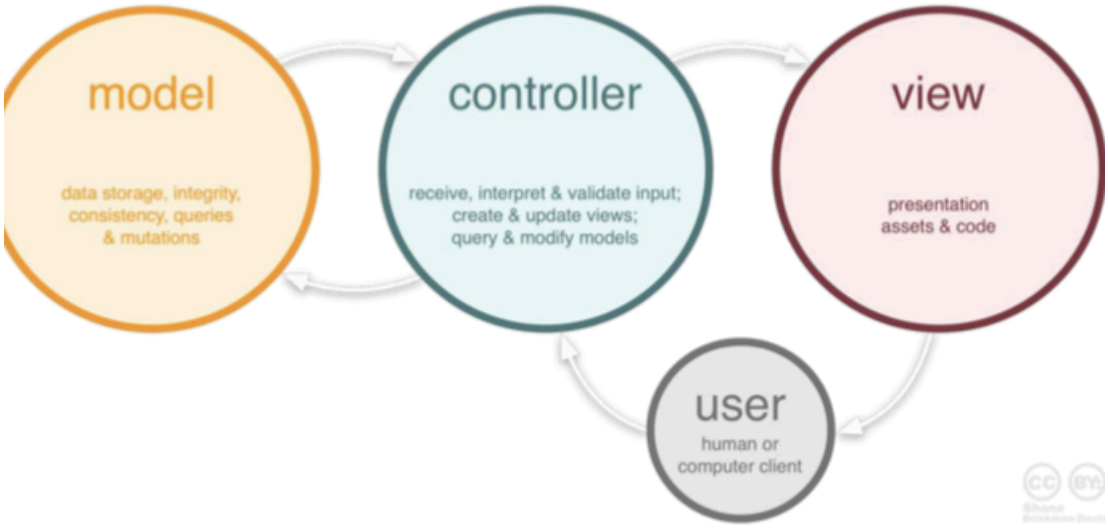
Для каждой единицы информации (info) определяется 5 действий. А именно:

GET /info/ *(Index)* – получает список всех объектов. Как правило, это упрощенный список, т.е. содержащий только поля идентификатора и названия объекта, без остальных данных.
GET /info/{id} *(View)* – получает полную информацию о объекте.

PUT /info/ или **POST /info/** *(Create)* – создает новый объект. Данные передаются в теле запроса без применения кодирования, даже urlencode.
POST /info/{id} или **PUT /info/{id}** *(Edit)* – изменяет данные с идентификатором {id}, возможно заменяет их. Данные так же передаются в теле запроса, но в отличие от PUT здесь есть некоторый нюанс. Дело в том, что POST-запрос подразумевает наличие urldecoded-post-data. Т.е. если не применять кодирования – это нарушение стандарта.
DELETE /info/{id} *(Delete)* – удаляет данные с идентификатором {id}.

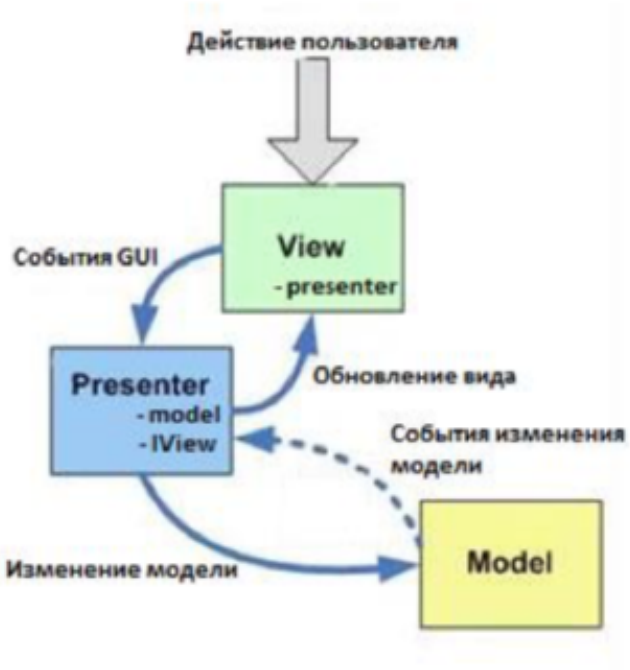
MVC
Под Моделью, обычно понимается часть содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей продукта. Модельный слой ничего не должен знать об элементах дизайна, и каким образом он будет отображаться. Достигается результат, позволяющий менять представление данных, то как они отображаются, не трогая саму Модель.

В обязанности Представления входит отображение данных полученных от Модели. Однако, представление не может напрямую влиять на модель. Можно говорить, что представление обладает доступом «только на чтение» к данным.



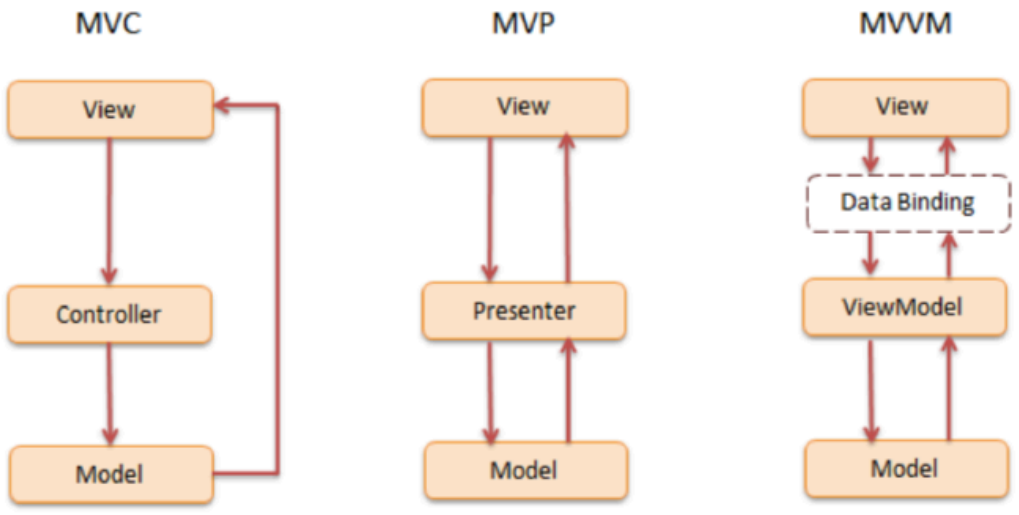
MTV

MVP
Задача MVP — тоже сделать Представления повторно используемыми. Для этого каждый View реализует определённый интерфейс и реализует механизм событий для обратной связи с Presenter'ом.



Для MVC — это там, где Представление обновляется каждый раз по какому-либо событию, а для MVP, когда Представление не нужно каждый раз пересоздавать.

MVVM
отличием является явное использование возможностей связывания данных (*databinding*) ViewModel не может общаться со View напрямую. Вместо этого она представляет легко связываемые свойства и методы в виде команд. View может привязываться к этим свойствам, чтобы получать информацию из ViewModel и вызывать на ней команды (методы).



Классификации клиент-серверных архитектур в Вебе

- MPA-SPA
- Толстый-Тонкий
- Изоморфный

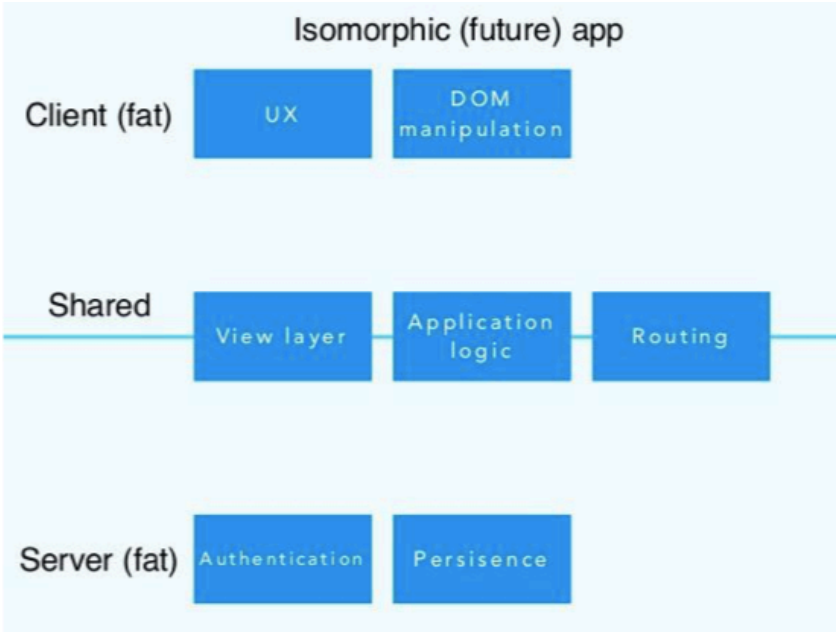
Multi Page Application (MPA)
Многостраничные приложения имеют более классическую архитектуру. Каждая страница отправляет запрос на сервер и полностью обновляет все данные. Даже если эти данные небольшие. Таким образом тратится производительность на отображение одних и тех же элементов. Соответственно это влияет на скорость и производительность.

Single Page Application (SPA)
Одностраничные приложения позволяют имитировать работу десктоп приложений. Архитектура устроена таким образом, что при переходе на новую страницу, обновляется только часть контента. Таким образом, нет необходимости повторно загружать одни и те же элементы. На практике это означает, что пользователь видит в браузере весь основной контент, а при прокрутке или переходах на другие страницы, вместо полной перезагрузки нужные элементы просто подгружаются.

SPA

Достоинства	Недостатки
Быстрая загрузка/обновление страниц	Тяжелые фреймворки
Удобство пользователя	Интерпретируемые языки
Изолирование фронтенда и бэкенда	SEO

Изоморфное приложение
в изоморфном приложении большая часть кода должна быть общей – как для клиентской, так и для серверной части приложения.



Там где возможно – MPA
• Скорость разработки
• Скорость работы
• Нет проблем с SEO
• Дешево
Сайты-визитки, персональные страницы, сайты компаний

Там где невозможно MPA - SPA
• Гибкая архитектура
• Максимальное удобство пользователя
• Простота поддержки и модификации
Любое веб-приложение, веб-интерфейс сервисов и т.д.

Выбор архитектуры: от задач
Там где невозможно SPA – изоморфное
• Одни достоинства
• Слишком модно и современно • Дорого
• Необходимо любить JS