Содержание

1 Вопросы – ответы

1.1 Архитектура программного обеспечения. Цель и задачи.

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

1.2 Существующие парадигмы программирования. Структурное программирование. Принципы структурного программирования. Достоинства и недостатки. Функциональное программирование. Идеи функционального подхода. Достоинства и недостатки.

Парадигмы программирования:

- **Структурное** программирование накладывает ограничение на прямую передачу управления (goto)
- **Объектно-ориентированное** программирование накладывает ограничение на косвенную передачу управления
- **Функциональное** программирование накладывает ограничение на присваивание

Структурное программирование

Принципы

1) Отказ от безусловного перехода.
2) Любая программа строится из трех базовых УК: последовательность, ветвление, цикл.
3) Базовые УК могут быть вложены друг в друга произвольным образом.
4) Декомпозиция в виде подпрограмм.
5) Каждую логически законченную группу инструкций следует оформить как блок.
6) Все перечисленные конструкции должны иметь один вход и один выход.
7) Разработка программы ведется пошагово "сверху вниз".

Следствия

1) Запрет на глобальные переменные.
2) Не более одного выхода их функции.
3) Не более одного выхода из цикла.
4) Вложенность любых блоков не превышает 4.
5) Размер функции 40-50 строк.
6) Декомпозиция: управление зависимостями (пример: разделение ввода, обработки и вывода данных).

Преимущества структурного программирования:

- Уменьшение вероятности воздействия одной функции на другую.
- Увеличение читабельности кода, поскольку глобальные переменные заменяются на локальные переменные и параметры.
- Уменьшение количества ошибок благодаря строгим правилам для проектирования, кодирования и отладки программы.
- Улучшение тестируемости программы благодаря использованию модульного подхода.
- Возможность использования функциональной абстракции, что упрощает понимание программы без знания ее внутренней реализации.

Недостатки структурного программирования:

- Структурированные программы часто содержат повторяющийся код и могут быть более длинными, чем неструктурированные программы.
- Структурированные программы могут использовать больше памяти и могут выполняться медленнее.
- Ограничение в использовании только трех базовых структур управления потоком может привести к сложности в разработке некоторых алгоритмов.
- Строгие правила структурного программирования могут привести к излишней сложности

в проектировании и кодировании программы.

Функциональное программирование — это парадигма программирования, в которой основной упор делается на использование математических функций. Основные идеи функционального подхода включают в себя:

- Функции как основной строительный блок программы. Функции в функциональном программировании рассматриваются как математические функции, которые принимают некоторые аргументы и возвращают результат. Функции в функциональном программировании не имеют побочных эффектов и не изменяют состояние программы.
- Функции высшего порядка. Функции высшего порядка - это функции, которые могут принимать другие функции в качестве аргументов или возвращать функции в качестве результатов.
- Рекурсия. Рекурсия — это процесс, при котором функция вызывает саму себя. Рекурсия является важной частью функционального программирования и используется для обхода структур данных, таких как списки и деревья.
- Неизменяемость. В функциональном программировании данные считаются неизменяемыми, то есть они не могут быть изменены после создания. Вместо изменения данных создается новый объект с измененным значением.
- Чистота функций. Чистые функции - это функции, которые возвращают результат только на основе своих аргументов и не имеют побочных эффектов. Чистые функции могут быть легко тестированы и переиспользованы.

Достоинства функционального программирования:

- Более простой и понятный код благодаря использованию математических функций и чистых функций.
- Более высокая надежность программы благодаря отсутствию побочных эффектов и использованию неизменяемых данных.
- Более простое тестирование и отладка программы благодаря чистым функциям и отсутствию побочных эффектов.
- Более простая параллельная обработка данных благодаря отсутствию состояния программы.

Недостатки функционального программирования:

- Не все задачи могут быть решены функциональным подходом, так как он не подходит для решения задач, которые требуют изменения состояния программы.
- Рекурсия может привести к переполнению стека и ухудшению производительности программы.

- Использование функций высшего порядка и рекурсии может привести к более сложному коду и ухудшению читаемости программы.

1.3 Паттерны ООП (GoF)

Шаблоны проектирования GoF (Gang of Four) - это наборы повторяющихся решений для общих проблем проектирования программного обеспечения.

Шаблоны проектирования GoF включают в себя:

- Порождающие шаблоны. Эти шаблоны используются для создания объектов:
  - Абстрактная фабрика
  - Строитель
  - Прототип
  - Фабричный метод
  - Одиночка
  - Статическая фабрика
  - Стратегия
  - Строительный блок
  - Менеджер зависимостей
- Структурные шаблоны. Эти шаблоны используются для организации классов и объектов:
  - Адаптер
  - Мост
  - Декоратор
  - Компоновщик
  - Фасад
  - Приспособленец
  - Заместитель
  - Легковес
  - Мост
- Поведенческие шаблоны. Эти шаблоны используются для управления взаимодействием между объектами:
  - Цепочка обязанностей
  - Команда
  - Наблюдатель
  - Итератор
  - Посетитель
  - Состояние

- Шаблонный метод
- Цепочка ответственности

Достоинства использования шаблонов проектирования GoF:

- Улучшение качества кода благодаря использованию проверенных решений для общих проблем проектирования.
- Улучшение читаемости кода благодаря использованию стандартных шаблонов.
- Улучшение повторного использования кода благодаря использованию стандартных шаблонов.
- Улучшение расширяемости программы благодаря использованию шаблонов, которые позволяют легко добавлять новый функционал.

Недостатки использования шаблонов проектирования GoF:

- Излишнее использование шаблонов может привести к избыточности кода и усложнению проекта.
- Некоторые шаблоны могут быть сложными для понимания и реализации, особенно для новичков в программировании.

1.4 Программные структуры нижнего уровня. Принципы SOLID.

Уровни проектирования

- уровень функций и методов
- уровень классов;
- уровень организации компонентов;
- архитектурный уровень

Все разговоры о проектировании компонентов сводятся к вопросу зависимости компонентов.

Программный компонент

Программный компонент — программная часть системы.

Программный компонент — это единица развертывания (.jar, .gem, .dll).

- независимое развертывание
- независимая разработка

Исполняемый файл развертывается как одна единица.

Программный компонент состоит из хорошо спроектированных программных структур среднего уровня.

Принципы SOLID

- SRP – single responsibility
- OCP – open-closed

- LSP – Liskov substitution
- ISP – Interface Segregation
- DIP – Dependency Inversion

SRP. Принцип Единственной Ответственности

- модуль должен отвечать за одного и только за одного актора;
- модуль должен иметь одну и только одну причину для изменения.

Универсальный подход

- уровень компонентов – принцип согласованного изменения (CCP)
- архитектурный уровень – принцип оси изменения (Axis of Change)

Два главных вопроса:

Что от чего зависит? Что разделяем, а что сливаем?

OCP. Принцип открытости/закрытости

Программные сущности должны быть открыты для расширения и закрыты для изменения.

Цель: легкая расширяемость и безопасность от влияния изменений.

Упорядочивание в иерархию, защищающую компоненты уровнем выше от изменения в компонентах уровнем ниже.

Чем выше политики – тем выше защита.

LSP. Принцип подстановки Барбары Лисков

Если для каждого объекта o1 типа S существует такой объект o2 типа T, что для всех программ P, определенных в терминах T, поведение P не изменяется при подстановке o1 вместо o2, то S является подтипом T.

Идея принципа состоит в том, что все реализации этого интерфейса, помимо того, что выглядят одинаково, сам метод должен вести себя одинаково (взаимодействовать с внешним миром одинаково). (связано с механизмом исключений)

Механизм исключений может приводить к тому, что разные реализации одного и того же интерфейса могут вести себя по разному.

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

ISP. Принцип разделения интерфейсов

Зависимости, несущие лишний груз ненужных и неиспользуемых особенностей, могут стать причиной неожиданных проблем.

DIP. Принцип инверсии зависимостей

Для максимальной гибкости зависимости должны быть направлены на абстракции, а не на конкретные реализации.

Несмотря на направление стека вызовов, можно инвертировать зависимость.

1.5 Инверсия управления и внедрение зависимости.

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 — процесс предоставления внешней зависимости программному компоненту.

В соответствии с SRP объект отдает заботу о построении требуемых ему зависимостей общему механизму.

1.6 Программные структуры среднего уровня. Понятие программного компонента. Принципы связности компонентов.

Программный компонент

Программный компонент — программная часть системы.

Программный компонент — это единица развертывания (.jar, .gem, .dll).

- независимое развертывание
- независимая разработка

Исполняемый файл развертывается как одна единица.

Программный компонент состоит из хорошо спроектированных программных структур среднего уровня.

Принципы связности компонентов:

REP: Reuse/Release Equivalence Principle (Принцип эквивалентности повторного использования и выпусков)

"Выпуск": номер версии, описание новой версии, лог изменений.

Единица повторного использования == единица выпуска.

Классы и модули, объединяемые в компонент, должны выпускаться *вместе*.

Объединение в один выпуск должно иметь *смысл* для автора и пользователей.

(мажорная версия программы, ломает обратную совместимость).(минорная версия, обратная совместимость).(номер билда, исправление багов)

CCP (Принцип согласованного изменения)

Развитие принципов "единственной ответственности"(SRP) и "открытости/закрытости"(OCP) из SOLID.

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

«Для большинства приложений простота сопровождения важнее возможности повторного использования»

Идея: объединение в компонент классов, закрытых для одного и того же вида изменения.

Изменение требований => изменение ***МИНИМАЛЬНОГО*** количество компонентов

Принцип совместного и повторного использования (CRP)

Развитие принципа разделения интерфейсов из SOLID.

- не вынуждайте пользователей компонента зависеть от того, что им не требуется;
- классы, не имеющие тесной связи, не должны включаться в компонент.

Не создавайте зависимостей от чего-то неиспользуемого.

1.7 Программные структуры среднего уровня. Понятие программного компонента. Принципы сочетаемости компонентов.

Программный компонент

Программный компонент — программная часть системы.

Программный компонент — это единица развертывания (.jar, .gem, .dll).

- независимое развертывание
- независимая разработка

Исполняемый файл развертывается как одна единица.

Программный компонент состоит из хорошо спроектированных программных структур среднего уровня.

Принципы сочетаемости компонентов

Принцип ацикличности зависимостей (ADP)

Циклы в графе зависимостей недопустимы.

Отдельные компоненты — отдельные разработчики (команды).

Появление цикла — появление одного БОЛЬШОГО компонента.

Граф зависимостей — ациклический ориентированный граф.

Разрыв цикла:

1) Применить принцип инверсии зависимостей.
2) Создать новый компонент, от которого зависят проблемные.

Проектирование сверху вниз?

- граф зависимостей формируется для защиты стабильных и ценных компонентов от влияния изменчивых компонентов;
- структура компонентов отражает удобство сборки и сопровождения и слабо отражает функции приложения.

Поэтому она не проектируется полностью в начале разработки

Принцип устойчивых зависимостей (SDP)

Зависимости должны быть направлены в сторону устойчивости.

Устойчивость – способность сохранять свое состояние при внешних воздействиях.

Метрика неустойчивости (I) = выходы / (входы + выходы)

I = 0 — максимальная устойчивость;

I = 1 — максимальная неустойчивость.

SDP: метрика неустойчивости компонента должна быть выше метрик неустойчивости компонентов, от которых он зависит.

Разного уровня зависимости: один метод и несколько.

Принцип устойчивости абстракций.

Устойчивость компонента должна быть пропорциональна его абстрактности.

Абстрактность (A) = число абстрактных классов и интерфейсов / число классов.

Пример: компоненты, содержащие только интерфейсы в C# или Java.

1.8 Программные структуры высшего уровня. Горизонтальные уровни, вертикальные срезы и границы в архитектуре информационных систем.

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

Границы

Разработка архитектуры — искусство проведения разделяющих линий — границ

Архитектура плагинов:

- независимые высокоуровневые компоненты
- ГРАНИЦА
- зависимые низкоуровневые компоненты

ЖЕСТКОСТЬ ГРАНИЦ — вопрос выбора архитектора

1.9 Чистая архитектура. Идея, принципы. Графическое изображение. КАК ПОСТРОИТЬ ЧИСТУЮ АРХИТЕКТУРУ

- независимость от фреймворков
- простота тестирования
- независимость от UI
- независимость от базы данных
- независимость от любых внешних агентов
- *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.

1.10 Архитектурные паттерны прикладных приложений с UI (MV*). MVC

Рисунок 1 – Чистая архитектура

Шаблон MVC (Модель-Вид-Контроллер или Модель-Состояние-Поведение) описывает простой способ построения структуры приложения, целью которого является отделение бизнес-логики от пользовательского интерфейса. В результате, приложение легче масштабируется, тестируется, сопровождается и конечно же реализуется.

1) Контроллера получает следующий запрос от пользователя.
2) Далее в зависимости от внутренней логики:
) Формируется представление какой-то страницы.
) Либо, вызываются методы модели.
3) Модель уведомляет представление об изменениях.
4) Представление обновляется (если в цепочке была задействована модель) и отображается пользователю.

MVP (Model-View-Presenter)

1) После того, как приложение получит запрос от пользователя, определяется запрошенный Presenter и действие. Приложение создает экземпляр этого Presenter'а и запускает метод

действия.

2) В методе действия могут содержаться вызовы модели, к примеру, считывающие информацию из базы данных.
3) Модель возвращает данные.
4) После этого действие формирует представление, в которое передаются данные полученные из модели.
5) Сформированное представление отображается пользователю.

MVVM

Первоначально MVVM был описан для Silverlight и имеет преимущества для сложных интерфейсов с определенной логикой, которая отличается от логики приложения. MVVM отличается более «тесной» связью между Моделью и Представлением посредством слоя Представление-Модель, который синхронизирует данные как при событии на стороне Модели, так и на стороне Представления.

В MVC логика зашита в Модели, ее можно также помещать в Контроллер, но это справедливо подвергается критике (Stupid Fat Controller). В MVVM, напротив, логика помещается в «промежуточный» слой ViewModel.

MVA

Реализация MVA основана на объектно-ориентированном подходе, где на каждый слой архитектуры будет реализован один или несколько классов. Каждый из слоев обладает рядом свойств.

Представление (View):

View содержит события, которые требуют взаимодействия с моделью View содержит контекст — ссылка на данные модели, которые необходимо отобразить пользователю View может содержать бизнес-логику, которая не требует взаимодействия с моделью.

Приложение (Application):

Выполняет роль связки представления и модели и является точкой входа в приложение. Имеет критерии запуска — набор параметров, которые определяют с какими параметрами необходимо запустить приложение. Обычно это параметры селекционного экрана.

Модель (Model):

Содержит публичные атрибуты, которые необходимы представлению. Содержит критерии расчета модели и метод инициализации.

1.11 Слои и виды моделей данных в стандартом прикладном приложении с БД и UI с точки зрения чистой архитектуры.

- Слой представления — отвечает за отображение данных пользователю и получение от него ввода. Этот слой может содержать контроллеры, которые принимают запросы от пользователей и представляют данные из слоя бизнес-логики.
- Слой бизнес-логики — содержит бизнес-логику и модели данных, которые являются самодостаточными и не зависят от слоя доступа к данным. Этот слой может содержать сущности и сервисы, которые предоставляют методы для выполнения операций над данными.
- Слой доступа к данным — отвечает за доступ к данным и сохранение их в базе данных. Этот слой может содержать репозитории, которые предоставляют методы для выполнения операций с базой данных.

1.12 Паттерны проектирования прикладных приложений (по Фаулеру).

Базовые паттерны:

Mapper (Распределитель), **Plugin** (Плагин), **Gateway** (Шлюз), **Separated Interface** (Выделенный интерфейс), **Registry** (Реестр), **Service Stub** (Сервисная заглушка), **Singleton** (Одиночка).

Паттерны веб-представления:

Template View (Шаблонизатор), **Application Controller** (Контроллер приложения), **MVC - Model View Controller** (Модель-Вид-Контроллер).

Паттерны Объектно-Реляционной логики:

Row Data Gateway (Шлюз к данным записи), **Active Record** (Активная запись), **Data Mapper**.

Паттерны логики сущности:

Domain Model (Модель области определения), **Service Layer** (Сервисный уровень).

Паттерны обработки Объектно-Реляционных метаданных:

Query Object (Объект-запрос), **Repository** (Репозиторий).

Паттерны распределение данных:

Remote Facade (Парадный вход), **Data Transfer Object** (Объект передачи данных).

1.13 Сервис-ориентированные и микросервисные архитектуры. Связь с чистой компонентной архитектурой. Цели, задачи и различия SOA и микросервисных архитектур.

Сервис-ориентированная архитектура (SOA) и микросервисная архитектура являются двумя популярными стилями архитектуры программного обеспечения. Обе архитектуры имеют схожие принципы и цели, но также имеют и ряд отличий.

SOA — это стиль архитектуры, который позволяет создавать приложения, построенные путём комбинации слабосвязанных взаимодействующих сервисов. Эти сервисы взаимодействуют на основе строго определённого платформенно- и языково-независимого интерфейса. Основные принципы SOA включают:

- использование независимых сервисов, выполняющих определённые задачи, вызываемые стандартным способом через точно определённые интерфейсы;
- инкапсуляция деталей реализации компонентов в интерфейсы, обеспечивающие независимость от используемых платформ и инструментов разработки, способствуя масштабируемости и управляемости создаваемых систем.

Микросервисная архитектура — это стиль архитектуры, в котором приложение состоит из набора мелких сервисов, каждый из которых выполняет свою определенную функцию и взаимодействует с другими сервисами через легковесные механизмы, такие как RESTful API. Основные принципы микросервисной архитектуры включают:

- разбиение приложения на мелкие, независимые сервисы, каждый из которых решает определенную задачу;
- использование легковесных механизмов взаимодействия между сервисами, таких как RESTful API;
- независимость каждого сервиса, позволяющая развивать и масштабировать приложение отдельно для каждого сервиса.

Связь с чистой компонентной архитектурой

Как SOA, так и микросервисная архитектура используют принцип инкапсуляции компонентов и независимость от платформы. Эти принципы также соответствуют принципам чистой компонентной архитектуры (Clean Architecture), которая также используется для построения сложных приложений.

Цели, задачи и различия SOA и микросервисных архитектур

Основные цели и задачи SOA и микросервисных архитектур:

- упрощение разработки, тестирования и развертывания приложений;

- улучшение масштабируемости и управляемости приложений;
- улучшение надежности и гибкости приложений.

Основные различия между микросервисной архитектурой и SOA включают:

- Масштаб: SOA может быть использована для построения крупных, монолитных приложений, в то время как микросервисная архитектура лучше подходит для построения распределенных систем.
- Сложность: SOA может быть более сложной в реализации, поскольку требует большего количества и более сложных служб.

1.14 Системная инженерия. Цель, основные идеи и методы. Постановка задачи разработки сложных систем.

Системная инженерия – это научно-методогическая дисциплина, изучающая вопросы проектирования, создания и эксплуатации структурно сложных систем. Предлагает принципы, методы и средства их разработки.

- системный подход
- общая теория систем
- математика (мат логика, мат. статистика, системный анализ, теория алгоритмов, теория игр, теория ситуаций, теория информации, комбинаторика)

Методы.

- обеспечение надежного проектного репозитория
- точная оценка доступной информации и определение недостающей
- точное определение критериев производительности и эффективности, которые определяют успех или неудачу системного проекта
- получение и анализ всех исходных требований, которые отражают запросы пользователей и цели заинтересованных сторон
- создание исполняемых моделей для верификации и валидации работы системы
- создание и поддержка плана управления системной инженерией (SEMP)

1.15 Жизненный цикл разработки ПО: модели. Гибкие и жесткие модели: достоинства и недостатки.

Разновидности:

- 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.

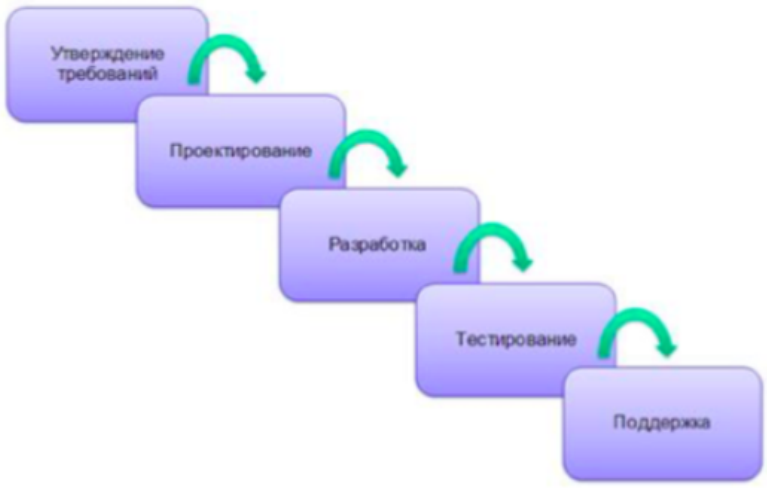

Рисунок 2 – Водопад

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

Рисунок 3 – V-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.

Рисунок 4 – Итеративная модель

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

1.16 Этап анализа в разработке ПО. Задачи и артефакты. SRD. BPMN. ER.

Этап анализа в разработке ПО является одним из самых важных этапов процесса разработки, поскольку позволяет определить требования к проекту и общую архитектуру системы. Он включает в себя ряд задач и артефактов, которые могут помочь в достижении целей этапа анализа.

Задачи этапа анализа в разработке ПО могут включать следующее:

- Изучение бизнес-процессов и требований к системе: это позволяет определить, как система будет использоваться и какие функциональные возможности должны быть реализованы. В этом процессе могут использоваться различные методы, такие как интервью с заинтересованными сторонами, анализ документации и наблюдение за работой существующих систем.
- Определение общей архитектуры системы: это включает в себя определение компонентов системы и их взаимодействия, а также определение технологий, используемых для реализации системы.
- Определение требований к данным: это позволяет определить, какие данные будут использоваться системой, а также как они будут храниться и обрабатываться.

- Определение требований к безопасности: это позволяет определить, какие меры безопасности должны быть реализованы для защиты системы от несанкционированного доступа и других угроз безопасности.

Артефакты этапа анализа включают:

- Software Requirements Document (SRD) или документ требований к программному обеспечению: это документ, который описывает требования к системе, включая функциональные и нефункциональные требования. SRD также может включать в себя описание архитектуры системы и требования к данным и безопасности.
- Business Process Model and Notation (BPMN) или модель бизнес-процесса и нотации: это графический язык для моделирования бизнес-процессов. Он может использоваться для визуализации бизнес-процессов и определения требований к системе.
- Entity-Relationship (ER) или диаграмма сущность-связь: это графическая диаграмма, которая описывает структуру базы данных. Она может использоваться для определения требований к данным и определения связей между различными сущностями в системе.

1.17 Менеджмент требований и качества ПО. Атрибуты качества ПО. ISO 9126.

Менеджмент требований и качества ПО — это важная составляющая процесса разработки ПО. Важно не только определить требования к системе, но и убедиться, что эти требования реализованы в конечном продукте на высоком уровне качества.

ISO 9126 — это стандарт, который определяет модель качества программного обеспечения, включающую в себя атрибуты качества и меры оценки качества. Он содержит следующие атрибуты качества:

- Функциональность: это определяет, насколько система соответствует требованиям и выполняет свои функции правильно.
- Надежность: это определяет, насколько система является надежной и устойчивой к сбоям и ошибкам.
- Эффективность: это определяет, насколько система эффективна в использовании ресурсов, таких как время и память.
- Удобство использования: это определяет, насколько система легка в использовании и понимании пользователем.
- Портативность: это определяет, насколько система может быть перенесена на другую платформу без необходимости внесения значительных изменений.
- Сопровождаемость: это определяет, насколько система легко сопровождается и изменяется с течением времени.

1.18 Этап проектирования в разработке ПО. Задачи и артефакты. SDD. UML.

Этап проектирования в разработке ПО является очень важным этапом, поскольку на этом этапе определяется общая архитектура системы и ее детали. На этом этапе разрабатываются артефакты, которые будут использоваться на этапе реализации.

Задачи этапа проектирования в разработке ПО могут включать следующее:

- Определение архитектуры системы: это включает в себя определение компонентов системы и их взаимодействия, а также определение технологий, используемых для реализации системы.
- Разработка детального дизайна: это включает в себя проектирование интерфейсов пользователя, базы данных и других деталей системы.
- Определение тестовых сценариев: это включает в себя определение тестовых сценариев, которые будут использоваться для проверки системы на соответствие требованиям.

Артефакты этапа проектирования включают:

- Software Design Document (SDD) или документ проектирования программного обеспечения: это документ, который описывает дизайн системы, включая архитектуру системы, детальный дизайн и тестовые сценарии. SDD также может включать в себя описание алгоритмов и структур данных.
- Unified Modeling Language (UML) или язык моделирования: это язык для визуализации и документации проектов по разработке программного обеспечения. Он может использоваться для моделирования архитектуры системы, дизайна интерфейсов пользователя и других деталей системы.

Следует отметить, что SDD и UML могут использоваться вместе для создания полной и понятной спецификации проекта по разработке программного обеспечения.

1.19 Язык UML для задач проектирования архитектуры ПО. Краткая характеристика других языков графического моделирования (Archimate, C4).

UML (Unified Modeling Language) — это язык моделирования, который широко используется для задач проектирования архитектуры ПО. UML включает в себя различные виды диаграмм, такие как диаграммы классов, диаграммы компонентов, диаграммы последовательностей и другие. Он предоставляет стандартизированный набор символов и обозначений для моделирования различных аспектов системы.

Archimate — это язык моделирования, который разработан для моделирования ар-

хитектуры предприятия. Он позволяет моделировать различные уровни архитектуры, такие как бизнес-архитектура, информационная архитектура и техническая архитектура. Archimate включает в себя различные виды диаграмм, такие как диаграммы бизнес-архитектуры, диаграммы информационной архитектуры и диаграммы технической архитектуры.

C4 — это модель, которая используется для описания архитектуры ПО. Он включает в себя четыре уровня абстракции (контекстная, контейнерная, компонентная и кодовая), каждый из которых представлен соответствующей диаграммой. C4 модель не является заменой UML, но скорее дополняет его, предоставляя более простой и понятный способ описания архитектуры системы.

1.20 Организация процесса разработки ПО. Декомпозиция задач. Оценка сложности задач. Таск- и баг- трекеры.

Организация процесса разработки ПО — это важная составляющая успешного завершения проекта. Организация процесса разработки ПО может включать в себя следующие шаги:

- Декомпозиция задач: это процесс разбиения проекта на меньшие задачи, которые могут быть выполнены отдельными командами или разработчиками. Декомпозиция задач позволяет лучше контролировать процесс разработки и управлять рисками.
- Оценка сложности задач: это процесс определения времени и усилий, необходимых для выполнения каждой задачи. Оценка сложности задач позволяет лучше понимать, какие задачи могут быть выполнены в рамках определенного бюджета и сроков.
- Использование таск- и баг- трекеров: это программные приложения, которые позволяют отслеживать задачи и ошибки в проекте. Таск- и баг- трекеры позволяют лучше организовать процесс разработки, управлять временем и ресурсами, а также улучшать качество продукта.

Декомпозиция задач и оценка сложности задач могут быть выполнены с помощью различных методов, таких как методика WBS (Work Breakdown Structure) или методика PERT (Program Evaluation and Review Technique). Эти методы позволяют разбить проект на меньшие задачи и определить время и усилия, необходимые для выполнения каждой задачи.

Таск- и баг- трекеры могут быть использованы для отслеживания задач и ошибок в проекте. Такие приложения, как Jira, Trello, Asana и другие, позволяют управлять задачами, отслеживать время, устанавливать сроки и отслеживать прогресс проекта. Они также позволяют вести коммуникацию между членами команды и управлять проектом на основе данных.

1.21 Проектная команда в разработке ПО: роли и ответственность. Виды команд.

Проектная команда в разработке ПО играет важную роль в успешном завершении проекта. Команда должна иметь определенные роли и ответственности, которые могут включать в себя:

- Product Owner: отвечает за определение требований к продукту и управление портфелем продуктов.
- Project Manager: отвечает за управление проектом, включая планирование, контроль и управление рисками.
- Разработчики: отвечают за создание кода и реализацию функциональности продукта.
- Тестировщики: отвечают за тестирование продукта и обеспечение его качества.
- Дизайнеры: отвечают за создание дизайна продукта и его пользовательского интерфейса.
- Аналитики: отвечают за анализ требований и данных и обеспечение соответствия продукта бизнес-целям.

Роли и ответственности могут варьироваться в зависимости от конкретного проекта и его потребностей.

В зависимости от типа проекта и его характеристик, команды могут быть различных типов, например:

- Команда разработки продукта: команда, которая отвечает за создание и разработку продукта.
- Команда технической поддержки: команда, которая отвечает за техническую поддержку продукта после его запуска.
- Команда обслуживания: команда, которая отвечает за обслуживание продукта после его запуска.

Каждая команда должна иметь свои роли и ответственности, которые определяются на основе характеристик проекта и его потребностей.

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

Системы контроля версий (Version Control Systems, VCS) представляют собой инструменты, которые помогают разработчикам ПО отслеживать изменения в исходном коде, управлять версиями кода, координировать работу нескольких разработчиков над одним проектом и восстанавливать предыдущие версии кода в случае необходимости.

Существует несколько моделей использования систем контроля версий, которые могут

быть применены в различных сценариях разработки ПО. Одна из наиболее распространенных моделей - централизованная модель (Centralized Version Control System, CVCS), где все изменения в репозитории (хранилище исходного кода) осуществляются через центральный сервер. Примерами таких систем являются Subversion (SVN) и Microsoft Team Foundation Server (TFS). Такими системами легко управлять из-за наличия единственного сервера. Но при этом наличие централизованного сервера приводит к возникновению единой точки отказа в виде этого самого сервера. В случае отключения этого сервера разработчики не смогут выкачивать файлы. Самым худшим сценарием является физическое уничтожение сервера (или вылет жесткого диска), он приводит к потерю кодовой базы.

Другая модель - распределенная модель (Distributed Version Control System, DVCS), в которой каждый разработчик имеет локальную копию репозитория и может работать над проектом независимо от других разработчиков. Изменения в репозитории синхронизируются между копиями репозитория разных разработчиков. Примерами таких систем являются Git, Mercurial и Bazaar. Для устранения единой точки отказа используются распределенные системы контроля версий. Они подразумевают, что клиент выкачает себе весь репозиторий целиком заместо выкачки конкретных интересующих клиента файлов. Если умрет любая копия репозитория, то это не приведет к потере кодовой базы, поскольку она может быть восстановлена с компьютера любого разработчика. Каждая копия является полным бэкапом данных.

Код-ревью (Code Review) - это процесс, в ходе которого другие разработчики анализируют код, написанный одним из разработчиков, с целью выявления ошибок, улучшения качества кода и обмена знаниями между командой разработчиков. Код-ревью может быть проведен как до того, как изменения будут включены в основную ветку кода, так и после этого. Системы контроля версий обычно предоставляют инструменты для управления процессом код-ревью, такие как pull requests в Git или code review в TFS.

1.23 Базовый (стандартный) процесс разработки ПО на примере разработки b2c-продукта и заказного b2b-проекта.

Базовый процесс разработки продукта

- Идея
- ТЗ на MVP (MVP – это минимально жизнеспособный продукт. То есть продукт, который содержит минимальный набор функций, но является уже ценным для пользователя)
- Верхнеуровневый архитектурный тех. проект, проект на MVP
- Итерации, в каждой – рабочая версия
- MVP (выюираються основыне задачи, более простой путь) (Главная задача MVP — минимизировать время и усилия, затраченные на тестирование реакции рынка на идею.)

- Набор ТЗ/Одно ТЗ
- Детализированное проектирование архитектуры системы
- Итерации, в каждой – сохраняем работоспособность системы
- Версия системы 1.0 (опытный образец) (тестирование на малом количестве пользователей)
- Версия системы 2.0 (промышленный) (тестирование на малом количестве пользователей)

B2C (Business to Consumer) предполагает продажу товаров и услуг физическим лицам или конечным потребителям B2B (Business to Business) – модель, когда клиенты компании – это другие фирмы

Процесс разработки ПО может немного различаться в зависимости от типа проекта. Например, в случае b2c-продукта, процесс может быть более быстрым и гибким, так как фокус сосредоточен на удовлетворении потребностей конечных пользователей и быстрой реакции на изменения рынка. В то же время, в случае заказного b2b-проекта, процесс может быть более формализованным и проходить через дополнительные этапы, такие как согласование требований с заказчиком, оценка рисков и управление изменениями.

Для б2с-продукта процесс разработки может быть организован в рамках Agile-методологии, например, Scrum, что позволит быстро реагировать на изменения, проводить тестирование и выявлять ошибки на ранних стадиях.

Для заказного b2b-проекта может быть применена Waterfall-методология, которая предполагает последовательное выполнение этапов разработки, начиная с анализа требований и заканчивая релизом и поддержкой продукта. Это поможет заказчику получить четкое понимание ожидаемых результатов и контролировать процесс разработки на каждом этапе.

Также для заказного b2b-проекта может быть применен DevOps-подход, который объединяет разработку, тестирование и внедрение продукта, а также управление инфраструктурой и автоматизацию процессов. Это позволяет ускорить процесс разработки и улучшить качество продукта, а также обеспечить более эффективное управление процессами и ресурсами.

1.24 Подходы к организации процесса разработки ПО (*DD)

Существует множество подходов к организации процесса разработки ПО, которые имеют свои особенности и принципы работы. Один из таких подходов - это *DD (сокращение от различных методологий, таких как TDD, BDD, DDD и т.д.), который представляет собой семейство методологий, основанных на тестировании и управлении требованиями.

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 посто-

янной проверкой прохождения тестов. Применяется не часто, так как никто не любить писать тесты.

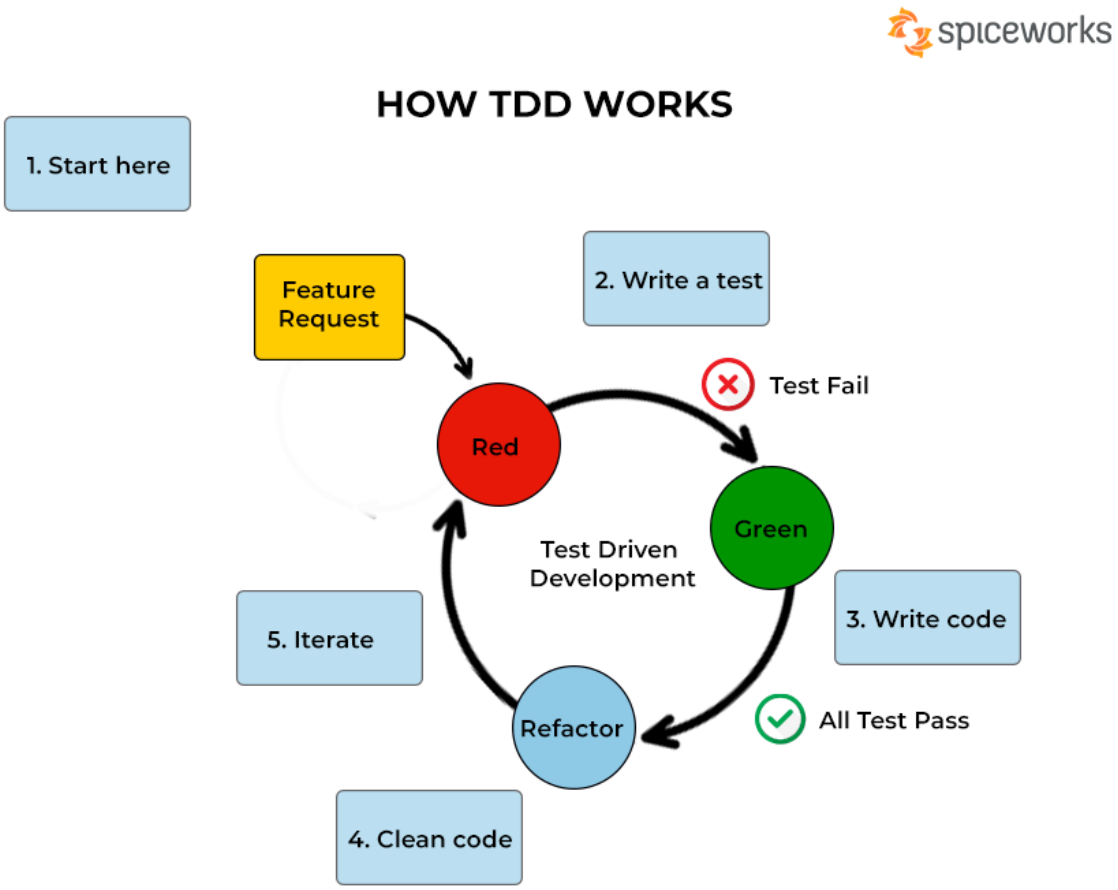

Рисунок 5 – TDD

Behavior-Driven Development (BDD) (на уровень выше TDD) - это методология, которая уделяет особое внимание описанию требований в форме поведения системы. Разработчики и бизнес-аналитики вместе определяют ожидаемое поведение системы и описывают его на языке бизнеса. Затем на основе этих описаний создаются тесты, которые проверяют соответствие реализации системы заданным требованиям. Такой подход позволяет улучшить коммуникацию между разработчиками и бизнесом, а также повысить качество кода. Из-за некоторого методологического сходства TDD (Test Driven Development) и BDD (Behaviour Driven Development) часто путают даже профессионалы. В чем же отличие? Концепции обоих подходов похожи, сначала идут тесты и только потом начинается разработка, но предназначение у них совершенно разное. 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.

Плюсы:

- Тесты читаемые для не программистов.
- Их легко изменять. Они часто пишутся почти на чистом английском.
- Их может писать product owner или другие заинтересованные лица.

Но у данного подхода есть и недостатки — это долго и дорого. BDD неудобен хотя бы тем, что требует привлечения специалистов тестирования уже на этапе проработки требований, а это удлиняет цикл разработки.

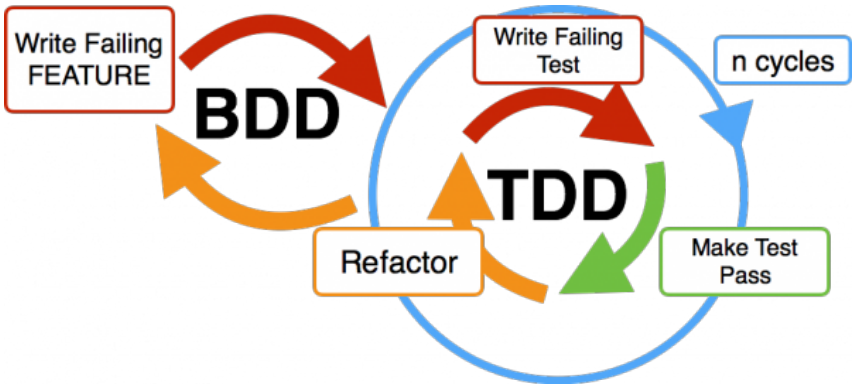

Рисунок 6 – BDD

Domain-Driven Design (DDD) - это методология, которая фокусируется на моделировании предметной области, на которой базируется система. Разработчики и бизнес-аналитики вместе определяют ключевые понятия предметной области и создают модели, которые отражают ее структуру и поведение. В эти модели входит бизнес-логика, устанавливающая связь между реальными условиями области применения продукта и кодом. Затем на основе этих моделей создаются архитектура и дизайн системы, что позволяет улучшить ее архитектуру и сделать ее более гибкой и адаптивной к изменениям в предметной области.

Подход DDD особо полезен в ситуациях, когда разработчик не является специалистом в области разрабатываемого продукта. К примеру: программист не может знать все области, в которых требуется создать ПО, но с помощью правильного представления структуры, посредством предметно-ориентированного подхода, может без труда спроектировать приложение, основываясь на ключевых моментах и знаниях рабочей области.

Основная цель Domain-Driven Design — это борьба со сложностью бизнес-процессов, их автоматизации и реализации в коде. «Domain» переводится как «предметная область», и именно от предметной области отталкивается разработка и проектирование в рамках данного подхода.

Ключевым понятием в DDD является «единый язык» (ubiquitous language). Ubiquitous language способствует прозрачному общению между участниками проекта. Единый он не в том

смысле, что он один на все случаи жизни. Как раз наоборот. Все участники общаются на нём, всё обсуждение происходит в терминах единого языка, и все артефакты максимально должны излагаться в терминах единого языка, то есть, начиная от ТЗ, и, заканчивая кодом.

Следующим понятием является "доменная модель". Данная модель представляет из себя словарь терминов из ubiquitous language. И доменная модель, и ubiquitous language ограничены контекстом, который в Domain-Driven Design называется bounded context. Он ограничивает доменную модель таким образом, чтобы все понятия внутри него были однозначными, и все понимали, о чём идёт речь.

Пример: возьмем сущность "человек"и поместим его в контекст "публичные выступления". В этом контексте, по DDD, он становится спикером или оратором. А в контексте "семья"— мужем или братом.

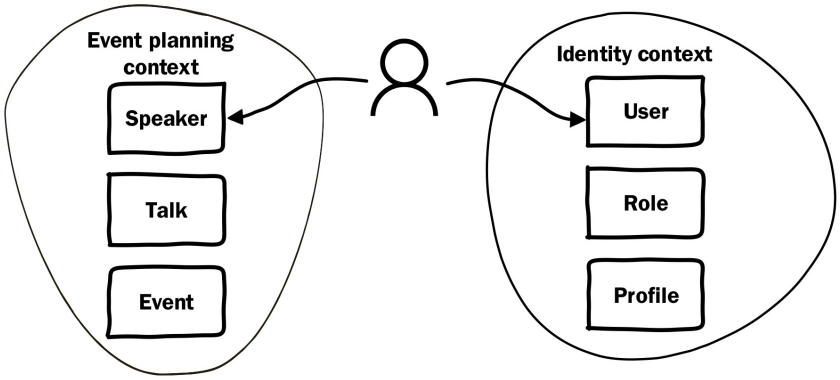

Рисунок 7 – Контекст

Если в языке проекта вы используете выражения "продукт был добавлен то следующий вариант не по DDD:

var product = new Product('apple')

product.save()

Почему? В коде написано, что мы создали продукт странным образом и сохранили его. Как же все таки добавить продукт? Нужно его добавить. Вот DDD код:

Product::add('apple');

С точки зрения Domain-Driven Design абсолютно всё равно, какую архитектуру вы выберете. Domain-Driven Design не про это, Domain-Driven Design про язык и про общение. Но DDD почти невозможен без чистой архитектуры проекта, так как при добавлении новой функциональности или изменении старой нужно стараться сохранять гибкость и прозрачность кодовой

базы.

Плюсы:

- почти все участники команды могут читать код проекта;
- постановка задач становится более явной;
- баги бизнес логики становиться проще искать;

Минусы: требуется высокая квалификация разработчиков, особенно, на старте проекта, не все клиенты готовы пойти на такие затраты, DDD нужно учиться всем участникам процесса разработки.

FDD — Эта методология (кратко именуемая FDD) была разработана Джеффом Де Люка (Jeff De Luca) и признанным гуру в области объектно-ориентированных технологий Питером Коадом (Peter Coad). FDD представляет собой попытку объединить наиболее признанные в индустрии разработки программного обеспечения методики, принимающие за основу важную для заказчика функциональность (свойства) разрабатываемого программного обеспечения. 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. Согласно FDD, одна итерация длится две недели. FDD насчитывает пять процессов. Первые три из них относятся к началу проекта:

- разработка общей модели;
- составление списка требуемых свойств системы;
- планирование работы над каждым свойством;
- проектирование каждого свойства;
- конструирование каждого свойства.

Последние два шага необходимо делать во время каждой итерации. При этом каждый процесс разбивается на задачи и имеет критерии верификации.

Плюсы:

- документация по свойствам системы;
- тщательное проектирование;
- тесты ориентированы на бизнес-задачи;
- проработанный процесс создания продукта;
- короткие итеративные циклы разработки позволяют быстрее наращивать функциональность и уменьшить количество ошибок.

Минусы:

- FDD больше подходит для больших проектов
- значительные затраты на внедрение и обучение.

Разработка, управляемая моделями, (англ. model-driven development) — это стиль разработки программного обеспечения, когда модели становятся основными артефактами разработки, из которых генерируется код и другие артефакты.

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

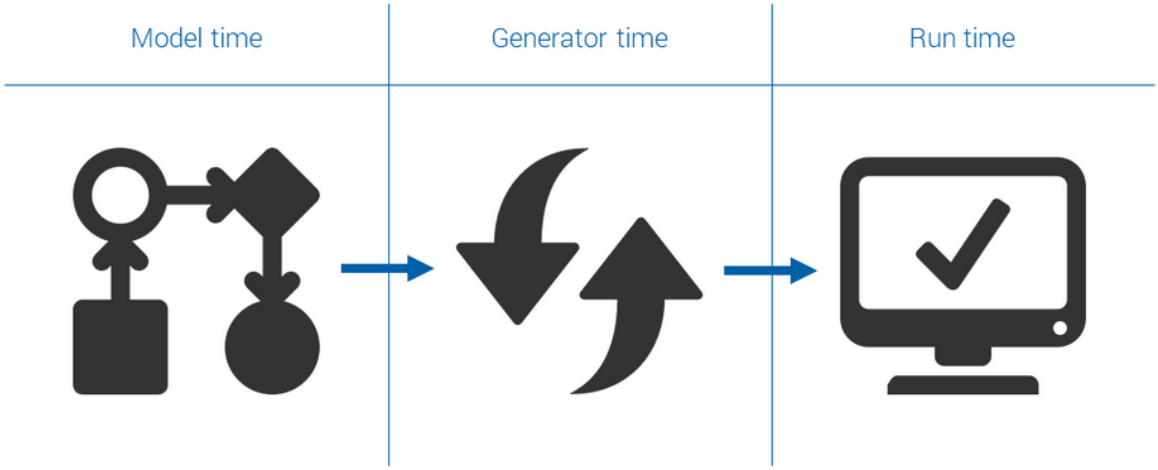

Рисунок 8 – MDD

Основная цель MDD — минимизация затрат, связанных с привязкой к конкретным системным платформам и программным инфраструктурам. Ведь основная бизнес-логика содержится в диаграммах и не сковывает нас рамками выбора языка программирования и инструментов разработки.

Идея MDD не нова - она использовались с переменным успехом и раньше. Причиной возросшего внимания к ним в настоящее время является то, что автоматизации поддается значительно больше процессов, чем раньше. Это развитие отражается в появлении MDD-стандартов, что ведет к унификации соответствующих средств. Одним из таких стандартов является пересмотренная версия Unified Modeling Language – UML 2.0.

Плюсы:

- ускоряется вывод минимального жизнеспособного продукта (Minimum Viable Product) на рынок;
- сокращается время на: генерацию каркаса приложения, модели классов, базы данных;
- постоянно обновляемая документация;
- для участников проекта диаграммы намного нагляднее кода.

Минусы:

- для внедрение MMD потребуется использовать специальные программные решения
- от программистов требуются внушительные знания проектирования диаграмм
- значительные затраты на внедрение и обучение.

PDD — Panic Driven Development. Если вы пробовали методологии agile разработки, то вы наверняка пробовали и PDD. Давайте посмотрим более подробно, каковы принципы этой методологии. PDD своеобразный антипаттерн разработки.

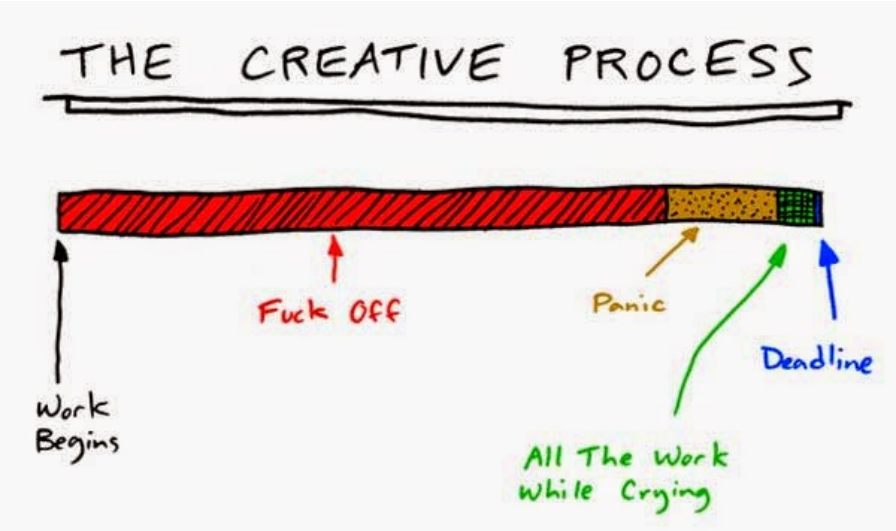

Рисунок 9 – PDD

Важные моменты:

- Новые задачи приоритетнее старых (ориентация на хотелки клиента)
- Пишите столько кода, сколько нужно, чтобы решить проблему (Сначала напишите решение, потом проверьте своё предположение по исправлению. Если исправление работает, проблема решена. Не тратим время попусту)
- Тесты должны писаться в конце (После того, как исправление внедрено, тесты могут быть запланированы как задача, которая будет сделана в будущем. Ручного тестирования должно быть достаточно, чтобы доказать работоспособность реализованного решения.)

Плюсы:

- высокая скорость разработки;
- дешево

Минусы:

- все плюсы разобьются о технический долг и сложность проекта.

1.25 Рефакторинг, оптимизация и исправление ошибок в ПО

Коротко

Причины:

- Модификация
- Ошибки
- Проблемы с разработкой

Подход:

- Небольшие, эквивалентные преобразования
- TDD BDD

Более подробно

Рефакторинг.

Внесение значительных изменений в программу возможно как на стадии разработки, так и на стадии сопровождения. При разработке возможны изменения требований, которые неизбежно приводят к изменениям кода. Современные методики снижают предсказуемость кодирования; т.к. эволюция - неизбежный процесс, его необходимо планировать и извлекать выгоду. Эволюция ПО должна повышать внутреннее качество программы. Важнейшей стратегией для достижения этой цели является рефакторинг.

Когда нужен рефакторинг? Код повторяется; метод слишком велик; цикл слишком велик или глубоко вложен в другие; класс имеет плохую связность; метод принимает слишком много параметров; класс слишком много знает о другом классе, или его метод слишком сильно зависит от другого класса. Наконец, если методы просто имеют неудачные имена, данные-члены сделаны открытыми, или код содержит глобальные переменные.

Рефакторинг можно поделить на три уровня: на уровне данных, на уровне отдельных операторов, на уровне отдельных методов, на уровне классов. На уровне данных: встраивание выражений в код, замена магических чисел на константы, инкапсуляция набора, преобразование массива в класс и т.п. На уровне операторов: декомпозиция логического выражения, замена условных операторов на вызов полиморфного метода, создание пустых объектов вместо проверки на null. На уровне методов: встраивание кода метода, извлечение метода, добавление или удаление параметров. На уровне классов: замена объектов-значений на объекты-ссылки и наоборот, перемещение специализированного кода в подкласс, объединение похожего кода в суперкласс.

Оптимизация

Оптимизация в проектировании программного обеспечения — это процесс улучшения производительности и эффективности программного продукта.

1) Алгоритмическая оптимизация: Этот подход заключается в улучшении алгоритмов, используемых в программном продукте. Это может быть достигнуто путем уменьшения

количества операций, ускорения выполнения алгоритма или уменьшения потребляемой памяти.

2) Оптимизация кода: Этот подход заключается в улучшении кода программного продукта. Это может быть достигнуто путем уменьшения количества повторяющегося кода, использования более эффективных структур данных или использования более эффективных алгоритмов.
3) Оптимизация базы данных: Этот подход заключается в улучшении базы данных, используемой программным продуктом. Это может быть достигнуто путем оптимизации запросов к базе данных, уменьшения количества данных, которые нужно хранить, или увеличения скорости доступа к данным.

Исправление ошибок

1) написание тестов на новую ошибку
2) исправление ошибки
3) надежда на светлое будущее (так в лекции написано... чее?)
4) TDD и BDD

1.26 CI/CD. Развертывание. Контейнеризация.

CI — continuous integration (непрерывная интеграция)

CD — continuous delivery (непрерывная доставка)

Это методология разработки ПО, которая позволяет автоматизировать процессы сборки, тестирования и доставки программного продукта.

Развертывание — это процесс установки и настройки программного обеспечения на конечном устройстве или сервере. Он включает в себя различные этапы, такие как установка зависимостей, настройка конфигурации и запуск приложения. Автоматизация этого процесса может значительно ускорить время выкатки новых версий программного продукта.

Контейнеризация — это технология, которая позволяет упаковывать приложения и их зависимости в контейнеры, которые могут быть запущены на любом компьютере, где есть подходящее окружение. Это позволяет упростить процесс развертывания, ускорить время выкатки новых версий и обеспечить большую гибкость в разработке и тестировании ПО.

1.27 Методология DDD. Цель и задачи. Основные принципы и идеи. Процесс разработки ПО по DDD. Связь с чистой архитектурой.

Методология DDD (Domain-Driven Design или предметно-ориентированное проектирование) - это подход к разработке программного обеспечения, который направлен на создание сложных приложений через упрощение их детализации с использованием основных понятий предметной области.

Цель DDD заключается в построении эффективной модели предметной области, которая бы отражала ключевые принципы и элементы реального мира. Это помогает улучшить общение между разработчиками и экспертами предметной области, обеспечивая точное и надежное программное решение.

Задачи DDD включают:

- Улучшение взаимопонимания между разработчиками и экспертами предметной области с использованием унифицированного языка (Ubiquitous Language).
- Построение четкой и гибкой архитектуры приложения.
- Отделение бизнес-логики от технических аспектов разработки.
- Обеспечение возможности масштабирования и развития программного решения.
- Уменьшение сложности приложения и рисков возникновения ошибок.

Основные принципы и идеи методологии DDD:

- Унифицированный язык (Ubiquitous Language): Общий язык, который используется всеми членами команды (разработчики, эксперты предметной области, менеджеры) для описания предметной области и бизнес-правил.
- Ограниченный контекст (Bounded Context): Границы, внутри которых применяется унифицированный язык и модель предметной области. Это позволяет разграничивать области ответственности и избегать сложных зависимостей между компонентами.
- Модель предметной области (Domain Model): Абстракция, которая отражает ключевые понятия и связи предметной области, а также бизнес-правила и логику.
- Сущности (Entities) и Значения (Value Objects): Сущности - это объекты предметной области, которые имеют уникальный идентификатор и изменяемое состояние. Значения представляют собой неизменяемые объекты, имеющие смысл только в контексте своих свойств.
- Агрегаты (Aggregates): Кластеры связанных сущностей и значений, которые обрабатываются как единое целое. Агрегаты ограничивают доступ к своим внутренним компонентам и гарантируют соблюдение бизнес-правил.
- Репозитории (Repositories): Репозитории предоставляют интерфейс для поиска, сохранения и управления агрегатами.
- Сервисы предметной области (Domain Services): Компоненты, обеспечивающие реализацию бизнес-логики или функций, которые не могут быть непосредственно связаны с сущностями и значениями.
- Фабрики (Factories): Компоненты, ответственные за создание объектов и агрегатов предметной области.
- События (Events): Уведомления о событиях, происходящих в предметной области, кото-

рые могут быть использованы для координации между различными компонентами системы.

Некоторые идэи это тупо паттерны.

Процесс разработки программного обеспечения по методологии Domain-Driven Design (DDD) представляет собой подход, который сфокусирован на организации работы над проектами вокруг основных бизнес-доменов и их сущностей. DDD ставит целью снижение сложности проектов, обеспечение их гибкости и устойчивости к изменениям.

- Определение бизнес-задач и выделение ядра предметной области (Domain Core). На этом этапе осуществляется исследование бизнес-домена, взаимодействия с экспертами в данной области и определение основных блоков — поддоменов. Также структурируется знание бизнес-домена, выделяются ключевые понятия, термины и контексты.
- Разработка модели предметной области (Domain Model) на основе ядра предметной области.
- Установление универсального языка (Ubiquitous Language): на этапе разработки универсального языка команда разрабатывает общий словарь, описывающий все ключевые термины и их отношения внутри определенного контекста. Это язык взаимодействия между разработчиками и экспертами, который позволяет избегать недопонимания и гарантирует жесткую связь между кодом и реальными продуктовыми метафорами.
- Разработка ограниченных контекстов (Bounded Contexts): каждый блок модели бизнес-домена имеет свой ограниченный контекст - четко определенную границу с находящимися внутри моделями, бизнес-правилами и логикой. Ограниченный контекст необходим для избегания влияния одной части модели на другую и упрощения иерархии зависимостей.
- Концепция упорядоченного дизайна (Организация кода по слоям и реализация ключевых подходов DDD). Код разделяется.
- Разработка интеграции и адаптеров: после того, как устанавливаются ограниченные контексты, проектируется модуль для интеграции и взаимодействия между ними. Здесь реализуются адаптеры, которые позволяют независимым модулям успешно работать с реализациями в других контекстах.
- Непрерывное совершенствование и рефакторинг: по мере расширения или изменения предметной области, происходит постоянная актуализация и улучшение набора доменных знаний, что отражается на кодовой базе. Таким образом, проектирование и разработка в DDD является итеративным и постоянно развивающимся процессом.

  Слои

- Доменный слой - набор доменных моделей, сущностей, агрегатов и контрактов, представ-

ляющих предметную область и ее бизнес-правила.

- Инфраструктурный слой - инструменты и сервисы для поддержки слоев приложения, назначен для работы с внешними средствами (например, БД, сетевые службы).
- Прикладной слой - слой, который координирует работу остальных слоев и обеспечивает связь между интерфейсами и доменной логикой.
- Интерфейсное слой - отвечает за взаимодействие с пользователем или другими системами.

В процессе разработки ПО по DDD важно уделить внимание следующим аспектам:

- Активное взаимодействие с экспертами предметной области для получения полного понимания бизнес-задач и требований.
- Использование языка, понятного для экспертов предметной области, при разработке модели предметной области.
- Разделение предметной области на поддомены и контексты для упрощения разработки и поддержки приложения.
- Использование тестирования для обеспечения корректной работы приложения и соответствия его требованиям

DDD имеет много общего с концепцией "чистой архитектуры"(Clean Architecture), предложенной Робертом Мартином. Оба подхода фокусируются на создании гибкой, легко поддерживаемой и расширяемой системы. Они также оба предлагают разделение системы на слои и использование паттернов проектирования. Однако, DDD добавляет еще один уровень абстракции - модель предметной области, и уделяет большее внимание моделированию и языку, что позволяет создать более точное отображение предметной области в системе. Таким образом, DDD и "чистая архитектура"взаимодополняют друг друга и могут использоваться вместе для создания более эффективной и качественной системы. Когда чистая архитектура и DDD применяются совместно, DDD отвечает за моделирование предметной области, в то время как чистая архитектура предоставляет подход к разделению слоев и зависимостей программного обеспечения.

1.28 Методология Event Storming. Связь с DDD.

Event Storming это интенсивный вид коллаборативной моделирования, который позволяет быстро раскрыть сложные бизнес-процессы и требования для разработки программного обеспечения. Event Storming объединяет экспертов по предметной области и разработчиков на одной площадке для генерации идеей и экспериментов для лучшего обнаружения предметной области. Он был разработан Альберто Бранделлини в 2013 году и с тех пор стал популярным подходом к моделированию и проектированию программных систем.

Domain-Driven Design (DDD) - это подход к разработке программного обеспечения, осно-

ванный на создании и внедрении наиболее точной модели предметной области и ее использовании для разработки сложных приложений. Это позволяет сосредоточиться на ядревом функционале, предоставляемом системой и упрощать разработку и поддержку ПО.

Связь между Event Storming и DDD заключается в том, что обе методологии ставят предметную область и ее осмысление в качестве ключевого фактора успешной разработки программного обеспечения.

Event Storming может быть использован как первый шаг в процессе DDD. На этапе Event Storming участники определяют важные события (Event), команды (Command), агрегаты (Aggregate), сущности (Entity) и ограничения (Invariant) в предметной области. Это позволяет получить хорошее понимание бизнес-процессов и полезно для создания так называемой "языковой модели"(Ubiquitous Language) - согласованного набора терминов и понятий, которые будут использоваться при дальнейшем проектировании и разработке системы.

После проведения сессии Event Storming дизайнеры и архитекторы могут начать применять принципы DDD для создания моделей и разрабатывать программные компоненты, строя их на основе выявленных в рамках Event Storming паттернов и структур. Это может включать создание агрегатов, репозиториев, сервисов, событий и других элементов, а также определение контекстов и границ (Bounded Context).

Вместе Event Storming и DDD обеспечивают рамки для превращения глубокого понимания предметной области в хорошо структурированное и легко поддерживаемое программное обеспечение.

1.29 TOGAF. Цель, основные идеи. Составные части процесса проектирования архитектуры предприятия по TOGAF.

The Open Group Architecture Framework — фреймворк для описания архитектуры предприятия, который предлагает подход для проектирования, планирования, внедрения IT-архитектуры предприятия и управления ей.

Основными идеями TOGAF являются:

- Системный подход к проектированию архитектуры предприятия, который включает в себя описание бизнес-процессов, информационной архитектуры, технической архитектуры и архитектуры приложений.
- Гибкость и адаптивность архитектуры предприятия, которые позволяют организациям быстро реагировать на изменения в бизнес-среде и технологическом ландшафте.
- Управление жизненным циклом архитектуры предприятия, которое включает в себя планирование, разработку, реализацию и управление архитектурой.

Базовая архитектура:

- набор наиболее общих служб и функций — Техническая Эталонная Модель
- набор элементарных архитектурных элементов (строительные архитектурные блоки и блоки реализации) (архитектурные блоки определяют требования и создают каркас)
- база данных стандартов

Рисунок 10 – TODAF

1.30 Задачи архитектора ПО. Технический (системный) архитектор. Архитектор решений (solution). Энтерпрайз архитектор.

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

Технический (системный) архитектор отвечает за разработку технической архитектуры системы. Он должен иметь глубокие знания в области программирования, баз данных,

сетевых технологий и операционных систем.

Архитектор решений (solution) отвечает за разработку конкретных решений на основе общей архитектуры предприятия. Он должен уметь анализировать бизнес-требования и выбирать подходящие технологии и решения.

Энтерпрайз архитектор отвечает за разработку общей архитектуры предприятия, которая охватывает все бизнес-процессы и технологическую инфраструктуру. Он должен иметь широкие знания в области бизнеса, информационных технологий и управления проектами.