1. Понятия требований, классификация.

https://stotis.sakhalin.gov.ru/upload/medialibrary/215/93d2xfb4xvm3w08mooscxugdcv3b2hzc.pdf

Требования - это исходные данные, на основании которых проектируются и создаются автоматизированные информационные системы. Первичные данные поступают из различных источников, характеризуются противоречивостью, неполнотой, нечеткостью, изменчивостью. Требования нужны в частности для того, чтобы Разработчик мог определить и согласовать с Заказчиком временные и финансовые перспективы проекта автоматизации.

Классификация требований:

* Функциональные – что система должна делать (например, авторизация пользователя).
* Нефункциональные – как система должна работать (производительность, безопасность, надежность).
* Бизнес-требования – цели организации.
* Пользовательские требования – потребности конечных пользователей.
* Системные требования – технические условия для работы ПО.

2. Уровни требований

1. Бизнес-требования (стратегические цели).
2. Пользовательские требования (сценарии использования).
3. Системные требования (технические спецификации).
4. Уровни требований.

Обычно выделяют три уровня требований.

• На верхнем уровне представлены так называемые бизнес-требования (business requirements). Примеры бизнес-требования: система должна сократить срок оборачиваемости обрабатываемых на предприятии заказов в три раза. Бизнестребования обычно формулируются топ-менеджерами, либо акционерами предприятия.

• Следующий уровень - уровень требований пользователей (user requirements). Пример требования пользователя: система должна представлять диалоговые средства для ввода исчерпывающей информации о заказе, последующей фиксации информации в базе данных и маршрутизации информации о заказе к сотруднику, отвечающему за его планирование и исполнение. Требования пользователей часто бывают плохо структурированными, дублирующимися, противоречивыми. Поэтому для создания системы важен третий уровень, в котором осуществляется формализация требований.

• Третий уровень - функциональный (functional requirements). Пример функциональных требований (или просто функций) по работе с электронным заказом: заказ может быть создан, отредактирован, удален и перемещен с участка на участок.

1. Методологии и стандарты, регламентирующие работу с требованиями.

Среди основополагающих нормативных документов в области работы с требованиями можно выделить следующие.

1. Разработки IEEE:

• IEEE 1362 "Concept of Operations Document".

• IEEE 1233 "Guide for Developing System Requirements Specifications".

• IEEE Standard 830-1998, "IEEE Recommended Practice for Software Requirements Specifications"

• IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990

• IEEE Guide to the Software Engineering Body of Knowledge (1) - SWEBOK®, 2004.

2. Отечественные ГОСТ:

• ГОСТ 34.601-90. Информационная технология. Автоматизированные системы. Стадии создания.

• ГОСТ 34.602-89. Информационная технология. Техническое задание на создание автоматизированной системы

• ГОСТ 19.201-78. Единая система программной документации. Техническое задание. Требования к содержанию и оформлению.

1. Современные принципы и методы разработки программных приложений.

<https://infourok.ru/prezentaciya-na-temu-sovremennye-principy-i-metody-razrabotki-prilozhenij-6282038.html>

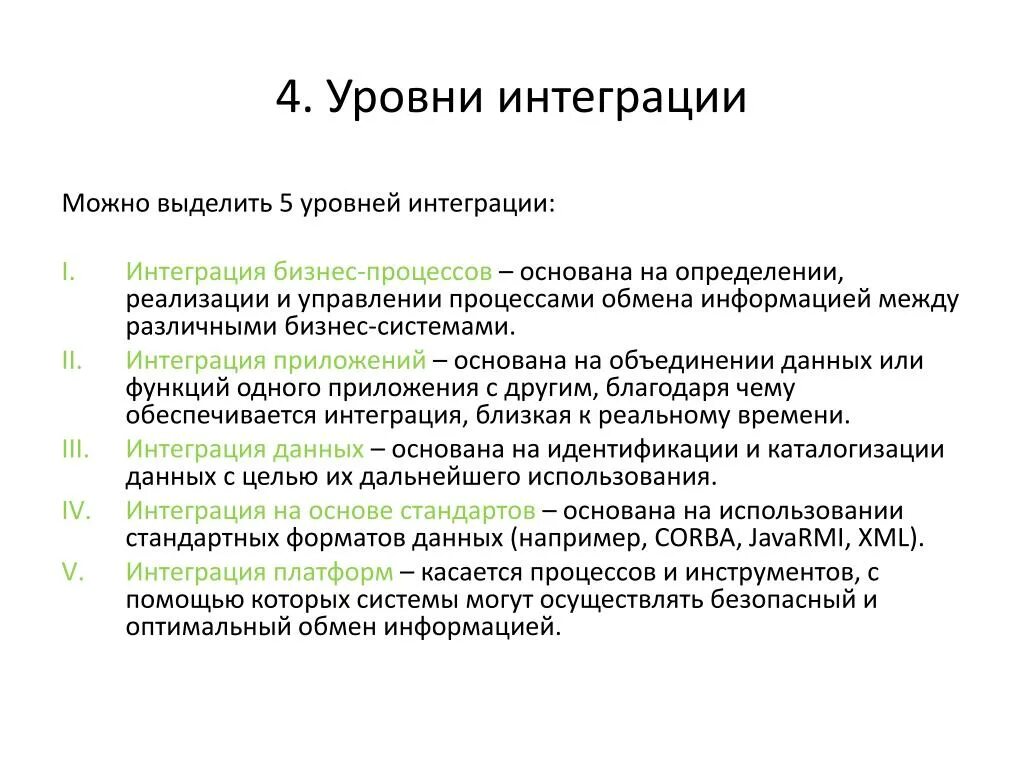
1. Методы организации работы в команде разработчиков.

<https://infourok.ru/metody-organizacii-raboty-v-komande-razrabotchikov-sistemy-kontrolya-versij-6575731.html>

1. Системы контроля версий.

<https://infourok.ru/lekciya-1-sistemy-kontrolya-versij-7279526.html>

1. Основные подходы к интегрированию программных модулей.



1. Стандарты кодирования.

**Стандарты кодирования** — это набор правил и соглашений, которые определяют, как писать и оформлять программный код. Они регламентируют всё: от именования переменных до структуры файлов проекта.

**Некоторые преимущества соблюдения стандартов кодирования:**

* **Повышение эффективности разработки**. Упрощает поддержку и обслуживание кода благодаря единому стилю написания. Сокращает время на понимание кода новыми участниками проекта.
* **Улучшение качества кода**. Снижает количество ошибок за счёт проверенных практик. Облегчает код-ревью благодаря унификации стиля.

**Каждый язык программирования имеет свои официальные стандарты кодирования**. Некоторые из них:

* **PEP 8**. Официальное руководство по стилю для языка программирования Python. Содержит рекомендации по отступам, соглашениям об именах и расположению кода.

1. Описание требований: унифицированный язык моделирования - краткий словарь.

**Унифицированный язык моделирования (UML)** — это графический язык для визуализации, спецификации, конструирования и документирования артефактов программных систем. [github.com](https://github.com/RichCake/diagram)

Некоторые термины из краткого словаря UML:

* **Класс** — шаблон для создания объектов, описывающий их состояние (атрибуты) и поведение (операции).
* **Объект** — экземпляр класса.
* **Атрибут** — свойство объекта или класса.
* **Операция (метод)** — действие, которое может выполнять объект или класс.
* **Ассоциация** — отношение между двумя классами.
* **Наследование** — отношение, при котором один класс (потомок) наследует свойства и методы другого класса (родителя).
* **Зависимость** — отношение, при котором изменение одного элемента влияет на другой.

1. Диаграммы UML.

UML — это набор правил, по которым нужно рисовать схемы. Зная его, можно быстро создавать универсальные графические представления сложных процессов и структур. Именно поэтому IT-специалисты во время разработки ПО часто используют UML-моделирование и проектирование процессов.

Язык UML нужен, чтобы описать и визуализировать какую-то абстрактную модель. На практике это может быть:  
  
● Создание модели объекта. Например, описание [структуры базы данных](https://practicum.yandex.ru/blog/chto-takoe-bazy-dannyh/?utm_campaign=Yan_Perfmax_RF_Data_Resk_b2c_Gener_Regular_Blog-13-tCPA-direct-boost_460&utm_content=sty_search%3As_none%3Acid_701648386%3Agid_5616732641%3Akw_%D0%B4%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B%20%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%20%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0%20uml%3Apid_55152283598%3Aaid_17154558697%3Acrid_0%3Arid_55152283598%3Ap_1%3Apty_premium%3Amty_syn%3Amkw_%3Adty_desktop%3Acgcid_0%3Arn_%D0%A0%D0%B5%D1%83%D1%82%D0%BE%D0%B2%3Arid_21621&utm_medium=cpc&utm_source=yandex&utm_term=%D0%B4%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B%20%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F%20%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0%20uml&yclid=3399951237576916991).  
● Создание модели процессов. Например, последовательность выполнения запросов ПО, чтобы клиент получил ожидаемый результат.  
  
Схему на языке UML можно составить по уже существующему объекту или процессу либо создать на этапе проектирования, чтобы разрабатывать объект или отлаживать процесс. Диаграммы UML применяют в проектировании, презентациях, описании или создании документации.

1. Описание и оформление требований (спецификация).

<https://infourok.ru/prezentaciya-na-temu-opisanie-i-oformlenie-trebovanij-specifikaciya-analiz-trebovanij-i-strategii-vybora-resheniya-6131599.html>

12. Анализ требований и стратегии выбора решения

<https://infourok.ru/prezentaciya-na-temu-opisanie-i-oformlenie-trebovanij-specifikaciya-analiz-trebovanij-i-strategii-vybora-resheniya-6131599.html>

13. Цели и задачи, и виды тестирования.

**ели тестирования** программного обеспечения:

* повысить вероятность того, что продукт будет работать правильно при любых обстоятельствах;
* проверить, что итоговый контент соответствует изначально выдвинутому набору требований;
* предоставить актуальную информацию о том, в каком состоянии программа находится на текущий момент;
* обнаружить ошибки, чтобы исправить их до релиза.

**Некоторые задачи тестирования**:

* проверить, что система работает в соответствии с определёнными временами отклика клиента и сервера;
* проверить, что наиболее критические последовательности действий с системой конечного пользователя выполняются верно;
* проверить работу пользовательских интерфейсов;
* проверить, что изменения в базах данных не оказывают неблагоприятного влияния на существующие программные модули.

**Некоторые виды тестирования**:

* **По характеру сценариев**:
  + **Позитивное тестирование** — проверка правильной работы программы в обычных условиях.
  + **Негативное тестирование** — проверка поведения программы при вводе некорректных данных.
* **По критериям запуска программы**:
  + **Статическое тестирование** — анализ документации, кода и требований без запуска программы.
  + **Динамическое тестирование** — проверка программы в процессе её выполнения.
* **По степени автоматизации**:
  + **Ручное тестирование** — выполняется тестировщиком без применения автоматизированных инструментов.
  + **Автоматизированное тестирование** — используются специальные инструменты для выполнения тестов.
* **По объектам тестирования**:
  + **Функциональное тестирование** — проверка соответствия функциональности требованиям.
  + **Нефункциональное тестирование** — проверка нефункциональных требований к системе, например производительности, безопасности, совместимости и других.
* **По степени знания системы**:
  + **Тестирование «чёрного ящика»** — проверка функциональности без доступа к внутреннему коду.
  + **Тестирование «белого ящика»** — анализ внутренней структуры и кода системы.

14. Стандарты качества программной документации.

<https://infourok.ru/prezentaciya-na-temu-principy-i-standarty-dokumentirovaniya-programmnyh-sredstv-4384903.html>

15. Меры и метрики.

**Меры** — это количественные показатели, используемые для оценки характеристик программного обеспечения или процесса разработки. Примеры мер: количество строк кода (LOC), количество дефектов, время выполнения теста. [github.com](https://github.com/RichCake/diagram)

**Метрики** — это стандартизированные способы измерения этих характеристик, часто выражаемые как отношение двух мер. Примеры метрик: плотность дефектов (дефектов на LOC), процент тестового покрытия, среднее время отклика

16. Тестовое покрытие.

**Тестовое покрытие** — это метрика, используемая для измерения качества тестирования программного обеспечения. Она показывает, какой процент кода приложения был выполнен в процессе тестирования. [sky.pro](https://sky.pro/media/chto-takoe-testovoe-pokrytie/)

**Существует несколько видов тестового покрытия**:

1. **Покрытие строк кода** (Line Coverage) — процент строк кода, которые были выполнены тестами. Важно учитывать, что не все строки кода необходимо тестировать, например, комментарии и пространства имён.
2. **Покрытие ветвей** (Branch Coverage) — процент ветвей кода (if, else, switch и т.д.), которые были выполнены тестами. Этот вид покрытия позволяет оценить тестирование разных сценариев выполнения кода.
3. **Покрытие функций** (Function Coverage) — процент функций и методов, которые были выполнены тестами. Это позволяет оценить, насколько хорошо каждая функция или метод приложения была протестирована.

17. Тестовый сценарий.

В тестовом сценарии подробно расписана последовательность действий, которую выполняет тестировщик для проверки работоспособности ПО. Он состоит из нескольких пунктов, в которых описано, что именно следует сделать с разработанной программой и какой должен быть результат.

Проходясь по тестовому сценарию, специалист последовательно выполняет эти пункты и смотрит, соответствует ли реальный результат ожидаемому. Если результат не тот или какой-то пункт выполнить невозможно — он составляет об этом отчёт.

18. Тестовый пакет.

**Тестовый пакет** (Test Suite) — это коллекция логически сгруппированных тестовых сценариев, которые предназначены для тестирования конкретной функциональности, модуля или всей системы.

**Некоторые элементы, которые включает в себя тестовый пакет:**

* план тестирования (цели и стратегия тестирования);
* тестовые случаи;
* тестовые данные;
* тестовые среды;
* другая соответствующая документация.

**Тестовый пакет важен**, так как обеспечивает систематический подход к тестированию и гарантирует, что все аспекты программного обеспечения или системы тщательно проверены. Он помогает выявлять дефекты, проверять функциональность и обеспечивать общее качество продукта

19. Анализ спецификаций.

**Анализ спецификаций** — это процесс критической оценки документации с требованиями (спецификаций) для выявления любых неточностей, неполноты, неоднозначностей, противоречий или избыточности.

Цель анализа — убедиться, что требования ясны, полны и могут быть реализованы.

20. Верификация и аттестация программного обеспечения.

<https://studfile.net/preview/1848692/page:59/>

21. Жизненный цикл программного обеспечения. Краткая характеристика каждого этапа.

<https://habr.com/ru/companies/kaiten/articles/893866/>

**Жизненный цикл разработки программного обеспечения (SDLC, Software Development Life Cycle)** описывает последовательность этапов, через которые проходит проект от идеи до завершения и поддержки программного продукта. Каждый этап имеет свои задачи, результаты и методы работы. Ниже приведены краткие характеристики четырёх этапов SDLC: планирования, анализа требований, проектирования и тестирования.

**Планирование**

**Цель**: разработать детальный план проекта, определить сроки, бюджет, ресурсы и задачи. Некоторые аспекты этапа:

* определение команды разработки, распределение обязанностей;
* создание графика работ;
* оценка рисков, разработка планов управления ими.

Планирование — основа для дальнейших этапов, поэтому важно уделить этому процессу достаточно времени.

**Анализ требований**

**Цель**: собрать и документировать требования к программному обеспечению. Некоторые особенности этапа:

* взаимодействие с заказчиками, пользователями и другими заинтересованными сторонами для определения функциональных и нефункциональных требований;
* создание документа с чёткими и понятными требованиями, который служит основой для проектирования.

Анализ требований — итерационный процесс, который продолжается на протяжении всего SDLC, и по ходу проекта могут происходить изменения.

**Проектирование**

**Цель**: создать архитектуру системы, чтобы разработчики точно понимали, как должна функционировать система. Работа делится на два уровня:

* **Верхнеуровневое проектирование** — отвечает за общую схему: из чего состоит система (модули, сервисы, подсистемы), как эти части взаимодействуют между собой.
* **Низкоуровневое проектирование** — отвечает за детали: определяют, какие алгоритмы и структуры данных будут использоваться, разрабатывают внутреннюю логику модулей.

Результат проектирования — создание архитектурной схемы и прототипов, которые помогут визуализировать систему.

**Тестирование**

**Цель**: проверить корректность работы продукта и соответствие заявленным характеристикам. На этом этапе проводится всестороннее тестирование: проверяют, как компоненты функционируют друг с другом, насколько система устойчива к нагрузке, и насколько она удобна для конечного пользователя.

В рамках тестирования могут использоваться разные подходы — от ручной проверки до автоматизированных сценариев. Если обнаруживаются ошибки, они возвращаются разработчикам на доработку.

Тестирование помогает выявить ошибки и недочёты, а также проверить, соответствует ли продукт требованиям.

22. Разработка пользовательских интерфейсов. Типы пользовательских интерфейсов и этапы их разработки.

**Пользовательский интерфейс (UI)** — это совокупность элементов, через которые пользователи взаимодействуют с программами или устройствами. Он включает в себя все визуальные компоненты, такие как кнопки, меню, иконки и формы, а также механизмы взаимодействия.

**Типы пользовательских интерфейсов** в зависимости от того, как представлен интерфейс и как человек с ним взаимодействует, выделяют такие типы:

* **Текстовые**. Для управления и взаимодействия с системой используют текст или программный код, а система отвечает таким же текстом или кодом. Разновидность текстового интерфейса — интерфейс командной строки.
* **Тактильные**. Для управления используют пальцы рук, двигая ими по экрану. Тактильные интерфейсы используют в системах навигации в больших ТЦ, аэропортах или музеях.
* **Голосовые**. Пользователь взаимодействует с системой или устройством с помощью голоса. Сюда относят все виды голосовых помощников, умные колонки, элементы умного дома, голосовые системы управления автомобилем.
* **Жестовые**. Управляются с помощью жестов рук, а иногда ещё и движений тела и головы. Жестовое управление используется, к примеру, в онлайн-играх — VR- или обычных.
* **Нейронные**. Самые продвинутые интерфейсы, где управлять системой можно с помощью нейронов мозга или, проще говоря, силой мысли.
* **Графические**. В виде окна с различными элементами меню, которыми можно управлять с помощью мыши, стилуса или пальцев рук.

**Этапы разработки пользовательского интерфейса**:

1. **Исследование и анализ**. Проводится исследование целевой аудитории и конкурентов, чтобы понять, какие потребности и ожидания есть у пользователей.
2. **Постановка задач и целей**. На основе собранной информации формулируются задачи и цели проекта.
3. **Создание пользовательских сценариев**. Разрабатываются сценарии использования, которые описывают, как пользователи будут взаимодействовать с интерфейсом.
4. **Создание каркасов**. Каркасы (wireframes) — это схематические эскизы, которые показывают структуру и компоновку элементов интерфейса без детальной проработки дизайна.
5. **Проектирование дизайна**. Разрабатывается визуальный дизайн интерфейса, включающий цветовые схемы, типографику, иконки и другие графические элементы.
6. **Разработка прототипов**. Прототипы — это интерактивные модели интерфейса, которые позволяют протестировать, как будет работать UI, до его фактической разработки.
7. **Запуск и мониторинг**. После завершения разработки и тестирования интерфейс запускается в рабочей среде.
8. **Поддержка и обновление**. Поддержка включает регулярное обновление интерфейса, добавление новых функций и исправление ошибок.

23. Техническое задание. Разделы, входящие в техническое задание.

Техническое задание — документ проекта, который помогает команде и заказчику понять, каким он должен получиться в итоге. По сути, ТЗ определяет требования к IT- продукту: например, какие функции он должен выполнять, какие пользовательские сценарии в нём есть, на чём пишется приложение и так далее.   
  
Обычно ТЗ включает такие блоки:  
  
● **Назначение проекта.** Описывает пользовательские и бизнес-цели продукта.  
● **Пользовательские группы.** Сюда входят разные категории целевых аудиторий проекта.  
● **Обзор содержания.** Большой блок технического задания, который включает его функции и сценарии использования.  
● **Взаимодействие с другими компонентами.** Описывает, как и с какими сторонними компонентами будет интегрироваться продукт, например, с Яндекс Картами.  
● **Обзор интерфейса.** В этот блок входят пожелания по структуре и дизайну страниц. Например, сюда можно включить использование фирменного стиля клиента.  
● **Безопасность.** Здесь описываются все системы безопасности, которые обязаны быть в продукте, допустим, файрволы, антивирусы и системы шифрования.  
● **Разработка.** Если у клиента есть пожелания к технической части, например, предпочтительный язык разработки, то его стоит указать в этом блоке.   
● **Системное окружению.** Технические характеристики серверов и устройств, на которых будет установлен продукт.

24. Использование языка UML при проектировании сложных программных систем.

**Язык UML (Unified Modeling Language) используется при проектировании сложных программных систем** для визуального описания, проектирования и документирования систем. Он не связан с конкретным языком программирования, подходит для проектирования систем любой сложности — от небольшого мобильного приложения до распределённой корпоративной платформы.

25. Диаграмма вариантов использования, ее назначение. Правила построения диаграммы вариантов использования.

**Диаграмма вариантов использования** (Use Case Diagram) — это визуальное представление взаимодействия пользователей с системой. Она относится к языку моделирования UML (Unified Modeling Language) и выступает ключевым инструментом на этапе сбора и анализа требований.

**Назначение** диаграммы — показать, **что** система делает с точки зрения пользователя, а не **как** она это делает технически. Некоторые преимущества использования таких диаграмм:

* **Общий язык общения** — заказчики и разработчики видят одну и ту же картину системы.
* **Упрощение планирования** — каждый вариант использования может стать основой для пользовательской истории.
* **Выявление требований** — процесс создания диаграммы часто помогает обнаружить скрытые функции.
* **Документирование** — наглядное представление функциональных требований к системе.
* **Оценка объёма работ** — прецеденты позволяют точнее оценить трудозатраты.

**Некоторые правила построения диаграммы вариантов использования**:

* **Использовать правильный уровень абстракции** — не углубляться в технические детали, но и не оставаться на слишком высоком уровне. Прецедент должен представлять законченное действие с ценностью для пользователя.
* **Применять цветовое кодирование** — выделять разные типы актёров или группы функциональности различными цветами для быстрого визуального восприятия.
* **Использовать слои** — многие онлайн-редакторы позволяют организовать элементы по слоям, что упрощает работу со сложными диаграммами.
* **Применять шаблоны** — не начинать с чистого листа, использовать готовые шаблоны для типовых сценариев (авторизация, управление профилем и т. д.).
* **Оставлять комментарии** — большинство онлайн-инструментов поддерживают комментирование, что полезно для командной работы и фиксации вопросов.

26. Понятие класса и объекта. Атрибут и операция.

**Класс** — это множество объектов, связанных общностью свойств, поведения, связей и семантики. Класс является абстрактным определением объекта и служит в качестве шаблона для создания объектов.

**Объект** — это конкретный представитель класса, построенный по хранящемуся в классе описанию. Объект может представлять собой абстракцию некоторой сущности предметной области (объект реального мира) или программной системы (архитектурный объект).

**Атрибут** — это значение, характеризующее объект в его классе. Среди атрибутов различаются постоянные атрибуты (константы) и переменные атрибуты. Постоянные атрибуты характеризуют объект в его классе (например, номер счёта, категория, имя человека и т. п.). Текущие значения переменных атрибутов характеризуют текущее состояние объекта (например, баланс счёта, возраст человека и т. п.).

**Операция** — это функция (или преобразование), которую можно применять к объектам данного класса. Примеры операций: проверить, снять, поместить (для объектов класса счёт), открыть\_на\_чтение, читать, закрыть (для объектов класса файл) и т. п.. Одна и та же операция может применяться к объектам разных классов: такая операция называется полиморфной, так как она может иметь разные формы для разных классов.

27. Диаграмма потоков данных. Основное назначение.

**Диаграмма потоков данных (DFD)** — это графическое представление потока данных в информационной системе. С её помощью можно описывать входящие и выходящие потоки данных и хранилища этих данных. [practicum.yandex.ru](https://practicum.yandex.ru/blog/diagramma-potokov-dannyh-dfd/)

**Основное назначение DFD-диаграмм**:

* **Моделирование функциональных требований** к проектируемой системе. Требования представляются в виде иерархии процессов, связанных потоками данных.
* **Понимание того, как данные обрабатываются, хранятся и передаются внутри системы**.
* **Идентификация потока данных и потенциальных узких мест в системе**.
* **Донесение структуры системы** до нетехнических заинтересованных сторон и обеспечение визуального представления архитектуры системы.



28. Основные принципы структурной методологии. Особенности структурных программ. Цели структурного программирования.

**Основные принципы структурного программирования**:

* **Разделение сложных задач на модули**. Каждый модуль решает конкретную задачу, что делает его понятным даже без глубокого анализа. Модули можно применять в различных частях или даже в других проектах.
* **Принцип однозначной ответственности**. Каждая часть программы должна быть спроектирована так, чтобы отвечать за выполнение одной конкретной задачи. Это упрощает внесение изменений, а также снижает вероятность того, что исправление одной ошибки приведёт к созданию новых проблем в других частях софта.
* **Иерархическое проектирование**. На верхнем уровне формулируются основные цели. На нижних уровнях прорабатываются детали реализации, представленные, например в виде процедур. Такая структура облегчает анализ, позволяет масштабировать проект без потери управляемости.
* **Алгоритмическая точность**. Кодирование реализуется с точным соответствием алгоритмам, разработанным заранее. Блок-схемы или псевдокод, используемые на этапе проектирования, должны полностью отражаться в реализации. Это помогает избежать логических ошибок и повышает предсказуемость работы.

**Особенности структурных программ**:

* **Чёткая структура**. Использование базовых структур, таких как последовательности, ветвления, циклы, помогает сделать кодирование логически простым и организованным.
* **Лёгкость в модификациях**. Когда программа организована в виде модулей, изменения в одной части не влияют на остальные её компоненты. Это позволяет без значительных усилий обновлять или расширять систему, обеспечивая гибкость и удобство в поддержке и добавлении новых функциональных возможностей.
* **Уменьшение количества вариантов построения программы**. Это значительно снижает сложность программы и облегчает понимание её другими разработчиками.

**Цели структурного программирования**:

* **Обеспечение дисциплины программирования** в процессе создания программных комплексов.
* **Улучшение читабельности программы**.
* **Повышение эффективности программ**. Этого можно достигнуть, если выполнять структурирование программы, разбивая её на модули так, чтобы можно было легко находить и корректировать ошибки.
* **Повышение надёжности программ**. Этого можно достигнуть, если программа будет легко поддаваться сквозному тестированию и не создаст проблем для организации процесса отладки.
* **Уменьшение времени и стоимости программной разработки**. Это происходит в том случае, если каждый программист команды разработчиков становится способным писать и отлаживать большее количество программного кода, чем раньше.

29. Модульное программирование (определение). Цели модульного программирования. Достоинства модульного программирования.

**Модульное программирование** — это методология разработки программного обеспечения, основанная на создании отдельных модулей, которые можно независимо разрабатывать, тестировать и поддерживать. Каждый модуль выполняет конкретную функцию и имеет определённые входные и выходные данные.

**Цели модульного программирования**:

* более быстрая разработка кода;
* увеличение надёжности программы;
* более простое тестирование продукта;
* взаимозаменяемость модулей в разных проектах.

**Некоторые достоинства модульного программирования**:

* **Упрощение разработки**. Разбиение программы на отдельные модули позволяет упростить процесс разработки, так как каждый модуль фокусируется на конкретной задаче или функциональности.
* **Улучшенная отладка**. Разбиение программы на модули упрощает процесс отладки и исправления ошибок, так как каждый модуль может быть тестирован и отлажен независимо от других частей программы.
* **Лёгкая масштабируемость**. Модульное программирование обеспечивает гибкость в масштабировании программных проектов. Новые модули могут быть добавлены или удалены без причинения значительных изменений в других частях программы.
* **Лёгкая поддержка и обновление**. Если необходимо внести изменения в систему, то модульное программирование облегчает этот процесс. Модули можно изменять или заменять независимо друг от друга, минимизируя риск возникновения ошибок и упрощая тестирование.
* **Повторное использование кода**. Модули разрабатываются независимо друг от друга и повторно используются в дальнейшем. Это сокращает время разработки и повышает эффективность команды.
* **Сохранность информации**. Использование модулей позволяет скрыть внутренние детали реализации и предоставить только интерфейс для взаимодействия с другими модулями.

30. Объектно-ориентированное программирование. Основные понятия: объект, свойство объекта, метод обработки, событие, класс объектов. Метод объектно-ориентированной декомпозиции, метод абстрактных типов данных, метод пересылки сообщений.

**Объектно-ориентированное программирование (ООП)** — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

**Некоторые основные понятия ООП:**

* **Объект** — структура данных в программировании, например пользователь, кнопка, сообщение. Объект обладает состоянием, поведением и идентичностью.
* **Свойства объекта** — это переменные и функции, которые являются атрибутами какого-то объекта (их называют полями объекта). Например, цвет поля или имя пользователя.
* **Методы** — это функции, которые описаны внутри объекта или класса. Они относятся к определённому объекту и позволяют взаимодействовать с ними или другими частями кода.
* **Класс** — это «шаблон» для объекта, который описывает его свойства. Несколько похожих между собой объектов, например профили разных пользователей, будут иметь одинаковую структуру, а значит, принадлежать к одному классу.
* **Сообщение** — это запрос на выполнение действия, содержащий набор необходимых параметров. Объекты взаимодействуют между собой, посылая и получая сообщения.

**Метод объектно-ориентированной декомпозиции** заключается в том, чтобы разбить программу на достаточное количество классов и модулей, каждый из которых предназначен для выполнения одной чётко очерченной задачи, которую будет выполнять отдельный объект.

**Метод абстрактных типов данных (АТД)**

**АТД** — это тип данных, который определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Внутренняя структура такого типа скрыта от разработчика программного обеспечения.

**Метод пересылки сообщений**

**Сообщение** — это запрос к объекту-получателю на выполнение определённого метода. Сообщение содержит имя метода и аргументы метода. 

31. Надежность программного изделия. Работоспособность программного изделия. Основные количественные показатели надежности программного изделия.

**Надёжность программного изделия** — это способность системы работать в соответствии с установленными требованиями и целями в заданных условиях использования. Это комплексное свойство включает две составляющие: безошибочность (соответствие спецификации) и устойчивость (отказоустойчивость).

**Работоспособность программного изделия** — это способность выполнять заданные функции, сохраняя значения основных параметров в пределах, установленных нормативно-технической документацией.

**Основные количественные показатели надёжности программного изделия**:

* **Вероятность безотказной работы** в течение определённого времени (P(t)).
* **Средняя наработка до первого отказа** (Т(ср)).
* **Вероятность отказа** (Q(t)).
* **Наработка на отказ** (t(ср)). Учитывает ситуации потери работоспособности, когда длительность восстановления достаточно велика и превышает пороговое значение времени, разделяющее события сбоя и отказа.
* **Коэффициент готовности**. Отражает вероятность иметь восстанавливаемую систему в работоспособном состоянии в произвольный момент времени.

32. Определение тестирования и отладки. Особенности и объекты тестирования. Автономное и комплексное тестирование.

**Тестирование** — это процесс многократного выполнения программы с целью выявления ошибок. **Отладка** — процесс устранения ошибок, выявленных на стадии тестирования. Хотя слова «отладка» и «тестирование» часто используются как синонимы, но это разные виды деятельности. Тестирование направлено на обнаружение ошибок, а отладка — на установление точной природы известной ошибки и её исправление.

**Особенности тестирования**:

* отсутствует программа-эталон, которой должны соответствовать все результаты тестирования проверяемой программы;
* невозможно создать тестовый набор для исчерпывающей проверки программы;
* следует избегать трудно или невоспроизводимых тестов;
* нужно создавать тесты, которые действительно находили бы ошибки, а не демонстрировали правильность выполнения программы;
* нет формализованных критериев качества программ и процесса их испытания;
* для тестирования нужно привлекать сторонних специалистов.

**Наиболее характерные объекты тестирования**: требования и спецификация, программные модули, группы программ, решающие законченные функциональные возможности.

**Автономное тестирование** — это метод, который включает автономную проверку отдельного модуля или подсистемы. Весь программный комплекс не тестируется, проверяется лишь структура компонента и вычисления и преобразования данных внутри него. Разделяют восходящее и нисходящее направления автономного тестирования:

* **Восходящее**. Тестируются физические модули самого нижнего уровня, затем — вызывающие их модули, и так до самого высшего. Плюсы: модули тестируются детально, это позволяет устранить большое количество ошибок, относительная простота подготовки тестов. Минусы: нужно постоянно обновлять набор тестовых данных при подключении нового модуля, большая вероятность появления новых ошибок при подключении модулей, необходимо имитировать работу вышестоящих модулей.
* **Нисходящее**. В первую очередь проверяется главный модуль, затем — модули более низких уровней и так далее. Плюсы: накапливаются тестовые данные, которые можно не менять при подключении модулей.

**Комплексное тестирование** — это контроль и/или испытание системы по отношению к исходным целям. Комплексное тестирование является процессом контроля, если оно выполняется в моделируемой среде, и процессом испытания, если выполняется в реальной среде.

33. Управление разработкой программных средств. Средства управления проектами. Основная цель управления жизненным циклом программных средств.

**Управление разработкой программных средств** — это управление программным проектом, которое включает в себя процессы выполнения требований к продукту путём планирования, реализации, мониторинга и контроля проектных работ.

**Средства управления проектами** включают в себя, например:

* **Системы управления разработкой программного обеспечения (СУРПО)**. Они участвуют в организации, планировании, контроле и выполнении всех аспектов разработки. Такие системы способствуют структурированному подходу к менеджменту проектов, включая создание детализированных графиков, определение стадий разработки и распределение задач среди членов команды.
* **Концепцию жизненного цикла (SDLC)**. Она определяет основные фазы, через которые проходит продукт от зарождения идеи до его вывода на рынок и дальнейшей поддержки. Использование методологии SDLC позволяет комплексно управлять этапами создания ПО с учётом требований, ресурсов и рисков.
* **Специализированные системы управления командной разработки ПО**. Они предоставляют широкий набор инструментов для планирования, контроля версий, управления задачами, аналитики на всех этапах цикла. Это помогает повысить скорость и качество разработки, снизить риски и издержки.

**Основная цель управления жизненным циклом программных средств** — достижение целей проекта при нахождении компромисса между объёмом работ, ресурсами, временем, качеством и рисками.

34. Инструментальные средства разработки программ. Инструментальные среды программирования. Средства автоматизации разработки программ (CASE-средства). Интегрированные среды.

**Инструментальные средства разработки программного обеспечения** — это программные инструменты, предназначенные для обеспечения полного цикла проектирования программного продукта (написание текста программы, компиляция, компоновка, отладка, тестирование, сопровождение и другие).

**Некоторые виды инструментальных средств**:

* **Специализированные редакторы текста**. Предназначены для ввода и редактирования исходного текста программы. Могут быть моноязычными, поддерживающими только лексику одного языка программирования, или мультиязычными, могут поставляться в виде отдельного приложения или встраиваться в интегрированную среду разработки (IDE).
* **Компиляторы**. Транслируют текст программы с языка программирования в машинный код (исполняемый файл) без её выполнения. Могут оптимизировать код программы с учётом версии языка программирования и особенностей аппаратной платформы, для которой производится трансляция.
* **Компоновщики (редакторы связей)**. Принимают на вход один или несколько объектных модулей и собирают по ним исполнимый модуль.
* **Отладчики**. Являются модулями среды разработки или отдельными приложениями, предназначенными для поиска ошибок в программе.
* **Программы для создания инсталляторов**. Обычно дистрибутивы программ создаются с интерфейсом «мастера», то есть пошагового диалога с пользователем.
* **Программы создания справочной системы**. Позволяют организовывать файлы помощи с нужной структурой, содержанием, возможностью поиска, контекстными подсказками, перекрёстными ссылками.

**Инструментальная среда программирования** предназначена в основном для поддержки процессов программирования (кодирования), тестирования и отладки программного обеспечения.

**Средства автоматизации разработки программ (CASE-средства)** — это программные комплексы, автоматизирующие весь технологический процесс жизненного цикла программного обеспечения. Главное преимущество CASE-технологий — поддержка коллективной работы разработчиков над проектом в локальной сети, экспорт и импорт любых фрагментов проекта, организованный процесс управления проектом до создания полного продукта.

**Интегрированная среда разработки (IDE)** — это оболочка, в которую объединяются большинство инструментальных средств, имеющая графический интерфейс. Такое решение позволяет увеличить производительность программистов за счёт унификации инструментальных средств и отсутствия необходимости переключения между отдельными компонентами.

35. Оценка качества программного обеспечения. Методы оценки свойств программного обеспечения.

**Оценка качества программного обеспечения** — это процесс, который позволяет определить качество и эффективность программного продукта в соответствии с требованиями заказчика. Он включает в себя анализ функциональных возможностей, надёжности работы, скорости выполнения задач и других параметров.

**Некоторые методы оценки качества программного обеспечения**:

* **Тестирование**. Наиболее распространённый метод. Может проводиться вручную или автоматически с помощью специальных инструментов. Тестирование позволяет выявлять ошибки и дефекты в работе программы, что даёт возможность исправить их до выпуска продукта.
* **Проверка кода**. Предполагает анализ кода программы на предмет соответствия стандартам и правилам написания кода. Этот метод позволяет выявлять потенциальные проблемы с производительностью или безопасностью.
* **Анализ требований**. Помогает определить, насколько корректно определены требования для создаваемого продукта. Анализ требований включает в себя проверку на соответствие бизнес-целям, адекватность и полноту.

**Методы оценки свойств программного обеспечения** различаются по способам получения информации о ПО и по источникам получения информации:

**По способам получения информации**:

* **Измерительный метод**. Основан на получении информации о свойствах и характеристиках ПО с использованием инструментальных средств. Например, определяется объём ПО, время выполнения ветви программы, время реакции и другие показатели.
* **Регистрационный метод**. Основан на получении информации во время испытаний или функционирования ПО, когда регистрируются и подсчитываются определённые события, например, время и число сбоев и отказов, время передачи управления другим модулям, время начала и окончания работы.
* **Органолептический метод**. Основан на использовании информации, получаемой в результате анализа восприятия органов чувств (зрения, слуха), и применяется для определения таких показателей, как удобство применения, эффективность и т. п..
* **Расчётный метод**. Основан на использовании теоретических и эмпирических зависимостей (на ранних этапах разработки), статистических данных, накапливаемых при испытаниях, эксплуатации и сопровождении ПО.

36. Внедрение программного комплекса. Подготовка тестовых данных. Анализ результатов испытаний.

**Внедрение программного комплекса**, **подготовка тестовых данных** и **анализ результатов испытаний** — ключевые аспекты жизненного цикла разработки программного обеспечения (ПО). Эти процессы связаны с установкой, настройкой и вводом в эксплуатацию ПО в рабочей среде пользователя, а также с оценкой работы программы для выявления ошибок и дефектов.

**Внедрение**

**Внедрение программного комплекса** — это процесс установки, настройки и ввода в эксплуатацию ПО в рабочей среде пользователя. Включает:

* планирование развёртывания;
* миграцию данных;
* обучение пользователей и переход на новую систему.

Точный процесс внедрения зависит от типа проекта: в заказном проекте его выполняют разработчики, в рыночном — покупатели.

**Подготовка тестовых данных**

**Подготовка тестовых данных** — это создание или сбор данных, которые будут использоваться при выполнении тестовых сценариев. Данные должны быть репрезентативными, разнообразными, включать как корректные, так и некорректные значения, а также граничные условия, чтобы максимально проверить функциональность системы.

Генерацию тестовых данных можно проводить несколькими способами:

* **Вручную** — создание данных в соответствии с требованиями к ИТ-продукту (например, вводят данные в форму, загружают файлы).
* **Автоматически** — подготовка данных с помощью специальных инструментов или скриптов, например, генераторов случайных данных, имитирующих различные сценарии использования.
* **Внедрение внутренних данных** — иногда для проверки качества ПО используют реальные данные из базы данных или из других систем организации.

**Анализ результатов испытаний**

**Анализ результатов испытаний** — это процесс оценки результатов выполнения тестовых сценариев. Включает:

* **Сравнение фактических результатов с ожидаемыми** — выявление расхождений (дефектов).
* **Документирование дефектов** — фиксация информации об ошибках (описание, шаги воспроизведения, серьёзность, приоритет).
* **Анализ корневых причин** — поиск причин возникновения дефектов.
* **Формирование отчётов** — предоставление информации о состоянии тестирования, количестве найденных дефектов, покрытии тестами и общем качестве ПО.
* **Принятие решений** — на основе анализа результатов принимаются решения о готовности продукта к релизу или необходимости доработки.