### **1. Технология программирования (определение, цели дисциплины). Отличие от программной инженерии**

**Технология программирования** — это совокупность методов и средств, используемых для разработки программного обеспечения (ПО), которое соответствует требованиям надежности, качества и эффективности.

### **Основные цели дисциплины**

1. **Повышение производительности труда программистов**
   * Ускорение процесса разработки.
2. **Улучшение качества создаваемого ПО**
   * Минимизация ошибок.
   * Оптимизация работы программ.
3. **Обеспечение надежности и технологичности программных продуктов**
   * Устойчивость ПО к сбоям.
   * Лёгкость в доработке и сопровождении.
4. **Снижение стоимости разработки и сопровождения ПО**
   * Рациональное использование ресурсов.
   * Повторное использование кода.

### **Отличие от программной инженерии**

* **Фокус технологии программирования**:  
   Основное внимание уделяется **техническим аспектам** разработки программного обеспечения, включая алгоритмы, языки программирования и инструменты.
* **Фокус программной инженерии**:  
   Рассматривает более широкий круг вопросов, включая:  
  + Управление проектами.
  + Экономические аспекты разработки.
  + Взаимодействие с заказчиком.
  + Юридические и организационные аспекты.

### **2. Основные этапы разработки (российский и международный стандарты)**

#### **Российский стандарт (ГОСТ 34.601-90)**

Этот стандарт регулирует создание автоматизированных систем (АС) и включает следующие этапы:

1. **Формирование требований к АС**
   * Сбор и анализ требований от заказчика.
   * Определение задач, которые должна решать система.
2. **Разработка концепции АС**
   * Определение общей структуры и принципов работы системы.
3. **Техническое задание**
   * Формирование детального документа с требованиями и ограничениями.
4. **Эскизный проект**
   * Предварительное проектирование с анализом альтернативных решений.
5. **Технический проект**
   * Разработка детализированного проекта системы.
6. **Рабочая документация**
   * Подготовка документации для реализации системы.
7. **Ввод в действие**
   * Установка, настройка и тестирование системы.
8. **Сопровождение АС**
   * Техническая поддержка и обновление системы.

#### **Международный стандарт (ISO/IEC 12207)**

ISO/IEC 12207 описывает жизненный цикл программного обеспечения и включает следующие этапы:

1. **Анализ требований (Requirements Analysis)**
   * Сбор, анализ и документирование требований.
2. **Проектирование архитектуры (Architectural Design)**
   * Разработка общей структуры системы.
3. **Детальное проектирование (Detailed Design)**
   * Проработка деталей компонентов системы.
4. **Кодирование и тестирование модулей (Coding and Unit Testing)**
   * Написание кода и тестирование отдельных модулей.
5. **Интеграция и тестирование системы (Integration and System Testing)**
   * Сборка компонентов в единую систему и их тестирование.
6. **Внедрение (Implementation)**
   * Установка системы у пользователя.
7. **Эксплуатация и сопровождение (Operation and Maintenance)**
   * Поддержка, обновление и исправление ошибок.

#### **Ключевые различия**

1. **Формализация**:  
   * ГОСТ 34.601-90 уделяет больше внимания созданию полного комплекта документации.
   * ISO/IEC 12207 более гибок и ориентирован на адаптацию к современным методологиям разработки.
2. **Предпроектное обследование**:  
   * В российском стандарте предпроектное обследование занимает важное место.
   * Международный стандарт концентрируется на начальных этапах анализа требований.
3. **Подход к разработке**:  
   * ГОСТ предполагает линейный подход с последовательным прохождением этапов.
   * ISO/IEC поддерживает итеративные и гибкие методы разработки, такие как Agile.

**3. Жизненный цикл ПО. Процессы ЖЦ. Модели ЖЦ ПО. Их достоинства и недостатки**

Каскадная

Инкрементная

Эволюционная

**Процессы** **ЖЦ**

Сбор и анализ требований

Проектирование

Кодирование

Тестирование

Развертывание

**Модели**



**Недостатки каскадной модели, которые необходимо учитывать при оценке возможности ее применения:**

• требования к объектам определены недостаточно четко;

• система обычно слишком велика, чтобы все работы по ее созданию выполнять однократно;

• предполагаемые скорые изменения в технологиях работ;

• возможные текущие изменения требований к системе;

• ограниченность ресурсов, например средств или персонала;

• промежуточный продукт может быть непригоден для использования.

*ГОСТ Р ИСО/МЭК ТО 15271-2002*

**Преимущества каскадной модели**

• однократное представление всех возможностей (характеристик) системы;

• необходимость только единственной фазы перехода от старой системы к новой.



**Достоинства**

Акцент на верификации и валидации

Четкие критерии перехода

Подходит для критических систем

**Недостатки**

Низкая гибкость

Высокая стоимость исправленных ошибок

Длительный цикл разработки



**Недостатки, которые необходимо учитывать при оценке возможности ее применения:**

• требования к объектам определены недостаточно четко;

• предполагаемые скорые изменения в технологиях работ;

• возможные текущие изменения требований к системе;

• привлечение ресурсов (средств или персонала) на длительный период ограничено.

**Преимущества использования данной модели:**

• необходимость изначального использования характеристик системы;

• пригодность для использования промежуточного продукта;

• возможности наращивания привлекаемого персонала и средств



**Достоинства спиральной модели:**

• наиболее реально отображает процесс разработки ПО;

• позволяет учитывать риск на каждом витке эволюции разработки;

• включает возможность оценки системы в итерационную структуру разработки • использует моделирование для уменьшения риска и совершенствования программного продукта

**Недостатки спиральной модели:**

• повышенные требования к заказчику;

• трудности контроля и управления временем разработки



### **4. Парадигмы программирования**

**Парадигма программирования** — это комплекс концепций, принципов и абстракций, определяющих фундаментальный стиль программирования. Парадигма задается использованием определённых сущностей, например:

* **Состояний программы и команд, изменяющих их** (императивное программирование).
* **Математических функций без состояний** (функциональное программирование).
* **Объектов и взаимодействий между ними** (объектно-ориентированное программирование).
* **Алгоритмов и контейнеров, работающих с типами данных, переданными как параметры** (обобщённое программирование).

#### **Основные парадигмы программирования**

**1. Императивное программирование**

* Основано на последовательном выполнении команд.
* Программа явно описывает последовательность шагов для достижения результата.
* Используются переменные, присваивания, циклы и условные операторы.
* **Примеры языков**: C, Pascal, Basic.
* **Подвиды**:
  + Процедурное программирование.
  + Структурное программирование.

**2. Функциональное программирование**

* Основано на математической концепции функций.
* Особенности:
  + Отсутствие побочных эффектов.
  + Неизменяемость данных (иммутабельность).
  + Использование функций высшего порядка.
  + Рекурсия вместо циклов.
* **Примеры языков**: Haskell, Lisp, Erlang.
* Характерные черты:
  + Чистые функции.
  + Ленивые вычисления.
  + Функции как объекты первого класса.

**3. Объектно-ориентированное программирование (ООП)**

* **Основные принципы**:
  + Инкапсуляция (сокрытие реализации).
  + Наследование (переиспользование кода).
  + Полиморфизм (множественность форм).
  + Абстракция (выделение существенных характеристик).
* **Примеры языков**: Java, C++, Python.
* Ключевые концепции:
  + Классы и объекты.
  + Методы и свойства.
  + Интерфейсы и абстрактные классы.

**4. Обобщённое программирование**

* Написание кода, который может работать с разными типами данных.
* Использование шаблонов и параметризованных типов.
* **Примеры**: шаблоны в C++, дженерики в Java.
* Характерные черты:
  + Переиспользование алгоритмов.
  + Типобезопасность.
  + Повышение абстракции кода.

**5. Декларативное программирование**

* Описание желаемого результата вместо алгоритма его получения.
* **Подвиды**:
  + Логическое программирование (например, Prolog).
  + SQL.
  + HTML/CSS.

**6. Событийно-ориентированное программирование**

* Программа реагирует на события.
* Широко используется в GUI-приложениях.
* Основано на обработчиках событий.
* **Примеры**: JavaScript в веб-разработке.

**7. Аспектно-ориентированное программирование**

* Разделение сквозной функциональности.
* Уменьшение связности кода.
* Примеры применения: логирование, безопасность, транзакции.

**8. Параллельное программирование**

* Одновременное выполнение нескольких задач.
* Использование многопоточности и многопроцессорности.
* Включает синхронизацию и взаимодействие между процессами.

#### **Важные моменты**

* Современные языки программирования часто поддерживают несколько парадигм (мультипарадигменные языки).
* Выбор парадигмы зависит от:
  + Задачи.
  + Требований к производительности.
  + Предметной области.
  + Опыта команды.

**Модели структурирования системы. Моделирование управление.**

### **1. Модели структурирования системы**

Модели структурирования описывают, как компоненты системы организованы и взаимодействуют между собой. Это помогает понять архитектуру системы и разрабатывать её структуру в соответствии с требованиями.

#### **Основные типы моделей структурирования:**

* **Иерархическая модель**:  
   Организует систему в виде дерева, где узлы представляют компоненты или подсистемы, а связи между ними определяют зависимости.  
   Пример: файловая система.
* **Клиент-серверная модель**:  
   Разделяет систему на клиентские (запрашивающие) и серверные (обрабатывающие) части.  
   Пример: веб-приложения.
* **Слойная модель**:  
   Система разделена на слои, каждый из которых взаимодействует только с соседними слоями.  
   Пример: сетевые протоколы (OSI).
* **Объектно-ориентированная модель**:  
   Представляет систему как набор объектов, которые взаимодействуют друг с другом через методы.  
   Пример: программы на C# или Java.

#### **Основные преимущества моделей структурирования:**

* Обеспечивают модульность и гибкость.
* Упрощают анализ и разработку.
* Способствуют повторному использованию компонентов.

### **2. Моделирование управления**

Моделирование управления направлено на описание потоков управления внутри системы, включая последовательности выполнения задач и взаимодействие между компонентами.

#### **Основные подходы к моделированию управления:**

* **Диаграммы состояний**:  
   Показывают, как объект или система переходят между различными состояниями под воздействием событий.  
   Пример: автомат управления светофором.
* **Диаграммы активности**:  
   Определяют последовательность действий или поток управления.  
   Пример: обработка заказа в интернет-магазине.
* **Диаграммы последовательностей (Sequence Diagram)**:  
   Показывают взаимодействие объектов во времени, акцентируя внимание на последовательности вызовов.  
   Пример: взаимодействие пользователя с системой авторизации.
* **Модели конечных автоматов**:  
   Формальная модель, описывающая поведение системы как набор состояний и событий, которые их изменяют.

#### **Важные аспекты моделирования управления:**

1. **События и действия**: события инициируют переходы между состояниями, а действия выполняются в ответ на события.
2. **Синхронизация**: моделирование точек, где несколько потоков управления должны координироваться.
3. **Параллелизм**: определение частей системы, которые могут работать независимо друг от друга.

**5. Методы разработки структуры программ (восходящая и нисходящая разработка, конструктивный и архитектурный подходы)**

### **1. Нисходящая разработка (Top-Down)**

Нисходящий подход основан на движении от общего к частному, начиная с общей концепции и заканчивая детализацией.

#### **Принципы:**

* Разработка начинается с общего описания задачи.
* Постепенная декомпозиция на более мелкие подзадачи.
* Последовательное движение от общего к частному.

#### **Преимущества:**

* Создаётся общий обзор всей системы с самого начала.
* Упрощает планирование и управление проектом.
* Позволяет увидеть структуру программы раньше.
* Облегчает документирование.

#### **Недостатки:**

* Базовые функции тестируются поздно.
* Сложно точно оценить время на реализацию деталей.
* Высокий риск переделки при обнаружении проблем на нижних уровнях.

### **2. Восходящая разработка (Bottom-Up)**

Восходящий подход строится на создании базовых элементов, которые объединяются в крупные компоненты.

#### **Принципы:**

* Начинается с разработки базовых компонентов.
* Постепенное объединение в более крупные части.
* Движение от частного к общему.

#### **Преимущества:**

* Раннее тестирование базовых элементов.
* Повторное использование кода.
* Точнее оценивается время разработки.
* Быстро создаются работающие компоненты.

#### **Недостатки:**

* Трудности с формированием общей картины системы.
* Риск создания ненужных элементов.
* Возможные проблемы с интеграцией компонентов.

### **3. Конструктивный подход**

Этот подход предполагает использование готовых компонентов и их сборку в единую систему.

#### **Основные принципы:**

* Использование готовых компонентов.
* Создание библиотек повторно используемых элементов.
* Сборка системы из проверенных блоков.

#### **Характеристики:**

* Акцент на повторное использование кода.
* Стандартизация интерфейсов.
* Модульная разработка.
* Тестирование компонентов по отдельности.

#### **Преимущества:**

* Высокая надежность.
* Сокращение времени разработки.
* Упрощение сопровождения.

### **4. Архитектурный подход**

Архитектурный подход фокусируется на создании общей структуры системы и определении её ключевых компонентов.

#### **Основные принципы:**

* Фокус на общей структуре системы.
* Определение ключевых компонентов и их взаимодействия.
* Учет нефункциональных требований.

#### **Ключевые аспекты:**

* Выбор архитектурного стиля.
* Определение интерфейсов между компонентами.
* Учет масштабируемости и производительности.
* Планирование развития системы.

#### **Типичные архитектурные стили:**

* Многослойная архитектура.
* Микросервисы.
* MVC (Model-View-Controller).
* Событийно-ориентированная архитектура.

#### **Преимущества:**

* Целостный взгляд на систему.
* Улучшенная масштабируемость.
* Управляемость сложности.
* Облегчение поддержки и развития.

### **Практические рекомендации**

1. Часто используется комбинация подходов.
2. Выбор метода зависит от:
   * Размера проекта.
   * Требований к системе.
   * Опыта команды.
   * Доступных ресурсов.
3. Важно учитывать:
   * Требования к производительности.
   * Необходимость повторного использования кода.
   * Сроки разработки.
   * Возможности команды.

### **6. Модульное программирование. Характеристики модуля (связность, сцепление, сложность)**

**Модульное программирование** — это метод разработки программ, при котором программа разбивается на отдельные, относительно независимые части (модули).

#### **1. Основные принципы модульного программирования:**

* Декомпозиция программы на модули.
* Независимость модулей друг от друга.
* Каждый модуль имеет единственную точку входа и выхода.
* Модули можно разрабатывать и тестировать независимо.

#### **2. Характеристики модуля:**

**А) Связность (Cohesion):**

* Связность определяет силу взаимосвязи элементов внутри модуля.
* **Типы связности (от лучшей к худшей):**
  + **Функциональная связность:**
    - Все элементы модуля работают для выполнения одной задачи.
    - Пример: модуль расчета зарплаты.
  + **Последовательная связность:**
    - Выход одной части является входом для другой.
    - Пример: считывание данных → обработка → запись.
  + **Коммуникационная связность:**
    - Части модуля работают с одними данными.
    - Пример: модуль работы с файлом (чтение/запись/удаление).
  + **Процедурная связность:**
    - Части выполняются в определенном порядке.
    - Пример: инициализация → обработка → завершение.
  + **Временная связность:**
    - Части связаны по времени выполнения.
    - Пример: модуль инициализации программы.
  + **Логическая связность:**
    - Части выполняют похожие функции.
    - Пример: модуль утилит.
  + **Случайная связность:**
    - Части никак не связаны между собой.
    - Наихудший вариант.

**Б) Сцепление (Coupling):**

* Сцепление определяет степень зависимости между модулями.
* **Типы сцепления (от лучшего к худшему):**
  + **Информационное сцепление:**
    - Модули связаны только через параметры.
    - Наилучший вариант.
  + **Сцепление по данным:**
    - Модули используют общие данные.
  + **Сцепление по управлению:**
    - Один модуль управляет работой другого.
  + **Сцепление по внешним ссылкам:**
    - Модули используют общие глобальные данные.
  + **Сцепление по общей области:**
    - Модули имеют доступ к одной области памяти.
  + **Содержательное сцепление:**
    - Один модуль напрямую обращается к внутренней работе другого.

**В) Сложность модуля:**

1. **Количественные метрики:**
   * Размер модуля (строки кода).
   * Цикломатическая сложность.
   * Количество параметров.
   * Глубина вложенности.
2. **Качественные характеристики:**
   * Понятность кода.
   * Тестируемость.
   * Сопровождаемость.
   * Возможность повторного использования.
3. **Рекомендации по сложности:**
   * Оптимальный размер 40–100 строк.
   * Не более 7 параметров.
   * Цикломатическая сложность не более 10.
   * Глубина вложенности не более 4.

#### **Правила хорошего модульного программирования:**

1. Высокая связность внутри модуля.
2. Слабое сцепление между модулями.
3. Единственная ответственность модуля.
4. Скрытие информации.
5. Четкие интерфейсы.
6. Независимое тестирование.

**7.Разработка требований. Определение. Классификация требований. Свойства требований. Документирование требований: • Концепция (Vision) • Пользовательские истории (User Stories) • Сценарии использования (Use Cases)**

**IEEE Standard Glossary of Software Engineering Terminology (1990)**

Требование - это:

1. условия или возможности, необходимые пользователю для решения проблем или достижения целей;

2. условия или возможности, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворять стандартам, спецификациям или другим формальным документам;

3. документированное представление условий или возможностей для пунктов 1 и 2.

**ISO/IEC/IEEE 29148:2011 Systems and software engineering — Life cycle processes — Requirements engineering**

Требование – это утверждение, которое переводит или выражает потребность и связанные с ней ограничения и условия

**Классификация требований**

Требования к ПО состоят из трех уровней —

бизнес-требования,

пользовательские,

функциональные требования



**по Карлу Вигерсу:**

1. Бизнес-требования (Business Requirements)

- Описывают цели и задачи организации/заказчика

- Определяют, почему создается система

- Документируются в:

\* Концепции (Vision document)

\* Технико-экономическом обосновании

- Примеры:

\* "Увеличить долю рынка на 15%"

\* "Сократить время обработки заказа на 30%"

\* "Выйти на новый рынок"

2. Пользовательские требования (User Requirements)

- Описывают задачи, которые пользователи должны иметь возможность решать с помощью системы

- Документируются в:

\* Сценариях использования (Use Cases)

\* Пользовательских историях (User Stories)

\* Сценариях взаимодействия

- Примеры:

\* "Пользователь должен иметь возможность создать новый заказ"

\* "Менеджер должен иметь возможность просматривать отчеты о продажах"

3. Функциональные требования (Functional Requirements)

- Описывают поведение системы

- Определяют что должна делать система

- Документируются в:

\* Спецификации требований к ПО

\* Функциональных спецификациях

- Примеры:

\* "Система должна отправлять email-уведомление при регистрации"

\* "Система должна автоматически рассчитывать итоговую сумму заказа"

4. Нефункциональные требования (Non-functional Requirements)

Подразделяются на несколько категорий:

А) Требования к производительности

- Время отклика

- Пропускная способность

- Объем обрабатываемых данных

- Примеры:

\* "Время отклика системы не должно превышать 2 секунды"

\* "Система должна поддерживать работу 1000 одновременных пользователей"

Б) Требования к безопасности

- Аутентификация

- Авторизация

- Защита данных

- Примеры:

\* "Пароли должны храниться в зашифрованном виде"

\* "Доступ к админ-панели только по двухфакторной аутентификации"

В) Требования к надежности

- Отказоустойчивость

- Восстановление после сбоев

- Доступность

- Примеры:

\* "Система должна быть доступна 99.9% времени"

\* "Автоматическое резервное копирование каждые 24 часа"

Г) Требования к удобству использования

- Интуитивность интерфейса

- Обучаемость

- Доступность для разных групп пользователей

- Примеры:

\* "Все основные функции должны быть доступны в 3 клика"

\* "Наличие справочной системы на русском языке"

5. Системные требования (System Requirements)

- Технические детали реализации

- Требования к интеграции

- Примеры:

\* "Система должна работать под управлением Windows Server 2019"

\* "Интеграция с платежной системой через API"

6. Ограничения (Constraints)

- Законодательные ограничения

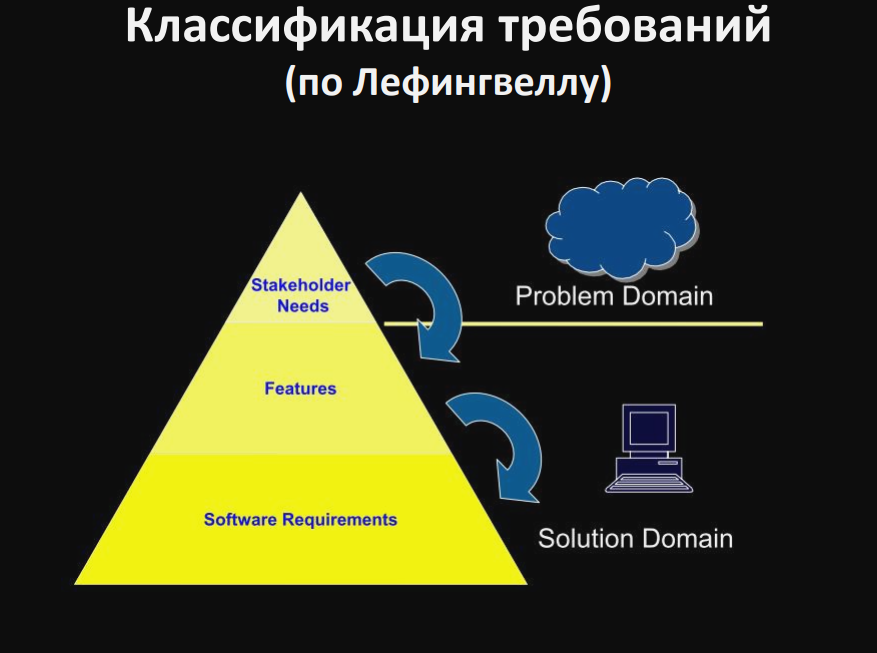
- Технологические ограничения

- Бюджетные ограничения

- Примеры:

\* "Использование только open-source компонентов"

\* "Соответствие GDPR"



**по Лефингвеллу:**

4. Требования к продукту (Product Requirements)

А) Функциональные требования

- Описывают поведение системы

- Определяют функции системы

- Примеры:

\* "Система должна отправлять уведомления пользователям"

\* "Система должна формировать ежемесячные отчеты"

\* "Возможность фильтрации данных по различным параметрам"

Б) Нефункциональные требования

- Атрибуты качества системы

- Подразделяются на:

1. Требования к удобству использования

\* Интуитивность интерфейса

\* Скорость обучения

\* Доступность для разных групп пользователей

\* Примеры:

- "Время обучения не более 2 часов"

- "Поддержка различных языков интерфейса"

2. Требования к производительности

\* Время отклика

\* Пропускная способность

\* Использование ресурсов

\* Примеры:

- "Загрузка страницы не более 3 секунд"

- "Обработка 1000 транзакций в секунду"

3. Требования к безопасности

\* Аутентификация

\* Авторизация

\* Шифрование

\* Примеры:

- "Блокировка учетной записи после 3 неудачных попыток"

- "Шифрование всех передаваемых данных"

4. Требования к надежности

\* Отказоустойчивость

\* Восстановление

\* Доступность

\* Примеры:

- "Время восстановления после сбоя не более 1 часа"

- "Доступность системы 99.9%"

5. Требования к реализации (Implementation Requirements)

- Технические ограничения

- Используемые технологии

- Примеры:

\* "Использование Java 11 и выше"

\* "Развертывание в облачной инфраструктуре"

6. Требования к интерфейсам (Interface Requirements)

- Взаимодействие с другими системами

- Форматы обмена данными

- Примеры:

\* "Интеграция с 1С через веб-сервисы"

\* "Поддержка формата JSON для API"

Особенности подхода Лефингвелла:

1. Иерархическая структура требований

2. Четкое разделение бизнес-правил и требований

3. Акцент на внешние факторы

4. Детальная классификация нефункциональных требований

Преимущества классификации:

- Помогает не упустить важные аспекты

- Облегчает управление требованиями

- Улучшает коммуникацию между заинтересованными сторонами

- Упрощает процесс документирования

**Свойства требований**

1. Полнота

- Требование должно полностью описывать необходимую функциональность

- Не должно быть недосказанности и пропущенных условий

- Пример хорошего требования: "Система должна отправлять email-уведомление пользователю при регистрации, содержащее логин, время регистрации и ссылку для подтверждения"

2. Ясность

- Однозначная интерпретация требования

- Отсутствие двусмысленностей

- Использование точных формулировок

- Пример: вместо "система должна работать быстро" → "время отклика системы не должно превышать 2 секунды"

3. Согласованность

- Отсутствие противоречий между требованиями

- Использование единой терминологии

- Соответствие общей концепции системы

- Пример противоречия: "система должна хранить пароли в открытом виде" и "система должна обеспечивать безопасное хранение данных"

4. Верифицируемость

- Возможность проверить выполнение требования

- Наличие измеримых критериев

- Пример верифицируемого требования: "система должна поддерживать одновременную работу 100 пользователей"

5. Необходимость

- Требование должно отражать реальную потребность

- Должно быть обосновано бизнес-целями

- Отсутствие избыточных требований

- Пример необходимого требования: "система должна формировать ежемесячный отчет о продажах" (если это нужно для бизнес-процессов)

6. Осуществимость

- Возможность реализации с учетом:

\* Технических ограничений

\* Временных рамок

\* Бюджета

\* Имеющихся ресурсов

- Пример неосуществимого требования: "система должна предсказывать будущие действия пользователя со 100% точностью"

7. Трассируемость

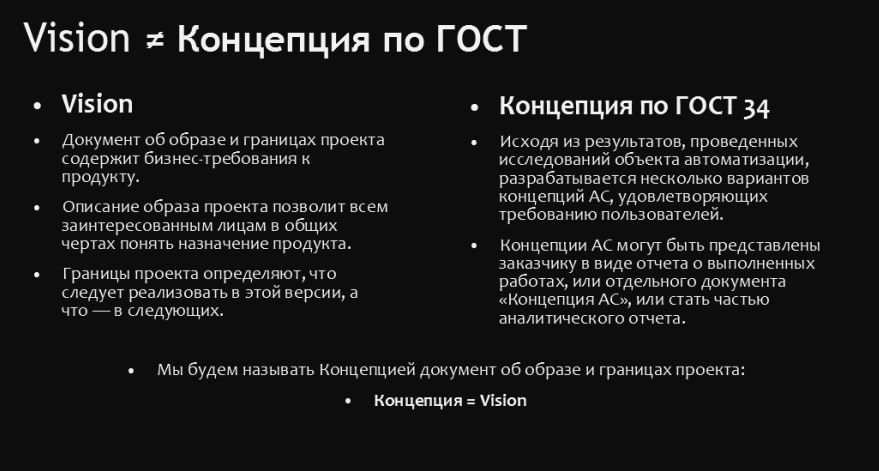
- Возможность проследить:

\* Источник требования

\* Связи с другими требованиями

\* Реализацию в системе

- Пример: требование должно иметь уникальный идентификатор и ссылки на связанные требования

**Документирование требований**

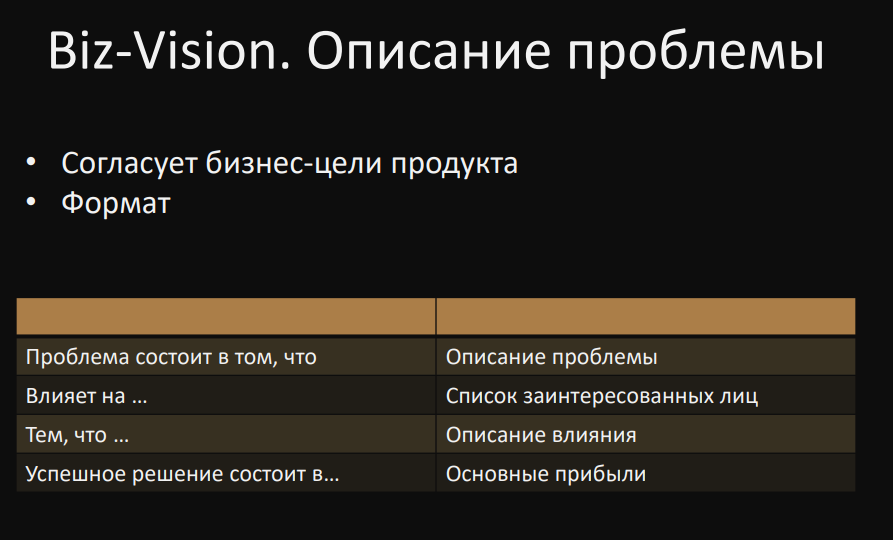
Концепция - Vision • Дает ответ на вопрос – Какую проблему решаем? – Каким способом? – Почему это правильный способ? • Согласует взгляды заинтересованных лиц и разработчиков • Формулируется до начала разработки

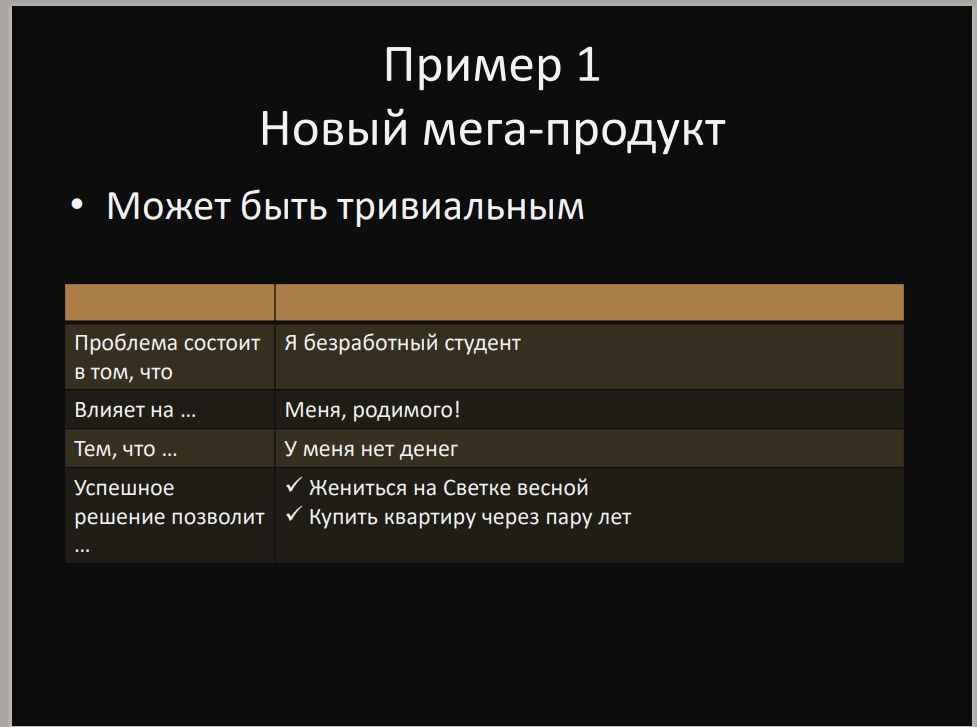
**Составляющие Vision**

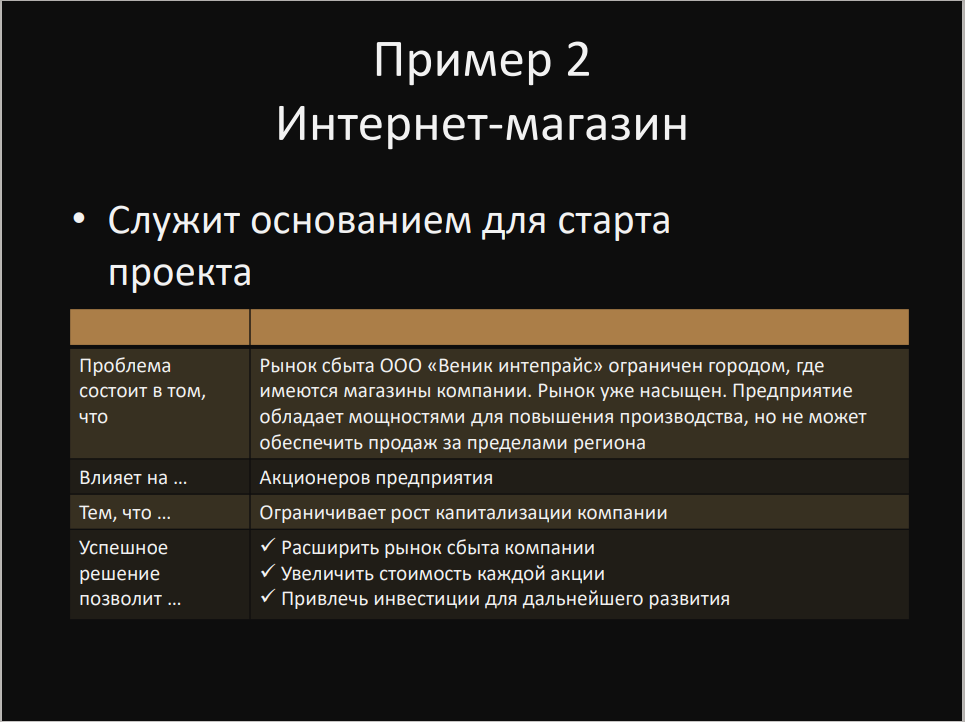
• Цель/проблема

• Заинтересованные лица

• Позиционирование продукта









**Пользовательские истории (User Stories)** представляют собой краткое описание функциональности системы с точки зрения пользователя, которое включает роль пользователя, желаемое действие и ожидаемую выгоду. Пользовательские истории записываются в формате "Как <роль> я хочу <действие> чтобы <ценность>" и сопровождаются критериями приемки, определяющими условия, при которых история считается реализованной. Они служат основой для обсуждения требований между заказчиком и командой разработки, позволяя сфокусироваться на потребностях пользователя и бизнес-ценности функционала. Пользовательские истории являются ключевым элементом гибких методологий разработки, обеспечивая баланс между достаточным уровнем детализации и гибкостью в реализации.

**Сценарии использования (Use Cases)** являются более формальным и детальным способом описания взаимодействия пользователя с системой, документирующим последовательность действий, которые система должна выполнить в ответ на действия пользователя. Сценарий использования включает название, действующих лиц, предусловия, основной поток событий, альтернативные потоки, постусловия и исключительные ситуации. Каждый сценарий описывает конкретную функциональность системы, учитывая различные варианты развития событий и возможные ошибки. Сценарии использования обеспечивают полноту и точность в описании требований, что особенно важно для сложных систем и критически важных бизнес-процессов. Они позволяют документировать не только успешные сценарии, но и обработку исключительных ситуаций, что делает их незаменимыми при разработке систем, требующих высокой надежности и предсказуемости поведения.

**8. Визуальное моделирование и проектирование с использованием UML ООАП. Анализ: разработка прецедентов. ООАП. Анализ: построение концептуальной модели ИС. ООАП. Анализ и проектирование: поведение системы (диаграммы последовательностей). ООАП. Анализ и проектирование: диаграммы кооперации.**

**Визуальное моделирование и проектирование с использованием UML**

UML (Unified Modeling Language) представляет собой унифицированный язык моделирования, предназначенный для визуализации, документирования и проектирования программных систем. UML включает набор стандартизированных графических нотаций для создания абстрактной модели системы. Основными диаграммами являются диаграммы структуры (классов, компонентов, объектов) и диаграммы поведения (последовательности, активности, состояний).

**ООАП. Анализ: разработка прецедентов**

Разработка прецедентов (Use Case) является ключевым этапом анализа системы, где описываются взаимодействия между системой и внешними акторами. Прецедент определяет последовательность действий, которые система выполняет в ответ на запрос пользователя или другого внешнего актора. Каждый прецедент включает основной поток событий, альтернативные потоки и исключительные ситуации. Диаграмма прецедентов отображает функциональные требования к системе в понятной для заказчика форме.

**ООАП. Анализ: построение концептуальной модели ИС**

Концептуальная модель информационной системы представляет собой абстрактное описание предметной области, включающее основные понятия, их атрибуты и взаимосвязи. При построении концептуальной модели используются диаграммы классов UML, где классы представляют сущности предметной области, а связи между ними отражают существующие отношения. Модель фокусируется на "что" система должна делать, а не "как" она это делает, абстрагируясь от деталей реализации.

**ООАП. Анализ и проектирование: поведение системы (диаграммы последовательностей)**

Диаграммы последовательностей отображают взаимодействие объектов системы во времени. Они показывают временную последовательность сообщений между объектами, участвующими в конкретном взаимодействии. Вертикальная ось представляет время, а горизонтальная - различные объекты. Диаграммы последовательностей особенно полезны для моделирования сценариев использования системы и детализации алгоритмов взаимодействия объектов.

**ООАП. Анализ и проектирование: диаграммы кооперации**

Диаграммы кооперации (или коммуникации в UML 2.0) показывают взаимодействие между объектами, фокусируясь на структурной организации объектов, которые отправляют и получают сообщения. В отличие от диаграмм последовательности, они подчеркивают пространственное расположение объектов и их связи, а не временную последовательность сообщений. Сообщения нумеруются для указания порядка их следования. Диаграммы кооперации помогают понять, как объекты работают вместе для реализации определенного поведения системы.

Все эти аспекты ООАП тесно взаимосвязаны и представляют разные взгляды на одну и ту же систему. Диаграммы прецедентов определяют функциональные требования, концептуальная модель описывает структуру предметной области, а диаграммы последовательности и кооперации показывают, как объекты взаимодействуют для реализации требуемого поведения. Вместе они обеспечивают полное представление о системе на разных уровнях абстракции и с разных точек зрения.

**9.Проектирование пользовательского интерфейса. Свойства эффективного интерфейса. Количественные характеристики качества интерфейса. Выбор структуры диалога**

**Свойства эффективного интерфейса**

Эстетическая привлекательность

Естественность интерфейса

Дружественность интерфейса

Принцип «обратной связи»

Согласованность интерфейса

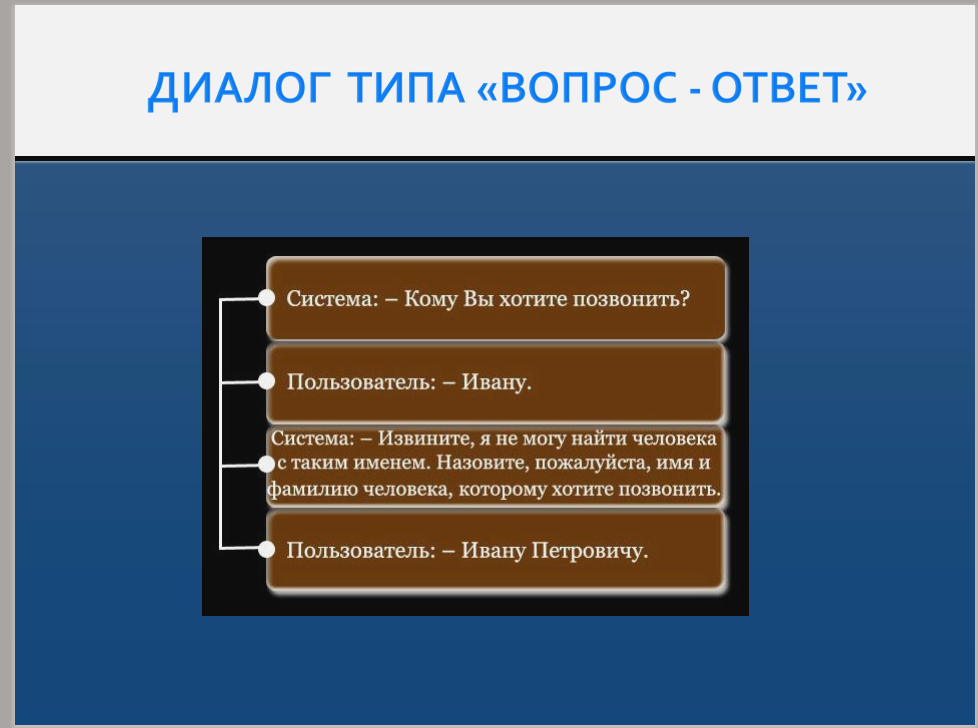
Гибкость интерфейса

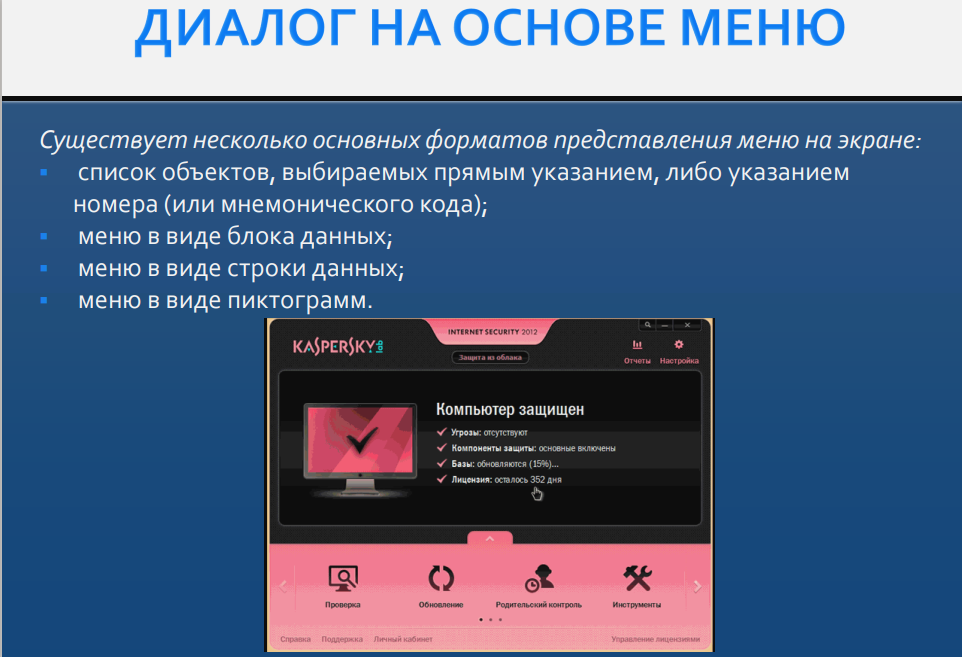
Простота интерфейса

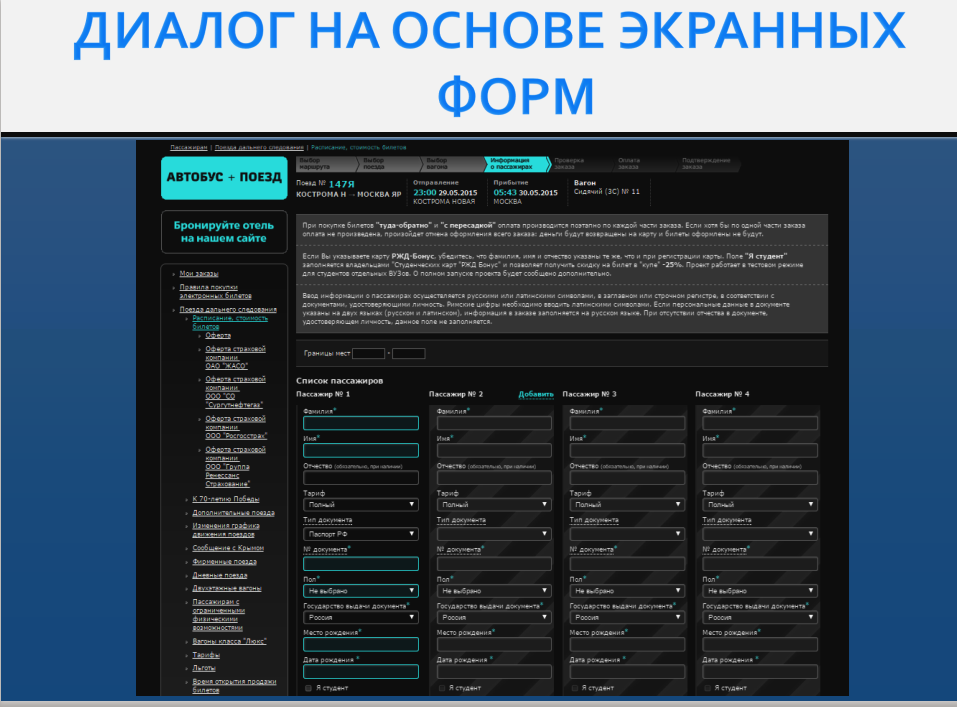
**Количественные характеристики качества интерфейса.**

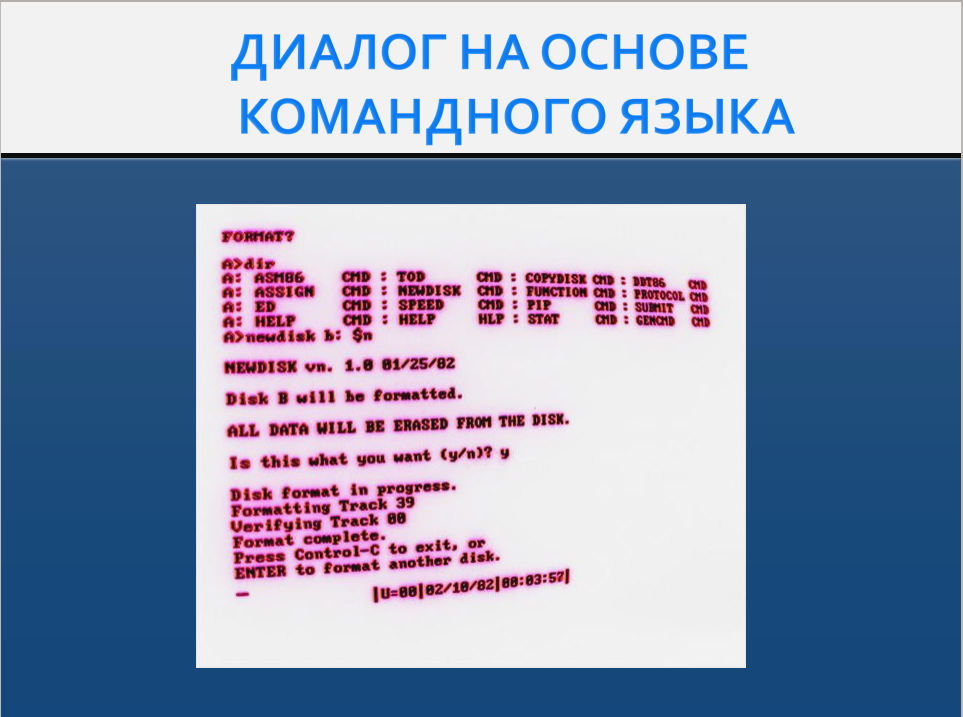
1. время, необходимое определенному пользователю, для достижения заданного уровня знаний навыков в работе с приложением. 2. сохранение полученных навыков по истечению времени 3. скорость решения задачи с помощью заданного приложения 4. субъективная удовлетворенность пользователя при работе системы

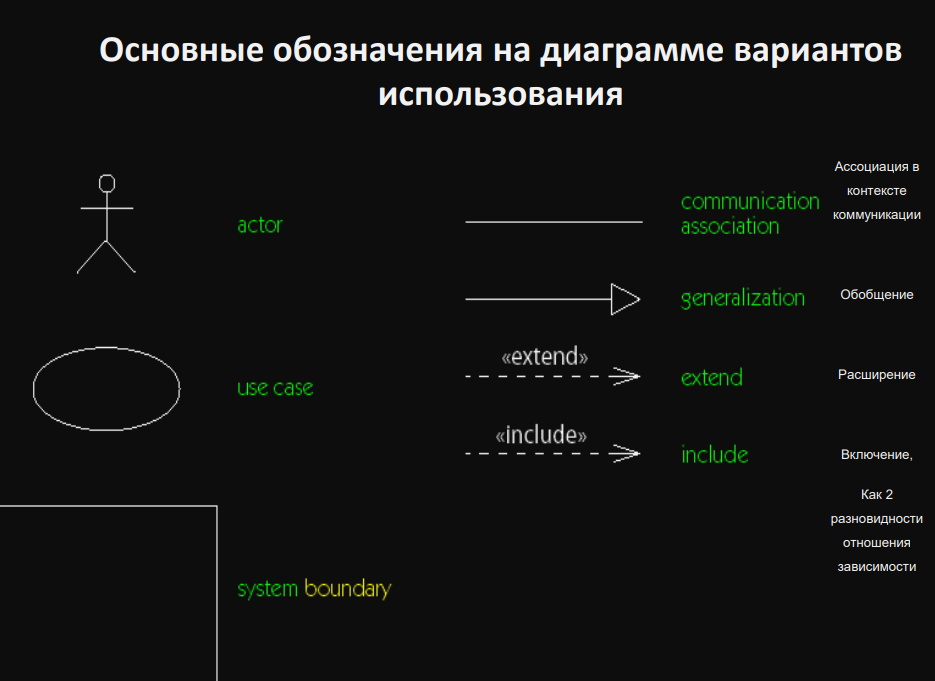
**Выбор диалога**

****

****

****

****

****