1. **Определение понятия «ФУНКЦИОНАЛЬНАЯ НАДЕЖНОСТЬ». Предмет функциональной надежности.**

*Надежность (Reability)* - является метрической характеристикой степени уверенности в том, что программа реагирует на поставленную задачу.

**Характеристики надежности сложных систем:**

Характеристиками надежности данных являются доступность, целостность, аутентификация, конфиденциальность и контроля участников информационного взаимодействия.

*Конфиденциальность* ассоциируется со способностью системы предотвратить получение неавторизованными пользователям конфиденциальной информации.

*Целостность* ассоциируется со способностью системы предотвращать несанкционированное изменение данных, состояние системы и информации, обеспечивая тем самым точность и достоверность информации и данных.

*Доступность* означает предотвращение утраты доступа к ресурсам информации, т.е. обеспечение предоставления авторизованным пользователям требуемой информации в случае необходимости.

**Предмет надежности функциональной надежности -** исследования безошибочности выполнения информационных технологий на предусмотренной информационной технике с учетом реализуемых в этих технологиях алгоритмов,требования к качеству управления и значимости ошибок.

**Метрики надежности.**

Метриками надежности являются: использование полосы пропускания, диагностические запросы, потеря или задержка пакетов, целостность пакетов.

Использование полосы пропускания характеризует полноту использования потенциальных транспортных возможностей сети.  
Мониторинг значения этого показателя, оценка тенденций его изменения, фиксация значимости отклонений от среднего значения создают основу для диагностирования факторов отказов, либо атак на систему.  
Диагностические запросы, ориентированы на изменение точности и временной ответа, и основанные на сопоставлении фактического ответа на запрос с ожидаемым. При анализе времени ответа следует учитывать случайную природу показателя, т.е. нужно ориентироваться на вероятностные оценки.  
Потери или задержки пакетов являются характеристикой надежности сервисов. Примерами причин, приводящим к потере пакетов и их задержкам, являются: недостаточная ширина полосы пропускания, проблемы установления соединений в сети, отказы оборудования, проблемы маршрутизации, конфигурация маршрутизаторов т.д.   
Целостность пакетов, определяется на основе значений контрольных сумм, являются основными характеристиками целостности данных.

1. **Концептуальная основа тестирования (50-е, 60-е годы).**

**(**книга по тестированию**) В 50–60-х годах** прошлого века процесс тестирования был предельно формализован, отделён от процесса непосредственной разработки ПО и «математизирован». Фактически тестирование представляло собой скорее отладку программ (debugging). Существовала концепция т. н. «исчерпывающего тестирования (exhaustive testing)» — проверки всех возможных путей выполнения кода со всеми возможными входными данными. Однако очень скоро было выяснено, что исчерпывающее тестирование невозможно, т. к. количество возможных путей и входных данных очень велико, а также при таком подходе сложно найти проблемы в документации.

**Концептуальная основа тестирования (50-е – 60-е годы)**

В 1950-х и 1960-х годах сформировались фундаментальные концепции тестирования, которые заложили основу для дальнейшего развития этой области. Этот период характеризуется становлением первых подходов, методов и практик тестирования, преимущественно в сфере программирования и вычислительных систем.

*1. Контекст и причины развития*

* **Рост сложности вычислительных систем**: В 1950-е годы вычислительные машины становились всё сложнее, а их программное обеспечение – объемнее.
* **Потребность в надёжности**: Программное обеспечение должно было обеспечивать безотказную работу систем в критически важных областях (например, военная техника, космические проекты).
* **Разработка первых языков программирования**: Такие языки, как Fortran (1957), COBOL (1959), стимулировали разработку методов проверки корректности программ.

*2. Ключевые концепции и подходы*

*2.1. Верификация и валидация*

* Эти концепции стали основополагающими в тестировании:
  + **Верификация**: проверка соответствия продукта его спецификациям (делаем ли мы продукт правильно?).
  + **Валидация**: проверка соответствия продукта ожиданиям пользователя (делаем ли мы правильный продукт?).

*2.2. Функциональное и структурное тестирование*

* **Функциональное тестирование**: проверка программы на соответствие требованиям (чёрный ящик).
* **Структурное тестирование**: анализ внутренней структуры программы (белый ящик).

***2.3. Теория тестирования***

* Первые теоретические исследования были сосредоточены на поиске всех возможных ошибок.
* **Работы Дж. Гуденафа и Г. Герста (Goodenough and Gerhart)**: Предложили основы для формализации тестирования, включая концепции покрытия (coverage).

*2.4. Автоматизация тестирования*

* Первыми инструментами автоматизации стали программы, которые помогали создавать тестовые данные и проводить тесты.
* **Примеры**: ранние системы автоматизации тестирования на мейнфреймах IBM.

*2.5. Принципы тестирования*

* **Тестирование не доказывает отсутствие ошибок**: Признание того, что тестирование может лишь выявить ошибки, но не доказать их полное отсутствие.
* **Ранняя фокусировка на тестировании**: Понимание, что тестирование должно начинаться уже на этапе проектирования.

*3. Ключевые достижения*

*3.1. Тестовые методологии*

* Разработка базовых подходов к тестированию (например, разбиение домена входных данных).
* Формализация понятий входных данных, ожидаемых результатов и критериев успешности теста.

*3.2. Появление стандартов*

* Начало разработки стандартов и руководств по тестированию. Например, появление первых практик тестирования ПО в военных и космических проектах (NASA).

*3.3. Исследование ошибок*

* Осознание значимости поиска ошибок на ранних этапах. Внимание к выявлению критических ошибок в сложных системах.

*4. Влияние на дальнейшее развитие*

* В 1970-х годах эти идеи стали основой для более системных подходов, включая:
  + Разработку комплексных методологий (например, Waterfall).
  + Акцент на управление качеством программного обеспечения.
  + Эволюцию автоматизированного тестирования и специализированных инструментов.

1. **Концептуальная основа тестирования (70-е годы). Философия «Белого» и «Черного» ящиков.**

(Из какой-то книги) **В 70-х годах** фактически родились две фундаментальные идеи тестирования: тестирование сначала рассматривалось как процесс доказательства работоспособности программы в некоторых заданных условиях (positive testing), а затем — строго наоборот: как процесс доказательства неработоспособности программы в некоторых заданных условиях (negative testing). Это внутреннее противоречие не только не исчезло со временем, но и в наши дни многими авторами совершенно справедливо отмечается как две взаимодополняющие цели тестирования.

Отметим, что «*процесс доказательства неработоспособности программы*» ценится чуть больше, т. к. не позволяет закрывать глаза на обнаруженные проблемы.

Итак, ещё раз самое важное, что тестирование «приобрело» в 70-е годы:

-тестирование позволяет удостовериться, что программа соответствует требованиям;

- тестирование позволяет определить условия, при которых программа ведёт себя некорректно.

(ГПТ ниже)

**Концептуальная основа тестирования (70-е годы): философия «Белого» и «Черного» ящиков**

1970-е годы стали периодом активного развития методологий тестирования, систематизации процессов, а также формализации философий тестирования, таких как **тестирование «чёрного ящика»** и **тестирование «белого ящика»**. Эти подходы начали широко использоваться в различных индустриях для обеспечения надёжности и качества программного обеспечения.

*1. Контекст 70-х годов*

* **Рост сложности ПО**: Программы становились более сложными, с увеличением объёма кода и числа зависимостей.
* **Системный подход**: Появление методологий разработки (например, Waterfall, предложенная У. Ройсом) сделало тестирование обязательным этапом.
* **Становление управления качеством**: Начало применения концепций Total Quality Management (TQM) в программной индустрии.
* **Фокус на экономию ресурсов**: Необходимость сокращать затраты на исправление ошибок за счёт раннего выявления дефектов.

**2. Философия «Чёрного ящика»**

***Определение:***

**Тестирование «чёрного ящика» (Black Box Testing)** — это подход, при котором тестировщик проверяет функциональность программного обеспечения, не зная его внутренней структуры или реализации.

*Основные характеристики:*

1. **Фокус на функциональности**: Проверяется соответствие программы заданным требованиям.
2. **Не требуется знание кода**: Тестировщик работает только с интерфейсами и спецификациями.
3. **Методы тестирования**:
   * Разбиение входных данных на классы эквивалентности.
   * Анализ граничных значений.
   * Таблицы решений и сценарии использования (use case testing).

*Преимущества***:**

* Простота в применении, так как тестировщику не нужно знать код.
* Независимость от разработчиков.
* Возможность выявления несоответствий спецификациям.

*Ограничения***:**

* Не позволяет оценить покрытие кода (можно пропустить внутренние дефекты).
* Эффективность зависит от качества спецификаций.

**3. Философия «Белого ящика»**

*Определение***:**

**Тестирование «белого ящика» (White Box Testing)** — это подход, при котором тестировщик проверяет внутреннюю структуру программы, используя знание кода.

*Основные характеристики:*

1. **Фокус на коде**: Тестировщик анализирует логику программы, внутренние структуры и потоки данных.
2. **Методы тестирования**:
   * Тестирование покрытия операторов (statement coverage).
   * Тестирование покрытия ветвей (branch coverage).
   * Тестирование путей (path testing).
   * Анализ циклов и условий.

*Преимущества***:**

* Помогает выявить скрытые дефекты в логике программы.
* Позволяет оценить степень покрытия кода.
* Обеспечивает глубокое понимание работы системы.

*Ограничения***:**

* Зависимость от знаний кода: требует квалифицированного тестировщика.
* Возможен упор на детали реализации вместо проверки бизнес-логики.

*4. Сравнение философий «Белого» и «Чёрного ящиков»*

| **Аспект** | **Чёрный ящик** | **Белый ящик** |
| --- | --- | --- |
| **Цель** | Проверка функциональности | Проверка структуры и логики |
| **Знание кода** | Не требуется | Обязательно |
| **Основной фокус** | Требования и спецификации | Логика реализации и код |
| **Примеры методов** | Разбиение на классы эквивалентности | Покрытие операторов, путей |
| **Преимущества** | Простота, независимость | Глубокий анализ внутренней логики |
| **Недостатки** | Пропуск внутренних дефектов | Зависимость от знаний кода |

*5. Развитие концепций в 70-е годы*

* **Формализация методов**:  
  В 70-х годах появились формальные методы и алгоритмы для тестирования, такие как критерии покрытия (Coverage Criteria) и систематические подходы к выбору тестовых данных.
* **Автоматизация тестирования**:  
  Начали разрабатываться первые инструменты для статического анализа кода и автоматического выполнения тестов.
* **Интеграция философий**:  
  В 70-е годы стало очевидным, что для достижения высокого качества необходимо использовать обе философии:
  + **«Чёрный ящик»** для проверки требований и внешнего поведения.
  + **«Белый ящик»** для анализа внутренней структуры.

*6. Влияние на современное тестирование*

* Концепции «чёрного» и «белого ящиков» легли в основу гибридного подхода — **«серого ящика»** (Gray Box Testing), который сочетает знания о внутренней структуре системы с функциональным анализом.
* Эти философии до сих пор являются основными подходами к тестированию в современных методологиях, таких как Agile и DevOps.

1. **. Концептуальная основа тестирования (70-е годы). Стратегии тестирования интеграции. Ограничения стратегий**

В 1970-е годы тестирование программного обеспечения стало более структурированным процессом. Одним из ключевых направлений стало развитие **стратегий тестирования интеграции** — этапа, на котором объединённые модули или компоненты системы тестируются на предмет корректного взаимодействия.

### 1. ****Контекст эпохи****

1. **Рост сложности программных систем**:
   * Разработка программного обеспечения усложнялась, увеличивалось количество взаимодействующих компонентов.
   * Интеграция модулей стала критическим этапом, так как ошибки на этом уровне оказывали значительное влияние на качество всей системы.
2. **Появление формальных методологий**:
   * Начали применяться структурированные подходы к проектированию и тестированию, такие как **структурное программирование** и **модульное проектирование**.

### 2. ****Основные стратегии тестирования интеграции****

1. **Нисходящее тестирование (Top-down testing)**
   * Тестирование начинается с верхнего уровня архитектуры и постепенно спускается к нижним модулям.
   * **Принципы**:
     + Сначала тестируется главный модуль.
     + Вспомогательные модули заменяются заглушками (stubs), которые имитируют их поведение.
   * **Преимущества**:
     + Обеспечивает раннюю проверку архитектуры и основных функций.
     + Локализует ошибки на верхнем уровне.
   * **Ограничения**:
     + Трудности с тестированием функциональности низкоуровневых модулей до их полной интеграции.
     + Требует значительного времени на разработку заглушек.
2. **Восходящее тестирование (Bottom-up testing)**
   * Тестирование начинается с самых низких модулей и постепенно поднимается вверх.
   * **Принципы**:
     + Сначала тестируются базовые модули.
     + Вспомогательные модули заменяются драйверами, которые передают данные для тестирования.
   * **Преимущества**:
     + Выявление ошибок в базовых модулях на раннем этапе.
     + Полная функциональность нижних модулей доступна к моменту тестирования верхних уровней.
   * **Ограничения**:
     + Функциональность системы в целом тестируется только на поздних этапах.
     + Высокие затраты на разработку драйверов.
3. **Сэндвич-тестирование (Sandwich testing)**
   * Комбинирует нисходящий и восходящий подходы.
   * **Принципы**:
     + Тестирование проводится параллельно на верхних и нижних уровнях.
     + Средний уровень системы становится точкой интеграции.
   * **Преимущества**:
     + Сбалансированный подход, позволяющий протестировать как верхние, так и нижние уровни одновременно.
   * **Ограничения**:
     + Сложность координации параллельного тестирования.
     + Трудности в локализации ошибок на среднем уровне.
4. **Инкрементное тестирование (Incremental testing)**
   * Модули добавляются и тестируются поэтапно, один за другим.
   * **Принципы**:
     + Последовательное тестирование всех компонентов.
     + Каждый новый модуль тестируется с уже интегрированными.
   * **Преимущества**:
     + Позволяет быстро локализовать ошибки.
     + Упрощает процесс диагностики.
   * **Ограничения**:
     + Требует значительных временных ресурсов.
     + Может быть неэффективным при большом количестве модулей.
5. **Большой взрыв (Big Bang testing)**
   * Все модули интегрируются одновременно и тестируются как единое целое.
   * **Преимущества**:
     + Минимальные затраты на подготовку тестов.
   * **Ограничения**:
     + Трудности в локализации ошибок.
     + Высокий риск пропустить дефекты взаимодействия модулей.
     + Большая вероятность полного провала тестирования.

### 3. ****Ограничения стратегий тестирования интеграции****

1. **Сложность управления процессом**:
   * При больших и сложных системах координация тестирования множества модулей требует значительных усилий.
2. **Зависимость от качества проектирования**:
   * Ошибки, допущенные на этапе проектирования системы, могут затруднить процесс интеграции и выявления ошибок.
3. **Необходимость заглушек и драйверов**:
   * Разработка вспомогательных компонентов увеличивает затраты времени и ресурсов.
4. **Риск пропуска дефектов на уровне взаимодействия**:
   * Неполное покрытие тестами может привести к обнаружению ошибок только на этапе эксплуатации.
5. **Сложность локализации дефектов**:
   * При одновременной интеграции большого числа модулей (например, в стратегии "Большой взрыв") становится сложно определить источник ошибки.
6. **Роль дисциплины при проектировании программных систем**

**Роль дисциплины при проектировании программных систем**

Проектирование программных систем — это многогранный и сложный процесс, который требует чёткого и системного подхода. Дисциплина в проектировании программных систем подразумевает использование строго определённых методологий, стандартов и практик, направленных на создание надёжных и качественных программных продуктов. Без дисциплины разработка может превратиться в хаос, что, в свою очередь, снижает качество конечного продукта и увеличивает затраты на его создание и поддержку.

**1. Значение дисциплины в проектировании**

Дисциплина важна для следующих аспектов проектирования:

1. **Управление сложностью**:  
   Современные программные системы включают большое количество компонентов и их взаимозависимостей. Без чёткой дисциплины проектирования сложно будет структурировать систему, что может привести к увеличению числа ошибок и затруднениям в её понимании и обслуживании.
2. **Системность подхода**:  
   Процесс проектирования требует учёта множества факторов — от функциональных и нефункциональных требований до архитектурных и технических ограничений. Дисциплина помогает сбалансировать все эти аспекты, обеспечивая целостность и последовательность решения задач.
3. **Снижение числа ошибок**:  
   Строгое следование методологиям, таким как UML, Agile или Waterfall, позволяет на ранних этапах проектирования выявлять и устранять возможные ошибки и риски, тем самым повышая качество конечного продукта.
4. **Обеспечение повторяемости**:  
   Применение проверенных методик и стандартов в проектировании позволяет использовать успешный опыт в разных проектах, повышая производительность и снижая риск ошибок.

**2. Основные аспекты дисциплины в проектировании**

**2.1. Использование методологий разработки**

* **Структурный подход**: Это разбиение системы на модули с чётким определением их функций и взаимодействий.
* **Объектно-ориентированный подход**: Применение принципов ООП (инкапсуляция, наследование, полиморфизм) позволяет эффективно управлять сложностью и обеспечивать повторное использование кода.
* **Адаптивные методологии**: Внедрение гибких подходов, таких как Agile, позволяет быстрее адаптироваться к изменениям требований в процессе разработки.

**2.2. Стандарты и процессы**

* Применение стандартов, таких как ISO/IEC 12207, помогает обеспечить единообразие и совместимость на всех этапах жизненного цикла программного обеспечения.
* Использование архитектурных паттернов (например, MVC или микросервисы) повышает качество проектирования и упрощает последующую поддержку системы.

**2.3. Инструменты и документация**

* **Моделирование**: Использование UML и других инструментов моделирования позволяет визуализировать архитектуру системы и её поведение.
* **Документация**: Создание полной и чёткой проектной документации обеспечивает передачу знаний и улучшает взаимодействие внутри команды.

**2.4. Управление качеством**

* Практики проверки и тестирования проектных решений, такие как код-ревью, анализ архитектурных рисков, помогают избежать ошибок и повысить надёжность системы.

**3. Последствия отсутствия дисциплины**

1. **Хаотичность разработки**:  
   Без дисциплины проектирование может стать непоследовательным, что усложняет поддержку системы и её дальнейшее развитие.
2. **Ошибки на ранних стадиях**:  
   Ошибки в проектировании, не выявленные на ранних этапах, часто становятся затратными и трудными для исправления в процессе разработки или эксплуатации.
3. **Низкое качество продукта**:  
   Недисциплинированный подход может привести к созданию систем с низкой надёжностью, производительностью и неудовлетворительными пользовательскими характеристиками.
4. **Зависимость от команды**:  
   Отсутствие дисциплины делает проектирование зависимым от личных навыков разработчиков, что затрудняет масштабирование и развитие проекта в будущем.

**4. Преимущества дисциплинированного подхода**

1. **Повышение надёжности**:  
   Следование чётким методологиям и стандартам помогает снизить вероятность ошибок и повысить стабильность работы системы.
2. **Эффективное использование ресурсов**:  
   Дисциплина в проектировании позволяет чётко распределить задачи, что способствует эффективному использованию времени и ресурсов.
3. **Простота сопровождения**:  
   Чёткое проектирование упрощает дальнейшую работу с системой, будь то исправление ошибок, добавление новых функций или её масштабирование.
4. **Совместная работа**:  
   Применение общих стандартов и документации улучшает взаимодействие между членами команды, обеспечивая единство и целостность работы.

**5. Роль дисциплины в современных подходах**

Современные методологии разработки, такие как Agile, DevOps и CI/CD, требуют строгой дисциплины для успешной реализации проектов. Важными аспектами являются:

* **Чёткое определение требований** на ранних стадиях проекта.
* **Тестирование на всех этапах разработки** для раннего выявления ошибок.
* **Автоматизация процессов** (например, сборка, тестирование, развертывание), что повышает скорость и надёжность разработки.