**1. IT-проект: основные понятия** - в этом контексте можно рассмотреть такие термины, как цель проекта, задачи, сроки, бюджет, команда проекта, методология управления проектом и т.д. IT-проект - это проект, связанный с разработкой, внедрением или поддержкой информационных технологий.

**2. Компьютерный червь: свойства, особенности** -компьютерный червь это вредоносная программа, способная самостоятельно распространяться через компьютерные сети, в основном, путем использования уязвимостей. Основные свойства и особенности компьютерных червей включают автономность, способность к самосохранению, способность к удаленному управлению и возможность нанесения вреда компьютерной системе, такой как уничтожение данных или рас

**3. Компьютерный троян: свойства, особенности.**

Компьютерный троян (Trojan) - это вид вредоносного программного обеспечения, которое маскируется под легитимное или полезное приложение или файл, чтобы получить доступ к компьютеру и нанести вред. В отличие от вирусов или червей, троянские программы не распространяются самостоятельно, они требуют участия пользователя для запуска.

Основные особенности и свойства компьютерных троянов:

1. Маскировка

2. Контроль над компьютером

3. Вредительство

4. Автоматическое распространение

**4. Компьютерный блокировщик экрана (WinLocker): свойства, особенности.**

Компьютерный блокировщик экрана, также известный как WinLocker, является видом вредоносного программного обеспечения, которое блокирует доступ к компьютеру и отображает на экране сообщение о требовании оплаты для разблокировки.

Основные свойства и особенности компьютерного блокировщика экрана:

1. Блокировка экрана

2. Вымогательство денег

3. Мошенничество

4. Распространение

5. Превращение в социальную инженерию

**5. Необходимые условия применения измерительных мониторов:**

- Наличие ясно определенной цели измерений

- Доступность и возможность установки мониторов

- Адекватность и точность измерений

- Надежность и стабильность мониторов

- Возможность калибровки и обслуживания

**6. Измерительные методы: назначение.**

Измерительные методы используются для определения значений различных параметров и характеристик. Они могут быть использованы в различных областях, таких как наука, промышленность, медицина и другие. Вот некоторые основные типы измерительных методов и их назначение:

- Физические методы

- Химические методы

- Биологические методы

- Оптические методы

- Электрические методы

Выбор и применение конкретного измерительного метода зависит от требуемой точности, доступности необходимого оборудования и параметров, которые необходимо измерить.

**7. Описание компонентов схемы измерений:**

- Измерительный прибор: это устройство, которое осуществляет измерение физической величины или параметра. Примеры включают в себя термометры, вольтметры, амперметры и другие специализированные приборы.

- Измерительные соединения: это соединения, которые необходимы для подключения измерительного прибора к объекту или системе, которую нужно измерить. Они могут включать в себя провода, кабели, разъемы и т. д.

- Источник измеряемого сигнала: это устройство, которое генерирует сигнал или импульс, который будет измеряться. Примеры включают в себя генераторы сигналов, источники тока или напряжения.

- Защитные средства: это компоненты, предназначенные для защиты измерительных приборов от повреждений или перегрузок. Они могут включать в себя предохранители, диоды или резисторы.

- Коммутационные компоненты: это компоненты, которые обеспечивают соединение или разъединение измерительного прибора с объектом или системой. Они могут включать в себя переключатели, реле или ручки.

- Датчики и преобразователи: это устройства, которые преобразуют физическую величину или параметр в электрический сигнал, который может быть измерен. Примеры включают в себя термисторы, датчики давления или датчики уровня.

**8. Аппаратные измерительные мониторы: виды, основные назначения.**

- Осциллографы: используются для измерения и отображения временных сигналов. Они позволяют наблюдать форму и амплитуду сигнала, а также измерять его частоту и фазу.

- Спектр анализаторы: используются для анализа спектра сигнала по частоте. Они позволяют определить амплитуду и фазу компонентов спектра, а также проводить спектральный анализ сигнала.

- Мультиметры: используются для измерения различных электрических величин, таких как напряжение, ток, сопротивление и емкость. Они представляют собой универсальные измерительные приборы.

- Логические анализаторы: используются для анализа цифровых сигналов в логическом уровне. Они позволяют наблюдать и анализировать последовательность сигналов, исследовать задержки и взаимодействия между сигналами.

- Частотомеры: используются для измерения частоты сигнала. Они могут быть аналоговыми или цифровыми и позволяют измерять частоту сигналов различного типа и диапазона.

- Анализаторы качества электроэнергии: используются для анализа качества электрической энергии в электрических сетях. Они позволяют измерять параметры, такие как напряжение, ток, гармонические искажения, обрывы и перекосы напряжения, а также проводить анализ электрической энергии.

**9. Измерительные мониторы** - это устройства или программное обеспечение, которые предназначены для измерения и мониторинга различных параметров или характеристик. Они используются во многих областях, включая науку, промышленность, здравоохранение и технику безопасности.

В зависимости от конкретного назначения, есть различные виды измерительных мониторов. Некоторые из них включают в себя мониторы окружающей среды, мониторы уровня шума, мониторы качества воздуха, мониторы сердечного ритма и мониторы потребления энергии. Они используются для измерения и контроля определенных параметров, чтобы обеспечить безопасность и эффективность процессов.

**10. Статическое тестирование** - это метод тестирования программного обеспечения, основанный на анализе и проверке кода без его фактического выполнения или запуска программы. Оно выполняется путем просмотра и анализа исходного кода, структуры программы, документации и других артефактов.

Основное значение статического тестирования заключается в выявлении потенциальных ошибок и дефектов до начала исполнения программы. Это помогает обнаруживать проблемы на ранних стадиях разработки и устранять их до того, как они приведут к серьезным проблемам в процессе эксплуатации.

Статическое тестирование позволяет контролировать качество кода, повышать его читаемость и понимаемость, а также облегчать сопровождение программы. Этот метод тестирования широко используется в различных областях разработки программного обеспечения, включая веб-разработку, разработку мобильных приложений и встроенные системы.

**11. Динамическое тестирование: общие сведения, значение.**

Динамическое тестирование – это процесс проверки программного обеспечения, при котором выполняются тестовые сценарии и анализируются результаты выполнения программы. Это активный подход к тестированию, который основан на запуске программы с целью выявления дефектов, ошибок и несоответствий.

Значение динамического тестирования заключается в том, что оно позволяет проверить работоспособность и функциональность программы в реальном времени. Путем выполнения тестовых сценариев можно выявить различные ошибки, такие как некорректное поведение программы, неработающие функции, проблемы с производительностью и т. д.

Динамическое тестирование дополняется статическим тестированием, которое происходит без запуска программы. Оба подхода важны для достижения высокого качества программного обеспечения.

**12. Этапы формирования эталонов для тестирования и их описание.**

Этапы формирования эталонов для тестирования включают в себя следующие шаги:

1. Анализ требований: На этом этапе происходит анализ требований к программе и определение функций, которые должны быть протестированы. Это позволяет определить набор тестовых случаев, которые будут использованы для формирования эталонов.

2. Выбор данных: На этом этапе выбираются тестовые данные, которые будут использоваться при выполнении тестовых сценариев. Это могут быть различные входные параметры, граничные значения, случаи ошибок и т. д. Выбор данных основывается на анализе требований и определении важных сценариев использования.

3. Создание эталонов: На этом этапе создаются эталоны, которые представляют собой правильные результаты работы программы для выбранных тестовых данных. Эталоны могут быть созданы вручную или автоматически, например, с использованием инструментов автоматического тестирования.

4. Проверка эталонов: На этом этапе происходит проверка эталонов на правильность и соответствие требованиям. Если найдены ошибки или несоответствия, они должны быть исправлены, чтобы обеспечить корректность результатов тестирования.

Формирование эталонов для тестирования является важной частью процесса тестирования, поскольку правильные эталоны позволяют сравнивать результаты выполнения программы с ожидаемыми результатами и выявлять потенциальные ошибки и дефекты.

**13. Классификация анализа вредоносных программ:**

1) Статический анализ: включает в себя исследование кода и структуры программы без ее активного выполнения. Статический анализ используется для обнаружения уязвимостей, анализа потока данных и процедур, анализа паттернов злоупотребления, анализа API и других методов.

2) Динамический анализ: осуществляется путем активного выполнения вредоносной программы в контролируемой среде. Включает в себя анализ поведения программы, отслеживание системных вызовов, мониторинг сетевой активности, анализ деобфускации кода и прочее.

3) Анализ памяти: представляет собой изучение работы вредоносной программы в памяти компьютера, обнаружение модификации структур данных и раскрытие скрытых алгоритмов.

4) Анализ сетевой активности: изучение сетевого взаимодействия вредоносной программы с другими компьютерами или серверами. Помогает выявить команды, передаваемые управляющим серверам, внешние домены или IP-адреса связанные с злоумышленниками.

5) Обратное инженерия: заключается в исследовании исполняемого кода или программного обеспечения с целью выявления его работы и функций, включая анализ алгоритмов шифрования, методов скрытия и других инструментов.

**14. Этапы изучения вредоносного программного обеспечения и их описание:**

1) Сбор образца: на этом этапе вредоносная программа должна быть обнаружена и зафиксирована, чтобы можно было провести дальнейший анализ.

2) Декомпиляция и деобфускация: вредоносная программа может быть зашифрована, обфусцирована или скрыта. Этот этап включает раскрытие и восстановление исходного кода или структуры программы.

3) Анализ структуры и функций: на этом этапе исследуются различные модули, функции и алгоритмы вредоносной программы для понимания ее целей и возможностей.

4) Изучение сетевой активности: данный этап фокусируется на анализе сетевой активности вредоносной программы, поиск связанных доменов и IP-адресов, анализ протоколов и команд, передаваемых по сети.

5) Анализ поведения: вредоносная программа может иметь различные поведенческие характеристики, такие как манипуляция файлами, регистрацией клавиатуры или мониторингом сетевой активности. На этом этапе изучается поведение программы, чтобы понять ее функциональность.

6) Построение модели угрозы: на основе полученных данных и анализа строится модель угрозы, которая помогает оценить риски и разработать меры безопасности для предотвращения атак и защиты от вредоносных программ.

7) Создание отчета и рекомендаций: результаты анализа вредоносной программы документируются в отчете, который содержит описание ее характеристик, функциональности и рекомендации по защите от подобных атак в будущем.

**15. Основные инструменты для исследования кода программ:**

1. Интегрированная среда разработки (IDE) - это программное обеспечение, которое предоставляет разработчику удобную среду для написания, отладки и тестирования кода программы. Основные функции IDE включают автодополнение кода, отладку, интеграцию с системой контроля версий и анализ кода.

2. Отладчик (Debugger) - это инструмент, который позволяет разработчику выполнять код программы по шагам и проверять состояние переменных и объектов в процессе выполнения. Отладчик позволяет обнаруживать и исправлять ошибки в коде и понять, как программные объекты взаимодействуют друг с другом.

3. Профилировщик (Profiler) - это инструмент, который помогает оптимизировать производительность программы. Профилировщик анализирует время, затраченное на выполнение каждого сегмента кода, и помогает выявить медленные участки программы. Это позволяет разработчикам устранить узкие места и повысить эффективность кода.

4. Система контроля версий (Version Control System, VCS) - это инструмент, который позволяет разработчику отслеживать и управлять изменениями в коде программы. Система контроля версий позволяет сохранять предыдущие версии кода, работать над изменениями вместе с другими разработчиками и откатывать изменения при необходимости. Некоторые популярные системы контроля версий - Git, SVN и Mercurial.

**16. Файловый монитор (FileMonitor)** - это инструмент, который позволяет отслеживать изменения, происходящие в файлах и директориях операционной системы. Файловый монитор предоставляет возможность отслеживать создание, удаление, изменение файлов и директорий, а также получать уведомления об этих событиях.

Возможности файлового монитора включают:

1. Отслеживание изменений в реальном времени - файловый монитор может непрерывно мониторить файлы и директории, чтобы обнаружить любые изменения и сразу же предоставить информацию о них.

2. Уведомления об изменениях - файловый монитор может отправлять уведомления или вызывать определенные действия при возникновении определенных событий, таких как создание, изменение или удаление файлов и директорий.

3. Журналирование - файловый монитор может вести журнал всех изменений, произошедших в файлах и директориях, чтобы можно было анализировать их позже.

4. Идентификация владельца и доступа - файловый монитор может предоставить информацию о владельцах файлов и директорий, а также о правах доступа к ним.

Файловый монитор широко используется для мониторинга файловых систем, обнаружения и предотвращения несанкционированного доступа и отслеживания изменений в важных системных файловых структурах.

**17. Методы, применяемые для обнаружения модифицированного кода. Преимущества и недостатки их применения.**

Для обнаружения модифицированного кода применяются различные методы, включая статический анализ, динамический анализ и обнаружение аномалий.

Статический анализ - это метод, основанный на анализе исходного кода программы, без ее реального выполнения. Он позволяет выявить изменения в программном коде, которые могут указывать на модификации или внесение вредоносного кода. Преимущества статического анализа включают его способность обнаружить различные типы изменений в коде и внести минимальные изменения в программу. Однако недостатками статического анализа являются высокие требования к экспертным знаниям и сложности в обнаружении некоторых типов вредоносного кода.

Динамический анализ - это метод, основанный на исполнении программы и наблюдении за поведением кода в реальном времени. Он может выявить изменения в программе, которые проявляются только при выполнении кода. Преимущества динамического анализа включают его способность обнаруживать изменения в программе в режиме реального времени и более низкие требования к экспертным знаниям. Однако недостатками динамического анализа являются его сложность в обнаружении некоторых типов изменений в программе и возможные негативные эффекты на работу системы.

Обнаружение аномалий - это метод, основанный на анализе поведения программы и выявлении отклонений от нормального поведения. Он позволяет выявлять неизвестные уязвимости и изменения в программе, которые могут указывать на наличие модифицированного кода. Преимуществом обнаружения аномалий является его способность обнаруживать новые типы изменений и уязвимостей. Однако недостатками этого метода являются его сложность в обнаружении изменений, которые могут быть схожи с нормальным поведением программы, и высокая вероятность ложных срабатываний.

**18. Отладчик** - это инструмент разработки программного обеспечения, который позволяет программисту исследовать и исправлять ошибки в программном коде. Он обычно предоставляет функции, такие как установка точек останова, наблюдение за переменными и выполнение кода шаг за шагом.

Виды отладчиков:

1. Локальный отладчик - это отладчик, который работает на локальной машине программиста и управляет отладкой программного кода, запущенного на этой же машине. Это самый распространенный вид отладчика, который обычно встроен в интегрированные среды разработки (IDE).

2. Удаленный отладчик - это отладчик, который работает на удаленной машине и управляет отладкой программного кода, запущенного на удаленной машине. Этот тип отладчика обычно используется при разработке и отладке программ для удаленных систем или при тестировании программного обеспечения на удаленных устройствах.

3. Пост-мортем отладчик - это отладчик, который используется для анализа сбоев и аварий, которые произошли в уже выпущенном продукте. Он позволяет программистам анализировать состояние программы на момент сбоев и находить ошибки и причины сбоев.

Отладчики являются важными инструментами разработки программного обеспечения, которые позволяют программистам сократить время и усилия, затрачиваемые на поиск и исправление ошибок.

**19. Трассировка кода программы** - это отладочный механизм, который позволяет программисту следить за выполнением программного кода и анализировать его работу поэтапно.

Назначение трассировки кода заключается в том, чтобы идентифицировать ошибки в программе и выявить неожиданное поведение. Она также может использоваться для изучения производительности программы и оптимизации.

Принцип работы трассировки кода состоит в том, чтобы программа записывала информацию о каждом шаге во время выполнения. Эта информация может включать значения переменных, вызовы функций и промежуточные результаты. Затем программист может анализировать эту информацию, чтобы определить, где возникают ошибки или проблемы.

**20. Контрольные точки останова** - это отладочный механизм, который позволяет программисту временно останавливать выполнение программы в определенных местах, чтобы проанализировать текущее состояние программы и переменных.

Назначение контрольных точек останова состоит в том, чтобы идентифицировать и исправить ошибки, определить исходное значение переменной в определенной точке программы или проанализировать последовательность выполнения кода.

Принцип работы контрольных точек останова заключается в том, чтобы программист установил точку останова в определенном месте программы. Когда программа достигает этой точки, выполнение приостанавливается, и программист может проанализировать текущие значения переменных, выполнить дополнительные действия (например, изменить значение переменной) или продолжить выполнение программы. Контрольные точки останова также могут быть установлены на условиях, что позволяет останавливать выполнение программы только в определенных ситуациях.

**21. Понятие тестирования программ** означает проверку программного обеспечения на соответствие требованиям и ожиданиям пользователя. Назначение тестирования программ состоит в обеспечении качества программного обеспечения и выявлении дефектов или ошибок.

Методы тестирования программ могут включать:

- Модульное тестирование: проверка отдельных частей программы (модулей) на правильность работы.

- Интеграционное тестирование: проверка взаимодействия различных модулей в рамках программы.

- Системное тестирование: проверка работы программы в рамках всей системы, включая взаимодействие с другими программами и оборудованием.

- Пользовательское тестирование: проверка программы пользователями на соответствие их требованиям и ожиданиям.

Для проведения тестирования программ могут использоваться различные методы, такие как тесты на функциональность, нагрузочное тестирование, тестирование стабильности и др. Результатами тестирования являются отчеты о найденных ошибках, а также уверенность в правильном функционировании программы.

**22. Понятие анализа программного кода** означает изучение и оценку исходного кода программы для выявления ошибок, уязвимостей или других проблем, которые могут влиять на работу программы.

Анализ программного кода включает в себя:

- Оценка качества кода: проверка стиля программирования, соответствия стандартам программирования, наличия дублирования кода и т. д.

- Выявление ошибок: поиск синтаксических, логических или других ошибок в коде.

- Поиск уязвимостей: обнаружение уязвимостей безопасности или потенциальных проблем с безопасностью в коде.

Анализ программного кода может быть выполнен как статически (без запуска программы), так и динамически (с запуском программы для проверки ее работы). Результатом анализа кода может быть создание отчета о найденных проблемах, а также рекомендации по их устранению или улучшению кода программы.

**23. Понятие и основные отличия тестирования моделей.**

Тестирование моделей – это процесс проверки моделей на соответствие требованиям и обнаружение возможных ошибок или дефектов. Модель может быть любым представлением системы, таким как математическая модель, программная модель или физическая модель.

Основные отличия тестирования моделей:

1) Цель тестирования. Если обычное тестирование программного обеспечения направлено на проверку функциональности и корректности работы программы, то тестирование моделей сконцентрировано на проверке достоверности представления системы в виде модели.

2) Виды ошибок/дефектов. В тестировании моделей проверяются такие ошибки, как несоответствие модели реальности, некорректность моделирования изменений системы и неправильное использование методов моделирования. В классическом тестировании программного обеспечения обычно уделяется большее внимание соблюдению требований к функциональности.

3) Процесс тестирования. Тестирование моделей включает выбор и разработку тестовых случаев, выполнение тестов, анализ результатов и отчетность. Кроме того, может применяться верификация и валидация моделей для проверки их корректности. В обычном тестировании программного обеспечения присутствуют более широкие варианты тестирования, такие как тестирование отдельных функций, тестирование производительности, тестирование безопасности и др.

**24. Характеристика и описание процесса тестирования методом "черный** **ящик".**

Метод "черный ящик" (black box testing) в тестировании – это подход, при котором тестирующий имеет доступ только к входным данным и ожидаемому результату, а внутреннее устройство и работа тестируемой системы остаются неизвестными.

Характеристики метода "черный ящик":

1) Основан на анализе требований и спецификации. Тестирующий использует требования и спецификации, чтобы определить ожидаемые результаты и разработать тестовые случаи.

2) Независимость от внутренней структуры системы. При использовании метода "черный ящик" тестирующий не знает, как работает система внутри, и на основе входных данных и ожидаемых результатов проверяет правильность работы системы.

3) Ориентирован на проведение функционального тестирования. Метод "черный ящик" обычно направлен на проверку функциональности системы, на соответствие требованиям и на поиск ошибок или дефектов в работе системы.

Процесс тестирования методом "черный ящик" включает следующие этапы:

1) Анализ требований и спецификации системы.

2) Разработка тестовых сценариев и создание тестовых данных.

3) Выполнение тестовых случаев и регистрация результатов.

4) Сравнение фактических результатов с ожидаемыми.

5) Оценка результатов и отчетность о найденных дефектах или ошибках.

6) Повторение процесса тестирования до достижения требуемого уровня качества системы.

**25. Метод "стеклянный ящик" (также известный как метод "белого ящика")** это метод тестирования программного обеспечения, который основывается на изучении внутренней структуры и логики программы.

Характеристики и описание процесса тестирования методом "стеклянный ящик" включают:

1. Знание внутренней структуры: Тестировщик имеет полное или частичное знание о внутренней структуре программы, включая код, алгоритмы и данные.

2. Использование кода: Тестирование включает использование знания о внутренней структуре программы, чтобы создавать тестовые случаи и наборы данных, которые позволяют изучать различные пути выполнения и подвергать программу проверке.

3. Разработка тестов: Тестировщик может использовать использование циклов, условных выражений, операторов и других элементов кода программы для разработки тестовых случаев, которые охватывают все возможные ветвления и состояния программы.

4. Управление ошибками: Тестировщик использует свое знание о программе для идентификации и устранения ошибок, связанных с кодом, алгоритмами, границами и другими внутренними аспектами программы.

5. Итеративность: Тестировщик проводит тестирование итеративно, изменяя набор тестов в соответствии с новыми знаниями о внутренней структуре программы и обнаруженных ошибках.

Метод "стеклянный ящик" позволяет более глубоко взглянуть на программу и увидеть ее внутреннюю работу. Этот метод способствует выявлению и устранению сложных ошибок и помогает улучшить качество и надежность программного обеспечения.

**26. Эмулирующий отладчик** - это инструмент разработки программного обеспечения, который позволяет программистам воссоздавать и проводить отладку определенных сценариев или ситуаций, чтобы исследовать код и выявить ошибки или неправильное поведение программы.

Основные особенности эмулирующего отладчика:

1. Эмуляция среды: Эмулирующий отладчик создает виртуальную среду, в которой можно воспроизводить определенные сценарии и устанавливать точки останова в коде для анализа его выполнения.

2. Имитация данных: Отладчик может имитировать различные входные данные или значения переменных для проверки реакции программы на разные условия.

3. Управление выполнением: Эмулирующий отладчик предоставляет возможность управлять выполнением программы, такими функциями, как выполнение по шагам, пропуск или остановка выполнения в точках останова.

4. Профилирование и анализ: Отладчик обычно предоставляет дополнительные инструменты для анализа производительности и профилирования кода, что позволяет программистам оптимизировать выполнение программы.

5. Выявление ошибок и уязвимостей: Эмулирующий отладчик может помочь выявить ошибки, исключения или уязвимости программы, позволяя программистам искать причины и исправлять их.

Эмулирующий отладчик является важным инструментом в процессе разработки программного обеспечения, который позволяет проверить и анализировать код в контролируемой среде, улучшая надежность и качество программы.

**27. Методы проникновения вредоносного программного обеспечения на персональный компьютер могут включать следующие способы:**

1. Фишинг: Злоумышленники могут отправлять поддельные электронные письма или создавать поддельные веб-сайты, чтобы обмануть пользователей и заставить их предоставить личную информацию или установить вредоносное ПО.

2. Социальная инженерия: Злоумышленники могут попытаться обмануть пользователей с помощью манипуляций и убедительных методов, чтобы убедить их установить или открыть вредоносный файл или ссылку.

3. Эксплойты: Злоумышленники могут использовать уязвимости в операционной системе или программном обеспечении для внедрения вредоносного ПО. Эксплойты могут быть внедрены через вредоносные веб-сайты, почтовые вложения или даже через непатченые программы.

4. Вредоносные вложения электронной почты: Злоумышленники могут включать вредоносные программы во вложения электронной почты и отправлять их пользователям. Если пользователи открывают вложения, вредоносное ПО может установиться на их компьютерах.

5. Зараженные веб-сайты: Злоумышленники могут создавать или загружать вредоносное ПО на веб-сайты, которые пользователи посещают. Когда пользователи посещают такие сайты, вредоносное ПО может загрузиться и установиться на их компьютерах.

**28. Вредоносное программное обеспечение (ВПО)** – это программы, созданные с целью нанести вред компьютеру и пользователям. Оно может быть разработано для сбора чувствительной информации, нарушения безопасности, уничтожения данных или похищения личных данных. Вредоносное ПО часто распространяется через поддельные электронные письма, веб-сайты, социальные сети или зараженные приставки USB. Оно может проникнуть на компьютер и выполняться в фоновом режиме, без ведома и согласия пользователя. Примеры вредоносного ПО включают вирусы, черви, трояны, шпионское ПО и рекламное ПО. Понимание понятия вредоносного ПО позволяет пользователям принимать меры для защиты своих компьютеров и важной информации.

**29. Понятие метрики измерений:**

Метрика измерений – это средство для оценки и измерения определенной характеристики или параметра в рамках определенной системы или процесса. Она может быть количественной или качественной и используется для оценки эффективности, производительности или качества данной системы или процесса. Метрики измерений помогают собирать, анализировать и интерпретировать данные для принятия решений и оптимизации процессов.

**30. Хранилище (Repository): основные понятия:**

Хранилище (Repository) – это централизованное место, где хранятся и организуются данные, файлы, документы и другие ресурсы. Он предоставляет единый доступ к данным для разных пользователей и приложений и обеспечивает их сохранность, надежность и консистентность.

В хранилище могут храниться различные типы данных, такие как программные файлы, документы, изображения, видео, аудио, базы данных и другие. Хранилище может предоставлять разные возможности для организации данных, такие как версионирование, контроль доступа, индексирование и поиск, а также функции совместной работы и управления изменениями.

Хранилище может быть реализовано с использованием различных технологий, таких как файловые системы, базы данных, облачные хранилища и другие. Оно является важным элементом в информационных системах и обеспечивает эффективное управление и использование данных.

**31. Рабочая копия (Working Copy)** - это копия репозитория в системе контроля версий (Version Control System, VCS), которая доступна для работы над изменениями без влияния на основной репозиторий. Рабочая копия содержит все файлы и историю изменений, связанные с проектом. При внесении изменений в рабочую копию, можно добавлять, изменять или удалять файлы, а затем сохранять изменения в репозитории.

**32. Тестирование ПО (Software Testing)** - это процесс проверки и оценки программного обеспечения на соответствие заданным требованиям и оценке его качества. Основная цель тестирования ПО заключается в выявлении дефектов, ошибок, недоработок и проблем, которые могут повлиять на корректность, надежность и производительность программы.

Некоторые ключевые понятия в тестировании ПО:

- Тестовый случай (Test Case) - это документированная последовательность шагов, которые следует выполнить для проверки определенной функциональности или аспекта программного обеспечения.

- Тестовый сценарий (Test Scenario) - это описание серии связанных тестовых случаев, которые должны быть выполнены для проверки определенного аспекта программного обеспечения.

- Результат тестирования (Test Result) - это информация о том, успешно ли выполнен тест или были обнаружены какие-либо ошибки или проблемы.

- Тестовая среда (Test Environment) - это комбинация аппаратных и программных средств, на которых выполняется тестирование программного обеспечения.

- Типы тестирования (Types of Testing) - это различные подходы или методологии, используемые для проверки различных аспектов программного обеспечения. Некоторые общие типы включают функциональное, нагрузочное, интеграционное, регрессионное и безопасности тестирование.

Различные виды тестирования включают в себя:

- Модульное тестирование (Unit Testing) - тестирование отдельных модулей или компонентов программы.

- Интеграционное тестирование (Integration Testing) - тестирование взаимодействия между различными модулями или компонентами программного обеспечения.

- Системное тестирование (System Testing) - тестирование всей системы или приложения в целом.

- Приемочное тестирование (Acceptance Testing) - тестирование программного обеспечения с целью проверки его соответствия требованиям заказчика или пользователей.

**33. Понятие управления контроля версиями (СКВ)** относится к процессу управления изменениями в программном обеспечении или других документах. СКВ позволяет отслеживать изменения, вносимые в файлы и документы, и хранить предыдущие версии содержимого. Он также предоставляет возможности сравнивать изменения и восстанавливать предыдущие версии, если это необходимо.

Применение системы контроля версий:

- Улучшение сотрудничества: СКВ позволяет нескольким пользователям работать над одним и тем же файлом, сливать изменения и смотреть, кто и когда внес какие изменения.

- Отслеживание изменений: СКВ позволяет легко отслеживать изменения в файлах, включая внесение новых функций, исправление ошибок и устранение недочетов.

- Восстановление истории версий: СКВ сохраняет историю всех изменений, сделанных в файле, и позволяет легко восстановить предыдущие версии в случае необходимости.

- Резервное копирование: СКВ обеспечивает резервное копирование файлов и документов, чтобы в случае потери данных можно было восстановить предыдущую версию.

Функции системы контроля версий:

- Запись изменений: СКВ записывает все изменения, сделанные в файле или документе, чтобы можно было отслеживать, кто внес изменения и когда это произошло.

- Сравнение и слияние изменений: СКВ позволяет сравнивать различные версии файла и объединять изменения, сделанные несколькими пользователями.

- Восстановление предыдущих версий: СКВ позволяет легко восстановить предыдущие версии файла или документа в случае потери или ошибки.

- Отслеживание истории изменений: СКВ сохраняет историю всех изменений, внесенных в файл или документ, чтобы можно было отследить, какие изменения были сделаны и кем.

**34. Классификация систем контроля версий (СКВ)** может быть основана на различных критериях. Одна из возможных классификаций включает следующие типы СКВ:

- Локальные СКВ: Это системы, в которых каждый пользователь хранит свои изменения локально на своем компьютере. Примером локальных СКВ является RCS (Revision Control System).

- Централизованные СКВ: Это системы, в которых все изменения хранятся на центральном сервере. Пользователи получают актуальные версии файлов из центрального сервера и вносят изменения в локальные копии файлов. Примерами централизованных СКВ являются CVS (Concurrent Versions System) и Subversion (SVN).

- Распределенные СКВ: Это системы, в которых каждый пользователь имеет полную копию репозитория, в которой хранятся все версии файлов. Пользователи могут вносить изменения и обмениваться ими между собой. Примерами распределенных СКВ являются Git и Mercurial.

Кроме того, можно также классифицировать СКВ по типу используемого хранилища (например, файловая система или база данных), типу доступа (только для чтения или с возможностью записи) и другим критериям.

**35. Контроль версий типа SVN (Subversion)** является одним из инструментов управления изменениями в программном обеспечении. Он позволяет разработчикам отслеживать и управлять изменениями в исходных кодах проектов, а также координировать работу над различными версиями программы.

Смысл контроля версий состоит в следующем:

1. Отслеживание изменений: SVN позволяет хранить исходные коды и все изменения, вносимые в них. Разработчики могут видеть, кому принадлежит изменение, что было изменено и когда происходило изменение. Это позволяет легко восстановить предыдущие версии кода и отслеживать историю изменений.

2. Координация работы: при работе в команде SVN обеспечивает синхронизацию изменений между различными разработчиками. Каждый разработчик может работать над своей версией кода, а затем объединить свои изменения с общим репозиторием. Это позволяет избежать конфликтов и упрощает совместную разработку.

3. Управление версиями: SVN позволяет хранить несколько версий программы одновременно. Это позволяет вернуться к предыдущей версии, если новые изменения вызовут проблемы. Кроме того, SVN позволяет создавать теги и ветки, что упрощает управление выпусками и параллельное развитие различных версий программы.

Современные подходы к анализу ПО включают:

1. Статический анализ: это анализ исходного кода и других артефактов программы без ее выполнения. Он позволяет выявить потенциальные ошибки, уязвимости, неправильное использование API и другие проблемы.

2. Динамический анализ: это анализ программы во время ее выполнения. Он позволяет выявить проблемы производительности, утечки памяти, неправильную работу алгоритмов и другие проблемы, которые могут проявиться только при выполнении программы.

3. Анализ зависимостей: это анализ связей между различными компонентами программы. Он позволяет выявить слабые места, лишние зависимости и проблемы с модульностью.

4. Анализ данных: это анализ данных, используемых или создаваемых программой. Он позволяет выявить проблемы с правильностью данных, нарушениями целостности и другие проблемы, связанные с обработкой данных.

**36. Уровни тестирования включают:**

1. Модульное тестирование: тестирование отдельных модулей программы (функций, классов, методов) для проверки их корректности и работоспособности.

2. Интеграционное тестирование: тестирование взаимодействия между различными модулями программы для проверки их совместной работоспособности.

3. Системное тестирование: тестирование всей системы или приложения в целом для проверки его соответствия требованиям и поведению в реальных условиях использования.

4. Приемочное тестирование: тестирование системы или приложения в соответствии с требованиями и ожиданиями заказчика для проверки его готовности к эксплуатации.

Цикл тестирования - это последовательность этапов тестирования, которые выполняются в определенном порядке. Он включает следующие этапы:

1. Планирование: определение целей, объема, временных рамок и ресурсов для тестирования.

2. Создание тестовых случаев: разработка конкретных сценариев тестирования, которые позволят проверить различные аспекты системы.

3. Подготовка среды тестирования: настройка тестовой среды, установка необходимых программных и аппаратных компонентов.

4. Исполнение тестовых случаев: запуск тестовых сценариев и регистрация результатов тестирования.

5. Анализ результатов: оценка результатов тестирования, выявление проблем и ошибок, а также принятие решений о дальнейшем действии.

6. Повторное тестирование: исправление ошибок и повторное тестирование для проверки их устранения.

7. Отчетность: создание отчетов о результатах тестирования, которые будут использованы для принятия решений и дальнейшего развития проекта.

Графическое отображение цикла тестирования в виде иерархического дерева позволяет визуализировать связи и последовательность этапов тестирования. Обычно корень дерева представляет собой этап планирования, дочерние узлы - этапы создания тестовых случаев, подготовки среды, исполнения тестов и анализа результатов, а листья дерева - этапы повторного тестирования и отчетности.

**37. Импорт и экспорт настроек: общие сведения, применение.**

Импорт и экспорт настроек - это процессы переноса настроек и конфигурационных данных между различными программами, приложениями или устройствами. Эти настройки могут включать параметры пользовательского интерфейса, предпочтения пользователей, настройки безопасности и другие релевантные данные.

Применение импорта и экспорта настроек может быть разнообразным. Некоторые из основных применений включают:

1. Перенос настроек на новое устройство: Пользователи могут экспортировать свои настройки с одного устройства и импортировать их на новый, чтобы сохранить предпочтения и параметры конфигурации.

2. Обмен данными между программами: Импорт и экспорт настроек позволяет программам передавать данные между собой, чтобы обеспечить совместимость и переносимость.

3. Создание резервной копии и восстановление настроек: Пользователи могут экспортировать настройки в файл или облачное хранилище для создания резервной копии, которую можно восстановить в случае сбоя или потери данных.

4. Массовое применение настроек: Некоторые программы позволяют экспортировать настройки в файл, который может быть быстро импортирован на другие устройства или учетные записи пользователей, чтобы упростить настройку.

В целом, импорт и экспорт настроек позволяют пользователям сохранять, передавать и восстанавливать свои конфигурационные данные, обеспечивая удобство и гибкость использования программ и устройств.

**38. Обратное проектирование: основные понятия, виды применяемых инструментов и описание процесса.**

Обратное проектирование (reverse engineering) - это процесс анализа уже существующего продукта или системы с целью извлечения информации о его структуре, функциональности и дизайне. Оно обычно применяется для восстановления документации, понимания работы системы или создания нового продукта, совместимого с исходным.

Виды применяемых инструментов обратного проектирования могут включать:

1. Дизассемблеры: Исследуют исполняемые файлы или код программы и восстанавливают исходный код, на котором была основана программа.

2. декомпиляторы: Извлекают исходный код из скомпилированного бинарного файла или исполняемого файла.

3. Статический анализатор кода: Используется для анализа кода и выявления зависимостей, структуры и функциональности.

4. Программы для анализа данных исходного кода: Эти инструменты позволяют обрабатывать и анализировать исходный код для выявления структуры и функциональности.

Процесс обратного проектирования может включать следующие шаги:

1. Сбор информации: Этот шаг включает сбор доступной информации о системе, такой как исполняемые файлы, исходный код, документация и т. д.

2. Анализ функциональности: Анализируется функциональность системы с целью выявления основных компонентов и взаимосвязей между ними.

3. Анализ дизайна: Исследуется архитектура и дизайн системы для понимания структуры и организации компонентов.

4. Извлечение исходного кода: С использованием инструментов обратного проектирования извлекается исходный код или его приближение.

5. Документирование: Результаты анализа и извлечения документируются для будущего использования и понимания.

Обратное проектирование может быть полезным во многих ситуациях, таких как модернизация или оптимизация существующих систем, понимание работы конкурентных продуктов или восстановление утраченной документации. Однако важно иметь в виду ограничения, связанные с правами интеллектуальной собственности и законодательством, касающимся обратного проектирования.

**39. Применение и функции обратного проектирования:**

Обратное проектирование (Reverse Engineering) является процессом извлечения информации о техническом объекте из уже существующего продукта или устройства. Применение и функции обратного проектирования включают:

1. Разработка резервных копий и восстановление: Обратное проектирование может использоваться для создания резервных копий устаревших или дефектных устройств, а также для восстановления утерянной документации или конструкции.

2. Исследование конкурентных продуктов: Обратное проектирование может использоваться для изучения конкурентных продуктов с целью определения их особенностей, структуры и функций. Это позволяет разработчикам улучшить свои собственные продукты и найти новые решения.

3. Модификация и улучшение существующих продуктов: Обратное проектирование позволяет анализировать и модифицировать существующие продукты, добавлять новые функции или улучшать их производительность и эффективность.

4. Интеграция систем и процессов: Обратное проектирование позволяет сопоставить различные системы и процессы для обеспечения их совместной работы и соответствия требованиям.

5. Улучшение безопасности и снижение рисков: Обратное проектирование помогает выявлять уязвимости и риски существующих систем и устройств, что позволяет разработать меры для улучшения безопасности и снижения возможных проблем.

**40. Описание анализа обратного проектирования:**

Анализ обратного проектирования включает в себя следующие этапы:

1. Сбор информации: На этом этапе проводится сбор исходных данных о продукте или устройстве, включая его физические свойства, электрические схемы, программное обеспечение и другую важную информацию.

2. Документация: Собранная информация о продукте документируется, чтобы обеспечить хранение и организацию данных для дальнейшего анализа и использования.

3. Обратный инжиниринг: На этом этапе проводится разбор продукта с целью выявления его структуры, функций и принципов работы. Для этого могут использоваться инструменты, такие как детальное изучение схем, разборка устройства или анализ программного кода.

4. Анализ данных: Полученная информация изучается и анализируется с целью выявления основных особенностей и характеристик продукта, его компонентов и процессов.

5. Восстановление и моделирование: На основе полученных данных и анализа происходит восстановление структуры и функций продукта. Это может включать создание 3D-модели, структурных диаграмм или программного кода.

6. Оценка и улучшение: Полученные результаты анализа используются для оценки существующего продукта и возможности его улучшения. Это может включать рекомендации по изменению дизайна, добавлению новых функций или улучшению качества и производительности.

Анализ обратного проектирования позволяет лучше понять существующие продукты и устройства, а также использовать эту информацию для разработки новых и улучшенных решений.

**41. Модульное тестирование (unit testing)** - это процесс тестирования отдельных модулей программного обеспечения (частей программы), чтобы проверить их корректность работы в изоляции от других компонентов системы. Оно выполняется покрытием функций или блоков кода с тестами, которые проверяют, что каждая функция работает должным образом и возвращает ожидаемые результаты.

Основные понятия модульного тестирования включают:

- Модуль: самостоятельная часть программного обеспечения, которая может быть протестирована независимо от других модулей.

- Тестовый случай: конкретная комбинация входных данных и ожидаемого результата, которые проверяются в модульном тесте.

- Пограничные значения: значения входных данных, которые находятся на границе допустимого диапазона и часто проверяются на возможные ошибки.

- Моки и стабы: заглушки или заменители, которые используются для создания необходимого окружения для тестирования, когда некоторые зависимости модуля не доступны или сложны для воспроизведения.

Результатом модульного тестирования является набор пройденных или неудачных тестов. Если все тесты успешно проходят, это означает, что модуль работает правильно. Если один или несколько тестов проваливаются, это указывает на наличие ошибок или проблем в коде модуля, которые нужно исправить перед интеграцией его со всеми остальными частями системы.

**42. Интеграционное тестирование (integration testing)** - это процесс тестирования взаимодействия и коммуникации между различными модулями или компонентами программного обеспечения для проверки их корректной интеграции. Оно выполняется после модульного тестирования для проверки, что отдельные модули взаимодействуют и работают вместе так, как должны.

Основные понятия интеграционного тестирования включают:

- Интеграционный тест: тест, который проверяет взаимодействие между двумя или более модулями или компонентами программного обеспечения.

- Интерфейсы: точки входа или выхода, через которые различные модули взаимодействуют друг с другом.

- Тестовые данные: данные, используемые для проверки правильности взаимодействия между модулями.

- Методы интеграции: различные стратегии и подходы к интеграционному тестированию, такие как "сверху вниз", "снизу вверх", "покрытие" и другие.

Результатом интеграционного тестирования является обнаружение ошибок в взаимодействии модулей или компонентов программного обеспечения, которые могут возникнуть при интеграции их вместе. Если все интеграционные тесты успешно проходят, это означает, что модули работают правильно в контексте своей интеграции. Если один или несколько тестов проваливаются, это указывает на наличие проблем в интеграции, которые нужно исправить перед продолжением разработки или релизом программного обеспечения.

**43. Системное тестирование: основные понятия, элементы, результат.**

Системное тестирование - это процесс проверки работы и функциональности всей системы в целом. Оно проводится после завершения интеграционного тестирования и обычно выполняется командой тестировщиков.

Основные понятия в системном тестировании:

1. Тест-кейсы: набор шагов и ожидаемых результатов, которые проверяют определенный аспект работы системы.

2. Тест-план: документ, который определяет объем, цели, подход и расписание для проведения системного тестирования.

3. Тестовые данные: данные, используемые для выполнения тест-кейсов, чтобы проверить различные сценарии использования системы.

4. Среда тестирования: оборудование, программное обеспечение и конфигурация, на которых выполняется тестирование.

5. Дефекты: несоответствия в работе системы или неожиданное поведение, которые должны быть исправлены разработчиками.

Элементы системного тестирования:

1. Подготовка: определение целей, разработка тест-плана, установление тестовой среды и создание тест-кейсов и данных.

2. Выполнение: запуск тест-кейсов, регистрация результатов и дефектов.

3. Анализ: оценка результатов тестирования и рассмотрение дефектов, чтобы определить, требуется ли исправление или изменение системы.

4. Отчетность: подготовка отчетов о выполненных тестах, выявленных дефектах и результатов анализа.

Результат системного тестирования включает в себя:

- Сводный отчет о выполненных тестах и их результатах.

- Описания выявленных дефектов и требований к их исправлению.

- Оценки качества работы системы и ее соответствия заявленным требованиям.

- Повторно используемая база знаний о системе, полученная в процессе тестирования.

**44. Локальный репозиторий: определение и функции.**

Локальный репозиторий - это хранилище кода и версий программы на компьютере разработчика. Он содержит все файлы, необходимые для работы с системой управления версиями (например, Git), и предоставляет возможность отслеживать изменения в коде, создавать и переключаться между разными ветками разработки, а также вносить изменения и сохранять их историю.

Функции локального репозитория:

1. Хранение версий программы: локальный репозиторий сохраняет все изменения в коде, включая создание, изменение и удаление файлов, чтобы разработчик мог отслеживать историю изменений.

2. Ветвление и слияние веток: разработчик может создавать разные ветки разработки, чтобы работать над разными функциями или исправлениями ошибок. Локальный репозиторий позволяет объединять изменения из разных веток или переключаться между ними.

3. Откат к предыдущим версиям: локальный репозиторий позволяет разработчику откатываться к предыдущим версиям программы, что полезно при исправлении ошибок или восстановлении работоспособности программы.

4. Работа в автономном режиме: локальный репозиторий позволяет разработчику работать с кодом без подключения к удаленному репозиторию. Это полезно, когда разработчик находится вне сети или не имеет доступа к удаленному репозиторию.

**45. Тестирование методом «серый ящик»** - это подход к тестированию программного обеспечения, который основан на знании некоторой информации о структуре и внутреннем устройстве тестируемой системы. В этом методе тестирования используется комбинация знаний о внутренней структуре системы и требованиях к ее функциональности. Тестирование «серым ящиком» позволяет тестировщику иметь представление о внутренней структуре системы и использовать эту информацию для разработки эффективных тестовых сценариев.

Процесс тестирования методом «серый ящик» обычно включает следующие шаги:

1. Анализ требований: изучение требований к системе и определение, какие аспекты внутренней структуры системы следует учесть при планировании и разработке тестовых случаев.

2. Изучение дизайна: анализ структуры и архитектуры системы, чтобы понять, как различные компоненты взаимодействуют, и определить, какие тестовые сценарии можно разработать.

3. Разработка тестовых случаев: на основе анализа требований и дизайна системы разработка тестовых сценариев для оценки функциональности и надежности системы.

4. Исполнение тестовых случаев: запуск разработанных тестовых сценариев и регистрация результатов тестирования.

5. Анализ результатов: анализ результатов тестирования, выявление дефектов и взаимодействие с разработчиками для исправления проблем.

**46. Получение доступа к GitHub** - это процесс создания аккаунта и получения доступа к платформе GitHub, где хранятся репозитории с открытым и закрытым кодом. Основные функции GitHub включают:

- Хранение кода: GitHub предоставляет возможность хранить и управлять версиями кода в репозиториях. Это позволяет командам разработчиков совместно работать над проектами и отслеживать изменения в коде.

- Контроль версий: GitHub основан на системе контроля версий Git, что позволяет разработчикам отслеживать изменения в коде, внести исправления и отменить изменения при необходимости.

- Коллаборация: разработчики могут совместно работать над проектами, создавая форки и подтверждая изменения через механизмы запросов на слияние (Pull Requests).

- Отслеживание ошибок: GitHub предоставляет возможность создания задач и отслеживания ошибок через встроенную систему отслеживания ошибок (Issue Tracking).

- Документация: разработчики могут создавать и поддерживать документацию для своих проектов на GitHub с помощью инструментов вроде Markdown и GitHub Pages.

Принцип GitHub основывается на открытой и коллаборативной разработке, предоставляя разработчикам возможность совместной работы над проектами и обмена кодом и идеями с другими участниками сообщества разработчиков.

**47. Организация разработки в сервисе GitHub:**

Организация разработки в сервисе GitHub может быть выполнена через использование функций и инструментов, предоставляемых платформой. Некоторые основные компоненты организации разработки в GitHub включают в себя:

- Репозитории: Репозитории являются центральными элементами разработки в GitHub. Они представляют собой места для хранения и управления кодом, документацией и другими файлами проекта.

- Коллабораторы: Коллабораторы - это люди, которым разрешено вносить изменения в репозитории. Они могут форкать репозитории, создавать ветки, делать коммиты и создавать запросы на слияние.

- Ветвление и слияние: GitHub предоставляет возможность создания веток для разработки новых функций или исправлений ошибок на основе существующего кода в репозитории. После завершения работы над веткой ее можно включить обратно в основную ветку с помощью запроса на слияние.

- Запросы на слияние: Запросы на слияние (Pull requests) предоставляют механизм для предложения изменений в коде репозитория и обсуждения с другими разработчиками. Они позволяют улучшить и контролировать качество кода перед его интеграцией в основную ветку.

- Проблемы и задачи: GitHub включает раздел для отслеживания проблем, задач и их обсуждения. Разработчики могут создавать задачи, указывать ответственных, обсуждать детали и контролировать прогресс.

- Обзор кода: GitHub предоставляет функциональность для обзора кода, позволяя другим разработчикам комментировать и оценивать предложенные изменения перед слиянием.

**48. Описание уровней доступа GitHub:**

В GitHub существуют различные уровни доступа, которые определяют, какие действия и функции доступны пользователям в рамках репозитория или организации. Некоторые общие уровни доступа в GitHub включают в себя:

- Владелец: Владелец - это уровень доступа, который предоставляет полный контроль над репозиторием или организацией. Владелец может управлять доступом и разрешениями других пользователей, создавать и удалять ветки, управлять запросами на слияние и другими аспектами разработки проекта.

- Администратор: Такой уровень доступа предоставляет пользователю возможность управлять настройками репозитория. Администратор может управлять коллабораторами, создавать и удалять ветки, отслеживать проблемы, настраивать предупреждения и т.д.

- Коллаборатор: Коллаборатор имеет доступ для внесения изменений в репозиторий. Он может создавать ветки, делать коммиты и создавать запросы на слияние.

- Член организации: Уровень доступа для пользователя, являющегося членом организации, определяет его роли и привилегии в контексте данной организации. Член организации может иметь различные уровни доступа в зависимости от настроек организации.

- Публичный доступ: Некоторые репозитории могут быть публичными, что означает, что любой пользователь может просматривать код, создавать запросы на слияние и вносить комментарии, но не может делать прямые изменения.

**49. Характеристика способов слияния копий репозитория.**

Существует несколько способов слияния копий репозитория. Вот некоторые из них:

- Слияние слиянием (Merge): Этот способ объединяет изменения из одной ветки (или веток) в другую. При слиянии Git автоматически пытается объединить изменения, если они не конфликтуют. Если возникают конфликты, Git позволяет разработчику решить их вручную. Слияние слиянием часто используется при работе с командой, где каждый член команды делает изменения в отдельной ветке и затем сливает их в основную ветку.

- Перебазирование (Rebase): В этом способе изменения из одной ветки ветки переносится в другую. В отличие от слияния слиянием, при перебазировании история коммитов может измениться, поскольку коммиты применяются поверх другой ветки. Это полезно, если вы хотите добавить свои изменения поверх актуальной версии кода. Однако перебазирование может создать сложности, если другие разработчики уже базируют свои изменения на вашей ветке.

- Вилка (Fork): Этот способ представляет собой создание копии репозитория на GitHub или другой платформе с хостингом Git. Вилка позволяет другим разработчикам вносить изменения в код, не влияя на оригинальный репозиторий. При необходимости владельцы оригинального репозитория могут интегрировать изменения из вилки.

**50. Модели создания кода в репозитории: описание и графическое представление.**

В репозитории можно использовать различные модели создания кода. Вот некоторые из них с их описанием и графическим представлением:

- Централизованная модель: В этой модели есть единственный центральный репозиторий, куда все разработчики вносят свои изменения. Один разработчик может захватить блокировку на изменение кода, что приводит к дублированию работы и замедляет процесс разработки.

Графическое представление:

```

┌─────────┐ pushes to ┌─────────┐

│ │ ─ ─ ─ ─ ─>│ │

│ Central │ │ Remote │

│ Repo │ < ─ ─ ─ ─ ┤ Repo │

│ │ < ─ ─ ─ ─ ┤ │

└─────────┘ pulls └─────────┘

```

- Распределенная модель: В этой модели каждый разработчик имеет свою копию репозитория. Они могут работать независимо, делая изменения и выкладывая их в центральный репозиторий или распространяя их между собой. Это позволяет разработчикам работать над разными частями проекта одновременно и ускоряет процесс разработки.

Графическое представление:

```

┌ ─ ─ ─ ─ ─ ─ ─ ┐ push ┌ ─ ─ ─ ─ ─ ─ ─ ┐

│ Developer 1 │ ─ ─ ─ ─ ─>│ Central │

│ Repo │ │ Repo │

│ │ < ─ ─ ─ ─ ┤ │

└ ─ ─ ─ ─ ─ ─ ─ ┘ pull └ ─ ─ ─ ─ ─ ─ ─ ┘

▲

│

│ pull

│

┌ ─ ─ ─ ─ ─ ─ ─ ┐ push ┌ ─ ─ ─ ─ ─ ─ ─ ┐

│ Developer 2 │ ─ ─ ─ ─ ─>│ Central │

│ Repo │ │ Repo │

│ │ < ─ ─ ─ ─ ┤ │

└ ─ ─ ─ ─ ─ ─ ─ ┘ pull └ ─ ─ ─ ─ ─ ─ ─ ┘

```

- Gitflow: Это модель, представляющая собой стратегию организации веток в репозитории. Она определяет основные ветки, такие как "master" (основная версия продукта) и "develop" (разрабатываемая версия продукта), а также вспомогательные ветки для функциональных и релизных версий. Эта модель обеспечивает хорошую организацию работы над разными функциональностями и выпускает стабильные релизы.

Графическое представление:

```

┌─────────────┐ ┌─────────────────┐

│ Release │ <───────│ Develop │

│ Branch │ └─────────────────┘

│ │

└─────────────┘

▲

│

┌───────┴───────┐

│ │

┌────────┴──────┐ ┌─────┴─────────┐

│ Feature │ │ Hotfix │

│ Branch │ │ Branch │

└────────┬──────┘ └─────┬─────────┘

│ │

**51. Архитектура MVC (Model-View-Controller)** – это популярный подход к организации структуры и разработке программного обеспечения. Основная идея заключается в разделении приложения на три основные составляющие:

- Модель (Model): отвечает за обработку данных, бизнес-логику и хранение состояния приложения. Модель не зависит от других компонентов архитектуры и не отображает данные напрямую.

- Представление (View): отвечает за отображение информации пользователю. Она получает данные от модели и представляет их в нужной форме, например, в виде HTML-страницы. Представление не содержит логики и не модифицирует данные.

- Контроллер (Controller): обрабатывает пользовательский ввод, взаимодействует с моделью и выбирает подходящее представление для отображения данных пользователю. Контроллер принимает решения на основе пользовательского ввода и взаимодействует с моделью, чтобы получить и обновить данные.

Общая структура MVC-архитектуры может быть представлена следующим образом:

+------------+

| Model |

+------------+

↑

| Получает данные

|

+-----+-----+-----+

| View |

+-----+-----+-----+

↑

| Отображает данные

|

+------------+-----+

| Controller |

+------------+-----+

↑

| Взаимодействует с моделью и представлением

**52. Процесс реализации проверки на стороне клиента (client-side validation) обычно включает следующие шаги:**

1. Добавление формы: Необходимо создать HTML-форму, содержащую поля, которые должны быть проверены на стороне клиента. Каждому полю формы можно присвоить атрибуты, такие как "required" или "pattern", чтобы указать необходимость заполнения поля или задать шаблон проверки.

2. Написание JavaScript-кода: С использованием JavaScript можно добавить проверку на стороне клиента для формы. Например, можно использовать объекты Document и Element для доступа к элементам формы, а затем провести проверку значений полей формы на основе определенных правил и условий.

3. Обработка событий: Чтобы активировать проверку на стороне клиента, нужно добавить обработчики событий к форме или отдельным элементам формы. Например, можно добавить событие "submit" к форме, чтобы проверить все поля перед отправкой данных на сервер.

4. Отображение сообщений об ошибке: Если проверка на стороне клиента обнаруживает ошибки, необходимо отобразить соответствующие сообщения об ошибках пользователю. Можно использовать элементы HTML, такие как <span> или <div>, чтобы отобразить сообщения рядом с соответствующими полями формы.

5. Комбинирование с серверной проверкой: Проверка на стороне клиента не должна заменять серверную проверку. Для обеспечения безопасности и целостности данных следует провести также проверку на стороне сервера.

В результате, процесс реализации проверки на стороне клиента позволяет проверить введенные пользователем данные до отправки их на сервер, что помогает повысить удобство использования и уменьшить нагрузку на сервер.

**53. JavaScript: инструменты языка. Состав среды разработки (IDE).**

JavaScript имеет широкий набор инструментов, которые помогают разработчикам в создании и отладке кода на этом языке. Некоторые из наиболее популярных инструментов включают в себя:

1. Редакторы кода: Редакторы кода, такие как Visual Studio Code, Sublime Text и Atom, обеспечивают функции подсветки синтаксиса, автодополнения кода, настраиваемые настройки форматирования и другие удобные функции для работы с JavaScript.

2. Отладчики: Браузеры, такие как Google Chrome и Firefox, предоставляют интегрированные инструменты для отладки JavaScript. Эти инструменты позволяют разработчикам просматривать и изменять значения переменных, отслеживать выполнение кода, а также находить и исправлять ошибки.

3. Пакетные менеджеры: JavaScript имеет несколько пакетных менеджеров, таких как npm (Node Package Manager) и yarn, которые позволяют разработчикам управлять зависимостями проекта и устанавливать сторонние библиотеки и фреймворки.

4. Компиляторы и сборщики: Некоторые разработчики JavaScript используют компиляторы и сборщики, такие как Babel и Webpack, чтобы преобразовывать код на JavaScript в совместимую версию, объединять и оптимизировать файлы и дополнительно обрабатывать код.

**54. Описание процесса реализации проверки на стороне клиента** Преимущества проверки на сервере.

Проверка на стороне клиента - это процесс валидации информации, который выполняется на компьютере или устройстве пользователя, перед отправкой данных на сервер. Принцип работы проверки на стороне клиента может варьироваться в зависимости от используемых инструментов и технологий, но обычно он включает следующие шаги:

1. Сбор данных: Данные, введенные пользователем в форму или другой интерфейс, собираются и сохраняются в переменных или объектах JavaScript.

2. Валидация данных: С помощью JavaScript разработчик может проверить, соответствуют ли введенные данные определенным правилам или ограничениям. Это может включать проверку формата данных (например, электронной почты), длину или диапазон значений, требования к паролю и другие параметры.

3. Отображение ошибок: Если данные не соответствуют заданным правилам, JavaScript может создать и отобразить сообщения об ошибках рядом с неправильными полями или в другом удобном месте на странице.

Преимущества проверки на стороне клиента включают:

1. Быстрая отзывчивость: Проверка на стороне клиента позволяет пользователям получать обратную связь по мере ввода данных без необходимости отправлять данные на сервер для проверки. Это может улучшить пользовательский опыт и ускорить процесс.

2. Оптимизация нагрузки на сервер: Предварительная проверка на стороне клиента помогает фильтровать некорректные данные еще до их отправки на сервер. Это позволяет снизить нагрузку на сервер и уменьшить объем обработки неправильных данных на стороне сервера.

3. Уменьшение ошибок: Проверка на стороне клиента позволяет предотвращать отправку неправильных данных на сервер и уменьшает вероятность возникновения ошибок при обработке данных на сервере.

Однако важно отметить, что проверка на стороне клиента может быть обходной или игнорироваться злоумышленниками, поэтому всегда следует выполнять также проверку данных на стороне сервера для обеспечения безопасности и целостности системы.

**55. Библиотека классов** - это набор готовых классов и методов, которые можно использовать для разработки приложений. Она предоставляет возможность повторного использования кода, упрощает разработку и улучшает поддерживаемость программного обеспечения.

Доступ к членам класса в библиотеке классов может быть разным. В общем случае, доступ к членам класса контролируется модификаторами доступа, такими как public, protected, private. Public позволяет получить доступ к членам класса из любого места кода, protected - только из класса и его потомков, private - только из самого класса.

**56. Язык Ассемблера** - это низкоуровневый язык программирования, который предоставляет прямое управление аппаратурой компьютера. Он использует мнемонические коды для представления команд процессора и адресов памяти.

Основная возможность языка Ассемблера - это полный контроль над аппаратурой компьютера, что позволяет максимально оптимизировать код и достичь высокой производительности. Кроме того, язык Ассемблера обеспечивает прямое взаимодействие с регистрами процессора и памятью.

Преимуществами языка Ассемблера являются:

- Высокая производительность и эффективное использование ресурсов компьютера;

- Полный контроль над аппаратурой;

- Возможность использования специфических инструкций и оптимизации.

Недостатки языка Ассемблера:

- Сложность и трудоемкость программирования;

- Отсутствие переносимости между различными аппаратурными платформами;

- Низкая читаемость и поддерживаемость кода.

**57. Сегментная адресация** - это система адресации в компьютерных системах, которая позволяет адресовать данные, используя комбинацию сегментного и смещенного адресов. В данной системе адресация происходит не только по одному целому числу, но и по сегменту, что позволяет увеличить размер доступной памяти и легче организовать работу с данными.

Примеры применения сегментной адресации:

- Виртуальная память: сегментация позволяет эффективно использовать ограниченные ресурсы физической памяти, разбивая ее на сегменты различных размеров и загружая в память только те сегменты, которые нужны в данный момент.

- Защита памяти: сегментация позволяет выделить отдельные сегменты памяти для различных процессов или пользователей, обеспечивая защиту от несанкционированного доступа.

- Исполнение кода: сегментация может использоваться для хранения и выполнения кода программы, где каждый сегмент содержит определенный тип инструкций или данных.

**58. Дизассемблер** - это программа, которая преобразует машинный код программы в читаемый человеком вид. Он позволяет анализировать и понимать, как работает программа, и может быть полезен в отладке, обратной разработке или изучении программного обеспечения.

Возможности дизассемблера:

- Преобразование машинного кода в ассемблерный код: дизассемблер может преобразовывать машинный код в код на языке ассемблера, что позволяет легко прочитать и понять код программы.

- Анализ исполняемых файлов: дизассемблер может анализировать исполняемые файлы и извлекать информацию о функциях, переменных, константах и других элементах программы.

- Отладка: дизассемблер может помочь в отладке программы, позволяя разработчику анализировать состояние памяти, значения регистров и последовательность выполнения инструкций.

Преимущества дизассемблера:

- Позволяет понять работу сложных программных систем и изучить код, написанный другими разработчиками.

- Позволяет проанализировать и исследовать вредоносные программы или другое вредоносное ПО.

- Используется в обратной разработке для восстановления исходного кода программы.

Недостатки дизассемблера:

- Машинный код может быть сложным для понимания и требует хорошего знания ассемблера и архитектуры процессора.

- Полная декомпиляция программы может быть затруднена из-за потери некоторой информации при компиляции или оптимизации.

- Дизассемблер может быть подвержен ошибкам при анализе исполняемых файлов с нестандартной структурой или защитой от обратной разработки.

**59. Мониторинг работоспособности системы контроля версий: назначение и инструменты мониторинга.**

Мониторинг работоспособности системы контроля версий (СКВ) имеет критическое значение для обеспечения эффективной и надежной разработки программного обеспечения. Назначение мониторинга состоит в том, чтобы непрерывно отслеживать производительность и стабильность СКВ, обнаруживать и устранять проблемы, связанные с доступностью, нагрузкой и безопасностью.

Для мониторинга работоспособности системы контроля версий могут использоваться различные инструменты. Некоторые из них включают:

1. Метрики производительности: мониторинг производительности СКВ, такой как время выполнения операций, использование памяти и процессора, может помочь выявить узкие места и оптимизировать работу системы.

2. Мониторинг доступности: система мониторинга может проверять доступность СКВ, осуществляя пинги или отправляя запросы на сервер. Это позволяет своевременно обнаруживать проблемы с доступностью и предотвращать длительные простои.

3. Мониторинг безопасности: следует отслеживать любые потенциальные уязвимости или атаки на СКВ, используя механизмы безопасности, такие как регистрация всех действий пользователей и анализ журналов для обнаружения подозрительной активности.

4. Уведомления и оповещения: система мониторинга должна быть настроена на отправку уведомлений и оповещений в случае обнаружения проблем или аварийных ситуаций, чтобы разработчики могли оперативно реагировать и устранять проблемы.

**60. Коллективная разработка: характеристика, структура.**

Коллективная разработка - это методология разработки программного обеспечения, в которой несколько разработчиков совместно работают над одним проектом, используя систему контроля версий и совместный доступ к репозиторию. Целью коллективной разработки является повышение эффективности и качества разработки путем распределения задач и интеграции вклада различных участников.

Структура коллективной разработки может варьироваться в зависимости от размера команды и конкретного проекта, но обычно включает в себя следующие элементы:

1. Репозиторий: центральное место, где хранится код проекта и с которым все участники могут синхронизироваться и вносить свои изменения.

2. Ветвление и слияние: разработчики создают различные ветки для работы над своими задачами, после чего вносят свои изменения обратно в главную ветку проекта путем слияния.

3. Код-ревью: участники команды проверяют код других разработчиков, обсуждают его и предлагают улучшения или исправления.

4. Итеративность: разработка основывается на коротких итерациях, где задачи и прогресс регулярно обновляются и обсуждаются всей командой.

5. Совместное общение: команда использует коммуникационные инструменты, такие как чаты, форумы и системы отслеживания ошибок, для общения и обмена информацией.

Коллективная разработка позволяет быстрее достигать результата, повышает качество и позволяет распределить задачи в соответствии с компетенциями участников команды.

**61. Проверка сервера: характеристика, принцип работы, особенности.**

Проверка сервера – это процесс, когда осуществляется проверка работоспособности и доступности сервера. Это важная задача для обеспечения стабильной работы сети и предотвращения возможных проблем или сбоев.

Характеристики проверки сервера могут включать следующие аспекты:

- Проверка соединения: проверяется, доступен ли сервер и может ли клиент успешно установить связь с ним.

- Проверка производительности: измерение нагрузки на сервер, скорости передачи данных и времени отклика.

- Проверка безопасности: анализ уязвимостей, обнаружение атак и проверка конфигурации сервера с целью защиты от несанкционированного доступа и вредоносного программного обеспечения.

- Проверка доступности сервисов: проверка работоспособности различных сервисов на сервере, таких как веб-сайты, электронная почта, базы данных и другие.

Принцип работы проверки сервера обычно основан на отправке запросов от клиента к серверу и анализе полученных ответов. Для проверки доступности сервера используется различные протоколы, такие как ICMP (Internet Control Message Protocol) для проверки наличия соединения, HTTP для проверки веб-сервера и другие.

Особенности проверки сервера:

- Регулярность: проверка сервера рекомендуется проводить в регулярном режиме, чтобы обнаружить проблемы и аномалии как можно раньше.

- Автоматизация: проверка сервера может быть автоматизирована с использованием специальных программных инструментов или сервисов, что упрощает процесс и позволяет получить быстрый результат.

- Мониторинг: результаты проверки сервера могут быть интегрированы в системы мониторинга, чтобы оперативно уведомлять администраторов о возможных проблемах или отклонениях от нормы.

- Логирование: результаты проверки сервера обычно записываются в журнал для дальнейшего анализа или отслеживания изменений со временем.

**62. Проверка клиента, характеристика, принцип работы**.

Проверка клиента – это процесс, при котором осуществляется проверка работоспособности и настройки клиентского устройства, подключенного к сети, для обеспечения надежной связи и устранения возможных проблем.

Характеристики проверки клиента могут включать следующие аспекты:

- Проверка подключения: проверка наличия и стабильности подключения клиента к сети.

- Проверка настроек сети: анализ и проверка настроек IP-адреса, DNS, шлюза по умолчанию и других параметров.

- Проверка пропускной способности: измерение скорости передачи данных от клиента до сервера и обратно.

- Проверка безопасности: проверка наличия антивирусного ПО, обновленных программ и настроек безопасности для защиты клиента от вредоносных атак или утечек данных.

Принцип работы проверки клиента часто основывается на отправке запросов от клиента к серверу и анализе полученных ответов. Для проверки доступности клиента можно использовать различные методы, такие как пинг (отправка ICMP-запросов), HTTP-запросы и другие протоколы в зависимости от цели проверки.

Однако следует отметить, что проверка клиента может иметь другие особенности в зависимости от конкретной сетевой среды или типа проверяемых устройств. Например, в безопасных сетях могут использоваться специальные протоколы или сертификаты для проверки легитимности клиента перед разрешением доступа.

**63. Отличительные особенности IDE (интегрированной среды разработки) C++/C#:**

1. Встроенная отладка: IDE C++/C# обеспечивает мощный отладчик, позволяющий запускать код по шагам, подсматривать значения переменных и исправлять ошибки.

2. Автодополнение и подсказки: IDE обладает функцией автодополнения, которая предлагает варианты кода на основе того, что вы уже напечатали. Также она предоставляет подсказки по доступным методам и свойствам.

3. Интеграция с системами контроля версий: IDE поддерживает интеграцию с такими системами, как Git, что упрощает работу с кодом в команде.

4. Удобная навигация по коду: IDE C++/C# предлагает функции навигации по коду, позволяющие быстро переходить к определенным классам, методам или свойствам.

5. Интегрированная справка: Идеальная среда разработки обычно предлагает встроенную справку, которая содержит информацию о синтаксисе языка, доступных классах и методах, а также примеры использования.

**64. Отличительные особенности IDE Java:**

1. Компиляция в реальном времени: IDE Java сканирует ваш код на наличие ошибок и предупреждений во время его написания, что позволяет быстро обнаруживать проблемы и устранять их.

2. Мощный отладчик: IDE Java предоставляет отладчик с широкими функциональными возможностями, такими как шаг за шагом выполнение кода, просмотр значений переменных и анализ стека вызовов.

3. Поддержка статического анализа кода: Некоторые IDE Java имеют встроенные инструменты статического анализа кода, способные обнаружить потенциальные проблемы или несоответствия стандартам кодирования.

4. Интеграция с серверами приложений: IDE Java обычно поддерживает интеграцию со сторонними серверами приложений, что упрощает разработку, тестирование и развертывание Java-приложений.

5. Поддержка фреймворков и библиотек: IDE Java позволяет легко интегрировать и использовать различные фреймворки и библиотеки Java, обеспечивая улучшенную производительность разработки.

**65. Отключение JavaScript на стороне клиента: особенности.**

Отключение JavaScript на стороне клиента - это процесс отключения выполнения JavaScript-кода на веб-сайте веб-браузером пользователя. Возможны несколько основных особенностей этого процесса:

1. Отключение функциональности: JavaScript является мощным языком программирования, который используется для добавления интерактивности и динамических функций на веб-сайты. При его отключении на стороне клиента, функции, зависящие от JavaScript, могут быть отключены или работать неправильно.

2. Отключение валидации данных: JavaScript часто используется для валидации данных на веб-сайтах, чтобы предотвратить отправку неправильных данных на сервер. Если JavaScript отключен, то эта валидация не будет выполняться, что может привести к проблемам с неправильными данными на сервере.

3. Отключение AJAX-запросов: JavaScript является ключевым инструментом для выполнения асинхронных запросов на сервер с использованием технологии AJAX. При отключении JavaScript возможность отправлять и обрабатывать AJAX-запросы будет потеряна.

4. Отключение аналитики и отслеживания: Многие веб-сайты используют JavaScript для сбора аналитических данных и отслеживания пользователей. Если JavaScript отключен, эта функциональность также будет недоступна.

5. Попытка отключить уязвимости: JavaScript иногда может быть использован для проведения атак на веб-сайты, таких как внедрение вредоносного кода или кросс-сайтовый сценарий (XSS). Отключение JavaScript может снизить уязвимости веб-сайта к таким атакам.

**66. Команды github: git init, git branch, git push/pull характеристика, основные действия.**

- `git init` - команда используется для создания нового репозитория Git. После выполнения этой команды, в текущей директории будет создан пустой репозиторий Git, который будет использоваться для отслеживания изменений в файлах.

- `git branch` - команда используется для создания, переименования или удаления веток в Git. Она также позволяет просматривать список существующих веток.

- `git push` - команда используется для отправки ("пуша") локальных изменений в удаленный репозиторий Git. Это позволяет вам обновить удаленный репозиторий с вашими последними изменениями.

- `git pull` - команда используется для получения ("пулла") обновлений из удаленного репозитория Git и объединения их с вашим локальным репозиторием. Этот процесс позволяет вам получить последние изменения, сделанные другими разработчиками, и синхронизировать свой репозиторий с удаленным.

Основные действия, связанные с этими командами, включают создание репозитория, создание и переключение веток, добавление файлов в индекс, фиксацию изменений, отправку локальных изменений в удаленный репозиторий (push), получение обновлений из удаленного репозитория и их объединение с локальным (pull).

**67. Приемочное тестирование** - это вид тестирования программного обеспечения, проводимый с целью проверки соответствия разработанного продукта заданным требованиям и ожиданиям конечного пользователя.

Основные понятия приемочного тестирования:

1. План приемочного тестирования - документ, в котором описывается подход к проведению приемочного тестирования, определяются его цели, задачи, ожидаемые результаты, а также ресурсы и расписание для выполнения тестирования.

2. Функциональные требования - это спецификация функциональных возможностей программного продукта, которые должны быть проверены во время приемочного тестирования.

3. Вариант использования - это сценарий или последовательность действий, которую пользователь проделывает с программным продуктом, чтобы достичь определенной цели. Во время приемочного тестирования тестируется поведение программы при различных вариантах использования.

Элементы приемочного тестирования:

1. План тестирования - подробное описание проведения приемочного тестирования, включающее в себя задачи тестирования, расписание, список требований и вариантов использования, список используемого оборудования и инструментов тестирования.

2. Тестовые сценарии - это последовательности действий, которые выполняются во время тестирования для проверки соответствия программного продукта функциональным требованиям.

3. Тестовые данные - это входные данные, которые используются для проведения тестирования программного продукта.

Результат приемочного тестирования - это отчет, который содержит информацию о выполненных тестовых сценариях, ошибках, обнаруженных в процессе тестирования, и общую информацию о соответствии программного продукта требованиям.

**68. Выходное тестирование** - это вид тестирования программного обеспечения, направленный на проверку корректности работы программы после ее модификации или исправления ошибок.

Основные понятия выходного тестирования:

1. Изменения - это изменения в программном коде или конфигурации программного продукта, которые были внесены для исправления ошибок или улучшения функциональности.

2. Ошибки - это несоответствия или неправильное поведение программного продукта, которые обнаружены в процессе тестирования.

3. Ожидаемое поведение - это ожидания относительно работы программного продукта после его модификации. Во время выходного тестирования проверяется, что изменения не привели к нежелательным изменениям в работе программы.

Элементы выходного тестирования:

1. Измененные тестовые сценарии - это обновленные или добавленные тестовые сценарии, которые учитывают изменения в программном продукте.

2. Тестовые данные - это входные данные, которые используются для проведения тестирования после модификации программного продукта.

3. Результаты тестирования - это отчет о выполнении тестовых сценариев и о найденных ошибках в процессе выходного тестирования.

Результат выходного тестирования - это подтверждение корректной работы программы после ее модификации или исправления ошибок.

**69. Основные элементы для тестирования ПО и их результаты:**

1. Планирование тестирования: результат - план тестирования, который включает в себя описание тестовых сценариев, ресурсы и расписание.

2. Проектирование тестов: результат - тестовая документация, включающая тестовые случаи, условия и ожидаемые результаты.

3. Создание тестовых данных: результат - набор данных, необходимых для проведения тестов.

4. Выполнение тестов: результат - запись результатов тестов и обнаруженных дефектов.

5. Отчет о тестировании: результат - отчет, содержащий информацию о проведенных тестах, обнаруженных дефектах и рекомендациях по их исправлению.

**70. Атрибуты и назначение команд git rebase, git clone и git log**:

1. git rebase: команда git rebase используется для переоснования (rebase) изменений из одной ветки на другую. Основные атрибуты команды git rebase включают:

- -i (--interactive): позволяет выполнять интерактивное переоснование, включая возможность редактирования коммитов и их порядка.

- -m (--merge): указывает git на выполнение merge при переосновании, а не простого применения изменений.

2. git clone: команда git clone используется для создания копии удаленного репозитория на локальном компьютере. Основные атрибуты команды git clone включают:

- --branch <branch\_name>: позволяет склонировать только указанную ветку.

- --depth <depth>: указывает максимальную глубину истории, которую нужно скачать.

3. git log: команда git log используется для просмотра информации о коммитах в репозитории. Основные атрибуты команды git log включают:

- --oneline: выводит краткую информацию о каждом коммите на одной строке.

- --author <author\_name>: фильтрует коммиты только по указанному автору.

Применение данных команд зависит от конкретных потребностей разработчика и команды. Например, команда git rebase может использоваться для объединения истории коммитов в более логичный порядок, а команда git clone - для получения локальной копии удаленного репозитория для работы над проектом. Команда git log может быть использована для анализа истории коммитов и поиска конкретных изменений.