**Вопросы сейчас не по порядку! Переставлены, чтобы не съезжали на другие страницы и можно было удобно печатать!**

**ПРОЧИТАТЬ:**

* Распределитесь в табличке снизу самостоятельно! При этом обязательно возьмите как минимум один свой доклад.
* Все, что останется в конце, я раскидаю как нибудь по таблице. В приоритете те, кто делал меньше до этого.
* Дедлайн в 17:00, чтобы можно было прочитать и что-то подправить. Если почему то не успеваете, то скажите, но я гулял и тд это не причины)
* Старайтесь в первую очередь писать то, что про ваш вопрос говорил дед. Пишем только важное, но при этом нужно написать все, что требуется по вопросу. Лучше сделать карточку чуть длиннее, чем бесполезную.
* После того как сделаете карточку, выделите её в изначальном распреде зеленым и напишите в новом её номер рядом с собой, чтобы было понятно, что вы это сделали
* Наши доклады лежат тут: [Доклады\_Тестирование\_3курс\_2025](https://drive.google.com/drive/folders/17dANSrvr7ya3HnSoszLRHtrCOsGER6Oe)

Форматирование:

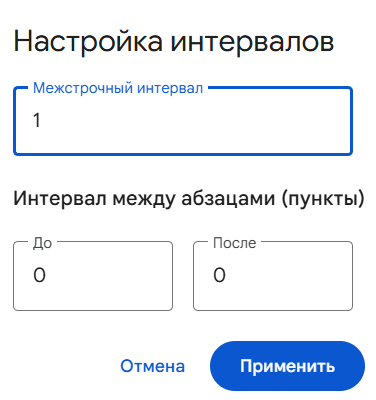
шрифт Montserrat

Обычный текст - размер шрифта 5

Название билета - размер шрифта 6 и жирный

Цвет текста - черный, в крайнем случае синий (всякий желтый не будет видно в чб печати)

Интервалы (важно тыкнуть применить, потому что когда открываешь посмотреть он и так по умолчанию 00 типо стоит, но это обманка):



Старый распред на доклады

| 1. Основные задачи тестирования, классификация тестирования | Богдан |
| --- | --- |
| 1. Стратегии тестирования | Луговенко |
| 1. Принципы тестирования | Ложкина |
| 1. Методы тестирования | Богданова |
| 1. Инспекция кода | Яшнова |
| 1. Сквозные просмотры | Астафьев |
| 1. Проверка за столом | Яшнова |
| 1. Статическое тестирование | Гаар |
| 1. Тестирование методом белого ящика | Богданова |
| 1. Тестирование методом черного ящика | Якунин / Плужникова |
| 1. Метод причинно-следственных диаграмм | Лобанов |
| 1. Модульное тестирование, инкрементное тестирование | Луговенко |
| 1. Нисходящее тестирование | чурова / Плужникова |
| 1. Восходящее тестирование | Черепнов |
| 1. Высокоуровневое тестирование | Якунин / Богдан |
| 1. Функциональное тестирование | Шклярова |
| 1. Системное тестирование | Гаар |
| 1. Приемочное тестирование, тестирование установки | Губковский / Емешкин |
| 1. Планирование и контроль тестирования | Ромашко / Золоев |
| 1. Критерии завершения тестов | Ромашко / Золоев |
| 1. Методы отладки, метод грубой силы | Лобанов |
| 1. Тестирование удобства пользователя | Губковский |
| 1. Гибкое тестирование | чурова / Григорьев |
| 1. Основные положения стандарта ISO/IEC 12207, требования к тестированию верхнего и нижнего уровня. | Тищенко |
| 25. Верификация программного обеспечения | Черепнов / Емешкин |
| 26. Валидация программного обеспечения | Шклярова / Шихалев |
| 27. Методы тестирования ПО для заинтересованных сторон, ГОСТ Р ИСО/МЭК 25010-2015 | Тищенко / Шихалев |
| 28. Тестирование ПО систем реального времени | Салимли / Михалец |
| 29. Тестирование ПО автономных систем | Астафьев / Михалец |
| 30. Основные задачи технологии тестирования Фаззинг | Салимли |
| 31. Требования к сертификации информационных систем, приказ ФСТЭК №76 от 02.06.2020 | Григорьев |
| 32. Методы автоматизированного тестирования | Ложкина |

**Распределение:**

| Имя | Вопросы |
| --- | --- |
| лизОК | 4, 9 |
| димуля я | 10, 28 |
| полик | 2, 12 |
| юля | 13 |
| сеня | 1, 15, 20 |
| настя | 3, 32 |
| XUXA | 16 |
| ДИДИ | 5,7 |
| Игорь | 6, 29 |
| Сочурка | 23 |
| Паша | 11,21 |
| Влад | 8, 17, 19 |
| Дима Г | 18,22 |
| Артём | 24 |
| Максим Ч | 14 |
| Леша | 26, 27 |
| Макс Е | 25 |

**Легенда:**

* Тема готова
* Тема делается
* Сомнительно, но окей

| **3. Принципы тестирования программного обеспечения** Для систематизации и повышения результативности тестирования разработаны принципы.  Эти принципы помогают тестировщикам, минимизировать риски и повысить качество программного обеспечения. **Основные принципы тестирования:** **Принцип 1:** Определение ожидаемых входных данных и результатов  Каждый тест должен включать четкое описание ожидаемых выходных данных. Без заранее определенных результатов тестировщик может ошибочно принять некорректные данные за правильные из-за психологической склонности интерпретировать результаты в пользу успеха. Поэтому любой тест должен включать две составляющие:  ■ Описание входных данных программы;  ■ точное описание корректных выходных данных для каждого набора входных данных.  **Принцип 2:** Избегание тестирования собственных программ Программисты не должны тестировать собственный код, так как они склонны подсознательно избегать выявления ошибок и могут опираться на неверные предположения, заложенные при разработке. Пример: Программист, создавший модуль регистрации, может не проверить сценарий с некорректным форматом электронной почты, считая свою реализацию правильной. Значение: Независимое тестирование увеличивает вероятность обнаружения ошибок и повышает качество проверки.  **Принцип 3:** . Компания-разработчик не должна тестировать собственное программное обеспечение  Компания-разработчик не должна тестировать собственное программное обеспечение, так как экономические и психологические факторы, такие как давление сроков и бюджета, могут снижать объективность. Пример: Компания, сосредоточенная на соблюдении сроков релиза, может недооценить важность тестирования редких сценариев, что приведет к пропуску ошибок. Передача тестирования независимой стороне способствует более тщательной проверке и повышает надежность продукта.  **Принцип 4**: Детальное изучение результатов  Каждый тест требует тщательного анализа выходных данных. Невнимательность при проверке результатов может привести к пропуску очевидных ошибок. Пример: При тестировании формы оплаты тестировщик может не заметить, что транзакция завершилась успешно, но данные клиента не сохранились, если не изучить все выходные данные. Тщательный анализ результатов увеличивает вероятность обнаружения ошибок.  **Принцип 5**: Тесты должны включать как ожидаемые, так и непредсказуемые входные данные, чтобы выявить ошибки, связанные с нестандартным использованием программы. Пример: Тестирование поля ввода возраста должно проверять корректные значения (например, 25) и некорректные (например, отрицательное число или буквы).Проверка некорректных данных улучшает ее надежность.  **Принцип 6:** Проверка, делает ли программа то, что должна, составляет лишь ползадачи; вторая половина задачи — выяснить, не делает ли программа того, чего не должна делать  Тестирование должно не только подтверждать выполнение функций программы, но и проверять, не выполняет ли она нежелательные действия. Пример: Программа для управления складом может корректно обновлять данные о запасах, но ошибочно удалять записи о предыдущих транзакциях. Выявление побочных эффектов обеспечивает безопасность и стабильность системы.  **Принцип 7**: Сохранение тестов  Тесты не следует удалять, за исключением случаев, когда программа предназначена для одноразового использования. Сохраненные тесты позволяют проводить регрессионное тестирование после внесения изменений. Пример: После добавления новой функции в приложение сохраненные тесты помогают убедиться, что обновление не нарушило существующую функциональность.Регрессионное тестирование минимизирует риск появления новых ошибок и поддерживает стабильность программы.  **Принцип 8**:Не приступайте к планированию тестов, заранее настраиваясь на то, что ошибки не будут обнаружены Тестирование не должно планироваться с предположением, что ошибок нет. Такой подход снижает эффективность процесса и увеличивает вероятность пропуска ошибок. Пример: Тестировщик, уверенный в стабильности модуля, может не проверить редкие сценарии, пропустив ошибку, появившуюся после последнего обновления. Настрой на поиск ошибок делает тестирование более деструктивным и результативным.  **Принцип 9:** Вероятность того, что в некоторой части программы остались необнаруженные ошибки, прямо пропорциональна количеству уже обнаруженных там ошибок  Вероятность наличия необнаруженных ошибок выше в тех частях программы, где уже найдено больше ошибок. Ошибки склонны группироваться в определенных модулях.: Концентрация усилий на проблемных модулях оптимизирует тестирование и повышает качество продукта.  **Принцип 10:** Творческий подход к тестированию  Тестирование — это интеллектуальный и творческий процесс, требующий нестандартного мышления для разработки эффективных тестовых сценариев. Творческий подход позволяет выявлять скрытые ошибок, недоступные стандартным методам тестирования.  Соблюдение этих принципов позволяет тестировщикам выявлять ошибки, минимизировать риски и создавать надежное программное обеспечение. | **4. Методы тестирования**  **1.** Долгое время большинство программистов считали, что программы пишутся исключительно для машинной обработки и не предназначены для чтения человеком. Единственным способом тестирования считалось выполнение кода на компьютере. Ситуация начала меняться в начале 1970-х годов, когда разработчики начали осознавать ценность просмотра кода как элемента процесса тестирования и отладки. **Ручное тестирование** — это процесс проверки программного кода без использования компьютера, непосредственно человеком.  **2. Методы ручного тестирования:**  **2.1 Инспекция кода** — метод обнаружения несоответствия спецификации через анализ (чтение) кода группой специалистов. **Состав группы:** 1) Координатор (не автор программы): - раздает материалы участникам; - составляет план заседания; - проводит заседание; - записывает все найденные несоответствия спецификации; - проверяет их устранение. 2) Программист — автор программы. 3)Проектировщик программы (если отличается от программиста). 4)Специалист по тестированию: - хорошо разбирается в методах тестирования ПО;- знает распространенные типы несоответствий спецификации. **Процедура проведения:** 1. За несколько дней до заседания координатор раздаёт листинг программы и спецификацию проекта всем участникам. 2. Во время заседания: - программист объясняет логику работы программы; - участники задают вопросы, направленные на выявление возможных несоответствий спецификации; - проводится анализ кода с использованием контрольных списков несоответствий спецификации. 3. После заседания: - программист получает список найденных несоответствий спецификации; - при необходимости проводится повторная инспекция; - найденные несоответствия спецификации добавляются в контрольный список для будущих проверок. **Особенности:**- Основная цель — обнаружение, а не исправление несоответствий спецификации. - Если ошибка мелкая, группа может предложить изменения. **Преимущества:** - Программист получает стороннюю оценку своего стиля и алгоритмов - Участники получают опыт через изучение несоответствий спецификации и подходов коллег.  **2.2 Сквозные просмотры** — метод обнаружения несоответствия спецификации через анализ (чтение) кода группой специалистов. Вместо простого чтения кода программы или ее проверки с помощью контрольного списка несоответствий спецификации, участники заседания “играют в компьютер”, вручную прогоняют тесты. **Состав группы:** 1) Координатор. 2) Секретарь (записывает несоответствия спецификации). 3) Тестировщик. 4) Другие участники могут быть: эксперт по языку программирования; начинающий программист; человек, который будет обслуживать программу; представитель другого проекта или той же команды. **Процедура:** 1. За несколько дней материалы раздаются участникам. 2. Во время заседания: - тестировщик предоставляет тестовые данные; - участники "прогоняют" тесты вручную, отслеживая состояние программы на бумаге или доске. **Особенности:** - Тесты должны быть простыми и ограниченными по количеству (ввиду медлительности ручного выполнения).  **2.3 Проверка за столом -** может рассматриваться как индивидуальная версия инспекции или сквозного просмотра. **Особенности:** - Выполняется одним человеком. - Может включать чтение кода, использование контрольных списков, прогон тестовых данных. **Недостатки:** - Неупорядоченный процесс. - Эффективность снижена, если проверку выполняет автор программы. - Лучше всего работает, когда проверку выполняет другой человек (например, программисты меняются программами для взаимопроверки). - Групповая работа (инспекции и сквозные просмотры) эффективнее, чем одиночная проверка.  **2.4 Рецензирование** - это процедура анонимной оценки общих характеристик качества, обслуживаемости, расширяемости, удобства использования и ясности программного обеспечения, не связано напрямую с тестированием, но также основано на чтении кода. **Цель:** получить стороннюю оценку качества программы. **Участники:** 1) Администратор (выбирается из числа программистов). 2)Группа рецензентов (6–20 человек), специализирующихся в одной области. **Процедура:** 1. Каждый участник предоставляет две свои программы: лучшую и худшую по его мнению. 2. Программы случайным образом распределяются между рецензентами. 3. На каждую программу тратится около 30 минут. 4. Заполняется оценочная анкета по десятибалльной шкале, примеры вопросов: - Легко ли понять программу? - Хорошо ли реализованы концепции проектирования?- Легко ли модифицировать программу? - Испытываете ли удовлетворение от её написания? 5. Участник получает анкеты с оценками своих программ. - Предоставляется статистическая сводка рейтинга программ и точности оценок рецензента.  **Заключение** - Ручное тестирование эффективно выявляет определённые типы несоответствий спецификации (например, использование неинициализированных переменных). - Компьютерное тестирование лучше справляется с другими типами несоответствий спецификации (например, деление на ноль). - Эти два подхода взаимно дополняют друг друга, и применение только одного из них снижает общую эффективность тестирования.  **Сводный список контрольных вопросов для выявления ошибок при инспекции кода ( пример какие могут быть вопросы):**  **Обращения к данным:** 1. Используются ли переменные с неинициализированными значениями? 2. Выходят ли индексы за границы массива? 3. Используются ли нецелочисленные индексы? **Вычисления:** 1. Участвуют ли в вычислениях ненарифметические переменные? 2. Выполняются ли вычисления с использованием данных разных типов? 3. Выполняются ли вычисления с переменными разной длины? 4. Присваиваются ли переменным значения типа данных большего размера? 5. Возможно ли деление на нуль? **Описания данных:** 1. Все ли переменные объявлены? 2. Правильно ли определены размеры, типы и классы памяти? 3. Согласуется ли инициализация с классом памяти? 4. Имеются ли переменные с похожими именами? **Сравнения:**1. Есть ли попытки сравнения несопоставимых переменных? 2. Есть ли попытки сравнения переменных разного типа? 3. Корректно ли используются операторы сравнения? **Передача управления:** 1. Будет ли завершен каждый цикл? 2. Будет завершена программа? 3. Есть ли цикл, который может не выполниться из-за входных условий? **Ввод-вывод:** 1. Согласуются ли спецификации формата с инструкциями ввода-вывода? 2. Открываются ли файлы перед обращением к ним? 3. Обрабатываются ли ошибки ввода-вывода? **Программные интерфейсы:** 1. Совпадает ли число аргументов, передаваемых вызываемым модулям, с числом ожидаемых параметров? 2. Совпадают ли единицы измерения аргументов, передаваемых вызываемым модулям, с единицами измерения ожидаемых параметров? |
| --- | --- |

| **5. Инспекция кода**  Инспекция кода — это набор процедур и методик обнаружения ошибок путем анализа (чтения) кода группой специалистов.  Группа инспектирования кода  Обычно в состав группы входят четыре человека, один из которых играет роль координатора. Координатор должен быть квалифицированным программистом, но не ав­тором тестируемой программы, детальное знание которой от него не требу­ется. В его обязанности входит следующее:  ■ раздача материалов участникам заседания инспекционной группы и составление плана его проведения;  ■ проведение заседания;  ■ запись всех обнаруженных ошибок;  ■ последующая проверка того, что обнаруженные ошибки устранены.  Вторым участником группы является программист, а остальными — проектировщик программы (если это не сам программист) и специалист по тестированию. Последний должен не только хорошо ориентироваться в методах тестирования ПО, но и знать наиболее распространенные типы ошибок.  *Порядок проведения заседаний по инспектированию кода*  За несколько дней до заседания координатор раздает листинг программы и спецификацию проекта всем участникам группы. Это делается для того, чтобы к моменту проведения заседания участники успели ознакомиться с необходимыми материалами. Инспекционное заседание проводится следующим образом.  **1.** Программист рассказывает присутствующим о логике работы про­граммы, подробно останавливаясь на каждой инструкции. В это вре­мя другие участники заседания должны задавать ему вопросы, кото­рые поспособствуют выявлению возможных ошибок. Вполне вероят­но, что по ходу дела часть ошибок обнаружит сам программист, а не остальные участники группы. Иными словами, одно лишь чтение про­граммы вслух при публичном объяснении принципов ее работы ста­новится весьма эффективным методом обнаружения программных ошибок.  **2.** Участники заседания анализируют код программы, сверяясь с состав­ленными на основе имеющегося опыта контрольными списками наи­более распространенных ошибок (подобный список обсуждается в следующем разделе).  Координатор обязан направлять дискуссию в конструктивное русло и следить за тем, чтобы участники концентрировали свое внимание не на устранении ошибок, а на их обнаружении (соответствующие исправления программист внесет самостоятельно после заседания).  По окончании заседания программисту передастся список найденных ошибок. Если этот список достаточно длинный или устранение ошибок требует внесения существенных изменений в программу, координатор мо­жет принять решение о повторной инспекции кода после того, как ошибки будут исправлены. Список анализируется и после разбиения ошибок на ка­тегории дополняет имеющийся контрольный список, что позволяет повы­сить эффективность будущих инспекций кода.  Время и место проведения инспекции должны быть спланированы так, чтобы никакие внешние факторы не могли мешать работе заседания. Опти­мальная длительность таких заседаний составляет от полутора до двух ча­сов. С увеличением их длительности продуктивность может резко падать. В большинстве случаев инспекция кода осуществляется со средней скоро­стью 150 строк в час. Поэтому для больших программ следует проводить не­сколько заседаний, на каждом из которых инспекции подвергается один или несколько модулей (подпрограмм).  Для того чтобы инспекция была эффективной, участники группы тести­рования должны выработать правильное отношение к этому процессу. Если программист воспринимает инспектирование своей программы как деятель­ность, направленную против него лично, и занимает оборонительную по­зицию, то процесс инспектирования не будет эффективным.  Важной частью процесса инспектирования кода является использование контрольных списков, облегчающих обнаружение наиболее распростра­ненных ошибок в программах. Такие вопросы можно разделить на 7 основных групп:(в скобках примеры вопросов)   | **Ошибки обращения к данным**(Не выходят ли значения индексов за установленные пределы при об­ращении к массиву?)  **Ошибки описания данных**( Правильно ли определяются размер и тип каждой переменной?)  **Ошибки вычислений**(Возможно ли обращение делимого в нуль при выполнении операции деления?))  **Ошибки ввода-вывода(**Все ли файлы закрываются после обращения к ним?) | **Ошибки сравнения**( Сравниваются ли в программе переменные различных типов, напри­мер символьная строка с адресом, датой или числом?)  **Ошибки в управляющей логике(** Сможет ли программа, модуль или подпрограмма в конечном итоге за­вершиться?)  **Ошибки программных интерфейсов**(Совпадают ли атрибуты (например, тип и размер) параметров с атри­бутами соответствующих аргументов?) | | --- | --- | | **6. Сквозные просмотры**  Как и инспекция, сквозной просмотр кода представляет собой совокупность процедур и методов обнаружения ошибок в коде путем его просмотра участниками группы тестирования.  Подобно инспекции, сквозной просмотр проводится в форме непрерывного заседания, длящегося не более 2-х часов. Число участников группы, выполняющей сквозной просмотр, составляет 3-5 человек. Один из них выполняет функции, аналогичные функциям координатора в группе инспектирования (из в.5: Координатор должен быть квалифицированным программистом, но не ав­тором тестируемой программы). Еще один человек играет роль секретаря и записывает все найденные ошибки, а третий участник — тестировщик. Мнения о том, кто должен входить в группу и стоит ли ее расширять до 5 человек, расходятся. Одним из этих людей должен быть программист, написавший программу. К числу других возможных кандидатур относятся:   * высококвалифицированный программист; * эксперт по используемому языку программирования; * начинающий программист (на точку зрения которого не повлияет предыдущий опыт); * человек, который в конечном счете будет обслуживать программу; * участник какого-либо другого проекта; * кто-нибудь из той же группы программистов, что и автор программы.   Процедура сквозного просмотра начинается точно так же, как и процедура инспекции: за несколько дней до заседания необходимые материалы раздают участникам, чтобы они могли заблаговременно ознакомиться с проверяемым программным обеспечением. Однако работа самого заседания организуется иначе. Вместо простого чтения кода программы или ее проверки с помощью контрольного списка ошибок участники заседания “играют в компьютер”. Тот, кто был назначен тестировщиком, приходит на совещание с предварительно записанными на листах бумаги тестами — представительными наборами входных (и ожидаемых выходных) данных для программы или модуля. Во время заседания участники “прогоняют в уме” каждый тест, т.е. подвергают тестовые данные обработке вручную в соответствии с логикой программы. Состояние программы (значения переменных) отслеживается на бумаге или на доске.  Тестов должно быть не слишком много, и они должны быть простыми по своей природе, потому что скорость ручного выполнения запрограммированных действий на много порядков меньше, чем скорость вычислений на компьютере. Следовательно, тесты сами по себе не играют критической роли; скорее они служат средством “разогрева” участников и основой для постановки вопросов программисту, касающихся логики проектирования и принятых допущений. В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.  Как и при инспекции кода, ключевую роль играет отношение участников к процессу. Объектом замечаний должна быть программа, а не программист. Ошибки не рассматриваются как признак профессиональной неподготовленности допустившего их человека, а считаются естественным проявлением трудностей.  Сквозные просмотры должны проходить так же, как и процесс инспектирования. Побочные эффекты, возникающие в ходе этого процесса (выявление подверженных ошибкам частей программы, обучение команды на основе анализа ошибок, стиля и методик программирования), характерны и для процесса сквозных просмотров.  *Главные отличия между инспекцией кода и сквозными просмотрами:*   * Методы обнаружения ошибок   Инспекция: Группа читает код и разбирает заранее подготовленные контрольные списки вопросов для обсуждения распространенных ошибок. Основной акцент сделан на анализе структуры и синтаксиса программы.  Сквозной просмотр: Группа имитирует работу компьютера («играет в компьютер»), прогоняя выбранные наборы тестовых данных вручную согласно логике программы. Основное внимание уделяется выполнению конкретных сценариев тестирования.   * Организация процесса   Инспекция: Регламентирована четкими этапами: подготовка материалов, распределение ролей (координатор, программист, тестировщик), выявление ошибок и их документирование.  Сквозной просмотр: Более гибкий подход, допускающий больший акцент на интерактивность и сотрудничество внутри группы. Акцент смещается на выполнение теста руками и обсуждения отдельных фрагментов программы.   * Цель и фокус внимания   Инспекция: выявить и зафиксировать максимальное количество ошибок путем анализа исходного текста программы и применения контрольных вопросов.  Сквозной просмотр: то же что и при инспекции, но дополнительно анализируется поведение программы в различных ситуациях благодаря ручному прогону тестов (подвергают тестовые данные обработке вручную в соответствии с логикой программы); анализ поведения программы помогает сформулировать вопросы касающиеся логики проектирования и принятых допущений. |
| --- | --- | --- | --- |
| **7.** **Проверка за столом**  **1. Определение:** Проверка за столом — это **метод ручного тестирования**, при котором **один человек** анализирует программный код без выполнения его на компьютере. Это индивидуальная версия инспекции кода или сквозного просмотра.  **2. Суть метода:**   * **Индивидуальный процесс:** Выполняется исключительно одним человеком (в отличие от групповых инспекций или сквозных просмотров). * **Ручной анализ:** Проверяющий вручную изучает код, логику программы и данные. * **Основные действия проверяющего:**   + **Чтение кода:** Последовательное изучение исходного кода.   + **Использование контрольных списков:** Сверка кода со списками типичных ошибок (ошибки данных, вычислений, логики, интерфейсов и т.д.).   + **"Прогон" тестовых данных:** Мысленное или на бумажное выполнение программы с конкретными тестовыми наборами данных ("игра в компьютер"). Проверяющий отслеживает состояние переменных и ход выполнения.   **3. Особенности и отличия от групповых методов:**   * **Неупорядоченность:** Процесс менее формализован и структурирован по сравнению с групповыми заседаниями. * **Отсутствие групповой динамики:** Нет коллективного обсуждения, мозгового штурма, здоровой конкуренции в поиске ошибок. * **Скорость и охват:** Позволяет проверить код быстрее, чем организация группового заседания, но глубина анализа может быть ниже.   **4. Ключевые недостатки:**   * **Низкая эффективность при самопроверке:** Если проверку выполняет **автор программы**, эффективность резко снижается из-за "профессиональной слепоты" (невозможности объективно увидеть свои ошибки - второй принцип тестирования). * **Относительно низкая продуктивность:** По сравнению с инспекциями и сквозными просмотрами, считается менее эффективным методом обнаружения дефектов. * **Отсутствие сторонней оценки:** Программист не получает обратной связи о стиле, алгоритмах или общих характеристиках качества кода от коллег.   **5. Рекомендации по повышению эффективности:**   * **Проверка другим человеком:** Наибольшая эффективность достигается, когда проверку выполняет **не автор кода**. Оптимальный вариант — **взаимная проверка** (программисты обмениваются программами). * **Использование структуры:** Четкое следование этапам (чтение, проверка по списку, прогон тестов) и использование контрольных списков. * **Фокус:** Концентрация на конкретных типах ошибок или модулях.   **6. Место в процессе тестирования:**   * **Дополнение к групповым методам:** Может использоваться для быстрой предварительной проверки модулей или исправлений перед групповой инспекцией. * **Альтернатива при ограниченных ресурсах:** При невозможности организовать групповое заседание (сроки, доступность людей). * **Взаимопроверка:** Как основной метод в паре программистов. * **Не замена:** Не является полноценной заменой ни компьютерному тестированию, ни групповым методам ручного анализа (инспекциям, сквозным просмотрам), а дополняет их.   **Заключение:** Проверка за столом — это полезный, но **наименее эффективный из описанных методов ручного тестирования**. Ее главное преимущество — скорость и простота организации для индивидуальной проверки, особенно если код анализирует не его автор. Однако для глубокого анализа и обнаружения сложных дефектов предпочтительнее групповые методы (инспекции, сквозные просмотры) или автоматизированное тестирование. | **8. Статическое тестирование**  Согласно ГОСТ Р 56920-2024, **статическое тестирование** — это оценка элемента тестирования, при которой не происходит выполнения кода, и которая может быть проведена вручную или с помощью инструментариев. Объектом тестирования может быть документация или исходный код, а сам процесс возможен на любом этапе жизненного цикла ПО.  Статическое тестирование включает в себя:   * *Проверку документации*: требования, спецификации, архитектурные решения. * *Анализ исходного кода*: поиск синтаксических ошибок, нарушений стандартов кодирования и потенциальных уязвимостей.   **Цель статического тестирования** — выявление дефектов на ранних стадиях разработки, что снижает затраты на их исправление.  **Основные задачи:**   * *Раннее обнаружение дефектов*: Например, ошибки в требованиях или архитектуре, которые сложно исправить на поздних этапах. * *Проверка соответствия стандартам*: ГОСТ Р 56920-2016 подчеркивает необходимость проверки документации и кода на соответствие внутренним и внешним стандартам. * *Улучшение читаемости кода*. * *Снижение рисков*: Выявление ошибок на ранних этапах снижает вероятность сбоев в работе ПО после релиза.   **Методы статического тестирования**  Проверки различаются по степени формальности и могут включать в себя инспектирование, технические, ознакомительные и неформальные проверки.   * **Инспекции:**   + Формальный процесс с участием команды разработчиков, тестировщиков и заказчиков.   + Пример из ГОСТ Р 56920-2016: инспекции включают проверку требований, архитектуры и кода на соответствие спецификациям.   + Как отмечает Майерс, инспекции особенно эффективны для критически важных систем, где ошибки недопустимы (например, медицинское ПО). * **Ревью кода:**   + Менее формальный подход, где код проверяют коллеги.   + Пример из практики: в Agile-командах ревью проводятся ежедневно, что ускоряет выявление дефектов.   + В книге «Искусство тестирования» подчеркивается, что ревью улучшают не только качество кода, но и навыки разработчиков. * **Статический анализ кода:**   + **Автоматизированные инструменты** (компилятор, цикломатический анализатор сложности или анализатор защищенности кода) анализируют код на наличие уязвимостей, ошибок и нарушений стиля.   + ГОСТ Р 56920-2024 рекомендует применять статический анализ до динамического тестирования, чтобы снизить количество дефектов на этапе выполнения.   + Можете привести пример на любом языке. Статический анализатор (например, Pylint) проверит код на синтаксические ошибки, потенциальные исключения, нарушение стиля кода.   **Преимущества статического тестирования:**   * *Экономия ресурсов:* Исправление ошибки на этапе требований в 10–100 раз дешевле, чем после релиза (данные из книги). * *Повышение качества документации:* Инспекции выявляют противоречия в ТЗ, что снижает риски недопонимания. * *Снижение рисков*: ГОСТ Р 56920-2016 относит статическое тестирование к обязательным практикам для проектов с высокими требованиями к безопасности.   **Ограничения статического тестирования:**   * *Не заменяет динамическое тестирование:* Статическое тестирование не проверяет поведение программы в реальных условиях. * *Требует экспертизы:* Как пишет Майерс, эффективность инспекций зависит от квалификации участников. * *Временные затраты:* Формальные проверки могут замедлять процесс, особенно в небольших проектах.   Сочетание инспекций, ревью и автоматизированного анализа позволяет минимизировать риски и сократить затраты. |

| **9. Тестирование методом белого ящика**  **Тестирование методом белого ящика** — это подход к тестированию программного обеспечения, при котором тестирующий имеет доступ к внутренней структуре программы и использует её для построения тестов. Оно также называется тестированием, управляемым логикой программы.  **1. Постановка задачи**  **Дано:** спецификация программы, блок-схема программы.  **Требуется** протестировать программу следующими методами: покрытие операторов, покрытие решений, покрытие условий, покрытие решений и условий, комбинаторное покрытие условий.  **2. Блок-схем программы (дорисуйте только овалы начало и конец на всякий случай).**  **3.1. Покрытие операторов / инструкций. Цель:** убедиться, что каждая инструкция(оператор) выполняется хотя бы один раз. Это необходимое, но недостаточное условие для приемлемого  тестирования по принципу белого ящика. **Пример теста:** A=2, B=0, X=3 → путь a-c-e Это приведет к однократному выполнению каждой инструкции программы. **Что проверяет:** все строки кода выполняются. **Недостатки:** не гарантирует проверку всех ветвей. Может пропустить ошибки в условиях. Критерий покрытия операторов/инструкций настолько слаб, что, как правило, оказывается бесполезным. **Примеры пропущенных ошибок:** -Пусть, в программе допущена ошибка и в первой точке ветвления должна была использоваться операция OR (||), а не and (&&). Или если бы во второй точке ветвления должно было использоваться выражение х>0, а не х>1, то тест не обнаружил бы такой ошибки. -В программе существует путь (a-b-d), вдоль которого значение х не изменяется. Если этого не должно быть, то и такая ошибка останется незамеченной.  **3.2. Покрытие решений / ветвлений. Цель:** каждое условие принимает значения true и false хотя бы раз, т.е. каждая логическая ветвь каждой инструкции ветвления в программе должна быть выполнена хотя бы один раз. Поскольку покрытие операторов считается необходимым условием тестирования,определение более сильного критерия покрытия решений должно включать и покрытие операторов. Следовательно, требование 100%-го покрытия решений означает, что условное выражение в каждой точке ветвления программы должно принимать каждое из двух возможных значений, true и false, хотя бы один раз, и при этом каждая инструкция программы должна выполняться но крайней мере один раз. Данный критерий может использоваться и в упрощенной форме: тесты должны охватывать как истинный, так и ложный результат вычисления условных выражений в точках ветвления, и при этом каждая точка входа в программу (включая блоки ON) должна быть пройдена по крайней мере один раз. **Примеры тестов:** 1) A=3, B=0, X=3 → путь a-c-e 2) A=2, B=1, X=1 → путь a-b-e **Что проверяет:** все ветви каждого условия выполняются. **Недостатки:** может не обнаружить ошибки в составных условиях.  **3.3. Покрытие условий. Цель:** каждое из элементарных условий, образующих проверочное выражение в точке ветвления, принимало каждое из двух возможных логических значений, true и false, по крайней мере один раз. Т.к., как и при покрытии решений, покрытие условий не всегда обеспечивает выполнение каждой инструкции, оно должно быть дополнено требованием, чтобы каждая точка входа в программу или подпрограмму, а также каждый блок ON были пройдены хотя бы один раз. **Условия в программе:** 1) A > 1 2)B == 0 3)A == 2 4) X > 1  Следовательно, количество тестов должно быть достаточным для создания ситуаций, в которых переменные имеют значения А>1, А<=1, в=0 и В<>0 вточке а и А=2, А<>2, Х>1 и X<=1 в точке b. **Примеры тестов**, удовлетворяющие критерию покрытия этих условий: 1) A=2, B=0, X=4 → A>1 (T), B==0 (T), A==2 (T), X>1 (T), путь a-c-e 2) A=1, B=1, X=1 → A>1 (F), B==0 (F), A==2 (F), X>1 (F), путь a-b-d **Что проверяет:**- Каждое подусловие проверяется независимо. **Недостатки:**- Не гарантирует проверки всех комбинаций условий.  **3.4. Покрытие решений и условий. Цель:** объединение двух предыдущих подходов:  Набор тестов является достаточно полным, если удовлетворяются следующие требования: каждое условие в решении принимает каждое возможное значение по крайней мере один раз, каждый возможный исход решения проверяется по крайней мере один раз и каждой точке входа управление передается по крайней мере один раз. Слабое место: несмотря на кажущуюся возможность охвата им всех возможных исходов решений для всех условий, это зачастую не обеспечивается из-за маскирования одних условий другими. **Примеры тестов:** 1) A=2, B=0, X=4 2)A=1, B=1, X=1  **Что проверяет:**- Комбинированный охват условий и ветвей. **Недостатки:**- По-прежнему может не обнаружить ошибки в сложных логических выражениях. Не проверяют действия программы при исходе false в точке решения Н и исходе true в точке решения К. Тестирование на основании критерия покрытия условий или критерия покрытия решений и условий не гарантирует обнаружение ошибок в логических выражениях  **3.5. Комбинаторное покрытие условий. Цель:** каждая возможная комбинация результатов вычисления условий в каждом решении и каждая точка входа проверяются по крайней мере один раз. **Количество комбинаций:** - Для первого условия (A > 1 && B == 0) → 4 комбинации - Для второго условия (A == 2 || X > 1) → 4 комбинации  Тестами должны быть покрыты восемь комбинаций условий: 1. А>1, В=0 2. А>1, В<>0 3. А<=1,В=0 4. А<=1,В<>0 5. А=2, Х>1 6. А=2, Х<=1 7. А<>2, Х>1 8. А<>2,Х<=1 **Примеры тестов:** Для того чтобы протестировать эти комбинации, вовсе не обязательно использовать восемь тестов. Фактически они могут быть покрыты четырьмя тестами. 1. A=2, B=0, X=4 → покрывает комбинации 1 и 5 2. A=2, B=1, X=1 → покрывает комбинации 2 и 6 3. A=1, B=0, X=2 → покрывает комбинации 3 и 7 4. A=1, B=1, X=1 → покрывает комбинации 4 и 8. Совпадение количества тестов с количеством различных путей на рисунке явля-  ется случайным. На самом деле указанные четыре теста не обеспечивают покрытия  всех путей, поскольку не охватывают путь a-c-d. Комбинаторное покрытие условий может быть достигнуто меньшим числом тестов, чем количество возможных путей, и не гарантирует полного покрытия всех возможных последовательностей выполнения кода.  **Что проверяет:**- Все возможные пути, связанные с условиями.- Выявляет ошибки в логике и маскировке условий. **Недостатки:**- Растёт количество тестов экспоненциально.- Не все комбинации могут быть реальными или достижимыми.  **4. Заключение:**  Белый ящик позволяет находить ошибки в реализации, но не заменяет других методов тестирования. Он должен быть частью комплексной стратегии обеспечения качества ПО. Покрытие операторов — минимальный уровень. Комбинаторное покрытие условий — наиболее надежный, но трудозатратный способ. | **2. Стратегии тестирования**  **Тестирование** - это проверка программы на соответствие требованиям, заданным в ТЗ.  **Стратегии тестирования** — это подходы и методы, используемые для планирования и выполнения тестирования программного обеспечения. Они помогают обеспечить соответствие продукта требованиям, проверить, что продукт работает корректно и удовлетворяет ожиданиям пользователей.  Рассмотрим следующие стратегии тестирования:   * **Статическое и динамическое тестирование.**   **Статическое тестирование** — это процесс анализа программного кода, документации и других артефактов разработки без выполнения кода. **Инструменты**: CppCheck для программы на C++(напр)  **Методы статического тестирования:**  *Инспекция кода* — это процесс просмотра программного кода другими разработчиками или специалистами по тестированию с целью выявления ошибок, нарушений стиля кода и других проблем.  *Статический анализ кода* — это автоматическая проверка кода специальными инструментами, которые помогают выявить ошибки, уязвимости и нарушения рекомендаций по качеству кода.  **Динамическое тестирование** — это процесс проверки программного обеспечения путем выполнения кода программы. **Инструменты**: JUnit, TestNG для Java, Selenium поддерживает Java, Python, C# и Ruby.  **Методы динамического тестирования**  *Модульное тестирование* — проверка отдельных компонентов программы на правильность работы и соответствие требованиям.  *Интеграционное тестирование* — проверка взаимодействия между модулями программы и их совместной работы.  *Системное тестирование* — проверка полной системы на соответствие требованиям и правильность работы в реальных условиях.   * **Ручное и автоматизированное тестирование.**   **Ручное тестирование** - процесс тестирования, не предполагающий использования компьютера и осуществляемый непосредственно человеком.  Тремя основными методами ручного тестирования являются:   1. **Инспекция кода.**   Инспекция включает в себя чтение или визуальную проверку программного кода группой лиц. В состав группы входят 4 человека: координатор(квалифиц. программист, но не автор), программист, проектировщик программы (если это не сам программист) и специалист по тестированию. Программист рассказывает присутствующим о логике работы программы, подробно останавливаясь на каждой инструкции. Участники заседания анализируют код программы, сверяясь с составленными на основе имеющегося опыта контрольными списками наиболее распространенных ошибок   1. **Сквозной просмотр.**   Код проверяется группой разработчиков (3-5 чел), координатор, секретарь, тестировщик, программист. Отличия от инспекции: нет списка вопросов, есть записанными на листах бумаги тесты - наборы  входных (и ожидаемых выходных) данных для программы или модуля. Во время заседания участники “прогоняют в уме” каждый тест, т.е. подвергают тестовые данные обработке вручную в соответствии с логикой программы. Состояние программы (значения переменных) отслеживается на бумаге или на доске. Большую часть программы тестирует не ее создатель, а другие члены команды разработчиков.   1. **Тестирование удобства использования.**   Следует подготовить набор практических, связанных с реальной деятельностью, повторяющихся тестовых заданий, которые должен будет выполнить каждый пользователь. Проектировать эти тестовые сценарии нужно так, чтобы в процессе их выполнения пользователь столкнулся со всеми аспектами эксплуатации программного обеспечения. Необходимо назначить наблюдателей, которые будут документировать впечатления пользователей в процессе выполнения каждым из них своих заданий на протяжении каждой фазы теста. По завершении теста нужно провести с пользователями интервью или предоставить им анкеты для документирования других аспектов взаимодействия с программным обеспечением. нужно написать для пользовательских тестов подробные инструкции, чтобы задействованная в каждом тестовом задании исходная информация и форма ее представления были одинаковыми для всех пользователей.  **Автоматизированное тестирование** — это процесс использования прогр. инструментов для выполнения тестов на ПО, чтобы проверить его функциональность, производительность. Инструменты: JUnit, TestNG(Java), Selenium(Java, C#, Python и др.)   * **Тестирование методами <<белого>> и <<черного>> ящиков.**   **Черный**: в соответствии с этим методом программа рассматривается как “черный ящик”, внутреннее  поведение и структура которого не имеют никакого значения. Вместо этого все внимание фокусируется на выяснении обстоятельств, при которых поведение программы не соответствует спецификации(нет блок-схемы).  **Белый**: разрешается исследовать внутреннюю структуру программы. Исходя из этой стратегии, тестировщик подбирает тестовые данные путем анализа логики программы(есть блок-схема).   * **Уровни тестирования: от модульного к приемочному.**   *Модульное тестирование* — это процесс тестирования отдельных блоков,  подпрограмм, классов или процедур, образующих крупную программу.  *Интеграционное тестирование* - на этом этапе отдельные модули или компоненты приложения объединяют и проверяют на совместимость друг с другом.  *Функциональное тестирование*  — выявление расхождений между поведением программы и внешней спецификацией(точное опис. поведения проги с точки зрения конечного пользователя).  *Системное тестирование* рассматривает всю систему в реальных условиях , чтобы убедиться, что она реализует цели проекта и готова к испол. Нужен документ, отражающий набор измеримых целей.  *Приемочное тестирование* — сравнения возможностей разработанной программы с исходными требованиями и потребностями конечных пользователей. Его выполнение возлагается на заказчика или конечного пользователя и не входит в обязанности компании-разработчика.  **Закл**: Тестирование программного обеспечения — это важный этап разработки, который помогает выявить ошибки, а также убедиться, что программа соответствует требованиям и удобна для пользователя. Лучший способ протестировать программу - это грамотно скомбинировать несколько стратегий тестирования. |
| --- | --- |
| **11. Метод причинно-следственных диаграмм**  У анализа граничных значений и разбиения на классы есть недостатки: не исследуем все комбинации входных условий, а перебирать все - очень тяжело так как даже используя 2 вышеизложенных метода  дают много вариантов. Можно выбирать произвольно - это не эффективно. Значит нужен новый метод.  **Метод причинно-следственных диаграмм** — метод, позволяющий выбирать высоко результативные тесты, а также обнаруживать неполноту и неоднозначность исходной спецификации.  **Причинно-следственная диаграмма** — формальный язык, на который транслируется спецификация, написанная на естественном языке. Является аналогом цифровой логической схемы, но используется более простая нотация. Для использования методом необходимо знание правил булевой алгебры  **Порядок построения причинно-следственных диаграмм (ПСД – неформально)**  1. Спецификация разбивается на части, с которыми легче работать. Нужно т.к ПСД для больших спецификаций сильно разрастаются  2. В спецификации определяются причины и следствия. Причины и следствия определяются путем последовательного (слово за словом) чтения спецификации и подчеркивания тех слов или фраз, которые описывают причины и следствия. Каждой причине и каждому следствию присваивается уникальный номер. **Причина** — это отдельное входное условие или класс эквивалентности входных условий. **Следствие** — это выходное условие или преобразование системы (долговременное воздействие, которое входное условие оказывает на состояние программы или системы).  3. Семантическое содержание спецификации анализируется и преобразуется в булев граф, связывающий причины и следствия. Граф = ПСД.  4. Д-мма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, реализация которых невозможна из-за синтаксических или внешних ограничений.  5. Путем методичного прослеживания состояний условий диаграмма преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.  6. Столбцы таблицы решений преобразуются в тесты. **Нотация:**  ***Тождество:*** if a=1 => b=1 else b=0 **Not:** if a = 1 => b= 0 else b =1  **Or:** if a = 1 | b =1 |c =1 => d=1 else d =0 **And:** if a=1 && b =1 => c =1 else c= 0  **Ограничения:**  **Е** — а = 1 или b =1 вместе не могут быть 1, но могут быть 0 .  **I** — мин. одна из a b c =1, не могут быть равны - одновременно).  **О** — одна и только одна из величин, a или b, была равна 1.  **R** — a = 1, только если b равно 1 **М** — if a =1, то b - автоматом 0. **Генерация таблицы решений**: **1.** Выберите следствие, которое должно находиться в состоянии 1 .  **2.** Продвигаясь вдоль линий диаграммы, ведущих к данному узлу, в обратном направлении, найдите все комбинации причин, устанавливающий следствие в 1. **3.** Создайте столбец в таблице решений для каждой комбинации причин. **4.**Определите для каждой комбинации состояния всех других следствий и поместите их в соответствующий столбец таблицы.  **Во время трассировки используйте эти правила :**  1. if идем через or, выход которого должен = 1 1 то ставим в 1 только 1 вход  2. if идем через and, выход которого должен = 0, то, нужно перечислить все комбинации входов, приводящие к 0. Но, if один вход 0, а несколько других 1, то рассматривается 1 случай.  3. if идем через and, выход которого должен = 0, то указываем 1 условие, при котором все входы = 0  **Примеры:**  Входы:  1 — символ в колонке 1 является буквой “А”; 2 — символ в колонке 1 является буквой “В”; 3 — символ в колонке 2 является цифрой,  Выходы:  70 — файл обновляется; 71 — выводится сообщение X12; 72 — выводится сообщение Х13.  **Пример трассировки правил**: Пусть в 7 – 0.  Тогда мы смотрим один случай, когда узлы 5 и 6 нули.  Для положения, 5 – 1, 6 – 0, мы ограничиваемся только одним случаем, когда узел 5 - 1, а не смотреть на все.  Также и для положения 5 – 0, 6 – 1, смотрим только на то, когда 6 принимает 1. Если узел 5 должен быть 1, то мы не ставим 1 в узлы 1 и 2 одновременно Так получаем всего 5 состояний покрывающих все, не рассматривая 13 возмозможных | **12. Модульное тестирование, инкрементное тестирование**  **Модульное тестирование** — это процесс тестирования отдельных блоков, подпрограмм, классов или процедур, образующих крупную программу. Прежде чем тестировать программу в целом, необходимо сосредоточить внимание на ее меньших по размеру компонентах. Во-первых, при таком подходе повышается эффективность тестирования сложных объектов, поскольку на первом этапе внимание фокусируется на небольших программных блоках, которые легче тестировать. Во-вторых, при таком подходе облегчается отладка программ (точная локализация и исправление обнаруженной ошибки), поскольку, обнаружив ошибку, мы всегда точно знаем, в каком именно модуле она содержится. Наконец, модульное тестирование позволяет распараллеливать процесс тестирования, что обеспечивает возможность одновременного тестирования нескольких модулей. Цель модульного тестирования — сравнение функций, реализуемых модулем, со спецификациями, описывающими его функциональные или интерфейсные характеристики. При проектировании модульных тестов используются два источника информации: спецификация модуля и его исходный код. Типичная спецификация описывает назначение модуля, а также его входные и выходные параметры. Модульное тестирование в основном ориентировано на использование метода “белого ящика”. Процесс модульного тестирования зависит от двух важных аспектов: проектирование эффективного набора тестов и то, каким образом модули объединяются в работающую программу.  **Инкрементное тестирование** является одним из подходов интеграционного тестирования. При  таком подходе модули не тестируются изолированно друг от друга, а поочередно подключаются к постепенно наращиваемому набору уже проверенных модулей и лишь после этого подвергаются тестированию. То есть мы объединяем модули постепенно, один за другим, пока все компоненты не будут добавлены в логику разрабатываемого приложения, вместо того чтобы интегрировать всю систему сразу и затем проводить тестирование конечного продукта.  **Цели инкрементного тестирования: 1.** Убедиться, что различные модули успешно работают вместе после интеграции. **2**. Облегчить локализацию возникновения дефекта. Например, если тестирование после интеграции M1 и M2 прошло успешно, но когда добавляется M3, тест не проходит, это поможет разработчику определить источник проблемы. **3.** Устранение проблем на ранней стадии без значительных доработок и с меньшими затратами.  **Методологии инкрементного интеграционного тестирования:**  Краткое введение о заглушках и драйверах: по сути, заглушки и драйверы – это псевдокод или фиктивный код, используемый в интеграционном тестировании, когда один или несколько модулей не разработаны, но необходимы для тестирования какого-либо другого модуля. **Заглушки** используются при тестировании сверху вниз и известны как “вызываемые программы”. Заглушки помогают имитировать интерфейс между модулями нижнего уровня, которые недоступны или не разработаны. **Драйверы** используются в подходе тестирования снизу вверх и известны как “вызывающие программы”. Драйверы помогают имитировать интерфейс между модулями верхнего уровня, которые не разработаны или недоступны.  **Метод “сверху вниз”**  Как следует из названия, тестирование происходит сверху вниз, т.е. от центрального модуля к подмодулям. Модули, составляющие верхний уровень приложения, тестируются в первую очередь.  Этот подход используется, если программа рассматривается как иерархическая структура, где центральный (главный) модуль управляет работой подчиненных ему подмодулей. Недоступные или не разработанные модули или компоненты заменяются заглушками.  Можно тестировать **“в ширину”**(сначала тестируются все модули на одном уровне): Будут разработаны следующие тест-кейсы: Модуль L и модуль O будут интегрированы и протестированы; Модули L, O и P будут интегрированы и протестированы; Модули L, O, P и R будут интегрированы и протестированы.  Для “в глубину” будут разработаны следующие тест-кейсы:  Модуль L и модуль O будут интегрированы и протестированы;  Модули L, O и OS будут интегрированы и протестированы Модули L, O, OS, P будут интегрированы и протестированы; Модули L, O, OS, P, CP будут интегрированы и протестированы.  **Метод “снизу вверх”(думаю сообразите по картинке, как модули интегрировать, места мала)**  При таком подходе тестирование происходит снизу вверх, т.е. сначала интегрируются и тестируются модули на нижнем уровне, а затем последовательно интегрируются другие модули по мере продвижения вверх. Недоступные или не разработанные модули заменяются драйверами.  **Закл**: Модульное и инкрементное тестирование играют ключевую роль в обеспечении качества ПО на ранних этапах разработки. Модульное тестирование позволяет проверить корректность работы отдельных компонентов программы. Инкрементное тестирование помогает поэтапно проверять взаимодействие модулей между собой, что облегчает локализацию проблем и снижает затраты на их устранение. Таким образом, сочетание модульного и инкрементного тестирования обеспечивает высокую надёжность программного продукта и гарантирует стабильную работу всей системы в целом. |

| **13. Нисходящее тестирование**  **Модульное тестирование —** процесс тестирования отдельных блоков, подпрограмм, классов или процедур, образующих крупную программу.  **Инкрементное тестирование** — метод, при котором модули тестируются не изолированно, а последовательно добавляются к уже проверенным.  Две стратегии инкрементного тестирования: **1**) ***Нисходящее тестирование***: начинается с верхних уровней иерархии, используются заглушки) **2)** *Восходящее тестировани*е: начинается с терминальных модулей (не вызывающих другие), используются драйверы.  **Нисходящее тестирование**  Нисходящее тестирование начинается с верхнего в иерархической структуре, или головного, модуля программы. Какой-либо единственно <<правильной>> процедуры, регламентирующей последующий выбор каждого очередного модуля, подлежащего инкрементному тестированию, не существует. Руководствоваться следует лишь тем, что подходящим для этих целей будет такой модуль, для которого найдется по крайней мере один вызывающий его модуль, уже прошедший тестирование.  ***Создание заглушек.*** Первый шаг — тестирование головного модуля. Необходимо написать заглушки, представляющие модули на следующем нижнем уровне. Заглушка — это объект, который заменяет реальную зависимость в тесте и возвращает заранее определенные значения. Задача заглушек заключается в выполнении некоторой полезной работы, которая должна завершаться определенным результатом (выходными данными). Если заглушка не возвращает никакого значимого результата, то тестируемый модуль аварийно завершит свою работу, причем это произойдет не из-за того, что в нем содержится ошибка, а из-за того, что заглушка не обеспечивает надлежащей имитации недостающего модуля.  **Передача тестовых данных.** Тестовые данные поступают в головной модуль из одной или нескольких связанных с ним заглушек. Поскольку вполне вероятно, что головной модуль вызывает заглушку всего один раз, нужно решить, каким образом передать в головной модуль несколько тестов. Решение - разработать несколько версий заглушки, каждая из которых содержит один фиксированный набор тестовых данных для возврата в головной модуль . Тогда для выполнения всех тестов достаточно запустить программу несколько раз, используя при каждом запуске разные версии заглушки. Возможен и другой вариант решения - тестовые данные помещаются во внешние файлы, откуда заглушка сможет прочитать их, а затем возвратить в головной модуль.  **Замена заглушек.** После того как головной модуль будет протестирован, одну из заглушек заменяют реальным модулем с последующим добавлением требуемых им дополнительных заглушек. Возможны самые разные варианты последовательности тестирования оставшихся модулей и их слияния для образования единой программы.  **Параллельное тестирование нескольких модулей**  При параллельном тестировании нескольких модулей возможны альтернативные варианты. Например, после тестирования модуля A один программист может тестировать комбинацию A-В, второй — А-С, третий — A-D. В общем случае сказать, какая последовательность будет лучшей, нельзя, но поэтому поводу можно дать две рекомендации- **1)** Если в программе имеются критически важные разделы), то целесообразно выбирать последовательность так, чтобы эти разделы включались в цепочку как можно раньше. В качестве критически важного раздела может выступать сложный модуль, модуль, реализующий новый алгоритм, или модуль, относительно которого есть опасения, что он может содержать ошибки. **2)** Модули, содержащие операции ввода-вывода, следует включать в цепочку тестирования как можно раньше.  **Промежуточное состояние программы в процессе нисходящего тестирования. Преимущества**  По достижении промежуточного состояния программы, представление тестов и анализ результатов тестирования существенно упрощаются. Это состояние обладает еще одним преимуществом, которое заключается в том, что вы получаете в свое распоряжение рабочую каркасную версию, выполняющую реальные операции ввода-вывода, в то время как часть остальных “внутренностей” программы по-прежнему имитируется заглушками. Эта ранняя каркасная версия программы предоставляет следующие возможности: **1)** Позволяет обнаруживать ошибки и проблемы, обусловленные человеческим фактором; **2)** Дает возможность продемонстрировать работу программы конечному пользователю; **3)** служит доказательством правильности подхода, лежащего в основе разработки программы.  **Недостатки. 1)** Из-за зависимости от более высокоуровневых модулей может оказаться невозможным предоставить все необходимые тестовые данные для проверки конкретных ситуаций в тестируемом модуле; **2)** В случае <<удаленности>> модуля от точки, в которой тестовые данные вводятся в программу, определение того, какие именно данные должны быть предоставлены модулю для тестирования этих ситуаций, часто будет требовать больших интеллектуальных усилий. **3)** Поскольку отображаемые выходные данные теста могут поступать из модуля, находящегося на большом “расстоянии” от тестируемого модуля, установление соответствия между наблюдаемыми результатами и тем, что на самом деле происходит в модуле, может представлять собой трудную или вообще неразрешимую задачу. | **14. Восходящее тестирование**  **Модульное тестирование —** это процесс тестирования отдельных блоков, подпрограмм, классов или процедур, образующих крупную программу.  **Инкрементное тестирование** — метод, при котором модули тестируются не изолированно, а последовательно добавляются к уже проверенным.  Две стратегии инкрементного тестирования:   * *Нисходящее тестирование*: начинается с верхних уровней иерархии, используются заглушки (stubs). * *Восходящее тестировани*е: начинается с терминальных модулей (не вызывающих другие), используются драйверы (drivers)   **Проблемы нисходящего тестирования**:   1. Сложность тестовых данных: для проверки модуля требуются входные данные, учитывающие взаимодействие через несколько уровней иерархии. 2. Трудность интерпретации результатов: выходные данные могут поступать из отдалённых модулей, что затрудняет локализацию ошибок. 3. Зависимость от заглушек: заглушки (stubs) для имитации нижних модулей сложны в разработке и могут требовать множества версий.   В соответствии с восходящей стратегией тестирование начинают с терминальных модулей программы (не вызывающих другие модули). Очередной выбираемый модуль должен быть таким, чтобы все его подчиненные (вызываемые им) модули предварительно были протестированы.  Для каждого модуля нужно создать **драйвер** — программу с встроенными тестовыми данными, которая вызывает тестируемый модуль и моделирует или сравнивает результаты его работы с ожидаемыми. В отличие от заглушек, драйвер не требует множества версий, так как может многократно вызывать модуль с разными наборами данных. Обычно создавать драйверы проще, чем заглушки. Как и при нисходящем тестировании, последовательность тестирования определяется прежде всего критичностью того или иного модуля.  **Плюсы восходящего тестирования:**   1. Отсутствуют сложности с созданием тестовых ситуаций, характерные для нисходящего тестирования. Восходящее тестирование работает с уже готовыми нижними модулями, поэтому нет необходимости моделировать или имитировать работу ещё не разработанных компонентов. 2. Драйверы применяются непосредственно к тестируемым модулям, без необходимости заглушек для промежуточных модулей. Такой подход снижает сложность тестовой среды, так как не нужно создавать дополнительные имитаторы модулей, что уменьшает объём вспомогательного кода и упрощает процесс тестирования 3. Легче локализовать ошибки, так как тестируется каждый модуль отдельно 4. Возможность тестирования функций ввода-вывода до полной интеграции. Даже до объединения всей программы можно проверить корректность работы отдельных интерфейсов и взаимодействий с внешними системами 5. Нет проблем с незавершённым тестированием одного модуля при переходе к другому — отсутствуют сложности с разными версиями заглушек. Поскольку каждый модуль тестируется отдельно с помощью собственного драйвера, не возникает путаницы с тестовыми данными и их совместимостью между версиями   **Минусы восходящего тестирования:**   1. Невозможность раннего формирования каркаса программы. Полноценная программа с интегрированными модулями появляется только после тестирования последнего модуля. 2. Необходимость написания множества драйверов. Для каждого модуля требуется отдельный драйвер, что увеличивает объём вспомогательного кода. 3. Большой объём отладочного программирования. Создание и сопровождение драйверов требуют времени и ресурсов. 4. Отсутствие преимуществ раннего тестирования интеграции. Системные ошибки могут выявиться только на поздних этапах. 5. Тестовые данные обычно не в форме, рассчитанной на пользователя, особенно на ранних этапах. 6. Необходимость отдельного тестирования взаимодействия модулей   Принимая во внимание серьезность последствий такого недостатка нисходящего тестирования, как значительная сложность создания тестовых условий, а также учитывая доступность инструментов тестирования, устраняющих потребность в драйверах, но не заглушках, можно сделать вывод о том, что предпочтение следует отдавать стратегии восходящего тестирования |
| --- | --- |
| **19. Планирование и контроль тестирования**  В условиях крупных проектов с множеством компонентов и участников необходим структурированный подход к тестированию. Простого выполнения тестов недостаточно — требуется системное управление процессом.  **Цель управления тестированием** — обеспечить *эффективность* и *результативность* процесса тестирования, минимизировать риски и достичь требуемого уровня качества продукта в рамках установленных сроков и бюджета.  ***Планирование тестирования***  Планирование определяет стратегию, объем работ, ресурсы и график тестирования.  Основным результатом является **тестовый план** — формальный документ, описывающий цели, объем, стратегию и ресурсы, включающий:   * ***Цели тестирования:*** Четкое определение того, что должно быть достигнуто в результате тестирования (например, проверка соответствия требованиям, оценка производительности, поиск дефектов в критически важных функциях). * ***Область тестирования:***Описание того, какие компоненты, функции или характеристики продукта будут и не будут тестироваться. * ***Стратегия тестирования:*** Описание подходов и методов тестирования, которые будут использоваться (например, ручное, автоматизированное, нагрузочное, тестирование использования). * ***Критерии начала и завершения тестирования:*** Условия, при которых тестирование может начаться (например, готовность тестового окружения, стабильность сборки) и условия, при которых оно считается завершенным (например, достижение определенного уровня покрытия кода тестами, отсутствие критических дефектов, истечение запланированного времени). * ***Графики и сроки:***Расписание выполнения различных этапов и задач тестирования, согласованное с общим планом проекта. * ***Ресурсы:*** Определение необходимых ресурсов, включая команду тестировщиков (с указанием ролей), оборудование, программное обеспечение и тестовые данные. * ***Ответственность:***Назначение ответственных лиц за выполнение конкретных задач и этапов тестирования. * ***Управление рисками:***Идентификация потенциальных рисков, связанных с процессом тестирования (например, нехватка ресурсов, задержки, нестабильность тестового окружения), и разработка планов по их минимизации. * ***Метрики и отчетность:*** Определение ключевых показателей для отслеживания прогресса (например, количество выполненных тестов, найденных дефектов) и формата отчетности.   **Основные проблемы:**   * Недооценка ресурсов и ошибочное предположение об отсутствии дефектов; * Сложность корректировки планов на финальных этапах разработки.   ***Контроль тестирования***  Контроль включает мониторинг хода тестирования, анализ полученных результатов, управление дефектами и актуализацию планов при необходимости:   * **Отслеживание выполнения** — непрерывный контроль выполнения тестовых активностей. * **Управление дефектами** — отслеживание полного жизненного цикла дефекта: от регистрации и анализа до исправления и повторной проверки. * **Регрессионное тестирование** — проверка, что внесенные изменения не нарушили ранее работающую функциональность. | **10. Тестирование методом черного ящика**  **Метод черного ящика** — это подход к тестированию программ на соответствие спецификации, при котором проверяется только соответствие входных и выходных данных требованиям, без учета внутренней структуры кода. Программа рассматривается как "черный ящик": важно, что она делает, а не как.  **Дано**: спецификация программы.  **Цель**: проверка соответствия программы спецификации. То есть программа должна работать правильно при всех допустимых входных данных и поведение программы должно быть предсказуемым и согласованным с требованиями.  **1. Основные приемы тестирования**  **1.1. Разбиение на эквивалентные классы**  Хорошо спроектированный тест должен охватывать значительную часть возможных случаев, выявляя ошибки даже в ситуациях, не проверяемых напрямую. Для этого область входных данных разбивают на классы эквивалентности. Тестирование одного значения из класса дает уверенность в корректности работы для всех значений этого класса.  - Цель: Сократить количество тестов, проверяя не все возможные значения, а только по одному из каждого класса.  - Пример классов: Если программа принимает числа от 1 до 100, допустимый класс — числа в этом диапазоне, недопустимые: 1) меньше 1; 2) больше 100.  Определение тестов по построенным классам эквивалентности состоит из трех этапов:   1. Назначить каждому классу эквивалентности уникальный номер; 2. Записать новые тесты, охватывающие как можно большее количество оставшихся неохваченными допустимых классов эквивалентности, пока не будут покрыты все допустимые классы; 3. Записать новые тесты, каждый из которых охватывает один и только один из оставшихся неохваченными недопустимых классов эквивалентности, пока не будут покрыты все недопустимые классы.   **1.2. Анализ граничных значений**  - Суть: Ошибки часто возникают на границах допустимых диапазонов, поэтому тестируются:   1. Сами граничные значения (например, 1 и 100). 2. Значения за границами (0 и 101). 3. Близкие к границе числа (0.999, 100.001).   - Пример: Для поля "Возраст" (допустимо 18–99 лет) тестируются 17, 18, 19, 98, 99, 100.  **1.3. Причинно-следственные диаграммы** (в основном 11 вопрос)  Слабая сторона двух рассмотренных приемов — они не исследуют комбинации входных условий.  Метод причинно-следственных диаграмм помогает это решить.  - Суть: Анализируются связи между входными данными (причины) и выходными результатами (следствия).  - Как это работает:  1. Выявляются все возможные комбинации входных данных.  2. Строится диаграмма с логическими связями (И, ИЛИ, НЕ).  3. На основе диаграммы создаются тестовые случаи.  - Зачем нужно: Позволяет проверить сложные комбинации входных данных, которые классы эквивалентности не охватывают.  **3. Вывод**  Метод черного ящика эффективен для проверки соответствия программы требованиям, но требует комбинации разных подходов:  - Эквивалентные классы — для базового тестирования.  - Граничные значения — для проверки крайних случаев.  - Причинно-следственные диаграммы — для анализа сложных комбинаций входных данных. |

| **17. Системное тестирование**  **Системное тестирование** — это один из ключевых этапов тестирования ПО, направленный на проверку соответствия всей системы или программы первоначально сформулированным целям и требованиям.  В отличие от функционального тестирования, которое ориентируется на **внешнюю спецификацию** и проверяет корректность отдельных функций, системное тестирование:   * рассматривает **программный продукт как целое**; * фокусируется на **достижении конечных целей**, определенных в постановке задачи или бизнес-требованиях; * оценивает, насколько **реализация удовлетворяет ожидаемому поведению с точки зрения пользователя**.   Из этого следуют два ключевых положения:   1. Системное тестирование не ограничивается только системами. Если продукт является программой, то системное тестирование в данном случае представляет собой попытку продемонстрировать, какие из стоящих перед программой целей не реализуются. 2. Системное тестирование по определению невозможно, если отсутствует документ, отражающий набор **четких** и **измеримых** целей, достижение которых возлагается на продукт.   (справа снизу зачеркнута внешняя спецификация). Внешняя спецификация, как правило, используется на этапе функционального тестирования, где она проверяется на соответствие реализованным функциям.  На этапе системного тестирования внешняя спецификация уже считается **реализованной**, и задача тестировщика — проверить, насколько реализация в целом отвечает исходным целям, опираясь, в том числе, на **пользовательскую документацию**, отражающую фактическое поведение системы.  Сложность составления системных тестов заключается в том, что нет общепризнанной методологии их написания. Обусловлено это тем, что формулировки целей устанавливают, что именно и насколько хорошо должна делать программа, но не определяют способа реализации ее функций.  Системные тесты были поделены на 15 категорий. Не все они применимы к каждой программе в равной степени, однако для того, чтобы ничего не упустить, лучше исследовать возможность применения каждой из них:    **Можно утверждать, что**   1. системное тестирование не должны выполнять программисты: человек, который будет выполнять системный тест, должен уметь рассуждать с позиций конечного пользователя. 2. из всех этапов тестирования этот — единственный, который ни при каких условиях не должен выполняться организацией, ответственной за разработку программ: компания-разработчик проявляет определенную привязанность к собственной программе и далеко не всегда заинтересована в нахождении несоответствий между программой и первоначальными требованиями, что препятствует эффективному тестированию. | **18. Приемочное тестирование, тестирование установки**  **Приемочное тестирование** – процесс сравнения возможностей разработанной программы с исходными требованиями и потребностями конечных пользователей.  Выполнение возлагается на заказчика или конечного пользователя и не входит в обязанности компании-разработчика.  Если программа разрабатывается в соответствии с контрактом, то организация-заказчик (пользователь) выполняет приемочное тестирование, сравнивая работу программы с оговоренными в исходном контракте требованиями.  В России перед выпуском ИТ-продуктов, особенно масштабных систем, принято проводить приемо-сдаточные испытания, основанные на программе и методике испытаний (ПМИ).  **ПМИ** — это документ с технической информацией о ПО и инструкцией, описывающей процесс проведения приемочных тестов. ПМИ — ключевой руководящий документ для проведения приемо сдаточных испытаний. Разработка методики регламентируется в России ГОСТом 19.301-79. В этом документе отображены требования к методике написания ПМИ и её оформлению.  **Виды приемочного тестирования:**   * **Приемочное пользовательское тестирование** - это оценка решения с точки зрения конечного пользователя и его восприятия. Такое тестирование можно определить как пользовательскую методологию, в рамках которой разработанное ПО тестируется для подтверждения того, что оно работает в соответствии с заданными спецификациями. * **Приемочное тестирование для разных сфер бизнеса** - направлена на то, чтобы убедиться, что решение соответствует целям и задачам бизнеса. Основано на понимании поведения конечного пользователя, а также экономических бизнес-выгод. Это требует глубокого знания продукта и целевой аудитории, особенно для команды тестирования и бизнес аналитиков. * **Законодательное приемочное тестирование** - процесс проверки того, что продукт или система соответствует всем требованиям, установленным регулирующими органами. Основная цель - снизить риск для компании и убедиться, что решение безопасно для человека и окружающей среды. Например, когда компания хочет выпустить медицинское ПО, ей необходимо доказать, что ее продукт не даст сбой и не причинит вред или травму людям * **Эксплуатационное приемочное тестирование** - это нефункциональное тестирование, используемое для определения эксплуатационной готовности продукта. Это тестирование позволяет убедиться, что все функции ПО будут работать правильно после сбоя, установки обновлений или доработки. * **Альфа-тестирование** - используется в среде тестирования специализированной командой QA-инженеров, известной как альфа-тестеры, которые исследуют продукт, имитируя поведение реальных пользователей, и дают обратную связь. * **Бета-тестирование** - Его проводят для оценки продукта путем предоставления решения реальным конечным пользователям, которых обычно называют бета-тестерами.   **Этапы приемочное тестирование:**   1. **Анализ требований** -QA-специалисты сначала анализирует документы с техническими требованиями, а затем на их основе определяют цели разрабатываемого ПО. В процессе используются документы с требованиями, блок-схемы и бизнес-сценарии. Заполняются документы с бизнес-требованиями, спецификациями системных требований и устав проекта. 2. **Формирование плана тестирования** - План описывает стратегию приемочного тестирования. В тест-плане указаны цели, подходы, график, оценки, сроки и ресурсы, необходимые для успешного завершения проекта. 3. **Разработка тест-кейсов** - На этом этапе разрабатываются тестовые примеры, охватывающие большинство сценариев на основе плана тестирования. 4. **Проведение тестирования** - проводятся приемочные тесты с использованием входных значений. Тестировщик собирает и выполняет все полученные значения, чтобы убедиться, что ПО работает корректно. 5. **Объективная оценка** - команда тестирования подтверждает, что ПО не содержит критических дефектов, соответствует заранее установленным критериям качества и готово к релизу. Или наоборот, тестировщики предоставляют отчет о найденных критических дефектах, которые необходимо устранить, или отчёт о несоответствии установленным критериям качества.   **Тестирование установки**  Цель - выявить не программные ошибки, а ошибки, происходящие в ходе установки программного продукта.  **Действия, которые происходят при установки программного обеспечения:**   * Пользователь выбирает множество опций * Выделяется память для размещения файлов и библиотек и осуществляется их загрузка * Предоставляются допустимые конфигурации оборудования * Программам могут потребоваться сетевые подключения для связи с другими программами   **Аспекты, на которые направлено тестирование установки:**   * Корректность установки - компоненты ПО устанавливаются правильно * Совместимость - ПО корректно устанавливается на различных операционных системах * Пользовательский интерфейс - оценка удобства и понятности интерфейса установщика * Обновление - проверка корректности процесса обновления существующих версий ПО * Удаление - программа удаляется полностью и не оставляет лишних файлов или записей в системе * Обработка ошибок - проверк,как установщик реагирует на ошибки(нет места на диске или нет прав)   **Установщик** — это специальная программа или скрипт, предназначенный для автоматизации процесса установки ПО. **Его основные функции включают**:   * Упрощение установки - делают процесс установки более удобным и быстрым для пользователей * Управление зависимостями - установщики могут автоматически загружать и устанавливать необходимые библиотеки и компоненты * Настройка окружения - установщики могут настраивать параметры системы или окружения для оптимальной работы установленного ПО * Обновление и удаление - установщики также могут управлять процессами обновления и удаления программного обеспечения   **Этапы тестирования установки:**   1. Подготовка к тестированию - Определение требований -> Настройка тестовой среды(подготовка различных конфигураций систем, на которых будет проводиться тестирование). 2. Тестирование процесса установки - Запуск установщика(установщик запускается без ошибок) -> Проверка пользовательского интерфейса(оценка удобства и понятности интерфейса установщика) -> Доступность необходимых ресурсов(установщик корректно определяет наличие необходимых библиотек и зависимостей) 3. Проверка установки - Корректность установки файлов(все файлы устанавливаются в правильные директории и у них ожидаемые размеры) -> Регистрация в системе(проверка того, что ПО зарегистрировано в системе) 4. Тестирование обновления - Процесс обновления(проверка корректности обновления) -> Сохранение данных(пользовательские данные и настройки должны сохраняться после обновления) 5. Тестирование удаления - Удаление программы(проверка того, что программа удаляется без ошибок, а все связанные файлы и записи в реестре удаляются полностью) 6. Обработка ошибок - Сценарии с ошибками(проверка реакции установщика на различные сценарии) -> Логирование ошибок(логируются и отображаются пользователю понятным образом) 7. Документация и отчеты - Составление отчётов -> Рекомендации по улучшению. |
| --- | --- |

| **21. Методы отладки, метод грубой силы**  **Отладка** — это процесс, который начинается после обнаружения ошибки (например, в результате успешного тестирования) и проводится в два этапа: **1.** Определение природы и местонахождения предполагаемой ошибки в программе (локализация) – составляет 95% от всей проблемы  **2**. Исправление ошибки Главное в методах отладки – быстро определить место и причину ошибки.  **Метод грубой силы –**  самый простой и часто используемый способ. Заключается в том, чтобы «засыпать» программу отладочными выводами, дампами памяти, логами, а затем анализировать полученные данные Не требует глубокого анализа структуры программы.  Метод грубой силы можно разделить на 3 категории по используемым инструментам: (дамп памяти, вывода на печать, инструменты IDE (т.останова)  **Преимущества** метода: простой, не нужно ничего знать о коде,всегда можно использовать, хорош для простых ошибок.  **Недостатки: т**рудоемкий и медленный, генерирует огромные объемы данных, неэффективен для больших программ. не дает понять логику возникновения ошибок. **Индуктивная отладка** – метод, использующий переход от частных наблюдений к общим выводам.  Отладка начинается с рассмотрения ряда ключевых факторов (симптомов ошибки и, возможно, результатов одного или нескольких тестов) с последующим переходом к установлению взаимосвязей между этими факторами. Если коротко:  1. Анализируются факты: когда и при каких условиях возникает ошибка.  2.Сравниваются ситуации, когда программа работает правильно и когда — нет.  3. На основе различий формулируется гипотеза о причине ошибки.  **Порядок отладки:**  1. Сбор всей доступной информации о том, какие действия выполняются программой корректно, а какие — некорректно  2. Структурирование обстоятельства возникновения ошибки, с целью выявления возможных закономерностей (таблица)  3. Изучение взаимосвязи, выдвигаем гипотезы, почему ошибка встречается.  4. Доказательство того, что ошибка встречается: сравнивая ее с первоначальными признаками ошибки или данными, дабы быть уверенным в том, что она полностью объясняет существование анализируемых признаков ошибки.  5. Исправление ошибки. После регрессионные тесты  **Пример:** Необходимо вычислить медиану баллов теста студентов **Ошибка:** при тестировании 51 студента среднее значение было правильным (73,2), но медиана вместо 82 показывала 26.  **Выделенные факты:** Ошибка проявляется только при нечётном количестве студентов.  В некоторых тестах медиана меньше или равна количеству студентов (26 <= 51). Повторный тест с другим набором оценок может подтвердить, что медиана не зависит от фактических оценок.  **Гипотеза:** медиана вычисляется как округлённый результат деления количества студентов на 2.  Проверяем гипотезу, если это так, то исправляем ошибку.  **Дедуктивная отладка** – метод, начинающийся с анализа общих теоритических положений или допущений и через цепочку логических умозаключений путем постепенного исключения и уточнения фактов приводит к конкретному решению (о местонахождении ошибки)  **Порядок отладки**  1. Составление списка всех мыслимых причин возникновения ошибки. Предположения о них не должны давать полное объяснение фактам.  2. Исключаем невозможные причины путем сбора информации о проявлении ошибки. При этом, если все гипотезы отвергнуты – выдвигаем новые. Если осталось несколько – рассматриваем наиболее вероятную  3. Уточнение выбранной гипотезы. Более детально разрабатываем причины, по которым могла развиться ошибка. (Пример: “ошибка возникает при обработке последней записи из файла” => ”возможно не удалили eot”  4. Доказательство выбранной гипотезы. Совпадает с аналогичным этапом индуктивной отладки.  5. Исправление ошибки. Совпадает с аналогичным этапом индуктивной отладки.  **Пример:**  Проблема: Медиана 26 вместо 82 (51 студ., среднее 73.2).  Возможные причины:Неверная сортировка данных., Ошибка в алгоритме медианы.  Исключение: Среднее верное ⟹ данные читаются правильно.  Уточнение гипотез: Наиболее вероятно: данные не отсортированы или отсортированы неверно.  Менее вероятно: ошибка в извлечении элемента (индекс).  Доказательство: Вручную (или программно) отсортировать баллы.Проверить 26-й элемент отсортированного списка.Если он 82, то причина подтверждена.  5. Исправление:Исправить сортировку данных перед вычислением медианы.  Убедиться в корректности индекса для нечетного числа элементов.  **Метод обратной трассировки (backtracking)** — прослеживание появления некорректных результатов в обратном направлении до той точки, в которой происходит сбой в работе логики программы.  Отладка начинается в точке программы, в которой программа выдает неправильный результат. При этом, исходя из наблюдаемого вывода, устанавливают, какими должны быть значения переменных в данной точке. Мысленно выполнив программу в обратном порядке и повторно применив логику “если, то”, определяется точное местонахождение ошибки. Это место будет находиться где-то между точкой, в которой состояние программы совпадает с ожидаемым, и первой точкой, в которой это соответствие нарушается.  **Отладка тестированием** — метод отладки, базирующийся на использовании тестов.  Используются тесты для отладки, цель которых — предоставить информацию, полезную для локализации ошибок, относительно которых уже имеются некоторые соображения.  Тесты бывают разные:   * Тест для тестирования Толстые — небольшим числом тестов собираемся покрыть максимально большое множество условий * Тест для отладки: Тонкие — каждый тест покрывает только одно или небольшую группу условий.   На основе симптома разрабатываем различные варианты исходного теста, сужаем область поиска.  Используется:   * в индуктивном методе для выдвижения и (или) доказательства гипотез об ошибках. * в дедуктивном методе для исключения возможных причин ошибки из анализируемого | **22. Тестирование удобства пользователя**  **Тестирование удобства использования** - тестирование с целью определения степени понятности, легкости в изучении и использовании, привлекательности программного продукта для пользователя при условии использования в заданных условиях эксплуатации.  **Цель** - определение, насколько программный продукт понятен, легок в освоении, прост в эксплуатации и привлекателен для пользователей при определенных условиях и требованиях.  Этот тип тестирования обычно выполняется реальными пользователями. Является одной из разновидностей тестирования по принципу «черного ящика».  Главные аспект тестирования - анализ человеческого фактора.  **Проверки, на которые направлены тесты:**   * Учитываются разные интеллектуальные и образовательные уровни пользователей. * Выходные данные понятны пользователю и отфильтрованы. Лишняя техническая информация не нужна. * Диагностическая информация и сообщения об ошибка понятны пользователю. * Доступ к разным опциям логичный и их количество сбалансировано(Хорошо продуманная программа отслеживает действия конкретных пользователей и предлагает им те пункты меню, которыми они пользуются чаще всего). * Система должны подтверждать любой ввод(Если для выбора элемента пользователю предоставляется список, то, после того как элемент выбран, его следует отобразить на экране. Кроме того, если для обработки выбранной команды требуется некоторое время, что часто происходит, когда осуществляется дистанционный доступ к системе, то в подобных случаях должно выводиться сообщение, информирующее пользователя о сути того, что происходит. Цель тестирования на этом уровне, которое иногда называют компонентным тестированием - проверка корректности работы и эффективности обратной связи с интерактивными компонентами системы) * Интерфейс должен быть продуман так, чтобы вероятность возникновения ошибок в результате неправильных действий пользователя была сведена к минимуму(? Один из возможных способов протестировать это заключается в подсчете количества ошибок, допускаемых каждым пользователем в процессе ввода данных или выбора опций программы). * Пользователю должно быть легко повторять ранее выполненные действия. * Пользователь должен чувствовать себя уверенно при навигации по пунктам меню.   В конечном счете тестирование удобства использования должно включать оценку того, соответствуют ли фактические функциональные возможности программы исходным спецификациям.  **Процесс тестирования удобства использования**:   1. Планирование - подготовка набора практических, связанных с реальной деятельностью, повторяющихся тестовых заданий для пользователя. 2. Назначение наблюдателей - документируют впечатления пользователей при выполнение тестовых заданий. 3. Написание подробных инструкций для пользователей. 4. Выбор типа пользователей, на которых ориентировано данное приложение(Например, медработники для медоборудования). Для приложений массового рынка обычно выбирают случайных пользователей. 5. Выбор количества пользователей. Формула для определения числа пользователей: , где 𝐸 - процент обнаруженных ошибок; 𝑛 - количество пользователей; 𝐿 - доля ошибок, обнаруживаемых одним пользователей. 6. Сбор данных - сбор результатов тестирования. Существую следующие методы:  * Комментирование вслух - видеозапись теста и запись наблюдателем комментариев пользователя, произносимых вслух. В ходе этого процесса участники теста описывают вслух свои задачи, высказывают замечания и делятся любыми соображениями, которые приходят им на ум по мере выполнения тестового сценария. * Дополнительные ПО для фиксации действий пользователя - актуально, если тестирование проходит удаленно(Плюсом является нахождение пользователя в привычной обстановке). Фиксируются нажатия клавиш пользователем и записывается время, необходимое пользователю для выполнения каждого тестового задания. * Отслеживание движения глаз - отслеживая движение глаз с помощью видеосистем или иных технологий, исследователи могут делать заключения о том, какие визуальные элементы привлекают внимание наблюдателя, в какой очередности и на какой период времени. * Анкетирование - В анкету включаются вопросы, ответы на которые поддаются количественной оценке и анализу по всей пользовательской базе. Например, Формулируется серию вопросов, на каждый из которых пользователь должен ответить, указав число в диапазоне от 1 до 5, где 5 означает полное согласие, а 1 — полное несогласие.   **ГОСТ Р ИСО 9241-210-2012 -** содержит руководство по человеко-ориентированному проектированию компьютерных интерактивных систем. В нем рассмотрены способы улучшения взаимодействия человек—система за счет аппаратных и программных компонентов интерактивных систем.  В разделе 4.4(Улучшение проекта за счет его оценки пользователями) отмечено: отзывы пользователей являются важным источником информации при человеко-ориентированном проектировании. Оценка проекта с участием пользователей и его улучшение на основе отзывов пользователей является эффективным средством минимизации риска несоответствия системы нуждам пользователей или организации-заказчика (включая трудно выявляемые требования). |
| --- | --- |
| **23. Гибкое тестирование**  По сути, **гибкое тестирование** — это форма совместного тестирования, характеризующаяся тем, что в процессы планирования, проектирования и выполнения тестов вовлекаются все участники проекта. Заказчики тоже принимают участие в подготовке приемочных тестов, определяя варианты использования и атрибуты программы. Разработчики совместно с тестировщиками создают тестовые драйверы,обеспечивающие автоматическое тестирование функциональности. Методология гибкого тестирования подразумевает интенсивное общение участников и коллективную форму работы. **Основные проблемы традиционного тестирования:**  Запаздывающая обратная связь, Высокая стоимость исправления ошибок на поздних этапа, • Несоответствие динамично меняющимся требованиям. **Agile Testing предлагает решение этих проблем** через непрерывное тестирование на всех этапах разработки. Cразу же после того, как разработчики получат стабильный базовый код, заказчики должны приступить к его приемочному тестированию и предоставить ответную информацию команде разработчиков. Это также означает, что тестирование не является отдельным этапом, а, скорее, “встраивается” в разработку, обеспечивая непрерывное протекание всего процесса. Чтобы заказчик получил действительно стабильный продукт, пригодный для последующего приемочного тестирования, разработчики сначала пишут модульные тесты для каждого модуля и лишь затем приступают к его кодированию. Поскольку тесты отражают требования, предъявляемые к модулям, то совершенно очевидно, что изначально они не должны проходить, так как соответствующий код еще не написан. Как это ни парадоксально, но выходит гибкое тестирование основывается на следующих ключевых принципах:• **Непрерывность**: Тестирование интегрировано в каждый спринт, **Раннее тестирование**: Тесты создаются до или параллельно с кодом, **Автоматизация:** Приоритет автоматизированным тестам , **Коллективная ответственность**: Вовлечение всей команды в процесс тестирования.  Обычно в agile-средах работают небольшие команды, в которых разработчики одновременно являются и тестировщиками. В крупных проектах, располагающих большими ресурсами, эту работу может выполнять отдельный тестировщик или группа специалистов по тестированию. В любом случае тестировщиков не следует воспринимать как “обвинителей”. Их задача состоит в том, чтобы содействовать продвижению проекта путем предоставления ответной информации о качестве программного обеспечения, чтобы ее можно было использовать для устранения дефектов программного обеспечения, изменения требований к нему и внесения необходимых улучшений. Гибкое тестирование хорошо вписывается в методологию экстремального программирования, в соответствии с которой разработчики сначала пишут модульные тесты и только потом — программное обеспечение. Оставшаяся часть билета посвящена более подробному изучению концепций экстремального программирования и экстремального тестирования.  **Экстремальное программирование** (XP) вносит следующие важные аспекты в процесс тестирования: Модульные тесты создаются до написания кода, Приемочные тесты разрабатываются совместно с заказчиком, Непрерывный рефакторинг с обязательной проверкой тестами, Парное программирование как способ повышения качества кода, Основной объем усилий приходится на модульное тестирование. Модульные тесты создаются с той целью, чтобы программное обеспечение не могло их пройти. Постоянное тестирование также приносит то дополнительное нематериальное благо, о котором уже упоминалось, — уверенность( и разработчиков и заказчика).  Экстремальное приемочное тестирование: Приемочные тесты создаются совместно с заказчиком на этапах проектирования и планирования. В отличие от других разновидностей тестирования, которые мы до этого обсуждали, приемочные тесты выполняются заказчиками, а не вами или вашими коллегами-программистами. Это дает заказчику возможность непредвзято проверить, удовлетворяет ли приложение его потребностям. Заказчики создают приемочные тесты на основе пользовательских историй. Обычно между пользовательскими историями и приемочными тестами существуют отношения “один ко многим”, т.е для каждой пользовательской истории может потребоваться более одного теста. Приемочные тесты в XT можно автоматизировать, а можно и не автоматизировать .Любое отклонение считается дефектом, о чем сообщается команде разработчиков. Важно отметить, что программа может проходить модульные тесты, но не проходить приемочные. Почему это может произойти? Потому что модульный тест проверяет, удовлетворяет ли программа требованиям определенной спецификации, а не полноту реализации заданной функциональности или эстетические характеристики программы.  **Преимущества:** Быстрая обратная связь, Снижение стоимости исправления дефектов, Лучшее соответствие требованиям заказчика, Повышение качества продукта  **Недостатки:** Высокие требования к квалификации команды, Необходимость в инструментах автоматизации, Сложность масштабирования на большие проекты, Зависимость от вовлеченности заказчика | **24. Основные положения стандарта ISO/IEC 12207, требования к тестированию верхнего и нижнего уровня.**  В прошлом семе я скатал про 34 ГОСТ всё, что смог, прямо со всем содержанием, деду понравилось, он поставил 5, поэтому скатывайте и не стесняйтесь. (Много получилось, можно что-нибудь скипнуть или сократить)  **ISO/IEC 12207 (ГОСТ Р ИСО/МЭК 12207)** - стандарт ISO и IEC, описывающий процессы жизненного цикла программного обеспечения. ISO (International Organization for Standardization) - Международная организация по стандартизации. IEC (International Electrotechnical Commission) - Международная электротехническая комиссия.  **Общие положения: Область применения (1.1)** Стандарт устанавливает общую структуру для всех этапов жизненного цикла программного продукта — от приобретения до прекращения его использования. Он охватывает различные виды деятельности, такие как поставка, разработка, сопровождение и завершение применения программного продукта. Стандарт применим как внутри организаций, так и между различными сторонами (например, покупателями и поставщиками).  **Общие положения: Назначение (1.2)** Стандарт предназначен для облегчения взаимодействия между сторонами, которые участвуют в жизненном цикле программного продукта. Стороны включают не только поставщиков и покупателей, но и разработчиков, пользователей, менеджеров и других участников. Стандарт помогает согласовать процессы, используемые при разработке, поставке и эксплуатации программных продуктов.  **Общие положения: Ограничения (1.3)** Стандарт не детализирует конкретные методы или процедуры, которые должны использоваться для выполнения процессов. Он не задает жесткие требования к  документированию, например, к формату или содержанию документации. Решение о  том, как именно применять стандарт, остается за пользователем. Стандарт также не  устанавливает модель жизненного цикла или методы разработки, оставляя выбор этих  аспектов за организациями, использующими данный стандарт.  **Организация стандарта (5.2)** Стандарт группирует различные виды деятельности, которые могут выполняться в течение жизненного цикла программных систем, в семь групп процессов. Каждый из  процессов жизненного цикла в пределах этих групп описывается в терминах цели и  желаемых выходов, списков действий и задач, которые необходимо выполнять для  достижения этих результатов. (5.2.1) \**Думаю определения групп можно не переписывать, ему хватит и оглавления, но на всякий случай оставлю\**  **1.** **процессы соглашения** - определяют действия, необходимые для выработки соглашений между двумя организациями, **2.** **процессы организационного обеспечения проекта** - осуществляют менеджмент возможностей организаций приобретать и поставлять продукты или услуги через инициализацию, поддержку и управление проектами. Эти процессы обеспечивают ресурсы и инфраструктуру, необходимые для поддержки проектов, и гарантируют удовлетворение организационных целей и установленных соглашений., **3. процессы проекта** - охватывают планирование, выполнение, оценку и управление проектом, **4.** **технические процессы** - охватывают определение требований, преобразование их в продукт, применение, поддержку и изъятие продукта, обеспечивая его функциональность, качество, безопасность, соответствие законодательству и снижение рисков, а также включают процессы от анализа требований до сопровождения и снятия продукта с эксплуатации., **5.** **процессы реализации программных средств** - используются для создания конкретного элемента системы (составной части), выполненного в виде программного средства. Эти процессы преобразуют заданные характеристики поведения, интерфейсы и ограничения на реализацию в действия, результатом которых становится системный элемент, удовлетворяющий требованиям, вытекающим из системных требований, **6.** **процессы поддержки программных средств** - включают управление документацией, конфигурацией, качеством, верификацию, валидацию, ревизию, аудит и решение проблем., **7. процессы повторного применения программных средств** - поддерживают возможности организации использовать повторно составные части программных средств за границами проекта.  **Описание процессов (5.1.9)** Каждый процесс стандарта описывается в терминах следующих атрибутов: 1. **наименование** — передает область применения процесса как целого; 2. **цель** — описывает конечные цели выполнения процесса; 3. **выходы** — представляют собой наблюдаемые результаты, ожидаемые при успешном выполнении процесса; 4. **деятельность** — является перечнем действий, используемых для достижения выходов; 5. **задачи** — представляют собой требования, рекомендации или допустимые действия, предназначенные для поддержки достижения выходов процесса. \**По такой схеме в ГОСТе описаны суммарно около 40 процессов\**  **Тестирование верхнего и нижнего уровня** \*я бы про это вообще не писал, а ток про гост, дед всё равно забудет, это тема как будто случайно в этот вопрос попала, но если хотите написать, то вот из моего доклада, он доклад видел, сильно не докапывался и вопросов не задавал, но всё равно это мои выдумки и гпт - в книжке про это нету\*  ■ Верхний уровень (системное тестирование) - проверка системы в целом, включая интеграцию всех компонентов, взаимодействие с внешними сервисами и соответствие пользовательским требованиям.  ■ Нижний уровень (модульное тестирование) - проверка отдельных компонентов системы (функций, классов, модулей) в изоляции от других частей.  **Требования к тестированию нижнего уровня**  1. Изоляция модулей: Каждый модуль должен тестироваться изолированно от других модулей. Для этих целей используются заглушки или драйверы.  2. Покрытие кода: Должны быть покрыты все возможные пути выполнения кода. Проверяются граничные значения и исключительные ситуации.  3. Автоматизация: Тесты должны быть автоматизированы и выполняться быстро, чтобы их можно было часто запускать.  4. Независимость: Тесты не должны зависеть друг от друга и от порядка выполнения.  **Требования к тестированию верхнего уровня**  1. Проверка соответствия спецификации.  2. Проверка интеграции: Проверка корректности взаимодействия всех компонентов системы и внешних сервисов.  3. Нефункциональные требования: Тестирование производительности, безопасности, отказоустойчивости и удобства использования.  4. Реальное окружение: Тестирование в условиях, максимально приближенных к реальным.  5. Реальные сценарии: Тестирование должно производиться на основе реальных пользовательских сценариев использования системы. |

| **25. Верификация программного обеспечения**  **Верификацией ПО** называется проверка соответствия результатов отдельных этапов разработки программной системы требованиям и ограничениям, сформулированным для них на предыдущих этапах. Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО решений другим, ранее созданным или используемым в качестве исходных данных, а также  соответствие этих решений и процессов их разработки правилам и стандартам.  **Верификация** — подтверждение на основе представления объективных свидетельств того, что установленные требования были выполнены (стандарт ИСО 9000:2000).  **Области применения.**  Верификация программного обеспечения — это очень дорогостоящий  процесс, и она оправдана в тех случаях, где цена ошибки значительно выше  возможных затрат на ее проведение. Примером могут служить программы, которые  запускаются только один раз, например ракеты, или системы, где ошибка  приводит к непоправимым последствиям.  **Методы верификации.**  Тестирование направлено на выявление несоответствий спецификаций в  программе. Однако тестирование не может показать соответствие спецификации,  этим занимается как раз верификация. На данный момент широко распространены два типа верификации программ. Первым является дедуктивный анализ, вторым — проверка на модели.  **Дедуктивный анализ. (Формальный метод)**  Для дедуктивного анализа спецификация и допустимые входные данные или начальные условия работы должны быть сформулированы на языке логики предикатов. Основой дедуктивного анализа является триада Хоара. Триада Хоара состоит из трех элементов: предусловия, программы и постусловия. Они связаны следующим образом: если выполнено предусловие, то после выполнения программы выполнится постусловие. Самой программой может выступать отдельный оператор, модуль или целая программа. В последнем случае предусловием будут являться допустимые входные значения, а постусловием спецификация. Верификация сводится к доказательству следования постусловия из предусловия и программы. Проблема в том, что доказательство почти полностью ложится на человека. Человеческий фактор может принести большие неприятности. Однако, существуют автоматические средства проверки или частичной проверки уже найденных доказательств. Примером использования дедуктивного анализа для доказательства работы целостной системы является разработанная микроядро операционной системы.  **Проверка модели. (Формальный метод)**  При этом подходе спецификация программы описывается при помощи  темпоральных логик. Для программы или для цифровой аппаратуры строится  модель в виде структуры Крипке. Структура Крипке — помеченный граф, каждая  вершина которого является состоянием системы. Очевидным ограничением, вытекающим отсюда, является конечность системы, то есть количество состояний системы должно быть ограничено.  Переходы между состояниями отражают логику работы программы. После того как  такой граф построен на нём можно проверить выполнимость спецификации при  помощи алгоритмов разметки структуры Крипке по формуле. Замечательно то, что и процесс построения структуры Крипке, и проверка формулы могут быть выполнены автоматически.  **Преимущества и недостатки.**  Верификация заключается в формальной проверке того, что для модели системы выполняются формально определенные свойства. **Преимуществом верификации** является то, что проводится исчерпывающая проверка всех возможных режимов работы модели системы, в том числе и тех, которые в реальном функционировании маловероятны, а потому и не могут быть проверены тестированием. Этот анализ проводится на ранних этапах разработки. **Недостатком верификации** является то. что проверяется не реальная система, а ее модель, которая может и не отражать существенных свойств поведения реальной системы. Далее, поскольку невозможно формально определить, что означает "полное отсутствие ошибок", верификация также не может гарантировать этого свойства у системы.  **Кратко про остальные методы.**  1) **Проверка согласованности (Формальный метод)** — анализ соответствия между двумя исполнимыми моделями, одна из которых моделирует проверяемый проект или реальную работу системы, а вторая — проверяемые свойства. 2) **Абстрактная интерпретация (Формальный метод)** — анализ на основе упрощенной модели программы. 3) **Тестирование (Динамический метод)** — проверяемое ПО выполняется в рамках заранее подготовленных сценариев. 4) **Мониторинг (Динамический метод)** — наблюдение, запись и оценка результатов работы ПО при его обычном использовании. 5) **Профилирование (Динамический метод)** — контроль операций с памятью и взаимодействия параллельных потоков и процессов в системе.  **Формальные методы** основаны на математическом моделировании программ и требований к ним.  **Динамические методы** анализируют и оценивают свойства ПО по результатам его реальной работы или работы некоторых моделей и прототипов. | **26. Валидация программного обеспечения**  **Валидация –** процесс подтверждения соответствия конечного продукта нуждам пользователя и возможность его применения для конкретных задач. Отвечает на вопрос: правильный ли продукт мы делаем? То есть на выходе получим ли мы правильный продукт?  **Валидация** (определение из ИСО 9000/2000) – подтверждение на основе представления объективных свидетельств того, что требования, предназначенные для конкретного использования или применения, выполнены. Используется для обозначения соответствия объекта (проекта, продукции, процесса) установленным требованиям. Имеет характер утверждения объекта (разрешения) для последующего использования.  Другими словами, **валидация** – процесс проверки соответствия ПО функциональным и нефункциональным требованиям и ожидаемым потребностям заказчика. С помощью валидации можно быть уверенным в том, что создан «правильный» продукт. Продукт, который полностью удовлетворяет заказчика.  **Процесс процесса валидации**: убедиться, что специфические требования для программного продукта выполнены, и осуществляется это с помощью:   * разработанной стратегии и критериев валидации для всех рабочих продуктов; * оговоренных действий по проведению валидации; * демонстрации соответствия разработанных программных продуктов требованиям заказчика и правилам их использования; * согласования с заказчиком полученных результатов валидации.   **Отличие валидации от верификации:**  Основные задачи процессов верификации и валидации состоят в том, чтобы проверить и подтвердить, что конечный программный продукт отвечает назначению и удовлетворяет требованиям заказчика. Эти процессы взаимосвязаны и определяются, как правило, одним общим термином "верификация и валидация" или (V&V). Если берется продукт, проверяется, исследуется, то есть тестируется на соответствие требованиям, этот процесс называется **верификацией**. Если продукт тестируется на соответствие, насколько он подходит самому заказчику, этот процесс называется **валидацией.** **Заказчика** интересует в первую очередь **валидация**, то есть исполнение собственных требований. А **исполнителя**, то есть нас, нашу компанию, волнует не только соблюдение всех норм качества, то есть верификация при реализации продукта, но и соответствие всех особенностей продукта желаниям заказчика.    **Валидация** используется для проверки того, насколько продукция, система или процессы отвечают требованиям клиента. В отличие от **верификации**, где производитель оценивает продукт на формальное соответствие техническим характеристикам, здесь выясняют, выполняет ли он свое предназначение. Технологическая линия может прекрасно выглядеть и отвечать всем требованиям к таким линиям, но выдавать бракованный продукт из-за мелкой и незаметной на первый взгляд неисправности. Валидация же позволяет провести внешний контроль качества.  Рассмотрим валидацию на примере сайта. Заказчик сформировал требования, по которым необходимо создать данный сайт.    Разработчики реализовали кнопку и программа по своей сути прошла верификацию. Сервис имеет кнопку проверки файла, кнопка имеет название проверить, при нажатии на кнопку запускается механизм проверки файла. Однако кнопку <<Проверить>> расположили за полем вывода результатов проверки, и в итоге кнопка перекрывается данным полем.    Текущая реализация не соответствует требованиям и ожиданиям заказчика. Формально требования реализованы, хотя в требованиях не были указаны, как и где должна располагаться кнопка проверки и как точно на нее надо было нажимать.  Поэтому при проработке требований аналитик организации разработчика обязан уточнить информацию у заказчика, а также все нюансы, которые он встретит в требованиях. Для задания четких требований нужен “Национальный стандарт РФ ГОСТ Р ИСО/МЭК 25010-2015. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов” (след.билет)  Если кнопка будет расположена не за полем вывода результатов, будет располагаться под ним, не будет прикрыта другими элементами интерфейса, то система по сути пройдет валидацию, так как система соответствует требованиям и ожиданиям заказчика и способна реализовать свое предназначение. |
| --- | --- |
| **27**. **Методы тестирования ПО для заинтересованных сторон, ГОСТ Р ИСО/МЭК 25010-2015.**  При разработке ПО можно выделить три заинтересованных стороны, имеющих различное отношение к разрабатываемому ПО:   * **Разработчик**; * **Конечный пользователь** (основной пользователь - лицо, взаимодействующее с системой для достижения основных целей); * **Вторичный пользователь ПО**, осуществляющий его эксплуатацию или поддержку - провайдер, установщик, специалист по обслуживанию и т.п. (иногда выделяют четвертую сторону - **косвенный пользователь** - тот, кто не использует продукт, но кого касаются результаты его работы).   Каждая из сторон обладает собственным уровнем компетентности и собственными интересами. **Задача ГОСТа** состоит в том, чтобы **учесть интересы каждой стороны при определении критериев качества продукта**, а также **разработать набор четких (измеримых) критериев качества продукта.**  Настоящий стандарт (**ГОСТ Р ИСО/МЭК 25010-2015)**  определяет:  **a)** **модель качества при использовании**, в состав которой входят пять характеристик, некоторые из которых, в свою очередь, подразделены на подхарактеристики. Эти характеристики касаются результата взаимодействия при использовании продукта в определенных условиях. Данная модель применима при использовании полных человеко-машинных систем, включая как вычислительные системы, так и программные продукты;  **b)** **модель качества продукта**, в состав которой входят восемь характеристик, которые, в свою очередь, подразделены на подхарактеристики. Характеристики относятся к статическим и динамическим свойствам программного обеспечения и вычислительных систем. Модель применима как к компьютерным системам, так и к программным продуктам.  Пункт 3 в ГОСТe. **Основы модели качества.**  **Качество системы** — это степень удовлетворения системой заявленных и подразумеваемых потребностей различных заинтересованных сторон, которая позволяет, таким образом, оценить достоинства. Эти заявленные и подразумеваемые потребности представлены в международных стандартах серии SQuaRE посредством моделей качества, которые представляют качество продукта в виде разбивки на классы характеристик, которые в отдельных случаях далее разделяются на подхарактеристики.  Измеримые, связанные с качеством свойства системы называют **свойствами качества**, связанными с соответствующими показателями качества. Чтобы прийти к показателям характеристики или подхарактеристики качества в случаях, когда характеристика или подхарактеристика не может быть непосредственно измерена, необходимо идентифицировать подмножество свойств, которое в совокупности покрывает характеристику или подхарактеристику, получить показатели качества для каждого свойства и, объединив их в вычислительном отношении, достигнуть полученного показателя качества, соответствующего характеристике или подхарактеристике качества.    Характеристики и подхарактеристики качества могут быть количественно определены с помощью функции измерения. **Функция измерения** -это алгоритм, используемый для объединения элементов показателя качества. Результат применения функции измерения называют показателем качества ПО.  К настоящему времени в серии SQuaRE имеются три модели качества: **модель качества при использовании** и **модель качества продукта**, определенные в настоящем стандарте, и **модель качества данных**, определенная в ИСО/МЭК 25012. Совместное использование моделей качества дает основание считать, что учтены все характеристики качества. Данные модели обеспечивают множество характеристик качества, в которых заинтересован широкий круг лиц, таких как: разработчики ПО, системные интеграторы, приобретатели, владельцы, специалисты по обслуживанию, подрядчики, профессионалы обеспечения и управления качеством и пользователи.  **Модель качества при использовании** определяет пять характеристик, связанных с результатами взаимодействия с системой: **результативность, производительность, удовлетворенность, свободу**  **от риска и покрытие контекста**. Качество при использовании системы **характеризует воздействие продукции** (система или программный продукт) на заинтересованную сторону. Оно определяется качествами программного обеспечения, аппаратных средств, операционной среды, а также характеристиками пользователей, задач и социальной среды. Все эти факторы вносят свой вклад в качество системы при использовании.  **Модель качества продукта** сводит свойства качества системы/программного продукта к **восьми характеристикам**, которыми являются: функциональная пригодность, уровень производительности, совместимость, удобство пользования, надежность, защищенность, сопровождаемость и переносимость (мобильность). Каждая характеристика, в свою очередь, состоит из ряда соответствующих подхарактеристик.  **Целью модели качества продукта** является компьютерная система, в которую входит целевой программный продукт, а **цель модели качества при использовании** — это совокупная человеко-машинная система, которая включает в себя и целевую компьютерную систему, и целевой программный продукт.    Модели качества продукции и качества при использовании могут быть использованы **для определения требований, выработки показателей и выполнения оценки качества**.  Требования к качеству должны быть определены с точки зрения заинтересованных лиц до разработки или приобретения программного обеспечения. Результатом анализа требований к использо ванию будут определенные требования к функциональности и качеству продукции, необходимые для достижения требований к использованию.  **Модели качества обеспечивают основу для сбора требований заинтересованных сторон.**  **Заинтересованная сторона** — это следующие три типа пользователя:  1. **Основной пользователь** — лицо, взаимодействующее с системой для достижения основных целей.  2. **Вторичные пользователи** — лица, осуществляющие поддержку, например:  a) провайдер контента, системные инженер/администратор, руководитель безопасности;  b) специалист по обслуживанию, анализатор, специалист по портированию, установщик.  3. **Косвенный пользователь** — лицо, которое получает результаты, но не взаимодействует с системой.  Некоторые термины (из ГОСТа): **Свобода от риска -** способность продукта или системы смягчать потенциальный риск для экономического положения, жизни, здоровья или окружающей среды.  **Покрытие контекста** - степень, в которой продукт или система могут быть использованы с эффективностью, результативностью, свободой от риска и в соответствии с требованиями как в первоначально определенных условиях использования, так и в условиях, выходящих за спецификации.  **Функциональная пригодность** - степень, в которой продукт или система обеспечивают выполнение функции в соответствии с заявленными и подразумеваемыми потребно стями при использовании в указанных условиях. | **28. Тестирование ПО систем реального времени**  Система реального времени — это система, которая должна обрабатывать события и выдавать результаты в строго заданные временные рамки, соответствующие реальному физическому времени. Важным отличием является критичность соблюдения временных ограничений для обеспечения корректной работы системы. Например, антиблокировочная система тормозов (ABS) в автомобиле, в ней критически важна задержка в работе системы.  Тестирование ПО таких систем— это процесс проверки компьютерных систем, которые должны обрабатывать данные и реагировать на события в строго определенные временные рамки. В отличие от обычного тестирования, здесь важно не только отсутствие несоответствий с спецификацией, но и соблюдение временных ограничений.  Особенности тестирования ПО реального времени:  1. Обязательная верификация соблюдения временных ограничений  2. Тестирование наихудшего времени выполнения  3. Проверка корректности обработки прерываний в заданные временные рамки  4. Валидация работы планировщика задач в условиях жесткого реального времени  5. Тестирование реакции системы на критические события с гарантированным временем отклика  **Методология тестирования**  Разработка тестов для систем реального времени включает четыре ключевых этапа:  1. Тестирование отдельных задач  - Проверка логики и синтаксиса каждой задачи в изоляции  - Использование статических методов анализа кода  - Пример: проверка модуля расчёта подачи топлива в двигателе (без замера времени на этом этапе)  2. Поведенческое тестирование  - Моделирование реакции системы на внешние события  - Использование автоматизированных инструментов тестирования  - Пример: тестирование системы удержания высоты дрона при изменении внешних условий.  3. Межзадачное тестирование  - Проверка взаимодействия между параллельными задачами  - Тестирование с различными скоростями передачи данных  - Пример: проверка синхронизации задач в системе управления лифтами многоэтажного здания  4. Системное тестирование  - Интеграционное тестирование программного и аппаратного обеспечения  - Полная проверка всех системных требований  - Пример: тестирование бортового компьютера автомобиля в условиях, приближенных к реальным  **Нагрузочное тестирование**  Нагрузочное тестирование особенно важно для систем реального времени, так как позволяет проверить:  - Работу в условиях пиковых нагрузок  - Реакцию на нештатные ситуации  - Устойчивость к превышению расчетных параметров  Примеры нагрузочного тестирования:  1. Система управления воздушным движением:  - Имитация одновременного появления 200+ самолетов в зоне контроля  - Тестирование реакции на экстренные ситуации (резкое изменение курса нескольких самолетов)  2. Мобильное приложение: (не уверен, не тригернет ли его этот пример и ниже с банком)  - Тестирование при низкой скорости интернета  - Проверка работы при частом переключении между Wi-Fi и мобильным интернетом  3. Промышленные системы:  - Имитация сбоя датчиков на производственной линии  - Проверка реакции на резкие скачки напряжения  Практические примеры тестирования  1. Банковские системы:  - Тестирование обработки транзакций в условиях высокой нагрузки (например, в час пик)  - Проверка реакции на одновременные запросы от тысяч пользователей  - Пример: тестирование системы онлайн-банкинга в день выплаты зарплат  2. Медицинское оборудование:  - Проверка работы аппарата ИВЛ при различных сценариях дыхания пациента  - Тестирование системы мониторинга в условиях помех  3. Транспортные системы:  - Проверка работы светофорного регулирования при аварийных ситуациях  - Тестирование системы автоматического торможения в различных дорожных условиях  В системах реального времени задержка — это не просто ошибка, а угроза. Ошибка в 100 мс может стоить жизни (авиация, медицина, авто) или привести к катастрофе (АЭС, промышленность). Здесь неприменимо «работает в среднем» — система должна быть отрабатывать за приемлемое время в худших случаях, включая экстремальные условия. |

| **29.** **Тестирование ПО автономных систем**  Автономные системы – это системы с распределенными вычислительными ресурсами с самоуправляемыми характеристиками, адаптирующиеся к непредсказуемым изменениям и скрывающие внутреннюю сложность от операторов и пользователей. Критерии автономной системы согласно IBM:   * Система должна знать о себе с точки зрения того, к каким ресурсам у нее есть доступ, каковы ее возможности и ограничения, а также как и почему она подключена к другим системам. * Система должна быть способна автоматически настраивать и реконфигурировать себя в зависимости от меняющейся среды. * Система должна быть способна оптимизировать свою производительность, чтобы обеспечить наиболее эффективный вычислительный процесс. * Система должна быть способна устранять возникающие проблемы. * Система должна обнаруживать, идентифицировать и защищать себя от различных типов атак для поддержания общей безопасности и целостности системы. * Система должна быть способна адаптироваться к изменяющейся среде, взаимодействуя с соседними системами. * Система должна предвидеть спрос на свои ресурсы, оставаясь при этом прозрачной для пользователей.   Основные сложности при тестировании автономных систем:   * Большие разнообразие и объем входных данных. * Очень сложное ПО с большим числом внутренних связей и внешних зависимостей * Применение машинного обучения, приводящего к изменению реакции на одинаковые входные данные с течением времени.   В состав автономной системы входит множество различных программных и аппаратных компонентов, тестируемых в том числе и по отдельности. На картинке представлена схема типов тестирования систем автопилотов автомобилей, применимая к автономным системам в целом. Формируются несколько видов тестовых систем: Моделирование модели в цикле (Model-in-the-Loop (MIL) simulation), Моделирование программного обеспечения в цикле (Software-in-the-Loop (SIL) simulation) и Моделирование аппаратного обеспечения в цикле (Hardware-in-the-Loop (HIL) Simulation).    Для ранних этапов тестирования в реальных условиях применяется симуляция на стенде, представляющем собой комбинацию аппаратных и программных средств ввода и вывода данных, подключенных к тестируемым компонентам. Стенд позволяет имитировать различные состояния тестируемой системы, испытывая различные компоненты программного и аппаратного обеспечения.  Отметим, что модели могут быть как моделями, разработанными в специальных средах (MATLAB, Simulink), так и реальными объектами материального мира.  Первым этапом является MIL, который в некотором роде выполняет роль проверки корректности модели. Выполняется проверка в первую очередь общей логики на основе построенной мат. модели. Сначала строится модель реальной установки, потом строится модель ее контроллера. Проверяется, как контроллер управляет самой установкой. Важно отметить, что на данном этапе данные вводятся вручную, а модель разрабатывается в специальных средах (например, Simulink). Результаты документируются.  На этапе SIL, который является следующим, выполняется замена блока контроллера с модели, управляемой вручную, на модель, управляемую разработанным ПО. На данном этапе все еще невозможно тестирование в реальном времени, так как управляемый объект управляется отдельно и в реальном времени не работает. На данном этапе проводится проверка того, как разработанное ПО воспринимает и передает данные (то есть как программа управляет установкой). При этом результаты также документируются и сравниваются с результатами полученными на предыдущем этапе. Если есть большие различия, то вносятся изменения либо в управляющее ПО, либо в модель на этапе MIL.  Заключительный этап HIL – проверка в реальном времени на реальной модели. Этот этап помогает определить проблемы, связанные с каналами связи и интерфейсом ввода-вывода, например, затухание и задержка, которые возникают из-за аналогового канала и могут привести к нестабильной работе контроллера. Это поведение не может быть зафиксировано при моделировании.  Рассмотрим пример: пусть тестируется автономный дрон, выполняющий задачи инспекции и съемки. Сначала, в рамках этапа MIL, будет смоделирована управляющая логика (построена мат. модель). В качестве аппаратной части будем использовать экран, показывающий текущее решение, принятое логикой, и клавиатуру, на которой вводятся параметры полета.  Переходя к SIL, следующему этапу, мы заменим программную составляющую стенда, сохранив аппаратную. Вместо мат. модели, использованной ранее, подключим полноценный программный модуль, управляющий навигацией дрона. На заключительном этапе, HIL, мы заменим аппаратную составляющую на реальный прототип дрона, запуская его в тестовой среде с моделированием реальных условий полета.  Требования, которые должны предъявляться к стенду HIL и SIL: 1) стенд должен предоставить возможность получать, оценивать и сохранять тестируемые параметры системы; 2) стенд должен обеспечивать покрытие большого числа тестов. | **30. Основные задачи технологии тестирования Фаззинг**  **Фаззинг** — техника тестирования программного обеспечения, часто автоматическая или  полуавтоматическая, заключающаяся в передаче приложению на вход неправильных,  неожиданных или случайных данных. Предметом интереса являются падения и зависания,  нарушения внутренней логики и проверок в коде приложения, утечки памяти, вызванные такими  данными на входе. Фаззинг является разновидностью выборочного тестирования (англ. random  testing), часто используемого для проверки проблем безопасности в программном обеспечении и  компьютерных системах. В качестве входных данных при этом могут выступать обрабатываемые  приложением файлы, информация, передающаяся по сетевым протоколам, функции прикладного  интерфейса и т. д.  В основном этапы фаззинга состоят из:  1 - Анализ исследуемого приложения  2 - Разработка фаззера (опционально)  3- Генерация данных  4- Сам фаззинг  5 - Анализ результатов.  Есть также **метод “серого ящика”**:  Это комбинация структурного и функционального тестирования. При тестировании данным методом исследователь не имеет полной спецификации программы и исходных кодов, как это бывает при тестировании методом «белого ящика», однако знаний о системе больше чем при тестировании методом «черного ящика».  Фаззинг берет лучшее от двух существующих подходов. Мы не ограничиваем себя конечным  количеством тестовых случаев и постоянно генерируем случайные входные данные для нашего  кода, но при этом каждая новая последовательность использует информацию из предыдущих  попыток для максимизации результата.  Сразу возникает резонный вопрос — могут ли случайные данные что-то протестировать в  действительности? Предположим, мы хотим проверить компилятор С/C++ и посылаем на вход  сгенерированные последовательности символов. Совершенно очевидно, что мы получим много  ошибок от компилятора на стадии лексикографического анализа и парсера выражений, но при  этом едва затронем оптимизацию и генерацию кода.  **Санитайзеры**: это инструменты для динамического тестирования, помогающие в поиске самых  разных ошибок в программах.  Повод для использования санитайзеров: переполнение буферов (глобальных, на стеке или в куче),  использование после освобождения, утечки памяти, обращение к неинициализированным  переменным.  Они также позволяют обнаруживать состояние гонки для потоков, ситуации взаимной  блокировки, обращения по нулевому указателю, деление на ноль (куда же без него),  переполнения для типов данных и некорректные битовые сдвиги.  Санитайзеры очень помогают во время отладки. Для их использования достаточно  скомпилировать исходники с включенной инструментацией, которая добавит специальные  команды в исполняемый файл. По ним можно будет следить за ходом выполнения программы и  состоянием памяти. При обнаружении ошибок будет сгенерирован отладочный вывод и  программа завершит работу.  Таким образом, мало просто сгенерировать фаззером некорректные входные данные, которые  обрушат программу. Ошибку следует устранить, а для этого надо собрать максимум информации.  Именно в этом и помогают санитайзеры. Их совместное использование повышает эффективность  тестирования.  **Типы фаззеров** есть - На основе грамматики, на основе мутаций, на основе покрытия кода, построение грамматики по данным.  на основе грамматики - грамматика построения входных данных (дает новые комбинации по правилам грамматики)  Мутация - случайным образом изменяет входные данные из  предыдущих попыток  Покрытие кода - по принципу генетического алгоритма и стремятся максимизировать  покрытие тестового кода. С практической точки зрения это один из самых эффективных на сегодня  типов фаззеров    Построение грамматики - комбинация на основе мутаций и грамматик.    Основные проблемы: 1 - уязвимость в архитектуре (сложно предугадать последствие) |
| --- | --- |
| **31. Требования к сертификации информационных систем, приказ ФСТЭК №76 от 02.06.2020**  Сетрификация информационных систем по Приказу №76 ФСТЭК — это процедура подтверждения соответствия системы требованиям безопасности информации. Она направлена на снижение рисков утечки, несанкционированного доступа и иных угроз. Приказ вступил в силу с 1 января 2021 года и обязателен для систем, обрабатывающих информацию ограниченного доступа, включая персональные данные и государственную тайну.  **Область применения**  Сертификация необходима для:  – Государственных организаций и учреждений;  – Объектов критической информационной инфраструктуры (КИИ), включая энергетику, транспорт, финансы, оборону, связь;  – Разработчиков отечественного программного обеспечения и СЗИ;  – Пользователей ИС, подлежащих контролю со стороны ФСТЭК.  Сертификация способствует формированию доверия к системам и минимизации рисков при их использовании в критически важных сферах.  **Этапы сертификации**  **Подача заявки**  Организация направляет заявку в аккредитованную лабораторию. Указываются характеристики системы, цели сертификации, область применения и уровень защищённости.  **Испытания**  Выполняются в условиях, приближенных к эксплуатационным. Включают:  – Анализ архитектуры и проектной документации;  – Проверку реализованных механизмов защиты;  – Тестирование устойчивости к типовым угрозам, включая НСД (несанкционированный доступ).  **Оформление результатов**  Составляется отчёт об испытаниях, на основании которого оформляется сертификат. Он содержит перечень защитных функций, класс защищённости, область применения и срок действия (до 5 лет).  **Контроль после сертификации**  Проводятся выборочные или плановые проверки для подтверждения, что система продолжает соответствовать требованиям. В случае нарушений сертификат может быть приостановлен или отозван.  **Требования к защищенности**  Системам присваивается класс защищенности (1–4), от которого зависят меры безопасности. Чем выше класс, тем строже требования. Включают:  – Идентификацию и аутентификацию пользователей;  – Контроль и разграничение прав доступа;  – Антивирусную защиту и анализ активности;  – Контроль целостности информации и программ;  – Регистрацию событий безопасности;  – Защиту сетевых взаимодействий;  – Резервное копирование и восстановление данных.  **Также обязательны:**  – Наличие модели угроз и нарушителя;  – Архитектура системы защиты;  – Полный комплект технической и эксплуатационной документации.  **Влияние на разработку и тестирование ИС**  Приказ №76 влияет на все стадии жизненного цикла ИС:  – **Проектирование**: архитектура системы должна включать встроенные механизмы защиты.  – **Разработка**: реализуются функции безопасности, ведётся документация.  – **Тестирование**: кроме функциональных тестов проводятся проверки устойчивости к сбоям, попыткам обхода защиты, соответствия требованиям моделей угроз.  – **Документирование**: все защитные функции должны быть проверены, протоколированы и связаны с требованиями.  **Пример проверок**  – Отработка попыток НСД;  – Проверка обработки инцидентов;  – Аудит защищённости каналов связи;  – Анализ логирования и восстановления после сбоев.  Приказ №76 — ключевой нормативный документ, формирующий единый подход к оценке защищённости ИС и СЗИ. Его выполнение позволяет организациям обеспечить соответствие требованиям ФСТЭК, минимизировать риски и повысить доверие к своим ИТ-решениям, особенно при взаимодействии с государственными и критическими инфраструктурами. | **32. Автоматизированное тестирование**:  **Автоматизированное тестирование** представляет собой процесс проверки программных продуктов с использованием специализированных инструментов и скриптов.Внедрение автоматизации позволило значительно упростить и ускорить процесс тестирования, минимизировать человеческий фактор. Автоматизация применяется на различных этапах разработки: от тестирования отдельных модулей до проверки производительности системы в условиях высокой нагрузки.  **Преимущества**   * Экономия времени: Быстрое выполнение тестов, особенно при повторных запусках. * Нагрузочное тестирование: Проверка производительности системы при больших нагрузках. * Минимум человеческого фактора: Снижение вероятности ошибок из-за невнимательности.   **Недостатки**   * Дороговизна: Высокие затраты на разработку и поддержку тестов. * UI-тестирование: Сложности с автоматизацией тестирования пользовательских интерфейсов. * Автоматизация не заменяет анализ, основанный на опыте тестировщика.   **Модульное тестирование (Unit Testing)** Модульное тестирование направлено на проверку отдельных частей кода — функций, методов или классов. Его цель — убедиться, что каждый модуль работает корректно и независимо от других компонентов системы.  *Преимущества*   * Раннее выявление ошибок на этапе разработки. * Упрощение отладки и рефакторинга кода.   *Инструменты*   * unittest: Встроенная библиотека Python для создания модульных тестов. Поддерживает написание тестов, организацию тестовых наборов и анализ результатов. * JUnit: Стандартный инструмент для модульного тестирования на Java. Широко используется в проектах на Java благодаря простоте и интеграции с IDE.   **Интеграционное тестирование (Integration Testing)** Интеграционное тестирование проверяет взаимодействие между различными модулями или компонентами системы. Его цель — обнаружить проблемы, возникающие при объединении частей системы.  *Пример сценария* Проверка взаимодействия базы данных и веб-сервиса.  *Преимущества*   * Проверка реальной работы системы в условиях интеграции.   *Инструменты*   * SoapUI: Инструмент для тестирования SOAP и REST сервисов. Поддерживает функциональное и нагрузочное тестирование API. * WireMock: Инструмент для создания заглушек (mock) REST API, что позволяет тестировать сервисы без зависимости от реальных серверов.   **Системное тестирование (System Testing)** Системное тестирование проверяет систему как единое целое, оценивая соответствие всем функциональным требованиям.  *Пример сценария* Проверка процесса входа в систему и обработки пользовательских данных *Преимущества*   * Проверка в условиях, близких к реальному использованию.   *Инструменты*   * Веб-приложения: Selenium: Лидер среди инструментов для автоматизации браузеров. Поддерживает Chrome, Firefox, Safari и другие браузеры. Интегрируется с языками программирования, такими как Python, Java, JavaScript. * Мобильные приложения: Appium: Кроссплатформенный инструмент для автоматизации тестирования приложений на iOS и Android. Использует API Selenium WebDriver, что упрощает переход от веб-тестирования к мобильному.   **Нагрузочное тестирование (Load Testing)** Нагрузочное тестирование оценивает производительность системы при увеличении количества пользователей или запросов. Его цель — убедиться, что система остается стабильной и быстрой при заданной нагрузке. 6.1  *Пример сценария* Проверка, как интернет-магазин обрабатывает 500 одновременных запросов на оформление заказа.  *Преимущества*   * Выявление узких мест и задержек.   *Инструменты*   * Apache JMeter: Многофункциональный инструмент для тестирования производительности. Поддерживает протоколы HTTP, FTP, JDBC и другие. Позволяет генерировать высокую нагрузку и анализировать результаты с помощью графиков и отчетов. * Locust: Инструмент для нагрузочного тестирования, написанный на Python. Позволяет описывать сценарии нагрузки в виде Python-кода, что упрощает настройку тестов. |
| **20. Критерии завершения тестов**  **Критерии завершения тестирования** представляют собой набор предопределенных условий, при достижении которых процесс тестирования считается выполненным в достаточном объеме для принятия решения о качестве ПО  **Регрессионное тестирование** — это вид тестирования, направленный на проверку того, что недавние изменения в коде (исправление ошибок, добавление новой функциональности) не привели к появлению новых дефектов или нарушению работы уже существующей функциональности.  **1) Проблема**: на практике условия окончания тестирования либо отсутствуют, либо являются размытыми.  «**Бесконечное тестирование**: При отсутствии четких условий окончания тестирования, процесс тестирования может затягиваться неопределенно долго. Команда производит поиск дефектов, когда как основные риски уже минимизированы. Это приводит к перерасходу бюджета и срыву сроков.  **Преждевременное прекращение тестирования**: завершение тестирования из-за давления сроков или ощущения «достаточности» проверок. Это чревато выпуском продукта с критическими дефектами, что влечет за собой репутационные потери, а также высокие затраты на исправление ошибок на стадии эксплуатации.  **Субъективность в принятии решений и конфликты интересов**: Без формальных критериев завершения принятие решения об окончании тестирования становится субъективным. Это может приводить к разногласиям между отделами и затруднять принятие согласованного решения.  **2) Ключевые типы компоненты критериев завершения**  Критерии завершения тестирования должны быть конкретными, измеримыми, достижимыми, релевантными и ограниченными по времени.  **Основные категории и составляющие таких критериев включают**:   * **Показатели покрытия**: Определяют, какая часть продукта или его функциональности была подвергнута тестированию. Сюда входит: покрытие требований, покрытие кода, покрытие тестовых сценариев. * **Показатели, связанные с дефектами**: Характеризуют текущее состояние продукта с точки зрения обнаруженных ошибок. Сюда входит: количество открытых дефектов, плотность дефектов. * **Временные рамки и бюджетные ограничения**: Завершение тестирования по истечении запланированного времени или бюджета, при условии достижения других качественных показателей. Этот критерий часто используется в комбинации с другими. * **Оценка рисков и стабильность продукта**: Уровень уверенности в том, что основные риски, связанные с качеством, минимизированы.   **3) Инструменты для отслеживания достижения критериев завершения**  Эффективный мониторинг достижения критериев завершения невозможен без использования специализированных инструментальных средств. Эти инструменты автоматизируют сбор данных, визуализируют прогресс и помогают принимать обоснованные решения:  **Системы управления тестированием**: Предоставляют централизованное хранилище для тестовых планов, тест-кейсов, результатов их выполнения и метрик покрытия требований. Примеры: TestRail, Zephyr, Qase.  **Инструменты для анализа покрытия кода**: Используются преимущественно с автоматизированными тестами для измерения процента кода, затронутого тестами. Примеры: JaCoCo (для Java), Coverage.py (для Python), gcov (для C/C++).  **Инструменты непрерывной интеграции и доставки** (CI/CD): Системы типа Jenkins, GitLab CI, TeamCity могут автоматически запускать тесты при каждом изменении кода и собирать метрики, включая результаты тестов и покрытие.  **Платформы для бизнес-аналитики**: Инструменты вроде Grafana, Tableau позволяют агрегировать данные из различных источников и представлять их в наглядном виде, облегчая визуальный контроль за достижением ключевых показателей.  **4) Взаимосвязь регрессионного тестирования и критериев завершения**  **Включение результатов регрессии в общие критерии**: успешное выполнение полного набора регрессионных тестов или критически важной его части является одним из критериев завершения тестирования.  **Оценка стабильности через регрессионные циклы**: Регулярные прогоны регрессионных наборов, особенно на поздних стадиях разработки, позволяют оценить стабильность продукта после исправлений и доработок.  **5) Выводы**  Четко сформулированные и согласованные критерии завершения тестирования преобразуют субъективные ощущения о «готовности» продукта в объективные, измеримые показатели, на основе которых можно принимать взвешенные решения.  **15. Высокоуровневое тестирование**  **Тестирование** - это проверка программы на соответствие требованиям, заданным в ТЗ.  **Высокоуровневое тестирование -** выявление ошибок искажения информации при переходе от одного этапа проектирования к другому.  **Внешняя спецификация** — это точное описание поведения программы с точки зрения конечного пользователя.  **1) Процесс разработки ПО и проблема трансляции информации**  Разработка — последовательная трансляция информации между этапами:    **Проблема**: Искажение информации при переходе между этапами — основная причина дефектов (70% ошибок возникают на стадиях анализа и проектирования).  ■ Требования определяют, для чего нужна программа.  ■ Цели устанавливают, что именно и насколько хорошо должна делать программа.  ■ Внешние спецификации дают точное описание способа представления программы пользователям.  ■ Документация, связанная с последующими процессами, опреде  **2) Подходы к предотвращению ошибок проектирования**   * Повышение точности разработки (выделить больше времени). * Верификация после каждого этапа — сравнение результатов этапа с предыдущим (например, целей и требований). * Связь тестирования с этапами разработки — каждому этапу соответствует свой тип тестирования для поиска ошибок при трансляции информации от этапа к этапу.   **3) Цели тестов**  ■ Цель функционального теста — продемонстрировать, что программа не соответствует внешним спецификациям.  ■ Цель системного теста — продемонстрировать, что свойства продукта не согласуются с первоначально сформулированными целями.  **4) Интеграционное тестирование**  Интеграционное тестирование, поскольку оно часто не считается отдельным этапом тестирования, а в случае использования инкрементного модульного тестирования выступает в качестве неявной составляющей модульного теста.  **5) Функциональное тестирование**  Сравнивает поведение программы с внешней спецификацией.  Используется стратегия «чёрного ящика»: **дана** спецификация программы, **требуется провести**  проверку соответствия программы заявленной спецификации. То есть нет никаких данных о том как устроена программа внутри.  Пример: Проверка корректности расчётов в калькуляторе согласно спецификации.  **6) Системное тестирование**  Является наиболее трудным для понимания и практического применения. Задача системного тестирование: сопоставить результат реализации системы или программы с первоначально сформулированными целями. Системные тесты проектируются на основе анализа документированных целей программы, а формулируются на основе анализа пользовательской документации:    Сложность составления системных тестов заключается в том, что нет общепризнанной методологии их написания. Однако системные тесты были поделены на 15 категорий. Не все они применимы к каждой программе в равной степени. Но для того, чтобы ничего не упустить, лучше исследовать возможность применения каждой из них:  **Вот перечисление части категорий:** возможности, предельные объемы данных, нагрузочное тестирование, безопасность, производительность. память, конфигурация, совместимость, установка, надёжность, документированность и восстанавливаемость  **7) Приемочное и тестирование установки**  Приемочное – оценка заказчиком соответствия программы первоначальным требованиям.  Тестирование установки – проверка корректности развертывания (настройки, конфигурация, сетевые подключения).  **8) Выводы**  Высокоуровневое тестирование обязательно требуется для больших систем: ошибки проектирования преобладают над ошибками кодирования. Каждый тип высокоуровнего тестирования решает уникальную задачу и требует специфических артефактов (спецификации, цели, документация). Эффективность зависит от формализации требований на ранних этапах. | **1. Основные задачи тестирования, классификация тестирования.**  **Тестирование** - это проверка продукта на соответствие требованиям, заданным в ТЗ.  **Требование** – потребность или ожидание, которое установлено, обычно предполагается или является обязательным  **1) Про тестирование в общем**  Требования выдвигает кто угодно со стороны заказчика. Нам бы неплохо эти требования формализовать. В целом требования делятся на формальные и неформальные. Формальные прописаны в ТЗ и их можно измерить. Исходя из того, что требования могут быть какими угодно, в разумных пределах, то и тесты, которые проверяют соответствия требованиям, также являются разнообразными. Если нам требуется, чтобы продукт мог работать при такой-то вибрации, то значит нужно сделать вибростенд, на котором можно будет проверить это соответствие продукта этому требованию. Важно понимать, что тестирование - довольно широкое понятие, которое относится к любому разрабатываемому продукту, к которому предъявлены требования, не только к программным продуктам. Тестирование – это не отдельная операция, а целый проект, который связан с аппаратурой, программным обеспечением, подготовкой тестовых задач с методами их измерения и порядком запуска.  **2) Основная задача тестирования** – проверка реализуется ли каждое требования в нужных ограничениях или нет.  **3) Программа испытаний**  Все тесты описываются в программе испытаний. Программа испытаний описывает то, каким образом производится проверка продукта на соответствие требованиям. Документ “программа и методика испытаний” должен содержать **следующие разделы**: Объект испытаний, цель испытаний, требования к программе, требования к программной документации, состав и порядок испытаний, методы испытаний.  **4) Классификация тестирования**  **4.1) Модульное тестирование** - процесс тестирования отдельных блоков, подпрограмм, классов или процедур, образующих крупную программу. Прежде чем тестировать программу в целом, необходимо сосредоточить внимание на меньших по размеру ее компонентах. Почему? - повышается эффективность тестирования сложных объектов, облегчается отладка (точная локализация и исправление обнаруженной ошибки) программ, позволяет распараллелить процесс тестирования.  **4.2) Интеграционное тестирование**  Второй уровень тестирования, на котором группа связанных модулей тестируется как интегрированный компонент. Целью является выявление проблем совместимости между модулями. Обычно интеграционное тестирование совмещено с модульным тестированием.  **4.3) Высокоуровневое тестирование -** выявление ошибок искажения информации при переходе от одного этапа проектирования к другому. **Проблема**: Искажение информации при переходе между этапами — основная причина дефектов (70% ошибок возникают на стадиях анализа и проектирования). В него входят такие подкатегории, как функциональное тестирование, системное тестирование, приемочное тестирование и тестирование установки.  **4.3.1) Функциональное тестирование**  Сравнивает поведение программы с внешней спецификацией.  Используется стратегия «чёрного ящика»: **дана** спецификация программы, **требуется провести**  проверку соответствия программы заявленной спецификации. То есть нет никаких данных о том как устроена программа внутри.  Пример: Проверка корректности расчётов в калькуляторе согласно спецификации.  **4.3.2) Системное тестирование**  Является наиболее трудным для понимания и практического применения. Задача системного тестирование: сопоставить результат реализации системы или программы с первоначально сформулированными целями. Системные тесты проектируются на основе анализа документированных целей программы, а формулируются на основе анализа пользовательской документации  **4.3.3) Приемочное тестирование и тестирование установки**  Приемочное – оценка заказчиком соответствия программы первоначальным требованиям.  Тестирование установки – проверка корректности развертывания (настройки, конфигурация, сетевые подключения).  **5) Выводы**  Тестирование представляет собой комплексный процесс верификации соответствия продукта установленным требованиям, формализованным в техническом задании. Эффективная организация тестов сокращает риски при разработке продукта. Ключевым фактором для составления качественных тестов является четкая формализация измеримых требований и методов их проверки.  **16. Функциональное тестирование**  ***Функциональное тестирование*** — это процесс, нацеленный на выявление расхождений между поведением программы и внешней спецификацией. (это НЕ демонстрация того, что поведение программы соответствует внешней спецификации. Функциональное тестирование фокусируется исключительно на том, *работает ли система правильно* .)  **Внешняя спецификация** — это точное описание поведения программы с точки зрения конечного пользователя.  (НЕфункциональное тестирование направлено на проверку качественных характеристик системы , таких как производительность, надежность, безопасность, удобство использования и т. д.)  Функциональное тестирование обычно выполняется в рамках стратегии “черного ящика”. При этом подразумевается, что достижение желаемых критериев покрытия логики по стратегии “белого ящика” обеспечивается предварительным проведением модульного тестирования.  Выполнению функционального тестирования предшествует анализ спецификации с целью получения набора тестов. Само же функциональное тестирование подразумевает, в частности, использование методов разбиения на эквивалентные классы, анализа граничных значений, выдвижения пред-  положений об ошибках и причинно-следственных диаграмм Этапы функционального тестирования1. Подготовка  * Анализ требований, технической документации. * Разработка плана тестирования и тест-кейсов. * Согласование сроков, количества итераций и возможных рисков с заказчиком. * Подготовка тестовой среды и данных.  2. Проведение  * Ручное или автоматизированное выполнение тест-кейсов. * Фиксация найденных багов в системе управления задачами (например, Jira, Trello). * Взаимодействие с командой разработки для уточнения и устранения проблем.  3. Отчетность  * Составление отчетов о проделанной работе. * Перечень выявленных ошибок с рекомендациями по их устранению   **Категории функционального тестирования (области проверки)**  Категории описывают что тестируется , то есть объект проверки и область применения функционального тестирования. Эти категории определяют, какие аспекты системы подвергаются анализу с точки зрения соответствия требованиям.  1. Тестирование пользовательского интерфейса (UI Testing) Проверка взаимодействия пользователя с элементами управления: кнопками, меню, формами и другими компонентами интерфейса. Цель — убедиться, что интерфейс интуитивно понятен и работает корректно.  2. Тестирование функциональности (Functionality Testing) Подтверждение того, что все функции программы работают согласно техническим требованиям и внешней спецификации.  3. Системное тестирование (System Testing) Проверка всей системы как единого целого в условиях, максимально приближенных к реальным. Включает тестирование взаимодействия между модулями и внешними системами.  4. Тестирование совместимости (Compatibility Testing) Оценка корректной работы программного обеспечения на различных устройствах, операционных системах, браузерах и аппаратных конфигурациях.  5. Тестирование безопасности (Security Testing) Проверка функциональных аспектов безопасности: аутентификация, авторизация, защита от SQL-инъекций, XSS и другие типы атак. Хотя формально относится к нефункциональному тестированию, имеет важные функциональные аспекты.  6. Тестирование локализации (Localization Testing) Проверка корректной адаптации программного обеспечения под языковые, культурные и региональные особенности: переводы, форматы дат, валют, календарей и других локализованных данных.  **Виды функционального тестирования (методы проверки)**  Виды описывают как именно проводится проверка , то есть методологию и подход к тестированию. Эти виды могут применяться в разных категориях.  1. Smoke-тестирование (Smoke Testing) Быстрая первоначальная проверка новой сборки для оценки её стабильности и готовности к дальнейшему тестированию. Обычно используется перед более глубокими проверками.  2. Sanity Testing Быстрая проверка ограниченного числа функций после небольших изменений или исправлений, чтобы убедиться в базовой работоспособности системы.  3. Регрессионное тестирование (Regression Testing) Проводится после изменений в коде (новые функции, багфиксы), чтобы убедиться, что изменения не нарушили ранее работавший функционал.  (4. Интеграционное тестирование (Integration Testing) Проверка взаимодействия нескольких модулей или сервисов по сквозным сценариям. Цель — убедиться, что система работает корректно как единое целое.  5. Приемочное тестирование (Acceptance Testing) Выполняется заказчиком или конечными пользователями для подтверждения того, что система соответствует бизнес-требованиям и готова к внедрению.)  (6. Модульное тестирование (Unit Testing) (ну это вроде как не совсем точно, что надо))  Проверка отдельных блоков кода (функций, методов) разработчиками с использованием тестовых примеров.) |