1. ***Тестирование: понятие, критерии. Требования к тестированию. Процесс проведения отладки.***

Тестирование ПО – это процесс проверки программного обеспечения на наличие ошибок и дефектов с целью обеспечения его надлежащего функционирования.

Критерии тестирования:

1. Функциональная корректность – проверка на выполнение всех функций и соответствие ожидаемому поведению;
2. Надёжность – оценка стабильности продукта в различных условиях, выявление и устранение сбоев.
3. Производительность – оценка скорости и эффективности работы продукта, проверка на высокие нагрузки и скорость отклика;
4. Совместимость – проверка работы продукта на разных платформах, браузерах, устройствах. Обеспечение работоспособности в различных окружениях;
5. Безопасность – проверка на уязвимости и угрозы безопасности, защита данных и конфиденциальности;
6. Удобство использования – оценка уровня удовлетворения пользователей, удобство навигации и интерфейса.

Требования к тестированию:

1. Корректность тестирования: каждое требование точно описывает желаемые инструменты и функции.
2. Проверяемость тестирования: требование формулируется так, чтобы существовали способы его однозадачной проверки на факт выполнения.
3. Полнота тестирования: каждое описание содержит информацию, которой достаточно разработчику для грамотной реализации того или иного функционала.
4. Недвусмысленность тестирования: сформулированные описания являются понятными и трактуются только одним способом.
5. Непротиворечивость тестирования: описание не должно содержать внутренних противоречий.
6. Приоритетность тестирования: приоритет требования представлен количественной оценкой степени важности.
7. Атомарность: описание нельзя разделить на более мелкие без потери завершенности.
8. Модифицируемость тестирования: указывает на простоту внесения изменений в отдельные описания или их наборы.
9. **Соблюдение стандартов и регулирований.**
10. **Полнота покрытия**. Тестирование должно охватывать все функциональные и нефункциональные требования к продукту.

Отладка – процесс поиска, локализация и исправления ошибок в программы. Она используется для выявления и устранения несоответствий между ожидаемым и фактическим поведением системы.

1. **Обнаружение ошибки.** Проблема существует и следует понять, какие условия приводят к ее возникновению.
2. **Изучение кода.**
3. **Использование отладочных инструментов.** Использование специальных инструментов и средств отладки, таких как отладчики, профилировщики, журналы событий и т. д., для отслеживания выполнения программы и выявления мест, где возникают ошибки.
4. **Выявление и исправление ошибок**.
5. **Тестирование исправлений.**
6. **Документация и обратная связь:**
7. ***Цели и основные принципы тестирования.***

Цели тестирования:

* обнаружение ошибок и дефектов программного обеспечения.
* проверка соответствия продукта требованиям и спецификациям.
* повышение качества и надежности продукта.

Принципы тестирования:

1. Полное тестирование невозможно – ограничены ресурсы и время. Фокус на наиболее критических аспектах;

*Ограниченные ресурсы и время делают невозможным проведение исчерпывающего тестирования продукта.*

1. Раннее тестирование – тестирование начинается на ранних этапах разработки, что экономит ресурсы и время;
2. Исчерпывающее тестирование – важно покрыть все функциональные и нефункциональные области. Тестовые случаи должны покрывать их все.
3. ***Стадии разработки программного обеспечения: описание.***

* ***Постановка задачи, разработка требований.* О**пределяются цели проекта, анализируются требования пользователей и формулируются функциональные и нефункциональные требования к программному продукту.
* ***Анализ.*** Производится более детальное изучение предметной области, выявление рисков, определение ограничений и составление плана проекта.
* ***Проектирование.*** Происходит разработка архитектуры программного решения. Включает в себя создание детальных планов структуры, интерфейсов, баз данных, алгоритмов и других аспектов программы.
* ***Разработка.*** На этом этапе реализуются проектные решения.
* ***Тестирование и отладка.*** Проводится проверка программного продукта на соответствие требованиям и выявление ошибок
* ***Сопровождение и эксплуатация.*** После завершения разработки и успешного прохождения тестирования начинается этап сопровождения, который включает в себя поддержку программного продукта, внесение изменений, исправление ошибок, обновления.

1. ***Подходы к формированию тестовых наборов: описание, примеры.***

2 подхода к формированию тестовых наборов:

1. *Структурный* – базируется на том, что известны алгоритмы работы программы. В основе структурного тестирования лежит концепция максимально полного тестирования всех маршрутов программы.

(Протестировать цикл с условием: проверить, что цикл корректно выполняется при верном условии и прерывается при ложном)

1. *Функциональный* – основывается на том, что алгоритм работы программного обеспечения не известен.

(Проверить, что форма принимает только корректные адреса электронной почты. Проверить, что пароль должен содержать не менее 8 символов)

1. ***Среда тестирования: понятие, настройка. Документированность процесса тестирования: тестовый план и отчёт.***

Среда тестирования – это настройка программного и аппаратного обеспечения для групп тестирования для выполнения тестовых случаев.

*Тестовая среда настраивается в соответствии с требованиями тестируемого приложения.*

Настройка среды тестирования включает в себя установку необходимых программных и аппаратных компонентов, настройку сетевых параметров, создание тестовых баз данных, установку и настройку средств автоматизации тестирования и других задач.

Цель настройки среды – обеспечить надлежащие условия для проведения тестирования и максимальную эффективность процесса.

Тестовый план – это документ, или набор документов, который содержит тестовые ресурсы, перечень функций и подсистем, подлежащих тестированию, тестовую стратегию, расписание тестовых циклов, фиксацию тестовой конфигурации (состава и конкретных параметров аппаратуры и программного окружения), определение списка тестовых метрик, которые на тестовом цикле необходимо собрать и проанализировать (например, метрик, оценивающих степень покрытия тестами набора требований)

**Тестовый план**: Это документ, который описывает общий подход к тестированию, включая цели, охват, методы, ресурсы и график тестирования. Тестовый план определяет стратегию тестирования и план действий, который должен быть выполнен для достижения поставленных целей.

**Тестовый отчет**: Это документ, который содержит результаты тестирования, включая описание выполненных тестов, найденные ошибки, отчеты об ошибках, рекомендации по улучшению программного продукта и другую информацию, полученную в процессе тестирования.

1. ***Методология тестирования сложных систем. Понятие сложной системы.***

*Сложная система* – система, состоящая их множества взаимодействующих составляющих, вследствие чего сложная система приобретает новые свойства, которые отсутствуют на подсистемном уровне и не могут быть сведены к свойствам подсистемного уровня.

Эффективно начинать тестирование комплексных систем на ранних стадиях разработки ПО.

Тестирование на протяжении процесса разработки сложной структуры из модулей выполняется на нескольких уровнях. Для каждого определяются категория объектов тестирования (ПС, компоненты, отдельные модули) и набор проверяемых тестируемых характеристик. На каждом уровне тестирование повторяется многократно, образуя циклы: тестирование — исправление — повторное тестирование.

Тестирование выполняется всегда для всех объектов системы вне зависимости от её критичности.

Традиционно выделяются три уровня тестирования ПО: автономное или модульное (unit testing), интеграционное (integrating testing) и системное (system testing). В стандарте ISO/IEC 12207 прослеживаются четыре уровня тестирования: ﻿﻿модульное (в процессе «Построение ПО»); ﻿﻿интеграционное (в процессе «Сборка ПО»); ﻿﻿тестирование ПС (как процесс); ﻿﻿системное (в процессе «Испытания ПС»)

1. ***Понятие жизненного цикла программного обеспечения. Стандарты и их классификация.***

*Жизненных цикл ПО* – это период времени, начинающийся с момента принятия решения о необходимости создания ПО и заканчивающийся в момент его полного изъятия из эксплуатации.

Стандарты:

1. ISO/IEC 12207:1995. Информационные технологии – Процесс ЖЦ программы.

*Определяет и регламентирует структуру ЖЦ ПО, состав процессов, действия и задачи.*

*Все процессы разделены на группы: основные, вспомогательные, организационные)*

*Модель ЖЦ – упрощённое и обобщенное представление о том, как развивается продукт.*

*Модели ЖЦ:*

1. *Последовательные (каскадная и v-образная)*
2. *Итерационные (инкрементная и спиральная). Предполагает разбиение проекта на части и прохождение этапов ЖЦ на каждом из них. Каждый этап является законченным сам по себе, а совокупность этапов формирует окончательный результат*

Основные категории стандартов:

* **Международные стандарты:** Это стандарты, принятые международными организациями, такими как ISO (Международная организация по стандартизации) и IEC (Международная электротехническая комиссия). Примеры включают ISO/IEC 12207 (ЖЦ ПО) и ISO/IEC 25000 (Качество ПО).
* **Национальные стандарты:** Это стандарты, принятые в определенных странах или регионах в соответствии с их законодательством и требованиями отрасли. Например, в США это может быть стандарт ANSI (American National Standards Institute).
* **Отраслевые стандарты:** Это стандарты, разработанные и принятые в определенных отраслях. Например, в авиационной отрасли могут применяться стандарты, разработанные Международной ассоциацией воздушного транспорта (IATA).
* **Корпоративные стандарты:** Это стандарты, разработанные и принятые внутри отдельных организаций для регулирования процессов разработки, внедрения и сопровождения программного обеспечения.

1. ***Каскадная модель жизненного цикла. Последовательность этапов. Особенности разработки по каскадной модели.***

*Каскадная модель жизненного цикла* – это классическая линейная модель разработки программного обеспечения. Этапы зависят друг от друга и каждый следующий этап начинается только после завершения предыдущего.

Недостатки: сложность выявления ошибок; трудно вернуться на предыдущий этап; нельзя добавлять какие-то требования.

*Этапы:*

1. **Постановка задачи и определение требований.**
2. **Анализ.**
3. **Проектирование (системной, программной архитектуры, техническое проектирование). С**оздание архитектуры системы, определение структуры и компонентов, разработка алгоритмов.
4. **Разработка**.
5. **Тестирование.** Проверка программного продукта на соответствие требованиям, выявление и исправление ошибок.
6. **Внедрение.** Развертывание программного продукта и его внедрение в рабочую среду.
7. **Сопровождение.** Обеспечение работы программного продукта, внесение изменений и улучшений, исправление ошибок после внедрения.

*Особенности разработки по каскадной модели:*

1. **Последовательность.** Фазы разработки выполняются последовательно, и каждая следующая фаза зависит от успешного завершения предыдущей.
2. **Фиксированные требования.** Требования к программному продукту должны быть четко определены и фиксированы на начальных этапах разработки.
3. **Отсутствие итераций.** В рамках каскадной модели нет возможности для повторного выполнения предыдущих фаз после завершения следующей.
4. **Возможность применения для стабильных проектов.** Каскадная модель хорошо подходит для проектов с четко определенными требованиями и стабильными условиями разработки.
5. **Риск изменений требований.** Изменения требований на поздних этапах разработки могут быть сложными и дорогостоящими.
6. Высокий уровень формализации процессов.
7. ***Спиральная модель жизненного цикла. Особенности и проблемы разработки по спиральной модели.***

Спиральная модель жизненного цикла программного обеспечения представляет собой гибридный подход, который комбинирует элементы последовательного и итеративного развития.

*Особенности:*

1. **Итеративный подход.** Разработка происходит через повторяющиеся циклы, в каждом из которых выполняются этапы определения целей, анализа рисков, разработки, тестирования и оценки.

2. **Управление рисками:** Спиральная модель сосредотачивается на анализе и управлении рисками на протяжении всего жизненного цикла проекта.

3. **Постепенное уточнение требований:** При каждом цикле разработки требования к программному продукту уточняются и дополняются.

4. **Гибкость и адаптивность:** Модель позволяет адаптироваться к изменяющимся требованиям и условиям проекта.

*Проблемы:*

1. **Сложность управления процессом:** Из-за гибкости и итеративности спиральной модели управление процессом разработки может быть сложным.

2. **Трудности оценки времени и затрат:** Итеративный подход затрудняет оценку времени и затрат на проект.

3. **Необходимость экспертизы и опыта:** Для эффективной работы по спиральной модели требуется высокий уровень экспертизы и опыта участников проекта.

4. **Риск недооценки рисков:** Неудачное управление рисками может привести к серьезным проблемам в разработке и внедрении программного продукта.

5. **Сложности в документировании процесса:** Поскольку процесс разработки непрерывно изменяется и уточняется, поддержание документации в актуальном состояние

*Невозможность регламентирования стадий выполнения.*

1. ***Каскадная и спиральная модель: сходства и различия.***

|  |  |
| --- | --- |
| ***сходства*** | ***различия*** |
| 1. **Этапы разработки:** Обе модели включают в себя определенные этапы разработки, такие как анализ требований, проектирование, разработка, тестирование и внедрение.  2. **Цель качества:** Обе модели стремятся к созданию качественного программного продукта, соответствующего требованиям заказчика. | ***Подход к разработке:***  ***Каскадная модель:*** Линейная модель, где каждый этап начинается только после завершения предыдущего и нет возможности возвращения к предыдущим этапам.  ***Спиральная модель:*** Итеративная модель, где разработка происходит через повторяющиеся циклы, в каждом из которых происходят этапы анализа, разработки, тестирования и оценки рисков.  ***Управление рисками:***  ***Каскадная модель:*** Риск управляется в основном на этапах анализа и проектирования, но учет рисков в последующих этапах затруднен.  ***Спиральная модель:*** Управление рисками является одной из ключевых особенностей модели, где каждый цикл включает этап анализа рисков и определения стратегии их управления.  ***Гибкость:***  ***Каскадная модель:*** Менее гибкая, сложно применять для проектов с изменяющимися требованиями.  ***Спиральная модель:*** Большая гибкость и возможность адаптации к изменяющимся требованиям или рискам.  ***Подход к изменениям:***  ***Каскадная модель:*** Затруднено внесение значительных изменений после завершения начальных этапов разработки.  ***Спиральная модель:*** Легче вносить изменения в результаты каждого цикла разработки. |

1. ***Характеристики программного продукта. Этапы решения задач. Понятие алгоритма и его свойства.***

*Алгоритм* – это система точно сформированных правил, определяющая процесс преобразования допустимых исходных данных в желаемый результат за конечное число шагов.

Свойства алгоритма:

1. Дискретность. Разбиение процесса обработки информации на более простые этапы.
2. Определенность. Однозначность выполнения каждого определенного шага преобразования информации.
3. Выполнимость. Конечность действия алгоритма решения задач, позволяющая получить желаемый результат при допустимых исходных данных за конечное число шагов.
4. Массовость. Пригодность алгоритма для решения определенного класса задач.

Программный продукт – комплекс взаимосвязанных программ для решения определенной проблемы массового спроса, подготовленный к реализации как любой вид промышленной продукции.

Характеристики:

* Мобильность. Их независимость от технического комплекса системы обработки данных операций среды, специфики предметной области и т.д.
* Надежность. Определяется бесперебойностью и устойчивостью в работе программ, точностью выполнения предписанных функций, возможностью диагностики возникающих ошибок.
* Эффективность.
* Коммуникативность. Основано на максимально возможной их интеграции с другими программами, обеспечение обмена данных и т.д.
* Модифицируемость. Способность к внесению изменений.
* Удобство использования.

Задачи:

* Технические (ставятся и решаются при организации технологического процесса обработки информации на компьютере)
* Функциональные (требуют решения при реализации функций управления в рамках информационных предметных областей)

Программы:

* Унитарные (предназначены для удовлетворения нужд разработчиков)
* Программные продукты (предназначенные для удовлетворения потребностей пользователей, широкого распространения и продажи)

Этапы:

* Постановка задачи, определение требований.
* Анализ.
* Проектирование
* Разработка.
* Тестирование.
* Эксплуатация и сопровождение.

1. ***Классификация видов тестирования.***

Все виды тестирования программного обеспечения, в зависимости от преследуемых целей, можно условно разделить на следующие группы:

* Функциональные
* Нефункциональные
* Связанные с изменениями

Виды тестирования:

* Функциональное (работоспособность) и нефункциональное (удобство);
* Статическое (документация) и динамическое (практическая проверка).

Другие виды тестирования:

* Нагрузочное;
* Тестирование UX;
* Конфигурационное (совместимость).

Классификация по знанию внутренней системы:

* Чёрный ящик (поведенческий) – тестировщик не знает, как устрое. Идеи идут от предполагаемых паттернов поведения пользователей;
* Белый ящик – тестировщик знает устройство логику программы;
* Серый ящик – смешанное тестирование.

1. ***Функциональное тестирование. Понятие, основные функции, условия ошибки, режим выполнения. Примеры функционального тестирования.***

|  |  |
| --- | --- |
| Функциональное тестирования | Нефункциональное тестирование |
| Выполняется с использованием функциональной спецификации, предоставленной клиентом, и проверяет систему на соответствие функциональным требованиям; | Проверяет производительность, надёжность, масштабируемость и другие нефункциональные аспекты программной системы |
| Выполняется в первую очередь | Проводится после функционального |
| Можно использовать ручное и автоматическое тестирование | Автоматическое тестирование более эффективно |
| Бизнес-требования – исходные данные для функционального тестирования | Параметры производительности (скорость, масштабируемость и т.д.) – входные данные для нефункционального тестирования |
| Описывает, что делает продукт | Описывает, как хорошо работает продукт |
| Легко тестировать вручную | Трудно тестировать вручную |
| Примеры:   * Модульное тестирование; * Дымное тестирование; * Санитарное тестирование; * Интеграционные тестирование; * Белый ящик; * Чёрный ящик; * Приёмочное тестирование пользователей; * Регрессивное тестирование. | Примеры:   * Тестирование производительности; * Нагрузочное тестирование; * Объёмное тестирование; * Стресс тестирование; * Тестирование безопасности; * Тестирование установки; * Проверка на проницаемость; * Тестирование совместимости; * Миграционное тестирование. |

*Функциональное тестирование* - это тип тестирования программного обеспечения, которое проверяет программную систему на соответствие функциональным требованиям/спецификациям. Целью функциональных тестов является тестирование каждой функции программного приложения путем предоставления соответствующих входных данных и проверки выходных данных на соответствие

функциональным требованиям.

Функциональное тестирование в основном включает в себя тестирование черного ящика и не касается исходного кода приложения. Это тестирование проверяет пользовательский интерфейс, API, базу данных, безопасность, связь между клиентом и сервером и другие функции тестируемого приложения. Тестирование может проводиться как вручную, так и с помощью автоматизации.

***Основные функции функционального тестирования:***

1. **Проверка функций продукта. П**роверка того, что каждая функция программы работает корректно и выполняет свою задачу.
2. **Проверка пользовательского интерфейса. П**роверку удобства использования пользовательского интерфейса и соответствия его дизайна требованиям.
3. **Верификация бизнес-логики:** Функциональное тестирование проверяет бизнес-логику программы, убеждаясь в правильности обработки входных данных и корректности выходных результатов.
4. **Тестирование различных сценариев использования.** В рамках функционального тестирования проверяются различные сценарии использования продукта для обеспечения его работоспособности в различных условиях.

**Условия ошибки в функциональном тестировании:**

1. Некорректное поведение функции программы при определенных входных данных.
2. Неправильная обработка исключительных ситуаций.
3. Некорректное отображение или функционирование пользовательского интерфейса.
4. Несоответствие требованиям и спецификации продукта.

*Процесс выполнения функционального тестирования:*

* Понимание функциональных требований
* Идентификация тестовых входных данных или тестовых данных на основе требований
* Вычислите ожидаемые результаты с выбранными тестовыми входными значениями
* Выполнение тестовых случаев
* Сравните фактические и вычисленные ожидаемые результаты

***Примеры функционального тестирования****:* проверка правильности вычислений в финансовом программном обеспечении; проверка функциональности регистрации и входа в систему на веб-сайте.

1. ***Нефункциональное тестирование. Понятие, основные функции, условия ошибки, режим выполнения. Примеры нефункционального тестирования.***

*Нефункциональное тестирование* – это тип тестирования программного обеспечения, который фокусируется на проверке качества продукта без учета его функциональной стороны.

*Основные функции нефункционального тестирования включают:*

* Производительность: Оценка отклика, эффективности и времени отклика системы при различных нагрузках и объемах данных.
* Масштабируемость: проверка способности приложения эффективно работать с большим количеством пользователей или данных.
* Надежность: тестирование стабильности и устойчивости приложения к сбоям и ошибкам.
* Восстанавливаемость: проверка способности приложения быстро восстанавливаться после сбоев и ошибок.
* **Совместимость.** Тестирование совместимости приложения с различными операционными системами, браузерами, устройствами и другими компонентами.
* Тестирование безопасности: поиск и устранение уязвимостей в приложении, которые могут использоваться для взлома или несанкционированного доступа. Совместимость: проверка работоспособности приложения на различных операционных системах, браузерах и устройствах.

*Ошибки в нефункциональном тестировании включают:*

* Низкая производительность или нестабильность работы приложения.
* Проблемы с масштабируемостью и невозможностью работы с большим количеством данных.
* Наличие уязвимостей безопасности и несанкционированный доступ к данным.
* Неработоспособность приложения на различных платформах и устройствах.
* Неправильное использование ресурсов, приводящее к ухудшению эффективности.

1. ***Модульное тестирование. Понятие, основные методы, алгоритм выполнения, преимущества и недостатки.***

*Модульное тестирование* - это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения.

Модульное тестирование *выполняется разработчиками* на этапе разработки (этапа кодирования) приложения. Модульные тесты изолируют участок кода и проверяют его правильность. Единицей может быть отдельная функция, метод, процедура, модуль или объект.

Модульное тестирование – это метод тестирования *WhiteBox*, который обычно выполняется разработчиком. Хотя на практике из-за нехватки времени или нежелания разработчиков проводить тесты инженеры по контролю качества также проводят модульное тестирование.

*Методы модульного тестирования в основном подразделяются на три части:* тестирование черного ящика, которое включает тестирование пользовательского интерфейса вместе с вводом и выводом, тестирование белого ящика, которое включает тестирование функционального поведения программного приложения, и тестирование серого ящика, которое используется для выполнения теста. наборы, методы тестирования, тестовые примеры и выполнение анализа рисков.

*Методы покрытия кода, используемые в модульном тестировании, перечислены ниже:*

1. Покрытие операторов - это метод тестирования белого ящика, при котором все исполняемые операторы в исходном коде выполняются хотя бы один раз.
2. Покрытие решений — это метод тестирования белого ящика, который сообщает об истинных или ложных результатах каждого логического выражения исходного кода.
3. Покрытие ветвей — это метод тестирования белого ящика, в котором проверяется каждый результат модуля кода (оператора или цикла).
4. Покрытие условий или покрытие выражений — это метод тестирования, используемый для проверки и

оценки переменных или подвыражений в условном операторе.

1. Покрытие конечного автомата, безусловно, является наиболее сложным типом метода покрытия кода.

*Преимущества:*

1. Разработчики, желающие узнать, какие функции предоставляет модуль и как его использовать, могут просмотреть модульные тесты, чтобы получить общее представление об API модуля.
2. Модульное тестирование позволяет программисту позже провести рефакторинг кода и убедиться, что модуль по-прежнему работает правильно (например, регрессионное тестирование). Процедура заключается в написании тестовых примеров для всех функций и методов, чтобы всякий раз, когда изменение вызывает ошибку, ее можно было быстро идентифицировать и исправить.
3. Благодаря модульному характеру модульного тестирования мы можем тестировать части проекта, не дожидаясь завершения других.

*Недостатки:*

1. Нельзя ожидать, что модульное тестирование выявит каждую ошибку в программе. Невозможно оценить все пути выполнения даже в самых тривиальных программах.
2. Модульное тестирование по своей природе фокусируется на единице кода. Следовательно, он не может обнаруживать ошибки интеграции или общие ошибки системного уровня.

Модульное тестирование может проводиться как вручную, так и автоматизировано.

Алгоритм проведения при автоматизированном подходе:

1. Разработчик пишет часть кода в приложении только для проверки функции. Позже они закомментируют и, наконец, удалят тестовый код при развертывании приложения.
2. Разработчик также может изолировать функцию, чтобы более тщательно протестировать ее. Это более тщательная практика модульного тестирования, которая включает копирование и вставку кода в собственную среду тестирования, а не в естественную среду. Изоляция кода помогает выявить ненужные зависимости между тестируемым кодом и другими единицами или пространствами данных в продукте.Затем эти зависимости могут быть устранены.
3. Кодировщик обычно использует UnitTest Framework для разработки автоматизированных тестовых случаев. Используя структуру автоматизации, разработчик вводит критерии в тест, чтобы проверить правильность кода. Во время выполнения тестовых случаев платформа регистрирует неудачные тестовые случаи. Многие фреймворки также автоматически помечают и сообщают об этих неудачных тестах. В зависимости от серьезности сбоя платформа может остановить последующее тестирование.
4. Рабочий процесс модульного тестирования: 1) Создание тестовых случаев 2) Обзор/переработка 3)Базовый план 4) Выполнение тестовых случаев.

*Или:*

1. Идентификация модулей тестирования.
2. Разработка тестовых случаев.
3. Подготовка тестовых данных.
4. Выполнение тестов.
5. Анализ результатов.
6. Исправление ошибок и т.д.

При автоматизированном подходе:

* Выбор инструментов для автоматизированного тестирования.
* Разработка текстовых сценариев.
* Написание автоматизированных тестов.
* Запуск тестов.

1. ***Дымовое тестирование. Понятие, основные методы, алгоритм выполнения, преимущества, примеры дымового тестирования, отличия от санитарного тестирования.***

*Дымовое тестирование* – это процесс тестирования программного обеспечения, который определяет, является ли развернутая сборка программного обеспечения стабильной или нет.

***Основные методы дымового тестирования:***

* **Проверка основных функций:** Тестируются основные функции приложения, чтобы убедиться, что они работают корректно.
* **Проверка стабильности:** Проверяется стабильность приложения при базовых нагрузках и входных данных.
* **Проверка взаимодействия с основными компонентами:** Убеждение в том, что приложение может взаимодействовать с внешними и внутренними компонентами, такими как базы данных, API и т. д.

*Алгоритм:*

* **Определение основных функций приложения:** Идентификация ключевых функций, которые необходимо протестировать.
* **Создание набора тестов:** Разработка небольшого набора тестов, которые проверяют основные функции приложения.
* **Запуск тестов:** Запуск набора тестов после каждого изменения в приложении или новой сборки.
* **Анализ результатов:** Оценка результатов тестирования для определения того, работает ли приложение в основном.
* **Принятие решения:** Принятие решения о том, следует ли приступать к более подробному тестированию или есть необходимость в исправлении выявленных проблем.

Он состоит из минимального набора тестов, запускаемых в каждой сборке для проверки функциональных возможностей программного обеспечения.

Проводят QA инженеры по обеспечению качества/руководители отдела контроля качества проводят тестирование с дымом.

Дымовое тестирование выполняется всякий раз, когда разрабатываются новые функции программного обеспечения и интегрируются с существующей сборкой, которая развертывается в среде обеспечения качества/тестирования. Это гарантирует, что все критические функции работают правильно или нет.

Берется подмножество тестовых случаев, а затем тестировщики запускают тестовые случаи в сборке. Команда контроля качества тестирует приложение на соответствие критическим функциям. Эти серии тестовых случаев предназначены для выявления ошибок в build. Если эти тесты пройдены, группа контроля качества продолжает функциональное тестирование.

Любой сбой указывает на необходимость вернуть систему команде разработчиков. Всякий раз, когда в сборку вносятся изменения, мы проводим тестирование с дымом, чтобы обеспечить стабильность.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *сценарии испытаний* | *описание* | *этап теста* | *ожидаемый результат* | *фактический результат* | *статус* |
| Действительные учетные данные для входа | Проверьте функцию входа в веб- приложение, чтобы убедиться, что зарегистрированному пользователю разрешен вход с именем пользователя и паролем. | 1.Запустите приложение  2.Перейдите на страницу входа  3.Введите правильное имя  пользователя  4.Введите правильный пароль  5.Нажмите кнопку входа | Логин должен быть успешным | как и ожидалось | PASS |
| Добавление функциональности элемента | Возможность добавить товар в корзину | 1.Выберите список категорий  2.Добавьте товар в корзину | Товар должен быть добавлен в корзину | Товар не добавляется в корзину | FAIL |
| Функциональность выхода | Проверьте функцию выхода | 1. выберите кнопку выхода | Пользователь должен иметь возможность выйти. | Пользователь не может выйти | FAIL |

*Преимущества:*

* Простота проведения тестирования
* Дефекты выявляются на ранних стадиях.
* Улучшает качество системы
* Снижает риск
* Доступ к прогрессу проще.
* Экономит усилия и время тестирования
* Легкое обнаружение критических ошибок и исправление ошибок.
* Он работает быстро
* Сводит к минимуму интеграционные риски

|  |  |
| --- | --- |
| *Дымное тестирование* | *Санитарное тестирование* |
| Дымовое тестирование выполняется, чтобы убедиться, что  критически важные функции программы работают  нормально. | Санитарное тестирование проводится для проверки того,  что новые функции/ошибки были исправлены. |
| Целью этого тестирования является проверка  «стабильности» системы, чтобы приступить к более  тщательному тестированию. | Цель тестирования — проверить «рациональность»  системы, чтобы приступить к более строгому тестированию. |
| Это тестирование выполняется разработчиками или  тестировщиками. | Проверка работоспособности при тестировании  программного обеспечения обычно выполняется  тестировщиками. |
| Тестирование дыма обычно документируется или  записывается в сценарии. | Проверка работоспособности обычно не документируется и  не записывается по сценарию. |
| Проверка работоспособности обычно не документируется и  не записывается по сценарию. | Санитарное тестирование является  подмножеством регрессионного тестирования . |
| Дымовое тестирование проверяет всю систему от начала до  конца | Санитарное тестирование проверяет только определенный  компонент всей системы. |
| Проверка дыма похожа на общую проверку здоровья | Санитарное тестирование похоже на специализированную  проверку здоровья |

1. ***Санитарное тестирование. Понятие, основные методы, алгоритм выполнения, преимущества, примеры санитарного тестирования, отличия от дымового тестирования.***

Санитарное тестирование – это вид тестирования программного обеспечения, выполняемого после получения сборки программного обеспечения с небольшими изменениями в коде или функциональности, чтобы убедиться, что ошибки были исправлены и что из-за этих изменений не возникло дополнительных проблем. Цель состоит в том, чтобы определить, работает ли предлагаемая функциональность примерно так, как ожидалось. Если тест работоспособности не проходит, сборка отклоняется, чтобы сэкономить время и деньги, связанные с более тщательным тестированием.

Цель состоит не в том, чтобы тщательно проверить новую функциональность, а в том, чтобы определить, что разработчик применил некоторую рациональность (здравомыслие) при создании программного обеспечения.

***Основные методы:***

1. **Проверка основных функций:** Проверяются основные функции приложения для подтверждения их работоспособности.
2. **Проверка критических сценариев использования:** Проверяются ключевые сценарии использования, которые влияют на общую функциональность приложения.
3. **Проверка целостности системы:** Убеждение в том, что изменения не нарушили целостность системы и не вызвали серьезных проблем.

***Алгоритм выполнения:***

1. **Идентификация основных функциональных элементов:** Определение ключевых функций и сценариев использования, которые необходимо протестировать.
2. **Разработка набора тестов:** Создание небольшого набора тестов, который охватывает основные функциональные элементы приложения.
3. **Запуск тестов:** Запуск набора тестов после внесения изменений в приложение или после новой сборки.
4. **Анализ результатов:** Оценка результатов тестирования для выявления любых критических проблем или несоответствий ожидаемому поведению.
5. **Принятие решения:** Принятие решения о том, следует ли приступать к более подробному тестированию или исправлению найденных проблем.

***Преимущества санитарного тестирования:***

1. **Эффективное использование времени:** Позволяет быстро проверить основные функциональные аспекты приложения.
2. **Раннее обнаружение проблем:** Помогает выявить критические проблемы на ранних этапах разработки.
3. **Улучшение качества продукта:** Позволяет быстро исправить выявленные проблемы и обеспечить качество продукта.

***Примеры санитарного тестирования:***

1. Проверка основных функций входа и навигации в веб-приложении.
2. Проверка основных функций добавления, удаления и обновления данных в приложении.
3. Проверка основных сценариев использования мобильного приложения.
4. ***Интеграционное тестирование. Понятие, необходимость интеграционного тестирования, основные методы, критерии входа и выхода для интеграционного тестирования.***

*Системное интеграционное тестирование* определяется как тип тестирования программного обеспечения, проводимого в интегрированной аппаратно-программной среде для проверки поведения всей системы. Это тестирование, проводимое на полной интегрированной системе для оценки соответствия системы заданным требованиям.

Системное интеграционное тестирование (SIT) выполняется для проверки взаимодействия между модулями программной системы. Он касается проверки требований к программному обеспечению высокого и низкого уровня, указанных в спецификации/данных требований к программному обеспечению и документе по проектированию программного обеспечения.

Он также проверяет сосуществование программной системы с другими и тестирует интерфейс между модулями программного приложения. В этом типе тестирования модули сначала тестируются по отдельности, а затем объединяются в систему.

*Необходимость:*

* Это помогает обнаружить дефект на ранней стадии
* Более ранняя обратная связь о приемлемости отдельного модуля будет доступна
* Планирование исправления дефектов является гибким, и его можно совмещать с разработкой.
* Правильный поток данных
* Правильный поток управления
* Правильное время
* Правильное использование памяти
* Исправьте требования к программному обеспечению

*Критерии входа и выхода:* Обычно при выполнении интеграционного тестирования используется стратегия ETVX (критерии входа, задачи, проверки и критерии выхода).

*Критерии входа:*

* Завершение модульного тестирования (Юнит-тесты)

*Входы:*

* Данные о требованиях к программному обеспечению
* Документ по дизайну программного обеспечения
* План проверки программного обеспечения
* Документы по интеграции программного обеспечения
* Критерии входа и выхода для интеграционного тестирования

*Мероприятия:*

1. На основе требований высокого и низкого уровня создайте тестовые примеры и процедуры.
2. Объединяйте низкоуровневые сборки модулей, реализующие общий функционал
3. Разработайте тестовую обвязку
4. Протестируйте сборку
5. После прохождения теста сборка объединяется с другими сборками и тестируется до тех пор, пока
6. система не будет интегрирована в целом.
7. Повторно выполнить все тесты на платформе с целевым процессором и получить результаты.

*Критерии выхода:*

Успешное завершение интеграции Программного модуля на целевом Аппаратном обеспечении

Корректная работа программного обеспечения в соответствии с указанными требованиями

*Выходы*

1. Отчеты об интеграционных испытаниях
2. Примеры и процедуры тестирования программного обеспечения [SVCP].

*Подходы:*

* *Подход большого взрыва.* все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование.
* *Инкрементный подход.* тестирование выполняется путем объединения двух или более логически связанных модулей. Затем другие связанные модули поэтапно добавляются и тестируются для правильного функционирования. Процесс продолжается до тех пор, пока все модули не будут соединены и успешно протестированы.

1. *Нисходящий.* Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые.
2. *Восходящий.* В восходящей стратегии каждый модуль на более низких уровнях последовательно тестируется с более высокоуровневыми модулями, пока не будут протестированы все модули.
3. *Гибридный / сэндвич-подход.* Представляет собой комбинацию восходящего и нисходящего подходов. Здесь целью является средний слой

Это систематический метод построения структуры программы при проведении тестов для выявления ошибок, связанных с интерфейсом.

Все модули интегрируются заранее, и вся программа тестируется как единое целое. Но во время этого процесса, скорее всего, возникнет множество ошибок.

Исправление таких ошибок затруднено, поскольку изоляция причин усложняется огромным расширением всей программы. Как только эти ошибки будут исправлены и исправлены, появится новая, и процесс продолжится в бесконечном цикле. Чтобы избежать этой ситуации, используется другой подход – добавочная интеграция.

Есть несколько дополнительных методов, таких как интеграционные тесты, которые проводятся в системе на основе целевого процессора. Используемая методология – тестирование черного ящика. Можно использовать как восходящую, так и нисходящую интеграцию.

Тестовые случаи определяются с использованием только высокоуровневых требований к программному обеспечению.

1. ***Тестирование методом «черного ящика». Типы и область тестирования, примеры.***

*Тестирование «черного ящика»* – это метод тестирования программного обеспечения, при котором функциональные возможности программных приложений проверяются без знания внутренней структуры кода, деталей реализации и внутренних путей.

Тестирование черного ящика в основном фокусируется на вводе и выводе программных приложений и полностью основано на требованиях и спецификациях программного обеспечения. Он также известен как поведенческое тестирование.

*Типы тестирования:*

1. Функциональное тестирование. Этот тип тестирования черного ящика связан с функциональными требованиями системы; это делают тестировщики программного обеспечения.
2. Нефункциональное тестирование. Этот тип тестирования «черного ящика» связан не с тестированием конкретной функциональности, а с нефункциональными требованиями, такими как производительность, масштабируемость, удобство использования.
3. Регрессионное тестирование. Регрессионное тестирование проводится после исправления кода, обновлений или любого другого обслуживания системы, чтобы убедиться, что новый код не повлиял на существующий код.

*Методы:*

1. Тестирование класса эквивалентности: оно используется для минимизации количества возможных тестовых случаев до оптимального уровня при сохранении разумного тестового покрытия.
2. Тестирование граничных значений: тестирование граничных значений сосредоточено на значениях на границах. Этот метод определяет, является ли определенный диапазон значений приемлемым для системы или нет. Это очень полезно для сокращения количества тестовых случаев. Это наиболее подходит для систем, где вход находится в определенных диапазонах.
3. Тестирование таблицы решений: таблица решений помещает причины и их последствия в матрицу. В каждом столбце есть уникальная комбинация.
4. ***Тестирование методом «белого ящика». Типы и область тестирования, примеры. Преимущества и недостатки метода.***

*Типы тестирования:*

1. Модульное тестирование: часто это первый тип тестирования приложения. Модульное тестирование выполняется для каждой единицы или блока кода по мере его разработки. Модульное тестирование в основном выполняется программистом. Как разработчик программного обеспечения, вы разрабатываете несколько строк кода, одну функцию или объект и тестируете их, чтобы убедиться, что они работают, прежде чем продолжить. Модульное тестирование помогает выявить большинство ошибок на ранних этапах жизненного цикла разработки программного обеспечения. Ошибки, обнаруженные на этом этапе, дешевле и их легко исправить.
2. Тестирование на наличие утечек памяти: Утечки памяти являются основной причиной замедления работы приложений. Специалист по обеспечению качества, имеющий опыт обнаружения утечек памяти, необходим в тех случаях, когда у вас медленно работает программное приложение.

*Преимущества:*

* Оптимизация кода путем поиска скрытых ошибок.
* Случаи тестирования белого ящика можно легко автоматизировать.
* Тестирование более тщательное, так как обычно охватываются все пути кода.
* Тестирование можно начать раньше в SDLC, даже если графический интерфейс недоступен.

*Недостатки:*

* Тестирование белого ящика может быть довольно сложным и дорогим.
* Разработчики, которые обычно выполняют тестовые случаи белого ящика, ненавидят его. Тестирование белого ящика разработчиками не является подробным, что может привести к производственным ошибкам.
* Для тестирования методом «белого ящика» требуются профессиональные ресурсы с подробным пониманием программирования и реализации.
* Тестирование методом «белого ящика» занимает много времени, для полного тестирования больших программных приложений требуется время.

Тестирование «белого ящика» включает в себя тестирование программного кода на предмет следующего:

* Дыры внутренней безопасности
* Сломанные или плохо структурированные пути в процессах кодирования
* Поток определенных входных данных через код
* Ожидаемый результат
* Функциональность условных циклов
* Тестирование каждого оператора, объекта и функции на индивидуальной основе

Основным методом тестирования «белого ящика» является анализ покрытия кода. Анализ покрытия кода устраняет пробелы в наборе тестовых примеров. Он определяет области программы, которые не проверяются набором тестовых примеров. После выявления пробелов вы создаете тест-кейсы для проверки непроверенных частей кода, тем самым повышая качество программного продукта.

1. ***Тестирование методом «серого ящика». Стратегия, преимущества и проблемы тестирования. Область тестирования. Этапы выполнения тестирования.***

*Тестирование серого ящика* – это метод тестирования программного обеспечения для тестирования программного продукта или приложения с частичным знанием внутренней структуры приложения. Целью тестирования серого ящика является поиск и выявление дефектов, связанных с неправильной структурой кода или неправильным использованием приложений.

В этом процессе обычно выявляются контекстно-зависимые ошибки, связанные с веб-системами. Это увеличивает охват тестированием, концентрируясь на всех уровнях любой сложной системы.

Тестирование серого ящика – это метод тестирования программного обеспечения, который представляет собой комбинацию методов тестирования белого ящика и метода тестирования черного ящика.

*Стратегии:*

Для выполнения тестирования серого ящика не обязательно, чтобы тестер имел доступ к исходному коду. Тест разработан на основе знания алгоритма, архитектуры, внутренних состояний или других высокоуровневых описаний поведения программы.

Чтобы выполнить тестирование серого ящика:

* Он применяет простую технику тестирования черного ящика.
* Он основан на генерации тестовых случаев требований, поэтому он предварительно устанавливает все условия перед тем, как программа будет протестирована методом утверждений.

*Преимущества:*

* **Объективность:** Тестировщики могут выявлять ошибки, не имея предвзятого представления о внутренней структуре программы.
* **Приоритизация функций:** Позволяет сконцентрироваться на тестировании ключевых функций программы.
* **Эффективность:** Может использоваться для обнаружения различных типов ошибок без необходимости изучения всей внутренней логики программы.

*Проблемы:*

* Когда тестируемый компонент сталкивается с каким-либо сбоем, это может привести к прерыванию текущей операции.
* Когда тест выполняется полностью, но содержание результата неверно.

*Шаги выполнения:*

* Определите входные данные
* Определите результаты
* Определите основные пути
* Определите подфункции
* Разработка входных данных для подфункций
* Разработка результатов для подфункций
* Выполните тестовый пример для подфункций
* Проверьте правильный результат для подфункций.
* Повторите шаги 4 и 8 для других подфункций.
* Повторите шаги 7 и 8 для других подфункций.

*Область тестирования* включает в себя функциональные и нефункциональные аспекты программы, такие как ввод данных, обработка, вывод, производительность, надежность и безопасность.

*Этапы:*

1. **Планирование:** Определение целей тестирования, выбор методов и инструментов, составление плана тестирования.

2.Создание тестовых случаев и сценариев на основе требований и спецификаций.

3. **Исполнение тестов:** Запуск тестовых случаев и регистрация результатов.

4. **Анализ результатов:** Оценка результатов тестирования, выявление ошибок, их документирование и отслеживание.

5. **Отчетность:** Подготовка отчета о выполненном тестировании с обзором найденных ошибок и рекомендациями.

1. ***Регрессивное тестирование. Понятие, необходимость, методы проведения регрессивного тестирования. Выбор регрессивного теста и тестовых случаев. Различия между повторным тестированием и регрессивным. Проблемы регрессивного тестирования.***

Регрессионное тестирование – тип тестирования программного обеспечения для подтверждения того, что недавнее изменение программы или кода не повлияло отрицательно на существующие функции. Регрессионное тестирование – это не что иное, как полный или частичный выбор уже выполненных тестовых случаев, которые выполняются повторно, чтобы убедиться, что существующие функции работают нормально.

Это тестирование проводится, чтобы убедиться, что новые изменения кода не будут иметь побочных эффектов для существующих функций. Это гарантирует, что старый код по-прежнему будет работать после внесения последних изменений в код.

*Необходимость:*

Необходимость регрессионного тестирования в основном возникает всякий раз, когда требуется изменить код, и нам нужно проверить, влияет ли измененный код на другую часть программного приложения или нет. Более того, регрессионное тестирование необходимо при добавлении новой функции в программное приложение и для устранения дефектов, а также устранения проблем с производительностью.

*Проведение регрессионного тестирования:*

* Сначала отладить код, чтобы выявить ошибки.
* После выявления ошибок вносятся необходимые изменения для их исправления.
* Затем выполняется регрессионное тестирование путем выбора соответствующих тестовых случаев из набора тестов, который охватывает как измененные, так и затронутые части кода.

Обслуживание программного обеспечения – это деятельность, которая включает улучшения, исправления ошибок, оптимизацию и удаление существующих функций. Эти модификации могут привести к неправильной работе системы. Поэтому регрессионное тестирование становится необходимым.

*Выбор регрессионного теста:*

Выбор регрессионного теста – это метод, при котором выполняются некоторые выбранные тестовые примеры из набора тестов, чтобы проверить, влияет ли измененный код на программное приложение или нет.

Тестовые наборы делятся на две части: повторно используемые тестовые наборы, которые можно использовать в дальнейших циклах регрессии, и устаревшие тестовые наборы, которые нельзя использовать в последующих циклах.

*Приоритизация тестовых случаев*

Расставьте приоритеты тестовых случаев в зависимости от влияния на бизнес, критических и часто используемых функций. Выбор тестовых случаев на основе приоритета значительно сократит набор регрессионных тестов.

*Выбор тестовых случаев:*

Эффективные регрессионные тесты можно выполнить, выбрав следующие тестовые примеры:

* Тест-кейсы с частыми дефектами
* Функциональность, которая более заметна для пользователей
* Тестовые случаи, которые проверяют основные функции продукта
* Тестовые случаи функциональных возможностей, которые претерпели больше и недавние изменения
* Все интеграционные тестовые случаи
* Все сложные тестовые случаи
* Тестовые примеры граничных значений
* Образец успешных тестовых случаев
* Пример тестовых случаев отказа

*Разница между регрессивным тестированием и повторным тестированием:*

Повторное тестирование означает повторное тестирование функциональности или ошибки, чтобы убедиться, что код исправлен. Если это не исправлено, необходимо повторно открыть дефект. Если исправлено, дефект закрывается.

Регрессионное тестирование означает тестирование вашего программного приложения, когда в нем происходит изменение кода, чтобы убедиться, что новый код не повлиял на другие части программного обеспечения.

*Проблемы регрессионного тестирования:*

* При последовательных запусках регрессии наборы тестов становятся довольно большими. Из-за ограничений по времени и бюджету весь набор регрессионных тестов не может быть выполнен.
* Сведение к минимуму набора тестов при достижении максимального покрытия тестами остается сложной задачей.
* Определение частоты регрессионных тестов, т. е. после каждой модификации или каждого обновления сборки или после множества исправлений ошибок, является сложной задачей.

*Методы:*

* Полное регрессионное тестирование. Весь набор тестовых случаев запускается после каждого изменения в программном коде.
* Выборочное. Тестировщики выбирают набор ключевых тестовых случаев, которые вероятнее всего будут затронуты изменениями.
* Автоматизированное. Использование инструментов автоматизации тестирования для запуска набора тестовых случаев после каждого изменения.
* Обратное. Изменения в коде не повлияли на уже протестированный функционал.

1. ***Определение и цель ручного тестирования. Преимущества и недостатки ручного тестирования. Типы тестирования.***

*Ручное тестирование* – это процесс тестирования программного обеспечения, выполняемый вручную без использования автоматизированных инструментов или скриптов.

***Цель ручного тестирования*** заключается в проверке функциональности, интерфейса, производительности и других аспектов программного продукта с точки зрения конечного пользователя.

***Преимущества:***

1. **Гибкость и адаптивность:** Возможность быстро адаптироваться к изменениям в требованиях или функциональности продукта.
2. **Интуиция и опыт:** Ручные тестировщики могут применять свой опыт и интуицию для выявления потенциальных проблем или ошибок.
3. **Тестирование пользовательского опыта:** Ручное тестирование позволяет оценить пользовательский опыт и взаимодействие с продуктом с точки зрения реального пользователя.
4. **Недорогостоящая альтернатива:** В начальных стадиях разработки или для небольших проектов ручное тестирование может быть более экономически выгодным вариантом.

***Недостатки ручного тестирования:***

1. **Времязатратность:** Ручное тестирование требует значительного времени и усилий, особенно для повторяющихся тестовых задач.

2. **Человеческий фактор:** Возможность допущения ошибок или упущений из-за человеческого фактора, особенно при монотонных задачах.

3. **Ограниченная масштабируемость:** Ручное тестирование может быть неэффективным для крупных проектов или продуктов с большим объемом функциональности.

4. **Невозможность полной автоматизации:** Некоторые аспекты тестирования, особенно связанные с большим объемом данных или сложной логикой, могут быть сложно или невозможно автоматизировать.

***Типы тестирования:***

1. **Функциональное тестирование:** Проверка функциональности продукта в соответствии с требованиями.

2. **Нагрузочное тестирование:** Оценка производительности и стабильности продукта при высоких нагрузках.

3. **Интеграционное тестирование:** Тестирование взаимодействия между различными компонентами системы.

4. **Приемочное тестирование:** Проверка соответствия продукта требованиям заказчика.

5. **Usability (юзабилити) тестирование:** Оценка удобства использования и пользовательского опыта.

6. **Безопасность (Security) тестирование:** Проверка на уязвимости и защищенность системы от атак.

7. **Регрессионное тестирование:** Проверка сохранения работоспособности всех функций после внесения изменений в код или интерфейс продукта.

1. ***Интеграционное тестирование. Понятие, цель и объект тестирования. Методы и инструменты тестирования.***

Интеграционное тестирование – тип тестирование, при котором модули объединяются логически и тестируются как группа.

Цель – выявление дефектов, появляющийся при взаимодействии между программными модулями.

Объект тестирования – потоки данных между модулями.

Тип тестирования – тестирование черного ящика.

Подходы интеграционного тестирования:

* Подход большой взрыва. Все модули собираются в единую систему и тестируются.
* Икрементальные подходы. Тестирование не зависит от готовности всех блоков. Дефекты локализуются в модулях.
* Нисходящий
* Восходящий
* Сэндвич – комбинация.

1. ***Юзабилити тестирование: понятие, основные подходы планирования и разработки программных продуктов.***

Юзабилити – степень, с которой продукт может быть использован определёнными пользователями при определённом контексте для достижения определённых целей с должной эффективностью, результативностью и удовлетворённостью. Юзабилити отражает степень удобства использования программного продукта конечными пользователями. Так как взаимодействие пользователя и программного обеспечения осуществляется посредством пользовательских интерфейсов, то понятие юзабилити прежде всего относится к процессу разработки пользовательских интерфейсов.

*Основные подходы планирования и разработки программных продуктов:*

* Проверка соответствия принципам обеспечения удобства пользования и корректного визуального представления в контексте функциональных требований посредством экспертной оценки (экспертный подход).
* Изучение опыта взаимодействия пользователя с приложением через имитацию поведения пользователей (пользовательский подход)

1. ***Экспертный подход к юзабилити-тестированию: критерии. Преимущества и недостатки подхода.***

*Преимущества:*

* быстрый в применении;
* эксперты гарантировано понимают общие задачи программного продукта.

*Недостатки:* субъективизм (эксперты не являются реальными пользователями)

При экспертном подходе в качестве пользователей выступают два и более экспертов (оптимальное количество для больших проектов 5-6 человек).

*Эксперты проходят основные сценарии поведения пользователей и анализируют их с точки зрения:*

* стандартов юзабилити для конкретного типа программного продукта (например, Android Material Design для мобильных приложений на платформе Android);
* общих принципов юзабилити (эвристики Якоба Нильсена);
* здравого смысла и опыта.

*Основные принципы, сформулированные Якобом Нильсеном:*

* Информированность о состоянии системы. Пользователь всегда должен ориентироваться и четко понимать, что происходит в системе. Реализация обратной связи, сообщений.
* Схожесть системы с реальным миром. Система должна общаться с пользователем на понятном ему языке. Использование инфографики, слов, фраз и понятий.
* Свобода действий. Необходимо предоставить пользователям возможность отмены действий, а также возврата к ранее осуществленным действиям.
* Единообразие и стандарты. Не следует вводить в заблуждения пользователя, описывая одни и те же вещи разными словами и терминами.
* Предотвращение ошибок. Важно свести к минимуму количество условий, в которых могут быть допущены ошибки. Подсказки, поясняющие, какую информацию надо вводить в текстовые поля.
* На виду, а не в памяти. Вся необходимая информация должна быть размещена в пределах доступности для пользователя.
* Гибкость и эффективность. Не следует нагружать пользователей лишней информацией – предоставьте им возможность совершать часто повторяющиеся действия как можно быстрее и проще.
* Эстетичный и минималистичный дизайн.
* Понимание проблем и их решение.
* Справочные материалы и документация. Даже если система может использоваться без документации, в процессе работы с ней все же может потребоваться справочная информация.

1. ***Пользовательский подход к юзабилити-тестированию: критерии, методы измерения. Преимущества и недостатки подхода.***

*Преимущества пользовательского подхода:*

* объективные результаты (участвуют реальные пользователи);
* процесс легко измерим.

*Недостатки пользовательского подхода:*

* длительный по времени;
* дорогой (если пользователей привлекают на платной основе);
* большое внимание следует уделить подбору пользователей.

Пользователям (3-5 человек из каждого сегмента целевой аудитории), согласившимся участвовать в тестировании, предлагают пройти наиболее распространенные и наиболее проблемные сценарии.

Эксперт протоколирует действия пользователя, фиксирует все в видео-формате, чтобы отследить реакцию (эмоции) пользователя, но никак не влияет на действия пользователя.

Возможные измерения при юзабилити-тестировании: успешность время выполнения выполнения задачи; задачи; эффект первого впечатления (например, сколько раз улыбнулся).

*Критерии:* простота использования, эффективность, удовлетворение, доступность.

*Методы измерения:*

* Тестирование сценариев использования. Пользователи выполняют типичные задачи, а исследователи наблюдают за их действиями и оценивают эффективность использования.
* Анкетирование и опросы.
* Интервью с пользователями.

1. ***Автоматизация тестирования: понятие, типы, принципы. Требования к тестировщику. Преимущества и недостатки автоматизации тестирования.***

Автоматизация тестирования – это процесс использования инструментов и программного обеспечения для выполнения тестовых сценариев, проверки функциональности, производительности, безопасности и других аспектов программного продукта без прямого участия человека.

Тестировщик, занимающийся автоматизацией, должен обладать навыками программирования, пониманием тестовых методик и процессов, уметь анализировать результаты тестирования и вносить коррективы в автоматизированные тесты при необходимости.

*Преимущества автоматизации тестирования:*

1. Экономия времени.
2. Исключение человеческого фактора.
3. Возможность эмулировать многопользовательскую работу – если рабочей нормой считается одновременное обращение к приложению нескольких тысяч пользователей, то средства автоматизации являются единственным способом решить проблему нагрузочного тестирования.
4. Отсутствие необходимости в графическом пользовательском интерфейсе – на ранних этапах развития программного продукта интерфейс, как правило, еще не согласован; это существенно также при тестировании обмена данными по протоколам
5. Наличие инструментария фиксации ошибок и результатов – это позволяет моделировать различные ошибочные ситуации, строить любые отчеты и диаграммы
6. *Недостатки:*
7. Необходимость программистских навыков у тестировщика – настоящая профессиональная автоматизация тестирования невозможна без работы непосредственно с кодом тестового скрипта
8. Чувствительность к среде, программному и аппаратному окружению тестируемого приложения – один и тот же тест одной и той же версии повторно может проходить совершенно иначе, чем в первый раз.

*Типы тестирования:*

1. функциональное (в том числе модульное, или unit-тестирование),
2. регрессионное (проверка работоспособности старого функционала и отсутствия ранее исправленных дефектов в новых версиях)
3. нагрузочное (поведение приложения под рабочей и стрессовой нагрузкой, влияние работающего приложения на системное окружение).

*Принципы:*

1. не следует пытаться автоматизировать все тесты, т.к. наиболее простые из них вполне могут быть выполнены в «ручном» режиме.

2. средства автоматизации – это всего лишь инструмент, поэтому особое внимание необходимо уделять качеству тест-плана.

3. аккуратное и адекватное планирование – залог успеха автоматизации.

1. ***Понятие утверждений и их параметры. Модели и основные виды утверждений. Понятие и категории директив.***

Утверждения представляют собой гипотезы, высказываемые тестировщиком относительно результатов выполнения того или иного теста. Если гипотеза подтвердилась, то начинает выполняться следующий тест (либо тестирование завершается), иначе возникает ошибка.

Однако для каждого метода существуют перегружаемые варианты, которые содержат дополнительные параметры, позволяющие сформировать строку сообщения.

*Параметры утверждений:*

Дополнительный параметр может быть обычной строкой, либо строкой со списком параметров, добавляемых в сообщение о результатах выполнения теста:

1. Assert.AreEqual( int expected, int actual, string message );
2. Assert.AreEqual( int expected, int actual, string message, params object[] parms );

*Параметры утверждений:*

* *Ожидаемое значение.*
* *Фактическое значение.*
* *Точность. Степень соответствия фактического значения ожидаемому.*
* *Метод сравнения. Алгоритм, по которому сравнивается ожидаемое и фактическое значения.*

Две модели для утверждений:

1. В nUnit поддерживаются две модели для утверждений – классическая и закрытая.

Классическая модель предполагает непосредственное обращение к методам класса Assert так, как это было сделано в вышеприведенных примерах.

В закрытой модели (constraint-based model) используется единственный метод класса Assert – метод That. Этот метод возвращает объект, в котором реализована вся логика, необходимая для проверки утверждения Assert.That( myString, Is.EqualTo("Hello") );

Основные виды утверждений. Все утверждения nUnit можно разделить на несколько групп:

1. утверждения равенства (Equality Asserts)
2. утверждения сравнения (Comparison Asserts)
3. утверждения о типах (Type Asserts)
4. утверждения о строках (StringAssert)

Директивы или атрибуты – это специальные предложения, используемые для структурирования тестовых заданий и описания дополнительных спецификаций теста. Все директивы содержатся в пространстве имен nunit.framework, которое должно быть включено в любой файл, содержащий тесты.

Категории директив:

1. Идентифицирующие
2. Селектирующие
3. Модифицирующие
4. Подготовки и очистки
5. Параметризующие
6. ***Определение, цель и этапы тестирования производительности. Критерии для измеряемых метрик.***

*Тестирование производительности* – тестирование, целью которого служит определение скорости работы вычислительной системы или ее частей под определёнными нагрузками.

*Этапы:*

* Анализ системы и рассмотрение требований.
* Подготовка стратегии.
* Настройка генератора нагрузки.
* Мониторинг серверов и генератора нагрузки.
* Подготовка тестовых данных.
* Разработка нагрузочных скриптов.
* Предварительные запуски тестов.
* Тестирование.
* Анализ результата и подготовка отчета
* полностью законченное приложение или система и правильно настроенная конфигурация ПО. Анализ системы включает в себя изучение свойств системы, её особенностей и режима работы. Требование – это критерий, которому система должна соответствовать с технической точки зрения.
* Стратегия – разрабатывается на основе детального анализа ПО и описывает подходы к тестированию производительности.
* Для качественного проведения тестирования необходимо установить инструмент тестирования на генератор нагрузки – виртуальную или физическую машину, расположенную максимально близко к серверу приложения.
* Выделенные на первом этапе метрики используются для оценки параметров производительности.
* 6 этап: Сценарии тестирования производительности создаются с использованием выбранного инструмента. Процесс разработки скриптов состоит из трех шагов: 1. Изучение сценарием – анализ подготовленных сценариев, для определения действия каждого. 2. Создание тест-кейсов – каждый сценарий записывается в шаблон без параметризации. 3. Отладка сценариев – запуск каждого сценария

*Критерии для измеряемых метрик:*

1. Количество пользователей – программа, которая при выполнении запросов к приложению действует как реальный пользователь.
2. Время отклика – время, затраченное системой на выполнение запроса пользователя
3. Запросы в секунду – измерение количества запросов, отправляемых на сервер.
4. Транзакции в секунду – измерение количества транзакций, отправляемых на целевой сервер.
5. Процент ошибок – процент ошибок от общего числа ответов в единицу времени.
6. Процессор – процентное соотношение времени, которое процессор тратит на выполнение рабочего потока.
7. Оперативная память – количество МВ в физической памяти, свободное для всех программ и процессов.
8. Жесткий диск – информация о процентах использования дискового пространства

Тестирование производительности – это процесс оценки скорости, масштабируемости и стабильности работы программного обеспечения при различных нагрузках. Целью такого тестирования является определение производительности системы в условиях ее нормального и максимального использования, а также выявление возможных узких мест и проблем, которые могут возникнуть при работе системы в реальных условиях эксплуатации.

Масштабируемость – способность системы справляться с увеличением рабочей нагрузки при добавлении ресурсов.

1. ***Структура и виды CASE-средств. Критерии при выборе.***

CASE (Computer-Aided Software Engineering) – средства автоматизированного проектирования программного обеспечения. Они помогают автоматизировать различные аспекты жизненного цикла разработки ПО, включая анализ, проектирование, реализацию, тестирование и управление проектом.

Case-средства представляют собой графически ориентированные инструменты для поддержки жизненного цикла программного обеспечения

***Критерии при выборе CASE-средства:***

* **Функциональные возможности**: CASE-средство поддерживало необходимый набор функций и инструментов для выполнения конкретных задач разработки ПО.
* **Интеграция с существующими системами**: CASE-средство должно хорошо интегрироваться с другими инструментами и средами разработки, используемыми в компании.
* **Простота использования**: Интерфейс и работа с CASE-средством должны быть интуитивно понятными и удобными для разработчиков и других участников проекта.
* **Поддержка и обновления**: Важно, чтобы CASE-средство имело активную поддержку от разработчиков и регулярные обновления для исправления ошибок и добавления новых функций.
* **Стоимость**: Цена CASE-средства должна быть доступной и соответствовать бюджету компании или проекта.
* **Сообщество и документация**: Наличие обширного сообщества пользователей и доступной документации может значительно облегчить работу с CASE-средством и решение возможных проблем.

Тип Case-средства:

* Средства анализа. Построение и анализ моделей предметной области.
* Средства анализа и проектирования. Создание проектных спецификаций компонентов и интерфейсов системы, архитектуры системы, алгоритмов и структур данных.
* Средства проектирования баз данных. Моделирование данных и генерацию схем баз данных для наиболее распространенных СУБД.
* Средства разработки приложений. Генерация программного кода компонентов системы.
* Средства реинжиниринга. Анализ программных кодов и схем БД и формирование на их основе различных моделей и проектных спецификаций.

Структура. Типовой case-пакет содержит 4 компонента:

* Центральный репозиторий.
* Средства ввода. Средства ввода предназначены для ввода данных в репозитарий, а также для организации взаимодействия пользователя с case-пакетом.
* Средства анализа, проектирования и разработки. Они предназначены для того, чтобы обеспечить проектирование и анализ различных описаний, а также различные преобразования этих описаний в процессе разработки.
* Средства вывода. Они предназначены для документирования, управления проектом и для генерации кода.

Структура CASE-средств может включать:

* Инструменты для моделирования и проектирования.
* Инструменты для генерации кода.
* Инструменты для управления проектом.
* Инструменты для анализа и тестирования.

1. ***Структуры среды разработки. Факторы при выборе.***

Структуры среды разработки (IDE – Integrated Development Environment) представляют собой комплексные программные среды, которые предоставляют разработчикам все необходимые инструменты для создания, отладки и тестирования программного обеспечения.

*Структура среды разработки включает:*

* Текстовый редактор. Это основной инструмент для написания кода. Обычно включает функции подсветки синтаксиса, автодополнения, отступов, автоформатирования и другие инструменты для улучшения продуктивности разработчика.
* Компилятор/Интерпретатор. Если вы работаете с компилируемыми языками программирования, компилятор в IDE позволяет компилировать исходный код в машинный код. Для интерпретируемых языков в IDE может быть встроен интерпретатор, который выполняет код напрямую.
* Отладчик. Инструмент, который помогает находить и исправлять ошибки в коде. Он позволяет устанавливать точки останова, следить за значениями переменных во время выполнения программы и выполнять код пошагово.
* Средства управления версиями. Интеграция с системами управления версиями (например, Git, SVN) позволяет отслеживать изменения в коде, создавать ветки, сливать изменения и восстанавливать предыдущие версии файлов.
* **Интеграция сборки и тестирования**. Некоторые IDE включают инструменты для автоматизации процесса сборки и тестирования программного обеспечения. Это позволяет разработчикам быстро и легко выполнять тесты и собирать проекты.

*Факторы при выборе среды разработки:*

* Поддержка языков программирования. IDE должна поддерживать язык или языки программирования, используемые в вашем проекте.
* Функциональность. IDE должна обладать необходимым набором функциональных возможностей, таких как автодополнение кода, отладка, рефакторинг, управление версиями, интеграция с системами сборки и тестирования и т. д.
* Интеграция с другими инструментами. IDE должно легко интегрироваться с другими инструментами и сервисами, которые используются в вашем процессе разработки, такими как системы управления версиями (например, Git).
* Совместимость. Если вы разрабатываете приложения для конкретных платформ, убедитесь, что выбранная вами IDE поддерживает нужные вам платформы.
* Производительность. IDE должна работать эффективно и не нагружать системные ресурсы вашего компьютера, особенно если вы работаете над крупными проектами.
* Цена и лицензирование.

1. ***Характеристика основных возможностей среды разработки.***

Основные возможности среды разработки включают:

* Поддержка языков программирования. Среда разработки должна поддерживать несколько языков программирования, таких как Java, C++, Python, JavaScript и других.
* **Текстовый редактор с подсветкой синтаксиса.** Предоставляет удобное окружение для написания кода с подсветкой ключевых слов и синтаксических конструкций.
* **Автодополнение кода.** Помогает ускорить процесс написания кода, предлагая варианты автозаполнения для методов, функций, переменных и других элементов кода.
* **Отладчик.**
* Интеграция с системой контроля версий, например, с Git. Это облегчает управление и совместную работу над кодом в команде.
* Расширяемость с помощью плагинов и расширений. Это позволяет адаптировать среду под конкретные потребности разработчика.
* Управление проектами, включая файлы и зависимости. Это способствует лёгкой организации кода и структуры проекта.

1. ***Основные инструменты среды для создания, исполнения и управления информационной системой.***

* Редактор кода: используется для написания и редактирования исходного кода программы.
* Компилятор или интерпретатор: преобразует исходный код в исполняемый файл или байт-код.
* Отладчик: инструмент для поиска и устранения ошибок в коде, а также для анализа производительности программы.
* Системы контроля версий: используются для управления изменениями в коде и обеспечения целостности проекта.
* Интегрированная среда разработки (IDE): объединяет в себе различные инструменты для создания, отладки и управления проектом. IDE предоставляет разработчикам удобное окружение для создания программного обеспечения. Она включает в себя текстовый редактор, отладчик, систему управления версиями, инструменты для компиляции и сборки кода, а также другие инструменты, упрощающие процесс разработки.
* Инструменты для работы с базами данных: позволяют создавать, изменять и управлять структурой базы данных. Обеспечивают надежное хранение, организацию и доступ к данным.
* Инструменты для управления проектами. Помогают организовать работу над проектом, распределять задачи, отслеживать прогресс и управлять ресурсами.
* **Среды виртуализации.** Позволяет создавать виртуальные окружения для разработки, тестирования и развертывания приложений без необходимости установки и настройки аппаратного оборудования.
* **Инструменты автоматизации тестирования.** Для тестирования информационной системы используются инструменты автоматизации тестирования, которые позволяют создавать и выполнять автоматизированные тесты для проверки работоспособности и качества приложения.

1. ***Средства обработки информации: определение, виды.***

Средства обработки информации - это инструменты и технологии, используемые для сбора, хранения, обработки, передачи и анализа данных в компьютерных системах.

Виды средств обработки информации включают программное обеспечение для баз данных, текстовых редакторов, электронных таблиц, систем управления контентом, систем управления проектами, а также аппаратное обеспечение, такое как компьютеры, серверы, сетевое оборудование и устройства хранения данных.

***Виды средств обработки информации:***

1. **Компьютеры и серверы:** Являются основными средствами обработки информации. Выполнение различных вычислительных задач, хранение и обработку данных, запуск прикладных программ и другие функции.
2. **Программное обеспечение:** Программное обеспечение включает в себя прикладные программы, операционные системы, базы данных, инструменты аналитики и другие приложения, которые используются для обработки информации в компьютерных системах.
3. **Системы хранения данных:** Это средства для хранения больших объемов данных.
4. **Сетевое оборудование:** Включает в себя коммутаторы, маршрутизаторы и другие устройства, которые обеспечивают связь и передачу данных между компьютерами и сетями.
5. **Средства защиты информации:** Это инструменты и технологии, которые обеспечивают безопасность данных и защиту информации от несанкционированного доступа, включая антивирусное программное обеспечение, брандмауэры, средства шифрования и аутентификации.
6. **Устройства ввода и вывода (ввода/вывода):** Клавиатуры, мыши, сканеры, принтеры, мониторы и другие устройства ввода и вывода используются для взаимодействия пользователя с компьютерной системой и отображения результатов обработки информации.
7. **Системы связи и передачи данных:** Системы связи, такие как сети передачи данных, Интернет, беспроводные сети и прочие технологии передачи данных, обеспечивают связь между различными компьютерами и устройствами для обмена информацией.
8. ***Организация работы в команде разработчиков. Виды моделей и роли***.

Организация работы в команде разработчиков включает в себя определение ролей, распределение задач и координацию работы между участниками команды.

1. Иерархическая модель – в этой модели команда организована по принципу пирамиды, где на вершине находится менеджер проекта, а ниже располагаются функциональные специалисты и разработчики. В этой модели команда разработчиков организована по иерархии, где каждый член имеет определенный уровень власти и ответственности. Верхние уровни принимают стратегические решения, а нижние уровни выполняют операционные задачи.
2. Гибкая модель (Scrum) – эта модель основана на использовании спринтов (коротких итераций) и предполагает наличие ролей владельца продукта, скрам-мастера, разработчиков и тестировщиков. Эта модель организации команды разработчиков основана на принципах Agile, таких как гибкость, быстрая адаптация к изменениям и сотрудничество. Команда работает в итерациях (спринтах), активно взаимодействует с заказчиком и стремится к непрерывному улучшению.
3. Гибридная модель – сочетает в себе элементы иерархической и гибкой моделей, позволяя использовать преимущества обеих подходов.
4. **Модель управления постановкой.** В этой модели команды разработчиков организованы вокруг функциональных областей продукта. Каждая команда (фича-команда) отвечает за определенный функционал продукта от начала до конца.
5. **Кросс-функциональные команды.** Эти команды включают разработчиков, тестировщиков, дизайнеров и других специалистов, необходимых для создания и поддержки продукта. Каждый член команды обладает навыками, необходимыми для выполнения различных задач.

*Модели:*

1. **Каскадная разработка.** Разработка происходит последовательно, каждая фаза зависит от завершения предыдущей: сначала идет анализ, потом проектирование, затем реализация, тестирование и поддержка.
2. **Прототипирование.** Создание прототипов позволяет быстро проверять концепции и вовлекать заказчика. Разработчики создают основной функционал, который затем дорабатывается на основе обратной связи.
3. **Итеративная и инкрементальная разработка.** Процесс разработки разбивается на короткие итерации или инкременты. На каждой итерации создается часть функционала, которая затем дорабатывается и расширяется.
4. **Гибкая разработка.** Ориентирована на гибкость и быструю адаптацию к изменениям. Работа идет в коротких циклах разработки (спринтах), с акцентом на обратную связь, гибкость и быструю поставку ценности.
5. **DevOps:** DevOps объединяет разработку и операции в единую команду, что позволяет ускорить разработку, повысить качество кода и автоматизировать процессы развертывания и мониторинга.

Роли в команде могут включать менеджера проекта, разработчиков, тестировщиков, аналитика, специалиста по качеству и других специалистов в зависимости от специфики проекта.

1. ***Система контроля версий: совместимость, установка, настройка.***

*СКВ* - механизм сохранения промежуточных состояний кода, разрабатываемого ПО. То есть с помощью этой системы программист может управлять своими файлами во времени: смотреть историю изменений файлов и каталогов, возвращаться к более ранним версиям кода и объединять несколько версий файла.

Основной областью применения контроля версий является коллективная разработка. СКВ – ПО, предназначенное для работы с постоянно изменяющейся информацией. ﻿﻿

1. Поддержка хранения файла в репозитории ﻿﻿
2. Поддержка истории версий в репозитории ﻿﻿
3. Отслеживание авторов произведений
4. ﻿﻿Нахождение конфликтов при изменении исходного кода и обеспечение синхронизации при работе в многопользовательской среде разработки.

*Git совместим с различными операционными системами (Windows, macOS, Linux) и может использоваться для управления версиями проектов любого размера.*

Совместимость: Ведущие СКВ, такие как Git, Subversion (SVN), Mercurial, совместимы с различными операционными системами (Windows, macOS, Linux) и популярными IDE.

Установка: Обычно системы контроля версий могут быть установлены через пакетные менеджеры (например, apt, yum, brew), скачав исполняемые файлы с официальных сайтов или установив через графические установщики.

Настройка: После установки, систему контроля версий следует настроить для работы с конкретным проектом. Это включает настройку имени пользователя и email (для Git), создание репозитория, настройку удаленных репозиториев (если требуется) и определение файлов, игнорируемых системой контроля версий.

Классификации СКВ.

* Централизованные / распределённые.

*Централизованных СКВ вся работа производится с центральным репозиторием, распределённых – у каждого разработчика есть локальная копия репозитория.*

* Блокирующие / неблокирующие.

*Блокирующие СКВ позволяют наложить запрет на изменение файла, пока один из разработчиков работает над ним, в неблокируюших один файд может одновременно изменяться несколькими разработчиками ﻿﻿﻿*

* Для текстовых данных / бинарных.

*Для СКВ с текстовыми данными важна поддержка слияния изменений. Для СКВ с бинарными данными важна возможность блокировки.*

1. ***Характеристика методов обеспечения кроссплатформенности информационной системы.***

Это позволяет приложению работать на различных операционных системах и устройствах, обеспечивая гибкость и доступность для широкой аудитории пользователей.

Кроссплатформенность информационной системы - это свойство системы, позволяющее ей работать на различных платформах без необходимости модификации или адаптации. Это достигается за счет использования стандартных технологий и интерфейсов, которые поддерживаются на разных платформах.

Методы обеспечения кроссплатформенности информационной системы включают:

* Использование кроссплатформенных языков программирования: Написание кода на языках, которые поддерживаются на различных платформах, таких как Java, Python, JavaScript.
* Использование кроссплатформенных фреймворков и библиотек: Использование фреймворков и библиотек, которые предоставляют абстракции для работы с операционной системой и обеспечивают кроссплатформенную совместимость.
* Виртуализация: Запуск приложения в виртуальной среде, которая может быть установлена на различных операционных системах.
* **Использование веб-технологий:** Разработка информационных систем как веб-приложений позволяет им работать на различных платформах. HTML, CSS и JavaScript являются кроссплатформенными технологиями, что обеспечивает широкую доступность приложений через браузеры на различных устройствах.
* Облачные сервисы: Разработка и развертывание приложения в облаке, что позволяет обеспечить доступ к приложению через веб-браузер с любой платформы.
* Адаптивный дизайн: Разработка пользовательского интерфейса, который адаптируется к разным размерам экранов и устройствам, что позволяет обеспечить кроссплатформенную совместимость приложения.

1. ***Особенности сервисно-ориентированной архитектуры, основные принципы.***

Сервисно-ориентированная архитектура (SOA) – это подход к проектированию информационных систем, в котором приложение разбивается на независимые компоненты, называемые сервисами, которые могут быть развернуты, управляемы и использованы независимо друг от друга.

***Особенности SOA:***

1. **Сервисы как основные компоненты:** Основной компонент SOA – сервисы, которые предоставляют конкретные функциональные возможности и могут быть вызваны другими приложениями или сервисами.
2. **Легкая связь между сервисами:** Сервисы в SOA взаимодействуют друг с другом через стандартизированные протоколы и интерфейсы, что позволяет им быть легко связанными и масштабируемыми.
3. **Независимость:** Сервисы в SOA могут быть разработаны, развернуты и использованы независимо друг от друга. Это позволяет модифицировать и обновлять один сервис без влияния на остальные компоненты системы.
4. **Переиспользование сервисов:** Поскольку сервисы в SOA являются независимыми и предоставляют конкретные функциональные возможности, их можно легко переиспользовать в различных контекстах и приложениях.
5. **Стандартизация интерфейсов и протоколов:** SOA использует стандартизированные протоколы и интерфейсы для взаимодействия между сервисами, такие как SOAP, REST, XML, JSON, что обеспечивает совместимость и универсальность.

***Основные принципы SOA:***

1. **Слабая связь.** Сервисы в SOA должны быть слабо связаны, чтобы изменения в одном сервисе не приводили к неожиданным последствиям в других.
2. **Открытость.** Сервисы должны быть открыты для использования и доступа через стандартные протоколы и интерфейсы.
3. **Абстракция.** Сервисы должны предоставлять абстрактный интерфейс, скрывающий детали реализации и обеспечивающий простоту в использовании.
4. **Переиспользование.** Сервисы должны быть разработаны с возможностью многократного использования в различных контекстах.
5. **Сервисная ориентация.** Архитектура должна быть построена вокруг сервисов, которые предоставляют конкретные функциональные возможности и могут быть использованы повторно.
6. **Автономность.** Сервисы должны быть автономными, то есть способными функционировать независимо от других сервисов и систем. SOA позволяет создавать гибкие, масштабируемые и легко поддерживаемые информационные системы, которые способны адаптироваться к изменяющимся бизнес-потребностям и технологическим требованиям.
7. ***Интегрированные среды разработки для создания независимых программ: назначение, сфера применения.***

Интегрированные среды разработки (IDE) представляют собой программные инструменты, предназначенные для упрощения процесса создания программного обеспечения. Они объединяют в себе различные инструменты, такие как редакторы кода, компиляторы, отладчики, средства управления версиями и другие, в одной среде.

***Назначение:***

1. **Создание программного обеспечения:** IDE предоставляют разработчикам все необходимые инструменты для создания программного обеспечения, начиная с написания кода и заканчивая его отладкой и тестированием.
2. **Улучшение производительности:** IDE облегчают и ускоряют процесс разработки благодаря интеграции различных инструментов и функций, таких как автодополнение кода, быстрый доступ к документации, интегрированные системы управления версиями и другие.
3. **Упрощение работы с большими проектами:** Благодаря функциям организации проектов, возможности работы с несколькими файлами и компонентами, IDE облегчают работу с крупными и сложными программными проектами.

***Сфера применения:***

1. **Разработка веб-приложений:** IDE используются для создания веб-приложений, включая фронтенд и бэкенд.
2. **Мобильная разработка:** Специализированные IDE позволяют разрабатывать мобильные приложения под различные платформы, такие как iOS, Android и Windows Phone, используя языки программирования и инструменты.
3. **Разработка настольных приложений:** IDE используются для создания настольных приложений под различные операционные системы, включая Windows, macOS и Linux, с использованием языков программирования.
4. **Разработка встроенного программного обеспечения:** IDE позволяют разрабатывать программное обеспечение для встроенных систем, таких как микроконтроллеры, микропроцессоры и другие электронные устройства.
5. ***Особенности объектно-ориентированных языков программирования.***

Объектно-ориентированные языки программирования (ООП) предоставляют специальные средства для организации программного кода в виде объектов, каждый из которых объединяет данные и методы (функции), которые могут оперировать этими данными

1. **Инкапсуляция:** ООП позволяет скрыть детали реализации объекта и предоставить только интерфейс для взаимодействия с ним. Это позволяет обеспечить защиту данных и избежать прямого доступа к ним извне объекта.
2. **Наследование:** Одним из ключевых принципов ООП является возможность создания новых классов на основе существующих, чтобы наследовать их свойства и методы. Это позволяет повторно использовать код и создавать иерархии классов для организации кода.
3. **Полиморфизм:** Полиморфизм позволяет объектам одного типа использовать методы другого типа, что делает код более гибким и упрощает его использование. Полиморфизм может быть достигнут с помощью перегрузки методов, виртуальных методов и интерфейсов.
4. **Абстракция:** Абстракция позволяет скрыть детали реализации объекта и сосредоточиться только на его важных характеристиках и функциях. Это упрощает понимание кода и его поддержку.
5. **Классы и объекты:** ООП позволяет создавать классы, которые являются шаблонами для создания объектов. Объекты, в свою очередь, являются экземплярами классов и имеют доступ к свойствам и методам, определенным в классе.
6. **События и обработчики событий:** ООП предоставляет механизмы для работы с событиями и их обработки. Это позволяет реализовывать асинхронное взаимодействие в программах.
7. **Гибкость и расширяемость:** ООП позволяет создавать модульный и гибкий код, который легко поддается изменениям и расширению. Это делает ООП подходящим для разработки крупных и сложных программных систем.
8. ***Диаграммы UML вариантов использования, последовательности, кооперации: цель построения и синтаксис.***

*Диаграмма UML вариантов использования*

Используются для описания функциональных требований к системе. Они представляют собой набор актеров и вариантов использования, которые описывают взаимодействие между актерами и системой.

Цель построения такой диаграммы – Показать функциональные требования системы и её взаимодействие с внешними акторами.

Синтаксис диаграммы вариантов использования включает в себя элементы: актеры, варианты использования, связи между ними и примечания.

Отношения:

* Ассоциации. Это связь между субъектом и вариантом использования, описывает причину связи и управляющие ею правила.
* Обобщения. Один элемент одели основан на другом.
* Включения. Один вариант использования содержит функциональные возможности другого.
* Расширения. Один вариант поведения одного расширяет поведение другого варианта использования.

*Диаграмма UML последовательности*

Показывают взаимодействие объектов во времени. Они отображают обмен сообщениями между объектами, порядок их отправки и получения, а также продолжительность каждого сообщения.

Цель построения диаграммы последовательности – моделирование взаимодействия между объектами в системе в определенной последовательности времени.

Синтаксис включает в себя объекты, стрелки синхронизации, сообщения, передаваемые между объектами, и временные оси, показывающие порядок выполнения действий. Линия жизни – вертикальная линия, представляет существование объекта в течении определенного периода времени. Фокус управления – указывает период времени, в течение которого объект выполняет некоторое действие, находясь в активном состоянии. Сообщения: синхронное (вызвать), ответное сообщение, отправить (асинхронное сообщение), создать, удалить.

*Диаграмма UML кооперации*

Отображает связи между объектами и их взаимодействие. Они показывают, какие объекты участвуют в определенном взаимодействии, какие сообщения они отправляют и получают, и как они связаны между собой.

Диаграмма кооперации используется для моделирования взаимодействия между объектами в системе, особенно акцентируя взаимодействие и передачу сообщений.

Цель: Показать взаимодействие между объектами и их связи в системе.

Синтаксис: Представляет объекты в виде прямоугольников с указанием их ролей и связи между объектами.

1. ***Диаграммы UML развертывания, компонентов, потоков данных: цель построения и синтаксис.***

*Диаграмма UML развертывания*

Показывает физическое размещение компонентов системы на узлах. Цель этой диаграммы - отобразить развертывание системы в окружающей среде, включая аппаратные средства, операционные системы и связи между ними. Синтаксис этой диаграммы включает узлы, компоненты, связи между узлами и компонентами, а также другие элементы (исполняемые файлы – артефакты).

*Диаграмма UML развертывания*

Отображает структуру системы, показывая ее компоненты и зависимости между ними.

Цель этой диаграммы состоит в том, чтобы показать, как система разбивается на компоненты и как они взаимодействуют друг с другом.

Синтаксис включает компоненты, интерфейсы, зависимости и другие элементы.

*Диаграмма UML потока данных*

Иллюстрирует поток данных в системе, показывая процессы, которые преобразуют входные данные в выходные, а также хранилища данных и внешние сущности.

Цель диаграммы потоков данных – моделирование потоков данных в системе, показывая, как данные перемещаются от источника к приемнику и как они обрабатываются на промежуточных этапах.

Синтаксис включает процессы (функции, операции), потоки данных, хранилища данных и внешние сущности, а также связи между ними.

1. ***Архитектура информационной системы: виды, типы групп, применение архитектурных описаний***.

Виды архитектуры ИС:

* Локальная. В этом случае все компоненты ИС располагаются на одном компьютере. Очевидным недостатком этой архитектуры является возможность работать в ИС только одному пользователю.
* Файл-серверная.
* Клиент-серверная. Наличие хранимых процедур дало возможность осуществлять некоторую часть обработки данных на сервере.
* Трехуровневая. В дополнение к хранимым процедурам разработчики стали использовать серверные языки программирования. Это дало возможность создавать в ИС промежуточный уровень - сервер приложений.

Виды архитектуры ИС:

* Логическая (временная, функциональная, поведенческая).
* Физическая.

***Логическая*** *архитектура* поддерживает функционирование системы на протяжении всего её жизненного цикла на логическом уровне. Она состоит из набора связанных технических концепций и принципов. Включает в себя функциональную архитектуру, поведенческую архитектуру и временную архитектуру. **Функциональная** архитектура представляет собой набор функций и их подфункций, определяющих преобразования, осуществляемые системой при выполнении своего назначения. **Поведенческая** архитектура соглашение о функциях и их подфункциях, а также интерфейсах (входы и выходы), которые определяют последовательность выполнения, условия для управления или потока данных, уровень производительности, необходимый для удовлетворения системных требований. Поведенческая архитектура может быть описана как совокупность взаимосвязанных сценариев, функций и/или эксплуатационных режимов. **Временная** архитектура является классификацией функций системы, которая получена в соответствии с уровнем частоты её исполнения. Временная архитектура включает в себя определение синхронных и асинхронных аспектов функций.

*Физическая архитектура* является систематизацией физических элементов (элементов системы и физических интерфейсов), которые реализуют спроектированные решения для продукта, услуги или предприятия. Она предназначена для удовлетворения требований к системе и элементам логической архитектуры и реализуется через технологические элементы системы.

Типы групп: функциональные, логические и физические.

Функциональная группа описаний обеспечивает представление с точки зрения пользователей или операторов, которое включает продукты, относящиеся к фазам, сценариям и потокам задач операционной системы. Также описываются и пользовательские интерфейсы. Примером продуктов, которые могут быть включены в это описание, будут функциональные данные или графики, сценарное описание (включая использование кейсов), блок-схемы задач, организационные диаграммы и схемы информационных потоков.

Логическое представление включает продукты, которые определяют системные границы с её окружением и функциональные интерфейсы с внешними системами, также основные функции и поведение системы, потоки информации, внутренние и внешние наборы данных, внутренних и внешних пользователей, и внутренние функциональные интерфейсы. Примером продуктов могут быть блочные диаграммы, IDEF0-диаграммы.

Физическая группа описаний обеспечивает представление с точки зрения проектировщиков. Продукт может включать в себя физические блок-схемы на довольно высоком уровне детализации, топологии базы данных, интерфейс управления документами и стандарты.

Применение архитектурных описаний:

- Планирование и проектирование: Описание структуры и взаимодействия компонентов системы.

- Документация: Создание документации для разработчиков, администраторов и пользователей.

- Анализ и моделирование: Использование описаний для анализа производительности, безопасности и других аспектов системы.

- Управление изменениями: Использование архитектурных описаний для оценки влияния изменений в системе.

1. ***Определение конфигурации информационной системы. Этапы, методы и средства конфигурирования.***

Конфигурация информационной системы (ИС) определяет способ, в котором её компоненты взаимодействуют друг с другом и как они настроены для достижения определенных целей. Это включает в себя аппаратное оборудование, программное обеспечение, сетевые настройки, базы данных и другие аспекты.

*Этапы конфигурации информационной системы (ИС) включают следующие действия:*

1. Изменение объектной модели.
2. Определение авторизаций пользователей.
3. Настройка интерфейса.
4. Создание типовых объектов данных: справочников, шаблонов, отчетов, процессов и т. д.
5. Настройка вариантов развертывания и взаимодействия программных компонентов, определение параметров системных служб, сервисов и т. д.

*Методы* конфигурирования могут варьироваться в зависимости от конкретных требований и особенностей проекта, но включают в себя ручную настройку, использование конфигурационных файлов, автоматизированные сценарии развертывания и управление изменениями.

*Средства* конфигурирования могут включать в себя специализированные программные продукты, системы управления конфигурациями (например, Puppet, Ansible, Chef), средства развертывания (например, Docker, Kubernetes) и инструменты управления версиями кода (например, Git). Кроме того, используются средства мониторинга и управления, чтобы обеспечить непрерывную работу системы и быструю реакцию на изменения.

1. ***Критерии выбора технических средств проекта информационной системы.***

Выбор технических средств для проекта информационной системы зависит от множества факторов, включая требования проекта, бюджет, сроки, особенности бизнеса и технические возможности.

1. **Функциональные требования:** Технические средства должны соответствовать функциональным требованиям проекта.
2. **Совместимость:** Технические средства должны быть совместимы между собой и с другими компонентами системы.
3. **Масштабируемость:** Выбранные технические средства должны быть способны масштабироваться в соответствии с ростом бизнеса и объемом данных.
4. **Безопасность:** Технические средства должны обеспечивать высокий уровень безопасности для защиты конфиденциальности и целостности данных.
5. Стоимость.
6. **Гибкость и открытость:** Выбранные технические средства должны быть гибкими и открытыми для интеграции с другими системами и расширения функциональности.
7. Надежность: технические средства должны быть надежными и обеспечивать стабильную работу системы в течение длительного времени.
8. ***Репозиторий проекта информационной системы: понятие и описание процесса создания.***

*Репозиторий проекта информационной системы* – это централизованное хранилище, в котором хранятся все файлы, документация, исходный код, конфигурационные файлы и другие ресурсы, необходимые для разработки, тестирования, управления версиями и совместной работы над проектом.

*Процесс создания:*

1. **Выбор системы контроля версий:** Наиболее распространенными системами контроля версий являются Git, Subversion (SVN), Mercurial и другие. Выбор системы контроля версий зависит от предпочтений команды разработки, требований проекта и удобства использования.
2. **Создание репозитория:** После выбора системы контроля версий необходимо создать сам репозиторий. Обычно это делается с помощью команды или интерфейса выбранной системы контроля версий.
3. **Настройка прав доступа:** В зависимости от требований проекта и политики безопасности, необходимо настроить права доступа к репозиторию. Это может включать в себя определение различных уровней доступа.
4. **Инициализация репозитория:** После создания репозитория необходимо инициализировать его, добавив в него начальные файлы, структуру каталогов и другие необходимые ресурсы.
5. **Настройка веток и ветвлений:** В зависимости от модели разработки (например, Git Flow) может потребоваться настройка веток и ветвлений в репозитории для управления различными версиями и функциональными потоками разработки.
6. **Добавление документации и руководств:** Хорошей практикой является добавление в репозиторий документации, инструкций по развертыванию, руководств пользователя и других ресурсов, необходимых для понимания и использования проекта.
7. **Непрерывная поддержка и обновление:** Репозиторий проекта должен поддерживаться и обновляться на протяжении всего жизненного цикла проекта. Это включает в себя регулярное добавление нового кода, исправление ошибок, обновление документации и управление версиями.
8. ***Алгоритм определения уровня доступа к репозиторию в системе контроля версий.***

Определение уровня доступа к репозиторию в системе контроля версий обычно осуществляется на основе ролей и задач участников проекта.

1. **Идентификация участников проекта:** Определите всех участников проекта, которым потребуется доступ к репозиторию.
2. **Определение ролей и задач:** Разделите участников проекта на роли в соответствии с их задачами и обязанностями.
3. **Выделение уровней доступа:** Определите несколько уровней доступа к репозиторию в зависимости от ролей участников. Обычно это включает уровни доступа, такие как чтение, запись и администрирование.
4. **Назначение прав доступа:** Назначьте уровни доступа каждой роли в соответствии с их обязанностями и требованиями проекта.
5. **Настройка прав доступа в системе контроля версий:** В зависимости от выбранной системы контроля версий (например, Git, SVN), настройте права доступа к репозиторию для каждой роли. Это может быть достигнуто с помощью настройки ACL (списков управления доступом), ролей пользователя или других механизмов, предоставляемых системой.
6. **Тестирование и администрирование:** После настройки прав доступа необходимо протестировать функциональность и убедиться, что каждый участник имеет необходимый уровень доступа. Также важно регулярно администрировать и обновлять права доступа в соответствии с изменениями в составе команды и требованиями проекта.
7. ***Процесс настройки среды разработки информационной системы: описание установки предпочитаемых инструментов и языков кода.***

Настройка среды разработки информационной системы включает в себя установку и настройку различных инструментов, языков программирования и других компонентов, необходимых для разработки и тестирования системы.

1. **Выбор интегрированной среды разработки (IDE).**
2. **Установка языков программирования.** Это может включать в себя установку компиляторов, интерпретаторов и других средств разработки для языков, таких как Java, Python, JavaScript, C#, PHP и других.
3. **Установка систем контроля версий.**
4. Настройку рабочего окружения, включая параметры редактора кода, отладчика, конфигураций сборки и т. д.
5. **Настройка базы данных и серверных технологий:**
6. ***Инструменты мониторинга разработки проекта.***

* Системы управления задачами (например, Jira, Trello, Asana), которые позволяют отслеживать задачи, назначать исполнителей, устанавливать сроки выполнения и контролировать прогресс.
* Системы контроля версий (например, Git, Mercurial, SVN), которые позволяют вести историю изменений кода, откатывать изменения, работать над проектом нескольким разработчикам одновременно.
* Средства непрерывной интеграции и доставки (CI/CD, Continuous Integration/Continuous Delivery), которые автоматизируют сборку, тестирование и развертывание приложения на разных платформах.
* Средства логирования и мониторинга (например, Logstash, Elasticsearch, Kibana), которые собирают и анализируют данные о работе приложения, выявляя проблемы и оптимизируя производительность.
* Средства анализа кода (SonarQube, CheckStyle, PMD), которые помогают находить и исправлять ошибки, улучшать качество и читаемость кода.

1. ***Требования к пользовательскому интерфейсу информационной системы.***

Требования к пользовательскому интерфейсу информационной системы зависят от целей системы, её целевой аудитории, контекста использования и других факторов.

1. **Простота использования:** Интерфейс должен быть интуитивно понятным и легко использоваться даже без специального обучения.
2. Доступность: интерфейс должен обеспечивать доступность для всех пользователей, включая людей с ограниченными возможностями.
3. **Эффективность и производительность:** Интерфейс должен быть спроектирован таким образом, чтобы пользователи могли выполнять задачи быстро и эффективно.
4. **Наглядность и понятность:** Интерфейс должен предоставлять пользователю достаточно информации о текущем состоянии системы, процессах и возможностях.
5. **Адаптивность и отзывчивость.**
6. Интерфейс должен обеспечивать безопасность данных и операций пользователя.
7. **Поддержка многоязычности и локализации:** Если система предназначена для использования в разных странах или регионах, интерфейс должен поддерживать многоязычность и локализацию, чтобы пользователи могли комфортно использовать систему на своем родном языке.
8. ***Принципы создания графического пользовательского интерфейса (GUI).***
9. Понятность: интерфейс должен быть понятным и очевидным для пользователя.
10. Интуитивность: интерфейс должен быть интуитивным и естественным для пользователя, чтобы он мог быстро и легко выполнять задачи.
11. Простота: интерфейс должен быть простым и минималистичным, без лишних элементов и сложностей.
12. Гибкость: интерфейс должен быть гибким и адаптивным, чтобы соответствовать различным устройствам, разрешениям и ориентациям экрана.
13. Эффективность: интерфейс должен быть эффективным и быстрым, чтобы пользователь мог выполнять свои задачи без задержек и проблем.
14. ***Понятие и формы спецификации языка программирования для разработки информационной системе.***

Спецификация языка программирования – это формальное описание синтаксиса, семантики, стандартов и других характеристик языка программирования. Она определяет правила, которыми должны следовать программисты при написании программ на данном языке.

Формы спецификации могут быть различными, например, это может быть текстовый файл с описанием языка, руководство пользователя, онлайн-документация и т.д.

*Формы спецификации языка программирования могут включать:*

1. **Грамматика:** Формальная грамматика языка программирования описывает структуру его синтаксиса, включая правила образования выражений, операторов, функций и других конструкций языка.

2. **Семантика:** Семантика определяет значение выражений и операций языка программирования. Она описывает, как интерпретировать и выполнять программы на данном языке, включая порядок вычислений, правила типизации, обработки ошибок и другие аспекты.

3. **Стандарты языка:** Стандарт языка программирования определяет его текущую версию, набор возможностей, функций, библиотек, а также правила совместимости между различными реализациями языка. 4. **Документация и руководства:** Документация по языку программирования предоставляет информацию о его синтаксисе, семантике, стандартах, библиотеках, а также примеры использования и советы по разработке.

5. **Формальные спецификации:** Некоторые языки программирования имеют формальные спецификации, которые могут быть использованы для автоматической генерации компиляторов, интерпретаторов и других инструментов разработки.

6. **Примеры кода:** Примеры кода на языке программирования могут служить спецификацией, показывая, как использовать различные функции и конструкции языка.

1. ***Методы организации ввода-вывода информации в информационной системе.***
2. **Графический пользовательский интерфейс (GUI):** GUI предоставляет графическую среду для взаимодействия пользователя с системой. Пользователи могут взаимодействовать с приложением с помощью элементов интерфейса, таких как кнопки, текстовые поля, списки, меню и др.
3. **Текстовый интерфейс командной строки (CLI):** CLI предоставляет интерфейс, основанный на тексте, в котором пользователь взаимодействует с системой, вводя команды и получая текстовые ответы.
4. **Ввод с помощью форм и веб-страниц:** Формы и веб-страницы позволяют пользователям вводить информацию, заполняя текстовые поля, выпадающие списки, радиокнопки и другие элементы веб-интерфейса.
5. **Файловый ввод-вывод:** Файловый ввод-вывод позволяет программам читать данные из файлов на диске и записывать данные в файлы.
6. **Ввод с помощью сенсорных устройств.**
7. **Ввод с помощью голосовых команд:** В некоторых системах пользовательский ввод может осуществляться с помощью голосовых команд.
8. ***Спецификация настроек типовой информационной системы.***

Спецификация настроек типовой информационной системы включает в себя описание конфигурации, параметров и параметров, необходимых для настройки и функционирования системы.

Конкретные настройки зависят от типа информационной системы и ее функциональности.

1. **Настройки подключения к базе данных**.
2. **Настройки безопасности:** (права доступа к функциональным возможностям системы для различных пользовательских ролей; методы аутентификации и авторизации пользователей; настройки шифрования данных и защиты от несанкционированного доступа).
3. **Настройки интерфейса пользователя.**
4. **Настройки резервного копирования и восстановления.**
5. **Настройки журналирования и мониторинга.**
6. ***Сообщения между модулями: назначения, вход и разделы модуля.***

В обмене информацией и координации работы в информационных системах.

**Назначение сообщений:**

* Сообщения используются для передачи данных между различными модулями системы.
* Сообщения позволяют модулям взаимодействовать и обмениваться информацией для выполнения определенных задач.
* Сообщения могут использоваться для синхронизации действий между модулями и обеспечения правильного порядка выполнения операций.

Вход: определяет входные данные, которые необходимы для его работы. Данные для обработки.

Разделы модуля определяют структуру и организацию кода внутри него.

Разделы модуля:

* Прием сообщений: Модуль должен иметь механизм для приема входящих сообщений от других модулей. Обычно это реализуется через специальные интерфейсы или обработчики событий.
* Обработка сообщений: После получения сообщения модуль должен обработать его в соответствии с его целями и задачами. Это может включать в себя анализ данных, выполнение операций и отправку ответных сообщений.
* Отправка сообщений: Модуль может отправлять сообщения другим модулям для совместной работы и взаимодействия. Это может быть, как ответ на полученное сообщение, так и инициированное модулем действие.

1. ***Приложения для моделирования процессов: виды моделей, требования.***

Приложения для моделирования процессов предоставляют инструменты для анализа, проектирования и оптимизации бизнес-процессов. Они используются для визуализации, анализа и управления процессами в организации.

Виды моделей:

1. **Диаграммы потока процесса (Flowcharts).**
2. **Диаграммы UML (Unified Modeling Language).**
3. **Диаграммы BPMN (Business Process Model and Notation):** Специализированные диаграммы для моделирования бизнес-процессов с акцентом на бизнес-ориентированные элементы, такие как задачи, события, потоки данных и т. д.
4. **Диаграммы сетевого планирования (PERT и Gantt):** Используются для моделирования проектных процессов, включая определение зависимостей задач, временных оценок, расписания и т. д.

Требования:

1. **Интерактивность и простота использования:**
2. **Поддержка различных видов моделей:** Приложение должно поддерживать широкий спектр моделей, таких как диаграммы потока процесса, UML, BPMN, сетевое планирование и другие.
3. **Возможности анализа и симуляции:** Пользователям должны быть доступны инструменты для анализа процессов, определения узких мест, оценки времени выполнения и оптимизации производительности.
4. **Коллаборативные возможности:** Приложение должно поддерживать работу нескольких пользователей над одной моделью, обмен комментариями, версионирование и контроль доступа.
5. **Интеграция с другими системами:** Возможность импорта и экспорта данных из других систем, таких как системы управления проектами, системы учета задач, CRM-системы и т. д.
6. **Генерация отчетов и документации:** Возможность автоматической генерации отчетов, документации и презентаций на основе моделей процессов.
7. **Поддержка расширений и настраиваемых компонентов:** Возможность расширения функциональности приложения путем добавления пользовательских компонентов, шаблонов и интеграции с внешними сервисами.
8. ***Разработка приложений: виды, этапы разработки.***

Виды: веб-приложения, мобильные приложения, десктопные приложения, встраиваемые приложения.

Этапы: Анализ требований, проектирование, разработка, тестирование, внедрение, сопровождение.

1. ***Отладки приложения: определение, существующие методы.***

Отладка приложения – это процесс поиска, идентификации и устранения ошибок (багов) в программном обеспечении.

1. **Использование отладочных инструментов IDE.**
2. Ручная отладка. Анализ состояния вычислительного процесса выполняет разработчик на основании своего опыта, знаний и интуиции.
3. Пошаговая отладка. Состояние вычислительного процесса анализируют после выполнения каждого оператора (команды) программы.
4. Отладка по предварительно заданным точкам останова программы. Позволяет после выполнения заданного заранее множества операторов приостановить на время выполнение программы.
5. Отладка обратного выполнения. Анализ вычислительного процесса начинается с состояния, в котором зафиксирована ошибка, и заканчивается состоянием, идентифицируемым как причина ошибки.
6. Трассировка. Предоставляет для анализа информацию о поведении программы.
7. ***Интеграция модуля: определение, возможность, способы интеграции.***

Интеграция модуля – это процесс объединения отдельных компонентов или модулей программного обеспечения в единое целое с целью создания полноценной работающей системы.

Возможность интеграции может быть определена на основе совместимости интерфейсов модулей, а способы интеграции могут включать использование стандартных интерфейсов, разработку собственных протоколов или использование сторонних решений.

Возможности интеграции модуля:

1. **Обмен данными:** Модули могут обмениваться данными, передавая информацию между собой для выполнения определенных задач.
2. Модули могут использовать общие ресурсы и сервисы, такие как базы данных, файловые хранилища, очереди сообщений и т. д.
3. Модули могут взаимодействовать через события, генерируемые и обрабатываемые в различных частях системы.
4. Интеграция на уровне данных: обмен информацией между модулями через общую базу данных или API.
5. Интеграция на уровне интерфейса: взаимодействие модулей через стандартизированные интерфейсы и протоколы обмена данными.
6. Интеграция на уровне бизнес-логики: согласование и совмещение бизнес-процессов и правил, реализованных в различных модулях.

Способы интеграции модуля:

1.Разработка приложения с использованием модульной архитектуры, где каждый модуль отвечает за определенную функциональность или часть приложения.

2. **Использование стандартов и протоколов:** Использование стандартных протоколов и форматов данных для обмена информацией между модулями, таких как RESTful API, SOAP, JSON, XML и т. д.

3. **API-интерфейсы:** Создание API-интерфейсов для взаимодействия между модулями, обеспечивающих доступ к функциональности через определенные точки взаимодействия.

4. **Интеграционное тестирование:** Проведение интеграционного тестирования для проверки взаимодействия между модулями и выявления возможных проблем.