**Оглавление**

[**1 Валидация (аттестация) и верификация.** 1](#_Toc185989728)

[**2 Общие вопросы верификации ПО. Цели и задачи верификации ПО.** 2](#_Toc185989729)

[**3 Статические и динамические методы верификации ПО.** 3](#_Toc185989730)

[**4 Факторы и атрибуты внешнего и внутреннего качества ПО.** 4](#_Toc185989731)

[**5 Методы тестирования.** 5](#_Toc185989732)

[**6 Классификация тестирования по уровням.** 6](#_Toc185989733)

[**7 Тестирование производительности.** 7](#_Toc185989734)

[**8 Смоук-тестирование.** 8](#_Toc185989735)

[**9 Регрессионное тестирование.** 9](#_Toc185989736)

[**10 Тестирование «черным ящиком».** 10](#_Toc185989737)

[**11 Тестирование нефункциональных требований.** 11](#_Toc185989738)

[**12 Взаимосвязь разработки и тестирования. V-модель разработки ПО.** 12](#_Toc185989739)

[**13 Уровни тестирования. Модульное (unit), интеграционное (integration).** 13](#_Toc185989740)

[**14 Системное (system), приемочное (acceptance) тестирование.** 14](#_Toc185989741)

[**15 Техники тест дизайна. Разбиение на классы эквивалентности и тестирование граничных значений.** 15](#_Toc185989742)

[**16 Понятие дефекта. Основные определения и классификация дефектов** 16](#_Toc185989743)

[**17 Виды ошибок. Методы отладки.** 17](#_Toc185989744)

[**18 Модульное тестирование.** 18](#_Toc185989745)

[**19 Интеграционное тестирование.** 19](#_Toc185989746)

[**20 Средства разработки технической документации.** 20](#_Toc185989747)

[**21 Технологии разработки документов.** 21](#_Toc185989748)

[**22 Документирование программного обеспечения в соответствии с Единой системой программной документации.** 22](#_Toc185989749)

[**23 Автоматизация разработки технической документации.** 23](#_Toc185989750)

[**24 Инструментальные средства поддержки тестирования.** 24](#_Toc185989751)

[**25 Артефакты разработки ПО, относящиеся к тестированию. Тест-кейсы (test cases).** 25](#_Toc185989752)

[**26 Артефакты разработки ПО, относящиеся к тестированию. План тестирования (test plan).** 26](#_Toc185989753)

[**27 Обеспечение качества (Quality Assurance).** 27](#_Toc185989754)

# **1 Валидация (аттестация) и верификация.**

**Верификация** — это процесс проверки того, что система или её компоненты соответствуют заданным требованиям на определённом этапе разработки. Верификация отвечает на вопрос: *"Создаётся ли правильный продукт?"* Это проверка соответствия проектной документации, кода и архитектуры установленным стандартам и спецификациям.

**Валидация** — это процесс подтверждения, что система удовлетворяет требованиям пользователя и решает поставленные задачи. Валидация отвечает на вопрос: *"Строится ли правильный продукт?"* Она подтверждает, что ПО выполняет нужные функции и соответствует требованиям конечного пользователя.

# **2 Общие вопросы верификации ПО. Цели и задачи верификации ПО.**

**Цели верификации:**

* Подтверждение, что продукт соответствует спецификациям и требованиям.
* Обеспечение того, чтобы не было ошибок на ранних стадиях разработки.
* Обеспечение качества продукта и его соответствия стандартам.

**Задачи верификации:**

* Анализ и проверка документации (например, требований и спецификаций).
* Проверка соответствия кода стандартам и спецификациям.
* Проверка на отсутствие ошибок при использовании различных статических методов (например, рецензии кода, статический анализ).
* Разработка тестов для дальнейшей валидации.

# **3 Статические и динамические методы верификации ПО.**

Статические методы верификации не требуют выполнения программы. Они анализируют код, документацию и проектные артефакты на соответствие стандартам и спецификациям. Примеры:

* Ревью кода — проверка кода коллегами.
* Анализ кода — использование инструментов для выявления потенциальных ошибок в коде (например, линтеры).
* Математическое моделирование — использование формальных методов для доказательства корректности.

Динамические методы верификации предполагают выполнение программы с целью выявления ошибок. Примеры:

* Тестирование — выполнение программы с различными входными данными для проверки правильности работы.
* Анализ покрытия — проверка, какие части программы были выполнены при тестировании.

# **4 Факторы и атрибуты внешнего и внутреннего качества ПО.**

Внешнее качество ПО относится к аспектам, которые влияют на восприятие пользователя. Атрибуты:

* Удобство использования — насколько интерфейс и функциональность понятны и интуитивно понятны.
* Производительность — как быстро и эффективно работает программа.
* Надежность — как часто программа выходит из строя и насколько она устойчива к сбоям.
* Безопасность — защита от несанкционированного доступа и утечек данных.

Внутреннее качество ПО связано с архитектурой, кодом и процессами разработки. Атрибуты:

* Поддерживаемость — насколько легко вносить изменения в код.
* Читаемость кода — насколько код понятен и легко модифицируем.
* Тестируемость — насколько легко проводить тестирование программы.
* Реализуемость — насколько легко реализовать функциональные требования.

# **5 Методы тестирования.**

**Ручное тестирование** — это тестирование, которое выполняется вручную, без использования автоматических инструментов. Примеры: функциональное тестирование, тестирование пользовательского интерфейса.

**Автоматизированное тестирование** — использование программных средств для выполнения тестов. Примеры: тесты на юнит-уровне, интеграционные тесты. Преимущества — высокая скорость выполнения, повторяемость.

**Тестирование черного ящика** — тестирование без учета внутренней структуры приложения, фокус на вводных и выходных данных.

**Тестирование белого ящика** — тестирование с учетом структуры и кода приложения, включая тестирование на уровне кода и логики.

**Интеграционное тестирование** — тестирование взаимодействия различных компонентов системы.

**Регрессионное тестирование** — тестирование для проверки, что новые изменения не сломали существующую функциональность.

**Тестирование нагрузки** — проверка, как система ведет себя при высокой нагрузке.

**Тестирование безопасности** — проверка системы на уязвимости и слабые места.

# **6 Классификация тестирования по уровням.**

Классификация тестирования по уровням описывает стадии проверки программного обеспечения в зависимости от его структуры и готовности.

**Уровни тестирования:**

1. **Модульное тестирование:**  
   Проверка отдельных компонентов или функций программы. Например, тестирование функции вычисления факториала в Python. Цель — убедиться, что каждый модуль работает правильно изолированно.
2. **Интеграционное тестирование:**  
   Проверка взаимодействия между модулями или компонентами. Например, тестирование совместной работы функций авторизации и проверки прав доступа. Цель — убедиться, что модули правильно работают вместе.
3. **Системное тестирование:**  
   Проверка всей системы в целом на соответствие требованиям. Например, тестирование приложения "КиберШкола", чтобы убедиться, что пользовательские сценарии (регистрация, добавление задания) выполняются корректно.
4. **Приёмочное тестирование:**  
   Финальная проверка системы пользователями или заказчиком перед вводом в эксплуатацию. Например, клиент тестирует приложение для оценки удобства интерфейса и правильности работы ключевых функций.

**Пример для Python:**

* **Модульное тестирование** — используется pytest для проверки функций.
* **Интеграционное** — проверяет связь между модулями через тестовые сценарии.
* **Системное** — запускается вся программа для тестирования полного сценария.
* **Приёмочное (тест-кейсы)** — проводится вручную с участием конечного пользователя.

Эти уровни помогают выявить и устранить ошибки на разных стадиях разработки.

# **7 Тестирование производительности.**

Тестирование производительности — это процесс оценки поведения системы при различных нагрузках, чтобы проверить её скорость, стабильность, масштабируемость и потребление ресурсов.

**Основные виды тестирования производительности:**

1. **Нагрузочное тестирование:**  
   Проверяет, как система работает при увеличении количества пользователей или запросов, оставаясь в пределах нормальных условий. Например, можно проверить, сколько пользователей одновременно могут работать с веб-приложением.
2. **Стресс-тестирование:**  
   Проверяет систему в условиях экстремальных нагрузок, чтобы выявить её пределы и устойчивость. Например, отправить на сервер запросов больше, чем обычно, чтобы увидеть, как он будет реагировать.
3. **Тестирование на стабильность:**  
   Проверяет, как долго система может работать под стабильной нагрузкой без ошибок. Например, работа сервера под нагрузкой в течение 24 часов.
4. **Тестирование на масштабируемость:**  
   Оценивает способность системы увеличивать производительность при добавлении ресурсов (например, процессоров или памяти).

**Инструменты и подходы:**

* В Python для тестирования производительности используются библиотеки, такие как time (для измерения времени выполнения функций) или cProfile (для анализа производительности кода).
* Для веб-приложений часто применяются внешние инструменты, например, JMeter или Locust.

**Пример в Python:**  
Измерение времени выполнения функции:

import time

start\_time = time.time()

# выполнение функции

end\_time = time.time()

print("Время выполнения:", end\_time - start\_time)

Тестирование производительности важно для обеспечения качества и надёжности системы при реальной эксплуатации.

# **8 Смоук-тестирование.**

**Смоук-тестирование** — это тип тестирования, который направлен на быструю проверку основных функций системы, чтобы убедиться, что они работают должным образом. Это своего рода "тест на стабильность", который помогает определить, можно ли продолжать более глубокое тестирование системы. Основная цель смоук-тестирования — удостовериться, что критически важные компоненты системы функционируют корректно после изменений, например, после новой сборки или исправлений.

**Смоук-тестирование с использованием тест-кейсов:**

Смоук-тестирование можно проводить с помощью тест-кейсов. Тест-кейсы — это заранее подготовленные шаги, которые описывают, как провести тестирование конкретной функции системы. Они включают входные данные, ожидаемые результаты и шаги выполнения теста. Тест-кейсы для смоук-тестирования обычно охватывают основные, критически важные функции. Например, для веб-приложения это может быть проверка загрузки страницы, входа в систему, отправки формы или других базовых операций. Если эти операции работают как ожидается, можно продолжать более подробное тестирование.

**Смоук-тестирование как тестирование "черным ящиком":**

Смоук-тестирование часто проводится в рамках метода тестирования "черным ящиком". Это означает, что тестировщик проверяет систему, не имея доступа к её внутренней структуре или исходному коду. Вместо этого он фокусируется на поведении системы через её интерфейсы, анализируя, как она выполняет свои функции, исходя из того, что доступно пользователю. Важно отметить, что тестировщик не рассматривает, как система решает задачи внутри, а лишь проверяет, правильно ли она выполняет те действия, которые от неё ожидаются с внешней стороны.

**Основные характеристики смоук-тестирования:**

1. **Цель:** проверить, работают ли основные функции системы, чтобы удостовериться, что она в целом стабильна для дальнейшего тестирования.
2. **Проверка ключевых операций:** например, в веб-приложении — это может быть проверка загрузки страницы, успешного входа в систему и выполнения основных действий.
3. **Простой процесс:** Смоук-тестирование не требует глубокого анализа всех функций, а лишь проверяет критические компоненты. Оно выполняется быстро и помогает обнаружить основные проблемы.
4. **Тестирование "черным ящиком":** Тестировщик проверяет только поведение системы, не зная её внутренней структуры или реализации. Это тестирование фокусируется на том, как система работает снаружи.

Смоук-тестирование важно для обеспечения того, чтобы система не имела крупных дефектов, которые могут помешать дальнейшему тестированию или использованию.

# **9 Регрессионное тестирование.**

**Регрессионное тестирование** — это процесс повторного тестирования программы после внесения изменений в её код (например, после исправления ошибок, добавления новых функций или обновлений), чтобы удостовериться, что эти изменения не привели к сбоям или ухудшению работы существующих функций. Его цель — проверить, что новая версия программы не нарушила ранее функционирующие части системы.

**Задачи регрессионного тестирования:**

1. **Проверка стабильности системы:** убедиться, что внесенные изменения не повлияли на корректность работы старых функций.
2. **Поиск новых ошибок:** несмотря на то что тестирование фокусируется на уже проверенных функциях, важно выявить и новые дефекты, которые могут возникнуть из-за изменений в коде.
3. **Подтверждение исправлений:** проверка, что ранее обнаруженные ошибки действительно были исправлены и не появляются снова.

**Важные моменты:**

* **Автоматизация:** Регрессионное тестирование часто автоматизируется, чтобы ускорить процесс повторного тестирования при каждом изменении кода. Это особенно важно для крупных систем, где множество функций могут быть затронуты.
* **Масштаб тестирования:** Регрессионное тестирование может охватывать все или только те участки системы, которые были изменены. В некоторых случаях проводят выборочное тестирование, проверяя лишь те части, которые были затронуты изменениями.

**Методы регрессионного тестирования:**

1. **Полное тестирование:** включает тестирование всей системы. Это более затратный и долгий процесс, но дает максимальную уверенность, что изменения не повлияли на систему в целом.
2. **Выборочное тестирование:** проверяются только те части программы, которые были изменены, а также те, которые могут быть косвенно затронуты этими изменениями.
3. **Тестирование по критериям:** выбираются наиболее важные функции, которые должны работать без сбоев, и проводится тестирование их стабильности.

# **10 Тестирование «черным ящиком».**

**Тестирование "черным ящиком"** (Black-box testing) — это метод тестирования программного обеспечения, при котором проверяется функциональность системы без учета её внутренней структуры и реализации. Тестировщик не имеет доступа к исходному коду и не интересуется тем, как работает программа внутри. Вместо этого внимание уделяется только входным данным и ожидаемым результатам.

**Основные принципы тестирования "черным ящиком":**

1. **Отсутствие знания о реализации:** Тестировщик не должен знать, как устроен код программы, а лишь использовать её интерфейсы и функциональные возможности.
2. **Проверка на основе требований:** Тестирование проводится на основе спецификаций, требований или документации, чтобы удостовериться, что система выполняет все предусмотренные функции правильно.
3. **Тестирование всех возможных входных данных:** Проводится проверка различных входных данных для того, чтобы убедиться в правильности работы программы при разных условиях.

**Виды тестирования "черным ящиком":**

1. **Функциональное тестирование:** Проверяется, выполняет ли программа все заявленные функции и соответствует ли результатам, указанным в спецификациях.
2. **Тестирование интерфейса:** Проверяется взаимодействие пользователя с программой через интерфейсы (например, графический интерфейс или API).
3. **Тестирование безопасности:** Анализируется, как программа реагирует на попытки несанкционированного доступа, неправильные вводы и другие угрозы безопасности.
4. **Тестирование производительности:** Оценивается, как система работает под различными нагрузками, например, при большом объеме входных данных.
5. **Тестирование совместимости:** Проверяется, как система работает в разных операционных системах, браузерах, на различных устройствах и т.д.

**Преимущества тестирования "черным ящиком":**

* **Независимость от разработчиков:** Тестировщик не зависит от внутренней структуры программы, что позволяет объективно оценить её поведение.
* **Оценка работы по функциональности:** Проверяется, соответствует ли приложение заявленным требованиям и работает ли оно корректно с точки зрения конечного пользователя.
* **Идеально для тестирования интерфейсов и пользовательского опыта:** Особенно полезно для тестирования веб-приложений, мобильных приложений, а также API, где важно взаимодействие с системой на уровне пользовательского ввода и вывода.

**Недостатки:**

* **Ограниченная проверка:** Без знания внутренней структуры приложения нельзя всегда проверить все возможные пути выполнения программы. Это может привести к пропуску некоторых ошибок.
* **Неопределенность в охвате тестов:** Поскольку тестирование происходит без понимания кода, сложно точно определить, какие части программы были протестированы, а какие — нет.

Таким образом, тестирование "черным ящиком" фокусируется на тестировании функциональности и поведения программы, что делает его особенно полезным для проверки работы системы с точки зрения конечного пользователя.

# **11 Тестирование нефункциональных требований.**

Тестирование нефункциональных требований — это важная часть процесса обеспечения качества программного обеспечения, которая фокусируется на том, как система работает, а не на том, что она делает.

Нефункциональные требования могут включать в себя производительность, безопасность, удобство использования, совместимость и другие аспекты.

**1. Производительность**

• Тестирование нагрузки: Определяет, как система справляется с высоким уровнем нагрузки. Используются инструменты, такие как Apache JMeter или LoadRunner.

• Тестирование стресса: Проверяет, как система работает при экстремальных условиях (например, при превышении максимальной нагрузки).

• Тестирование стабильности: Оценивает, как система ведет себя в течение длительного времени под нагрузкой.

**2. Безопасность**

• Тестирование уязвимостей: Используются инструменты для выявления уязвимостей, такие как OWASP ZAP или Burp Suite.

• Тестирование аутентификации и авторизации: Проверка правильности работы механизмов доступа к системе.

• Тестирование защиты данных: Проверка шифрования данных и их хранения.

**3. Удобство использования (Usability)**

• Юзабилити-тестирование: Оценка удобства интерфейса пользователями. Это может включать в себя наблюдение за пользователями, опросы и интервью.

• Тестирование доступности: Проверка того, насколько приложение доступно для людей с ограниченными возможностями.

4. **Совместимость**

• Тестирование на разных платформах: Проверка работы приложения на различных операционных системах, браузерах и устройствах.

• Тестирование интеграции: Проверка взаимодействия системы с другими системами и сервисами.

5. **Надежность**

• Тестирование восстановления: Проверка способности системы восстанавливаться после сбоев.

• Тестирование отказоустойчивости: Оценка поведения системы при отключении или сбоях отдельных компонентов.

6. **Поддерживаемость**

• Тестирование документации: Оценка качества документации для разработчиков и пользователей.

• Тестирование обновлений: Проверка процесса обновления системы и совместимости новых версий с предыдущими.

7. **Локализация**

• Тестирование перевода: Проверка правильности перевода интерфейса и содержания на разные языки.

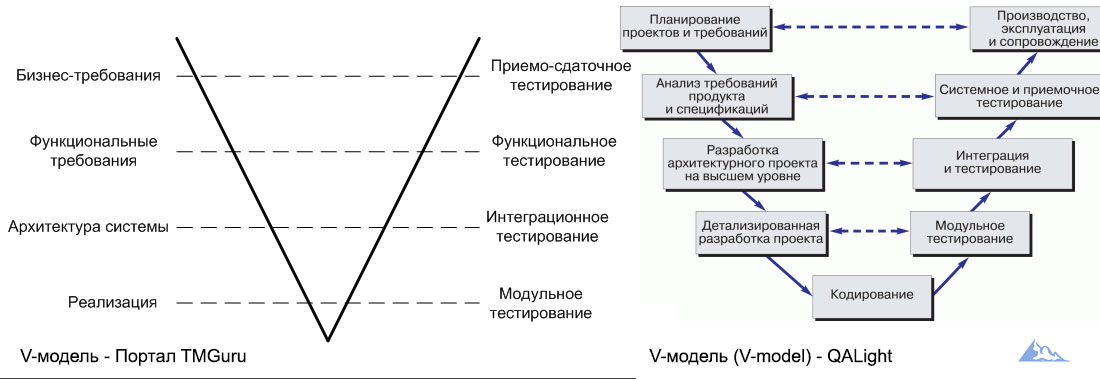
• Тестирование форматов данных: Оценка корректности отображения дат, валют и других локализованных данных.

# **12 Взаимосвязь разработки и тестирования. V-модель разработки ПО.**

Взаимосвязь разработки и тестирования заключается в тесном сотрудничестве между командами для создания качественного и надёжного программного обеспечения.

Обсуждение требований и постановка задач. Это помогает тестировщику разработать эффективные тестовые сценарии и определить ключевые области для тестирования. Проходит проведение различных видов тестирования, таких как модульное, интеграционное и системное.

Важно документировать все найденные дефекты и передавать их разработчикам для исправления. Исправленные ошибки не должны вызвать новых проблем в системе. Разработчики и тестировщики совместно анализируют результаты модульных тестов, определяя приоритеты для дальнейшей отладки и доработки кода.



**V-модель** — это тип модели **SDLC**, в которой процесс выполняется последовательно в**V-образной форме**. Модель основана на объединении фазы тестирования с каждой соответствующей стадией разработки. Разработка каждого шага напрямую связана с этапом тестирования. Следующая фаза начинается только после завершения предыдущей. Каждый этап разработки, напрямую связан с тестированием этого этапа.

# **13 Уровни тестирования. Модульное (unit), интеграционное (integration).**

Выделяют 4 основных уровня тестирования:

* + 1. Компонентное/модульное тестирование (Component/Unit Testing)
    2. Интеграционное тестирование (Integration Testing)
    3. Системное тестирование (System Testing)
    4. Приемочное тестирование (Acceptance Testing)

Модульное.

Этот вид тестирования выполняется на самой ранней стадии разработки программы — во время написания кода. Обычно его выполняет сам программист, ошибки исправляются сразу же и не попадают к специалистам по тестированию.

Направлено на тестирование отдельных модулей и компонентов программы, которые изолированы от других модулей и компонентов. Поэтому его стоит совмещать с другими видами тестирования, сам по себе он малоэффективен.

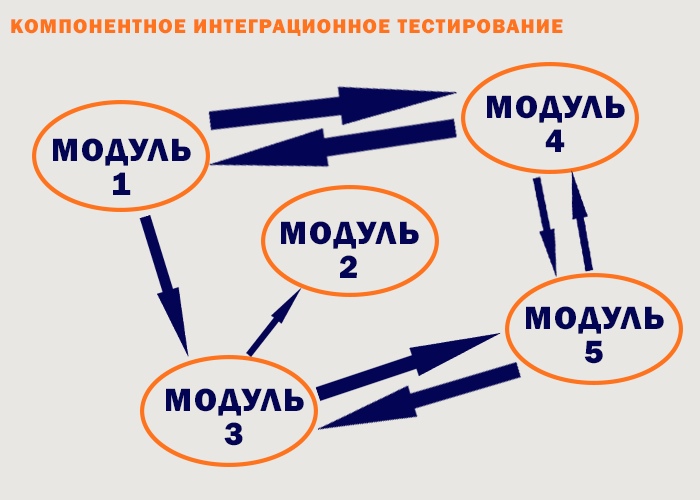
Для этого уровня тестирования характерно несколько целей:

1. Проверка компонента на соответствие требованиям,
2. Обнаружение ошибок в компоненте,
3. Предотвращение пропуска ошибок на более высокие уровни тестирования.

Интеграционное тестирование необходимо для того, чтобы тестировать взаимосвязь между чем-либо.

В общем случае различают *два вида интеграционного тестирования*:

—**Компонентное** интеграционное тестирование. Как видно из названия, оно необходимо для того, чтобы протестировать работу модулей в связке друг с другом.



— ***Системное*** интеграционное тестирование.

Предположим, мы разрабатываем игру для смартфона. Сама игра является системой, которую необходимо протестировать. Кроме этого, есть еще сервисы, которые взаимодействуют с игрой и такое взаимодействие тоже должно быть проверено. Например, игра может поддерживать социальную сеть Facebook, чтобы можно было играть с друзьями. В этом случае, необходимо протестировать корректно ли работает вход в социальную игру через нашу игру, можем ли мы видеть список друзей и т. д.

Для этого уровня тестирования также характерно несколько целей:  
1. Проверка интерфейсов на соответствие требованиям.  
2. Обнаружение ошибок в интерфейсах.  
3. Предотвращение пропуска ошибок на более высокие уровни тестирования.

С помощью интеграционного тестирования мы снижаем риски и укрепляем свою уверенность в качестве продукта.

# **14 Системное (system), приемочное (acceptance) тестирование.**

Системное тестирование — это тестирование еще более высокого уровня.

Задача уже состоит в том, чтобы убедиться в корректности работы в целом всей системы. Программа в этом случае должна быть максимально приближена к конечному результату. А наше внимание должно быть сосредоточено на общем поведении системы с точки зрения конечных пользователей

Для этого уровня тестирования также характерно несколько целей:  
1. Проверка системы на соответствие требованиям.  
2. Обнаружение ошибок в системе.  
3. Предотвращение пропуска ошибок на более высокие уровни тестирования

Приемочное тестирование — это высший уровень тестирования, направленный на проверку готовности программы к использованию. Оно проводится в конце цикла разработки и должно подтвердить, что программа соответствует требованиям и функционалу, ожидаемым пользователями.

Основные цели приемочного тестирования:

1. Подтвердить завершенность программы.

2. Проверить соответствие установленному ТЗ.

Согласно ISTQB, существуют несколько форм приемочного тестирования:

1. Пользовательское приемочное тестирование — проверка программы с точки зрения конечного пользователя.

2. Эксплуатационное приемочное тестирование — проверка с позиции технической поддержки, включая аспекты установки, обновления и восстановления.

3. Контрактное приемочное тестирование — проверка по условиям контракта с заказчиком.

4. Альфа- и Бета-тестирование — получение обратной связи от пользователей; альфа-тестирование проводится внутри компании, а бета-тестирование — среди ограниченного круга пользователей.

# **15 Техники тест дизайна. Разбиение на классы эквивалентности и тестирование граничных значений.**

Разбиение на классы эквивалентности — это метод, при котором входные данные разделяются на несколько классов эквивалентных значений, из которых можно выбрать одно значение для тестирования. Все значения внутри одного класса считаются эквивалентными, и тестирование одного из них будет достаточно для проверки всей группы. Пример: если входное значение должно быть числом от 1 до 100, то классы могут быть:

* Класс 1: значения от 1 до 100.
* Класс 2: значения меньше 1.
* Класс 3: значения больше 100.

Тестирование граничных значений — это метод, при котором тестируются значения на границах классов эквивалентности, поскольку ошибки часто происходят именно в этих местах. Пример: для диапазона от 1 до 100 тестируются следующие значения:

* Минимум: 1.
* Максимум: 100.
* За границами: 0 и 101.

# **16 Понятие дефекта. Основные определения и классификация дефектов**

Дефект — это отклонение от ожидаемого или требуемого поведения программы. Это ошибка в программе, которая влияет на её работу, нарушая функциональность, производительность или безопасность.

Классификация дефектов:

* Функциональные дефекты — несоответствие функциональным требованиям системы.
* Нефункциональные дефекты — проблемы с производительностью, удобством использования, безопасностью, совместимостью.
* Серьезные дефекты — такие, которые блокируют работу приложения или могут вызвать его сбой.
* Технические дефекты — ошибки в коде, такие как синтаксические ошибки, неправильные алгоритмы и т.д.

# **17 Виды ошибок. Методы отладки.**

Виды ошибок:

* Синтаксические ошибки — ошибки, связанные с неверным использованием языка программирования (например, пропущенная скобка).
* Логические ошибки — ошибки, которые не приводят к сбоям, но нарушают правильность выполнения программы (например, неправильное вычисление).
* Ошибки выполнения (Runtime errors) — ошибки, возникающие в процессе работы программы (например, деление на ноль).
* Ошибки компиляции — ошибки, которые препятствуют компиляции программы (например, неопределенные переменные).

Методы отладки:

* Использование отладчика — позволяет пошагово выполнять программу, исследовать значения переменных и потоки выполнения.
* Логирование — выводит информацию о выполнении программы на разных этапах (например, используя logging в Python).
* Ручное тестирование — ручная проверка программы на разных этапах её работы.
* Юнит-тесты — автоматические тесты, которые помогают выявить ошибки в конкретных частях программы.

# **18 Модульное тестирование.**

Модульное тестирование — это метод тестирования, при котором тестируются отдельные компоненты программы (модули) в изоляции от других частей системы. Модульные тесты проверяют, что каждый отдельный модуль работает корректно.

Основные характеристики:

* Тестируются небольшие единицы кода (функции, методы).
* Обычно выполняется в рамках разработки (Test-Driven Development, TDD).
* Позволяет быстро находить ошибки на ранних стадиях разработки.
* В Python для модульного тестирования обычно используется встроенный модуль unittest.

Пример:

import unittest

def add(a, b):

return a + b

class TestMathOperations(unittest.TestCase):

def test\_add(self):

self.assertEqual(add(1, 2), 3)

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

# **19 Интеграционное тестирование.**

Интеграционное тестирование — это процесс тестирования, при котором проверяется взаимодействие между несколькими модулями или компонентами системы. Цель — выявить проблемы на стыке компонентов, которые могут не проявляться при тестировании отдельных частей системы.

В отличие от модульного тестирования, где проверяются отдельные функции или классы, интеграционное тестирование фокусируется на взаимодействии и интеграции этих компонентов.

Методы:

* Инкрементальное интеграционное тестирование — тестирование системы по частям, начиная с малого набора модулей.
* Большим банком (Big Bang) — когда все модули интегрируются сразу, и система тестируется целиком.
* Тестирование на основе контрактов — проверка, что модули правильно взаимодействуют в соответствии с заранее определёнными интерфейсами.

Пример: проверка взаимодействия между модулем авторизации и модулем базы данных для корректной работы с пользователями.

# **20 Средства разработки технической документации.**

При создании технической документации на ПО можно использовать программы и сервисы, которые упрощают процесс:

● Doxygen. Бесплатный сервис, который анализирует исходный код и комментарии к нему, автоматически извлекает информацию о классах, функциях, переменных и других элементах программы. В основном используется для C# и C++, но также поддерживает другие языки, в том числе Python, Java, IDL и PHP. Doxygen позволяет создавать документацию в различных форматах, например, HTML или RTF, а также конвертировать данные в PDF.

● GitBook. Платформа, которая позволяет нескольким авторам работать над одной документацией одновременно. Включает настраиваемые темы и шаблоны, функцию поиска и имеет простой интерфейс. Удобна для создания всех видов технической документации ПО. Стоимость: от $6,7 в месяц, есть бесплатная версия. GitBook поддерживает несколько форматов итоговой документации, включая HTML, PDF и форматы электронных книг EPUB и MOBI.

Средства разработки технической документации — это инструменты, предназначенные для создания, редактирования, хранения и публикации документации, связанной с разработкой и эксплуатацией программного обеспечения. Основные виды средств включают:

* Текстовые процессоры: Microsoft Word, Google Docs — используют для создания документации в текстовом формате, включая спецификации, инструкции и отчеты.
* Системы управления документацией (DMS): Confluence, SharePoint — обеспечивают централизованное хранение, совместную работу и управление версиями документов.
* Системы для создания и поддержания документации в разметке: Markdown, reStructuredText, LaTeX — часто используются для разработки технической документации в виде статичных страниц, обеспечивая легкость форматирования и поддержание документации в репозиториях.
* Средства для создания схем и диаграмм: Microsoft Visio, Lucidchart, Draw.io — помогают визуализировать архитектуру, процессы и схемы в документации.

# **21 Технологии разработки документов.**

Технологии разработки документов включают методы, подходы и инструменты, которые позволяют создавать документацию эффективным и стандартизированным образом. Среди них можно выделить:

* Автоматизация документооборота: Использование шаблонов и автоматических генераторов документации (например, Javadoc для документации кода).
* Системы контроля версий: Git, Subversion — применяются для контроля и отслеживания изменений в документации.
* Системы генерации документации: Doxygen, Sphinx, Jekyll — позволяют генерировать документацию на основе комментариев в коде или текстовых файлов.
* Методологии документирования: Например, использование Markdown для документов, предназначенных для публикации в GitHub, или разметки LaTeX для научных и технических документов.

Основной принцип — четкая структура, использование стандартов и поддержка актуальности документации.

# **22 Документирование программного обеспечения в соответствии с Единой системой программной документации.**

Единая система программной документации (ЕСПД) — это набор стандартов, методов и правил, предназначенных для унификации и систематизации документации на программное обеспечение. В рамках ЕСПД важно:

* Типология документов: включает описание функциональных и нефункциональных требований, проектной документации, инструкций по эксплуатации и тестированиям.
* Стандарты и нормативы: документы должны соответствовать стандартам ГОСТ, ISO, а также внутренним корпоративным стандартам, которые регулируют содержание, формат и структуру документации.
* Типы документации в ЕСПД:
  + Технические задания (ТЗ)
  + Технические отчеты
  + Инструкции по эксплуатации
  + Документация для тестирования ПО

Документирование должно быть логически структурировано и обеспечивать полное описание всех стадий жизненного цикла программного обеспечения.

# **23 Автоматизация разработки технической документации.**

Автоматизация разработки технической документации включает использование различных инструментов и подходов для ускорения и улучшения качества создания документов. К таким средствам относятся:

* Генераторы документации: например, Sphinx для Python или Doxygen для C++, которые автоматически генерируют документацию по комментариям в коде.
* Интеграция с системами контроля версий: для обеспечения актуальности документации при изменении кода (например, использование Git для синхронизации документации с кодом).
* Шаблоны и стандартные форматы: Использование заранее подготовленных шаблонов, которые помогут поддерживать единый стиль и структуру документации.
* API документация: Автоматическое создание документации по API через такие инструменты, как Swagger или Postman.

Автоматизация позволяет сократить время на поддержание документации и снизить риск ошибок из-за человеческого фактора.

# **24 Инструментальные средства поддержки тестирования.**

Инструментальные средства поддержки тестирования помогают улучшить процесс тестирования программного обеспечения, ускорить его выполнение и повысить качество. В Python существует множество таких инструментов:

* Фреймворки для юнит-тестирования:
  + unittest: Встроенный фреймворк в Python, предназначенный для написания и запуска модульных тестов.
  + pytest: Популярный фреймворк с расширенными возможностями, такими как фикстуры, параметры тестов, ассерты и другие.
* Тестирование производительности:
  + Locust: Используется для тестирования нагрузки на систему.
  + Apache JMeter: Инструмент для тестирования производительности веб-приложений.
* Тестирование безопасности:
  + Bandit: Инструмент для статического анализа безопасности кода Python.
* Инструменты для автоматизации тестирования UI:
  + Selenium: Для автоматизации взаимодействия с веб-интерфейсами.
  + PyAutoGUI: Для тестирования приложений с графическим интерфейсом пользователя.
* CI/CD системы: Jenkins, GitLab CI, CircleCI — интеграция тестов в процессы непрерывной интеграции и доставки (CI/CD) помогает автоматизировать запуск тестов при каждом изменении кода.

# **25 Артефакты разработки ПО, относящиеся к тестированию. Тест-кейсы (test cases).**

Тест-кейсы — это основные артефакты в процессе тестирования программного обеспечения, которые описывают конкретные шаги и условия, при которых проводится тестирование. Тест-кейсы предназначены для проверки функциональности и других характеристик системы. Тест-кейсы обычно включают следующие элементы:

* Идентификатор теста: Уникальный номер или имя теста для его отслеживания.
* Описание: Краткое описание того, что проверяет данный тест-кейс.
* Предусловия: Условия, которые должны быть выполнены до выполнения теста (например, начальные данные, настройки системы).
* Шаги теста: Последовательность действий, которые необходимо выполнить в рамках тестирования.
* Ожидаемый результат: Описание того, что должно произойти после выполнения теста (например, изменение данных, появление сообщения).
* Фактический результат: Результат выполнения теста, который сравнивается с ожидаемым.
* Статус теста: Указывает, прошел ли тест (например, "успех" или "неудача").

Пример тест-кейса:

* Идентификатор: TC\_001
* Описание: Проверка правильности ввода логина.
* Предусловия: Пользователь находится на странице авторизации.
* Шаги:
  1. Ввести в поле логина "user123".
  2. Нажать на кнопку "Войти".
* Ожидаемый результат: Ожидается, что пользователь будет перенаправлен на страницу профиля.
* Фактический результат: Успешное перенаправление на страницу профиля.
* Статус: Прошел.

Тест-кейсы являются основой для систематического и повторяемого тестирования. Их наличие помогает тестировщикам удостовериться в том, что все важные аспекты функциональности приложения протестированы.

# **26 Артефакты разработки ПО, относящиеся к тестированию. План тестирования (test plan).**

План тестирования (test plan) — это документ, который описывает общую стратегию тестирования, задачи, подходы, критерии и ресурсы, необходимые для успешного выполнения тестирования программного обеспечения. План тестирования обычно включает следующие компоненты:

* Цели тестирования: Определение того, какие аспекты системы должны быть проверены (например, функциональность, производительность, безопасность).
* Объем тестирования: Описание того, что будет протестировано, а что не будет (например, только основные функции, а не все возможные варианты использования).
* Типы тестирования: Указание, какие виды тестов будут проводиться (модульное тестирование, интеграционное, системное, приемочное и др.).
* Ресурсы: Описание необходимых ресурсов, таких как оборудование, программное обеспечение, люди (кто будет выполнять тесты).
* Ожидаемые результаты: Краткое описание того, как должны выглядеть ожидаемые результаты для различных тестов.
* График тестирования: Планирование сроков, когда будут проводиться различные этапы тестирования.
* Критерии завершения: Условия, при которых тестирование может быть завершено (например, когда все критические баги исправлены и пройдены тесты).

Пример плана тестирования:

* Цели тестирования: Проверить функциональность поиска по базе данных, убедиться в правильности отображения результатов.
* Типы тестирования: Модульное тестирование для каждого компонента поиска, системное тестирование всей поисковой системы.
* Ресурсы: Тестировщики, тестовый сервер, инструменты для автоматизации тестирования (например, Selenium).
* Ожидаемые результаты: Поиск по ключевым словам должен возвращать релевантные результаты, порядок сортировки должен соответствовать настройкам.
* График: Тестирование начнется с 1 января и закончится 15 января.
* Критерии завершения: Все критические баги должны быть исправлены, количество найденных ошибок должно быть минимальным.

План тестирования помогает систематизировать процесс тестирования, обеспечивает его согласованность и согласование между командой разработки и тестирования.

# **27 Обеспечение качества (Quality Assurance).**

**Обеспечение качества (Quality Assurance, QA)** — это процесс, направленный на создание и поддержку процессов разработки и тестирования программного обеспечения, которые гарантируют, что продукт будет соответствовать установленным требованиям качества. QA включает в себя не только тестирование, но и несколько других аспектов:

* **Процессное управление**: Разработка и внедрение процессов, которые обеспечат высокое качество на всех стадиях жизненного цикла продукта, начиная от планирования и разработки до релиза.
* **Планирование качества**: Определение стандартов качества, которые должны быть достигнуты, и методов для их обеспечения.
* **Предотвращение дефектов**: QA направлена на предотвращение возникновения дефектов, а не только на их выявление. Это включает в себя использование практик код-ревью, статического анализа кода, автоматизированного тестирования.
* **Метрики качества**: Использование различных метрик для оценки качества продукта и процесса, таких как покрытие тестами, количество багов, время на исправление ошибок и т.д.
* **Система управления рисками**: Идентификация и управление рисками, которые могут повлиять на качество продукта. Это может включать в себя анализ рисков, проведение стресс-тестов и оценку уязвимостей.

Таким образом, QA является проактивным процессом, который охватывает все этапы разработки ПО, с целью минимизировать дефекты и улучшить качество конечного продукта. Важно отметить, что QA не ограничивается тестированием, а охватывает все аспекты разработки, включая процессы и стандарты, которые предотвращают появление дефектов и обеспечивают высокое качество продукта.