ТЕСТИРОВАНИЕ ПО И ТЕСТОВАЯ ДОКУМЕНТАЦИЯ

ВВЕДЕНИЕ

Тестирование программного обеспечения — процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта.

Ошибка — ошибка, допущенная разработчиком.

Дефект/баг — несоответствие кода требованиям.

Неисправность — некорректное поведение программы в процессе работы.

Верификация – «соответствие требованиям» предполагает, что требования должны быть настолько четко определены, что не могут быть поняты и интерпретированы некорректно. Любые несоответствия должны рассматриваться как дефекты – отсутствие качества.

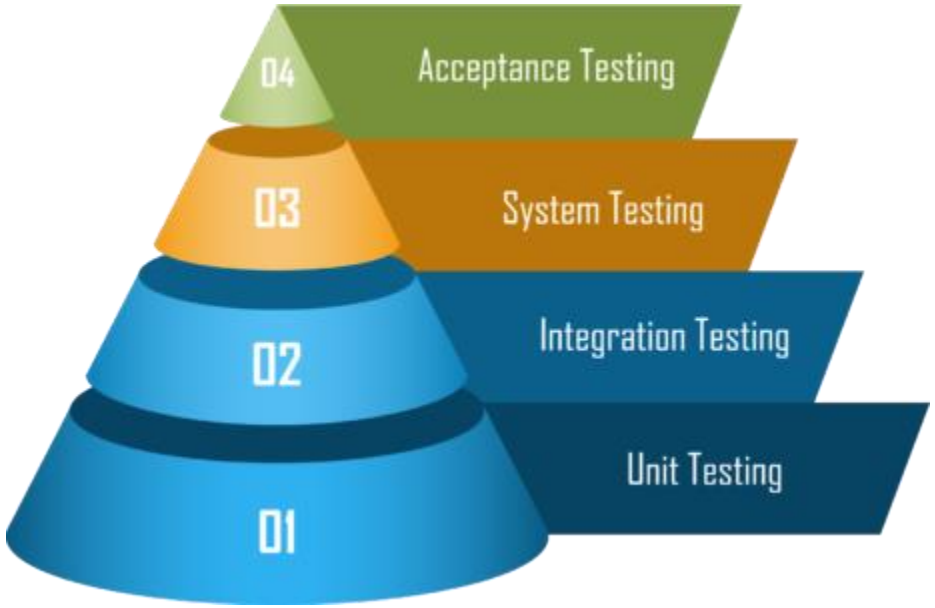
Валидация – «пригодность к использованию» принимает во внимание требования и ожидания конечных пользователей продукта, т.е. продукт или предоставляемый сервис будет удобным для их нужд.

УРОВНИ ТЕСТИРОВАНИЯ

Пирамида тестирования, также часто говорят **уровни тестирования**, это группировка тестов по уровню детализации и их назначению.

Пирамиду разбивают на 4 уровня (снизу вверх), например, по ISTQB:

- модульное тестирование (юнит);
- интеграционное тестирование;
- системное тестирования;
- приемочное тестирование.



МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Модульное тестирование — это базовый уровень тестирования, который проверяет отдельные компоненты программы (функции, классы, модули). Оно используется на протяжении всего жизненного цикла разработки и тестирует каждый модуль изолированно. Этот вид тестирования особенно важен на всех этапах, так как помогает локализовать проблемы и быстро реагировать на изменения.

Цель: проверить каждый отдельный компонент системы (модуль) в изоляции от других.

Что тестируется: отдельные функции, методы или классы, которые представляют собой "модули" в коде. Эти модули тестируются отдельно от остальной системы.

Преимущества:

- Раннее обнаружение ошибок.
- Локализация проблем для более легкой отладки.
- Улучшение качества кода, так как разработчик фокусируется на каждом модуле по отдельности.

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Пример:
Есть функция на Python, которая вычисляет факториал числа

```python
# Пример функции на Python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)


# Модульный тест для этой функции
import unittest

class TestFactorial(unittest.TestCase):
    def test_factorial(self):
        self.assertEqual(factorial(5), 120)
        self.assertEqual(factorial(0), 1)

if __name__ == '__main__':
    unittest.main()

```

Инструменты для модульного тестирования:

- Jest, Mocha для JavaScript.
- JUnit для Java.
- pytest, unittest для Python.
- NUnit для .NET.

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ

Интеграционное тестирование проверяет взаимодействие между отдельными модулями, чтобы убедиться, что они работают корректно вместе. Даже если отдельные модули работают правильно в изоляции, их интеграция может выявить ошибки из-за неправильного взаимодействия.

Цель: проверить, как модули работают вместе, когда они интегрированы (объединены) в одно целое.

Что тестируется: взаимодействие между модулями системы, которые уже прошли модульное тестирование. Задача — убедиться, что модули корректно "общаются" друг с другом. *Пример использования:* тестирование взаимодействия модуля авторизации и модуля управления сессиями в веб-приложении.
Методы интеграционного тестирования:

- Нисходящее тестирование — начинается с высокоуровневых модулей, постепенно интегрируя нижестоящие.
- Восходящее тестирование — начинается с низкоуровневых модулей, и по мере их интеграции проверяются взаимодействия с вышестоящими.
- Большой взрыв — все модули интегрируются и тестируются одновременно, что может быть рискованным, если модули не тестировались по отдельности.

Преимущества:

- Обнаружение дефектов интерфейсов между модулями.
- Проверка корректности передачи данных между компонентами.
- Выявление проблем, связанных с несовместимостью модулей.

СИСТЕМНОЕ ТЕСТИРОВАНИЕ

Системное тестирование проверяет всю систему как единое целое. Этот этап важен для того, чтобы убедиться, что система соответствует всем функциональным и нефункциональным требованиям.

Цель: проверить всю систему в целом, убедиться, что она выполняет все заявленные требования и работает корректно в реальных условиях.

Типы системного тестирования:
- Функциональное тестирование — проверка, что система выполняет свои функциональные обязанности.
- Нагрузочное тестирование — проверка системы на устойчивость при высоких нагрузках.
- Тестирование безопасности — оценка защищенности системы от атак и уязвимостей.
- Тестирование удобства использования (Usability testing) — проверка удобства работы пользователей с системой.

Преимущества:
- Полное тестирование системы на соответствие заявленным требованиям.
- Обнаружение дефектов, связанных с целостностью системы.
- Проверка системы в условиях, приближенных к реальным.

Пример использования: проверка функционирования банковской системы после интеграции всех модулей (клиентский интерфейс, обработка транзакций, управление счетами).

ПРИЁМОЧНОЕ ТЕСТИРОВАНИЕ

Приемочное тестирование проводится для проверки того, что система соответствует требованиям заказчика и готова к релизу. Обычно его проводят конечные пользователи или заказчики.

Цель: убедиться, что продукт удовлетворяет ожиданиям заказчика и готов к использованию.

Этапы тестирования с привлечением пользователей:
- Альфа-тестирование
- Бета-тестирование

Преимущества:
- Финальная проверка продукта перед поставкой.
- Снижение рисков возврата или недовольства клиента.

Пример использования: тестирование веб-приложения конечным пользователем, чтобы убедиться, что оно выполняет все заявленные функции и удобно в использовании.

АЛЬФА И БЕТА ТЕСТИРОВАНИЕ

Альфа-тестирование проводится внутри компании, как правило, силами разработчиков или сотрудников, не занятых непосредственно в проекте. Этот вид тестирования проводится на стадии разработки.

Бета-тестирование проводится внешними пользователями после альфа-тестирования, когда продукт уже считается достаточно стабильным для публичного тестирования.

Цель альфа и бета тестирования: выявление реальных ошибок, возникающих при использовании системы.

Преимущества:
- Обратная связь от пользователей для улучшения качества продукта.
- Проверка продукта в реальных условиях эксплуатации.

Пример использования: публичный релиз бета-версии мобильного приложения для сбора отзывов и улучшений перед официальным запуском.

ТИПЫ ТЕСТИРОВАНИЯ

Типы тестирования помогают структурировать процесс проверки программного обеспечения, охватывая разные уровни и цели, чтобы обеспечить качество, надежность и соответствие продукта требованиям. Каждый тип ориентирован на конкретные аспекты или этапы разработки и эксплуатации системы.

По типу проверки:
- Функциональное тестирование
- Нефункциональное тестирование

По знанию кода:
- Тестирование «черного ящика»
- Тестирование «белого ящика»
- Тестирование «серого ящика»

По целям и задачам:
- Дымовое тестирование
- Санитарное тестирование
- Регрессионное тестирование
- Альфа-тестирование
- Бета-тестирование

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Функциональное тестирование проверяет, что система или ее компоненты выполняют свои функции согласно спецификации.

Цель: подтвердить, что система работает в соответствии с функциональными требованиями.

Преимущества:
- Проверка работоспособности основных функций системы.
- Возможность использования как ручного, так и автоматизированного тестирования.

Пример использования: проверка, что кнопка "Добавить в корзину" в интернет-магазине правильно добавляет товар в корзину.

НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Нефункциональное тестирование направлено на проверку аспектов системы, не связанных с её функциональностью, таких как производительность, безопасность и удобство использования.

Типы нефункционального тестирования:
- Нагрузочное тестирование — оценка производительности системы при нормальных и повышенных нагрузках.
- Тестирование безопасности — проверка защиты данных и устойчивости системы к атакам.
- Тестирование удобства использования (Usability Testing) — проверка удобства взаимодействия пользователей с интерфейсом системы.
- Тестирование производительности (Performance Testing) — анализ быстродействия системы в различных условиях.

Преимущества:
- Улучшение показателей производительности системы.
- Снижение уязвимостей безопасности.
- Оптимизация пользовательского опыта.

Пример использования: проверка, как долго работает веб-приложение под высокой нагрузкой пользователей.

ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ

Тестирование безопасности — это процесс оценки программного обеспечения на устойчивость к угрозам, уязвимостям и несанкционированному доступу.

Цель: найти уязвимости, которые могут позволить злоумышленникам получить доступ к данным или системным функциям, нарушить работу системы или скомпрометировать конфиденциальность пользователей.

Что тестируется: безопасность системы, включая защиту данных, аутентификацию, авторизацию, управление сессиями, криптографию и т.д.

Когда используется: на поздних этапах разработки, перед релизом и периодически после релизов для обеспечения непрерывной защиты.

Пример: проверка на уязвимость к SQL-инъекциям. Например, при попытке обмануть систему входа через вредоносный ввод данных.

ТЕСТИРОВАНИЕ ЧЕРНОГО И БЕЛОГО ЯЩИКА

Тестирование черного ящика (Black Box Testing): проверка системы без знания внутренней структуры кода, фокусируется на вводе и выводе данных.

Пример: тестировщик проверяет пользовательский интерфейс, не зная, как устроены внутренние процессы.

Тестирование белого ящика (White Box Testing): тестирование с доступом к исходному коду системы, что позволяет проверять внутренние механизмы программы.

Пример: тестировщик знает алгоритмы и логику программы, проверяет внутренние функции и условия.

Тестирование серого ящика (Gray Box Testing) — это метод тестирования, при котором у тестировщика есть частичное знание внутренней структуры или логики приложения, но основная проверка проводится с точки зрения пользователя (черного ящика).

Пример: тестировщик знает, как устроена база данных и основные алгоритмы обработки данных, но тестирует функциональность через пользовательский интерфейс.

РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ

Регрессионное тестирование проверяет, что добавление нового кода или функционала не нарушило существующую работу системы. Это один из самых важных этапов тестирования, поскольку любое изменение может негативно повлиять на стабильность работы системы.

Цель: убедиться, что изменения в коде (новый функционал или исправления) не сломали уже существующую функциональность.

Что тестируется: весь продукт или его части, которые уже были протестированы ранее. Регрессионное тестирование направлено на проверку того, что система по-прежнему работает правильно после внесения изменений.

Когда используется: после каждого внесения изменений в систему, будь то исправление ошибки или добавление нового функционала.

Пример: после изменения функции оплаты в интернет-магазине, нужно проверить, что она не сломала существующие функции добавления товаров в корзину.

ДЫМОВОЕ ТЕСТИРОВАНИЕ

Дымовое тестирование — это кратковременная проверка, которая проводится для оценки стабильности основных функций системы после нового билда или деплоя.

Цель: убедиться, что основные функции системы не нарушены и продукт готов для более глубокого тестирования.

Преимущества:

- Быстрая проверка работоспособности системы.
- Снижение времени на анализ ошибок после деплоя.

Пример использования: после внедрения нового релиза проверяются ключевые функции: авторизация, навигация, базовые транзакции.

ТЕСТИРОВАНИЕ СОВМЕСТИМОСТИ, ВОССТАНОВЛЕНИЯ ПОСЛЕ СБОЕВ И КОНФИГУРАЦИИ

Тестирование совместимости проверяет, насколько хорошо система работает на различных устройствах, операционных системах, браузерах и их версиях.

Цель: убедиться, что продукт корректно работает на всех заявленных платформах.

Пример использования: проверка, что веб-приложение одинаково работает в браузерах Chrome, Firefox и Safari.

Тестирование восстановления проверяет способность системы восстанавливаться после отказов или сбоев.

Цель: убедиться, что система может вернуться к нормальной работе после критических ошибок.

Пример использования: тестирование, как быстро восстанавливается база данных после сбоя питания.

Тестирование конфигурации проверяет, как система работает с различными конфигурациями оборудования и программного обеспечения (например, с разными версиями ОС, различными драйверами).

Цель: убедиться, что система корректно работает в различных средах.

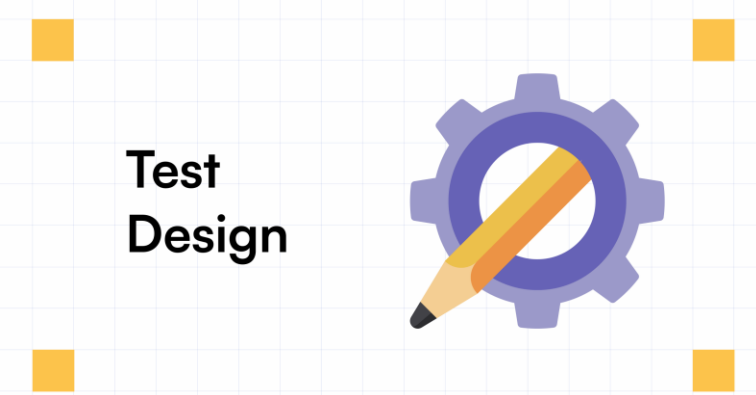
Пример использования: проверка работы приложения на устройствах с разным объемом оперативной памяти.

ТЕХНИКИ ТЕСТ-ДИЗАЙНА

Техники тестирования — это формальные методы и подходы, которые помогают выбрать оптимальный набор тестовых данных и сценариев из множества возможных вариантов. Их цель — повысить эффективность тестирования, сократить количество тестов при максимальном покрытии функций и обнаружении багов. Эти техники позволяют системно и рационально строить тестовые случаи, ориентируясь на спецификации и требования к ПО.

Некоторые техники тестирования:

- Классы эквивалентности
- Анализ граничных значений
- Комбинаторное тестирование
- Табличное тестирование (Decision Table Testing)
- Тестирование переходов состояний (State Transition Testing)
- Исследовательское тестирование (Exploratory Testing)



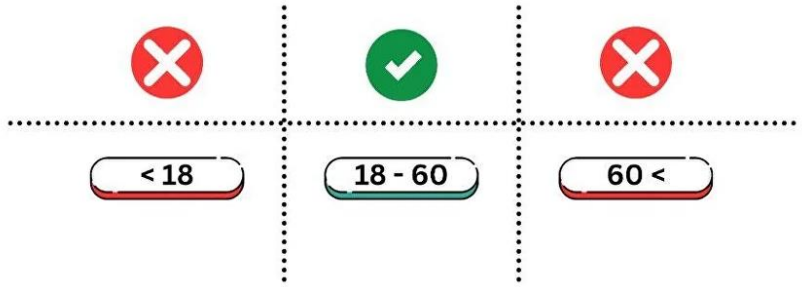
КЛАССЫ ЭКВИВАЛЕНТНОСТИ

Класс эквивалентности — набор данных, обрабатываемых одинаковым образом и приводящих к одинаковому результату, т.е. это множества.

Если от выполнения двух тестов ожидается один и тот же результат, то они считаются эквивалентными. Группа тестов представляет собой класс эквивалентности, если выполняются следующие условия:

- Все тесты предназначены для выявления одной и той же ошибки
- Если один из тестов выявит ошибку, остальные, скорее всего, тоже это сделают
- Если один из тестов не выявит ошибки, остальные, скорее всего, тоже этого не сделают

Пример: класс валидных номеров телефона, класс пустых строк, класс номеров с неправильным форматом.



АНАЛИЗ ГРАНИЧНЫХ ЗНАЧЕНИЙ

Граничное условие — значение, находящееся на границе классов эквивалентности. Тестирование значений на границах и около них, так как ошибки чаще всего возникают именно там.

Весьма перспективной областью для поиска ошибок являются границы классов эквивалентности функции. Как правило, анализ, который ведет к определению классов эквивалентности, определяет и эти границы. Включение нескольких пограничных значений в тестовые случаи для данной функции поможет проверить удовлетворение ожиданий (требований) пользователя.

Пример: если поле принимает число от 1 до 100, протестировать 0, 1, 2, 99, 100, 101.

КОМБИНАТОРНОЕ ТЕСТИРОВАНИЕ

Комбинаторное тестирование — способ выбрать подходящий набор комбинаций тестовых данных для достижения установленного уровня тестового покрытия в случае, когда проверка всех возможных наборов значений тестовых данных невозможна за имеющееся время.

Некоторые комбинаторные техники:

- Тестирование всех комбинаций — тестирование всех возможных комбинаций всех значений всех тестовых данных (например, всех параметров функции).
- Попарное тестирование - основан на следующей идее: подавляющее большинство багов выявляется тестом, проверяющим либо один параметр, либо сочетание двух. Ошибки, причиной которых явились комбинации трех и более параметров, как правило, значительно менее критичны.

Пример: проверка всех сочетаний оплатой картой / наличными и доставки курьером / самовывозом.

ТАБЛИЦА ПРИНЯТИЯ РЕШЕНИЙ

Таблица принятия решений — техника тестирования (по методу чёрного ящика), в которой тест-кейсы разрабатываются на основе т.н. таблицы принятия решений, в которой отражены входные данные (и их комбинации) и воздействия на приложение, а также соответствующие им выходные данные и реакции приложения.

Помогает обеспечить проверку всех комбинаций условий и соответствующих реакций системы.

Пример: таблица с комбинациями условий по валидации формы и соответствующими сообщениями об ошибках.

	Rule 1	Rule 2	Rule 3	Rule 4
Conditions				
Math Score: 40-70	Y	N	Y	N
Average Score: 51-99	Y	Y	N	N
Action				
10% discount	N	Y	Y	N
20% discount	Y	N	N	N

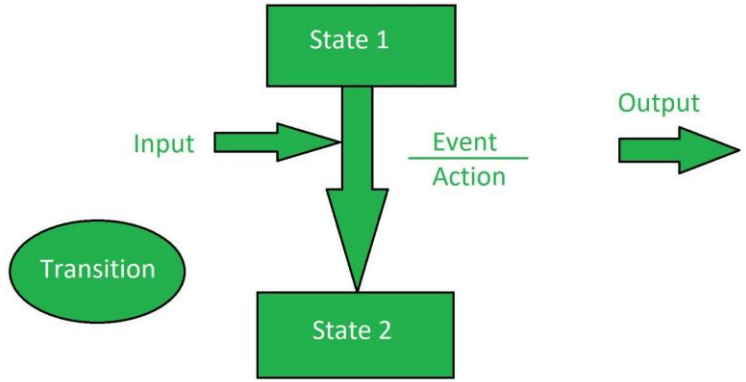
ТЕСТИРОВАНИЕ ПЕРЕХОДОВ СОСТОЯНИЙ

Тестирование переходов состояний — техника тестирования, в которой тест-кейсы разрабатываются для проверки переходов приложения из одного состояния в другое. Состояния могут быть описаны диаграммой состояний или таблицей состояний. Иногда эту технику тестирования также называют «тестированием по принципу конечного автомата». Важным преимуществом этой техники является возможность применения в ней теории конечных автоматов (которая хорошо формализована), а также возможность использования автоматизации для генерации комбинаций входных данных.

Диаграмма перехода состояний представляет собой одну специфическую сущность (например, процесс резервирования).

Применяется для систем, поведение которых зависит от внутреннего состояния и внешних событий. Проверяются переходы из одного состояния в другое, включая корректные и некорректные сценарии.

Пример: изменение статуса заказа — новый, в обработке, отправлен, отменён.



ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ

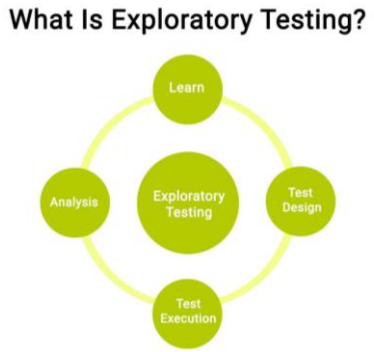
Исследовательское тестирование — это подход к тестированию, при котором тестировщик активно исследует приложение, создавая тест-кейсы и выполняя их одновременно. Это более гибкий метод, который фокусируется на выявлении дефектов через изучение системы без использования заранее подготовленных сценариев.

Преимущества

- Быстрое обнаружение неожиданных дефектов.
- Гибкость в подходе и адаптация к поведению системы.
- Экономия времени на подготовку тест-кейсов.
- Выявление проблем, связанных с пользовательским опытом (UX).

Ограничения

- Сложно повторить найденные дефекты, если шаги не задокументированы.
- Не всегда охватывает все области системы из-за отсутствия формального плана.
- Зависит от опыта и знаний тестировщика.



МЕТОДЫ ОБЕСПЕЧЕНИЯ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Обеспечение качества программного обеспечения (Quality Assurance, QA) – это процесс, направленный на поддержание высокого уровня качества при разработке ПО. Это не просто тестирование конечного продукта, а многоэтапный процесс, который начинается с проектирования системы и продолжается до самого завершения жизненного цикла ПО.

Цели обеспечения качества:

- Снижение количества дефектов: минимизация багов и уязвимостей.
- Повышение надежности: обеспечение устойчивой и предсказуемой работы ПО.
- Улучшение производительности: оптимизация системы для различных условий эксплуатации.
- Обеспечение безопасности: защита от угроз и уязвимостей.
- Ускорение выпуска новых версий: создание эффективных процессов разработки.

Существует множество методов и инструментов, которые применяются на разных этапах разработки, чтобы гарантировать, что продукт отвечает стандартам качества. *Три* наиболее популярных *подхода*:

- Ревью кода
- Статический анализ
- Автоматизированное тестирование

РЕВЬЮ КОДА

Ревью кода – это процесс, при котором другие разработчики проверяют код, написанный их коллегами, чтобы найти и устранить ошибки, улучшить архитектуру и повысить читаемость кода. Этот метод фокусируется не только на поиске багов, но и на улучшении стиля, логики и структуры кода.

Цели ревью кода:

- Обнаружение дефектов на ранних стадиях.
- Повышение уровня качества кода.
- Обучение менее опытных разработчиков.
- Совместная работа и обмен знаниями.

Как проводится ревью:

1. Отправка на ревью. Разработчик завершает работу над задачей и отправляет свой код на проверку через систему управления версиями (например, GitHub, GitLab или Bitbucket).
2. Рецензирование. Один или несколько коллег анализируют код. Они обращают внимание на правильность реализации, возможные ошибки, соответствие архитектурным решениям и стилевым рекомендациям.
3. Обратная связь. Ревьюеры оставляют комментарии по коду, указывают на ошибки, а также дают рекомендации по улучшению.
4. Исправления. Автор вносит необходимые изменения и отправляет код повторно.
5. Принятие изменений. После одобрения ревьюерами код сливается в основную ветку проекта.

РЕВЬЮ КОДА

Преимущества:
- Улучшение читаемости и поддерживаемости кода.
- Обнаружение ошибок и багов до того, как они попадут в продакшн.
- Обучение и повышение квалификации разработчиков.

Недостатки:
- Может затянуть процесс разработки, если ревью проводится медленно.
- В некоторых случаях ревью может быть формальностью, без глубокой проверки.

Пример:
Разработчик добавил новую функцию для обработки данных, но сделал это неэффективно (использовал вложенные циклы вместо более оптимального алгоритма). Во время ревью его коллега заметит этот момент и предложит улучшить алгоритм, что не только улучшит производительность, но и снизит сложность поддержки кода.

СТАТИЧЕСКИЙ АНАЛИЗ

Статический анализ – это метод проверки кода без его выполнения. В отличие от динамического анализа (когда код запускается и проверяется во время выполнения), статический анализ анализирует исходный код программы на предмет синтаксических, логических и других ошибок. Этот анализ проводится автоматически с помощью специальных инструментов, таких как линтеры, анализаторы кода и компиляторы.

Цели статического анализа:

- Обнаружение потенциальных ошибок до выполнения кода.
- Проверка соответствия кода стандартам и стилям.
- Поиск уязвимостей безопасности.
- Анализ производительности и возможных проблем с памятью.

Как проводится статический анализ:

1. Запуск инструмента статического анализа. Разработчик или система непрерывной интеграции (CI) запускает инструмент (например, ESLint, Pylint, SonarQube, или Clang), который анализирует код.
2. Отчет об ошибках. Инструмент выводит отчет с найденными проблемами — от синтаксических ошибок до потенциальных уязвимостей безопасности.
3. Исправление. Разработчик исправляет обнаруженные проблемы на основе полученных отчетов.

СТАТИЧЕСКИЙ АНАЛИЗ

Преимущества:

- Мгновенное обнаружение ошибок, которые трудно заметить вручную.
- Проверка на наличие уязвимостей, которые могут быть использованы злоумышленниками.
- Интеграция в процесс CI позволяет быстро проверять код на соответствие стандартам.

Недостатки:

- Не обнаруживает логические ошибки, которые проявляются только при выполнении кода.
- Может создавать "ложные срабатывания", когда инструмент отмечает код как проблемный, хотя на самом деле ошибки нет.

Пример:

В Python-коде используется переменная, которая была объявлена, но не используется. Статический анализатор, такой как Pylint, отметит это как избыточный код, и разработчик сможет удалить ненужную переменную.

```
# Пример с избыточной переменной
def process_data(data):
    result = []  # Статический анализатор укажет, что эта переменная не используется
    for item in data:
        result.append(item * 2)
    return result

```

МЕТРИКИ КАЧЕСТВА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Метрики качества программного обеспечения (Quality Metrics) — это измеримые показатели, которые позволяют оценить различные аспекты системы, такие как её надёжность, производительность, безопасность и другие параметры. Оценка качества на основе метрик помогает разработчикам и менеджерам принимать решения о дальнейших улучшениях системы, выявлять потенциальные проблемы и обеспечивать соответствие продукта стандартам.

Важность метрик

- Объективная оценка. Предоставляют числовые данные для оценки качества продукта и процесса разработки.
- Принятие решений. Помогают принимать обоснованные решения на основе реальных данных, а не только интуиции.
- Управление рисками. Позволяют выявлять и устранять проблемы до того, как они усугубятся и повлияют на проект.
- Постановка целей. Помогают ставить измеримые цели и отслеживать прогресс в их достижении.

НАДЁЖНОСТЬ

Надёжность программного обеспечения — это способность системы выполнять свои функции в течение определённого времени без сбоев и ошибок. Это одна из ключевых метрик качества, так как пользователи ожидают стабильной работы ПО, особенно в критических приложениях, таких как медицинские системы или системы управления транспортом.

Основные аспекты надёжности:

- Безотказность — способность системы функционировать без критических ошибок и отказов. Безотказность измеряется как среднее время между сбоями (MTBF, Mean Time Between Failures).
- Восстановимость — способность системы восстанавливаться после сбоев и продолжать работу. Оценка происходит через метрику MTTR (Mean Time to Repair), которая показывает, сколько времени требуется на устранение проблем.
- Предсказуемость — система должна демонстрировать предсказуемое поведение при нормальных и аварийных условиях.

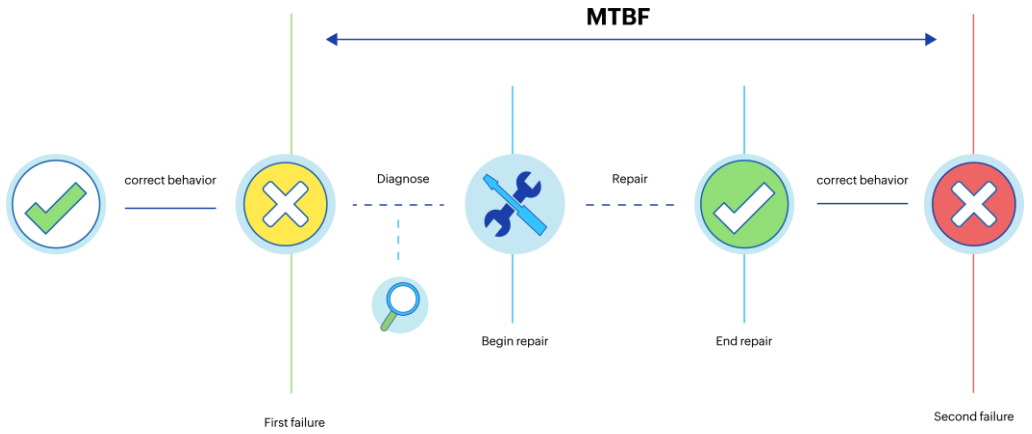
Метрики для оценки надёжности:

- MTBF (Mean Time Between Failures): среднее время между сбоями. Чем выше это значение, тем надёжнее система.
- MTTR (Mean Time to Repair): среднее время на восстановление после сбоя. Чем ниже это значение, тем быстрее система восстанавливается.
- Количество сбоев в единицу времени: показывает, как часто происходят ошибки.

НАДЁЖНОСТЬ

Пример:
Система мониторинга серверов испытывает редкие сбои, но после них быстро восстанавливается и продолжает свою работу. В этом случае метрика MTBF будет высокой (например, один сбой каждые 1000 часов), а MTTR — низкой (например, 5 минут на восстановление).

Улучшение надёжности:
- Внедрение системы резервирования данных и компонент.
- Использование средств мониторинга для раннего обнаружения проблем.
- Автоматическое восстановление при ошибках (например, перезапуск серверов).



ПРОИЗВОДИТЕЛЬНОСТЬ

Производительность программного обеспечения — это способность системы эффективно выполнять задачи за минимальное время с использованием минимальных ресурсов. Пользователи ожидают, что программа будет отвечать быстро, даже под нагрузкой, и не будет зависать при обработке больших объёмов данных.

Основные аспекты производительности:

- Время отклика — сколько времени требуется системе для выполнения запроса пользователя или задачи. Эта метрика особенно важна для веб-приложений и пользовательских интерфейсов.
- Пропускная способность — количество операций или запросов, которые система может обработать за единицу времени. Примером может быть веб-сайт, который способен обслуживать 1000 запросов в секунду.
- Использование ресурсов — сколько оперативной памяти, процессорного времени или дискового пространства использует система при выполнении задач

Метрики для оценки производительности:

- Время отклика: измеряется в миллисекундах или секундах, определяет, насколько быстро система реагирует на действия пользователей.
- Пропускная способность (Throughput): количество задач, которые система может выполнить за определённое время (например, количество обработанных запросов за секунду).
- Загрузка процессора/памяти: процент использования процессора и памяти системы, который позволяет оценить эффективность использования ресурсов.

ПРОИЗВОДИТЕЛЬНОСТЬ

Пример:
В интернет-магазине время отклика сервера при добавлении товара в корзину составляет 0.2 секунды, а сервер может обслуживать 5000 запросов в минуту. Если магазин начинает обслуживать больше пользователей, система должна сохранять эти показатели или масштабироваться для обработки увеличенной нагрузки.

```python
# Пример нагрузки API: тестирование времени отклика
import requests
import time

start_time = time.time()
response = requests.get("https://example.com/api/products")
end_time = time.time()

response_time = end_time - start_time
print(f"Response time: {response_time} seconds")

```

Улучшение производительности:
- Оптимизация алгоритмов и структур данных.
- Внедрение кэширующих решений, таких как Redis или Memcached.
- Масштабирование системы (добавление серверов или использование облачных ресурсов).
- Профилирование системы и устранение узких мест (например, долгие запросы к базе данных).

БЕЗОПАСНОСТЬ

Безопасность программного обеспечения — это способность системы защищаться от несанкционированного доступа, атак и утечек данных. В современном мире вопросы безопасности становятся особенно важными, так как кибератаки могут привести к огромным финансовым и репутационным потерям.

Основные аспекты безопасности:

- Конфиденциальность — защита данных от несанкционированного доступа. Система должна обеспечивать, чтобы только авторизованные пользователи могли получать доступ к конфиденциальной информации.
- Целостность данных — гарантия того, что данные не будут изменены или повреждены в ходе передачи или хранения.
- Доступность — система должна быть доступна пользователям даже в условиях атак (например, DDoS).
- Аутентификация и авторизация — проверка личности пользователя (аутентификация) и распределение прав доступа (авторизация).

Метрики для оценки безопасности:

- Количество уязвимостей: количество выявленных уязвимостей в системе, которые могут быть использованы для атак.
- Время на исправление уязвимости: время, которое требуется для исправления найденных проблем безопасности.
- Количество успешных атак: измерение количества атак, которые привели к реальной утечке данных или отказу системы.

БЕЗОПАСНОСТЬ

Пример:
Веб-приложение, такое как система интернет-банкинга, регулярно тестируется на наличие уязвимостей с помощью инструментов статического и динамического анализа, чтобы предотвратить SQL-инъекции и XSS-атаки.

```
# Пример уязвимости: SQL-инъекция
user_input = "' OR 1=1; --"
query = f"SELECT * FROM users WHERE username = '{user_input}'"
# Этот запрос вернёт всех пользователей, так как условие будет всегда истинно

# Защита от SQL-инъекции с использованием параметризованных запросов
query = "SELECT * FROM users WHERE username = %s"
cursor.execute(query, (user_input,))

```

Улучшение безопасности:
- Проведение регулярных аудитов безопасности и тестов на проникновение (penetration testing).
- Внедрение практик защиты от SQL-инъекций, XSS и CSRF-атак.
- Шифрование данных на этапе передачи и хранения.
- Ограничение прав доступа пользователей в соответствии с принципами минимизации привилегий.

ТЕСТОВАЯ ДОКУМЕНТАЦИЯ

Тестовая документация — это набор записей и инструкций, которые помогают проверять программу или приложение. В ней описывают, что именно нужно проверить, как это сделать и какой результат считать правильным. Благодаря такой документации команда тестировщиков не пропускает ошибки и делает продукт лучше.

Кто использует тестовую документацию:

- QA-инженеры пишут и выполняют тест-кейсы, фиксируют баги.
- Разработчики сверяются с тестами, чтобы убедиться, что всё работает верно.
- Менеджеры проектов отслеживают прогресс тестирования, анализируют риски, связанные с выпуском продукта, чтобы своевременно принимать решения и избегать проблем.
- Аналитики контролируют, чтобы тесты проверяли все нужные функции и задачи, которые были запланированы для продукта.
- Инженеры по автоматизации тестирования берут тест-кейсы за основу для написания автотестов.
- Аудиторы и заказчики изучают документацию, проверяя соответствие продукта установленным требованиям и стандартам качества перед выпуском или сертификацией.

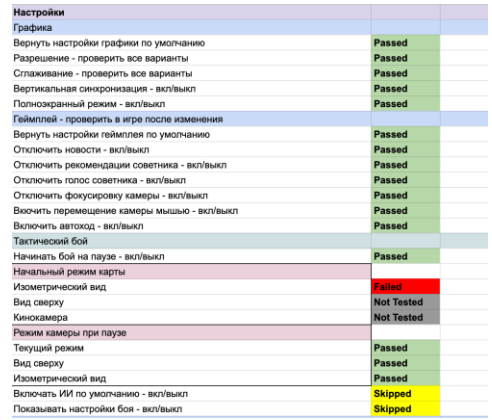
ЧЕК-ЛИСТЫ

Чек-лист – это документ, описывающий то, что должно быть протестировано. Чек-листы чаще всего составляются без детализации, они не избыточны, и их можно скомпоновать в наборы и проверять для любого функционала либо нового, либо регрессионного. Чек-листы лучше сразу писать по требованиям перед стартом тестирования функционала.
Такие списки не подходят новичкам, поскольку не содержат никакой подробной информации и нужно объяснять людям, что предполагается под конкретными пунктами.

Правила написания чек-листа:
- Максимальное покрытие (несмотря на краткость, они должны максимально покрывать тестируемый функционал)
- Четкая структура (элементы чек-листа должны быть читаемы, расположены логично и последовательно)
- Краткость (не пишутся сложные предложения, подробные описания и т.д.)
- Актуальность (должны поддерживаться в актуальном состоянии и обновляться)

Атрибуты чек-листа:
- Область проверки (что подлежит проверке)
- Статус (passed, failed, blocked и т.д.)
- Опционально: комментарии, ссылки, и т.д.

Свойства чек-листа:
- Логичность
- Последовательность и структурированность
- Полнота и не избыточность



Настройки		
Графика		
Вернуть настройки графики по умолчанию	Passed	
Разрешение - проверить все варианты	Passed	
Сглаживание - проверить все варианты	Passed	
Вертикальная синхронизация - вкл/выкл	Passed	
Полноэкранный режим - вкл/выкл	Passed	
Геймплей - проверить в игре после изменения		
Вернуть настройки геймплея по умолчанию	Passed	
Отключить новости - вкл/выкл	Passed	
Отключить рекомендации советника - вкл/выкл	Passed	
Отключить голос советника - вкл/выкл	Passed	
Отключить фокусировку камеры - вкл/выкл	Passed	
Вкючить перемещение камеры мышью - вкл/выкл	Passed	
Включить автоход - вкл/выкл	Passed	
Тактический бой		
Начинать бой на паузе - вкл/выкл	Passed	
Начальный режим карты		
Изометрический вид	Failed	
Вид сверху	Not Tested	
Кинокамера	Not Tested	
Режим камеры при паузе		
Текущий режим	Passed	
Вид сверху	Passed	
Изометрический вид	Passed	
Включать ИИ по умолчанию - вкл/выкл	Skipped	
Показывать настройки боя - вкл/выкл	Skipped	

ТЕСТ-КЕЙС

Тест-кейс – это тестовый документ с последовательностью действий, цель которого проверить функциональность на соответствие фактического результата ожидаемому.

Они разделяются по ожидаемому результату на позитивные и негативные:

- Позитивный – использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.
- Негативный – оперирует как корректными данными, так и некорректными (минимум 1 некорректный параметр) и ставит цель проверить исключительные ситуации (срабатывание валидаторов).

Правила написания тест-кейсов:

- Независимость – результат выполнения одних кейсов не используется в других.
- Детальность шагов – незнакомый с продуктом человек должен суметь повторить написанные кейсы.
- Полнота охвата – кейсы должны максимально охватывать функционал ПО.
- Актуальность – кейсы нужно поддерживать в актуальном состоянии и обновлять при изменении функциональности.

ТЕСТ-КЕЙС

Основные атрибуты тест-кейса:

- ID – представляет собой уникальное значение
- Название – для упрощения и ускорения понимания основной идеи (цели) тест-кейса без обращения к его остальным атрибутам
- Шаги выполнения – описывают последовательность действий, которые необходимо реализовать в процессе выполнения тест-кейса
- Ожидаемый результат – в ожидаемом результате ВСЕГДА описывается КОРРЕКТНАЯ работа приложения

Также могут использоваться дополнительные атрибуты, например, приоритет, начальные условия, вид теста и т.д.

Пример:

46625_1. Проверка ПФ «Расшифровка калькуляции (Зарплата)» от роли «Менеджер коммерческого отдела»
Приоритет: Средний
Требование: SR00046625
Модуль: УСП
Исходные данные: Войти в систему под ролью «Менеджер коммерческого отдела». Заполнены справочники «Физические лица», «Контрагенты», «Суда», «Договоры с контрагентами». Заполнен раздел «Графики ремонта», «Судоремонтные заказы».
Шаги:

1. Нажать слева на панели «Судоремонтные заказы»
2. Выбрать заказ
3. Нажать на командной панели «Печать»
4. В выпавшем меню нажать «Расшифровка калькуляции (Зарплата)»

Ожидаемый результат: Должна открыться печатная форма «Расшифровка основной заработной платы к калькуляции на выполнение работ по надводному ремонту ""»

ОТЧЕТ ОБ ОШИБКЕ ИЛИ БАГ-РЕПОРТ

Баг-репорт — это документ, описывающий ошибку в работе программы или приложения. Он составляется тестировщиком для разработчиков, чтобы те могли быстро понять, воспроизвести и исправить найденный дефект.
Качественный баг-репорт должен включать: краткое описание проблемы, шаги для её воспроизведения, ожидаемый и фактический результат, а также другую необходимую информацию, такую как версия ПО и окружения.

Дефект — расхождение ожидаемого и фактического результата.

Ожидаемый результат — поведение системы, описанное в требованиях.

Фактический результат — поведение системы, наблюдаемое в процессе тестирования.

Список атрибутов отчета об ошибке:

1. ID
2. Название
3. Проект
4. Компонент приложения
5. Номер версии
6. Серьезность
7. Приоритет
8. Статус
9. Автор
10. Исполнитель
11. Краткое описание
12. Окружение
13. Шаги выполнения
14. Ожидаемый результат
15. Фактический результат
16. Приложения

Пункты 3-5, 7-12 при необходимости могут быть опущены.

ОТЧЕТ ОБ ОШИБКЕ ИЛИ БАГ-РЕПОРТ

Серьезность – это атрибут, характеризующий влияние дефекта на работоспособность приложения.

Уровни серьезности дефекта:

1. Блокирующий (Blocker) – назначается дефекту, приводящему приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.
2. Критический (Critical) – назначается дефекту, способствующему неправильной работе ключевой бизнес-логики приложения, наличию дыр в системе безопасности, наличию проблем, ставших причиной временного отказа работоспособности некоторой части системы, без возможности решения проблемы, используя другие входные точки.
3. Значительный (Major) – назначается дефекту, приводящему к отказам в части основной бизнес-логики. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.
4. Незначительный (Minor) – назначается дефекту, затрагивающему некоторые вспомогательные функции приложения, и не приводящему к нарушениям бизнес-логики тестируемой части приложения.
5. Тривиальный (Trivial) – назначается дефекту, не затрагивающему бизнес-логику приложения. Не оказывает никакого влияния на общее качество продукта.

ОТЧЕТ ОБ ОШИБКЕ ИЛИ БАГ-РЕПОРТ

Приоритет – это атрибут, указывающий на очередность выполнения задачи или устранения дефекта.

Уровни приоритета могут быть:

- Высокий (High) - назначается дефекту, который должен быть исправлен как можно быстрее, так как его наличие является критическим для проекта.
- Средний (Medium) – назначается дефекту, который должен быть исправлен, его наличие не является критичным, но требует обязательного решения.
- Низкий (Low) – назначается дефекту, который должен быть исправлен, его наличие не является критичным, и не требует срочного решения.

В каждой организации могут быть свои общепринятые уровни.

Статус – это атрибут, указывающий на стадию жизненного цикла дефекта.

Статусы могут быть:

- Новый
- Отклонен
- Отсрочен
- Открыт
- Открыт повторно
- Закрыт

ОТЧЕТ ОБ ОШИБКЕ ИЛИ БАГ-РЕПОРТ

Пример:
T-2378. [Регистрация пользователя] При попытке регистрации с некорректным email появляется ошибка сервера, и регистрация не завершается.
Окружение: Windows 10, браузер Chrome v115, версия веб-приложения 3.4.1, тестовый сервер INTG.
Приоритет: высокий.
Предусловия: Открыта страница регистрации пользователя на веб-сайте. Браузер Chrome последней версии.
Шаги:
1. В поле "Email" ввести некорректный адрес (например, "user@@example").
2. Заполнить остальные обязательные поля валидными данными.
3. Нажать кнопку «Зарегистрироваться».

Фактический результат: Появляется сообщение об ошибке сервера (500 Internal Server Error), регистрация не происходит, форма не выдает пояснений по корректности email. В консоли браузера и серверных логах зафиксирована ошибка <Error...>.
Ожидаемый результат: Пользователь получает валидационное сообщение о некорректном формате email, регистрация не проходит, но сервер работает без ошибок. В логах отсутствуют ошибки.
Доп. информация: При вводе корректного email регистрация проходит успешно. Ошибка воспроизводится во всех современных браузерах. Приложены логи сервера, скриншоты и консольные ошибки.

<Приложены скриншоты дефекта>