Наталия Матвеева

ШПАРГАЛКА начинающего ТЕСТИРОВЩИКА

Версия 28766790.**009**

2025

СОДЕРЖАНИЕ

Readme 6
История тестирования 7

Информационные технологии

Базовые понятия 9
Виды программного обеспечения 10
Виды ПО по доступу 11
Десктопное ПО vs Мобильные приложения 12
Типы мобильных приложений 13
No-Code, Low-Cod, Mini Apps 14
Направления IT-индустрии 15
Сфера применения IT-технологий 16
Data Science, Machine learning, AI 17
Ключевые области применения AI 18

Разработка ПО

Жизненный цикл ПО (SDLC) 20
Планирование разработки. Требования 21
User Story 22
Use Case 23
Роли в разработке ИТ-продукта 24
Модели, методологии, фреймворки 26
Водопадная модель 27
V-образная модель 28
Итеративная и инкрементальная модель 29
Спиральная модель 30
RAD (Быстрая разработка) 31
Agile. Гибкие методологии разработки 32
Scrum 33
Kanban 34
Extreme Programming 35
Lean 36
Shift-Left Testing 36
DevOps 37
Shift-Right Testing 38
Среды разработки и тестирования ПО 39

Система контроля версий 40
Программирование 41
Объектно-ориентированное программирование 42
Системы счисления 43
UV/UX дизайн 44
Юзабилити 45
Адаптивность vs Отзывчивость 46
Геймификация 47

Web

Интернет 49
Клиент-серверная архитектура 50
Frontend и Backend 51
Аутентификация. Авторизация 52
Монолит vs Микросервисы 53
API 54
REST API 54
SOAP API 54
GraphQL 56
Форматы передачи данных 56
HTTP протокол 57
HTTP-запросы 58
HTTP-методы 59
HTTP-ответы 60
Базы данных 61
Реляционная БД 61
Нереляционная БД 62
SQL 63
Брокеры сообщений 65
Логирование 66
HTML 67
CSS 69
JavaScript 70
Кэш. Куки. Сессии 71
DevTools 72

Введение в тестирование ПО

Тестирование 74
Принципы тестирования 74
QA, QC и Testing 75
Баг 76
Верификация и Валидация 76
Жизненный цикл тестирования (STLC) 77

Тестовая документация

Виды тестовой документации 79
Тест-план vs Тестовая стратегия 80
Оценка трудозатрат 81
Чек-лист. Тест-кейс 82
Жизненный цикл тест-кейса 83
Баг-репорт 84
Приоритет и серьезность бага 85
Жизненный цикл и статусы бага 86
Отчет о тестировании 87
Метрики 87
ПМИ 87

Виды тестирования

Классификация тестирования 89
Функциональное тестирование 90
Альфа и бета тестирование 90
По уровням детализации 91
Модульное. Интергационное. Системное. Приёмочное
Пирамида тестирования 92
Нефункциональное тестирование 93
Инсталляционное. Нагрузочное. Безопасности. Совместимости
Юзабилити-тестирование 94
Интернационализация/Локализация 95
Тестирование требований 96

Тестирование связанное с изменениями 97
Подтверждающее. Регрессионное. Дымовое. Санитарное. Тестирование сборки
По степени важности тестируемых функций 98
Дымовое. Критического пути. Расширенное
По характеру сценария 98
Позитивное и негативное
По степени формализации 99
На основе тест-кейсов. Исследовательское. Свободное
По доступу к коду 100
Тестирование белого, чёрного и серого ящика
По запуску кода на исполнение 101
Статическое и динамическое тестирование
По степени автоматизации 102
Ручное и автоматизированное тестирование
Тестирование API 103
Mock Testing 103
Мобильное тестирование 104

Техники тестирования (тест-дизайн)

Тест-дизайн 106
Предугадывание ошибок 106
Доменное тестирование 107
Эквивалентное разбиение 108
Анализ граничных значений 109
Состояния и переходы 110
Таблица решений 111
Попарное тестирование 112

Напоследок

Инструменты 114
Conclusions 115

README | ТЕЗИСЫ

Тестирование – это НЕ легкий путь в IT, *вопреки навязанному инфоцыганами мнению.*

*Крылатая сущность на страницах Шпаргалки – это **Федя**. Федя – жук, но вовсе не баг! Федя – лицо и душа Шпаргалки. Он не дал мне бросить этот сложный проект, помогая довести его до финала, разбавляя море текста мимимишными картинками.*

*Помимо непосредственно **Шпаргалки**, это ещё и **Roadmap** начинающего тестировщика программного обеспечения.*

*Материал в Шпаргалке изложен максимально кратко, но при этом содержит достаточно полезной информации, **помогая сложить пазл (пусть и низкого разрешения) про индустрию IT** в целом и про тестирование в частности.*

***Шпаргалка кликабельна!** Кликабельно как само содержание, так и кнопка «**содержание**» вверху каждой страницы. А ещё вот так подчёркнутые слова. Но есть нюанс, в некоторых смартфонах стандартное средство просмотра PDF может не поддерживать кликабельность. Просто смените приложение для чтения PDF файлов.*

*Хочу поблагодарить офигенных людей, **которые вдохновили меня** на создание этой Шпаргалки: **Раю Петросян**, **Инну Когинову**, **Алину Шеянову** и **Дмитрия Матвеева**.*

***В конце Шпаргалки** есть пространное послесловие, кому предисловия показалось мало.*

История тестирования:

- **Первые программные разработки** создавались преимущественно для нужд министерств обороны разных стран. (Можно конечно вспомнить Аду Лавлейс с аналитической машиной в XIX веке, но не будем.) На тот момент проверка программы выполнялась самими разработчиками.
- **В 50-х годах** XX века тестирование отделилось в самостоятельную тему и проводилось «от и до», то есть анализ всех путей выполнения кода со всеми возможными данными. Но из-за возрастающего объема информации такой подход стал излишне трудозатратным, по причине нехватки человеческих, временных и прочих ресурсов.
- **К 70-м годам** тестирование сосредоточилось на демонстрации корректности работы продукта и поиске ошибок.
- **К 80-м годам** тестирование обзавелось своей методологией, появилось такое понятие, как предупреждение дефектов. Начали разрабатываться и использоваться первые инструменты для автоматизированного тестирования.
- **К 90-м годам** тестирование переросло в более широкое понятие – обеспечение качества. Оно включало планирование, проектирование, создание и выполнение тестов, и охватывало весь цикл разработки ПО.
- **К 2000-м годам** основными задачми тестирования стало: проверка соответствия ПО спецификации, обнаружение ошибок, предотвращение возможных дефектов.

  Появилось исследовательское тестирование, оно характеризовалось минимум документации, но показывало высокую эффективность.
- **В 2004 году** появился Selenium – набор программ с открытым исходным кодом, созданный для автоматизации тестирования, что продвинуло тестирование на новый уровень. Еще одним ключевым моментом в 2000-х стало внедрение тестирования API (взаимодействие с системой без использования пользовательского интерфейса).
- **2040 год** искусственный интеллект применяется повсеместно. Он сам разрабатывает необходимое программное обеспечение. Тестирование более не нужно, так как машины не ошибаются. Это была шутка. Или нет...

Информационные технологии

БАЗОВЫЕ ПОНЯТИЯ

Информационные технологии (*Information Technology, IT*) – отрасль, которая занимается разработкой, внедрением и поддержкой высокотехнологичных систем и цифровых инноваций.

Данные *(первичная информация)* → ***Информационные технологии*** → **Программный продукт**

Программное обеспечение (*ПО, Software, софт*) – это ИТ, способная выполнить определенную задачу или процесс с помощью компьютера. Результатом разработки ПО является – **программный продукт** *(программа)*, который поставляется заказчику вместе с документацией об установке и использовании.

Свойства хорошего ПО:

- **Надёжность и эффективность.** Способность безотказно выполнять заданные функции при заданных условиях.
- **Удобство применения.** Степень удобства ПО для пользователей, легкость его эксплуатации и изучения.
- **Функциональность.** Способность ПО выполнять возложенные на него задачи, отвечать заданным потребностям.
- **Универсальность.** Адаптируемость ПО к новым функциональным требованиям.
- **Защищённость.** Минимизация угроз, связанных с несанкционированным чтением, изменением информации и т. д.
- **Совместимость.** Способность программных компонентов взаимодействовать друг с другом.
- **Удобство сопровождения.** Степень простоты анализа, тестирования, коррекции компонентов ПО и его обслуживания.

Промежуточное ПО (*Middleware*) – это связующее звено между **железом** и **софтом**. Сюда, например, относят драйверы.

Аппаратное обеспечение (*Hardware, железо*) – материальная часть компьютера, включает в себя все физические части ПК.

Являются ли Веб-сайты ПО?

Скорее да, чем нет, хотя существуют разные мнения по этому поводу. Однако, большинство сайтов предоставляют пользователю возможность управлять ими: выполнять поиск, оставлять комментарии, добавлять публикации, применять разграничение доступа и т.д. То есть сайт способен решить задачи, поставленные пользователем, а это уже определение программного продукта.

ВИДЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

***по назначению** (сфере применения)*

• **Системное ПО**. Состоит из **операционных систем**, утилит, а также драйверов устройств, необходимых для управления компьютером и обеспечения его работоспособности.

Операционная система (ОС) – набор программ, обеспечивающих работу компьютера (или другого устройства) и взаимодействие с ним пользователя.

Наиболее актуальные семейства ОС:

- ***Windows.*** *Платная и закрытая ОС с поддержкой большого количества аппаратного обеспечения. Имеет максимально широкий выбор программ и игр для установки.*
- ***MacOS.*** *Платная и закрытая ОС, устанавливается только на компьютеры Apple. На игры не ориентирована. Имеет высокую скорость работы и высокий уровень безопасности.*
- ***Linux.*** *Открытая и бесплатная ОС с некоторой возможностью выполнения программ от Windows. Широко используется в качестве серверной ОС, в качестве ОС для различных микрокомпьютеров, систем умного дома и тому подобного. В качестве десктопной ОС очень любима разработчиками по причине гибкости настройки и наличию предустановленных средств разработки.*
- ***iOS.*** *Закрытая ОС для мобильных устройств Apple. Ценится благодаря удобному управлению, плавности работы и эстетичному дизайну.*
- ***Android.*** *Открытая и бесплатная ОС для мобильных устройств с возможностью установки приложений из альтернативных источников.*

• **Прикладное ПО**. Различные офисные приложения (например, Microsoft Office), графические редакторы, аудиоплееры, браузеры, игры и т.д. То есть любое ПО, используемое для решения повседневных задач. Прикладное ПО разделяют на *десктопное* и *мобильное*.

• **Инструментальное ПО**. Включает в себя программы и инструменты, предназначенные для создания другого программного обеспечения. К этой категории относятся интегрированные среды разработки (IDE), компиляторы и интерпретаторы (переводят *код программы* в *машинный код*), отладчики, *системы управления версиями*, средства автоматизации тестирования и другие инструменты, упрощающие процесс разработки программного обеспечения.

ВИДЫ ПРОГРАМНОГО ОБЕСПЕЧЕНИЯ
по доступу

Открытое ПО *(Open Source Software)* – ПО с «открытым» исходным кодом (общедоступен для просмотра и изменения).

«Открытая» лицензия не требует, чтобы ПО предоставлялось бесплатно, но чаще всего оно бесплатно.

Закрытое / **Проприетарное ПО** *(Closed Source Software /Proprietary Software)* – частное программное обеспечение, где правообладатель имеет абсолютную монополию на его использование, копирование и модификацию.

Свободное ПО – это ПО, в отношении которого пользователь обладает «четырьмя свободами»:

- *Программу можно использовать с любой целью.,*
- *Можно изучать, как программа работает и адаптировать её для своих целей.*
- *Можно распространять копии программы.*
- *Программу можно улучшать и публиковать свою версию.*

Коммерческое ПО *(Commercial Software)* – ПО, созданное коммерческой организацией с целью получения прибыли.

Бесплатное ПО *(Freeware)* – ПО, лицензионное соглашение которого не требует каких-либо выплат правообладателю.

ОТКРЫТОЕ vs СВОБОДНОЕ

*Большинство **«открытых»** программ является одновременно **«свободными»** и наоборот. По сути эти понятия синонимы, отличие заключается лишь в философии:*

- *Сторонники **открытого ПО** делают упор на **эффективность открытых исходников как метода разработки**.*
- *Сторонники **свободного ПО** считают, что именно **права на распространение, модификацию** и изучение программ являются главным достоинством свободного ПО.*

СВОБОДНОЕ vs БЕСПЛАТНОЕ

*В англоязычных текстах часто возникает путаница, поскольку слово «free» в переводе означает и **«свободное»**, и **«бесплатное»** и соответственно это слово употребляют часто в обоих случаях. Бесплатное ПО распространяется без взимания платы, но недоступно для изменения сообществом и не является свободным. Свободное ПО вполне можно распространять взимая плату, соблюдая при этом критерии свободы.*

ДЕСКТОПНОЕ ПО vs МОБИЛЬНЫЕ ПРИЛОЖЕНИЯ

Десктопное ПО *(Desktop Software)* – это компьютерные программы, разработанные для десктопных операционных систем (Windows, MacOS или Linux) и, соответственно, предназначенные для установки и использования на персональных компьютерах и ноутбуках. В большинстве случаев десктопное ПО может функционировать без доступа к интернету.

Мобильное приложение *(Mobile Application)* – это разновидность прикладного ПО предназначенная для работы на смартфонах, планшетах и других мобильных устройствах.

Мобильные приложения разделяют на:

- ***Нативные приложения***
- ***Кроссплатформенные приложения***
- ***Веб-приложения***
- ***Гибридные приложения***

В принципе это разделение можно применить и к десктопу, но в рамках данной Шпаргалки эти понятия будут использоваться только применительно к мобилкам.

Десктоп *(Desktop Computer, компьютер на столе) – стационарный (настольный) персональный компьютер. В России слово* ***десктоп*** *не прижилось, однако повсеместно употребляется в значении* ***рабочего стола операционной системы.***

Ноутбук *(Notebook, записная книжка) – переносной ПК.*

Лэптоп *(Laptop Computer, компьютер на коленях) – то же, что и ноутбук.* ***В русском языке предпочитают слово ноутбук****, тогда как англоязычные, наоборот, чаще говорят лэптоп.*

Планшет *(от фр. Planchette – маленькая дощечка; англ. tablet computer) – портативное компьютерное устройство с операционной системой и сенсорным экраном для управления.*

Смартфон *(Smartphone, умный телефон) – мобильное устройство, которое сочетает в себе функциональность мобильного телефона и портативного компьютера.*

ТИПЫ МОБИЛЬНЫХ ПРИЛОЖЕНИЙ

НАТИВНЫЕ ПРИЛОЖЕНИЯ

Это прикладное ПО, разработанное для конкретной ОС с использованием своего языка программирования. Нативные приложения требуют установки на устройство пользователя, могут работать без подключения к интернету. Имеют доступ к функциям (Bluetooth, контакты, камера, NFC и др.)

КРОССПЛАТФОРМЕННЫЕ ПРИЛОЖЕНИЯ

Приложение разрабатывается с помощью кроссплатформенных фреймворков и имеют общий код для iOS и Android. Этот тип приложений иногда путают с **гибридными**.

ВЕБ-ПРИЛОЖЕНИЯ

Представляют собой адаптированные веб-сайты, которые открываются в браузере. То есть пользователь не скачивает приложение и не хранит его на своем устройстве, приложение не требует настроек под ОС и не требует обновлений.

Выделяют веб-приложения нескольких категорий:

- **Multi Page Applications** (MPA) – многостраничные веб-приложения. MPA отправляет запрос на *сервер* и страница полностью обновляется в результате *ответа*.
- **Single Page Applications** (SPA) – одностраничное веб-приложение. SPA загружает одну *HTML-страницу* и динамически обновляет контент (без полной перезагрузки) по мере взаимодействия пользователя.
- **Progressive Web Application** (PWA) – технология (можно рассматривать как гибридную) которая сохраняет свою функциональность, даже когда приложение работает в режиме офлайн (без интернета). PWA-приложение можно установить на телефон и запускать иконкой с рабочего стола.

ГИБРИДНЫЕ ПРИЛОЖЕНИЯ

Сочетают в себе элементы нативного и веб-приложения. Необходимо загружать на устройство, но написаны они с помощью *HTML*, *CSS* и *JavaScript*. Основным преимуществом таких приложений является единая кодовая база для разных платформ.

*Для создания используется технология, которая встраивает веб-контент в нативное приложение – **WebView**.*

NO-CODE и LOW-CODE

No-Code и **Low-Code** – это технологии создания программного продукта (*баз данных*, мобильных приложений или сайтов) без *программирования* и написания кода.

No-Code (нет кода). Визуальное программирование **методом перетаскивания элементов** (Drag-and-Drop): кнопок, текстовых блоков, символов, форм для заполнения, фигур, картинок, то есть того, из чего «складывается» сайт или приложение.

Low-Code (низкий код). Технологии визуального программирования, без работы с кодом, но **с возможностью его редактирования**.

Алгоритм создания ПО выглядит так:

- **Создание логики приложения.** Сборка логических последовательностей из готовых блоков.
- **Перетаскивание компонентов на рабочую область.** Выбор и расстановка готовых компонентов, таких как кнопки, текстовые поля, диаграммы и другие.
- **Генерация кода**. Платформа автоматически генерирует код на основе визуально созданной логики и интерфейса.
- **Интеграция и расширяемость**. В продукт легко можно добавить подключение к базам данных или API внешних сервисов.
- **Ручное кодирование**. Для сложных и нестандартных продуктов стоит выбирать Low-code для возможности допилить ПО под свои нужды.

MINI APPS

Mini Apps (мини-приложения) – сервисы внутри больших приложений. Характеризуются быстрой скоростью загрузки и не требуют отдельной установки или регистрации.

Хороший пример – **VK Mini Apps**. Это открытая платформа мини-приложений внутри соцсети «ВКонтакте». Мини-приложения представляют собой интернет-магазины, игры, сервисы доставки и многое другое. Благодаря такому разнообразию любой бизнес сможет создать приложение для решения своих задач.

VK Mini Apps нельзя собрать из готовых элементов по типу No-code. VK Mini Apps это платформа, на которую нужно загружать полноценный код приложения, платформа работает на HTML, JavaScript и CSS.

НАПРАВЛЕНИЯ IT-ИНДУСТРИИ

Современная IT-индустрия охватывает множество направлений. Эти направления, а также сферы применения, могут пересекаться между собой. Стоит выделить ***несколько основных направлений****:*

Разработка ПО *(Software Development)* – включает в себя создание приложений для компьютеров, мобильных устройств, игр, ПО для управления бизнесом и многое другое.

Веб-разработка *(Web Development)* – процесс создания и поддержки веб-сайтов и веб-приложений. Например, от одностраничных сайтов для малого бизнеса до интернет-магазинов и социальных сетей.

Мобильная разработка *(Mobile App Development)* – разработка ПО для портативных устройств. Сюда входят приложения для общения, поиска информации, оплаты товаров, навигации, чтения новостей и других задач.

Искусственный интеллект *(Artificial Intelligence, AI)* – Технология, позволяющая компьютерам выполнять задачи, которые обычно выполняют люди. Включает в себя распознавание речи, компьютерное зрение, машинное обучение, анализ данных и многое другое. *Подробнее здесь*.

Интернет вещей *(Internet of Things, IoT)* – это концепция сети передачи данных между устройствами. Внутри IoT люди могут общаться с «вещами», а «вещи» общаться между собой. Примеры варьируются от **умных часов**, до **умного дома**, который способен контролировать, например, степень освещения помещения. Еще одним примером является концепция **умного производства** *(Smart Factory)*, известная также как «Индустрия 4.0», или IV промышленная революция. Технология помогает оптимизировать производство – обучаясь в реальном времени, она способна управлять производственными процессами.

Робототехника *(Robotics)* – создание роботов для нужд промышленности, медицины, науки и т.д.

Кибербезопасность *(Cybersecurity)* – практика защиты данных и связанных с ними технологий от киберугроз, включая хакерские атаки, вирусы, вредоносное ПО и другие цифровые угрозы.

Облачные технологии *(Cloud Technologies)* – позволяют хранить данные на удаленных серверах, делая их доступными из любой точки мира.

Телеком – область IT связанная с услугами связи: телефон, интернет, кабельное телевидение.

СФЕРЫ ПРИМЕНЕНИЯ IT-ТЕХНОЛОГИЙ

Сфер и отраслей применения IT-технологий десятки, если уже не сотни. ***Самые популярные*** *из них:*

Госсектор – ИТ в деятельности государственных структур. Нужны для решения задач по учёту и планированию бюджетов, переходу к электронному документообороту, для развития информационных сервисов для населения и др.

FinTech *(финансовые технологии, финтех)* – сервисы, которые помогают работать с финансами, платежами, переводами, управлением инвестициями и расчётными счетами. Финтех, как отрасль, включает в себя различные компании, использующие IT технологии для предложения финансовых продуктов и услуг. Сегодняшний финтех охватывает гораздо больше, чем просто мобильные платежи или банкинг. Он включает в себя развитие искусственного интеллекта, блокчейн технологий и даже расширенной и виртуальной реальности в финансовой сфере.

Блокчейн – цифровой реестр осуществленных сделок, транзакций или выполненных контрактов. Технология создана для безопасного хранения и передачи цифровых активов.

EdTech *(технологии образования)*. Под EdTech обычно представляют всё, что связано с онлайн-образованием, например банальные онлайн-курсы. Но понятие EdTech гораздо шире, так как эта сфера связана с применением технологий для улучшения любых процессов обучения. EdTech включает в себя адаптацию учебных курсов и программ под разные категории людей, создание платформ по *геймификации* в обучении, различные технологии, связанные с машинным обучением и искусственным интеллектом, которые помогают оптимизировать обучающие материалы.

MedTech – IT-разработки и инновационные решения в сфере медицины. В MedTech существуют свои направления развития, такие как **HealthTech** и **BioTech**. Эти технологии позволяют диагностировать заболевания, помогать следить за состоянием здоровья или, например, оптимизировать работу врачей.

Gamedev *(геймдев, разработка игр)* – игры для мобильных телефонов, компьютеров, консолей, VR-устройств и других гаджетов относятся к продуктам геймдев индустрии.

DATA SCIENCE | MACHINE LEARNING | AI

Эти термины часто понимают некорректно, путая или пересекаясь в определениях. **Data Science**, **Machine Learning** и **Artificial Intelligence** действительно относятся к одной и той же области науки и технологий, где **общим знаменателем являются данные**.

Data Science *(DS, Наука о данных)* – это дисциплина, которая позволяет проанализировать и **понять** большие объемы информации, обнаружить закономерности и сделать данные полезными.

Artificial Intelligence *(AI, Искусственный Интеллект)* – это раздел IT, ориентированный на создание алгоритмов и технологий, позволяющих имитировать ряд возможностей человеческого мозга. Искусственному интеллекту требуется непрерывная подача данных, на основе которых он способен понимать, планировать, **действовать** и учиться, подобно тому, как это делают люди.

Machine Learning *(ML, Машинное обучение)* – раздел AI, активно применяющийся на практике. ML-алгоритмы на основе набора данных **прогнозируют,** какая информация актуальна для поставленной задачи. Сегодня, когда речь заходит об использовании AI в бизнесе, чаще всего имеется в виду именно ML.

> ***Искусственный Интеллект*** *– инструмент, который помогает* ***Науке о данных*** *получать результаты и решения конкретных проблем. Однако* ***Машинное обучение*** *– это то, что помогает в достижении этой цели.*

Одно из определений во всемирной паутине: *Искусственный интеллект – это сверхчеловек, который никогда не спит, способен легко впитывать любую информацию, не прокрастинирует и анализирует события, не полагаясь на собственные эмоции.*

КЛЮЧЕВЫЕ ОБЛАСТИ AI

Глубокое обучение *(Deep Learning) – ML с использованием многослойных самообучающихся нейронных сетей. Используется при создании таких решений, как чат-боты, маркетинговые рекомендательные системы, автоматические переводчики.*

Нейронная сеть *(Neural Network) – система, моделирующая нейронные связи человеческого мозга и обеспечивающая глубокое обучение. Применение: распознавание голоса, изображений, речи, образов, классификация и обнаружение аномалий и др.*

Когнитивные вычисления *(Cognitive Computing) – технология, стремящаяся имитировать логику работы человеческого сознания в компьютерной модели. Цель – улучшить взаимодействие между людьми и машинами. Это происходит, например, через распознавание человеческого языка и значения изображений для последующего самообучения.*

Обработка естественного языка *или* ***NLP*** *(Natural Language Processing) – технология, которая позволяет компьютерам понимать, распознавать, интерпретировать и воспроизводить человеческий язык и речь.*

Компьютерное зрение *или* ***CV*** *(Computer Vision) – интерпретирует содержимое изображения: фото, растровой и векторной графики, таблиц, PDF и видео и т.д. На компьютерном зрении основана работа автопилотов самоуправляемых транспортных средств (например, Tesla Autopilot), а также системы AI, обрабатывающие информацию с дорожных камер.*

Роботизированная автоматизация процессов *или* ***RPA*** *(Robotic Process Automation) – RPA-системы применяются в бухгалтерии (обработка счетов), розничная торговля (управление запасами), организация служб технической поддержки и др.*

Самая популярная нейросеть в мире на сегодняшний день – ChatGPT *от компании OpenAI, запущенный 30 ноября 2022г.*

ChatGPT *– это чат-бот, обученный на огромном наборе данных человеческого языка. Может поддерживать диалог, писать статьи, сочинять стихи, отвечать на вопросы, давать советы, спорить.*

ChatGPT также может генерировать код на различных языках программирования, включая Python, Java и JavaScript.

Разработка Программного обеспечения

ЖИЗНЕННЫЙ ЦИКЛ ПО (SDLC)

Жизненный цикл разработки программного обеспечения *(Software Development Lifecycle)* – период времени от момента принятия решения о необходимости создания продукта до ввода системы в эксплуатацию.

Жизненный цикл разработки ПО – это часть **жизненного цикла программного обеспечения**, включающего в себя ещё и вывод ПО из эксплуатации.

Модель SDLC с основными этапами разработки

- ***Планирование*** *отвечает на вопрос «Что мы хотим сделать?».*
- ***Анализ требований*** *отвечает на вопрос «Какие проблемы требуют решений?».*
- ***Проектирование и дизайн*** *отвечает на вопрос «Как мы добьемся наших целей?».*
- ***Разработка и тестирование*** *регулирует процесс создания качественного продукта.*
- ***Развертывание*** *регулирует использование финального ПО.*
- ***Поддержка*** *отвечает за обслуживание продукта и устранение проблем, возникающих у пользователей.*

Частью SDLC является ***STLC*** *– жизненный цикл тестирования ПО.* ***STLC фокусируется на качестве*** *разрабатываемого программного продукта.*

ПЛАНИРОВАНИЕ РАЗРАБОТКИ ПО

Планирование – самая первая фаза разработки, начинается со сбора требований и последующей разработки детального плана проекта с определением сроков, бюджета, ресурсов и задач.

Требования – спецификация того, что должно быть реализовано, т.е. описание поведения системы, её свойства или атрибуты. Требования – это основа для проектирования и разработки ПО.

Для понимания реализуемости требований, а также для оптимизации работы команды существует необходимая процедура – тестирование требований.

Виды требований по уровням

Бизнес-требования. Определяют цели и задачи проекта, а также ожидания от его результатов. Бизнес-требования важны для понимания того, что именно хочет получить заказчик от продукта, и как он будет использоваться.

Пользовательские требования. Цели и задачи, которые пользователи могут выполнять с помощью ПО. Могут выражаться в виде фраз утверждений, например, в виде **сценариев использования** (*Use Case*) или **пользовательских историй** (*Use Story*).

Функциональные требования описывают конкретные функции, возможности и поведение системы в тех или иных условиях. Являются основой для разработки тестовых сценариев (*Test Case*) и плана тестирования (*Test Plan*).

*По итогам сбора требований формируется **техническое задание** (ТЗ) на разработку. В ТЗ обычно включают следующие разделы: информация о продукте, глоссарий, бизнес-требования, функциональные требования, технические требования (нагрузка, производительность, требования к технологиям, серверам, скорости работы сервиса и т. д.), спецификация на интерфейс.*

Риски при разработке ПО

***Риски** – это возможные события, препятствующие достижению командой разработки поставленных целей, например:*

- ***Риск превышения** выделенного на проект **бюджета**.*
- ***Риски, связанные с кадрами и знаниями**. Риск потери или нехватки членов проектной команды (болезнь, увольнение) или отсутствие у команды необходимых навыков и знаний.*
- ***Временные риски**. Риски задержки релизов в результате неправильного планирования или меняющихся требований.*

USER STORY

User Story *(пользовательская история)* – способ записи требований с точки зрения пользователя. Метод, который помогает понять нужду клиента и после обсуждения **выбрать**, описать и утвердить **то решение, которое удовлетворит эту нужду**.

Шаблон написания стори: ***Как*** *<роль пользователя>,* ***я*** *<что-то хочу получить>, <с такой-то целью>*

Пример результата: ***Как*** *пациент,* ***я*** *хочу созваниваться с врачами по видеосвязи,* ***чтобы*** *иметь возможность обсудить вопросы здоровья).*

То есть, текст **user story** должен объяснять роль/действия пользователя в системе, его потребность и профит, который пользователь получит после того, как история случится.

User Story не стоит путать с Use Case, это два разных подхода к описанию требований.

User Story можно оценить по критериям «INVEST»

Independent – независимая от других историй, то есть истории могут быть реализованы в любом порядке.

Negotiable – обсуждаемая, отражает суть, а не детали, не содержит конкретных шагов реализации.

Valuable – ценная для клиентов и бизнеса.

Estimable – оцениваемая по сложности и трудозатратам.

Small – компактная, может быть сделана за одну итерацию.

Testable – тестируемая, имеет критерии приёмки.

User Story Mapping (USM)

Карта пользовательских историй – это визуальное представление пользовательских историй, упорядоченных по этапам, приоритету, и предварительно сгруппированных по итерациям (спринтам или релизам).

MVP (Minimum Viable Product)

Минимально работоспособная версия продукта

MVP *– версия продукта с минимальным набором функций, достаточным для проверки на первых потребителях.*

MVP *позволяет проверить жизнеспособность идеи, то есть* ***определить реальный спрос*** *у потребителя и* ***есть ли он вообще****.*

USE CASE

Use Case (*UC, юзкейс, сценарий использования*) – сценарий взаимодействия пользователя с ПО для достижения конкретной цели. Юзкейсы используются для описания *функциональных требований* к системе, они описывают **не каким образом программа делает что-либо, а что именно она делает**.

Юзкейсы содержат следующие сведения:

- **Кто** использует сайт или приложение
- **Что** пользователь хочет сделать
- **Цель** пользователя
- **Шаги**, которые делает пользователь, чтобы совершить определенное действие
- **Описание** того, как сайт или приложение реагируют на действия пользователя.

Юзкейсы – это база для формирования *тест-кейсов*. Use Case тестирование позволяет выявить проблемы в системе, которые могли быть пропущены при тестировании отдельных компонентов ПО (*Unit Testing*). А также оно помогает убедиться, что продукт удобен и понятен для пользователя.

USER STORY vs USE CASE

***User Story** и **Use Case** – это методы формализации требований к ПО, которые часто путают или считают, что это одно и тоже. Между ними действительно есть некоторое сходство, но они не взаимозаменяемы.*

Use Case

Детально и структурировано описывает взаимодействие между системой и пользователями, представляя собой набор шагов, которые пользователь выполняет для достижения определённой цели.

*В **Use Case** может быть описано взаимодействие нескольких участников.*

Use Case сосредоточен на поведении системы с технической точки зрения.

User Story

Выражена в неформальном формате с использованием коротких предложений. Фокусируется на потребностях конкретного пользователя. Она более гибкая и позволяет изменять требования в процессе разработки.

***User Story** излагается от имени какого-то конкретного пользователя.*

User Story сосредоточена на потребностях и целях пользователя.

РОЛИ В РАЗРАБОТКЕ ИТ-ПРОДУКТА

В разработке ПО участвует большое количество специалистов. Приведенный ниже ***список не является исчерпывающим и может различаться*** *в разных командах и проектах.*

Также стоит отметить, что ***понимание этих позиций может быть неодинаковым****, как в разных странах, так и в разных компаниях в пределах одного города.*

Ну и конечно не стоит забывать, что в реальности один ***специалист может совмещать несколько ролей*** *параллельно.*

Стандартный список ролей:

Аккаунт-менеджер – связующее звено между клиентом и командой. Отвечает за коммуникацию с заказчиком, является гарантом того, что фирма выполнит свои обязательства.

Продакт – отвечает за успешный выход создаваемого продукта на рынок, успех, в том числе и коммерческий. Его также называют product-менеджером, продукт-менеджером или менеджером по продукту.

Проджект-менеджер – менеджер проекта, координирует процесс разработки. Следит за соблюдением дедлайнов и требований по проекту, ведёт документацию (бюджет, сбор требований, метрики, делегирование задач и тд.).

Аккаунт-менеджер vs Проджект-менеджер

Аккаунт *представляет сторону заказчика, борется за положительный опыт клиента и выполнение условий сделок.* ***Проджект*** *представитель со стороны команды разработчиков.*

Продакт-менеджер vs Проджект-менеджер

Продакт-менеджер *определяет, что необходимо реализовать, чтобы продукт был конкурентоспособным.*

Проджект-менеджер *оценивает имеющиеся ресурсы и предлагает эффективные способы произвести соответствующий требованиям продукт в обозначенные сроки.*

Бизнес аналитик составляет концепцию продукта, его конечный вид и функционал.

Системный аналитик работает в паре с бизнес-аналитиком или же вовсе совмещает в себе две эти роли. Знает, какие задачи должна решать система. На основе этого составляет техническое задание и список требований.

Бизнес-аналитик фокусируется на оптимизации бизнес-процессов, а системный аналитик – отвечает на проекте за техническую постановку задачи разработки ПО.

Архитектор занимается проектированием системы и её интерфейсов.

Технический писатель рассказывает о продукте и составляет инструкции по эксплуатации.

UI/UX дизайнер отвечает за внешний вид продукта, учитывая удобство его использования клиентом.

Верстальщик *(Frontend Developer)* создаёт вёрстку страниц по макетам дизайнера.

Разработчик *(Backend Developer)* занимается реализацией функционала продукта, то есть пишет код.

DevOps-инженер поддерживает инфраструктуру для разработки и развертывания ПО.

Тестировщик обеспечивает качество продукта.

Тимлид *(Team Lead)* планирует и организует работу команды (10-12 человек) распределяя задачи между сотрудниками, мотивирует их, отвечает за результаты перед заказчиком.

В IT существуют градации специалистов (грейды), но стандартизированных критериев оценки нет, компании сами определяют, что должен знать и уметь сотрудник на каждой ступени:

Стажёр *(Intern) – специалист в начале трудовой карьеры. Часто стажёры работают даже не за еду.*

Джун *(Junior) – сотрудник, выполняющий простые задачи под наставничеством опытных коллег;*

Мидл *(Middle) – специалист с достаточным опытом, решающий сложные задачи и действующий самостоятельно.*

Сеньор *(Senior) – решает сложные задачи и отвечает за результаты.*

МЕТОДОЛОГИИ | МОДЕЛИ | ФРЕЙМВОРКИ

*Главная цель компаний-разработчиков – **создание продуктов высокого качества по привлекательным ценам в короткие сроки**. Задача эта не тривиальная и для её решения используются различные методологии управления проектами.*

Методология *(Methodology)* – описывает заранее определенные принципы и правила управления проектами, включая роли на проекте, а также ценности и рекомендации, которые необходимо соблюдать.

Модель разработки – это абстрактное представление методологии процесса, то есть упрощенный взгляд на жизненный цикл разработки. Модель определяет этапы жизненного цикла, но ничего не говорит о том, как они выполняются.

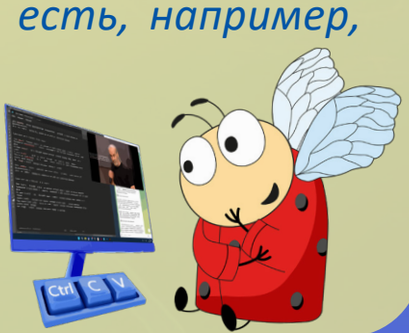
Фреймворк *(Framework)* – готовая система рабочих процессов. Тоже своего рода методология, содержит большое количество правил и практик, но позволяет не использовать их все, оставляя возможность выбрать только то, что нужно для процесса создания ПО. Такой подход позволяет проектам хорошо реагировать на изменения на любом этапе разработки.

__Методология описывает принципы__, ценности и лучшие рекомендации, которые нужно соблюдать, в то время, как __фреймворк показывает способ__ соблюдения. Фреймворки обозначают что, где, когда и как сделать, чего опасаться, и на что обратить внимание. То есть фреймворк, в теории, позволяет ничего не знать об управлении проектами, но собрав команду и предоставив ей выбранный фреймворк – можно ожидать нужного результата.

Фреймворки для управления проектами, такие как: Scrum, Kanban, XP – используются в философии Agile.

Философия в управлении проектами – свод идеологических принципов, которые отвечают на вопрос *«Что мы хотим учитывать при управлении проектом?»*.

***Фреймворки** – это инструменты, и они используются не только для управления проектами, есть, например, фреймворки для написания кода, такие как: **бэкенд-фреймворки** (Django, Symfony, Laravel), **фронтенд-фреймворки** (Angular, React), **фуллстек-фреймворки** (JS Meteor) и др.*



ВОДОПАДНАЯ МОДЕЛЬ

Waterfall Model *(водопадная/каскадная)* – это классическая модель жизненного цикла разработки ПО. Была придумана в 1950-х годах, и долгое время все продукты и проекты реализовывали с её помощью.

Модель подразумевает **последовательное прохождение стадий** (анализ, требования, проектирование, реализация, тестирование), каждая из которых должна завершиться полностью до начала следующей. Однако **это работает только тогда, когда требования стабильны и не меняются в ходе разработки**.

Особенности:

- *Разработка проходит быстро.*
- *Стоимость и сроки заранее определены.*
- *Хороший результат только в проектах с чётко и заранее определёнными требованиями.*
- *Нет возможности сделать шаг назад.*

Сейчас в чистом виде Waterfall почти не применяют (бытует мнение, что в классическом понимании водопадной модели никогда и не существовало). Связано это в первую очередь с человеческим фактором – где всего из-за одной ошибки на любом этапе разработки может появиться другая, которая повлечёт за собой третью, и так далее. Второй момент актуален в настоящее время, когда за полгода-год банально могут устареть технологии.

V-ОБРАЗНАЯ МОДЕЛЬ

V-Model – усовершенствованная водопадная модель. В V-модели следующий этап начинается только после завершения предыдущего. Но в отличие от водопада **в V-модели для каждого действия по разработке** (спуск буквы V) **существует соответствующее ему действие по тестированию** (подъём буквы V).

V-Model используется в разработке ПО для клиник или в создании программного обеспечения для управления подушками безопасности и т.д.

Особенности:

- *Тестирование продукта с ранних стадий проектирования.*
- *Если при разработке архитектуры была допущена ошибка, то вернуться и исправить её будет стоить дорого.*

В русскоязычных материалах и статьях по разработке ПО, можно встретить упоминание о другом названии V-модели – разработка через тестирование (Test Driven Development, TDD).

***TDD** – это метод, при котором тесты разрабатываются до написания программного кода. Относится к гибким методологиям разработки программного обеспечения. **Такое сравнение, конечно, не является корректным.** Есть варианты, при которых TDD можно встроить в V-модель, тем самым добавив ей гибкости, но называть синонимами эти понятия не стоит.*

ИТЕРАТИВНАЯ И ИНКРЕМЕНТНАЯ РАЗРАБОТКА

Эти подходы часто путают друг с другом или используют, как взаимозаменяемые понятия, что не верно.

Инкремент – завершённая и полезная часть работы, которая повышает ценность продукта. **Итерация** – это отдельный цикл разработки внутри проекта.

Итеративная разработка по своей природе является инкрементальной, но инкрементная не обязательно должна быть итеративной.

Итеративная и инкрементная модель, это комбинация из двух подходов, она включает в себя принципы постепенного добавления функций, циклического выпуска, постоянного экспериментирования и цикла обратной связи.

- **Итеративная разработка** – подход, при котором **продукт разрабатывается, тестируется, а затем модифицируется на основе обратной связи на каждой итерации**. То есть после каждой итерации выпускается новая версия продукта с улучшением уже имеющегося функционала или добавлением нового. Основная цель этой разработки – улучшить качество продукта за счет обеспечения глубокого тестирования и обратной связи.
- **Инкрементная разработка** – это процесс в ходе которого **проект разбивается на небольшие функциональные компоненты и разработка движется линейно** (по этапам). Каждый этап добавляет продукту функциональности. Подход используется, когда конечная цель четко определена и требования проекта ясны. *Иногда эту модель сравнивают с Водопадом*.

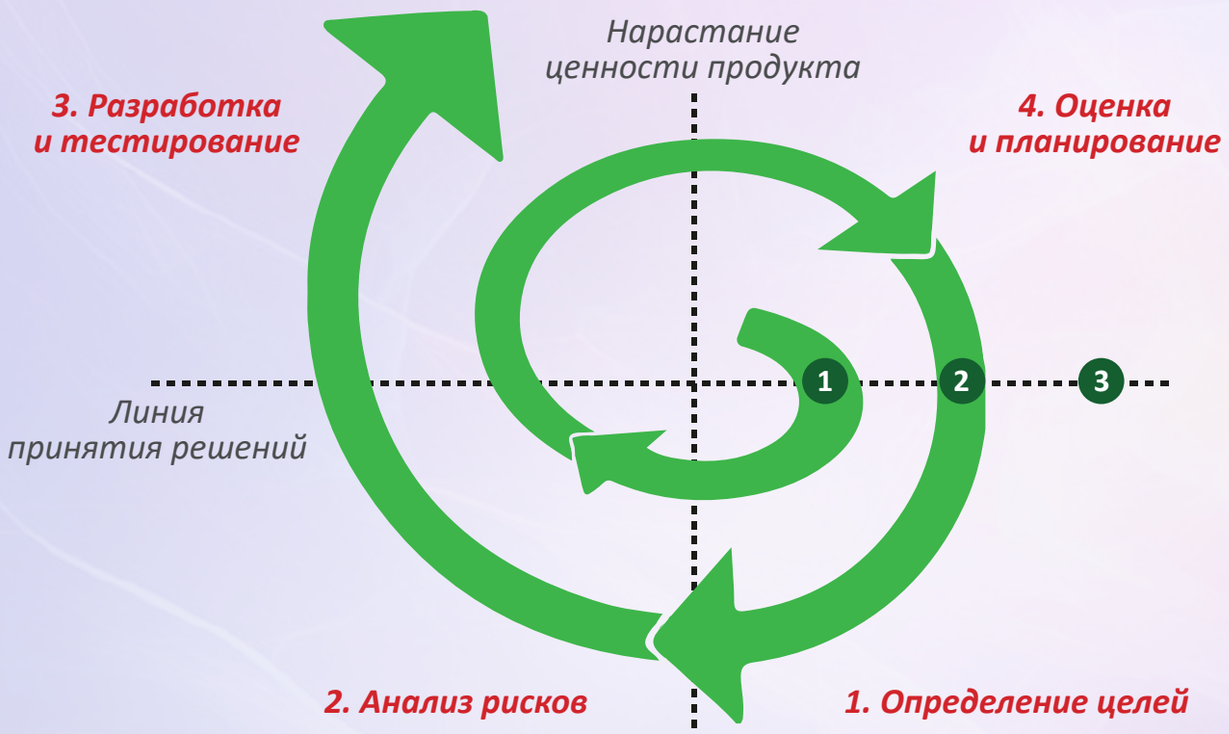
Вывод: **итеративная инкрементная разработка** с точки зрения жизненного цикла является итерационной (многократное повторение одних и тех же стадий), с точки зрения развития продукта модель является инкрементальной (приращение новых полезных функций).

СПИРАЛЬНАЯ МОДЕЛЬ

Spiral Model – представляет собой частный случай *итерационной инкрементной модели*. Особенностью спирали является **фокусировка на выявлении всех возможных рисков**, присвоение им определённого уровня приоритета на основе их значимости, и разработка стратегий преодоления этих *рисков*.

Процесс разработки

Состоит из последовательных итераций, каждая из которых включает в себя несколько повторяющихся стадий:



1. Определение целей. Сбор требований и запросов на изменения от заказчика, разработка и анализ целей. По возможности, в присутствии клиента создается прототип для оценки качества продукта, предлагаются альтернативные решения.

2. Анализ рисков. Оценка возможных решений для выбора лучшего и выявление рисков, связанных с этим решением. Устранение этих рисков и создание прототипа.

Прототипы могут быть нарисованы на бумаге или созданы в Фигме, соответственно отличаться уровнем детализации и кликабельностью элементов.

3. Разработка и тестирование. Выполнение задач, связанных с созданием программного продукта в соответствии с требованиями, проведение тестирования. На этом этапе должны быть реализованы все изменения, предложенные заказчиком во время оценки на предыдущей итерации.

4. Оценка и планирование. Оценка клиентом разрабатываемого продукта. После получения отзыва заказчика команда разработки планирует следующую итерацию.

RAD (RAPID APPLICATION DEVELOPMENT)

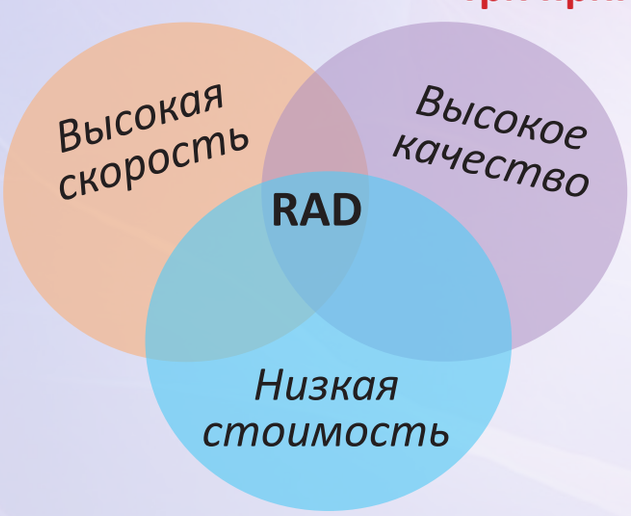
RAD (Быстрая разработка приложений) – методология, ориентированная на максимально быстрое получение результата в условиях сильных ограничений по срокам и бюджету и часто без чётко определённых требований к продукту.

RAD и Agile созданы, как ответ на предполагаемые ограничения структурированных традиционных методов управления проектами, таких как водопадная и итеративная модели.

Особенности RAD разработки

Работа в RAD строится маленькими итерациями. Такой подход дает возможность, наблюдая за реакцией пользователей на продукт, переделывать его на лету. Очередной цикл разработки начинается не раньше, чем пользователь оценил результаты предыдущего.

Три принципа RAD



1. Высокая скорость выдаваемого результата.
2. Низкая стоимость при фиксированном бюджете.
3. Высокое качество. Сделанное, должно работать без ошибок, но объём может быть меньше запланированного.

Преимущества:

- ***Можно выпускать программы при малом или гибком бюджете.*** *Есть деньги на один модуль – идёт его разработка. Появились деньги на другие модули – пилят их. Деньги закончились – но созданным софтом можно пользоваться. RAD часто применяют при создании MVP.*
- ***Можно менять возможности программы в любой момент разработки.***

Недостатки:

- ***Плохо подходит для разработки больших программных комплексов****, например для управления производством. Так как в такое ПО сложно вносить изменения на ходу – работа одного модуля может повлиять на работу других.*
- ***Заказчик должен постоянно участвовать в обсуждении и тестировании****.*

ФИЛОСОФИЯ AGILE | ГИБКИЕ МЕТОДОЛОГИИ

(Agile Software Development)

AGILE – обобщающий термин для целого ряда подходов и практик, целью которых является оперативная реакция на изменения в ходе процесса разработки.

Agile – это не методология, а философия со своими ценностями и принципами разработки программного обеспечения.

Ценности Agile

1. ***Люди*** *важнее процессов и инструментов.*
2. ***Работающий продукт*** *важнее исчерпывающей документации.*
3. ***Услышать позицию клиента и сотрудничать с ним*** *важнее согласования условий контракта.*
4. ***Готовность к изменениям*** *важнее первоначального плана.*

Суть принципов Agile-манифеста:

- Главная цель – удовлетворение потребностей заказчика. Любые изменения в проекте приветствуются.
- Работа делится на итерации (циклы по 2 – 8 недель). В конце каждой итерации заказчик получает готовый минимально работающий продукт (его часть), которую можно использовать.
- По итогам каждой итерации команда анализирует результаты и меняет приоритеты для следующего цикла.
- Разработчики и заказчик должны работать вместе ежедневно, обмениваясь идеями и полезной информацией.
- Над проектом должны работать мотивированные специалисты.

Примеры Agile: *Scrum* – фреймворк «по умолчанию», для многих является синонимом Agile. Работа строится по итерациям. *Kanban* – фреймворк с акцентом на визуализации задач на досках и оптимизации рабочего процесса.

RAD vs AGILE

RAD ***и Agile имеют ряд сходств****, таких как: быстрое создание высококачественного ПО, гибкость, высокий уровень взаимодействия с клиентами.*

Но ***RAD*** *в первую очередь* ***ориентирован на быстрое прототипирование,*** *уменьшая необходимость в обширном планировании и документации, в то время как* ***Agile ориентирован на разбиение проекта на этапы*** *с постепенными улучшениями на протяжении всего проекта.*

SCRUM

Скрам – фреймворк гибкой разработки. Процесс управления состоит из коротких итераций – спринтов. **Каждый спринт предполагает взаимодействие** команды разработчиков ***(Development Team)***, заказчика ***(Product Owner)*** и ***Scrum-мастера***.

Product Owner. *Общается с заказчиками и потребителями, планирует задачи и подробно описывает их.*

Development Team. *Команда, которая создает продукт.*

Scrum-мастер. *Анализирует выполненные задачи и ищет способы улучшить процесс работы, организует встречи.*

Мероприятия внутри каждого спринта:

Sprint Planning Meeting *(планирование)*. Собрание в начале спринта – команда просматривает задачи, анализирует их сложность и определяет, сколько успеет сделать за спринт.

Daily Scrum Meeting *(ежедневное собрание)*. Короткое ежедневное собрание, где все члены команды должны рассказать о проделанной вчера работе и о планах на сегодняшний день.

Retrospective Meeting *(ретроспектива)*. Проводится в конце спринта. Обсуждаются результаты выполненных задач и идеи, как можно улучшить процесс работы.

Суть фреймворка:

- Работа делится на спринты длительностью 1-3 недели.
- Перед каждым спринтом команда формирует список задач.
- После выполнения спринта проводится ретроспектива.

Артефакты в Скрам:

Бэклог продукта – упорядоченный и постоянно обновляемый список с кратким описанием того, что необходимо для улучшения продукта.

Бэклог спринта – это подборка задач из бэклога продукта, выбранных для исполнения в текущем спринте.

Инкремент – это список всех элементов бэклога, которые были отмечены как выполненные.

Product Owner vs Product Manager

Продукт менеджер *–* ***профессия*** *(специалист отвечает за успех продукта),* ***Продукт оунер*** *–* ***роль в Scrum*** *(следит за тем, чтобы соответствовать тому, что прописано в Scrum.).*

KANBAN

Канбан – гибкий фреймворк, в котором процесс разработки прозрачен для всех членов команды и распределяется равномерно. Визуализация работы осуществляется с помощью канбан-доски.

Kanban-доска *(физическая и виртуальная) – инструмент визуализации и документирования процесса разработки.*

Основные элементы Kanban-доски:

Колонки. *Представляют различные стадии рабочего процесса, отражая этап, через который проходят задачи.*

Карточки. *Каждая задача представлена карточкой. Карточки содержат информацию о задаче, её описание, сроки выполнения и другие полезные сведения.*

Система Канбан основана на принципах:

- **Визуализация.** Канбан-доска, с этапами выполнения текущей задачи: «запланировано» / «в работе» / «сделано» и др.
- **Разделение задач.** Четкое понимание целей упрощает процесс.
- **Фокусировка на работе.** Невыполненные задачи требуют особого внимания – перераспределения ресурсов и возможность подключения дополнительных сотрудников.

Scrum vs Kanban

*В **Scrum** работа ведётся **короткими спринтами**. В **Kanban** **нет ограниченных по времени** спринтов и поступление новых задач в процессе – абсолютно нормальная практика.*

*В **Scrum** формируются **универсальные команды**, в **Kanban** может работать **несколько узкопрофильных команд**.*

Scrumban

***Гибридный подход к разработке, сочетает в себе элементы Scrum и Kanban**. Используется, когда Scrum или Kanban, в «чистом» виде, не подходят для решения задач проекта.*

EXTREME PROGRAMMING

Экстремальное программирование *(XP)* – фреймворк гибкой методологии, напоминает *Scrum* – разработка ведется короткими итерациями при активной взаимосвязи с заказчиком.

Практики экстремального программирования:

Короткий цикл обратной связи

- Разработка через тестирование *(Test-driven Development)*
- Игра в планирование *(Planning Game)*
- Заказчик всегда рядом *(Onsite Customer)*
- Парное программирование *(Pair Programming)*

Непрерывный процесс

- Непрерывная интеграция *(Continuous Integration)*
- Рефакторинг *(Design Improvement)*
- Частые небольшие релизы *(Small Releases)*

Понимание, разделяемое всеми

- Простота проектирования *(Simple Design)*
- Метафора системы
- Коллективное владение кодом *(Collective Code Ownership)*
- Стандарт оформления кода *(Coding Standard)*

Социальная защищённость

- 40-часовая рабочая неделя *(Sustainable Pace, Forty-hour Week)*

Разработка через тестирование (TDD)

TDD – методология разработки ПО, которая основывается на повторении коротких циклов разработки:

- Пишется тест, покрывающий желаемое изменение.
- Пишется программный код, который реализует желаемое поведение системы и позволяет пройти написанный тест.
- Проводится рефакторинг написанного кода с постоянной проверкой прохождения всех тестов.

Опираясь на тесты, разработчики понимают, какая функциональность необходима пользователю. Часто детали интерфейса появляются задолго до окончательной реализации решения.

Близкие с TDD методологии:

Разработка через поведение (BDD). Тестируются возможности кода на: «что он должен уметь делать в разных ситуациях».

Разработка через приёмочное тестирование (ATDD). Критерии, описанные заказчиком, автоматизируются в приёмочные тесты, используемые потом в обычном процессе разработки.

LEAN

Lean Software Development *(LSD, бережливая разработка)* – подход, основанный на оптимизации времени и ресурсов, устранении потерь и предоставлении только того, что нужно продукту.

Lean разработана на базе Agile, но направлена на повышение эффективности за счёт ***исключения из процесса разработки всего лишнего****, при этом с акцентом на непрерывное совершенствование и ценность для клиента.*

Принципы бережливой разработки

- **Исключение потерь.** Потери это всё то, что не добавляет ценности для потребителя (бюрократия, ненужный код, функции).
- **Забота о качестве.** Для этого используются различные тактики, например разработка на основе тестов.
- **Постоянное совершенствование.** Короткие циклы разработки, раннее тестирование, анализ и обратная связь с заказчиком.
- **Предельно отсроченное принятие решений.** Решение следует принимать только на основе существенных фактов.
- **Быстрая доставка.** Короткие итерации, разработка MVP.
- **Мотивация команды.** Нельзя рассматривать людей исключительно как ресурс.

SHIFT-LEFT TESTING

Shift-Left Testing *(тестирование со сдвигом влево)* – это методология упреждающего тестирования, которая уделяет приоритетное внимание раннему обнаружению ошибок в *жизненном цикле разработки* программного обеспечения.

Shift-Left *– это визуальное представление, где на временной шкале процесса разработки тестирование сдвигается к началу.*

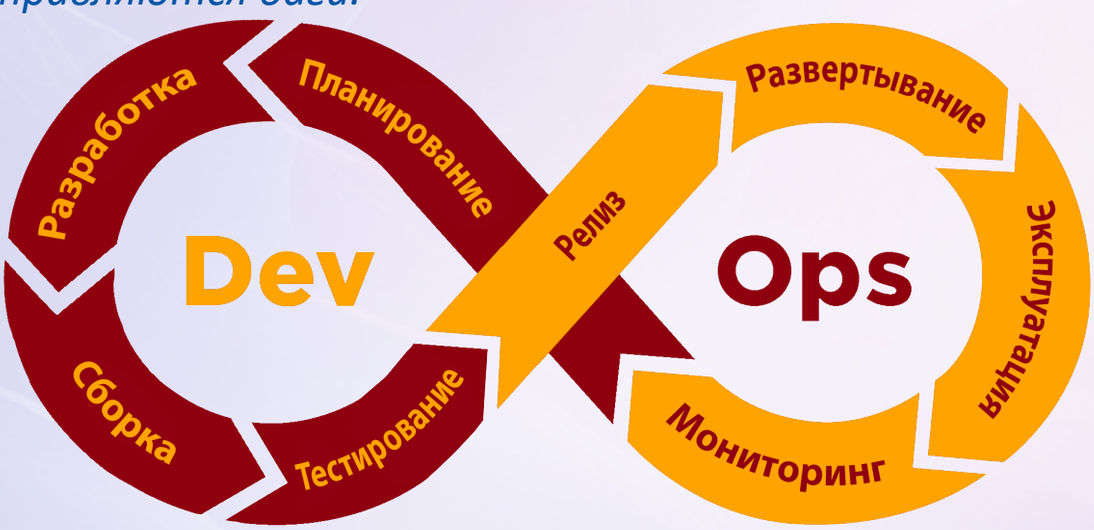
Существует и противоположный Shift-Left тестированию подход – **Shift-Right Testing**. *Подробнее здесь*.

DEVOPS

DevOps от Development (разработка) & **Operations** (управление) – методология разработки ПО, в основе которой лежат практики, нацеленные на автоматизацию процессов разработки и взаимодействия команды между собой и с заказчиком.

Основные инструменты Devops, это **Docker**, **k8s (Kubernetes)**, **Jenkins**, они позволяют отслеживать задачи, быстро интегрировать изменения и оперативно выявлять проблемы.

DevOps своими практиками внедряет философию Agile непосредственно в процесс разработки. *Благодаря сочетанию DevOps и Agile процесс выпуска ПО упрощается (и становится прозрачным), быстрее находятся и исправляются баги.*



Основные практики DevOps:

CI/CD *(Continuous Integration, Continuous Delivery)* – непрерывная интеграция, поставка и развёртывание. Базовая часть культуры DevOps. После сборки все изменения кода автоматически развёртываются в *тестовой среде*. После этого развёрнутый код прогоняют через *автоматические тесты*, и запускается развёртывание в производственной среде.

Непрерывное тестирование. Практика непрерывного тестирования помогает как можно раньше выявлять возможные риски на всех стадиях разработки ПО – чтобы ошибки не повлияли на конечных пользователей.

Непрерывный мониторинг. За уже выложенным, развёрнутым приложением постоянный сбор данных с помощью автоматизированных систем. Затем эти данные анализируются для понимания состояния процесса разработки.

Микросервисы. Это архитектура приложения, согласно которой оно разбито на множество мелких и независимых друг от друга модулей. *Подробнее здесь.*

SHIFT-RIGHT TESTING

Shift-Right Testing *(тестирование со сдвигом вправо)* – это подход к разработке, при котором тестирование проводится на поздних этапах жизненного цикла разработки ПО и происходит сразу на *проде*.

Цель Shift-Right тестирования – непрерывно проверять производительность и удобство использования системы, получая обратную связь от реальных пользователей в реальных условиях использования. Подход позволяет выявлять новые, иногда неожиданные, сценарии и проблемы, которые могли быть упущены на ранних этапах *SDLC*.

Практики Shift-Right Testing:

- **A/B-тестирование**. Это маркетинговое исследование, где на рынок выпускаются две похожие версии продукта с небольшими различиями, для двух схожих наборов пользователей. Метод позволяет понять, какая из версий лучше справляется с поставленными задачами.
- **Бета-тестирование**. Продукт выпускается на рынок для конкретного набора пользователей, для получения обратной связи.
- **Канареечное тестирование**. Постепенное развертывание новых версий ПО сначала для небольшого количества пользователей.
- **Производственный мониторинг**. Мониторинг работы системы для быстрого выявления любых проблем. Сюда входит получение информации о показателях производительности, времени отклика, частоты ошибок, работоспособности сервера и др.
- **Сине-зеленое развертывание**. Стратегия управления релизами, которая минимизирует риски за счёт использования двух идентичных производственных сред. На одной из них (например "синей") развернута стабильная версия, а на другой ("зеленой") производится тестирование следующей версии. По окончании тестирования пользовательский трафик переключается на "синюю" среду, а на "зеленой" начинают тестирование следующей версии.

Shift-Left vs *Shift-Right*

Эти подходы сосредоточены на разных этапах SDLC, но они не являются взаимоисключающими. Комбинация Shift-Left и Shift-Right помогает быстро реагировать на отзывы пользователей и на любые проблемы, возникающие на проде.

СРЕДЫ РАЗРАБОТКИ И ТЕСТИРОВАНИЯ ПО

(Окружения развёртывания)

Окружение (Environment) – это управляемая конфигурация ресурсов, на которых запускается ПО. Эти ресурсы могут включать в себя физические серверы, виртуальные машины, сетевые устройства и другие компоненты. В зависимости от решаемых задач, таких как: тестирование, отладка, добавление новой *фичи* или наполнение *БД*, одновременно приходится запускать несколько (разных или идентичных) версий приложения.

Почти все компании-разработчики ПО в мире используют по крайней мере две среды развертывания – **Development** (DEV, окружение разработчика) и **Production** (PROD, используют конечные пользователи), часто с промежуточными этапами посередине (DEMO, TEST, STAGE и др.).

DEV – среда разработки, где происходит создание продукта. Здесь всегда самая свежая версия кода и продукта.

DEMO – среда, содержащая промежуточный результат (в основном для заказчика), который можно «потрогать руками» и посмотреть, так ли всё работает, как должно.

TEST – здесь происходит полноценное тестирование продукта на стабильность. По сути, это испытательный полигон, который при необходимости можно зачистить и начать снова.

STAGE – среда, которая максимально приближена к тому, что видят перед собой конечные пользователи. Её еще называют предпродакшн.

PROD – среда, с которой работают пользователи. При заходе на сайт, в приложение или игру взаимодействие происходит с тем, что опубликовано на PROD-сервере.

Следует отметить, что точные определения, количество сред и границы между ними могут сильно варьироваться в разных компаниях и на разных проектах.

***Huyak-huyak и в продакшн** (ХХП)*

*IT вариант известной фразы – «рас-рас и готово». По сути еще одна неофициальная разновидность гибкой методологии разработки ПО. **Подразумевает выкатку непроверенных функций/исправлений сразу на PROD**, минуя все остальные окружения.*

Существует даже мнение, что это и есть истинный Agile. Однако радует тот факт, что в большинстве случаев ХХП упоминается в качестве мема.

СИСТЕМА КОНТРОЛЯ ВЕРСИЙ

Version Control System (VCS, СКВ)

Инструмент разработки, позволяющий фиксировать все изменения в проекте. **Самая популярная распределённая система контроля версий на сегодняшний день – Git**.

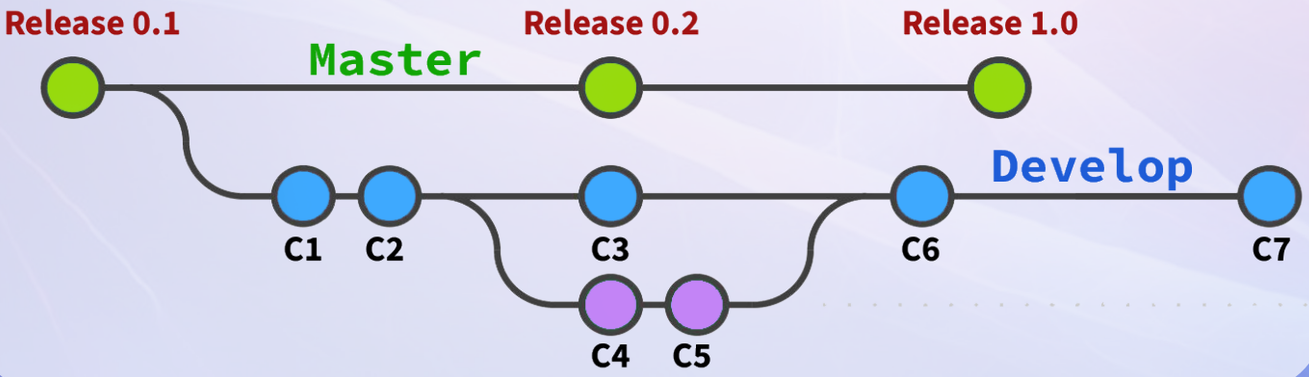
Git фиксирует все изменения, сохраняя различные версии и при необходимости дает вернуться к более раннему состоянию. Позволяет организовать удобную командную работу с одним и тем же кодом, не мешая друг другу.

*Основными сервисами он-лайн хостинга репозиториев для работы с Git являются **GitHub** и **GitLab**.*

Репозиторий Git – место *(по сути папка)*, в котором хранится весь код проекта и вся история его изменений.

Коммит – логическая единица изменений в проекте, точка, к которой можно вернуться, если возникнут какие-то проблемы.

Ветка – последовательность коммитов. По умолчанию репозиторий содержит две главные ветки – **Master** (создаётся при инициализации репозитория) и **Develop** (ветка для разработки). Дополнительные ветки используются для создания нового функционала, исправления ошибок или для подготовки выпуска новой версии проекта. Количество веток зависит от проекта, компании и команды разработки. Вносимые изменения в файлы на одной ветке никак не скажутся на файлах в другой ветке.



КОММИТИМ, ЛОГИРУЕМ, ПУШИМ, МЁРЖИМ

или основной набор команд в Git

git commit – фиксация изменений

git log – просмотр журнала коммитов в обратном хронологическом порядке

git push – отправка изменений из локального репозитория на сервер

git merge – слияния одной или нескольких веток

git pull – извлечение в локальный репозиторий последней версии проекта.

ПРОГРАММИРОВАНИЕ

Программирование – это процесс создания ПО с помощью написания программного кода.

Программный код – программа в текстовом виде написанная на языке программирования, содержит инструкции, которые компьютер может понять и выполнить. **Код следует разделять на**:

- **Исходный код** – текст программы, написанный человеком на каком-либо языке программирования.
- **Исполняемый код** – результат преобразования исходного кода. Различают исходный код по методу выполнения:
  - ***Компилируемый***. *Исходный код сначала компилируется в машинный код, а затем выполняется процессором.*
  - ***Интерпретируемый***. *Исходный код выполняется непосредственно интерпретатором, который читает и исполняет код построчно, без предварительной компиляции в машинный код.*

Кодинг – это процесс написания кода (или *скрипта*) на языке программирования.

Программировать *– находить оптимальное решение поставленной задачи.* ***Кодить*** *– писать код программы.*

Язык программирования (ЯП) – формальный язык, предназначенный для записи (создания) программ. **У каждого языка есть свой набор парадигм, которым следуют при разработке**.

Большинство популярных языков программирования – высокоуровневые. Такие языки близки по написанию к обычному человеческому языку или иным привычным знаковым системам (например, математическим формулам).

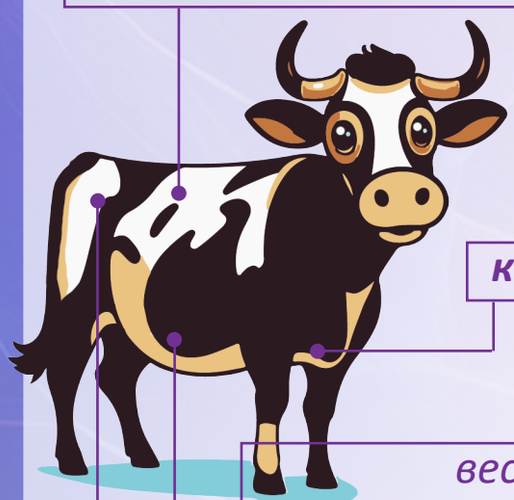
Парадигма программирования – это структурированная совокупность идей и правил (иногда запретов на определенные действия), которые определяют способ написания кода. Выделяют два основных класса парадигм: **императивные** (описывает последовательность действий, которые преобразуют состояние программы) и **декларативные** (описывает то, чего вы хотите достичь с помощью программы, а не то, как она должна работать).

Легаси *(legacy, наследие) – код, написанный некоторое время назад и переданный «по наследству» от разработчика разработчику. Он может быть единственным нормально работающим куском кода, но гораздо чаще легаси является причиной дергающегося глаза у разработчиков.*

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. Является одним из представителей *императивной* парадигмы, почти всё современное программирование построено на принципах ООП. Примеры языков: Java, Python, C++, C#, PHP и др.

Элементы ООП:



КЛАССЫ. Шаблон, на базе которого создаются **объекты**. Классы определяют, какие атрибуты будут иметь объекты, но не значение этих **атрибутов** для конкретного **объекта**.

ОБЪЕКТЫ. Экземпляры **класса** со своими данными (значениями **атрибутов**) и набором доступных действий (**методов**).

АТРИБУТЫ. Характеристики, определяющие состояние объекта. Атрибуты объекта определены в шаблоне **класса**.

МЕТОДЫ. Методы представляют поведение **объекта,** определяя действия, которые он способен выполнять.

Основные принципы ООП

- **Инкапсуляция.** Информация, необходимая для работы конкретного объекта, храниться внутри этого объекта. И только определённые данные доступны для внешних функций и объектов.
- **Абстракция.** Способ выделить общие, наиболее важные атрибуты и методы и исключить (абстрагировать) незначимые.
- **Наследование.** Позволяет создавать аналогичные объекты, которые поддерживают базовое количество свойств (атрибутов) и возможностей (методов) и добавлять свои.
- **Полиморфизм.** Один и тот же метод может работать по-разному в зависимости от объекта, где он вызван, и данных, которые ему передали.

СИСТЕМЫ СЧИСЛЕНИЯ

Система счисления – это система описывающая способы записи чисел. Подразделяется на:

- ***Непозиционная***. *Каждая цифра числа имеет величину, не зависящую от её позиции (разряда).*
- ***Позиционная***. *Каждая цифра имеет значение в зависимости от её позиции.*

Основные позиционные системы счисления:

Десятичная. Применяется в обыденной жизни повсеместно. Каждый разряд – это только одна цифра в диапазоне от **0** до **9**.

Двоичная. Применяется повсеместно в информационных технологиях. Представляет собой двоичный (или бинарый) код. Бинарный код – это понятный компьютеру язык, называемый ещё *машинным кодом*. Используются символы **0** и **1**.

*Например, десятичное число – **2024**, в двоичной системе будет записано как – **11111101000**.*

Восьмеричная. Диапазон цифр для записи от **0** до **7**. Представляет собой компактный способ представления двоичных чисел, сейчас почти полностью вытеснена шестнадцатеричной.

Шестнадцатеричная. Применяется в низкоуровневом программировании и компьютерной документации, в графическом дизайне, оформлении сайтов и прочих работах с цветом. Используются цифры от **0** до **9** и латинские буквы от **A** до **F**.

*Например, **голубой цвет** будет записан так – **#00FFFF**.*

БИТЫ и БАЙТЫ

Бит *– минимальная единица информации в **двоичной системе**. Один бит может принимать одно из двух значений: **0** или **1**. Интерпретировать эти значения можно, как: «истина» или «ложь», «да» или «нет», «вкл» или «выкл», «+» или «–».*

*Вся информация хранящаяся в компьютере представлена в виде последовательности нолей и единиц. Носителем информации является электрический заряд. Оперативная память, SSD или флешка состоят из множества ячеек, в которых заряд может либо **отсутствовать**, либо **присутствовать**, что соответствует **0** и **1**. Комбинации таких ячеек являются физическим кодированием информации.*

Байт *– группа из 8 бит. Всем известные объемы компьютерной информации измеряются в байтах, где 1024 байта – это килобайт, 1024 Кб – мегабайт, 1024 Мб – гигабайт и тд.*

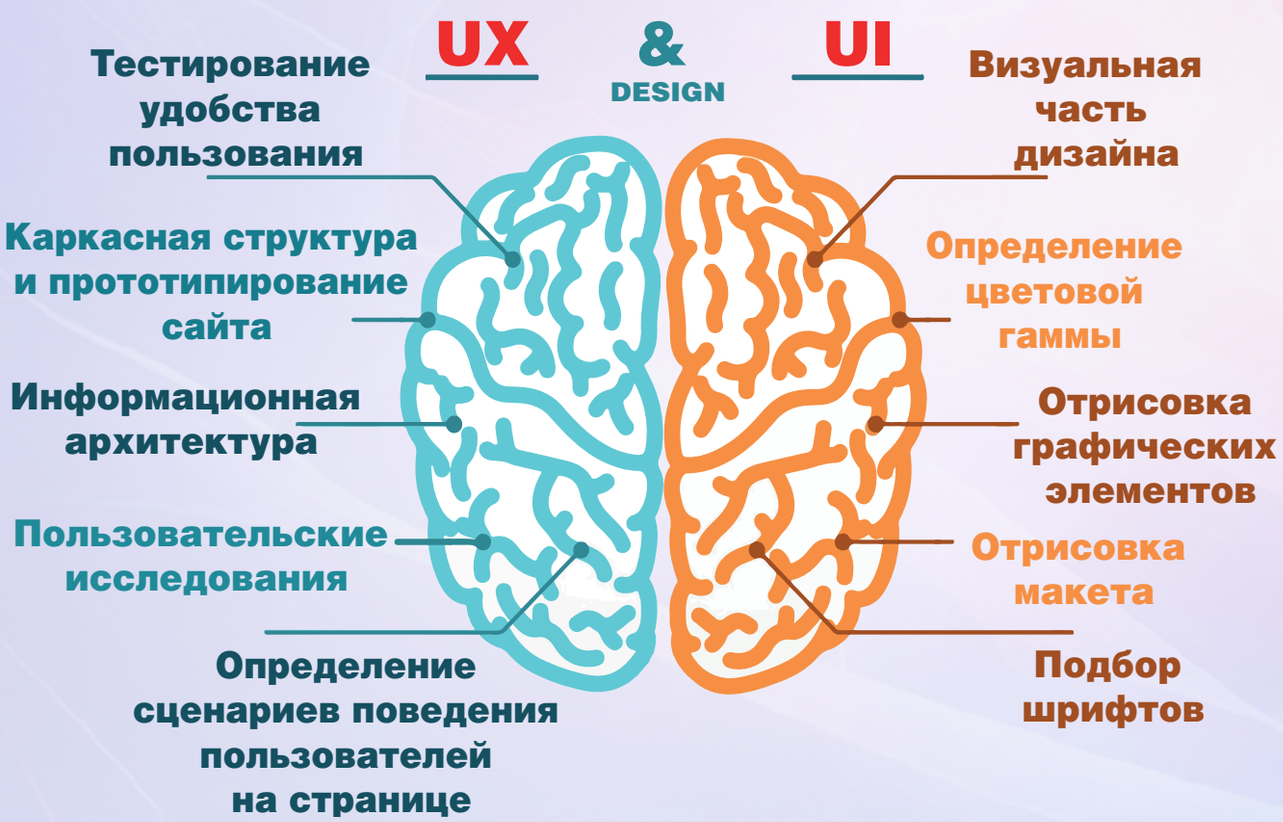
UX/UI ДИЗАЙН

UX *(User Experience, опыт пользователя)*. То, какой опыт/впечатление получает юзер от работы с интерфейсом. Удаётся ли ему достичь цели, и насколько просто/сложно это сделать.

UI *(User Interface, пользовательский интерфейс)*. То, как выглядит интерфейс и то, какие физические характеристики приобретает. Определяет, например, какого цвета будет продукт, какого размера сделать кнопки, чтобы по ним было удобно попадать пальцем, выбор читабельного шрифта и многое другое.

UI тесно связан с frontend. Фронтенд-разработчик «оживляет» нарисованный дизайнером макет с помощью кода.

UX/UI дизайн – проектирование интерфейсов, в которых удобство использования так же важно, как и внешний вид.



Прямая обязанность UX/UI дизайнера – «продать» *товар или услугу через интерфейс.*

*Именно на основе работы UX/UI дизайнера пользователь принимает решение: «**Быть или не быть?**» Нравится или не нравится. Купить или не купить.*

UX дизайнер vs UI дизайнер

UX дизайнер планирует то, как вы будете взаимодействовать с интерфейсом**, и какие шаги вам нужно предпринять, чтобы сделать что-то. **А UI дизайнер придумывает, как каждый из этих шагов будет выглядеть.

USABILITY

Юзабилити – то, как оценивают сайт, мобильное приложение или ПО пользователи с точки зрения удобства работы с ним.

Требования к юзабилити:

- **Интуитивно понятный интерфейс**. Пользователи не должны испытывать трудностей в поиске нужной кнопки/функции.
- **Отсутствуют ошибочные действия.** Пользователь сразу находит нужные кнопки и может на них нажать не промахиваясь.
- **Продукт приносит удовлетворение.**

Элементы пользовательского интерфейса

Юзабилити и дизайн – не одно и то же. Но дизайн является частью юзабилити, именно элементы дизайна (интерфейса) позволяют пользователю взаимодействовать с ПО. Тестировщику для работы необходимо знать названия элементов и понимать их назначение. UI-элементов много (кнопки, чекбоксы, тоглы, поля ввода и тд), но в основном их делят на категории:

- **Input Controls** *(ввод информации в систему)*
- **Navigation Components** *(перемещение по продукту)*
- **Informational Components** (*информация для пользователя*)
- **Containers** (*содержат связанный контент вместе*)

FIGMA

Figma – инструмент, в котором дизайнеры создают макеты приложений или сайтов. Figma полностью вытеснила Photoshop с рынка разработки.

Тестировщику Figma нужна для:

- Предварительного просмотра и анализа прототипа.
- Проверки соответствия макета и вёрстки. То есть для проверки соответсвия кнопок, полей, размеров, шрифтов и прочего.

КЛИКАБЕЛЬНЫЙ vs КЛИКБЕЙТНЫЙ

Кликабельный заголовок *– заголовок (или любые элементы дизайна), который содержит активную ссылку.*

Кликбейтный заголовок *– заголовок, содержащий кричащую информацию, которая привлекает внимание и мотивирует человека перейти по ссылке.*

***ПРИМЕР**: «**10 способов заработать миллион за ночь без вложений!**»*

АДАПТИВНОСТЬ vs ОТЗЫВЧИВОСТЬ

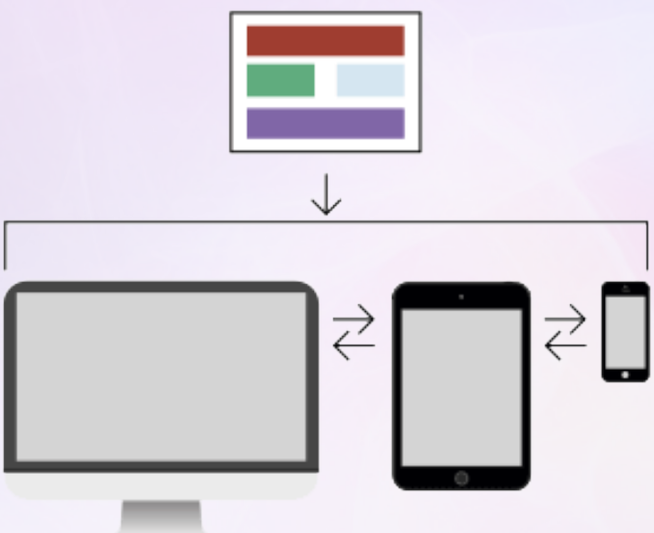
Эти понятия **чаще всего употребляют, как синонимы, или же путают их определения.** Связано это с тем, что адаптивный и отзывчивый дизайн имеют общую цель – позволяют отображать сайт на экранах устройств, различных по разрешению и соотношению сторон, но методы их реализации отличаются. Выбор метода зависит от потребностей и ресурсов проекта. Иногда приходится даже комбинировать оба подхода для оптимальных результатов.

ОТЗЫВЧИВЫЙ ДИЗАЙН

(Responsive Design)

Единый дизайн, который может изменяться. Использует гибкую сетку и медиазапросы CSS для того, чтобы элементы сайта адаптировались к разным размерам экранов.

Сайт с отзывчивым дизайном обычно имеет один *URL* для всех устройств.

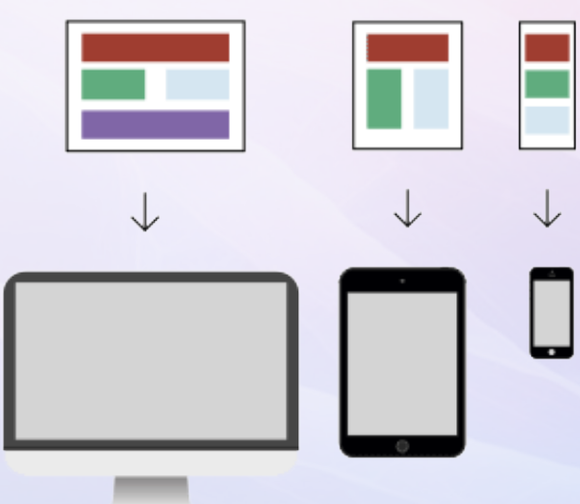


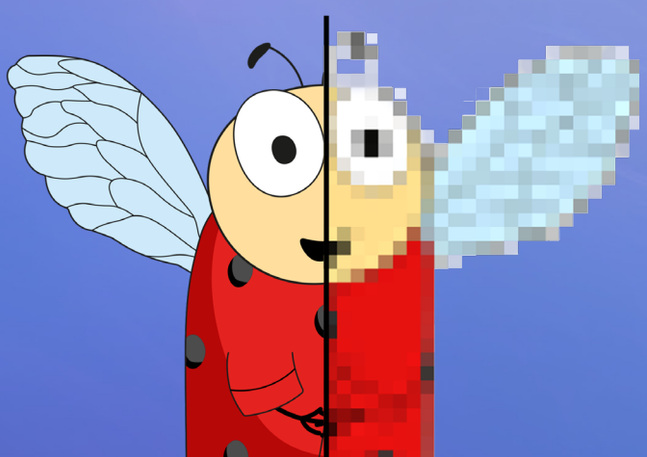
АДАПТИВНЫЙ ДИЗАЙН

(Adaptive Design, AWD)

Предусматривает **создание разных версий сайта или приложения для разных категорий устройств** (например, мобильные, планшеты, ПК).

Каждая версия может быть оптимизирована под конкретные требования устройства, тем самым предоставляя более широкие возможности для кастомизации и создания уникальных интерфейсов.





Высокое разрешение | Низкое разрешение

***Разрешение экрана** – размер, выраженный в пикселях двумя числами – количество пикселей по ширине и по высоте экрана. От разрешения зависит качество картинки.*

***Размер экрана** – его диагональ, измеряется в дюймах.*

ГЕЙМИФИКАЦИЯ

Геймификация *(Gamification)* – это внедрение игровых механик и стратегий в неигровой контекст.

Это может быть как система простых наград и достижений, так и сложные сценарии виртуальных миров, созданные для стимулирования вовлеченности, удовлетворения и мотивации пользователей.

Цель геймификации – вовлечь или привлечь пользователя (покупателя, сотрудника, ученика и др.) в процесс, стимулируя его активность, так же, как это делают компьютерные игры.

Цели геймификации в разных сферах:

Внутренняя геймификация для сотрудников. Цель – сплотить коллектив для повышения продуктивности. Это повысит качество производимого продукта и увеличит объём продаж, поможет улучшить обслуживание клиентов.

Внешняя геймификация для клиентов и пользователей. Цель – повысить узнаваемость продукта, удержать имеющихся и привлечь новых клиентов, повысить лояльность, увеличить доход.

Меняющая поведение геймификация. Используется, например, в обучении или здравоохранении. Цель – сформировать новые привычки для здорового образа жизни или повышение мотивации и результативности в обучении чему-либо.

Пример геймификации:

Duolingo *(сервис для изучения языков). Предлагает копить внутреннюю валюту, чтобы обменивать ее на бонусы, начисляет баллы и даёт значки достижений за выполнение заданий.*

Вовлечённость *– это уровень взаимодействия пользователя с приложением или контентом.*

Вовлеченность на примере VK:

Низкая вовлечённость*: пользователь иногда заходит в VK, пассивно просматривает – нет ли новых сообщений и постов. Выходит не взаимодействуя.*

Высокая вовлечённость*: пользователь активно публикует контент, репостит и комментирует посты.*

Web

ИНТЕРНЕТ
(межсетевой протокол)

Интернет – глобальная сеть взаимосвязанных умных устройств, которые **общаются,** обмениваясь информацией друг с другом по компьютерным сетям, используя архитектуру *клиент-сервер*. «Общение» подробно описано в моделях сетевого взаимодействия (**OSI** и **TCP/IP**) и происходит с помощью стандартизированных протоколов.

Введя в адресную строку браузера **доменное имя** сайта, например, **vk.com** – браузер (используя **DNS**-сервер) определит его **IP-адрес**, направит запрос на сервер и, получив ответ, отобразит сайт.

- ***Доменное имя**, это буквально имя вашего сайта, но у сайта есть еще URL-адрес, он указывает точное местоположение конкретной страницы на этом сайте.*
- ***DNS** (Domain Name System) – это система, которая связывает доменное имя с его цифровым адресом (**IP-адресом**).*
- ***URL-адрес** главной страницы ВК будет выглядеть так: **https://vk.com/feed**, где **vk.com** – доменное имя.*
- ***IP-адрес** – сетевой адрес (уникальная последовательность чисел). На одном IP-адресе могут быть сотни сайтов, поэтому для подключения к нужному ресурсу нужно доменное имя.*

Модели сетевого взаимодействия OSI и TCP/IP

***Модель OSI** (Open Systems Interconnection) – **эталонная** модель, описывает сетевое взаимодействие в целом.*

***Модель TCP/IP** (Transmission Control Protocol/Internet Protocol) – ориентирована на **практическое применение**, она обеспечивает универсальность и совместимость между различными устройствами и операционными системами. А также надёжность и порядок доставки данных между компьютерами.*

Модель TCP/IP состоит из четырёх уровней

- ***Канальный уровень**. Отвечает за взаимодействие по сетевому оборудованию, например по Ethernet-кабелю или Wi-Fi.*
- ***Межсетевой уровень**. Помогает отдельным сетям общаться друг с другом.*
- ***Транспортный уровень**. Отвечает за передачу данных между устройствами, например, по протоколам TCP и UDP.*
- ***Прикладной уровень**. Помогает приложениям общаться друг с другом с помощью интерфейсов или API. Используются протоколы, например HTTP, SMTP или FTP.*

КЛИЕНТ-СЕРВЕРНАЯ АРХИТЕКТУРА

– это распределенная система, в которой одни устройства (***клиенты***) обращаются к другим (***серверам***) для получения необходимой информации или выполнения каких-либо задач.

Другими словами – это **модель взаимодействия компонентов системы**, где клиент (потребитель услуг) отправляет запросы серверу (поставщику услуг), который в свою очередь обрабатывает эти запросы и возвращает результаты обратно клиенту.

Модель клиент-сервер является основой для создания веб-продуктов, но применима также к desktop и mobile ПО.

Компоненты клиент-серверной архитектуры:

Клиент – это устройство или приложение на стороне пользователя, которое **отсылает запросы серверу** и **принимает** от него **ответы**.

*Различают **тонкий** и **толстый клиент**, отличие между ними в том, где физически программа выполняет большую часть вычислений – на компьютере пользователя (**толстый клиент**) или на сервере (**тонкий клиент**).*

Сервер – это специальный компьютер, который предоставляет запрашиваемую информацию или услуги **клиентам**. Серверы могут выполнять такие функции, как хранение данных, обработка запросов, вычисления и др.

Протоколы обмена данными – правила, определяющие, как **клиенты** и **серверы** обмениваются информацией. Например: HTTP, протоколы отправки и получения почты (SMTP, POP3, IMAP), протокол передачи файлов FTP.

Базы данных – хранилища информации, которые используются на серверной стороне. *Подробнее здесь*.

Сеть – обеспечивает обмен информацией между **клиентом** и **сервером** по протоколам обмена данными. Сети могут быть локальными (LAN) и глобальными (WAN).

Система безопасности – обеспечивает защиту данных, передаваемых между **клиентами** и **серверами** (шифрование данных, *аутентификацию* и авторизацию клиентов, защиту от несанкционированного доступа и другие меры безопасности).

Хранение и обработка данных – это компоненты, включающие в себя серверные операционные системы, системы управления базами данных (*СУБД*), серверы приложений и др.

FRONTEND и BACKEND

Составляют целостную структуру любого веб-продукта

Фронтенд

(Клиентский уровень)

Часть ПО, которую видит юзер и с которой может взаимодействать (картинки, меню, анимация, кнопки, чекбоксы и др. интерактивные элементы). То есть – это разработка пользовательских функций и *интерфейса*.

Фронт работает на мощности ПК или смартфона. Создаётся с помощью *HTML*, *CSS* и *JavaScript*.

Бэкенд

(Серверный уровень)

Логика работы сайта, скрытая от пользователя. Включает:

- **Сервер**. *Подробнее здесь*.
- **База данных (БД)**. *Подробнее здесь*.
- **Бизнес-логика**. Правила, которые определяют поведение веб-приложения. Включает алгоритмы обработки данных, авторизацию, аутентификацию и другие бизнес-процессы.

ПРОЦЕСС ДОСТУПА К РЕСУРСАМ

Идентификация. Процесс определения пользователя в системе, например, путем ввода им логина или электронной почты.

Аутентификация. Проверка подлинности личности юзера для доступа к учетной записи и информации в ней. Осуществляется, например, посредством ввода пароля.

Авторизация. После аутентификации система определяет набор привилегий и прав, доступных конкретному пользователю.

Токены аутентификации JWT и OAuth

JWT (JSON Web Tokens) – строка символов с данными пользователя (ID, имя, уровень доступа). После первой аутентификации сервис предоставляет пользователю два токена – короткоживущий **access token** (с его помощью осуществляется авторизация) и **refresh token** (нужен для получения свежего access token). При последующих обращениях access token передаётся приложению в *запросах* от клиента. Получив токен, приложение извлекает сведения о пользователе и на их основе авторизует его. Если access token будет перехвачен злоумышленниками, это позволит им получить доступ к приложению лишь на ограниченный срок, в течение которого токен будет валидным (несколько минут).

OAuth предоставляет токены доступа к сторонним сервисам, например авторизация через соцсеть.

КЛИЕНТ-СЕРВЕРНОЕ ВЗАИМОДЕЙСТВИЕ

Архитектура «клиент-сервер» определяет общие принципы организации взаимодействия в сети, но **практические реализации такой архитектуры могут различаться**:

Клиент

Клиент отправляет HTTP запрос к API, а в ответ получает HTTP ответ и обрабатывает его.

Сервер приложения

Клиент общается с сервером через API. Сервер принимает HTTP запрос, обрабатывает его и выдает клиенту HTTP ответ.

Сервер БД + База данных

Сервер базы данных (БД) принимает SQL запрос от сервера и выдает ответ обратно.

- **Двухзвенная** (двухуровневая). Распределение трёх базовых компонентов (клиент, сервер, база данных) между двумя узлами – клиентом и сервером.
- **Трехзвенная** (трехуровневая) – частный случай многоуровневой архитектуры.
- **Многозвенная** (многоуровневая). Суть многоуровневой архитектуры заключается в том, что запрос клиента обрабатывается сразу несколькими серверами.

Полезные понятия

Сервер приложений – физическая или виртуальная машина, на которой размещается бэкенд приложения.

Сервер базы данных – серверная платформа с установленным специализированным программным обеспечением (СУБД) для управления базой данных (БД).

СУБД – система управления базами данных. СУБД получает команды (запросы), что сделать с БД, и выполняет их.

Локальные БД. Все элементы системы и базы данных находятся на одном сервере, как правило.

Распределённые БД. Набор логически взаимосвязанных баз данных, физически распределенных на разных серверах.

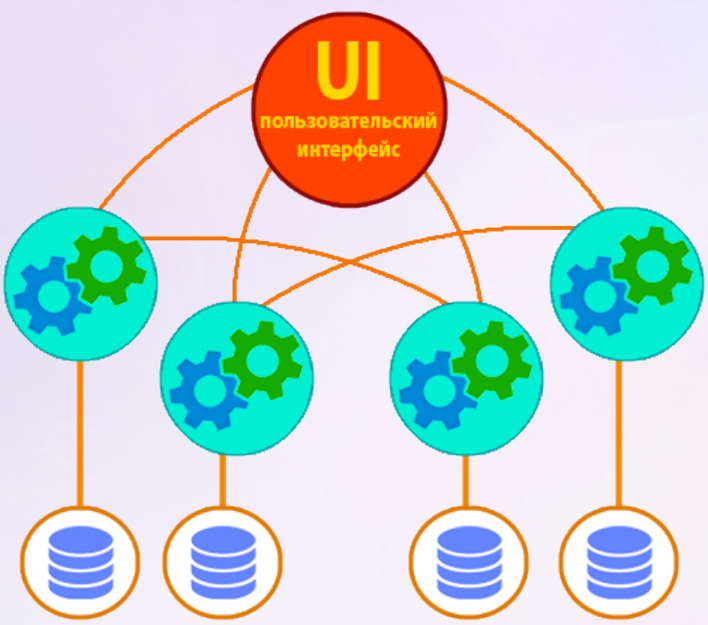
МОНОЛИТ vs МИКРОСЕРВИСЫ

Монолит и **Миросервисы** – два разных, но самых актуальных на сегодня подхода к построению **архитектуры ПО**.

Архитектура ПО – верхнеуровневая структура, на базе которой создается приложение. Содержит описание системы в целом, отдельных компонентов и связей между ними.

Монолит

Микросервисы



Монолитная архитектура. Традиционно, это подход к построению ПО, где все компоненты (пользовательский интерфейс, бизнес-логика, база данных), находятся внутри одной кодовой базы. Но в веб-разработке монолит разделяют на *фронтенд* и *бэкенд* получая классический вариант трехзвенной клиент-серверной архитектуры, это тоже будет считаться монолитом.

Микросервисная архитектура. Микросервисы можно рассматривать как следующую стадию развития клиент-серверной архитектуры, где серверная часть приложения разбивается на набор небольших, независимо развёртываемых компонентов, которые фокусируются на определённых задачах.

База данных создается для каждого микросервиса *(Database Per Service)*, так как общая БД, в микросервисной архитектуре, это моветон. Невозможно независимо тестировать и масштабировать два сервиса, использующие одну базу данных.

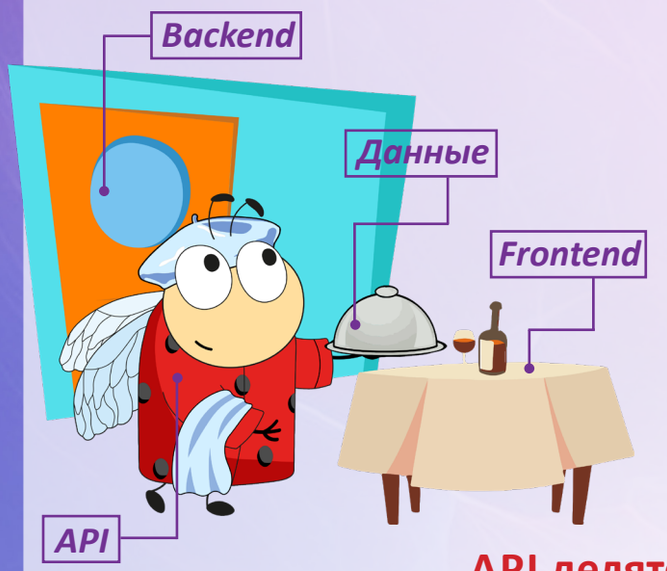
Микросервисы взаимодействуют между собой с помощью API и брокера сообщений (выступает в качестве центрального роутера для координации отдельных сервисов).

Подход, когда разные сервисы имеют единую БД является разновидностью сервис-ориентированной архитектуры (SOA), в рамках данной Шпаргалки она не рассматривается.

API

Application Programming Interface

API *(программный интерфейс приложения)* – описание способов и правил, по которым программы или компоненты программ взаимодействуют (обмениваются данными) между собой. API позволяет одной программе использовать функционал другой, то есть это возможность внедрять готовые наработки в продукт.



*Для взаимодействия приложений (или компонентов) используется **API endpoint**, он представляет собой URL (или часть URL). По сути, **эндпоинт – это адресат**, к которому можно обратиться для выполнения нужного действия или получения данных.*

API делятся на группы:

Открытые API (общедоступные, внешние API). Готовые решения доступные для сторонних разработчиков ПО, например: яндекс-карты, модуль прогноза погоды, регистрация через соц. сети и др.

*Для интеграции стороннего сервиса нужен ключ. **API Key – это уникальный идентификатор**, представляет собой длинную строку букв и цифр. Создается один раз и не имеет срока действия, если его не отозвать. API-ключ, часто называют токеном, однако это разные понятия. Ключ используется для связи между серверами, предоставляет приложениям доступ к API (например к API погоды). Токен используется для аутентификации пользователей.*

Партнерские API. Доступны только партнёрам, поставщикам услуг и клиентам для обмена информацией между компаниями.

Внутренние API (частные API). Разрабатываются под нужды конкретного сайта или приложения. Предназначены для облегчения связи и обмена данными между внутренними компонентами системы или различными приложениями внутри одной организации.

Составные API. Это система API, которая объединяет два или больше разных API, чтобы решить поставленную задачу (например объединять запросы и получать единый ответ с разных серверов).

При разработке API используют разные технологии обмена информацией, такие как: REST, SOAP и GraphQL. Выбор технологии зависит от конкретной задачи и требований к API.

REST API

REST API *(Representational State Transfer, передача самоописываемого состояния)* – архитектурный стиль содержащий в себе набор принципов и ограничений, которыми руководствуются при разработке API. Приложения, которые **строго следуют правилам REST**, называются **RESTful API**.

REST использует стандартные *HTTP-методы* для выполнения операций с ресурсами и передачей данных в формате *JSON*, *XML* и другими (в REST нет фиксированного формата).

Набор правил и принципов REST:

- **Клиент-серверная архитектура** (Client-Server). Отделение клиента от сервера – код запросов остаётся на стороне клиента, а код для доступа к данным – на стороне сервера.
- **Отсутствие состояния** (Statelessness). Сервер не хранит данные о предыдущих взаимодействиях с клиентом. Запрос содержит всю необходимую информацию для его обработки.
- **Кэшируемость** (Cacheability). В запросе к серверу должна содержаться информация о кэшировании данных в буфере. Ответы REST API должны указывать, могут ли они быть кэшированы клиентом. *Подробнее про кеш здесь*.
- **Единообразие интерфейса** (Uniform Interface). Все данные должны запрашиваться с одного URL, используя методы HTTP.
- **Слои** (Layered System). REST API состоит из серверов, шлюзов, балансировщиков нагрузки и т.д. Клиент может общаться напрямую с сервером, а может с промежуточным узлом.
- **Код по требованию** (Code on Demand). Необязательный принцип. Сервер может отправлять код, который будет выполняться уже на стороне клиента.

SOAP API

SOAP API *(Simple Object Access Protocol)* – протокол для обмена структурированной информацией в веб-сервисах с использованием формата XML.

SOAP обладает высокой расширяемостью и позволяет осуществлять обмен данными через несколько различных транспортных протоколов, включая HTTP. API SOAP предлагает встроенную обработку ошибок и поддержку обмена сообщениями с отслеживанием состояния (либо без). Благодаря этим особенностям SOAP используется в ряде корпоративных сервисов, особенно связанных с финансами. Однако, из-за тяжеловесности XML-формата и правил формирования XML-сообщений SOAP теряет популярность.

GRAPHQL

GraphQL – язык запросов и обработки данных с открытым исходным кодом для API, а также серверная среда выполнения запросов (GraphQL Server). ***GraphQL Server*** *выступает прокладкой между пользователем приложения и веб-сервером с базой данных.*

GraphQL предоставляет полное и понятное описание данных в API, позволяя клиентам запрашивать именно то, что им нужно, и ничего лишнего. При этом с помощью GraphQL можно получить данные из любого хранилища. Запрос отправляется в среду выполнения, а в ответ приходят уже готовые данные в формате JSON.

Язык был разработан в 2012 году в Facebook в качестве альтернативы **REST API**. GraphQL, в отличие от REST API, позволяет при помощи одного запроса получить сразу все необходимые данные, даже если они находятся на разных серверах. В REST API для этого понадобилось бы отправлять отдельный запрос на каждый сервер.

Форматы передачи данных

Данные, передаваемые через HTTP, технически могут быть любого формата, но чаще всего это форматы **JSON** и **XML**.

XML *используется в SOAP (всегда) и реже в REST-запросах.*
JSON *используется в REST.*

JSON *(JavaScript Object Notation)* – простой и удобный формат данных в виде текста, который стал повсеместным при разработке приложений и взаимодействии между клиентом и сервером. Основа JSON – это синтаксис объектов *JavaScript*. Данные в формате JSON хранятся как пары «ключ:значение», разделенные запятыми. С помощью JSON различные типы данных могут быть сериализованы и представлены строками.

Сериализация *– процесс преобразования структуры данных в удобный для передачи формат, например, JSON.*
То есть, объект сериализуется на одной стороне, затем передаётся по сети и десериализуется на другой стороне для дальнейшего использования.

XML *(**E**X**t**ensible **M**arkup **L**anguage)* – язык разметки для создания логической структуры данных, их хранения и передачи в виде, удобном и для компьютера, и для человека. XML позволяет описывать документы с помощью тегов.

HTTP ПРОТОКОЛ

HTTP протокол *(HyperText Transfer Protocol, протокол передачи гипертекста)* – изначально предназначался для передачи гипертекстовых документов в формате HTML, сейчас это универсальное средство взаимодействия для обмена информацией между клиентами и серверами.

HTTP используется также в качестве «транспорта» для других протоколов прикладного уровня, например SOAP и XML-RPC.

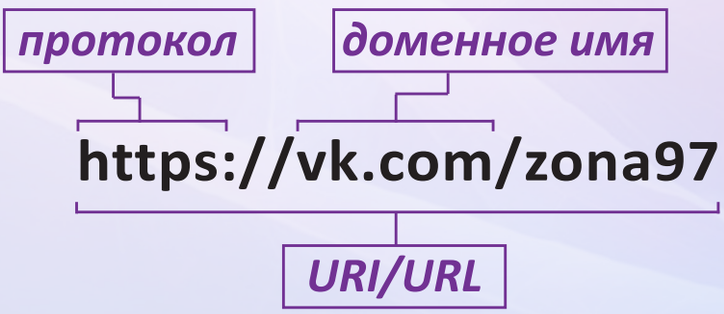
Благодаря протоколу HTTP обеспечивается работа Всемирной паутины. Для идентификации ресурсов **HTTP** использует **URI**.

URI (Uniform Resource Identifier) – последовательность символов, которая отличает (идентифицирует) один ресурс от другого. Если просто – это строка, указывающая на ресурс: HTML-страница, CSS-файл, изображение и т.д.

URI может быть либо указателем ресурса (**URL**), **либо именем ресурса** (**URN**), **либо одновременно обоими**. То есть URN и URL являются URI, но обратное неверно.

URL *(Uniform Resource Locator) – адрес ресурса в сети, определяет местонахождение и способ обращения к нему.*

URN *(Unifrom Resource Name) – имя ресурса, определяет только название, без способа подключения к нему.*



Термины URL и URN технически устарели и вместо них следует использовать URI. Но на сегодня в народе URL – это наиболее часто используемый термин при описании ссылок. URN же активно используется для обеспечения уникальности и постоянства идентификаторов в различных сферах, например, в цифровых библиотеках для идентификации книг по ISBN (*уникальный номер книжного издания*).

HTTP vs HTTPS

HTTP *– **это открытый протокол** передачи данных между браузером и сервером.* ***HTTPS*** *– тот же HTTP, но с добавленными методами шифрования данных и проверки безопасности, то есть **защищенный протокол** передачи данных.*

HTTP ЗАПРОСЫ и HTTP ОТВЕТЫ

Структурированный и эффективный обмен данными обеспечивается HTTP протоколом посредством **запросов** и **ответов**.

HTTP ЗАПРОСЫ *(HTTP Requests)* – это сообщения, отправляемые *клиентом* на *сервер*, чтобы инициировать выполнение определенных действий.

Запрос, отправленный клиентом на сервер, получает свой уникальный идентификатор, с которым затем будет связан лог об его успешной или неудачной обработке.

HTTP ОТВЕТЫ *(HTTP Responses)* – это сообщения, которые сервер отправляет клиенту в ответ на запрос.

СТРУКТУРА HTTP ЗАПРОСА

- ***Стартовая строка*** *(Start Line/Request Line). В строке запроса указывается* ***метод передачи****, версия протокола HTTP и URI (URL), к которому должен обратиться сервер.*
  - **Метод HTTP** *(HTTP Method) –* ***команда серверу, указывает на операцию над ресурсом.*** *Подробнее здесь.*
- ***Заголовки*** *(Request Headers). Заголовки содержат дополнительную информацию (метаданные) о запросе, либо описывают содержимое тела сообщения.*
- ***Пустая строка*** *(Empty Line) сообщает, что все метаданные для конкретного запроса (или ответа) были отправлены.*
- ***Тело запроса*** *(Request Body) – завершающая часть HTTP-запроса, поддерживается не всеми методами, содержит данные для внесения изменений в БД. Отсутствует, например у* ***GET****,* ***HEAD*** *и* ***DELETE****. А методы* ***POST*** *и* ***PUT*** *могут содержать данные для отправки на сервер.*

Что такое CRUD?

CRUD *– это сокращение от* ***Create*** *(создание),* ***Read*** *(чтение),* ***Update*** *(модификация) и* ***Delete*** *(удаление).*

CRUD представляет собой четыре основные операции, которые совершаются с любой информацией, в любых системах (на сайтах, в приложениях, базах данных). Обычно для каждой операции используют свой HTTP-метод:

C (create) – POST
R (read) – GET
U (update) – PATCH/PUT
D (delete) – DELETE

HTTP-МЕТОДЫ

Это команды определяющие тип операции, которую клиент хочет выполнить с ресурсами на сервере. **Основные методы:**

- **GET.** Запрос содержимого ресурса (например, загрузка HTML-страницы) в режиме «только чтение». **Параметры запроса передаются только в URL** (URI) и ограничены по количеству символов используемым браузером.
- **HEAD.** Метод для получения заголовков ресурса. Запрашивает ресурс так же, как и метод GET, но без возврата тела ответа.
- **POST.** Создает новый ресурс (или загружает файлы на сервер) по данным, которые указаны в теле запроса.
- **PUT.** Создает новый ресурс или полностью заменяет существующий ресурс данными, которые указаны в теле запроса.
- **PATCH.** Используется для частичного изменения ресурса.
- **DELETE.** Удаляет указанный ресурс.

Методы можно классифицировать:

- **Идемпотентный метод**. Может вызываться несколько раз с одним и тем же вводом и давать один и тот же результат.
- **Безопасный метод**. Может вызываться несколько раз с одним и тем же вводом, и он не изменяет ресурс на стороне сервера. Этот метод не должен иметь побочных эффектов, за исключением безобидных (таких, как *логирование* и *кеширование*) Все безопасные методы являются идемпотентными, но не наоборот.

Безопасность, в данном контексте, это не про перехват информации злоумышленниками, а про то, могут ли запросы своими методами изменить (поломать) ресурс.

Канонически, исходя из понятий **идемпотентности** и **безопасности**, принято разделять методы на три группы:

1. **GET**, **HEAD**, **OPTIONS** – *безопасны и идемпотентны.*
2. **PUT**, **DELETE** – *не безопасны, но идемпотентны.*
3. **POST**, **PATCH** – *не безопасны и не идемпотентны.*

*Однажды вы столкнетесь с противоречиями **между тем, как принято** и положено, **и тем, как на это** по факту **положено**. Например, метод **GET**, в теории, создан для получения данных, но он может успешно реализовывать удаление запрошенного ресурса. С другой стороны, **HTTP-методы** (и соответствующие им действия) – **это соглашения, которые желательно, но не обязательно соблюдать**.*

СТРУКТУРА HTTP ОТВЕТА

- ***Стартовая строка*** *ответа HTTP, называемая строкой статуса (**Status Line**), содержит в себе информацию:*
  - ***Версию протокола*** *– HTTP/1.1, HTTP/2, HTTP/3.*
  - ***Код состояния*** *(Status Code). Информирует о результате запроса, состоит он из трех цифр.*
  - ***Пояснение*** *(Status Text). Краткое описание кода состояния.*
- ***Заголовки*** *(Response Headers). Служебная информация.*
- ***Пустая строка*** *как разделитель между headers и body.*
- ***Тело сообщения*** *(Response Body). Данные, которые посылает сервер в ответ на запрос. Тело есть не всегда, у ответов с кодом состояния **201** или **204** оно обычно отсутствует.*

Группы кодов состояний HTTP ответа, с примерами:

1** **Информационные сообщения**
102 Processing *– запрос получен сервером, но его обработка ещё не завершена.*

2** **Сообщения об УСПЕХЕ**
200 OK *– запрос принят и корректно обработан сервером.*
201 Created *– запрос корректно обработан, ресурс создан.*

3** **Редиректы.** Говорят о перемещении ресурса в другое место.
301 Moved Permanently *– перемещено навсегда.*
302 Found *– перемещено временно.*

4** **Сообщение об ошибке на сервере по вине КЛИЕНТА**
400 Bad Request *– неверный запрос. Возможно синтаксическая ошибка, например, опечатка.*
401 Unauthorized *– ресурс доступен только аутентифицированным пользователям.*
403 Forbidden *– у клиента нет прав для доступа к ресурсу.*
404 Not Found *– запрашиваемый ресурс не найден.*
405 Method Not Allowed *– HTTP-метод нельзя применить к текущему ресурсу.*

5** **Сообщение об ошибке на сервере по вине СЕРВЕРА**
500 Internal Server Error *– непредвиденная ошибка на сервере.*
501 Not Implemented *– HTTP-метод не поддерживается.*
503 Service Unavailable *– сервер не готов обработать запрос (например, из-за технического обслуживания).*
505 HTTP Version Not Supported *– версия HTTP, используемая в запросе, не поддерживается сервером.*

БАЗЫ ДАННЫХ

База данных (БД) – это организованное хранилище цифровой информации. База данных позволяет не только хранить, но и изменять, извлекать и удалять информацию. БД содержит все данные, касающиеся различных компонентов приложения.

Существует множество видов баз данных, но основные это **реляционные** и *нереляционные*.

РЕЛЯЦИОННАЯ БД

Реляционная БД имеет **табличную** организацию хранения данных с предопределенными связями между ними. В **столбце** хранится определенный тип данных, в каждой **ячейке** – значение атрибута. **Строка** представляет собой набор связанных значений, относящихся к одной сущности.

Клиент

Id _кл	Фамилия	Имя	Отчество
15	Иванов	Иван	Иванович
16	Петров	Петр	Петрович
17	Николаев	Николай	Николаевич

Товар

Id_тов	Название
1	Шкаф
2	Стул
3	Стол

Ключи

Заказ

Id_зак	Клиент	Товар	Дата	Количество
1	15	1	15.09.2003	1
2	17	1	17.09.2003	2
3	15	2	20.09.2003	12

Ключ СУБД *– это атрибут или набор атрибутов, который помогает идентифицировать строку в таблице.*

- ***Первичный ключ*** *(Primary Key) – поле, каждый элемент которого однозначно определяет запись таблицы. Используется для идентификации объекта.*
- ***Ключ внешний*** *(Foreign Key) – это столбец или сочетание столбцов, применяется для принудительного установления связи между таблицами одной БД.*

Реляционные БД используют структурированный язык запросов (SQL) для управления данными. *Подробнее здесь.*

Нормализация *(декомпозиция)* ***БД*** *– метод проектирования реляционных БД, направленный на создание системы с четким представлением информации и взаимосвязей, без избыточности данных. Под избыточностью данных понимается хранение одной и той же информации в базе сразу в нескольких местах.*

Балансировщики нагрузки *– сервисы для распределения запросов по нескольким серверам.*

НЕРЕЛЯЦИОННЫЕ БД (NoSQL)

Семейство баз данных, в которых модель хранения информации оптимизирована под конкретные нужды ПО. NoSQL БД имеют распределённую архитектуру, обеспечивающую высокую производительность и масштабируемость.

Типы баз данных NoSQL

- **Ключ-значение.** Информация хранится в таблице, содержащей ключи со значениями, которые могут быть разных типов: числа, строки или более сложные объекты. *Пример СУБД – Redis.*
- **Документоориентированная БД** (Хранилище документов). Данные хранятся в виде документов JSON. Каждый документ – самостоятельный объект, в нём есть все данные для описания одной сущности, эти документы группируются в коллекции. *Пример СУБД – MongoDB, Couchbase.*
- **Графовые** (Сетевые) **БД.** Данные представляются в виде **узлов** (объектов) и **рёбер** (определяющих связи между объектами), узлы могут иметь любое количество связанных с ними **свойств**. Графовые БД применяются, например, в социальных сетях. *Пример СУБД – Neo4j.*
- **Столбцовые БД.** Напоминают реляционную БД, также хранят данные, используя строки и столбцы, но с иной связью между элементами – данные каждого столбца хранятся отдельно (независимо) от других столбцов. *Пример СУБД – Hbase.*

Масштабирование БД

Партицирование *– разделение таблицы на несколько таблиц по какому-либо принципу, чаще по функциональности.*

Шардирование *– это принцип проектирования баз данных, при котором данные разбиваются на независимые части и размещаются на разных серверах для распределения нагрузки. Деление бывает* ***горизонтальным*** *и* ***вертикальным****.*

Часто ***шардирование путают с партицированием****, но это разные понятия, которые могут быть реализованы в пределах одной БД.* ***При партицировании база данных остаётся на одном сервере, а при шардировании распределяется на разные серверы****.*

Репликация БД *– процесс синхронизации информации в нескольких копиях (репликах) одной БД на разных серверах. Выделяют* ***асинхронную*** *(после записи в оригинал информация дублируется в реплику) и* ***синхронную*** *репликацию (данные дублируются в реальном времени).*

S Q L
(Structured Query Language)

SQL – язык структурированных запросов, позволяющий хранить, изменять и извлекать данные из *реляционных БД*.

В императивных *ЯП*, таких, как Python или *JavaScript*, пишется последовательность действий. Но **SQL – декларативный язык, на нём нужно прописывать то, что хотим получить** (обозначить результат). Программа, выполняющая запрос (СУБД), сама получает и возвращает нужные данные, т.е. выполняет заданную команду. Стоит отметить, что на языке SQL нельзя написать программу, он предназначен только для взаимодействия с БД.

***СУБД** – системы управления базами данных. Разные системы управления БД имеют собственные расширения языка SQL – **диалекты**, но базовый синтаксис у всех одинаковый, это удобно при переходе от одной СУБД к другой. Примеры СУБД: MySQL, PostgreSQL, SQLite, Microsoft SQL Server.*

SQL-запросы

SQL-запросы – это набор команд для работы с реляционными базами данных. **С помощью SQL-запросов можно**:

- **Создать базу данных.** Описать её структуру и настроить правила расположения информации.
- **Проводить операции** с данными в БД. Добавить, обновить, извлечь или удалить данные.
- **Управлять доступом** к объектам базы данных. Предоставить или отозвать права доступа.
- **Управлять транзакциями**. Обеспечить целостность и согласованность данных при выполнении нескольких операций, позволяя откатить изменения.

***SQL-транзакция** (Transact-SQL)*

Последовательность различных SQL-запросов, выполняющихся как одно целое, которые должны завершится успешно.

Свойства транзакций (ACID):

***Атомарность** (Atomicity). Все команды в транзакции либо полностью выполняются, либо ничего не выполняется.*

***Согласованность** (Consistency). После успешной транзакции база данных не будет содержать несогласованных данных.*

***Изолированность** (Isolation). Отделяет все параллельные транзакции друг от друга.*

***Долговечность** (Durability). Сохраняемость данных.*

ТИПЫ SQL ЗАПРОСОВ

DDL (Data Definition Language) – определение данных.

- **CREATE** – создать. Например, **CREATE TABLE** – создать таблицу или **CREATE USER** – создать пользователя.
- **ALTER** – модифицировать. Этот запрос используется при внесении изменений в саму БД или в ее часть.
- **DROP** – удалить. Запрос относится к БД и её частям.

DML (Data Manipulation Language) – изменение данных.

- **SELECT** – выборка данных.
- **INSERT** – вставка новых данных.
- **UPDATE** – обновление данных.
- **DELETE** – удаление данных.
- **MERGE** – слияние данных.

DCL (Data Control Language) – управление данными.

- **GRANT** – разрешение пользователю на проведение определенных операций с БД или данными.
- **REVOKE** – отзыв выданного разрешения.
- **DENY** – установка запрета, имеет приоритет над разрешением.

TCL (Transaction Control Language) – управление транзакциями.

- **BEGIN/COMMIT** – команда, которая применяется для объявления транзакции.
- **ROLLBACK** – откатывает текущую транзакцию.

СТРУКТУРА SQL-ЗАПРОСОВ

- *Поля, значения которых требуется вернуть.*
- *Названия таблиц, из которых необходимо извлечь данные.*
- *Связи между таблицами.*
- *Условия выборки.*
- *Вспомогательные критерии отбора (ограничения, способы представления информации, тип сортировки).*

***НАПРИМЕР**, для выборки по клиентам, приносящим интернет-магазину наибольшую прибыль, для работы с БД фирмы строится запрос, имеющий вид:*

> ***SELECT col1, col2, col3** (перечисление колонок, которые нужно отобразить) **from table** (имя таблицы) **where** (указание на последующий фильтр) **clause** (критерий отбора).*

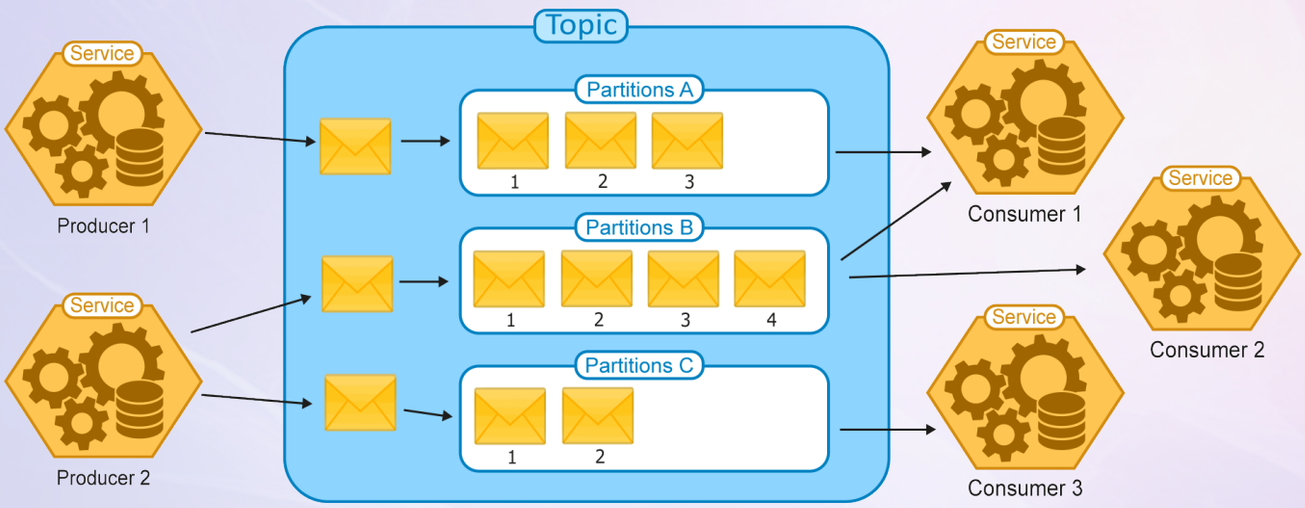
БРОКЕРЫ СООБЩЕНИЙ

В микросервисной архитектуре БД создается для каждого сервиса, при этом – **сервис может получать данные другого сервиса только через API посредством сетевых запросов или сообщений**, без прямого подключения к его БД.

Брокер сообщений – это ПО, которое служит посредником для пересылки сообщений между различными компонентами или приложениями в распределённой системе.

Сообщения могут быть, как текстовыми или бинарными данными, так и графическими изображениями.

В работе брокера используются две основные сущности: Producer (Publisher/отправитель/издатель) **и Consumer** (потребитель/подписчик). Микросервис может быть и продюсером, и консьюмером в зависимости от контекста работы с брокером.



- **Producer** отправляет сообщение брокеру, указывая **топик,** к которому относятся эти данные.

  Брокер принимает сообщение, размещая его в соответствующем топике, анализирует, какие консьюмеры подписаны на данную тему и пересылает им это сообщение.

- **Consumer** подписываются на **топики** и обрабатывают поступающие сообщения, они могут читать данные из одного или нескольких **разделов**.

- **Топик** (Topic/тема). Логическая категория, которая представляет собой поток сообщений, объединенных по некоторому критерию. **Каждый топик разделен на несколько разделов**.
- **Разделы** (Partitions) служат для физического распределения данных внутри топика. Это позволяет параллельно обрабатывать сообщения и повышает производительность системы.
- **Оффсет** (Offset). Каждое сообщение в теме имеет уникальный числовой идентификатор, называемый оффсетом. Он указывает позицию сообщения внутри раздела.

ЛОГИРОВАНИЕ

Лог (Log) – это файл, содержащий хронологическую запись наиболее значимой информации о работе сервера или любой другой программы. Например, в какой именно момент, какой пользователь спровоцировал то или иное событие, какие ошибки возникли и т.д.

Упрощенно процесс логирования выглядит так:

- Клиент отправляет запрос на сервер
- Запрос получает уникальный идентификатор (ID)
- По результатам запроса сохраняется лог об успешной обработке запроса или, если произошла ошибка, сервер должен обработать её и записать следующие данные: ID запроса, все заголовки, тело запроса, параметры запроса, отметку времени и информацию об ошибке.

***С помощью логов можно**: найти причины сбоя и локализовать ошибки, определить вид взлома (возможно найти хулигана), проверить правильность работы сервиса и др.*

Типы логов

- **Логи ошибок** (logs/error.log). Хранят запросы, которые завершились с ошибкой.
- **Логи доступа** (logs/access.log). Хранят информацию обо всех запросах, которые были отправлены на сервер.

*Одним из самых распространенных инструментов просмотра логов бэка и работы с ними, является **Kibana**.*

Уровни логирования:

FATAL – тотальный сбой работоспособности, когда нет доступа к базе данных или сети.

ERROR – типичные ошибки, которые могут привести к некорректной работе приложения.

WARN – предупреждения о потенциальных проблемах, не являющихся критическими ошибками.

DEBUG – подробности о ходе выполнения приложения, обращение к базе данных, старт/пауза сервиса, успешная обработка записи и пр.

TRACE – пошаговые записи процесса, полезен при локализации ошибки.

INFO – общая информация о ходе работы приложения и важных событиях.

HTML | CSS | JAVASCRIPT

Клиентская часть сайта (*фронтенд*) выполняется в браузере. Для реализации фронтенда используются языки текстовой разметки HTML и CSS, а также язык программирования JavaScript. Если очень кратко, то работает это так:

- **HTML** Используется для создания каркаса сайта. Наполняет страницы объектами, которые видит и с которыми может взаимодействовать пользователь.
- **CSS** отвечает за визуальное оформление (цвет, шрифт и т.д.)
- **JavaScript** обеспечивает взаимодействие с пользователем и динамическое обновление страницы.

HTML

Язык HTML (HyperText Markup Language) – язык гипертекстовой разметки. Базовый язык для создания веб-страниц.

*Страница сайта – это обычный текстовый файл с расширением **.html**. Внутри этого файла и хранится текст страницы вместе с тегами.*

HTML состоит из элементов, тегов, атрибутов и значений атрибутов.

- **HTML теги** – это специальные команды для браузера. Они говорят, например, что следует считать заголовком страницы, а что абзацем. Теги строятся по такому принципу: уголок <, потом имя тега, а потом уголок >. Имя тега может состоять из английских букв и цифр. Примеры тегов: <h1>, , <b>.

  Теги обычно пишутся парами – открывающий тег и закрывающий. Разница между ними в том, что в закрывающем теге после уголка < стоит слеш /. **Пример: ..... ** . Все, что попадает между открывающим и закрывающим тегами, **подпадает под воздействие нашего тега**. Бывают теги, которые не нужно закрывать, например,
 или <img>.

- **АТРИБУТЫ** – команды, которые расширяют действие тега. Атрибуты размещаются внутри открывающего тега в таком формате: **<тег** атрибут1=’’значение’’ атрибут2=’’значение’’**>**.

  Кавычки могут быть любыми – одинарными или двойными, допустимо вовсе их не ставить, если значение атрибута состоит из одного слова.

БАЗОВАЯ СТРУКТУРА HTML-ДОКУМЕНТА

- **<html>** – указывает, что это HTML код.
  Перед тегом <html> обычно пишется конструкция DOCTYPE (**<!DOCTYPE** html**>**). DOCTYPE отвечает за корректное отображение веб-страницы браузером.
- **<head>** – содержит техническую информацию, например, заголовок страницы, подключаемые стили и скрипты, мета-информацию и т.д.
- **<title>** (заголовок) – содержит заголовок HTML-документа, который отображается во вкладке браузера.
- **<body>** – содержимое документа, которое видит пользователь.
- **<meta>** – задаёт описание содержимого страницы и ключевые слова, автора HTML-документа и др. свойства метаданных, например, кодировку (ставится в атрибуте **charset** и обычно имеет значение utf-8).

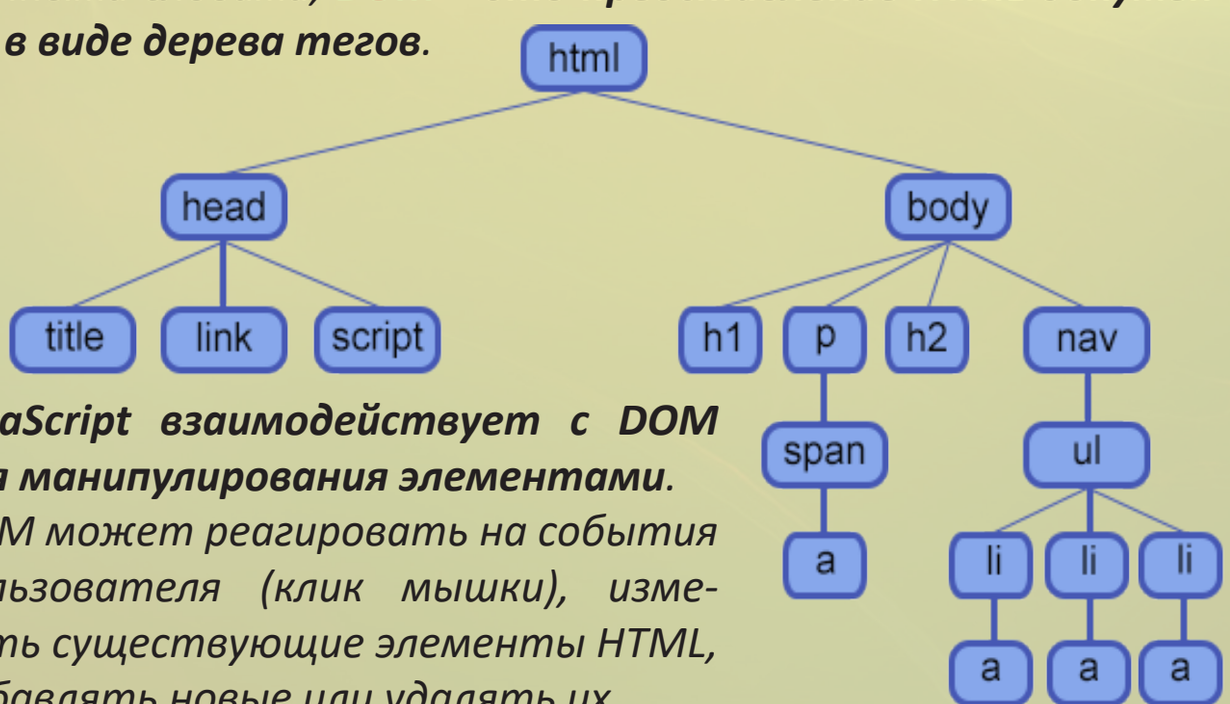
АБЗАЦЫ Каждый абзац начинается с новой строки и имеет так называемую красную строку. Создается с помощью тега <p>

ЗАГОЛОВКИ ТЕКСТА Используются для определения нового раздела или подраздела. Создаются с помощью тегов <h1>, <h2>, <h3>, <h4>, <h5>, <h6>. Имеют разную степень важности. В заголовке **h1** следует располагать название всей HTML страницы, в **h2** – название блоков страницы, в **h3** – название подблоков и т.д. Все заголовки по умолчанию жирные и имеют разный размер.

DOM (Document Object Model)

Это объектная модель документа, созданная браузером в памяти ПК на основании HTML-кода, полученного от сервера.

*Иными словами, **DOM** – **это представление HTML-документа в виде дерева тегов**.*



JavaScript взаимодействует с DOM для манипулирования элементами.
DOM может реагировать на события пользователя (клик мышки), изменять существующие элементы HTML, добавлять новые или удалять их.

CSS

CSS расшифровывается как **Cascading Style Sheets**, что в переводе означает «Каскадные Таблицы Стилей».

ЯЗЫК CSS – это язык разметки, используемый для визуального оформления веб-сайтов. CSS работает в связке с языком разметки HTML. Сначала на HTML создается структура документа – заголовки и подзаголовки, маркированные списки и ссылки, таблицы. А потом с помощью CSS – определяется стиль всех элементов.

Синтаксис

В основе синтаксиса CSS лежат правила, которые определяют стиль HTML-элементов страницы. Чтобы прописать правило, необходимо задать **селектор**.

```
h1 {color:red; font-size:10px;}
```

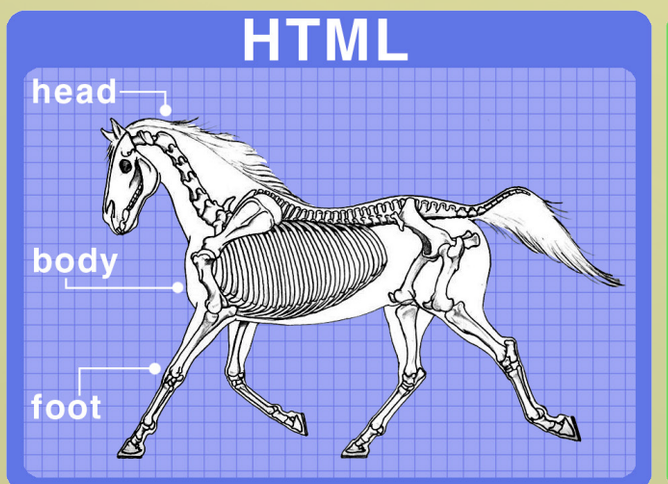
Селектор — h1; *Свойство* — color; *Значение* — red; *Свойство* — font-size; *Значение* — 10px

Селектор – это ссылка на элемент в HTML, находится в начале CSS-правила и определяет, к каким HTML-элементам применятся **свойства** и **значения** из правила.

- ***Свойство*** *(Property) – определенная характеристика элемента, которую нужно изменить. Например, размер или цвет. Указывается после фигурной скобки.*
- ***Значение*** *(Value) – цифровое или текстовое обозначение для выбранного свойства. Описывает, как именно свойство будет обработано браузером.*

HTML vs CSS

*В отличие от **HTML**, который служит для определения структуры и семантики содержимого сайта, **CSS** отвечает за его внешний вид и отображение.*



JAVASCRIPT

JavaScript (JS) – высокоуровневый интерпретируемый язык программирования, который используется для создания динамических и интерактивных элементов в веб-продуктах.

Программы, написанные на JavaScript, называются сценариями или скриптами. Добавление их на страницу выполняется через тег <script>. При этом их можно как непосредственно вставлять на веб-страницу, так и размещать в отдельном файле.

***Интерпретируемый язык программирования** – язык, в котором исходный код с помощью интерпретатора выполняется без предварительной компиляции в машинный код.*

***Интерпретатор** – ПО, которое построчно читает код и сразу выполняет написанные команды. **Интерпретатор для JavaScript встроен во все основные браузеры**, именно поэтому они могут выполнять скрипты на странице.*

На JS написан практически весь *фронтенд*. JavaScript позволяет добавлять функциональность веб-продуктам, используя множество существующих *API*. Например, браузерные API используют скрипты для отображения веб-страниц, взаимодействия с пользователем, отправки запросов на серверы и получения ответов. API сторонних производителей позволяют получать доступ к функциям сторонних сервисов.

Некоторые возможности JavaScript:

- **Адаптация контента** под окно браузера.
- **Добавление интерактивности**. Например, при нажатии на «сердечко» под постом или фото в социальной сети число лайков увеличивается без обновления страницы.
- **Автозаполнение полей**. Браузер выдаёт рекомендации при повторном заполнении полей (имя, пароль, телефон и т. д.).
- **Добавление анимации**. Например, скролл до раздела по нажатию кнопки, анимацию фона, движущиеся объекты.
- **Выполнение математических операций**. Например, подсчет денежной суммы на товары в корзине.
- **Создание браузерных игр**. На JavaScript можно писать простые игры.

КЭШ. КУКИ. СЕССИИ

Кэш (Cache) – **копии загруженных ранее веб-страниц, хранящиеся в памяти персонального компьютера** (или другого устройства) включая: изображения, видео, музыку. При следующем посещении сайта загрузка происходит с жесткого диска, что позволяет существенно повышать скорость загрузки.

На сайты регулярно вносятся изменения и тогда возникают различия между кэшированной версией сайта и версией, загружаемой с сервера. Это приводит к некорректности отображения веб-ресурса. Перед проведением тестирования нужно обязательно очистить кэш браузера.

Куки (Cookie) – **служебные файлы, отправляемые с сервера и хранящиеся на клиентском компьютере**. Куки хранят информацию, относящуюся к конкретному пользователю, например: идентификационную информацию, пользовательские предпочтения, выбранные пользователем настройки сайта и др. Эта информация собирается и анализируется сервером, например для: авторизации посетителя, персонализации выбранных клиентом настроек при запросах. Некоторые куки хранятся только в течение одной сессии, они удаляются после закрытия браузера.

Сессия (Session) – **данные хранящиеся на сервере,** позволяющие идентифицировать браузер и обрабатывать запросы в течение одного сеанса. Со стартом сессии на сервере создаётся документ, где находятся данные о клиенте, его манипуляциях и событиях, произошедших в пределах одной сессии.

БРАУЗЕР и ДВИЖОК БРАУЗЕРА

***Браузер** – ПО для просмотра веб-страниц, для этого в интерфейсе любого браузера есть адресная строка.*

***Движок браузера** – это программное ядро любого браузера, отвечающее за рендеринг (перевод кода веб-ресурса, загруженного с сервера, в понятное для восприятия пользователем изображение на экране). Некоторые популярные движки:*

- ***Blink**. Применяется в Google Chrome, Opera, Microsoft Edge, Yandex Browser.*
- ***WebKit**. Оптимизирован под устройства компании Apple, используется в собственном браузере – Safari.*
- ***Gecko**. Кроссплатформенный движок, поддерживает Windows, macOS, Linux и Android. Применяется в Firefox.*
- ***Quantum**. Обновление движка Gecko.*

DEVTOOLS
(Инструменты разработчика)

DevTools – это набор инструментов для веб-разработчиков, встроенный непосредственно во все современные браузеры. **DevTools** показывает изнанку сайта, предоставляя информацию о коде, сетевых запросах и других аспектах разработки.

Тестировщику DevTools позволяет выполнять различные задачи, связанные с обеспечением качества продукта. Например, эмуляцию мобильных устройств (**Toggle Device Toolbar)** – инструмент помогает протестировать отображение интерфейса для разных устройств и разных размеров экранов.

Запустить DevTools можно так:

- **Перейдя в раздел** «Дополнительные инструменты / Инструменты Разработчика».
- **Нажать на правую** клавишу мыши, выбрать «Просмотреть код».
- **Или с помощью** сочетания клавиш
  - **Cmd+Alt+I** (macOS)
  - **F12** (Windows)
  - **Ctrl+Alt+I** (Linux)

Вкладки DevTools

- **Elements**. Используется для выбора и редактирования любых HTML элементов на странице. Позволяет свободно манипулировать DOM и CSS.
- **Console**. Отображение вывода JavaScript и выполнение кода для тестирования и отладки страницы. Все ошибки в JS коде будут выводиться здесь с указанием файла и конкретного места в нём, где произошла ошибка.
- **Sources**. Отображение загруженных файлов из всех источников, к которым обращался сайт. Просмотр содержимого с возможностью сохранения. Данную вкладку можно использовать и как полноценный редактор кода.
- **Network**. Отображение сетевых запросов, которые делает сайт. Позволяет отслеживать запросы HTTP/HTTPS, помогая выявлять ошибки в запросах и неправильные ответы сервера.
- **Performances**. Отображение нагрузки, которую создает сайт на компьютер пользователя.
- **Memory**. Инструменты для отслеживания нагрузки на систему при выполнении кода.
- **Security**. Оценка безопасности текущей страницы.
- **Lighthouse**. Проверка производительности сайта.

Введение в тестирование ПО

ТЕСТИРОВАНИЕ

Тестирование (Software Testing) – проверка соответствия реальных и ожидаемых результатов поведения программы, проводимая на конечном наборе тестов, выбранном определённым образом.

Цели тестирования:

- Выявление дефектов до того, как их обнаружат пользователи.
- Предоставление актуальной информации о состоянии продукта на данный момент.
- Проверка на соответствие ПО всем заявленным требованиям.

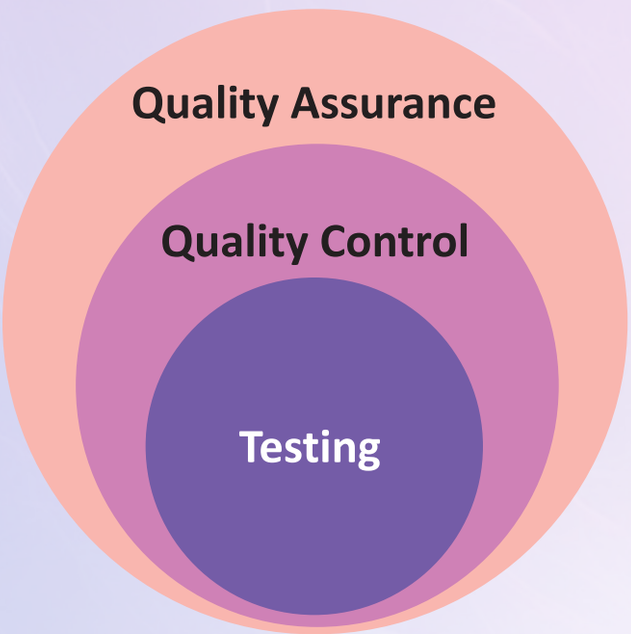
Принципы тестирования

- **Тестирование демонстрирует наличие дефектов.** Тестирование только снижает вероятность наличия дефектов, которые находятся в ПО, но не гарантирует их отсутствия.
- **Исчерпывающее тестирование невозможно.** Полное тестирование с использованием всех входных комбинаций данных, результатов и предусловий физически невыполнимо.
- **Раннее тестирование.** Следует начинать тестирование на ранних стадиях жизненного цикла разработки ПО, чтобы найти дефекты как можно раньше.
- **Скопление дефектов.** Большая часть дефектов находится в ограниченном количестве модулей.
- **Эффект (парадокс) пестицида.** Если повторять те же тестовые сценарии снова и снова, в какой-то момент этот набор тестов перестанет выявлять новые дефекты.

  Суть его в том, что если вы долго проводите одни и те же проверки, скорее всего новых багов вы не найдете. Именно поэтому периодически нужно «встряхивать» тестовую базу, ревьюить её новыми сотрудниками, проводить исследовательское тестирование.
- **Тестирование зависит от контекста.** Например, программное обеспечение, в котором критически важна безопасность, тестируется иначе, чем новостной портал.
- **Заблуждение об отсутствии ошибок.** Отсутствие найденных дефектов и ошибок при тестировании не всегда означает готовность продукта к релизу. Система должна быть удобна пользователю в использовании и удовлетворять его ожиданиям и потребностям.

QA, QC и TESTING

Часто, под **Quality Assurance** (QA, обеспечение качества) подразумевается процесс **тестирования** (Testing). Обеспечение качества тесно связано с тестированием, но на самом деле это разные понятия. QA, в сравнении с тестированием, является более широким понятием. Кроме того, QA также включает в себя понятие – **контролем качества** (Quality Control, QC).



Атрибуты качества ПО:

- *Функциональность*
- *Безопасность*
- *Совместимость*
- *Надёжность*
- *Удобство использования*
- *Производительность*
- *Удобство сопровождения*
- *Переносимость*

Подробнее здесь

Quality Assurance (QA). Это совокупность мероприятий направленных на создание и выпуск качественного продукта. В случаях когда качество не достигло ожидаемого уровня QA анализирует проблемы и пытается решить, каким образом можно сделать продукт лучше.

Quality Control (QC). Предполагает контроль соблюдения требований, выполняет анализ результатов тестирования и даёт оценку готовности и качества компонента или системы.

Testing – деятельность, направленная на оценку системы или её компонентов для проверки соответствия конкретным требованиям, является частью как QC, так и QA. Цель тестирования — выявить любые дефекты или проблемы в продукте перед его релизом для конечных пользователей.

ТЕСТИРОВЩИК

Тестировщик – специалист, который проверяет качество ПО и уровень его соответствия заранее определённым требованиям.

Роль тестировщика на проекте:

- Поиск проблем с точки зрения пользователя и передача информации для исправления найденных недостатков.
- Предоставление информации о работе ПО заинтересованным лицам.

БАГ

БАГ *(Bug)* – отклонение **фактического** результата от **ожидаемого**. Найденные баги оформляются в *баг-репорты*.

Ожидаемый результат *(Expected Result) – то, как должна работать система в соответствии с документацией.*

Фактический результат *(Actual Result) – то, как система работает на самом деле (результат, полученный тестировщиком во время тестирования).*

И вроде казалось бы, что с термином баг (он же дефект) все предельно понятно, но есть нюанс.

- **Баг** (дефект). Любой недостаток в продукте, из-за которого этот продукт не соответствует требованиям.
- **Ошибка** (Error, Mistake). Действие, приводящее к некорректным результатам, например ошибки в коде при разработке.
- **Сбой** (Fault). Ситуация, когда приложение не может правильно функционировать. Например, из-за недостатка памяти.
- **Отказ** (Failure). Сочетание дефектов, приводящие к полному отказу приложения.

  Отказ – это постоянное нарушение работы системы, то есть потеря работоспособности системы, а сбой – временное нарушение, часто самоустраняющиеся.

Верификация и валидация

Верификация – это процесс проверки соответствия формальным требованиям.

Валидация – проверка соответствия продукта требованиям пользователя *(хотелкам пользователя)*.

ПО может на 100 % соответствовать спецификации, но при этом выполнять совершенно не то, что хотел пользователь/заказчик.

ПРИМЕР. Релиз – «ОСЁЛ». *Заказчик по ТЗ хотел: непарнокопытное животное с 4 ногами и умеющее скакать.* ***Верификация пройдена. Валидация – нет. Заказчик хотел коня.***

БАГ vs ФИЧА

Популярная фраза ***«Не баг, а фича»*** *заключается в том, что* ***баг – случайно допущенная ошибка*** *в приложении, а* ***фича – специально сделанная функция****, возможно даже необычная.*

ЖИЗНЕННЫЙ ЦИКЛ ТЕСТИРОВАНИЯ (STLC)

Жизненный цикл тестирования ПО (Software Testing Life Cycle) – это последовательность действий, проводимых в процессе тестирования, с помощью которых гарантируется качество ПО и его соответствие требованиям.

ЭТАПЫ STLC-ЦИКЛА

- ***Общее планирование и анализ требований.*** *Необходимо получить ответы на следующие вопросы: что нам предстоит тестировать; как много будет работы; какие есть сложности; всё ли необходимое у нас есть и т.п.*
- ***Уточнение критериев приёмки.*** *Позволяет сформулировать метрики и признаки необходимости начала (приостановки/возобновления/завершения) тестирования.*
- ***Уточнение стратегии тестирования.*** *Рассматриваются и уточняются те части стратегии тестирования, которые актуальны для текущей итерации.*
- ***Разработка чек-листов и/или тест-кейсов.*** *Посвящена разработке, пересмотру и уточнению тест-кейсов.*
- ***Выполнение тест-кейсов и фиксация найденных дефектов.*** *Дефекты исправляются сразу по факту их обнаружения в процессе выполнения тест-кейсов.*
- ***Анализ результатов тестирования и отчётность.*** *Создается отчет о результатах тестирования. QA-команда обсуждает и анализирует баги, делает выводы из возникших проблем, чтобы избежать подобных проблем в будущем.*

Следует начинать тестирование на ранних стадиях жизненного цикла разработки ПО, чтобы найти дефекты как можно раньше.

Принцип – Раннее тестирование (Early Testing).

Тестовая документация

ТЕСТОВАЯ ДОКУМЕНТАЦИЯ

Тестовая документация – набор документов, который создается на протяжении всего цикла тестирования. Они описывают покрытие тестами и процесс выполнения тестов, содержат перечень необходимых для тестирования вещей, приводится основная терминология и т. д. Документация помогает однозначно трактовать шаги, сроки тестирования, результаты, а также обращаться к этой информации в спорных моментах.

На практике тестовая документация называется ещё ***артефактами тестирования.***

Виды тестовой документации:

- **Тест-план** (Test Plan). Описывает весь объем работ по тестированию программного обеспечения.
- **Тестовая стратегия** (Test Strategy). Определяет то, как тестируем ПО. То есть это набор идей, которые направляют процесс тестирования.
- **Чек-лист** (Check-List). Список, содержащий ряд необходимых проверок.
- **Тест-кейс** (Test Case). Описывает наши тесты. Говорит, как их выполнить, при каких условиях и что должно получиться после выполнения шагов, которые заложены в тест-кейсе.
- **Тест-сьют** (Test Suite). Комплект тест-кейсов для исследуемого компонента или системы. Результат выполнения одного тест-кейса в тест-сьюте может быть предусловием для выполнения следующего.
- **Баг-репорт** (Bug Report). Содержит полное описание дефекта.
- **Отчет о тестировании** (Test Report). Здесь обобщаются все результаты работ по тестированию.

Матрица трассировки

(Traceability Matri)

Это таблица, содержащая соответствие функциональных требований продукта (Functional Requirements) и подготовленных тестовых сценариев (Test Cases).

Позволяет обеспечить полноту тестирования. Помогает обнаружить функциональные пробелы и непокрытые требования до начала тестирования. Может быть как самостоятельным документом, так и частью документации по требованиям, или же частью плана тестирования.

ТЕСТ-ПЛАН vs ТЕСТОВАЯ СТРАТЕГИЯ

План тестирования и **стратегия тестирования** являются частью любой системы контроля качества программного обеспечения, но на разных уровнях. Тест-план детализирует тестовую стратегию. **Однако в реальности граница между этими понятиями сильно размыта.**

Тест-план
(Test Plan)

Документ уровня проекта, то есть он ориентирован на конкретный программный продукт. Создается на одном из первых этапов жизненного цикла тестирования (*STLC)*, чтобы подробно описать, как будет проводиться тестирование для конкретного продукта.

Тест-план содержит пункты:

- **Определение целей и задач тестирования**.
- **Описание объектов тестирования.** Список функций и описание тестируемой системы и её компонентов в отдельности.
- **Определение методов тестирования.** Указываются те виды тестирования, которые будут использованы при тестировании.
- **Описание ресурсов и инструментов.** Перечень необходимого для тестирования оборудования и программных средств., например тестовый стенд и его конфигурация, программы для автоматизированного тестирования и т.д.
- **Роли и обязанности**. Все обязанности и роль каждого члена конкретной команды тестирования должны быть записаны.
- **Оценку трудозатрат** на каждую из работ.
- **Риски и пути их разрешения**.
- **Описание процедур контроля качества**.

На практике же тест-план имеет скорее условный характер, так как требования к проекту могут корректироваться по ходу разработки.

Стратегия тестирования
(Test Strategy)

Документ уровня организации, который устанавливает общий подход к тестированию – что должно быть достигнуто и как этого достичь. Этот документ выходит за рамки STLC и не определяет требования к тестированию для конкретного проекта.

ОЦЕНКА ТРУДОЗАТРАТ

Программное обеспечение может считаться завершенным (готовым) только в том случае, если оно было разработано и тщательно протестировано.

Оценка трудозатрат – это процесс прогнозирования количества времени, усилий и ресурсов, которые потребуются для выполнения работ по тестированию в проекте разработки программного обеспечения.

Задачи оценки трудозатрат

- Планирование и контроль общего графика проекта.
- Определение сроков на проведение тестирования.
- Расчёт приблизительной стоимости тестирования.
- Оптимизация существующих процессов.
- Управление рисками.

Методы оценки трудозатрат

- **Пальцем в небо.** Оценка на основе интуиции тестировщика.
- **Оценка на основе опыта.** Оценка происходит учитывая предыдущий опыт, например, прошедшую итерацию.
- **Эстимация на основе разработки.** Предположение на основе того, сколько времени потратят на разработку девелоперы.
- **Эстимация на основе тест-кейсов.** Оценка на основе активности тестировщика: составление и запуск тест-кейсов, оформление баг-репортов, ретесты, коммуникации с командой.
- **Эстимация по трём точкам.** Устанавливаются три временных точки: оптимистичное, пессимистичное и среднее время, потраченное на тестирование.
- **Покер планирование.** Прогнозы строятся учитывая мнения всех членов команды. Проводится с помощью карт.

Attention!

Данная Шпаргалка интерактивна (кликабельна). Если это не так и при нажатии на ссылки, либо на кнопку «содержание» перехода на другую страницу не происходит – ***вы используете пиратскую версию!***

Санкций ни каких не предусмотрено, но как-то нелогично пользоваться кастрированной версией, когда ***в сети СВОБОДНО распространяется полноценная****. Ищите и обязательно найдете!*

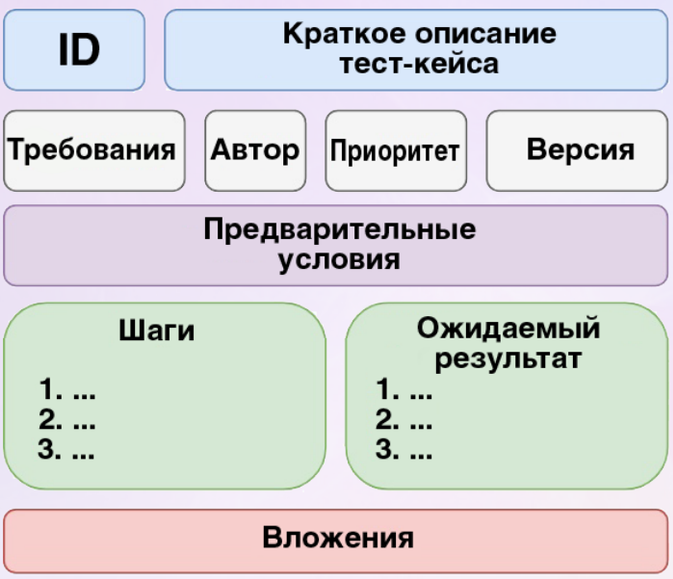
ЧЕК-ЛИСТ

Чек-лист в тестировании ПО – это список пунктов, которые тестировщик должен проверить во время тестирования. Пункты чек-листа, это по сути список тест-кейсов, если смотреть на них как на названия тест-кейсов. То есть чек-лист – это список того, что нужно проверить, но без детализации, как это сделать.

ТЕСТ-КЕЙС

Тест-кейс – это документ, описывающий алгоритм тестирования функции или системы.

Один тест-кейс проверяет одну конкретную функцию или пользовательский сценарий.

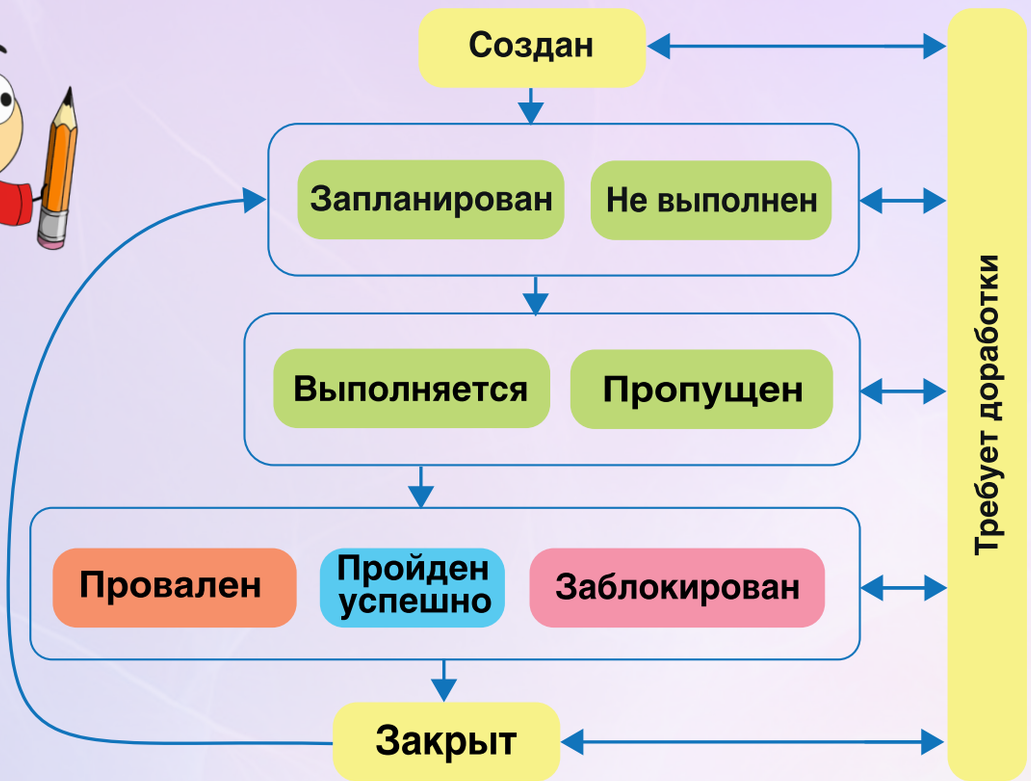


Атрибуты тест-кейса

- **ID – уникальный номер** (Test Case ID). Обычно проставляется автоматически в системах хранения тест-кейсов.
- **Краткое описание / Заголовок кейса** (Name/ Test Case Summary). Краткое и емкое название проводимой проверки.
- **Связанное требование** (Related Requirement). Ссылка на требование или ТЗ, на основе которого был составлен тест-кейс.
- **Автор** (Author). Тестировщик, написавший тест-кейс.
- **Приоритет** (Priority) – насколько важен этот тест-кейс, в какую очередь его стоит выполнять.
- **Название/модуль/версия продукта** (Component/Version) – описание ПО, на котором можно выполнить тест-кейс.
- **Тестовое окружение** (Test Environment). Оборудование / программное обеспечение / сеть, в которых выполнялся тест.
- **Предварительные условия** (Precondition/ Prerequisites) – шаги, которые необходимо выполнить перед выполнением теста.
- **Шаги** (Steps/ Test Script). Шаги выполнения теста.
- **Ожидаемый результат** (Expected Result).
- **Приложения/вложения** (Attachments). Дополнительная информация, например, скриншоты, текстовые файлы и др.
- **Заметки** (Remarks). Любые комментарии к тесту.
- **Статус** (Status). Пройден, не пройден или другие.

ЖИЗНЕННЫЙ ЦИКЛ ТЕСТ-КЕЙСА

Набор состояний, в которых может находиться тест-кейс



- **Создан** (New). Тест-кейс автоматически принимает это состояние после создания.
- **Запланирован** (Planned, Ready for Testing). Состояния тест-кейса, когда он включён в план ближайшей итерации тестирования или готов для выполнения.
- **Не выполнен** (Not Tested). Тест-кейс готов к выполнению, но ещё не был выполнен.
- **Выполняется** (Work in Progress). Работа идёт, и скоро можно ожидать её результатов. Если выполнение тест-кейса занимает мало времени, это состояние, как правило, пропускается, а тест-кейс сразу переводится в состояние – «провален»/«пройден успешно»/«заблокирован».
- **Пропущен** (Skipped). Выполнение тест-кейса отменяется.
- **Провален** (Failed). Был обнаружен дефект.
- **Пройден успешно** (Passed). Не было обнаружено дефектов, связанных с расхождением ожидаемых и фактических результатов.
- **Заблокирован** (Blocked). Выполнение тест-кейса невозможно.
- **Закрыт** (Closed). На данной итерации тестирования все действия с тест-кейсом завершены.
- **Требует доработки** (Not Ready). Как видно из схемы, в это состояние (или из него) тест-кейс может быть переведён в любой момент времени, если в нём будет обнаружена ошибка, если изменятся требования, по которым он был написан.

БАГ-РЕПОРТ

БАГ РЕПОРТ (Bug Report, отчёт об ошибке) – это технический документ, который подробно описывает ошибку в работе программы, приложения или другого ПО для ее последующего устранения разработчиками. *Что такое баг здесь.*

Атрибуты баг-репорта

Количество атрибутов и правила оформления могут варьироваться в зависимости от баг-трекинговой системы, внутренних правил компании и особенностей проекта.

- **Уникальный идентификатор** (ID). Присваивается автоматически системой при создании баг-репорта.
- **Заголовок** (Summary) отвечает на три вопроса:
  - Что произошло?
  - Где появилась ошибка?
  - Когда или при каких условиях найден дефект?
- **Описание ошибки** (Description). Подробное описание дефекта (указывается опционально).
- **Автор баг репорта** (Author). Обычно это Тестировщик.
- **Назначен на** (Assigned to). Имя сотрудника, назначенного на решение проблемы.
- **Серьёзность** (Severity). *Подробно здесь.*
- **Приоритет** (Priority). *Подробно здесь.*
- **Окружение** (Environment)
  - **Компоненты и среда.** Атрибуты дефекта, которые указывают, где воспроизводится этот дефект. Например, iOS, Android, Windows, Mac и их версии. А также названия и версии браузеров, в которых воспроизводится дефект.
  - **Build**. Атрибут, указывающий номер сборки.
- **Предусловия** (Prerequisites). В случае, если есть специфичные действия или шаги воспроизведения достаточно объёмные, то указываются начальные условия.
- **Шаги воспроизведения** (Steps to Reproduce). Описание четкой последовательности действий, при которых повторяется найденная ошибка.
- **Ожидаемый результат** (Expected result). Описание того, как должна работать система в соответствии с документацией.
- **Фактический результат** (Actual result). Описывается поведение системы на момент обнаружения дефекта в ней.
- **Вложения** (Attachment). Скриншоты, видео или *логи*.
- **Статус** (Status). Определяет текущее *состояние бага*.

ПРИОРИТЕТ и СЕРЬЁЗНОСТЬ

Серьёзность

(Severity, критичность)

Атрибут, характеризующий влияние дефекта на работоспособность программного обеспечения.

Уровни серьёзности бага:

- **S1. Блокирующий** (Blocker). Баг полностью блокирует работу приложения. Например, приложение не запускается или выдаёт ошибку при загрузке.
- **S2. Критический** (Critical). Баг блокирует какую-то часть функционала приложения.
- **S3. Значительный** (Major). Часть функционала приложения работает некорректно, но есть возможность решить проблему другим способом.
- **S4. Незначительный** (Minor). Не нарушает работу приложения, то есть не относится к функционалу продукта. Обычно проставляется для тех ошибок, которые в большей степени относятся к удобству использования разработанного ПО.
- **S5. Тривиальный** (Trivial). Часто неотличим от уровня Minor. Не относится к функционалу ПО и не влияет на качество продукта. Но портит общее впечатление от использования, например, это могут быть опечатки в тексте.

Приоритет

(Priority)

Атрибут, указывающий на очерёдность выполнения задачи или устранения дефекта.

С точки зрения функционала баг может быть несерьёзный, но при этом важный (приоритетный) для бизнеса, например: Опечатка на главной странице бизнес-портала. Системе не вредит, а репутации – да.

Уровни приоритетности бага:

- **P1. Высокий** (High). Требуется исправить в первую очередь;
- **P2. Средний** (Medium). Требуется исправить во вторую очередь, когда нет дефектов с высоким приоритетом;
- **P3. Низкий** (Low). Исправляется в последнюю очередь, когда разобрались с багами других приоритетов.

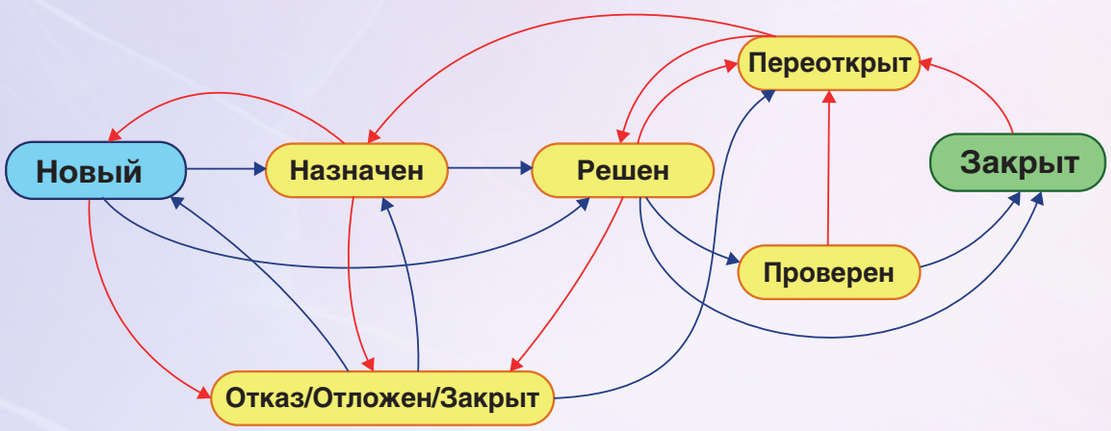
ЖИЗНЕННЫЙ ЦИКЛ И СТАТУСЫ БАГА

Жизненный цикл бага – это определенный набор состояний, через которые проходит баг в течение своей жизни.

Статус бага – это текущее состояние, в котором находится баг. Назначение статуса багу делает процесс его устранения системным и эффективным, позволяя отслеживать фактический прогресс его жизненного цикла.

Количество состояний (и их названий), через которые проходит баг, варьируется. В основном это зависит от баг-трекинговой системы.

Примеры возможных вариантов жизненного цикла бага



- ***Новый*** *(New). Впервые найденный баг, занесённый в систему.*
- ***Назначен*** *(Assigned). После публикации баг просмотрен и утвержден, то есть признан для исправления.*
- ***Открыт*** *(Open). Разработчик приступает к анализу и работает над устранением дефекта.*
- ***Решен*** *(Fixed). Баг исправили, и он в этом состоянии требует повторного тестирования (ретеста).*
- ***Проверен*** *(Verified). Статус бага после ретеста, если он после исправления больше не воспроизводится.*
- ***Переоткрыт*** *(Re-opened). Баг после ретеста не исправлен или исправлен не полностью.*
- ***Закрыт*** *(Closed). Баг больше не существует.*
- ***Отказ*** *(Rejected). Баг отклонён. Причины: некачественное описание или такой дефект уже существует, невозможность воспроизвести баг.*
- ***Дубликат*** *(Duplicate). Если баг повторяется или соответствует концепции другого бага.*
- ***Отложен*** *(Deffered). Если данный баг не является приоритетным или планируется его исправление в следующем релизе.*
- ***Не баг*** *(Not a Bug). Баг по факту оказался не багом.*

ОТЧЕТ О ТЕСТИРОВАНИИ

Отчет о результатах тестирования – документ, обобщающий результаты работ по тестированию и содержащий информацию, достаточную для соотнесения текущей ситуации с тест-планом и принятия необходимых управленческих решений.

Отчёт содержит следующие разделы:

- **Краткое описание**. Отражает основные достижения, проблемы, выводы и рекомендации.
- Список **команды тестировщиков** с указанием их должностей.
- **Описание процесса тестирования**. Последовательное описание того, какие работы были выполнены за подотчётный период.
- **Расписание** работы команды тестировщиков.
- **Статистика по новым дефектам**.
- **Список новых дефектов** с их краткими описаниями.
- **Статистика по всем дефектам**.
- **Рекомендации**.
- **Приложения**. Фактические данные (как правило, значения метрик и графическое представление их изменения во времени).

МЕТРИКИ

Метрика тестирования ПО определяется как количественная мера, которая помогает оценить прогресс, качество и и производительность команды QA.

Примеры метрик тестирования ПО:

- Количество найденных и исправленных дефектов.
- Процент выполненных тестов.
- Время исправления дефектов.
- Количество пропущенных дефектов.

ПМИ

Программа и методика испытаний (ПМИ) – это технический документ, который устанавливает порядок проведения приемочных испытаний системы в целом или отдельных ее компонентов.

ПМИ включает следующие разделы: объект испытаний, цель проведения испытаний, основание для проведения испытаний, объём испытаний, методы и средства испытаний.

ПМИ разрабатывается на основе требований технического задания, перед испытаниями документ согласовывает заказчик.

Виды тестирования

ВИДЫ ТЕСТИРОВАНИЯ

Классификаций видов тестирования существует великое множество и в разных источниках можно найти значимые отличия информации. В любом случае, приведенная ниже классификация не является полной, а отображает лишь основные виды тестирования.

По целям (объектам) тестирования
- **Функциональное** тестирование
  - ***По уровню детализации приложения***
    - *Модульное*
    - *Интеграционное*
    - *Системное*
    - *Приемочное*
- **Нефункциональное** тестирование
  - ***Нефункциональное тестирование***
    - *Инсталляционное*
    - *Нагрузочное*
    - *Юзабилити*
    - *Локализации/интернационализации*
    - *Документации*

Связанное с изменениями в коде продукта
- **Подтверждающее** тестирование
- **Регрессионное** тестирование
- **Дымовое** тестирование
- **Санитарное** тестирование

По степени важности тестируемых функций
- **Дымовое** тестирование
- **Тестирование** критического пути
- **Расширенное** тестирование

По характеру сценария
- **Позитивное** тестирование
- **Негативное** тестирование

По степени формализации
- **На основе тест-кейсов**
- **Исследовательское** тестирование
- **Свободное** тестирование

По доступу к коду
- **Белый ящик** (есть доступ к коду)
- **Чёрный ящик** (нет доступа к коду)

По запуску кода на исполнение
- **Статическое** тестирование (без запуска)
- **Динамическое** тестирование (с запуском)

По степени автоматизации
- **Ручное** тестирование
- **Автоматизированное** тестирование

КЛАССИФИКАЦИЯ ПО ЦЕЛЯМ (ОБЪЕКТАМ) ТЕСТИРОВАНИЯ

По целям можно разделить тестирование на **функциональное** и *нефункциональное*. Некоторые авторы добавляют в эту градацию и *тестирование, связанное с изменениями*.

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

(Functional Testing)

Это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям.

Этапы функционального тестирования

1. Планирование. Исследование функциональных требований для понимания того, что должна делать система в соответствии с требованиями заказчика. Создание плана тестирования.

2. Составление чек-листов и написание тест-кейсов.

3. Подготовка тестовых данных. Тестовые данные должны покрывать различные входные условия и быть максимально приближены к тем, что могут использовать пользователи.

4. Тестирование и анализ результатов. Оценка соответствия поведения системы ожидаемым результатам, выявление ошибок или несоответствий требованиям. Регистрация дефектов и проведение регрессионных тестов.

5. Составление отчета о результатах тестирования, в котором фиксируется подробная информация о выполнении тестирования и его результатах, а также об оставшейся работе.

Некоторые примеры функционального тестирования:

- *Модульное (Юнит)*
- *Интеграционное*
- *Системное*
- *Приёмочное*
- *Дымовое*
- *Регрессионное*
- *Подтверждающее*
- *Тестирование API*

АЛЬФА-ТЕСТИРОВАНИЕ vs БЕТА-ТЕСТИРОВАНИЕ

Альфа-тестирование *– тестирование внутри компании специальной командой тестировщиков.*

Бета-тестирование *– тестирование независимыми тестерами (потребителями/юзерами).*

КЛАССИФИКАЦИЯ ПО УРОВНЯМ ДЕТАЛИЗАЦИИ СИСТЕМЫ

Это тестирование так же называется – тестирование по уровням, то есть эта **классификация основана на разделении процесса тестирования на несколько уровней**.

Такой подход к тестированию направлен на персональную проверку определенной функциональности и взаимодействия компонентов приложения. Это помогает находить дефекты на разных этапах разработки – от изолированного тестирования отдельных модулей до проверки работы всей системы.

Классификация по **уровням детализации системы** включает в себя четыре вида тестирования: модульное, интеграционное, системное, приёмочное. Эти тесты визуально можно собрать в *пирамиду функционального тестирования*.

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ
(Unit Testing)

Тестирование отдельных компонентов ПО. Каждый модуль тестируется независимо от других. Тестировщики анализируют входные и выходные данные, проверяя функционирование каждого компонента. Компонент – наименьший элемент ПО, который может быть протестирован отдельно. Модульное тестирование позволяет выявить дефекты на ранней стадии их появления.

*Ряд источников отдельно выделяют **Unit-тестирование**, как компонентное тестирование методом белого ящика, выполняемое самими разработчиками.*

ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ
(Integrationl Testing)

Фаза тестирования ПО, при которой отдельные модули объединяются и тестируются в группе.

СИСТЕМНОЕ ТЕСТИРОВАНИЕ
(System Testing)

Выполняемое на полной, интегрированной системе на предмет соответствия всей системы исходным требованиям. Это тестирование *чёрного ящика*.

ПРИЁМОЧНОЕ ТЕСТИРОВАНИЕ
(Acceptance Testing)

Это последнее тестирование перед развертыванием, делается для проверки готовности ПО выполнять задачи и функции, поставленные при разработке.

ПИРАМИДА ТЕСТИРОВАНИЯ

При классификации на уровни часто пользуются визуальными представлениями в виде ромбов, пирамид, перевернутых пирамид. Одним из основных визуальных представлений является пирамида тестирования.

Пирамида тестирования – это схематическое отображение **количества** тестов на разных этапах. Тесты более высокого уровня должны проверять только то, что невозможно проверить на предыдущих уровнях.

Пирамида тестирования диктует, как выстраивать тесты на проекте. В идеале должно быть много юнит-тестов, меньше интеграционных тестов, ещё меньше системных тестов и минимально приемочных.

- *ПО принимается заказчиком на соответствие заявленным требованиям либо тестировщики проходят **end-to-end** сценарии с точки зрения пользователя.*
- *Тестируется весь продукт полностью.*
- *Несколько модулей продукта тестируются вместе.*
- *Проверяется минимально-атомарный модуль ПО*

End-to-end (E2E) или сквозное тестирование *– тщательное тестирование ПО командой тестировщиков на основании сценариев «от начала до конца»* ***со всеми задействованными функциями****. То есть проверка всех уровней системы, начиная от пользовательского интерфейса и заканчивая БД и серверной логикой. В литературе часто используется, как* ***синоним системному тестированию****, реже как приемочному.*

Частный случай end-to-end тестирования – это тестирование критического пути. ***Тестирование критического пути*** *– тестирование, во время которого проверяется* ***основная функциональность*** *ПО, критичная для конечного пользователя при стандартном его использовании.*

НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ
(Non-functional Testing)

Нефункциональное тестирование. Анализ атрибутов качества компонента или системы по нефункциональным параметрам.

Примеры нефункциональных видов тестирования:

- **Инсталляционное тестирование**. Проверка успешной установки (инсталляции), выборочной установки, удаления (деинсталляции), настройки и обновления приложения в среде, максимально приближенной к эксплуатационной.
- **Нагрузочное тестирование**. Проверка устойчивости и производительности программного обеспечения под нагрузкой, сопоставимой с реальными условиями использования. Различают:
  - ***Непосредственно нагрузочное.*** *Проверяется, работа приложения или сайта при заданной нагрузке.*
  - ***Тестирование производительности.*** *Проверка поведения системы в различных условиях. Например, при каком максимальном количестве запросов пользователей она в состоянии работать быстро, без ошибок.*
  - ***Тесты на стабильность.*** *Проверяется работоспособность на длительном промежутке времени.*
  - ***Тестирование отказоустойчивости.*** *Например, при отключении электричества, как быстро система возобновит работу, переключившись на резервный сервер.*
  - ***Стресс-тесты****. Проверка работоспособности продукта в период предельной нагрузки, к примеру, онлайн-магазин в «Черную Пятницу».*
  - ***Тесты масштабируемости****. Тестирование стабильной работы системы при ожидаемом увеличении нагрузки, например, при большем количестве пользователей или объеме данных.*
- **Тестирование безопасности**. Проверка на уязвимости ПО, которые могут привести к потере информации компании. Проверки включают – тестирование на проникновение и этический взлом, оценку рисков и сканирование уязвимостей, и ряд других мер.
- **Тестирование совместимости**. Проверка работы программного обеспечения на разных платформах, браузерах и устройствах.
- *Юзабилити-тестирование*
- *Тестирование локализации/интернационализации*
- *Тестирование требований*

ЮЗАБИЛИТИ-ТЕСТИРОВАНИЕ

(Usability Testing)

Метод **оценки интерфейса со стороны удобства и эффективности его использования** конечным пользователем.

Тестирование включает в себя:

- Проходимость пользовательских сценариев.
- Востребованность и впечатления от системы.
- Понятность навигации.
- Полноту и доступность контента.
- Восприятие дизайна.

ТЕСТИРОВАНИЕ ДОСТУПНОСТИ

(Accessibility Testing)

Это **проверка ПО на пригодность к использованию людьми с нарушениями слуха, зрения, двигательной активности**. Тестирование доступности это часть Usability Testing.

Доступность важна всем пользователям. Например, находясь без наушников в шумной обстановке или напротив, когда необходимо соблюдать тишину – должна быть возможность просмотреть видео с субтитрами.

Создаваемое ПО в идеале должно быть доступным (содержать ряд встроенных функций), но необходимо также учитывать возможность использования **вспомогательных технологий**. Пример таких технологий: программы для распознавания речи; скринридеры озвучивают текст, появляющийся на экране; программы-лупы позволяющие увеличить изображение; специальные клавиатуры для облегчения ввода текста.

Доступность (Accessibility) vs Доступность (Availability)

***Accessibility** – доступность продукта для людей с различными физическими ограничениями, а **Availability** – это доступность системы для обращений к ней. Эти два понятия из-за одинакового перевода регулярно путают.*

Примеры проблем с доступностью (Availability):

- ***Некорректное отображение** элементов интерфейса.*
- ***Истекший SSL-сертификат** (SSL-сертификат удостоверяет подлинность сайта и позволяет использовать зашифрованное соединение).*
- ***Катастрофически низкая скорость загрузки сайта.***

ТЕСТИРОВАНИЕ ИНТЕРНАЦИОНАЛИЗАЦИИ
(Internationalization Testing)

Интернационализация – процесс проектирования и разработки ПО таким образом, чтобы оно могло быть адаптировано к различным языковым, культурным и региональным особенностям без необходимости проведения дополнительных инженерных изменений. **Тестирование интернационализации** проверяет, может ли программный код работать со всей международной поддержкой без нарушения функциональности.

Примеры тестирования:

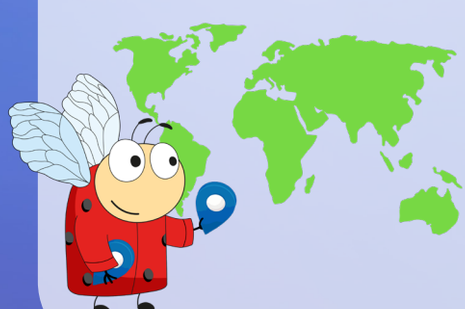
- **Возможность кодовой поддержки** местных, региональных, языковых или относящихся к культуре особенностей.
- **Функциональность.** Тестирование функциональности в различных языковых средах и ввод строк на родном языке.
- **Проверка пользовательского интерфейса.** Выявление таких проблем, как наложение текста, усечение текста и др.
- **Тестирование установки.** Попытка установить приложение на разных языках и проверка, правильно ли отображаются все сообщения об установке в языковых настройках.

ТЕСТИРОВАНИЕ ЛОКАЛИЗАЦИИ
(Localization Testing)

Локализация фокусируется на адаптации уже интернационализированного продукта для конкретного региона или языковой группы. **Тестирование локализации** проверяет, что контент и интерфейс соответствуют языковым, культурным и другим требованиям определённого целевого рынка.

Проверяют адаптацию следующих аспектов:

- **Форматы времени и даты, денежные единицы.**
- **Использование клавиатуры.**
- **Символы, знаки и цвета.**
- **Текст и графические изображения,** которые в данной культурной среде могут быть неправильно истолкованы.
- **Различные правовые требования.**
- **И многое другое**.



*Подготовка приложения к интернациональному использованию называется **глобализацией**. Глобализованное ПО должно поддерживать ввод, вывод на любом языке и обработку информации по правилам этого языка.*

ТЕСТИРОВАНИЕ ТРЕБОВАНИЙ
(Requirements Testing)

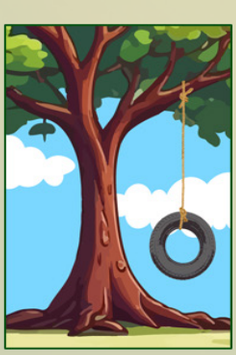
Требования к ПО – это спецификация того, что должно быть реализовано. В требованиях описано поведение системы, свойства системы или её атрибуты.

***Цель тестирования** – устранить возможные ошибки на начальных этапах проектирования системы.*

Свойства хорошего документа

- **Требования должны быть полными**, правильно и в полной мере описывать функцию, которую необходимо реализовать.
- **Однозначность**. Одинаковое восприятие документа всей командой.
- **Непротиворечивость**. Отсутствие противоречивых требований, конфликтующих между собой.
- **Необходимость**. Требования должны отражать функциональности, необходимые для пользователя, чтобы гарантировать, что система или продукт отвечают его потребностям.
- **Выполнимость**. Насколько требования возможно реализовать.
- **Тестируемость**. Возможность проверить все прописанные требования после их реализации.
- **Модифицируемость**. Характеризует простоту внесения изменений в отдельные требования и в набор требований.
- **Атомарность**. Требование описывает только одну ситуацию.
- **Прослеживаемость**. Позволяет описывать и отслеживать жизненный цикл требования как в прямом, так и в обратном направлении. То есть от его происхождения, через разработку и спецификацию, до последующего развёртывания и использования.
- **Актуальность**. Требование не устарело по прошествии времени.

Когда нет продуманного и чёткого технического задания

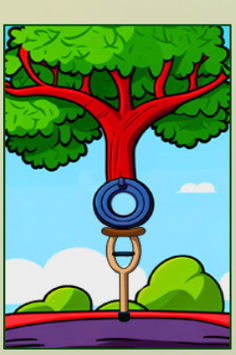


Что хотел заказчик

Как он это объяснил

Как понял руководитель проекта

Как спроектировал дизайнер



Как реализовал программист

*В реализации иллюстрации мне помогла нейросетка **Kandinsky** и её недопиленная (на август 2024) функция «Вариация изображения». Хотя возможно это тот самый случай, когда **«это не баг, а фича!»***

ТЕСТИРОВАНИЕ, СВЯЗАННОЕ С ИЗМЕНЕНИЯМИ

Задача этого вида тестирования – подтвердить устранение выявленных в процессе тестирования багов и убедиться, что внесенные изменения не сказались на работе других частей программного обеспечения.

ПОДТВЕРЖДАЮЩЕЕ ТЕСТИРОВАНИЕ

(Re-testing/ретест)

Повторное тестирование после фикса бага (устранения дефекта), чтобы понять – исправлен баг или нет. Проводится по шагам указанным в *баг-репорте*. Это позволяет разработчикам самостоятельно проверить исправление бага.

РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ

(Regression Testing)

Относится к углубленному и всестороннему исследованию программного обеспечения с целью убедиться, что недавнее изменение кода не оказало негативного влияния на существующие ранее функции, то есть на проверку того, что новый функционал не сломал старый функционал, который до этого работал. Проводится, соответственно, если в ПО была введена новая функция или внесены изменения в код, например после исправления бага.

*Часто понятие **регрессионное тестирование** используют **как собирательное название** для всех видов **тестирования ПО, связанных с изменениями**.*

ДЫМОВОЕ ТЕСТИРОВАНИЕ

(Smoke Testing)

Проверка критически важных функций и стабильности системы в целом перед тщательным тестированием. *Подробнее здесь*.

САНИТАРНОЕ ТЕСТИРОВАНИЕ

(Sanity Testing)

Проверка работоспособности отдельных функциональных элементов. Обычно санитарные проверки выполняются до старта полного цикла проведения регрессионных проверок, но только после окончания дымового тестирования.

ТЕСТИРОВАНИЕ СБОРКИ

(Build Verification Test)

Проверка стабильности системы в целом, т.е. проверка критически важных функций конкретной сборки. Является аналогом дымового тестирования, но может проникать глубже.

КЛАССИФИКАЦИЯ ПО СТЕПЕНИ ВАЖНОСТИ ТЕСТИРУЕМЫХ ФУНКЦИЙ

ДЫМОВОЕ ТЕСТИРОВАНИЕ
(Smoke Testing)

Направлено на проверку ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования ПО. Обычно выполняется после установки существенных обновлений или исправлений приложения.

Термин Smoke Testing возник в электронной промышленности, где новое устройство, при подключении к нему питания, сначала проверяли на отсутствие дыма.

ТЕСТИРОВАНИЕ КРИТИЧЕСКОГО ПУТИ
(Critical Path Test)

Одно из самых распространенных видов функционального тестирования. Если дымовое тестирование пройдено успешно, то на уровне критического пути должны быть проверены **все сценарии стандартного использования приложения пользователем**, исключая негативные сценарии.

РАСШИРЕННОЕ ТЕСТИРОВАНИЕ
(Extended Test)

Тестирование направлено на исследование всей заявленной в требованиях функциональности. Тест-кейсы этого уровня могут затронуть даже самые низкоприоритетные требования.

КЛАССИФИКАЦИЯ ПО ХАРАКТЕРУ СЦЕНАРИЯ

ПОЗИТИВНОЕ ТЕСТИРОВАНИЕ
(Positive Testing)

Это тестирование проверяет, правильно ли функционирует ПО и **выдает ли ожидаемый результат при адекватном поведении пользователя** (например, введение валидных данных). По сути это проверка, что система делает то, для чего и была создана.

НЕГАТИВНОЕ ТЕСТИРОВАНИЕ
(Negative Testing)

Это тестирование направлено на проверку того, что **система ведёт себя должным образом в нештатных ситуациях**. Например, при введении неверных данных. Негативные тесты проводятся только после успешного прохождения позитивных.

КЛАССИФИКАЦИЯ ПО СТЕПЕНИ ФОРМАЛИЗАЦИИ

ТЕСТИРОВАНИЕ НА ОСНОВЕ ТЕСТ-КЕЙСОВ

(Test-Case Based Testing)

Формализованный подход, в котором **тестирование производится на основе заранее подготовленных тест-кейсов** и иной документации. Этот вид тестирования представляет собой необдуманную деятельность, при которой тестировщики, выполняя сценарии, сравнивают фактические результаты с ожидаемыми. Такая деятельность не требует серьезных когнитивных навыков и может быть автоматизирована.

ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ

(Exploratory Testing)

Частично формализованный подход. Тестировщик выполняет работу с продуктом по выбранному сценарию, который дорабатывается в процессе выполнения с целью более полного исследования приложения. То есть **процесс разработки тестов и их прохождение выполняются одновременно**.

СВОБОДНОЕ (ИНТУИТИВНОЕ) ТЕСТИРОВАНИЕ

(Ad-hoc Testing)

Полностью неформализованный подход, в котором **не предполагается использования ни тест-кейсов, ни чек-листов, ни сценариев**. Тестировщик опирается на свою интуицию и опыт для спонтанного выполнения с продуктом действий, которые, как он считает, могут обнаружить ошибку.

Одним из видов Ad-hoc тестирования является **Monkey Testing**. Это произвольное тестирование программы с целью её сломать. То есть буквально тыкаем куда попало без цели и плана.

СВОБОДНОЕ vs ИССЛЕДОВАТЕЛЬСКОЕ

Разница между Ad-hoc и исследовательским тестированием в том, что, свободное может провести кто угодно. А для проведения исследовательского необходимо мастерство и владение определёнными техниками.

По сути Ad-hoc тестированием занимаются бета-тестировщики, они как раз понятия не имеют ни о техниках тестирования, ни о его методах и принципах.

ТЕСТИРОВАНИЕ ПО ДОСТУПУ К КОДУ

ТЕСТИРОВАНИЕ БЕЛОГО ЯЩИКА

(White Box Testing)

У тестировщика есть доступ к внутренней структуре и коду приложения. Это тестирование в основном проводится на модульном уровне. Тестирование белого ящика выполняется разработчиками, реже тестировщиками (обязательно имеющими навыки программирования).

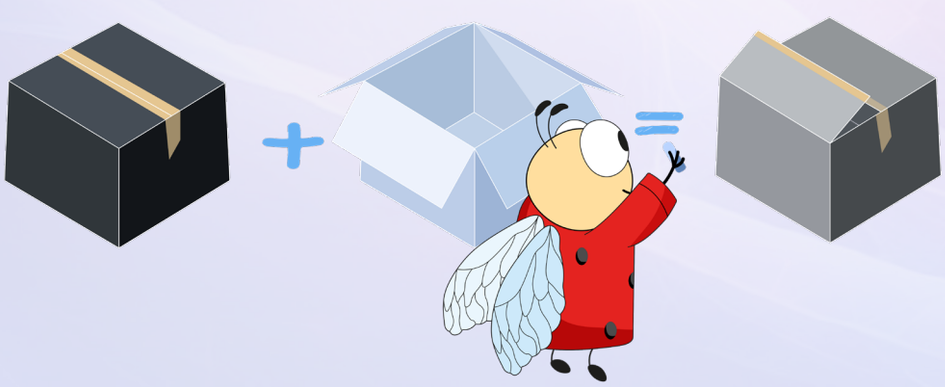
Некоторые авторы склонны связывать этот метод со статическим тестированием, но ничто не мешает запустить код на выполнение и при этом периодически обращаться к самому коду.

ТЕСТИРОВАНИЕ ЧЁРНОГО ЯЩИКА

(Black Box Testing)

У тестировщика либо нет доступа к внутренней структуре и коду ПО, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.

Большинство видов тестирования работают по методу чёрного ящика, где тестировщик оказывает на ПО воздействие (и проверяет реакцию) тем же способом, каким при реальной эксплуатации на приложения воздействовали бы пользователи.



ТЕСТИРОВАНИЕ СЕРОГО ЯЩИКА

(Grey Box Testing)

Комбинация методов белого и чёрного ящиков, к части кода и архитектуры у тестировщика доступ есть, а к части – нет.

Основная область применения – интеграционное тестирование. Примерами методов тестирования серого ящика являются тестирование API и запросы к базе данных.

Однако. В классификации ISTQB (международная система квалификации тестировщиков ПО, унифицирующая стандарты и подходы к тестированию) такого вида тестирования не существует.

ТЕСТИРОВАНИЕ ПО ЗАПУСКУ КОДА НА ИСПОЛНЕНИЕ

Тестирование не всегда предполагает взаимодействие с работающим приложением. Отсюда и классификация тестирования по запуску кода на исполнителя.

СТАТИЧЕСКОЕ ТЕСТИРОВАНИЕ

(Static Testing)

При статическом тестировании код не выполняется, то есть программа не запускается. При этом, само тестирование может быть как *ручным*, так и *автоматизированным*.

Проверяется:

- **Документы** (требования, тест-кейсы, описания архитектуры приложения, схемы баз данных и т.д.).
- **Графические прототипы** (дизайн).
- **Код приложения** (выполняется программистами в рамках аудита кода (code review), являющегося специфической вариацией взаимного просмотра в применении к исходному коду).

*Статическое тестирование **начинается на ранних этапах** жизненного цикла ПО и является, соответственно, частью процесса **верификации** (насколько продукт соответствует требованиям и спецификации).*

ДИНАМИЧЕСКОЕ ТЕСТИРОВАНИЕ

(Dynamic Testing)

Тестирование с запуском кода на исполнение, как всего приложения целиком (*системное тестирование*), так и нескольких взаимосвязанных частей (*интеграционное тестирование*), отдельных частей (*модульное тестирование*) и даже отдельных участков кода.

Простыми словами, динамическое тестирование – это метод оценки приложения путем выполнения его кода (то есть его фактическое использование) и наблюдения за его поведением в различных условиях.

Обычно динамическое тестирование используется после статического, то есть когда программный код уже был проверен на наличие очевидных ошибок.

*Динамическое тестирование включает в себя **тестирование ПО в режиме реального времени** путем предоставления входных данных и изучения результата поведения ПО. Проверка осуществляется с помощью заранее подготовленного набора тестов. Является частью процесса валидации программного обеспечения.*

КЛАССИФИКАЦИЯ ПО СТЕПЕНИ АВТОМАТИЗАЦИИ

РУЧНОЕ (МАНУАЛЬНОЕ) ТЕСТИРОВАНИЕ

(Manual Testing)

Это подход, который требует человеческого вмешательства на каждом этапе тестирования. Он включает идентификацию ошибок, генерацию входных данных и оценку полученных результатов, документацию обнаруженных багов.

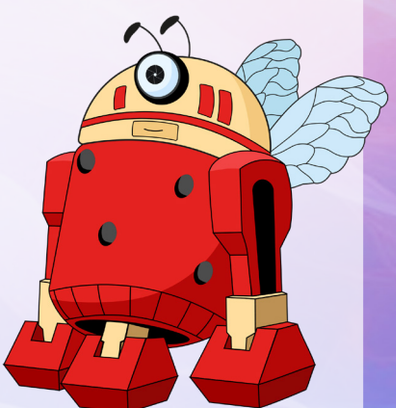
При ручном тестировании специалист взаимодействует с ПО как обычный пользователь, но наблюдает за результатом и проверяет, нет ли ошибок или неожиданного поведения.

Ручное тестирование особенно полезно для тестирования удобства использования и проверки сложных и нестандартных сценариев. Но так же позволяет проверить другие аспекты программы, такие, как внешний интерфейс, функциональность и взаимодействие ПО с пользователем, и многое другое.

Ручное тестирование является дешевым, но оно может быть подвержено человеческим ошибкам.

АВТОМАТИЗИРОВАННОЕ ТЕСТИРОВАНИЕ

(Automated Testing)



Метод тестирования ПО, при котором **проверки выполняются с использованием программных средств** для прохождения тестовых сценариев, с автоматически формируемыми отчетами о результатах тестирования.

При автоматизированном тестировании заранее подготовленные тесты можно выполнять многократно с минимальным вмешательством человека, обеспечивая быстрые результаты при тестировании большого объема данных.

Автотесты позволяют проводить интеграционное, дымовое и модульное тестирование. Особенно хорошо они подходят для повторяющихся тестов, таких, как регрессионное тестирование и тестирование производительности. А нагрузочное тестирование – это тип тестирования, который можно выполнить только автоматизированно.

Автоматизированное тестирование требует первоначальных вложений, но в долгосрочной перспективе экономит время и ресурсы за счет повышения точности и быстродействия машин.

ТЕСТИРОВАНИЕ API

API – это набор правил, с помощью которых разные программы общаются друг с другом. *Подробнее здесь.*

Основная задача тестирования API – гарантировать правильное взаимодействие между компонентами системы. При подключении API к программе могут возникать разные проблемы (ошибки в коде API или неправильное его подключение). **Задача тестировщика** убедиться, что все работает как надо, либо найти эти проблемы для их устранения разработчиками.

Тестирования API включает проверки:

- **Тестирование методов**. Каждый метод проверяют по отдельности, чтобы убедиться, что он работает правильно и возвращает ожидаемые результаты. *Про методы здесь.*
- **Взаимодействие API** с другими API, компонентами ПО и сервисами. Проверяется успешность отправки и получения данных, а также отработка различных сценариев использования.

  *Тестирование интеграции API можно проводить с помощью **Mock-тестирования**.*
- **Авторизация и аутентификация**.
- **Обработка ошибок**. API должен правильно обрабатывать **исключения** и передавать в программу верные коды ошибок.

  ***Исключения в API** – это ситуации, когда во время работы приложения возникают исключительные ситуации, например, когда у пользователя отсутствует доступ или он передал некорректные данные.*
- **Производительность**. Проверка работы API при повышенных нагрузках, его пропускной способности и производительности.
- **Безопасность**. Проверка уязвимости безопасности API, для предотвращения несанкционированного доступа и утечки данных.

МОСК-ТЕСТИРОВАНИЕ

(Mock Testing)

Mock-тестирование позволяет создавать имитации API для тестирования без необходимости взаимодействия с реальными системами. То есть реальные компоненты системы заменяются дублерами (моками). Ими могут быть фейковые базы данных, почтовые серверы и другие сложные системы. Тестовые объекты лишь подражают настоящим, но не содержат реальной логики или данных. Для Mock-тестирования можно использовать различные инструменты, например, **Postman** или **SoapUI**.

МОБИЛЬНОЕ ТЕСТИРОВАНИЕ

Это процесс тестирования приложений для современных мобильных устройств на функциональность, удобство использования, производительность и др.

Например, проверке подвергаются:

- **Функциональность.** Проверка работы функционала ПО.
- **Удобство использования**. Достаточно ли простое и понятное приложение в использовании.
- **Совместимость**. Установка на разные операционные системы, платформы, модели устройства и т.п.

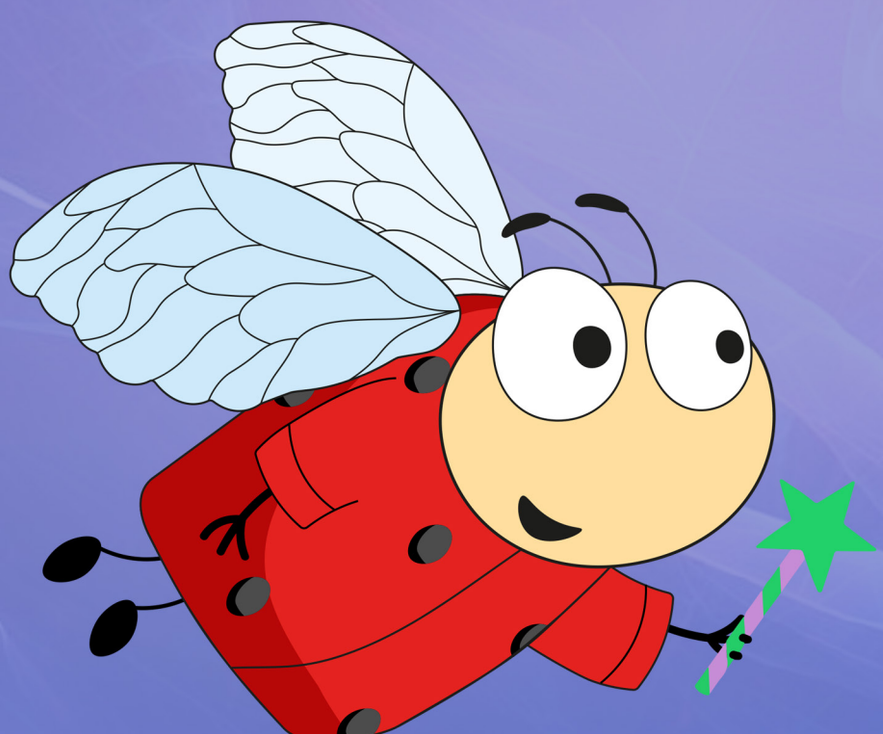
  Кроссбраузерность *– способность ресурса отображаться одинаково и работать во всех популярных браузерах без перебоев в функциональности.*

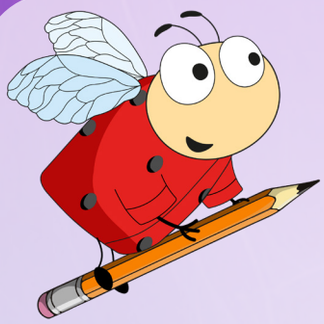
  Кроссплатформенность *– способность ПО работать с несколькими аппаратными платформами или ОС.*
- **Безопасность.** Проверка на уязвимости.
- **Локализация.**
- **Тестирование прерываний.** Проверка реакции мобильного приложения на прерывание и возвращение к своему предыдущему состоянию. **Примеры прерываний**:
  - *Низкий заряд батареи.*
  - *Входящий звонок/смс.*
  - *Входящее оповещение из другого приложения.*
  - *Включение/выключение зарядки.*
  - *Устройство выключено.*
  - *Напоминания об обновлении приложения.*
  - *Аварийная сигнализация.*
  - *Потеря/восстановление сетевого подключения.*
- **Подключение**. Проверка на разных типах подключения (wi-fi, мобильная сеть), переключение типов и оффлайн работа.
- **Производительность.**

Для тестирования мобильного приложения используют как физические устройства, так и симуляторы и эмуляторы. ***Эмуляторы*** *– это виртуальные устройства, которые имитируют программную и «железную» части физического устройства.* ***Симуляторы*** *– имитируют оболочку.*

Тестирование прерываний применимо и к десктопным приложениям со своими особенностями, например: адаптация к разным размерам экранов; работа приложения в режимах энергосбережения, спящем режиме, гибернации и быстрого запуска; обработка перехода в фоновый режим и др.

Тест-дизайн





ТЕСТ-ДИЗАЙН
(Test Design)

Тест-дизайн – это процесс разработки техник и методов тестирования, суть которых – оптимизация процесса тестирования.

Цель тест-дизайна – за минимальное количество проверок обеспечить плотное покрытие тестами проверяемого функционала продукта.

Техники тест-дизайна содержат рекомендации, советы и правила, которых стоит придерживаться при разработке тестов.

При этом разные тестировщики, работая над одним проектом, могут создать (и создают) различный набор тестов. Но именно этот интересный нюанс помогает обойти парадокс пестицида.

Хороший тестировщик, учитывая особенности продукта, на интуитивном уровне подбирает техники тест-дизайна, при необходимости комбинируя их. Интуиция тестировщика – это навык, который нарабатывается с опытом и приобретенными знаниями. На основе одного только опыта тестировщика существует отдельная техника тест-дизайна – **Предугадывание ошибок**.

ПРЕДУГАДЫВАНИЕ ОШИБОК
(Error Guessing)

Метод основан на интуиции тестировщика, подкрепленной знаниями о работе приложения в прошлом и о типах дефектов, которые были обнаружены в схожем программном обеспечении. То есть **тестировщик интуитивно понимает, где могут быть проблемные места в приложении**.

Предугадывание ошибок не имеет явных правил тестирования и часто не документируется или документируется способом, понятным и доступным для воспроизведения только автору.

Предугадывание ошибок не заменяет формальные методы тестирования, а дополняет их.

Error Guessing vs *Ad-hoc Testing*

Важно отличать предугадывание ошибок и исследовательское тестирование. В то время как первое основывается на обоснованных предположениях, для второго характерно отсутствие предопределенных гипотез.

ДОМЕННОЕ ТЕСТИРОВАНИЕ
(Domain Testing)

Домен (Domain) – **предметная область**, которая описывает совокупность проблем и целей бизнеса компании-заказчика IT-продукта. Другими словами – это область знаний, в которой функционирует бизнес.

*Знание **предметной области** тестируемого ПО играет важную роль в проведении качественной проверки его работоспособности. То есть тестировщик должен обладать базовыми знаниями о сфере, в которой приложение призвано решать поставленные бизнесом задачи.*

***Например**, если компания занимается интернет-банкингом, тестировщик должен иметь хорошее представление о таких операциях, как вход в систему, оплата счетов, переводы и т.п.*

Доменное тестирование (доменный анализ) – техника, направленная на создание эффективных тест-кейсов в случае, когда несколько переменных могут или должны быть протестированы одновременно. То есть – это проверка работы системы на ввод корректных данных и на её реакцию на ввод недопустимых значений.

Это тестирование представляет собой комбинацию из двух техник: *эквивалентное разбиение* и *анализ граничных значений*, с использованием *негативного тестирования*.

***Допускается проведение нескольких позитивных тестов** (например, ввод данных в несколько полей), **в одной проверке сразу**. Если что-то пойдёт не так, всегда можно декомпозировать проверки, чтобы понять, где именно система реагирует неверно.*

*Однако, считается, что **негативные тесты комбинировать нельзя**, так как теряется возможность отследить, при каких условиях произошла ошибка. Но альтернативщики уверяют, что именно в комбинации нескольких негативных данных может вылезти баг.*

Domain** vs **Domain

Домен, как предметная область, не имеет отношения к домену, который является основной частью веб-адреса сайта в Интернете.

ЭКВИВАЛЕНТНОЕ РАЗБИЕНИЕ
(Equivalence Partitioning)

Эта техника исследует различные входные данные, необходимые для использования ПО. Но сначала эти **данные разбиваются на** группы, так называемые **классы эквивалентности**, и уже внутри этих классов выбирается значения для проверок.

Принято считать, что **любой тест, выполненный из одного и того же класса эквивалентности, приведет к точно такому же результату, как и выполнение всех остальных тестов из этого же класса**. То есть нет смысла тестировать все значения внутри одного класса, но это надо сделать по крайней мере один раз в каждом классе эквивалентности.

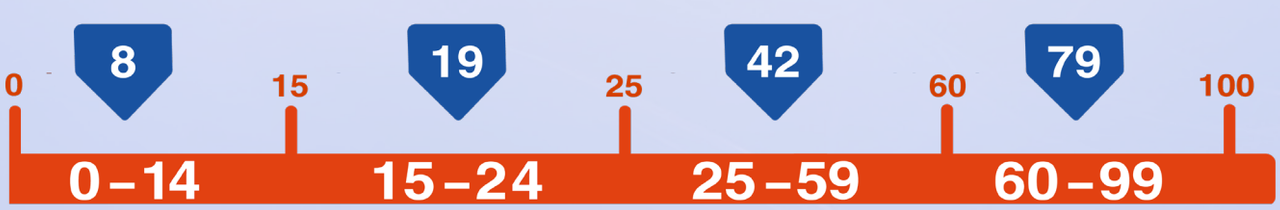
ПРИМЕР 1 Приложение позволяет пользователю вводить **пароль длиной от 7 до 15 цифр** (минимум 7 и максимум 15 цифр). Классы эквивалентности разбиваются следующим образом:

- Неверный **класс эквивалентности <7**
- Действительный **класс эквивалентности 7-15**
- Неверный **класс эквивалентности >15**

ПРИМЕР 2 Дано четыре возрастные группы получателей суслико-пособия. Эти группы и есть классы эквивалентности. До рождения и после 100 лет – суслики не положены.

- **С рождения до 14 лет** (включительно) – 5 сусликов в месяц
- **От 15 до 24 лет** (включительно) – 3 суслика в месяц
- **От 25 до 59 лет** (включительно) – 1 суслик в месяц
- **От 60 до 99 лет** (включительно) – 20 сусликов в месяц

Синими маркерами помечены рандомно выбранные проверки внутри классов: 8, 19, 42 и 79 лет.



На картинке представлено разбиение только на корректные классы, то есть позитивные проверки. Негативные тесты – это классы со значениями **<0** или **>100**.

АНАЛИЗ ГРАНИЧНЫХ ЗНАЧЕНИЙ

(Boundary Value Analysis)

Это **техника проверки ошибок на границах классов эквивалентности**. В основе лежит идея о том, что программа может сломаться в области граничных значений.

Канонически (по ISTQB) принято выделять у каждого класса эквивалентности **четыре значения**:

1. Нижняя граница
2. Верхняя граница
3. Нижняя граница минус 1
4. Верхняя граница плюс 1

Другой подход, с большим покрытием. **Каждая граница имеет по три значения**:

1. Нижняя граница минус 1
2. Нижняя граница
3. Нижняя граница плюс 1
4. Верхняя граница минус 1
5. Верхняя граница
6. Верхняя граница плюс 1

ПРИМЕР Продолжение *примера 2* из темы про эквивалентное разбиение, где рассматриваются получатели суслико-пособия. Упростим разбиение до допустимых и не допустимых значений. Где ноль, это начало отсчета жизни – то есть рождение.

- **Пособие положено** – с рождения до достижения 100 лет.
- **Суслики не положены** – до рождения и после 100 лет .

Нижняя граница:

- Негативные тесты *(недопустимые значения)* – **-1**
- Позитивные тесты *(допустимые значения)* – **0** и **1**

Верхняя граница:

- Позитивные тесты *(допустимые значения)* – **98** и **99**
- Негативные тесты *(недопустимые значения)* – **100**

*Если определять тестируемые данные по ISTQB, то значения **1** и **98** проверке не подлежат.*

*Ряд авторов отмечает, что **необходимость тестировать граничные значения доказывает неэффективность эквивалентного разбиения**. Связано это с тем, что часть граничных значений, находящихся непосредственно на границах классов, технически принадлежат этим классам эквивалентности. **И соответственно тестируется более одного значения каждого класса**.*

СОСТОЯНИЯ И ПЕРЕХОДЫ
(State & Transition)

Метод тестирования, при котором **изменения, вносимые во входные условия, вызывают изменения состояния** или выходных данных в тестируемом приложении. То есть тестирование объекта, который **может менять своё состояние** в зависимости от действий, которые с ним совершают.

ПРИМЕР 1 Вода (объект) имеет **жидкое состояние**. После воздействия высокой температуры **переходит в газообразное состояние**.

Элементы метода «Состояния и переходы»:

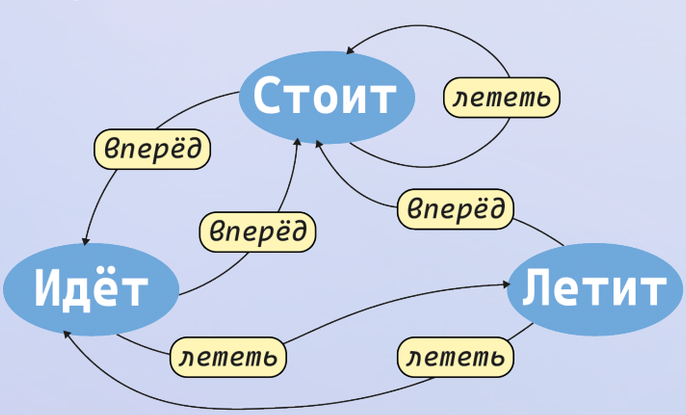
- **Состояния.** Режимы, в которым может находиться объект.
- **События**. Стимулы, заставляющие объект (систему) переходить из одного состояния в другое. Например «нажатие кнопки» или «получение сообщения».
- **Переходы**. Изменение состояния системы в ответ на внешнее событие. Например, система может переключиться в состояние «вход в систему» после события «нажатие кнопки».

Этапы тестирования:

- **Исследовать компонент** и собрать информацию.
- **Нарисовать диаграмму** и описание модели компонента.
- **Создать таблицу** на основе диаграммы.

ПРИМЕР 2 Представим, что Федя – персонаж компьютерной игры.
У Феди есть **три состояния: Стоит, Идёт, Летит**
У пользователя есть **две кнопки**:

- **ВПЕРЕД** Федя движется при коротком нажатии кнопки. Останавливается при повторном нажатии.
- **ЛЕТЕТЬ** Работает от короткого нажатия, но только если Федя уже идёт. Если Федя летит, то нажатие переведет его в шаг.



	Нажать кнопку Вперед	Нажать кнопку Полёт
Стоит	*Идёт*	*Стоит*
Идёт	*Стоит*	*Летит*
Летит	*Стоит*	*Идёт*

Данная техника подходит, когда необходимо покрыть функциональность *смоук-тестами*.

ТАБЛИЦА РЕШЕНИЙ
(Decision Table)

Таблица решений также известна, как **причинно-следственная таблица** (однако некоторые источники разделяют эти техники).

Таблица решений представляет собой визуальное отображение поведения ПО в зависимости от заданных условий. Составляется на основе требований, где указываются входные **условия** (причины) и их **результаты** (действия/следствия).

Таблица позволяет тестировщику не держать в голове все возможные варианты комбинаций входных условий, а **каждый её столбец – это по сути готовый тест-кейс**.

Где применять таблицу решений?

- **Там, где комбинации множества условий определяют поведение продукта**. Например, рассчёт скидки в магазине.
- **Когда есть проблемы с документацией**, документации нет вовсе или она содержит неточности.
- **В условиях постоянно меняющихся требований**.

По канону (ISQB) таблица заполняется бинарными значениями, такими как: да/нет, true/false или +/-, но иногда значений может быть больше.

ПРИМЕР Составление таблицы решений для проверки применения скидки в корзине покупателя.

По горизонтали:

- **УСЛОВИЯ**, которые влияют на результат. Обычно прописываются в виде вопроса.
- **ДЕЙСТВИЕ** или следствие (описание ожидаемого результата). Обязательно утверждение и только одно.

По вертикали:

- **ПРАВИЛА** – комбинация входных условий, или проще **ТЕСТЫ**.

Условия	Тест_1	Тест_2	Тест_3	Тест_4	Тест_5	Тест_6	Тест_7	Тест_8
Пользователь авторизован?	+	+	+	+	-	-	-	-
Количество товара от 1 до 3 шт.?	+	-	-	-	+	-	-	-
Количество товара от 4 до 9 шт.?	-	+	-	-	-	+	-	-
Количество товара от 10 шт.?	-	-	+	-	-	-	+	-
Действия								
Скидка за авторизацию 3%	+	+	+	+	-	-	-	-
Скидка за количество (1-3 шт.) - 0%	+				+			
Скидка за количество (4-9 шт.) - 5%		+				+		
Скидка за количество (>10 шт.) - 10%			+				+	

МЕТОД ПОПАРНОГО ТЕСТИРОВАНИЯ

(Pairwise Testing)

Эта техника направлена на создание тестов таким образом, чтобы **каждое значение параметра хотя бы раз побывало в одной паре с каждым значением другого параметра**. Приверженцы этой техники ссылаются на теорию, которая гласит, что подавляющее большинство багов вызвано взаимодействием не более двух факторов (параметров). Баги, причиной которых явились комбинации трех и более параметров, встречаются реже и (если верить легенде) значительно менее критичны.

ПРИМЕР Федя тестит фичу **Конструктор бургера** в приложении ресторана **«Сытно в рассрочку»**. Параметры со своими значениями у бургера такие: **Мясо** *(кенгуру или суслик)*, **Булка** *(пшеничная или ржаная)*, **Наличие мух** *(есть или нет)*.

Алгоритм действий такой:

- Составляем таблицу, обычно в Excel или Google таблицах.
- Подбираем уникальные комбинации двух параметров из первых двух столбцов. Затем из второго и третьего. И наконец подбираем пары для первого и третьего столбца.
- Комбинируем полученные пары.

	Мясо	Булка	Мухи
1	*Кенгуру*	*Пшеничная*	*Есть*
2	*Кенгуру*	*Пшеничная*	*Нет*
3	*Кенгуру*	*Ржаная*	*Есть*
4	*Кенгуру*	*Ржаная*	*Нет*
5	*Суслик*	*Пшеничная*	*Есть*
6	*Суслик*	*Пшеничная*	*Нет*
7	*Суслик*	*Ржаная*	*Есть*
8	*Суслик*	*Ржаная*	*Нет*

- На выходе имеем четыре тест-кейса, вместо восьми.

	Мясо	Булка	Мухи
1	*Кенгуру*	*Пшеничная*	*Есть*
2	*Кенгуру*	*Ржаная*	*Нет*
3	*Суслик*	*Пшеничная*	*Нет*
4	*Суслик*	*Ржаная*	*Есть*

Для тестирования большого количества параметров и значений, вместо мозгового штурма, желательно пользоваться специальными сервисами, такими как: **PICT**, **Pairwise online tool** и др.

*Существует мнение, что **попарное тестирование не результативно**. Попарные баги могут представлять собой большинство ошибок, а могут и не представлять. На это влияют фактические зависимости между переменными в продукте. Результаты контролируемого исследования продемонстрировали идентичность количества найденных багов при попарном тестировании и рандомно выбранными значениями параметров для проведения тестов.*

Напоследок

ИНСТРУМЕНТЫ

Умение работать с разнообразными инструментами в настоящее время тестировщику просто необходимо. Инструменты, обозначенные ниже - это лишь малая часть того, что существует.

- **Управление тестированием.** Специальные программные средства коллективной работы, позволяющие управлять командой и задачами. Вся коммуникация осуществляется прямо в программе. Например: **Jira, YouTrack, Redmine, TestRail**.
- **Скриншоты и работа с ними.** В версиях Windows 10 и 11 скриншот можно сделать **сочетанием клавиш Win + Shift + S**, откроется программа «Ножницы». Другие программные средства: **Snagit, Monosnap, GreenShot**.
- **Генераторы данных.** Инструменты генерации тестовых данных. Например: **Mockaroo.com**, **Bugmagnet**, **Generatedata.com**.
- **Визуальное отображение и интеллект-карты.** Человеческий мозг лучше обрабатывает визуальную информацию, поэтому инструментами визуального отображения очень полезно пользоваться. Например: **Coggle**, **Xmind**, **Freemind**.
- **Чек-листы.** Самый простой и доступный инструмент, это **Google-таблицы**, но есть и специализированное ПО. Например: **Testpad, Sitechco, Teamsuccess**.
- **Тестирование API.** API не получится тестировать вручную, для тестирования API необходимы инструменты. Например: **Postman, SoapUI, JMeter, REST-assured, Swagge**r.
- **Брокеры сообщений.** Технология, обеспечивающая связь между приложениями и их компонентами. Например: **Apache Kafka, RabbitMQ**.
- **Контейнеризация приложения.** Технология контейнеризации упрощает разработку и развертывание приложений изолировано от операционной системы. Например: **Docker, Kubernetes**.
- **Снифферы.** Инструменты для регистрации, анализа и изменения запросов между веб-приложением и сервером с помощью перехвата HTTP и HTTPS трафика. Например: **Fiddler, Charles Proxy, Proxyman**.
- **Просмотр макета/прототипа.** Помогают сверить размеры, отступы, шрифты, цвета элементов и ряд других параметров пользовательского интерфейса. Например: **Figma**.
- **DevTools/Инструменты разработчика.** DevTools помогает проводить функциональное и UI-тестирование сайта.

CONCLUSIONS | В ВОПРОСАХ

Для кого?

В интернетах можно найти миллион статей и видео на тему, «Как стать тестировщиком/программистом после 40 (лет) с нуля». Но в 99% случаев есть нюанс – эти «нулевые» товарищи, когда-то в молодости отучились на программиста, а некоторые даже работали по специальности, но после ушли работать в другую сферу деятельности. Но теперь преподносят себя, как «с нуля»...

Этот долгий заход к тому, что ***Шпаргалка в первую очередь рассчитана на помощь тем, кто действительно далек от мира IT****.*

Что ещё почитать из популярного?

Начну сначала с того, что ***читать не стоит*** *– широко известная книга Романа Савина «Тестирование dot com». Она когда-то была актуальна, но не сейчас. Сейчас это скорее художественная книга.*

Что ***рекомендую****. Если вы фанатеете от комиксов и прочих книг с картинками, и вам легче так воспринимать информацию, то книги* ***Ольги Назиной*** *ваш вариант.*

Если вам в кайф читать лонгриды без картинок, то советую учебники ***Станислава Куликова****.*

Где-то там написано не так, как здесь, почему?

Никакое из определений в этой Шпаргалке не является догмой*. У максимально большого количества определений в IT отсутствует однозначность трактования. А ещё в русскоязычном мире встречаются трудности перевода первоисточников, благодаря чему рождаются новые понятия, которые затем успешно тиражируются и даже обретают смысл.*

Что меня огорчило?

В прошлой версии Шпаргалки кто-то отрезал вступление с моими координатами и способами задонатить. Но не это самое печальное, отрезав вступление и перегнав шпору по новой в пдф – ***убили всю её интерактивность****.*

И это всё?

*Да. Но закончить хочу на фразе, найденной на просторах рунета – «****если бы мы пытались находить понятия в родном языке, а не заимствовали без перевода и понимания иностранные, то IT было бы менее загадочной областью****». (c) dvpingvin*

End!