**Теория тестирования**

**Что такое тестирование? С какой целью его проводят?**

Это проверка, насколько реальное поведение приложения совпадает с ожидаемым.

Цели:

**техническая**: предоставление информации о состоянии приложения на данный момент;

**коммерческая**: повышение лояльности к компании и приложению, так как любой обнаруженный дефект негативно влияет на доверие пользователей.

**Что делает тестировщик?**

Тестировщик сравнивает требования к приложению с тем, что получилось разработать. В процессе он обнаруживает ошибки и предлагает улучшения.

**Что такое QA, QC, Testing? В чём между ними разница?**

**QA** — Quality Assurance (обеспечение качества). Отслеживание процесса разработки приложения. Тестировщик ищет возможные проблемы и предлагает решения. Например, разработчики не переводят баг в трекере в статус «Тестирование», и тестировщики не понимают, что баг исправлен и его можно проверять.

**QC** — Quality Control (контроль качества продукта). Анализ результатов тестирования. Тестировщик делает вывод, готов ли продукт к релизу. Например, сообщает, что в приложении есть блокирующие баги, и релиз придется отложить.

Testing (тестирование). Сам процесс проверки. Тестировщик выполняет подготовленные тесты. Например, проверяет новую форму регистрации.

**QA** включает в себя QC и Testing.

**QC** включает в себя Testing.

**Что такое верификация и валидация?**

**Верификация** — подтверждение, что функциональность работает согласно требованиям.

Например, команда проверяет, что новый баннер с акциями на главной странице такой же, как на макетах: по ширине, высоте, цвету и содержанию.

**Валидация** — подтверждение, что функциональность выполняет ту цель, которую в неё закладывали изначально.

Например, команда проверяет, что новый баннер с акциями действительно привлекает больше пользователей совершать покупки.

**Расскажи про принципы тестирования.**

* **Все протестировать невозможно** Тестирование всех комбинаций входных значений и предусловий неосуществимо — за исключением тривиальных случаев.
* **Тестирование показывает наличие дефектов, а не их отсутствие** Тестирование снижает вероятность, что в приложении есть баги, но не доказывает, что их нет.
* **Раннее тестирование экономит время и деньги** Нужно начинать тестирование на ранних этапах разработки приложения — чем раньше приступить, тем дешевле будет исправить дефект.
* **Дефекты группируются вместе** Небольшое количество модулей приложения содержит большинство дефектов.
* **Нужно остерегаться** «**парадокса пестицидов**» Через некоторое время пестициды перестают влиять на насекомых. То же самое с тестами и багами 🐞. Если одни и те же тесты повторяются снова и снова, то в конечном итоге они больше не обнаруживают новых дефектов. Чтобы это исправить, нужно изменить существующие тесты и тестовые данные, а также написать новые тесты.
* **Тестирование зависит от контекста** Тестирование проводится по-разному в разных контекстах. Например, жизненно важное медицинское ПО тестируют не так, как мобильное приложение пиццерии.
* **Отсутствие ошибок — это заблуждение** Тестировщики не смогут выполнить все возможные тесты и найти все возможные дефекты. А если бы даже смогли, это необязательно обеспечит успех приложения, потому что оно может оказаться неудобным или просто уступать конкурентам.

**Методы, типы и уровни тестирования**

**Какие есть типы/виды тестирования?**

Их три:

* **функциональное** тестирование,
* **нефункциональное** тестирование,
* тестирование **изменений**.

**Какие есть методы тестирования?**

Методов четыре:

* метод **белого ящика**,
* метод **черного ящика**,
* метод **серого ящика**,
* метод, основанный **на опыте** (например, сюда относится исследовательское тестирование).

***Какие есть уровни тестирования?***

Существует четыре уровня тестирования:

* компонентное,
* системное,
* приемочное,
* интеграционное.

**Что такое регрессионное тестирование?**

Изменение, внесенное в одну часть приложения, может случайно повлиять на поведение других частей. Такие побочные эффекты называют регрессиями. А регрессионное тестирование — это проверка всего приложения с целью убедиться, что регрессий нет.

**Что такое функциональное тестирование?**

Функциональное тестирование проверяет, корректно ли работает **функциональность приложения**.

***Что такое нефункциональное тестирование?***

Нефункциональное тестирование направлено на проверку тех **частей приложения**, которые **не относятся к функциональности**. Например, насколько быстро загружаются данные или как много пользователей могут подключиться одновременно.

**Что такое интеграционное тестирование?**

Интеграционное тестирование проверяет, **как приложение взаимодействует с другими приложениями**. Например, можно протестировать интеграцию Яндекс Самоката и банковского приложения: получается ли оформить заказ в рассрочку.

**Что такое системное тестирование?**

На системном уровне приложение **тестируют целиком** — с учётом функциональных и нефункциональных требований.

**Что такое смоук-тестирование?**

Цель смоук-тестирования (англ. smoke testing), или дымового тестирования — проверить **работоспособность основных функций приложения**. Обычно это функциональность, которая критически важна для взаимодействия между пользователем и приложением. Например, можно протестировать, открывается ли приложение, полностью ли загружается интерфейс, отвечает ли база данных.

**Что такое компонентное тестирование?**

Компонентное, или модульное тестирование проверяет **только часть приложения**.

**Что такое санитарное тестирование?**

Санитарное тестирование (англ. sanity testing) проверяет **работоспособность отдельных важных компонентов приложения**. Например, можно проверить, отправляются ли нотификации, проводится ли оплата товара в интернет-магазине.

**Что такое исследовательское тестирование?**

В исследовательском тестировании **нет тестовой документации**, которую собрали заранее. Продукт проверяют и изучают одновременно. При этом каждый последующий тест выбирают на основании предыдущей проверки.

**Что такое тестирование безопасности?**

Нефункциональное тестирование, при котором тестировщик пытается **атаковать систему**, **чтобы найти в ней уязвимости**. Например, проверяет, легко ли взломать личный кабинет и получить доступ к личным данным пользователей.

**Что такое «тестирование белым (стеклянным) ящиком»?**

При проверке методом белого ящика тестировщик **анализирует код приложения и логику работы бэкенда**. Чтобы проектировать и выполнять такие тесты, придётся разобраться в языках программирования.

Этот метод позволяет:

* убедиться, что выходные данные из кода соответствуют ОР;
* обнаружить уязвимости в структуре кода.

Важное преимущество — для тестирования не нужен готовый фронтенд. Ошибки можно обнаружить и исправить до завершения разработки, просто изучив код.

**Что такое «тестирование черным ящиком»?**

Тестирование методом чёрного ящика — это проверка приложения только **по макетам и требованиям**. Тестировщик **не смотрит в код** и может не знать, как устроен бэкенд. Так можно сконцентрироваться на пользовательских сценариях, а не на внутренней логике приложения.

**Что такое «тестирование серым ящиком»?**

Метод серого ящика **совмещает методы чёрного и белого** ящиков. Тестировщик работает с требованиями и макетами, но при необходимости обращается к коду.

**В чем разница между кроссбраузерным и кроссплатформенным тестированием?**

Кроссбраузерное тестирование — проверка, что во всех **популярных браузерах** приложение работает и выглядит так, как описано в требованиях и макетах.

Кроссплатформенное тестирование — проверка, что приложение соответствует требованиям и макетам на **любых платформах** и ОС.

Что такое «end-to-end (сквозное) тестирование»?

Сквозное тестирование (End-to-end, E2E, Chain testing) проверяет работу приложения **от начала до конца**, **а также его интеграцию с другими сервисами**. Цель такого тестирования — убедиться в целостности данных и связи приложения с другими системами, интерфейсами и базами данных.

**Тестовая документация**

**Что такое тест-план? Из каких элементов он состоит?**

Тест-план — это документ, который **содержит**:

* **данные** о том, что, как и когда нужно протестировать;
* доступные **ресурсы** и потенциальные **риски**;
* критерии для **входа** и **выхода** из тестирования: когда начинать и заканчивать проверку;
* **планы** на **регресс** и **ретест** багов;
* планы на **автоматизированное тестирование**.

**Что такое «парадокс пестицида»?**

Через некоторое время пестициды перестают влиять на насекомых. То же самое с тестами и багами 🐞. Если одни и те же тесты повторяются снова и снова, то в конечном итоге они больше не обнаруживают новых дефектов. Чтобы это исправить, нужно **изменить существующие тесты** и тестовые данные, а также написать новые тесты.

**Что такое отчёт о тестировании и из каких элементов он состоит?**

После проверки должен остаться **документ, который содержал бы в себе всю информацию о проделанной работе.**

Вот необходимые пункты для такого отчёта:

* название фичи;
* кто проводил тестирование;
* период тестирования;
* использованные девайсы и версия сборки;
* выполненные проверки (обычно это набор тест-кейсов);
* блокирующие и критичные ошибки (баги);
* вывод: принята фича или нет.

В некоторых случаях стоит приложить фото или видео, доказывающие работоспособность фичи.

Строгих правил для оформления отчётов нет: главное — отразить всю важную информацию о тестировании.

**Отчёт о тестировании релиза**

Релиз с точки зрения тестировщика обычно состоит из нескольких фич, проверки багов и регресса. Поэтому отчёт о тестировании релиза — это документ с информацией о проверках, которые были выполнены на релизной сборке, и о состоянии продукта в целом.

В отчёте указывают:

* период тестирования;
* состав команды тестирования;
* версию релизной сборки и девайсы, на которых проводились проверки;
* ссылки на результаты регрессивного тестирования;
* ссылки на результаты автоматического тестирования;
* количество ошибок с указанием их приоритетности, ссылки на баг-репорты;
* вывод: можно выпускать релиз или нет.

**Что такое баг-репорт? Какие у него основные элементы?**

Баг-репорт — технический документ, в котором тестировщик подробно описывает ошибку, найденную в приложении. **Основные элементы** баг-репорта: **заголовок, шаги воспроизведения, ФР, ОР, окружение, приоритет, предусловия, постусловия, дополнительные материалы.**

**Что такое ОР и ФР? В чём разница?**

Ожидаемый результат (**ОР**) — то, как приложение **должно работать** согласно требованиям.

Фактический результат (**ФР**) — то, как приложение **работает** на самом деле.

**В чем разница между чек-листом и тест-кейсом? Из каких элементов они состоят?**

**Чек-лист** — список проверок, а **тест-кейс** — подробное пошаговое описание пункта из этого списка. На один пункт чек-листа может приходиться несколько тест-кейсов.

**В каких случаях больше подходит чек-лист, а в каких — тест-кейсы?**

Используй чек-листы в следующих ситуациях:

* Требования постоянно меняются. Нет смысла тратить время на детальное описание, если всё может поменяться.
* Мало времени на тестирование. Если не успеваешь написать тест-кейсы, а задача срочная, подойдёт чек-лист. Позднее его можно превратить в тест-кейсы.
* Несложная, неважная или редкая проверка. Например, изменили цвет кнопки оформления заказа. На саму кнопку заказа тест-кейс уже есть — можно обойтись чек-листом.

Во всех остальных случаях лучше прописывать тест-кейсы: с детальным описанием ты точно не забудешь о важных нюансах.

Если времени мало, сначала напиши чек-лист, а позднее детально распиши отдельные пункты.

При этом превращать в тест-кейсы все пункты не стоит, на их поддержку нужно время. Важно соблюдать баланс, чтобы тестирование не превратилось только в работу с документацией.

**Процессы**

**Опиши жизненный цикл разработки и выпуска фичи в релиз.**

**Фича** (feature — особенность) — изменение или дополнение к функциональности приложения.

Например, менеджер хочет добавить в приложение Яндекс Самокат фичу — поиск самоката по сигнализации. Прежде чем эта задача попадёт к тестировщику, она пройдёт несколько этапов разработки — **жизненный путь**.

Сначала менеджер инициирует задачу.

Когда разработчики выполнят задачу, её передадут тестировщику. Тестировщик проверит приложение и сообщит о багах разработке.



**Что такое waterfall, scrum и agile? В чём их особенности?**

**Waterfall** (англ. «водопад») — линейная модель разработки. Ее отличают заранее определённые и зафиксированные процессы, сроки и бюджет. Такая модель плохо подстраивается под изменения требований.

**Scrum** — гибкая модель. В рамках Scrum работа делится на двухнедельные отрезки — спринты. За один спринт команда должна подготовить к релизу определённую функциональность.

**Agile** — набор принципов разработки. На их основе появились гибкие модели.

**Какие существуют этапы тестирования фичи?**

Тестировщик проходит следующие этапы:

* выполняет тест-анализ требований;
* проектирует тесты;
* устанавливает сборку на тестовый стенд;
* выполняет проверки по своим тест-кейсам и чек-листам;
* оформляет баг-репорты, если есть ошибки;
* контролирует, чтобы блокирующие и критические баги исправили;
* сообщает команде, готова ли фича к релизу;
* переходит к следующей фиче.

**Какие существуют этапы тестирования релиза?**

При подготовке релиза тестировщик:

1. Планирует тестирование.
2. Готовится и выполняет проверки.
3. Составляет отчёт по результатам.

**Планирование**

Когда менеджер совместно с командой определил, какую функциональность реализовывать в следующем спринте, старший тестировщик планирует объём работы.

**Подготовка и тестирование**

Пока разработчики пишут код, тестировщики готовятся к тестированию:

* изучают требования;
* задают уточняющие вопросы;
* проектируют тесты: составляют чек-листы и тест-кейсы.

Когда код готов, тестировщики приступают к проверкам. Например, проводят смоук-тестирование и регрессионное тестирование.

**Составление отчёта**

По результатам тестирования готовят отчёт. В нём старший тестировщик указывает, сколько ошибок удалось найти, и оценивает готовность к релизу. Если приложение не готово, даёт рекомендации: например, исправить блокирующие баги и провести повторный регресс.

**Чем релиз отличается от патча?**

Релиз включает изменение функциональности и исправление багов, а патч — только исправление багов. Поэтому для него обычно не проводят регрессионное тестирование.

**Расскажи про критерии начала и окончания тестирования фичи.**

**Критерии начала тестирования**:

* написана большая часть кода или весь код;
* проведена декомпозиция требований и написаны тест-кейсы;
* приложение доступно для проверки на тестовом стенде;
* окружение работает исправно, все инструменты тестировщика доступны;
* определён список фич, которые нужно протестировать.

**Критерии окончания тестирования**:

* приложение работает согласно требованиям;
* все тест-кейсы пройдены;
* нет багов со средним или высоким приоритетом.

**Расскажи про критерии начала и окончания тестирования релиза.**

**Критерии начала тестирования** — это список условий, которые должны быть выполнены перед началом проверок. Например:

* требования и макеты разработаны,
* фичи реализованы,
* тестовая версия приложения готова,
* стенды настроены и готовы к раскатке тестовой версии.

**Критерии окончания тестирования** — это условия, которые должны быть выполнены перед завершением проверок. Их тоже прописывают в тест-плане:

* приложение работает согласно требованиям;
* все тест-кейсы пройдены;
* нет багов со средним или высоким приоритетом;
* проведено регрессионное тестирование;
* новых багов не найдено.

**Какие роли существуют в команде продукта? В чём особенности каждой?**

**Менеджеры**

Владелец продукта, продакт-оунер (Product Owner) — капитан корабля, который задаёт курс: решает, как должен выглядеть конечный продукт, доносит ключевые идеи до команды, определяет стратегию развития.

**Менеджер продукта**, продакт-менеджер (Product Manager) исследует рынок и аудиторию, придумывает, как улучшить метрики: например, увеличить прибыль компании и количество новых пользователей.

**Менеджер проекта, проджект-менеджер** (Project manager) координирует команду: ставит задачи разработке и тестированию, следит за их выполнением в срок.

**Другие роли**

Команда, которая непосредственно готовит приложение к релизу: **дизайнеры**, **разработчики** и **тестировщики**.

Когда приложение готово к выпуску, опытный разработчик или тестировщик берёт на себя роль **релиз-менеджера**: проверяет, всё ли готово, анонсирует релиз, отчитывается, что именно сделала команда.

После релиза пользователям помогает **сотрудник службы поддержки**: отвечает на вопросы, собирает отзывы и предложения, передаёт информацию об ошибках тестировщикам и разработчикам.

**Опиши жизненный цикл бага.**

Чтобы отслеживать, кто отвечает за определённую ошибку в каждый момент времени, багам проставляют статусы:

1. Открыт (open) — тестировщик обнаруживает баг и сообщает о нём команде. Дальше разработчики могут взять его в работу или отклонить.
2. В работе (in progress) — разработчик исправляет баг.
3. Исправлен (fixed) — разработчик устранил ошибку.
4. Тестируется (testing) — тестировщик повторно проверяет баг.
5. Переоткрыт (re-opened) — баг исправили не полностью или не исправили вообще.
6. Закрыт (closed) — баг исправили или решили не чинить.

На разных проектах названия статусов и их количество могут отличаться.

**Как понять, что баг исправлен?**

Убедиться, что:

1. Статус баг-репорта — «исправлен» или «готов к тестированию».
2. К баг-репорту приложен MR (merge request) или другой артефакт, который содержит код исправления.
3. Баг поправлен так, как описано в баг-репорте.

**Что делать, если проверка показала, что баг не исправлен? А если исправлен?**

Если баг не исправлен, тестировщик возвращает баг-репорт разработчику. В комментарии нужно написать, что баг не исправлен, и указать номер версии, на которой осуществлялась проверка.

Если баг исправлен, тестировщик закрывает баг-репорт. В комментарии нужно написать, что баг исправлен, и указать номер версии, на которой осуществлялась проверка.

**Работа с требованиями**

**Что такое бизнес-требования, системные требования и пользовательские требования? [2 спринт]**

**Бизнес-требования**

Это требования верхнего уровня. Обычно их формирует заказчик: коротко описывает идею приложения, объясняет, зачем и для кого его создавать. В таких требованиях нет деталей реализации, они нужны менеджерам и аналитикам.

Например, бизнес-требования к Яндекс Самокату могут выглядеть так:

Перед запуском нового сервиса нужно провести эксперимент: запустить промостраницу и проверить спрос. На промостранице пользователь сможет оформить предзаказ самоката.

**Пользовательские требования**

Пользовательские требования нужны дизайнерам, разработчикам и тестировщикам. Это тоже верхний уровень.

Например,

1. Время на таймере обновляется раз в секунду.
2. При нажатии кнопки «Предзаказ» страница прокручивается вниз до блока с формой предзаказа.
3. При нажатии кнопки «Подробнее» страница прокручивается вниз до блока с дополнительной информацией — «Новое поколение самокатов».

**Системные требования**

Системные требования — это нижний уровень. Они определяют, как будет устроено всё приложение: какие базы данных использовать для хранения информации о пользователях или какие ограничения есть в полях ввода.

Например:

При нажатии кнопки «Заказать» происходит валидация: приложение проверяет корректность заполнения полей, а также уникальность номера телефона и адреса электронной почты.

Ограничения полей:

* Имя. Только буквы русского и английского алфавитов. Длина не менее 2 и не более 25 символов.
* Электронный адрес. Только буквы английского алфавита, цифры, тире, точка, @. Длина не менее 5 и не более 50 символов.
* Номер телефона. Только цифры и знак +. Длина не менее 10 и не более 12 символов. Ввод символов начинается в скобках после +7.

**Опиши основные характеристики хороших требований. [2 спринт]**

1. **Завершённость**. Требования содержат исчерпывающую информацию о приложении.
2. **Ясность**. Требования должны быть понятными, ни один из пунктов не нужно додумывать.
3. **Корректность и согласованность**. Из требований понятно, как должно и не должно работать приложение. Логика работы разных функциональностей согласована между собой.
4. **Проверяемость**. Понятно, как проверить, выполняется ли требование.
5. **Необходимость и полезность при эксплуатации**. В документации нет лишних условий, которые не нужно проверять.
6. **Осуществимость**. Требования должны быть реалистичными. Это определяется балансом между ценностью и требуемыми ресурсами.
7. **Модифицируемость**. Набор требований должен быть таким, чтобы его можно было легко изменять и дорабатывать, не модифицируя требования в других местах.
8. **Прослеживаемость**. Легко отследить связь между требованиями и другими артефактами проекта — например, макетами.
9. **Упорядоченность**. По важности, стабильности и срочности исполнения.
10. **Отсутствие распространённых ошибок**.

**Дай определение серой зоны в требованиях. [2 спринт]**

**Серые зоны** — несостыковки, противоречия и пропуски в требованиях. Это обычная ситуация: менеджер может не указать детали, которые кажутся очевидными, или просто ошибиться по невнимательности. Задача тестировщика — проанализировать требования и уточнить всё, что осталось непонятным.

**Как найти серые зоны? [2 спринт]**

**Декомпозировать требования**. Разбирайся с каждым отдельным пунктом требований, но не забывай, что система работает как одно целое.

**Отклоняться от требований**. Важно понять, как отреагирует приложение, если требование не выполняется. Например, в требованиях к Самокату есть такой пункт: «Телефон и адрес электронной почты должны быть уникальными: их нельзя использовать повторно, чтобы оформить другой предзаказ». Что будет, если попытаться использовать один и тот же номер дважды?

**Визуализировать требования**. Составь mindmap, чтобы лучше представлять, как устроено приложение. Так ты сразу увидишь, если в требованиях чего-то не хватает. А ещё вспомни блок-схемы: ромб означает момент принятия решения, поэтому от него рисуют две стрелки с надписями «Да» и «Нет». Если стрелка одна — скорее всего, требования неполные.

**Задавать вопросы**. Во время тест-анализа тебе предстоит много общаться с командой.

**Что такое тест-анализ? Какова его цель? [2 спринт]**

**Тест-анализ** (test analysis) — один из этапов тестирования. Тестировщик изучает требования и макеты, чтобы ответить на вопрос: что именно предстоит тестировать. А потом составляет список объектов тестирования.

**Что такое матрица покрытия требований? [Нет ответа в программе]**

Матрица покрытия требований — это инструмент, который помогает тестировщикам понять, насколько требования покрыты тест-кейсами и надо ли то-то дописать или удалить.

Матрицу покрытия требований оформляют в виде таблицы. По горизонтали, в заголовках столбцов — требования, по которым надо написать тесты. По вертикали, в строках — тест-кейсы. В ячейках на их пересечении обозначают, покрывается требование тест-кейсом или нет.

**Что делать, если требований нет или они неактуальны? [2 спринт]**

**Требования есть, но они неактуальны**

Представь, что в твоей колонке «Можно тестировать» на доске с общекомандными задачами появилась новая карточка: «Поиск самоката по сигнализации». Разработчики написали код и передали в тестирование.

Требования к новой функциональности приложены прямо к карточке с задачей, но они неактуальны. В таком случае:

1. Не удаляй текст и не вноси правки самостоятельно. Команде пригодятся все **артефакты**, в том числе старые требования.
2. Укажи на несоответствия в комментариях к документу.
3. Покажи команде документ с комментариями и уточни, актуальна ли информация.
4. После этого напиши менеджеру или аналитику, который составлял требования.

**Требований нет**

Если на проекте есть требования, их всегда можно актуализировать. Но бывает, что требований нет совсем. Тогда команда может самостоятельно их собрать. Для этого нужно:

1. Определить целевую аудиторию — кто пользуется приложением?
2. Понять, как должно выглядеть приложение, какие задачи решать. Описать всё в отдельном документе.
3. Устроить командный созвон и договориться дополнять и редактировать требования общими усилиями.
4. Дополнять требования, опираясь на логику, общепринятые стандарты и профессиональный опыт коллег.
5. Документировать в требованиях все обновления приложения.
6. Регулярно отправлять документацию команде на ревью.
7. Чтобы не забывать вовремя вносить изменения, нужно завести на доске задачи на обновление требований. Иначе времени на документацию может не найтись — особенно в момент активной разработки фичи.

**Что такое объект тестирования? [2 спринт]**

**Объекты тестирования** (test objects) — части приложения, которые нужно проверить.

**Что такое декомпозиция требований? Как ее выполнять? [2 спринт]**

**Декомпозиция**

**Декомпозиция** — разделение целого на части. На этом этапе тебе предстоит разбить крупные объекты тестирования на более мелкие. Так с ними удобнее работать.

Декомпозиция требований — это подход тест-анализа, при котором тестировщик разбивает крупные объекты тестирования на более мелкие. Так проще проектировать проверки.

**Правило первое**

Обычно одно требование можно разложить на несколько частей.

Например, тебе удалось изучить требования к Яндекс Самокату и выделить объект тестирования — форму предзаказа. Форма состоит из нескольких полей ввода: «Имя», «Фамилия», «Отчество», «Email» и «Номер телефона».

**Правило второе**

Требования декомпозируют до атомарного уровня: так, чтобы поделить их ещё раз было уже нельзя.

Например, поле «Имя» на форме предзаказа можно декомпозировать так:

* русские буквы,
* английские буквы.

Это и есть атомарный уровень: разбить требование ещё раз не получится.

**Правило третье**

Требования декомпозируют в рамках существующего описания.

Например, если в требованиях к полю «Имя» не написано про заглавные и прописные буквы, то достаточно проверить любые буквы русского и английского алфавитов. Декомпозировать их до заглавных и прописных не нужно.

**Что делать, если в требованиях обнаружена ошибка? [Нет ответа в программе]**

1. Завести баг-репорт.
2. Сообщить автору требований (например, менеджеру или аналитику).
3. Убедиться, что ошибка в требованиях исправлена.

**Какие инструменты тестировщик использует при работе с требованиями? [2 спринт]**

* Системы хранения требований.
* Инструменты разработки mindmap и блок-схем для декомпозиции**.**

**Проектирование тестов**

**Зачем нужна тестовая документация? Какие виды тестовой документации существуют? [3 спринт]**

Есть три типа тестовой документации.

**Тест-план** — это документ, который содержит:

* данные о том, что, как и когда нужно протестировать;
* доступные ресурсы и потенциальные риски;
* критерии для входа и выхода из тестирования: когда начинать и заканчивать проверку;
* планы на регресс и ретест багов;
* планы на автоматизированное тестирование.

**Чек-лист** — список элементов системы, которые нужно проверить. Каждый пункт чек-листа обязательно содержит элемент.

**Тест-кейс** — это детальное пошаговое описание проверки. Оно показывает определённый путь программы или сценарий проверки.

**Что такое тест-дизайн? Какова его цель? [3 спринт]**

**Тест-дизайн** — проектирование тестовой документации.

Если тест-анализ — это ответ на вопрос «Что тестировать?», то тест-дизайн — «Как тестировать?».

**Перечисли самые известные техники тест-дизайна. [3 спринт]**

Классы эквивалентности, граничные значения, таблица принятия решений, попарное тестирование, диаграмма состояний и переходов, таблица состояний и переходов.

**Что такое классы эквивалентности? [3 спринт]**

**Класс эквивалентности** (КЭ) — это набор входных или выходных данных ПО, которые обрабатываются программой по одному алгоритму или приводят к одному результату.

**Как применять технику разбиения требований на классы эквивалентности? Какова цель этой техники? [3 спринт]**

Реакция приложения на все тестовые значения из одного класса эквивалентности одинаковая. Поэтому для проверки достаточно взять по одному значению из каждого класса — на остальные приложение должно отреагировать так же.

Это допущение позволяет сократить количество тест-кейсов и сэкономить время.

## Диапазон

Класс эквивалентности типа диапазон — это интервал значений.

* Все объекты этого класса можно сравнить между собой: например, отсортировать от меньшего к большему.
* Элементы такого класса могут быть представлены на числовой прямой, где разделением между классами будет условие требований.
* Любой объект класса можно сравнить с условием требований: определить, \*\*больше, меньше или равно значение из класса некоторому значению условия.

## Набор

Класс эквивалентности типа набор — это совокупность значений, объединенных одним свойством.

* Объекты внутри набора нельзя упорядочить или сравнить между собой.
* Объекты набора должны принадлежать множеству допустимых значений.
* Необходимо качественное, а не количественное сравнение значений набора с условием требования.

## Другие проверки

**Отрицательные и слишком большие значения.** Возраст не может быть отрицательным, но этот вариант нужно проверить. Например, пользователь может случайно ввести в поле -35. Программа определит такого пользователя как несовершеннолетнего. Важно уточнить у аналитика нижнюю границу допустимых значений — 0 или 1 год.

То же касается и слишком больших значений: вряд ли приложением будет пользоваться человек, которому 200 лет. Такой кейс — повод уточнить у аналитика, нужно ли установить ограничение на максимальный возраст.

**Буквы и спецсимволы.** Пользователь может ввести значения вроде «10 лет», «сорок два» или просто случайно поставить не тот символ — «24а». Для таких случаев тоже нужно предусмотреть класс эквивалентности.

**Вещественные значения.** Сейчас программа определит пользователя с возрастом 17,5 как несовершеннолетнего. Но корректно ли принимать такие значения?

**Ноль.** Данное число является нестандартным для кода. Во многих системах важно проверять это значение — хорошо, если оно вынесено в отдельный тест-кейс.

Чтобы выявить все КЭ, требование нужно максимально декомпозировать. Для каждого условия — свой класс.

**Что такое граничные значения? [3 спринт]**

**Граничные значения** (ГЗ) — это первая и последняя точки класса эквивалентности. Их важно проверять, потому что на границах часто возникают ошибки.

Обрати внимание: граничные значения есть только у диапазона. У наборов их не бывает.

Техника выделения ГЗ помогает проверить, корректно ли приложение обрабатывает границы КЭ, а также дополнить проверки КЭ типа «диапазон» тестами на границах.

Разные ошибки в условиях могут привести к разным последствиям. Техника выделения граничных значений помогает найти такие ошибки с минимальным количеством проверок. Нужно только выделить границы класса и взять по три значения на каждой из границ:

* значение на самой границе,
* значение около границы внутри класса,
* значение около границы за пределами класса.

Этого хватит, чтобы протестировать весь диапазон.

**Как выглядит алгоритм выделения граничных значений? Какая цель у этой техники тест-дизайна? [3 спринт]**

Разные ошибки в условиях могут привести к разным последствиям. Техника выделения граничных значений помогает найти такие ошибки с минимальным количеством проверок. Нужно только выделить границы класса и взять по три значения на каждой из границ:

* значение на самой границе,
* значение около границы внутри класса,
* значение около границы за пределами класса.

Этого хватит, чтобы протестировать весь диапазон.

**Как связаны техники КЭ и ГЗ? [3 спринт]**

**Граничные значения** (ГЗ) — это первая и последняя точки класса эквивалентности. Граничные значения есть только у классов эквивалентности типа **диапазон**. У **наборов** их не бывает.

Техника выделения граничных значений работает только в паре с определением классов эквивалентности. Она дополняет набор тестовыми данными.

**В каких случаях лучше использовать технику разбиения требований на классы эквивалентности, а в каких — граничные значения? [Нет ответа в программе]**

Разбиение на классы эквивалентности используется, когда требования можно декомпозировать на набор или диапазон значений.

Выделение граничных значений используется, когда требования уже разбиты на классы эквивалентности с типом диапазон.

**Как оптимизировать тесты? [3 спринт]**

Бывает, что после применения техник тест-дизайна остаются повторяющиеся тестовые значения. Перед проверкой их нужно сократить — это основной принцип оптимизации.

Чтобы оптимизировать проверки, пройдись по каждому тестовому значению и ответь на вопрос: оно принадлежит этому классу?

Алгоритм следующий:

* Если значение принадлежит тому классу, в котором указано, переходи к следующему значению.
* Если нет, найди класс, которому это значение должно принадлежать.
* Если подходящего класса нет, это ошибка — у тебя не хватает условия.
* Если подходящий класс есть, проверь: здесь указано тестовое значение?
* Если значение не указано в нужном КЭ — скорее всего, границы выделены неверно.
* Если КЭ есть, и его тестовые значения содержат тестовое значение — всё в порядке. Вычеркни это значение из неподходящего КЭ.

**Как применять технику попарного тестирования (pairwaise)? Для чего она нужна? [3 спринт]**

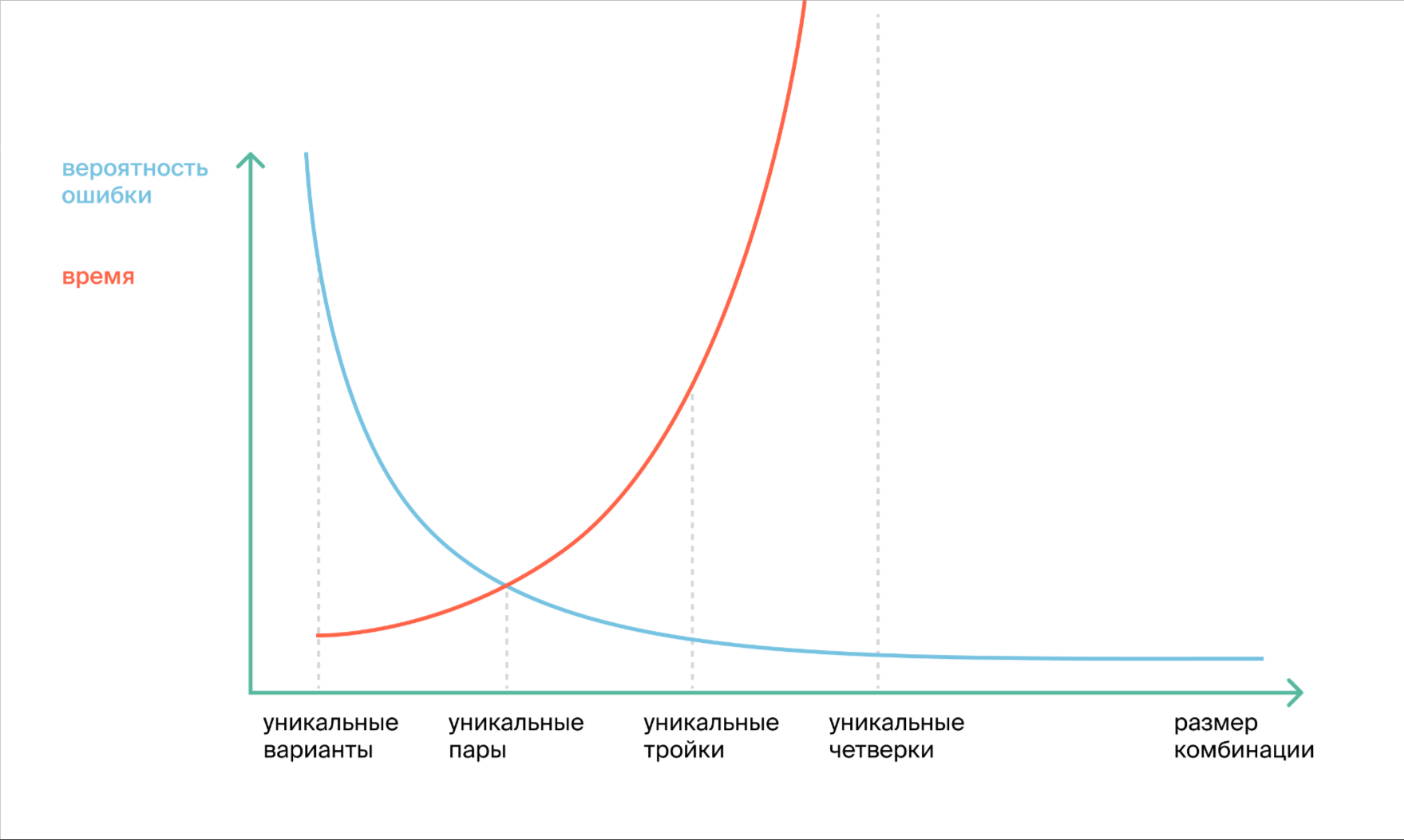
**Попарное тестирование** (pairwise testing) — техника тест-дизайна, в которой проверяют комбинации значений для каждой пары параметров.

В тестировании есть гипотеза, подтверждённая статистически: «Ошибки возникают в одном параметре чаще всего; реже — на стыке двух параметров; и редко — при комбинации трёх и более параметров».

Значит, если пара параметров вызвала ошибку в комбинации с другими параметрами, с большой вероятностью та же пара вызовет ошибку и в другой комбинации. Поэтому достаточно протестировать только уникальные пары вариантов.

Посмотри на график. Он показывает следующее:

* вероятность ошибки уменьшается, если проверить больше комбинаций;
* время на тестирование при этом увеличивается — чем больше комбинаций, тем больше времени понадобится.



### 

### **Как использовать технику попарного тестирования**

Таблица заполняется так же, как таблица принятия решений. Разница в том, что в ней указаны не все комбинации условий, а только уникальные пары.

Алгоритм такой:

1. Сначала по вертикали указывают параметры.
2. В последний столбец вносят ожидаемый результат. Можно разделить на несколько столбцов.
3. В ячейках указывают значения параметров.
4. Каждая строка — это уникальная комбинация условий.

**Как применять таблицу принятия решений? Зачем она нужна? [3 спринт]**

**Таблицу принятия решений используют**, когда в требованиях есть разные комбинации условий. Их удобно представить графически, в виде таблицы, чтобы точно покрыть тестами все требования и ничего не упустить.

## **Алгоритм техники**

1. Проанализировать требования и выделить объекты тестирования.
2. Определить условия.
3. Сделать заготовку таблицы.
4. Заполнить таблицу.

**Условие** — это параметр с определённым значением.

## **Как заполнять таблицу**

1. Сначала по вертикали указывают параметры.
2. В последний столбец вносят ожидаемый результат. Можно разделить на несколько столбцов. Например, «Пишем продавцу» и «Не пишем продавцу» с вариантами «Да» или «Нет».
3. В ячейках указывают значения параметров.
4. Каждая строка — это уникальная комбинация условий.
5. Таблица содержит все возможные комбинации условий.

**Как применять диаграмму состояний и переходов? В каких случаях эта техника подходит? [3 спринт]**

**Диаграмма состояний и переходов** — это схема, которая описывает, в каких состояниях может находиться приложение или его часть и какие переходы есть между разными состояниями.

Диаграмма состояний и переходов хорошо подходит для описания процессов.

Действие или событие, которое переводит одно состояние в другое, называется **переходом**.

* Переход, вызванный пользователем, — это действие. Например, пользователь нажимает на кнопку, чтобы выбрать чек-бокс.
* Переход, вызванный самой системой, — это событие. Например, для входа в приложение может понадобиться ввести код из смс-сообщения. Такой код работает около минуты, время отсчитывается по таймеру. Когда минута пройдёт, поле для ввода кода исчезнет — это и есть событие.

## **Алгоритм составления диаграммы**

1. Определить объект тестирования: чек-бокс.
2. Определить состояния объекта: выбран или не выбран.
3. Определить начальное состояние: не выбран.
4. Нарисовать начальное состояние: обычно это прямоугольник.
5. Обозначить начало схемы, или псевдосостояние. Обозначают чёрной точкой со стрелкой.
6. Нарисовать остальные состояния — обычно тоже в виде прямоугольников.
7. Определить переходы — действия или события, которые переводят объект из одного состояния в другое.
8. Нарисовать переходы. Переходы обозначают стрелками. Они показывают, из какого состояния и в какое переходит объект.
9. Изобразить метку перехода рядом со стрелкой. Метка описывает действие или событие, которое ведет к переходу.
10. Можно отметить конечное псевдосостояние. Это круг с точкой.

**Как применять таблицу состояний и переходов? В каких случаях эта техника подходит? [3 спринт]**

Таблица может быть более наглядной и систематизированной, чем диаграмма. С ней легче формировать тест-кейсы. А ещё в таблице можно указать негативные проверки.

Её заполняют по следующему алгоритму:

**Шаг 1.** В первый столбец, начиная со второй строки, выписывают все состояния.

**Шаг 2.** В первую строку, начиная со второго столбца, выписывают все действия и события.

**Шаг 3.** В ячейках указывают состояния, в которых объект окажется после перехода.

**Когда стоит использовать следующие техники: технику попарного тестирования, таблицу принятия решений, диаграмму состояний и переходов, таблицу состояний и переходов? [3 спринт]**

**Таблица принятия решений** удобна, когда у объекта тестирования не слишком много параметров и их значений.

С **попарным тестированием** наоборот: его применяют, когда параметров и их значений много.

**Диаграмма состояний и переходов** хорошо подходит для описания процессов и сценариев работы функциональности.

**Таблица состояний и переходов** может быть более наглядной и систематизированной, чем диаграмма. С ней легче формировать тест-кейсы. А ещё в таблице можно указать негативные проверки.

**Как приоритизировать тесты? Какие бывают приоритеты? [Задачник и материалы для собеседований]**

Приложение дорабатывают и развивают, увеличивается число компонентов — а значит, и тест-кейсов. Последовательная проверка всех тестовых значений увеличивает сроки и стоимость тестирования, поэтому начинать нужно с самой важной функциональности.

Приоритет как раз отражает важность проверки.

Выделяют четыре приоритета проверок:

* критичный,
* высокий,
* средний,
* низкий.

**Критичный**

Тест с критичным приоритетом содержит сценарий для проверки основной логики приложения. Это базовая функциональность, с которой работают пользователи. С ней же связаны остальные компоненты.

Если на этом этапе происходит сбой, остальные проверки проводить бессмысленно.

**Высокий**

Сценарии тестов с высоким приоритетом проверяют дополнительные возможности функциональности, их важные особенности и соответствие требованиям.

Ошибки, найденные в ходе таких проверок, не блокируют работу основной логики приложения, но мешают пользоваться некоторыми фичами.

**Средний**

Проверки со средним приоритетом позволяют найти недочеты в работающем приложении. Такие тесты помогают усовершенствовать приложение — сделать его удобнее для пользователя.

Ошибки, обнаруженные при проверках со средним приоритетом, не нарушают логику приложения. Скорее всего, их исправят в следующих релизах.

**Низкий**

Тесты с низким приоритетом проверяют мелкие недочёты. Обычный пользователь может никогда не столкнуться с таким багом. Но если его удалось найти, нужно исправить.

**Что такое позитивные и негативные проверки? [3 спринт]**

**Позитивные тесты-кейсы** проверяют, что приложение работает без ошибок в двух случаях:

* если использовать его по назначению и не пытаться сломать,
* если вводить корректные данные согласно требованиям.

**Негативные тест-кейсы** проверяют, как поведёт себя приложение, если:

* использовать приложение не так, как задумывали разработчики;
* вводить данные не из требований.

**Какими инструментами пользуется тестировщик при работе с тестами? [3 спринт]**

* Система управления тестами (TMS);
* Инструменты для работы с таблицами;
* Инструменты для разработки диаграмм.

**Тестирование приложений**

**Объясни принцип работы клиент-серверной архитектуры. [4 спринт]**

Веб-приложение устроено по определённым принципам. Они определяют, из каких элементов состоит приложение и как они связаны.

Такие принципы называют архитектурой. Это способ организации работы приложения.

Самая распространённая архитектура веб-приложений — клиент-серверная.

Клиент, сервер и интернет — элементы клиент-серверной архитектуры. Клиент и сервер работают по отдельности: клиент отвечает за взаимодействие с пользователем, сервер — за логические операции, вычисления и хранение данных, а интернет, или сеть — за связь клиента и сервера.

**Клиент** — это система, которая связывается с сервером и запрашивает нужную пользователю информацию.

Клиент преобразует твои действия в запросы и отправляет на сервер.

**Сервер** — система, которая обрабатывает запросы клиента и формирует ответ. Например, присылает информацию о видео: лайки, комментарии, количество просмотров.

Клиент и сервер «общаются» по сети.

**Интернет**, или **Сеть** — система связанных между собой устройств, которая помогает клиенту и серверу обмениваться данными.

Если ты знаешь, как устроена клиент-серверная архитектура, то сможешь понять, где именно в приложении произошла ошибка: на стороне клиента или сервера. Так разработчики быстрее исправят баг.

**Что такое REST API, SOAP? В чем между ними разница? [7 спринт]**

**REST** — это способ использования API.

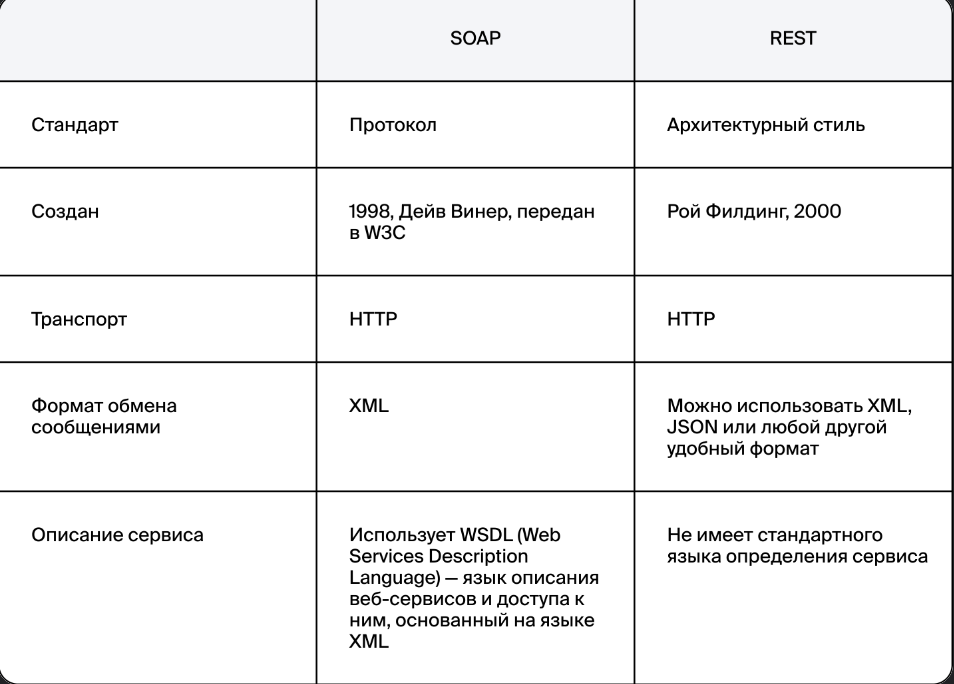
**REST** расшифровывается как Representational State Transfer — «передача состояния представления». Клиент отправляет запросы, а сервер присылает ответы на эти запросы согласно правилам. Их определяет REST-архитектура.

**SOAP** (Simple Object Access Protocol) — протокол обмена данными. Это означает, что клиент и сервер «общаются» по определённым правилам. Так они точно знают, что друг от друга ожидать: в каком формате придут данные и как прислать ответ. Получается меньше сбоев и ошибок.

**Разница** между **SOAP** и **REST** — это как разница между обычном магазином и магазином самообслуживания. В SOAP ты общаешься с продавцом, узнаёшь стоимость товара, просишь его взвесить. А в REST просто берёшь с полки нужное и идёшь оплачивать.

О разнице между SOAP и REST часто спрашивают на собеседовании, поэтому сохрани табличку ниже в шпаргалках.

Постарайся запомнить, что REST — это архитектура, а SOAP— протокол, на котором тоже можно построить архитектуру. REST популярнее, чем SOAP, потому что проще в использовании, но SOAP всё ещё встречается. Для тестирования SOAP обычно используют Postman или SoapUI.



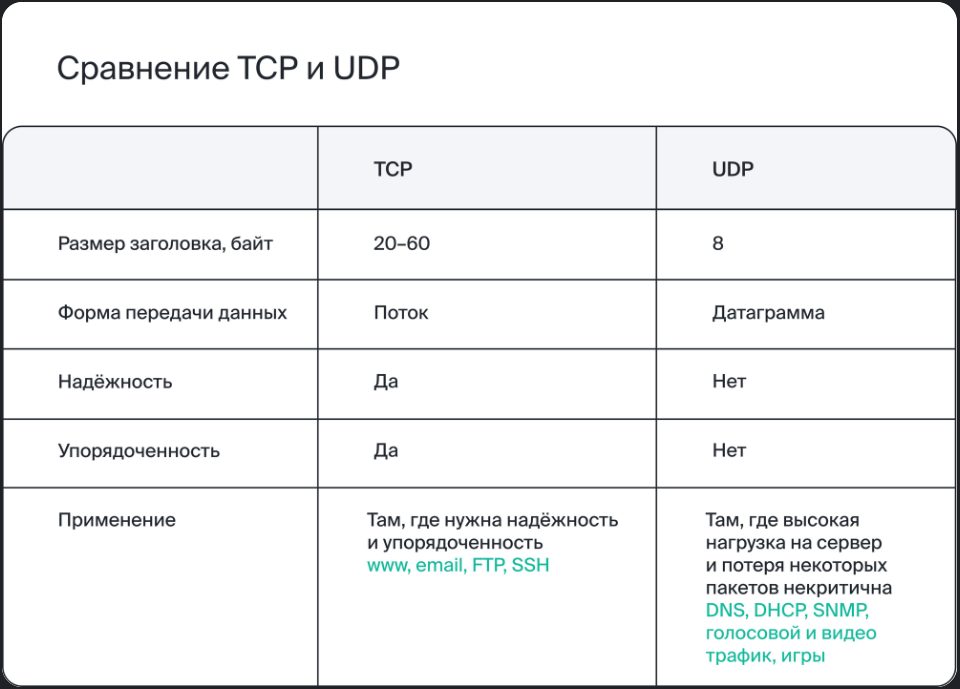
**Что такое TCP/IP и UDP? В чем между ними разница? [Задачник и материалы для собеседований]**

**TCP/IP** — это связь протоколов TCP (Transmission Control Protocol) и IP (Internet Protocol), благодаря которым работает интернет.

**TCP** разбивает информацию на маленькие пакеты данных, а IP контролирует отправление и доставку этой информации от отправителя к получателю.

**UDP** (протокол пользовательских датаграмм) — ненадёжный, он занимается передачей автономных датаграмм.

Датаграмма — это блок информации, который передаётся протоколом через сеть связи без предварительного установления соединения.

UDP формирует и отправляет пакеты, но не гарантирует, что все датаграммы дойдут до получателя. Сами датаграммы содержат всю необходимую информацию, чтобы дойти до получателя, но они всё равно могут быть потеряны или доставлены не в том порядке.

**Что такое HTTP? URL? [4 спринт]**

**Протокол передачи данных** — набор правил, по которым устройства обмениваются информацией.

Один из таких протоколов — **HTTP** (HyperText Transfer Protocol — протокол передачи гипертекста). Сейчас по HTTP передаётся не только гипертекст, но и другие данные: картинки, аудио, видео.

**URL** (Uniform Resource Locator — унифицированный указатель ресурса) — это адрес веб-ресурса. URL показывает, где находится веб-приложение, веб-страница или фрагмент веб-страницы и как к ним обратиться.

**Что такое запрос и ответ? Из каких элементов они состоят? [4 спринт]**

**Запрос** — это специальное сообщение. В нём фронтенд «просит» у бэкенда совершить определённые действия с данными: например, отправить, сохранить или изменить их.

Запрос структурирован по правилам передачи данных HTTP. Он состоит из трёх блоков:

* стартовая строка,
* заголовки,
* тело сообщения.

**Ответ** тоже специальное сообщение, в котором бэкенд отдаёт результат выполнения запроса.

**HTTP-ответ** включает:

* строку состояния,
* заголовки,
* тело сообщения.

**Какие наиболее распространенные методы HTTP-запроса ты знаешь? [4 спринт]**

* **GET** — запросить у бэкенда данные по определённому адресу.
* **POST** — отправить данные на бэкенд.
* **PUT** — изменить данные на бэкенде.
* **DELETE** — удалить данные на бэкенде.

**Чем метод POST отличается от GET? [Нет ответа в программе]**

* Метод GET используется для чтения информации, а метод POST — для отправки данных.
* У GET-запроса информация передаётся в заголовке, у POST — в теле.
* GET-запрос не подходит для передачи приватных данных, а POST-запрос — да. Он поддерживает кодирование данных, и информация там передаётся в теле.

**Чем метод PUT отличается от PATCH? [Нет ответа в программе]**

PUT обновляет весь ресурс целиком, то есть создаёт новую версию ресурса на сервере.

PATCH обновляет ресурс в той части, которая указана в теле запроса. То есть обновляет ту версию, которая уже находится на сервере.

**Какие коды состояния ты знаешь? [4 спринт]**

**1xx** — информационные сообщения,

**2xx** — сообщения об успехе,

**3xx** — перенаправление,

**4xx** — клиентские ошибки,

**5xx** — ошибки сервера.

**Что означают следующие коды состояния: 200, 300, 400, 500? [4 спринт]**

**202** - Запрос принят в обработку, но обработка еще не завершена.

Этот код состояния означает, что сервер принял запрос на обработку, но сам процесс обработки ещё не завершён. Запрос будет обработан позже, и результат будет предоставлен в будущем.

**302** - Перенаправление временно.

Этот код указывает, что ресурс, запрошенный клиентом, временно находится по другому URL-адресу. Клиент должен использовать новый URL для временного доступа, но не должен запоминать его для будущих запросов.

**403** - Доступ запрещен.

Этот код указывает, что сервер понял запрос, но отказывается его выполнять из-за отсутствия у клиента соответствующих прав доступа. Это может происходить, например, при попытке доступа к ресурсам, требующим авторизации.

**404** - Ресурс не найден.

Этот код состояния означает, что сервер не может найти запрашиваемый ресурс. Это обычно происходит, когда URL-адрес неверен или страница была удалена.

**502** - Ошибка шлюза.

Этот код состояния обозначает, что сервер, выполняющий роль шлюза или прокси, получил недействительный ответ от вышестоящего сервера. Это может быть связано с проблемами в сети или ошибками на стороне других серверов.

**Что такое фронтенд и бэкенд? Как они связаны с API? [4 спринт]**

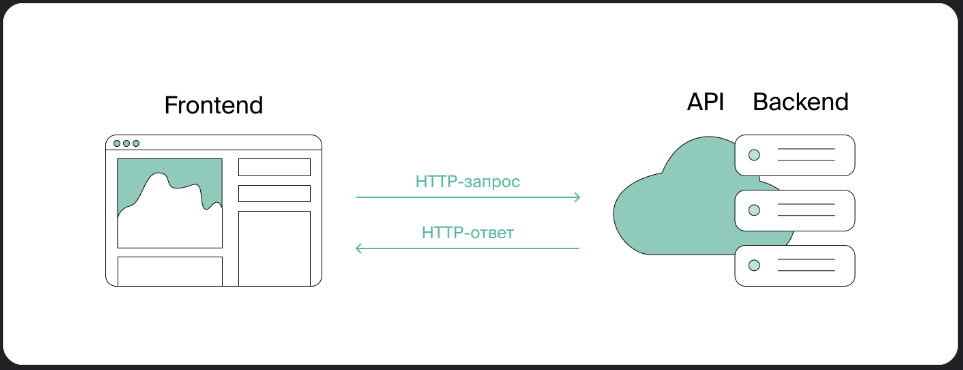
Фронтенд и бэкенд — две части одного веб-приложения.

**Фронтенд** (frontend) — видимая часть приложения. С ней пользователь взаимодействует: например, нажимает на кнопки или вводит текст. В модели «клиент-сервер» фронтенд — это код, который обрабатывается на стороне клиента.

**Бэкенд** (backend) — скрытая часть приложения. Она отвечает за вычисления, логику, хранение данных. В модели «клиент-сервер» бэкенд — это код, который работает на удалённом сервере.

Вспомни клиент-серверную архитектуру: фронтенд общается с бэкендом, отправляя запросы и получая ответы. Чтобы сделать запрос к бэкенду, фронтенд обращается к API бэкенда.

Здесь API бэкенда — это фиксированный набор URL-адресов. По ним можно передавать данные в определённом формате.



**Что такое JSON и XML? Какие у них особенности? Какая между ними разница? [7 спринт]**

## **JSON**

**JSON** (англ. JavaScript Object Notation) — текстовый формат обмена данными, основан на распространённом языке программирования JavaScript (JS). При этом формат независим от JS и может использоваться в любом языке.

В качестве значений в JSON могут встретиться:

* JSON-объект;
* массив, или список;
* число — целое или вещественное;
* литералы true (логическое значение «истина»), false (логическое значение «ложь») и null;
* строка.

JSON используется в REST API. При этом одну и ту же структуру JSON можно преобразовать в XML.

## **XML**

Документ **XML** состоит из элементов — как из кирпичиков. Элемент состоит из **тегов** и **содержимого**.

Тег в начале называется **открывающим**, а в конце — **закрывающим**. Оба тега обязательные.

Элементы в XML можно группировать. Элемент-контейнер для остальных называется **родителем**, а элементы внутри — **потомками**. Самый «главный» родитель называется **корневым элементом**.

Также у XML присутствуют XML-декларация и Неймспейсы

## **Разница**

JSON — «лёгкий» формат данных. XML — ещё один способ размечать данные, которыми обмениваются приложения. В XML объём больше — потому что разметка тегами, а указывать данные массивами нельзя.

**Как посмотреть запросы, которые отправляет фронтенд, и ответы, которые присылает бэкенд? [4 спринт]**

Чтобы просматривать запросы и ответы, обычно пользуются DevTools (Tools for Web Developers — инструменты разработчика). Инструменты разработчика доступны в современных браузерах.

На вкладке Network доступна следующая информация:

* названия ресурсов, которые фронтенд и бэкенд передают по сети (колонка Name);
* длительность загрузки (колонка Time);
* размер файлов (колонка Size);
* сводная информация о запросах к серверу: например, объём трафика.

**Где содержится информация о методах, реализованных в API приложения? [7 спринт]**

Эта информация содержится в документации API.

Документация помогает разобраться, как работает API определённого компонента или приложения. Внутри описаны:

* Действия, которые можно делать с системой; данные, которые можно запросить.
* Ограничения передаваемых данных. Например, у некоторых параметров может быть ограничение на тип данных (число или строка), на длину (от 2 до 50 символов).
* Структура запросов, ответов и адресов, по которым можно отправлять запросы.

**Что такое Swagger? [7 спринт]**

Чтобы упростить работу с документацией, пользуются инструментами, которые генерируют её автоматически. Например, Swagger.

Разработчик готовит специальный файл с описанием запросов для API; инструмент автоматически переводит её в структуру.

**Какие инструменты нужны, чтобы протестировать API? [7 спринт]**

* Инструменты для просмотра, создания, хранения, отправки запросов (Postman, cURL).
* JSON-валидаторы. Проверяют синтаксис и подсвечивают ошибки в JSON-запросе.
* Инструменты разработчика (Devtools).

**Перечисли основные функции Postman. [Нет ответа в программе]**

Он позволяет создавать коллекции запросов к любому API, применять к ним разные окружения, настраивать мок-серверы, писать автотесты на JavaScript, анализировать и визуализировать результаты запросов (просматривать ответы сразу в окошке приложения).

**Что такое валидация на стороне бэкенда? А на стороне фронтенда? В чём между ними разница? [Нет ответа в программе]**

Валидация на фронтенде — проверка, что пользователь ввёл данные и они соответствую требованиям.

Валидация на бэкенде — механизм, который проверяет, что запрос в API составили правильно: указали данные в нужном формате и не ошиблись в структуре.

Правила валидации разработчики прописывают в коде. Это делают как на стороне клиента, так и на стороне сервера. Лучше «зашивать» валидацию на бэкенде: на клиенте её может обойти злоумышленник, а к серверу у него обычно нет доступа.

Валидация может предупреждать о том, что:

* **структура запроса неправильная**: например, нет всех нужных параметров или тегов;
* **данные записаны** **некорректно**: слишком много символов или не тот формат.

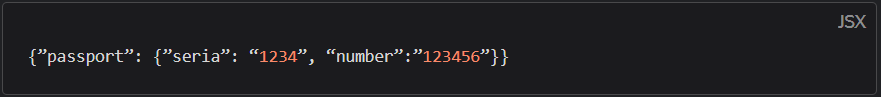
**Какие особенности стоит учитывать при тестировании API? [7 спринт]**

## Корректные значения параметров

Метод может получать несколько входных параметров. Каждый из них предстоит проверить отдельно как некорректный. Для этого остальные параметры нужно зафиксировать корректными значениями.

## Проверка параметров во вложенных объектах и списках

На вход принимается структура:



Здесь заданы три параметра:

**passport** с типом словарь;

**seria** с типом строка;

**number** с типом строка;

**seria** и **number** — это вложенные параметры, их тоже нужно проверять.

Что будет, если передать параметру **seria** значение с типом целое число? А если передать несуществующее значение? Как поведёт себя система, если в параметрах **seria** и **number** будет превышено количество допустимых символов? Каждый из этих вопросов должен быть покрыт проверкой.

## Проверка URL-параметров

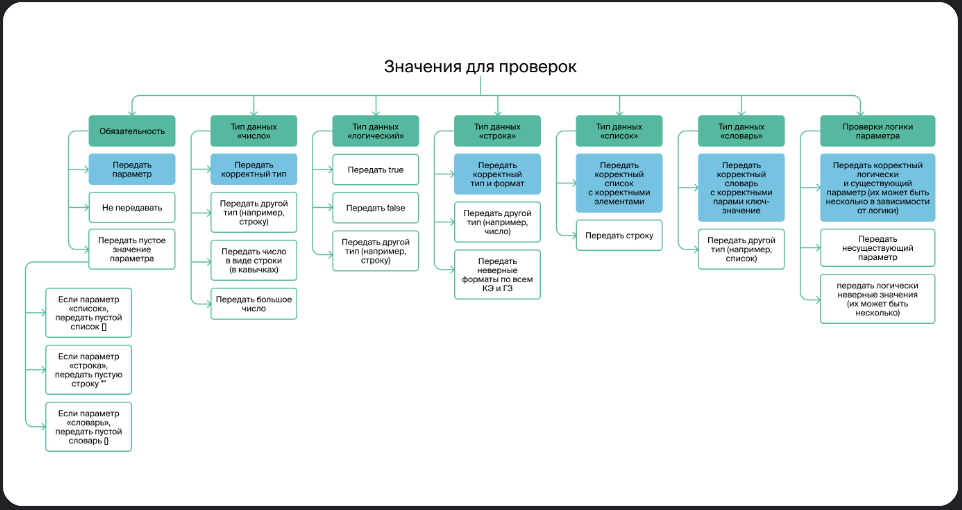
Для параметров, которые передаются в стартовой строке метода, нужно проверить:

* существующие значения,
* несуществующие значения.

**Что стоит проверить, когда тестируешь API? [7 спринт]**

Вот какую последовательность действий можно выделить при тестировании API:

* 1. Определи все параметры, которые передаются в запросе. Не забудь проверить вложенные запросы.
* 2. Составь запрос со всеми корректными параметрами. Так ты зафиксируешь корректные значения и сможешь проводить отдельные негативные проверки для каждого параметра.
* 3. Если есть URL-параметр, проверь случаи, когда:
* передаётся существующее значение параметра,
* передаётся несуществующее значение параметра.
* 4. Подбери значения для проверок каждого body-параметра.

****

Обрати внимание: все пункты, выделенные синим цветом, выполняются одним запросом.

1. Составь тестовую документацию. Подойдёт чек-лист, в котором описаны все проверки. Опирайся на требования и документацию.
2. Убедись, что все КЭ и ГЗ выделены по логике параметров.
3. Убедись, что для каждой проверки приведено конкретное тестовое значение.
4. Проведи тестирование.

**Что такое адаптивная вёрстка? Почему нужно тестировать вёрстку на адаптивность? [4 спринт]**

Бывает, что элементы интерфейса адаптируются под разрешение экрана не так, как задумано.

**Адаптивность** — это способность приложения подстраиваться под экраны с разными разрешениями.

Чтобы «подогнать» вёрстку под разные разрешения, применяют один из подходов: **адаптивный** и **респонсивный** дизайн, или **адаптив** и **респонсив**. И то и другое часто просто называют «адаптивным» дизайном.

Тестировщику всегда нужно проверять, как вёрстка подстраивается под устройства с разными разрешениями. Это называется тестирование **адаптивности**.

При тестировании адаптивности важно убедиться, что веб-приложение в разных разрешениях соответствует макетам.

**Для чего тестировщики используют Devtools? [4 спринт]**

Тестировщики используют Devtools, чтобы:

* Просматривать запросы фронтенда и ответы бэкенда с помощью вкладки Network (Сеть).
* Тестировать вёрстку с помощью Toggle Device Toolbar.
* Просматривать логи для локализации ошибок с помощью вкладки Console (Консоль).
* Просматривать куки прямо в браузере с помощью вкладки Application (Приложение). Раздел «Storage» (Хранилище) → «Cookies» (Файлы куки).
* Просматривать данные, используемые только клиентской частью приложения, — с помощью вкладки Local Storage.
* Тестировать приложение в разных окружениях, не меняя при этом техническое устройство, — с помощью User-Agent на вкладке Elements.

**Объясни назначение вкладки Console в Devtools. [4 спринт]**

С помощью вкладки **Console** (Консоль) тестировщик просматривает логи для локализации ошибок.

Здесь есть классификация логов по степени важности.

* **Errors** — ошибки, которые влияют на взаимодействие пользователя с приложением. Например, из-за таких ошибок может не работать часть приложения.
* **Warnings** — предупреждения о потенциальных ошибках, при этом приложение всё равно смогло выполнить запрос.
* **Information** — информационные сообщения. Например, о работе базы данных.

**Объясни назначение Toggle device toolbar в Devtools. [4 спринт]**

Toggle Device Toolbar имитирует разрешение и ориентацию экрана мобильных устройств:

* как вёрстка веб-приложения адаптируется под разные разрешения экрана;
* как вёрстка ведёт себя, когда меняется ориентация экрана;
* некоторые жесты: например, свайп — скольжение пальцем в сторону.

**Объясни назначение Application в Devtools. [Нет ответа в программе]**

Это вкладка для инспектирования и очистки всех загруженных ресурсов. Например, можно очистить куки, кэш, изображения, шрифты и так далее.

**В чём главное различие браузеров? [4 спринт]**

В движке.

**Браузерный движок** — это программа внутри браузера, которая помогает отображать веб-страницы и её элементы.

Некоторые браузеры работают на одном движке: это значит, что логика отображения у них одинаковая. Например, Google Chrome, Яндекс Браузер, Opera и Microsoft Edge.

Для каждого движка можно выбрать самый популярный браузер и проверять только в нём.

Вот примеры движков:

* Движок Blink — Яндекс Браузер, Google Chrome, Opera 15+, Microsoft Edge 79+.
* Движок WebKit — Apple Safari, Google Chrome на iOS.
* Движок Gecko — Mozilla Firefox.

Некоторые баги возникают только в одном браузере и не воспроизводятся в других — даже на том же движке. Такие баги называют специфичными.

Если всегда проверять только в самом популярном браузере на одном движке, можно не заметить некоторые ошибки: например, баг может воспроизводиться в Яндекс Браузере, а в Google Chrome — нет.

**Что такое cookie (куки) и cache (кэш)? В чём их особенности? [4 спринт]**

**Cookie** (куки, печенье) — данные, которые хранятся на устройстве пользователя. Эти данные приходят с сервера и сохраняются на компьютере или смартфоне, а когда используешь приложение повторно — снова передаются на сервер.

**Кэш** (Cache Storage) — это данные веб-страниц. Они сохраняются на компьютере, когда ты открываешь изображения, аудио-, видео-, CSS-, HTML-, JS-файлы.

Cookie и Local Storage упрощают работу с приложением. Пользователю не нужно каждый раз вводить одни и те же данные, например: логин, пароль, номер телефона.

Браузерный кэш нужен для другого. Когда ты откроешь веб-страницу повторно, она загрузится быстрее: браузер возьмёт файлы из кэш-хранилища, а не с сервера.

**Как составить набор для регрессионного тестирования? [Задачник и материалы для собеседований]**

Регрессионное тестирование — это расширенная проверка. Сюда относят:

1️⃣ Проверку функциональности, которая уже была в предыдущих версиях и в текущей не дорабатывалась, но могла быть затронута другими изменениями. Например, даже если в форме регистрации ничего не меняли, из-за доработок в других частях приложения здесь могут возникнуть ошибки.

2️⃣ Расширенную проверку функциональности, которая изменилась в новой версии. Например, в форму регистрации добавили новое поле. Регресс должен проверить всю форму целиком: работают ли остальные поля, что происходит при вводе валидных и невалидных данных, как отображается форма на экранах с разным разрешением.

3️⃣ Сценарии, которые редко используются на практике. Так можно найти те части приложения, которые пока не покрыты проверкам.

**Как составить набор для смоук-тестирования? [Задачник и материалы для собеседований]**

Смоук-тесты обычно запускают в таких случаях:

* Вместо регресса. Например, в приложении поменялась функциональность, но на регресс нет времени. Тестировщикам нужно убедиться, что изменения ничего не сломали.
* Перед регрессом. Даже если на регресс есть время, в первую очередь всё равно нужно запустить смоук-набор — он включает самые важные проверки.
* После регресса. Иногда смоук-набор запускают повторно, но уже на стенде продакшена.

Это очень частые проверки, поэтому их стараются автоматизировать. Но вот определить, какие тест-кейсы включить в набор, — задача тестировщика. Чтобы это сделать:

* Изучи требования и макеты.
* Подумай, какие требования приоритетны.
* Собери и запусти набор.

**Что такое логи? Где их можно посмотреть? [6 спринт]**

Современные сервисы и приложения — это сложные системы. Всё, что происходит в программе, чаще всего фиксируют в журнале событий. Такие записи — это **логи**.

Логи помогают тестировщику и разработчику быстрее локализовать баг. В логах фиксируются разные данные: например, когда пользователь нажал кнопку в приложении и произошла ошибка.

Разработчик управляет уровнями логирования и может обозначить, какую информацию записывать в лог. Чаще всего встречаются такие уровни:

* **ERROR**. Информация о проблемах, которые повлияли на работу приложения. Например, не выполнился какой-нибудь запрос или не сделаны определённые расчёты, но приложение может выполнять другие функции.
* **CRITICAL**. Сообщение о критических ошибках в приложении, которые остановили работу: например, повреждение базы данных.
* **WARNING**. Предупреждение о проблемах, которые могут приводить к ошибкам в приложении.
* **INFO**. Обычное логирование действий, которые происходят в приложении и не требуют реакции.
* **DEBUG**. Подробное информирование о действиях в приложении. Этот уровень применяют разработчики, чтобы отладить приложение.

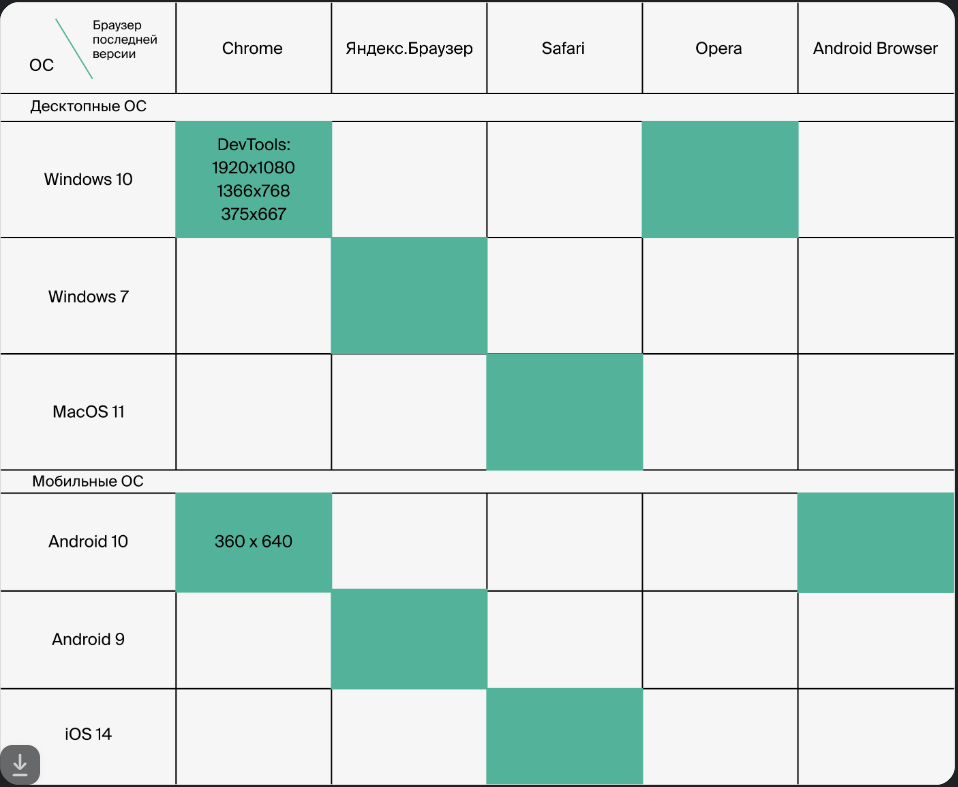
Приложения могут записывать логи в текстовые файлы, выводить в консоль или отправлять их по сети в базы данных.

**Как подобрать окружение для тестирования? [4 спринт]**

Составить матрицу поддерживаемых окружений — таблицу, строками которой являются операционные системы, а столбцами — браузеры.

Пригодится специальный алгоритм:

* Последовательно сопоставь каждый браузер с одной десктопной и одной мобильной ОС.
* Не забудь, что некоторые браузеры работают не со всеми ОС.
* У тебя получится таблица — опирайся на неё, чтобы хотя бы раз проверить каждую ОС и каждый браузер.
* Выдели зелёным цветом ячейки, в которых пересекаются ОС и браузер, — это поддерживаемые окружения.
* Впиши поддерживаемые разрешения в те зелёные ячейки, в которых можно их проверить. Разрешения можно тестировать на реальных устройствах или в DevTools.

****

**Что такое тестовый стенд и стенд продакшна? [1 спринт]**

**Тестовый стенд** — тестовая версия приложения в определённом окружении.

Тестировщик проверяет новую функциональность на своём тестовом стенде.

**Продакшен** (англ. production) — это полностью настроенная среда, в которой работают пользователи.

**Как часто следует проводить регрессионное тестирование? [Нет ответа в программе]**

Перед каждым релизом.

**Что делать, если разработчик не может воспроизвести баг? [Нет ответа в программе]**

1. Свериться по окружению, версии приложения и шагам теста.
2. Если всё идентично, то созвониться и воспроизвести баг на твоем компьютере с демонстрацией экрана.
3. Попросить коллег-тестировщиков воспроизвести баг у себя.
4. Если у коллег нет такой ошибки, то это не массовый случай. У такого бага можно понизить приоритет.

**Перечисли основные функции Charles/Fiddler. [4 спринт]**

Они позволяют:

* просматривать трафик,
* перехватывать запросы и ответы,
* вносить изменения в запросы и ответы: вручную, автоматически или из файла.

**Базы данных и SQL**

**Что такое база данных? [7 спринт]**

**База данных или БД** — элемент архитектуры приложения. Это хранилище

информации внутри приложения.

С БД взаимодействует бэкенд: он отправляет ей запросы и получает данные в ответ

**Что такое СУБД? [8 спринт]**

Это специальная **система управления базами данных, или СУБД**. СУБД способна на многое, например, с её помощью можно:

* создать базу или таблицу в базе;
* внести новые данные или удалить устаревшие;
* выгрузить нужную информацию, задав условие;
* обеспечить безопасный доступ к данным.

**Что такое реляционная база данных? В чём её особенности? [8 спринт]**

Если данные в базе представлены в виде связанных таблиц, такую базу данных называют **реляционной** (англ. relation — «связь»). Реляционные базы данных особенно популярны благодаря своей простоте и надёжности.

У таблиц в реляционных базах есть несколько особенностей. Читая документацию или общаясь с аналитиками, ты обнаружишь, что элементы таблицы часто называют так: столбцы — **полями**, строки — **записями**, объекты на пересечении полей и записей — **ячейками**. Слова «столбец» и «поле» могут быть взаимозаменяемыми, но не забудь об этой терминологии, когда встретишь непривычное слово.

Другая особенность — в таблице не должно быть дубликатов. Если все записи в таблице уникальны, ты будешь знать наверняка, что работаешь с нужной строкой. Для этого к каждой записи добавляется уникальный признак — **первичный ключ**. Такой ключ не позволит создать абсолютно одинаковые записи.

**Что выведет в пропусках FULL JOIN? [8 спринт]**

Оператор FULL OUTER JOIN объединяет все данные из левой и правой таблиц. Если не нашлось совпадения, на месте значения будет NULL.

**В чем разница между RIGHT JOIN и LEFT JOIN? [8 спринт]**

LEFT OUTER JOIN — тип объединения, который предполагает, что в результат слияния войдут все записи из левой таблицы. Записи из правой таблицы сохранятся только в том случае, если значения в нужном поле совпадают со значениями в левой таблице.

Тип RIGHT OUTER JOIN похож на предыдущий тип объединения, только в этом случае больший приоритет у правой таблицы. Оператор RIGHT OUTER JOIN предполагает, что записи из правой таблицы обязательно войдут в результат слияния. А записи из левой таблицы сохранятся только в том случае, если значения в нужном поле совпадают со значениями в правой таблице. На месте некоторых фамилий указано значение NULL — в том случае, если нужных данных в левой таблице нет.

**Какие существуют типы связей между таблицами? [8 спринт]**

## Связь «один к одному»

В английском языке такую связь называют ”one-to-one”. Такой вид связи предполагает, что одна запись первой таблицы связана только с одной записью другой таблицы.

## Связь «один ко многим»

Этот вид связи, наоборот, самый популярный. Иногда его обозначают ”one-to-many”. Такой вид связи предполагает, что одной записи первой таблицы соответствует несколько записей другой таблицы.

## Связь «многие ко многим»

Другое название связи — ”many-to-many”. Этот вид связи предполагает, что одна запись одной таблицы соответствует нескольким записям другой таблицы и наоборот. Кажется, такой вид связи напоминает «один ко многим», но это не так.

**Что такое первичный ключ? Что такое внешний ключ? [8 спринт]**

**Первичный ключ** делает запись уникальной и позволит избежать полных дубликатов в таблице.

**Внешний ключ** — это поле, которое отсылает к первичному ключу другой таблицы. Другими словами, внешний ключ указывает на запись другой таблицы.

**Типы данных в PostgreSQL**

