Тестирование

Тестирование – поиск несоответствий между фактическим и ожидаемым результатом.

ЖЦ ПО (жизненный цикл программного обеспечения)

1. Инициация (идея)
2. Разработка и сбор требований
3. Дизайн
4. Разработка
5. Тестирование
6. Ввод в эксплуатацию
7. Вывод из эксплуатации

**Цель тестирования** — проверка соответствия ПО предъявляемым требованиям, обеспечение уверенности в качестве ПО, поиск очевидных ошибок в программном обеспечении, которые должны быть выявлены до того, как их обнаружат пользователи программы.  
  
**Для чего проводится тестирование ПО?**

* Для проверки соответствия требованиям.
* Для обнаружение проблем на более ранних этапах разработки и предотвращение повышения стоимости продукта.
* Обнаружение вариантов использования, которые не были предусмотрены при разработке. А также взгляд на продукт со стороны пользователя.
* Повышение лояльности к компании и продукту, т.к. любой обнаруженный дефект негативно влияет на доверие пользователей.

**Уровни тестирования**

1. Компонентное/модульное (тестирование отдельных модулей)
2. Интеграционное тестирование (проверка взаимодействий всех этих модулей)
3. Системное – полная проверка приложения

Виды тестирования

1. **Регрессионное тестирование (regression testing)** — тестирование уже проверенной ранее функциональности после внесения изменений в код приложения, для уверенности в том, что эти изменения не внесли ошибки в областях, которые не подверглись изменениям.
2. **Тестирование интерфейса (GUI/UI testing)** — проверка требований к пользовательскому интерфейсу.
3. **Стрессовое тестирование (stress testing)** — тип тестирования направленный для проверки, как система обращается с нарастающей нагрузкой (количеством одновременных пользователей).
4. **Позитивное тестирование** — тестирование, при котором используются только корректные данные.
5. **Негативное тестирование** — тестирование приложения, при котором используются некорректные данные и выполняются некорректные операции.
6. **Интеграционное тестирование** — тестирование, направленное на проверку корректности взаимодействия нескольких модулей, объединенных в единое целое.

**Модели разработки ПО**

Водопадная

Изображение выглядит как текст, знак, снимок экрана

Автоматически созданное описание

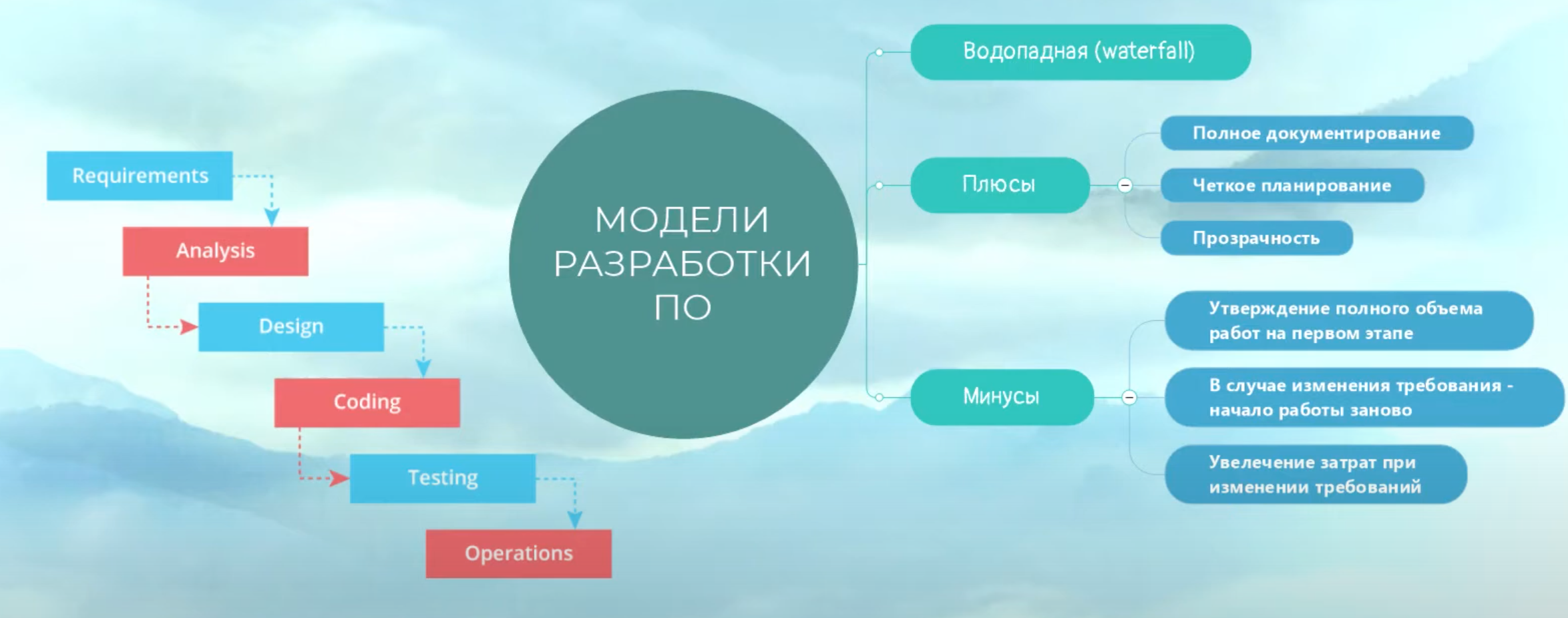
Для того, чтобы перейти на новый этап, мы полностью должны завершить текущий.

Плюсы:

* Полное документирование каждого этапа
* Всегда можем четко спланировать сроки и затраты
* Для заказчика наши процессы будут прозрачны (когда запуск, сколько времени потрачено)

Минусы

* Увеличение средств и времени, в случае изменении требований

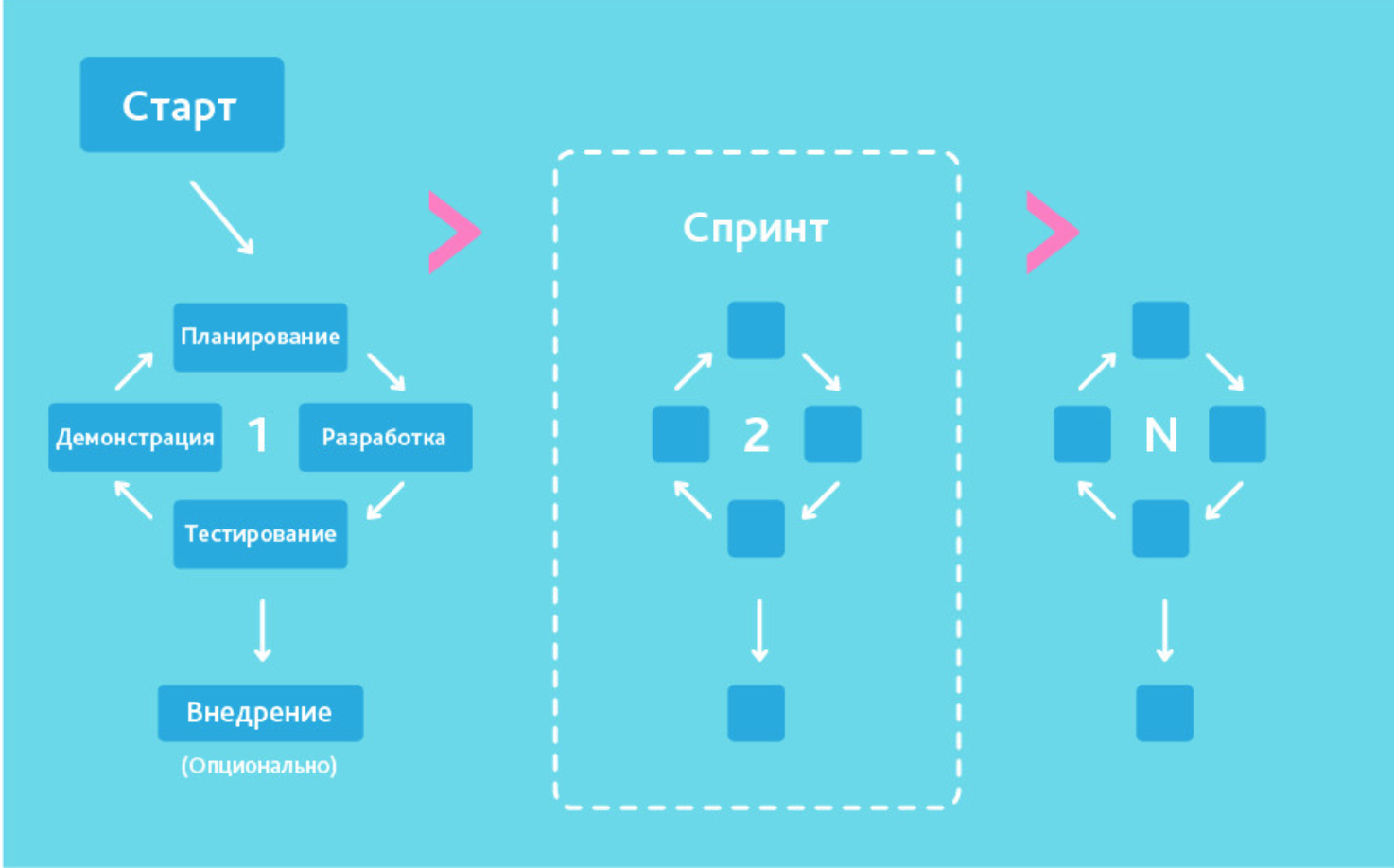


V-модель

На каждом этапе происходит контроль текузего процесса, для того чтобы убедиться в возможности перехода к следующему уровню.



## **«Agile Model» (гибкая методология разработки)**



В «гибкой» методологии разработки после каждой итерации заказчик может наблюдать результат и понимать, удовлетворяет он его или нет. Это одно из преимуществ гибкой модели. К ее недостаткам относят то, что из-за отсутствия конкретных формулировок результатов сложно оценить трудозатраты и стоимость, требуемые на разработку. Экстремальное программирование (XP) является одним из наиболее известных применений гибкой модели на практике.  
  
В основе такого типа — непродолжительные ежедневные встречи — «Scrum» и регулярно повторяющиеся собрания (раз в неделю, раз в две недели или раз в месяц), которые называются «Sprint». На ежедневных совещаниях участники команды обсуждают:

* отчёт о проделанной работе с момента последнего Scrum’a;
* список задач, которые сотрудник должен выполнить до следующего собрания;
* затруднения, возникшие в ходе работы.

Методология подходит для больших или нацеленных на длительный жизненный цикл проектов, постоянно адаптируемых к условиям рынка. Соответственно, в процессе реализации требования изменяются. Стоит вспомнить класс творческих людей, которым свойственно генерировать, выдавать и опробовать новые идеи еженедельно или даже ежедневно. Гибкая разработка лучше всего подходит для этого психотипа руководителей. Внутренние стартапы компании мы разрабатываем по Agile. Примером клиентских проектов является [Электронная Система Медицинских Осмотров](http://www.kvzrm.ru/), созданная для проведения массовых медосмотров в считанные минуты. Во втором абзаце [этого отзыва](http://www.edsd.ru/public/images/otzyv-ndvision.jpg), наши американские партнеры описали очень важную вещь, принципиальную для успеха на Agile.  
  
**Когда использовать Agile?**

* Когда потребности пользователей постоянно меняются в динамическом бизнесе.
* Изменения на Agile реализуются за меньшую цену из-за частых инкрементов.
* В отличие от модели водопада, в гибкой модели для старта проекта достаточно лишь небольшого планирования.

**Scrum и Kanban** — представители методологий Agile-семейства. Обе считаются гибкими и итеративными. Перед тем, как разобраться в разнице между ними, вспомним кратко о том, что их объединяет.

## **В чем разница между Scrum и Kanban?**

Основу Scrum составляют короткие итерации или спринты, как правило, 2-3-х недельные. Перед началом спринта команда сама формирует список фич на итерацию, далее запускается спринт.   
  
После окончания спринта выполненные фичи заливаются на продакшн, а невыполненные — переносятся в другой спринт. Как правило, фичи, которые делаются во время спринта, не меняются: что было на старте спринта — должно быть сделано любой ценой к окончанию спринта.  
  
На Kanban мы посмотрим там, где он и возник. Представьте себе конвейер, на котором делают детали для машин Toyota. Есть станок, он делает зеркала для машин. Он умеет делать левые зеркала, правые зеркала, задние и зеркала для солнцезащитного козырька. Принцип прост: нажми на кнопку, поменяй режим — получи новую продукцию.   
  
Вот вы заказываете в Москве на Кутузовском новую Toyota Camry на «максималке», и для вас уже делают зеркала в козырьке (вы выбрали «максималку» как раз из-за зеркал в козырьке). Важный момент тут — мы можем менять приоритеты в любой момент. Мы очень быстро можем переключать станок в другой режим.   
  
**Основная разница между Scrum и Канбан — в длине итераций. В Scrum итерации — 2 недели, в Kanban задачи программисту можно «подсовывать» хоть каждый день.**  
  
Kanban дает больше гибкости, если под гибкостью понимать частоту смены приоритетов. Вчера вы залили на прод новую фичу, а сегодня получили данные с передовой и узнали, что вот эта штука не работает так, как было задумано — люди не нажимают кнопку «купить». Вы «даете по шапке» UX, он дает вам новые требования. Вы поднимаете наверх очереди эту задачу, программист берет эту задачу «сверху», выполняет ее и, к вечеру fix уже на проде, конверсия в платежи выросли на 12%. Это победа.  
  
В Scrum задачи принято оценивать в Story points или в часах. Без оценки не получится сформировать спринт: ведь нам нужно знать, успеем ли мы сделать задачи за 2 недели. Через 2 недели мы получаем ценную статистику — сколько часов или Story points команда смогла сделать за спринт. Velocity — это производительность команды за один спринт. Этот параметр позволяет Scrum менеджеру предсказать, где команда будет через 2 недели.   
  
В Kanban не принято делать оценку. Это опционально, команда решает сама. Здесь нет понятия «скорость работы команды», считается только среднее время на задачу. Время это считается с помощью специального отчета — Cycle Time.   
  
Cycle Time для задачи = время выполнения задачи минус время начала работы над задачей. Например, у вас есть колонки: to do, reopened, developing, testing, stage testing, deployed. Cycle time для задачи будет равен deployed-developing, то есть сколько времени прошло от момента, когда задачу начали делать до момента пока она попала в deployed.  
  
Итак, в Scrum наша цель — закончить спринт, в Kanban — задачу.   
  
**Scrum — это автобус, который останавливается лишь на определенных остановках, где люди выходят группами. А Kanban — это маршрутка: захотел пассажир выйти, попросил водителя и вышел там, где ему нужно**

Тестовая документация

1. Чек-лист – список проверок, в котором показано что мы будем тестировать и как следствие результат и статус проверок. Список содержит в себе.

Изображение выглядит как стол

Автоматически созданное описание

Build – версия нашей сборки

Enviroment – окружение/операционная система, на котором проводилось тестирование

Test date – дата тестирования

Tester – фамилия тестировщика

Test type – Тип тестов

Checking – Название проверок

Result - Результат

Это тут все на английском, ты будешь на русском все писать.

Тест-кейс

Пошаговый сценарий/в отличии от чек-листа, где мы говорили «что будем тестировать», здесь мы описываем «как будем тестировать». Так же имеет поля – Заголовок, приоритет, результат. Итог – тест-кейс в обязательном порядке имеет три элемента: шаги проверок, фактический результат и ожидаемый результат.



Набор тест-кейсов собирается в тест-сьют/тест набор.

**Тест план (Test Plan)** — это документ, который описывает весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков.  
  
**Тест план должен отвечать на следующие вопросы:**

* Что необходимо протестировать?
* Как будет проводиться тестирование?
* Когда будет проводиться тестирование?
* Критерии начала тестирования.
* Критерии окончания тестирования.

Техники тест-дизайна

**Тест-дизайн** — это этап тестирования ПО, на котором проектируются и создаются тестовые случаи (тест-кейсы).

Тест-дизайн – разработка тест-кейсов. Есть две основные цели:

1. Придумать такие кейсы, которые могли бы обнаружить наиболее серьезные ошибки для нашего продукты
2. Минимизация таких тестов.

1 методика – Эквивалентное разбиение

Изображение выглядит как текст

Автоматически созданное описание

Пример был следующим – на работу нанимаются люди в определенном возрасте. Нужно для этого создать кейсы. Выше описаны правила.

2 методика – Анализ граничных значений

Изображение выглядит как текст

Автоматически созданное описание

Баг-репорт/отчет о дефекте

**Дефект (bug)** — отклонение фактического результата от ожидаемого.  
  
**Отчёт о дефекте (bug report)** — документ, который содержит отчет о любом недостатке в компоненте или системе, который потенциально может привести компонент или систему к невозможности выполнить требуемую функцию.

Атрибуты:

1. Описание дефекта (шаги воспроизведения дефекта)
2. Название проекта
3. Компонент (система, где обнаружен дефект)
4. Версия приложения
5. Серьезность – критичность дефекта (насколько критично влияет на приложение)
6. Приоритет (очередь исправления дефекта)
7. Статус дефекта в жизненном цикле
8. Автор
9. Назначение

Подробнее о серьезности и приоритете

На сегодняшний день, приоритет принято разделять на три уровня, а серьезность – на пять:

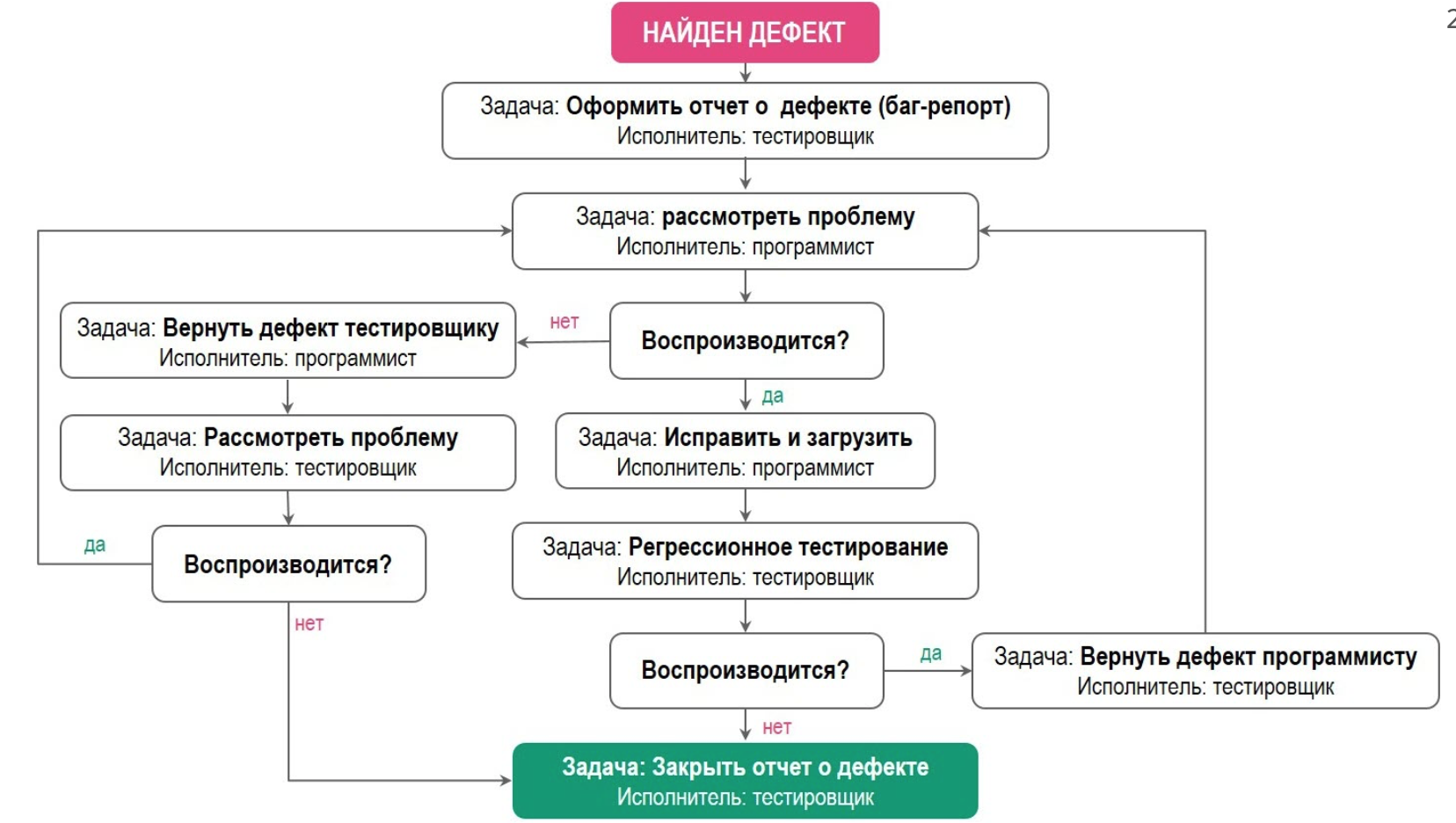
**Приоритет (Priority)** – это атрибут, указывающий на очередность выполнения задачи или устранения дефекта. Проставляется руководителем или менеджером проекта.

* **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 все дефекты с более высоким приоритетом уже исправлены.

**Серьезность (Severity**) – это атрибут, характеризующий влияние дефекта на работоспособность приложения. Проставляется тестировщиком или техническим специалистом, который может оценить степень влияния дефекта на работу системы.

* **S1 – Блокирующий (Blocker)** – дефект полностью блокирует выполнение функционала, нет никакого способа его обойти. Если провести аналогию с закрытым помещением и дверью – то дверь закрыта, у вас нет никакой возможности её открыть и покинуть помещение. Окон нет, ключ к двери не подходит.
* **S2 – Критический (Critical)** – дефект блокирует часть функциональности, но есть альтернативный путь для его обхода. По аналогии с помещением и дверью: вы можете покинуть помещение через окно, хотя дверь по-прежнему закрыта и ключ к ней не подходит.
* **S3 – Значительный (Major)** – дефект, указывающий на некорректную работу части функциональности. Зачастую связан не с тем, что функция не работает, а с тем, что она работает неправильно. В любом случае, существует более одной точки входа для инициации нужной функциональности. Так, вы можете покинуть помещение без использования ключа (дыра в безопасности), через вентиляцию (другая точка входа) или дверь открывается не в ту сторону (как следствие, упирается в угол и открывается только частично – некорректная реализация). Наиболее часто встречаются дефекты, которые можно отнести именно к этому уровню серьезности.
* **S4 – Незначительный (Minor)** – дефект, не относящийся к функциональности системы. Обычно серьезность Minor проставляется для тех дефектов, которые относятся к удобству использования или интерфейсу. По аналогии с помещением и дверью – на двери написано «От себя», хотя она открывается на себя, неудобное расположение замочной скважины и т.д.
* **S5 – Тривиальный (Trivial)** – дефект, не затрагивающий функциональность системы, а также оказывающий минимальное влияние на общее качество системы. Часто неотличим от уровня «minor». Обычно это грамматические дефекты в сопроводительной документации к системе. Иногда дефект относится к «невидимым» проблемам с точки зрения пользователя или пользовательского интерфейса и рассматривает сторонние библиотеки или сервисы, не относящиеся к самой разработанной системе. По аналогии с помещением и дверью – замок и ключ не одного производителя, в помещении слышится шум сверху (не относится к самому помещению) и т.д.

Жизненный цикл дефекта



Тестировщик находит дефект оформляет отчет о дефекте в баг-трекинговую систему (статус «Новый» (New)) и назначает на разработчика (статус «Назначен» (Assigned)).

1. Разработчик проверяет дефект, воспроизводится он или нет, и присваивает ему один из статусов:

* «Дубликат» (Duplicate) – похожий дефект уже есть в баг-трекинговой системе;
* «Отклонен» (Rejected) – дефект не является веским;
* «Отсрочен» (Deferred) – исправление дефекта можно перенести в следующие версии программного продукта;
* «Не баг» (Not a bug) – в функционал программного продукта не будет внесено никаких изменений;
* «Открыт» (Open) – разработчик взял дефект в работу;
* «Исправлен» (Fixed) – разработчик внес изменения в код и проверил их.

1. Тестировщик проводит повторное тестирование дефекта (статус «Повторное тестирование» (Re-testing)).
2. Если дефект не воспроизводится, тестировщик закрывает его (статусы «Проверен» (Verified), «Закрыт» (Closed)).
3. Если дефект воспроизводится, тестировщик возвращает его разработчику на исправление (статусы «Переоткрыт» (Reopened), «Назначен» (Assigned)) и такой дефект проходит этот жизненный цикл еще раз.

SQL — это язык структурированных запросов (Structured Query Language), позволяющий хранить, манипулировать и извлекать данные из реляционных баз данных (далее — РБД, БД)

SELECT, FROM

SELECT, FROM — обязательные элементы запроса, которые определяют выбранные столбцы, их порядок и источник данных.   
  
Выбрать все (обозначается как \*) из таблицы Customers:

**SELECT** \* **FROM** Customers

Выбрать столбцы CustomerID, CustomerName из таблицы Customers:

**SELECT** CustomerID, CustomerName **FROM** Customers

WHERE

WHERE — необязательный элемент запроса, который используется, когда нужно отфильтровать данные по нужному условию. Очень часто внутри элемента where используются IN / NOT IN для фильтрации столбца по нескольким значениям, AND / OR для фильтрации таблицы по нескольким столбцам.  
  
Фильтрация по одному условию и одному значению:

**select** \* **from** Customers

**WHERE** City = 'London'

Фильтрация по одному условию и нескольким значениям с применением IN (включение) или NOT IN (исключение):

**select** \* **from** Customers

**where** City **IN** ('London', 'Berlin')

**select** \* **from** Customers

**where** City **NOT** **IN** ('Madrid', 'Berlin','Bern')

Фильтрация по нескольким условиям с применением AND (выполняются все условия) или OR (выполняется хотя бы одно условие) и нескольким значениям:

**select** \* **from** Customers

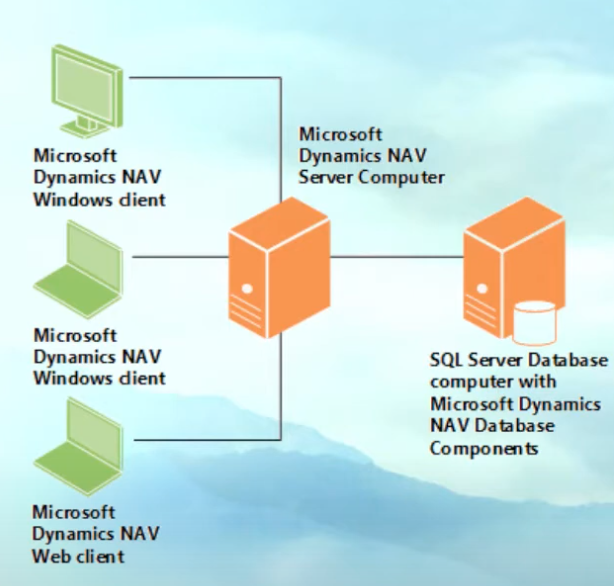
**where** Country = 'Germany' **AND** City **not** **in** ('Berlin', 'Aachen') **AND** CustomerID > 15

**select** \* **from** Customers

**where** City **in** ('London', 'Berlin') **OR** CustomerID > 4

# Клиент-серверная архитектура

Клиент-серверная архитектура – это архитектура, в которой сетевая нагрузка (не обязательно знать, что такое сетевая нагрузка) распределяется между поставщиками услуг (серверы) и заказчиками услуг (клиенты). Данная архитектура построена на основе http протокола.



КЛИЕНТ. СЕРВЕР СЕРВЕР Базы данных

Клиент отправляет http запрос на сервер, сервер отправляет запрос на базу данных. База данных обрабатывает запрос и отправляет ответ на сервер, после чего сервер отправляет ответ клиенту.

**Уровни требований:**

1. **Уровень Бизнес-требований**
2. **Уровень Пользовательских требований**
3. **Уровень Продуктных требований**

**Бизнес-требования (Business Requirements)**через них выражается цель, ради которой создается продукт (для чего продукт, какая от продукта польза, как будем получать прибыль).

Результатом сформированных требований является:

**Общее видение** (**Vision**and **Scope**) - это документ с простым текстом и таблицами, в нем нет деталей и технических характеристик.

Пример бизнес-требований:

* Нужен инструмент, в реальном времени отражающий наиболее выгодный курс покупки и продажи валюты
* Необходимо в два-три раза повысить количество заявок, обрабатываемых одним оператором за смену
* Нужно автоматизировать процесс выписки товарно-транспортных накладных на основе договоров

**Пользовательские требования (User Requirements)**описывают задачи, которые пользователь может выполнять с помощью разрабатываемого продукта, а также способы (сценарии) их решения в системе (реакция продукта на действия пользователя, сценарии работы пользователя).

Пользовательские требования представлены в виде:

* вариантов использования (uses cases)
* пользовательских историй (user stories)
* пользовательских сценариев (user scenarios)

**Бизнес-правила (Business Rules)** - описывают возможности принятых процессов, огарничений, правил.

**Продуктные требования:**

**Функциональные требования (Functional Requirements)** — охватывают предполагаемое поведение системы, определяя действия, которые система способна выполнять. Описывается в системной спецификации. В основном влияют на дизайн системы.

Пример функциональных требований:

* В процессе инсталяции приложение должно проверять остаток свободного места на целевом носителе
* Приложение не должно выгружать из памяти фоновые документы в течение 30 минут с момента выполнения с ними последней операции

**Нефункциональные требования (Non-functional Requirements) -**охватывают свойства системы (удобства использования, надежность, масштабируемость), которыми она должна обладать при реализации своего поведения.

Нефункциональные требования  в основоном влияют на архитектуру продукта.

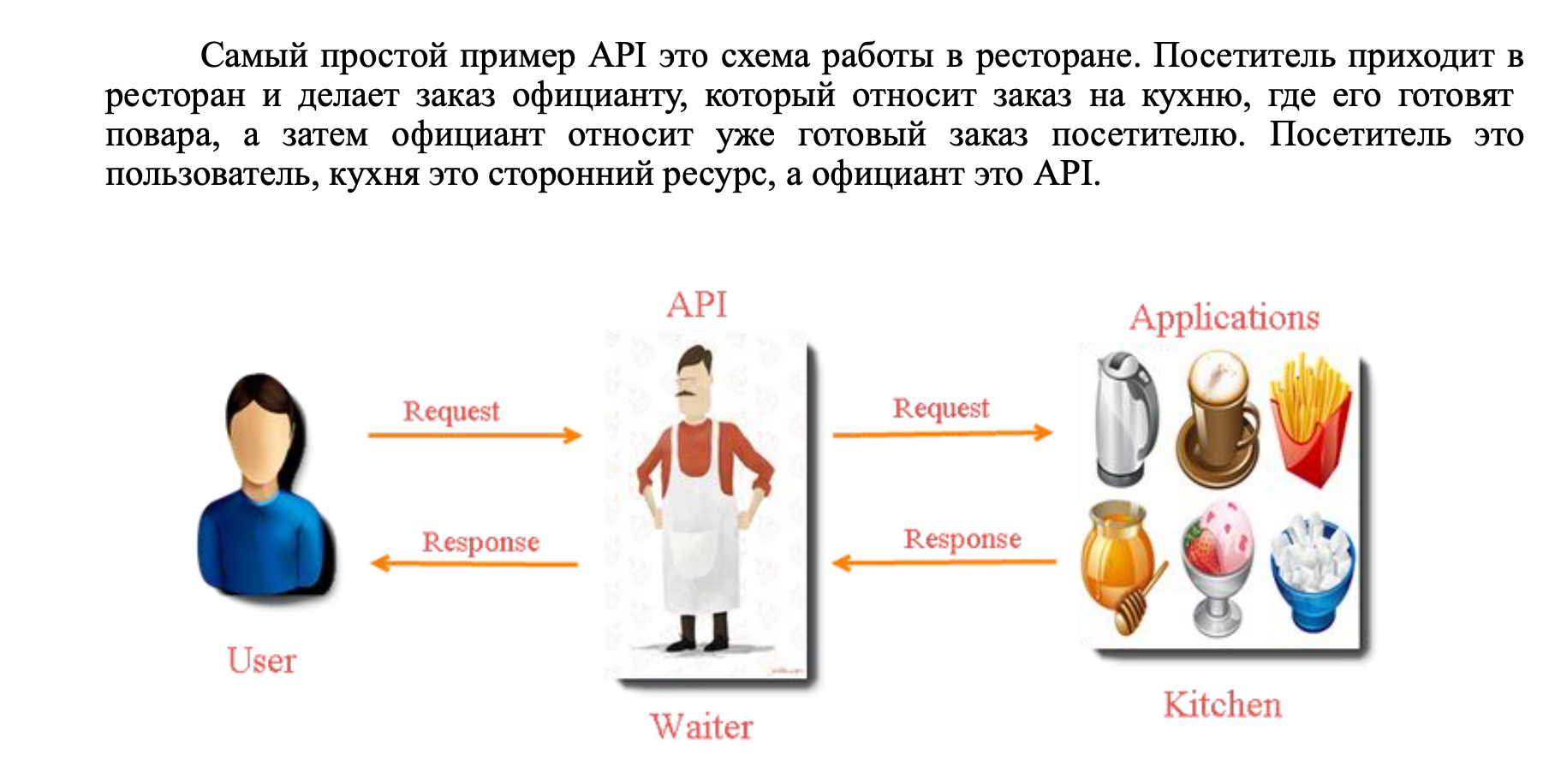
Пример нефункциональных требований:

* Вне зависимости от условий работы, общий объем используемой продуктом оперативной памяти не должен превышать 2ГБ
* При одновременной непрерывной работе с системой 1000 юзеров, время между возникновением сбоев должно быть более или равно 200 часов.

Тестирование API

Начнем с того, что такое API.

АПИ – штука/платорма, которая позволяет **общаться разным сервисам между собой**, получать/изменять информацию



От сюда делаем вывод, что АПИ – это интерфейс для интеграции («**общения/взаимодействия**») одной программки/системы с другой. Есть два способа обмена информациями между системами – Soup and Rest

Soup протокол – протокол обмена структурированными сообщениями, т.е. используется для обмена сообщениями в формате xml.

Rest – архитектурный стиль.

Отличие рест от соап

1. Рест поддерживает различные форматы (xml, json…). Соап поддерживает только хмл
2. Рест работает только с протоколами http/htpps, соап работает с различными протоколами
3. Рест не правилозависим

В процессе «общения» одной системы с другой, применяются методы, в зависимости от того, что мы хотим сделать с системой.

Метод – это тип http запроса, который указывает серверу на то, какое действие мы хотим произвести с ресурсом.

POST – добавление информации в наш ресурс (загрузить музыку, ввести пароль в поля ввода)

GET – получить информацию от нашего сервера

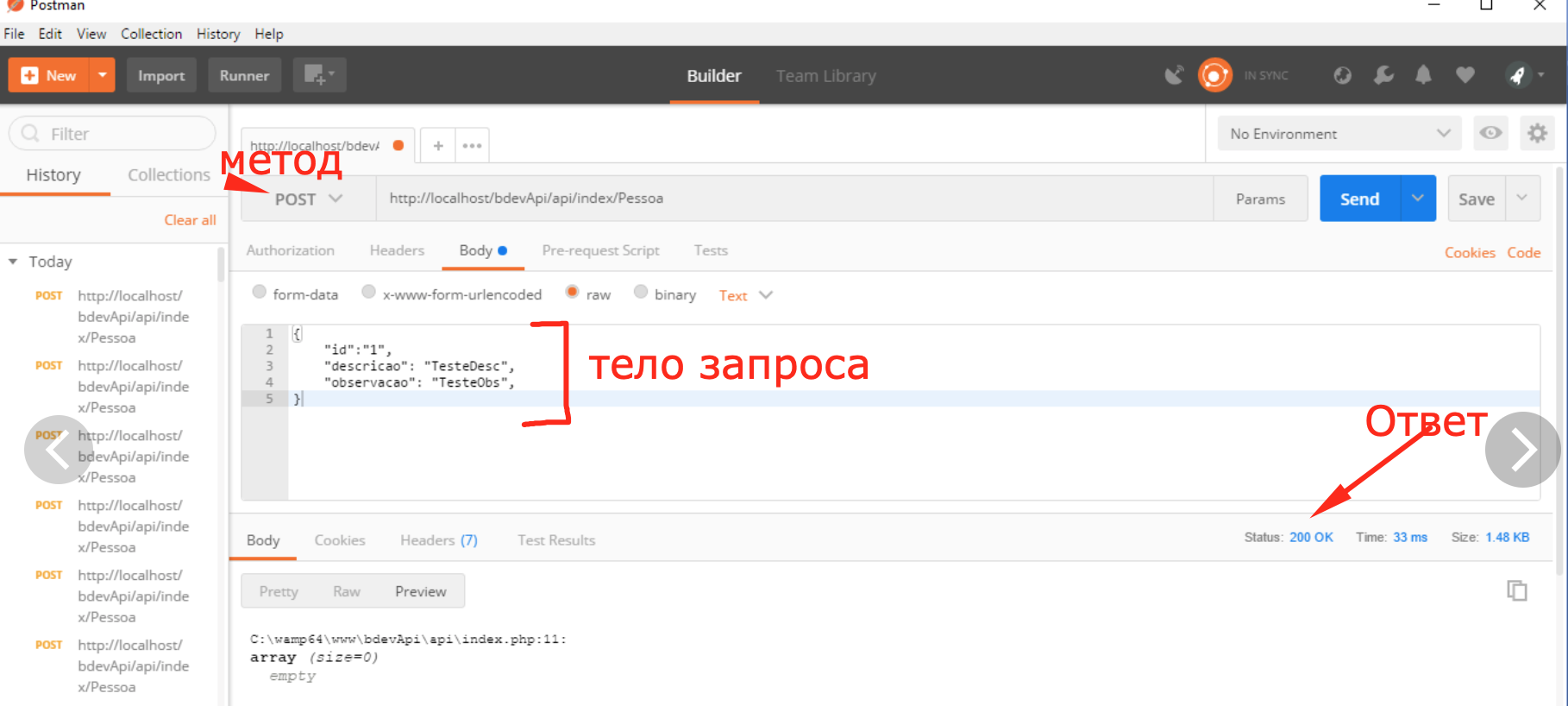
DELETE – удаление информации

PUT – заменить информацию

Отличие post от get –

Гет метод очень легко обнаружить, так как GET метод всегда это наша адресная строка, т.е. то что мы пишем в адресную строку мы хотим получить какую-то информацию + в отличии от метода пост у гет нет тела запроса (смотри скрин), мы не можем с помощью гет загрузить информацию (текст, музыка, код). Гет ограничен длинной адресной строки.

Вот так это выглядит в программе Postman (через нее тестируется АПИ)



На скрине видно название метода, там же можно выбрать и другие методы.

Изображение выглядит как стол

Автоматически созданное описание

Расскажи о себе..

Почему тестирование?

1. В чем цель тестирования?

Предоставление актуальной информации о соответсвии требованиям разработываемого продукта.

1. Тест план?

Тест план – документ, который включает в себя критерии начала и конца тестирования, расписание и дополнительный сведения о том какие техники тест дизайна будут использоваться, виды тестирования, пользователи и тд.

1. Тест кейс?

Подробный набор действий для тестирования определенного функционала. Основные поля – Название, предусловие, шаги, ОР,

1. Что такое дефект?

Основные поля в баг-репорте. Название/шаги/ор/фр/скриншоты/видео/серьезность/приоритет/версия/устройство/тестовые данные

Техники тест дизайна?

Позволяют оптимизировать процесс тестирования, уменьшить количество проверок при максимальном покрытии функционала.

Жизненный цикл ПО/Дефекта?

Виды методологий тестирования. Методология Agile (скрам/камбан).

Тестирование API??

Отличие рест от соап?

Методы запросов? Отличие пост от гет.

Статус кодов?

Виды тестирования?

Алгоритм тестирования:

- Запрашиваем требования

- Позитивные тесты

- Негативные тесты

- Не функциональные проверки (ошибки в консоли, кроссбраузерное тестирование)