

1.5.1. Список вопросов для проведения собеседования

- Теория тестирования
 - Что такое тестирование? В чем цель тестирования?
 - Какие принципы тестирования вам знакомы?
 - Что такое ошибка?
 - Priority и Severity в дефекте
 - Какие бывают требования?
 - Какие виды/типы/классы/методы тестирования вы знаете, и чем они различаются?
 - Уровни тестирования (дополнить)
 - Тестовая документация: виды, цели
 - Из каких этапов состоит процесс тестирования? Что вы знаете о жизненном цикле тестирования?
 - Какие техники тест-дизайна вам знакомы?
 - Что вы знаете о пирамиде тестирования?
- API тестирование
 - Что такое REST?
 - Что вы знаете о принципах Restful?
 - Какие методы запросов вам знакомы?
 - Какова структура REST-запроса?
 - Какова структура REST-ответа?
 - Может ли быть GET-запрос с телом?
 - Может ли быть POST-запрос без тела?
 - Что такое SOAP? В чем его основные особенности?
 - Stateless и Stateful - о чем говорят эти два понятия в контексте API?
 - Что можно протестировать на уровне API и нельзя на уровне GUI?
 - Инструменты для API тестирования
- Работа с БД
 - Типы БД
 - Три типа нормализации реляционной БД
 - Запросы в БД
 - создание бд
 - создание таблицы
 - удаление таблицы
 - обновление таблицы
 - добавление записи в таблицу
 - обновление записи в таблице
 - удаление записи из таблицы
 - получение данных из таблицы
 - копирование данных из одной таблицы в другую и добавление данных в таблицу
 - копирование данных из одной таблицы в другую
 - получение уникальных данных из таблицы
 - Разница LEFT, RIGHT, OUTER, INNER JOIN
 - Оператор SELF JOIN
 - Оператор UNION
 - Подстановочные знаки
 - Вложенные запросы
- Автоматизированное тестирование
 - Java
 - Принципы ООП
 - Модификаторы доступа
 - Перегрузка и переопределение
 - Коллекции
 - Работа с коллекциями. Работа со stream
 - Виды исключений
 - Обработка исключений
 - Final, finally, finalize (добавить)
 - Применение ключевого слова final (добавить)
 - Ключевое слово static
 - Работа gradle
 - Автоматизация
 - Автоматизированное тестирование – отдельный вид тестирования?
 - Когда требуется вводить автоматизацию тестирования на проект?
 - Что требуется, чтобы начать автоматизацию?
 - Этапы автоматизации тестирования
 - Что имеет смысл автоматизировать?
 - Что не следует автоматизировать?
 - Виды автотестов
 - Какие есть типы фреймворков автоматизации?

- Преимущества автотестирования
- Недостатки автотестирования
- Паттерны автоматизации
- Работа с css и xpath локаторами
- Selenium
- Selenide
- Явные и неявные ожидания
- Rest Assured
- Работа с отчетами
- DDT, TDD, BDD
- CI/CD/CDP
- Pipeline
- Logging
- Все ли ошибки стоит логировать? (дополнить)

Теория тестирования

Что такое тестирование? В чем цель тестирования?

Тестирование – комплекс мероприятий, направленный на проведение проверок на соответствие производимого продукта требованиям, к нему предъявляемым (прямым и косвенным). НО НЕ поиск ошибок/инцидентов/замечаний.

Цель тестирования – предоставление актуальной информации о соответствии производимого продукта требованиям.

Какие принципы тестирования вам знакомы?

- Тестирование показывает наличие дефектов

Тестирование может показать наличие дефектов в программе, но не доказать их отсутствие. Тем не менее, важно составлять тест-кейсы, которые будут находить как можно больше багов. Таким образом, при должном тестовом покрытии, тестирование позволяет снизить вероятность наличия дефектов в программном обеспечении. В то же время, даже если дефекты не были найдены в процессе тестирования, нельзя утверждать, что их нет.

- Исчерпывающее тестирование невозможно

Невозможно провести исчерпывающее тестирование, которое бы покрывало все комбинации пользовательского ввода и состояний системы, за исключением совсем уж примитивных случаев. Вместо этого необходимо использовать анализ рисков и расстановку приоритетов, что позволит более эффективно распределять усилия по обеспечению качества ПО.

- Раннее тестирование

Тестирование должно начинаться как можно раньше в жизненном цикле разработки программного обеспечения, и его усилия должны быть сконцентрированы на определенных целях.

- Скопление дефектов

Разные модули системы могут содержать разное количество дефектов – то есть, плотность скопления дефектов в разных элементах программы может отличаться. Усилия по тестированию должны распределяться пропорционально фактической плотности дефектов. В основном, большую часть критических дефектов находят в ограниченном количестве модулей. Это проявление принципа Парето: 80% проблем содержатся в 20% модулей.

- Парадокс пестицида

Прогоняя одни и те же тесты вновь и вновь, Вы столкнетесь с тем, что они находят все меньше новых ошибок. Поскольку система эволюционирует, многие из ранее найденных дефектов исправляют и старые тест-кейсы больше не срабатывают.

Чтобы преодолеть этот парадокс, необходимо периодически вносить изменения в используемые наборы тестов, рецензировать и корректировать их с тем, чтобы они отвечали новому состоянию системы и позволяли находить как можно большее количество дефектов.

- Тестирование зависит от контекста

Выбор методологии, техники и типа тестирования будет напрямую зависеть от природы самой программы. Например, программное обеспечение для медицинских нужд требует гораздо более строгой и тщательной проверки, чем, скажем, компьютерная игра. Из тех же соображений, сайт с большой посещаемостью должен пройти через серьезное тестирование производительности, чтобы показать возможность работы в условиях высокой нагрузки.

- Заблуждение об отсутствии ошибок

Тот факт, что тестирование не обнаружило дефектов, еще не значит, что программа готова к релизу. Нахождение и исправление дефектов будут не важны, если система окажется неудобной в использовании, и не будет удовлетворять ожиданиям и потребностям пользователя.

И еще несколько важных принципов:

- тестирование должно производиться независимыми специалистами;
- привлекайте лучших профессионалов;
- тестируйте как позитивные, так и негативные сценарии;
- не допускайте изменений в программе в процессе тестирования;
- указывайте ожидаемый результат выполнения тестов.

Что такое ошибка?

Ошибка (баг, дефект, баг-репорт) – несоответствие производимого продукта требованиям.

Шаблон баг-репорта

Поле	Описание
Короткое описание (Summary)	Короткое описание проблемы, явно указывающее на причину и тип ошибочной ситуации.
Проект (Project)	Название тестируемого проекта
Компонент приложения (Component)	Название части или функции тестируемого продукта
Номер версии (Version)	Версия на которой была найдена ошибка
Серьезность (Severity)	
Приоритет (Priority)	
Статус (Status)	Статус бага. Зависит от используемой процедуры и жизненного цикла бага (bug workflow and life cycle)
Автор (Author)	Создатель баг репорта
Назначен на (Assigned To)	Имя сотрудника, назначенного на решение проблемы
Окружение	
С / Сервис Пак и т.д. / Браузера + версия / ...	Информация об окружении, на котором был найден баг: операционная система, сервис пак, для WEB тестирования - имя и версия браузера и т.д.
Описание	
Шаги воспроизведения (Steps to Reproduce)	Шаги, по которым можно легко воспроизвести ситуацию, приведшую к ошибке.
Фактический Результат (Result)	Результат, полученный после прохождения шагов к воспроизведению
Ожидаемый результат (Expected Result)	Ожидаемый правильный результат
Дополнения	
Прикрепленный файл (Attachment)	Файл с логами, скриншот или любой другой документ, который может помочь прояснить причину ошибки или указать на способ решения проблемы

Priority и Severity в дефекте

Классификация дефектов, с точки зрения степени влияния (Severity) на работоспособность ПО:

- **Blocker.** Ошибка, которая приводит программу в нерабочее состояние. Дальнейшая работа с программной системой или ее функциями – невозможна.
- **Critical.** Критический дефект, приводящий некоторый ключевой функционал в нерабочее состояние. Так же это может быть существенное отклонение от бизнес логики, неправильная реализация требуемых функций, потеря пользовательских данных и т.д.
- **Major.** Весьма серьезная ошибка, свидетельствующая об отклонении от бизнес логики или нарушающая работу программы. Не имеет критического воздействия на приложение.
- **Minor.** Незначительный дефект, не нарушающий функционал тестируемого приложения, но который является несоответствием ожидаемому результату. Например, ошибка дизайна.
- **Trivial.** Баг, не имеющий влияние на функционал или работу программы, но который может быть обнаружен визуально. Например, ошибка в тексте.

Градация дефектов, с точки зрения приоритетности исправления (Priority):

- **High.** Баг должен быть исправлен как можно быстрее, т.к. он критически влияет на работоспособность программы.
- **Medium.** Дефект должен быть обязательно исправлен, но он не оказывает критическое воздействие на работу приложения.

- Low. Ошибка должна быть исправлена, но она не имеет критического влияния на программу и устранение может быть отложено, в зависимости от наличия других более приоритетных дефектов.

Какие бывают требования?

- прямые (т. е. формализованные в технической документации, спеках, юзер-стори и прочих формальных артефактах). Прямые требования всегда приоритетнее косвенных
- косвенные (т. е. проистекающие из прямых, либо являющимися негласным стандартом для данной продукции или основывающиеся на опыте и здравом смысле использования данного продукта или продуктов, подобных ему)
- функциональные (описывающие какие функции должен выполнять продукт)
- нефункциональные (требования к окружению, поддерживаемости, надежности и прочим характеристикам продукта)

Какие виды/типы/классы/методы тестирования вы знаете, и чем они различаются?

Методы тестирования

- черный ящик
- белый ящик
- серый ящик

Виды тестирования (по Куликову):

По запуску кода на исполнение:

- Статическое тестирование - без запуска
- Динамическое тестирование - с запуском

По задачам:

- функциональное
- нефункциональное (удобство использования, совместимость, производительность, безопасность и т.д.)

По степени автоматизации:

- Ручное тестирование - тест-кейсы выполняет человек.
- Автоматизированное тестирование - тест-кейсы частично или полностью выполняет специальное инструментальное средство.

По уровню детализации приложения (по уровню тестирования):

- Приемочное тестирование (acceptance testing) - формализованное тестирование, направленное на проверку приложения с точки зрения конечного пользователя/заказчика и вынесения решения о том, принимает ли заказчик работу у исполнителя (проектной команды).
- Модульное (компонентное) тестирование - проверяются отдельные небольшие части приложения.
- Интеграционное тестирование - проверяется взаимодействие между несколькими частями приложения.
- Системное тестирование - приложение проверяется как единое целое.

По (убыванию) степени важности тестируемых функций (по уровню функционального тестирования):

- Дымовое тестирование (smoke) - проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения.
- Тестирование критического пути (critical path) - проверка функциональности, используемой типичными пользователями в типичной повседневной деятельности.
- Расширенное тестирование (extended test) - проверка всей (остальной) функциональности, заявленной в требованиях.

По принципам работы с приложением:

- Позитивное тестирование - все действия с приложением выполняются строго по инструкции без никаких недопустимых действий, некорректных данных и т.д. Можно образно сказать, что приложение исследуется в «тепличных условиях».
- Негативное тестирование - в работе с приложением выполняются (некорректные) операции и используются данные, потенциально приводящие к ошибкам (классика жанра - деление на ноль).

Классификация по природе приложения:

- Тестирование веб-приложений (web-applications testing) сопряжено с интенсивной деятельностью в области тестирования совместимости (в особенности - кросс-браузерного тестирования), тестирования производительности, автоматизации тестирования с использованием широкого спектра инструментальных средств.
- Тестирование мобильных приложений (mobile applications testing) также требует повышенного внимания к тестированию совместимости, оптимизации производительности (в том числе клиентской части с точки зрения снижения энергопотребления), автоматизации тестирования с применением эмуляторов мобильных устройств.

- Тестирование настольных приложений (desktop applications testing) является самым классическим среди всех перечисленных в данной классификации, и его особенности зависят от предметной области приложения, нюансов архитектуры, ключевых показателей качества и т.д.

Классификация по фокусировке на уровне архитектуры приложения:

- Тестирование уровня представления (presentation tier testing) сконцентрировано на той части приложения, которая отвечает за взаимодействие с «внешним миром» (как пользователями, так и другими приложениями). Здесь исследуются вопросы удобства использования, скорости отклика интерфейса, совместимости с браузерами, корректности работы интерфейсов.
- Тестирование уровня бизнес-логики (business logic tier testing) отвечает за проверку основного набора функций приложения и строится на базе ключевых требований к приложению, бизнес-правил и общей проверки функциональности.
- Тестирование уровня данных (data tier testing) сконцентрировано на той части приложения, которая отвечает за хранение и некоторую обработку данных (чаще всего - в базе данных или ином хранилище). Здесь особый интерес представляет тестирование данных, проверка соблюдения бизнес-правил, тестирование производительности.

Классификация по привлечению конечных пользователей:

- Альфа-тестирование (alpha testing) выполняется внутри организации-разработчика с возможным частичным привлечением конечных пользователей. Может являться формой внутреннего приёмочного тестирования. В некоторых источниках отмечается, что это тестирование должно проводиться без привлечения команды разработчиков, но другие источники не выдвигают такого требования. Суть этого вида вкратце: продукт уже можно периодически показывать внешним пользователям, но он ещё достаточно «сырой», потому основное тестирование выполняется организацией-разработчиком.
- Бета-тестирование (beta testing) выполняется вне организации-разработчика с активным привлечением конечных пользователей/заказчиков. Может являться формой внешнего приёмочного тестирования. Суть этого вида вкратце: продукт уже можно открыто показывать внешним пользователям, он уже достаточно стабилен, но проблемы всё ещё могут быть, и для их выявления нужна обратная связь от реальных пользователей.
- Гамма-тестирование (gamma testing) - финальная стадия тестирования перед выпуском продукта, направленная на исправление незначительных дефектов, обнаруженных в бета-тестировании. Как правило, также выполняется с максимальным привлечением конечных пользователей/заказчиков. Может являться формой внешнего приёмочного тестирования. Суть этого вида вкратце: продукт уже почти готов, и сейчас обратная связь от реальных пользователей используется для устранения последних недоработок.

Иные виды:

- Инсталляционное тестирование (installation testing, installability testing) - тестирование, направленное на выявление дефектов, влияющих на протекание стадии инсталляции (установки) приложения.
- Регрессионное тестирование (regression testing) - тестирование, направленное на проверку того факта, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования.
- Повторное тестирование (re-testing) - тестирование, проводимое в реальной или приближенной к реальной операционной среде (operational environment), включающей операционную систему, системы управления базами данных, серверы приложений, веб-серверы, аппаратное обеспечение и т.д.
- Тестирование удобства использования (usability testing) - тестирование, направленное на исследование того, насколько конечному пользователю понятно, как работать с продуктом (understandability, learnability, operability), а также на то, насколько ему нравится использовать продукт (attractiveness), confirmation testing) - выполнение тест-кейсов, которые ранее обнаружили дефекты, с целью подтверждения устранения дефектов.
- Тестирование доступности (accessibility testing) - тестирование, направленное на исследование пригодности продукта к использованию людьми с ограниченными возможностями (слабым зрением и т.д.).
- Тестирование интерфейса (interface testing) - тестирование, направленное на проверку интерфейсов приложения или его компонентов.
- Тестирование безопасности (security testing) - тестирование, направленное на проверку способности приложения противостоять злонамеренным попыткам получения доступа к данным или функциям, права на доступ к которым у злоумышленника нет.
- Тестирование интернационализации (internationalization testing, in testing, globalization testing, localizability testing) - тестирование, направленное на проверку готовности продукта к работе с использованием различных языков и с учётом различных национальных и культурных особенностей. Этот вид тестирования не подразумевает проверки качества соответствующей адаптации.
- Тестирование локализации (localization testing) - тестирование, направленное на проверку корректности и качества адаптации продукта к использованию на том или ином языке с учётом национальных и культурных особенностей. Это тестирование следует за тестированием интернационализации и проверяет корректность перевода и адаптации продукта, а не готовность продукта к таким действиям.
- Тестирование совместимости (compatibility testing, interoperability testing) - тестирование, направленное на проверку способности приложения работать в указанном окружении.
- Тестирование данных (data quality testing) и баз данных (database integrity testing) - два близких по смыслу вида тестирования, направленных на исследование таких характеристик данных, как полнота, непротиворечивость, целостность, структурированность и т.д.
- Тестирование использования ресурсов (resource utilization testing, efficiency testing, storage testing) - совокупность видов тестирования, проверяющих эффективность использования приложением доступных ему ресурсов и зависимость результатов работы приложения от количества доступных ему ресурсов.
- Сравнительное тестирование (comparison testing) - тестирование, направленное на сравнительный анализ преимуществ и недостатков разрабатываемого продукта по отношению к его основным конкурентам.

- Демонстрационное тестирование (qualification testing) - формальный процесс демонстрации заказчику продукта с целью подтверждения, что продукт соответствует всем заявленным требованиям. В отличие от приёмочного тестирования этот процесс более строгий и всеобъемлющий, но может проводиться и на промежуточных стадиях разработки продукта.
- Исчерпывающее тестирование (exhaustive testing) - тестирование приложения со всеми возможными комбинациями всех возможных входных данных во всех возможных условиях выполнения. Для сколь бы то ни было сложной системы нереализуемо, но может применяться для проверки отдельных крайне простых компонентов.
- Тестирование надёжности (reliability testing) - тестирование способности приложения выполнять свои функции в заданных условиях на протяжении заданного времени или заданного количества операций.
- Тестирование восстанавливаемости (recoverability testing) - тестирование способности приложения восстанавливать свои функции и заданный уровень производительности, а также восстанавливать данные в случае возникновения критической ситуации, приводящей к временной (частичной) утрате работоспособности приложения.
- Тестирование отказоустойчивости (failover testing) - тестирование, заключающееся в эмуляции или реальном создании критических ситуаций с целью проверки способности приложения задействовать соответствующие механизмы, предотвращающие нарушение работоспособности, производительности и повреждения данных.
- Тестирование производительности (performance testing) - исследование показателей скорости реакции приложения на внешние воздействия при различной по характеру и интенсивности нагрузке. В рамках тестирования производительности выделяют следующие подвиды:
 - Нагрузочное тестирование (load testing, capacity testing) - исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).
 - Тестирование масштабируемости (scalability testing) - исследование способности приложения увеличивать показатели производительности в соответствии с увеличением количества доступных приложению ресурсов.
 - Объёмное тестирование (volume testing) - исследование производительности приложения при обработке различных (как правило, больших) объёмов данных.
 - Стрессовое тестирование (stress testing) - исследование поведения приложения при нештатных изменениях нагрузки, значительно превышающих расчётный уровень, или в ситуациях недоступности значительной части необходимых приложению ресурсов. Стрессовое тестирование может выполняться и вне контекста нагрузочного тестирования: тогда оно, как правило, называется «тестированием на разрушение» (destructive testing) и представляет собой крайнюю форму негативного тестирования}.
 - Конкурентное тестирование (concurrency testing) - исследование поведения приложения в ситуации, когда ему приходится обрабатывать большое количество одновременно поступающих запросов, что вызывает конкуренцию между запросами за ресурсы (базу данных, память, канал передачи данных, дисковую подсистему и т.д.). Иногда под конкурентным тестированием понимают также исследование работы многопоточных приложений и корректность синхронизации действий, производимых в разных потоках.
- Санитарное тестирование - это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Используется для определения работоспособности определенной части приложения после изменений произведенных в ней или окружающей среде. Обычно выполняется вручную.

Отличие санитарного тестирования от дымового (Sanity vs Smoke testing)

В некоторых источниках ошибочно полагают, что санитарное и дымовое тестирование - это одно и то же. Мы же полагаем, что эти виды тестирования имеют "вектора движения", направления в разные стороны. В отличие от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

Уровни тестирования (дополнить)

Тестовая документация: виды, цели

Внешняя документация:

Замечание – короткая записка, комментарий о небольшой неточности в реализации продукта.

Баг-репорт – описание выявленного случая несоответствия производимого продукта требованиям, к нему выдвигаемым – ошибки или ее проявления. Он обязательно должен содержать следующие элементы:

- Идею тестового случая, вызвавшего ошибку.
- Описание исходного состояния системы для выполнения кейса.
- Шаги, необходимые для того, чтобы выявить ошибку или ее проявление.
- Ожидаемый результат, т. е. то, что должно было произойти в соответствии с требованиями.
- Фактический результат, т. е. то, что произошло на самом деле.
- Входные данные, которые использовались во время воспроизведения кейса.
- Прочую информацию, без которой повторить кейс не получится.
- Критичность и/или приоритет.
- Экранный снимок (скрин).
- Версию, сборку, ресурс и другие данные об окружении.

Запрос на изменение (улучшение) – описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути /рекомендации по модификации продукта для соответствия им.

Отчет о тестировании (тест репорт) – документ, предоставляющий сведения о соответствии/ несоответствии продукта требованиям. Может так же содержать описание некоторых подробностей проведенной сессии тестирования, например, затраченное время, использованные виды тестирования, перечень проверенных случаев и т. п. В идеальном варианте фраза вида «Тест пройден. Ошибка не воспроизводится/Функционал работает корректно/Соответствует требованиям» означает, что продукт или его часть полностью соответствует требованиям прямым и косвенным (в производстве ПО).

Внутренняя документация:

Тест-план (план тестирования) – формализованное и укрупненное описание одной сессии тестирования по одному или нескольким направлениям проверок. Т.е. перечень направлений проверок, которые должны быть проведены в рамках сессии тестирования (и, сообразных этим направлениям, требований). Также может содержать в себе необходимую информацию об окружении, методике, прочих условиях важных для показательности данной сессии тестирования. Под направлением проверок также может пониматься более детализированная тестовая документация (в виде ссылки на нее): чек листы, тестовые комплекты, тестовые сценарии, на которую необходимо опираться при проведении сессии тестирования. Основная цель документа – описать границы сессии тестирования, стабилизировать показательности данной сессии.

Тестовый сценарий – последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразных им проверок корректности поведения продукта в ходе этих действий. Может содержать информацию об исходном состоянии продукта для запуска сценария, входных данных и прочие сведения, имеющие определяющее значение для успешного и показательного проведения проверок по сценарию. Особенностью является линейность действий и проверок, т.е. зависимость последующих действий и проверок от успешности предыдущих. Цель документа – стабилизация покрытия аспектов продукта, необходимых для выполнения функциональной задачи, показательными необходимыми и достаточными проверками. Фактически при успешном прохождении всего тестового сценария мы можем сделать заключение о том, что продукт может выполнять ту или иную возложенную на него функцию.

Тестовый комплект – некоторый набор формализованных тестовых случаев объединенных между собой по общему логическому признаку.

Чек-лист (лист проверок) – перечень формализованных тестовых случаев в виде удобном для проведения проверок. Тестовые случаи в чек-листе не должны быть зависимыми друг от друга. Обязательно должен содержать в себе информацию о: идеях проверок, наборах входных данных, ожидаемых результатах, булеву отметку о прохождении /непрохождении тестового случая, булеву отметку о совпадении/несовпадении фактического и ожидаемого результата по каждой проверке. Может так же содержать шаги для проведения проверки, данные об особенностях окружения и прочую информацию необходимую для проведения проверок. Цель – обеспечить стабильность покрытия требований проверками необходимыми и достаточными для заключения о соответствии им продукта. Особенностью является то, что чек-листы komponуются теми тестовыми случаями, которые показательны для определенного требования.

Тестовый случай (тест-кейс) – формализованное описание одной показательной проверки на соответствие требованиям прямым или косвенным. Обязательно должен содержать следующую информацию:

- Идея проверки.
- Описание проверяемого требования или проверяемой части требования.
- Используемое для проверки тестовое окружение.
- Исходное состояние продукта перед началом проверки.
- Шаги для приведения продукта в состояние, подлежащее проверке.
- Входные данные для использования при воспроизведении шагов.
- Ожидаемый результат.
- Прочую информацию, необходимую для проведения проверки.

Цель – зафиксировать сгенерированную и отобранную показательную проверку в виде, позволяющем тестировщику любой квалификации ее провести и суметь проанализировать полученные результаты.

Как видно, каждый последующий вид внутренней тестовой документации в определенной мере детализирует предыдущий. У каждого документа есть свое назначение и все вместе они – инструмент для облегчения генерации, отбора и воспроизведения тестовых случаев. Кроме того хорошо структурированная, поддерживаемая, читаемая, организованная и доступная тестовая документация позволяет в долгосрочной перспективе:

- Обеспечить стабильность покрытия требований проверками.
- Обеспечить показательность всех проводимых проверок.
- Обеспечить необходимость и достаточность проводимых проверок.
- Сэкономить время на этапах тестирования, сводя их к проведению проверок и анализу и передаче результатов.
- Снизить входной уровень квалификации тестировщика для проведения проверок.
- Повысить прогнозируемость сессий тестирования в части затрат времени и ресурсов.
- Повысить прозрачность процесса тестирования для других участников процесса производства продукта.
- Обеспечить базу знаний о продукте и истории его развития.

Из каких этапов состоит процесс тестирования? Что вы знаете о жизненном цикле тестирования?

Инициация – событие, которое извещает команду тестирования о необходимости сессии тестирования, а также гарантирует выполнение требований к продукту для проведения тестирования.

Для производства ПО требования включают:

- доступно необходимое тестовое окружение,
- доступен билд/ресурс/предмет тестирования,
- код, БД, прочие компоненты объекта тестирования «заморожены», т. е. не изменяются в период всей сессии тестирования,
- модификация требований (хотя бы прямых) «заморожена»,
- известно направление тестирования,
- известны сроки на сессию тестирования.

Выявление требований – пожалуй, один из главных шагов в процессе тестирования. Неизвестны требования – нет тестирования. Необходимо собрать всю доступную информацию о предмете тестирования, вариантах использования и т. п. Первый источник – техническая документация и юзер-стори – это прямые требования. Качество же косвенных требований во многом зависит от добросовестности, ответственности, квалификации тестировщика и всей команды проекта.

Генерация тестовых случаев – выявление всех возможных случаев использования продукта, его характеристик и особенностей в процессе эксплуатации. Это значит: всех случаев, которые тестировщик может «придумать» на основе прямых и косвенных требований, известных ему. Этот этап требует высокой квалификации специалиста по тестированию.

Отбор тестовых случаев – отбор наиболее показательных, значимых и воспроизводимых тестовых случаев. От этого этапа зависит, насколько тестирование будет полезным, эффективным и анализируемым.

Проведение проверок – тут все понятно. Либо согласно документации, либо ad hoc (интуитивно, свободный поиск, без документации). В любом случае это проводится согласно списку отобранных проверок. Почему-то большинство именно этот пункт называет тестированием. И в голове обывателя, незнакомого с профессией, только один этот пункт и содержится J.

Фиксация результатов – создание внутренней и внешней тестовой документации в формализованном виде или в виде записей и т. п. На данном этапе отчет о тестировании даже если и создается, то не считается законченным.

Анализ результатов – вынесение решения о соответствии проверенного продукта требованиям. Формализация данного решения и его обоснование в виде отчета о тестировании. Сюда также входят процедуры по оценке покрытия требований проверками, тайм-шitting и пр. Таким образом, проводится анализ не только результатов, но и самой сессии тестирования.

Передача информации о соответствии продукта требованиям. Формально: передача внешней тестовой документации заинтересованным в ней сторонам, зачастую инициатору сессии тестирования. В общем случае: помимо документации предоставляется информация о рисках, которые были выявлены в продукте, требованиях, процессах, передаются рекомендации по обработке этих рисков и т. п.

Какие техники тест-дизайна вам знакомы?

- Эквивалентное Разделение (Equivalence Partitioning - EP)

Тестовые данные разбиваются на определенные классы допустимых значений. В рамках каждого класса выполнение теста с любым значением тестовых данных приводит к эквивалентному результату. После определения классов необходимо выполнить хотя бы один тест в каждом классе.

Тестируется модуль для HR, который определяет возможность принятия на работу кандидата в зависимости от его возраста.

Установлены следующие условия отбора:

при возрасте от 0 до 16 лет – не нанимать;
при возрасте от 16 до 18 лет – можно нанять только на part time;
при возрасте от 18 до 55 лет – можно нанять на full time;
при возрасте от 55 до 99 лет – не нанимать.

- Анализ Граничных Значений (Boundary Value Analysis - BVA)

Эта техника основана на том факте, что одним из самых слабых мест любого программного продукта является область граничных значений. Для начала выбираются диапазоны значений – как правило, это классы эквивалентности. Затем определяются границы диапазонов. На каждую из границ создается 3 тест-кейса: первый проверяет значение границы, второй – значение ниже границы, третий – значение выше границы.

- Причина / Следствие (Cause/Effect - CE)

Эта техника помогает:

- а) определить минимальное количество тестов для нахождения максимума ошибок.
- б) выяснить все причины и следствия – таким образом, мы убедимся, что на любые манипуляции с системой у системы будет ответ.
- в) найти возможные недочеты в логике описания приложения (что, в свою очередь, поможет улучшить документацию).

Например, QA-специалист тестирует приложение типа “записная книжка”. После ввода всех данных нового контакта и нажатия кнопки Создать (причина) приложение должно автоматически создать карточку с номером телефона, фотографией и ФИО человека (следствие). Тесты покажут, можно ли оставлять одно или несколько полей пустыми, распознает ли система кириллицу, латиницу или оба алфавита, а также другие параметры.

- Предугадывание ошибки (Error Guessing - EG)

Используя свои знания о системе, QA-специалист может «предугадать», при каких входных условиях есть риск ошибок. Для этого важно иметь опыт, хорошо знать продукт и уметь выстроить коммуникации с коллегами. Например, в спецификации указано, что поле должно принимать код из четырех цифр. В числе возможных тестов:

- Что произойдет, если не ввести код?
- Что произойдет, если не ввести спецсимволы?
- Что произойдет, если ввести не цифры, а другие символы?
- Что произойдет, если ввести не четыре цифры, а другое количество?

Преимущества:

1. Эта проверка эффективна в качестве дополнения к другим техникам.
2. Выявляет тестовые случаи, которые “никогда не должны случиться”.

Недостатки:

1. Техника в значительной степени основана на интуиции.
2. Необходим опыт в тестировании подобных систем.
3. Малое покрытие тестами.

- **Исчерпывающее тестирование (Exhaustive Testing - ET)**

В пределах этой техники необходимо проверить все возможные комбинации входных значений, и в принципе, это должно найти все проблемы. На практике применение этого метода не представляется возможным, из-за огромного количества входных значений.

- **Ad-hoc тестирование**

Под ad-hoc тестированием будем понимать тестирование без использования спецификаций, планов и разработанных тест-кейсов: чистая импровизация.

- **Исследовательское тестирование**

Более формальная версия ad-hoc: тестирование, не требующее написания тест-кейсов, но подразумевающее, что каждый последующий тест выбирается на основании результата предыдущего теста.

- **Попарные значения**

Суть этого метода, также известного как pairwise testing, в том, что каждое значение каждого проверяемого параметра должно быть протестировано на взаимодействие с каждым значением всех остальных параметров. После составления такой матрицы мы убираем тесты, которые дублируют друг друга, оставляя максимальное покрытие при минимальном необходимом наборе сценариев.

№	Браузер	Операционная система	Язык
1	Opera	Windows	RU
2	Google Chrome	Linux	RU
3	Opera	Linux	EN
4	Google Chrome	Windows	EN

- **Таблица принятия решений**

Таблицы решений – это удобный инструмент для фиксирования требований и описания функциональности приложения. Таблицами очень удобно описывать бизнес-логику приложения, и они могут служить отличной основой для создания тест-кейсов.

	Тест 1	Тест 2	Тест 3	Тест 4
Условия				
Состоит в браке	Да	Да	Нет	Нет
Хороший студент	Да	Нет	Да	Нет

- **Техника состояний и переходов**

Система переходит в то или иное состояние в зависимости от того, какие операции над ней выполняются.

Состояние (state, представленное в виде круга на диаграмме) – это состояние приложения, в котором оно ожидает одно или более событий. Состояние помнит входные данные, полученные до этого, и показывает, как приложение будет реагировать на полученные события. События могут вызывать смену состояния и/или инициировать действия.

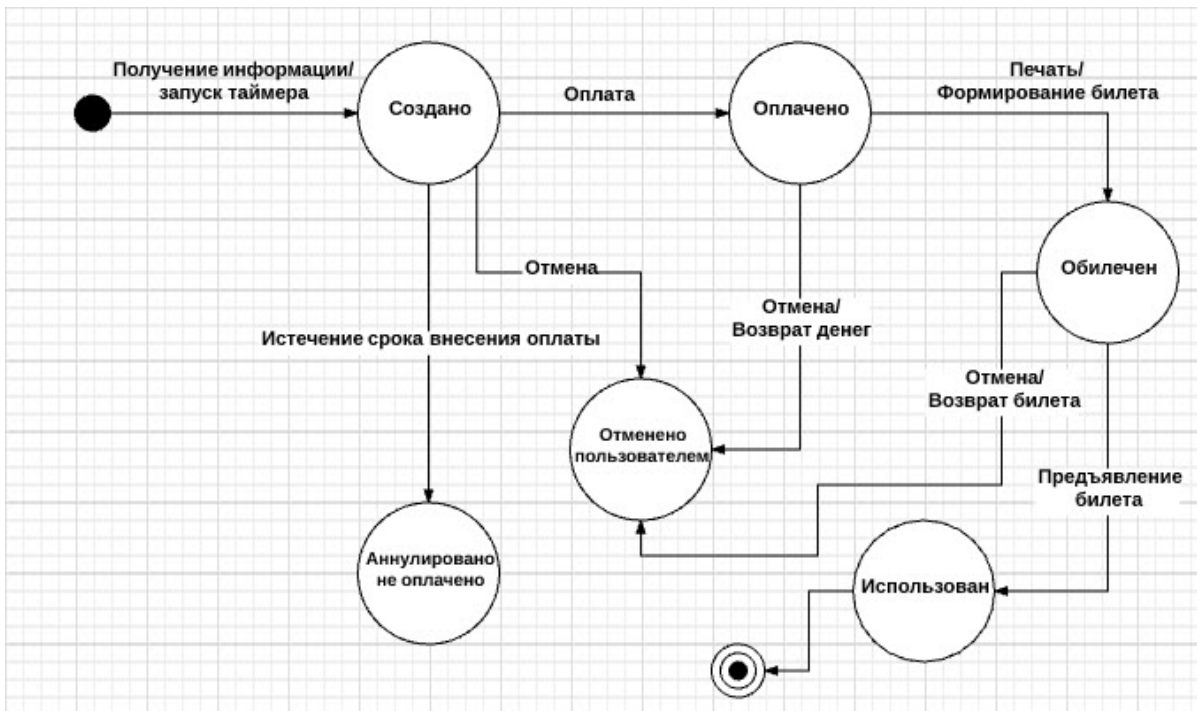
Переход (transition, представлено в виде стрелки на диаграмме) – это преобразование одного состояния в другое, происходящее по событию.

Событие (event, представленное ярлыком над стрелкой) – это что-то, что заставляет приложение поменять свое состояние. События могут поступать извне приложения, через интерфейс самого приложения. Само приложение также может генерировать события (например, событие «истек таймер»). Когда происходит событие, приложение может поменять (или не поменять) состояние и выполнить (или не выполнить) действие. События могут иметь параметры (например, событие «Оплата» может иметь параметры «Наличные деньги», «Чек», «Приходная карта» или «Кредитная карта»).

Действие (action, представлено после «/» в ярлыке над переходом) инициируется сменой состояния («напечатать билет», «показать на экране» и др.). Обычно действия создают что-то, что является выходными/возвращаемыми данными системы. Действия возникают при переходах, сами по себе состояния пассивны.

Точка входа обозначается черным кружком.

Точка выхода показывается на диаграмме в виде мишени.



• Mind Map

Интеллект-карта содержит в себе не только все важные элементы, но и наглядно иллюстрирует взаимосвязи между ними. Благодаря этому, в случае появления бага в одной части программы, невозможно пропустить все связанные с этой частью места и элементы и проверить все функциональности, которые могли оказаться сломанными. А расставив на карте приоритеты, не будут упущены из внимания наиболее важные части функционала.

К ключевым преимуществам интеллект-карт относятся:

- четкое структурирование информации;
- возможность собрать в одном месте всю информацию, относящуюся к проекту;
- охват всех связей проекта;
- представление единой картины проекта.
- **Доменный анализ**

Это техника основана на разбиении диапазона возможных значений переменной (или переменных) на поддиапазоны (или домены), с последующим выбором одного или нескольких значений из каждого домена для тестирования. Во многом доменное тестирование пересекается с известными нам техниками разбиения на классы эквивалентности и анализа граничных значений. Но доменное тестирование не ограничивается перечисленными техниками. Оно включает в себя как анализ зависимостей между переменными, так и поиск тех значений переменных, которые несут в себе большой риск (не только на границах).

• Сценарий использования

Use Case описывает сценарий взаимодействия двух и более участников (как правило – пользователя и системы). Пользователем может выступать как человек, так и другая система. Для тестировщиков Use Case являются отличной базой для формирования тестовых сценариев (тест-кейсов), так как они описывают, в каком контексте должно производиться каждое действие пользователя. Use Case, по умолчанию, являются тестируемыми требованиями, так как в них всегда указана цель, которой нужно достигнуть, и шаги, которые надо для этого воспроизвести.

Что вы знаете о пирамиде тестирования?

«Пирамида тестов» - абстракция, которая означает группировку тестов программного обеспечения по разным уровням детализации.

Она также даёт представление, сколько тестов должно быть в каждой из этих групп.



Два принципа:

1. Писать тесты разной детализации.
2. Чем выше уровень, тем меньше тестов.

Придерживайтесь формы пирамиды, чтобы придумать здоровый, быстрый и поддерживаемый набор тестов.

- Напишите *много* маленьких и быстрых юнит-тестов
- Напишите *несколько* более общих тестов
- и *совсем мало* высокоуровневых сквозных тестов, которые проверяют приложение от начала до конца.

Подробнее о Классической пирамиде тестирования:

Тестовая пирамида - визуализация, описывающая различные уровни тестирования и объем тестирования на каждом слое тестирования.

Модульные тесты должны составлять основную часть автоматизированного тестирования.

- Задачи автоматизации не закрываются до тех пор, пока эти скрипты не будут запущены на реализованной функциональности;
- Разработка одновременно с модульными тестами заставляет разработчиков задуматься о проблеме, которую они решают, и о любых крайних случаях, с которыми они могут столкнуться;
- Тесты являются детальными и могут помочь точно определить дефект;
- Время выполнения невероятно быстрое, потому что им не нужно полагаться на какой-либо пользовательский интерфейс или внешние системы, такие как база данных или API;
- Они недорогие, просто пишутся, легко поддерживать.

Интеграционные тесты должны занимать середину пирамиды.

Используйте этот уровень для проверки бизнес-логики без использования пользовательского интерфейса (UI);

Тестируя за пределами пользовательского интерфейса, вы можете тестировать входы и выходы API или сервисов без всех сложностей, которые вводит пользовательский интерфейс;

Эти тесты медленнее и сложнее, чем модульные тесты, потому что им может потребоваться доступ к базе данных или другим компонентам.

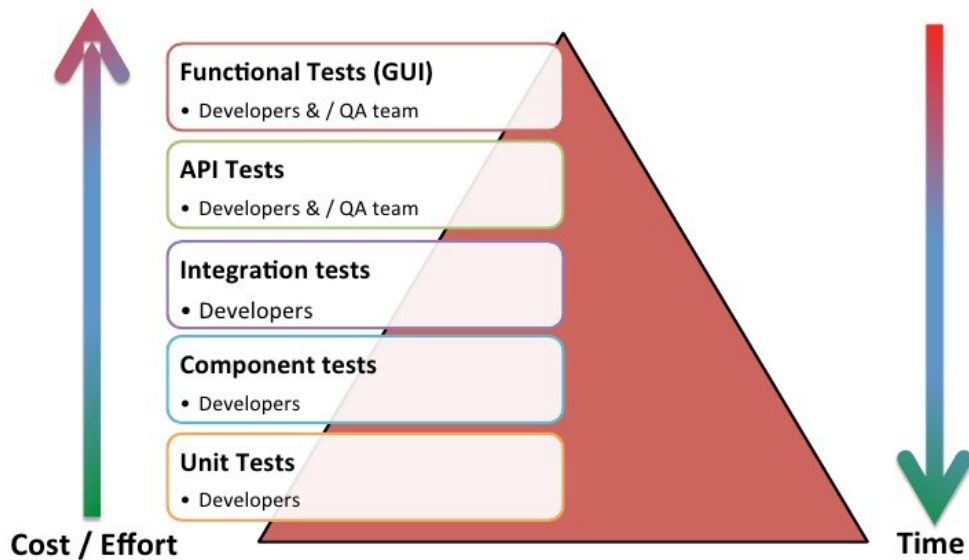
Тесты пользовательского интерфейса должны размещаться на вершине пирамиды.

Большая часть вашего кода и бизнес-логики должна быть уже протестирована до этого уровня;

Тесты интерфейса пишутся, чтобы убедиться, что сам интерфейс работает правильно;

Тесты пользовательского интерфейса медленнее и тяжелее в написании и поддержке, поэтому необходимо сводить их к минимуму.

Ideal Test Pyramid

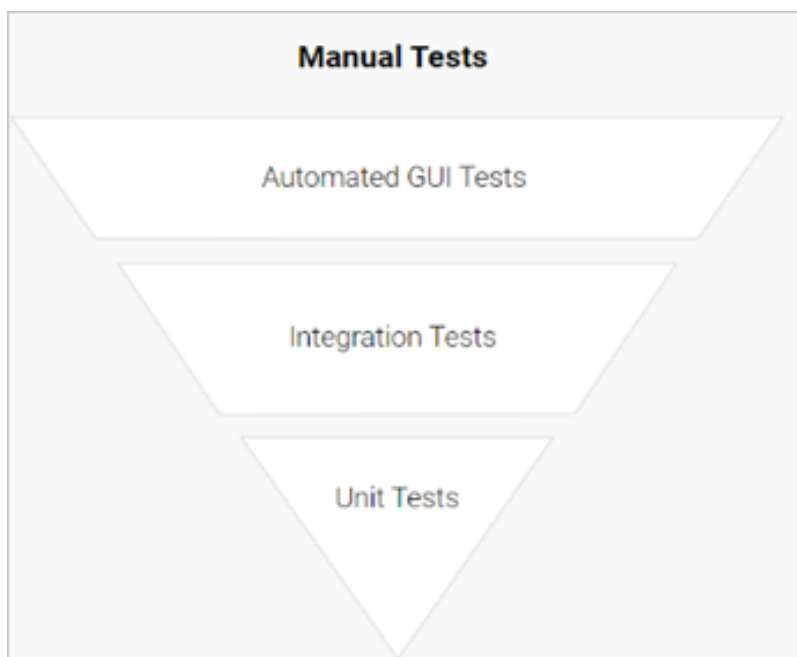


Перевернутая пирамида тестирования или "Рожок мороженого"

Перевернутая пирамида считается не рекомендованной, хотя в практике такой подход встречается.

Основные тезисы:

1. Тесты пользовательского интерфейса должны автоматизироваться в большей мере.
2. Длинные тестовые прогоны. Время выполнения занимает намного больше времени, чем другие типы тестов, потому что оно основано на взаимодействии с визуальными элементами пользовательского интерфейса и не обязательно имеет хуки в исходном коде;
3. Сложно поддерживать, так как тесты пользовательского интерфейса сложно писать и они очень сильно зависят даже от малейших изменений;
4. Больше подходит для сценариев позитивного пути. Тестирование отрицательных путей в сквозных тестах очень затратно и долго выполняется по сравнению с тестами более низкого уровня;
5. Ожидание написания модульных тестов до тех пор, пока функции не будут завершены, может привести к тому, что каждому придется несколько раз выполнить большую работу для решения проблемы.



API тестирование

Что такое REST?

Representational State Transfer - передача состояния представления.

Это архитектурный стиль взаимодействия компонентов распределенной системы в компьютерной сети. Проще говоря, REST определяет стиль взаимодействия (обмена данными) между разными компонентами системы, каждая из которых может физически располагаться в разных местах.

Данный архитектурный стиль представляет собой согласованный набор ограничений, учитываемых при проектировании распределенной системы.

Преимущества REST:

- надёжность (не нужно сохранять информацию о состоянии клиента, которая может быть утеряна);
- производительность (за счёт использования кэша);
- масштабируемость;
- прозрачность системы взаимодействия;
- простота интерфейсов;
- портативность компонентов;
- лёгкость внесения изменений;
- способность эволюционировать, приспосабливаясь к новым требованиям.

Что вы знаете о принципах Restfull?

• Приведение архитектуры к модели клиент-сервер

В основе данного ограничения лежит разграничение потребностей. Необходимо отделять потребности клиентского интерфейса от потребностей сервера, хранящего данные. Данное ограничение повышает переносимость клиентского кода на другие платформы, а упрощение серверной части улучшает масштабируемость системы. Само разграничение на "клиент" и "сервер" позволяет им развиваться независимо друг от друга.

• Отсутствие состояния

Архитектура REST требует соблюдения следующего условия. В период между запросами серверу не нужно хранить информацию о состоянии клиента и наоборот. Все запросы от клиента должны быть составлены так, чтобы сервер получил всю необходимую информацию для выполнения запроса. Таким образом и сервер, и клиент могут "понимать" любое принятое сообщение, не опираясь при этом на предыдущие сообщения.

• Кэширование

Клиенты могут выполнять кэширование ответов сервера. У тех, в свою очередь, должно быть явное или неявное обозначение как кэшируемых или некаэшируемых, чтобы клиенты в ответ на последующие запросы не получали устаревшие или неверные данные.

Правильное использование кэширования помогает полностью или частично устранить некоторые клиент-серверные взаимодействия, ещё больше повышая производительность и расширяемость системы.

• Единообразие интерфейса

К фундаментальным требованиям REST архитектуры относится и унифицированный, единообразный интерфейс. Клиент должен всегда понимать, в каком формате и на какие адреса ему нужно слать запрос, а сервер, в свою очередь, также должен понимать, в каком формате ему следует отвечать на запросы клиента. Этот единый формат клиент-серверного взаимодействия, который описывает, что, куда, в каком виде и как отсылать и является унифицированным интерфейсом.

• Слои

Под слоями подразумевается иерархическая структура сетей. Иногда клиент может общаться напрямую с сервером, а иногда - просто с промежуточным узлом. Применение промежуточных серверов способно повысить масштабируемость за счёт балансировки нагрузки и распределённого кэширования.

Пример: представим себе некоторое мобильное приложение, которое пользуется популярностью во всем мире. Его неотъемлемая часть - загрузка картинок. Так как пользователей - миллионы человек, один сервер не смог бы выдержать такой большой нагрузки.

Разграничение системы на слои решит эту проблему. Клиент запросит картинку у промежуточного узла, промежуточный узел запросит картинку у сервера, который наименее загружен в данный момент, и вернет картинку клиенту. Если здесь на каждом уровне иерархии правильно применить кэширование, то можно добиться хорошей масштабируемости системы.

• Код по требованию (необязательное ограничение)

Данное ограничение подразумевает, что клиент может расширять свою функциональность, за счет загрузки кода с сервера в виде апплетов или сценариев.

Какие методы запросов вам знакомы?

- GET запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.
- HEAD запрашивает ресурс так же, как и метод GET, но без тела ответа.
- POST используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.
- PUT заменяет все текущие представления ресурса данными запроса.
- DELETE удаляет указанный ресурс.
- CONNECT устанавливает "туннель" к серверу, определённому по ресурсу.
- OPTIONS используется для описания параметров соединения с ресурсом.
- TRACE выполняет вызов возвращаемого тестового сообщения с ресурса.
- PATCH используется для частичного изменения ресурса.

Какова структура REST-запроса?

- URI - символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д. Прежде всего, речь идёт о ресурсах сети Интернет и Всемирной паутины. URI предоставляет простой и расширяемый способ идентификации ресурсов
- Параметры (если необходимо)
- Метод
- Заголовки
- Тело (если необходимо)

Какова структура REST-ответа?

- URI
- Код ответа:
 - 100 - 199 Информационные
 - 200 - 299 Успешные
 - 300 - 399 Перенаправления
 - 400 - 499 Клиентские ошибки
 - 500 - 599 Серверные ошибки
- Заголовки
- Тело (если есть)



Информация по кодам ответа: <https://developer.mozilla.org/ru/docs/Web/HTTP/Status>

Может ли быть GET-запрос с телом?

Может, но так делать не стоит:

A payload within a GET request message has no defined semantics; sending a payload body on a GET request might cause some existing implementations to reject the request.

Т.е. полезная нагрузка в сообщении запроса GET не имеет определенной семантики; отправка тела полезной нагрузки в запросе GET может привести к тому, что некоторые существующие реализации отклонят запрос. Если обратиться к более старым спецификациям, то там вообще написано, что надо игнорировать тело при GET.

Может ли быть POST-запрос без тела?

Может. Данное поведение считается штатным для данного метода.

Если использовать POST без тела, это похоже на использование функции, которая не принимает аргумент; поэтому разумно иметь функцию для вашего класса ресурсов, которая может изменять состояние объекта без аргумента.

Что такое SOAP? В чем его основные особенности?

SOAP (Simple Object Access Protocol) является протоколом и имеет спецификацию, характеризуется использованием HTTP(S)-протокола лишь как транспорта (чаще всего, методом POST). Все детали сообщений (в обе стороны – от клиента к серверу и обратно) передаются в стандартизованном XML-документе. SOAP может работать и с другими протоколами прикладного уровня (SMTP, FTP), но чаще всего он применяется поверх HTTP(S).

Любое сообщение в протоколе SOAP - это XML документ, состоящий из следующих элементов (тегов):

- Envelope. Корневой обязательный элемент. Определяет начало и окончание сообщения.
- Header. Необязательный элемент - заголовок. Содержит элементы, необходимые для обработки самого сообщения. Например, идентификатор сессии.
- Body. Основной элемент, содержит основную информацию сообщения. Обязательный.

- **Fault.** Элемент, содержащий информацию об ошибках, возникающих в процессе обработки сообщения. Необязательный.

SOAP использует WSDL (Web Services Description Language) - язык описания веб-сервисов и доступа к ним, основанный на языке XML.

Каждый документ WSDL 1.1 можно разбить на следующие логические части:

- определение типов данных (types) - определение вида отправляемых и получаемых сервисом XML-сообщений
- элементы данных (message) - сообщения, используемые web-сервисом
- абстрактные операции (portType) - список операций, которые могут быть выполнены с сообщениями
- связывание сервисов (binding) - способ, которым сообщение будет доставлено

SOAP не накладывает никаких ограничений на тип транспортного протокола. Вы можете использовать либо Web протокол HTTP, либо MQ.

XSD - это язык описания структуры XML документа. Его также называют XML Schema. При использовании XML Schema XML парсер может проверить не только правильность синтаксиса XML документа, но также его структуру, модель содержания и типы данных.

Такой подход позволяет объектно-ориентированным языкам программирования легко создавать объекты в памяти, что, несомненно, удобнее, чем разбирать XML как обычный текстовый файл. Кроме того, XSD расширяем, и позволяет подключать уже готовые словари для описания типовых задач, например веб-сервисов, таких как SOAP. В XSD есть встроенные средства документирования, что позволяет создавать самодостаточные документы, не требующие дополнительного описания.

Преимущества SOAP:

- отраслевой стандарт по версии W3C;
- наличие строгой спецификации;
- широкая поддержка в продуктах Microsoft,
- однозначность.

Недостатки SOAP:

- сложность реализации;
- сложность/ресурсоемкость парсинга XML-данных.

Stateless и Stateful - о чем говорят эти два понятия в контексте API?

- **Stateful** может хранить какое-либо состояние между запросами.
- **Stateless** никакого состояния не хранит, а следовательно может использоваться один и тот же экземпляр.

Что можно протестировать на уровне API и нельзя на уровне GUI?

Например, передачу некоторых данных либо параметров в промежуточные системы.

Яркий пример на проекте: пасскод и передача динамических полей по API. На GUI можно увидеть лишь в Альфа-Мобайл.

Инструменты для API тестирования

- Postman
- Jmeter
- SOAP UI



Обзор: <https://habr.com/ru/post/418313/>

Работа с БД


Типы БД

1. **Простейшие типы баз данных**
 - a. Простые структуры данных
 - b. Иерархические базы данных
 - c. Сетевые базы данных
2. **Реляционные БД**
 - a. SQL базы данных
3. **NoSQL базы данных**
 - a. Базы данных «ключ-значение»

- b. Документная база данных
- c. Графовая база данных
- d. Колоночные базы данных
- e. Базы данных временных рядов

4. Комбинированные типы

- a. NewSQL базы данных
- b. Многомодельные базы данных

 Подробнее: <https://proglib.io/p/11-tipov-sovremennyh-baz-dannyh-kratkie-opisaniya-shemy-i-primery-bd-2020-01-07>

Три типа нормализации реляционной БД

Нормализация - это процесс организации данных в базе данных. Это включает создание таблиц и установление связей между этими таблицами в соответствии с правилами, предназначенными как для защиты данных, так и для того, чтобы сделать базу данных более гибкой за счет устранения избыточности и непоследовательной зависимости.

Первая нормальная форма

- Исключить повторяющиеся группы в отдельных таблицах.
- Создайте отдельную таблицу для каждого набора связанных данных.
- Определите каждый набор связанных данных с помощью основного ключа.

Плохой пример

Производитель конфет	Вид конфет
ООО "Спартак"	Грильяж, Трюфель, Леденец

Пример нормализации

Производитель конфет	Вид конфет
ООО "Спартак"	Грильяж
ООО "Спартак"	Трюфель

Вторая нормальная форма

- Создайте отдельные таблицы для наборов значений, применимых к нескольким записям.
- Соотносим эти таблицы с иностранным ключом.

Плохой пример

Производитель конфет	Вид конфет
ООО "Спартак"	Грильяж

Пример нормализации

id	Производитель конфет	Вид конфет
000-989-242	ООО "Спартак"	Грильяж
000-989-243	ООО "Спартак"	Трюфель

Третья нормальная форма

- Устранение полей, которые не зависят от ключа.

Плохой пример

id	Производитель конфет	Вид конфет
000-989-242	ООО "Спартак"	Грильяж
000-989-243	ООО "Спартак"	Трюфель

Пример нормализации

id	Производитель конфет	Код конфет
000-989-242	ООО "Спартак"	245-09
000-989-243	ООО "Спартак"	246-01

id	Вид конфет
245-09	Грильяж
246-09	Трюфель

Запросы в БД

создание бд

```
CREATE DATABASE databasename;
```

создание таблицы

```
CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    PRIMARY KEY (ID)  
);
```

удаление таблицы

```
DROP TABLE table_name;
```

обновление таблицы

```
ALTER TABLE table_name  
ADD column_name datatype;
```

добавление записи в таблицу

```
INSERT INTO table_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);
```

обновление записи в таблице

```
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;
```

удаление записи из таблицы

```
DELETE FROM table_name WHERE condition;
```

получение данных из таблицы

```
SELECT column1, column2, ...  
FROM table_name;
```

копирование данных из одной таблицы в другую и добавление данных в таблицу

```
INSERT INTO table2  
SELECT * FROM table1  
WHERE condition;
```

копирование данных из одной таблицы в другую

```
SELECT column1, column2, column3, ...  
INTO newtable [IN externaldb]  
FROM oldtable  
WHERE condition;
```

получение уникальных данных из таблицы

```
SELECT DISTINCT column1, column2, ...  
FROM table_name;
```

Разница LEFT, RIGHT, OUTER, INNER JOIN

Ключевое слово **LEFT JOIN** возвращает все записи из левой таблицы (table1) и соответствующие записи из правой таблицы (table2).

```
SELECT column_name(s)  
FROM table1  
LEFT JOIN table2  
ON table1.column_name = table2.column_name;
```

Ключевое слово **RIGHT JOIN** возвращает все записи из правой таблицы (table2) и соответствующие записи из левой таблицы (table1).

```
SELECT column_name(s)  
FROM table1  
RIGHT JOIN table2  
ON table1.column_name = table2.column_name;
```

Ключевое слово **INNER JOIN** выбирает записи, которые имеют совпадающие значения в обеих таблицах.

```
SELECT column_name(s)  
FROM table1  
INNER JOIN table2  
ON table1.column_name = table2.column_name;
```

Ключевое слово **FULL OUTER JOIN** возвращает все записи, если есть совпадения в левой (таблица1) или правой (таблица2) записях таблицы.

```
SELECT column_name(s)  
FROM table1  
FULL OUTER JOIN table2  
ON table1.column_name = table2.column_name  
WHERE condition;
```

Оператор SELF JOIN

Это выражение используется для того, чтобы таблица объединилась сама с собой, словно это две разные таблицы. Чтобы такое реализовать, одна из таких «таблиц» временно переименовывается.

Например, следующий SQL-запрос объединяет клиентов из одного города:

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

Оператор UNION

Он используется для объединения полученных данных из двух или более запросов, которые должны иметь одинаковое количество столбцов с одинаковыми типами данных и расположенных в том же порядке.

Пример использования:

```
SELECT column(s) FROM first_table
UNION
SELECT column(s) FROM second_table;
```

Подстановочные знаки

Это специальные символы, которые нужны для замены каких-либо знаков в запросе. Они используются вместе с оператором LIKE, с помощью которого можно отфильтровать запрашиваемые данные.

% - заменить ноль или более символов;

_ - заменить один символ.

Примеры:

Данный запрос позволяет найти данные всех пользователей, имена которых содержат в себе «test».

```
SELECT * FROM user WHERE name LIKE '%test%';
```

А в этом случае имена искомых пользователей начинаются на «t», после содержат какой-либо символ и «est» в конце.

```
SELECT * FROM user WHERE name LIKE 't_est';
```

Вложенные запросы

Вложенный запрос – это запрос, который находится внутри другого SQL запроса и встроен внутри условного оператора WHERE. Данный вид запросов используется для возвращения данных, которые будут использоваться в основном запросе, как условие для ограничения получаемых данных.

Вложенные запросы должны следовать следующим правилам:

- Вложенный запрос должен быть заключён в родительский запрос.
- Вложенный запрос может содержать только одну колонку в операторе SELECT.
- Оператор ORDER BY не может быть использован во вложенном запросе. Для обеспечения функционала ORDER BY, во вложенном запросе может быть использован GROUP BY.
- Вложенные запросы, возвращающие более одной записи могут использоваться с операторами нескольких значений, как оператор IN.
- Вложенный запрос не может заканчиваться в функции.
- SELECT не может включать никаких ссылок на значения BLOB, ARRAY, CLOB и NCLOB.
- Оператор BETWEEN не может быть использован вместе с вложенным запросом.

```
SELECT * FROM developers
WHERE ID IN (SELECT ID
            FROM developers
            WHERE SALARY > 2000);
```

Автоматизированное тестирование

Java

Принципы ООП

- **Абстракция** - отделение концепции от ее экземпляра.
- **Полиморфизм** - реализация задач одной и той же идеи разными способами.
- **Наследование** - способность объекта или класса базироваться на другом объекте или классе. Это главный механизм для повторного использования кода. Наследственное отношение классов четко определяет их иерархию.
- **Инкапсуляция** - размещение одного объекта или класса внутри другого для разграничения доступа к ним.

Модификаторы доступа

- **public** – полный доступ к сущности (полю или методу класса) из любого пакета;
- **protected** – доступ к сущности только для классов своего пакета и наследников класса;
- **default** – **неявный модификатор по умолчанию (при отсутствии трёх явных)** – доступ к сущности только для классов своего пакета;
- **private** – доступ только внутри класса, в котором объявлена сущность.

Перегрузка и переопределение

Перегрузка методов позволяет использовать одно и то же имя для нескольких методов с разными сигнатурами.

Для перегрузки методов необходимо определить методы с одним и тем же именем, которые отличаются:

- количеством параметров
- типом параметров
- последовательностью типов параметров

В процессе компиляции определяется, какой метод следует вызвать, основываясь на количестве, типе и последовательности передаваемых параметров.

Имена параметров, тип возвращаемого значения и модификатор доступа метода в выборе требуемого метода не участвуют.

```
public double summa(double x1, double x2) {
    return x1 + x2;
}

public double summa(double x1, double x2, double x3) {
    return x1 + x2 + x3;
}

public double summa(double x1, double x2, double x3, double x4) {
    return x1 + x2 + x3 + x4;
}
```

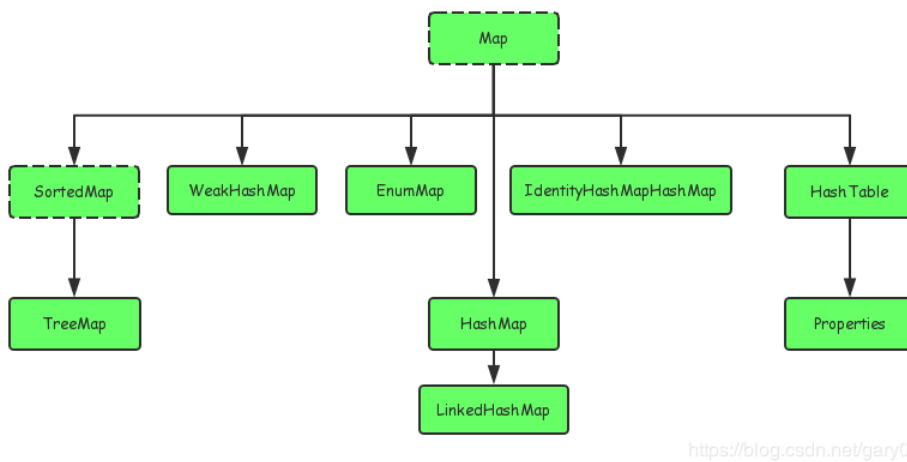
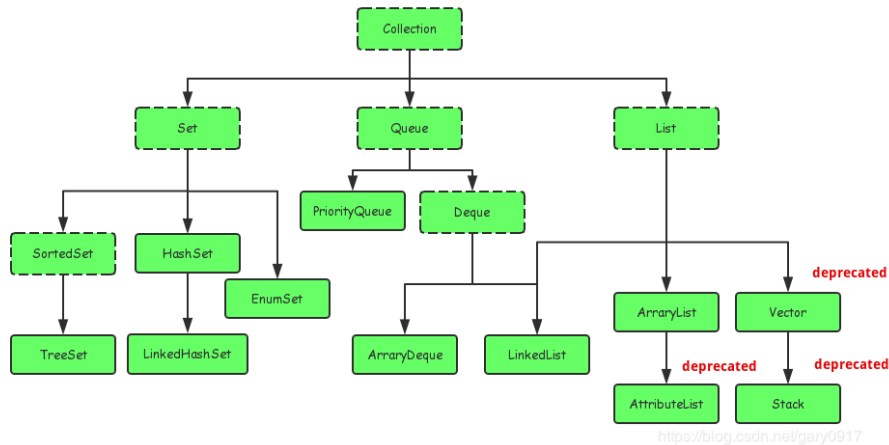
Переопределение – это изменение реализации уже существующего в суперклассе метода (override). В новом методе должны быть те же сигнатура и тип возвращаемого результата, что и у метода родительского класса.

```
public class Robot
{
    private String name;

    public Robot(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "name=" + name;
    }
}
```

Коллекции



Интерфейсы:

- **Collection:** базовый интерфейс для всех коллекций и других интерфейсов коллекций.
- **List:** наследует интерфейс Collection и представляет функциональность простых списков, в котором у каждого элемента есть индекс. Дубликаты значений допускаются.
- **Queue:** наследует интерфейс Collection и представляет функционал для структур данных в виде очереди. В таком списке элементы можно добавлять только в хвост, а удалять - только из начала. Так реализуется концепция **FIFO (first in, first out)** - «первым пришёл - первым ушёл». Пример из жизни: очередь в супермаркете. Ещё есть **LIFO (last in, first out)**, то есть «последним пришёл - первым ушёл». Пример из жизни: стопка буклетов на ресепшен в отеле.
- **Set:** также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов.
- **SortedSet:** расширяет интерфейс Set для создания сортированных коллекций.
- **Map:** предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. Ключи уникальны, а значения могут повторяться. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection.

List

ArrayList

Класс ArrayList поддерживает динамические массивы, которые могут расти по мере необходимости. Элементы ArrayList могут быть абсолютно любых типов, в том числе и null. Элементы могут повторяться.

Класс ArrayList реализует интерфейс List.

Объект класса ArrayList, содержит свойства `elementData` и `size`. Хранилище значений `elementData` есть не что иное, как массив определенного типа (указанного в generic).

Минимальная длина массива = 10, увеличивается по формуле (текущий размер * 2) / 3 (в 1,5 раза).

Достоинства класса ArrayList:

- быстрый доступ по индексу
- быстрая вставка и удаление элементов с конца

Недостатки класса ArrayList:

- медленная вставка и удаление элементов в середину

LinkedList

LinkedList - класс, реализующий два интерфейса - List и Deque. Это обеспечивает возможность создания двунаправленной очереди из любых (в том числе и null) элементов. Каждый объект, помещенный в связанный список, является узлом (нодом). Каждый узел содержит элемент, ссылку на предыдущий и следующий узел. Фактически связанный список состоит из последовательности узлов, каждый из которых предназначен для хранения объекта определенного при создании типа.

Особенности:

- может содержать элементы любого типа;
- позволяет хранить повторяющиеся значения;
- сохраняет порядок размещения;
- быстрее, чем массивы, поскольку при вставке не происходит смещения элементов
- операции, индексирующие список, будут проходить по списку от начала или до конца, в зависимости от того, что ближе к указанному индексу
- не синхронизирован
- Iterator и ListIterator предрасположены к сбою (это означает, что после создания итератора, если список будет изменен, будет выдано исключение ConcurrentModificationException)
- каждый элемент является узлом, который хранит ссылку на следующий и предыдущий

Сравнение производительности между ArrayList и LinkedList

ArrayList - это очередь массива, эквивалентная динамическому массиву. Это реализуется массивом, с высокой эффективностью произвольного доступа, низкой случайной вставкой и эффективностью случайного удаления. ArrayList должен использовать произвольный доступ (т. Е. Доступ по номеру индекса) для обхода элементов коллекции.

LinkedList - это дважды связанный список. Он также может работать как стек, очередь или очередь. Эффективность произвольного доступа в LinkedList низкая, но эффективность произвольной вставки и случайного удаления высока. LinkedList должен пройти элементы коллекции один за другим.

СРАВНЕНИЕ:

• Структура

ArrayList – это структура данных на основе индекса, поддерживаемая массивом. Он обеспечивает произвольный доступ к своим элементам с производительностью, равной $O(1)$.

С другой стороны, LinkedList хранит свои данные в виде списка элементов, и каждый элемент связан со своим предыдущим и следующим элементом. В этом случае операция поиска элемента имеет время выполнения, равное $O(n)$.

• Операции

Операции вставки, добавления и удаления элемента выполняются быстрее в LinkedList, поскольку нет необходимости изменять размер массива или обновлять индекс при добавлении элемента в произвольную позицию внутри коллекции, изменяются только ссылки в окружающих элементах.

• Использование памяти

Связанный список потребляет больше памяти, чем ArrayList, потому что каждый узел в Связанном списке хранит две ссылки, одну для своего предыдущего элемента и одну для следующего элемента, в то время как ArrayList содержит только данные и его индекс.

• Применение

Вот несколько примеров кода, которые показывают, как вы можете использовать LinkedList:

1. Создание

```
LinkedList linkedList = new LinkedList<>();
```
2. Добавление элемента
 LinkedList реализует List и Deque интерфейс, помимо стандартных add() и addAll() методов, которые вы можете найти addFirst() и addLast(), которые добавляют элемент в начале или в конце соответственно.
3. Удаление элемента
 Аналогично добавлению элементов, реализация этого списка предлагает removeFirst() и removeLast(). Кроме того, существует удобный метод removeFirstOccurrence() и removeLastOccurrence(), который возвращает логическое значение (true, если коллекция содержала указанный элемент).

4. Операции с очередями

Deque интерфейс обеспечивает поведение, подобное очереди (на самом деле Deque расширяет Очередь интерфейс):

- `linkedList.poll();`
- `linkedList.pop();`

Эти методы извлекают первый элемент и удаляют его из списка.

Разница между `poll()` и `pop()` заключается в том, что `pop` вызовет `NoSuchElementException()` в пустом списке, тогда как `poll` возвращает `null`. Также доступны API `pollFirst()` и `pollLast()`.

Вот, например, как работает API `push`:

```
linkedList.push(Object o);
```

вставляет элемент в качестве главы коллекции.

Заключение

- (01) для быстрой вставки и удаления элементов следует использовать `LinkedList`;
- (02) для элементов, требующих быстрого произвольного доступа, следует использовать `ArrayList`;
- (03) для «однопоточной среды» или «многопоточной среды, но `List` будет работать только одним потоком», вы должны использовать несинхронные классы (такие как `ArrayList`). Для «многопоточной среды, и `List` может работать несколькими потоками одновременно», вы должны использовать синхронизированные классы (такие как `Vector`).

Queue

Deque наследует интерфейс `Queue` и представляет функционал для двунаправленных очередей. Это значит, что элементы можно добавлять как в её начало, так и в конец. То же относится к удалению элементов из очереди.

PriorityQueue – это класс очереди с приоритетами. По умолчанию очередь с приоритетами размещает элементы согласно естественному порядку сортировки используя `Comparable`. Элементу с наименьшим значением присваивается наибольший приоритет. Если несколько элементов имеют одинаковый наивысший элемент – связь определяется произвольно. Также можно указать специальный порядок размещения, используя `Comparator`.

ArrayDeque создает двунаправленную очередь



Подробнее: <https://www.examclouds.com/ru/java/java-core-russian/interface-queue>

Set

Интерфейс SortedSet описывает упорядоченное множество, отсортированное в возрастающем порядке или по порядку, заданному реализацией интерфейса `Comparator`.

Методы интерфейса `SortedSet`:

- `Comparator<? super E> comparator()` - возвращает компаратор отсортированного множества. Если для множества применяется естественный порядок сортировки, возвращается `null`.
- `E first()` - возвращает первый элемент вызывающего отсортированного множества.
- `E last()` - возвращает последний элемент вызывающего отсортированного множества.
- `SortedSet headSet(E toElement)` - возвращает `SortedSet`, содержащий элементы из вызывающего множества, которые предшествуют `end`.
- `SortedSet subSet(E fromElement, E toElement)` - возвращает `SortedSet`, содержащий элементы из вызывающего множества, находящиеся между `start` и `end-1`.
- `SortedSet tailSet(E fromElement)` - возвращает `SortedSet`, содержащий элементы из вызывающего множества, которые следуют за `end`.

Интерфейс NavigableSet расширяет `SortedSet` и добавляет методы для более удобного поиска по коллекции:

- `E ceiling(E obj)` - ищет в наборе наименьший элемент `e`, для которого истинно `e >= obj`. Если такой элемент найден, он возвращается. В противном случае возвращается `null`.
- `E floor(E obj)` - ищет в наборе наибольший элемент `e`, для которого истинно `e <= obj`. Если такой элемент найден, он возвращается. В противном случае возвращается `null`.
- `E higher(E obj)` - ищет в наборе наибольший элемент `e`, для которого истинно `e > obj`. Если такой элемент найден, он возвращается. В противном случае возвращается `null`.
- `E lower(E obj)` - ищет в наборе наименьший элемент `e`, для которого истинно `e < obj`. Если такой элемент найден, он возвращается. В противном случае возвращается `null`.
- `NavigableSet headSet(E upperBound, boolean incl)` - возвращает `NavigableSet`, включающий все элементы вызывающего набора, меньшие `upperBound`. Результирующий набор поддерживается вызывающим набором.
- `NavigableSet subSet(E lowerBound, boolean lowIncl, E upperBound, boolean highIncl)` - возвращает `NavigableSet`, включающий все элементы вызывающего набора, которые больше `lowerBound` и меньше `upperBound`. Если `lowIncl` равно `true`, то элемент, равный `lowerBound`, включается. Если `highIncl` равно `true`, также включается элемент, равный `upperBound`.
- `E pollLast()` - возвращает последний элемент, удаляя его в процессе. Поскольку набор отсортирован, это будет элемент с наибольшим значением. Возвращает `null` в случае пустого набора.

- `pollFirst()` - возвращает первый элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наименьшим значением. Возвращает `null` в случае пустого набора.
- `Iterator descendingIterator()` - возвращает итератор, перемещающийся от большего к меньшему, другими словами, обратный итератор.
- `NavigableSet descendingSet()` - возвращает `NavigableSet`, представляющий собой обратную версию вызывающего набора. Результирующий набор поддерживается вызывающим набором.

TreeSet реализует интерфейс `NavigableSet`, который поддерживает элементы в отсортированном по возрастанию порядке. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

HashSet реализует интерфейс `Set` и создает коллекцию, которая используется для хранения хеш-таблиц.

Элементы хеш-таблицы хранятся в виде пар ключ-значение. Ключ определяет ячейку для хранения значения. Содержимое ключа служит для определения однозначного значения, называемого хеш-кодом.

Мы можем считать, что хеш-код это ID объекта, хотя он не должен быть уникальным. Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом.

Правила написания методов `hashCode()` и `equals()`:

- для одного и того же объекта, хеш-код всегда будет одинаковым;
- если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот);
- если хеш-коды равны, то входные объекты не всегда равны;
- если хеш-коды разные, то и объекты гарантированно будут разные.

Выгода от хеширования состоит в том, что оно обеспечивает постоянство время выполнения операций `add()`, `contains()`, `remove()` и `size()`, даже для больших наборов.

Класс `HashSet` не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств.

LinkedHashSet расширяет `HashSet`, не добавляя никаких новых методов. `LinkedHashSet` поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.

Работает дольше чем класс `HashSet`.

EnumSet - это специализированная коллекция `Set` для работы с классами перечисления. Реализует интерфейс `Set`.

Особенности:

- может содержать только `enum`-значения, и все значения должны принадлежать одному и тому же `enum`-классу;
- не позволяет добавлять `null`, иначе исключение `NullPointerException` при попытке сделать это;
- не является потокобезопасным, поэтому при необходимости нам нужно синхронизировать его внешне;
- элементы хранятся в том порядке, в котором они объявлены в `enum`;
- использует отказоустойчивый итератор, который работает с копией, поэтому он не будет генерировать исключение `ConcurrentModificationException`, если коллекция будет изменена при итерации по ней.

Map

Интерфейс SortedMap гарантирует, что элементы размещаются в возрастающем порядке значений ключей.

Методы интерфейса SortedMap:

- `Comparator<? super K> comparator()` - возвращает компаратор вызывающей сортированной карты. Если картой используется естественный порядок, возвращается `null`.
- `K firstKey()` - возвращает первый ключ вызывающей карты.
- `K lastKey()` - возвращает последний ключ вызывающей карты.
- `SortedMap<K, V> headMap(K end)` - Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше `end`.
- `SortedMap<K, V> subMap(K start, K end)` - возвращает карту, содержащую элементы вызывающей карты, чей ключ больше или равен `start` и меньше `end`.
- `SortedMap<K, V> tailMap(K start)` - возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых больше `start`.

Интерфейс NavigableMap расширяет `SortedMap` и определяет поведение карты, поддерживающей извлечение элементов на основе ближайшего соответствия заданному ключу или ключам.

Интерфейс Map.Entry позволяет работать с элементами карты. Метод `entrySet()`, объявленный в интерфейсе `Map`, возвращает `Set`, содержащий элементы карты. Каждый из элементов этого набора представляет собой объект типа `Map.Entry`.

TreeMap хранит элементы в порядке сортировки. `TreeMap` сортирует элементы по возрастанию от первого к последнему. Порядок сортировки может задаваться реализацией интерфейсов `Comparator` и `Comparable`. Реализация `Comparator` передается в конструктор `TreeMap`, `Comparable` используется при добавлении элемента в карту.

HashMap реализует интерфейс Map. Он использует хеш-таблицу для хранения карты. Это позволяет обеспечить константное время выполнения методов get() и put() даже при больших наборах. Ключи и значения могут быть любых типов, в том числе и null.

LinkedHashMap расширяет HashMap. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать итерацию по карте в порядке вставки.

EnumMap реализует интерфейс Map для перечисления типов данных.

Особенности EnumMap:

- упорядоченный набор элементов типа данных enum;
- не синхронизирована;
- производительность выше, чем у HashMap;
- все ключи являются экземплярами элементов в Enum;
- нельзя хранить null ключи;
- EnumMap хранит данные внутри в виде массивов.

Hashtable реализует интерфейс Map, которая хранит пары ключ-значение. Hashtable является синхронизированной и потокобезопасной коллекцией. Hashtable не допускается null-ключей и дублирующих ключей, а также null-значений.

Сравнение Hashtable и HashMap

Характеристика	Hashtable	HashMap
Потокобезопасность	+	-
Синхронизированность	+	-
Производительность	Потокобезопасность и синхронизированность делает ее медленной	В однопоточной среде намного быстрее, чем Hashtable. Если вы не работаете с многопоточностью, то рекомендуется HashMap
Ключ == null?	Нельзя	Позволяет null ключ и null значение
Альтернатива	Нет альтернативы	Можно использовать ConcurrentHashMap для работы в многопоточной среде

WeakHashMap, фактически, хранит не пары "ключ - значение", а пары "слабая ссылка на ключ - значение". Особенность слабых ссылок (WeakReference) состоит в том, что они игнорируются сборщиком мусора, т.е. если на объект-ключ нет других ссылок, он уничтожается.

Перед любым обращением к WeakHashMap (get(), put(), size() и т.д.) анализируются невалидные ссылки и соответствующая пара удаляется.



Подробнее про Map: <https://www.examclouds.com/ru/java/java-core-russian/map>

Работа с коллекциями. Работа со stream

Начиная с JDK 8 в Java появился новый API - Stream API.

Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете java.util.stream.

Одной из отличительных черт Stream API является применение лямбда-выражений, которые позволяют значительно сократить запись выполняемых действий.

При работе со Stream API важно понимать, что все операции с потоками бывают либо **терминальными (terminal)**, либо **промежуточными (intermediate)**

- промежуточные операции модифицируют стрим. На одном потоке можно вызвать сколько угодно промежуточных операций;
- терминальная операция «потребляет» поток. Она может быть только одна, в конце работы с отдельно взятым стримом. Стримы работают лениво – вся цепочка промежуточных операций не начнет выполняться до вызова терминальной.

Преимущества

- стримы избавляют от написания стереотипного кода всякий раз, когда нужно сделать что-то с набором элементов. То есть благодаря стримам не приходится думать о деталях реализации;
- стримы поддерживают один из основных принципов хорошего проектирования - слабую связанность (low coupling). Чем меньше класс знает про другие классы - тем лучше. Алгоритму сортировки не должно быть важно, что конкретно он сортирует. Это и делают стримы;
- с помощью стримов операции с коллекциями проще распараллелить: в императивном подходе для этого бы понадобился минимум ещё один цикл;
- стримы позволяют уменьшить число побочных эффектов: методы Stream API не меняют исходные коллекции;
- со Stream API лаконично записываются сложные алгоритмы обработки данных.

Отличие коллекций от потоков:

- потоки не хранят элементы. Элементы, используемые в потоках, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
- операции с потоками не изменяют источника данных. Операции с потоками лишь возвращают новый поток с результатами этих операций.
- для потоков характерно отложенное выполнение. То есть выполнение всех операций с потоком происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый поток.

Лямбда - выражение в программировании - специальный синтаксис для определения функциональных объектов, заимствованный из λ-исчисления. То есть, используя функциональные объекты, можно объявлять функции в любом месте кода. Stream API в Java8 используется для работы с коллекциями, позволяя писать код в функциональном стиле.

Использование λ-выражения в Java дает возможность внедрять функциональное программирование в парадигме объектно-ориентированного. Лямбда выражения позволяют писать быстрее и делают код более ясным.

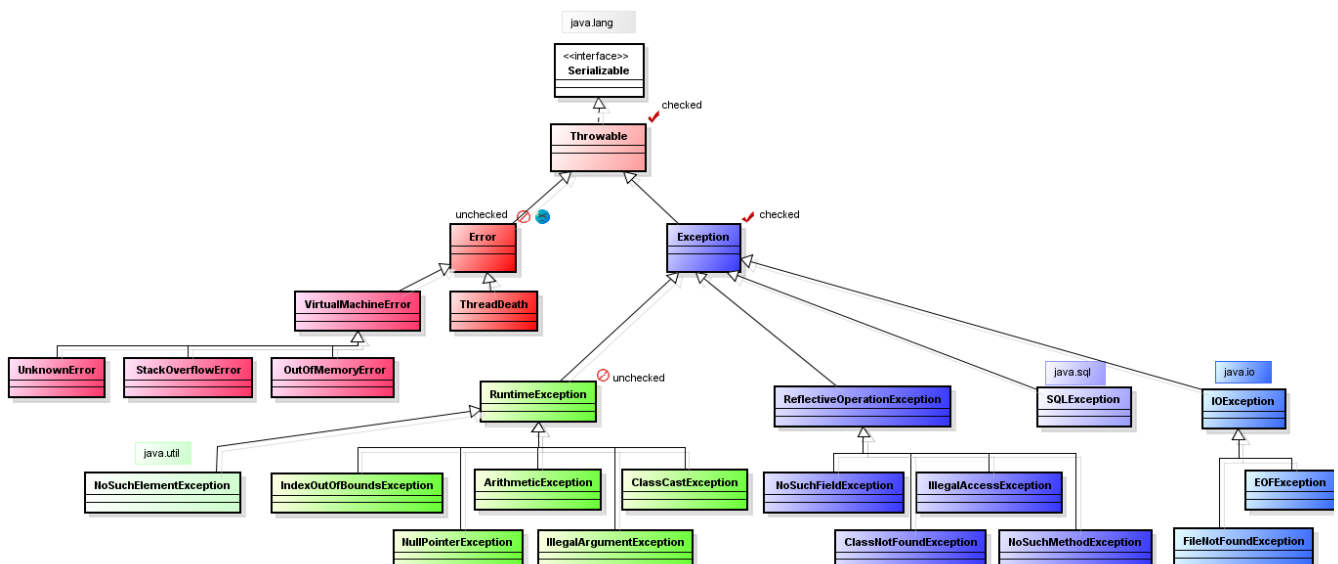
Особенности:

- лямбда-выражение является блоком кода с параметрами;
- используйте лямбда-выражение, когда хотите выполнить блок кода в более поздний момент времени;
- лямбда-выражения могут быть преобразованы в функциональные интерфейсы;
- лямбда-выражения имеют доступ к final переменным из охватывающей области видимости;
- ссылки на метод и конструктор ссылаются на методы или конструкторы без их вызова;
- теперь вы можете добавлять методы по умолчанию и статические методы к интерфейсам, которые обеспечивают конкретные реализации;
- вы должны разрешать любые конфликты между методами по умолчанию из нескольких интерфейсов;



Шпаргалка по работе со stream: <https://habr.com/ru/company/luxoft/blog/270383/>

Виды исключений



Исключение – это проблема(ошибка) возникающая во время выполнения программы.

Исключения могут возникать во многих случаях, например:

- пользователь ввел некорректные данные
- файл, к которому обращается программа, не найден
- сетевое соединение с сервером было утеряно во время передачи данных
- все исключения в Java являются объектами. Поэтому они могут порождаться не только автоматически при возникновении исключительной ситуации, но и создаваться самим разработчиком.

Исключения делятся на несколько классов, но все они имеют общего предка - класс `Throwable`. Его потомками являются подклассы `Exception` и `Error`.

Исключения (`Exceptions`) являются результатом проблем в программе, которые в принципе решаемые и предсказуемые. Например, произошло деление на ноль в целых числах.

Ошибки (`Errors`) представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине. Программа дополнительную память всё равно не сможет обеспечить для JVM.

В Java все исключения делятся на два типа: контролируемые исключения (`checked`) и неконтролируемые исключения (`unchecked`), к которым относятся ошибки (`Errors`) и исключения времени выполнения (`RuntimeExceptions`, потомок класса `Exception`).

- `Checked` исключения, это те, которые должны обрабатываться блоком `catch` или описываться в сигнатуре метода. `Unchecked` могут не обрабатываться и не быть описанными.
- `Unchecked` исключения в Java – наследованные от `RuntimeException`, `checked` – от `Exception` (не включая `unchecked`).

Можно/нужно ли обрабатывать ошибки `jvm`?

Обрабатывать можно, но делать этого не стоит. Разработчику не предоставлены инструменты для обработки ошибок системы и виртуальной машины.

Обработка исключений

В Java есть пять ключевых слов для работы с исключениями:

- `try` – данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке
- `catch` – ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений
- `finally` – ключевое слово для отметки начала блока кода, который является дополнительным. Этот блок помещается после последнего блока `'catch'`. Управление обычно передаётся в блок `'finally'` в любом случае
- `throw` – служит для генерации исключений
- `throws` – ключевое слово, которое прописывается в сигнатуре метода, и обозначающее что метод потенциально может выбросить исключение с указанным типом.

В чем особенность блока `finally`? Всегда ли он исполняется?

Когда исключение передано, выполнение метода направляется по нелинейному пути. Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе. Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм `finally`.

Ключевое слово `finally` создаёт блок кода, который будет выполнен после завершения блока `try/catch`, но перед кодом, следующим за ним. Блок будет выполнен, независимо от того, передано исключение или нет. Оператор `finally` не обязателен, однако каждый оператор `try` требует наличия либо `catch`, либо `finally`. Код в блоке `finally` будет выполнен всегда.

Может ли не быть ни одного блока `catch` при отлавливании исключений?

Такая запись допустима, если имеется связка `try{} finally {}`. Но смысла в такой записи не так много, всё же лучше иметь блок `catch` в котором будет обрабатываться необходимое исключение.

Могли бы вы придумать ситуацию, когда блок `finally` не будет выполнен?

- бесконечный цикл в блоке `try`
- `System.exit(0)` в блоке `try`
- ошибка JVM

Может ли один блок `catch` отлавливать несколько исключений (с одной и разных веток наследований)?

В Java 7 стала доступна новая конструкция, с помощью которой можно перехватывать несколько исключений одним блоком `catch`:

```
try {
    ...
} catch( IOException | SQLException ex ) {
    logger.log(ex);
    throw ex;
}
```



Больше инфо об исключениях: <https://javastudy.ru/interview/exceptions/>

Final, finally, finalize (добавить)

Применение ключевого слова final (добавить)

Ключевое слово static

Static в Java используется как ключевое слово используется для управления памятью. Его можно применять с переменными, методами, блоками и внутренними классами. В объявлении static относится к классу, а не экземпляру класса.

Модификатор **static** в Java напрямую связан с классом. Если поле статично, значит оно принадлежит классу, если метод статичный - аналогично: он принадлежит классу. Исходя из этого, можно обращаться к статическому методу или полю, используя имя класса. Например, если поле count статично в классе Counter, значит, вы можете обратиться к переменной запросом вида: Counter.count.

Статической может быть:

- переменная (переменная класса);
- метод (метод класса);
- блок;
- внутренний класс.



Подробнее: <https://www.internet-technologies.ru/articles/klyuchevoe-slovo-static-v-java.html>

Работа gradle

Gradle - система автоматизации сборки с открытым исходным кодом.

Список функций, которые предоставляет Gradle:

- **Декларативные сборки и сборка по соглашению.** Gradle доступен с отдельным предметно-ориентированным языком (DSL) на основе языка Groovy. Gradle предоставляет элементы декларативного языка. Эти элементы также обеспечивают поддержку сборки по соглашению для Java, Groovy, OSGI, Web и Scala.
- **Язык программирования на основе зависимостей.** Декларативный язык находится на вершине графа задач общего назначения, который вы можете полностью использовать в своей сборке.
- **Структурирование сборки.** Gradle позволяет применять общие принципы проектирования к вашей сборке. Это даст вам идеальную структуру для сборки, так что вы сможете разработать хорошо структурированную и легко поддерживаемую, понятную сборку.
- **Глубокий API.** Используя этот API, он позволяет отслеживать и настраивать его конфигурацию и поведение при выполнении.
- **Масштабирование Gradle.** Gradle может легко увеличить свою производительность, от простых сборок одного проекта до огромных многопроектных сборок предприятия.
- **Многопроектные сборки.** Gradle поддерживает многопроектные сборки и поддерживает частичные сборки. Если вы строите подпроект, Gradle позаботится о создании всех подпроектов, от которых он зависит.
- **Различные способы управления вашими сборками.** Gradle поддерживает различные стратегии для управления вашими зависимостями.
- **Gradle** - это первый инструмент интеграции сборки. Gradle полностью поддерживается для задач ANT, инфраструктура репозитория Maven и Ivy для публикации и получения зависимостей. Gradle также предоставляет конвертер для превращения Maven pom.xml в скрипт Gradle.
- **Легкость миграции.** Gradle может легко адаптироваться к любой вашей структуре. Поэтому всегда можно разработать свою сборку Gradle в той же ветке, где можно создать живой скрипт.
- **Gradle Wrapper.** Gradle Wrapper позволяет выполнять сборки Gradle на машинах, где Gradle не установлен. Это полезно для непрерывной интеграции серверов.
- **Бесплатный открытый исходный код.** Gradle - это проект с открытым исходным кодом, который распространяется под лицензией Apache Software License (ASL).
- **Groovy** - скрипт сборки Gradle написан на Groovy. Весь дизайн Gradle ориентирован на использование в качестве языка, а не жесткой структуры. А Groovy позволяет вам написать собственный скрипт с некоторыми абстракциями. Весь API Gradle полностью разработан на языке Groovy.

Почему Groovy?

Полный API Gradle разработан с использованием языка Groovy. Это преимущество внутреннего DSL над XML. Gradle - это универсальный инструмент для сборки; основное внимание уделяется Java-проектам. В таких проектах члены команды будут очень хорошо знакомы с Java, и лучше, чтобы сборка была максимально прозрачной для всех членов команды.

Такие языки, как Python, Groovy или Ruby, лучше подходят для сборки фреймворка. Почему Groovy был выбран, так это потому, что он предлагает наибольшую прозрачность для людей, использующих Java. Базовый синтаксис Groovy такой же, как Java. Groovy предлагает гораздо больше.

Инициализация проекта Gradle

Gradle - это про выполнение задач, называемых task. Таски предоставляются различными плагинами (plugins).

Файл [build.gradle](#) - главный файл, в котором описывается то, какие библиотеки и фреймворки использует наш проект, какие плагины нужно подключить к проекту и описывает различные задачи.



Подробнее о написании task в gradle: <https://coderlessons.com/tutorials/raznoe/vyuchit-gradle/gradle-zadachi>

Управление зависимостями

- объявление зависимостей:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

- конфигурации зависимостей:
 - implementation - зависимость доступна другим модулям только во время исполнения + есть ссылки на классы компиляции.
 - testImplementation - зависимость доступна другим модулям только во время исполнения тестов + есть ссылки на классы компиляции.
 - runtimeOnly - зависимость доступна другим модулям только во время исполнения, но нет ссылок на классы компиляции.
 - api - зависимость доступна другим модулям как во время компиляции, так и во время исполнения.
 - compileOnly - зависимость доступна другим модулям только во время компиляции, не доступна в рантайме.
 - compile | testCompile | debugCompile - **deprecated**
- типы зависимостей:
 - внешние (на мавен репозиторий и иные)
 - внутренние (на библиотеки проекта)

Работа с хранилищами

При добавлении внешних зависимостей Gradle ищет их в хранилище. Репозиторий - это набор файлов, упорядоченный по группам, именам и версиям. По умолчанию Gradle не определяет никаких репозиториев. Необходимо определить хотя бы одно хранилище явно.

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

Работа с плагинами

Типы плагинов:

- скриптовые плагины - могут быть загружены из локального файла или удаленного репозитория

```
apply from: 'other.gradle'
```

- бинарные плагины - идентифицируются по id плагина

```
plugins {
    id 'java'
}
```


Сборка проекта

- gradle(w) build - сборка проекта
- gradle(w) clean test - запуск тестов
- gradle(w) task test - запуск задачи на сборку проекта

Автоматизация

Автоматизированное тестирование – отдельный вид тестирования?

Нет, не отдельный.

Автоматизированное тестирование предполагает использование специального программного обеспечения (помимо тестируемого) для контроля выполнения тестов и сравнения ожидаемого фактического результата работы программы. Этот тип тестирования помогает автоматизировать часто повторяющиеся, но необходимые для максимизации тестового покрытия задачи.

Когда требуется вводить автоматизацию тестирования на проект?

Ответить на 6 вопросов:

1. ВАШ ПРОГРАММНЫЙ ПРОДУКТ СТАБИЛЕН?
2. ПОТРЕБУЕТСЯ ЛИ НА ПРОЕКТЕ РЕГУЛЯРНОЕ ВЫПОЛНЕНИЕ ОДИНАКОВЫХ ТЕСТОВ?
3. СКОЛЬКО РАЗ БУДЕТ ЗАПУЩЕН НАБОР АВТОТЕСТОВ?
4. У ВАС В ШТАТЕ ЕСТЬ ПРОФЕССИОНАЛЬНАЯ КОМАНДА АВТОМАТИЗАТОРОВ?
5. В ВАШИ ПЛАНЫ ВХОДИТ ЗАМЕНА ВСЕГО РУЧНОГО ТЕСТИРОВАНИЯ НА АВТОМАТИЗИРОВАННОЕ?
6. ПОЧЕМУ ВЫ ЗАДУМАЛИСЬ ОБ АВТОМАТИЗАЦИИ?

О необходимости автоматизации тестирования говорят такие факторы:

- большое количество ручных тестов и не хватает времени на регулярное проведение полного регресса;
- большой процент пропуска ошибок по вине человеческого фактора;
- большой промежуток времени между внесением ошибки, ее обнаружением и исправлением;
- подготовка к тестированию (настройка конфигурации, генерация тестовых данных) занимает много времени;
- большие команды, в которых нужна уверенность, что новый код не сломает код других разработчиков;
- поддержка старых версий ПО, в которых нужно тестировать новые патчи и сервис-паки.

Что требуется, чтобы начать автоматизацию?

- выбрать стратегию автотестирования
- выбрать инструменты для автотестирования
- подготовить окружение
- подготовить тестовые данные
- определить объем автоматизации
- определить, какие именно кейсы требуют автоматизации
- определить варианты отчетности
- подготовить специалистов

Этапы автоматизации тестирования

В процессе автоматизации выполняются следующие шаги:

1. Выбор тестового инструмента
Выбор подходящего инструмента может оказаться сложной задачей. Следующие критерии помогут выбрать лучший инструмент для ваших требований:
 - a. поддержка окружающей среды;
 - b. легкость использования;
 - c. тестирование базы данных;
 - d. идентификация объекта;
 - e. тестирование изображений;
 - f. тестирование восстановления после ошибок;
 - g. отображение объектов;
 - h. используемый язык сценариев;
 - i. поддержка различных типов тестирования, в том числе функционального, тестового управления, мобильного и т. д.;
 - j. поддержка нескольких фреймворков тестирования;
 - k. легко отлаживать сценарии программного обеспечения автоматизации;
 - l. умение распознавать предметы в любой среде;
 - m. обширные отчеты об испытаниях и их результаты;
 - n. минимизация затрат на обучение выбранным инструментам

2. Определение объема автоматизации

Объем автоматизации - это область тестируемого приложения, которая будет автоматизирована. Его помогают определить следующие пункты:

- a. Функции, важные для бизнеса
- b. Сценарии с большим объемом данных
- c. Общие функции приложений
- d. Техническая осуществимость
- e. Частота повторного использования бизнес-компонентов
- f. Сложность тестовых случаев
- g. Возможность использовать одни и те же тестовые сценарии для кросс-браузерного тестирования

3. Планирование, дизайн и разработка

Предполагает создание стратегии и плана автоматизации, которые содержат следующие детали:

- a. Выбранные инструменты автоматизации
- b. Конструкция каркаса и его особенности
- c. Входящие и выходящие за рамки элементы автоматизации
- d. Подготовка стендов автоматизации
- e. График и временная шкала сценариев и выполнения
- f. Результаты тестирования автоматизации

4. Выполнение теста

На этом этапе выполняются сценарии автоматизации. Сценариям необходимо ввести тестовые данные, прежде чем они будут запущены. После выполнения они предоставляют подробные отчеты об испытаниях. Выполнение может быть выполнено с использованием инструмента автоматизации напрямую или с помощью инструмента управления тестированием, который вызовет инструмент автоматизации.

5. Техническое обслуживание

Этот этап автоматизированного тестирования проводится для проверки того, как работают новые функции, добавленные в программное обеспечение: нормально или нет. Сопровождение в автотестировании выполняется, когда добавляются новые сценарии автоматизации, и их необходимо проверять и поддерживать, чтобы повышать эффективность сценариев автоматизации с каждым последующим циклом выпуска.

Что имеет смысл автоматизировать?

- высокорисковые для бизнеса кейсы
- тестовые сценарии, которые регулярно повторяются (регресс)
- тестовые сценарии, которые очень сложны и утомительны для выполнения вручную (е2е тесты)
- тестовые примеры, отнимающие много времени
- смоук-тестирование

Что не следует автоматизировать?

- новые тестовые примеры, которые не выполнялись вручную хотя бы один раз
- сценарии тестирования, требования к которым часто меняются
- тестовые примеры, которые выполняются на разовой основе
- капчи/баркод
- сложные и многоуровневые интеграции

Виды автотестов

- **автоматизация тестирования кода (Code-driven testing)** – тестирование на уровне программных модулей, классов и библиотек (фактически, автоматические юнит-тесты);
- **автоматизация тестирования графического пользовательского интерфейса (Graphical user interface testing)** – специальная программа (фреймворк автоматизации тестирования) позволяет генерировать пользовательские события – нажатия клавиш, клики мышкой, и отслеживать реакцию программы на эти действия – соответствует ли она спецификации.
- **автоматизация тестирования API (Application Programming Interface)** – программного интерфейса программы. Тестируются интерфейсы, предназначенные для взаимодействия, например, с другими программами или с пользователем. Здесь опять же, как правило, используются специальные фреймворки.

Какие есть типы фреймворков автоматизации?

Фреймворк - программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

- **линейные**

Особенность: второе название "Запись и воспроизведение". Шаги записываются в линейном последовательном порядке (один за другим). Нет необходимости писать код самостоятельно, так как чаще "рекордеры" это делают за тестировщика. Самый примитивный тип тестовых фреймворков.

Плюсы:

- Нет необходимости писать собственный код, поэтому опыт в автоматизации тестирования не требуется.

- Один из самых быстрых способов создания тестовых сценариев, поскольку их можно легко записать за минимальное время.
- Рабочий процесс тестирования легче понять любой стороне, участвующей в тестировании, поскольку сценарии построены последовательно.
- Самый простой способ начать автотестирование, особенно с новым инструментом. Большинство современных инструментов автоматизированного тестирования предоставляют функции записи и воспроизведения, поэтому вам также не нужно будет тщательно планировать с помощью этой структуры.

Минусы:

- Скрипты, разработанные с использованием этой платформы, не подлежат повторному использованию.
- Данные жестко запрограммированы в тестах, что означает, что тестовые примеры не могут быть повторно запущены с несколькими наборами данных.
- Техническое обслуживание считается трудозатратным. Эта модель не отличается особой масштабируемостью.

• **модульные**

Особенность: раздел тестируемого приложения на отдельные модули, блоки, функции или разделы, каждый из которых тестируется изолированно. После разделения приложения на отдельные модули для каждой части создаются тесты, которые затем объединяются в тестовые сюиты в иерархическом порядке.

Плюсы:

- Если в приложение вносятся какие-либо изменения, необходимо будет исправить только модуль и связанный с ним отдельный тест, а это означает, что не придется возиться с остальной частью приложения.
- Создание тестов требует меньше усилий, поскольку тестовые сценарии для разных модулей можно повторно использовать.

Минусы:

- Тестовые данные чаще захаардкожены.
- Требуются знания ЯП и автотестирования.

• **library architecture**

Особенность: основана на модульном типе фреймворков, но имеет некоторые дополнительные преимущества. Вместо того, чтобы разделять тестируемое приложение на различные модули, создаются аналогичные функции (методы), которые затем группируются по классам. Эти функции хранятся в библиотеке, которые при необходимости вызываются в тестах.

Плюсы:

- Масштабируемость.
- Возможность повторного использования методов.

Минусы:

- Тестовые данные чаще захаардкожены.
- Требуются знания ЯП и автотестирования.
- На разработку тестов уходит много времени.

• **data driven**

Особенность: отделение тестовых данных от логики сценария, что означает, что тестирующие могут хранить данные в properties-файлах, Excel, текстовых файлах, файлах CSV, таблицах и т.д. Тесты "подключаются" к внешнему источнику данных, и им предлагается вычитать или заполнить необходимые данные.

Плюсы:

- Тесты можно выполнять с несколькими наборами данных.
- Можно быстро протестировать несколько сценариев.
- Можно избежать жесткого кодирования данных, поэтому любые изменения в тестовых сценариях не влияют на используемые данные и наоборот.
- Экономия временных и человеческих ресурсов.

Минусы:

- Требуются знания ЯП и автотестирования.
- На разработку тестов уходит много времени.
- Настройка фреймворка занимает значительное время.

- **keyword driven**

Особенность: предполагает размещение ключевых слов во внешнем источнике, а также "инструкций" по работе с ними. Файл читается линейно и происходит выполнение сценария теста.

После того, как таблица настроена, все, что нужно сделать тестировщику, - это написать код, который предложит необходимые действия на основе ключевых слов. Когда тест запускается, тестовые данные считываются и указываются на соответствующее ключевое слово, которое затем выполняет соответствующий сценарий.

Плюсы:

- Требуются минимальные знания сценариев.
- Одно ключевое слово может использоваться в нескольких тестах, поэтому код можно использовать повторно.
- Широкая применимость независимо от типа приложения.

Минусы:

- Трудоемкая и затратная по времени настройка и разработка фреймворка.
- Требуются знания ЯП и автотестирования.
- Ключевые слова могут быть проблемой при масштабировании фреймворка. Потребуется расширить список ключевых слов для описания процессов и функциональности.

- **behavior driven**

Особенность: позволяет автоматизировать сценарии в формате, легко читаемом и понятном бизнес-аналитикам, разработчикам, тестировщикам и т. д. Такие фреймворки не обязательно требуют от пользователя знания языка программирования. Для BDD доступны различные инструменты, такие как Cucumber, Jbehave и т.д.

- **гибридные**

Особенность: комбинация двух или более фреймворков, упомянутых выше.

Преимущества автотестирования

- На 70% быстрее, чем при ручном тестировании.
- Более широкий тестовый охват функций приложения.
- Надежные в результаты ?.
- Обеспечивает согласованность тестовых моделей.
- Экономит время и деньги.
- Повышает точность.
- Позволяет исполнять процесс тестирования без вмешательства человека.
- Повышает эффективность.
- Увеличивает скорость выполнения тестирования.
- Повторно использует тестовые скрипты.
- Позволяет тестировать часто и тщательно.
- Большой цикл выполнения может быть достигнут за счет автоматизации.
- Сокращает время выхода продукта на рынок

Недостатки автотестирования

- Отсутствие обратной связи.
- Отсутствие тестирования глазами пользователя.
- Отсутствие возможности тестирования цвета, дизайна и эргономики.
- Спорная надежность, требуется постоянная поддержка.
- Стоимость.
- Требуются навыки программирования.
- Требуются соот-щие специалисты.

Паттерны автоматизации

- Page Object
 - Page Factory
 - Page Element
 - Data Provider
 - Value Object
 - Object Pool
 - Chain Of invocations
 - Builder
 - Decorator
 - и иные
-



Подробнее: <https://habr.com/ru/company/jugru/blog/338836/>

Работа с CSS и xpath локаторами

CSS селектор – это составная часть CSS правила отвечающая за определение конкретных html тегов, к которым будут применены стили оформления, прописанные в этом правиле.

Виды:

- простые
- псевдо-селекторы
- селекторы атрибутов

Особенности:

- поиск только вглубь по DOM-дереву
- простой и компактный синтаксис
- читабелен



Подробнее о типах CSS-селекторов: <https://html5book.ru/osnovy-css/>

XPath использует выражения путей для выбора элементов в документе XML (или в HTML-документе).

Особенности:

- поиск вглубь и вверх по DOM-дереву
- наличие функций (например, contains)
- сложный синтаксис
- возможность делать подзапросы

Некоторые особенности и ограничения обоих инструментов:

- XPATH – мощный и универсальный инструмент, знание которого спасёт в любой ситуации, связанной с поиском элементов, однако за его мощь нужно будет заплатить производительностью.
- CSS-локаторы были разработаны специально для HTML, а XPath-локаторы – это универсальный механизм для поиска по XML DOM. Поэтому для CSS идентификатор и имя – это особые атрибуты, он строит табличку-индекс для быстрого обращения к элементам, снабженным этими атрибутами. Поскольку эти атрибуты должны быть уникальными, индексация работает очень эффективно. А с точки зрения XPath – это "просто атрибуты", поиск ведётся не по таблице-индексу, а по всему DOM-дереву. Поиск по классу уже не так эффективен.
- В большинстве случаев локатор на CSS выглядит лаконичнее и дружелюбнее чем локатор на XPATH.
- В старых версиях IE XPATH обрабатывал значительно медленнее CSS поскольку возможность использования XPATH осуществлялась с помощью JavaScript.
- XPATH движки различны для разных браузеров.

Selenium

Selenium – это проект, в рамках которого разрабатывается серия программных продуктов с открытым исходным кодом (open source):

- Selenium WebDriver – это программная библиотека для управления браузерами.
- **deprecated** Selenium RC – это предыдущая версия библиотеки для управления браузерами. Аббревиатура RC в названии этого продукта расшифровывается как Remote Control, то есть это средство для «удалённого» управления браузером.
- Selenium Server – это сервер, который позволяет управлять браузером с удалённой машины, по сети.
- Selenium Grid – это кластер, состоящий из нескольких Selenium-серверов. Он предназначен для организации распределённой сети, позволяющей параллельно запускать много браузеров на большом количестве машин.
- **deprecated** Selenium IDE – плагин к браузеру Firefox, который может записывать действия пользователя, воспроизводить их, а также генерировать код для WebDriver или Selenium RC, в котором выполняются те же самые действия. В общем, это «Selenium-рекордер».

Selenium WebDriver – инструмент для автоматизации реального браузера, как локально, так и удалённо, наиболее близко имитирующий действия пользователя.

Selenium 2 (или Webdriver) – последнее пополнение в пакете инструментов Selenium которое является основным вектором развития проекта. Это абсолютно новый инструмент автоматизации. По сравнению с Selenium RC Webdriver использует совершенно иной способ взаимодействия с браузерами. Он напрямую вызывает команды браузера, используя родной для каждого конкретного браузера API. Как совершаются эти вызовы и какие функции они выполняют зависит от конкретного браузера. В то же время Selenium RC внедрял javascript код в браузер при запуске и использовал его для управления веб-приложением. Таким образом, Webdriver использует способ взаимодействия с браузером более близкий к действиям реального пользователя.

Самое главное изменение новой версии Selenium - это Webdriver API.

Selenium 1.0 (RC) + WebDriver = Selenium 2.0

По сравнению с более старым интерфейсом он обладает рядом преимуществ:

- Интерфейс Webdriver был спроектирован более простым и выразительным;
- Webdriver обладает более компактным и объектно-ориентированным API;
- Webdriver управляет браузером более эффективно, а также справляется с некоторыми ограничениями, характерными для Selenium RC, как загрузка и отправление файлов, попапы и диалоги.

Для работы с Webdriver необходимо 3 основных программных компонента:

- Браузер, работу которого пользователь хочет автоматизировать. Это реальный браузер определённой версии, установленный на определённой ОС и имеющий свои настройки (по умолчанию или кастомные). На самом деле Webdriver может работать и с "ненастоящими" браузерами, но подробно о них позже.
- Для управления браузером совершенно необходим driver браузера. Driver на самом деле является веб-сервером, который запускает браузер и отправляет ему команды, а также закрывает его. У каждого браузера свой driver. Связано это с тем, что у каждого браузера свои отличные команды управления и реализованы они по-своему. Найти список доступных драйверов и ссылки для скачивания можно на официальном сайте Selenium проекта.
- Скрипт/тест, который содержит набор команд на определённом языке программирования для драйвера браузера. Такие скрипты используют Selenium Webdriver bindings (готовые библиотеки), которые доступны пользователям на различных языках.

Основными понятиями в Selenium Webdriver являются:

- Webdriver - самая важная сущность, ответственная за управление браузером. Основной ход скрипта/теста строится именно вокруг экземпляра этой сущности.
- WebElement - вторая важная сущность, представляющая собой абстракцию над веб-элементом (кнопки, ссылки, поля ввода и др.). WebElement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текущего статуса.
- By - абстракция над локатором веб-элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

Поиск элементов осуществляется благодаря методам:

- findElement
- findElements

Отличие driver.close() и driver.quit():

- первый закрывает окно браузера
- второй закрывает сам инстанс браузера

Selenide

Selenide - это фреймворк для автоматизированного тестирования веб-приложений на основе Selenium WebDriver, дающий следующие преимущества:

Selenide предлагает лаконичный API для использования Selenium WebDriver в UI тестах:

- Умные ожидания
- Новый механизм поиска элементов через \$ и \$\$?, \$x
- Автоматическое управление браузером

Вам больше не надо явно открывать браузер и думать о том, где его хранить и когда закрывать. Selenide сам откроет браузер, когда он впервые понадобится, и закроет, когда он больше не будет нужен.

- Удобные методы

Selenide предлагает лаконичный и мощный API, который поможет вам писать короткие и хорошо читаемые тесты. Selenide есть масса удобных методов для заполнения полей, выбора чекбоксов, выпадающих списков, поиска элементов по тексту и т.д.

```
@Test
public void canFillComplexForm() {
    open("/client/registration");
    $(By.name("user.name")).val("johny");
    $(By.name("user.gender")).selectRadio("male");
    $("#user.preferredLayout").selectOption("plain");
    $("#user.securityQuestion").selectOptionByText("What is my first car?");
}
```

- Поддержка Ajax

При тестировании современных динамических приложений, полных аякса и яваскрипта, нам часто нужно подождать, пока изменится состояние какого-либо элемента. Selenide делает это из коробки. Автоматически. Вам даже не нужно задумываться о том, требуется ли ожидание в том или ином месте.

Все нижеследующие методы могут немножко подождать, если условие не выполнено сразу (по умолчанию до 4 секунд):

```
$("#topic").should(appear);
$("#topic").shouldBe(visible);
$("#topic").should(disappear);
$("h1").shouldHave(text("Hello"));
$(".message").shouldNotHave(text("Wait for loading..."));
$(".password").shouldNotHave(cssClass("errorField"));
$(".error").should(disappear);
```

- **Автоматические скриншоты**

Когда тест падает, Selenide автоматически делает скриншот.



Сравнение Selenium vs Selenide: <https://github.com/selenide/selenide/wiki/Selenide-vs-Selenium>

Явные и неявные ожидания

Неявные ожидания - Implicit Waits - конфигурируют экземпляр WebDriver делать многократные попытки найти элемент (элементы) на странице в течение заданного периода времени, если элемент не найден сразу. Только по истечении этого времени WebDriver бросит `ElementNotFoundException`.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://some_url");
WebElement dynamicElement = driver.findElement(By.id("dynamicElement_id"));
```

Явные ожидания - Explicit Waits - это код, который ждет наступления какого-то события (чаще всего на AUT), прежде чем продолжит выполнение команд скрипта. Такое ожидание срабатывает один раз в указанном месте. Самым худшим вариантом является использование `Thread.sleep(1000)`, в случае с которым скрипт просто будет ждать определенное количество времени. Это не гарантирует наступление нужного события, либо будет слишком избыточным и увеличит время выполнения теста.

Более предпочтительно использовать `WebDriverWait` и `ExpectedCondition`:

```
WebDriver driver = new FirefoxDriver();
driver.get("http://some_url");
WebElement dynamicElement = (new WebDriverWait(driver, 10))
    .until(ExpectedConditions.presenceOfElementLocated(By.id("dynamicElement_id")));
```

В данном случае скрипт будет ждать элемента с `id = dynamicElement_id` в течении 10 секунд, но продолжит выполнение, как только элемент будет найден.

Rest Assured

Rest Assured - это Java-библиотека для тестирования RESTful API. Код, написанный с помощью этой библиотеки, имеет простой и понятный синтаксис. В тесте можно выполнить запрос к API буквально в одну строчку кода и при помощи синтаксиса проверять полученный результат.

Возможности библиотеки:

- использован BDD-подход, что делает библиотеку интуитивно понятной;
- простая отправка HTTP-запросов с необходимыми параметрами (поддерживает отправки запросов с хидерами, параметрами и телом);
- методы проверки ответов (проверка статус кода, ожидаемого тела и т.д.);
- создание POJO или классов-обертки и конвертация их в json;
- создание кастомных спецификаций.

Хорошие практики:

- вынесение эндпоинтов в отдельный класс(ы);
- использование встроенных механизмов проверок;
- использование `response-request-спецификаций`;
- использование нативных возможностей работы с заголовками (например, `contentType(ContentType.JSON)`);
- возможность создавать и преобразовывать POJO.

Работа с отчетами

Популярные инструменты для создания автогенерирующихся отчетов:

- Allure
- Report Portal
- Extent Test Framework

В идеале отчет о прогоне автотестов включает:

- номер кейса
- название кейса
- название фичи
- ссылка на кейс
- дата прогона
- время прогона
- уровень кейса
- шаги выполнения
- статус
- логи
- иные артефакты (видео, фото)
- процент соотношения успешно пройденных кейсов и неуспешных

DDT, TDD, BDD

DDT (Data Driven Testing) – подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще всего применимо к backend тестированию), при котором тест умеет принимать набор входных параметров, и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров.



Подробнее: <https://jazzteam.org/ru/technical-articles/data-driven-testing/>

TDD (Test Driven Development) – разработка на основе тестов.

BDD (Behavior Driven Development) – разработка на основе поведения. BDD, на самом деле, является расширением TDD-подхода. Тем не менее, они предназначены для разных целей и для их реализации используются разные инструменты. В разных командах эти понятия могут интерпретировать по-разному, и часто возникает путаница между ними.

В чем разница:

- TDD хорошо подходит для юнит-тестирования, т.е. для проверки работы отдельных модулей самих по себе. BDD – для интеграционного (т.е. для проверки, как отдельные модули работают друг с другом) и e2e (т.е. для проверки всей системы целиком) тестирования.
- TDD: тесты сразу реализуются в коде, для BDD чаще всего описываются шаги на языке, понятном всем, а не только разработчикам.
- TDD: юнит-тесты пишут сами разработчики. BDD требует объединения усилий разных членов команды. Обычно тест-кейсы (шаги) описываются ручным тестировщиком или аналитиком и воплощаются в код тестировщиком-автоматизатором. В нашей команде мы (фронтендеры) описываем шаги вместе с тестировщиками, а код тестов пишет фронтенд-команда.
- TDD проверяет работу функций, BDD – пользовательские сценарии.

Основной идеей BDD является совмещение в процессе разработки чисто технических интересов и интересов бизнеса, позволяя тем самым управляющему персоналу и программистам говорить на одном языке. Для общения между этими группами персонала используется предметно-ориентированный язык, основу которого представляют конструкции из естественного языка, понятные неспециалисту, обычно выражающие поведение программного продукта и ожидаемые результаты.

BDD фокусируется на следующих вопросах:

1. С чего начинается процесс.
2. Что нужно тестировать, а что нет.
3. Сколько проверок должно быть совершено за один раз.
4. Что можно назвать проверкой.
5. Как понять, почему тест не прошёл.

Исходя из этих вопросов, BDD требует, чтобы имена тестов были целыми предложениями, которые начинаются с глагола в сослагательном наклонении и следовали бизнес-целям. Описание приемочных тестов должно вестись на гибком языке пользовательской истории, например, *Как [роль того, чьи бизнес-интересы удовлетворяются] я хочу, чтобы [описание функциональности так, как она должна работать], для того чтобы [описание выгоды].*

Критерии приёмки должны быть описаны через сценарий, который реализует пользователь, чтобы достигнуть результата.

Тесты для некоторой единицы программного обеспечения должны быть описаны с точки зрения желаемого поведения программируемого устройства. Под желаемым поведением понимается такое, которое имеет ценность для бизнеса. Описание желаемого поведения даётся с помощью спецификации поведения (англ. behavioral specification).

Спецификация поведения строится в полужформальной форме. В настоящее время в практике BDD устоялась следующая структура:

- Заголовок (англ. Title). В сослагательной форме должно быть дано описание бизнес-цели.
- Описание (англ. Narrative). В краткой и свободной форме должны быть раскрыты следующие вопросы:
 - Кто является заинтересованным лицом данной истории;
 - Что входит в состав данной истории;
 - Какую ценность данная история предоставляет для бизнеса.
- Сценарии (англ. Scenarios). В одной спецификации может быть один и более сценариев, каждый из которых раскрывает одну из ситуаций поведения пользователя, тем самым конкретизируя описание спецификации. Каждый сценарий обычно строится по одной и той же схеме:
- Начальные условия (одно или несколько);
- Событие, которое инициирует начало этого сценария;
- Ожидаемый результат или результаты.

BDD не предоставляет каких-либо формальных правил, но настаивает на том, чтобы использовался ограниченный стандартный набор фраз, который включал бы все элементы спецификации поведения. В 2007 году Дэном Нормом был предложен шаблон для спецификации, который получил популярность и впоследствии стал известен как язык **Gherkin**.

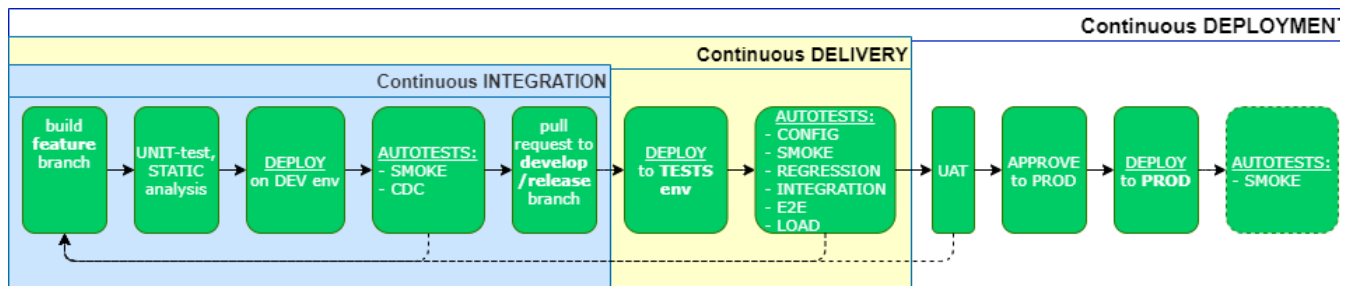
Основные фразы языка Gherkin представлены в таблице.

Ключевое слово на английском языке	Русскоязычная адаптация	Описание
Story (Feature)	История	Каждая новая спецификация начинается с этого ключевого слова, после которого через двоеточие в сослагательной форме пишется имя истории.
As a	Как (в роли)	Роль того лица в бизнес-модели, которому данная функциональность интересна.
In order to	Чтобы достичь	В краткой форме какие цели преследует лицо.
I want to	Я хочу, чтобы	В краткой форме описывается конечный результат.
Scenario	Сценарий	Каждый сценарий одной истории начинается с этого слова, после которого через двоеточие в сослагательной форме пишется цель сценария. Если сценариев в одной истории несколько, то после ключевого слова должен писаться его порядковый номер.
Given	Дано	Начальное условие. Если начальных условий несколько, то каждое новое условие добавляется с новой строки с помощью ключевого слова And.
When	Когда (прим.: что-то происходит)	Событие, которое инициирует данный сценарий. Если событие нельзя раскрыть одним предложением, то все последующие детали раскрываются через ключевые слова And и But.
Then	Тогда	Результат, который пользователь должен наблюдать в конечном итоге. Если результат нельзя раскрыть одним предложением, то все последующие детали раскрываются через ключевые слова And и But.
And	И	Вспомогательное ключевое слово, аналог конъюнкции.
But	Но	Вспомогательное ключевое слово, аналог отрицания.

Пример сценария:

```
:
  1:
  2:
```

CI/CD/CDP



Continuous integration, CI = автоматизированная интеграция программного кода в существующий проект в репозитории с последующей компиляцией, формированием сборки и прогоном базовых автотестов. Считается, что должно занимать ≤ 10 минут.

Continuous delivery, CD = CI + автоматизированная поставка готовой сборки ПО с изменениями на сервера разработки и тестирования с прогоном автотестов.

Continuous deployment, CDP = CD + автоматизированное развёртывание изменений в Пром.

Pipeline

Jenkins Pipeline - набор плагинов, позволяющий определить жизненный цикл сборки и доставки приложения как код. Он представляет собой Groovy-скрипт с использованием Jenkins Pipeline DSL и хранится стандартно в системе контроля версий.

Существует два способа описания пайплайнов - скриптовый и декларативный.

1. Scripted

```
node {
    stage('Example') {
        try {
            sh 'exit 1'
        }
        catch (exc) {
            throw exc
        }
    }
}
```

2. Declarative

```
pipeline {
    agent any
    stages {
        stage("Stage name") {
            steps {}
        }
    }
}
```

Они оба имеют структуру, но в скриптовом она вольная - достаточно указать, на каком слейве запускаться (node), и стадию сборки (stage), а также написать Groovy-код для запуска атомарных степов.

Декларативный пайплайн определен более жестко, и, соответственно, его структура читается лучше.

- В структуре должна быть определена директива pipeline.
- Также нужно определить, на каком агенте (agent) будет запущена сборка.
- Далее идет определение stages, которые будут содержаться в пайплайне, и обязательно должен быть конкретный стейдж с названием stage("name"). Если имени нет, тест упадет в runtime с ошибкой «Добавьте имя стейджа».
- Обязательно должна быть директива steps, в которой уже содержатся атомарные шаги сборки. Например, вы можете вывести в консоль «Hello».

```

pipeline { // pipeline
  agent any // ,

  stages { //
    stage("Stage name") { //
      steps { //
        echo "Hello work" //
      }
    }
  }
}

```

Delivery Pipeline - это способ разделить процесс сборки на этапы. Ранние этапы - самые быстрые и должны находить наибольшее количество ошибок, в то время как поздние - значительно медленнее и чем дальше от начала они находятся, тем больше опираются на исследования.

Определение шагов Delivery Pipeline

- *Этап фиксации изменений*

Самый первый, технический уровень Delivery Pipeline. Обычно он состоит из следующих шагов:

1. Компиляция
2. Выполнение Unit-тестов (на код, архитектуру и инфраструктуру)
3. Статический анализ кода

После прохождения этого этапа, разработчик может продолжить разработку. Этап должен быть максимально быстрым.

- *Этап автоматизированного приемочного тестирования*

Второй этап - функциональный. Он состоит как минимум из двух шагов:

1. Выполнение автоматических функциональных/приемочных тестов
2. Проверка атрибутов качества, таких как производительность, безопасность, масштабируемость.

- *Этап релиза в боевую среду*

Поставка продукта пользователю. Хорошим тоном является отделить поставку от релиза, тогда появляется возможность поставлять изменения на боевую среду автоматически, даже если какая-то функциональность готова не полностью, а релизить её, то есть показывать клиенту, только тогда, когда это действительно нужно.

Для этих целей можно использовать, например, Feature Toggles/Flags. Это своего рода тумблер, позволяющий включать и выключать фичу через конфигурационные параметры. Применение Feature Flags позволяет отделить поставку от релиза и отключить функциональность на боевых серверах в случае необходимости. Это узкое определение. Более широкое - дает полный контроль над жизненным циклом отдельно взятой фичи.

Стоит предусмотреть способы включения/выключения, это могут быть:

- Командная строка
- База данных
- Админка
- API (Rest)
- Условный оператор в коде

Logging

Логированием называют запись логов. Оно позволяет ответить на вопросы, что происходило, когда и при каких обстоятельствах. Без логов сложно понять, из-за чего появляется ошибка, если она возникает периодически и только при определенных условиях. Чтобы облегчить задачу администраторам и программистам, в лог записывается информация не только об ошибках, но и о причинах их возникновения.

Уровни логирования в порядке возрастания приоритета:

- ALL - логирование всех событий. Существенно уменьшает производительность приложения.
- DEBUG - логирование всех событий при отладке.
- INFO - логирование ошибок, предупреждений и сообщений.
- WARN - логирование ошибок и предупреждений.
- ERROR - логирование всех ошибок.
- FATAL - логирование только ошибок, приводящих к прекращению работы компонента, для которого ведется логирование.
- OFF - логирование отключено.

Из известных решений по логированию в Java можно выделить:

- log4j
- JUL - java.util.logging
- JCL - jakarta commons logging
- Logback
- SLF4J - simple logging facade for java

Все ли ошибки стоит логировать? (дополнить)