**Software Test Engineer** – выполняет функциональное тестирование продукта;

**Quality Control Engineer** – проверяет на соответствие требованиям;

**Quality Assurance Engineer** – обеспечивает контроль качества на всех этапах планирования, проектирования и разработки.

**Цель тестировщиков и QC** – обнаружить неисправности и расхождения с требованиями, а QA – не допустить появление дефектов через улучшение процессов разработки и тестирования. **Quality Assurance** – это обеспечение качества, а тестирование и Quality Control – часть этого процесса.

В наших реалиях нет разделения на подкатегории, а перечисленными выше задачами занимается Manual QA инженер – он планирует, тестирует и участвует в улучшении процессов.

QA-инженер – QA Engineer, или Quality Assurance engineer (Quality Assurance дословно с английского переводится как «обеспечение качества»), – это специалист по функциональному тестированию программного обеспечения на этапе разработки. В отличие от тестировщика, который занимается проверкой работоспособности продукта, QA-инженер видит полную картину и налаживает процессы таким образом, чтобы минимизировать риск ошибок и уменьшить количество дефектов. Тестирование лишь часть этого процесса, а задача QA – эффективно интегрировать его с разработкой.

**Позитивное тестирование** – когда применяются сценарии, которые соответствуют нормальному, ожидаемому поведению системы (н-р. Калькулятор 2+2=4)

**Негативное тестирование** – нештатное поведение системы, например, деление на 0

**Классификация тестирования.**

**Тесты по степени важности**

**Smoke тест**. Дымное тестирование. Имеет 2 варианта проверки, да или нет, работает или нет, если нет, дальше не тестирует. Например, печники построили печь, развели огонь и если из щелей идёт дым, значит нарушили строительство печи и надо переделывать.

**По цели тестирования** (Critical plan testing), проверяется осн. масса требований к продукту

**Расширенный тест** (extended test), проверка нестандартного использования программного продукта.

**По цели тестирования**

**New Feature test** – проверка качества новой функциональности и проходит все этапы, обычно проходит все этапы тестирования.

**Regression testing** – тестирование раннее проверенной функциональности, чтобы убедиться что новые добавления, не повлияли на старое. Проводится в каждом билде. Проверка исправленных багов. Проводится несколько раз, часто автоматизируется.

Всегда нужно включать тестирование безопасности или важные функции для бизнеса. Вкл. те области которые чаще всего меняются в ходе разработки. Функции с высокой вероятностью ошибок.

**Re-test** – если был баг и его исправили, это нужно проверить.

**По степени автоматизации**:

Ручное

Автоматизированное

**По знанию кода:**

**Тестир. Чёрного ящика** (Black box testing) - мы не знаем как устроена тестируемая система, как устроен код, основана на работе с внешним. интерфейсом (поля ввода кнопки), т.е. тестируем только GUI

**Тестир. Серого ящика** (Grey box testing) – известны только некоторые особенности реализации системы, внтр. устройство программы нам известно частично, само тестирование проводится только с позиции black box

**Тестир. Белого ящика** (White box testing) – известны все детали реализации тестируемой программы…тестируемого ПО, определяет как правильные так и не правильные пользовательские вводы, ожидаемый результат определяется именно тем, как должен работать код программы.

**Функциональное и нефункциональное тестирование.**

**Функц. –** осн. задача подтверждение того, что разрабатываемый продукт обладает всем функционалом, который требуется заказчику.

**Нефункц. –** является проверка свойств приложения, с его нефункциональными требованиями, по сути это тест. свойств которые не относятся к функциональности системы, например надёжность.

**тестирование на отказ и восстановление**

**тестирование производительности** (опред. Работоспособности, стабильности, потребление ресурсов)

Нагрузочное, стресс, тестирование стабильности, объёмное

**Тестир. Удобства использования**

**Тестир. Безопасности**

**Тестир. Установки (портированность)**

**Конфиг. Тестирование. – кроссплатформенное и кроссбраузерное**

**Тестир. Локализации (l10n)**

**Тестир. Интернационализации (i18n)**

**Тестирование GUI**

**Тестирование доступности -** для людей с огранич. возвожностями

**Тестирование по исполнению сценария (ad hoc тестирование)**

**Ad-hoc тестирование** – без использования спецификаций, планов, импровизация … первый раз видите приложение и понимаете как с ним нужно работать

**Исследовательское тестир. –**  не требует написания тест кейсов, но подразумевает, что каждый. послед. тест опирается на результаты предыдущего тестирования.

**Сценарное тестирование** – классич. тестир. по предварительным написанным и задокументир. тестовым сценариям.

**По запуску кода**

**Статическое –** не предполагает, что программный код при тестировании будет выполнятся, начинается на ранних этапах жизненного цикла ПО.

**Динамическое -**  предполагает запуск программного кода.

**Модели разработки ПО**

**Водопадная (каскадная (waterfall)) –** каждый этап разработки ЖЦ ПО продолжает предыдущий. Т.е. идёт сверху вниз.

1. Анализ требований
2. Стадия дизайна (архитектура, база данных)
3. Разработка (кодинг)
4. Тестирование (тестовые данные, системы для тестирования, выводы)
5. Поддержка приложения, поддержка клиентов

Плюсы, полное документирование каждого этапа.

Минусы, до начала проекта, нужно утвердить полный объём требований к системе, увеличение затрат средств, в случае изменения требований.

Используется в проектах с чёткими определён. требованиями, н-р: гос. структура, банковская система, для проектов, которые мигрируют с одной платформы на другую.

**V-модель –** нисходящая последовательность

Плюсы, строгие этапы, планирование тестирование и верификация на ранних этапах, промежуточное тестирование.

Минусы, нет гибкости модели, написание кода только в середине процесса, нет динамического внесения изменений.

**Итерационная** (инкрементальная (iterative & incremental)) модель – в каждой итерации, отдельная разработка компонента.

Плюсы, раннее создание работающего ПО, готовы к изменениям требований на каждом этапе разработки

Минусы, каждая фаза самостоятельная, не все требования известны.

**Гибкая методология Agile model** (Scram, Kanban) – для работы небольшого кол-ва людей, спринт 2-4 недели, итерация 24 часа.

Есть ещё **Rad model, Spiral model, prototype model**…

**Agile (эжайл)**

**Agile** не говорит, что делать, он говорит как это делать.

**Agile-манифест (ценности):**

Люди и взаимодействие важнее процессов и инструментов

Работающий продукт важнее исчерпывающей документации

Сотрудничество с заказчиком важнее согласования условий контракта

Готовность к изменениям важнее следования первоначальному плану

**Подробнее о ценностях:**

1. Высший приоритет, удовлетворить заказчика с помощью раннего и перманентного деливери рабочего софта.
2. Изменение требование приветствуется.
3. Мы должны доставлять изменения часто (неделя - месяц)
4. Бизнес и разработчики должны работать вместе.
5. Нужно строить проект вокруг тех кому он важен, поддерживать и дать, что им нужно.
6. Самый эффективный способ коммуникации, это лицом к лицу.
7. **Рабочий софт является мерилом прогресса.**
8. Процесс должен поддерживать устойчивую работу.
9. Постоянно нужно уделять внимание хорошим практикам и хорошему дизайну.
10. Простота
11. Самоорганизующиеся команды.
12. Команда должна работать эффективно и когда регулярно показывает эффективность, тогда вносить правки в процесс и менять её поведение.

Sprint – некий временной отрезок 2-4 недели, в течении которого создаётся инкремент продуктаготовый к использованию.

Традиционное тестирование – тестировщик в первую очередь сфокусирован на том, чтобы найти расхождение, между тем, что он видит в разработанном продукте и тем, что указано в требованиях.

Agile тестирование – главная задача тестировщика организовать тестирование таким образом, чтобы предупредить о возможных расхождениях между разработанным продуктом и сущ-ми требованиями.

**Scrum**

Один из главных Фреймворков Agile.

Концепция Scrum: Product backlog (Список требований) –> Sprint backlog (выполнение короткой операции) -> итерация проходит 2-4 недели -> Shippable product (Готовый продукт)

**Scrum roles**

**Roles**:

Product Owner – человек который решает, что должно быть в продукте, какой спринт сейчас важнее, знает, что должно быть в продукте, планирует.

Scrum master – следит, чтобы все участники проекта следовали правилам Scrum. Помогает Product owner и разработчикам.

Development team – команда разработчиков, дизайнеров и т.п. 3-9 человек.

**Events:**

Sprint grooming (backlog refinement) – мероприятие, когда мы выбираем, что будет в спринте, какие вещи в нём будут. История задач и задаём вопрос Product Owner, что делать в след. Спринте. Чётко понять чего хотят от нас в сторе.

Sprint planning – планирование спринта. Разбиение больших задач на стори, на маленькие таски, и оцениваем сколько времени потребуется. Выбираем что будем делать в стори. Сколько времени займёт.

Daily Scrum/Standup – небольшое совещание (пятиминутка). Где делимся инфф. Со своей командой. Каждый день. Каждый говорит, кто что сделал, вчера, что планирует, есть ли сложности.

Sprint review / Demo – Показываем, что было сделано, кто ответственен за задачу, тот и показывает. Получаем фитбэк, классно или не классно и т.п. Раз за один спринт.

Sprint Retrospective – что было классно в спринте (мне понравилось, что в спринте, не было зависимости отдой фичи от другой или подобное), что можно улучшить. Нужен человек и сроки, чтобы исправить предыдущие ошибки, сделать след. спринт лучше.

**Burndown chart –** график, в одной оси время в другой усилия (ресурсы, человеко-часы).

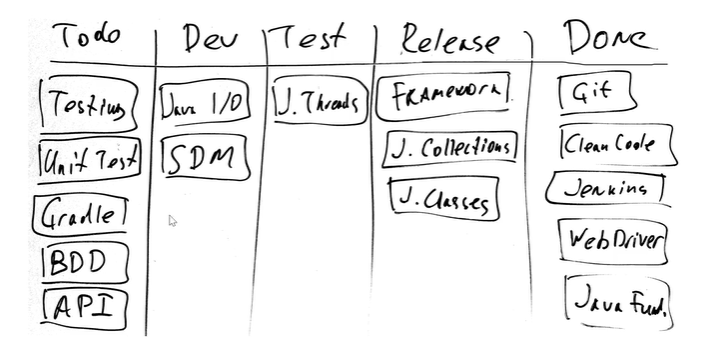
**Artifacts** (см. концепция Scrum)

**Other**

**Kanban**

Модель разработки Kanbun.

Доска:



Нет жёстких правил. Не нужно планировать сколько чего нужно сделать за 2 недели.

Подходит для поддержки.

В начале разработки.

Приходящие специалисты.

**Экстремальное программирование.**

Улучшение процесса разработки продукта.

Осн. моменты:

Планирование, короткие релизы (часто маленькие кусочки), метафора (сложные вещи простыми словами, примерами), простой дизайн (перед тем как его усложнять), тестирование, рефакторинг (практика по улучшению организации кода, без изменения его функционала), парное программирование, 40-к часов в неделю, стандарты написания кода.

**Test-Driven Development**

Одна из практик экстремального программирования.

Сначала пишем тесты, а затем уже код. Сначала создаём тесты, кода нет. Затем пишется код, тесты проходят, затем код шлифуется, делается «красивым».

Чистый, читаемый код. Легко и безопасно рефакторить (меняется название класса, функции), юнит-тестирование помогает улучшать код, большая часть команды подтверждает работоспособность.

**Анализ требований.**

**Уровень бизнес требований** – цель, ради которой создаётся наш продукт (для чего, какая польза)

**Уровень пользовательских требований** – покрывает задачи которые пользователь может выполнять с помощью продукта.

**Уровень продуктовых требований** – как это всё будет реализовано в нашем продукте

**Функциональные** – что система должна делать

**Нефункциональные** – как система должна это делать

**Тест-стратегия, тест-план, отчёт о тестировании**

**Тест – стратегия –** высокоуровневый тест-план.

Виды: Отдельный документ. Часть тест-плана

Этапы: Сбор инф-ии, Анализ инф-ии, Принятие решений, Презентация

Кто составляет? Менеджер. Тестировщик (middle +)

Что включает? Цель тестирования, Инф-ия о продукте, Виды тестирования, критерии начала и окончания тестирования

**Тест-план –**

Кто составляет? Менеджер. Тестировщик (middle +)

Виды: Мастер (для всей команды в одном проекте), детальный (Для каждой команды, итерации, релиза), План приёмочных испытаний

Стратегии: Виды тестирования, Приоритет тестир., Стратегия тестир., Окружение, Задачи тестировщиков, Тестовая документация, Техники тест-дизайна, Баг-трекинг, Критерии начала и окончания тестирования, Инструменты для работы, Инструменты для работы, Оценки качества, метрики.

**Отчёт о тестировании –** результирующий документ.

Типы: Промежуточный (дневной, недельный, месячный, Версионный-отчёт по итерации), Финальный (Договорённость с заказчиком, о том какие баги допустимы, например, medium or low)

Части отчёта: Состав команды, Сроки, Описание процессов тестирования, Дополнение к тестовым кейсам, Процент пройденных кейсов, Критичные баги, Результаты регрессионного тестирования, Планы (только для промежуточных)

**Тестирование клиент-серверной архитектуры**

Архитектура, в которой сетевая нагрузка распределяется между поставщиками услуг (сервер) и заказчиками услуг (клиент). Фактически клиент и сервер это некое ПО, расположены на разных вычислит. машинах и взаимодействую между собой через сетевые протоколы. Но так же могут располагаться на одной машине. Распространённый вид протокола http.

2-х уровневая система: Сервер – Клиент

3-х уровневая система: база данных – сервер – клиент

Клиент: Тонкий (компьютер или программа которая переносит все или большую часть обработки информации на сервер), толстый (приложение которое обеспечивает расширенную функциональность независимо от центрального сервера, сервер только как хранилище данных Например 1С, Wot).

Web- сервисы – интеграционный уровень. Это прикладной программный интерфейс API. Который работает на сервере и предоставляет клиенту данные через протокол http, через стандартизированную систему обмена сообщениями. Веб-сервисы подразделяются на Soap и Rest. Предпочитают использовать Rest. Soap – стандартизированный, а Rest более гибкий и интересный с точки зрения веб-разработки. Soap – xml и по объёму инфы больше чем в Rest (Json – сокращение времени запросы и ответы на сервере)

Web-приложения – интерактивные комп-ые приложения, которые разрабатываются для сети интернет и позволяют пользователям вводить, получать и манипулировать данными с помощью взаимодействия. Такие программы имеют тесную связь с сервером и отправляют на него много запросов. Могут быть встроены в веб-страницы или сами веб-страницы могут являться приложениями, например Facebook, Gmail, youtube, twitter… веб-приложения используют имя пользователя и пароль для аутентификации и позволяют своим посетителям обмениваться мгновенными сообщениями если это соц. сети. Отличие от веб-сайтов многие приложения могут не иметь реального информативного содержания. Они используются для выполнения каких-то дополнительных задач, какие-то интернет переводчики. Мессенджеры, конвертеры файлов. Валют.

Web-сайты – носят какой-то инф-ный хар-тер, состоят из неких веб-страниц, объединённых в единый ресурс, имеют простую архитектуру на основе html кода. Не предоставляют возможностей для взаимодействия с нашей программой. (Новостные, кулинарные, погода)

**Introduction to Testing (Введение в тестирование)**

Тестовая стратегия.

1. Какой продукт, какая у него специфика?
2. Какие у нас временные рамки на его тестирование?

Тестовая стратегия:

1. Какие тестовые типы у нас будут выполнены в рамках тестирования?
2. Как будем выполнять эти типы тестирования, как можем объединить или оптимизировать?
3. Что будет выполняться в рамках определённых фаз?

## Основные моменты по настройке GIT

Загрузить гит можно тут: <http://git-scm.com/downloads>

Устанавливайте гит с настройками по умолчанию.

Генерация пары ssh ключей:

     ssh-keygen -t rsa -C "vitali\_shulha@epam.com"

Публичный ключ (id\_rsa) нужно отправить владельцу репозитория для получения прав работы. Или загрузить в настройки профиля в bitbucket/github/gitlab

Настройка имени пользователя и емейла:

     git config --global user.name “Vitali Shulha“

     git config --global user.email [vitali\_Shulha@epam.com](mailto:vitali_Shulha@epam.com)

* **Переход в нужную папку**: cd /d/Git/ (диск D, папка Git), для перехода в подпапку cd git-demo/
* **pwd** – посмотреть **в какой папке сейчас находишься**.
* **git clone** – **клонирование**. Спросит, хотите ли вы продолжить (yes/no), затем попросит пароль от git bash.
* **Вставка инф-ии** находящейся в буфере обмена **Shift+Insert**
* **Ls** – **просмотр всех папок и файлов** в папке в которой сейчас находимся.
* Чтобы убедиться, что это гитхаб, а не другой хостинг набрать **git remote -v**
* **git status** – говорит где находимся и что есть нового между локальными данными и на сервере git.
* Если нужно **добавить что-то в git** используется команда **git add имя файла**, н-р git add index.php
* **git reset имя файла** – для отмены изменений
* **git add имя файла** – для добавления файла в индекс.
* **git commit** – сохранить изменения. С флагом –m для добавления сообщения к коммиту (н-р git commit –m “first change”)
* **git log** – история коммитов (сохранений)
* **git push** – синхронизация локального репозитория с удалённым.
* Просмотр информации последнего коммита: **git show –s --pretty=raw 6601e** (4-5 символов последнего коммита)
* Просмотреть дерево после git show: **git ls-tree e8e9b** (символы из строки tree)
* **git show 2c2cf** (просмотр изменений в нужном файле )

Какой командой можно воспользоваться чтобы "подтянуть" изменения с удаленного репозитория в локальный? git pull

Какая комбинация команд в git bash будет правильной для добавления нового файла в index, коммита и отправки на github? git add, git commit, git push

сначала добавляем файлы через add, потом делаем коммит и синхронизируем локальный репозиторий с удаленным через git push

**Файл .git/refs/heads/master содержит в себе следующий текст: "5c2fdbaff3b0b9b5eae3c48047aafcee2201e5c4".** Это sha1 последнего коммита в master ветке

**Что означают имена директорий в папке .git/objects?** Первые два символа из sha1

**Git Gui & gitk**

Для вызова из git bash **Git Gui** набрать команду **git gui&**

Для вызова из git bash **gitk** набрать команду **gitk&**

**Отмена изменений сделанные в Git**

**Working directory (добавили изменения в файл или папку):**

* **git checkout -- имя файла** – команда отмены изменений в указанном файле (в существующем файле).
* **git checkout** . – команда отмены изменений во всех существующих файлах (отменённые изменения вернуть назад нельзя)
* **git clean –xdf –** команда которая удаляет новые файлы (которые ещё не коммитились и не передавались на сервер Git). Флаги: **X- git ignore, D – Удалять директории, F – force** (без неё ничего не произойдёт и файлы или папки не удалятся)

**Staging area (Index)(выполнили git add, т.е. подготовили для commit (сохранение на сервере git)):**

* **git reset – имя файла -** Если файл добавлен в index (выполнена команда **git add имя файла**), но не был ещё произведён commit.

**Local branch (Добавили изменения, проиндексировали и закоммитили)(отмена commit):**

* **git commit -- amend -m “commit mesage” –** если commit хороший, но что-то забыли, этой командой можно обновить инф-ию, а не делать новый коммит **-m “commit mesage”** – если хотим добавить пояснение к обновлению данных.
* **git reset HEAD^ или (HEAD~1) -** полная отмена commit, кол-во ^ указывает сколько коммитов хотим удалить, если их много можно прописать HEAD~кол-во отмен.

git reset -- soft HEAD^ - возвращает закоммиченный файл в индекс (самый безопасный вариант)

git reset -- mixed HEAD^ - файл модифицирован на диске и не добавлен в индекс.

git reset -- hard HEAD^ - все изменения которые были в коммите полностью исчезают.

**Remote repository (Удаление репозитория):**

* **git revert <sha1> -** отмена изменений которые уже находятся в github

**git revert «первые 5 символов нужного коммита»** н-р.: **git revert 06052,** узнать можно запустив команду **git log**.

Для того, чтобы редактировать текст в тестовом редакторе **Vim** нужно нажать кнопку insert, для того чтобы выйти из режима insert (из режима редактирования) нажать ESC.

**:wq** – записываем контент файла на диск и выходим из **Vim**.

**:q!** – не менять, не вносить никакие правки и просто выйти.

**Ctrl+Ins –** копировать в Git Bash

**Shift+Ins –** вставить в Git Bash

**Файл .gitignore**

Нужен для того, чтобы git не видел каких-то файлов или папок в папке с проектом. Создать в локальной папке github где расположен проект файл .gitignore и далее записываем в него.

\*.log – скрыть все файлы с расширением log (имена и расширения могут указываться любые)

!errore.log – открыть только этот файл.

/TODO – скрыть все директории

Build/ - скрыть все файлы после /

**git checkout feature –** для перехода в другую ветку, эта ветка называется **feature**, название может быть любым. В основном и в самом начале работаем в ветке **main или master**.

Чем чаще делаешь коммиты, тем лучше. Лучше 10 коммитов по одной фиче (правке, изменению), чем один, который будет содержать 10 фич.

**Git Conflict**

Merge – слияние двух веток

Git merge --aboort – отмена merge (слияния )

Git checkout --Xours - git если есть конфликт, считай, что моя версия правильная

Git checkout --Xtheirs - git если есть конфликт, считай, что чужая версия правильная.

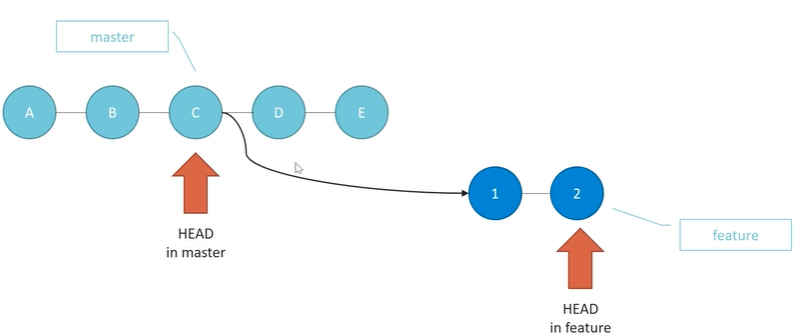
Git diff – решать в ручную

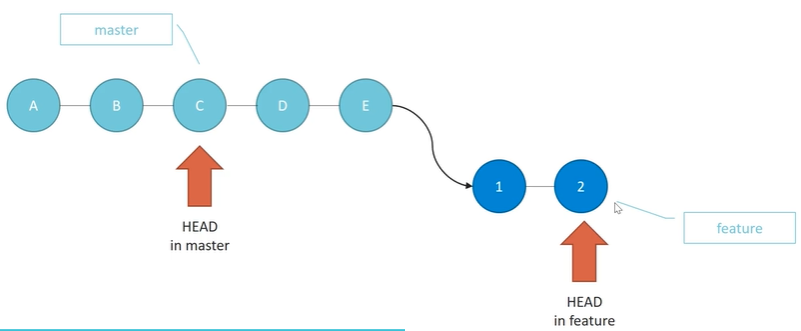
Git revert 09fe472 – вернуться к версии такой-то

Так же есть различные user merge tool – программы которые помогают найти различия, конфликты в файлах (например различия в excel документе)

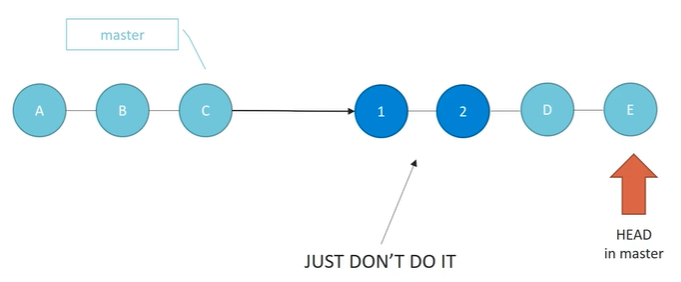
**Команда Rebase**

Изменение базы, например с коммита С перенесли в коммит Е.

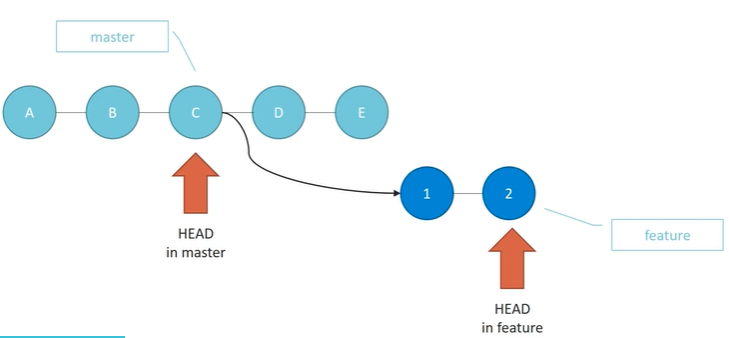




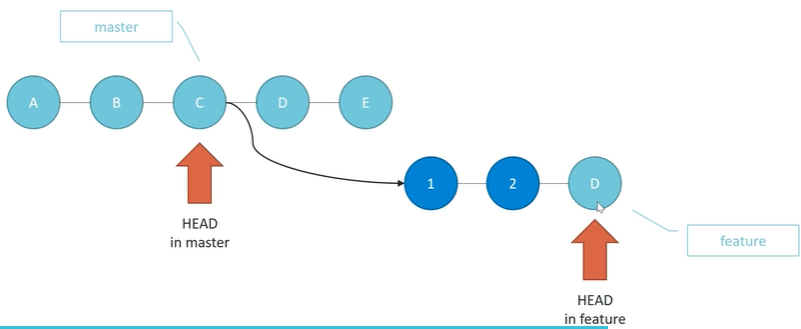
ТАК ДЕЛАТЬ НЕ НАДО:



**Cherry-pick**



Например вышла нужная фича или исправлен баг в коммите D, но наша фича 2 ещё не закончена, по этому merge (заливать нашу фичу в общую ветку) делать не хочется.



### Сколько занимает branch (ветка) в файловой системе? **41 байт** Как посмотреть список веток в репозитории? **git branch --all** Как решить конфликт при слиянии веток (merge conflict)? **Исправить содержимое файлов с конфликтами, сделать коммит изменений или сделать git merge --continue.**

### **Tags (теги)**

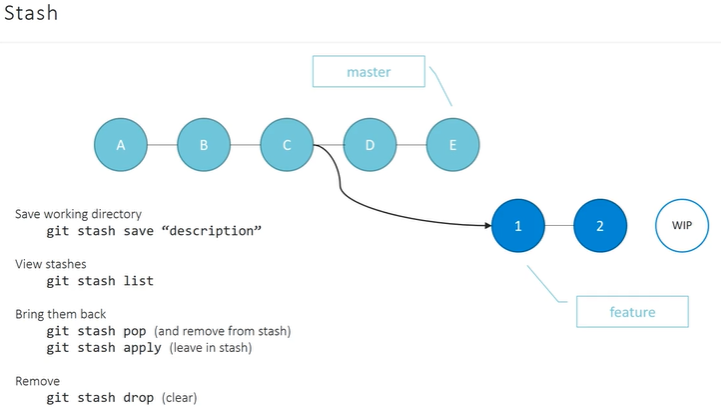
### 

### Подсветить коммит тегом – **git tag ver1** (ver1 и есть тег, можно написать всё, что хочешь) в **gitk&** этот маркер будет выделен жёлтым. Чтобы сразу перейти в нужный коммит помеченный тегом, можно набрать **git checkout ver1**

Просмотреть список тегов – **git tag --list**

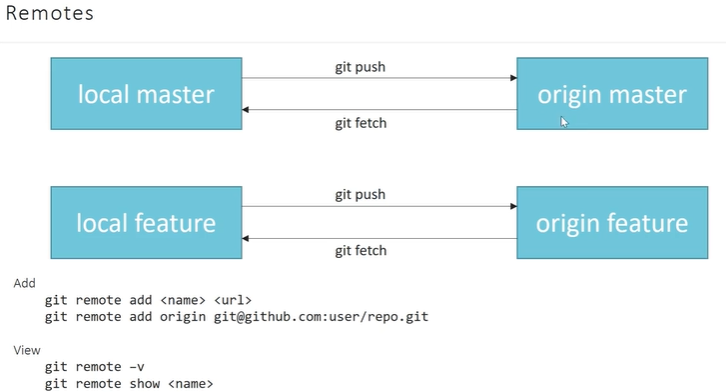
**Stashing (тайник, временное хранилище)**

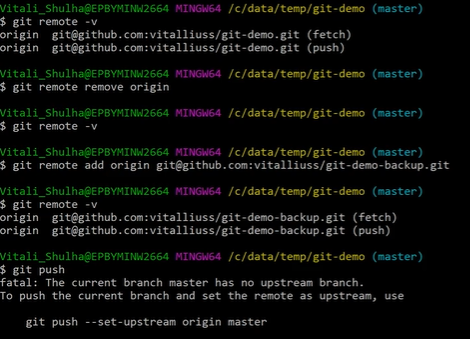
Осн. идея Stash, сохранить свою работу, которая не закоммичена и не готова к коммиту в какое-нибудь хранилище.



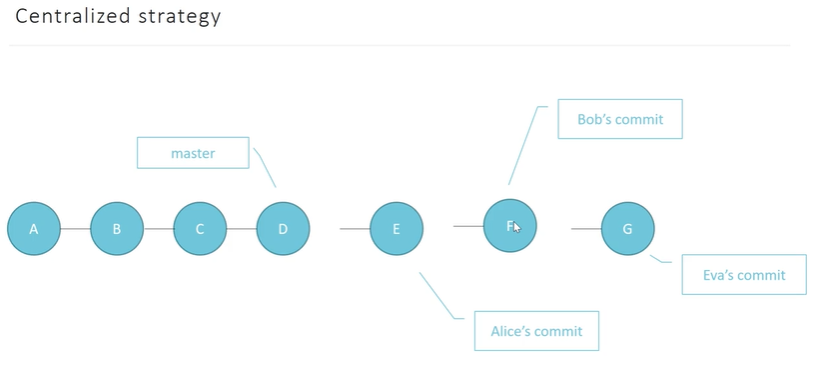
**Remotes**

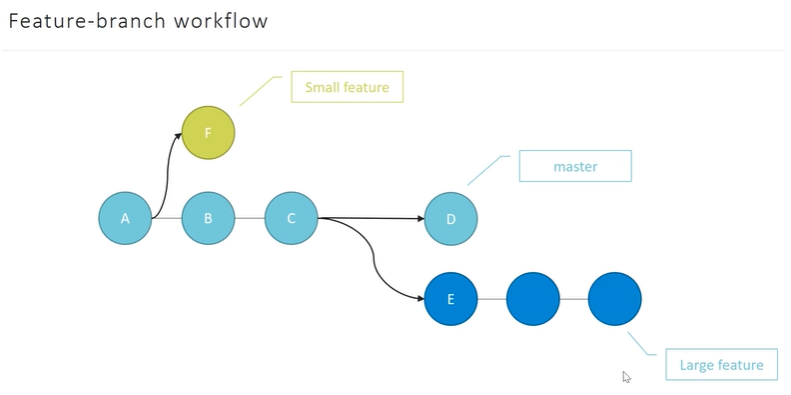
Если по какой-то причине удалился (потерялся) проект в github. Можно создать пустой проект и перенести всё в него.

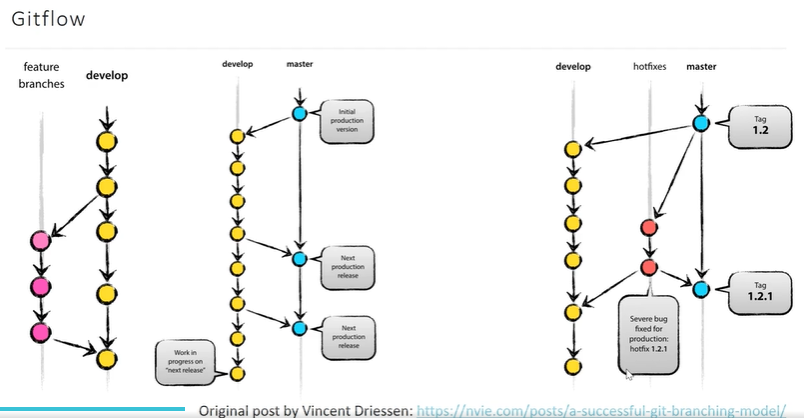


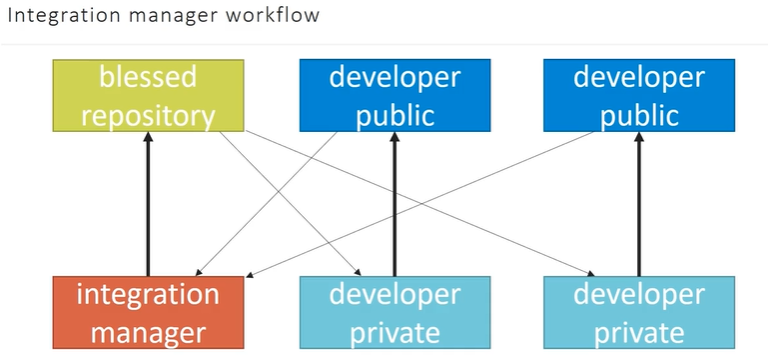


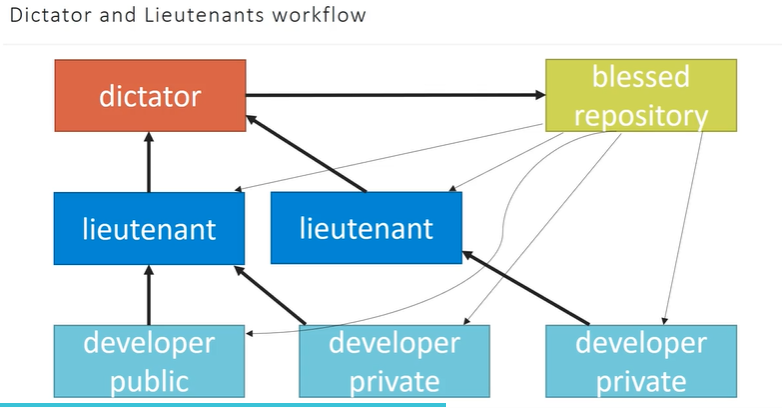
**Branching strategies**

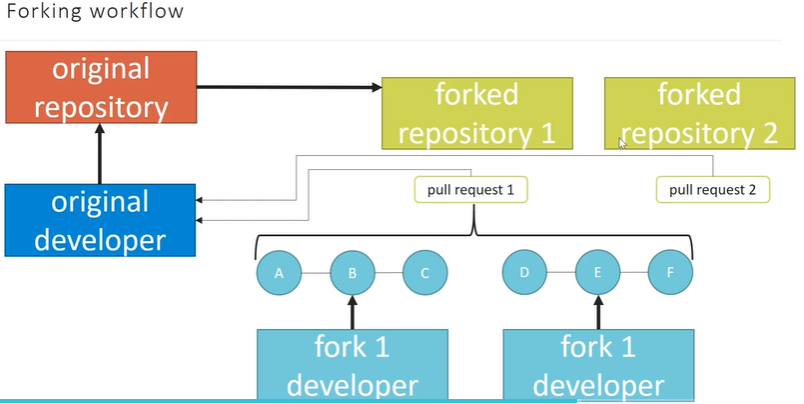




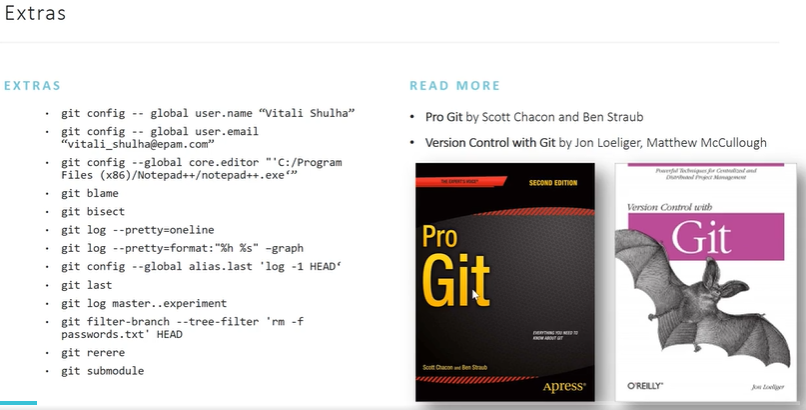








Extras (дополнительно)



Git config – для конфигурации имени и почты

Так же для установки по дефолту какого-то приложения, например notepad++

Git blame - увидеть кто имеет отношение к файлу, кто его правил

Git bisect – делать бинарный поиск, может за 6-7 прыжков найти проблемный участок

Git log -- pretty=oneline – можем посмотреть все коммиты в виде одной строки

Git log -- pretty=format… - можно увидеть некий график

Git log master..experiment – разница между двумя бранчами

Git rerere – можно записать паттерны, и тем самым научить гит фиксить конфликты, если есть узнаваемый паттерн

Git submodule – гит репозиторий внутри гит репозитория

## Полезные ссылки

Презентация тренинга: [DevTestOps-Version\_Control\_with\_Git.pdf](https://elearn.epam.com/assets/courseware/v1/d8b3970d06d567a94e0bb179b8d84acb/asset-v1:EPAM+VCG+RU+type@asset+block/DevTestOps-Version_Control_with_Git.pdf)

Официальный сайт Git: <https://git-scm.com/>

Книга "ProGit", второе издание, на русском: <https://git-scm.com/book/ru/v2>

**Maven (сборщик проектов).**

Скачать от сюда: <https://maven.apache.org/download.cgi> файл в формате zip (н-р.: [apache-maven-3.8.6-bin.zip](https://dlcdn.apache.org/maven/maven-3/3.8.6/binaries/apache-maven-3.8.6-bin.zip).

Распаковать архив в любое место, желательно, чтобы не было русских папок и пробелов в именах папок. Оптимально распаковать в корневой каталог диска С.

Далее нужно сделать изменения в «Изменение системных переменных среды» (Edit the system environment variables) ищем в «Пуске».

На вкладке дополнительно (Advanced) нажать на кнопку «переменные среды».

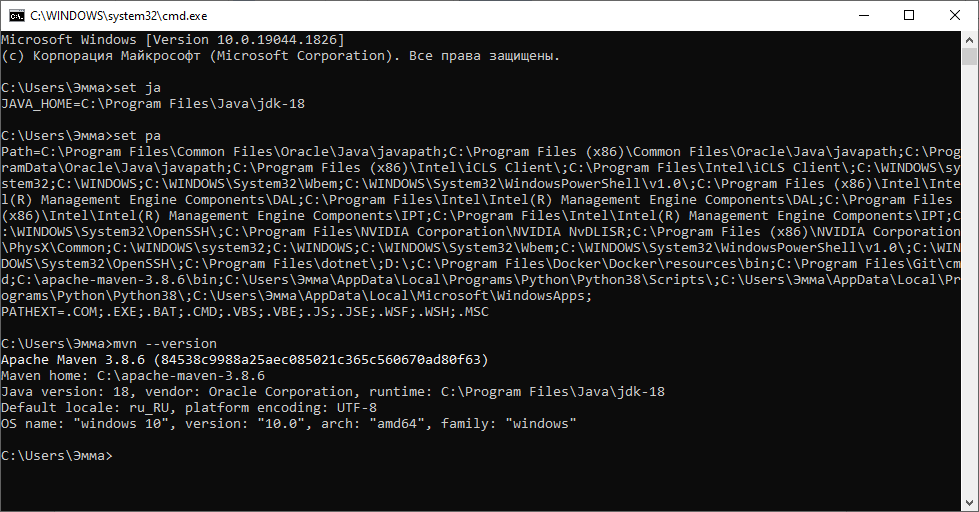
В нижнем окне (системные переменные) нужно добавить 2 переменных.



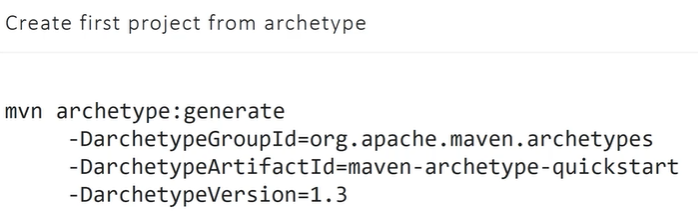
И изменить переменную **Path**, дописав в конце: **;%MAVEN\_HOME%\bin;**

**Если нет Java JDK скачиваем и устанавливаем с инэта.**

Открываем CMD и проверяем: set mave, set ja, set pa, mvn –version. Должны увидеть следующее.



**Create project & import in IDE**



В командной строке переходим в папку где будет у нас проект, вставляем текст ниже и жмём Enter

mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4

Следовать указаниям установщика, задать имя группы и имя папки где будет проект. Затем открыть эту папку через IntelliJ IDEA. Нажать Import Project и выбрать что это Maven.

**Build lifecycle (жизненный цикл сборки)**

Самые используемые команды:

Clean – очищает проект и удаляет файлы которые были сделаны при прошлой сборке

Test – компилирует приложение и запускает unit тесты

Verify - верификация

Package – запаковывает в бинарный артефакт

Install – устанавливает локально в систему

Deploy – развёртывает на некий удалённый сервер.

**mvn test** – скомпилирует Java приложение и запустит Unit тест. (запускается в командной строке находясь в папке с проектом: )

Остальные команды используются аналогично. Можно, например, использовать такую конструкцию:



**Dependency management (Управление зависимостями)**

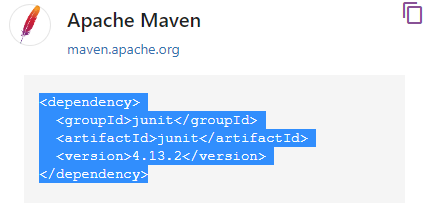
Если нужна какая-то библиотека в нашем проекте делаем следующее:

Чтобы найти нужную библиотеку заходим на search.maven.org

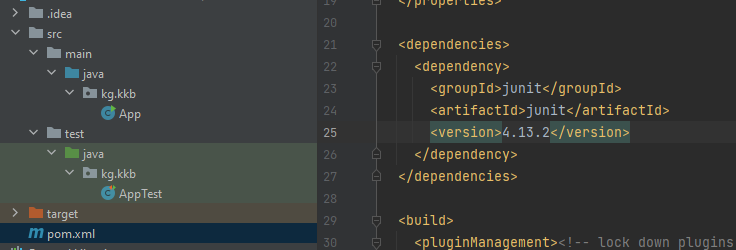
Ищем нужную библиотеку н-р.: junit

Находим нужную или последнюю из библиотек н-р: 

Кликаем на версию и копируем на открывшейся странице текст из раздела Apache Maven



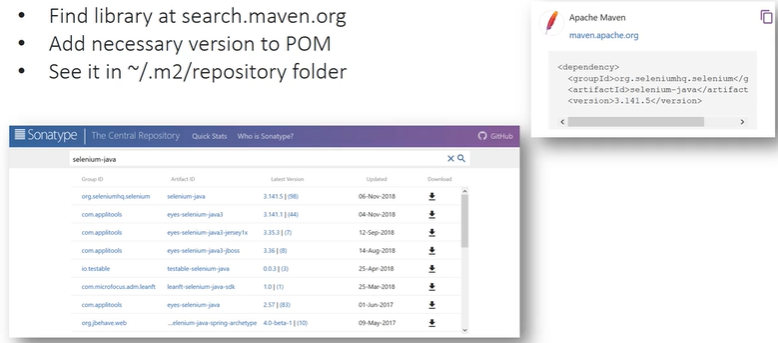
И вставляем его в наш проект, в pom.xml



После этого в Intellij IDEA появится всплывающее окошко, предлагающее установить новый пакет.

А хранятся эти пакеты в папке users/user/.m2/repository. (н-р. C:\Users\Эмма\.m2\repository)

Краткое описание:



Презентация тренинга: [DevTestOps-Maven\_build\_tool.pdf](https://elearn.epam.com/assets/courseware/v1/3237f828d1c1b341e4a0b548389df6b7/asset-v1:EPAM+MBT+RU+type@asset+block/DevTestOps-Maven_build_tool.pdf)

Официальный сайт проекта Maven: <http://maven.apache.org/>

Центральный maven репозиторий для поиска библиотек: <https://search.maven.org/>

**Why need CI & CD (continuous integration & continuous delivery)**

Зачем нужны CI и CD (непрерывная интеграция и непрерывная доставка)

**Концепция continuous integration – Происходит интеграция всей системы. З**акомиченная фича сразу идёт в интеграцию. И тут же идут тесты. Если тесты проходят, всё хорошо, если нет, то разработчику становится известно в каком тесте не прошло и где ошибка. На это уходит в среднем 5-7 минут. Баг находится быстро и его цена низкая. Чем меньше живёт баг, тем дешевле он обходится.

1. Код должен хранится в системе версионного контроля (н-р. Git).
2. Сборка должна быть автоматизирована.
3. Должны быть автоматизированные тесты.
4. Делать коммиты часто. (не раз в неделю, а например каждые 2 часа)
5. Сборка по изменениям. Как появился новый коммит начинается весь **continuous integration.**

Для чего нужно **continuous integration?** Осн. задача иметь в конце собранную протестированную систему в которой есть новое изменение.

**continuous delivery –** когда в самом конце, все тесты прошли ещё и деплой (развёртывание) происходит автоматически. Т.е. если все тесты прошли, обновление уходит пользователям.

Benefits (выгода):

1. Рано обнаруживаем дефекты
2. Имеем возможность автоматически установить приложение, и оно готово для ручного и автоматического тестирования, для ручного деплоя на сервере.
3. Есть прозрачный монитор здоровья проекта, видно проблемы сразу.
4. Есть возможность визуализировать пайплайн (процесс разработки) (цепочка сборок)
5. Распараллеливание некоторых задач. В зависимости от вычислительных мощностей можно расспараллелить задачи и тем самым сократить время.

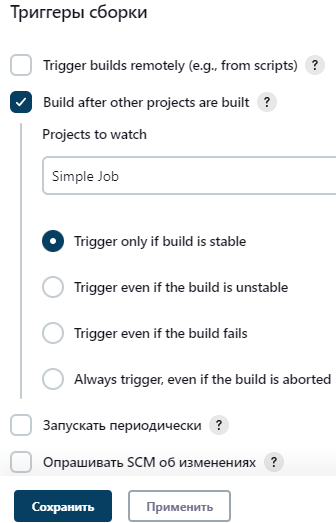
**Jenkins – один из лучших CI серверов.**

**Build triggers.**

Задачи тригерров:

1. Запуск по клику (on-demand run)
2. Вызов одной задачи после завершения другой (other job finish)
3. По расписанию, через Cron (Cron shedule)
4. Посмотреть наличие коммитов в системе версионного контроля, н-р: Git (SCM check)

Например, выбрали 2 вариант. Для этого заходим в настройки нужной задачи и указываем после какой задачи запустить выбранную.



**Синтаксис Cron.**

Сборки или события по расписанию.

