Система контроля версий (Version Control System

Что такое система контроля версий?

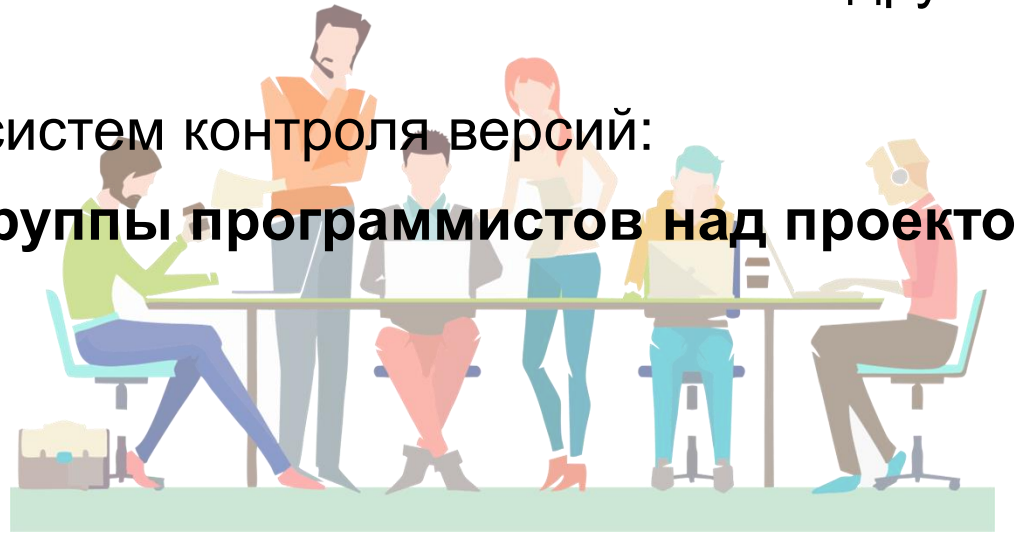
Система контроля версий – это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определенной версии.

Задачи:

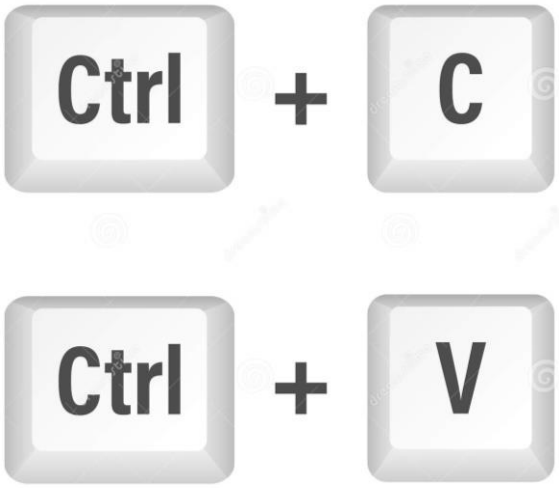
- хранение всех выполненных изменений, возможность посмотреть кем и когда они были выполнены, возможность «откатить» неудачные изменения
- возможность параллельной и независимой работы над данными
- возможность использовать изменения выполненные другими членами команды

Основное применение систем контроля версий:

организация работы группы программистов над проектом



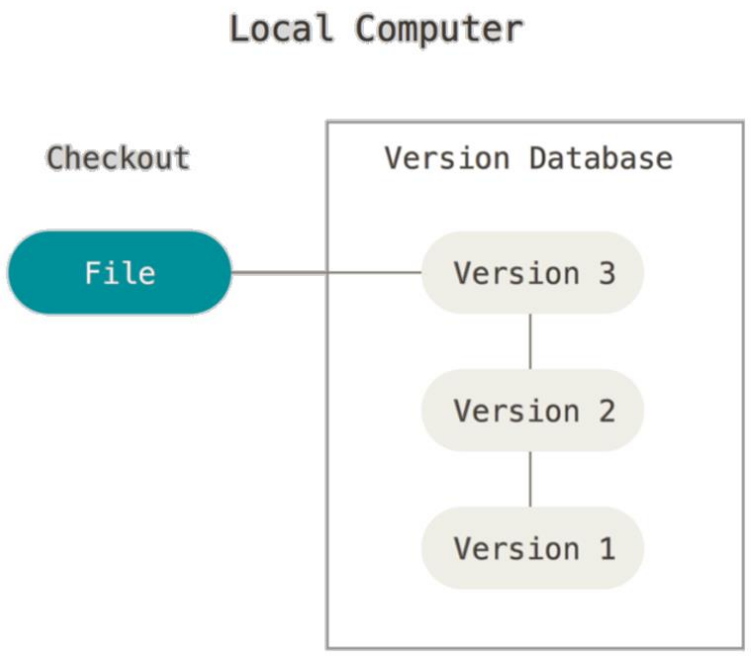
Виды СКВ: **Copy-paste**



Известный метод при применении к данной задаче может выглядеть следующим образом: будем называть файлы по шаблону filename_{version}, возможно с добавлением времени создания или изменения.

Данный способ является очень простым, но он подвержен различным ошибкам: можно случайно изменить не тот файл, можно скопировать не из той директории (ведь именно так переносятся файлы в этой модели).

Локальная система контроля версий

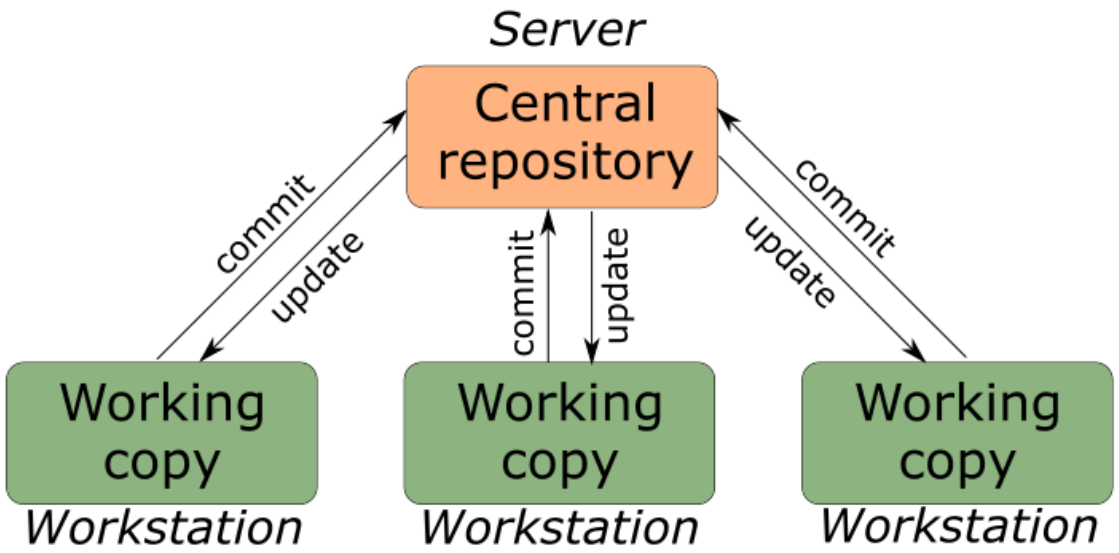


Следующим шагом в развитии систем контроля - **локальные системы контроля версий**. Они представляли из себя простейшую базу данных, которая хранит записи обо всех изменениях в файлах.

Одним из примеров таких систем является система контроля версий RCS, которая была разработана в 1985 году (последний патч был написан в 2015 году) и хранит изменений в файлах (патчи), осуществляя контроль версий. Набор этих изменений позволяет восстановить любое состояние файла. RCS поставляется с Linux'ом.

Локальная система контроля версий хорошо решает поставленную перед ней задачу, однако ее проблемой является основное свойство — локальность. Она совершенно не предназначена для коллективного использования.

Централизованная система контроля версий

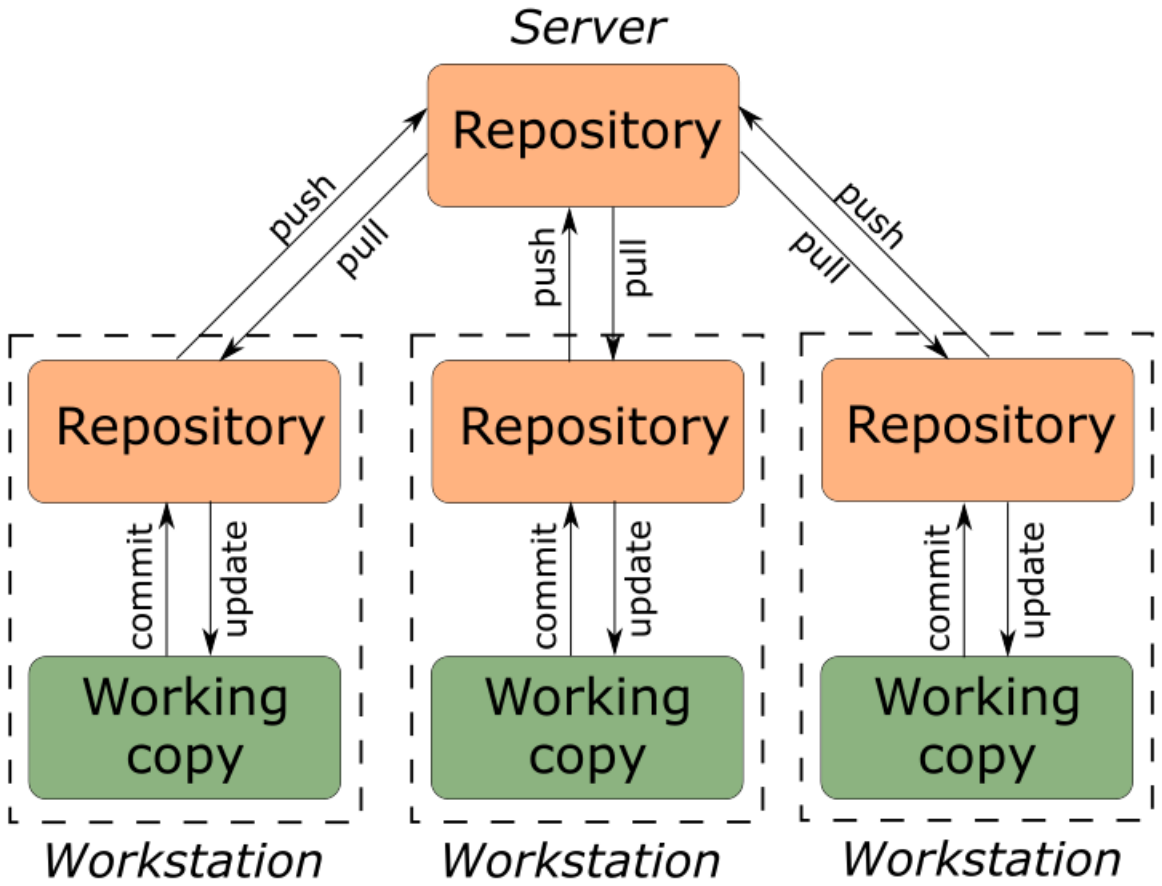


Централизованная система контроля версий предназначена решает вышеописанную проблему.

Для организации такой системы контроля версий используется **единственный сервер**, который содержит **все версии файлов**. Клиенты, обращаясь к этому серверу, получают их из этого централизованного хранилища. На протяжении многих лет являлись стандартом. К ним относятся CVS, Subversion, Perforce.

Такими системами легко управлять из-за наличия единственного сервера. Но это приводит к возникновению единой точки отказа в виде этого самого сервера. В случае отключения нельзя будет выкачивать файлы. В худшем сценарии (физическое уничтожение сервера или вылет жесткого диска) – полная потеря данных.

Распределенная система контроля версий



Для устранения единой точки отказа используются **распределенные системы контроля версий**. Они подразумевают, что клиент **выкачает себе весь репозиторий целиком** заместо выкачки конкретных интересующих клиента файлов. Если умрет любая копия репозитория, то это не приведет к потере кодовой базы, поскольку она может быть восстановлена с компьютера любого разработчика. Каждая копия является полным бэкапом данных.

Все копии являются равноправным и могут синхронизироваться между собой.

К данному виду систем контроля версий относятся Mercurial, Bazaar, Darcs и Git. Последняя система контроля версий и будет рассмотрена нами далее более детально.

Краткая история Git

Как и многие вещи в жизни, Git начинался с капелькой творческого хаоса и бурных споров.

Ядро **Linux** — это достаточно большой проект с открытым исходным кодом. Большую часть времени разработки ядра Linux (1991–2002 гг.) изменения передавались между разработчиками в виде патчей и архивов. В 2002 году проект ядра Linux начал использовать проприетарную децентрализованную СКВ BitKeeper.

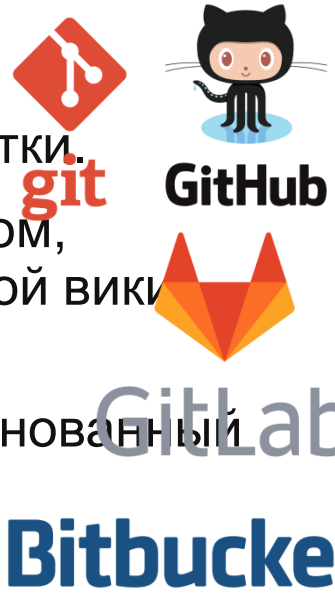
В 2005 году отношения между сообществом разработчиков ядра Linux и коммерческой компанией, которая разрабатывала BitKeeper, прекратились, и бесплатное использование утилиты стало невозможным. Это сподвигло сообщество разработчиков ядра Linux (а в частности Линуса Торвальдса — создателя Linux) разработать свою собственную утилиту, учитывая уроки, полученные при работе с BitKeeper. Некоторыми **целями, которые преследовала новая система, были**:

- Скорость
- Простая архитектура
- Хорошая поддержка нелинейной разработки (тысячи параллельных веток)
- Полная децентрализация

Возможность эффективного управления большими проектами, такими как ядро Linux (скорость работы и разумное использование дискового пространства).

Git - крайне полезный инструмент при командной разработке. Возможность вносить изменения в репозиторий нелинейно (независимые изменения в разных ветках) позволяет организовать коллективную работу над проектом. Каждый участник разработки имеет собственную копию репозитория (*локальный репозиторий, local repository*), в которой может вести работу независимо от остальных. Для синхронизации внесенных изменений используется общая копия репозитория (*удаленный репозиторий, remote repository*), которая размещается на отдельном сервере (GitHub, GitLab, Bitbucket, собственный сервер и т.д.).

Какие бывают удаленные репозитории?



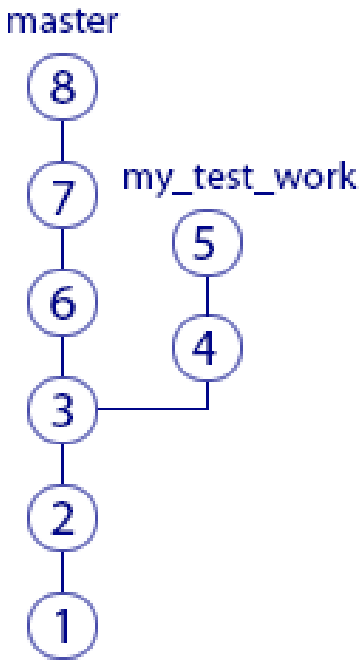
- **GitHub** — это крупнейшее хранилище для репозиториев и совместной разработки.
- **GitLab** — веб-инструмент жизненного цикла DevOps с открытым исходным кодом, представляющий систему управления репозиториями кода для Git с собственной вики-системой отслеживания ошибок, CI/CD пайплайн и другими функциями.
- **BitBucket** — веб-сервис для хостинга проектов и их совместной разработки, основанный на системе контроля версий Mercurial и Git.
- И другие (зеркало/национальный репозиторий)

Наличие удаленного репозитория может быть полезным и при одиночной разработке: оно позволяет синхронизировать состояние проекта на разных компьютерах и просто сохранить проект на внешнем сервере.

Основные термины

- ***Репозиторий (repository)*** — совокупность файлов, состояние которых отслеживается, и история их изменений.
- ***Коммит (commit)*** — сохраненное состояние (версия) файлов репозитория.
- ***Ветка (branch)*** — последовательность коммитов (история изменения состояния репозитория). Каждый коммит в ветке имеет «родителя» (*parent commit*) — коммит, на основе которого был получен текущий. В репозитории может быть несколько веток (в случаях, когда к одной версии репозитория применяется несколько независимых изменений).
- ***Мердж (слияние, merge)*** — объединение двух или более веток. В процессе мерджа изменения с указанной ветки переносятся (копируются) в текущую.
- ***Мастер (master, main)*** — основная ветка репозитория, создается автоматически при создании репозитория.
- ***Мердж коммит (merge commit)*** — коммит, который создается автоматически по завершению процесса слияния веток. Мердж коммит содержит в себе все изменения целевой ветки мерджа, которые отсутствуют в текущей (все коммиты целевой ветки, которые начиная с базы слияния, но не включая её).
- ***Мердж конфликт (merge conflict)*** — ситуация, когда при слиянии веток в один или несколько файлов вносились независимые изменения. В некоторых случаях (например, если изменялись разные, не пересекающиеся части одного файла) git способен самостоятельно решить, как выполнять слияние таких файлов. Если автоматически это сделать не удалось — возникает конфликт. В таком случае необходимо самостоятельно указать, как выполнять слияние конфликтующих версий (*решить конфликт, resolve merge conflict*). Изменения, внесенные в процессе решения конфликта автоматически попадают в мердж коммит.

Ветки (branch)



Представьте, что вам нужно внести множество изменений в файлы вашего рабочего каталога, но эта работа **экспериментальная** – не факт, что всё получится хорошо. Вы бы не хотели, чтобы ваши изменения увидели другие сотрудники до тех пор, пока работа не будет закончена. Может просто ничего не коммитить до тех пор? **Это плохой вариант.** ***Частые коммиты и пуши – залог сохранности вашей работы,*** а также возможность посмотреть историю изменений. К счастью, в Git есть механизм веток, который позволит нам коммитить и пушить, но не мешать другим сотрудникам.

Перед началом экспериментальных изменений вы должны создать **ветку**. У ветки есть имя. Пусть она будет называться ***my test work***. Теперь все ваши коммиты будут идти именно туда. До этого они шли в основную ветку разработки – будем называть её master. Другими словами, раньше вы были в ветке master (хоть и не знали этого), а сейчас переключились на ветку my test work.

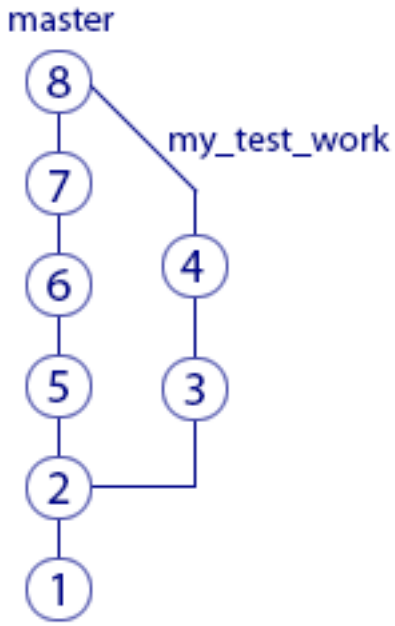
После коммита «3» создана ветка и ваши новые коммиты «4»и «5» пошли в неё. А ваши коллеги остались в ветке master, поэтому их новые коммиты «6», «7», «8» добавляются в ветку master. История перестала быть линейной.

На что это повлияло? Сотрудники теперь не видят изменений файлов, которые вы делаете. А вы не видите их изменений в своих рабочих файлах. Хотя

Слияние веток (merge)

Теперь мы знаем, что каждый может создать ветки и работать независимо. Можно по очереди работать то в одной ветке, то в другой – переключаясь между ними. Ветки переключает команда **checkout**.

Но вернёмся к примеру с вашей экспериментальной работой. Допустим мы решили, что она не удалась. Вы вернулись в ветку ***master*** и потеряли изменения, сделанные в ветке my test work. А если все получилось? Вы хотите перенести свои изменения в ветку ***master***, чтобы их увидели сотрудники, которые с ней работают. **Git** может помочь – выполним команду **merge** ветки ***my test work*** в ветку

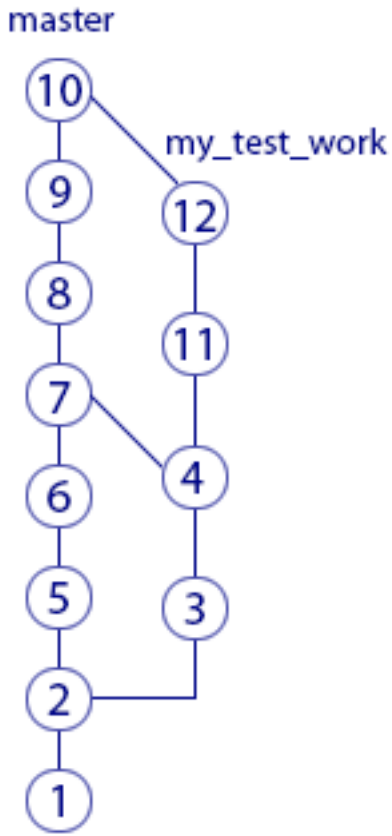


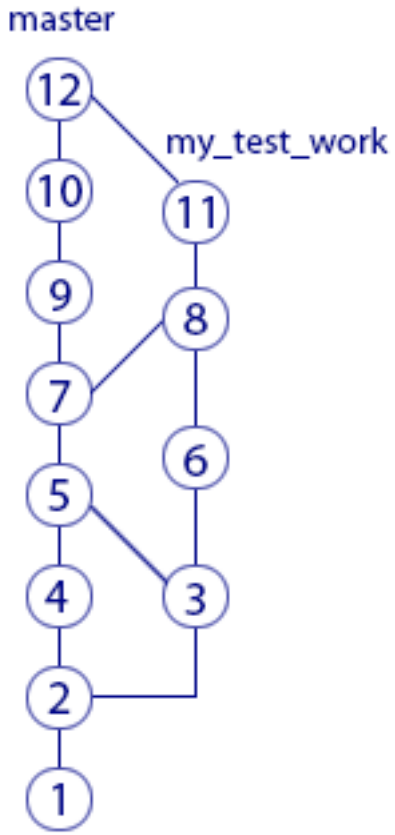
Здесь коммит «8» – это **merge-commit**. Когда мы выполняем команду merge, система сама создает этот коммит. В нём объединены изменения ваших коллег из коммитов «5», «6», «7», а также ваша работа из коммитов «3», «4».

Изменения из коммитов «1» и «2» объединять не нужно, ведь они были сделаны до создания ветки. А значит изначально были и в ветке ***master***, и в ветке ***my test work***.

Команда **merge** ничего не посылает в ***origin***. Единственный ее результат – это **merge-commit** (на рисунке кружок с номером 8), который появится у вас на компьютере. Его нужно запушить, как и ваши обычные коммиты. Только после этого **merge-commit** отправится на ***origin*** – тогда коллеги увидят результат вашей работы, сделав **pull**.

Несколько слияний



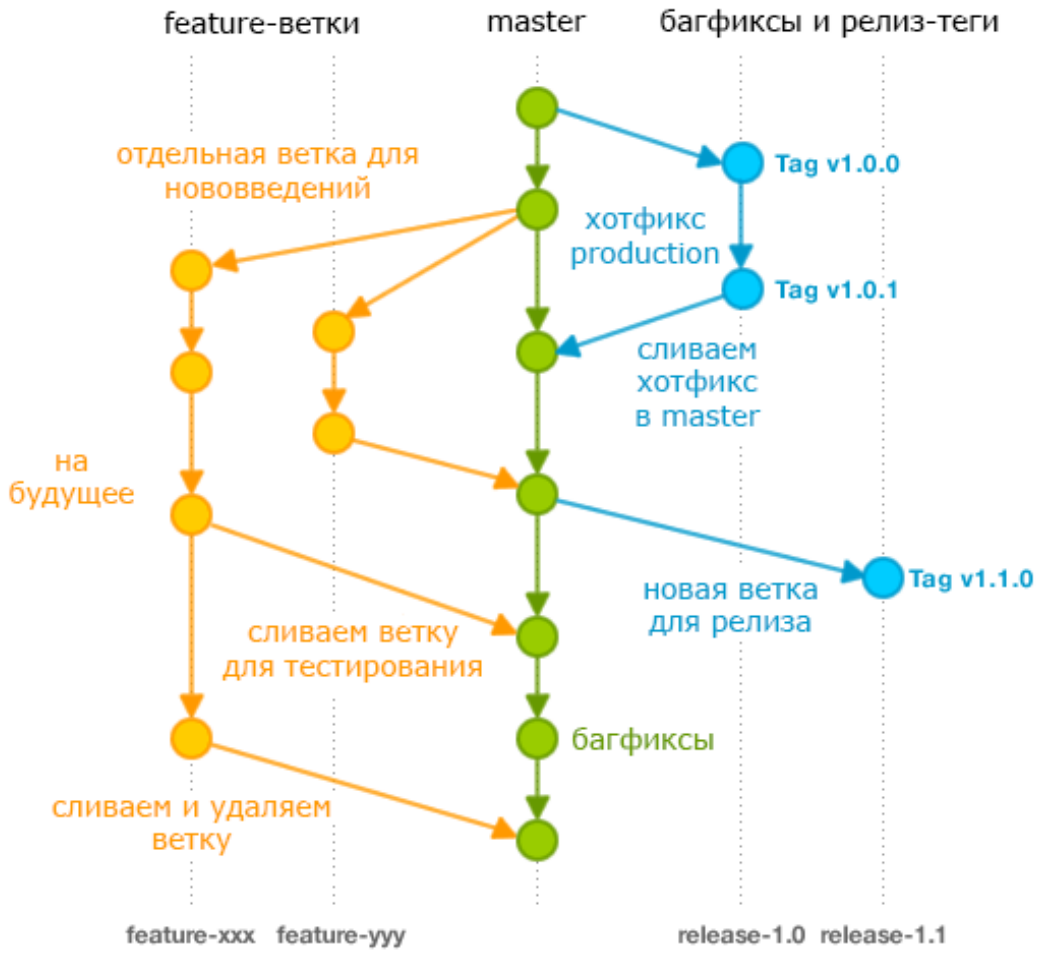


Мы уже узнали, как сделать новую ветку, поработать в ней и залить изменения в главную ветку. На картинке после объединения ветки слились вместе. Означает ли это, что в ветке ***my test work*** теперь работать нельзя – она ведь уже объединилась с ***master***? Нет, вы можете продолжать коммитить в ветку ***my test work*** и периодически мержить в главную ветку.

Можно ли мержить в обратную сторону и есть ли в этом смысл? Можно. Есть.

Если вы долго работаете в своей ветке, рекомендуется периодически делать мерж в неё из главной ветки. Это необходимо, чтобы вы работали с актуальными версиями файлов, которые меняют другие люди.

Коммиты и их хеши, теги



Как Git различает коммиты? На картинках мы для простоты помечали их порядковыми номерами (или буквами). На самом деле каждый коммит в Git обозначается вот такой строкой:

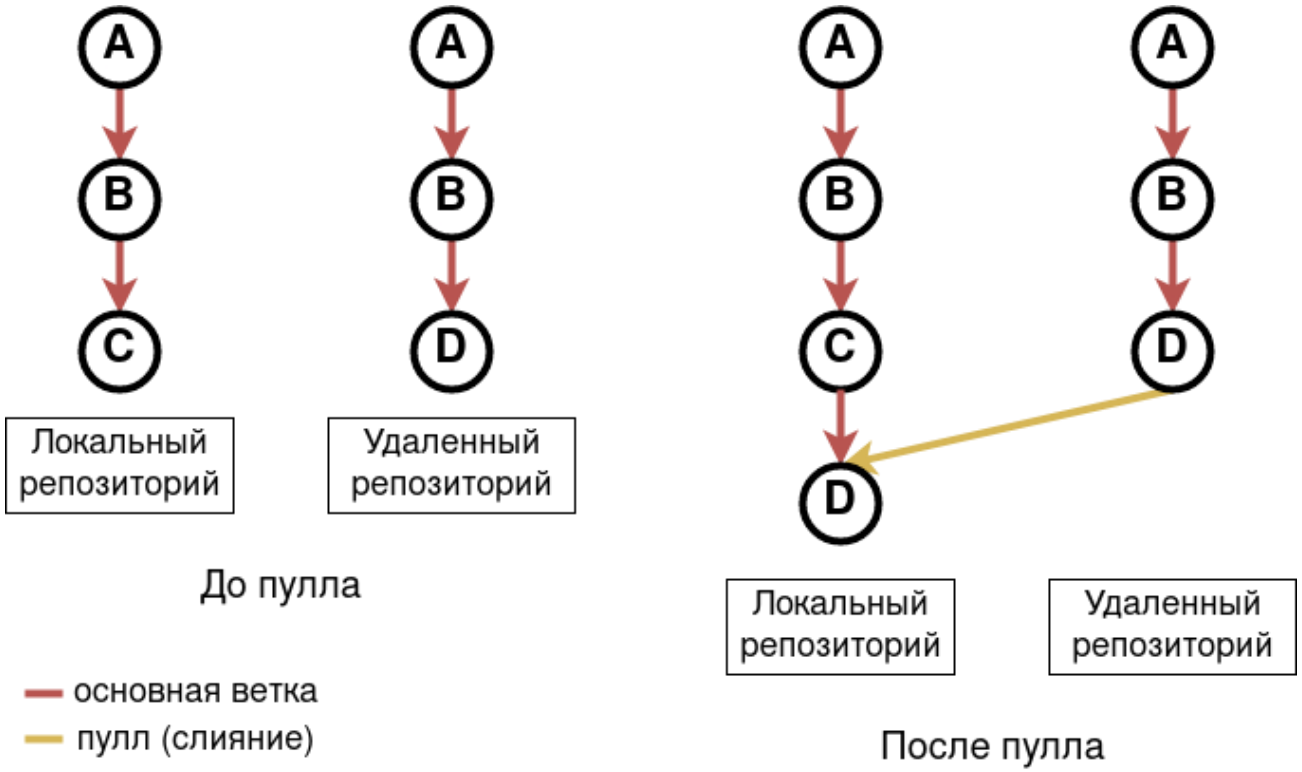
e09844739f6f355e169f701a5b7ae02c214d5fb0

Это «названия» коммитов, которые Git автоматически даёт им при создании. Вообще, такие строки принято называть «хеш». У каждого коммита хеш разный. Если вы хотите кому-то сообщить об определённом коммите, можно отправить человеку хеш этого коммита. Зная хеш, он сможет найти этот коммит (если это ваш коммит, то, конечно, его надо сначала запушить).

Теги (Tags) — это ссылки, указывающие на определенные точки в истории Git. **Команда git tag** обычно используется для захвата некой точки в истории, которая используется для релиза нумерованной версии (например, v1.0.1). Теги похожи на неизменяемые ветки, но они, в отличие от веток, не имеют истории коммитов после создания.

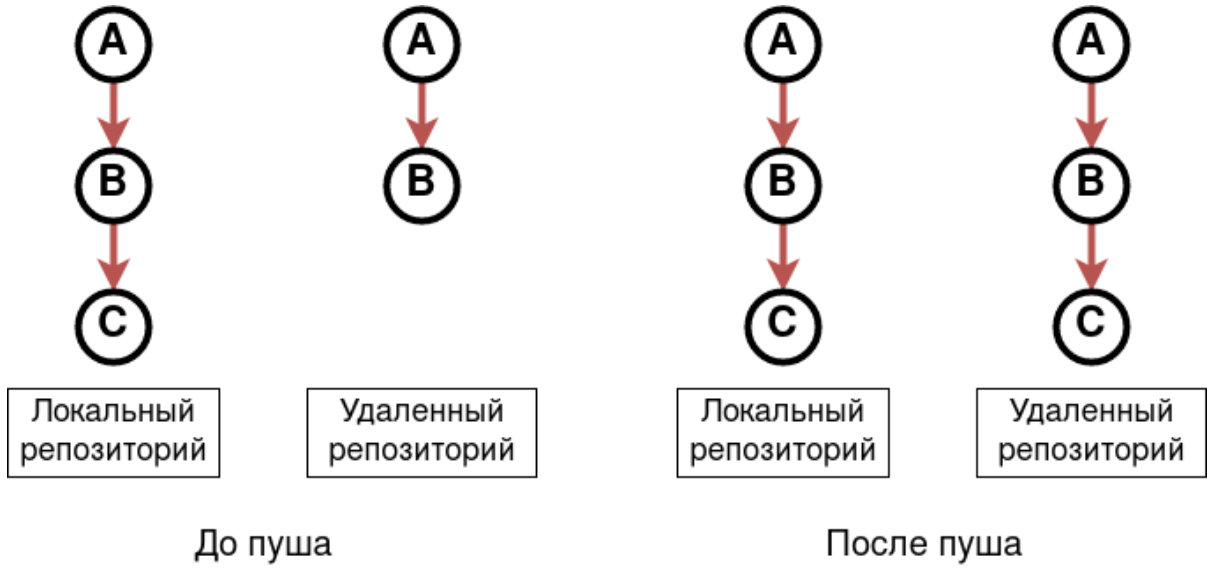
Выше мы часто упоминали **пуши (push)**, так что же это такое? Есть два варианта синхронизации изменений проекта.

пулл (pull)



пулл (pull) — слияние состояния удаленного репозитория и локального (обычно — в отдельной ветке). **Пулл** может выполняться как для одной и той же ветки (с одинаковым именем), так и для разных. **Пулл** являет собою обычный **мердж**, но целевая ветка при этом находится не в том же репозитории, в котором выполняется **пулл**, а в удаленном. Как следствие, при **пулле** так же создается **мердж коммит**, пулл можно **отменить (заревертить)** и в его процессе может возникнуть **мердж конфликт**.

пуш (push)



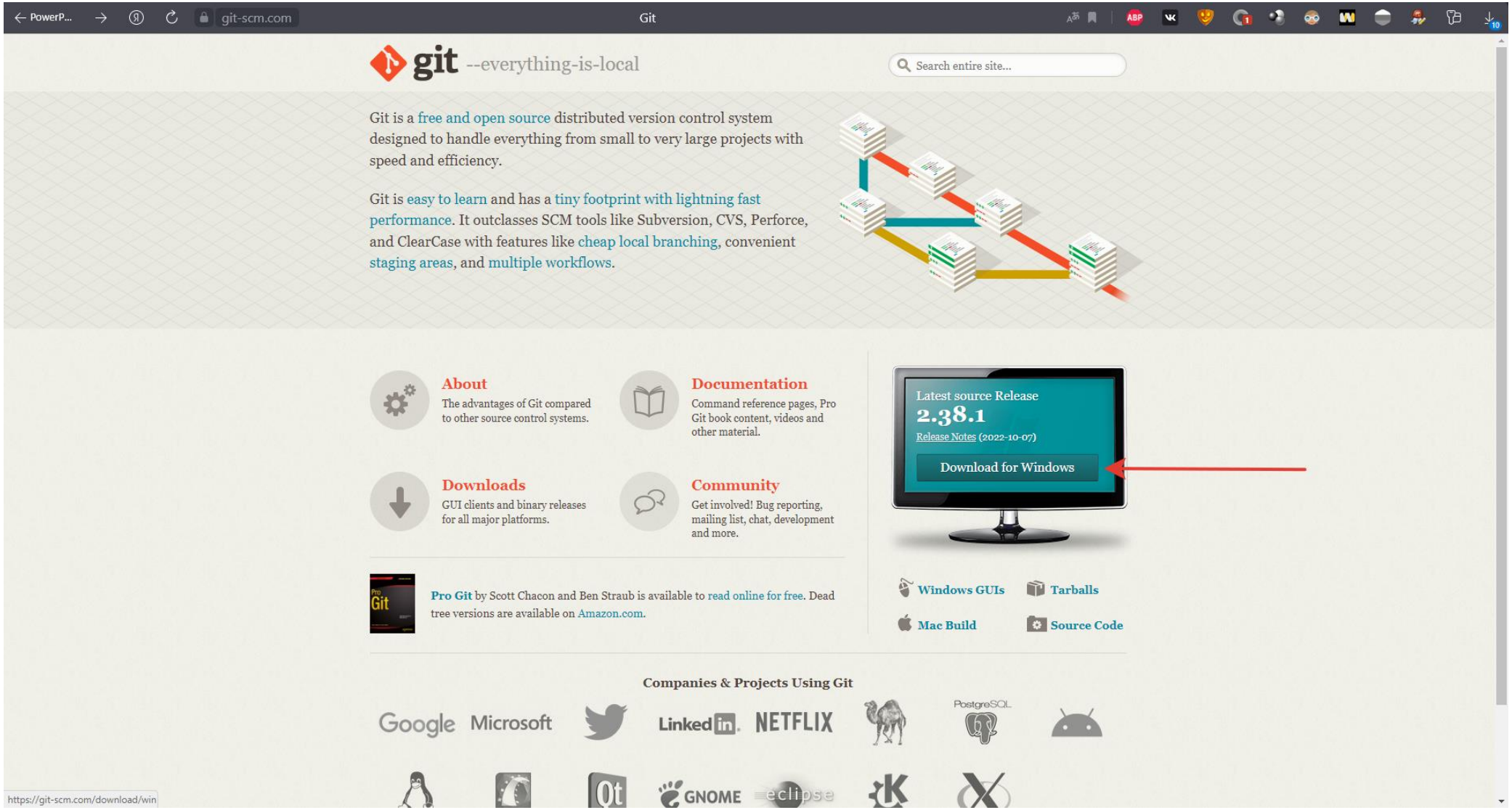
пуш (push) — обратный **пуллу** процесс. При пуше изменения из локального репозитория переносятся в удаленный. Пуш обновляет состояние текущей ветки в удаленном репозитории и **не является мерджем** (не создает дополнительные коммиты и не может привести к конфликтам). Если в ветке удаленного репозитория присутствуют коммиты, которых нет в локальном репозитории, сигнализируется ошибка о несовпадении истории изменений (***non fast-forward merge)***, пуш выполнить не получится. В таком случае необходимо сначала синхронизировать состояние локального репозитория (получить недостающие коммиты с помощью пулла), и только после этого повторить процесс пуша.

Нередко возникает необходимость обновить информацию о состоянии удаленного репозитория (существующих ветках и коммитах в них) без выполнения **слияния (пулла).** Такой процесс называется ***фетчем (fetch)***. Таким образом, **пулл** является комбинаций **фетча** и **мерджа**: сперва обновляется информация о состоянии целевой ветки в удаленном репозитории, а затем ее изменения вливаются в текущую ветку в локальном репозитории.

Лабораторная работа

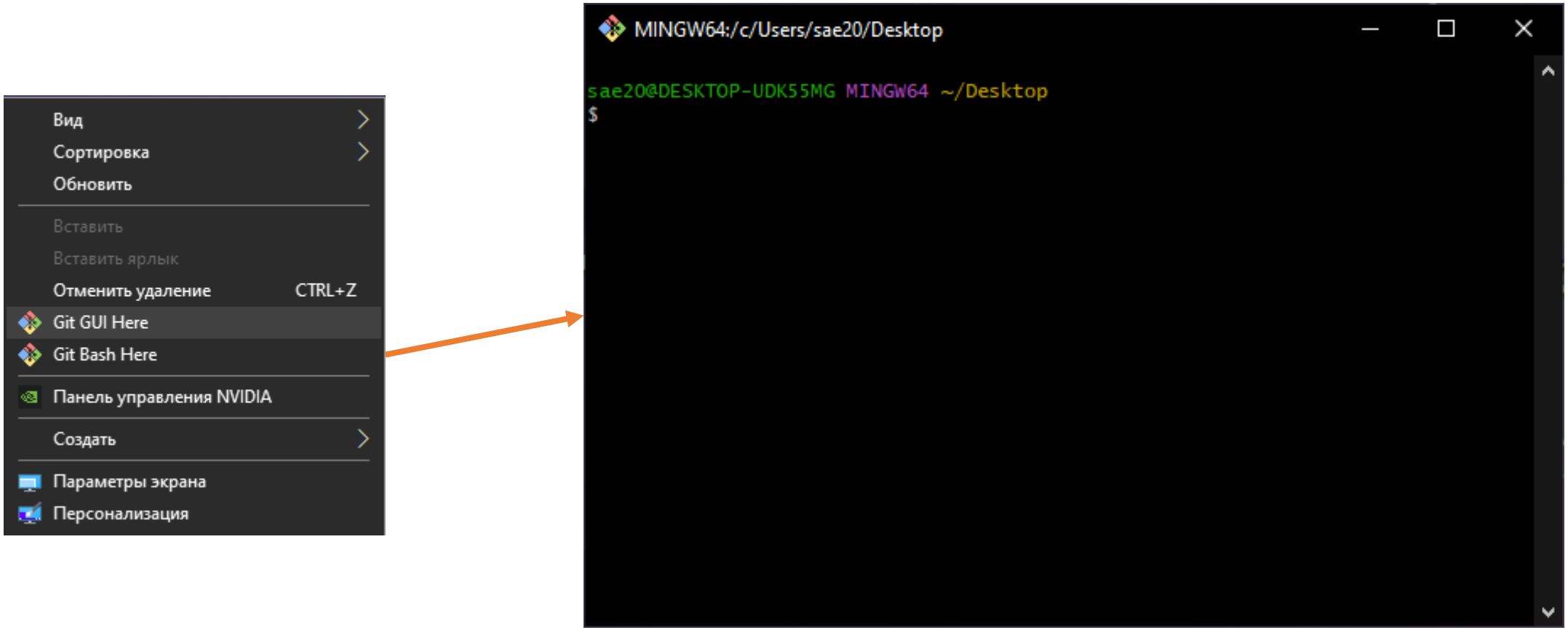
- установить СКВ
- создать репозиторий, выполнить операции: добавление файлов и папок в репозиторий, создание версий файлов (подробнее в задании).
- залить ваши лабораторные в репозиторий https://github.com/OrangeRedeng/Spring_2024.git

Установка



Официальный сайт: http://git-scm.com/ (установочные параметры стандартные)

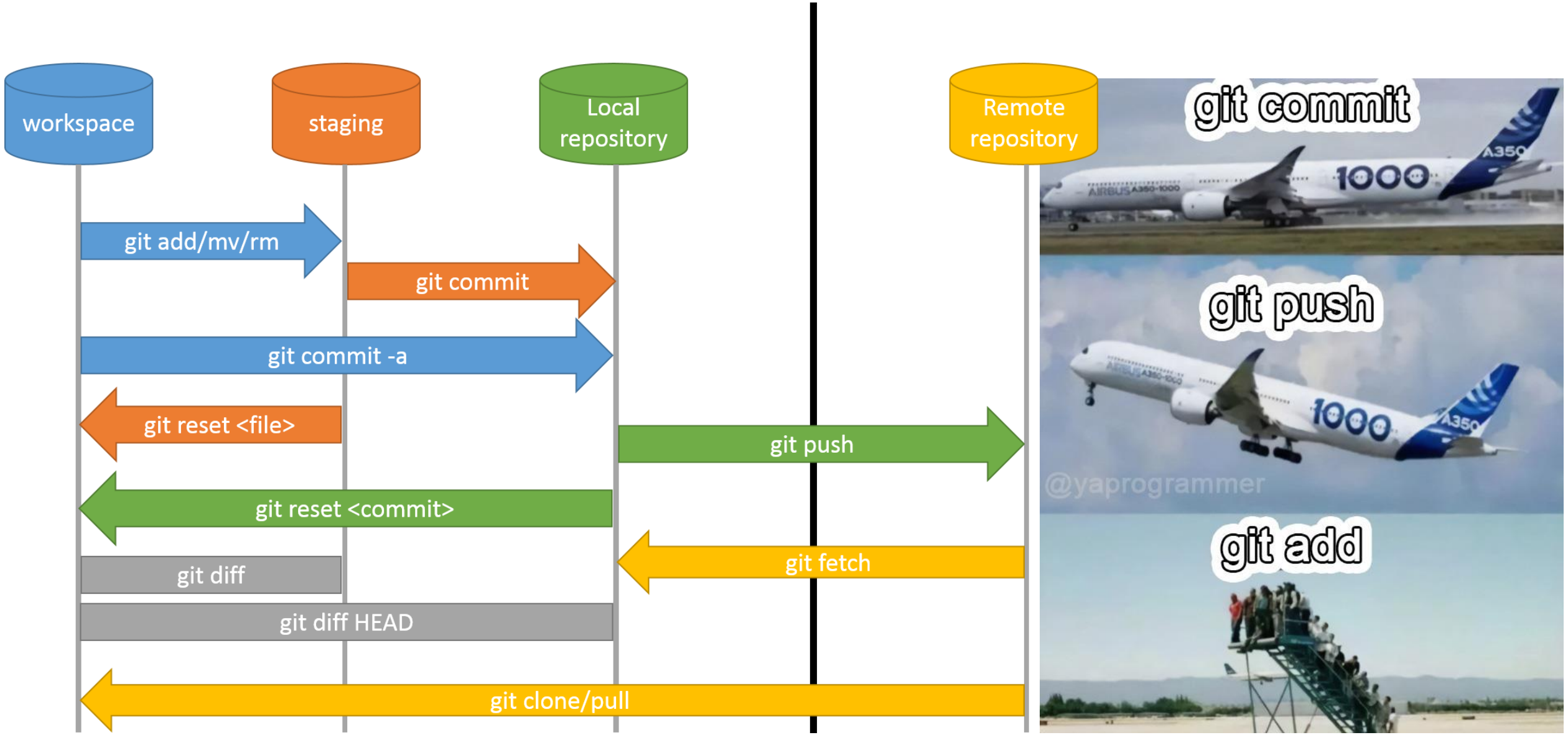
Командная строка



Навигация по файловой системе: команда **cd**(использование: cd<путь к папке>)
Посмотреть содержимое папки: команда **ls**
Создать папку: команда **mkdir** (использование: mkdir<имя папки>)
Создать файл: команда **touch** (использование: touch<имя файла>)

Основные команды

git clone – клонирование репозитория
git status – проверка статуса репозитория
git add – добавление изменений из индекса
git reset – удаление изменений из индекса
git commit – зафиксировать в коммите проиндексированные изменения
git tag – создать тег с указанным именем на коммите
git branch – показать список веток
git branch <name of new branch> - создать новую ветку
git checkout <name of branch> - переключиться на ветку
git merge – слияние веток
git pull – влить изменения в локальный репозиторий
git push – влить изменения в удаленный репозиторий


workspace
staging
Local repository
Remote repository
git add/mv/rm
git commit
git commit -a
git reset <file>
git push
git reset <commit>
git fetch
git diff
git diff HEAD
git clone/pull
git commit
1000
A350
git push
1000
A350
@yaprogrammer
git add

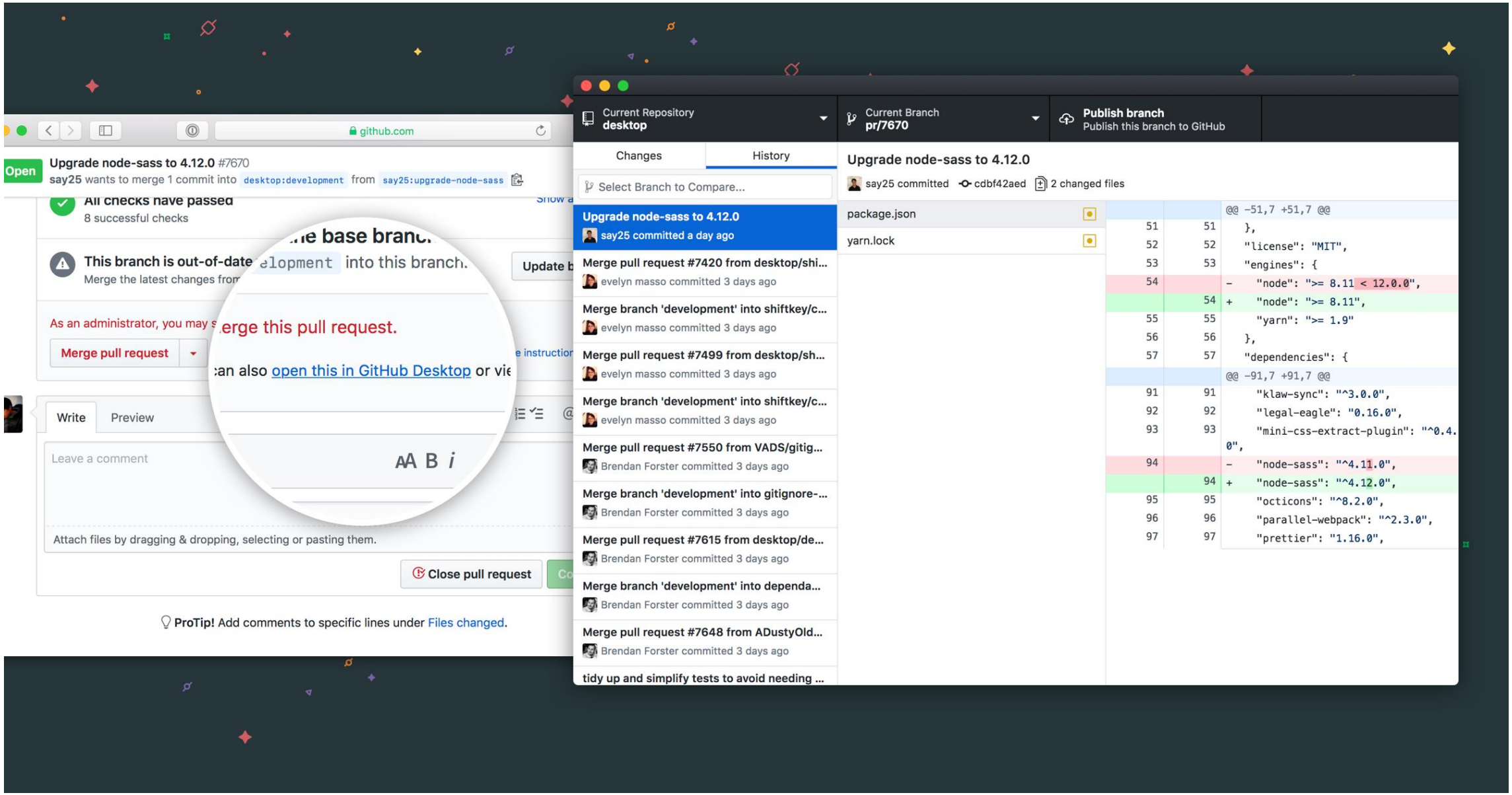
.git & .gitignore

- Все изменения **git** сохраняются в системной папке **.git**, находящейся в корне репозитория, и ее лучше не трогать
- Мы хотим хранить только исходный код и ничего другого в репозитории. А что может быть еще? Как минимум, скомпилированные классы и/или файлы, которые создают среды разработки. Чтобы гит их игнорировал, есть специальный файл, который нужно создать. Делаем это: создаем файл в корне проекта с названием **.gitignore**, и в этом файле каждая строка будет шаблоном для игнорирования. Пример:

```
*.class
target/
*.iml
.idea/
```

- первая строка — это игнорирование всех файлов с расширением .class;
- вторая строка — это игнорирование папки target и всего, что она содержит;
- третья строка — это игнорирование всех файлов с расширением .iml;
- четвертая строка — это игнорирование папки .idea.

GitHub Desktop



Полезные ссылки

- https://habr.com/ru/sandbox/156522/ - основные понятия
- https://habr.com/ru/post/541258/ - Git для новичков
- https://github.com/cyberspacedk/Git-commands - шпаргалка по командам git