

# Воркшоп по git



Команда миграции и  
внедрения бизнес  
процессов



# git

Зачем вообще нужен  
этот ваш git?





Представим себе программиста *Геннадия*. Гена уже несколько месяцев работает над большим проектом. У заказчиков постоянно появляются новые требования, а сам Гена регулярно находит новые баги.

Перед каждым большим изменением Гена делает полную резервную копию всего кода:

Перед изменением я копирую проект в новую папку **Мой проект v.0.xxx** и вношу изменения.

Так я не боюсь что-то поломать, ведь я всегда могу откатиться на предыдущую версию.

– *Геннадий*





Через пару месяцев работы над проектом Гена понимает, что следить за изменениями кода сложно:

- *Гена не помнит, что менялось от версии к версии.*

Гена начинает записывать изменения в блокнот, но и это скоро перестаёт помогать:

- *Записи в блокноте никак не связаны с кодом, и Гене неудобно сравнивать код двух разных версий.*

В конце концов Гена перестает справляться с проектом и ему выделяют стажера Валеру.

Валера начинает создавать свои копии папок с версиями и вести свой блокнотик...





**ГЕНА**



## Как Гене решить его проблемы?

Геннадий устал от того ада, в который превратилась его работа с кодом, и начал искать решение своей проблемы в интернете.

Так Гена узнал про *системы контроля версий*.





# Что такое СКВ?

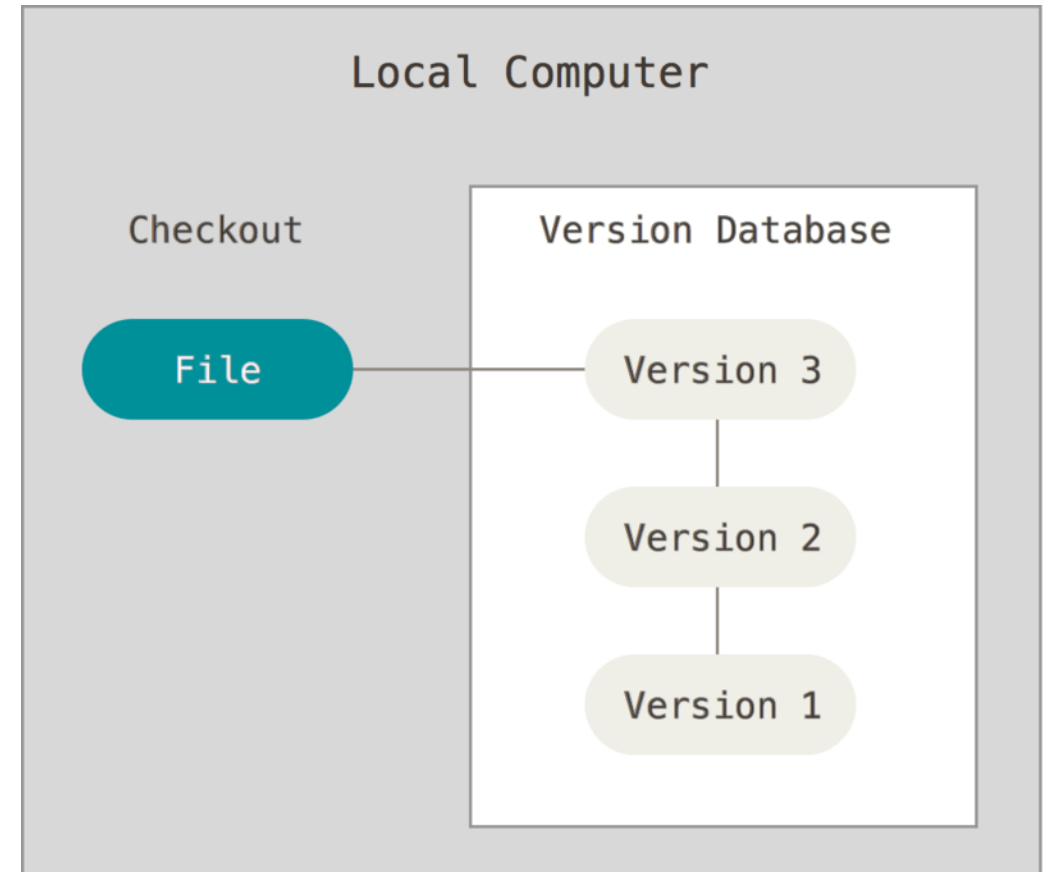
Системы контроля версий (сокращенно СКВ) — это инструменты, которые помогают разработчикам ориентироваться в коде и отслеживать изменения.

СКВ могут "откатить" отдельные файлы в исходное состояние или показать вам только те изменения, которые вы сделали за определённое время.

Сейчас есть три принципиальных модели реализации СКВ:  
**локальная, централизованная и распределенная.**

## Локальная СКВ

Локальная СКВ – это самый простой вид СКВ. Все изменения и резервные копии хранятся у вас на компьютере в специальной базе данных.

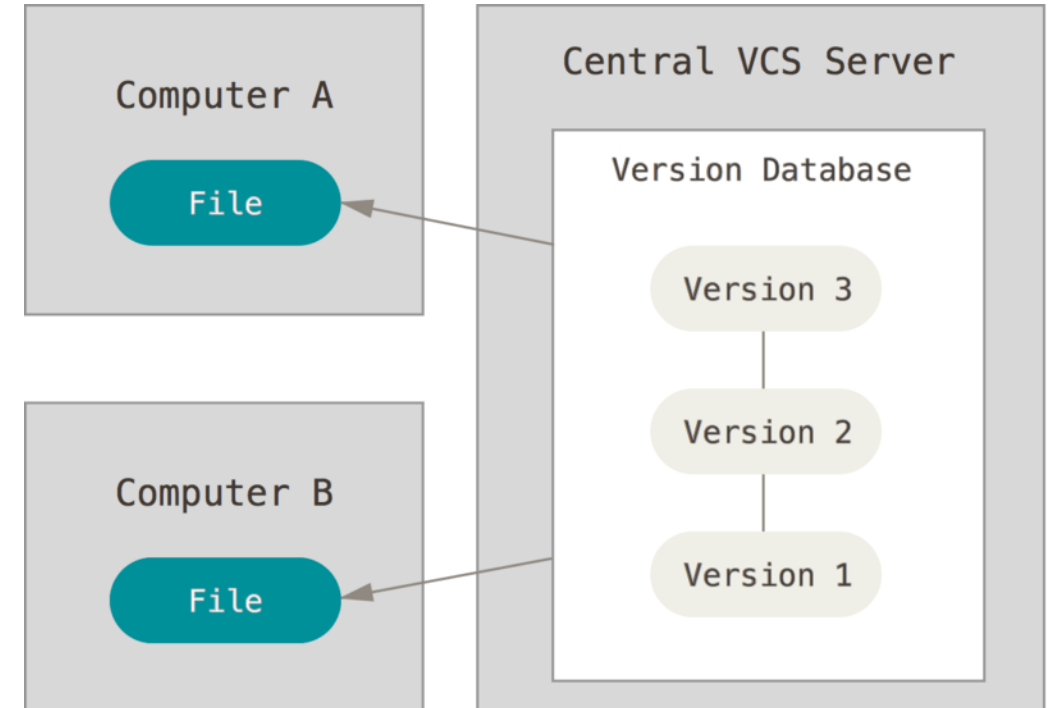


# Централизованная СКВ

Централизованные СКВ придумали, чтобы несколько программистов могли работать над одним проектом не мешая друг другу.

Такая система контроля версий состоит из:

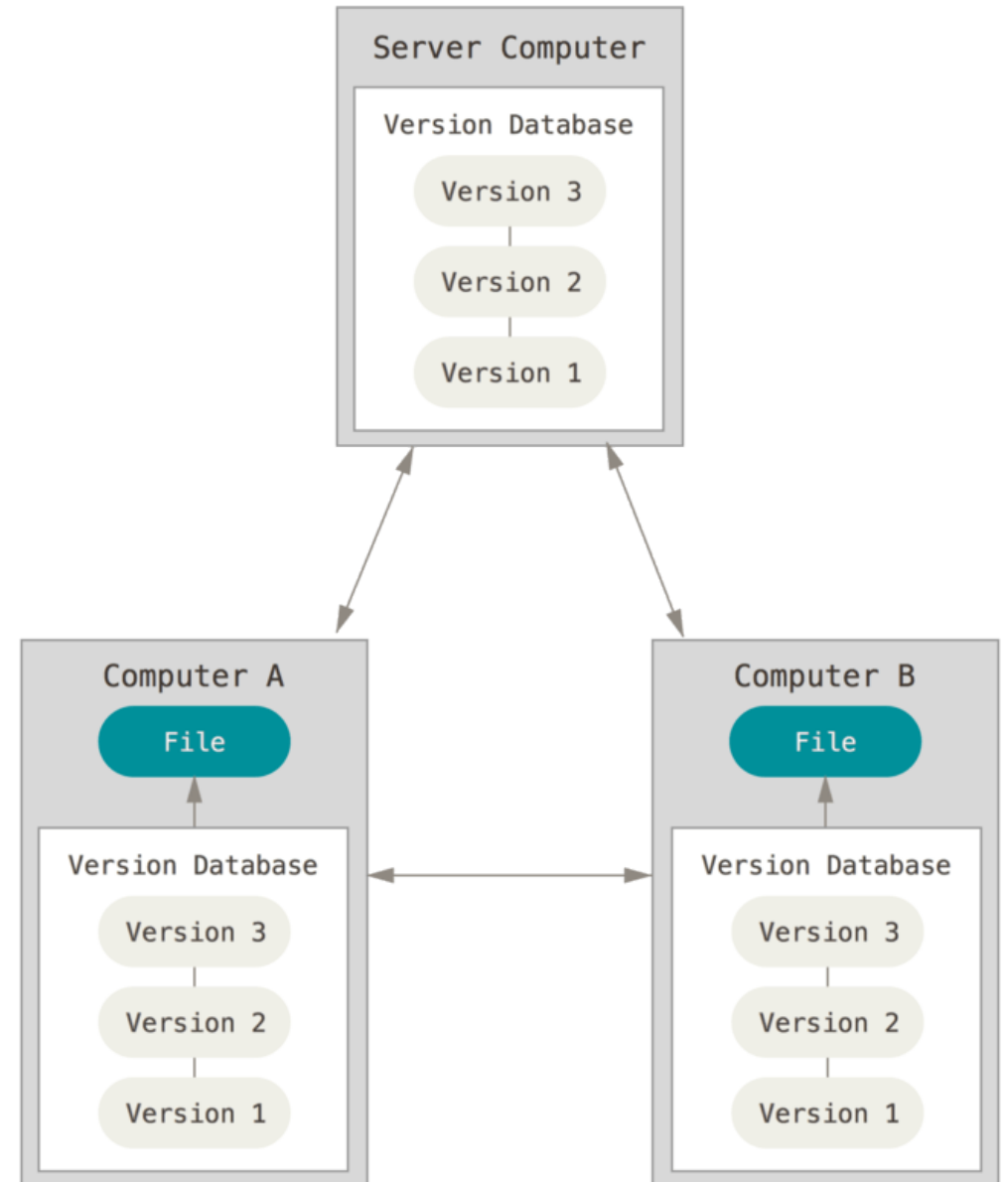
- сервера с репозиториями
- рабочих станций, которые подключаются к серверу для работы над файлами.



# Распределенная СКВ

В распределенной СКВ каждый разработчик получает не просто несколько нужных для работы файлов, а весь репозиторий целиком.

Если сервер выйдет из строя, он просто возьмёт полный репозиторий у любого программиста и скачает его себе.



## Что же такое git?

**Git** – это распределенная система контроля версий. **Git** повсеместно используется разработчиками как инструмент для совместной работы над кодом.

Основное преимущество **Git** – скорость работы, простота и работа с большими проектами. В отличие от других систем контроля версий, **Git** не записывает изменения к каждому файлу, а как бы фотографирует весь проект целиком.

Не путайте с **GitHub** – это онлайн-сервис, который основан на технологии **Git**. Он хранит репозитории в интернете, автоматически синхронизирует их с репозиториями у разработчиков

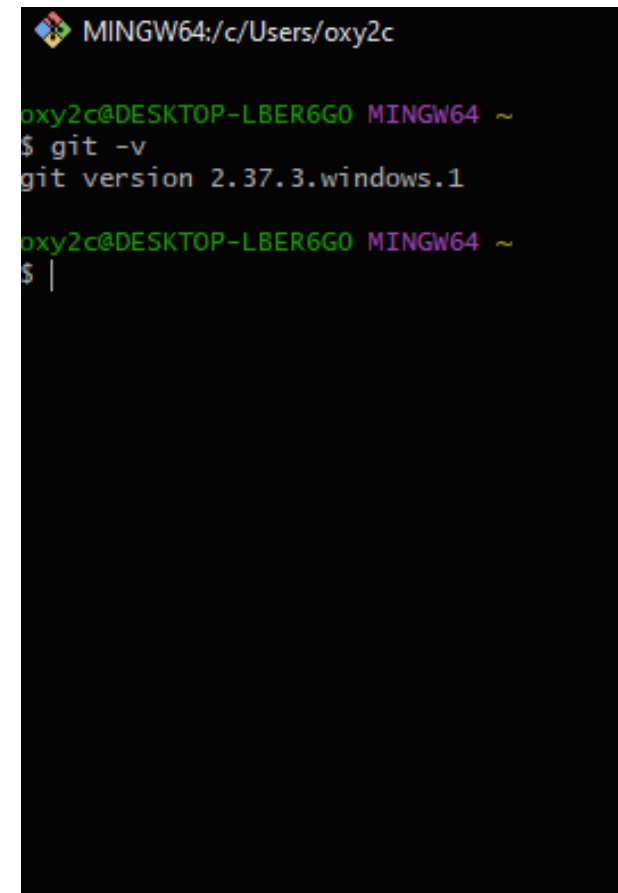
# Как начать пользоваться git?

Git – это в первую очередь инструмент для командной строки, у него нет графического интерфейса.

Конечно, для git существует довольно много приложений с графическим интерфейсом, однако лучше все же уметь пользоваться командами git

Скачать git для Windows можно по ссылке:

<https://git-scm.com/download/win>

A screenshot of a Windows terminal window with a black background. The title bar at the top reads 'MINGW64:/c/Users/oxy2c'. The terminal shows two lines of command and output. The first line shows the command '\$ git -v' followed by the output 'git version 2.37.3.windows.1'. The second line shows the command '\$ |' with a cursor at the end.

```
MINGW64:/c/Users/oxy2c  
oxy2c@DESKTOP-LBER6G0 MINGW64 ~  
$ git -v  
git version 2.37.3.windows.1  
oxy2c@DESKTOP-LBER6G0 MINGW64 ~  
$ |
```

# Первоначальная настройка

Первое, что нужно сделать после установки **Git** – указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git должен содержать эту информацию.

Чтобы настроить имя пользователя введите команды:

```
git config --global user.name "Gennadiy Bookin"  
git config --global user.email genabookin@example.com
```

Также можно поменять текстовый редактор (например на VS Code):

```
git config --global core.editor "code --wait"
```



# Начало работы с репозиториями

# Получение и создания репозитория

Существует два способа создать Git репозиторий:

1. `git clone` – клонирование репозитория из существующего репозитория
2. `git init` – создание репозитория в существующем каталоге.

## Создание репозитория в существующем каталоге

Если у нас уже есть папка с проектом, который не находится под версионным контролем Git, то для начала нужно перейти в эту папку:

```
cd C:/Users/Gena/My_Project
```

а затем выполните команду:

```
git init
```

Эта команда создаст в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы репозитория – структуру Git репозитория.

# Клонирование существующего репозитория

Для получения копии существующего Git-репозитория, например, проекта, в который мы хотим внести свой вклад, необходимо *клонировать* репозиторий.

Клонирование репозитория осуществляется командой `git clone <url>`.

Например, если мы хотим клонировать библиотеку libgit2, то можем сделать это следующим образом:

```
git clone https://github.com/libgit2/libgit2
```

Эта команда создаёт каталог `libgit2`, инициализирует в нём подкаталог `.git`, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии.

# Локальная работа в git

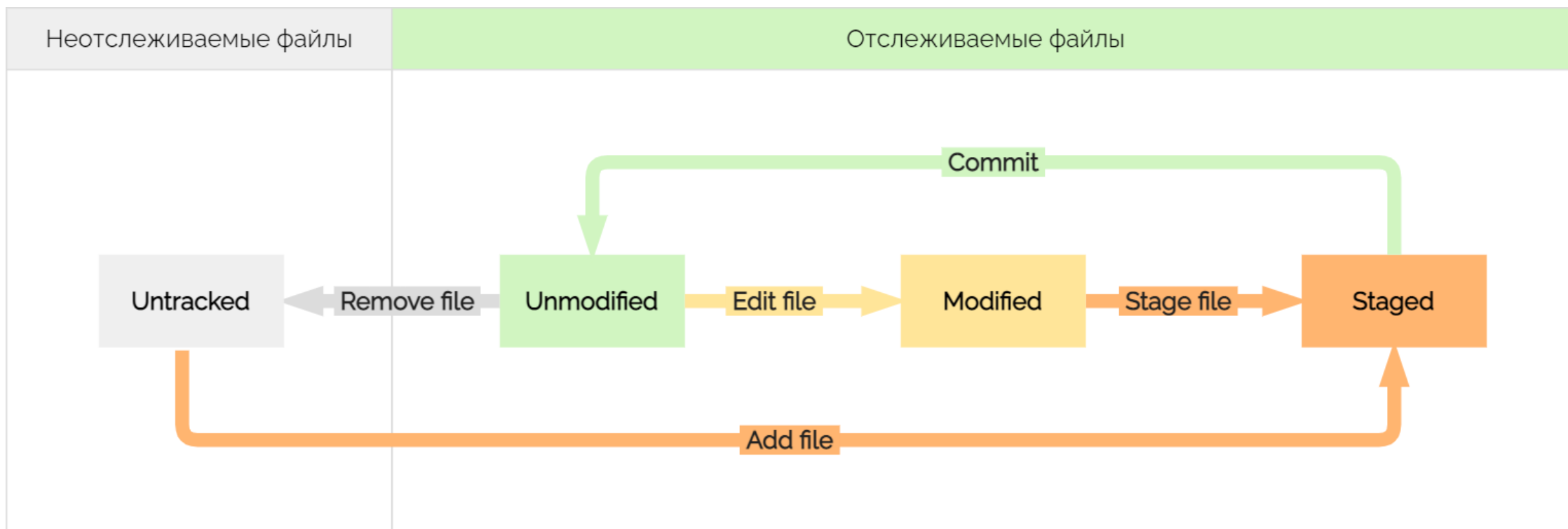
## Запись изменений

Итак, у нас есть локальная копия репозитория. Мы внесли в код какие-то изменения и хотим, чтобы git запомнил текущее состояние нашего проекта – сделал "снимок" (**snapshot**).

Файлы в папке вашего проекта могут находиться в одном из двух состояний:

- **Отслеживаемые (tracked)** – это те файлы, которые были в последнем **snapshot**'е проекта.
- **Неотслеживаемые (untracked)** – это всё остальное, любые файлы в папке проекта, которые не входили в ваш последний **snapshot** и не подготовлены к коммиту

# Статусы файлов в git





## Как узнать статусы файлов?

Чтобы узнать в каком состоянии находятся файлы репозитория git, используется команда `git status`.

Если выполнить эту команду сразу после клонирования репозитория, вы увидите что-то вроде этого:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

Это означает, что у вас чистый рабочий каталог, другими словами — в нем нет отслеживаемых измененных файлов.

# Добавление новых файлов

Для того, чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда `git add`.

Чтобы, например, начать отслеживание файла `README`, нужно выполнить:

```
git add README
```

Если в проекте появилось несколько новых файлов и вы хотите добавить сразу все файлы, можно использовать *wildcard*:

```
git add *
```

Подробнее о паттернах, которые можно использовать при работе с файлами, можно узнать тут:

[https://en.wikipedia.org/wiki/Glob\\_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming))

# Игнорирование файлов

По мере разработки, в папке проекта зачастую появляются файлы, которые не нужно добавлять под версионный контроль. К таким файлам обычно относятся автоматически генерируемые файлы (логи, результаты сборки и т. п.).

Если вы не хотите чтобы git отслеживал такие файлы, вы можете создать файл `.gitignore`. Вот пример такого файла:

```
# My Project ignore rules
*.log
build/
```

1-ая строка – комментарий, начинается с символа `#`

2-ая строка – игнорирование любых файлов с расширением `.log`.

3-я строка – игнорирование всех файлов в каталоге `build/`

# Коммит изменений

Теперь у нас есть все файлы, которые должны попасть в обновленную версию кода. А сам git благодаря файлу `.gitignore` знает какие файлы игнорировать.

Самое время зафиксировать изменения – сделать *коммит*, вот так:

```
git commit
```

Эта команда откроет выбранный вами ранее текстовый редактор, чтобы вы ввели описание сделанных изменений, которое будет прикреплено к коммиту.

Если вам неудобно вводить описание изменений в текстовом редакторе, можно воспользоваться опцией `-m` и сразу добавить сообщение с описанием изменений:

```
git commit -m "Main.ltw: в Exception Handler добавлена съёмка скриншотов."
```

## Отмена коммита

Отмена может потребоваться, если вы сделали коммит слишком рано, например, забыв добавить какие-то файлы или комментарий к коммиту.

Например, если вы сделали коммит и поняли, что забыли проиндексировать изменения в файле, который хотели добавить в коммит, то можно сделать следующее:

```
$ git commit -m 'Initial commit'  
$ git add forgotten_file  
$ git commit --amend
```

В итоге получится единый коммит — второй коммит заменит результаты первого.

# Отмена изменений в файле

Допустим, вы начали работать над каким-то изменением и в ходе работы поняли, что все сделали неправильно.

Что делать, если вы хотите отменить все изменения и вернуть файл файл в исходное состояние?

Чтобы отменить внесенные изменения можно сделать следующее:

```
git checkout -- Process.ltw
```

**Важно:** `git checkout -- <file>` — опасная команда. Все локальные изменения в файле пропадут — Git просто заменит его версией из последнего коммита.

## Удаление файлов

Если вы просто удалите файл из папки своего проекта и наберете команду `git status`, то увидите, что git все еще отслеживает файл в статусе `deleted` и предлагает добавить (`git add`) это изменение к коммиту.

Чтобы git сразу удалил файл и добавил это как изменение к коммиту, удобно использовать команду `git rm`:

```
git rm TEST.ltw
```



## Удаление файла из отслеживаемых

Допустим, вы забыли добавить что-то в файл `.gitignore` и по ошибке проиндексировали ненужный файл.

Теперь вы хотите оставить сам файл в папке проекта, но чтобы git перестал отслеживать этот файл. Чтобы сделать это, используйте команду `git rm` с опцией `--cached`:

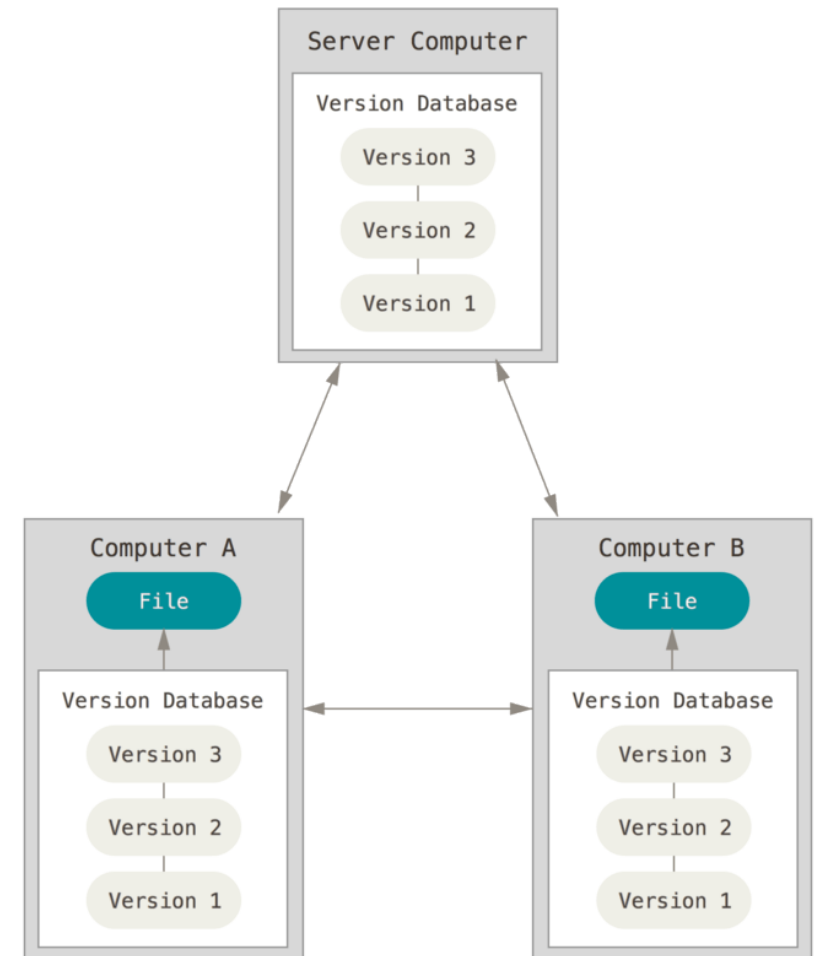
```
git rm --cached debug.log
```

# Совместная работа в git

# Удаленные репозитории

До сих пор мы работали с локальной копией репозитория, однако при разработке больших проектов над кодом проекта может работать целая команда разработчиков. В таком случае проект обычно располагается в удаленном репозитории.

Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети.



# Работа с удаленными репозиториями

Рассмотрим на примере как может выглядеть командная работа с удаленными репозиториями.

В начале недели команда разработки выкатила в прод новую версию продукта – v2.0. До конца недели они собирали фидбек от пользователей продукта и, в результате, узнали о двух новых ошибках:

- **Ошибка 1** (не все данные попали в отчёт)
- **Ошибка 2** (сломанный селектор)

Чтобы быстрее исправить ошибки, разработчики *Гена* и *Валера* решили разделить их между собой: *Гене* досталась **Ошибка 1**, *Валере* – **Ошибка 2**

# Работа с удаленными репозиториями

Вот что сделали Гена и Валера:

1. Гена и Валера клонировали в свои локальные репозитории проект версии **v2.0** из их удаленного репозитория
2. Каждый создал себе отдельную *ветку* на базе главной ветки проекта и начал исправлять ошибки
3. Валера справился с **Ошибкой 2** быстрее. Он сделал коммит своих изменений, выполнил *слияние* своей ветки с **главной веткой** и отправил изменения в удаленный репозиторий
4. Через пару часов Гена тоже закончил свое исправление **Ошибки 2** и тоже отправил свои изменения в удаленный репозиторий
5. Разработчики назвали новую версию **v2.1**, запустили её в прод и пошли праздновать

## Что происходит?

Здорово что у Гены с Валерой все получилось, но пока ничего непонятно:

- Что такое *ветки*?
- Как отправлять изменения на удаленный сервер?
- Что такое *слияние* веток?

Давайте разберемся с этими вопросами подробнее.

# Получение изменений из удалённого репозитория

Предположим, ранее вы клонировали себе удаленный репозиторий с помощью команды `git clone`. Вы пару дней не занимались проектом и другой разработчик уже обновил код в удаленном репозитории.

В данной ситуации, самый удобный способ получить изменения — использовать команду `git pull`.

По умолчанию команда `git clone` автоматически настраивает вашу локальную ветку `master` на отслеживание удалённой ветки `master` на сервере, с которого вы клонировали репозиторий.

Название веток может быть другим и зависит от ветки по умолчанию на сервере.



## Отправка изменений в удаленный репозиторий

Когда вы хотите поделиться своими изменениями, вам необходимо отправить их в удалённый репозиторий. Чтобы сделать это используется команда: `git push` `<remote-name> <branch-name>`.

Чтобы отправить вашу локальную ветку `master` на сервер `origin` (клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

```
git push origin master
```

## Ветвление в git

Когда вы делаете *коммит*, Git сохраняет его в виде объекта, который содержит:

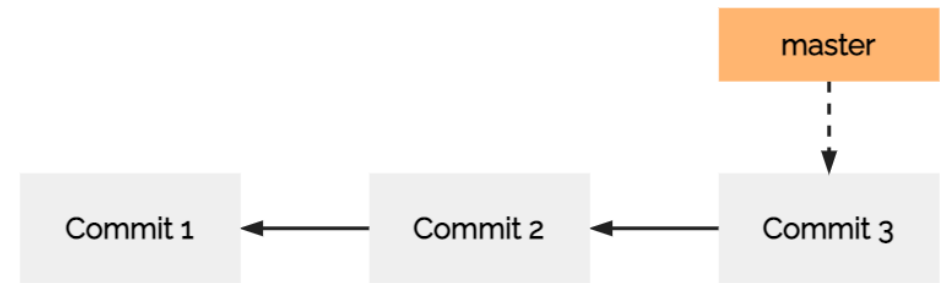
- Снимок (snapshot) всех данных
- Имя автора и его email
- Сообщение об изменениях
- Указатель на предшествующий (родительский) коммит

Если вы сделаете изменения и создадите ещё один коммит, то он будет содержать указатель на предыдущий коммит.

# Ветвление в git

Ветка в Git — это просто указатель на один из таких коммитов.

По умолчанию, имя основной ветки в Git — `master`. Как только вы начнёте создавать коммиты, ветка `master` будет всегда указывать на последний коммит.

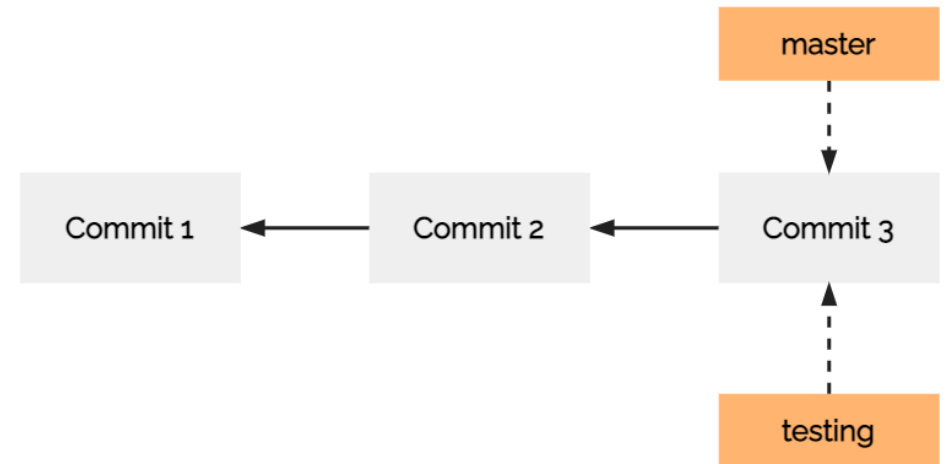


## Создание новой ветки

Что же на самом деле происходит при создании ветки? Всего лишь создаётся новый указатель на один из коммитов.

Допустим вы хотите создать новую ветку с именем `testing`. Вы можете это сделать командой `git branch`:

```
git branch testing
```



## Переключение между ветками

Для переключения на существующую ветку выполните команду `git checkout` .  
Например, чтобы переключиться на только что созданную ветку `testing` :

```
git checkout testing
```

# Слияние веток

Допустим мы выполнили какое-то изменение в ветке `testing`, и протестировали его. Теперь мы хотим применить эти изменения к основной ветке `master`.

Операция, которая делает это называется *слиянием* (*merge*) веток.

Итак, чтобы выполнить слияние ветки `testing` с веткой `master` нужно:

1. Переключиться на ветку `master`:

```
git checkout master
```

2. Выполнить слияние с помощью команды `git merge`:

```
git merge testing
```

Удачи при использовании git!

## Полезные ссылки

1. Скачать Git для Windows можно [тут](#)
2. Онлайн-книга по Git: [на русском](#), [на английском](#)