Оглавление

[Системы контроля версий 2](#_Toc210047313)

[Определение 2](#_Toc210047314)

[Применение 2](#_Toc210047315)

[Задачи СКВ 2](#_Toc210047316)

[История развития 3](#_Toc210047317)

[Локальные системы контроля версий 3](#_Toc210047318)

[Централизованные системы контроля версий 4](#_Toc210047319)

[Распределенные системы контроля версий 4](#_Toc210047320)

[Основные понятия 5](#_Toc210047321)

[Репозиторий 5](#_Toc210047322)

[Версия 6](#_Toc210047323)

[Рабочая копия 6](#_Toc210047324)

[Ветка 7](#_Toc210047325)

[Конфликт версий 8](#_Toc210047326)

[Тег 8](#_Toc210047327)

[Основные операции 8](#_Toc210047328)

[Клонирование 8](#_Toc210047329)

[Коммит 9](#_Toc210047330)

[Слияние 10](#_Toc210047331)

[Индексация 10](#_Toc210047332)

[Push 10](#_Toc210047333)

[Fetch 11](#_Toc210047334)

[Pull 11](#_Toc210047335)

[Процесс работы с проектами в Git 12](#_Toc210047336)

# Системы контроля версий

## Определение

Система контроля версий (СКВ, Version Control System, VCS) — это система, которая сохраняет историю изменений в файлах проекта, позволяя отслеживать, управлять версиями и работать над проектом совместно. Системы контроля версий помогают возвращаться к предыдущим состояниям, видеть кто и когда внес изменения, и устранять ошибки без потери данных.

## Применение

СКВ могут применяться в различных областях, где требуется управление изменениями в файлах и организация совместной работы.

* **Разработка программного обеспечения**

Это самая распространённая и классическая область применения СКВ. Они помогают управлять исходным кодом, фиксируют историю изменений, позволяют разработчикам работать параллельно над разными частями проекта, сливать изменения и откатывать ошибки.

* **Техническая документация и научные исследования**

СКВ используют для отслеживания изменений в документации, чтобы видеть эволюцию текста, исправления и дополнения. В научных проектах система помогает сохранять версии результатов экспериментов и работ коллективов.

* **Управление контентом и файлами**

СКВ применяются для контроля изменений в больших объемах файлов (например, в веб-проектах, дизайн-файлах, учебных материалах), что облегчает совместную работу и восстановление предыдущих версий при ошибках.

* **Командная работа и проектное управление**

СКВ позволяют эффективно координировать действия в командах, избегая конфликтов при одновременной работе, обеспечивая сохранность информации и прозрачность истории изменений.

* **Непрерывная интеграция и развертывание (CI/CD)**

СКВ являются основой современных DevOps-процессов, позволяя автоматически тестировать, собирать и выкатывать программные продукты при изменениях в репозитории.

*В данном тексте* будут рассматриваться примеры применения СКВ в самой распространённой области – управление версиями исходного кода программного обеспечения.

## Задачи СКВ

В последние годы наличие СКВ стало необходимым для эффективного управления большими проектами из-за того, что она решает следующие задачи:

* **Отслеживание изменений.** СКВ сохраняет историю всех изменений, что позволяет разработчикам видеть, кто и когда внес изменения, а также вернуться к любой предыдущей версии кода.
* **Совместная работа.** СКВ позволяет нескольким разработчикам работать над одним проектом одновременно, синхронизируя их изменения и предотвращая конфликты.
* **Ветвление и слияние.** СКВ поддерживает создание параллельных версий проекта (веток), что позволяет разработчикам работать над новыми функциями или исправлениями ошибок без влияния на основную версию кода. После завершения работы изменения можно объединить.
* **Резервное копирование и восстановление.** СКВ обеспечивает надежное хранение всех версий проекта, что защищает от потери данных и позволяет восстановить проект в случае ошибок или сбоев.
* **Документирование изменений.** СКВ позволяет добавлять комментарии к каждому изменению, что помогает лучше понимать причины и контекст изменений.
* **Управление релизами.** СКВ облегчает процесс выпуска новых версий программного обеспечения, позволяя четко отслеживать, какие изменения вошли в каждую версию.

## История развития

История появления и развития СКВ можно разбить на три поколения:

### Локальные системы контроля версий

В 1970-х годах была создана первая система контроля версий — SCCS. Она позволяла отслеживать изменения в файлах проекта и сохранять их историю.

С развитием технологий СКВ стали больше удовлетворять запросам разработчиков, стали сложнее и функциональнее. В СКВ появились возможности сравнивать версии файлов, создавать патчи, а также выполнять автоматическое слияние изменений. Так в 1982 году была разработана RCS. Эта система была совместима с Linux, имела открытый исходный код, а также более простой пользовательский интерфейс.

СКВ первого поколения отслеживали изменения в отдельных файлах, а редактирование поддерживалось только локально и одним пользователем за раз. Системы строились на предположении, что все пользователи будут заходить по своим учётными записям на один и тот же общий узел Unix. Такие системы называют **локальными СКВ**.

Локальный компьютер

Файл

Рабочая копия

База данных версий

Версия 1

Версия 2

Версия 3

Специфика локальных СКВ – это хранение как исходного кода, так и всех его версий в одном месте – на компьютере разработчика. Эта особенность является одновременно и главным минусом таких систем. В случае сбоя может быть потеряна не только какая-то из версий, но и весь проект целиком.

Также в локальных СКВ отсутствует поддержка командной работы и совместного доступа к версиям, что делает их неудобными для организации командной разработки ПО.

### Централизованные системы контроля версий

Следующим шагом в развитии СКВ стало появление сети и централизованных репозиториев.

**Централизованные** **СКВ** — второе поколение СКВ — в отличие от предшественников поддерживают возможность командной работы. Такие системы предполагают хранение проекта на едином сервере. Чтобы изменить исходный код, разработчик должен скачать с сервера необходимые файлы и исправить их, а затем вернуть эти файлы на сервер. Это позволяет организовать совместную, параллельную работу над проектом, где каждый из разработчиков работает с какой-либо скопированной частью исходного кода.

Компьютер 1

Рабочая копия

Центральный сервер СКВ

База данных версий

Версия 1

Версия 2

Версия 3

Компьютер 2

Рабочая копия

Файл

Файл

Однако такая архитектура также обладает существенными недостатками – уязвимость сервера. В случае сбоя на сервере или проблем с подключением к нему, данные могут быть потеряны или недоступны сразу для всех разработчиков. Также для внесения своих изменений в репозиторий разработчику необходим постоянный сетевой доступ.

Первая централизованная система версий — CVS — была разработана в 1986 году. Также к централизованным СКВ относятся Subversion (SVN) и Perforce.

### Распределенные системы контроля версий

Третьим поколением СКВ считается создание **распределенных СКВ**.

В распределенной СКВ все копии репозитория считаются равными, нет центрального репозитория. Центральный сервер может использоваться как хранилище, но не является обязательным элементом архитектуры. Это позволяет делать для коммиты, ветви и слияния, которые создаются локально без доступа к сети и перемещаются в другие репозитории по мере необходимости.

Центральный сервер СКВ

База данных версий

Компьютер 1

Компьютер 3

Компьютер 2

Рабочая копия

Рабочая копия

Рабочая копия

База данных версий

База данных версий

База данных версий

Системы распределенного типа позволили избавиться от основного недостатка централизованных СКВ — уязвимости сервера. В распределенной версии СКВ каждый разработчик скачивает с сервера не только нужный ему файл, но и весь набор данных. Таким образом, при сбое на сервере восстановить данные можно будет с компьютера любого из разработчиков.

К таким системам относится разработанный в 2003 году Darcs, а также Git, Mercurial и Bazaar, разработанные в 2005 году, Fossil, разработанный в 2006 году.

В настоящее время системы контроля версий такого типа используются чаще всего и являются стандартом в командной разработке ПО. В дальнейшем будут подробно рассматриваться особенности распределенных СКВ на примере системы Git.

# Основные понятия

## Репозиторий

Репозиторий в системе контроля версий — это специальное хранилище, в котором сохраняются все файлы проекта и история всех изменений, внесённых в них. Репозиторий может находиться как на удалённом сервере (централизованное или облачное хранение), так и локально на компьютере разработчика.

**Локальный репозиторий** — это полная копия репозитория, которая хранится на компьютере разработчика. Он включает в себя:

1. Рабочую копию – текущие файлы проекта, с которыми работает разработчик.
2. Историю изменений – полная история всех коммитов, веток и меток.
3. Индекс (staging area) – промежуточная область, где изменения подготавливаются перед коммитом (используется в Git, аналог в Mercurial – расширение Mercurial Queues).

Преимущества локального репозитория:

* Автономная работа. Разработчик может выполнять коммиты, создавать ветки и сливать изменения без подключения к сети.
* Быстрота операций. Большинство операций выполняются локально и очень быстро.
* Безопасность данных. Полная копия репозитория на каждом компьютере обеспечивает резервное копирование.

**Удалённый репозиторий** — это репозиторий, который хранится на сервере и доступен через сеть. Он служит центральной точкой для синхронизации изменений между разработчиками.

Основные операции с удалённым репозиторием:

* Клонирование (clone). Создание локальной копии удалённого репозитория.
* Извлечение (fetch). Получение изменений из удалённого репозитория без их автоматического слияния с локальными изменениями.
* Слияние (merge). Объединение изменений из удалённого репозитория с локальными изменениями.
* Отправка (push). Отправка локальных изменений в удалённый репозиторий.
* Вытягивание (pull). Комбинированная операция извлечения и слияния изменений из удалённого репозитория.

Преимущества удалённого репозитория:

* Централизованное хранилище. Обеспечивает единое место для хранения и обмена изменениями между разработчиками.
* Совместная работа. Упрощает координацию и синхронизацию работы в команде.
* Резервное копирование. Дополнительный уровень защиты данных.

Рабочий процесс с репозиторием включает его создание, добавление файлов, коммиты, работу с ветками и слияниями, а также синхронизацию локальных изменений с удалённым репозиторием. Репозиторий обеспечивает комфортное ведение истории проекта и совместную работу команды разработчиков.

## Версия

Версия (ревизия, revision) – это конкретное состояние файла или набора файлов в определённый момент времени, зафиксированное в истории изменений. Версия представляет собой «снимок» проекта, который сохраняет все изменения, сделанные до этого момента.

Благодаря версиям можно эффективно отслеживать эволюцию проекта, понимать природу изменений, восстанавливать и сравнивать предыдущие состояния в случае необходимости.

Каждый раз при сохранении изменений в проекте создаётся новая версия, связанная с уникальным идентификатором и метаданными — датой, автором изменений и сообщением, поясняющим что было изменено. Версия включает не только содержимое файлов, но и структуру проекта, например, расположение файлов, ветки разработки.

СКВ позволяют работать с несколькими версиями одновременно, используя ветки, и объединять их изменения.

## Рабочая копия

Рабочая копия (working copy, working tree, working directory) — это локальная версия файлов проекта, с которой непосредственно работает пользователь. Она представляет собой текущее состояние файлов на компьютере разработчика, готовое к редактированию, тестированию и другим видам работы.

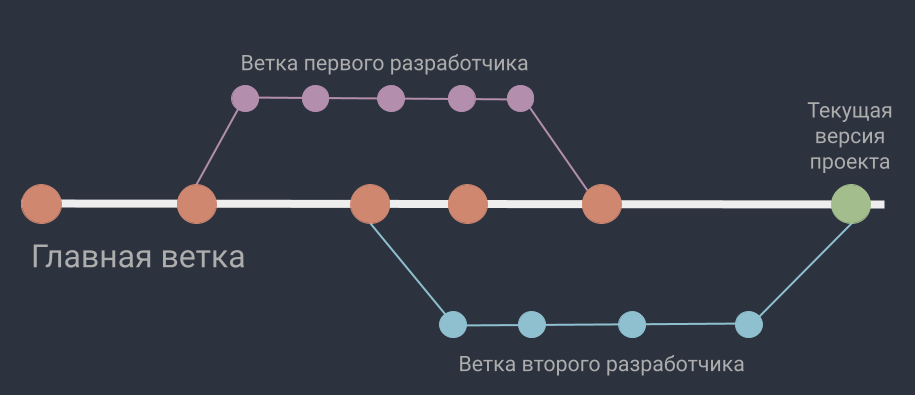
Рабочая копия не содержит полной истории версий — она отражает только текущее состояние на момент последнего обновления. Чтобы изменения из рабочей копии стали частью истории проекта, их нужно зафиксировать (сделать коммит) через систему контроля версий.

Рабочая копия непрерывно синхронизируется с локальным или удалённым репозиторием: её изменения можно загрузить (commit, push), а обновления из репозитория получить (pull, fetch). При конфликте изменений СКВ помогает выявить разногласия между рабочей копией и репозиторием и предлагает способы их разрешения.

Рабочая копия отображает промежуточное состояние между сохранёнными версиями в репозитории и изменениями на локальном уровне. Она даёт возможность свободно экспериментировать с кодом без немедленного внесения изменений в основную историю проекта.

## Ветка

Ветка — это специальная, созданная параллельно с основным репозиторием, копия проекта. Ветвление позволяет изолировать изменения, экспериментировать с новыми функциями и исправлять ошибки, не влияя на стабильность основной базы кода — master-ветку.



Основные этапы работы с ветками:

1. Создание новой ветки в репозитории. Для этого от уже существующей в репозитории ветки — master-ветки или любой другой — откалывают новую ветку.
2. Работа с файлами проекта, сохранение изменений и создание коммитов в созданной ветке.
3. Создание пулл реквеста (Pull Request) — запроса на сохранение в родительской ветке изменений, которые были внесены в рабочую ветку. В пулл реквесте другие разработчики могут обсудить правки, добавить свои коммиты при необходимости, принять или отклонить предложенные изменения.
4. Слияние (Merge) веток — рабочую ветку вливают в родительскую. Теперь изменения внесены и доступны всем пользователям.

Ветвление позволяет организовать командную разработку проекта. Поскольку разработчики делают коммиты в специально созданных ветках, исходный код защищен от ошибок или случайного удаления. Ветвление полезно, если нужно внести изменения, которые могут повлиять на другие части проекта. Кроме того, оно позволяет протестировать новые функции перед их интеграцией в master-ветку.

## Конфликт версий

Конфликт версий — это ситуация, когда одновременно внесённые разными разработчиками изменения в одни и те же строки одного файла не могут быть автоматически объединены системой и требуют ручного разрешения.

СКВ предоставляет инструменты для выявления конфликтов, сравнения версий и удобного их устранения. Однако, если при попытке слияния версий СКВ не может однозначно определить, какая версия должна остаться, система помечает конфликтные участки и требует от разработчиков вручную выбрать, какие изменения сохранять.

При разрешении конфликтов важно следить за следующими моментами:

1. Понимание изменений. Должны быть изучены различия между ветками и выбраны наиболее подходящие изменения.
2. Изменение файлов. Должны быть внесены необходимые изменения в файлы проекта таким образом, чтобы сохранялась логика и целостность кода.
3. Проверка работоспособности. После разрешения конфликтов необходимо убедится, что код все еще работает корректно, и нет новых ошибок.

Слияние веток и разрешение конфликтов требует внимательности и осторожности, особенно в крупных проектах с множеством разработчиков.

Хорошие практики работы с СКВ, такие как частые коммиты, ветвление и синхронизация, помогают минимизировать количество конфликтов.

## Тег

Тег в системе контроля версий — это символическая метка, которая присваивается определённой версии (коммиту) в репозитории для удобной идентификации и быстрого доступа. Тег используется для обозначения важных моментов в истории проекта, например, выпусков (релизов), стабильных версий или контрольных точек.

В отличие от веток, которые продолжают развиваться, тег фиксирует конкретное состояние репозитория и обычно не меняется.

Теги помогают легко находить и переключаться на определённые версии проекта без необходимости запоминать длинные идентификаторы коммитов.

В системах контроля версий, таких как Git, существуют аннотированные теги с дополнительной информацией (автор, дата, сообщение) и лёгкие теги — простые метки, указывающие на определённый коммит.

Таким образом, тег — это удобный способ пометить ключевые версии в истории проекта и обеспечить быстрый доступ к ним.

# Основные операции

## Клонирование

Клонирование — это операция создания полной копии существующего репозитория, включая все файлы проекта, историю коммитов, ветки и ссылки на удалённые репозитории. При клонировании появляется полноценная локальная рабочая копия, с которой можно работать автономно, вносить изменения, создавать коммиты и синхронизироваться с удалённым репозиторием.

Клонированный репозиторий остаётся связан с удалённым источником, что позволяет получать обновления или отправлять свои изменения.

Основные задачи клонирования следующие:

* Предоставление полного доступа к проекту и его истории для разработки и анализа.
* Обеспечение удобной совместной работы в распределённых командах.
* Предоставление возможности работать офлайн и синхронизации по сети.

В Git клонирование выполняется командой ***git clone <URL>***, которая создает каталог с проектом, который содержит полный рабочий репозиторий, где ***<URL>*** - ссылка на удалённый репозиторий (может быть HTTPS или SSH).

Варианты клонирования:

* Через командную строку — более универсально и гибко.
* Через графические интерфейсы Git GUI (GitHub Desktop, Sourcetree, GitKraken, GitVerse) — для удобства визуального управления.
* Через интеграцию в IDE (например, Visual Studio) — для упрощённой работы программиста.

Клонирование является стартовой операцией, с которой обычно начинается работа над новым проектом. В отличие от простого скачивания файлов, клонирование обеспечивает полноценное управление версиями и полный набор инструментов распределённой системы контроля версий.

## Коммит

СКВ помогает сохранять так называемые «снимки» состояния проекта, его промежуточные версии. Это своего рода опорные точки, к которым разработчик может вернуться в случае необходимости. Такие сохранения называются ***коммитами***. Коммиты содержат информацию о том, какие изменения были внесены в исходный код, но не весь проект целиком. Таким образом система контроля версий может сохранять большое количество изменений с минимальными затратами на ресурсы.

*Коммит в Git — это основная операция, которая фиксирует изменения в репозитории. Он создаёт снимок текущего состояния файлов в рабочей директории и сохраняет его в истории изменений.*

Коммиты могут быть небольшими и частыми, что помогает лучше отслеживать прогресс и упрощает процесс отладки.

Основными характеристиками коммита являются:

1. Снимок состояния. Коммит сохраняет текущее состояние файлов, которые были добавлены в индекс (staging area). Это позволяет восстановить проект до состояния на момент коммита.
2. Уникальный идентификатор. Каждый коммит имеет уникальный SHA-1 хеш, который идентифицирует его в истории репозитория.
3. Сообщение коммита. Коммит сопровождается сообщением, которое описывает внесённые изменения. Хорошо написанные сообщения коммитов помогают понять, что было изменено и почему.
4. Ссылки на предыдущие коммиты. Коммиты образуют цепочку, где каждый коммит (кроме первого) ссылается на предыдущий. Это позволяет Git отслеживать историю изменений и строить граф изменений.

## Слияние

Слияние (merge) — это процесс объединения изменений из одной ветки в другую. Слияние позволяет интегрировать новые функции или исправления ошибок в основную ветку проекта. При слиянии могут возникать конфликты, если изменения в разных ветках противоречат друг другу.

Слияние может быть автоматическим или ручным, в зависимости от сложности изменений. Автоматическое слияние происходит, когда изменения не конфликтуют, а ручное — требует вмешательства разработчика для разрешения конфликтов. Важно регулярно сливать изменения из основной ветки в рабочие ветки, чтобы минимизировать вероятность конфликтов.

## Индексация

Индексация в системе контроля версий — процесс добавления изменений из рабочей копии в промежуточную область, называемую индексом или staging area, для последующего включения этих изменений в коммит.

*Индексация (staging) — это промежуточный этап подготовки изменений перед фиксацией в коммите.*

*Индекс — это промежуточная зона между рабочей копией и репозиторием, где формируется набор изменений, которые войдут в следующий коммит.*

Индексация позволяет выборочно подготовить изменения к сохранению, например, можно добавить только часть изменений в файле. Если после индексации в файле были сделаны дополнительные правки, они не попадут в текущий коммит, пока не будут добавлены повторно в индекс.

Индексация упрощает контроль над тем, какие именно изменения фиксируются, что полезно при работе с большим количеством правок.

В Git индексация выполняется командой ***git add***, которая помещает изменения файла или его части в индекс.

Команда ***git status*** показывает, какие изменения проиндексированы, а какие ещё нет.

Индексация — ключевой этап в работе с Git, обеспечивающий точное формирование коммитов и управление историей проекта.

## Push

Операция ***push*** в системе контроля версий — это процесс отправки локальных изменений, сохранённых коммитами, из локального репозитория в удалённый репозиторий. Она позволяет синхронизировать работу разработчика с общим центральным репозиторием или сервером, делая изменения доступными для всей команды.

Операция ***push*** работает следующим образом:

1. Разработчик делает коммиты локально, сохраняя изменения с комментариями.
2. Команда *push* отправляет эти коммиты и связанные с ними изменения на сервер.
3. При успешном выполнении команды изменения появляются в удалённом репозитории.
4. Если в удалённом репозитории появились новые изменения, которых нет в локальном, проведение операции *push* может быть отклонено. Тогда сначала нужно выполнить операцию *pull*, чтобы получить и слить эти изменения локально.

В Git операция выполняется командой ***git push <remote> <branch>***, которая помещает изменения файла или его части в индекс.

* Пример: *git push origin main* — отправить изменения из ветки main на сервер origin.

Особенности операции ***push***:

* Операция обновляет только ветки, с которыми участвует разработчик.
* Операция требует актуальной локальной версии, иначе возникает конфликт.
* В распределённых системах *push* — это лишь часть цикла синхронизации, вместе с *pull* и *fetch*.

Операция ***push*** — ключевой механизм обмена изменениями между разработчиками и совместного прогресса над проектом. Она гарантирует, что результаты работы, локально сохранённые разработчиком, станут доступны команде, и все будут работать с актуальной версией кода.

## Fetch

Операция ***fetch*** — это операция, которая загружает последние изменения из удалённого репозитория в локальное хранилище без изменения файлов в текущей рабочей директории. Операция ***fetch*** обновляет локальные данные о состоянии удалённого репозитория, но не влияет на текущие рабочие файлы и ветки.

Операция ***fetch*** выполняется для того, чтобы узнать, какие изменения появились в удалённом репозитории, не вмешиваясь в локальную работу, подготовить локальный репозиторий к последующим операциям слияния и запланировать дальнейшую работу, предварительно изучая изменения коллег.

Операция ***fetch*** выполняетследующиефункции:

* Установление соединения с удалённым репозиторием.
* Загрузка новых коммитов, веток, тегов и других объектов.
* Обновление локальных ссылок на удалённые ветки (так называемые ветки удалённого отслеживания).

При этом текущее состояние рабочей копии не изменяется и изменения не сливаются автоматически.

В Git операция выполняется командой ***git fetch <remote>***.

Операция ***fetch***часто используется для проверки обновлений перед слиянием или обновлением локального репозитория, так как она позволяет подготавливать данные без риска вмешательства в текущую работу.

## Pull

Операция ***pull*** — это процесс загрузки последних изменений из удалённого репозитория в локальный репозиторий с их автоматическим слиянием с текущей локальной веткой. Проще говоря, ***pull*** объединяет две операции: получение обновлений (fetch) и их слияние в текущую рабочую копию (merge).

Процесс выполнения операции ***pull*** выглядит следующим образом:

* Устанавливается соединение с удалённым репозиторием.
* Загружаются все новые коммиты, файлы, ветки и теги, которые появились с момента последней синхронизации.
* Происходит автоматическое слияние загруженных изменений с локальной веткой, над которой работает разработчик.
* Если конфликты отсутствуют, merge происходит автоматически. При конфликтах требуется вмешательство пользователя для разрешения.

В Git операция выполняется командой ***git pull <remote> <branch>***. Для операции есть возможность задать опцию ***–rebase***, которая применит изменения поверх локальных коммитов, что помогает сохранять линейную историю.

Отличия ***fetch*** от ***pull***:

* ***fetch*** скачивает изменения, но не объединяет их с локальными. Это безопасная операция для инспекции изменений.
* ***pull*** сначала делает ***fetch***, а затем автоматически сливает изменения с текущей веткой, обновляя рабочие файлы.

Операция ***pull*** используется для того, чтобы регулярно получать последние изменения от команды, обновлять локальную копию проекта перед началом работы и интеграции изменений, сделанных другими разработчиками.

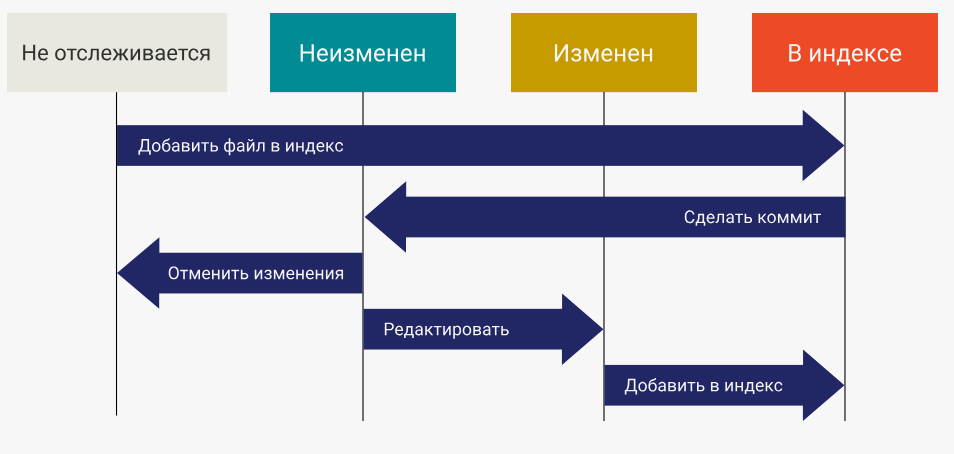
Операция ***pull*** может вызвать конфликты, если изменения в удалённом репозитории и локальные изменения затрагивают одни и те же файлы. Таким образом, рекомендуется перед операцией ***push*** выполнять операцию ***pull***, чтобы обновить локальную версию и избежать конфликтов при отправке своих изменений.

# Процесс работы с проектами в Git

У проектных файлов в Git есть 3 базовых состояния

* **Измененные** (modified) — файлы в процессе рабочего редактирования.
* **Индексированные** (staged) — та часть измененных файлов, которая уже подготовлена к фиксации после редактирования.
* **Зафиксированные** (committed) — файлы, уже сохраненные в локальном репозитории.

Схематичный процесс внесения изменений в рамках системы контроля версий:



Основные шаги при работе с проектами в Git:

1. **Инициализация репозитория (Initialize repository)**. Создание нового локального репозитория для отслеживания изменений. Команда  Git — *git init*, Mercurial — *hg init*. Это первый шаг перед началом работы с проектом.
2. **Клонирование репозитория (Clone repository)**. Получение копии удалённого репозитория на локальный компьютер для работы офлайн. Команда Git — *git clone <url>*.
3. **Отслеживание изменений (Track changes).** Добавление новых или изменённых файлов в список отслеживаемых файлов для последующего сохранения. Команда Git — *git add,* Mercurial - *hg add*. Это подготовительный этап перед коммитом.
4. **Фиксация изменений (Commit changes).** Сохранение изменений текущего состояния проекта в истории с комментарием. Коммит создает снимок проекта, к которому можно будет вернуться. Команда Git — *git commit*, Mercurial — *hg commit*.
5. **Просмотр статуса (Check status).** Позволяет увидеть, какие изменения внесены, какие файлы подготовлены к коммиту, а какие нет. Команда Git — *git status*, Mercurial — *hg status*.
6. **Просмотр истории (View history)**. Просмотр всех коммитов и изменений за время ведения проекта. Команда Git — *git log*, Mercurial — *hg log*.
7. **Создание и переключение веток (Branching and switching)**. Создание независимой ветки разработки и переключение между ними для параллельной работы. Команды Git – *git branch <branch>, git checkout <branch>*; Mercurial – *hg branch <branch>, hg update <branch>*.
8. **Слияние веток (Merge branches)**. Объединение изменений из одной ветки в другую с разрешением конфликтов при необходимости. Команда Git – *git merge <branch>*, Mercurial – *hg merge <branch>*.
9. **Синхронизация с удалённым репозиторием (Push / Pull / Fetch):**
   * **Push** — отправка локальных изменений в удалённый репозиторий (*git push, hg push*).
   * **Pull** — загрузка изменений из удалённого репозитория и их автоматическое слияние (*git pull, hg pull && hg update*).
   * **Fetch** — просто загрузка изменений без слияния (*git fetch <remote>*).
10. **Разрешение конфликтов (Resolve conflicts)**. При слиянии веток или получении изменений с удалённого репозитория могут возникать конфликты. Распределённые СКВ предоставляют инструменты для их обнаружения и разрешения.