



МОСКОВСКИЙ УНИВЕРСИТЕТ ИМЕНИ С.Ю. ВИТТЕ

Кафедра информационных систем

А.А. Булычев

**СИСТЕМА УПРАВЛЕНИЯ ВЕРСИЯМИ GIT
И РОССИЙСКИЙ СЕРВИС ХРАНЕНИЯ
ИСХОДНОГО КОДА GITFLIC**

УЧЕБНОЕ ПОСОБИЕ

**Москва
2022**

УДК 004.3(075.8)

ББК 32.973.2я73

Б90

Рецензенты:

Дерябин С.А. – исполнительный директор ООО «Ресолют Образование»;

Зайцев С.А. – канд. техн. наук, доцент, декан факультета информационных технологий Московского университета им. С.Ю. Витте

Б90 **Булычев А.А.**

Система управления версиями GIT и российский сервис хранения исходного кода GitFlic: учеб. пособие / А.А. Булычев; Моск. ун-т им. С.Ю. Витте; каф. ИС [Электронное издание]. – Москва: изд. ЧОУВО «МУ им. С.Ю. Витте», 2022. – 2,3 Мб.

Эффективное управление программным проектом, выполняемым командой программистов, немыслимо без использования системы управления версиями. В учебном пособии рассмотрены общие принципы построения систем управления версиями, подробно описана внутренняя организация одной из популярных систем: GIT. Показана возможность использования репозитория GitFlic в учебном процессе для подготовки программистов.

Издание предназначено для студентов всех форм обучения ИТ-направлений подготовки в соответствии с требованиями Федеральных государственных образовательных стандартов. Может быть использовано аспирантами, преподавателями вузов, руководителями организаций, менеджерами всех уровней управления и практикующими разработчиками программного обеспечения.

*Рекомендовано к изданию решением Научно-методического совета
МУ им. С.Ю. Витте № 4 от 03 ноября 2022 г.*

ISBN 978-5-9580-0635-9

© Булычев А.А., 2022

© ЧОУВО «МУ им. С.Ю. Витте», 2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	6
1. ИСТОРИЯ РАЗВИТИЯ СИСТЕМ УПРАВЛЕНИЯ ВЕРСИЯМИ	8
1.1. Назначение систем управления версиями.....	8
1.2. Краткая история развития систем управления версиями	11
1.3. Системы управления версиями первого поколения.....	12
1.4. Системы управления версиями второго поколения.....	16
1.5. Системы управления версиями третьего поколения	20
1.6. Встроенные системы управления версиями	25
2. СИСТЕМА УПРАВЛЕНИЯ ВЕРСИЯМИ GIT	29
2.1. Особенности и преимущества Git.....	29
2.2. Установка и настройка Git для Windows	30
2.3. Основные понятия	31
2.4. Настройка Git, команда git config	32
2.5. Создание локального репозитория, команда git init.....	33
2.6. Логическая структура Git	34
2.7. Состояния файлов, команда git status	35
2.8. Объекты Git.....	37
2.9. Добавление файла в индекс, команда git add.....	39
2.10. Создание коммита, команда git commit.....	44
2.11. Технология использования веток.....	50
2.12. Создание ветки.....	53
2.13. Команда git stash	56
2.14. Просмотр истории коммитов	58
2.15. Сравнение версий файлов, команды git diff, git show	60
2.16. Удаление файлов, команда git rm	65
2.17. Восстановление файлов, команда git restore.....	66
2.18. Перемещение указателя HEAD, команда git checkout (продолжение).....	67
2.19. Удаление и объединение коммитов, команда git reset.....	70
2.19.1. Редактирование последнего коммита.....	71
2.19.2. Удаление последних коммитов	72
2.19.3. Объединение коммитов	73
2.19.4. Отмена действия команды git reset	73
2.20. Удаление веток	73
2.21. Слияние веток, принципы выполнения операции.....	74
2.22. Слияние веток, команда git merge.....	77
2.22.1. Устранение конфликтов слияния.....	78

2.23. Перенос начала ветки, команда git rebase	82
2.24. Копирование коммита, команда git cherry-pick	85
2.25. Работа с внешним репозиторием	87
2.25.1. Подключение внешнего репозитория, группа команд git remote	87
2.25.2. Отправка изменений в удалённый репозиторий, команда git push	89
2.25.3. Клонирование удалённого репозитория, команда git clone	90
2.25.4. Просмотр внешнего репозитория, команда git fetch	94
2.25.5. Получение изменений из локального репозитория, команда git pull	96
3. ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ GIT	98
3.1. Управление репозиторием из среды программирования	98
3.1.1. Операции с репозиторием Git в среде PyCharm	98
3.1.1.1. Подключение локального репозитория к проекту Pycharm	98
3.1.1.2. Клонирование внешнего репозитория в проект Pycharm	99
3.1.1.3. Выполнение команд Commit и push из проекта PyCharm	101
3.1.2. Операции с репозиторием Git в среде Visual Studio	103
3.1.2.1. Клонирование репозитория	103
3.1.2.2. Графическая схема репозитория	104
3.1.2.3. Выполнение команд Commit и push из проекта Visual Studio	105
3.2. Учебный портал сайта GitFlic	106
3.2.1. Вход на учебный портал GitFlic	107
3.2.2. Регистрация преподавателей и студентов	108
3.2.3. Взаимодействие преподавателя и студентов	109
3.2.4. Создание заданий	110
3.2.4.1. Создание учебного курса	110
3.2.4.2. Добавление на курс преподавателей и студенческих групп	110
3.2.4.3. Добавление лекций	111
3.2.4.4. Добавление задания	112
3.2.4.5. Распределение заданий между студентами	112
3.2.4.6. Создание репозитория	113
3.2.4.7. Отправка задания на проверку	114
3.2.4.8. Проверка заданий	115
3.2.4.9. Просмотр результатов проверки	115
3.3. Репозиторий GitFlic	116
3.3.1. Структура окна сайта GitFlic	116
3.3.2. Создание нового репозитория	118
3.3.3. Управление проектом	119
3.3.4. Администрирование и просмотр публичных проектов	123
3.3.4.1. Клонирование репозитория	123

3.3.4.2. Избранное	123
3.3.4.3. Переименование проекта	124
3.3.5. Видимость проекта	124
3.3.6. Передача репозитория другому владельцу	125
3.3.7. Добавление пользователей в проект	125
3.3.8. Просмотр списка публичных проектов	125
3.3.9. Создание нового проекта и управление настройками	126
3.3.9.1. Создание проекта	126
3.3.9.2. Изменение настроек проекта	127
3.3.9.3. Настройка основных веток проекта	128
3.3.10. Работа с аккаунтом	129
3.3.10.1. Профиль пользователя	129
3.3.10.2. Настройка аккаунта	130
3.3.11. Авторизация и регистрация	132
3.3.12. Уведомления	133
3.3.13. Создание README для профиля	134
3.3.14. README для организации	135
3.3.15. Создание команды разработчиков	136
3.3.16. Создание компании	137
3.3.17. Добавление сотрудников	138
ЗАКЛЮЧЕНИЕ	140
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	142

ВВЕДЕНИЕ

На каком бы алгоритмическом языке ни писалась программа, она остаётся текстом, который хранится в файле (одном или нескольких). В ходе работы над программным проектом создаются новые файлы с текстом программ, удаляются и изменяются старые. До тех пор, пока проект небольшой, никаких серьёзных проблем не возникает. С ростом объёма текста и количества файлов появляются проблемы, например, как вернуться к старой версии текста или как одновременно заниматься сопровождением рабочей версии и разработкой перспективной?

Представьте себе, что программист в течение нескольких дней занимался решением задачи, расширяющей возможности текущего проекта, он внёс большое количество исправлений в несколько файлов. Неожиданно оказалось, что решаемая задача больше не актуальна (из-за нехватки времени заказчик отказался от расширения проекта), необходимо вернуться к старой версии проекта. Это неприятность, а, может быть, и большая проблема.

В программный продукт, уже сданный заказчику, необходимо внести изменения, при этом не отменяется задача исправления небольших погрешностей в этом же продукте. Разумеется, вносить изменения в эксплуатирующийся код нельзя. Снова неприятность, а, может быть, проблема.

Кажется, что решение проблем лежит на поверхности: создавать копии каталога с текстами программ и сохранять их в хронологическом порядке. При реализации этого подхода очень скоро возникнет ситуация, когда захочется внести изменения в файлы каталога, сохранённого три шага назад, потом в файлы каталога, сохранённого два шага назад, а потом соединить эти изменения. И вот уже возникла путаница. Дальше – больше... путаницы.

Следующий круг проблем возникает, когда над проектом работают несколько человек. Необходимо избежать внесения изменений в чужой код и обеспечить объединение результатов работы членов команды.

Программисты, столкнувшись с описанными проблемами, нашли решение. Нет, не нашли, а написали. Они написали (создали) систему управления версиями, которая обеспечивает:

- быстрый возврат к любой из прошлых версий программного продукта;
- удобный просмотр цепочки сделанных изменений;
- безопасную и эффективную коллективную работу.

Настоящее учебное пособие посвящено изучению системы управления версиями Git и репозитория GitFlic, включая возможность его использования в учебном процессе. Также в главе 3 приведены справочники по системе GitFlic и рекомендации по использованию Git в средах программирования Visual Studio и PyCharm. Учебное пособие состоит из трёх глав.

В первой главе рассмотрена история развития систем управления версиями с последовательным описанием этапов становления их современных реализаций. В начале главы, на примере одной из моделей жизненного цикла программного продукта, объяснено, что такое система управления версиями и почему её применение является обязательным при ведении разработки программ и программных комплексов различного назначения.

Во второй главе подробно рассмотрена система управления версиями Git. Описаны внутренние механизмы её функционирования. Многочисленные примеры позволяют понять устройство и взаимосвязь структур данных, создаваемых при работе с Git. Детально рассмотрены приёмы работы с ветвями.

Третья глава посвящена описанию методики использования репозитория GitFlic при организации обучения студентов программированию. Рассмотрены способы подключения к репозиторию. Описана процедура обмена данными между преподавателем и студентом в ходе выполнения учебных заданий. Подробно описан интерфейс учебного портала GitFlic.

Кроме того, приведены инструкции по интеграции репозитория Git в среды программирования и использованию графического интерфейса для выполнения основных команд Git.

1. ИСТОРИЯ РАЗВИТИЯ СИСТЕМ УПРАВЛЕНИЯ ВЕРСИЯМИ

Перед тем, как перейти к изложению истории развития систем управления версиями, считаем полезным объяснить, что они собой представляют и зачем нужны.

1.1. Назначение систем управления версиями

Для того чтобы понять, что такое система управления версиями и зачем она нужна, рассмотрим модель жизненного цикла программного продукта, известную как V-модель. Жизненный цикл программного продукта включает 6 этапов:

1. Планирование и анализ требований.
2. Создание документации.
3. Разработка архитектуры проекта.
4. Реализация.
5. Тестирование.
6. Развёртывание и сопровождение.

Модель жизненного цикла показывает последовательность реализации названных этапов и взаимосвязи между ними. Схема V-модели представлена на рисунке 1.1. Модель основана на объединении стадий разработки с соответствующими стадиями тестирования. Следующая фаза разработки начинается только после завершения предыдущей. Каждый этап разработки напрямую связан с соответствующим этапом тестирования. В левой части диаграммы расположены этапы проектирования, в правой части – этапы тестирования.

На этапах проектирования выполняются следующие работы:

- планирование и анализ требований – сбор информации о проекте, требованиях заказчика;
- создание документации – согласованные требования заказчика оформляются в виде договора, составляется рабочая документация проекта;

- разработка архитектуры проекта на системном уровне – определяется состав базовых модулей проекта, выбираются или разрабатываются математические модели и алгоритмы;
- детализированная разработка проекта – проект разбивается на модули, составляются требования, предъявляемые к каждому модулю, выбираются инструменты разработки;
- кодирование – пишется программный код для реализации всех модулей.



Рисунок 1.1 – V-модель жизненного цикла программного продукта

На этапах тестирования выполняются следующие работы:

- модульное тестирование – выполняется тестирование отдельных модулей с целью устранения ошибок в программном коде;
- интеграционное тестирование – совместное тестирование модулей, имеющих связи с целью устранения ошибок проектирования межмодульных интерфейсов;

- системное и приёмное тестирование – полностью готовый программный продукт тестируется на соответствие требованиям заказчика;
- эксплуатация и сопровождение – устраняются ошибки, обнаруженные в процессе эксплуатации; по просьбе заказчика вносятся изменения, улучшающие функционал программного продукта.

Такая модель жизненного цикла предполагает прототипирование. Например, для тестирования высокоуровневой архитектуры проекта необходим прототип готового продукта. Под прототипом понимается функционирующий программный продукт, созданный на основании промежуточной документации, возможно с частично ограниченным функционалом. В V-модели используются горизонтальные и вертикальные прототипы. Горизонтальные прототипы применяются для прояснения неясных требований, когда архитектура продукта не затрагивается. Вертикальные прототипы используются для проверки архитектурных решений.

Горизонтальные связи, присутствующие в V-модели, свидетельствуют о возможности возврата к предыдущим этапам разработки для уточнения системных требований и внесения изменений в программный код по результатам тестирования прототипов. В результате многократных изменений, отражающих процесс поиска оптимального решения, возникает большое количество вариантов программной реализации отдельных модулей.

В любой момент может возникнуть необходимость вернуться к предыдущей версии или проверить несколько альтернативных решений. Такой стиль разработки предполагает частое сохранение результатов, соответствующих большому количеству контрольных точек, и возможность быстрого возврата в любую из этих контрольных точек. Объём программного кода быстро растёт, всё труднее становится разобраться, что, где и почему было изменено.

Возникает необходимость в специальном инструменте, позволяющем документировать ход разработки и быстро переходить от одной версии кода к другой вперёд, назад и в бок. Таким инструментом, позволяющим решать перечисленные задачи, является **система управления версиями**.

Систему управления версиями можно сравнить с сохранением состояния компьютерной игры. Перед прохождением особенно сложного игрового эпизода игрок сохраняет текущее состояние, чтобы, в случае неудачи, была возможность вернуться, а не проходить игру заново. Или игрок хочет

попробовать несколько вариантов прохождения. Он сохраняет текущее состояние и многократно возвращается к нему, пробуя новые стратегии прохождения.

Таким образом,

Система управления версиями – это инструмент, позволяющий документировать ход разработки, выполняемой коллективом программистов, и быстро переходить от одной версии кода к другой.

1.2. Краткая история развития систем управления версиями

История систем управления версиями (VCS) насчитывает три поколения. На рисунке 1.2 представлена хронология их создания.

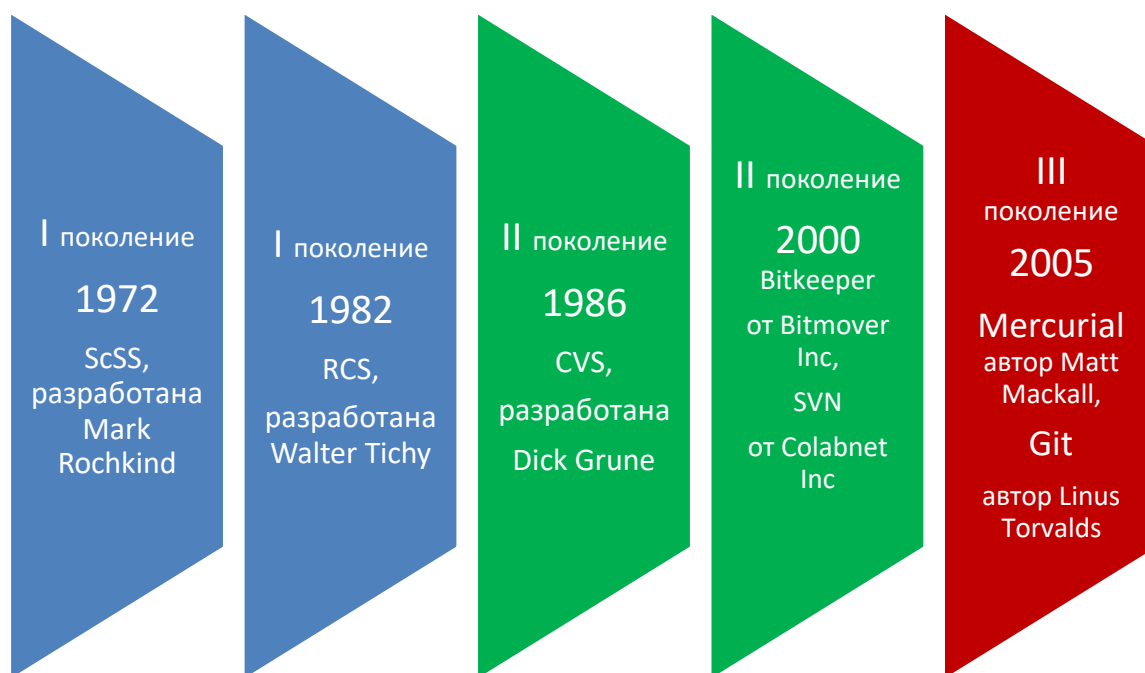


Рисунок 1.2 – Хронология создания систем управления версиями

VCS первого поколения предназначались для отслеживания изменений в одном, изолированном файле. Редактировать этот файл в каждый момент времени мог только один человек. VCS первого поколения, в основном, предназначались для использования одним программистом на одном, изолированном компьютере. В очень редких случаях организовывался сетевой доступ к одному компьютеру, и все участники проекта должны были

заходить на этот компьютер и редактировать отдельные файлы. В каждый момент времени доступ к VCS имел только один участник проекта.

Коллективный доступ к файлам появился в VCS второго поколения. Стала доступна коллективная работа. Несколько человек одновременно могли редактировать файлы и сохранять изменения в одном централизованном хранилище (репозитории). В репозитории в каждый момент времени хранилась «официальная» версия проекта. При работе с репозиторием к нему требовался постоянный сетевой доступ. Фактически репозиторий представлял собой сетевую папку, с которой одновременно работали несколько человек.

VSC третьего поколения реализованы как распределённые хранилища. Централизованного репозитория нет. Все VCS на компьютерах всех разработчиков считаются равноправными. Каждый из разработчиков работает автономно. По согласованию между разработчиками одно из хранилищ считается местом хранения текущей «официальной» версии проекта. Разработчики синхронизируют свои VSC с «официальной» версией по мере накопления готовых частей кода.

1.3. Системы управления версиями первого поколения

Первой системой управления версиями, получившей широкое распространение, была SCCS (Source Code Control System) (J. Schilling, 2020). SCCS была разработана сотрудником Bell Labs Марком Рочкингом в 1972 году и предназначалась для отслеживания версий одного, изолированного файла. Базовые идеи, заложенные в SCCS, сохраняются и в современных системах управления версиями (Oracle Corporation, 2010).

В момент добавления в SCCS нового файла, за версиями которого необходимо следить, в системе создаётся контейнер, называемый файлом истории. Имя этого контейнера состоит из имени исходного файла, к которому добавлен префикс *s*. Например, для файла с именем *file1.txt* имя контейнера будет *s.file1.txt*. Контейнер хранится в директории с именем SCCS, являющейся поддиректорией директории, в которой хранится исходный файл. В момент создания файл истории содержит копию исходного

файла и служебную информацию, предназначенную для отслеживания версий. Одной из составляющих служебной информации является контрольная сумма, предназначенная для проверки целостности файла истории. Файл истории не кодируется и не сжимается, он хранится «как есть», и существенно увеличивается в размере по мере сохранения истории изменений.

При попытке сохранить новую версию исходного файла в файл истории добавляется только фрагмент, содержащий различия между версиями. Такой фрагмент называется дельтой. Идея хранения дельт останется основополагающей для всех последующих поколений систем управления версиями. Такой способ хранения позволяет существенно уменьшить размер файла истории, даже не прибегая к сжатию. Каждая дельта хранится в специальной структуре, называемой дельта-таблицей. В дельта-таблице сохраняется текст, который добавляется или удаляется из исходного файла. К этому тексту добавляются управляющие последовательности, маркирующие начало и конец очередной дельты.

Добавление каждой новой дельты (описывающей добавление или удаление фрагмента текста) приводит к увеличению номера версии файла истории. Любую из версий можно извлечь в рабочую директорию для просмотра, редактирования или компиляции. В SCCS используется метод хранения под названием «чередующиеся дельты», который гарантирует, что время извлечения более ранних версий не будет отличаться от времени извлечения более поздних версий.

SCCS позволяет хранить историю версий для одного файла, в то время как программный проект состоит из нескольких файлов. При стандартном использовании SCCS версии файлов, входящих в один проект, будут отличаться. Чтобы синхронизировать номера версий в рамках одного проекта, используют следующее ухищрение. В те файлы, которые не редактировались, добавляют пустое изменение, вызывающее изменение номера версии и добавление пустой дельты. Надо заметить, если проект содержит много файлов, подобная синхронизация номеров их версий является очень трудоёмкой.

Когда один из программистов извлекает из файла истории версию для редактирования, сам файл истории блокируется и становится недоступным для других программистов. Это позволяет избежать перезаписи чужих правок, но существенно замедляет коллективную работу над проектом, так как

с одним файлом в один момент времени может работать только один программист.

В SCCS была реализована идея веток, ставшая в дальнейшем классической. Под ветками подразумеваются отдельные файлы истории, хранящие различные части решения одной задачи. После завершения работы над всеми частями файлы истории соединяются, образуя общий файл истории решения всей задачи.

Для понимания того, какие действия позволяет выполнить SCCS, рассмотрим некоторые команды этой системы управления версиями.

- `sccs create` – добавляет новый файл в рабочий каталог и создаёт для него файл истории в каталоге `/SCCS`;
- `sccs get` – извлекает последнюю версию рабочего файла из файла истории и помещает её в рабочий каталог для чтения (редактирование запрещено);
- `sccs edit` – извлекает последнюю версию рабочего файла из файла истории и помещает её в рабочий каталог для редактирования. Файл истории будет заблокирован и недоступен для других пользователей;
- `sccs delta` – изменения, сделанные в рабочем файле, будут добавлены в файл истории, который будет разблокирован;
- `sccs prt` – показать цепочку изменений, хранящуюся в файле истории;
- `sccs diffs` – показать различия между текущим рабочим файлом и соответствующим ему заблокированным файлом истории.

Перечисленные команды демонстрируют основные этапы работы с файлом истории: создание, просмотр последней версии, редактирование, сравнение с предыдущей версией, просмотр истории изменений, сохранение сделанных изменений.

Система управления версиями SCCS была коммерческой (платной), и из-за этого её доступность была ограничена. Как обычно случается в таких ситуациях, появилась система управления версиями с открытым кодом. Её создал Уолтер Тихи в 1982 году и назвал Revision Control System (RCS) (Nguyen, 2022). SCCS и RCS во многом похожи, их общие свойства:

- сохранение истории только для одного файла;

- версии различных файлов, составляющих один проект, могут отличаться;
- отсутствует многопользовательский режим;
- отсутствует поддержка сети.

Отличие RCS от SCCS является, скорее, её недостатком, чем усовершенствованием, вероятно, выполненным для того, чтобы избежать патентных споров с SCCS. Тем не менее, о нём необходимо рассказать, поскольку на RCS базируются системы управления версиями второго поколения.

Для сохранения изменений в RCS используется система обратных дельт. При создании первоначального файла истории в него помещается содержимое исходного файла. Процесс сохранения изменений заключается в следующем. Вычисляется дельта между новой версией и версией, хранящейся в файле истории. Версия, хранящаяся в файле истории, из него удаляется, и вместо неё записывается новая версия и вычисленная дельта. Для получения последней версии файла никаких действий не требуется, так как она всегда хранится в файле истории. Для получения предыдущей версии необходимо из текущей версии вычесть дельту. Чем более раннюю версию требуется получить, тем больше дельт необходимо вычесть. В отличие от SCCS, в RCS доступ к последней версии осуществляется очень быстро, и чем старше версия, к которой необходимо получить доступ, тем больше времени требуется затратить. В SCCS время доступа ко всем версиям одинаковое.

Список и содержимое команд RCS похожи на список и содержимое команд SCCS:

- `si` – добавляет новый файл в рабочий каталог и создаёт для него файл истории в каталоге `/RCS`. Если файл уже имеется в каталоге `/RCS`, то вычисляет дельту, удаляет старую версию и добавляет новую и соответствующую ей дельту;
- `co` – извлекает последнюю версию рабочего файла из файла истории и помещает её в рабочий каталог для чтения (редактирование запрещено);
- `co-l` – извлекает последнюю версию рабочего файла из файла истории и помещает её в рабочий каталог для редактирования. Файл истории будет заблокирован и недоступен для других пользователей;
- `merge` – производит слияние двух отредактированных копий одного исходного файла;

– rcsdiff – показывает различия между текущим рабочим файлом и соответствующим ему заблокированным файлом истории.

1.4. Системы управления версиями второго поколения

Эффективному использованию систем управления версиями первого поколения мешало отсутствие поддержки многопользовательской удалённой работы. В 1986 году Дик Грун разработал систему управления версиями под названием Concurrent Version System (CVS) (GNU, 2015). CVS открывает эру систем управления версиями второго поколения, главным отличием которых от систем управления версиями первого поколения является поддержка многопользовательского режима работы (поддержка сети). С появлением этой системы географически распределённые команды разработчиков получили эффективный инструмент коллективной работы. Программисты, находящиеся в разных точках земного шара, теперь могли взаимодействовать так же эффективно, как при нахождении в одной комнате.

CVS представляет собой сетевой интерфейс для системы управления версиями первого поколения RCS. С точки зрения правил формирования файлов истории, в CVS нет никаких отличий от RCS. Отличия заключаются в процедуре доступа к файлам истории, которая реализует модель централизованного репозитория. Под централизованным репозиторием понимается сетевое хранилище файлов, доступ к которому осуществляется через локальную или глобальную сеть.

Первый шаг по организации работы с CVS заключается в создании на удалённом сервере централизованного репозитория. Для выполнения этой работы используются специальные команды CVS. После того, как репозиторий создан, в него импортируется заготовка проекта, содержащая минимальный набор файлов. Вместо заготовки может быть импортирован проект, подлежащий доработке. Работу по созданию репозитория и импорту проекта обычно выполняет руководитель или инициатор проекта. В результате импорта в репозитории создаются файлы истории для всех файлов проекта.

После создания репозитория каждый из участников проекта имеет

возможность скачать на свой локальный компьютер файлы истории. В отличие от RCS, файлы истории в репозитории не блокируются после их скачивания, что позволяет работать с проектом неограниченному количеству программистов. Программисты редактируют файлы на своих локальных компьютерах и, по мере необходимости, фиксируют изменения в файлах истории центрального репозитория. Если один из программистов обновил файл истории в репозитории, его коллеги должны обновить соответствующий рабочий файл на своих локальных компьютерах. Сделать это необходимо перед фиксацией изменений, внесённых в тот же файл, или для того, чтобы использовать (посмотреть) результаты работы коллеги. Процедура обновления рабочих копий, как правило, автоматизирована и не требует ручной работы. Правда, иногда возникают коллизии, которые необходимо устранить в ручном режиме перед фиксацией изменений.

История не сохранила сведений о том, когда появился термин «коммит», но при описании приёмов работы с системами управления версиями второго поколения он широко используется. Значит, необходимо дать определение этого термина.

Коммит – это синоним выражения «зафиксировать изменения в файле истории».
--

Система команд CVS объединяет в себе команды системы RCS и специфические команды, обеспечивающие коллективную работу с центральным репозиторием. Вот список этих команд:

- `export CVSOOT` – задаёт корневой каталог репозитория CVS для того, чтобы его не нужно было в дальнейшем указывать в каждой команде;
- `cvs import` – импортировать директорию проекта в репозиторий CVS. Перед выполнением команды происходит автоматический переход в корневой каталог проекта;
- `cvs checkout` – копировать файлы проекта на локальный компьютер;
- `cvs commit` – коммит изменённого файла;
- `cvs add` – добавляет новый файл в репозиторий для отслеживания изменений;
- `cvs update` – обновить рабочую версию из репозитория для учёта

коммитов, сделанных другими программистами;

- `cvststatus` – показать общую информацию об извлечённой рабочей версии;
- `cvstag` – добавить информационный заголовок (тег) в файл или группу файлов;
- `cvstag -b` – создать новую ветвь в репозитории;
- `cvsc checkout -r` – извлечь ветвь из репозитория;
- `cvsupdate -j` – объединить ветвь из репозитория с локальной рабочей копией.

Дальнейшим развитием систем управления версиями второго поколения стала система Subversion (С. Michael Pilato, 2008). Как и CVS, она основана на технологии центрального репозитория.

В CVS возможна ситуация, когда коммит будет сохранён не полностью, например, из-за проблем с сетевым подключением. Частично сохранённый коммит будет рассматриваться как полный и при загрузке в рабочие версии создаст массу проблем. Для исключения таких ситуаций в Subversion предусмотрено журналирование. Перед началом записи очередного коммита в специальном файле, называемом журналом, делается запись о попытке выполнить коммит с подробной информацией о нём. Если операция была выполнена успешно, к этой записи добавляется отметка о завершении сохранения коммита. В случае сбоя отметка о завершении коммита не будет размещена в журнале. После восстановления работоспособности системы проверяется журнал на предмет наличия записей без отметок о завершении. При наличии таких записей повторяются указанные в них коммиты.

Важной особенностью Subversion является то, что коммиты могут включать в себя не один файл, а группу файлов или каталогов. Это позволяет рассматривать набор связанных изменений как один коммит, что увеличивает информативность сохранённых изменений.

В настоящее время Subversion поддерживается Apache Software Foundation и использует файловую систему FSFS (File System atop the File System). В FSFS директории являются полноправными объектами, которые возможно включать в коммит. Каждый коммит хранится в отдельном файле. Этим файлам присваиваются имена в виде последовательных числовых

идентификаторов. При первом коммите в файле сохраняется полный текст файла, при последующих – только дельты. Кроме того, коммиты сжимаются с помощью алгоритмов lz4 или zlib. Поскольку Subversion является наследницей RCS, в ней используется система обратных дельт. При большом количестве коммитов, относящихся к одному файлу, время восстановления ранних версий значительно увеличивается. Для борьбы с этим неудобством с Subversion используется соглашение о том, что после сохранения 1023 дельт снова сохраняется полная версия исходного файла.

Subversion использует не обычную систему ветвления. Шаблон репозитория содержит три директории:

- trunk;
- branches;
- tags.

Директория используется в следующих целях:

- trunk для хранения окончательных версий проекта, готовых для передачи пользователю;
- branches для хранения вложенных папок, соответствующих различным ветвям;
- tags для хранения обособленных частей проекта.

Ниже перечислены основные команды системы Subversion:

- svn create – создаёт новый пустой шаблон репозитория;
- svn import – импортировать проект в репозиторий (создать файлы истории для всех файлов проекта);
- svn checkout – загрузить рабочие версии из репозитория на локальный компьютер;
- svn commit -m – создать коммит для набора файлов и директорий;
- svn add – добавить в репозиторий новый файл для отслеживания;
- svn update – обновить рабочую версию проекта; будут скачаны и добавлены изменения, внесённые другими программистами;
- svn status – отобразить список файлов, которые изменились (полезна в связке с предыдущей командой);
- svn info – общие сведения об извлечённой рабочей копии проекта;
- svn copy – создать новую ветку путём копирования существующей;

- `svn switch` – сделать ветку рабочей директорией;
- `svn merge` – объединить ветвь из репозитория с текущей ветвью на рабочем компьютере (объединение произойдёт только на рабочем компьютере, для сохранения изменений в файле истории нужно будет сделать коммит);
- `svn log` – показать историю коммитов.

1.5. Системы управления версиями третьего поколения

Системы управления версиями второго поколения обеспечили возможность коллективной удалённой работы. Географически распределённые команды разработчиков смогли выполнять совместные проекты. Однако, некоторые недостатки, связанные с удалённой работой, всё же остались. Эти недостатки связаны с необходимостью использовать центральный репозиторий, через который происходит обмен коммитами. Если по какой-то причине центральный репозиторий недоступен, то коллективная работа над проектом становится невозможной.

Проблемы, связанные с недоступностью центрального хранилища, решаются путём его исключения из архитектуры сети. Архитектура, в которой все компьютеры могут общаться друг с другом напрямую, называется распределённой сетью. Было логично ожидать, что появится система управления версиями, основанная на распределённой сети. И она появилась. В 2005 году Линус Торвальдс создал систему управления версиями Git (Scott Chacon, 2014). Будучи создателем операционной системы Linux, Торвальдс разрабатывал Git, прежде всего, для координации работы программистов, принимающих участие в разработке ядра Linux. Со временем Git стали использовать и программисты, не связанные с работой над Linux. Сфера применения Git расширилась, и в настоящее время это наиболее популярная система управления версиями.

Как было сказано, Git является распределённой системой, центрального репозитория нет, и программисты могут обмениваться версиями разрабатываемых программ напрямую. По правде говоря, централизованное хранилище в Git обычно присутствует, но его функция отличается от функ-

ции центрального репозитория в системах управления версиями второго поколения. В Git централизованное хранилище используется, в основном, для хранения окончательных версий продукта в целом или его подсистем. Даже если централизованное хранилище используется для обмена версиями в процессе работы, оно рассматривается как участник рабочей группы.

Ключевой особенностью систем управления версиями третьего поколения и Git в частности является то, что на локальном компьютере хранятся не только рабочие файлы, но и файлы истории. В результате коммиты можно выполнять локально, даже без доступа к сети. Работая с Git, программист выполняет коммиты на локальном компьютере до тех пор, пока не будет готов поделиться результатами своей работы со всеми членами группы разработчиков или с некоторыми её участниками, работающими над одной и той же подсистемой. Как будет производиться обмен, через центральный репозиторий или непосредственно между компьютерами членов группы, решают сами программисты, участвующие в обмене.

Дельты в Git используются только для пересылки файлов в удалённые репозитории. Для создания коммитов дельты не используются, и это делает Git очень быстрым, так как в коммитах хранятся полные версии файлов, и нет необходимости тратить время на восстановление версии, применяя одну дельту за другой. Когда файл добавляется для отслеживания в Git, он сжимается с использованием алгоритма zlib, и результат хэшируется с помощью хэш-функции SHA-1. Полученный уникальный хэш однозначно соответствует содержимому файла. Все файлы хранятся в папке git/objects. Совокупность хранимых файлов называется базой объектов. Именами файлов в базе объектов являются полученные хэши. Сами сжатые файлы называются блобами. Блобы создаются каждый раз, когда новый файл добавляется в Git для отслеживания или когда сохраняется изменённая версия существующего файла.

При подготовке коммита Git создаёт промежуточный индексный файл (*staging index*). По мере добавления изменений в коммит ссылки на блобы, соответствующие этим изменениям, добавляются в индексный файл. Содержимое индексного файла представляет собой структуру, описывающую дерево. Дерево является ещё одним объектом Git наряду с блобами. В узлах дерева хранится информация о реальном имени файла, правах доступа к

файлу, имени его автора и ссылки на предыдущие и следующие версии файла. Также узел дерева может хранить ссылку на другое дерево, реализуя инструмент ветвей. Когда все данные с историей изменений подготовлены, даётся команда на создание коммита. Коммит является третьим объектом Git (первые два – блов и дерево). Коммит хранится в базе коммитов и содержит дату создания, комментарий, имя автора, адрес его электронной почты и ссылку на объект дерева. Каждый коммит, кроме того, содержит ссылку на родительский коммит, в результате чего создаётся история изменений проекта.

В целях экономии места все объекты Git (бловы, деревья и коммиты) сжимаются, дельты не используются. Они называются свободными объектами (*loose objects*). Однако есть исключение. Передача коммитов в удалённый репозиторий, хранение очень больших наборов объектов или выполнение команды сборки мусора вызывают переупаковку объектов и создание пакетных файлов. Пакетные файлы сжимаются и имеют расширение *pack*. Для каждого пакетного файла создаётся индексный файл с расширением *idx*. Индексный файл содержит ссылки на упакованные объекты и их расположение в пакетном файле.

При выполнении обмена коммитами по сети передаются пакетные файлы. После получения пакетного файла адресатом он распаковывается, и в дальнейшем для работы используются свободные объекты.

Набор команд Git более обширный и разнообразный, чем у систем первого и второго поколений. Ниже приведены некоторые из них:

- `git init` – текущая папка на локальном компьютере инициализируется как репозиторий Git, в ней создаётся скрытый каталог `git`;
- `git clone` – на локальный компьютер загружается репозиторий, находящийся по адресу, указанному в команде. Существенное отличие от систем управления версиями предыдущих поколений заключается в том, что загружаются файлы истории, без преобразования в рабочие файлы;
- `git add` – создаёт для нового файла блов и добавляет информацию о нём в дерево готовящегося коммита;
- `git commit` – создать коммит и поместить его в базу коммитов;
- `git status` – показать статус файлов в рабочем каталоге (добавленные в коммит, изменённые, не отслеживаемые и т.п.);
- `git branch` – создаёт новую ветвь;

- `git checkout` – извлечь ветвь в рабочий каталог;
- `git merge` – объединить ветвь с ранее извлечённой в рабочий каталог;
- `git pull` – получить из удалённого репозитория обновления, отсутствующие в рабочем каталоге;
- `git push` – объединить файлы коммитов для текущей ветви в пакетный файл и передать пакетный файл в удалённый репозиторий;
- `git log` – показать историю коммитов для активной ветви;
- `git stash` – сохранить все незафиксированные изменения в кэш, чтобы извлечь их позже.

В 2005 году появилась ещё одна система управления версиями третьего поколения под названием Mercurial. Она была создана Мэттом Макколлом и, как и Git, предназначалась для управления версиями ядра Linux. В конкурентной борьбе между Mercurial и Git победил Git. В настоящее время Mercurial является второй по распространённости системой управления версиями, хотя и используется гораздо реже по сравнению с Git.

Mercurial, как и Git, является распределённой системой управления версиями. Она позволяет большому количеству программистов работать с локальной версией репозитория, сохраняя изменения по мере необходимости. В Mercurial тоже используются сжатие и хэширование, но архитектура системы коренным образом отличается от Git. Идеологически Mercurial ближе к CVS и Subversion.

Когда новый файл добавляется для отслеживания, в специальном скрытом каталоге `hg/store/data` создаётся соответствующий ему файл `revlog` (журнал изменений). Этот файл можно рассматривать как модификацию файла истории системы CVS. Если Git создаёт блоб для каждой модификации файла, Mercurial создаёт новую запись в журнале истории. Для экономии места каждая новая запись содержит только дельту. Когда число дельт достигает порогового значения, новая версия файла сохраняется целиком. Это позволяет уменьшить время извлечения рабочего файла для ранних версий.

Пока всё очень похоже на то, как устроены более ранние системы управления версиями. Но есть и отличия. Имя каждого `revlog` совпадает с именем файла, история изменений которого в нём отслеживается. Для большинства файлов создаются два файла `revlog` с одинаковыми именами, но с

разными расширениями. У одного из них расширение `i`, у другого `d`. Расширение `d` имеет `revlog`, содержащий дельты, а расширение `i` имеет индексный файл, используемый для быстрого отслеживания версий внутри файла с индексом `d`. Для небольших файлов, содержащих малое количество дельт, дельты хранятся внутри файла с индексом `i`, файл с индексом `d` при этом не создаётся. Записи `revlog` сжимаются и хэшируются. Хэши являются идентификаторами дельт и называются `nodeid`.

При каждом коммите Mercurial создаёт файл манифеста, который тоже носит название `revlog`, но это `revlog` другого типа. Манифест содержит список файлов, включённых в коммит, и список `nodeid`, указывающих на изменения, фиксирующиеся в данном коммите. Все записи манифеста сжимаются и хэшируются. Хэши, идентифицирующие записи манифеста, тоже называются `nodeid`.

Третий тип `revlog`, который использует Mercurial, называется `changelog` (журнал изменений). В журнале изменений для каждого коммита содержатся следующие данные:

- `nodeid` манифеста, который содержит информацию обо всех версиях файлов, доступных на момент включения записи в журнал изменений;
- один или два `nodeid` родительских коммитов. Если коммит не является коммитом слияния двух ветвей, то сохраняется один `nodeid` родителя. В противном случае у коммита два родителя, и поэтому сохраняются два `nodeid`. Такая структура позволяет отслеживать изменение проекта во времени;
- имя автора коммита;
- дата коммита;
- комментарий к коммиту.

Каждая запись журнала изменений тоже хэшируется, и хэш называется `nodeid` журнала изменений.

По смыслу команды Mercurial, приведённые ниже, во многом похожи на команды Git:

- `hg init` – инициализировать текущую папку как репозиторий Mercurial (создать скрытый каталог `hg` и его структуру);
- `hg clone` – загрузить копию сетевого репозитория Mercurial на локальный компьютер;
- `hg add` – добавить новый файл для отслеживания изменений (создаются соответствующие `revlog` с индексами `i` и `d`);

- hg commit – зафиксировать изменения, создать манифест и запись в журнале истории;
- hg status – показать информацию о состоянии файлов текущего каталога (изменённые файлы, не отслеживаемые файлы и т.д.);
- hg update – извлечь указанную ветвь в рабочий каталог;
- hg merge – объединить указанную ветвь с ветвью, находящейся в рабочем каталоге;
- hg pull – скачать новые версии из удалённого репозитория, но не распаковывать их в рабочую директорию;
- hg push – перенести новые версии из локального каталога в удалённый репозиторий;
- hg log – показать историю коммитов и соответствующих им комментариев для текущей ветви.

1.6. Встроенные системы управления версиями

В предыдущих разделах рассмотрены системы управления версиями, представляющие собой самостоятельный, отдельно стоящий инструмент. Эти системы используются в процессе профессиональной разработки программного обеспечения. Наряду с профессиональными существуют системы управления версиями, встроенные в программные инструменты, используемые для решения офисных и даже повседневных задач. Большинство работающих с этими инструментами людей даже не подозревают, что пользуются системами управления версиями. Рассмотрим несколько примеров.

Сервисы синхронизации файлов между устройствами, такие как Dropbox, отслеживают версии файлов, с которыми работают пользователи. Процесс выглядит следующим образом. Файловая система компьютера автоматически сохраняет информацию о времени последнего изменения файла. Сервис синхронизации периодически сравнивает файлы, находящиеся на сервере, с файлами на локальном компьютере. Если файлы отличаются и файл, находящийся на сервере, старше файла, находящегося на локальном компьютере, то выполняется процесс синхронизации. Файл, находящийся на сервере, становится частью истории изменений, а на его место скачива-

ется файл с локального компьютера. Dropbox сохраняет файл в истории каждый раз, когда пользователь нажимает кнопку «Сохранить». Файл сохраняется целиком и называется снимком. Благодаря такой процедуре Dropbox в состоянии восстановить любую версию файла. Недостаток такого подхода очевиден – значительный расход дискового пространства. Возможно, с точки зрения экономии дискового пространства было бы эффективнее хранить дельты, как это делается во многих профессиональных системах управления версиями. Но в этом случае время восстановления версий будет значительным. Экономия дискового пространства или малое время восстановления предыдущих версий? Эта дилемма возникает всегда при разработке систем управления версиями.

Системы управления версиями являются неотъемлемой частью текстовых редакторов, особенно онлайнowych. Google Docs выполняет процедуру автосохранения приблизительно каждые 5 секунд. Если файл изменился, то делается снимок, который сохраняется в истории изменений.

История изменений иначе называется историей ревизий. Ревизия более общее понятие, чем снимок. Ревизия – это любое зафиксированное изменение в файле, как правило, сохранённое в истории изменений. Ревизия может выполняться с использованием снимка или дельты. Ревизия – это контейнер для снимка или дельты, помимо самого снимка (дельты), она содержит дополнительную информацию, например, дату сохранения, имя автора и т.п. Процесс переключения между ревизиями имеет своё название в рамках систем управления версиями. Этот процесс называется checkout. Например, в Git существует команда `git checkout`.

Современные текстовые редакторы являются инструментом коллективной работы над документами. Любой коммерческий договор проходит несколько стадий согласования и зачастую возвращается к автору или рецензенту несколько раз. Разумеется, сотрудник, получивший документ с исправлениями (замечаниями), прежде всего, хочет узнать, какие правки в него были внесены. Для того чтобы увидеть изменения, необходимо сравнить ревизии. Если ревизия была сохранена с использованием дельты, то достаточно извлечь нужную дельту и отобразить её. Если ревизия хранится в виде снимков, то необходимо вычислить дельту перед отображением. Например, Microsoft Word хранит историю ревизий и позволяет сравнивать версии документа, отображая в наглядной форме, что и где изменилось. Для

Microsoft Word существуют два вида ревизий. Одна из них называется режимом записи исправлений и сохраняет только удаления или изменения текста. В режиме записи исправлений возможно сравнивать оригинал и изменённый документ, располагая их в двух соседних окнах. Такая история ревизий хранится непосредственно в документе в виде дельт. Другой вид истории ревизий сохраняет снимки и позволяет восстановить любую из предыдущих версий. Для того чтобы воспользоваться этим режимом, необходимо подключиться к One Drive. При сохранении снимков Microsoft Word обеспечивает сравнение версий, вычисляя дельты.

Какой бы из систем управления версиями вы ни воспользовались, рабочий процесс будет выглядеть одинаково:

- создание репозитория;
- добавление новых файлов;
- коммит;
- внесение изменений в файлы (редактирование, добавление, удаление);
- коммит;
- и так далее до завершения работы над проектом (документом).

Термины «коммит» и «ревизия» являются синонимами. В данной работе в дальнейшем будет использоваться термин «коммит».

С практической точки зрения к коммиту предъявляются два следующих требования.

1. Коммит должен иметь комментарий. В большинстве систем управления версиями коммит невозможно сохранить, если комментарий отсутствует. Комментарий состоит из краткого заголовка, описывающего суть изменений, зафиксированных в коммите. За заголовком, если требуется, следуют подробные объяснения. Это правило обеспечивает самодокументированность истории версий и способствует эффективной работе с ней.

2. Коммит должен фиксировать решение одной задачи, как можно более полное. Это правило называется атомарностью коммита. Как и первое правило, оно способствует эффективной работе с историей версий, кроме того, помогает понять, какие коммиты необходимо отменить, чтобы вернуться к предыдущей версии, если это необходимо.

Контрольные вопросы

1. Какие задачи решает система управления версиями?

2. Дайте сравнительную характеристику систем управления версиями 1-го, 2-го и 3-го поколений.
3. Опишите принципы формирования файла истории. Что такое дельта (прямая и обратная)?
4. Раскройте понятие термина «ветка» по отношению к системе управления версиями.
5. Что такое «централизованный репозиторий»? Какова его роль в развитии систем управления версиями?
6. Что такое «коммит»? Какие проблемы, возникающие при использовании коммитов, решает «журналирование»?
7. Что такое «распределённый репозиторий»? Опишите его преимущества по отношению к «централизованному репозиторию».
8. Раскройте смысл термина «встроенная система управления версиями». Приведите пример встроенной системы управления версиями.

2. СИСТЕМА УПРАВЛЕНИЯ ВЕРСИЯМИ GIT

Вторая глава посвящена системе управления версиями Git. Будут подробно рассмотрены все этапы использования репозитория Git от создания до управления ветками. В ходе описания команд Git будет продемонстрировано внутреннее устройство репозитория. Из этой главы читатель получит полное представление о Git как инструменте управления версиями и сможет начать использовать его в своей работе. Дальше практика, практика и ещё раз практика.

2.1. Особенности и преимущества Git

Git является распределённой системой управления версиями. Это значит, что полная версия репозитория находится на локальном компьютере. Нет необходимости постоянно сохранять изменения в удалённом репозитории и загружать из него обновления. Всё, что происходит на локальном компьютере, остаётся только на локальном компьютере. Во внешний мир уходят только проверенные версии, поэтому никто не узнает о неудачах, ошибках и откровенных ляпах. Копии репозитория могут находиться на нескольких компьютерах: домашнем, офисном, дачном, и на всех компьютерах будут одни и те же версии файлов. Если есть необходимость выполнить работу на «новом» компьютере, репозиторий может быть перенесён на него очень быстро. В общем, полная свобода перемещений.

Git даёт полный контроль над версиями. Коммиты можно удалять, изменять, менять местами, объединять, разделять, перемещать между ветками.

Основной инструмент Git – это **ветки**. Ветки создаются для проверки решения подзадач, для выбора одного варианта из нескольких, для помощи коллеге, для сохранения идеи «на будущее». Создать ветку, переключиться между ветками, слить ветки, удалить ветку – эти операции выполняются постоянно в ходе работы над проектом. Ветки – то, ради чего существует система управления версиями. **Ветки и есть версии.**

Git предоставляет гибкий инструмент управления коммитами.

Перед созданием коммита можно посмотреть, что в него войдёт и отредактировать список файлов, при необходимости. Можно посмотреть историю коммитов для одного файла или для всей ветки. Сравнить версии файла в различных коммитах. Удалить коммит, восстановить рабочую версию из коммита.

Git позволяет заморозить изменения без создания коммита. Если нужно срочно переключиться на другую ветку, а текущая ветка ещё не готова для коммита, то её можно заморозить. Изменения не будут потеряны, и не придётся создавать «технологический коммит», не несущий смысловой нагрузки.

Git позволяет организовать работу в команде. Каждый программист работает в своей ветке, что ускоряет разработку за счёт разделения её на несколько потоков. Перед добавлением ветки в проект она отправляется на проверку и рецензию. При необходимости код, содержащийся в ветке, дорабатывается. В результате в проект попадает только проверенный код. Программисты видят результаты работы коллег и могут обмениваться идеями, устраивать консультации, совместные обсуждения, оказывать помощь.

Для Git существует большое количество бесплатных удалённых репозиторийев, таких как GitHub, GitFlic, BitBucket.

2.2. Установка и настройка Git для Windows

Для использования Git его необходимо установить на компьютер и настроить. Для установки Git на компьютер, работающий под управлением операционной системы Windows, необходимо скачать дистрибутив с официального сайта Git: www.git-scm.com, запустить его и выбрать необходимые опции, предлагаемые установщиком. Здесь будут пояснены две наиболее важные опции. Не беда, если в процессе установки будут выбраны «не те» опции, их можно будет изменить после установки с помощью команды `git config`.

Выбор имени основной ветки. Git предлагает имя для основной ветки, но лучше придумать своё. В этом случае, при обновлении версии Git, имя основной ветки не изменится. Выберите опцию “Override the default branch name for new repositories” и введите имя основной ветки в поле ввода.

Настройка запуска Git из консоли. Выберите опцию “Git from the command line and also from 3rd-party software”. Это позволит запускать Git не только из консоли Git Bash, но и из консоли Windows и других приложений (например, сред разработки).

После завершения установки на компьютере появится папка, содержащая программы:

- Git Bash – оболочка работы с Git, основанная на Bash;
- Git CMD – окно командной строки Windows для работы с Git;
- Git FAQs – ссылка на сайт с ответами на наиболее частые вопросы про Git;
- Git GUI – графическая оболочка для работы с Git;
- Git release notes – ссылка на сайт с информацией о текущей и предыдущих версиях Git.

Что использовать для работы с Git: Git Bash, Git CMD, Git GUI – каждый из пользователей должен решить самостоятельно. В настоящем издании для демонстрации примеров будет использоваться Git Bash, поскольку Bash является оболочкой, доступной и в Windows, и в Linux, что делает примеры воспроизводимыми в различных операционных системах.

2.3. Основные понятия

Ниже приведено формальное описание терминов, которые будут использоваться в процессе изложения материала.

1. **Репозиторий.** Любая папка файловой системы может быть преобразована в репозиторий Git. После преобразования внутри этой папки появится папка `.git/`, содержащая дерево изменений проекта.

2. **Рабочая директория.** Все папки и файлы внутри репозитория за исключением `.git/`. С файлами, находящимися в рабочей директории, происходят основные изменения в рамках работы над проектом.

3. **Индекс.** Файл, расположенный в папке `.git/`, содержит список файлов, подготовленных для добавления в коммит.

4. **Коммит.** Файл, который хранит изменённые файлы, имя автора коммита, время создания коммита. Каждый коммит имеет уникальное имя,

позволяющее вернуть рабочую директорию в состояние, которое было до внесения изменений, зафиксированных в коммите.

5. **Указатель.** Ссылка, которую использует Git или программист, чтобы сослаться на коммит.

6. **Ветка.** Последовательность коммитов. Для ссылки на ветку используется указатель на последний коммит ветки. В репозитории изначально содержится одна ветка. В дальнейшем может быть добавлено неограниченное количество независимых веток.

2.4. Настройка Git, команда `git config`

После установки Git встраивается в операционную систему и влияет на её поведение на системном, глобальном и локальном уровне. В соответствии с этим, существуют три файла настроек:

1. **Системные.** Распространяются на всех пользователей операционной системы. Файл с системными настройками Git называется `gitconfig` и расположен в директории `C:\Program Files\Git\etc\`.

2. **Глобальные.** Распространяются на все репозитории, созданные под логином пользователя. Файл с глобальными настройками Git называется `.gitconfig` и расположен в директории `C:/User/<имя пользователя>/`.

3. **Локальные.** Влияют на настройки одного репозитория. Файл с настройками называется `config` и расположен в директории `.git/` репозитория.

Для изменения настроек можно отредактировать файл конфигурации в текстовом редакторе или воспользоваться командой `git config`. Эта команда позволяет изменить или посмотреть значение параметров конфигурации Git.

Формат команды `git config`:

`git config <ключ> <параметр> <значение>`

Ключ задаёт область действия параметра. Возможные значения:

`--global` – указывает на то, что будут изменяться настройки файла `gitconfig`;

`--system` – указывает на то, что будут изменяться настройки файла `.gitconfig`;

--local – указывает на то, что будут изменяться настройки файла config (это значение ключа установлено по умолчанию).

Параметр указывает, какая настройка будет изменена. Возможные значения:

- user.name – имя пользователя;
- user.email – адрес электронной почты пользователя.

Значение – текстовая строка, задающая значение параметра.

В следующем примере изменяются имя пользователя и его e-mail в глобальных настройках.

```
$ git config --global user.name 'Git_Book'
$ git config --global user.email 'git_book@home.com'
```

После выполнения приведённых команд файл .gitconfig будет выглядеть следующим образом:

```
[user]
  name = Git_Book
  email = git_book@home.com
[credential "https://gitflic.ru"]
  provider = generic
```

Поля name и email содержат значения, заданные соответствующими командами Git, приведёнными выше.

2.5. Создание локального репозитория, команда git init

Команда git init создаёт новый локальный репозиторий в той папке, в которой она была выполнена.

Формат команды git init:

git init <параметр>

Один из возможных параметров задаёт имя главной ветки проекта:

--initial-branch=<имя ветки>

В примере ниже создаётся репозиторий в папке Learn_git. Имя главной ветки этого репозитория Main_branch.

```
/c
$ mkdir Learn_git
/c
$ cd Learn_git
/c/Learn_git
$ git init --initial-branch='Main_branch'
Initialized empty Git repository in C:/Learn_git/.git/
/c/Learn_git (Main_branch)
```

Признаком того, что папка является репозиторием Git, будет название ветки в скобках после указания пути к папке.

В папке Learn_git командой git init была создана папка .git со следующей структурой вложенных папок.



Назначение этих папок и их содержимое будет поясняться далее, по мере изучения команд Git и создаваемых ими объектов.

2.6. Логическая структура Git

Логически репозиторий состоит из трёх частей: рабочая директория; файлы, подготовленные для включения в очередной коммит (Staging area); хранилище файлов предыдущих версий (архив).

В рабочей директории находятся файлы, в которые программист вносит текущие изменения.

Staging area включает изменённые файлы, которые планируется включить в очередной коммит.

Архив содержит все предыдущие версии файлов проекта.

На рисунке 2.1 показана диаграмма «перемещения» файлов между логическими частями репозитория.

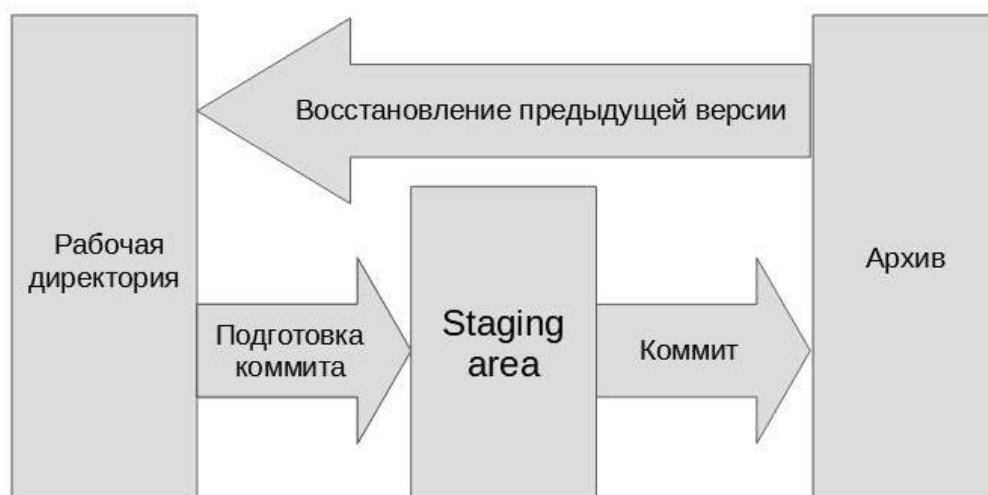


Рисунок 2.1 – Диаграмма перемещения файлов

Работа программиста, использующего Git, выглядит следующим образом. Редактирование файлов в рабочей директории. Git следит за тем, какие файлы были изменены, и может сформировать список изменённых файлов. Когда программист решит, что файл содержит изменения, которые необходимо зафиксировать, он даёт Git команду подготовить файл для сохранения в архиве. Git преобразует файл в формат, удобный для хранения, и размещает его в архивной папке `.git`. Этот файл находится в промежуточном состоянии. Он уже в архиве, но ещё не включён в историю изменений. Это Staging area. Когда все файлы, включающие логически завершённую часть решаемой задачи, размещены в Staging Area, программист даёт Git команду включить файлы из Staging area в историю изменений. Git формирует структуры, описывающие произошедшие изменения, и сохраняет их в папке `.git`.

2.7. Состояния файлов, команда `git status`

Файлы, находящиеся в рабочей директории, могут находиться в одном из двух состояний: отслеживаемые и не отслеживаемые. В число отслеживаемых входят все файлы, которые хотя бы один раз попадали в Staging area. Про файлы, ни разу не попадавшие в Staging area, Git знает только то, что они существуют, и не отслеживает происходящие в них изменения.

Отслеживаемые файлы делятся на три категории: неизменённые, изменённые, подготовленные к коммиту. На рисунке 2.2 показана диаграмма

перехода файлов из одного состояния в другое.

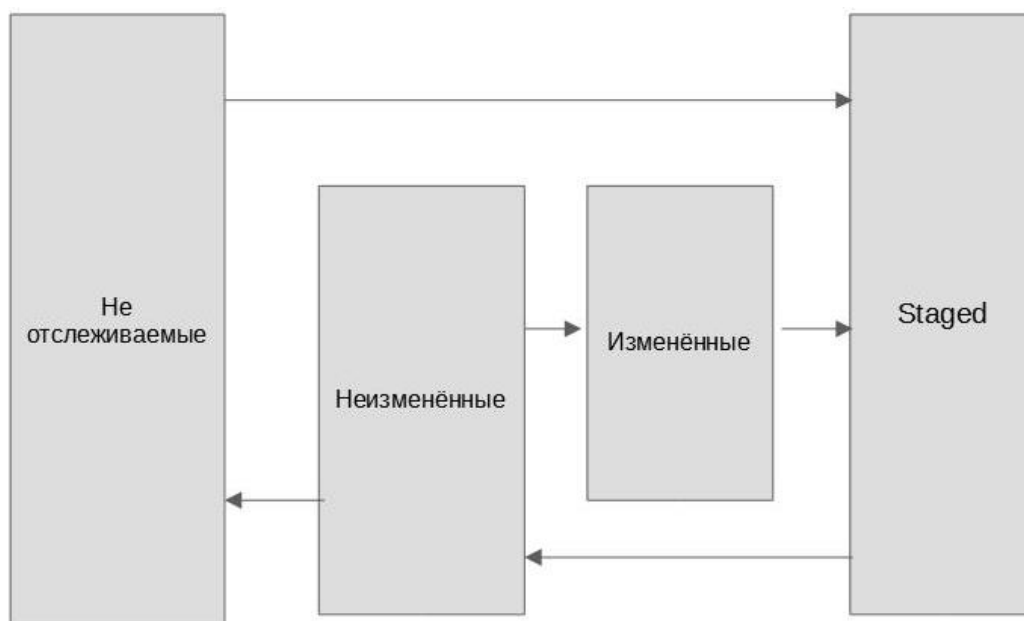


Рисунок 2.2 – Диаграмма изменения состояния файла

Для того чтобы файл стал отслеживаемым, его необходимо поместить в Staging area. Например, для того чтобы файл, скопированный из другой папки, стал частью репозитория, его необходимо включить в коммит (даже, если в этот файл не будут внесены никакие изменения).

Для того чтобы файл стал не отслеживаемым, его необходимо удалить из рабочей директории. После удаления файл останется в архиве, и его можно будет восстановить.

Файл становится изменённым, если в него внесены изменения.

Файл становится Staged (подготовленным к коммиту) после соответствующей команды программиста. При этом новая версия файла будет сохранена в архиве, но не внесена в историю изменений.

После того, как был выполнен коммит, все файлы из Staged area переходят в состояние неизменённых и остаются отслеживаемыми.

Для определения состояния файлов, находящихся в рабочей директории, предназначена команда `git status`.

Формат команды `git status`:

`git status <флаги>`

Возможные значения флагов:

- s – короткий формат вывода информации;
- b – показывать название ветви даже при коротком формате вывода;

-v – для изменённых файлов показывать текст, который был изменён.

Для примера, рассмотренного ранее, в директорию Learn_git были добавлены два файла (File1.txt, File2.txt) и пустая директория Dir1. File1 содержит текст: Example1. File2 содержит текст: Example2. Ниже показан результат выполнения команды git status после добавления файлов и директории.

```
$ git status
On branch Main_branch

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  File1.txt
  File2.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Интерпретация информации, выведенной Git по команде git status, следующая:

- текущая ветка – Main_branch;
- до сих пор не было сделано ни одного коммита;
- имеются не отслеживаемые файлы File1.txt, File2.txt. Git рекомендует добавить их в Staging area с помощью команды add;
- сообщается, что ничего не добавлено в коммит, но имеются не отслеживаемые файлы.

Перед тем, как выполнять действия по добавлению файлов в Staging area и затем в коммит, необходимо рассмотреть, как Git хранит внутреннюю информацию.

2.8. Объекты Git

Всё, что сохраняет Git в архиве, называется объектом. Объекты хранятся в директории .git/objects. В настоящий момент, для рассматриваемого примера, директория .git/objects содержит две директории info и pack (см. раздел 2.5) и ни одного файла с объектами.

Объекты в Git бывают трёх типов:

1. **Blob** (binary large object) – большой двоичный объект. Из названия следует, что это двоичный файл. Blob создаётся для каждого файла и всех

его версий, сохранённых в архиве. Blob содержит имя файла и его содержимое в сжатом виде, Blob формируется, когда файл добавляется в Staging area.

2. **Tree** (дерево) – двоичное представление графа, описывающего связи между файлами, хранящимися в архиве. Граф формируется для каждой директории, включённой в архив. Формирование графа происходит в процессе выполнения коммита. Граф содержит информацию о том, какие файлы и директории содержатся в директории, которую описывает граф. Описание хранится в виде имён blob-объектов и имён других объектов типа Tree.

3. **Коммит** – содержит информацию об авторе коммита, дате и времени выполнения коммита, комментарий и ссылку на объект Tree, который описывает корневую директорию репозитория.

Ещё один элемент внутреннего устройства Git, не являющийся объектом, это двоичный файл index (индекс). Этот файл хранит ссылки на blob-объекты, которые добавляются в Staging area. Фактически файл index и есть Staging area. В дальнейшем термин «добавить в Staging area» будет применяться как синоним термина «добавить в индекс». Во время выполнения команды add, упоминавшейся в примере прошлого раздела, происходит формирование blob-объекта для добавляемого файла. Blob-объект добавляется в директорию .git/objects, а имя этого blob-объекта записывается в файл index.

Blob-объект формируется по следующему алгоритму.

1. На первом шаге формируется текстовый файл, который начинается с ключевого слова blob. После слова blob через пробел записывается длина исходного текстового файла в байтах и нулевой байт (\0). После нулевого байта записывается содержимое исходного текстового файла. В рассмотренном выше примере файл File1.txt содержит текст Example1. Для этого файла результат выполнения первого шага создания blob-объекта будет выглядеть следующим образом: blob 8\0Example1.

2. Для полученного на первом шаге файла с помощью алгоритма SHA-1 формируется хеш-значение, состоящее из 40 шестнадцатеричных цифр. Первые две цифры хеш-значения являются именем подкаталога каталога .git/objects, в котором будет храниться blob-объект. Остальные 38 шестнадцатеричных цифр являются именем blob-объекта. Хеш-значение используется для того, чтобы все хранящиеся в архиве файлы имели различные имена хранения. Даже если два файла имеют одинаковое исходное имя, но различное содержание, их хеш-значения будут различными. Первые две

цифры хеш-значения используются для обозначения директории для того, чтобы файл было легче искать.

3. Текстовый файл, полученный на первом шаге, сжимается. После сжатия получается двоичный файл, и blob-объект начинает оправдывать своё название.

2.9. Добавление файла в индекс, команда `git add`

Команда `git add` выполняет три действия:

- Создаёт blob-объект для выбранного файла.
- Сохраняет blob-объект в директории `.git/objects`.
- Записывает сведения о blob-объекте в файл `index`.

Файл `index` хранится в директории `.git/`. В каждой строке файла хранится служебная информация, хеш-значение, соответствующее файлу, который планируется добавить в коммит, и имя файла. Все три части разделены пробелами. Файл `index` хранится в сжатом виде.

Формат команды `git add`:

```
git add <имя файла> <флаги>
```

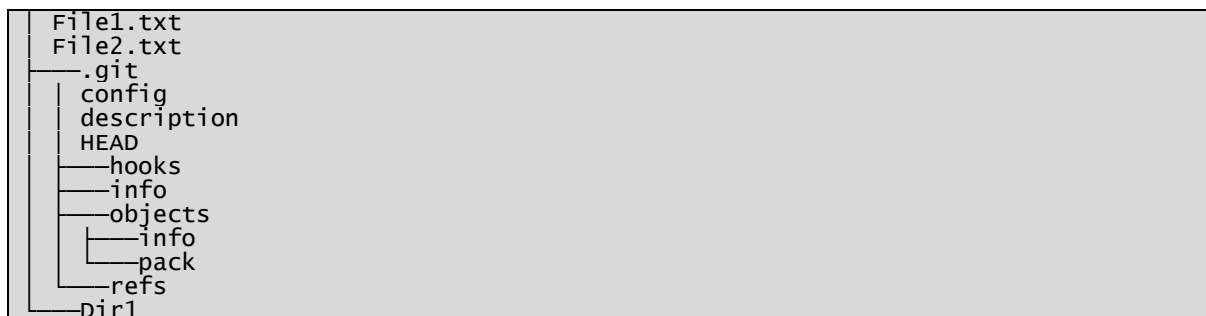
В имени файла допускается использовать символы групповых операций для добавления нескольких файлов в одной команде.

Некоторые из возможных флагов:

-А или `-all` – в индекс будут добавлены все файлы, находящиеся в рабочей директории.

Продолжение примера.

После добавления файлов `File1.txt` и `File2.txt` репозиторий содержит следующие файлы и директории:



Ниже показан результат последовательного выполнения команд:

- `git add File1.txt;`

- git ststus.

```
$ git add File_1.txt
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   File1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    File2.txt
```

Файл File_1.txt поменял статус. Теперь он находится в числе файлов, которые планируется добавить в коммит. Файл File_2.txt по-прежнему не отслеживаемый.

Теперь директория .git/ выглядит так:

```
File1.txt
File2.txt
.git
├── config
├── description
├── HEAD
├── index
├── hooks
├── info
├── objects
│   ├── 84
│   └── 9327df401a74dd0148b99b532d290f7da80eae
│       ├── info
│       └── pack
├── refs
└── Dir1
```

Произошли следующие изменения:

- В директории .git/ появился файл index.
- В директории ./git/objects появилась директория 84 и в ней файл 9327df401a74dd0148b99b532d290f7da80eae.

Файл index – двоичный сжатый файл. Посмотреть его содержимое обычными средствами Windows не представляется возможным. Однако это возможно сделать, используя команду git: git ls-file –stage.

```
$ git ls-files --stage
100644 849327df401a74dd0148b99b532d290f7da80eae 0 File1.txt
```

Файл index содержит одну строку, соответствующую одному файлу, подготовленному для коммита (см. результат команды git ststus).

Строка состоит из четырёх секций:

- Служебная информация 100644 говорит о том, что в строке описывается обычный файл.

- 849327df401a74dd0148b99b532d290f7da80eae – хеш-значение.

- 0 – признак отсутствия конфликтов в готовящемся коммите.

- File1.txt – имя файла, который предполагается добавить в коммит.

В процессе выполнения команды `git add File1.txt` был создан blob-объект с хеш-значением 849327df401a74dd0148b99b532d290f7da80eae. Этот blob-объект хранится в директории `./git/objects/84` под именем 9327df401a74dd0148b99b532d290f7da80eae (см. дерево директории выше).

Для просмотра blob-объекта используется команда `git:`

`git cat-file -p <имя blob-объекта>`

```
/c/Learn_git (Main_branch)
$ git cat-file -p 849327df401a74dd0148b99b532d290f7da80eae
Example1
```

Таким образом, при выполнении команды `git add File1.txt` произошло следующее:

- Для файла `File1.txt` был создан blob-объект с хеш-значением 849327df401a74dd0148b99b532d290f7da80eae;

- blob-объект был записан в файл с именем 9327df401a74dd0148b99b532d290f7da80eae и размещён в директории `./git/objects/84`;

- в файл `index` была добавлена строка:

100644 849327df401a74dd0148b99b532d290f7da80eae 0 File1.txt

Добавим в индекс файл `File2.txt` и определим статус файлов в репозитории.

```
/c/Learn_git (Main_branch)
$ git add File2.txt

/c/Learn_git (Main_branch)
$ git status
On branch Main_branch

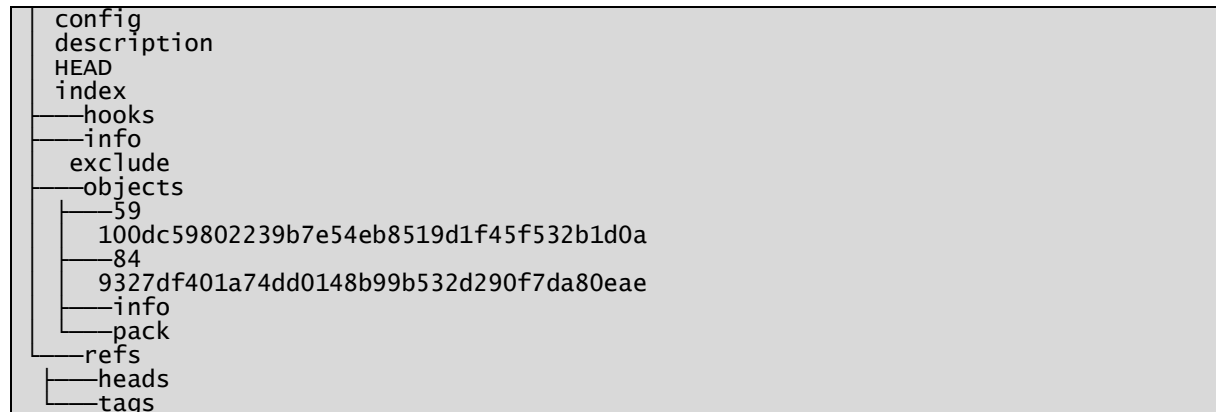
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   File1.txt
    new file:   File2.txt
```

Теперь оба файла являются отслеживаемыми и занесены в индекс, который выглядит так:

```
$ git ls-files --stage
100644 849327df401a74dd0148b99b532d290f7da80eae 0 File1.txt
100644 59100dc59802239b7e54eb8519d1f45f532b1d0a 0 File2.txt
```

В директории `.git/objects` появилась директория `59`, в которой содержится blob-объект, соответствующий файлу `File2.txt`.



Изменим содержимое файла `File2.txt` и снова выполним команду `git status`.

```
/c/Learn_git (Main_branch)
$ git status
On branch Main_branch

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   File1.txt
    new file:   File2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   File2.txt
```

Файлы `File1` и `File2.txt` по-прежнему отслеживаемые и находятся в файле индекса. В то же время про файл `File2.txt` сказано, что он изменён и не включён в индекс. Таким образом, создаётся впечатление, что файл `File2.txt` находится сразу в двух состояниях. Для того чтобы объяснить, что произошло, сначала необходимо сказать, как Git определяет, что файл изменился. Для сравнения файлов Git сравнивает хеш-значения этих файлов. Если значения разные, значит, содержимое файлов различно. При выполне-

нии команды `git status` Git обращается к каждому из файлов в рабочей директории и определяет, были ли они включены в предыдущие коммиты или в текущий файл индекса. Если файл ни разу не был включён в коммит и не содержится в текущем файле индекса, значит, для него не вычислялось хеш-значение, и сравнивать текущее содержимое файла не с чем. Файл не отслеживаемый. Если предыдущее хеш-значение существует, то оно сравнивается с текущим и определяется, был ли файл изменён.

При выполнении команды `git status` в рассматриваемом примере Git обнаружил, что для файла `File2.txt` ранее вычислялось хеш-значение (файл находится в Staging area). Предыдущее и текущее хеш-значения различны, следовательно, после внесения файла `File2.txt` в индекс он был изменён. Об этом Git и сообщил в результатах команды `git status`. Если при таком состоянии репозитория выполнить коммит, то в архиве будет сохранено неизменённое содержимое файла `File2.txt` (то, которое хранится в директории `.git/objects/59`). Таким образом, Git хранит в папке `.git/objects` все версии файлов, но включает в коммит только последнюю из них.

Для рассматриваемого примера существуют три варианта действий:

1. Выполнить коммит и сохранить в архиве старое содержание файла `File2.txt`.
2. Восстановить содержимое файла `File2.txt` из blob-объекта.
3. Выполнить команду `git add` для файла `File2.txt` и добавить в индекс ссылку на его новое содержание.

Выберем третий вариант.

```
/c/Learn_git (Main_branch)
$ git add File2.txt

/c/Learn_git (Main_branch)
$ git status
On branch Main_branch

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   File1.txt
    new file:   File2.txt

/c/Learn_git (Main_branch)
$ git ls-files --stage
100644 849327df401a74dd0148b99b532d290f7da80eae 0 File1.txt
100644 36d9b23df2cdcd91759d69a291d04beddab2b091 0 File2.txt
```

Теперь `File1.txt` и `File2.txt` находятся в Staging area, но для файла

File2.txt был создан новый blob-объект, который хранится в директории .git/objects/36.

Закончим готовить тестовый репозиторий к коммиту. Создадим в директории Dir1 файл File3.txt, содержащий текст Example3. Добавим файл File3.txt в индекс и посмотрим, что получилось.

```
/c/Learn_git (Main_branch)
$ cd Dir1
/c/Learn_git/Dir1 (Main_branch)
$ echo Example3 > File3.txt
/c/Learn_git/Dir1 (Main_branch)
$ git add File3.txt
/c/Learn_git/Dir1 (Main_branch)
$ cd ..
/c/Learn_git (Main_branch)
$ git ls-files --stage
100644 30aa3732af149122998338bcd99fc8a6fb52c988 0 Dir1/File3.txt
100644 849327df401a74dd0148b99b532d290f7da80eae 0 File1.txt
100644 36d9b23df2cdcd91759d69a291d04beddab2b091 0 File2.txt
```

В Staging area подготовлены для коммита три файла, blob-объекты для которых находятся в директориях .git/objects/30, .git/objects/84, .git/objects/36. Два из этих файлов находятся в корне рабочей директории, один – в директории Dir1.

Теперь всё готово для объяснения того, что происходит при создании коммита.

2.10. Создание коммита, команда git commit

Формат команды git commit:

git commit <флаги>

Возможные значения флагов:

-m <описание> – коммит обязательно должен содержать текст, описывающий те изменения, ради которых был сделан коммит. Если команда git commit будет использована без флагов, указывающих, откуда взять описание, автоматически откроется редактор и будет предложено ввести описание;

-с <коммит> – описание для нового коммита будет взято из указанного существующего коммита, будет автоматически открыт редактор и предложено внести изменения в описание;

-С – то же самое, что -с, только без открытия редактора.

При выполнении команды `git commit` происходят три действия:

1. Создаётся объект типа «дерево», который описывает текущую рабочую директорию.
2. Создаётся объект типа «коммит», который содержит информацию о коммите.
3. Изменяются указатели, отвечающие за создание цепочки коммитов (ветки). Для этого новый коммит указывает на предыдущий коммит, а заголовок ветки указывает на новый коммит.

Далее будет создан коммит для Staging area, подготовленной в предыдущем разделе, и на примере описан формат всех появившихся объектов.

```
/c/Learn_git (Main_branch)
$ git commit
[Main_branch (root-commit) 00edc80] Try commit
3 files changed, 3 insertions(+)
create mode 100644 Dir1/File3.txt
create mode 100644 File1.txt
create mode 100644 File2.txt

/c/Learn_git (Main_branch)
$ git cat-file -p 00edc80c522d4b79c517c7f377e99a3249e7b24b
tree fa0d51469255146719fe3aebdfedbe5e046a8278
author Git_Book <git_book@home.com> 1658435516 +0300
committer Git_Book <git_book@home.com> 1658435516 +0300

Try commit

/c/Learn_git (Main_branch)
$ git cat-file -p fa0d51469255146719fe3aebdfedbe5e046a8278
040000 tree 8a63ac25a734b6e00b9808209fc0e9e41c76b893 Dir1
100644 blob 849327df401a74dd0148b99b532d290f7da80eae File1.txt
100644 blob 36d9b23df2cdcd91759d69a291d04beddab2b091 File2.txt

/c/Learn_git (Main_branch)
$ git cat-file -p 8a63ac25a734b6e00b9808209fc0e9e41c76b893
100644 blob 30aa3732af149122998338bcd99fc8a6fb52c988 File3.txt
```

После успешного выполнения команды `git commit` будут выведены сообщения, описывающие результат создания коммита.

Строка `[main_branch (root-commit) 00edc80] Try commit` говорит о том, что:

- коммит добавлен в ветку `Main_branch`;
- хеш-значение объекта, описывающего коммит, начинается на **00edc80** и, следовательно, находится в директории `.git/objects/00`;
- описание коммита `Try commit`;
- в коммит добавлены 3 перечисленных blob-объекта.

Переходим в директорию `.git/objects/00` и видим в ней файл с именем `edc80c522d4b79c517c7f377e99a3249e7b24b`, содержащий объект типа «коммит». Выполнив команду `git cat-file` для этого объекта, получаем хранящееся

в нём описание коммита. Описание коммита состоит из четырёх строк. Если текст, описывающий коммит, занимает более одной строки, то общее количество строк в описании коммита может быть более 4.

В первой строке описания коммита находится ссылка на объект «дерево» для корня рабочей директории репозитория. Ссылка состоит из 2 полей:

- текстового обозначения объекта «дерево» (tree);
- хеш-значения объекта «дерево».

Во второй строке находятся данные об авторе коммита и времени выполнения коммита.

В третьей строке находятся данные о лице, выполнившем коммит, и времени выполнения коммита.

Вторая и третья строки будут содержать различные данные в том случае, если происходит копирование коммита (один программист создал коммит, а другой скопировал его в свою ветку).

В четвёртой строке находится описание коммита.

Выполняем команду `git cat-file` для объекта «дерево», хеш-значение которого указано в первой строке описания коммита:

```
$ git cat-file -p fa0d51469255146719fe3aebdfedbe5e046a8278
```

Получаем описание дерева рабочей директории, состоящее из нескольких строк, имеющих одинаковый формат.

Каждая строка объекта «дерево» описывает объект, входящий в дерево и состоит из четырёх частей:

- числового обозначения типа объекта, которое может быть:
 - ✓ 100644 – обычный файл (blob);
 - ✓ 100755 – исполняемый файл;
 - ✓ 040000 – дерево;
 - ✓ 120000 – ссылка;
- текстового обозначения типа объекта;
- хеш-значения объекта;
- имени объекта.

В примере дерево рабочей директории состоит из трёх объектов: двух blob и одного дерева.

Выполняем команду `git cat-file` для объекта, описывающего дерево каталога `Dir1`:

```
$ git cat-file -p 8a63ac25a734b6e00b9808209fc0e9e41c76b893
```

Получаем описание дерева каталога Dir1. Это дерево состоит из одного blob-объекта для файла File3.txt.

На рисунке 2.3 показано дерево рабочей директории.

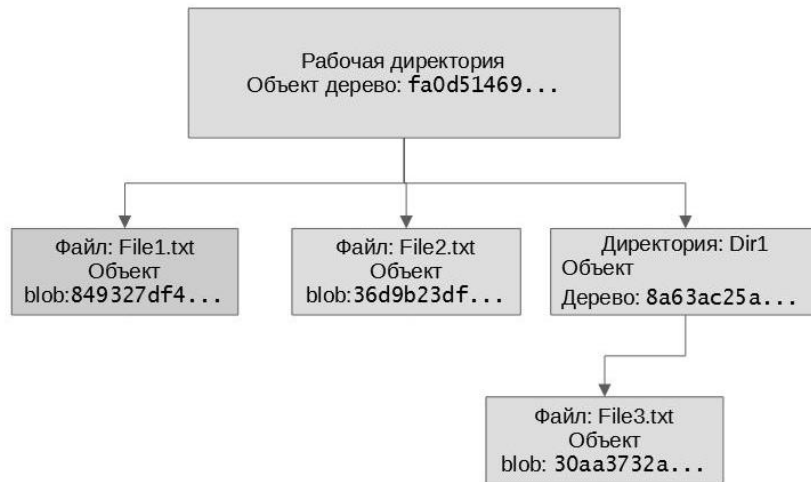


Рисунок 2.3 – Дерево рабочей директории (первый коммит)

Два из трёх действий, выполняемых в процессе коммита, завершены: созданы объекты дерева рабочей директории и коммит. Осталось поменять указатели. Поскольку текущий коммит является первым в ветке Main_branch, он не может ссылаться на предыдущие коммиты, поэтому указатель на предыдущий коммит изменяться не будет. Указатель на начало текущей ветки (заголовок ветки) находится в файле .git/HEAD. Файл .git/HEAD текстовой, и его можно прочитать в текстовом редакторе. Для рассматриваемого примера содержимое файла .git/HEAD:

```
ref: refs/heads/Main_branch
```

Эта запись говорит о том, что указатель на начало ветки находится в файле, размещённом в директории .git/refs/heads. Название файла совпадает с названием текущей ветки: Main_branch. Файл Main_branch текстовой, его содержимое для рассматриваемого примера:

```
00edc80c522d4b79c517c7f377e99a3249e7b24b
```

Это хеш-значение для объекта коммита, с которого было начато построение дерева рабочей директории.

Таким образом, для того чтобы восстановить состояние рабочей директории, соответствующей последнему коммиту, необходимо:

- прочитать файл .git/HEAD;
- из файла, на который указывает содержимое .git/HEAD, узнать хеш-

значение объекта «коммит»;

- из объекта коммита узнать хеш-значение объекта «дерево рабочей директории»;
- построить дерево рабочей директории;
- в листьях дерева рабочей директории находятся blob-объекты файлов, содержащихся в рабочей директории. Из этих blob-объектов возможно восстановить оригиналы файлов.

Внесём изменения в файлы File1.txt и File3.txt, добавим их в индекс и выполним коммит.

Состояние до внесения изменений:

```
/c/Learn_git (Main_branch)
$ git ls-files --stage
100644 30aa3732af149122998338bcd99fc8a6fb52c988 0 Dir1/File3.txt
100644 849327df401a74dd0148b99b532d290f7da80eae 0 File1.txt
100644 36d9b23df2cdcd91759d69a291d04beddab2b091 0 File2.txt
/c/Learn_git (Main_branch)
$ git status
On branch Main_branch
nothing to commit, working tree clean
```

В индексе находятся те же хеш-значения, которые в нём были на момент выполнения предыдущего коммита. Хеш-значения для файлов, находящихся в рабочей директории, совпадают с хеш-значениями, хранящимися в индексе, поэтому Git делает вывод, что никаких изменений не было.

Состояние после внесения изменений в файлы File1.txt и File3.txt:

```
/c/Learn_git (Main_branch)
$ git status
On branch Main_branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified: Dir1/File3.txt
    modified: File1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Теперь хеш-значения для файлов File1.txt и File3.txt, находящихся в рабочей директории, не совпадают с хеш-значениями для этих же файлов, хранящихся в индексе. Git делает вывод, что в эти файлы были внесены изменения.

Добавление модифицированных файлов File1.txt, File3.txt в индекс:

```
/c/Learn_git (Main_branch)
$ git add File1.txt
/c/Learn_git (Main_branch)
$ git add Dir1/File3.txt
/c/Learn_git (Main_branch)
```

```
$ git ls-files --stage
100644 09478d7f26086b969ea8edac54bc98d0c1c9e750 0 Dir1/File3.txt
100644 d4a854445e5dab327199bdeca32f16878c874e05 0 File1.txt
100644 36d9b23df2cdcd91759d69a291d04beddab2b091 0 File2.txt
```

Хеш-значения в индексе для файлов File1.txt, File3.txt изменились и теперь они не совпадают со значениями, хранящимися в последнем коммите. Git понимает, что файлы File2.txt, File3.txt подготовлены для добавления в новый коммит.

Выполнение коммита:

```
/c/Learn_git (Main_branch)
$ git commit -m 'Test commit 2'
[Main_branch d58ea97] Test commit 2
2 files changed, 2 insertions(+), 2 deletions(-)
```

В файле .git/HEAD хранится запись:

ref: refs/heads/Main_branch

В файле refs/heads/Main_branch хранится запись:

d58ea9721672b176fdeda4a6550030a43f982d61

Это имя объекта, соответствующего новому коммиту.

Содержание нового коммита:

```
/c/Learn_git (Main_branch)
$ git cat-file -p d58ea9721672b176fdeda4a6550030a43f982d61
tree a7325622d8c15306bc4f2c3b8825d7ffeaec0d1c
parent 00edc80c522d4b79c517c7f377e99a3249e7b24b
author Git_Book <git_book@home.com> 1658515697 +0300
committer Git_Book <git_book@home.com> 1658515697 +0300

Test commit 2
```

В отличие от предыдущего (первого) коммита, добавлена строка с признаком parent. В этой строке находится имя объекта, содержащего предыдущий коммит. Начиная со второго коммита в ветке, строка parent будет присутствовать во всех коммитах, что даёт возможность перемещаться от одного коммита к другому в обратном хронологическом порядке и, при необходимости, восстановить рабочую директорию для прошлого коммита.

Чтение содержимого объектов «дерево» текущего коммита.

```
/c/Learn_git (Main_branch)
$ git cat-file -p a7325622d8c15306bc4f2c3b8825d7ffeaec0d1c
040000 tree 44fdafa264bef7c64378a1d070a034bf57d02979 Dir1
100644 blob d4a854445e5dab327199bdeca32f16878c874e05 File1.txt
100644 blob 36d9b23df2cdcd91759d69a291d04beddab2b091 File2.txt

/c/Learn_git (Main_branch)
$ git cat-file -p 44fdafa264bef7c64378a1d070a034bf57d02979
100644 blob 09478d7f26086b969ea8edac54bc98d0c1c9e750 File3.txt
```

Графическое представление дерева рабочей директории для второго

коммита представлено на рисунке 2.4. Сравнив деревья рисунков 2.3 и 2.4, видим, что для файлов File1.txt, File3.txt созданы новые blob-объекты. Для File2.txt используется blob-объект из прошлого коммита, что позволяет экономить место на диске.

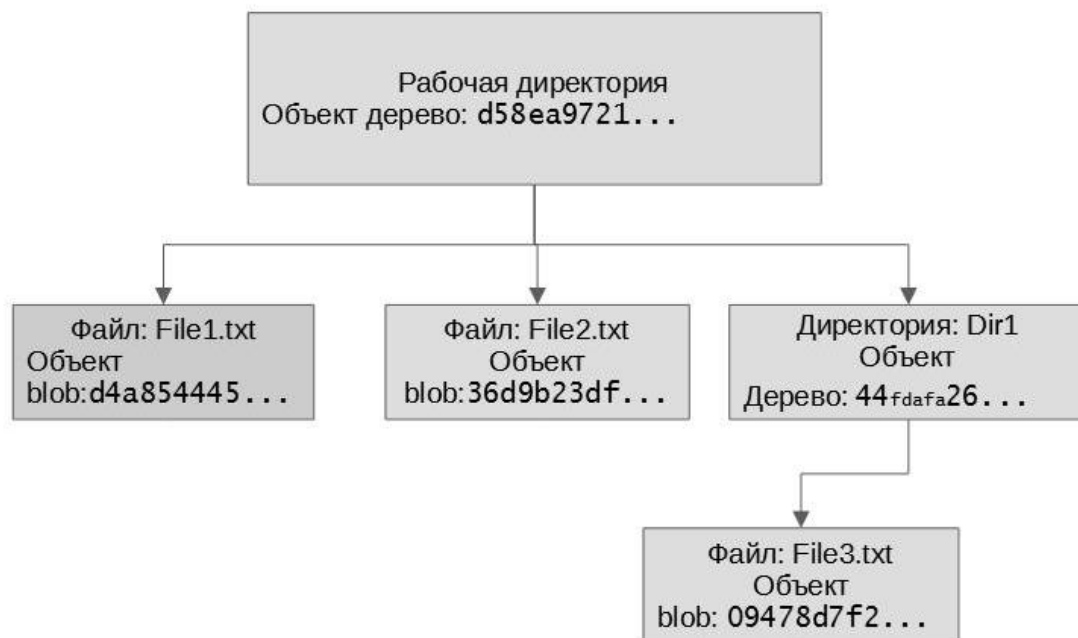


Рисунок 2.4 – Дерево рабочей директории (второй коммит)

2.11. Технология использования веток

Работа над проектом разработки программного продукта может строиться по-разному. Как бы работа не строилась, предполагаем, что для сохранения истории создания кода используется Git. Самый простой вариант построения репозитория – это репозиторий с одной веткой. Как говорилось ранее, ветка – это последовательность коммитов. Репозиторий, содержащий одну ветку, изображён на рисунке 2.5.

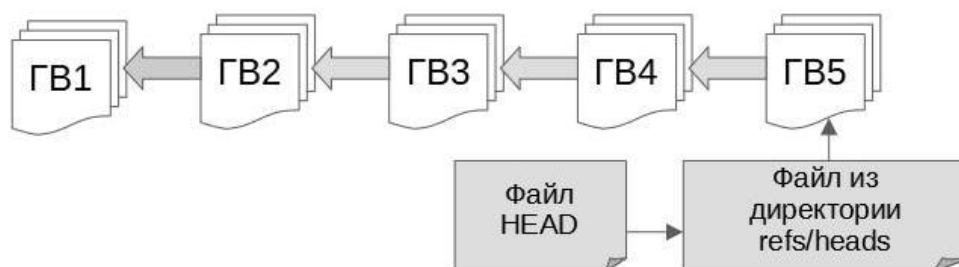


Рисунок 2.5 – Репозиторий с одной веткой

На рисунке показаны 5 коммитов главной ветки (ГВ) ГВ1, ГВ2, ГВ3, ГВ4, ГВ5. Коммиты ссылаются друг на друга в обратном хронологическом порядке. В файле HEAD находится имя файла из директории .git/refs/head, в котором записано хеш-значение последнего в ветке объекта «коммит».

При таком построении репозитория, для управления состоянием рабочей директории, возможно выполнить следующие действия:

1. Переместить указатель HEAD на любой из коммитов. При таком перемещении Git автоматически восстановит в рабочей директории состояние файлов, каким оно было на момент коммита, к которому переместился указатель HEAD. Указатель HEAD в любой момент можно вернуть к последнему коммиту, и Git восстановит соответствующее состояние рабочей директории.

2. Удалить часть коммитов, идущих подряд в обратном хронологическом порядке. Git восстановит состояние рабочей директории, каким оно было до выполнения удалённых коммитов.

3. Посмотреть историю изменения файлов рабочей директории.

4. Сравнить версии файлов из различных коммитов.

Репозиторий с одной веткой возникает крайне редко в ситуации, когда проект выполняет один программист. Среди программистов существует негласное правило, согласно которому в главную ветку помещается только проверенный код, не содержащий грубых ошибок. Именно грубых, так как ни один программист не будет утверждать, что в его коде нет ни одной ошибки. Тестирование кода перед размещением в главной ветке происходит в боковых ветках. На рисунке 2.6 показан репозиторий с боковой веткой, когда HEAD указывает на боковую ветку. На рисунке 2.7 показан репозиторий с боковой веткой, когда HEAD указывает на главную ветку.

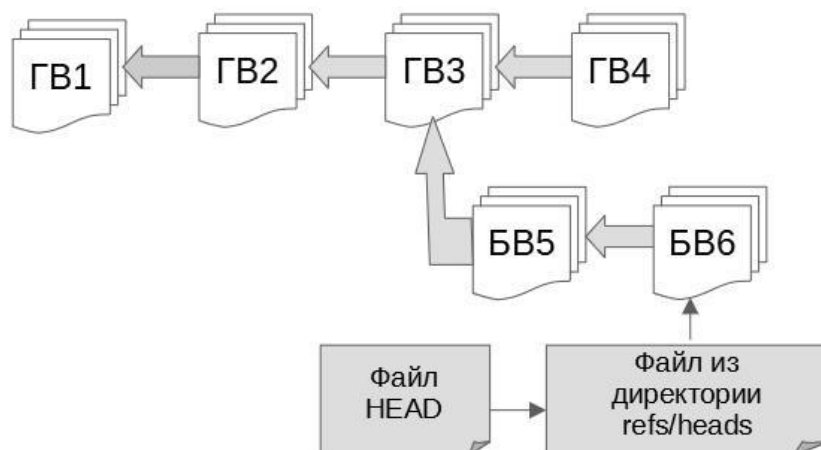


Рисунок 2.6 – Репозиторий с двумя ветками (HEAD указывает на боковую ветку)

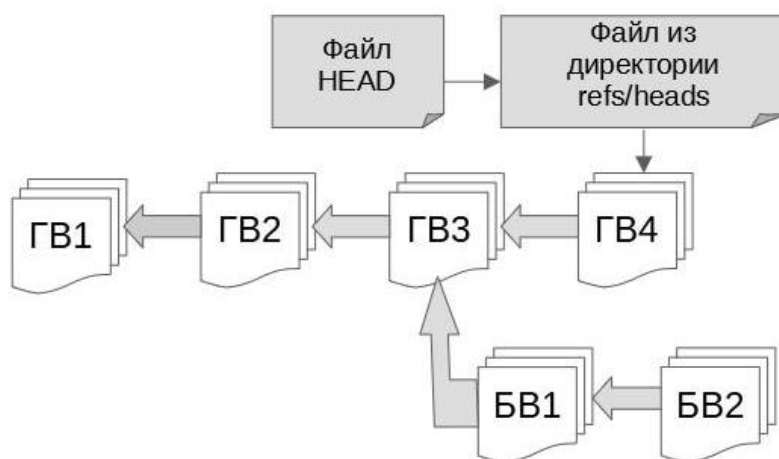


Рисунок 2.7 – Репозиторий с двумя ветками (HEAD указывает на главную ветку)

В репозитории может быть неограниченное количество веток, однако стараются сохранять такое количество веток, при котором не теряется обзорность репозитория. При построении репозитория по схеме с несколькими ветками, для управления состоянием рабочей директории возможно выполнять все действия, которые доступны для репозитория с одной веткой, и в дополнение ряд других действий:

1. Переключение с одной ветки на другую. При этом Git обеспечивает состояние рабочей директории, соответствующее последнему коммиту ветки.
2. Сохранение состояния рабочей директории из одной ветки в коммите для другой ветки. Необходимость такого сохранения возникает, когда работа над кодом в боковой ветке завершена, и необходимо перенести ре-

зультат в главную ветку. При этом боковая ветка сохраняется, и в дальнейшем возможно продолжить вносить изменения в код этой ветки.

3. Слияние последних коммитов двух веток. Выполняется, когда окончательный результат складывается из двух коммитов. В этом случае продолжить работу с боковой веткой нельзя.

4. Удаление ветки. Когда задача решена или стало понятно, что был выбран ошибочный путь решения задачи, боковая ветка удаляется без вреда для остальных веток проекта.

Проект, над которым работают несколько программистов, обязательно содержит боковые ветки (их минимальное количество равно количеству программистов).

Таким образом, технология использования веток включает следующие действия:

1. Создание ветки.
2. Удаление ветки.
3. Переключение между ветками.
4. Слияние веток.

В следующих разделах будет подробно рассмотрено каждое из этих действий.

2.12. Создание ветки

Для создания ветки используется команда `git branch`.

Формат команды `git branch`:

`git branch <ключ> <имя ветки>`

Одно из возможных значений ключа: `-m`. При использовании команды `git branch` с ключом `-m` произойдёт переименование текущей ветки.

Команду `git branch` можно использовать без ключа, в этом случае будет выполнено создание новой ветки.

При выполнении команды `git branch` без ключа происходят следующие действия:

1. В директории `.git/refs/heads` создаётся файл, имя которого совпадает с именем новой ветки, указанным в команде `git branch`.
2. В созданный файл записывается хеш-значение текущего коммита,

того, на который указывает HEAD через соответствующий файл в директории .git/refs/heads.

На рисунке 2.8 показана ситуация, которая возникнет при добавлении ветки New_branch в репозиторий, главная ветка которого называется Main_branch. Если продолжить делать коммиты, то они будут добавляться в главную ветку, после коммита ГВ3. Для добавления коммитов в новую ветку на неё необходимо переключиться. Для переключения между ветками предназначена команда `git checkout`.

Формат команды `git checkout`:

`git checkout <ключ> <имя ветки>`

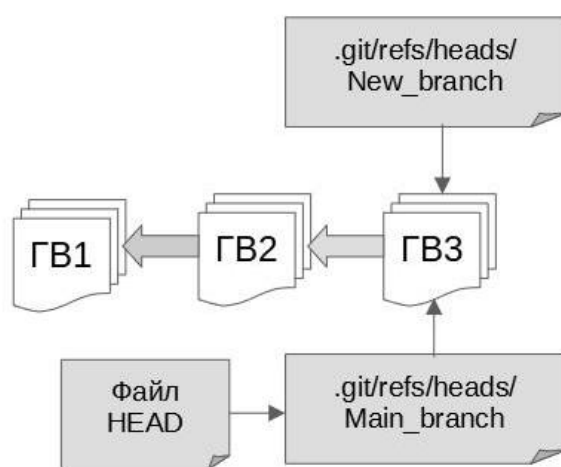


Рисунок 2.8 – Создание новой ветки

Одним из возможных ключей является `-b`. Использование команды `git checkout` с ключом `-b` эквивалентно последовательности команд `git branch` и `git checkout`. Произойдёт создание новой ветки и переключение на неё.

После выполнения команды `git checkout` в файл HEAD будет записано имя файла из директории .git/refs/heads, совпадающее с именем ветки, на которую произошло переключение. Если выполнить несколько коммитов, то ситуация будет выглядеть, как показано на рисунке 2.9.

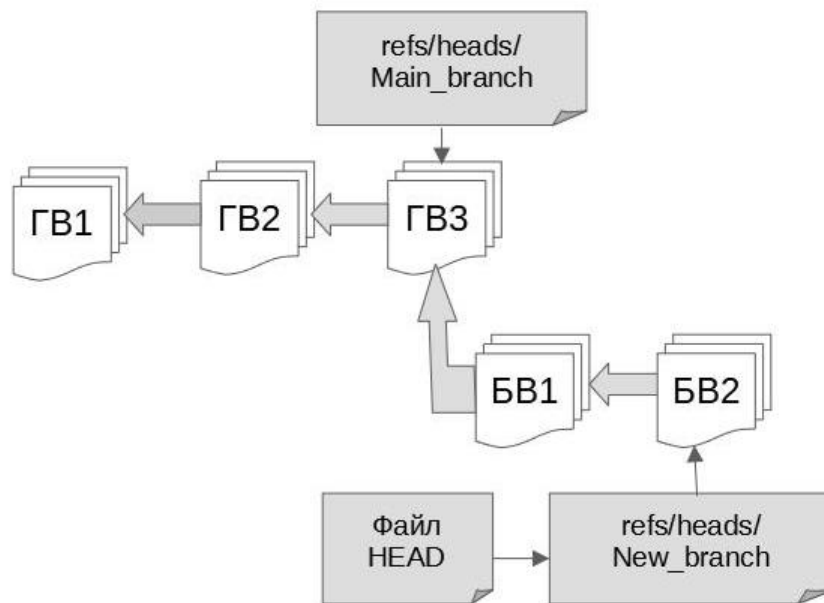


Рисунок 2.9 – Добавление коммитов в новую ветку

Для просмотра веток, существующих в репозитории, используется команда `git branch` с ключами `-a` или `-g` или без ключей.

Использование ключа `-a` или команды `git branch` без ключей приведёт к выводу полного списка веток.

Использование ключа `-g` приведёт к выводу списка веток из удалённого репозитория (об удалённых репозиториях речь пойдёт в одном из следующих разделов).

Так выглядит список веток из примера:

```

$ git branch -a
* main_branch
  New_branch
  
```

Текущая ветка в списке отмечена звёздочкой.

При переключении на другую ветку происходит не только изменение содержимого файла `HEAD`, но и изменение состояния рабочей директории. Проще всего представить себе перестройку рабочей директории следующим образом. Назовём последний коммит в текущей ветке «источником», а последний коммит в ветке, на которую происходит переход, «приёмником». Все файлы, имена которых присутствуют одновременно в «источнике» и «приёмнике», удаляются из рабочей директории. После этого в рабочую директорию записываются файлы, восстановленные из `blob`-объектов «приёмника».

При переходе между ветками нужно иметь в виду следующее:

1. Все изменения, сделанные в текущей рабочей директории в отслеживаемых файлах и не включённые в коммит, будут потеряны.
2. Все неотслеживаемые файлы перейдут в «приёмник».

Эти особенности перехода между ветками могут стать источником серьёзных проблем. Для того чтобы избежать проблем, есть два пути:

- сделать коммит перед переходом на другую ветку;
- использовать команду `stash` того, чтобы «складировать» изменения во временном хранилище.

2.13. Команда `git stash`

Если выполненные изменения рано добавлять в коммит, и есть необходимость переключиться на другую ветку для выполнения срочной работы, чтобы не потерять изменения, их можно «складировать».

Для «складирования» изменений нужно выполнить команду:

`git stash save` или **`git stash`**. `git stash save` позволяет добавить комментарий к «складируемым» файлам.

При выполнении команды `git stash` происходит следующее:

- создаётся резервный коммит;
- резервный коммит сохраняется в изолированном хранилище, построенном по принципу стека;
- состояние рабочей директории восстанавливается из последнего коммита текущей ветки.

Для того чтобы добавить в резервный коммит неотслеживаемые файлы, необходимо использовать ключ `-u` (например, `git stash save -u 'Резервная копия 1'`).

Посмотреть, что хранится на «складе», можно с помощью команды:

`git stash list`

Пример выполнения команды `git stash list`:

```
$ git stash list
stash@{0}: On Main_branch: Склад для ветки Main_branch
stash@{1}: On New_Branch: Склад для ветки New_Branch
```

Каждый хранимый на «складе» временный коммит имеет номер `stash@{число}`. К временному коммиту можно обращаться либо с использованием правила стека («последний пришёл, первый ушёл»), либо по номеру.

При извлечении временного коммита со «склада» происходит восстановление рабочей директории в соответствии с описанием, хранящимся в этом коммите.

Команда:

git stash apply

извлекает со «склада» временный коммит с номером 0.

Можно указать номер того коммита, который необходимо извлечь:

git stash apply stash@{число}

Команда `git stash pop` выполняет те же действия, что и `git stash apply`, и удаляет временный коммит со «склада». Без указания номера команда `git stash pop` будет применена к временному коммиту с номером 0.

Команда ***git stash show*** показывает, какие изменения были сохранены во временном коммите с номером 0.

```
$ git stash show
File_1.txt | 3 ++-
1 file changed, 2 insertions(+), 1 deletion(-)
```

Детали изменения можно посмотреть при использовании ключа `-p`.

```
$ git stash show -p
diff --git a/File_1.txt b/File_1.txt
index 1da2014..2fb2027 100644
--- a/File_1.txt
+++ b/File_1.txt
@@ -1,2 @@
-This is first file we will use for tests.
\ No newline at end of file
+This is first file we will use for tests.
+Changes for test stash.
\ No newline at end of file
```

Как и в других командах, относящихся к `stash`, можно указать номер временного коммита, который будет использован для просмотра изменений.

Команда ***git stash branch <имя новой ветки>*** создаёт новую ветку с указанным именем и переносит в неё содержимое временного коммита. Без указания номера для создания ветки будет использован временный коммит с номером 0.

Команда ***git stash drop*** удаляет временный коммит. Без указания номера будет удалён временный коммит с номером 0.

Команда ***git stash clear*** удаляет весь «склад».

2.14. Просмотр истории коммитов

История коммитов бывает полезна в следующих ситуациях:

- необходимо посмотреть историю коммитов, чтобы составить представление о том, как развивается проект;
- необходимо выяснить, кто внёс те или иные изменения;
- необходимо определить хеш-номер коммита, к которому нужно вернуться.

Для просмотра истории коммитов используется команда ***git log***.

Формат команды *git log*:

git log <ключи> --<путь>

Возможные значения ключей:

- <число> или -п<число> – выводит последние коммиты в количестве, заданном числом;
- pretty = <значение> – значение задаёт формат вывода, может быть oneline, short, medium, full;
- p – показывает изменения, сделанные в текущем коммите;
- all – выводит историю всех коммитов для всех веток.

Правило, которым руководствуется Git при выводе истории, гласит: «Переходить от одного предка к другому, пока они не закончатся».

Рассмотрим для примера репозиторий, граф которого изображён на рисунке 2.10.

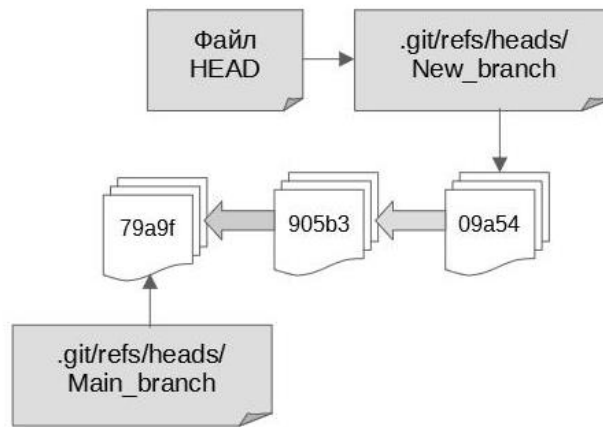


Рисунок 2.10 – Тестовый репозиторий

Выполним для этого репозитория команду `git log --pretty = full`.

Git начал формировать список коммитов с того из них, на который указывает содержимое файла HEAD через файл с именем New_Branch. В список коммитов попали все коммиты, составляющие дерево репозитория. Два из них принадлежат ветке New_Branch и один – ветке Main_branch.

```

git log --pretty=full
commit 09a543aaa4b8adaced3ea2e4b98b9973dfcfc002 (HEAD -> New_Branch)
Author: Git_Book <git_book@home.com>
Commit: Git_Book <git_book@home.com>

Третий коммит.

commit 905b38d0d881bb229252127c0ee1c988ab84acd3
Author: Git_Book <git_book@home.com>
Commit: Git_Book <git_book@home.com>

Проверка перехода между ветками

commit 79a9f3a521f77bed2c94115cbe7a6a7316abd276 (Main_branch)
Author: Git_Book <git_book@home.com>
Commit: Git_Book <git_book@home.com>

Test commit 1
  
```

В первой строке результата выполнения команды в скобках записано, что текущая ветка New_Branch (на неё указывает HEAD). Следующий коммит принадлежит текущей ветке (отсутствует информация, заключённая в скобки). Последний коммит принадлежит ветке Main_branch (название ветки в скобках).

Если необходимо вывести список коммитов только для одной ветки, то для ветки Main_branch это сделать легко:

```

git log Main_branch
commit 79a9f3a521f77bed2c94115cbe7a6a7316abd276 (Main_branch)
Author: Git_Book <git_book@home.com>
Date: Thu Jul 21 21:12:24 2022 +0300

Test commit 1
  
```

Для ветки New_Branch команда будет выглядеть сложнее:

```
$ git log Main_Branch..New_Branch
commit 09a543aaa4b8adaced3ea2e4b98b9973dfcfc002 (HEAD -> New_Branch)
Author: gitflic-muiv-gmbulan <gmbulan@gmail.com>
Date: Sun Jul 24 21:52:43 2022 +0300

    Третий коммит.

commit 905b38d0d881bb229252127c0ee1c988ab84acd3
Author: gitflic-muiv-gmbulan <gmbulan@gmail.com>
Date: Sun Jul 24 19:36:06 2022 +0300

    Проверка перехода между ветками
```

В команде `git log Main_Branch..New_Branch` указано, что нужно вывести сведения о коммитах, расположенных в ветке `New_Branch` от её начала до того места на дереве репозитория, где эта ветка примыкает к ветке `Main_Branch`.

2.15. Сравнение версий файлов, команды `git diff`, `git show`

При анализе истории изменений файлов в репозитории нередко бывает необходимо сравнить версии одного и того же файла. Для выполнения этой операции используется команда `git diff`.

Формат команды `git diff`:

`Git diff <ключ> <файл1> <файл2>`

Возможные значения ключей:

`--diff-filter=<признак>`; признак может иметь одно из значений: А – сравнение для файлов, добавленных в индекс; D – сравнение для удалённых файлов; М – сравнение для изменённых файлов;

`--word-diff=color` – выделять удалённые слова красным цветом, а добавленные – зелёным. Этот ключ действует по умолчанию.

Файл1, файл2 – имена файлов и/или коммитов, для которых будет производиться сравнение. `<ключ> <файл1> <файл2>` являются необязательными атрибутами команды `git diff`.

Рассмотрим на примере, как задаются файлы, для которых будет выполняться операция сравнения, и как «расшифровывать» результаты сравнения.

В рабочую директорию был добавлен файл с именем `Learn_diff.txt`. Вначале этот файл содержал следующие строки (версия 1):

Файл Изменения, добавленный в коммит.

Неизменяемая строка.

Изменяемая строка: коммит.

Этот файл был добавлен в коммит:

```
/c/Learn_git (Main_branch)
$ git add Learn_diff.txt

/c/Learn_git (Main_branch)
$ git commit -m 'Изучение команды git diff'
[Main_branch b5fd14d] Изучение команды git diff
2 files changed, 5 insertions(+), 2 deletions(-)
create mode 100644 Learn_diff.txt
```

Содержимое файла Learn_diff.txt было изменено (версия 2):

Файл Изменения, добавленный в индекс.

Неизменяемая строка.

Изменяемая строка: индекс.

Изменённый файл был добавлен в индекс:

```
/c/Learn_git (Main_branch)
$ git add Learn_diff.txt
```

Файл Learn_diff.txt был изменён ещё раз (версия 3):

Файл Изменения, не отслеживаемый.

Неизменяемая строка.

Изменяемая строка: рабочая директория.

После выполнения этих действий существуют три версии файла Learn_diff.txt: первая хранится в последнем коммите, вторая хранится в индексе, третья хранится в рабочей директории.

Выполним команду git diff без ключей и указания имён файлов:

```
1 /c/Learn_git (Main_branch)
2 $ git diff
3 diff --git a/Learn_diff.txt b/Learn_diff.txt
4 index c0901fb..d687242 100644
5 --- a/Learn_diff.txt
6 +++ b/Learn_diff.txt
7 @@ -1,3 +1,3 @@
8 -Файл Изменения, добавленный в индекс.
9 +Файл Изменения, не отслеживаемый.
10  Неизменяемая строка.
11 -Изменяемая строка: индекс.
12 \ No newline at end of file
13 +Изменяемая строка: рабочая директория.
14 \ No newline at end of file
```

Результат выполнения команды расшифровывается следующим образом.

Строка 3. Git сообщает, что будут сравниваться два файла a/Learn_diff.txt и b/Learn_diff.txt.

Строка 4. Git выводит хеш-значения сравниваемых файлов.

Замечание!

Не обязательно указывать в команде 40 символов хеш-значения, достаточно указать несколько первых символов (не менее 5).

Строки 5, 6. Git сообщает, что для обозначения изменений в файле a/Learn_diff.txt будет использоваться знак минус, а для обозначения изменений в файле b/Learn_diff.txt – знак плюс.

Строка 7. Заголовок блока описания изменений. Заголовок с двух сторон ограничен символами @@ и указывает, что из файла **a** в блок сравнения будут добавлены 3 строки, начиная с первой, и из файла **b** – три строки, начиная с первой.

Строки с 8 по 14. Git описывает, какие изменения были выполнены в файле **a**, чтобы получить файл **b**. Красным цветом выделены удалённые строки, зелёным цветом – добавленные строки и чёрным цветом – неизменившиеся строки. Текст «\ No newline at end of file» говорит о том, что строка, указанная перед ним, последняя в файле.

Таким образом, из файла **a** были удалены строки:

- **Файл Изменения, добавленный в индекс.**
- **Изменяемая строка: индекс.**

В файл **b** были добавлены строки:

- **Файл Изменения, не отслеживаемый.**
- **Изменяемая строка: рабочая директория.**

Строка «Неизменяемая строка.» присутствует в обоих файлах.

Сравнивая приведённые выше версии файлов, понимаем, что сравнивались файлы, сохранённые в индексе и в рабочей директории.

Делаем вывод, что команда git diff без указания ключа и имён файлов сравнивает изменения в файлах, хранящихся в индексе и в рабочей директории. Если существуют несколько файлов, в которых произошли изменения, описания изменений будут для всех файлов.

Выполним команду

git diff HEAD:

```
/c/Learn_git (Main_branch)
$ git diff HEAD
diff --git a/Learn_diff.txt b/Learn_diff.txt
index 0f83f71..d687242 100644
--- a/Learn_diff.txt
+++ b/Learn_diff.txt
@@ -1,3 +1,3 @@
-файл Изменения, добавленный в коммит.
+файл Изменения, не отслеживаемый.
Неизменяемая строка.
```

```
-Изменяемая строка: коммит.  
\ No newline at end of file  
+Изменяемая строка: рабочая директория.  
\ No newline at end of file
```

Проведя анализ результата сравнения, понимаем, что команда `git diff HEAD` сравнивает файлы, сохранённые в последнем коммите и в рабочей директории.

Выполним команду

git diff --staged:

```
/c/Learn_git (Main_branch)  
$ git diff --staged  
diff --git a/Learn_diff.txt b/Learn_diff.txt  
index 0f83f71..c0901fb 100644  
--- a/Learn_diff.txt  
+++ b/Learn_diff.txt  
@@ -1,3 +1,3 @@  
-Файл Изменения, добавленный в коммит.  
+Файл Изменения, добавленный в индекс.  
Неизменяемая строка.  
-Изменяемая строка: коммит.  
\ No newline at end of file  
+Изменяемая строка: индекс.  
\ No newline at end of file
```

Анализ результатов сравнения показывает, что команда `git diff --staged` сравнивает файлы, хранящиеся в последнем коммите и в индексе.

Приведённые три варианта команды `git diff` позволяют проводить сравнение файлов в пределах последнего коммита. Рассмотрим другие варианты:

- `git diff HEAD^` – сравнение файлов из предпоследнего коммита и из рабочей директории;
- `git diff <имя ветки1> <имя ветки 2>` – сравнение файлов из коммитов, расположенных в головных коммитах указанных веток;
- `git diff <имя ветки>` – сравнение файлов из головного коммита текущей ветки и головного коммита ветки, указанной в команде;
- `Git diff <хеш-значение 1> <хеш-значение 2>` – сравнение файлов из коммитов, заданных их хеш-значениями.

Указанные команды сравнивают ВСЕ файлы, расположенные в областях, указанных в команде `git diff`. Если необходимо сравнить только один файл, то его имя нужно добавить в конец соответствующей команды:

- `git diff <имя файла>`
- `git diff HEAD < имя файла >`
- `git diff --staged < имя файла >`

- `git diff HEAD^ < имя файла >`
- `git diff < имя ветки 1> < имя ветки 2> < имя файла >`
- `git diff < имя ветки> < имя файла >`
- `git diff < хеш-значение 1> < хеш-значение 2> < имя файла >`

Команда **git show** показывает содержимое указанного объекта.

Формат команды git show:

git show <объект>

В зависимости от типа объекта будет показано:

- для blob-объекта – текст сохранённого файла;
- для дерева – список объектов, входящих в дерево;
- для коммита – информация о коммите и различия между файлами из заданного в команде коммита и файлами из предшествующего коммита (для формирования списка различий используется команда `git diff`).

Пример выполнения команды git show:

```
/c/Learn_git (Main_branch)
$ git show
commit b5fd14da49b6c21759266118fd34ea78ff51d2d2 (HEAD -> Main_branch)
Author: Git_Book <git_book@home.com>
Date: Mon Jul 25 21:55:20 2022 +0300

Изучение команды git diff

diff --git a/Learn_diff.txt b/Learn_diff.txt
new file mode 100644
index 0000000..0f83f71
--- /dev/null
+++ b/Learn_diff.txt
@@ -0,0 +1,3 @@
+Файл Изменения, добавленный в коммит.
+Неизменяемая строка.
+Изменяемая строка: коммит.
\ No newline at end of file
```

В приведённом примере в команде `git show` не указан объект. В этом случае в качестве объекта используется коммит `HEAD`. Файлы, содержащиеся в головном коммите, будут сравниваться с файлами из предыдущего коммита. Поскольку в предыдущем коммите файла **a** не было (его хеш-значение равно 0), в качестве разницы выведено содержимое файла **b** (нет удалённых и неизменных строк, есть только добавленные строки).

2.16. Удаление файлов, команда `git rm`

Результат удаления файла из рабочей директории, с точки зрения Git, будет зависеть от того, в каком состоянии файл находился перед удалением. Рассмотрим варианты.

1. Удаляемый файл находится в состоянии не отслеживаемый. Удаление такого файла никак не повлияет на состояние Git, поскольку Git ничего не «знает» об этом файле.

2. Удаляемый файл находится в состоянии отслеживаемый, не изменённый. При выполнении ближайшего коммита файл будет исключён из дерева рабочей директории, но его копия будет храниться в предыдущих коммитах. Потери информации не произойдёт.

3. Удаляемый файл находится в состоянии отслеживаемый, изменённый, не добавленный в индекс. Случай аналогичен рассмотренному в п. 2, за исключением того, что изменения, сделанные в файле, будут потеряны без возможности восстановления.

4. Удаляемый файл находится в состоянии отслеживаемый, изменённый, добавленный в индекс. Копия удалённого файла хранится в индексе и будет добавлена в ближайший коммит. Удаление файла из рабочей директории не произойдёт.

Удаление файла в ситуациях, описанных в первых трёх пунктах, не требует использования команд Git. Но необходимо быть внимательным, чтобы избежать возможных негативных последствий. Для удаления файла в четвёртой ситуации требуется специальная команда, которая позволит удалить файл из индекса и из рабочей директории. Последствия выполнения такой команды аналогичны последствиям действий, описанных в п. 3. Команда, позволяющая удалить файл в ситуации п. 4, называется `git rm`.

Формат команды `git rm`:

`git rm <ключ> <имя удаляемого файла>`

Возможные значения ключа:

--forced – удаление с игнорированием всех предупреждений;

--cached – удаление файла только из индекса. Файл становится не отслеживаемым.

2.17. Восстановление файлов, команда `git restore`

Git не оправдывал бы своё существование, если бы не позволял восстанавливать случайно удалённые или некорректно изменённые файлы. Разумеется, Git позволяет выполнять восстановление. Для этого предназначена команда `git restore`.

Формат команды `git restore`:

`Git restore <ключ> <имя восстанавливаемого файла>`

Возможные значения ключа:

`--source=<коммит>` – задаёт коммит, из которого будет взят восстанавливаемый файл;

`--worktree` – записать восстановленную копию в рабочую директорию (действует как значение по умолчанию);

`--staged` – записывает восстановленную копию в индекс и отменяет запись в рабочую директорию. Для записи в индекс и в рабочую директорию необходимо указать оба ключа: `--worktree` и `--staged`.

Рассмотрим пример восстановления файла. В рабочей директории, в индексе и в последнем коммите хранится файл с именем `File1.txt`, содержащий текст «`Example1_V1`».

В рабочей директории изменим текст в файле `File1.txt` на «`Text for restore`» и выполним команду `git add File1.txt`. После этого удалим файл `File1.txt` из рабочей директории и восстановим его двумя способами.

Первый способ: восстановление с помощью команды `git restore File1.txt`

```
/c/Learn_git (Main_branch)
$ dir
Dir1 File1.txt File2.txt Learn_diff.txt
/c/Learn_git (Main_branch)
$ cat File1.txt
Text for restore
/c/Learn_git (Main_branch)
$ rm File1.txt
/c/Learn_git (Main_branch)
$ dir
Dir1 File2.txt Learn_diff.txt
/c/Learn_git (Main_branch)
$ git restore File1.txt
/c/Learn_git (Main_branch)
$ cat File1.txt
Example1_for restore
```

Файл `File1.txt` был восстановлен из индекса.

Второй способ: восстановление с помощью команды: `git restore --source=HEAD File1.txt`

```
/c/Learn_git (Main_branch)
$ dir
Dir1 File1.txt File2.txt Learn_diff.txt
/c/Learn_git (Main_branch)
$ cat File1.txt
Example1_for restore
/c/Learn_git (Main_branch)
$ rm File1.txt
/c/Learn_git (Main_branch)
$ dir
Dir1 File2.txt Learn_diff.txt
/c/Learn_git (Main_branch)
$ git restore --source=HEAD File1.txt
/c/Learn_git (Main_branch)
$ dir
Dir1 File1.txt File2.txt Learn_diff.txt
/c/Learn_git (Main_branch)
$ cat File1.txt
Example1_v1
```

Файл File1.txt был восстановлен из коммита.

2.18. Перемещение указателя HEAD, команда `git checkout` (продолжение)

Git даёт возможность не только восстанавливать файлы, но и возвращаться к прежним состояниям рабочей директории. Возврат может потребоваться в следующих случаях:

1. Необходимо просмотреть файлы проекта в том состоянии, в котором проект был некоторое время назад.
2. Необходимо вернуться к предыдущему состоянию проекта, создать ветку и проверить альтернативное решение.

Для возврата к предыдущему коммиту необходимо изменить значение, записанное в файл HEAD, так как этот файл ссылается на последний коммит. Изменение содержимого файла HEAD называют перемещением указателя. Файл HEAD может содержать или имя ветки или хеш-значение. В первом случае HEAD указывает на последний коммит в ветке, во втором случае HEAD указывает на любой коммит. На рисунке 2.11 показано состояние указателей при перемещении HEAD на произвольный коммит.

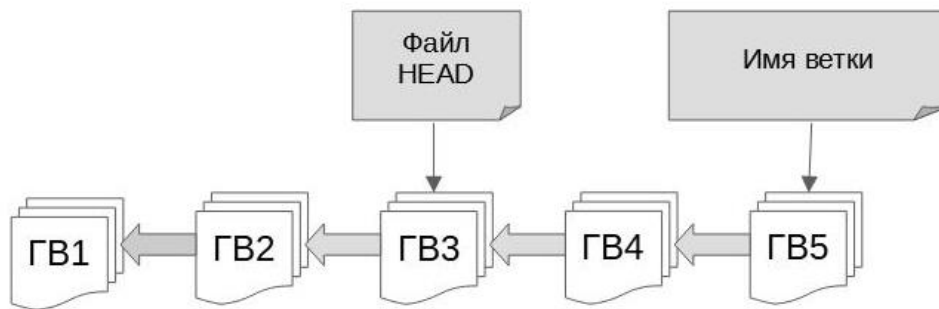


Рисунок 2.11 – Состояние “detached head”

Файл с именем ветки в директории `.git/refs/heads` по-прежнему содержит хеш-значение последнего коммита в ветке (указывает на ветку). Файл HEAD содержит хеш-значение произвольного коммита. Такое состояние называется «Свободный указатель» (Detached head). В таком состоянии команда `git log` выведет список не всех коммитов ветки, а только тех, которые расположены до коммита, на который указывает HEAD. Посмотреть все коммиты в ветке можно с помощью команды `git log -all`. Пример выполнения этой команды приведён ниже.

```
/c/Learn_git ((b5fd14d...))
$ git log --all --pretty=oneline
6aa60021e322f099c14f8008d3ea8ce3f34dc5ef (Main_branch) перемещение HEAD
b5fd14da49b6c21759266118fd34ea78ff51d2d2 (HEAD) изучение клманды git diff
d240a8b99799c7921a99cb059b93be6c0c6d6917 изучение команды git diff
d58ea9721672b176fdeda4a6550030a43f982d61 Test commit 2
00edc80c522d4b79c517c7f377e99a3249e7b24b Try commit
```

Из примера видно, что в ветке есть два указателя: `Main_branch`, указывающий на коммит `6aa600` и `HEAD`, указывающий на коммит `b5fd1`.

Для того чтобы выйти из состояния «Свободный указатель», достаточно выполнить команду `git checkout 'Main_branch'`.

```
/c/Learn_git ((b5fd14d...))
$ git checkout 'Main_branch'
Previous HEAD position was b5fd14d изучение клманды git diff
Switched to branch 'Main_branch'

Andrey@DESKTOP-1RNVJK6 MINGW64 /c/Learn_git (Main_branch)
$ git log --all --pretty=oneline
6aa60021e322f099c14f8008d3ea8ce3f34dc5ef (HEAD -> Main_branch) перемещение HEAD
b5fd14da49b6c21759266118fd34ea78ff51d2d2 изучение клманды git diff
d240a8b99799c7921a99cb059b93be6c0c6d6917 изучение команды git diff
d58ea9721672b176fdeda4a6550030a43f982d61 Test commit 2
00edc80c522d4b79c517c7f377e99a3249e7b24b Try commit
```

Теперь HEAD указывает на файл с именем ветки, который содержит хеш-значение последнего коммита в ветке (всё, как мы привыкли видеть).

Указатель HEAD можно перемещать в пределах одной ветки. Для его перемещения используется команда `checkout`, рассмотренная в разделе 2.12. Для создания состояния «Свободный указатель» в этой команде вместо

имени ветки необходимо указать коммит. Сделать это можно либо с помощью прямой адресации, указав хеш-значение коммита, либо с помощью косвенной адресации, указав смещение относительно текущего значения HEAD:

- HEAD[^] – предыдущий коммит;
- HEAD~<n> – смещение назад по ветке на n коммитов.
- Рассмотрим пример выполнения команды `git checkout HEAD^`.

```
/c/Learn_git (Main_branch)
$ git checkout HEAD^
Note: switching to 'HEAD^'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedhead to false
HEAD is now at b5fd14d изучение команды git diff
```

В результате выполнения команды был не только перемещён указатель HEAD, но и получены подробные рекомендации по поводу возможных действий в состоянии ‘detached HEAD’. Рассмотрим эти рекомендации.

1. Можно изучить состояние рабочей директории (look around). Дело в том, что при изменении положения указателя HEAD рабочая директория восстанавливается из того коммита, на который в данный момент указывает HEAD.

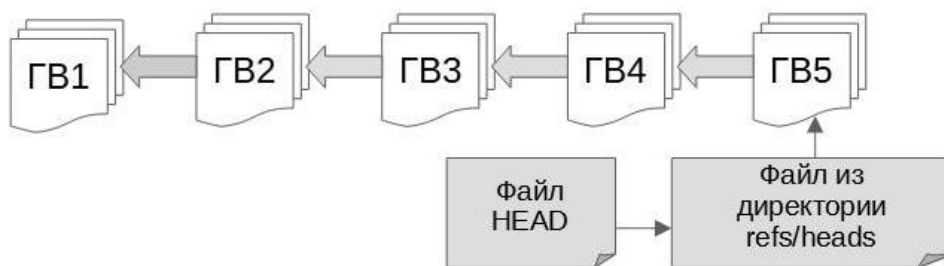
2. Можно поэкспериментировать (make experimental changes). Все изменения, внесённые в рабочую директорию, будут потеряны после выхода из состояния ‘detached head’.

3. Можно сохранить результаты экспериментов, создав новую ветку, «растущую» из коммита, на который указывает HEAD. Ветку можно создать как с помощью команды `git switch -c ‘имя новой ветки’`, так и с помощью команды `git checkout -b ‘имя новой ветки’`. Новую ветку можно создать как до начала серии коммитов, так и после окончания серии коммитов, перед выходом из состояния ‘detached head’.

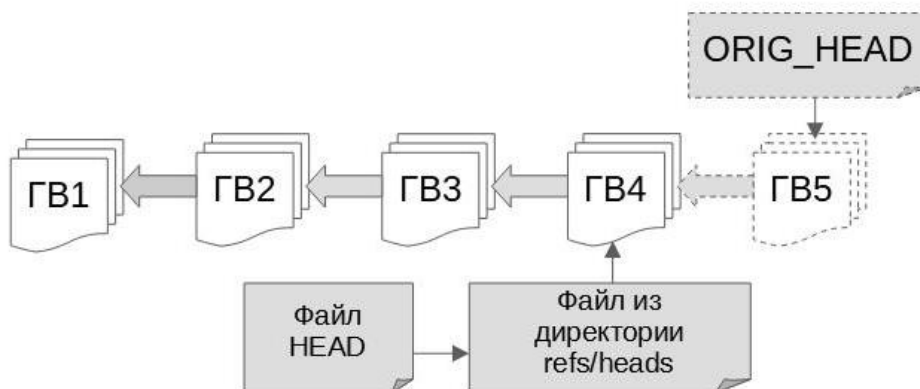
2.19. Удаление и объединение коммитов, команда git reset

Для того чтобы обозначить коммит, с которым в данный момент работает пользователь, в Git существуют два указателя: HEAD и файл с именем ветки. В предыдущем разделе была рассмотрена ситуация, когда изменяется содержимое указателя HEAD. В настоящем разделе будет рассмотрена ситуация, когда изменяется содержимое файла с именем ветки. Для такого изменения используется команда git reset.

На рисунке 2.12 показано, как изменяется положение указателей после применения команды git reset.



а) Дерево коммитов до выполнения команды git reset



а) Дерево коммитов после выполнения команды git reset HEAD^

Рисунок 2.12 – Положение указателей при выполнении команды git reset

До выполнения команды git reset указатель HEAD ссылался на файл с именем ветки, а файл с именем ветки ссылался на хеш-значение последнего коммита в ветке. После выполнения команды git reset указатель HEAD не изменился, а файл с именем ветки стал указывать на хеш-значение коммита, который был указан в команде. Кроме того, появился ещё один указатель ORIG_HEAD, содержащий копию хеш-значения файла с именем ветки до его изменения. Команда git reset может выполняться в двух режимах:

– после перемещения указателя содержимое рабочей директории не изменяется;

– после перемещения указателя рабочая директория восстанавливается из коммита, на который указывает файл с именем ветки.

От того, в каком режиме выполняется команда `git reset`, зависит алгоритм её применения. Рассмотрим всё по порядку.

Формат команды `git reset`:

`git reset` <ключи> <адрес коммита>

Возможные значения ключей:

--soft – команда не изменяет ни рабочую директорию, ни индекс. Все файлы останутся неизменными, но указатель будет передвинут;

--hard – будут удалены все файлы рабочей директории и индекса, после чего состояние рабочей директории и индекса будет восстановлено из коммита, указанного в команде.

Адрес коммита – абсолютный или относительный адрес коммита, на который необходимо передвинуть указатель.

2.19.1. Редактирование последнего коммита

Бывают ситуации, когда после выполнения коммита обнаруживается, что в него необходимо внести изменения. Это возможно сделать с использованием команды `git reset`. Команда `git reset --soft HEAD^` переместит указатель на предпоследний коммит. Восстановления состояния рабочей директории из предпоследнего коммита не произойдёт. В результате состояние архива будет таким, каким оно было непосредственно перед выполнением последнего коммита. Можно внести исправления в подготавливаемый коммит (отредактировать файлы, добавить/удалить файлы в/из индекса, ...).

После того, как исправления будут завершены, необходимо выполнить команду:

`git commit -c ORIG_HEAD`.

Ключ `-c` говорит о том, что комментарий к коммиту должен быть введён заново. Ключ `-C` позволит скопировать комментарий из исправляемого коммита.

Возникающая в результате описанных действий ситуация показана на рисунке 2.13.

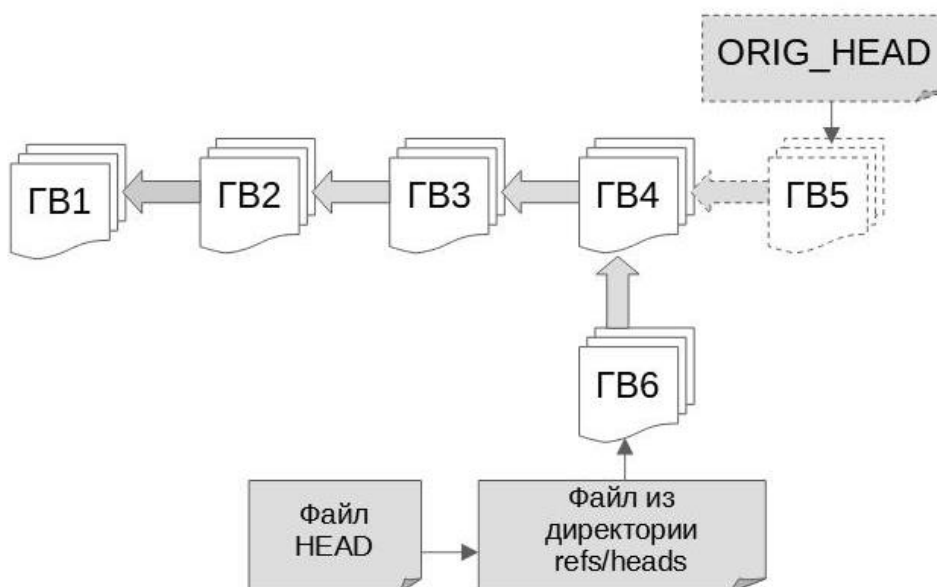


Рисунок 2.13 – Редактирование последнего коммита

Коммит, который был исправлен, остался в архиве, на него возможно переключиться по указателю ORIG_HEAD до тех пор, пока ORIG_HEAD не поменяет своё значение после выполнения новой команды `git reset`. Но даже после этого к исправленному коммиту можно обратиться, если знать его хеш-значение.

Существует другой путь внесения изменений в последний коммит. Вносятся необходимые изменения в рабочую директорию и индекс, после чего выполняется команда:

```
git commit --amend.
```

Эта команда эквивалентна последовательности команд:

```
git reset --soft HEAD^
```

```
git commit -C ORIG_HEAD
```

Если необходимо ввести новый комментарий для исправленного коммита, то команда будет выглядеть так:

```
git commit --amend -m <новый комментарий>
```

2.19.2. Удаление последних коммитов

Для удаления последнего коммита и возврата к предпоследнему используется команда:

```
git reset --hard HEAD^
```

Произойдёт восстановление рабочей директории и индекса из предпоследнего коммита. После внесения изменений и очередного коммита ситуация будет аналогичной, изображённой на рисунке 2.13. «Удалённый» коммит по-прежнему доступен по ссылке ORIG_HEAD.

Для удаления нескольких коммитов используется команда:

```
git reset --hard HEAD~n
```

Здесь *n* – количество коммитов, которые необходимо удалить. Обратиться к цепочке «удалённых» коммитов возможно по указателю ORIG_HEAD (до тех пор, пока он не изменится).

2.19.3. Объединение коммитов

Для объединения нескольких коммитов используется команда:

```
git reset --soft HEAD~n
```

Здесь *n* – количество коммитов, которые необходимо изменить.

После выполнения команды произойдёт перемещение указателя на *n* коммитов назад. Состояние рабочей директории и индекса изменено не будет. И рабочая директория, и индекс будут включать все изменения, которые были зафиксированы в последних *n* коммитах, выполненных после того, на который произошло переключение.

Команда `git commit -c ORIG_HEAD` зафиксирует все изменения, как бы собрав последние коммиты вместе. Ключ `-C` обычно не применяется, так как для объединённого коммита нужен новый комментарий.

На цепочку объединённых коммитов можно переключиться с использованием указателя ORIG_HEAD.

2.19.4. Отмена действия команды *git reset*

Если по какой-то причине была выполнена нежелательная команда `git reset`, то все изменения можно отменить, выполнив команду:

```
git reset ORIG_HEAD
```

Отмена возможна, пока не будет выполнена ещё одна команда `git reset`, которая изменит значение указателя ORIG_HEAD.

2.20. Удаление веток

Ветки – это, как правило, «черновики». Они нужны, чтобы что-то проверить, разобраться в сложной ситуации, создать временную «заплатку» и т.п. После завершения работы с веткой её, как правило, удаляют. Бывают и исключения. Например, ветка, в которой готовится новый релиз, имеет шанс не только остаться в репозитории, но со временем стать основной. Но всё-таки ветки часто удаляют.

Для удаления ветки используется одна из двух команд:

- `git branch -d <имя ветки>;`
- `git branch -D <имя ветки>.`

Команда с ключом `-d` удалит ветку только в том случае, если она была слита с другой (про слияние см. следующий раздел). Если слияние не выполнялось, то Git остановит операцию удаления ветки.

Команда с ключом `-D` удалит ветку без проверки, была ли она слита с другой или нет.

В Git ничего не теряется и не удаляется безвозвратно. Единожды попав в директорию `.git/objects`, объект остаётся в ней на всё время существования репозитория. Для обращения к объекту необходимо знать несколько первых цифр его хеш-значения. Ветка представляет собой цепочку коммитов, каждый из которых ссылается на предыдущий. Попасть на «удалённую» ветку можно, зная хеш-значение её последнего коммита и указав его в команде `git checkout <хеш-значение>`. После выполнения такой команды возникнет состояние «Свободный указатель» (`detached head`), разобранный в разделе 2.18.

Необходимо помнить, что все изменения, не добавленные в коммит, при удалении ветки будут потеряны, так как они не попадут ни в один из объектов в директории `.git/objects`.

2.21. Слияние веток, принципы выполнения операции

При создании репозитория в нём существует одна ветка, которой можно присвоить имя в ходе выполнения команды `git init`. Это главная ветка проекта. Исходя из практики использования Git, в главной ветке сохраняется только проверенный код. Выполнение этого кода не должно приводить к проблемам, которые необходимо решать в срочном порядке (наличие не критичных проблем, в некоторых случаях, допускается). При такой технологии работы над проектом, для разработки кода каждого модуля создаётся отдельная ветка (назовём её

рабочей веткой модуля). В процессе написания кода модуля все коммиты сохраняются только в рабочей ветке модуля. После того, как модуль разработан и протестирован, финальная версия кода (финальный коммит) добавляется в главную ветку. Структура репозитория, соответствующая описанной технологии работы над проектом, показана на рисунке 2.14.

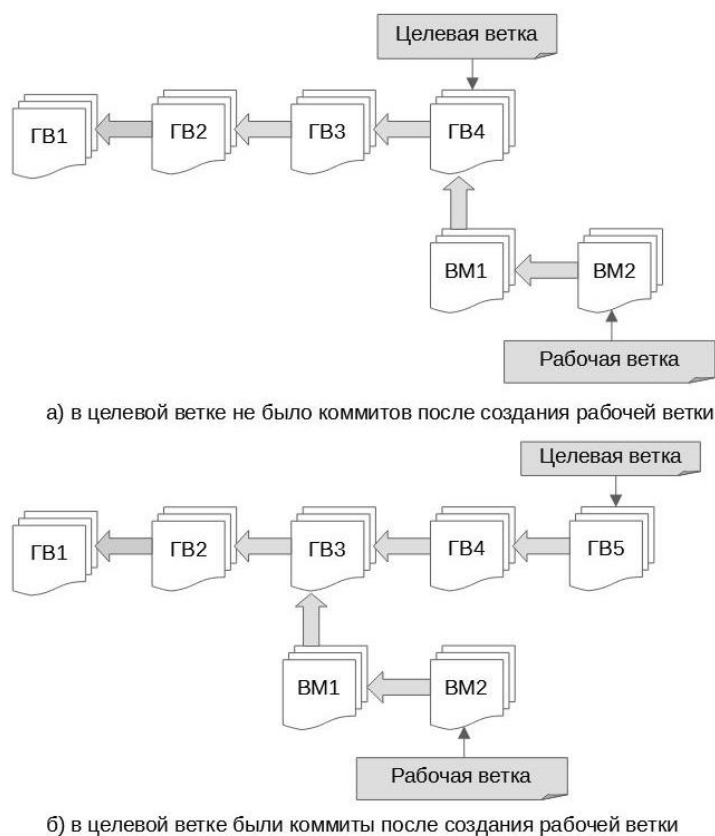


Рисунок 2.14 – Структура проекта при использовании веток

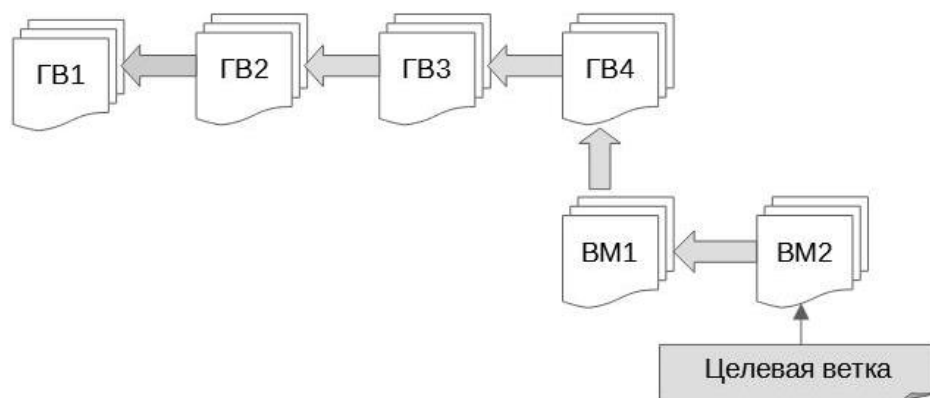
На рисунке 2.14 термин «рабочая ветка» заменён на термин «целевая ветка». В рабочей ветке модуля могут быть дополнительные ответвления, например, для разработки кода функции. Принцип остаётся неизменным. Код создаётся в рабочей ветке функции, тестируется и сливается с рабочей веткой модуля. В общем случае ветка, в которой хранится финальный код, называется целевой. Последней целевой веткой в этой иерархической структуре является главная ветка.

В ситуации (рисунки 2.14а и 2.14б) слияние будет происходить по-разному. В Git существуют два термина для обозначения слияния.

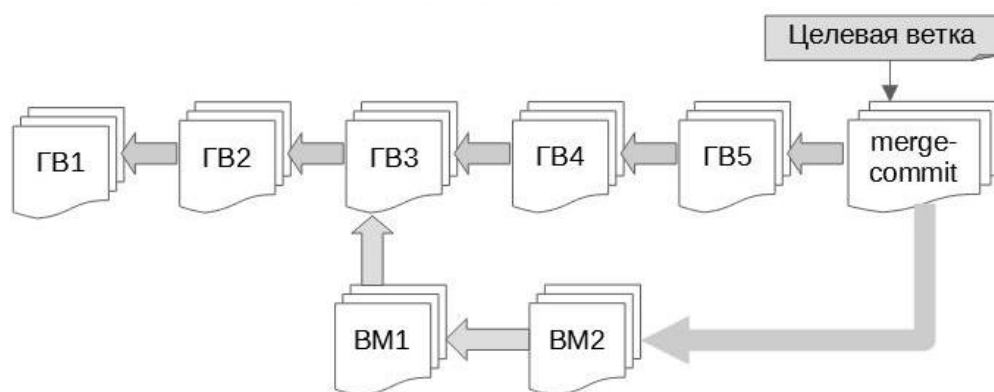
1. **Явное слияние** выполняется в случае, когда в целевой ветке были коммиты после создания рабочей ветки (рисунок 2.14б).
2. **Неявное слияние** выполняется в случае, когда в целевой ветке не

было коммитов после создания рабочей ветки (рисунок 2.14а).

При выполнении неявного слияния указатель целевой ветки переносится на последний коммит рабочей ветки (рисунок 2.15а). Новый коммит не создаётся.



а) неявное слияние



б) явное слияние

Рисунок 2.15 – Явное и неявное слияние

При выполнении явного слияния создаётся новый коммит, у которого два родителя (рисунок 2.15б). Такой коммит обозначается термином merge-commit. Сделать возврат из merge-commit сложно, поэтому его нужно создавать, когда вероятность наличия ошибок в коде рабочей ветки практически исключена.

Возможность выполнения явного и неявного слияния ограничена:

- неявное слияние возможно создать, только если в целевой ветке не выполнялись коммиты, после присоединения к ней рабочей ветки. Это ограничение снять невозможно;
- явное слияние возможно выполнить, если отсутствуют конфликты.

Под конфликтом понимается ситуация, когда в рабочей и целевой ветке присутствуют файлы с одинаковым именем и различным содержанием. Конфликт возможно устранить, поэтому явное слияние, в отличие от неявного, можно выполнить всегда (приложив дополнительные усилия, если требуется устранить конфликт).

2.22. Слияние веток, команда `git merge`

Команда `git merge` выполняет слияние веток.

Формат команды: `git merge <ключи> <рабочая ветка>`

При слиянии текущей веткой должна быть целевая. По логике Git изменения, сделанные в рабочей ветке, «вливаются» в целевую.

Возможные значения ключей:

--ff – выполнить неявное слияние, если возможно. Другое название явного слияния `fast-forward`;

--no-ff – отключить возможность неявного слияния, выполнять явное слияние;

--ff-only – выполнять только неявное слияние, если это невозможно, то остановить выполнение команды `git merge`.

--strategy – стратегия выполнения явного слияния. Git поддерживает следующие стратегии слияния:

- `resolve`;
- `recursive`;
- `ours`;
- `octopus`;
- `subtree`.

Неявное слияние не вызывает никаких сложностей, оно или выполняется, или не выполняется. С явным слиянием не так просто. Сложности таются в стратегии выполнения слияния. По умолчанию используется стратегия `resolve`, и большинство пользователей Git это устраивает. Рассмотрим эту стратегию, а с остальными читатель сможет познакомиться самостоятельно (например, в статье А. Прокопюк, 2013).

Цель любой из стратегий – построить дерево рабочей директории, которое будет записано в merge-commit. Стратегия resolve заключается в выполнении следующих пяти шагов.

1. Поиск общего предка. Например, при выполнении явного слияния на рисунке 2.15б общим предком будет коммит ГВ3.

2. Поиск файлов, изменившихся в сливаемых ветках относительно общего предка.

3. Блоки, оставшиеся без изменений, включаются в дерево как неизменившиеся.

4. Блоки, изменившиеся только в одной из сливаемых веток, добавляются в дерево как изменившиеся.

5. Блоки, изменившиеся в обеих ветках, добавляются в дерево только при условии, если изменения в них одинаковые. Если изменения различные, то возникает ситуация «конфликт», и выполнение команды git merge приостанавливается. Программист, выполняющий слияние, должен или устранить конфликт и продолжить слияние, или отменить слияние.

Недостаток стратегии resolve проявляется в случае, если в рабочей ветке продолжается редактирование кода и создание новых коммитов. В этой ситуации через некоторое время появится необходимость выполнить ещё одно слияние. Окажется, что для нового слияния общий предок тот же, что и для предыдущего. В результате придётся повторно устранять конфликты. В остальных 4 стратегиях сделаны попытки устранить указанный недостаток, но идеальное решение пока не найдено.

2.22.1. Устранение конфликтов слияния

Рассмотрим на примере, как устраняются файловые конфликты, возникающие при использовании стратегии слияния resolve. Возникновение конфликта означает, что имеются две различные версии файла, и Git не знает, какую из них добавить в рабочую директорию merge-commit. Git просит программиста, выполняющего слияние, дать ему указания, какую версию включать в рабочую директорию.

Предположим, что в проекте есть файл с именем Test_merge.txt. На рисунке 2.16 показано, как изменялось содержимое этого файла.

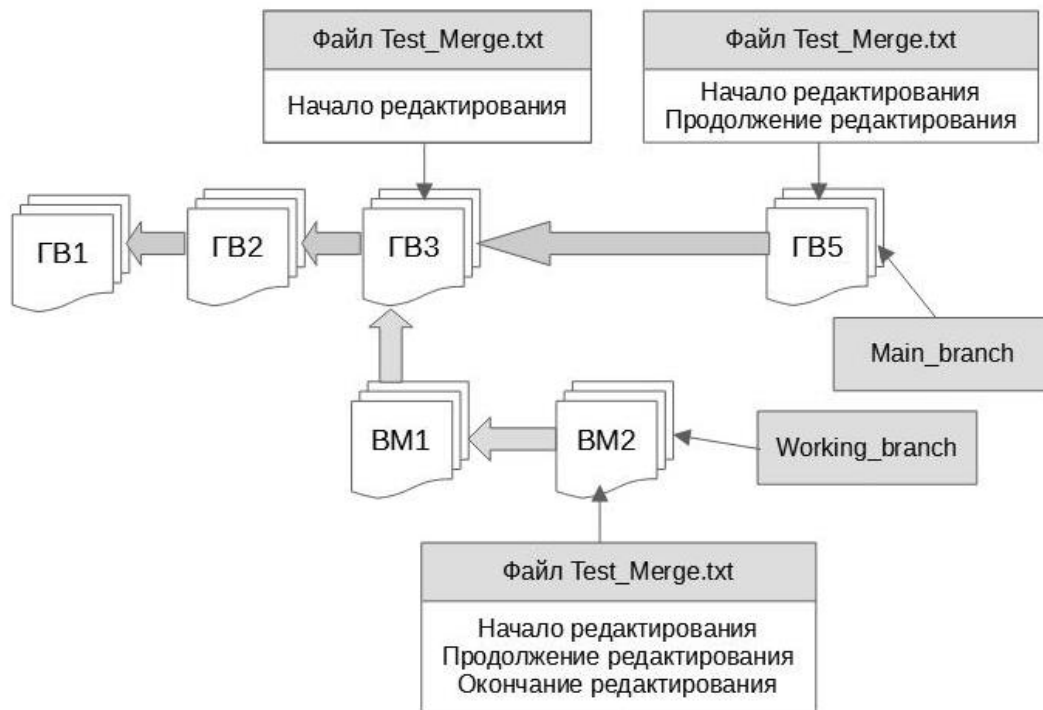


Рисунок 2.16 – Пример слияния веток с конфликтом

Выполним слияние веток Main_branch и Working_branch:

```
/c/Learn_git (working_branch)
$ git checkout 'Main_branch'
Switched to branch 'Main_branch'

/c/Learn_git (Main_branch)
$ git merge 'working_branch'
Auto-merging Test_Merge.txt
CONFLICT (content): Merge conflict in Test_Merge.txt
Automatic merge failed; fix conflicts and then commit the result.
```

В строке “CONFLICT (content): Merge conflict in Test_Merge.txt” Git сообщил, что возник конфликт в содержимом (content) файла Test_Merge.txt.

В строке “Automatic merge failed; fix conflicts and then commit the result.” Git сообщил, что автоматически устранить конфликт не удалось, и просит устранить конфликт и создать коммит.

В сложившейся ситуации возможны три варианта действий:

1. Отредактировать конфликтующие файлы, добившись того, чтобы они были одинаковыми.
 2. Выбрать один из двух файлов.
 3. Прервать слияние веток, выполнив команду `git merge --abort`.
- Рассмотрим первые два варианта.

Ручное редактирование конфликтующих файлов. Файл Test_Merge.

txt в текущей рабочей директории выглядит следующим образом:

```
Начало редактирования
<<<<<<< HEAD
Продолжение редактирования
=====
Продолжение редактирования
Окончание редактирования
>>>>>>> Working_branch
```

Git разметил текст, чтобы помочь в устранении конфликта. Весь текст, находящийся до строчки <<<<<<< **HEAD**, одинаковый в конфликтующих файлах.

Текст между строчками <<<<<<< HEAD и ===== есть только в файле, находящемся в целевой ветке (HEAD). В нашем случае это ветка Main_branch.

Текст между строчками ===== и >>>>>>> Working_branch есть только в файле, находящемся в рабочей ветке. В нашем случае это ветка Working_branch.

Для устранения конфликта необходимо отредактировать конфликтующий файл, находящийся в текущей директории. Не обязательно оставлять в файле текст, который поместил в него Git. Можно ввести новый текст, никак не связанный с конфликтующими файлами. Обязательно нужно удалить разметку <<<<<<< **HEAD**, =====, >>>>>>> Working_branch.

После окончания редактирования и сохранения файла конфликт считается устранённым, и можно завершить слияние командой git merge –continue.

В сложных случаях устранению конфликта может помочь наличие в редактируемом файле общего предка. Чтобы Git добавлял общего предка в заготовку для устранения конфликта, необходимо изменить настройки Git, выполнив команду:

```
git config –global merge.conflictstyle diff3
```

Остановим предыдущее слияние, выполнив команду git merge –abort. И снова запустим процесс слияния командой git merge ‘Working_branch’. Сообщение о том, что выполнить слияние невозможно, не изменится. Изменится содержание файла Test_Merge.txt. Теперь оно выглядит следующим образом:

```
<<<<<<< HEAD
Начало редактирования
Продолжение редактирования
||||||| e83aaf9
Начало редактирования
=====
Начало редактирования
Продолжение редактирования
```

```
Окончание редактирования
>>>>>> working_branch
```

Текст между строчкой ||||| e83aaf9 и ===== представляет собой общего предка. Цифры e83aaf9 – это хеш-значение коммита, в котором находится общий предок конфликтующих файлов. Из сравнения общего предка с конфликтующими файлами видно, что:

- в файл Test_Merge.txt, находящийся в целевой ветке (HEAD), была добавлена строчка «Продолжение редактирования»;
- в файл Test_Merge.txt, находящийся в рабочей ветке (Working_branch), были добавлены две строчки «Продолжение редактирования» и «Окончание редактирования».

Изменим файл Test_Merge.txt так, чтобы он содержал текст, который должен быть добавлен в merge-commit. Например, так:

```
Начало редактирования
Продолжение редактирования (добавлено в целевой ветке)
Окончание редактирования (добавлено в рабочей ветке).
```

Добавим изменённый файл Test_Merge.txt в индекс (git add Test_Merge.txt) и продолжим процесс слияния, выполнив команду git merge – continue. В процессе выполнения последней команды автоматически откроется редактор для того, чтобы можно было ввести комментарий к merge-commit. Результат слияния:

```
$ git add Test_Merge.txt
/c/Learn_git (Main_branch|MERGING)
$ git merge --continue
[Main_branch 2e9bb1b] Merge branch 'working_branch' into Main_branch
```

Слияние прошло успешно. Был создан merge-commit с хеш-значением 2e9bb1b, находящийся в ветке Main_branch. В объекте merge-commit содержится следующая информация:

```
/c/Learn_git (Main_branch)
$ git cat-file -p 2e9bb1b
tree cc7737e7d54a2bb5b64bee090ab0b48a09f2a776
parent a4649d09a31c61b7a4d382f0d7ab27fd61b27249
parent 93bc4e1c6548223a5d6b40c3f11de56b7f1837d2
author Git_Book <git_book@home.com> 1659106632 +0300
committer Git_Book <git_book@home.com> 1659106632 +0300

Merge branch 'working_branch' into Main_branch
```

Существенным является то, что merge-commit содержит ссылки на два родителя (две строчки parent).

Выбор одного из двух конфликтующих файлов. Для разрешения

конфликта путём выбора одного из двух файлов предусмотрены две команды:

- `git checkout –ours <имя файла>`. При выполнении этой команды в текущую рабочую директорию будет скопирован конфликтующий файл из целевой ветки. Он заменит тот, который Git подготовил для ручного устранения конфликта;

- `git checkout –their <имя файла>`. При выполнении этой команды в текущую рабочую директорию будет скопирован конфликтующий файл из рабочей ветки. Он заменит тот, который Git подготовил для ручного устранения конфликта.

После выбора файла его необходимо добавить в индекс (`git add <имя файла>`) и продолжить выполнение слияния (`git merge –continue`).

2.23. Перенос начала ветки, команда `git rebase`

Перед тем, как продолжить, необходимо определить, что такое начало ветки и что такое конец ветки. Начало ветки – это коммит, самый близкий к ветке, из которой «растёт» текущая. Конец ветки – это последний коммит в текущей ветке.

Обычно при структуре дерева коммитов, соответствующей рисунку 2.14а, выполняется неявное слияние, а при структуре дерева коммитов, соответствующей рисунку 2.14б, выполняется явное слияние. Однако, бывают случаи, когда при структуре дерева коммитов, изображённой на рисунке 2.17, необходимо добавить в целевую ветку коммиты VM1 и VM2. Выполнить такую операцию невозможно ни с помощью явного слияния, ни с помощью неявного слияния.

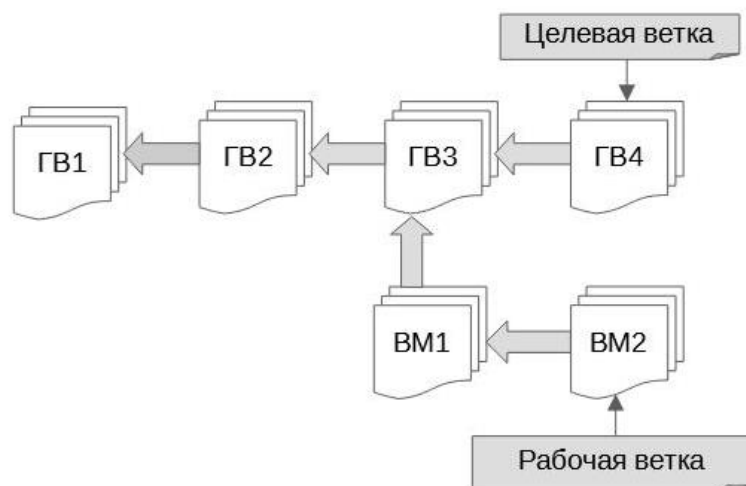


Рисунок 2.17 – Структура дерева коммитов до преобразования

Для решения поставленной задачи Git предлагает команду `git rebase`, которая может изменить структуру дерева коммитов, как показано на рисунке 2.18.

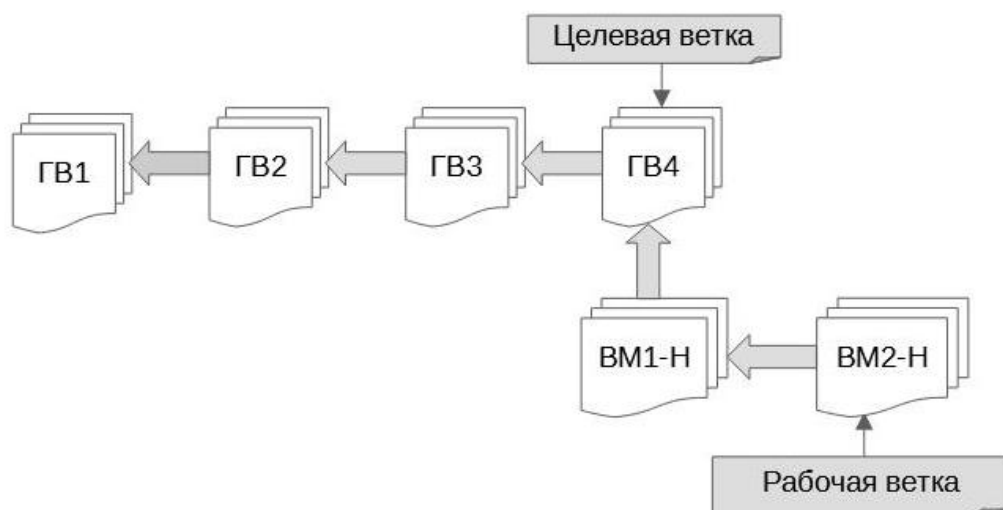


Рисунок 2.18 – Структура дерева коммитов после преобразования

После такого преобразования появляется возможность выполнить неявное слияние, перенеся указатель целевой ветки на коммит BM2-N. В результате коммиты BM1-N и BM2-N окажутся включёнными в целевую ветку. У коммитов BM1-N, BM2-N хеш-значения не совпадают с хеш-значениями коммитов BM1, BM2, однако изменение состояния рабочей директории и индекса описываются так же, как в дереве на рисунке 2.17.

Формат команды `git rebase`:

`git rebase <ключи> <целевая ветка>`

Команда `git rebase` выполняется, когда текущей веткой является рабочая.

Возможные значения ключей:

--abort – используется при возникновении конфликтов для прерывания выполнения команды git rebase;

--continue – используется для продолжения выполнения команды git rebase после устранения конфликта;

--skip – позволяет пропустить текущий коммит при устранении конфликтов.

В процессе преобразования дерева коммитов командой git rebase выполняются следующие шаги:

1. Определяется база преобразования, для чего производится поиск общего коммита двух веток (на рисунке 2.17 это коммит ГВ3).

2. Вычисляется разница между предпоследним коммитом целевой ветки и первым коммитом рабочей ветки: $d1 = \text{ВМ1} - \text{ГВ3}$.

3. Добавляется разница $d1$ к коммиту ГВ4, в результате получается коммит ВМ1-Н.

4. Для каждой пары коммитов в рабочей ветке вычисляется разница и добавляется к последнему преобразованному коммиту. Для примера (рисунок 2.17) будут выполнены следующие преобразования: $d2 = \text{ВМ2} - \text{ВМ1}$; $\text{ВМ2-Н} = \text{ВМ1-Н} + d2$.

5. После преобразования всех коммитов указатель рабочей ветки переносится на последний из преобразованных коммитов.

6. Создаётся указатель ORIG_HEAD, указывающий на последний коммит в оригинальной рабочей ветке.

Для примера (рисунок 2.17) результат преобразования, выполненного командой git rebase, показан на рисунке 2.19.

Когда в процессе добавления разницы возникает конфликт, выполнение преобразования приостанавливается до тех пор, пока конфликт не будет устранён. Устранение конфликта происходит так, как описано в разделе 2.22.1.

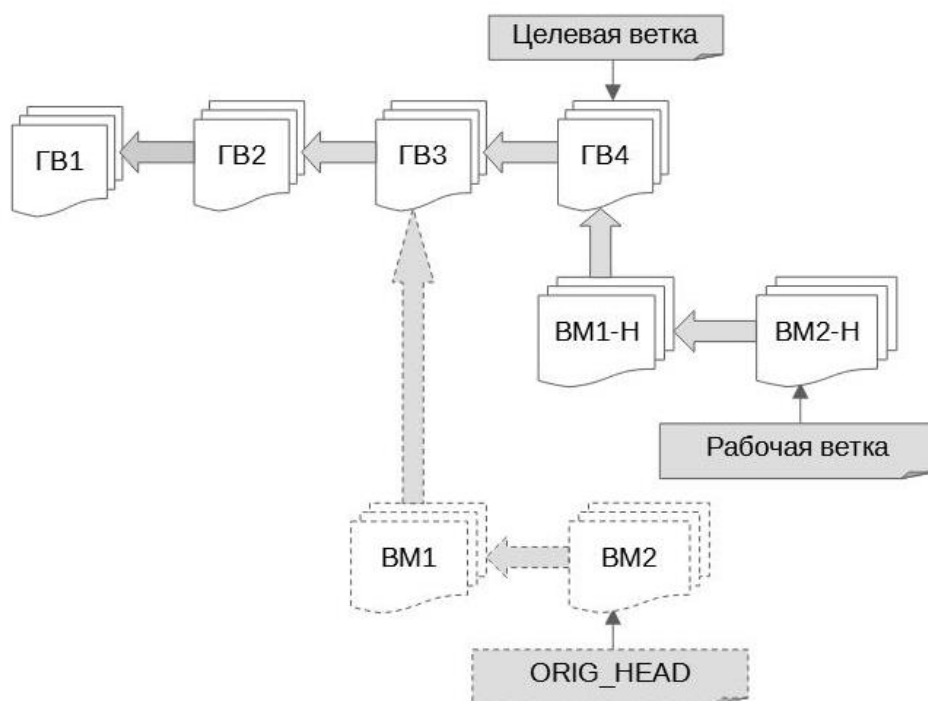


Рисунок 2.19 – Результат выполнения команды rebase

Если в процессе анализа конфликтной ситуации оказывается, что файлы из текущего коммита можно не включать в рабочую директорию (они полностью замещаются файлами из следующего коммита), то текущее преобразование можно пропустить, выполнив команду `git rebase –skip`.

Прервать выполнение команды `git rebase` возможно только в случае её приостановки для устранения конфликта. Прерывание осуществляется командой `git rebase –break`. Однако даже после завершения преобразования возможно вернуть дерево коммитов в состояние, в котором оно находилось до начала преобразования. Для этого используется указатель `ORIG_HEAD` и команда:

```
git reset –hard ORIG_HEAD
```

2.24. Копирование коммита, команда `git cherry-pick`

В процессе работы над проектом может возникнуть ситуация, когда необходимо скопировать коммит из одной ветки в другую. Например, два разработчика создают части программного модуля, ведя работу в разных ветках. Одному из них необходимо получить код своего коллеги, в котором

описывается общий интерфейс. При этом сливать ветки пока рано. Или другой пример. В результате ошибочных действий был потерян важный коммит. Его хеш-значение известно. Исправить ситуацию может копирование «потерянного» коммита в текущую ветку.

Команда, позволяющая копировать коммиты, называется `git cherry-pick`.

Формат команды `git cherry-pick`:

а) `git cherry-pick <ключи> <хеш-значение коммита, подлежащего копированию>`

б) `git cherry-pick <ключи> <хеш-значение первого коммита> <хеш-значение последнего коммита>`

В случае а будет скопирован один коммит. В случае б будет скопирована последовательность коммитов.

Возможные значения ключей:

--edit – позволяет отредактировать комментарий к коммиту;

--no-commit – в рабочую директорию будут скопированы различия между её текущим содержимым и коммитом, указанным в команде, новый коммит создан не будет;

--abort – используется только при устранении конфликтов для прерывания выполнения команды;

--continue – используется только при устранении конфликтов для продолжения выполнения команды после устранения конфликта.

Команда `git cherry-pick` копирует указанный коммит в текущую директорию. Работает она следующим образом. Вычисляется разница между текущим состоянием рабочей директории и рабочей директорией, зафиксированной в переносимом коммите. Разница добавляется в текущую рабочую директорию. Если возникает конфликт, выполнение команды останавливается до устранения конфликта. После добавления разницы создаётся новый коммит в текущей ветке.

2.25. Работа с внешним репозиторием

Git является распределённой системой контроля версий. Это значит, что каждый пользователь Git хранит свои репозитории отдельно и независимо от других пользователей. Хранить репозитории возможно или на локальном компьютере, или в сетевом хранилище, называемом внешним репозиторием. Использование внешнего репозитория удобно при коллективной работе, для обмена данными между участниками проекта или, когда индивидуальный разработчик использует несколько компьютеров для выполнения работы.

Для работы с Git не требуется постоянная связь с внешним репозиторием. Рабочая директория всегда находится на локальном компьютере. В процессе работы коммиты тоже сохраняются на локальном компьютере. И только, когда разработчик готов поделиться с коллегами результатом, или когда индивидуальному разработчику необходимо сменить рабочее место, коммиты копируются во внешний репозиторий. Внешний репозиторий обычно защищён паролем, и им может пользоваться ограниченный круг разработчиков. Исключение составляют open-source проекты, для которых внешний репозиторий паролем не защищают, но ограничивают права на запись коммитов.

В качестве внешнего репозитория возможно использовать любое файловое хранилище. Если файловое хранилище не подготовлено для хранения репозитория, не имеет специальных инструментов для создания репозитория, просмотра их содержимого и подобных операций, то пользоваться им неудобно. Разработчикам программного обеспечения повезло, для них создан ряд внешних репозитория с удобным интерфейсом и набором инструментов для коллективной работы. Изложение материала в настоящем учебном пособии базируется на внешнем репозитории GitFlic (GitFlic, 2022). Нужно иметь в виду, что команды Git, предназначенные для работы с внешним репозиторием, не зависят от репозитория, поэтому изложенный материал остаётся универсальным.

2.25.1. Подключение внешнего репозитория, группа команд `git remote`

О сайте GitFlic будем думать, как об оболочке, внутри которой хранится множество репозитория. Единицей хранения в GitFlic является проект.

Проект – это внешняя копия локального репозитория. Процедура, позволяющая установить соответствие между локальным репозиторием и проектом, называется подключением внешнего репозитория.

Перед тем, как подключать внешний репозиторий (проект), его необходимо создать. Сейчас нас интересует адрес проекта, который будет его однозначно идентифицировать.

Создать проект

Репозиторий содержит все файлы проекта и историю их изменений.

The screenshot shows a web form for creating a project on GitFlic. It consists of several input fields and dropdown menus. The first section is for the owner, labeled 'Владелец', with a dropdown menu showing 'Baatest'. The second section is for the project name, labeled 'Название проекта', with a text input field containing 'Project_1'. The third section is for the programming language, labeled 'Язык программирования', with a dropdown menu showing 'Markdown'. The fourth section is for the project URL, labeled 'URL проекта', with a text input field containing 'https://gitflic.ru/project/baa-test/' and a sub-input field containing 'project-1'.

Рисунок 2.20 – Создание проекта GitFlic

На рисунке 2.20 показана часть экранной формы GitFlic, предназначенной для создания проекта. Адрес проекта находится в поле **URL проекта**. Адресом является весь текст, находящийся в этом поле, в конце строки необходимо указать расширение `.git`. В итоге получится:

`https://gitflic.ru/project/baa-test/project-1.git`

Это и есть адрес внешнего репозитория.

Для подключения внешнего репозитория к локальному используется команда `git remote add`.

Формат команды `git remote add`:

`git remote add <название внешнего репозитория> <адрес внешнего репозитория>`

Название внешнего репозитория – произвольная строка, которую в

дальнейшем можно будет использовать для ссылки на него.

В процессе выполнения команды `git remote add` потребуется указать логин и пароль учётной записи, содержащей проект, к которому происходит подключение.

Внешний репозиторий является полноценным репозиторием, у которого есть ветки с их указателями, история коммитов, блобы и так далее. Хранятся объекты, принадлежащие внешнему репозиторию, в локальной директории `.git/refs/remote/<имя внешнего репозитория>`. Ветки удалённого репозитория могут отличаться от веток локального репозитория до момента, пока не произведена синхронизация локального и внешнего репозитория. Чтобы было ясно, какой объект имеется в виду, в документации к имени ветки внешнего репозитория добавляется имя репозитория. Например, ветка `Branch_1` репозитория `ex_per` будет обозначаться так: `ex_per/Branch1`.

После подключения с внешним репозиторием можно выполнить следующие действия:

1. Отключить. Для этого используется команда:

git remote remove <название удалённого репозитория>.

Внимание! Команда `git remote remove` не удаляет внешний репозиторий, а только отключает его.

2. Переименовать. Для этого используется команда:

git remote rename <старое название> <новое название>

К одному локальному репозиторию могут быть подключены несколько внешних репозитория. Их список можно сформировать командой:

git remote show

Если в этой команде указать имя внешнего репозитория, то будут выведены сведения о репозитории.

2.25.2. Отправка изменений в удалённый репозиторий, команда `git push`

После того, как было выполнено подключение к внешнему репозиторию, в него можно скопировать локальный репозиторий. Для выполнения этой операции используется команда `git push`.

Формат команды `git push`:

`git push <ключи> <имя внешнего репозитория> <имя ветки>`

Возможные значения ключей:

--all – копировать во внешний репозиторий все ветки. Если этот ключ отсутствует, то обязательно должна быть указана ветка, которая будет копироваться;

--force – перезаписывает ветку во внешнем репозитории, не проверяя наличие конфликтов. Ветка во внешнем репозитории будет точно такой, как в локальном репозитории. Все изменения, внесённые в ветку внешнего репозитория до выполнения команды push, будут потеряны;

--force-with-lease – удаляет из внешнего репозитория все коммиты, которых нет в локальном репозитории. Если коммит во внешний репозиторий был добавлен другим пользователем, то выполнение команды будет прервано.

При выполнении команды push выполняется проверка, есть ли в ветке внешнего репозитория коммиты, добавленные другими пользователями после последнего копирования ветки локального репозитория. Если таких коммитов нет, то команда push будет выполнена успешно. Если такие коммиты есть, что означает невозможность добавления новых коммитов в режиме fast-forward, то выполнение команды push будет прервано и завершится с ошибкой.

Когда разработчик выполняет индивидуальный проект, никаких проблем с копированием локального репозитория не возникает. При командной работе разрабатывается регламент обновления внешнего репозитория. Одной из составляющих регламента являются merge-запросы (запросы на слияние). Хранилища репозитория, такие как GitFlic, предоставляют инструментальные средства для работы с запросами на слияние. Технология работы с запросами на слияние заключается в ограничении права разработчика напрямую вносить изменения во внешний репозиторий. Вместо прямого изменения разработчик должен отправить запрос на слияние руководителю проекта (или члену команды, ответственному за обновление репозитория). После проверки коммиты, включённые в запрос на слияние, будут добавлены во внешний репозиторий или будет выполнено устранение конфликтов.

2.25.3. Клонирование удалённого репозитория, команда git clone

Под клонированием репозитория понимается создание на локальном компьютере копии удалённого репозитория. Необходимость клонирования

возникает, когда разработчик хочет подключиться к новому проекту или продолжить работу после смены компьютера.

Формат команды `git clone`:

`git clone <адрес внешнего репозитория> <директория>`

Директория – это полный путь к директории, в которой будет размещена копия репозитория. Если параметр «директория» не указан, то в директории, из которой вызвана команда `git clone`, будет создана поддиректория, и в неё будет скопирован внешний репозиторий. Имя вновь созданной поддиректории совпадает с именем проекта GitFlic.

Рассмотрим пример клонирования внешнего репозитория (имя проекта `git-book`) в корневой каталог диска `C:` локального компьютера. В проекте `git-book` имеются две ветки: `Main_branch` и `Working_branch`. Содержимое рабочей директории ветки `Main_branch` показано на рисунке 2.21.

Main_branch		git-book
G Check clone comand		
Dir1	Test commit 2	
File1.txt	Test clone	
File2.txt	Try commit	
File3.txt	Check clone comand	
Learn_diff.txt	Изучение клманды git diff	
Test_Merge.txt	Merge branch 'Working_branch' into Main_branch	
Изменения.txt	Изучение клманды git diff	

Рисунок 2.21 – Рабочая директория ветки `Main_branch`

Выполним команду `git clone`:

```
$ cd c:/
$ git clone https://gitflic.ru/project/Gituser/git-book.git
Cloning into 'git-book'...
remote: Counting objects: 29, done
remote: Finding sources: 100% (29/29)
remote: Getting sizes: 100% (12/12)
remote: Total 29 (delta 10), reused 29 (delta 10)
Receiving objects: 100% (29/29), done.
Resolving deltas: 100% (10/10), done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.
```

Была создана директория git-book, и в неё скопирован внешний репозиторий. Копирование прошло успешно, но в последней строчке отчёта команды git clone имеется предупреждение: remote HEAD refers to nonexistent ref, unable to checkout, которое говорит о том, что невозможно переключиться на главную ветку репозитория.

Перейдём в директорию git-book и посмотрим, какие в ней находятся файлы.

```
$ cd git-book
/c/git-book (master)
$ dir
```

Директория git-book является репозиторием, о чём говорит имя ветки в скобках, после имени директории. Имя ветки master. Git создал ветку master в процессе копирования. Но такой ветки в репозитории нет, поэтому команда dir показывает, что рабочая директория пуста. Переключимся на ветку Main_branch:

```
$ git checkout Main_branch
Switched to a new branch 'Main_branch'
branch 'Main_branch' set up to track 'origin/Main_branch'.
```

Git сообщил, что произошло переключение на ветку Main_branch, и что эта ветка соответствует ветке Main_branch репозитория (origin/Main_branch). Теперь дерево каталога git-book выглядит следующим образом:

```
File1.txt
File2.txt
File3.txt
Learn_diff.txt
Test_Merge.txt
Изменения.txt
+---.git
|   config
|   description
|   HEAD
|   index
|   packed-refs
+---hooks
+---info
+---logs
+---objects
|   +---info
|   L---pack
|       pack-f06005ad11ffcd12f3e780ec62357cdadf02d2a1.idx
|       pack-f06005ad11ffcd12f3e780ec62357cdadf02d2a1.pack
L---refs
|   +---heads
|       Main_branch
L---tags
```

В рабочей директории присутствуют те же файлы, что и во внешнем репозитории. Вместе с тем, папка `objects` выглядит непривычно. В ней отсутствуют объекты и присутствует папка `pack`, содержащая два файла. Первый файл имеет расширение `idx`, второй имеет расширение `pack`. Эти два файла хранят объекты репозитория в упакованном виде. Файл с расширением `idx` хранит ссылки на объекты, размещённые в файле с расширением `pack`. Файл с расширением `pack` хранит разности для объектов репозитория. Такая форма хранения объектов сама является объектом Git и называется `pack-файл`. `Pack-файлы` гораздо компактнее, чем независимые объекты, они создаются при пересылке репозитория между компьютерами. Кроме того, `pack-файлы` могут создаваться, если репозиторий на локальном компьютере стал слишком большим.

Рассмотрим ещё одну особенность клонированного репозитория. Она заключается в том, что в клонированном репозитории находятся не только локальные ветки, но и ветки внешнего репозитория. Список веток внешнего репозитория можно посмотреть с помощью команды `git branch` с ключом `-r`:

```
$ git branch -r
  origin/Main_branch
  origin/working_branch
```

На ветку внешнего репозитория возможно переключиться:

```
$ git checkout origin/Main_branch
Note: switching to 'Git-book/Main_branch'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false
HEAD is now at 15870e5 Check clone comand
```

Видим, что ветка внешнего репозитория доступна в режиме “detached HEAD”. Её можно только просматривать.

2.25.4. Просмотр внешнего репозитория, команда *git fetch*

Команда `git fetch` позволяет загрузить обновлённые файлы из внешнего репозитория на локальный компьютер. При выполнении команды `git fetch` состояние рабочей директории не изменяется. Загружаемые обновлённые файлы сохраняются в ветках внешнего репозитория и не оказывают влияния на локальный репозиторий. Такой режим обновления позволяет проанализировать изменения, полученные из внешнего репозитория. Если изменения не конфликтуют с кодом, находящимся в локальном репозитории, то они могут быть добавлены в него командой `git merge`.

Формат команды *git fetch*:

`git fetch <ключи> <имя удалённого репозитория>`

Возможные значения ключей:

`--all` – получить изменения из всех подключённых репозиториях;

`--depth=n` – получить не более `n` коммитов из каждой ветки;

`--shallow-since = date` – получить только коммиты, созданные после даты `date`.

Рассмотрим работу команды `fetch` на примере. Во внешнем репозитории, в ветке `Main_branch` рабочая директория выглядит так, как показано на рисунке 2.22.

Main_branch

▼

git-book

G Fetch commit		
Dir1		Test commit 2
Fetch_main.txt		Fetch commit
File1.txt		Test clone
File2.txt		Try commit
File3.txt		Check clone comand
Learn_diff.txt		Изучение клманды git diff
Test_Merge.txt		Merge branch 'Working_branch' into Main_branch
Изменения.txt		Изучение клманды git diff

Рисунок 2.22 – Внешний репозиторий для демонстрации команды `fetch`

Из директории `Git-book` выполним команду `git fetch Git-book` и выведем список файлов, находящихся в рабочей директории:

```
/c/git-book (Main_branch)
$ git fetch Git-book
remote: Counting objects: 8, done
remote: Finding sources: 100% (6/6)
```

```

remote: Getting sizes: 100% (4/4)
remote: Total 6 (delta 2), reused 6 (delta 2)
Unpacking objects: 100% (6/6), 574 bytes | 95.00 KiB/s, done.
From https://gitflic.ru/project/abulychev/git-book
 15870e5..e7ade33 Main_branch -> Git-book/Main_branch
* [new branch] working_branch -> Git-book/Working_branch

$ dir
Dir1 File2.txt Learn_diff.txt Изменения.txt
File1.txt File3.txt Test_Merge.txt

```

В отчёте команды `git fetch` сказано, что изменения в коммитах ветки `Main_branch` загружены в ветку `Git-book/Main_branch`, а изменения в ветке `Working_branch` загружены в ветку `Git_book/Working_branch`. В списке файлов, находящихся в рабочей директории, отсутствует файл `Fetch_main.txt`, который есть во внешнем репозитории. Этого следовало ожидать, так как команда `git fetch` не обновляет рабочую директорию. Выполним смену ветки `Main_branch` на ветку `Git-book/Main_branch` и снова выведем список файлов, находящихся в рабочей директории:

```

- $ git checkout Git-book/Main_branch
- Note: switching to 'Git-book/Main_branch'.
-
- You are in 'detached HEAD' state. You can look around, make experi-
- mental
- changes and commit them, and you can discard any commits you make in
- this
- state without impacting any branches by switching back to a branch.
-
- If you want to create a new branch to retain commits you create, you
- may
- do so (now or later) by using -c with the switch command. Example:
-
-   git switch -c <new-branch-name>
-
- Or undo this operation with:
-
-   git switch -
-
- Turn off this advice by setting config variable advice.detachedHead to
- false
-
- HEAD is now at e7ade33 Fetch commit
-
- /c/git-book ((e7ade33...))
- $ dir
- Dir1 File1.txt File3.txt Test_Merge.txt
- Fetch_main.txt File2.txt Learn_diff.txt Изменения.txt
-

```

Файл `Fetch_main.txt` появился в рабочей директории, но это рабочая директория ветки внешнего репозитория, которая находится в режиме “detached HEAD”. В этой рабочей директории файлы можно только просмотреть с целью принятия решения о добавлении изменений в локальную

рабочую директорию. Допустим, принято решение добавить файл Fetch_main.txt в локальную рабочую директорию. Для этого:

- сменим ветку на Main_branch;
- выполним команду git merge Git-book/Main_branch.

```
$ git checkout Main_branch
Previous HEAD position was e7ade33 Fetch commit
Switched to branch 'Main_branch'
Your branch is ahead of 'origin/Main_branch' by 2 commits.
(use "git push" to publish your local commits)
$ git merge Git-book/Main_branch
Updating 15870e5..e7ade33
Fast-forward
 Fetch_main.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 Fetch_main.txt
/c/git-book (Main_branch)
$ dir
Dir1  File1.txt File3.txt Test_Merge.txt
Fetch_main.txt File2.txt Learn_diff.txt Изменения.txt
```

Теперь файл Fetch_main.txt находится в рабочей директории локального репозитория. Для фиксации этого изменения необходимо добавить этот файл в индекс и сделать коммит.

2.25.5. Получение изменений из локального репозитория, команда git pull

Команда git pull загружает изменения из внешнего репозитория во внутреннюю ветку (то же, что делает git fetch) и пытается перенести эти изменения в локальную рабочую директорию (то же, что делает команда git merge). Если при слиянии веток не возникает конфликтов, то изменения появятся в локальной рабочей директории сразу же после выполнения команды git pull.

Формат команды git pull:

git pull <ключи> <имя удалённого репозитория>

Возможные значения ключей:

- ff – использовать режим fast-forward при слиянии веток, если это возможно;
- no-ff – отключить режим fast-forward;
- ff-only – остановить выполнение команды, если слияние невозможно выполнить в режиме fast-forward.

Контрольные вопросы

1. Что такое рабочая директория в системе управления версиями Git?
2. Что такое «индекс» в системе управления версиями Git?
3. Опишите, на что влияет содержание каждого из файлов настроек Git: системного, локального, глобального?
4. Опишите принцип отнесения файла проекта к одной из категорий: отслеживаемые/не отслеживаемые, изменённые/не изменённые.
5. Опишите назначение и принципы организации таких объектов Git, как Blob и Tree.
6. Какие действия необходимо выполнить для добавления коммита во внешний репозиторий?
7. Для чего используется файл Head?
8. Опишите варианты, которые могут возникнуть при удалении файла из репозитория, в зависимости от его текущего состояния.
9. Чем «неявное слияние» отличается от «явного слияния»?
10. Опишите процесс устранения конфликтов, возникающих при слиянии веток.
11. В чём отличия команд, предназначенных для получения данных из внешнего репозитория `git clone`, `git pull`, `git fetch`?

3. ПРАКТИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ GIT

3.1. Управление репозиторием из среды программирования

Рассмотренные в предыдущих разделах команды Git позволяют в полной мере управлять репозиторием: создавать и переключать ветки, просматривать изменения, добавлять файлы в индекс, делать коммиты и т.п. Работа в режиме командной строки даёт чувство полного контроля над компьютером. Это увлекательный, но не очень удобный и совсем не наглядный процесс. К тому же разработка программного обеспечения всё реже ведётся в режиме командной строки и всё чаще с использованием интегрированных сред разработки (IDE), таких как Microsoft Visual Studio, Borland Embarcadero. Выполняя работу с использованием IDE, не всегда удобно переключаться в режим командной строки для управления репозиторием, поэтому IDE предоставляют встроенные графические средства для работы с Git. Эти средства позволяют как управлять локальным репозиторием, так и осуществлять обмен данными с удалённым. В настоящем разделе описана работа с репозиторием Git в JetBrains PyCharm и Microsoft Visual Studio. Изучив принципы работы с Git в указанных IDE, читатель в дальнейшем сможет самостоятельно освоить управление репозиторием в других IDE.

3.1.1. Операции с репозиторием Git в среде PyCharm

Материал, изложенный в настоящем разделе, актуален для версий PyCharm, начиная с 2020. В более ранних версиях средства работы с системами управления версиями представлены в ограниченном объёме.

3.1.1.1. Подключение локального репозитория к проекту PyCharm

Локальный репозиторий может быть создан на компьютере вне IDE PyCharm. Если локальный репозиторий уже существует на момент запуска PyCharm, то выберите соответствующую директорию в качестве директории проекта. После открытия проекта в главном меню будет присутствовать раздел Git (рисунок 3.1), из которого можно осуществлять управление репозиторием. Доступны знакомые читателю из предыдущего раздела команды `git commit`, `git push`, `git fetch` и другие. Все команды выполняются в графическом режиме. Это значит, что программисту достаточно ввести в соответствующие

поля экранной формы параметры команды и нажать кнопку подтверждения выполнения. PyCharm автоматически сформирует команду git и выполнит её.

Если директория проекта не является репозиторием, то вместо раздела Git в главном меню будет присутствовать раздел VCS, содержащий команды, позволяющие, в случае необходимости, подключить к проекту внешний репозиторий.

3.1.1.2. Клонирование внешнего репозитория в проект PyCharm

PyCharm позволяет выполнить команду git clone с использованием графического интерфейса и перенести на локальный компьютер файлы из удалённого репозитория.

Создайте новый проект, не являющийся репозиторием Git. В главном меню выберите команду VCS/ Get from Version Control (рисунок 3.2).

PyCharm запустит процесс клонирования внешнего репозитория. На первом шаге появится окно, позволяющее указать адрес внешнего репозитория и авторизоваться (рисунок 3.3). В поле URL введите URL-адрес внешнего репозитория (например: https://edu.gitflic.ru/project/student/project_1.git), логин и пароль. Нажмите кнопку Clone. PyCharm запустит на выполнение команду git clone (см. раздел 2.25.3). Если авторизация и клонирование пройдут успешно, пункт меню VSC (рисунок 3.2) превратится в пункт меню Git (рисунок 3.1).

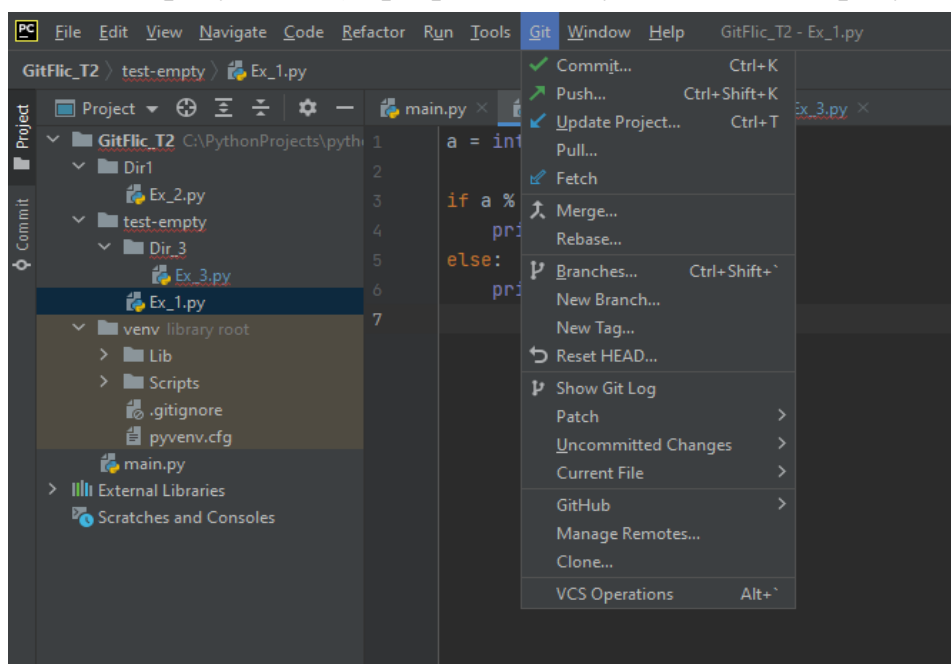


Рисунок 3.1 – Меню Git

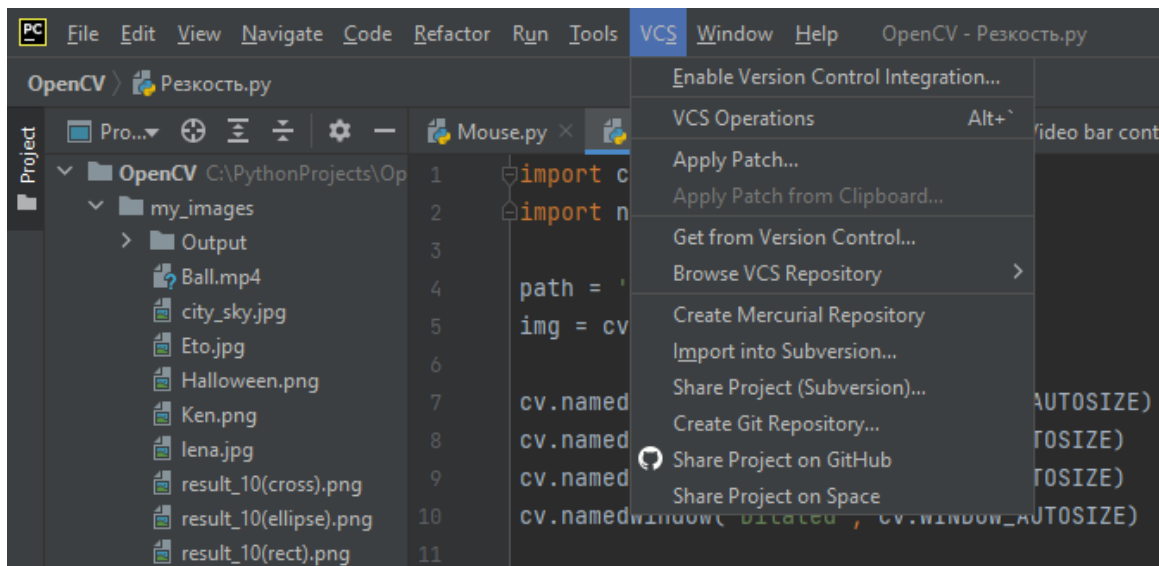


Рисунок 3.2 – Клонирование внешнего репозитория

Если во внешнем репозитории отсутствует ветка с именем master, то рабочая директория проекта будет пустой. Для того чтобы извлечь файлы проекта из архива, необходимо выполнить команду `git checkout` с указанием имени ветки, которая существует в проекте. Команду `git checkout` нужно будет выполнить из командной строки операционной системы или из вкладки `terminal` PyCharm, находясь в директории локального репозитория.

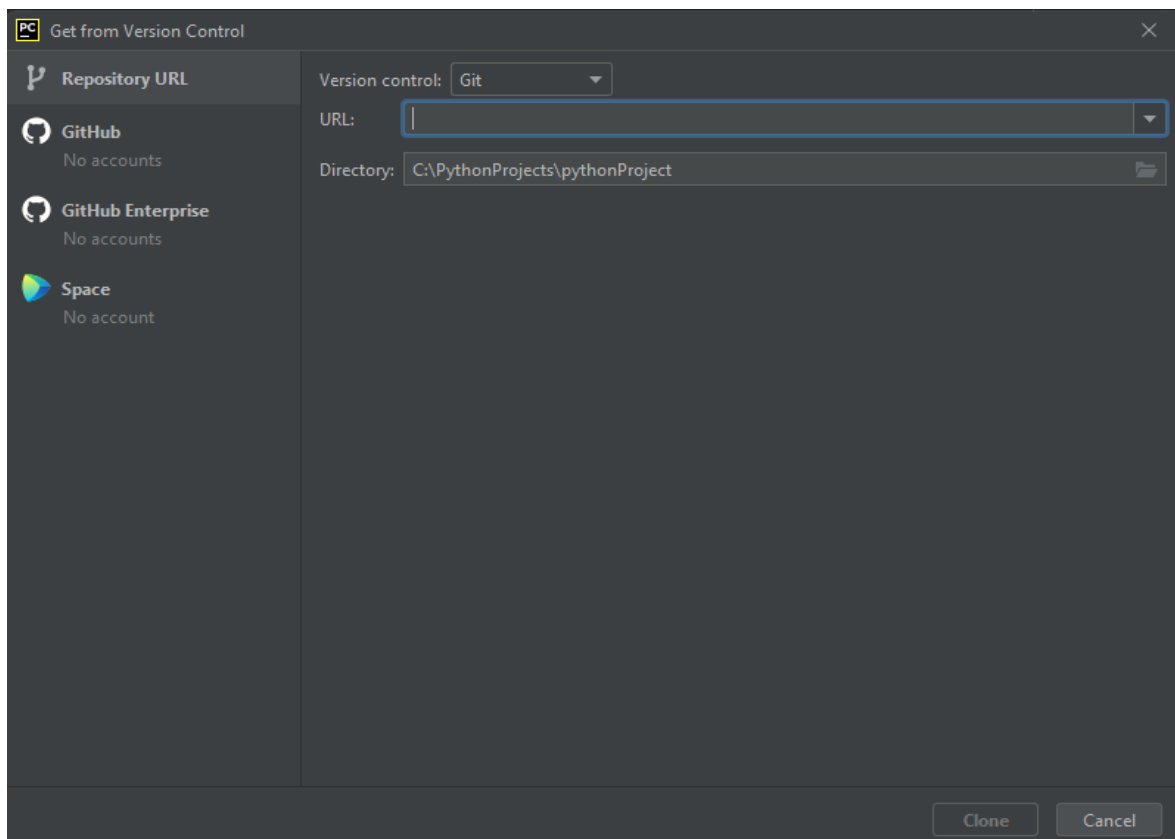


Рисунок 3.3 – Ввод адреса внешнего репозитория

В дальнейшем проект с клонированным репозиторием нужно открывать так, как описано в разделе 3.1.1.1.

3.1.1.3. Выполнение команд *Commit* и *push* из проекта PyCharm

В PyCharm команда `git add` объединена в одном графическом интерфейсе с командой `git commit`.

Для выполнения коммита выберите из главного меню PyCharm команду `Git/Commit...` (рисунок 3.1).

Появится вкладка `Commit` (рисунок 3.4). В верхней части вкладки отображается список файлов, в которые были внесены изменения. Из этого списка необходимо выбрать файлы, предназначенные для добавления в коммит. Выбор осуществляется отметкой соответствующего чек-бокса (например, на рисунке 3.4 выбран файл `Ex_3.py`). В нижней части окна находится текстовое поле, в которое необходимо ввести комментарий к коммиту. После выбора всех необходимых файлов и ввода комментария нажмите кнопку `Commit`. PyCharm выполнит команду `git add` для всех выбранных файлов и затем выполнит команду `git commit`, указав в качестве комментария текст из нижнего текстового поля.

Для записи коммита во внешний репозиторий выберите из главного меню PyCharm команду `Git/Push...` (рисунок 3.1). В появившемся диалоговом окне будет отображено имя текущей ветки и имя ветки внешнего репозитория, в который будет производиться копирование. Кроме того, будут перечислены все коммиты, имеющиеся в этой ветке. Для копирования ветки целиком необходимо выбрать её имя. Это соответствует команде `git push` с ключом `-all`. В правой части диалогового окна будет выведен список файлов, включённых в коммиты ветки. В списке зелёным цветом маркируются файлы, которые ранее были скопированы во внешний репозиторий и не были изменены. Синим цветом маркируются файлы, которые были изменены, добавлены в один из коммитов, но не были скопированы во внешний репозиторий.

Для копирования одного из коммитов ветки необходимо выбрать его из списка. После выбора коммита подробная информация о нём будет отображена в правой части окна.

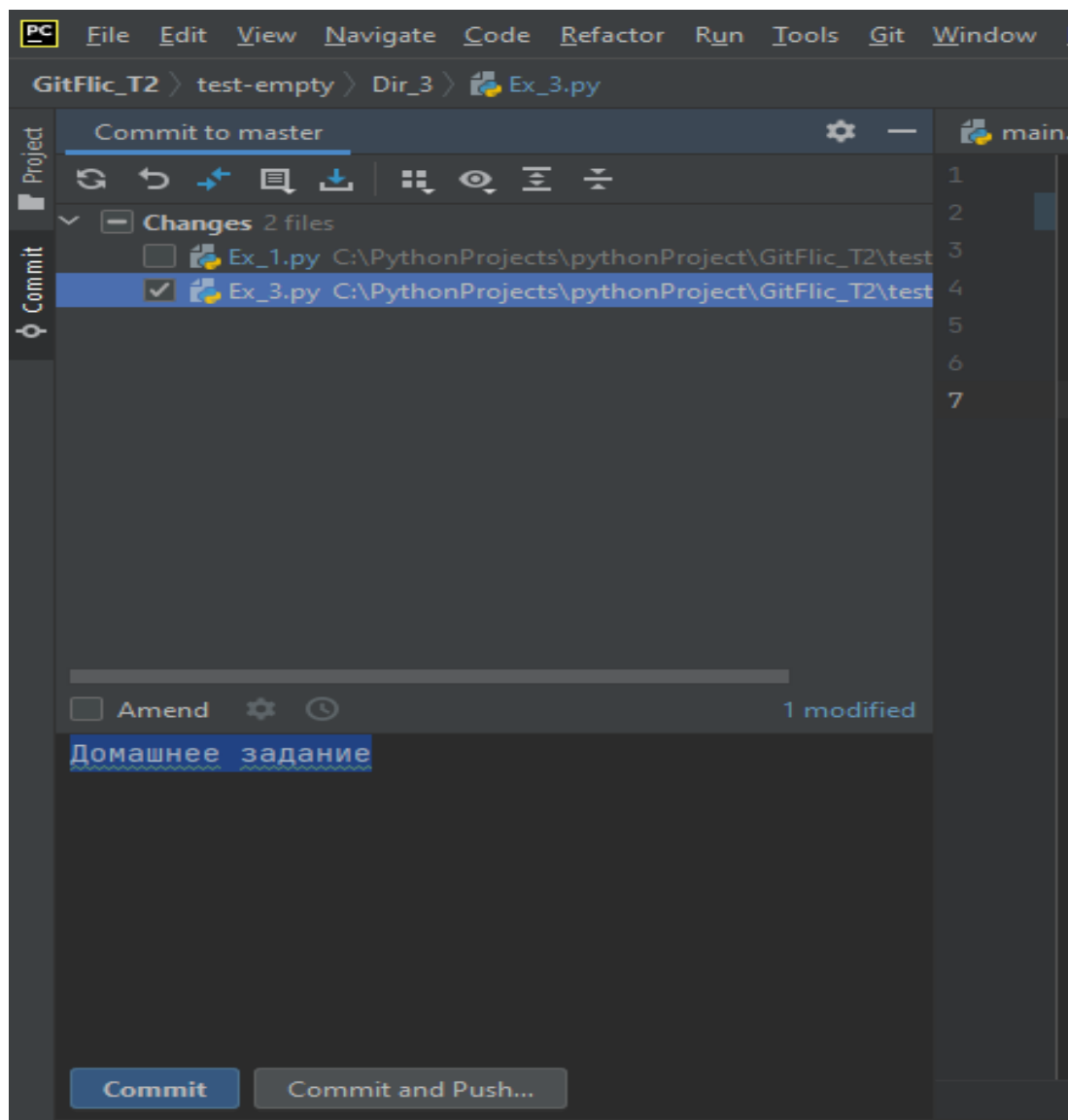


Рисунок 3.4 – Создание коммита

Для активации процедуры копирования необходимо нажать кнопку Push.

Переключение между ветками репозитория осуществляется путём выбора команды Git/rebase из главного меню PyCharm.

Если во внешний репозиторий были внесены изменения, например, вами с другого компьютера, то необходимо обновить проект:

- команда PyCharm Git/Update Project (рисунок 3.1) является максимально автоматизированной командой git fetch с последующим обновлением веток;
- команда PyCharm Git/pull реализует графический интерфейс для команды git pull с возможностью указания ключей;

– команда PyCharm Git/Fetch реализует графический интерфейс для команды `git fetch`.

3.1.2. Операции с репозиторием Git в среде Visual Studio

3.1.2.1. Клонирование репозитория

Есть два пути клонировать репозиторий с использованием возможностей Visual Studio. Первый из них заключается в выборе опции «Клонирование репозитория» при создании проекта. Второй путь заключается в выборе команды «Git/Клонировать репозиторий» из главного меню. Каждый из этих вариантов приведёт к появлению на экране формы описания параметров клонирования. На форме присутствуют два текстовых поля. В одно из них необходимо ввести URL-адрес внешнего репозитория, в другое – путь к пустой директории, в которой будет размещён результат клонирования. После подтверждения данных начнётся процесс клонирования.

Ход процесса клонирования будет отображаться в окне «Вывод», расположенном в нижней части рабочей области Visual Studio (рисунок 3.5). По умолчанию, после клонирования Visual Studio пытается сделать активной ветку `master`. Если такой ветки в репозитории нет, то будет выведено предупреждение: `Warning: remote HEAD refers to nonexistent ref, unable to checkout`. Файлы проекта не будут извлечены из архива, рабочая директория будет пустой.

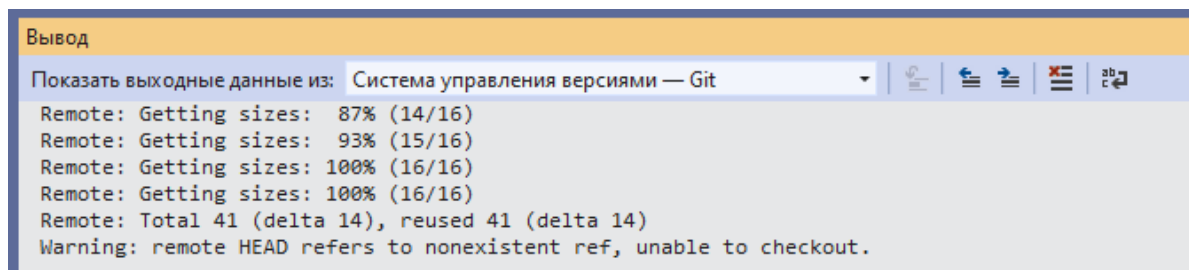


Рисунок 3.5. – Клонирование репозитория

Для извлечения файлов из архива в рабочую директорию необходимо переключиться на одну из веток, существующих в проекте:

```
C:\Visualstudio_Git\GitBook>git checkout Main_branch
Switched to branch 'Main_branch'
Your branch is up to date with 'origin/Main_branch'.
```

После извлечения файлов проекта в рабочую директорию они будут

отображены в обозревателе решений Visual Studio (рисунок 3.6).

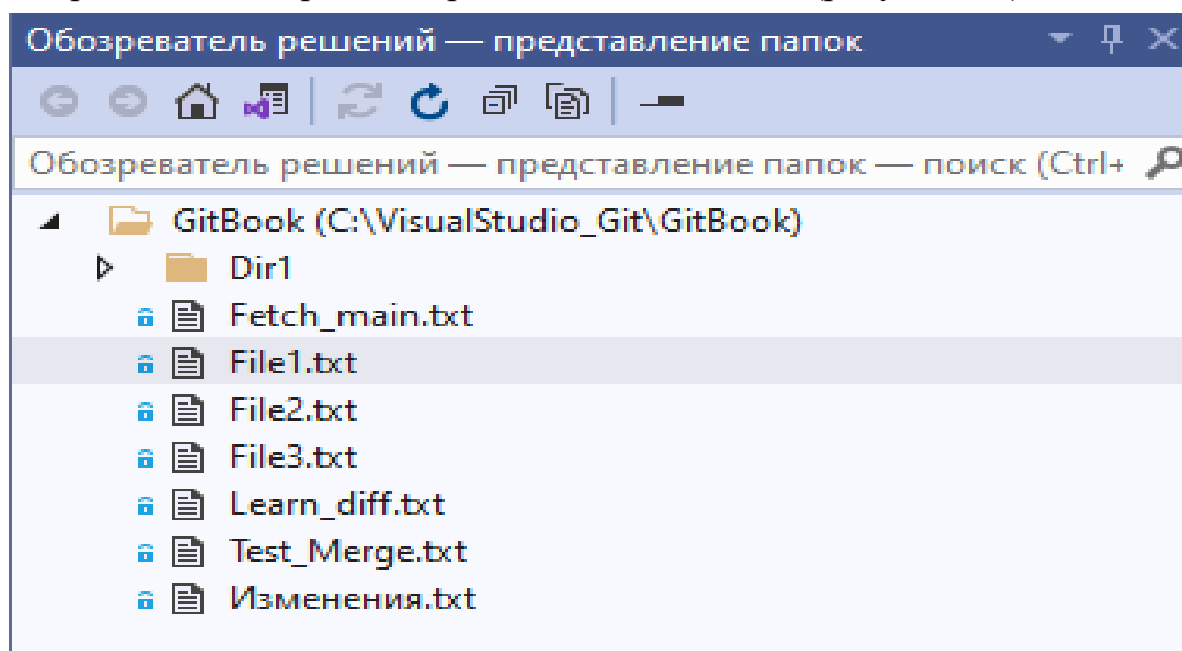


Рисунок 3.6 – Содержимое рабочей директории

3.1.2.2. Графическая схема репозитория

Visual Studio позволяет вывести графическую схему репозитория в центральной части рабочей области. Для добавления графической схемы необходимо:

- выбрать репозиторий из главного меню «Git/Локальные репозитории»;
- выбрать способ отображения графической информации из главного меню: «Git/Просмотреть журнал ветвей» или «Git/Управление ветвями». В первом случае будет выведена схема коммитов для текущей ветви. Во втором случае, в дополнение к схеме коммитов, будет выведено окно, в котором отображаются названия всех ветвей проекта.

Пример графической схемы в режиме «Управление ветвями» изображён на рисунке 3.7. При однократном щелчке левой кнопкой мышки по названию ветви в левой части окна произойдёт смена схемы коммитов в правой части окна. При двойном щелчке мышки по коммиту будет отображено состояние файлов проекта, соответствующее выбранному коммиту, и состояние файлов проекта, соответствующее предыдущему коммиту. Разметка файлов выделяет различия между коммитами.

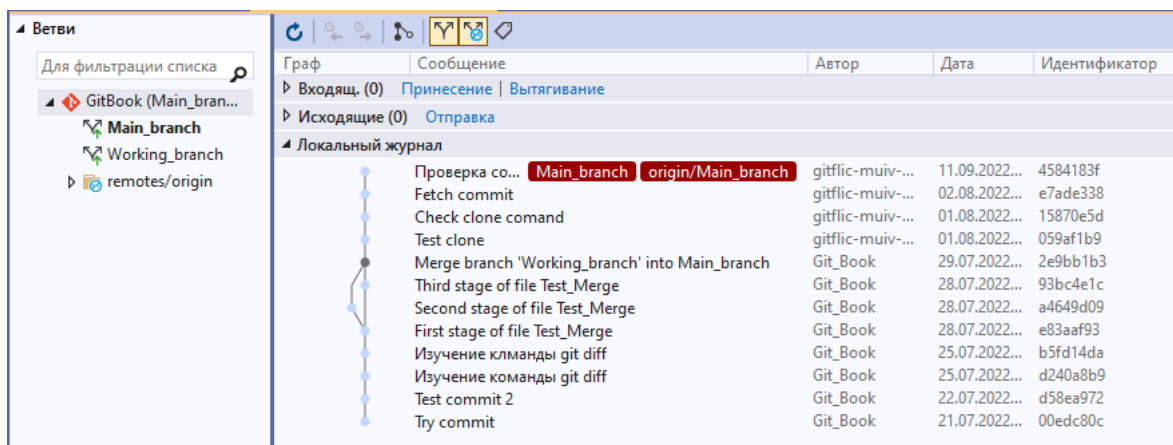


Рисунок 3.7. – Графическая схема коммитов
в режиме «Управление ветвями»

3.1.2.3. Выполнение команд *Commit* и *push* из проекта *Visual Studio*

Операции добавления файлов в индекс, создание коммита и выгрузка коммита во внешний репозиторий выполняются с помощью вкладки «Изменения Git», расположенной в правой части рабочей области Visual Studio (рисунок 3.8). Вкладка «Изменения Git» содержит список файлов проекта, которые были изменены, но не добавлены в коммит. При наведении курсора мышки на имя файла отобразятся иконки с кнопками управления:

- стрелка влево – щелчок мышкой по этой кнопке приведёт к отмене последних изменений;
- знак плюс – щелчок мышкой по этой кнопке приведёт к добавлению файла в индекс.

В верхнее окно редактирования вводится комментарий, который будет сохранён в коммите. Выпадающий список «Зафиксировать все» позволяет выбрать команды для выполнения следующих действий:

- зафиксировать все – будет создан коммит в локальном репозитории, включающий все файлы, которые были добавлены в индекс;
- зафиксировать все и отправить – будут последовательно выполнены команды `git commit` и `git push`;
- зафиксировать всё и синхронизировать – будут последовательно выполнены команды `git commit`, `git fetch` и `git push`;
- спрятать все – будет выполнена команда `git stash` с ключом `–include-untracked`;

– спрятать все и сохранить промежуточные данные – будет выполнена команда `git stash` с ключом `–keep-index`.

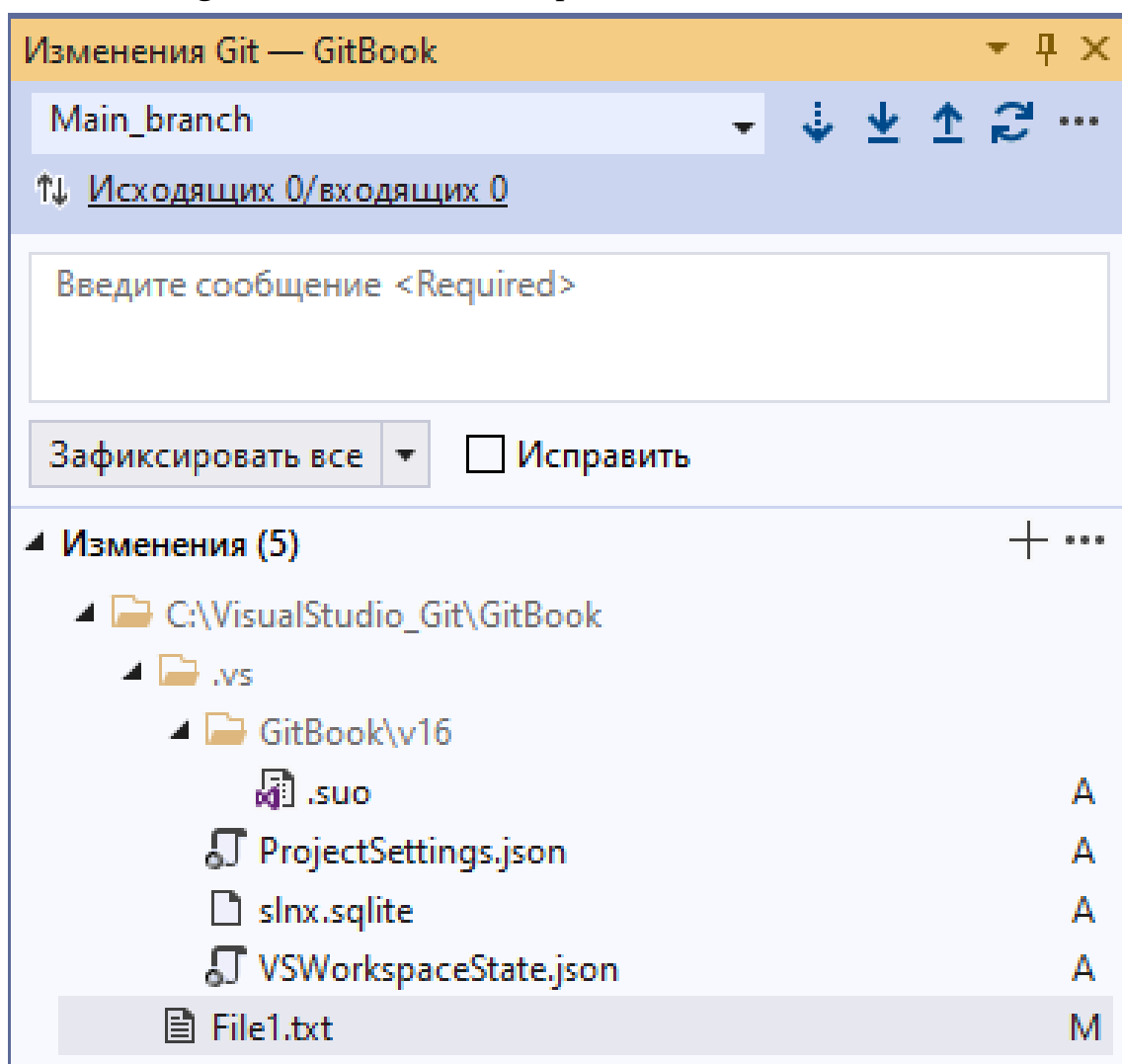


Рисунок 3.8 – Окно фиксации изменений файлов проекта

3.2. Учебный портал сайта GitFlic

Репозитории системы контроля версий Git могут располагаться не только на локальном компьютере, но и в облаке. Облачное размещение позволяет упростить организацию коллективного использования репозитория разработчиками программного обеспечения. Один из членов коллектива создаёт репозиторий, остальные клонируют его на локальные компьютеры и, по мере необходимости, сохраняют коммиты в облаке.

Сохранить репозиторий можно в любом облачном хранилище. Существуют и специализированные хранилища. Названия некоторых из них

начинаются со слова Git: GitHub, GitLab, GitLinux, GitFlic. В названиях других слова Git нет: Bitbucket, Beanstalk, SourceForge. Ни одно из таких хранилищ не является Git. Система контроля версий Git должна быть установлена на локальном компьютере, независимо от того, где размещается репозиторий, в одном из облачных хранилищ или на локальном диске.

Git можно использовать не только непосредственно для управления версиями, но и для других целей, связанных с хранением и обработкой программного кода. Одним из примеров такого использования является учебный портал сайта GitFlic. GitFlic является Git-репозиторием, и в то же время его учебный портал может использоваться для обучения студентов программированию как на ранних стадиях обучения, так и при курсовом (дипломном) проектировании.

В процессе обучения студенты получают от преподавателя задания на создание программного кода. В процессе подготовки решения студенты сохраняют в репозитории историю своей работы в виде git-коммитов. Преподаватель имеет возможность просматривать коммиты и оставлять комментарии с целью помочь студентам исправить имеющиеся ошибки.

В настоящем разделе описывается работа с учебным порталом сайта GitFlic.

3.2.1. Вход на учебный портал GitFlic

Для перехода к учебному portalу GitFlic наберите в адресной строке браузера:

edu.gitflic.ru

Введите логин и пароль. В случае успешного входа откроется приветственная страница портала.

Кликните мышкой по строчке со своими именем и фамилией в правом верхнем углу экрана. Откроется главная страница личного кабинета GitFlic.

Для перехода на главную страницу из любого места личного кабинета кликните мышкой на строчке со своими именем и фамилией в левом верхнем углу экрана.

Главная страница содержит:

- вертикальное меню, расположенное в её левой части;

- горизонтальное меню со вспомогательными ссылками, расположенное в нижней части;
- горизонтальное меню со списком вкладок, расположенное в верхней части;
- рабочую область, в центре.

3.2.2. Регистрация преподавателей и студентов

Преподавателей на учебном портале GitFlic регистрирует администратор портала. После регистрации преподавателю передаётся логин и пароль.

Студенты регистрируются с использованием ссылки-приглашения, которая создаётся преподавателем для каждой учебной группы. Для создания ссылки-приглашения необходимо выполнить следующие действия:

- выбрать из вертикального меню пункт «Сообщество»;
- из окна, расположенного в рабочей области, выбрать «Московский университет им. С.Ю. Витте»;
- из верхнего горизонтального меню выбрать вкладку «Группы»;
- кликнуть мышкой по кнопке «Добавить группу»;
- заполнить сведения о группе, кликнуть мышкой по кнопке «Сохранить»;
- найти вновь созданную группу в списке групп прямым поиском или с помощью поля «Поиск»;
- кликнуть по иконке «цепочка», расположенной справа от названия группы, в результате чего откроется ссылка-приглашение.

Каждый из студентов учебной группы переходит по полученной ссылке-приглашению и создаёт для себя логин и пароль, после чего происходит автоматическая регистрация.

После регистрации студенту необходимо отредактировать сведения о фамилии и имени в профиле. Для чего:

- щёлкнуть мышкой по временному имени в верхнем углу главного экрана (произойдёт переход на вкладку «Профиль»);
- щёлкнуть мышкой по кнопке «Редактировать профиль» (кнопка находится под большим кружком с буквой);

- ввести в соответствующие поля имя и фамилию;
- щёлкнуть мышкой по кнопке «Сохранить».

После регистрации студентов преподаватель, проводящий регистрацию студентов, должен подтвердить введенные студентом фамилию и имя. Для чего:

- перейти на вкладку «Группы» (см. выше);
- найти в списке требуемую группу, для чего в поле «Поиск» ввести номер группы. Кликнуть мышкой по названию группы. Откроется список студентов группы;
- если в списке студентов вместо фамилии и имени указан псевдоним, то кликнуть мышкой по имени студента. На открывшейся вкладке кликнуть мышкой по кнопке «Редактировать»;
- в окне редактирования сведений о студенте подтвердить фамилию и имя студента, щёлкнув мышкой по соответствующим подсказкам, расположенным под полями «Имя студента» и «Фамилия студента»;
- щёлкнуть мышкой по кнопке «Сохранить».

3.2.3. Взаимодействие преподавателя и студентов

Взаимодействие преподавателя и студента при использовании учебного портала GitFlic происходит по следующей схеме:

- преподаватель создаёт задания и распределяет их между студентами;
- получив задание, студент создаёт репозиторий на учебном портале GitFlic и связывает его с локальным репозиторием на рабочем компьютере;
- в ходе выполнения задания студент сохраняет промежуточные результаты в виде коммитов в локальном репозитории и, по мере готовности решения, синхронизирует локальный репозиторий с GitFlic;
- после завершения выполнения задания студент отправляет преподавателю ссылку на окончательную версию программного кода, хранящуюся в репозитории GitFlic;
- преподаватель проверяет правильность выполнения задания;
- в случае отсутствия ошибок преподаватель подтверждает факт выполнения задания;

– при наличии ошибок преподаватель добавляет в код студента комментарии с указанием на ошибки и возвращает работу на доработку.

В следующих разделах дано описание действий, выполняемых на всех этапах взаимодействия преподавателя и студента.

3.2.4. Создание заданий

Задания существуют в рамках лекций. В одной лекции может быть неограниченное количество заданий. Лекции существуют в рамках учебного курса. В рамках учебного курса может быть неограниченное количество лекций.

3.2.4.1. Создание учебного курса

Действие выполняет преподаватель.

Из левого вертикального меню выбрать пункт «Курсы». Из верхнего меню выбрать пункт «Неопубликованные». Кликнуть мышкой по прямоугольной кнопке «Новый курс». Появится окно ввода общей информации о курсе.

Ввести название и краткое описание курса. Выбрать из выпадающего списка название университета и кафедру.

ВНИМАНИЕ!

Информация о редакторе курса справочная и, с точки зрения дальнейшей работы с GitFlic, ни на что не влияет. Автор – преподаватель (несколько преподавателей), который сможет вносить изменения в структуру курса.

После окончания ввода всей информации нажать кнопку «Сохранить». В случае успешного добавления курса на экране появится титульная страница курса.

3.2.4.2. Добавление на курс преподавателей и студенческих групп

Действие выполняет преподаватель.

Из левого вертикального меню выбрать пункт «Курсы». Из верхнего меню выбрать пункт «Неопубликованные». Найти создаваемый курс и кликнуть мышкой по его названию. Появится титульная страница курса.

Кликнуть мышкой по кнопке «Перейти к курсу». Появится вкладка,

предназначенная для распределения заданий между студентами. Из верхнего горизонтального меню выбрать вкладку «Преподаватели и их группы». Кликнуть по кнопке «Назначить преподавателя и группу». Появится диалоговое окно добавления преподавателя.

Выбрать группу, с которой работает преподаватель, выбрать преподавателя. Если в выпадающем списке отсутствует нужная группа, начать вводить её название в поле «Группа». Если нужная группа уже зарегистрирована в GitFlic, её название появится в выпадающем списке.

После окончания ввода всей информации кликнуть мышкой по кнопке «Добавить». Повторить описанные действия для всех групп, обучающихся на курсе.

ВНИМАНИЕ!

Студенческие группы должны быть зарегистрированы как указано в п. 3.3.2.

3.2.4.3. Добавление лекций

Действие выполняет преподаватель.

Из левого вертикального меню выбрать пункт «Курсы». Из верхнего меню выбрать пункт «Неопубликованные». Найти создаваемый курс и кликнуть мышкой по его названию. Появится титульная страница курса.

Кликнуть мышкой по кнопке «Редактировать». Появится информация о курсе с возможностью редактирования содержания. Из верхнего горизонтального меню выбрать пункт «Структура».

Появится список занятий (изначально пустой). Нажать на кнопку «Добавить занятие» (кнопка расположена в рабочей области). Появится форма ввода данных о занятии.

Заполните форму:

- выберите тип занятия из выпадающего меню в верхнем поле ввода;
- введите название лекции и её краткое содержание.

После заполнения формы нажмите кнопку «Сохранить». Произойдёт возврат на вкладку «Структура», в рабочем окне отобразится информация о созданной лекции.

Для того чтобы задания, относящиеся к лекции, были доступны, лекцию необходимо опубликовать. Для публикации лекции кликните мышкой

по её названию на вкладке «Структура». Кликните мышкой по кнопке «Опубликовать», расположенной справа от названия лекции.

3.2.4.4. Добавление задания

Действие выполняет преподаватель.

Из левого вертикального меню выбрать пункт «Курсы». Из верхнего меню выбрать пункт «Неопубликованные». Найти создаваемый курс и кликнуть мышкой по его названию. Появится титульная страница курса.

Кликнуть мышкой по кнопке «Редактировать». Появится информация о курсе с возможностью редактирования содержания. Из верхнего горизонтального меню выбрать пункт «Структура».

Появится список занятий. Найти в списке лекцию, для которой будут добавляться задания. Справа от названия лекции расположены две иконки: «карандаш» и «стрелка». Кликните мышкой по иконке «стрелка». Появится кнопка «Добавить задание». Кликните мышкой по этой кнопке. В появившейся форме заполните поле «Название задания» и кликните мышкой по кнопке «Сохранить». Произойдёт возврат на вкладку «Структура», и ниже названия лекции будет отображено название созданного задания (при необходимости кликните мышкой по иконке «стрелка», чтобы раскрыть описание лекции).

Кликните мышкой по названию задания. Появится форма для ввода содержательной части задания. Введите содержание задания в текстовое поле «Текст задания». При необходимости прикрепите файлы с материалами задания. Для того чтобы сделать задание доступным для отправки студентам, кликните по кнопке «Опубликовать». После окончания редактирования кликните по кнопке «Сохранить».

В правой части рабочего окна находится список заданий для текущей лекции. При клике мышкой по названию лекции откроется чистая форма для ввода нового задания. При клике мышкой на названии задания откроется существующее задание для редактирования.

После окончания ввода всех заданий кликните по пункту «Структура» в верхнем горизонтальном меню. Произойдёт возврат к списку занятий.

3.2.4.5. Распределение заданий между студентами

Действие выполняет преподаватель.

Из левого вертикального меню выбрать пункт «Курсы». Из верхнего

меню выбрать пункт «Неопубликованные». Найти создаваемый курс и кликнуть мышкой по его названию. Появится титульная страница курса.

Кликнуть мышкой по кнопке «Перейти к курсу». Откроется форма «Управление». Выбрать «Назначение занятий и задач» из верхнего горизонтального меню. Появится форма со списком занятий и задач.

Для выбранного занятия кликните мышкой по иконке «Человечек». В появившейся форме выберите группу, которой необходимо назначить задание.

ВНИМАНИЕ!

Можно выбирать только те группы, которые были добавлены на курс (см. раздел 3.3.4.2).

3.2.4.6. Создание репозитория

Действие выполняет студент.

Репозиторий (проект) создаётся студентом перед началом выполнения задания. Один и тот же репозиторий можно использовать для выполнения различных заданий, однако рекомендуется создавать репозиторий для каждого задания, чтобы сделать распределение решений систематизированным.

Для создания репозитория (проекта) кликните мышкой по значку + в правом верхнем углу экрана, рядом с буквой G в зелёном кружке. В выпадающем списке кликните мышкой по тексту «Новый проект». Появится вкладка с шаблоном для описания нового проекта.

Введите название проекта латинскими буквами. Выберите язык программирования из выпадающего списка.

Обратите особое внимание на поле «URL проекта». В нём указан сетевой адрес, который позволит связать репозиторий GitFlic с локальным репозиторием вашего рабочего компьютера.

Адрес, указанный в поле «URL команды», нужно изменить. Если исходный адрес:

`https://gitflic.ru/project/student/my_project,`

то адрес для связи с репозиторием будет:

`https://edu.gitflic.ru/project/student/my_project.git`

(полужирным выделены кодовые слова **edu** и **.git**, которые нужно добавить в исходную адресную строку).

После ввода всей информации о проекте кликните по кнопке «Создать

проект».

Существующий проект можно найти на вкладке «Проекты» главной страницы личного кабинета.

3.2.4.7. Отправка задания на проверку

Действие выполняет студент.

Перейдите на главную страницу личного кабинета GitFlic. Кликните мышкой по пункту «Задания» в левом вертикальном меню. Появится вкладка со списком заданий, назначенных студенту для выполнения. Задания разделены на четыре группы:

- новые (не выполненные задания);
- на проверке (выполненные задания, находящиеся на проверке у преподавателя);
- проверенные (выполненные задания, по которым преподаватель принял положительное решение);
- отклонённые (задания, которые преподаватель вернул на доработку).

Для просмотра группы её название необходимо выбрать из верхнего горизонтального меню.

Найдите в группе «Новые» название задания, которое вы хотите отправить на проверку. Кликните мышкой по кнопке «Отправить решение» справа от названия задания. Откроется вкладка для выбора решения.

В поле «Проекты» выберите из выпадающего списка название репозитория, в котором хранится выполненное задание.

В поле «Путь к файлу» выберите из выпадающего списка название файла, содержащего решение. Если названия файла в выпадающем списке нет, начните вводить название, и оно появится в выпадающем списке.

Если необходимо отправить несколько файлов, то они должны быть собраны вместе, в одной директории. Имя этой директории нужно указать в поле «Путь к папке».

После заполнения всех полей нажмите кнопку «Сохранить».

3.2.4.8. Проверка заданий

Действие выполняет преподаватель.

Перейдите на главную страницу личного кабинета GitFlic. Кликните мышкой по пункту «Задания» в левом вертикальном меню. Выберите «Панель преподавателя» из верхнего горизонтального меню.

Выберите пункт «На проверке» из верхнего, горизонтального меню. Отобразится список работ, которые студенты прислали на проверку.

Кликните мышкой по названию работы, которую вы хотите проверить. Откроется вкладка с описанием работы.

Кликните мышкой по пункту «Файлы» в верхнем горизонтальном меню. Откроется текстовый редактор, в котором можно просматривать содержимое всех файлов, присланных студентом для текущей работы.

При наведении курсора мышки на текст слева появится небольшой синий квадратик со значком + внутри. Если кликнуть мышкой по этому квадратику, то откроется поле, в котором можно ввести комментарий. После ввода комментария кликните мышкой по кнопке «Комментировать». Комментарий будет сохранён в том месте файла, где находился курсор мышки при нажатии на синий квадратик.

После проверки работы кликните мышкой по кнопке «Выполнено», если решение верное, или по кнопке «Не выполнено», если решение содержит ошибки.

На панели преподавателя работа переместится во вкладку «Проверенные», если решение было верное, или во вкладку «Отклонённые», если решение содержит ошибки.

3.2.4.9. Просмотр результатов проверки

Действие выполняет студент.

Перейдите на главную страницу личного кабинета GitFlic. Кликните мышкой по пункту «Задания» в левом вертикальном меню. Выберите «Проверенные» или «Отклонённые» из верхнего горизонтального меню. Все задания, находящиеся в группе «Проверенные», приняты преподавателем, никаких дополнительных действий выполнять не нужно.

Задания, находящиеся в группе «Отклонённые», содержат ошибки. Кликните мышкой по названию задания из этой группы. Откроется вкладка с описанием работы.

Выберите пункт «Файлы» из верхнего горизонтального меню. Откроется редактор, в котором можно просмотреть текст работы с комментариями преподавателя.

После изучения комментариев преподавателя необходимо исправить указанные ошибки, сохранить исправленный файл/файлы в репозитории и отправить работу на повторную проверку.

Для отправки работы на повторную проверку необходимо кликнуть мышкой по кнопке «Изменить решение», в появившейся форме заполнить поля (см. п. 3.3.4.7) и кликнуть мышкой по кнопке «Сохранить».

3.3. Репозиторий GitFlic

Проект GitFlic включает в себя не только учебный портал, но и репозиторий общего назначения. Этим репозиторием могут воспользоваться программисты, разрабатывающие как коммерческое, так и некоммерческое программное обеспечение. Для небольших групп программистов использование GitFlic бесплатно. Крупные программистские коллективы должны сотрудничать с GitFlic на коммерческой основе.

3.3.1. Структура окна сайта GitFlic

После входа на сайт GitFlic открывается окно, состоящее из следующих четырёх областей:

Горизонтальное меню

Представляет собой чёрную полосу в верхней части экрана.

В левой части горизонтального меню расположена эмблема компании GitFlic – стилизованное изображение головы медведя, рядом с которой расположена надпись GitFlic. Щелчок мышкой по эмблеме приводит к переходу на страницу профиля пользователя.

В правой части горизонтального меню расположены:

- Строка поиска, помогающая найти нужный проект.
- Иконка «колокольчик», клик мышкой по которой открывает экран уведомлений.
- Знак плюс, позволяющий переключиться на один из диалогов: создание нового проекта, создание новой команды разработчиков, создание новой компании.
- Иконка с обозначением текущего пользователя. Клик мышкой по ней позволяет выполнить одно из трёх действий: открыть профиль пользователя, открыть экран с настройками или выйти из системы.

Вертикальное меню

Расположено в левой части экрана и содержит следующие шесть пунктов:

- Проекты. Переход к экрану работы с проектами.
- Команды. Переход к экрану создания и редактирования команд разработчиков.
- Компании. Переход к экрану создания и редактирования производственных компаний.
- Уведомления. Переход к экрану работы с уведомлениями.
- Помощь. Переход к контекстной справке по системе.
- Выйти. Выход из системы.

Нижнее меню

Расположено в нижней части экрана. Позволяет обратиться к юридическим документам, регламентирующим работу с GitFlic, найти контакты разработчиков GitFlic и перейти на страницу, содержащую публичные проекты.

Центральная часть экрана

Содержимое этой части зависит от того, с какой из страниц интерфейса в данный момент работает пользователь. Центральная часть экрана имеет собственное меню, назначение пунктов которого будет описано при рассмотрении соответствующей страницы интерфейса.

3.3.2. Создание нового репозитория

В верхнем меню щёлкните мышкой по знаку плюс и выберите «Новый проект». Заполните форму создания проекта. Обратите внимание на поле URL. Оно состоит из двух частей: фиксированной и редактируемой. Фиксированная часть задаёт сетевой путь к папке с проектами пользователя. Редактируемая часть позволяет задать имя вложенной папки, в которой будет храниться создаваемый проект. Запомните полный путь, он пригодится для подключения к проекту локального репозитория.

Переключатель «Публичный проект»/«Приватный проект» позволяет выбрать уровень доступа к создаваемому проекту. Публичный проект будет виден всем посетителям сайта GitFlic, даже не зарегистрированным. Приватный проект будет доступен только пользователю, создавшему его (хозяин проекта) и тем пользователям, кому хозяин проекта предоставит доступ.

После заполнения полей нажмите кнопку «Создать проект». Откроется страница вновь созданного проекта. До добавления файлов в проект на этой странице будет находиться подробная информация, содержащая команды git, позволяющие подключить к проекту локальный репозиторий. На странице приводятся следующие группы команд.

Глобальные настройки Git. Эти команды необходимо выполнить один раз, после установки git на компьютер.

```
git config --global user.name "MyName"  
git config --global user.email MyMail@MyServer.com
```

Клонирование удалённого репозитория (проекта) на локальный компьютер. При клонировании автоматически создаётся новая директория, имя которой совпадает с именем проекта.

```
git clone https://gitflic.ru/project/MyName/testovyyj-proekt.git  
cd testovyyj-proekt  
touch README.md  
git add README.md  
git commit -m "add README"  
git push -u origin master
```

Преобразовать существующую директорию в репозиторий git и копирование в неё проекта.

```
cd existing_folder  
git init  
git remote add origin https://gitflic.ru/project/abulychev/testovyyj-proekt.git  
git add  
git commit -m "Initial commit"  
git push -u origin master
```

Переименовать удалённый репозиторий и скопировать его на локальный компьютер.

```
cd existing_folder
```

```
git remote rename origin old-origin
git remote add origin https://gitflic.ru/project/abulychev/testovyj-proekt.git
git push -u origin --all
git push -u origin --tags
```

3.3.3. Управление проектом

При выборе проекта открывается обзорная страница, на которой находятся:

- описание проекта;
- адрес сайта проекта;
- используемый язык программирования;
- содержание приложенного README-файла.

Для управления состоянием проекта предназначены кнопки «Наблюдать», «Форк», «Избранное».

Кнопка «Наблюдать» используется для управления потоком уведомлений. При её нажатии появляется выпадающее меню, в котором возможно выбрать режим для получения уведомлений:

- получать все уведомления;
- получать только уведомления, в которых упоминается пользователь;
- не получать уведомления.

Кнопка «Форк» позволяет создать новую ветку, используя текущий проект в качестве начальной точки.

Кнопка «Избранное» позволяет добавить проект в список избранных. Статус «Избранное» учитывается при использовании фильтра «Избранное» на странице «Проекты».

В верхней части окна проекта расположено дополнительное горизонтальное меню, включающее пункты:

- Файлы – обзор файлов проекта.
- Проблемы – создание и управление задачами, возникающими в ходе выполнения проекта.
- Запросы на слияние – список запросов на слияние веток проекта и инструменты для управления слияниями.
- Коммиты – список выполненных коммитов.
- Ветки – список веток проекта.

- Теги – список тегов, используемых в проекте.
- Релизы – информация о выпущенных релизах.
- Статистика – статистические данные о проекте.
- Настройки – управление настройками проекта.

Ветки проекта выбираются в выпадающем меню над списком файлов (рисунок 3.9). После выбора ветки можно просмотреть загруженные файлы проекта.

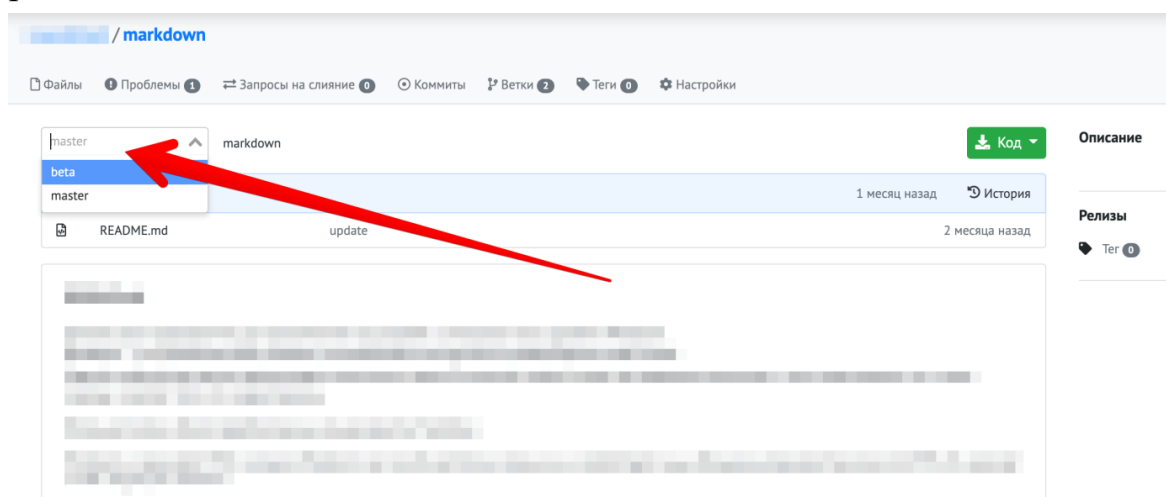


Рисунок 3.9 – Выбор ветки проекта

Копирование кода проекта (рисунок 3.10). Чтобы скопировать код проекта, откройте меню «Код» (расположено на зелёной плашке). Выберите вариант авторизации. Авторизация, при копировании проекта, может производиться по протоколу HTTPS или по протоколу SSH.

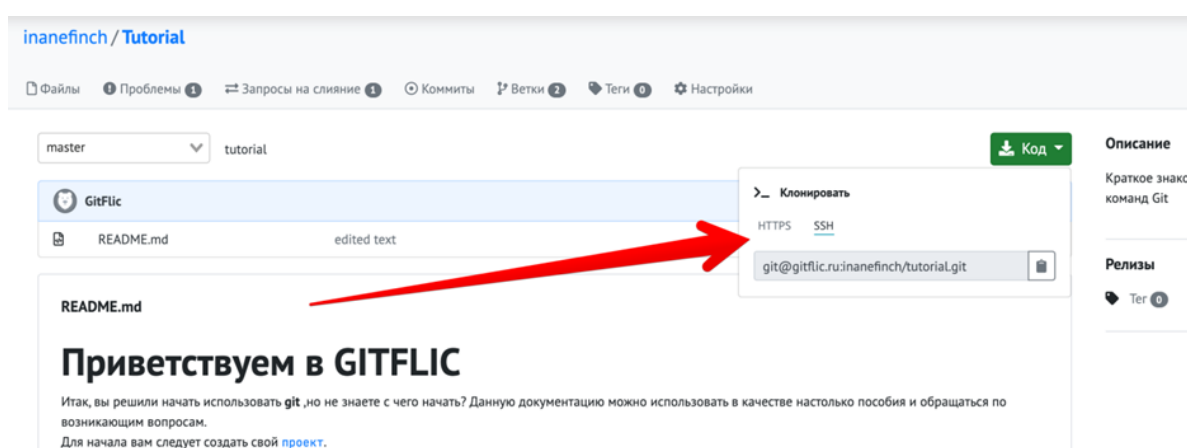


Рисунок 3.10 – Копирование кода проекта

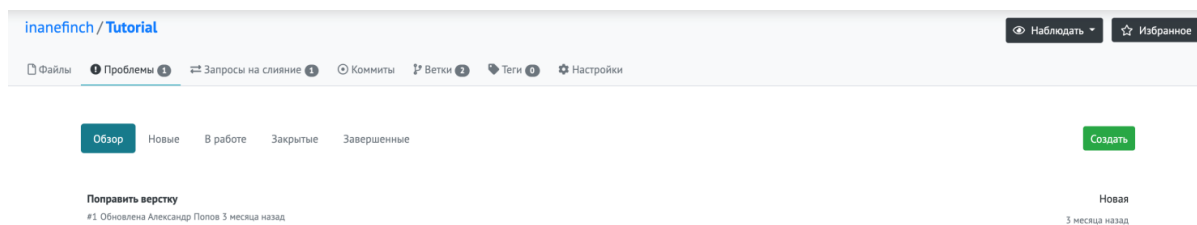


Рисунок 3.11 – Создание проблемы

«Проблемы». На вкладке «Проблемы» (рисунок 3.11) находятся описания текущих задач, выполняемых в ходе проекта. В проблемах описываются ошибки, которые необходимо исправить, и задачи, которые необходимо выполнить для исправления/улучшения проекта. У проблем имеются статусы выполнения. По статусам проблемы делятся на «Новые», «В работе», «Закрывать», «Завершённые». Предусмотрена возможность включения фильтра для выбора проблем на основании их статуса. На предпросмотре для проблемы указаны: статус; дата последнего обновления; имя пользователя, который последним взаимодействовал с проблемой; тег проблемы.

Для создания новой «Проблемы» щёлкните мышкой по кнопке «Создать проблему». На странице создания проблемы укажите заголовок и описание проблемы. В правой части формы укажите начальный статус проблемы, назначьте ответственных (их можно выбрать из списка членов команды или компании) и присвойте проблеме лейбл (релиз, фикс, ошибка, фича). После заполнения всех полей нажмите «Создать». Новая проблема отобразится в списке.

Запросы на слияние. На странице запросов на слияние находится список созданных запросов и кнопка для создания новых. В списке отображён статус созданных запросов, также можно открыть запрос для просмотра изменений. После открытия запроса появятся кнопки, используя которые, возможно выполнить следующие действия:

- обсудить запрос на слияние в дискуссиях;
- просмотреть вкладку с коммитами, которые входят в данный запрос;
- просмотреть вкладку, которая наглядно отображает изменения в файлах.

Для создания нового запроса на слияние в выпадающем меню выберите ветку, которую собираетесь слить, и нажмите кнопку «Создать». На странице создания выберите ветку, с которой будет происходить слияние. Укажите название и описание для запроса на слияние, укажите ответственных пользователей, рецензентов, которые будут подавать свой голос за принятие или отклонение запроса на слияние. Также можете указать лейблы к запросу на слияние. После заполнения всех полей нажмите кнопку «Создать».

Для принятия запроса на слияние внизу страницы нажмите «Слить» (рисунок 3.12). Чтобы отклонить запрос на слияние или закрыть его, на кнопке рядом со «Слить» нажмите стрелку и выберите необходимый вариант.

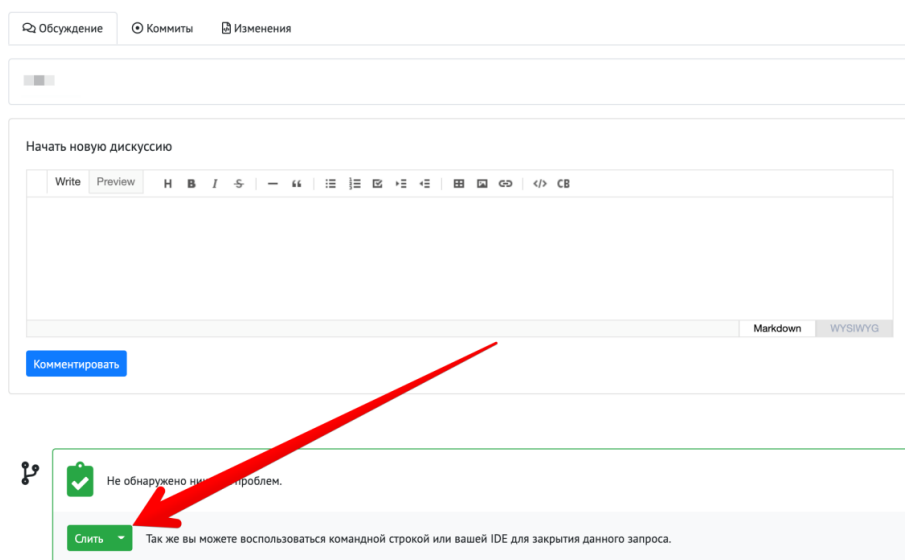


Рисунок 3.12 – Подтверждение слияния веток

Коммиты. На вкладке коммиты (рисунок 3.13) находится история всех коммитов проекта. Для просмотра коммитов, относящихся к одной ветке, переключите ветку в выпадающем меню. Нажмите на интересующий вас коммит для просмотра изменений, зафиксированных в этом коммите.

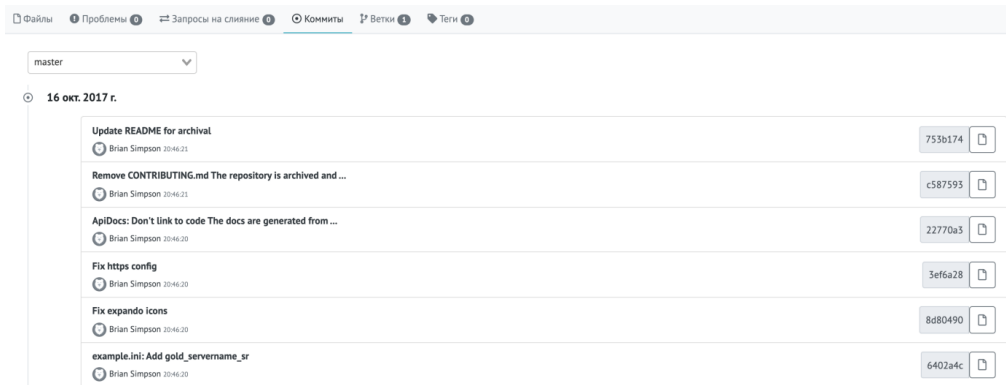


Рисунок 3.13 – Список коммитов 1

Ветки. На вкладке ветки находится информация обо всех созданных ветках и их текущем статусе. На странице отображена информация о стандартной ветке, рабочей, активной, а также все заброшенные ветки, которые ранее были использованы в работе с проектом.

3.3.4. Администрирование и просмотр публичных проектов

3.3.4.1. Клонирование репозитория

Клонирование репозитория выполняется с использованием команды меню «Код» (расположено на зелёной плашке). Выберите протокол подключения (HTTPS или SSH) и выполните команду клонирования для создания локальной копии. При использовании протокола ssh необходимо произвести дополнительную настройку аккаунта, добавив SSH ключ в меню Настройки/Ключи.

3.3.4.2. Избранное

Добавить проект в избранное можно, нажав кнопку «Избранное» (рисунок 3.14) в горизонтальном меню на вкладке проекта. После добавления проекта в избранное он будет доступен на вкладке с избранными проектами на главной странице личного кабинета пользователя GitFlic. Для удаления проекта из списка избранных повторно нажмите на кнопку «Избранное». Список избранных проектов является непубличным и доступен только пользователю, создавшему его.

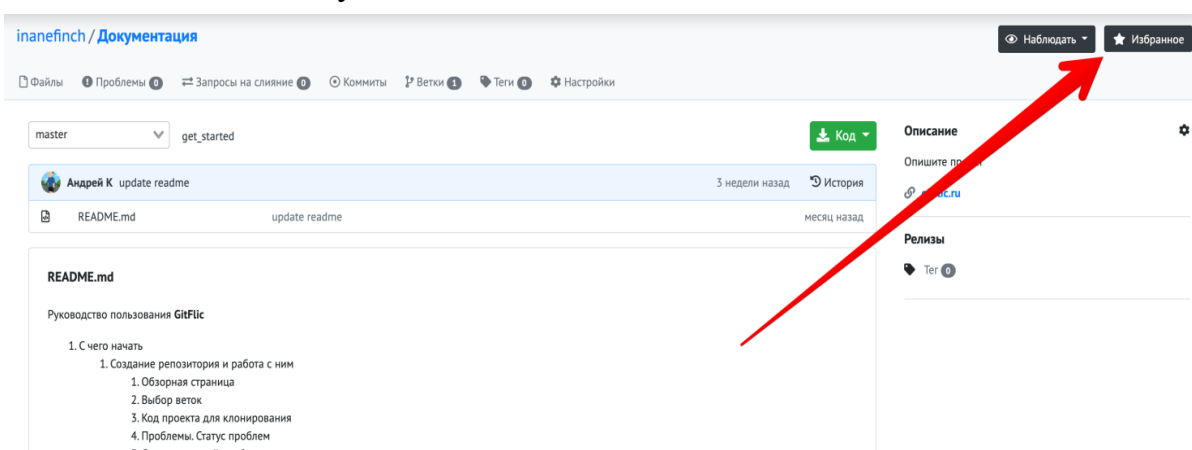


Рисунок 3.14 – Добавление проекта в избранное

3.3.4.3. Переименование проекта

Для переименования проекта откройте страницу настроек проекта. При переименовании существуют два сценария действий:

1. В поле с именем необходимо изменить название проекта. При этом в поле внутри жёлтой зоны настроек изменится его веб-адрес. Изменения произойдут в переменной части адреса. Поменяется транслитическое написание в соответствии с новым названием.

2. Чтобы изменить только URL репозитория, в поле внутри жёлтой зоны настроек укажите желаемое название переменной части адреса.

После нажатия кнопки «Переименовать» прежний URL репозитория будет недоступен. Все внешние ссылки на проект будет необходимо заменить на новый адрес.

ВНИМАНИЕ!

Не меняйте URL репозитория, если не уверены в своих действиях и не осознаете всех возможных последствий.

3.3.5. Видимость проекта

Возможно изменить видимость проекта, сделав его публичным или приватным. Для изменения откройте страницу с настройками, опуститесь к красной зоне и нажмите «Изменить видимость» напротив надписи «Изменить видимость репозитория» (рисунок 3.15).

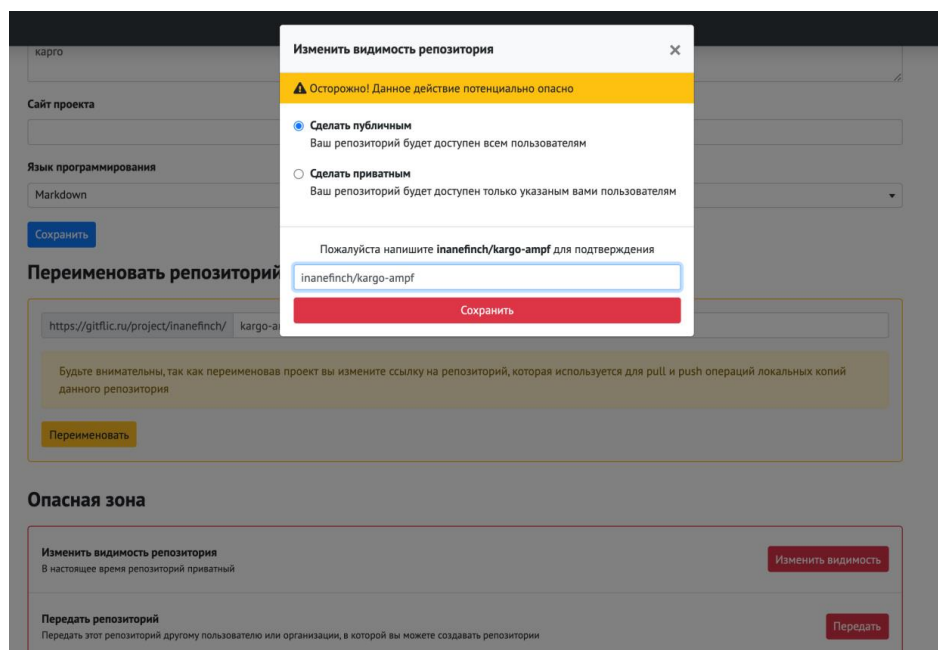


Рисунок 3.15 – Изменение видимости проекта

Укажите необходимую опцию (публичный/приватный) и напишите контрольную фразу для подтверждения действия, затем нажмите кнопку «Сохранить».

3.3.6. Передача репозитория другому владельцу

Для передачи репозитория перейдите в настройки к красной зоне. Откройте меню передачи, нажав кнопку «Передать». Укажите, кому хотите передать репозиторий (у вас должны быть права на создание проектов в команде или компании, которой вы передаёте проект).

3.3.7. Добавление пользователей в проект

Добавление пользователей в проект происходит на странице настроек проекта. На этой странице предусмотрено правое вертикальное меню. В этом меню выберите «Управление доступом». Произойдёт переключение на страницу «Участники проекта». Выберите вкладку «Пригласить пользователя». Укажите имя пользователя и дайте ему роль в проекте (пользователь должен быть зарегистрирован на сайте GitFlic). На бесплатном тарифе вы можете добавить только 4 новых участников. Для добавления большего количества участников проекта необходимо перейти на платный тариф.

3.3.8. Просмотр списка публичных проектов

Для того чтобы перейти к списку публичных проектов, в нижнем меню выберите пункт «хранилище кода». На экране появится список всех публичных проектов, созданных на сайте GitFlic.

На сайте реализован поиск по списку публичных проектов. Поле поиска находится в верхнем, горизонтальном меню. Для поиска по названию введите в поле часть названия проекта и нажмите на кнопку с увеличительным стеклом. Произойдёт переход на страницу с результатами поиска. В правой, верхней части страницы результатов находится таблица со статистикой распределения проектов по языкам программирования. Эта таблица является интерактивной. Для уточнения поиска воспользуйтесь фильтром по языкам программирования, выбрав соответствующую строчку таблицы. Чтобы найти конкретный проект, введите в поле поиска полное, максимально точное название проекта.

Авторизованный пользователь может просматривать в разделе с проектами все публичные проекты пользователей, личные проекты, проекты, отмеченные как избранные, и проекты, в которых пользователь принимает участие (проекты других пользователей или компаний, в которые был приглашен другими пользователями сервиса).

3.3.9. Создание нового проекта и управление настройками

3.3.9.1. Создание проекта

В верхнем меню откройте пункт, обозначенный знаком плюс (рисунок 3.16), выберите команду «Новый проект». Откроется страница создания нового проекта, на которой расположена форма создания проекта. Заполните форму и нажмите кнопку «Создать проект».

Владельцем проекта устанавливается текущий активный пользователь. Для смены владельца проекта нажмите на выпадающее меню и выберите нужного владельца для создаваемого проекта. Владелец проекта может быть как пользователь, так и команда или компания. В меню выбора владельца будут отображены все доступные на данный момент команды и компании, в которые текущий пользователь может добавлять проекты.

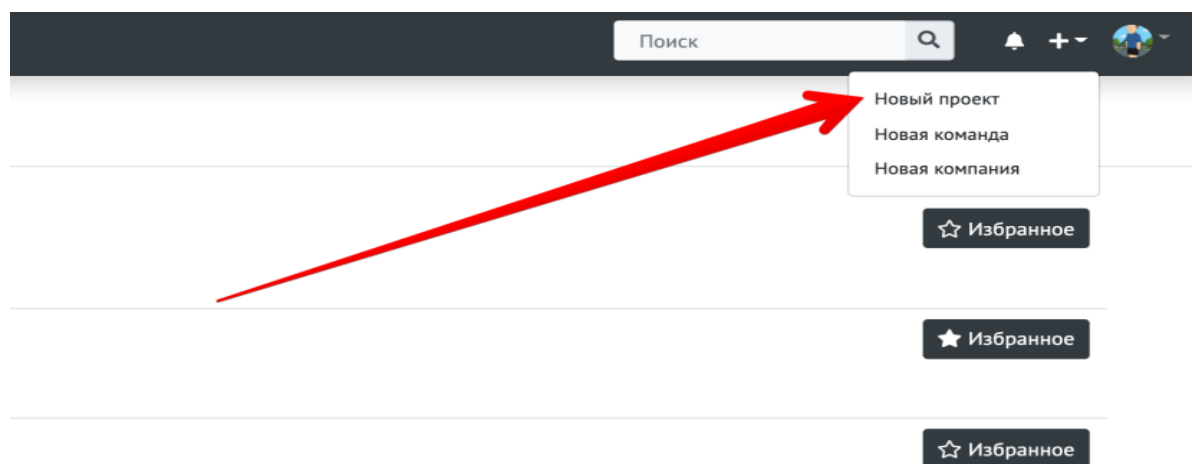


Рисунок 3.16 – Создание нового проекта

В поле «Название проекта» введите желаемое название проекта.

Укажите, если требуется, язык программирования для проекта из выпадающего меню. По умолчанию установлен Markdown.

Вы можете указать свой URL для проекта, если вам не подходит автоматический вариант названия.

Поле с описанием является необязательным. Если присутствует, то содержит краткое описание проекта.

После заполнения поля с описанием следует выбрать настройку приватности вашего проекта. Возможные варианты:

- «Публичный», проект будет доступен для всех в интернете.
- «Приватный» проект доступен только автору и пользователям, которым автор явно предоставил доступ.

После заполнения всех полей нажмите «Создать проект».

3.3.9.2. Изменение настроек проекта

Процедура изменения настроек проекта подробно описана в разделе 3.3.4. В текущем разделе приводятся только краткие сведения.

Для изменений настроек проекта выберите пункт «Настройки» дополнительного горизонтального меню. Откроется страница настроек проекта, на которой можно изменить параметры, указанные при создании проекта.

Внимание! Доступ к настройкам проекта имеет только владелец, либо назначенный администратор.

На странице настроек можно изменить URL проекта. Нужно иметь в виду, что старый адрес после сохранения перестанет работать.

Внимание! Не изменяйте адрес проекта, если не уверены в последствиях.

Чтобы изменить видимость проекта, нажмите на указанную кнопку. В открывшемся окне измените параметр видимости проекта и введите контрольную строку для подтверждения действия (её можно скопировать и вставить).

Передача проекта происходит аналогично процессу смены видимости и требует ввода фразы для подтверждения действия, однако, для передачи

проекта новый владелец должен подтвердить действие передачи, в противном случае проект останется связанным с текущим профилем.

Для добавления новых участников проекта откройте вкладку настройки/управление доступом (рисунок 3.17). Для поиска пользователя, которого вы собираетесь добавить, нажмите на поле «Пользователь», откроется выпадающее меню поиска. Начните вводить имя пользователя или имя профиля, поиск начнётся после ввода не менее 3 символов. После выбора пользователя укажите роль для нового участника вашего проекта. Для удаления пользователей из списка участников напротив имени пользователя нажмите на кнопку «Удаление».

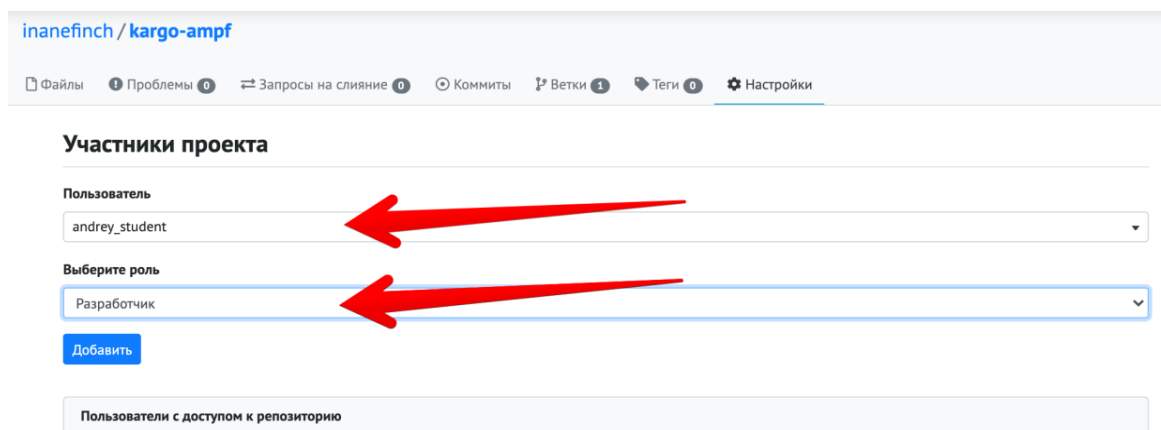


Рисунок 3.17 – Добавление пользователя

3.3.9.3. Настройка основных веток проекта

Для настройки веток по умолчанию выберите пункт «Ветки» из правого вертикального меню на вкладке «Настройки» (рисунок 3.18).

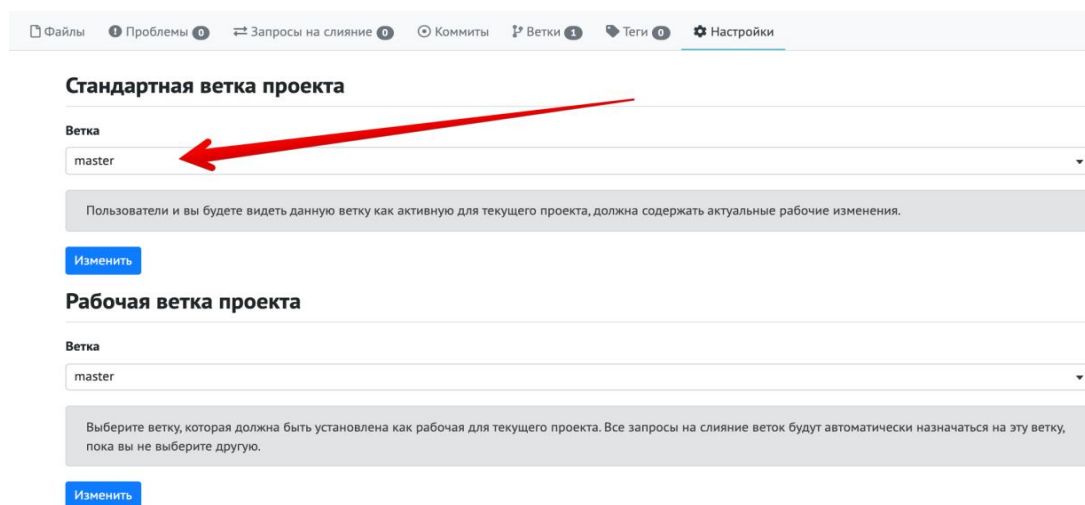


Рисунок 3.18 – Настройка веток проекта

Можно настроить значения по умолчанию для следующих веток проекта:

- «Стандартная ветка проекта» – ветка, которая будет отображена при просмотре проекта;
- «Рабочая ветка проекта» – ветка, в которую по умолчанию будут загружаться изменения, если не указана какая-либо другая ветка.

3.3.10. Работа с аккаунтом

В настоящем разделе описываются действия с аккаунтом пользователя, такие как создание, заполнение профиля, изменение и т.п.

3.3.10.1. Профиль пользователя

Профиль пользователя (рисунок 3.19) содержит личную информацию о пользователе GitFlic. Пользователь сам решает, какую информацию включать в свой профиль, чтобы гости и другие пользователи имели представление о нём на GitFlic.

Публичный профиль



Имя
GitFlic

Фамилия
User

Биография
лучший профиль на весь край...

Сайт
internet.ru

Компания
ru-internet

Выберите файл Обзор

Сохранить

Профиль
Аккаунт
Ключи
Пароль
Приложения OAuth

Рисунок 3.19 – Создание профиля пользователя

Поле для ввода имени и фамилии находится на главной странице настроек профиля.

Биография – краткая, деловая характеристика пользователя. Приветствуется описание опыта работы, списка компаний, в которых пользователь работал, профессиональных навыков, уровня образования.

Если у пользователя есть персональный сайт, можно указать его URL. Вместо URL возможно разместить ссылку на аккаунты в социальных сетях.

Имеется возможность указать название компании, в которой пользователь работает или которую представляет.

Незаменимым атрибутом профиля является аватар. Добавьте изображение, чтобы профиль стал уникальным и узнаваемым. Приветствуются как личные фото, так и изображения, характеризующие личность пользователя. Для добавления фото нажмите кнопку «Обзор». Появится стандартное окно выбора файла. Выберите файл с необходимым изображением.

Текущее имя можно изменить в любой момент на любое свободное. Если желаемое имя занято, то рекомендуется использовать альтернативное написание, например, с использованием чисел. Если пользователь обнаружил неправомерное использование своего имени (имени компании), следует подать в службу поддержки жалобу по поводу неправомерного использования товарного знака.

3.3.10.2. Настройка аккаунта

Перейдите на главную страницу своей учётной записи, кликнув мышкой на своём логине в левом верхнем углу экрана.

На главной странице нажмите кнопку «Редактировать профиль». Из правого вертикального меню выберите пункт «Аккаунт». Откроется страница редактирования данных аккаунта, на которой можно выполнить следующие действия.

Изменить логин GitFlic на любой свободный (рисунок 3.20). Следует учитывать, что после сохранения нового логина предыдущий URL проекта станет недоступен, так как он будет изменён в соответствии с новым именем профиля (имя профиля является частью URL проекта).

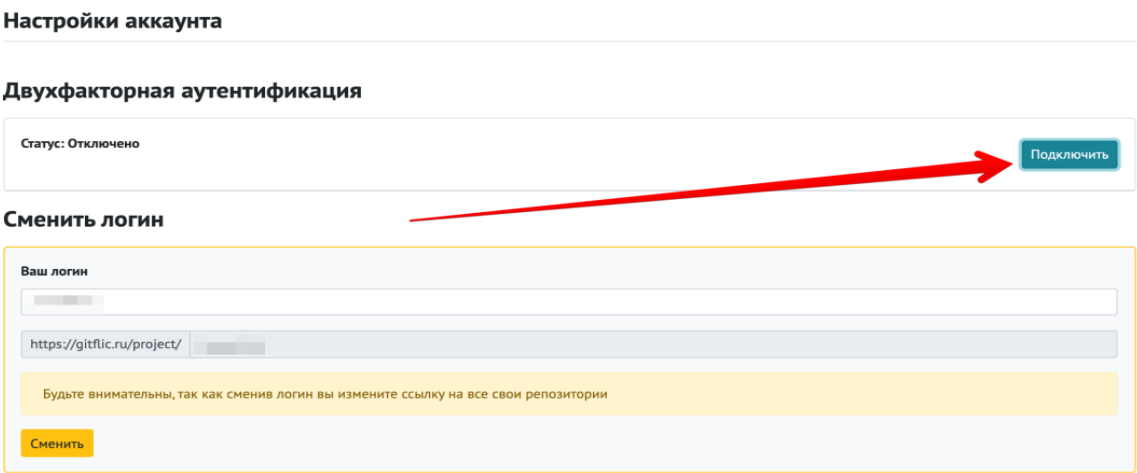


Рисунок 3.20 – Смена логина пользователя

Включение двухфакторной аутентификации. Двухфакторная аутентификация позволяет защитить аккаунт при помощи генерации временных секретных кодов. Для подключения аутентификации потребуется использование мобильного телефона.

Настроить аутентификацию по SSH-ключу. В GitFlic есть возможность подключить аутентификацию по публичному SSH-ключу. Следующая процедура используется для проверки наличия SSH-ключа и создания нового SSH-ключа (при необходимости).

Для того чтобы убедиться в отсутствии SSH-ключа, на локальном компьютере выполните следующие команды в командной строке операционной системы:

```
cd ~/.ssh  
ls
```

Найдите в списке файл с именем *id_dsa* или *id_rsa* и одноименный файл с расширением *.pub*. Файл с расширением *.pub* – это публичный ключ, а второй файл – приватный ключ. Если указанные файлы отсутствуют (или отсутствует директория *.ssh*), то у пользователя нет SSH-ключа. Для создания SSH-ключа используются команды:

```
cd ~/.  
ssh-keygen -o
```

Если на экране появился следующий текст, значит всё сделано верно, и ключ был создан.

```
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/gitflic_user/.ssh/id_rsa):  
Created directory '/home/gitflic_user/.ssh'.  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /home/gitflic_user/.ssh/id_rsa.  
Your public key has been saved in /home/gitflic_user/.ssh/id_rsa.pub.  
The key fingerprint is:  
d0:82:24:8e:d7:f1:bb:xx:yy:zz:96:93:49:da:9b:e3 gitflic\_user@gitflic.ru
```

Следующий шаг – получение публичного SSH-ключа из файла. Откройте файл с публичным ключом (расширение *pub*) в текстовом редакторе и полностью скопируйте содержимое.

На странице профиля выберите пункт «Ключ» из правого вертикального меню. Появится вкладка «SSH-ключи». Вставьте публичный ключ в соответствующее поле, введите название ключа (произвольное). Если хотите, чтобы ключ был бессрочным, не заполняйте поле «Дата окончания действия».

Изменение пароля. Имеется возможность в любой момент заменить

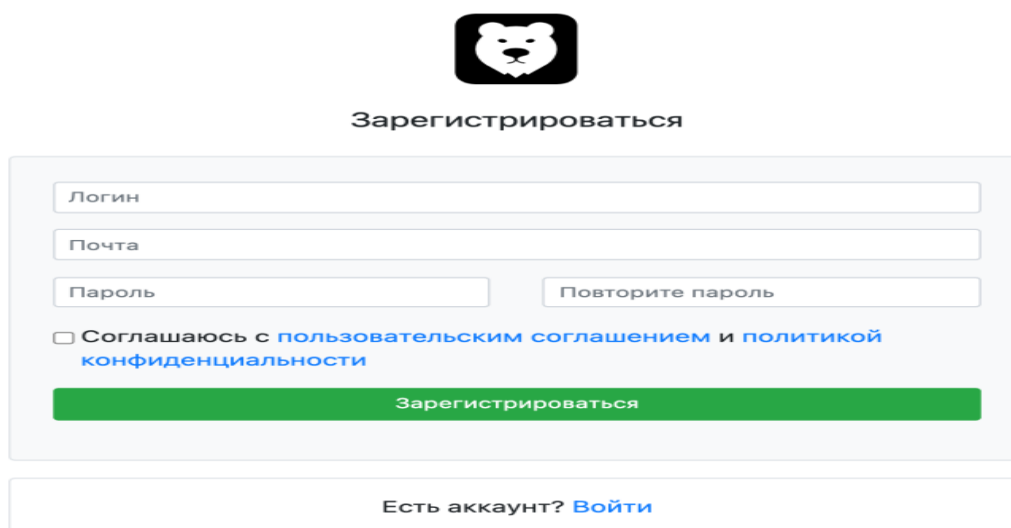
текущий пароль на новый. Для этого выполните следующие действия. Выберите пункт «Пароль» из правого вертикального меню на вкладке «Профиль». Появится форма для смены пароля. Укажите старый пароль, введите новый, повторите ввод нового пароля и нажмите кнопку «Сохранить». Изменить пароль можно также при входе в систему. Для этого нажмите кнопку «Забыли пароль». Рекомендуем не использовать одни и те же пароли на различных сайтах. Добавьте несколько чисел или ассоциацию с сайтом, это повысит надёжность.

Для завершения работы с «Профилем» нажмите кнопку «Сохранить» внизу экрана.

3.3.11. Авторизация и регистрация

Вход в систему GitFlic осуществляется при помощи почты и пароля. Также возможно использовать опцию продления сессии авторизации, для этого поставьте галочку напротив «Запомнить меня».

Если пользователь забыл пароль, который указывал при регистрации, то возможно воспользоваться функцией восстановления пароля. Для этого нажмите кнопку «Забыли пароль». В появившемся окне ввода почты укажите e-mail, который был использован при регистрации, и нажмите «Отправить письмо». В указанный почтовый ящик придёт письмо с инструкцией для создания нового пароля.



The image shows a registration form for GitFlic. At the top, there is a black square icon with a white bear's head. Below the icon, the text "Зарегистрироваться" (Register) is displayed. The form itself is a light gray box containing several input fields: "Логин" (Login), "Почта" (Email), "Пароль" (Password), and "Повторите пароль" (Repeat password). Below these fields is a checkbox labeled "Соглашаюсь с пользовательским соглашением и политикой конфиденциальности" (I agree with the user agreement and privacy policy). At the bottom of the form is a green button labeled "Зарегистрироваться". Below the registration form is a separate box containing the text "Есть аккаунт? Войти" (Have an account? Log in).

Рисунок 3.21 – Регистрация нового пользователя

Для регистрации нового пользователя GitFlic на странице регистрации (рисунок 3.21) укажите почтовый адрес и пароль (пароль создаёт пользователь). Ознакомьтесь с Пользовательским соглашением и Политикой конфиденциальности. Поставьте галочку, подтверждающую ваше согласие с документами, и нажмите «Зарегистрироваться». В указанный почтовый ящик будет отправлено письмо с подтверждением. Перейдите по вложенной в письмо ссылке для активации профиля.

3.3.12. Уведомления

В сервисе GitFlic реализована система внутренних уведомлений для пользователей. Пользователи уведомляются об изменениях в проектах, приглашениях к дискуссии и прочих действиях внутри сервиса GitFlic. На страницу с уведомлениями можно попасть, нажав на кнопку «Уведомления» в левом вертикальном меню, либо на кнопку «колокольчик» в верхнем, горизонтальном меню.

На странице «Уведомления» отображены все уведомления, которые направлены пользователю. Они расположены в хронологическом порядке от новых к старым.

Уведомлениям можно присвоить статусы:

- прочитанные;
- сохранённые.

На вкладке «Прочитанные» отображены уведомления, которые пользователь отметил как прочитанные.

Пользователь имеет возможность сохранить для себя важные уведомления, нажав на кнопку «Сохранить» напротив уведомления. Такие уведомления будут отображаться на вкладке «Сохранённые».

GitFlic позволяет управлять потоком уведомлений с помощью «Подписки» (рисунок 3.22). Меню управления подпиской находится на странице проекта.

Нажмите на кнопку «Наблюдать». Откроется меню, в котором можно выбрать нужный тип подписки:

- все уведомления (подписка «Вся активность»);

- только уведомления, в которых упоминается пользователь, устанавливающий данный тип подписки;
- отключить все уведомления, относящиеся к проекту.

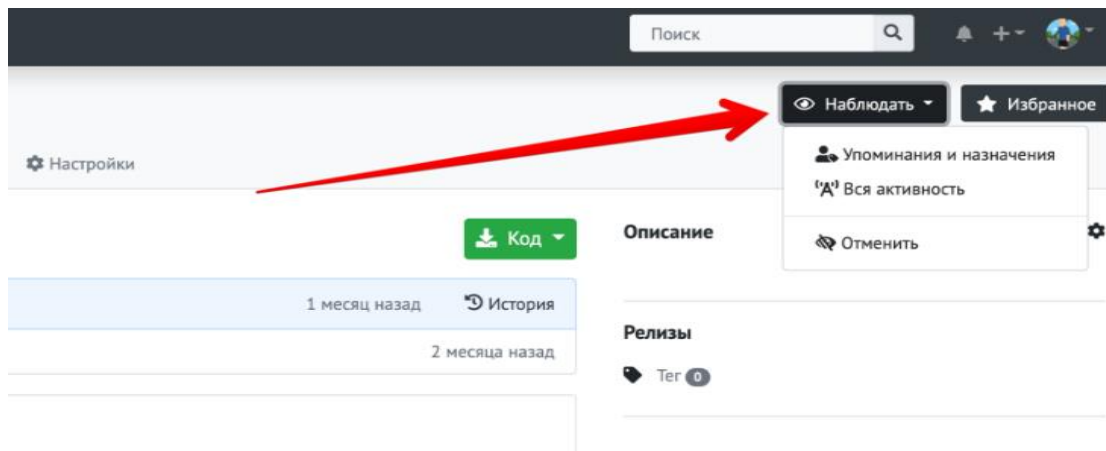


Рисунок 3.22 – Управление подписками

3.3.13. Создание README для профиля

Пользователь может поделиться информацией о себе и своём проекте, создав README для своего профиля. README будет показан в верхней части страницы профиля пользователя.

Пользователь сам решает, какую информацию включать в README, ему дан полный контроль над тем, как он будет представлен на GitFlic. Ниже приведены несколько примеров информации, которую посетители могут найти интересной, забавной или полезной в вашем профиле README.

- Раздел «Обо мне», в котором описывается ваша работа и интересы.
- Проекты, которыми вы гордитесь.

README профиля пользователя будет отображён при условии соблюдения следующих требований:

- Пользователь создал репозиторий с именем, совпадающим с именем пользователя Gitflic.
- Репозиторий публичный.
- Репозиторий содержит файл с именем README.md в корне.
- Файл README.md не пустой.

Для создания README для профиля выполните следующие действия:

- В правом верхнем углу любой страницы нажмите «+» и создайте новый проект.

- В разделе «Название проекта» введите имя проекта, соответствующее вашему имени пользователя GitFlic. Например, если ваше имя пользователя “gitflicuser”, имя проекта (репозитория) должно быть “gitflicuser”.
 - По желанию, добавьте описание вашего проекта.
 - Отметьте чекбокс «Публичный проект» и нажмите кнопку «Создать проект».
 - Выполните клонирование репозитория или создайте подключение к удалённому репозиторию.
 - Создайте в корневой папке локального репозитория файл README.md с необходимым текстом. Используйте Markdown при необходимости.
 - Добавьте README.md в индекс и создайте новый коммит. Выполните операцию push для сохранения изменений в удалённом репозитории.
- Если все шаги выполнены верно, README будет отображён в вашем профиле на вкладке «Обзор».

3.3.14. README для организации

Пользователь может поделиться информацией о том, как взаимодействовать с организацией, для которой он выполняет проект (рисунок 3.23). Для этого необходимо создать README профиля организации. Gitflic показывает README профиля вашей организации на вкладке «Обзор».

Пользователь сам выбирает, какую информацию включать в README.

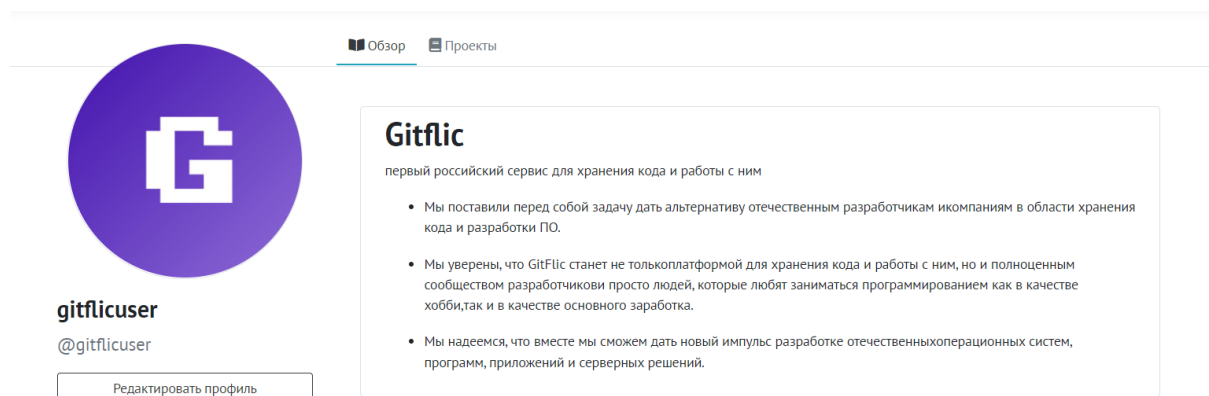


Рисунок 3.23 – Профиль организации с заполненным README

Вот несколько примеров информации, которая может оказаться полезной.

- Раздел «Об организации», описывающий вашу организацию.

– Руководство для получения помощи в организации, контакты организации.

Для создания README организации необходимо выполнить следующие действия:

– Создайте публичный проект, который будет иметь название, совпадающее с названием вашей организации в Gitflic. Подробнее о создании проектов можно прочитать в разделе 3.3.9.

– Выполните операцию клонирования или создайте удалённое подключение к существующему репозиторию.

– В корне локального репозитория создайте файл README.md и заполните его по своему усмотрению. При необходимости используйте Markdown.

– Добавьте README.md в индекс и создайте коммит. Выполните операцию push для сохранения изменений в удалённом репозитории.

Если все шаги выполнены верно, README вашей организации будет отображён на вкладке «Обзор компании».

3.3.15. Создание команды разработчиков

Команды – это группа пользователей, которые объединены общими целями и/или задачами в совместном проекте.

Для создания новой команды откройте меню создания (знак плюс в верхней части рабочей страницы GitFlic) и выберите пункт «Новая команда». Откроется форма для ввода данных о команде (рисунок 3.24). Заполните поля формы. Выберите для новой команды уникальное название и укажите приватность.

Создать новую команду

Команды позволяют управлять несколькими проектами и взаимодействовать с ними. Участники команды имеют доступ ко всем ее проектам.

Владелец: Название:

URL команды:

Описание:

☐ Публичная команда
Команда и любые публичные проекты доступны всем в интернете. Любой авторизованный пользователь может подать заявку на вступление в ряды команды.

☒ Приватная команда
Команда и проекты команды доступны только ее членам. Публичные репозитории команды доступны всем в интернете.

Рисунок 3.24 – Создание команды разработчиков

Для добавления участников в команду выполните следующие действия:

- В левом вертикальном меню выберите пункт «Команды». На открывшейся странице перейдите на вкладку «Обзор команды».

Если вы обладаете правами владельца или администратора, найдите пользователя, которого необходимо добавить (воспользуйтесь полем поиска). Укажите роль участника в команде, определяющую права доступа. После приглашения нового пользователя в команду ему на почту придёт приглашение. Пользователь должен подтвердить своё согласие на вступление в команду путём перехода по ссылке из приглашения.

Возможные роли членов команды:

- *Гость* – пользователь сможет только просматривать проекты команды, вложенные задачи и запросы на слияние. Такая роль подойдёт аудиторам.

- *Доносчик* – пользователь сможет создавать задачи по проектам команды. Такая роль подойдёт тестировщикам или менеджерам команды.

- *Разработчик* – пользователь сможет полноценно управлять проектами и выполнять команды git.

- *Администратор* – пользователь сможет редактировать настройки команды. ***Данную роль следует назначать с осторожностью.***

Удаление членов команды осуществляется на вкладке со списком участников команды.

Внимание! В сервисе существует ограничение на количество участников команды для бесплатных тарифов. Лимит составляет 5 пользователей в рамках одной команды/проекта. После превышения лимита предлагается создать компанию и перейти на платный тариф.

3.3.16. Создание компании

Компания – это обобщённая учётная запись, в которой собраны пользователи и команды, ведущие совместную работу над проектами. Администраторы и владелец (пользователь GitFlic) могут управлять доступом участников компании к проектам.

Для создания новой компании откройте меню создания (знак плюс в

верхней части рабочей страницы GitFlic) и выберите пункт «Новая компания». Откроется форма создания новой компании. Необходимо заполнить поля формы, указав базовую информацию: название, описание и тип приватности. URL компании заполняется автоматически на основании названия, но его, при желании, можно изменить.

3.3.17. Добавление сотрудников

Приглашать пользователей в компанию можно напрямую или через команды разработчиков. Для добавления пользователей перейдите на страницу «Компании», выбрав одноимённый пункт в левом вертикальном меню. Выберите компанию и пройдите на вкладку «Сотрудники компании». Выберите пользователя GitFlic через поле поиска и назначьте ему роль. Нажмите «Добавить». Новый сотрудник появится в списке сотрудников.

Для добавления команды к компании создайте новую команду (см. п. 3.3.15). Владелец команды укажите вашу компанию. После создания команды добавьте в число участников пользователей GitFlic согласно процедуре, описанной в п. 3.3.15.

Удаление сотрудников из компании или команды внутри компании доступно только пользователям с правами администратора компании. Для удаления откройте список участников компании, выбрав вкладку «Сотрудники компании» (рисунок 3.25). Напротив имени пользователя нажмите кнопку «Удалить». Если в компании существуют команды, то сотрудников можно удалять из команды. Стоит учитывать, что удалённый пользователь потеряет доступ к проектам компании, но у него могут остаться локальные копии проектов, с которыми он работал.

Сотрудник компании с правами администратора может вносить изменения в профиль компании. Для этого на странице компании выберите из дополнительного горизонтального меню пункт «Настройки». Откроется страница профиля компании, на которой доступны изменения в названии, описании, контактном телефоне, адресе электронной почты и ссылки на сайт компании. Также есть возможность изменить URL. Помните, что после смены URL старый адрес будет недоступен, также он будет свободен для всех остальных пользователей GitFlic. В дополнение, администратор компании может изменить видимость компании.

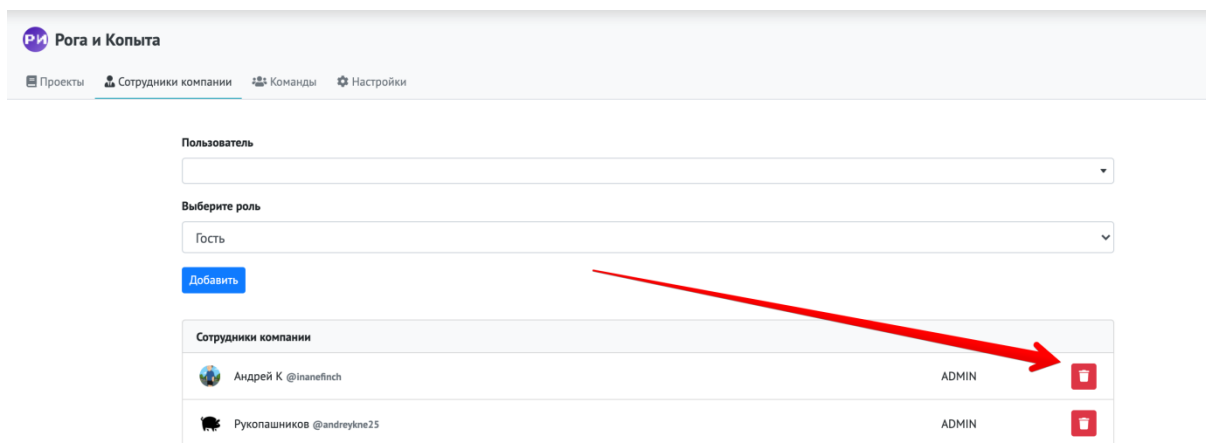


Рисунок 3.25 – Удаление сотрудника компании

Просмотр публичных компаний доступен всем авторизованным пользователям. Имеется возможность просмотреть информацию о компании, публичные проекты, список пользователей и команд.

Контрольные вопросы

1. Опишите принципы клонирования внешнего репозитория в среде программирования PyCharm.
2. Опишите особенности выполнения команды commit в среде программирования PyCharm.
3. Опишите способы клонирования репозитория в среде программирования Visual Studio
4. Опишите инструмент, предназначенный для создания и использования графической схемы репозитория в среде программирования Visual Studio.
5. Что такое учебный портал GitFlic?
6. Опишите процесс взаимодействия преподавателя и студента в ходе выполнения заданий на учебном портале GitFlic.
7. Как работает инструмент «наблюдать» в среде GitFlic?
8. Опишите процесс обработки запроса на слияние в среде GitFlic.

ЗАКЛЮЧЕНИЕ

В основу любого технологического инструмента заложена идея, позволяющая в той или иной мере облегчить труд некоторого круга профессионалов. Система управления версиями не является исключением. В её основе лежит идея систематизации хранения истории развития программного продукта. В первой главе настоящего пособия представлена историческая ретроспектива, показывающая путь, который прошли системы управления версиями, начиная с набора упорядоченных файлов и заканчивая сложными структурами данных. Последние основаны на хранении разностных текстов и систематизации связанной информации на основании деревьев и линейных упорядоченных списков. Изложенный материал позволяет проследить развитие концепций организации систем управления версиями, понять современное состояние этого инструмента и увидеть перспективы его развития.

Любой профессионал должен досконально знать свой рабочий инструмент. Не только основные приёмы его использования, но и детали внутреннего устройства. Целью второй главы является раскрытие глубинного устройства и структуры одной из наиболее популярных систем управления версиями, каковой, несомненно, является Git. Описание правил перемещения файлов программного проекта между внутренними хранилищами Git нацелено на создание у читателя целостного представления о структуре рабочих каталогов репозитория. Детальное рассмотрение операций с ветвями проекта позволяет понять концепцию одновременной отработки множества идей и решений в рамках одного проекта. Владение инструментарием создания, переключения и слияния ветвей даёт возможность управлять проектом любой сложности, не запутаться в хитросплетениях множества одновременно решаемых задач. Вернуться к проверенному решению из аварийной ситуации, опубликовать предыдущую версию без прекращения работы над новым релизом, устранить конфликты в текстах частей проекта. Всё это и многое другое доступно при глубоком понимании организации локальных и удалённых репозиториях Git.

После изучения теории необходимо переходить к практике. В третьей главе учебного пособия рассмотрены приёмы практического использования

Git. На примерах сред программирования PyCharm и Visual Studio рассмотрены приёмы организации работы с репозиторием непосредственно в среде программирования. Объединение преимуществ среды программирования и системы управления версиями делает работу программиста более продуктивной, а его взаимодействие с коллегами удобным и необременительным по времени. В результате группы программистов, разделённых географически, могут работать не менее эффективно, чем их коллеги, находящиеся в пределах одного офиса.

Вторая тема, затронутая в практической части настоящего учебного пособия, касается учебного портала GitFlic. GitFlic возник на стыке двух технологий: инструмента для организации удалённого учебного процесса и системы управления версиями Git. Учебная часть GitFlic обеспечивает создание модели учебного процесса в высшем учебном заведении. Позволяет описывать не только административную организацию факультетов и кафедр, но и структуру лекций и практических занятий. Git, в рамках учебного портала GitFlic, обеспечивает хранение истории учебных проектов, выполняемых студентами. Преподавателям предоставляется возможность ознакомиться со всеми этапами работы их учеников, указать на неточности и помочь найти правильное решение.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Прокопюк, А. Тонкости благополучного git-merge. Сентябрь 2013 г. / А. Прокопюк. – URL: <https://habr.com/ru/post/195674/> (дата обращения: 20.05.2022). – Текст: электронный.
2. Chacon, S. Pro Git / S. Chacon, B. Straub. – Apress, 2014. – 496 p.
3. GitFlic. 2022. – URL: <https://www.gitflic.ru/> (дата обращения: 25.05.2022). – Текст: электронный.
4. GNU. 28 December 2015 г. CVS – Concurrent Versions System v1.11.23. – URL: https://www.gnu.org/software/trans-coord/manual/cvs/html_node/ (дата обращения: 20.05.2022). – Текст: электронный.
5. Nguyen, T.-T. GNU RCS 5.10.1. January 2022 г. / T.-T. Nguyen. – URL: <https://www.gnu.org/software/rcs/manual/rcs.html> (дата обращения: 20.05.2022). – Текст: электронный.
6. Oracle Corporation. Chapter 5 SCCS Source Code Control System. 2010. – URL: <https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dhp/index.html> (дата обращения: 20.05.2022). – Текст: электронный.
7. Pilato, C. Michael. Version Control with Subversion / C. Michael Pilato. – 2nd Edition. – O'Reilly Media, Inc, 2008. – 407 p.
8. Schilling, J.E. An Introduction to the Source Code Control System / J.E. Schilling. – Berkeley: University of California at Berkeley, 2020.

УЧЕБНОЕ ИЗДАНИЕ

Булычев Андрей Александрович

**СИСТЕМА УПРАВЛЕНИЯ ВЕРСИЯМИ GIT И РОССИЙСКИЙ
СЕРВИС ХРАНЕНИЯ ИСХОДНОГО КОДА GITFLIC**

УЧЕБНОЕ ПОСОБИЕ

Редактор	<i>Демиденко В.К.</i>
Корректор	<i>Демиденко В.К.</i>
Компьютерная верстка	<i>Луговая С.А.</i>

Электронное издание.

Подписано в тираж 09.11.2022 г.

Печ. л. 9. Усл.-печ. л. 8,37. Уч.-изд. л. 5,39.

Объем 2,3 Мб. Тираж – 500 (первый завод – 30) экз.

Заказ №

Отпечатано в ООО «Минэлла Трэйд»,
115419, Россия, Москва, ул. Орджоникидзе, д. 9, корп. 2, пом. 5,
тел.: 8 (495) 730-41-88.

Макет подготовлен в редакционно-издательском отделе
ЧОУВО «Московский университет им. С.Ю. Витте»,
115432, Россия, Москва, 2-й Кожуховский проезд, д. 12, стр. 1,
тел.: 8(495) 783-68-48, доб. 53-53.