МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение   
высшего образования

**«Сибирский государственный университет науки и технологий   
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**ОТЧЕТ**

**по лабораторной работе № 3**

по дисциплине

*Информационные технологии в цифровой экономике*

Выполнила студентка группы БИСЗ19-01

Заочной формы обучения Тарасова К.Л.

Руководитель: ктн, доцент Буряченко В.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Красноярск 2023 г.

***Лабораторная работа №3. Знакомство с ситемой GIT.***

1. ***Цель работы*:** изучить систему управления версиями Git.

***Задачи работы*:** Используя систему управления версиями Git и программный код из ранее выполняемых лабораторных работ по программированию, продемонстрировать в отчете знания по решению следующих задач:

•создание проекта и репозитория;

•проверка состояния репозитория;

•индексация изменений;

•история версий;

•создание алиасов;

•перемещение по версиям;

•тегирование;

•клонирование репозиториев;

•отмена ошибочных изменений, коммитов;

•ветвление и разрешение конфликтов при ветвлении;

•создание чистого репозитория.

2. Ход работы

2.1 Подготовка к работе с Git

Первым шагом установим имя и адрес электронной почты. Для этого в окне Git выполним следующие команды:

gitconfig--globaluser.name "Xenia Tarasova"

git config--global user.email "taral.krsk@bk.ru"

Пример выполнения команд представлен на рисунке 2.1.

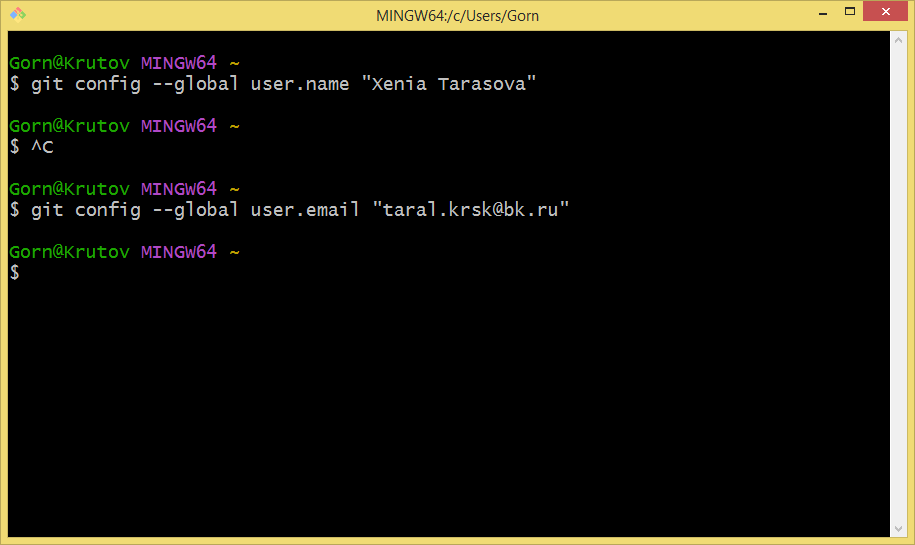


Рисунок 2.1 –Настройка имени и электронной почты.

Следующим шагом установиv параметры окончаний строк. Для этого выполним следующие команды:

git config --global core.autocrlf true

git config --global core.safecrlf true

Пример выполнения команд представлен на рисунке 2.2.

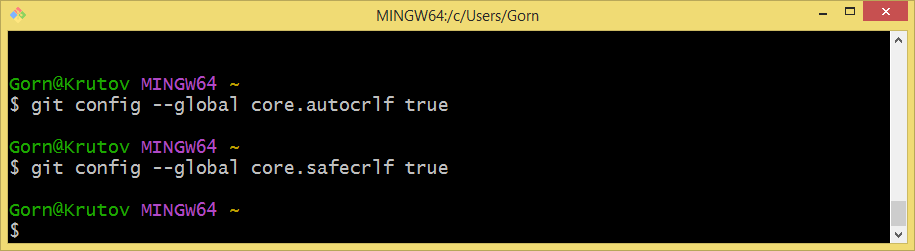


Рисунок 2.2 –Настройка параметров окончания строк.

2.2 Создание проекта и репозитория

Выберем каталог, в котором будем хранить данные об изменениях версий проекта.

Выполним следующие команды:

mkdir «your initials»

cd «your initials»

Пример выполнения команд представлен на рисунке 2.3.



Рисунок 2.3 –Создание каталога.

Создадим сам файл проекта.

Для создания файла «new\_project.cs» нужно выполнить команду:

touch new\_project.cs

Пример выполнения команды представлен на рисунке 2.4.



Рисунок 2.4 –Создание файла

Пример созданного файла представлен на рисунке 2.5.

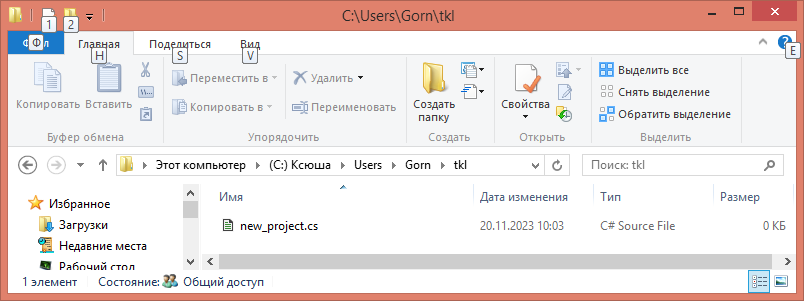
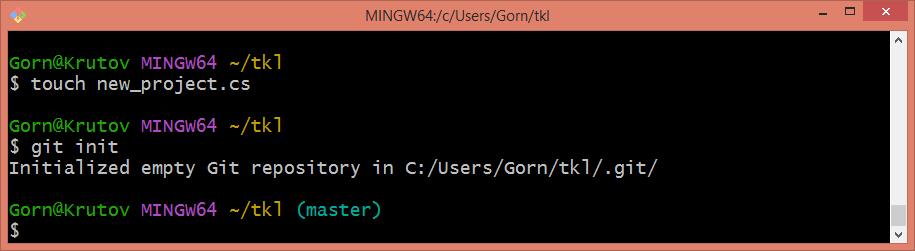


Рисунок 2.5 –Вновь созданный файл.

Создадим репозиторий, то есть хранилище всех изменений файлов в этом каталоге. Выполним следующую команду:

git init

Результат выполнения команды изображен на рисунке 2.6



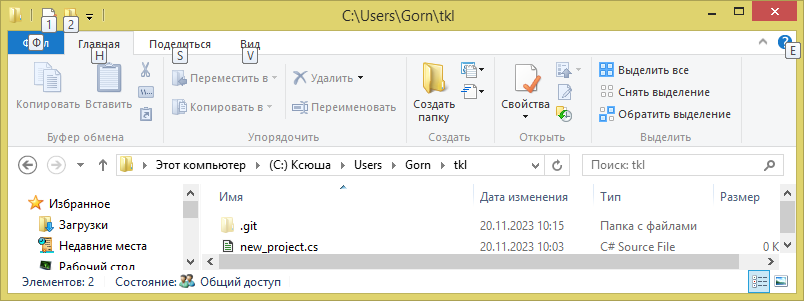


Рисунок 2.6 –Результат создания репозитория.

Чтобы система начала контроль версий файлов, необходимо добавить их в репозиторий. Добавим файл «new\_project.cs» в созданный репозиторий и добавим коммит, говорящий о том, что текущая версия файла не содержит никакого программного кода, т.е. файл пустой. Для этого нужно выполнить следующие команды:

git add new\_project.cs

git commit -m "Just empty file"

Результат выполнения команд представлен на рисунке 2.7.

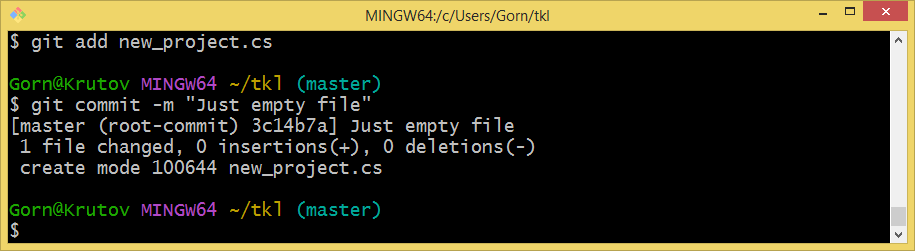


Рисунок 2.7 –Результат добавления первого коммита.

2.3 Проверка состояния репозитория

Проверим состояние репозитория, чтобы узнать, в каком он сейчас состоянии. Для этого нужно выполнить следующую команду:

git status

Результат выполнения команды представлен на рисунке 2.8.

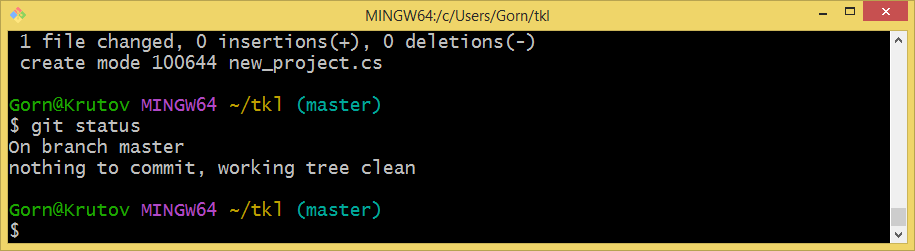


Рисунок 2.8 –Проверка состояния репозитория.

2.4 Индексация изменений.

Теперь, когда контроль версий, добавленных в репозиторий файлов осуществляется, начнем работу над проектом. После внесения изменений в файл проекта, вновь проверим состояние репозитория.

Результат такой проверки представлен на рисунке 2.9.



Рисунок 2.9 –Результат проверки состояния репозитория, после изменения исходного файла.

Чтобы добавить текущую версию в систему контроля, проиндексируем изменения и добавим коммит.

Результат представлен на рисунке 2.10.

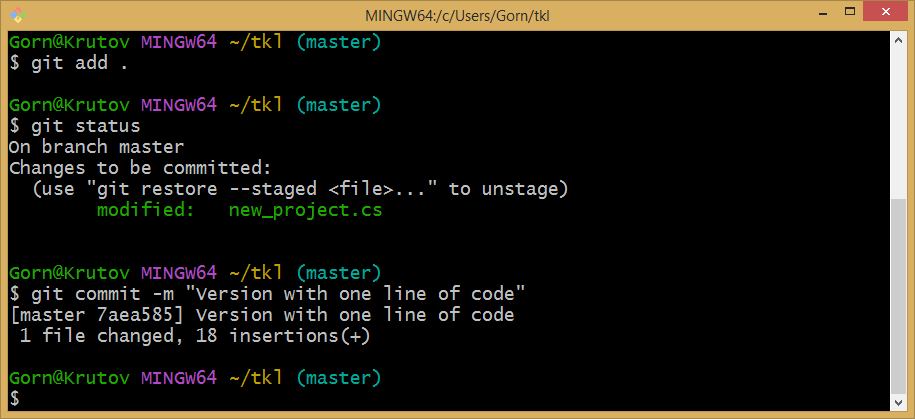


Рисунок 2.10 –Проверка состояние репозитория и добавление коммита.

Теперь, внесем изменения в файл проекта, проиндексируем изменения, вновь изменим файл и проверим состояние репозитория. Для более полного понимания того, что система контроля версий Git работает с изменениями, а не с целыми файлами, как подавляющее большинство других систем контроля версий, результат подобных операций представлен на рисунке 2.11.

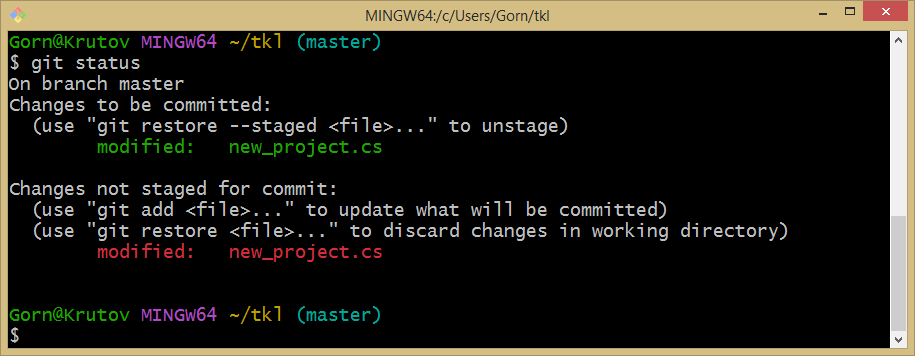


Рисунок 2.11 –Очередная проверка состояния репозитория.

Как можно видеть, система различает версии одного и того же файла, одна из которых проиндексирована и готова к добавлению коммита, а вторая не проиндексирована. Следующим шагом необходимо добавим коммит к проиндексированной версии, проиндексируем последнюю версию и добавим коммит к ней.

2.5 История версий

Для того, чтобы посмотреть всю историю версий, нужно выполнить следующую команду:

git log

Результат выполнения команды представлен на рисунке 2.12



Рисунок 2.12 –История проекта.

Очевидно, что при таком выводе истории, происходит дублирование большого количества информации. Для удобства восприятия истории, более подходит режим «одной строки». Для этого, нужно выполнить следующую команду:

git log –pretty=oneline

В этом режиме информация выводится в построчном виде. Пример изображен на рисунке 2.13.

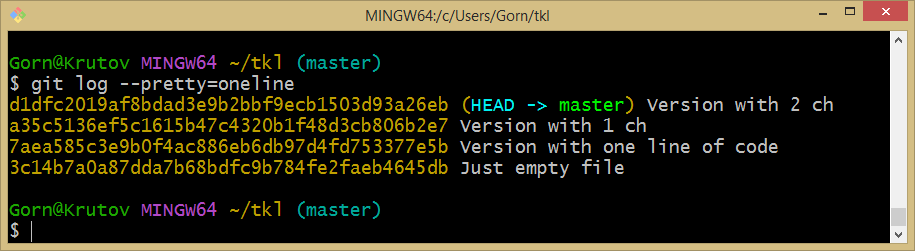


Рисунок 2.13 –История проекта в режиме «одной строки».

Для ещё более удобного вывода истории можно использовать различные способы форматирования. Пример одного из них представлен на рисунке 2.14

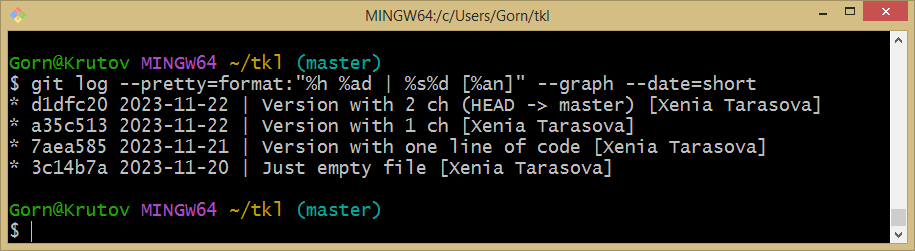


Рисунок 2.14 –Пример форматированной истории.

Очевидно, что набирать команды целиком порой достаточно утомительно, поэтому целесообразно использовать алиасы.

2.6 Алиасы.

Алиас –псевдоним/сокращение команды. Примеры использования алиасов представлены на рисунке 2.15.

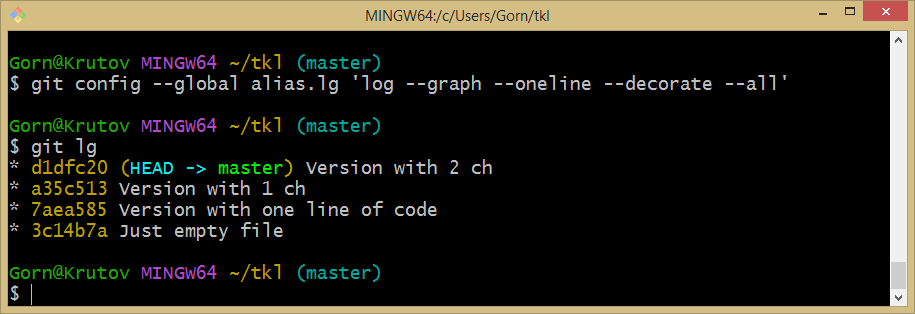


Рисунок 2.15 –Использование алиасов.

2.7 Возврат к прошлым версиям.

В том случае, если необходимо вернуться к прошлой версии кода программы, используют команду:

git checkout <hash>

Результат выполнения команды представлен на рисунке 2.16.

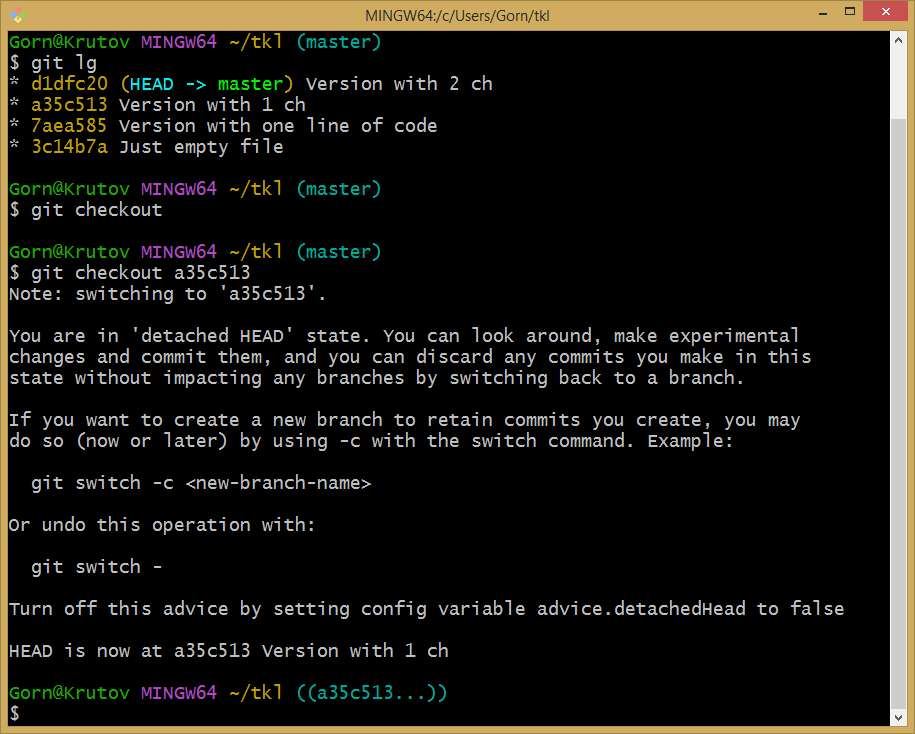


Рисунок 2.16 –Возврат к прошлым версиям.

В случае необходимости возврата к последней версии файла, достаточно дать аналогичную команду. Результат представлен на рисунке 2.17.

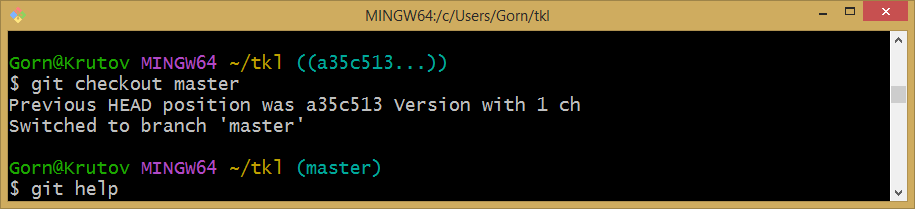


Рисунок 2.17 –Пример возврата к последней версии файла.

2.8 Тегирование версий.

В случае, когда разработка проекта ведется в несколько этапов, целесообразно использовать тегирование. Для того чтобы задать тег той или иной версии файла, необходимо использовать команду:

git tag <some\_tag>

Пример выполнения команды представлен на рисунке 2.18.

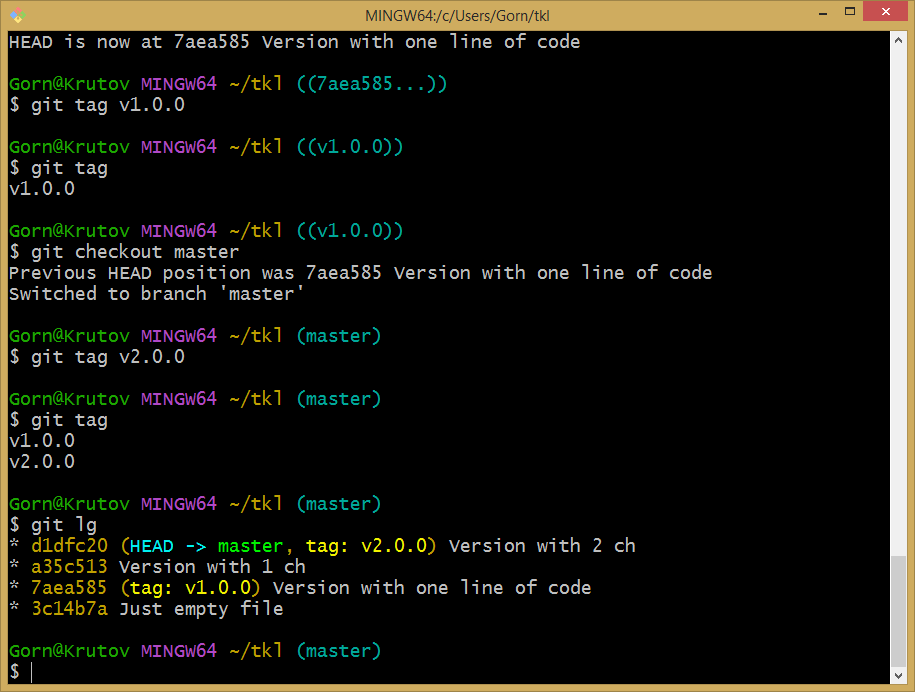


Рисунок 2.18 –Пример использования тегов в отслеживании версий.

2.9 Отмена локальных изменений.

Бывают случаи, когда в код были добавлены ненужные строки или комментарии, либо в результате попытки улучшить программу, были допущены ошибки, и необходимо вернуться на последнюю проиндексированную версию. Пример выполнения подобной операции представлен на рисунке 2.19.

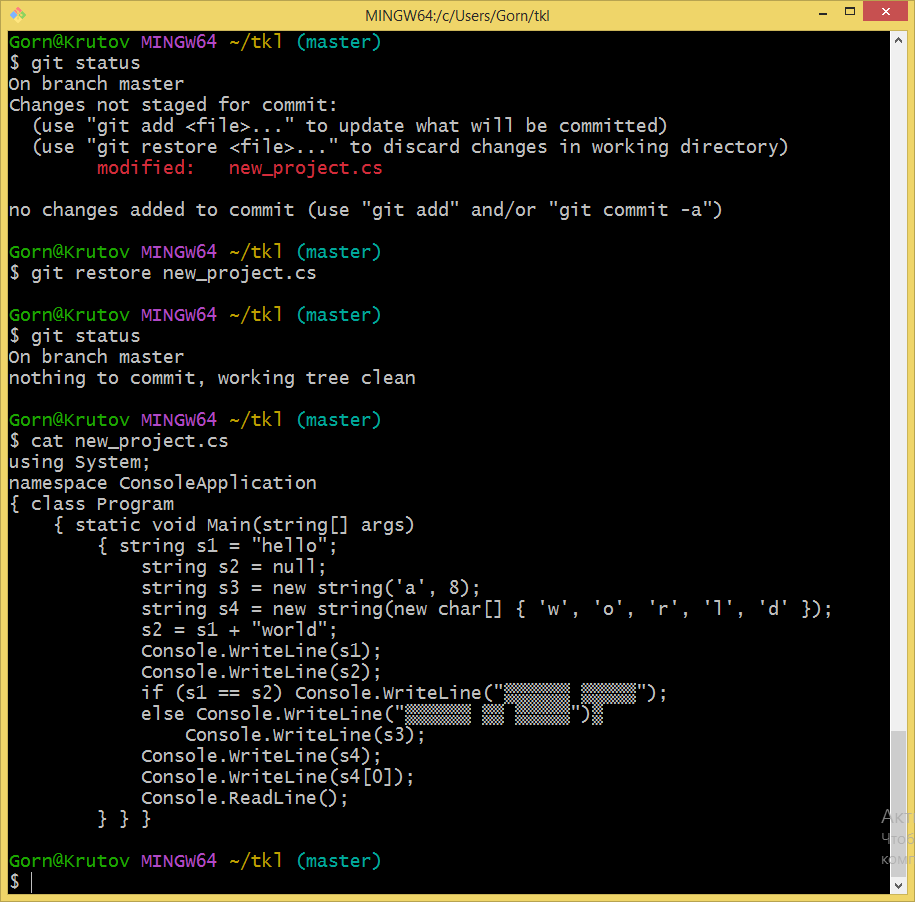


Рисунок 2.19–Пример отмены изменений.

Если же случилось так, что файл с ошибками был проиндексирован, но к нему не добавлен коммит, и его необходимо откатить, нужно выполнить команду:

git reset HEAD <file’s\_name>

Пример выполнения подобной операции приведен на рисунке 2.20.

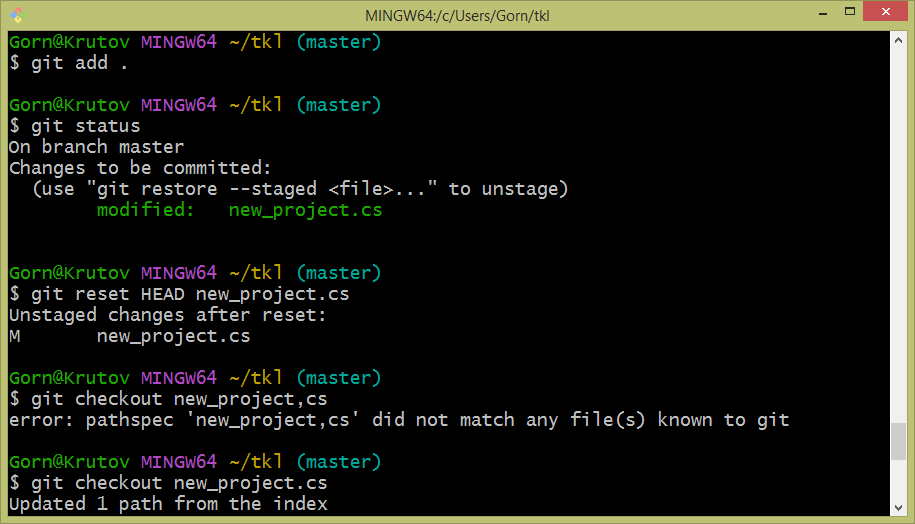


Рисунок 2.20–Пример отмены проиндексированных изменений.

Если же случилось так, что файл с ошибками был проиндексирован, и к нему добавлен коммит, и его необходимо откатить, нужно выполнить команду:

git revert HEAD

Пример выполнения команды представлен на рисунке 2.21

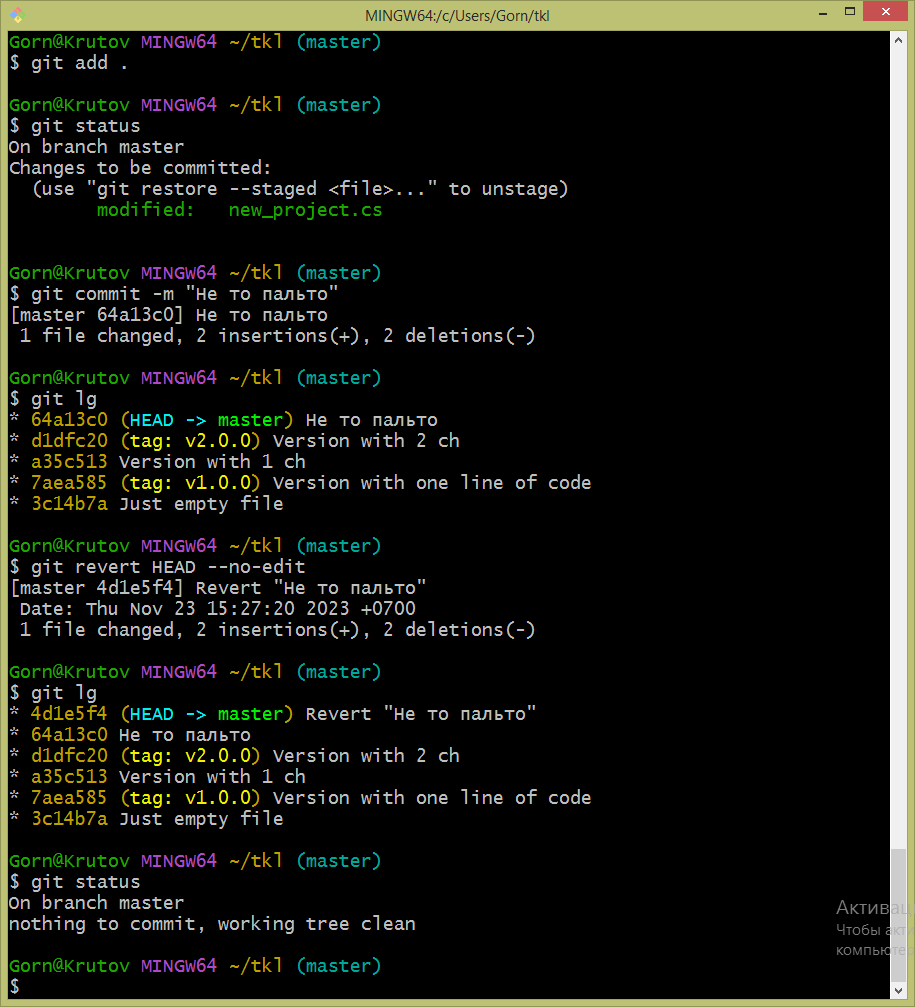


Рисунок 2.21–Пример отмены неправильного коммита.

Как видно на предыдущем рисунке, отмененный коммит остался в истории. Для его отмены для начала необходимо протегировать текущую версию файла. После этого выполнить сброс к предыдущей версии. Пример выполнения операций приведены на рисунке 2.22.

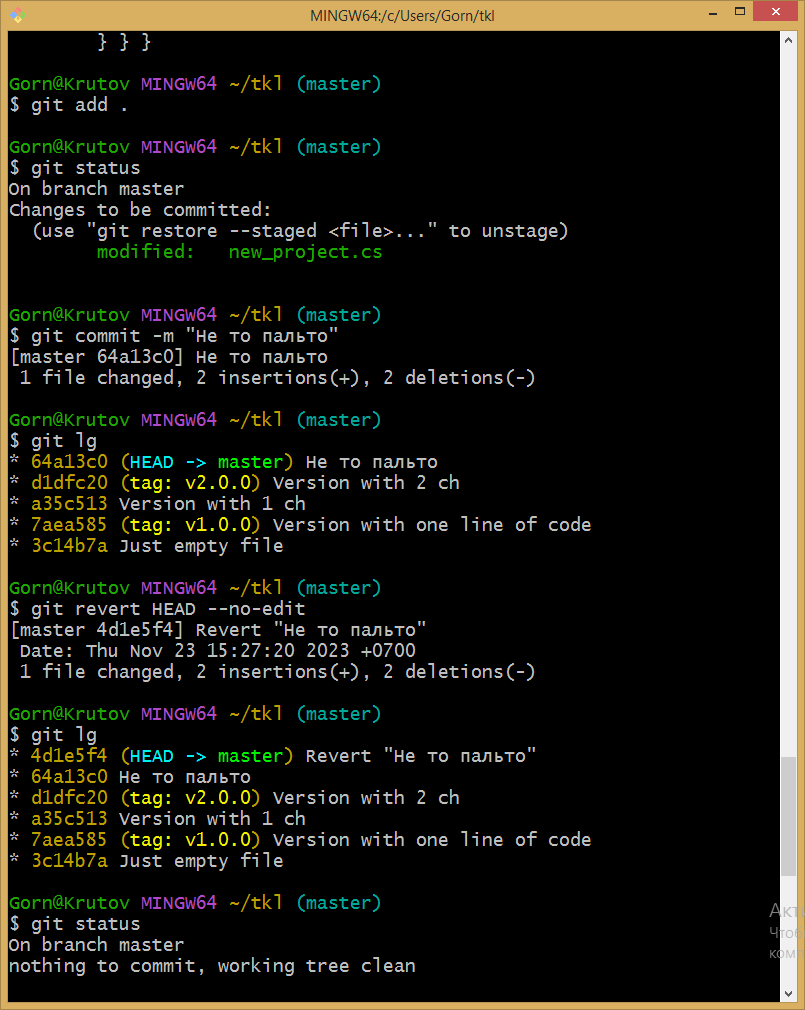


Рисунок 2.22–Пример удаления.

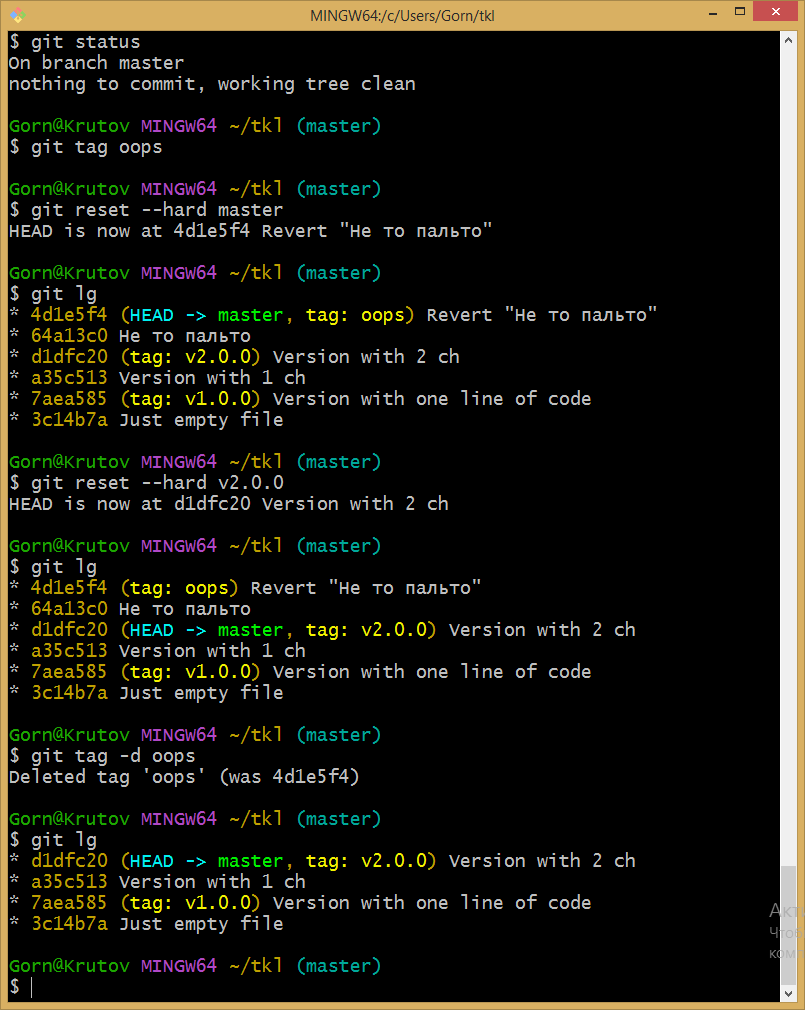


Рисунок 2.22(1)–Пример удаления.

2.10 Создание ветки.

Одним из самых важных и удобных инструментов системы контроля версий является ветвление. Таким образом, один проект может разветвиться на несколько отдельно развивающихся ветвей, преследующих разные цели, оперирующие разными версиями файлов и так далее. Кроме того, в ходе разработки ветвей, можно устроить их слияние. Пример создание ветки представлен на рисунке 2.24.

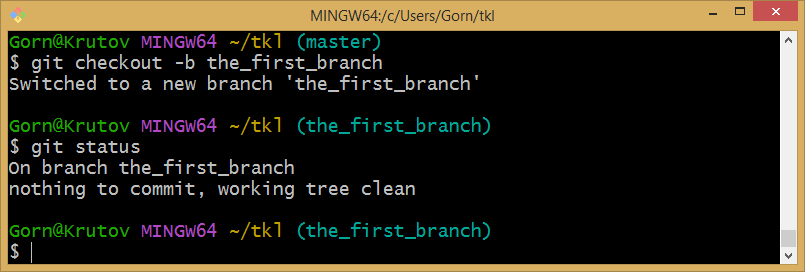


Рисунок 2.24–Создание ветки the\_first\_branch.

Пример работы с ветками изображен на рисунках 2.25, 2.26.

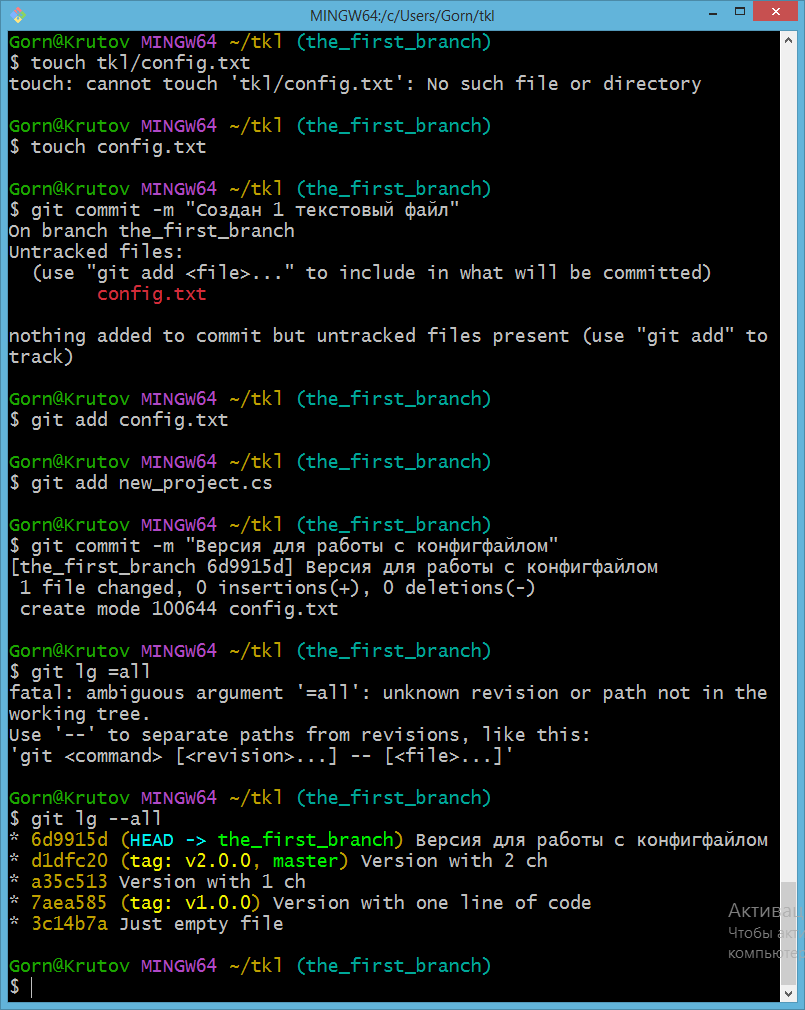


Рисунок 2.25–Работа с ветками.

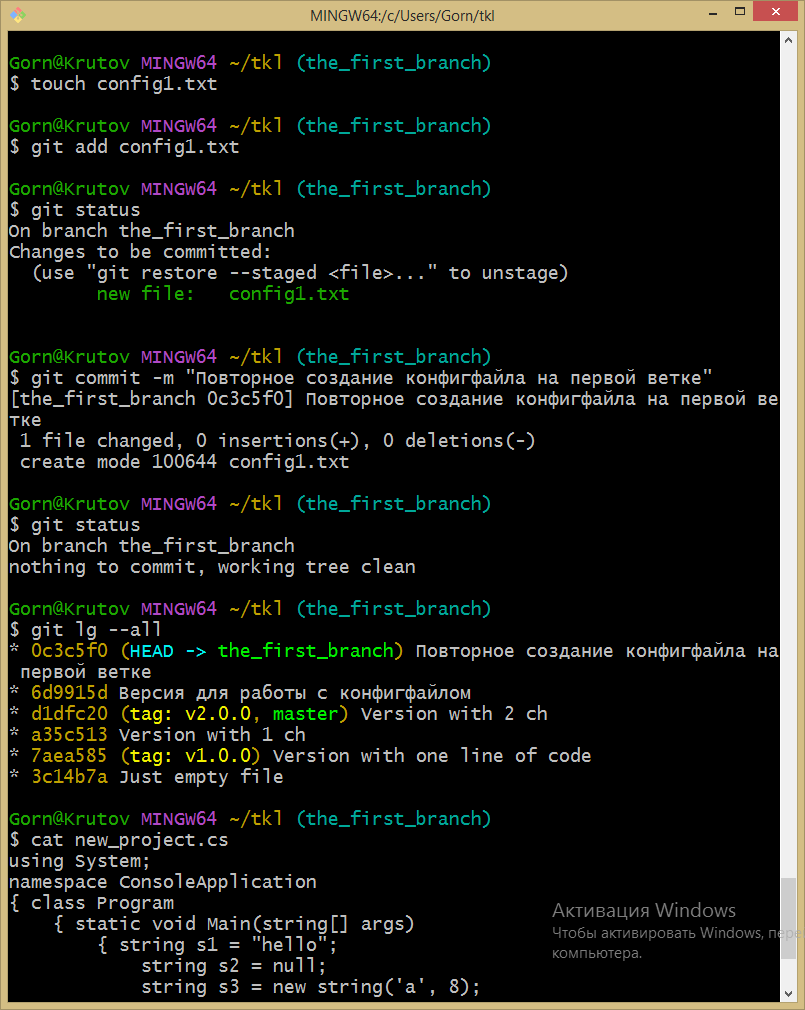
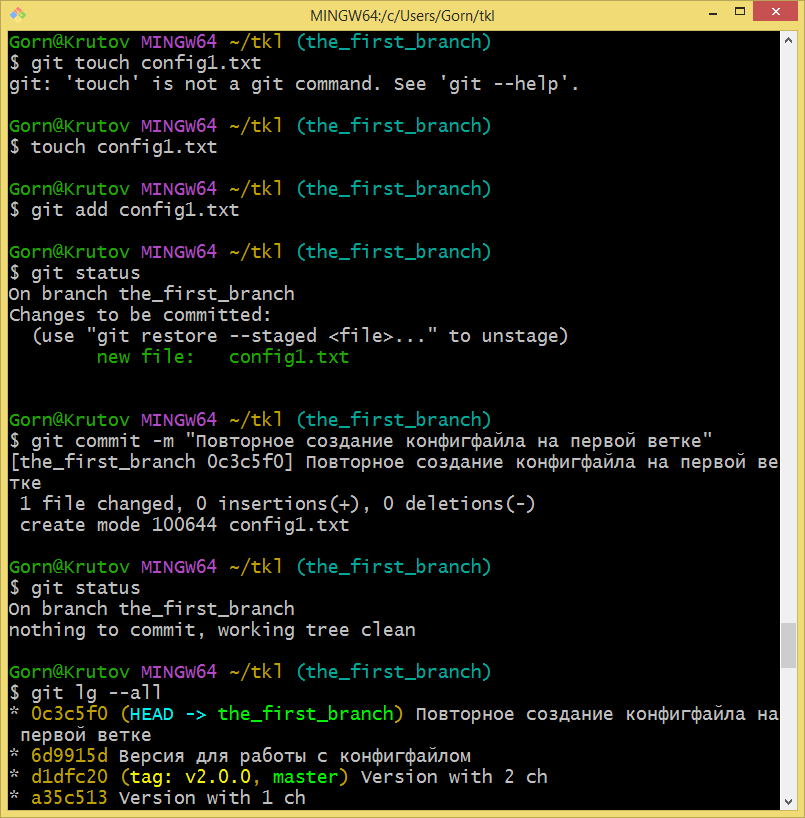
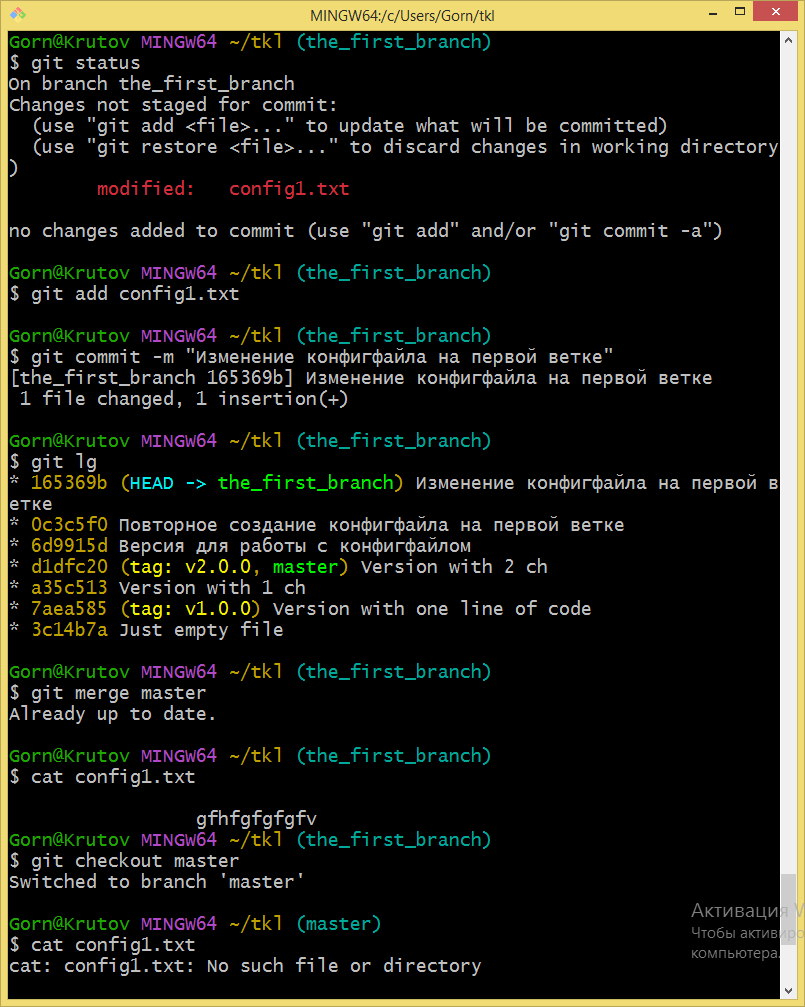


Рисунок 2.26–Работа с ветками.

2.11 Слияние веток

Пример слияния веток изображен на рисунке 2.27.





2.12 Конфликт

Для рассмотрения механизма разрешения конфликтов, необходимо создать конфликт. Для этого необходимо, чтобы у одного файла оказалось разное содержимое в разных ветках. Пример создания конфликта изображён на рисунке 2.28.

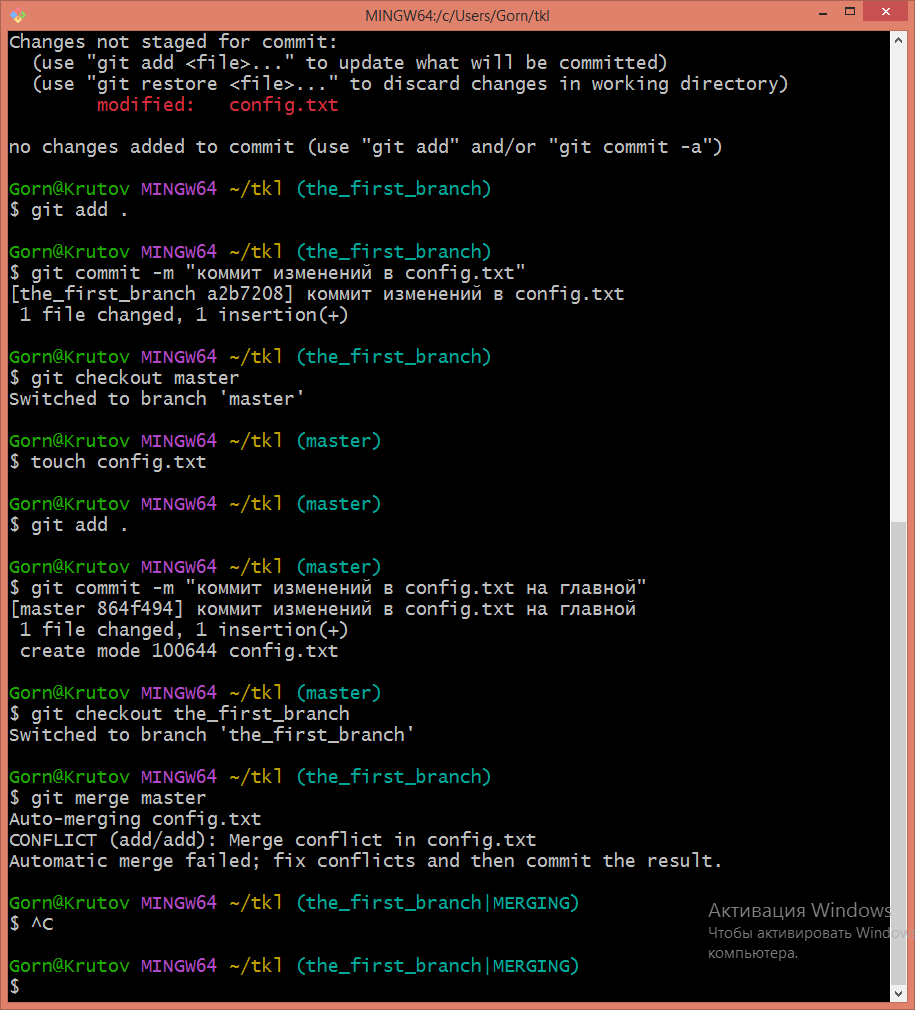


Рисунок 2.28–Конфликт веток.

Кроме того, Git делает необходимые подсказки в самих файлах. Пример этого представлен на рисунке 2.29.

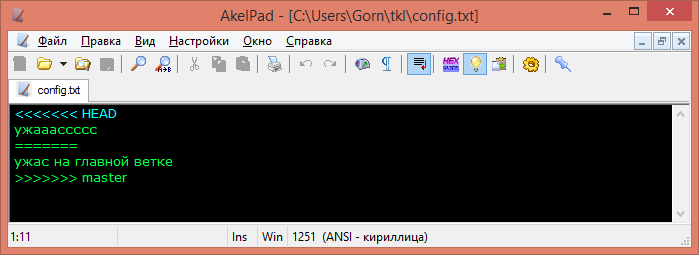


Рисунок 2.29–Комментарии Git относительно места конфликта.

2.13 Разрешение конфликтов

Исправив конфликт вручную, в файле, то есть, привести файл к версии одной из веток, необходимо проиндексировать и добавить коммит во второй ветке. Пример разрешения конфликта изображен на рисунке 2.30.

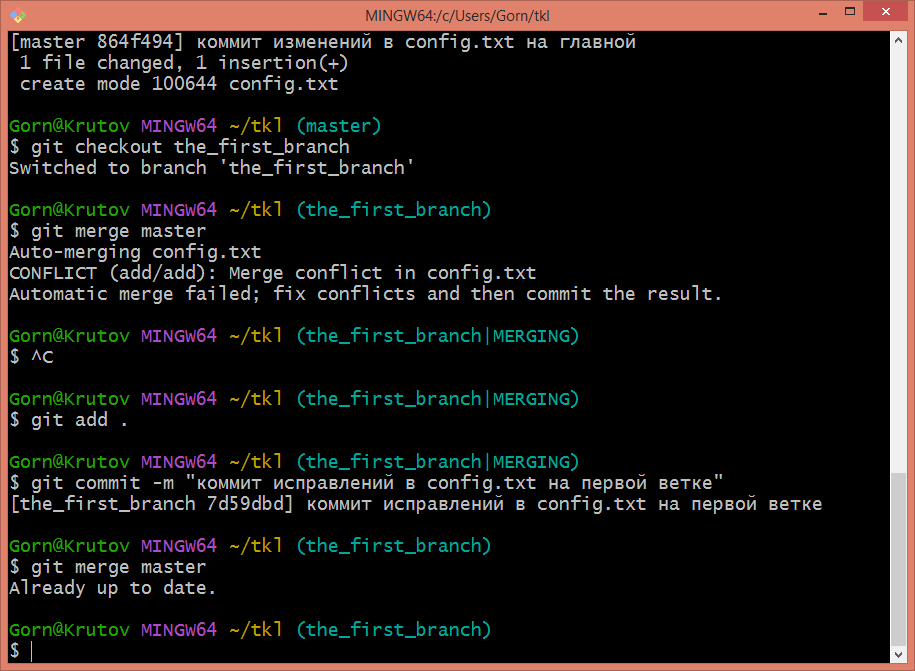
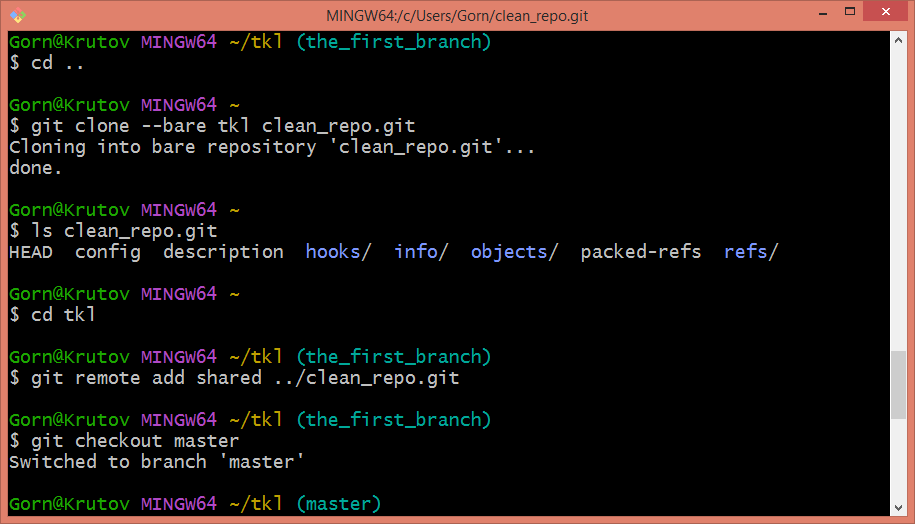


Рисунок 2.30–Разрешенный конфликт.

2.14 Создание чистого репозитория.

Обычный git-репозиторий подразумевает, что вы будете использоватьего как рабочую директорию, поэтому вместе с файлами проекта в актуальной версии, git хранит все служебные, «чисто-репозиториевские» файлы в поддиректории .git. В удаленных репозиториях нет смысла хранить рабочие файлы на диске (как это делается в рабочих копиях), а все что им действительно нужно –это дельты изменений и другие бинарные данные репозитория. Вот это и есть «чистый репозиторий». Пример создания чистого репозитория и работы с ним изображен на рисунках 2.31и 2.32.



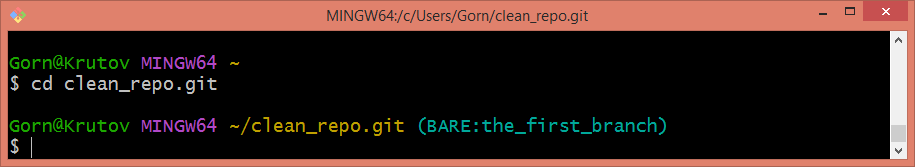


Рисунок 2.31–Создание репозитория clean\_repo.

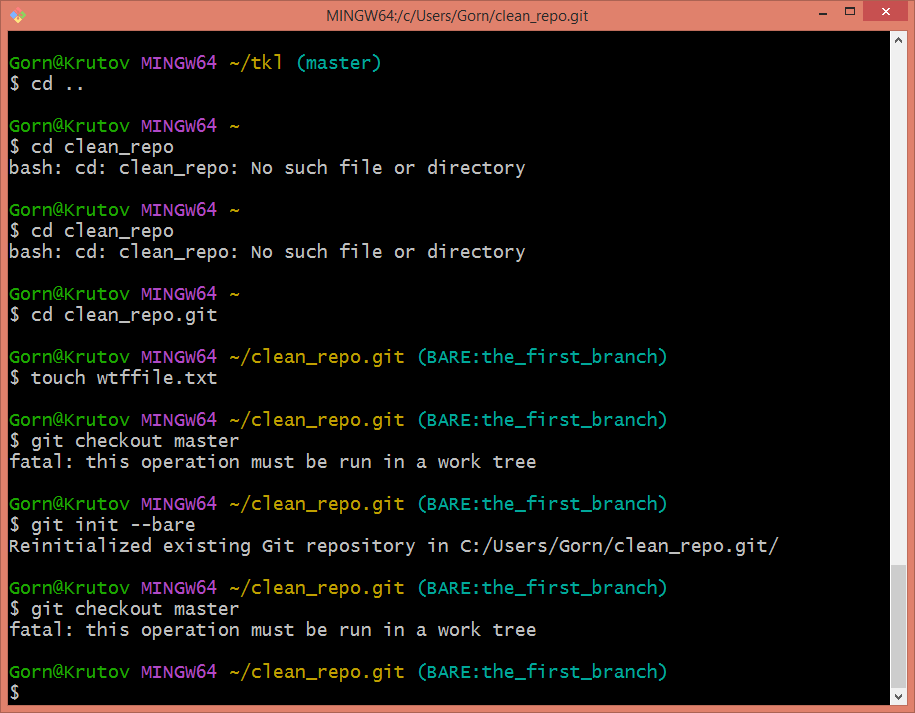


Рисунок 2.32–Работа с удаленным репозиторием.

3.Контрольные вопросы.

1.Что представляет собой система GIT?

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

2.Перечислите возможности систем управления версиями.

Сохранение исходного кода. Информация сваливается на удаленный сервер и в репозитории остаются даже файлы, удаленные с компьютера разработчика.

Возможность привлекать группу программистов, не покупая отдельно специальные инструменты для командной работы. Каждый выполняет свою задачу на персональном компьютере, обновляя файлы, когда это нужно.

Отмена внесенных изменений. Всегда есть возможность вернуться к контрольной точке, провести ревью исходного кода и текущего, а затем обновить основную ветку.

Распределенная работа над проектом. То есть, программисты могут создавать видоизмененный плагин, пока основная его версия спокойно функционирует на сайте.

3.Что такое коммит и когда он выполняется?

В системах управления версиями коммит добавляет последние изменения в [часть] исходного кода в хранилище, делая эти изменения частью основной версии хранилища.

4.Как посмотреть историю коммитов?

Для просмотра истории предыдущих коммитов в обратном хронологическом порядке можно использовать команду git log. Ей можно передать разные опции:

-p показывает изменения в каждом коммите;

--stat показывает сокращённую статистику для коммитов, например изменённые файлы и количество добавленных/удалённых строк в каждом их них;

-n показывает n последних коммитов;

--since=\_\_\_ и --until=\_\_\_ позволяет отфильтровать коммиты по промежутку времени, например --since="2019-01-01" покажет коммиты с 1 января 2019 года;

--pretty позволяет указать формат логов (например, --pretty=oneline), также можно использовать --pretty=format для большей кастомизации, например --pretty=format:"%h %s";

--grep и -S фильтруют коммиты с сообщениями/изменениями кода, которые содержат указанную строку, например, git log -S имя\_функции позволяет посмотреть добавление/удаление функции;

--no-merges пропускает коммиты со слиянием веток;

.ветка2 позволяет посмотреть, какие коммиты из ветки 2 не находятся в ветке 1 (полезно при слиянии веток). Например, git log master..test покажет, каких коммитов из ветки test нет в master (о ветках поговорим чуть позже).

--left-right ветка1...ветка2 показывает коммиты, которые есть либо в ветке 1, либо в ветке 2, но не в обеих; знак < обозначает коммиты из ветка1, а > — из ветка2. Обратите внимание: используется три точки, а не две;

-L принимает аргумент начало,конец:файл или :функция:файл и показывает историю изменений переданного набора строк или функции в файле.

5.Как создать пустой GIT репозиторий?

Если у вас уже есть проект в директории, которая не находится под версионным контролем Git, то для начала нужно перейти в эту директорию. Если вы не делали этого раньше, то для разных операционных систем это выглядит по-разному:

для Windows:

$ cd C:/Users/user/my\_project

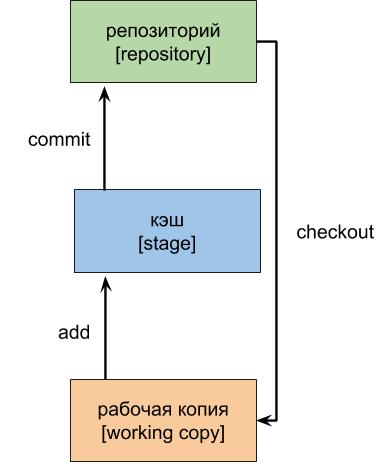
а затем выполните команду:

git init

Эта команда создаёт в текущей директории новую поддиректорию с именем .git, содержащую все необходимые файлы репозитория — структуру Git-репозитория.

6.Какую архитектуру имеет система контроля версий GIT?

Система контроля версий *git*использует архитектуру трех деревьев. Схематично она выглядит так как показано на рисунке ниже.



Суть ее заключается в том, что дополнительно добавляется ещё одно место, которое можно назвать кэшем или *stage*в английской терминологии. Рабочая копия и репозиторий идейно не отличается от их аналогов в архитектуре двух деревьев. Наличие дополнительного элемента меняет регламент работы, которой в этом случае выглядит так:

* Перед началом работы разработчик делает *checkout*, для того чтобы быть уверенным, что он будет работать с актуальной рабочей копией.
* Разработчик вносит необходимые изменения в исходный код.
* Разработчик отправляет необходимый набор файлов, изменения в которые внесены, в *stage*, для того, чтобы потом построить из них коммит. До того, как изменения будут отправлены в репозиторий, разработчик может добавлять и удалять файлы из *stage*. Набор файлов в *stage*, как правило, идеологически связан между собой.
* Разработчик отправляет изменения в репозиторий (коммитит их).
* Повторить необходимое количество раз пункты 2 – 4.
* Наличие *stage*добавляет гибкости в процесс разработки, вы можете внести изменения в довольно большое количество файлов, но отправить их в репозиторий в разных коммитах со своими специфическими комментариями.

7.Как создать новую ветку в GIT и соединить ветки?

Для создания новой ветки используется команда

git branch имя\_ветки.

Для просмотра имеющихся в репозитории веток нужно ввести эту команду без аргументов:

git branch.

Эта команда только создает ветку, но не изменяет указатель на текущую ветку HEAD. Для того, чтобы сделать ветку текущей (переместить HEAD) нужно выполнить команду

git checkout имя\_ветки.

Соединение двух ветвей на github.

Переключитесь на ветку, в которую собираетесь делать слияние: $ git checkout master.

И слейте с той веткой, из которой собираетесь делать слияние: $ git merge develop.

Для этого **выполните команду git merge <название ветки>, где <название ветки> — название ветки, которая будет объединена с принимающей**.

8.Поясните понятие «staging area».

Область подготовленных файлов (**staging area**) — это обычный файл, обычно хранящийся в каталоге Git, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов. Область подготовленных файлов это уже не рабочий каталог, но ещё и не коммит.

9.Как сравнить версии файла в GIT?

**Сравнение с последним коммитом**

Для вывода изменений в файлах по сравнению с последним коммитом, используется git diff без параметров:

git diff

Команда выводит изменения в файлах, которые еще не были добавлены в индекс. Сравнение происходит с последним коммитом.

**Сравнение с последним коммитом, включая файлы в индексе**

Если вы изменили какие-нибудь файлы в вашем рабочем каталоге и добавили один или несколько из них в индекс (с помощью git add), то команда git diff не покажет изменения в этих файлах. Чтобы показать изменения в файлах, включая файлы, добавленные в индекс, используется ключ --cached:

git diff --cached

**Сравнение коммитов**

Команда git diff позволяет сравнивать два различных коммита. Сначала нужно определить хеш (ID) коммитов, которые требуется сравнивать. Можно воспользоваться командой git log, чтобы вывести список коммитов и их идентификаторы:

git log --oneline

Теперь сравним два коммита. Для этого в качестве первого аргумента команде git diff указывается хеш первого коммита, а вторым аргументом хеш второго коммита, например:

git diff 4612297 5e356cf

**Сравнение двух веток**

Для вывода всех изменений между концами двух веток, необходимо для git diff указать имена веток:

git diff branch1 branch2

**Сравнение файлов между двумя ветками**

Чтобы сравнить конкретные файлы между двумя ветками используется команда:

git diff branch1 branch2 ./myfile.cpp

Вместо *branch1*, *branch2* нужно указать название веток, а вместо *myfile.cpp* путь до сравниваемого файла. Через пробел можно добавить еще файлы для сравнения.

**Исключить некоторые файлы из сравнения**

Иногда нужно выполнить **git diff**, но исключить один или несколько файлов, чтобы команда **git diff** их проигнорировала. Для этого используется запись вида ’:(exclude)имя\_файла’ или короткая запись ’:!имя\_файла’

Пример:

git diff -- . ':(exclude)file1.abc' ':(exclude)file2.abc'

Или более короткая запись:

git diff -- . ':!file1.abc' ':!file2.abc'

**Сравнивать только изменения в файлах (игнорировать новые и удаленные файлы)**

Чтобы исключить из сравнения новые и удаленные файлы, можно воспользоваться опцией --diff-filter, которая позволяет выбирать какие именно изменения (файлы) нужно сравнивать.

Чтобы выполнить сравнение только изменений внутри файлов (игнорируя новые, удаленные, переименованные файлы) используется ключ Modified (M) — --diff-filter=M:

git diff --diff-filter=M

4. Заключение

В ходе выполнения лабораторной работы были изучены основы системы управления версиями Git.