## 1.2 Общая характеристика курса «Контроль версий с помощью Git»

В ходе учебной практики нами был изучен курс от EPAM “[Version Control with Git](https://elearn.epam.com/learning/course/course-v1:EPAM+VCG+RU/block-v1:EPAM+VCG+RU+type@sequential+block@ab5e12cb97ca4c3983af26e9bd1f715c/block-v1:EPAM+VCG+RU+type@vertical+block@11c00f5c20d54ad29e1a78da13289d6c)”

## 1.2.1 Основы системы контроля версий

Зачастую при совместной разработке проекта в какой бы то ни было сфере целой командой специалистов исключительно важными становятся возможности вносить правки в содержимое проекта одновременно, синхронизировать действия разработчиков, а также сохранять полную историю изменений. Облачные хранилища годятся в этом аспекте для создания текстового документа, презентации, таблицы Excel, но совершенно не подходят для работы с объёмными проектами, например с файлами языков программирования, так как выбирать, какие изменения от какого автора нужно записать, а какие – удалить, придётся каждый раз вручную. В такой ситуации на помощь разработчикам приходят системы контроля версий.

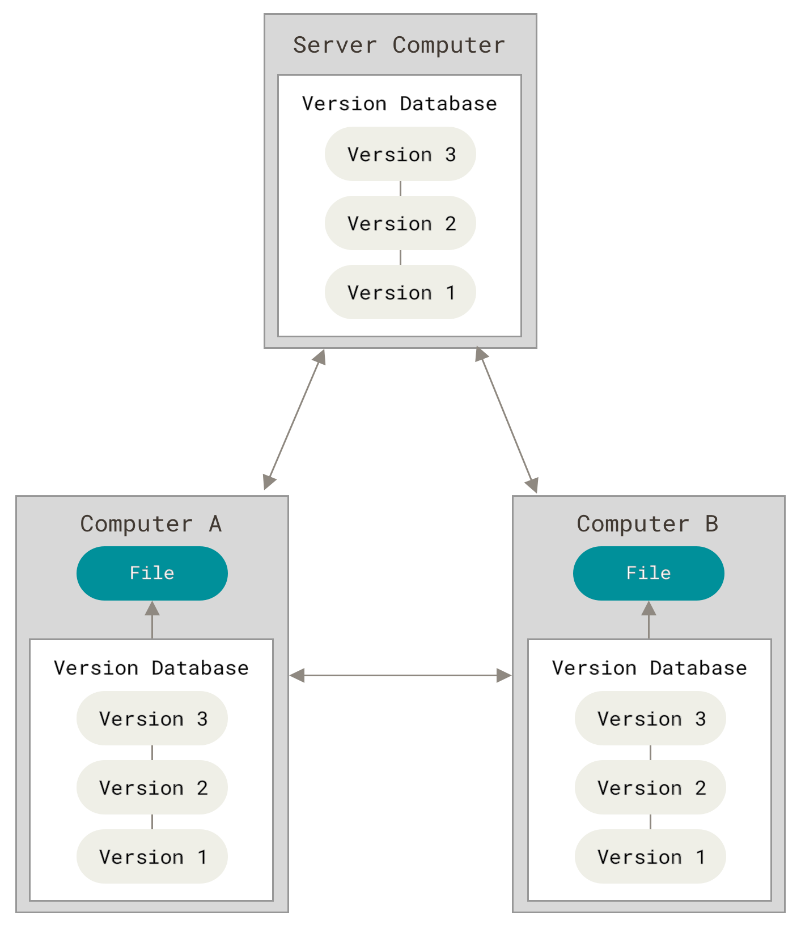
Итак, **системы контроля версий** (англ. Version Control System, VCS)— это программные инструменты, помогающие командам разработчиков управлять изменениями в исходном коде с течением времени.

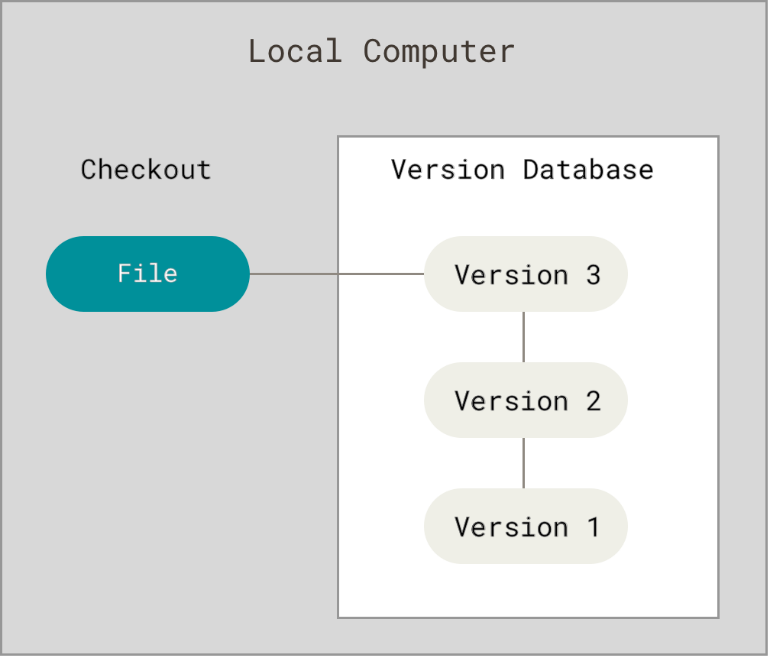
Хорошая VCS должна решать следующие задачи:

* Создание резервной копии и восстановление данных из неё
* Синхронизация работы нескольких разработчиков
* Возможность отмены изменений
* Отслеживание всех изменений и авторов изменений
* Branching, т.е. отклонение от основного кода и последующая разработка проекта независимо от него
* Sandboxing, т.е. возможность создания изолированных систем для запуска и тестирования непроверенных изменений в коде без вреда для всего проекта

Многочисленные системы контроля версий принято разделять на две большие группы в зависимости от механизма, в соответствии с которым системы работают с файлами: **lock-modify-unlock** (человек, вносящий изменения в содержимое проекта, блокирует редактирование для остальных участников и снимает блокировку после окончания правки) и **copy-modify-merge** (все участники могут работать параллельно; если разработчик хочет отправить на сервер новую версию файла, не совпадающую с актуальной на сервере, то возникает конфликт версий, который он должен разрешить, определив верное содержимое файла) стратегии.

Также системы разделяют на 2 группы по принципу взаимодействия с сервером: **централизованные** (существует единый сервер с записями о всех версиях, а у локальных участников есть только файлы проекта; участники «подтягивают» актуальную версию проекта с единого сервера и напрямую на него же отправляют свои файлы, а сервер сам обеспечивает упорядочивание и хранение данных) и **распределённые** (кроме общего удалённого сервера (репозитория), у каждого пользователя на компьютере размещён свой локальный; в этом случае участники не просто скачивают снимок всех файлов, а делают резервную копию на свой локальный сервер). Преимуществом централизованной системы является простота организации, распределённой – возможность скопировать любой репозиторий на новый сервер для продолжения работы при выходе из строя одного из задействованных в обмене данными серверов и возможность работать с несколькими удалёнными репозиториями одновременно.





*Схемы работы централизованной и локальной VCS*[*[1]*](https://git-scm.com/book/ru/v2/%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%9E-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B5-%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F-%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9)

Наиболее популярной на сегодняшний день системой контроля версий является программа **Git** – распределённая система с открытым кодом, построенная на command line интерфейсе и использующая copy-modify-merge стратегию работы с файлами.

Популярность Git обусловлена её заметным превосходством в быстродействии над другими VCS, простым созданием веток и работой с ними, гарантией сохранности целостности данных благодаря оригинальной системе хранения: Git делает «снимок» (snapshot) всех отслеживаемых файлов в конкретный момент времени, подвергая контент каждого файла односторонней хэш-функции SHA-1. SHA-1 ставит в соответствие содержимому файла последовательность из 40 символов, обозначающих число в шестнадцатиричной системе счисления.

## 1.2.2 Настройка Git

Для работы с системой контроля версий Git, необходимо скачать его с официального сайта git-scm.com и установить на свою локальную машину.

Для работы с Git необходимо найти и запустить приложение Git Bash, позволяющее работать с Git через command line интерфейс. При первом запуске выполняется базовая настройка (конфигурация):

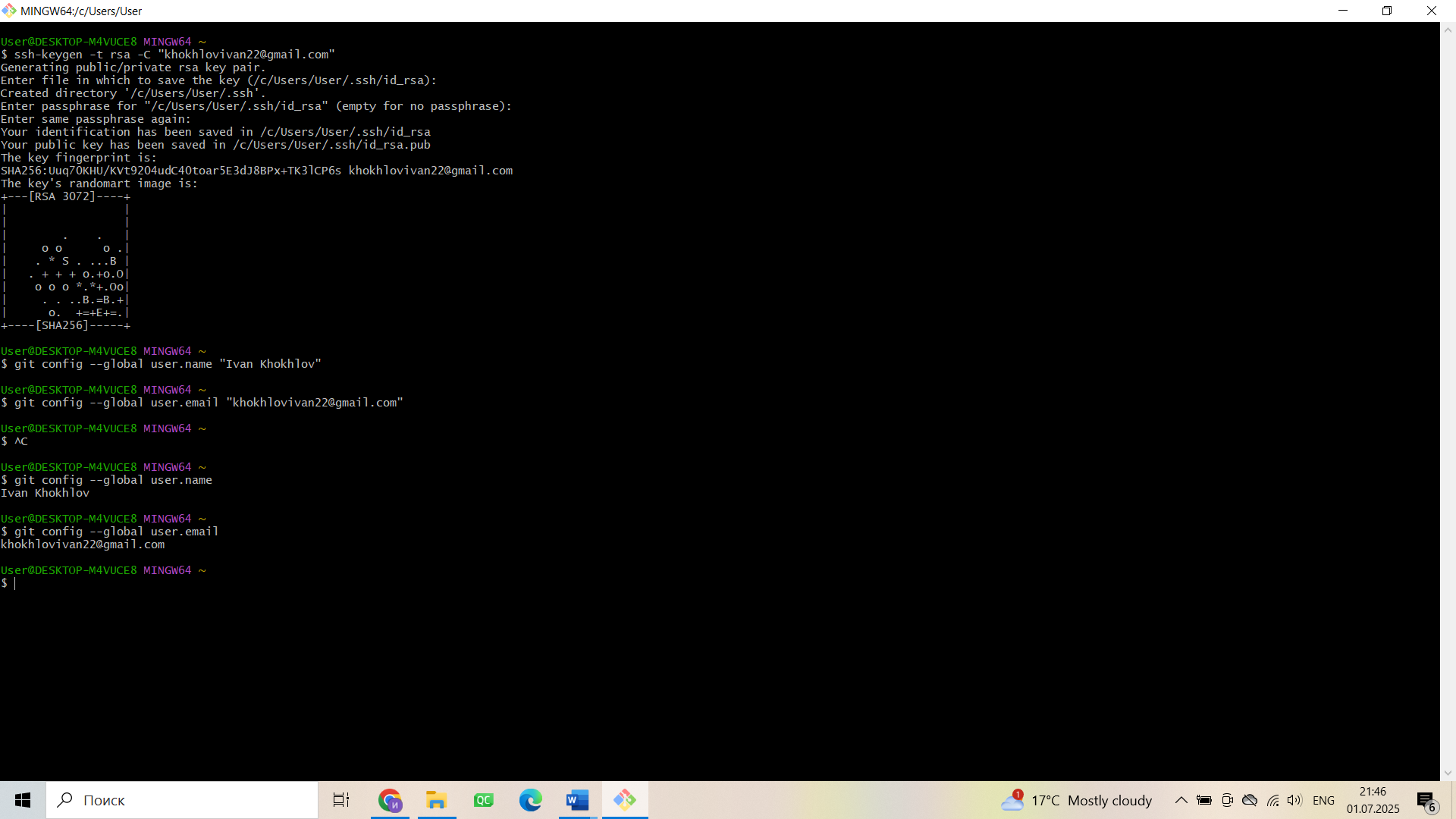
* Используем для установки имени и адреса электронной почты пользователя команды:

*git config --global user.name "Your Name"*

*git config --global user.email "your.email@example.com”*

* Генерируем публичный и приватный SSH-ключи, используемые при подключении к удалённым серверам и репозиториям, используя команду:

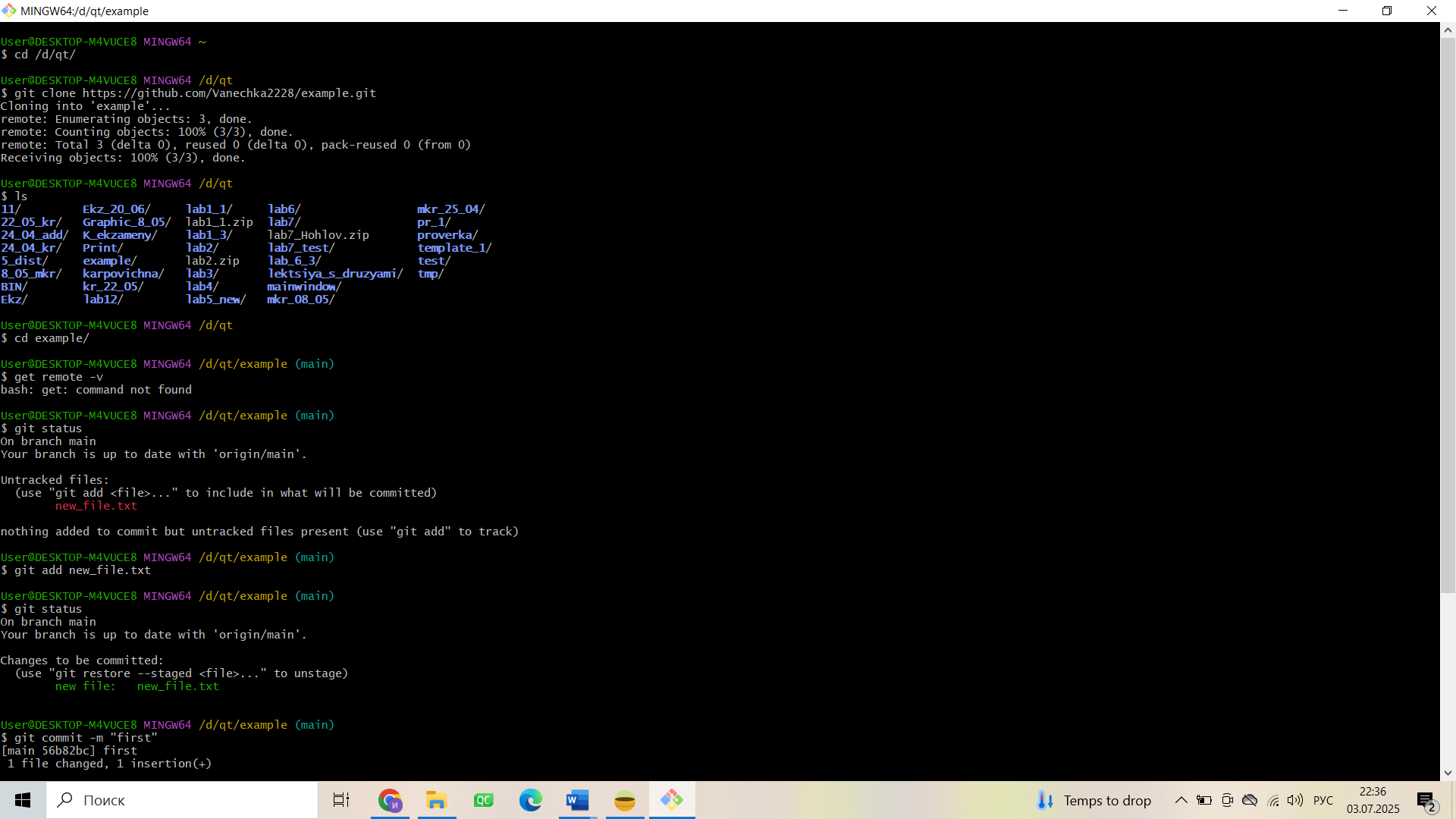
*ssh-keygen -t rsa -C “your.email@example.com”*



## Базовые операции

Для создания удалённых репозиториев в ходе учебной практики использовался GitHub –облачная платформа для размещения и управления Git-репозиториями, обеспечивающая хранение и контроль версий кода с помощью Git, совместную разработку и обзор кода, управление задачами. Связь Git с GitHub осуществляется с помощью SSH- и HTTPS-ключей, генерирующихся на GitHub при создании нового удалённого репозитория. Для связи удалённого репозитория с локальной машиной используют команду:

***git clone key***, где key – SSH- или HTTPS-ключ удалённого репозитория.

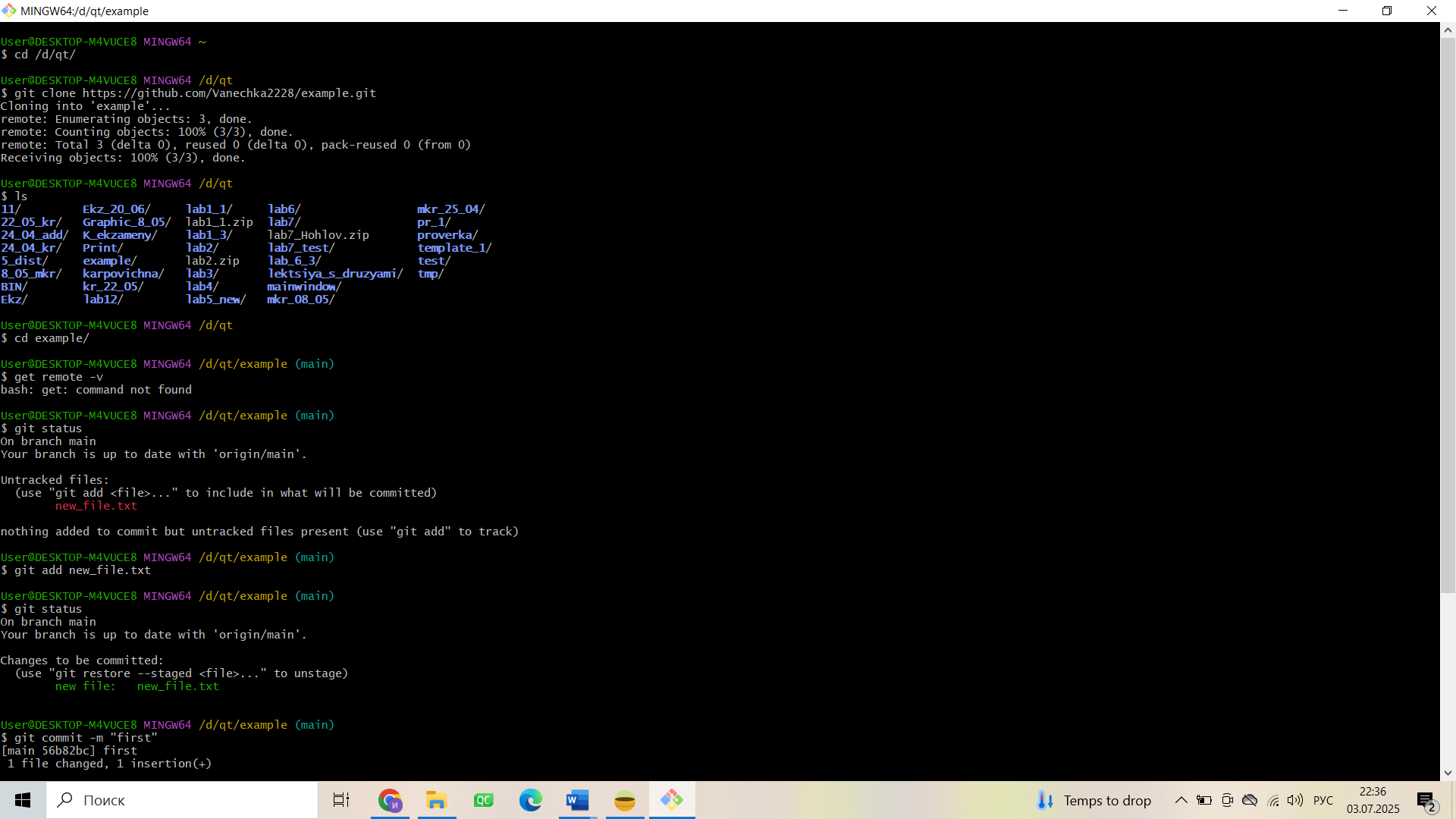
**

*Пример использования команды git clone*

*Команда* ***cd <path>*** *(change directory) позволяет сменить директорию на локальной машине, с который пользователь работает в данный момент.*

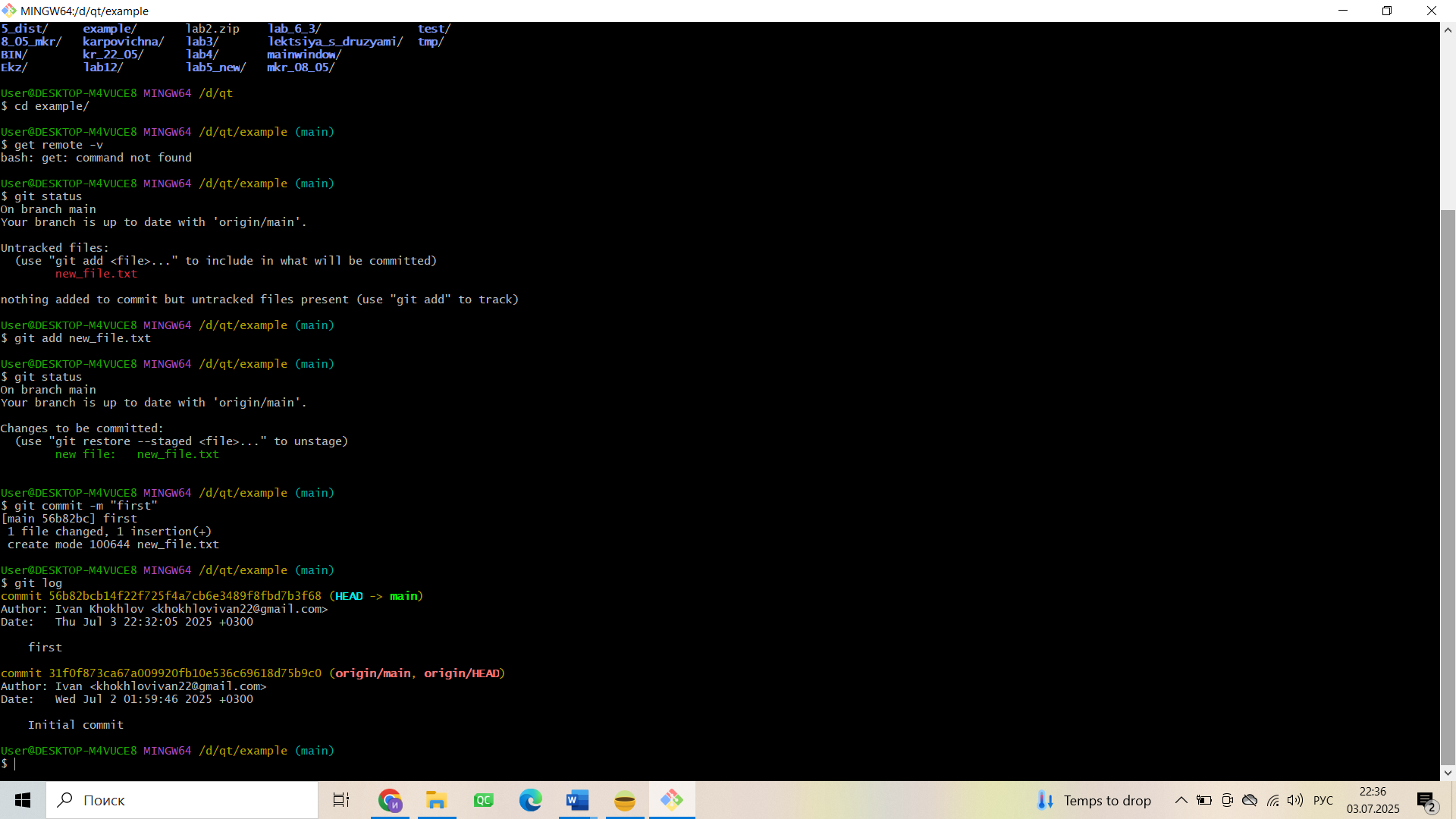
Git предоставляет широкий набор команд для работы с изменениями файлов репозитория:

* **git add** <**file>** – добавляет изменённые файлы в индекс (staging area) – промежуточную область между рабочей директорией и локальным репозиторием, где накапливаются файлы, готовые к отправке в репозиторий
* **git status** – отображает сообщение, оповещающее о наличии/отсутствии в рабочей директории изменённых с момента последнего сохранения файлов, не добавленных в индекс



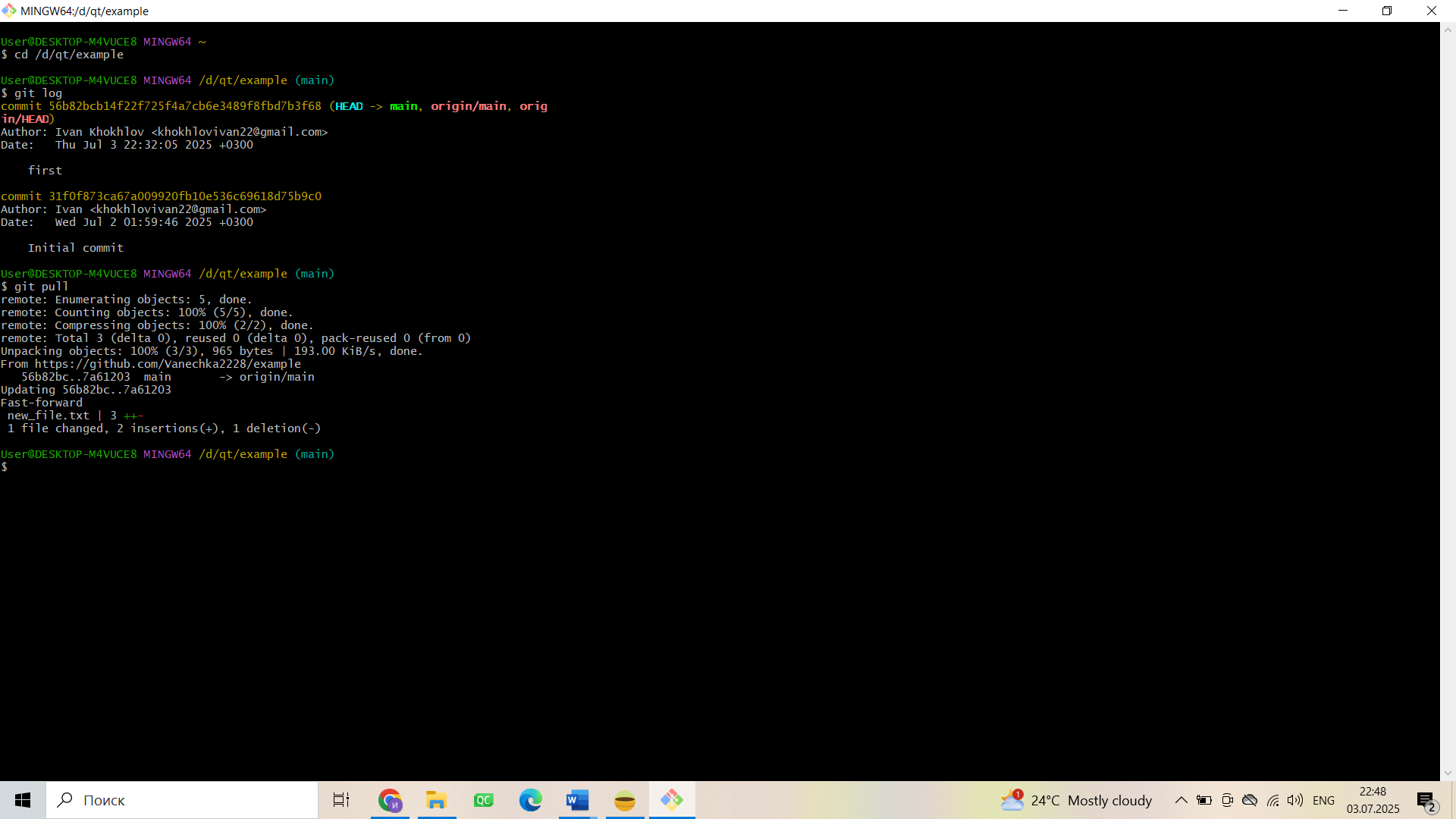
*Пример использования команд git status и git add <file>*

* **git commit -m "Commit message"** — делает коммит (сохраняет снимок текущего состояния индекса в локальном репозитории) с сообщением, введённым пользователем
* **git log** – показывает историю коммитов



*Пример использования команд git commit -m “Commit message” и git log*

* **git push** – передаёт изменения из локального репозитория в удалённый
* **git pull** – извлекает последнее обновление из удалённого репозитория и сохраняет его в локальный, соответствует последовательному вызову двух команд: git fetch и git merge

****

*Пример использования команды git pull*

* **git checkout <commit> -- <file>** – перезаписывает указанный файл содержимым из коммита <commit>, причём в качестве нужного коммита указываются начальные символы его SHA-1-кода
* **git commit --amend** – редактирование коммита (позволяет пользователю внести дополнительные изменения, отправить их в индекс и отредактировать коммит)
* **git reset HEAD <file>** – отмена индексации изменённого файла
* **git restore --staged <file>** – также отменяет индексацию файла
* **git restore <file>** –возвращает содержимое изменённого файла к последней зафиксированной версии

## Ветвление и слияние

Ветвление является неотъемлемой частью командных разработок, так как даёт возможность работать над разными версиями исходного кода. Основная идея ветвления – отклонение от основного кода и последующая разработка проекта независимо от него.

Под веткой обычно понимают независимую хронологически упорядоченную последовательность коммитов. Однако конкретно в Git, в силу его реализации в виде графа, вершины которого – это указатели на коммиты, соединённые с предыдущими вершинами-указателями, ветка фактически является указателем на последний коммит в ней, а создание ветки сводится к созданию нового указателя, ссылающегося на текущий коммит.

По умолчанию главной веткой репозитория принято считать ветку под именем master (либо main), которая создаётся автоматически сразу при инициализации репозитория.

Команда **git branch** — главный инструмент для работы с ветвлением:

с её помощью можно как добавлять новые ветки, так и удалять их.

* **git branch <branchname>** – добавляет новую ветку

*После вызова git branch <branchname> новая ветка уже создана, но пользователь по-прежнему находится в предыдущей ветке. Для перехода в новую ветку используется команда* ***git checkout <branchname>****.*

* **git branch -d <branchname> – удаление ветки**

*Для корректного удаления нужно помнить правила:*

* *Нельзя удалить ветку, в которой мы находимся.*
* *Git не позволит удалить ветку, у которой есть несохраненные изменения.*

Чаще всего при создании новой ветки разработчику нужно сразу же переключиться на неё. Это можно сделать за одну команду:

**git checkout -b <branchname>**

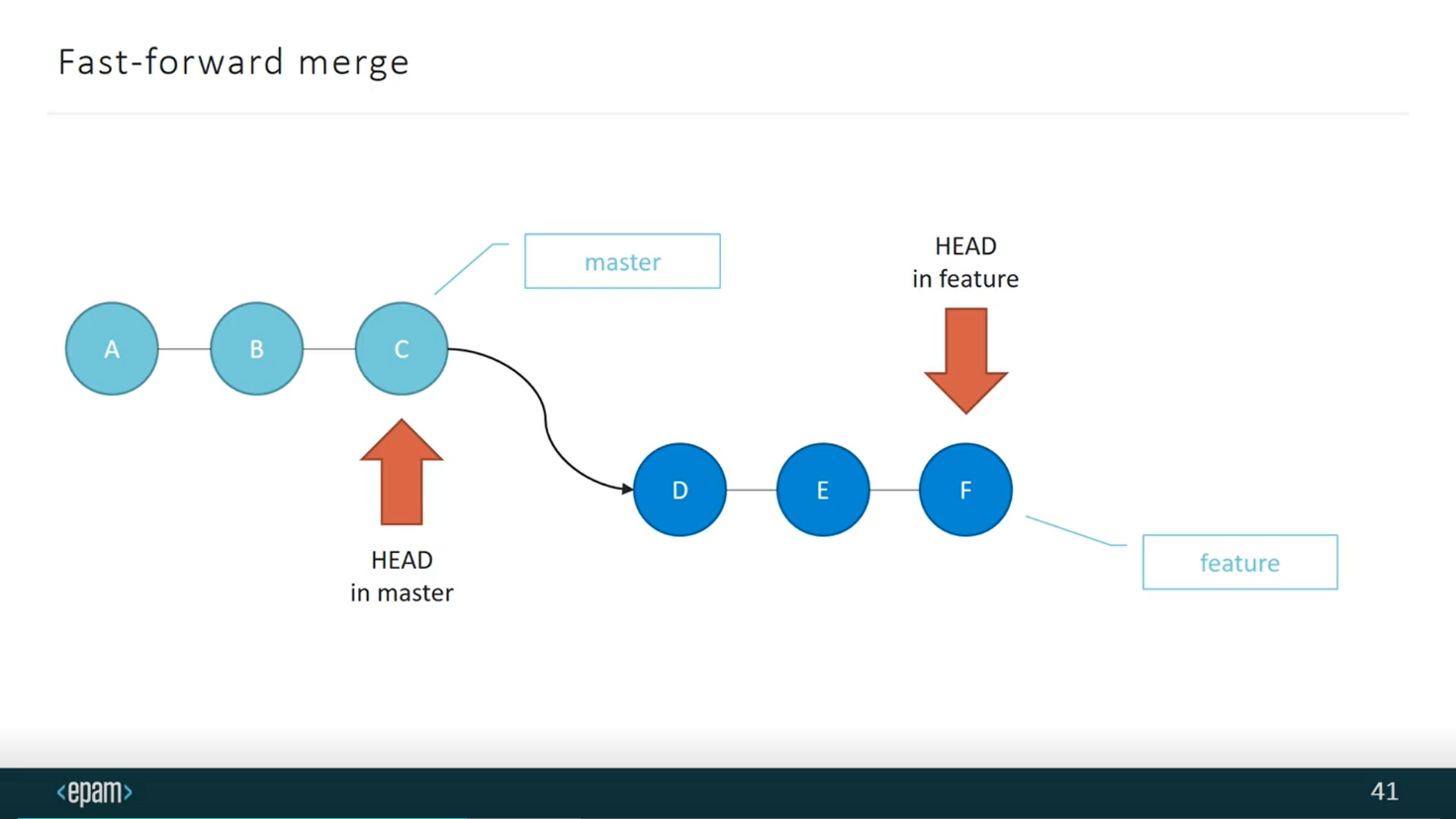
Или же:

**git checkout branch <branchname>**

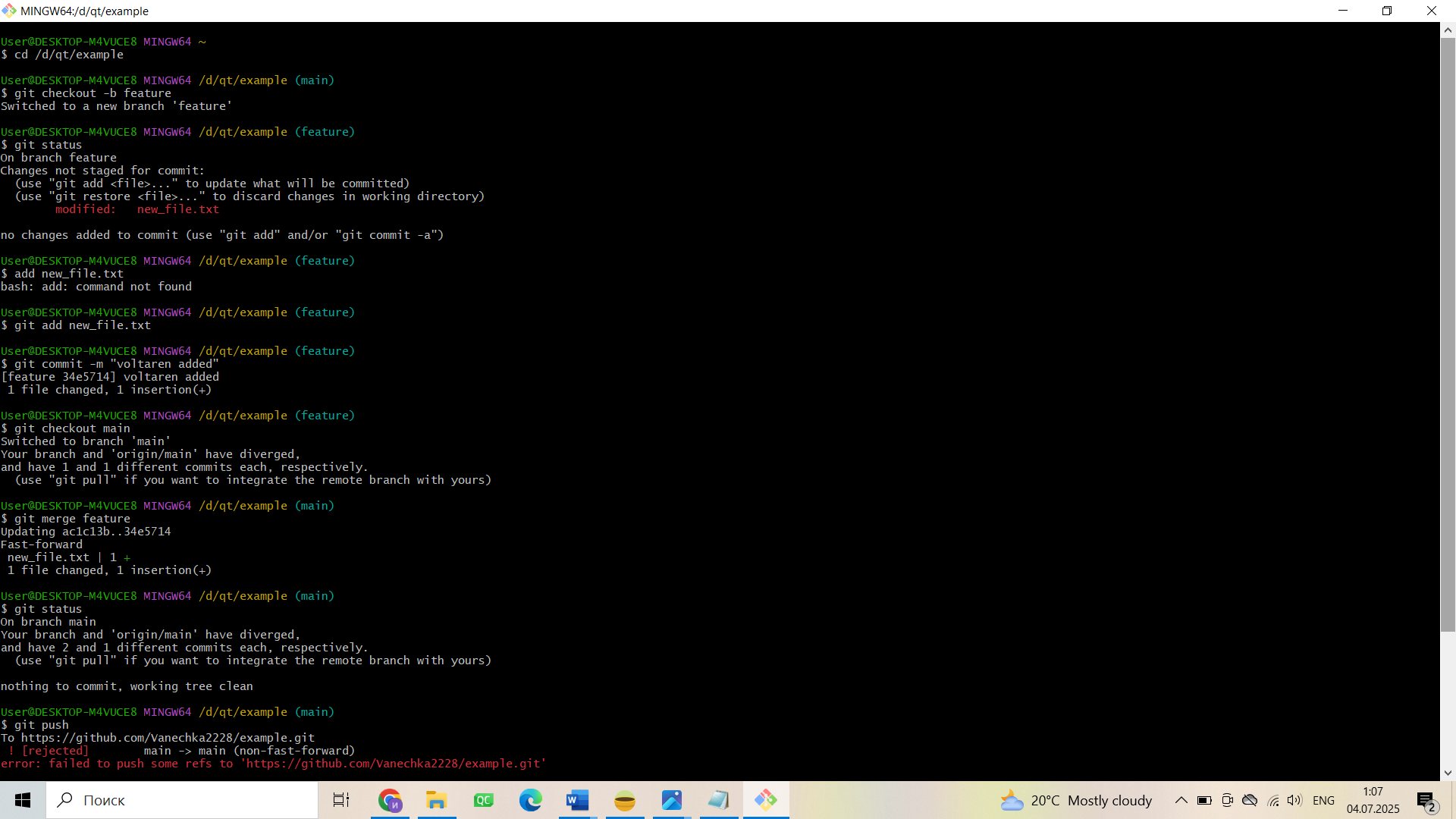
Завершив работу на неким новым функционалом, ради разработки которого и создавалась отдельная ветка, разработчики обычно переносят новшества в другую, основную ветку. Специально для этой цели в Git существует операция слияния (команда **git merge <branchname>** – ветка <branchname> вливается в ту ветку, в которой пользователь находится на момент вызова команды). К слову, вливать друг в друга можно любые ветки – с точки зрения функционала Git нет никакой разницы, вливается ветка с новым функционалом в основную или наоборот.

Выделяют два вида слияния: **fast-forward merge** и **non fast-forward merge**.

Fast-forward merge применяется в ситуации, когда дочерняя ветка «отпочковывается» от последнего коммита родительской. В этом случае слияния веток Git производит автоматически: указатель на последний коммит родительской ветки просто заменятся на указатель на последний коммит дочерней, т.е. меняется HEAD родительской ветки.

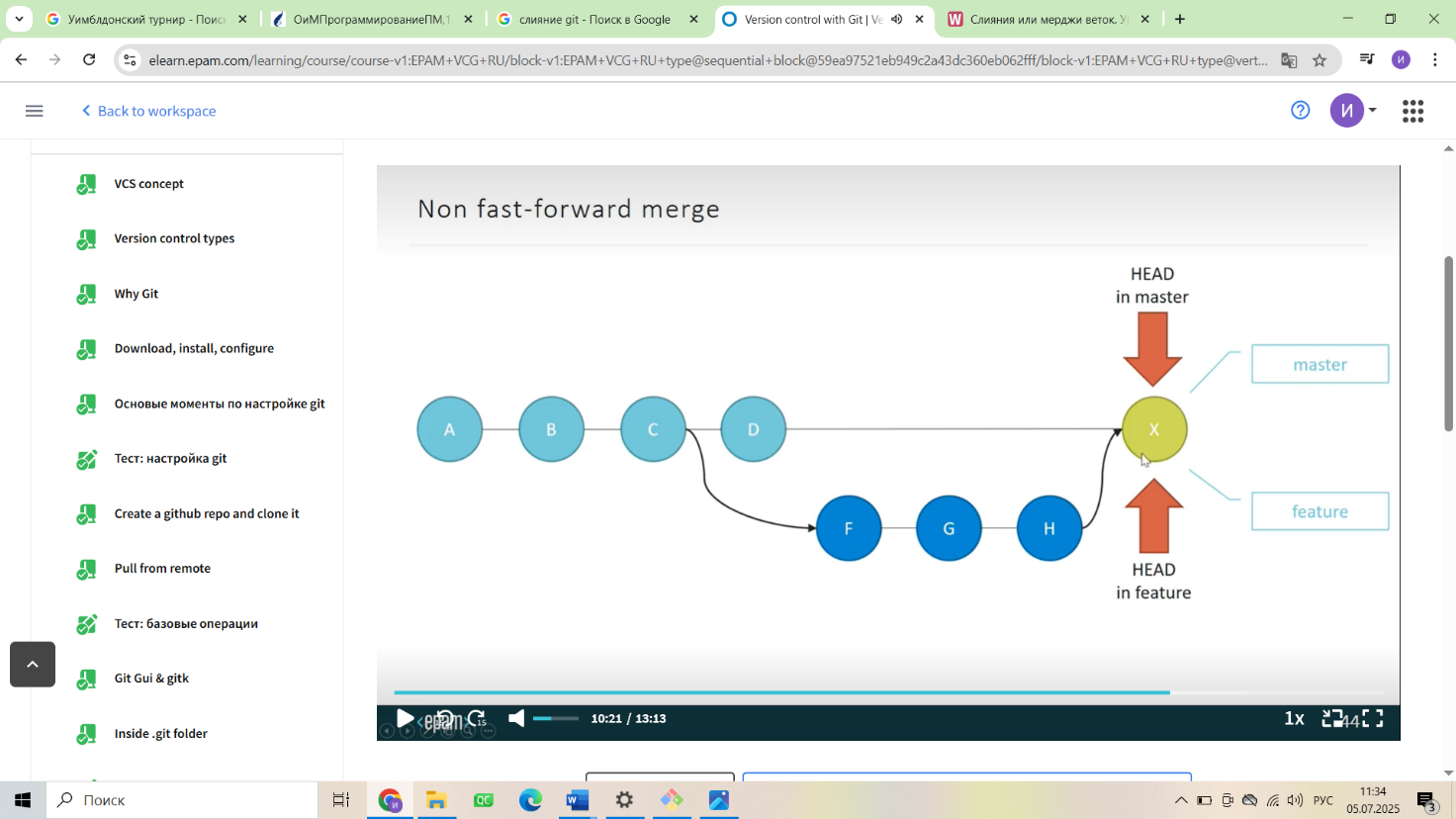


*Иллюстрация ситуации, в которой применяется fast-forward merge*[*[2]*](https://elearn.epam.com/learning/course/course-v1:EPAM+VCG+RU/block-v1:EPAM+VCG+RU+type@sequential+block@59ea97521eb949c2a43dc360eb062fff/block-v1:EPAM+VCG+RU+type@vertical+block@87bfd681590e465fbd82a3123eae68c7)

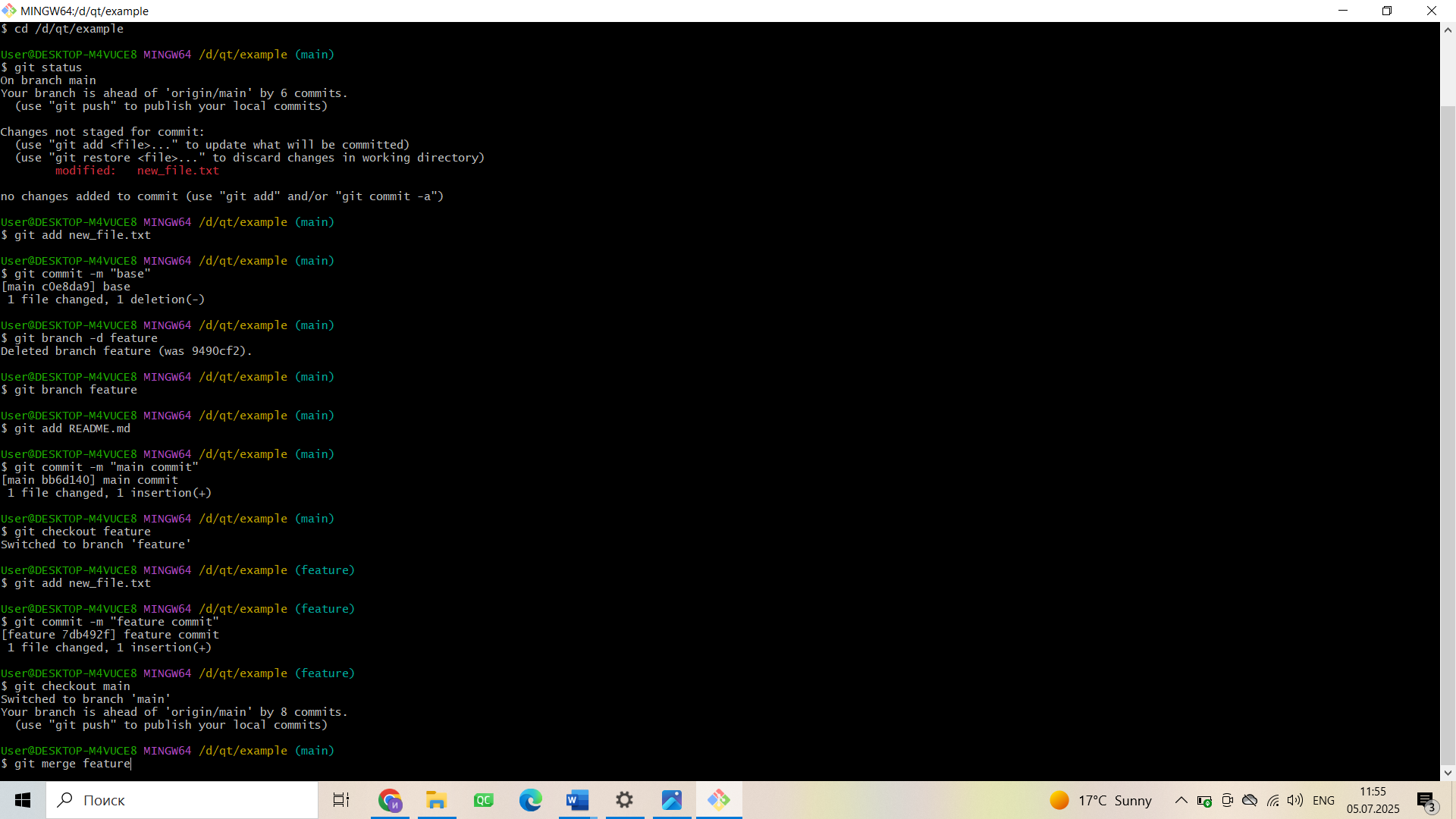


*Пример процесса fast-forward слияния веток*

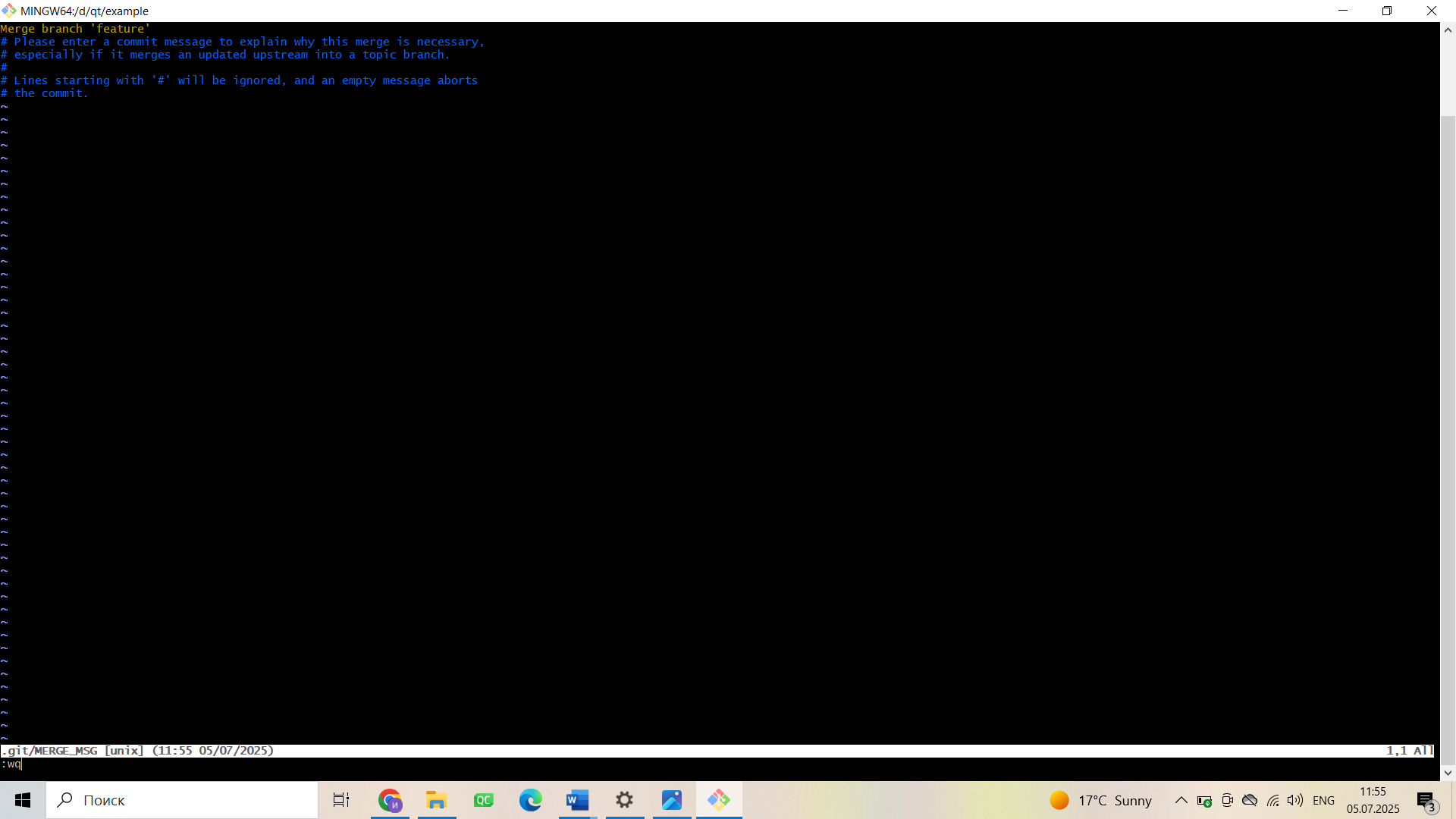
В некоторых случаях применить fast-forward слияние невозможно, так как от коммита основной ветки, от которого «отпочковалась» дочерняя, отходят коммиты и в родительской ветке, т.е. пользователь просто потерял бы все изменения в родительской ветке, выполненные после создания новой ветки. В таком случае создаётся новый результирующий коммит.



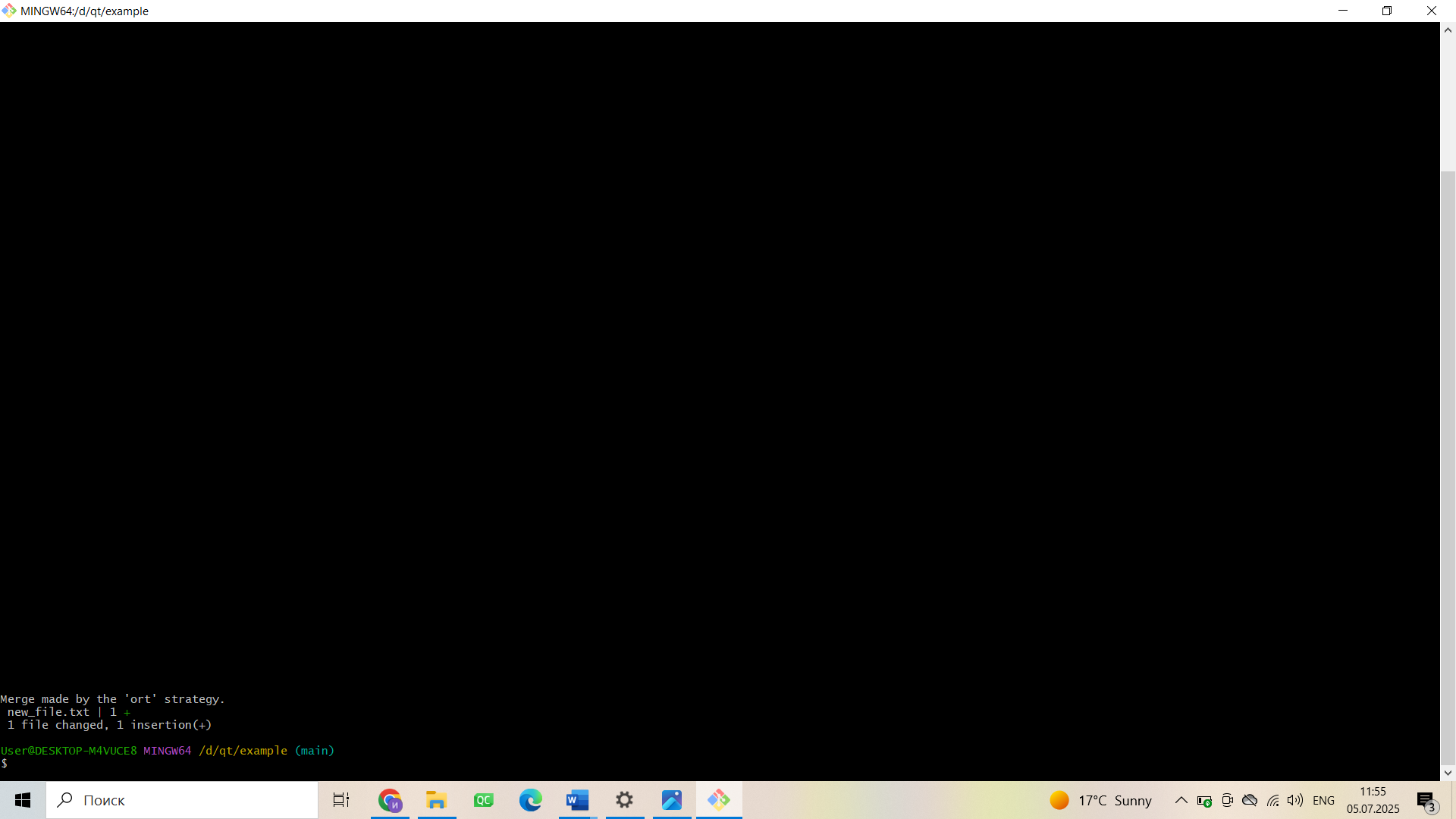
*Схема non fast-forward merge*[*[3]*](https://elearn.epam.com/learning/course/course-v1:EPAM+VCG+RU/block-v1:EPAM+VCG+RU+type@sequential+block@59ea97521eb949c2a43dc360eb062fff/block-v1:EPAM+VCG+RU+type@vertical+block@87bfd681590e465fbd82a3123eae68c7)

**

Часть кода с операцией non fast-forward merge. В таком случае при вызове команды git merge открывается встроенный в Git Bash текстовый редактор Vi, требующий указать commit message:

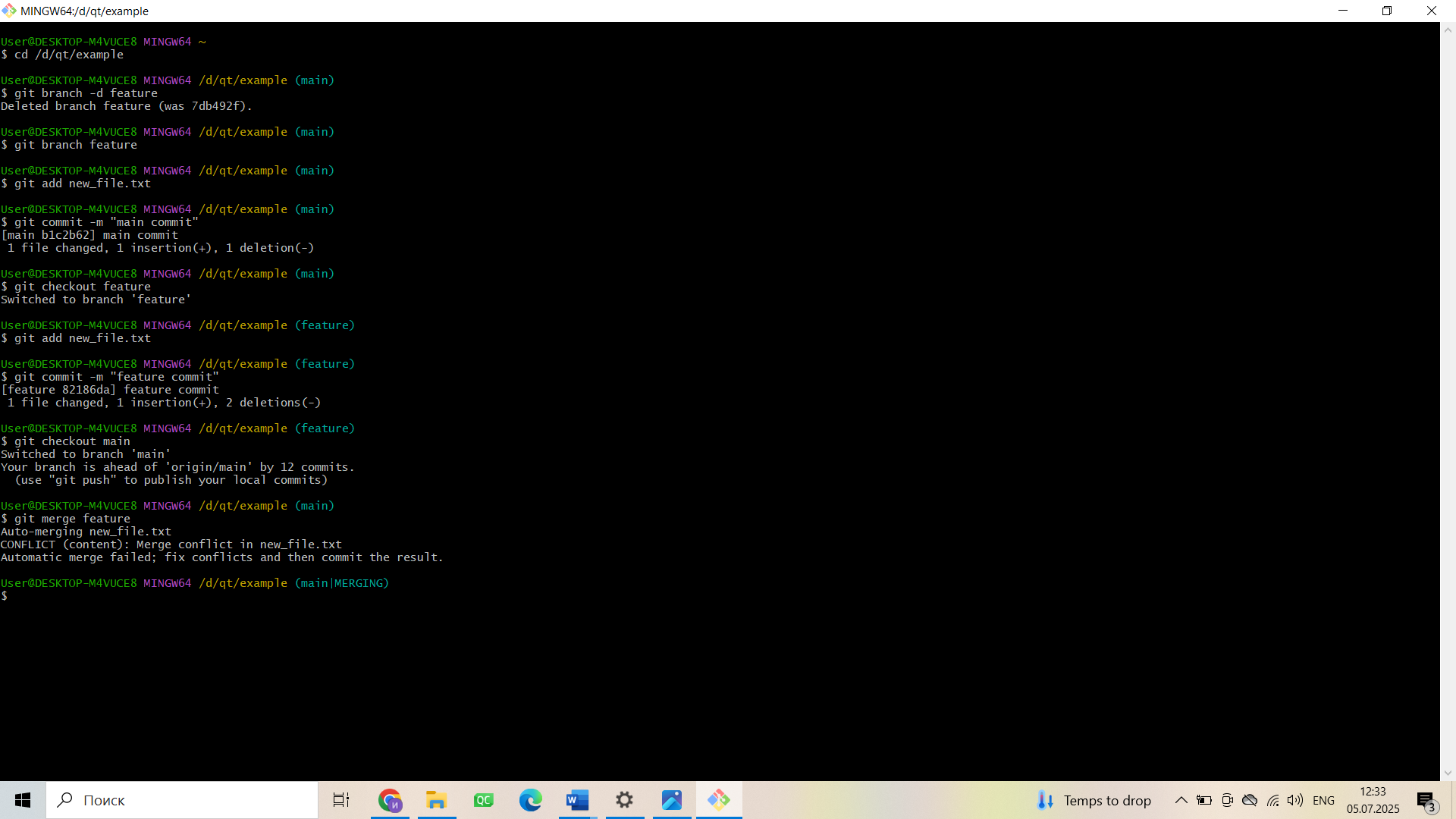
**

*Вводим команду :wq (write & quit) и нажимаем ENTER, после чего редактор закрывается и Git производит слияние веток:*

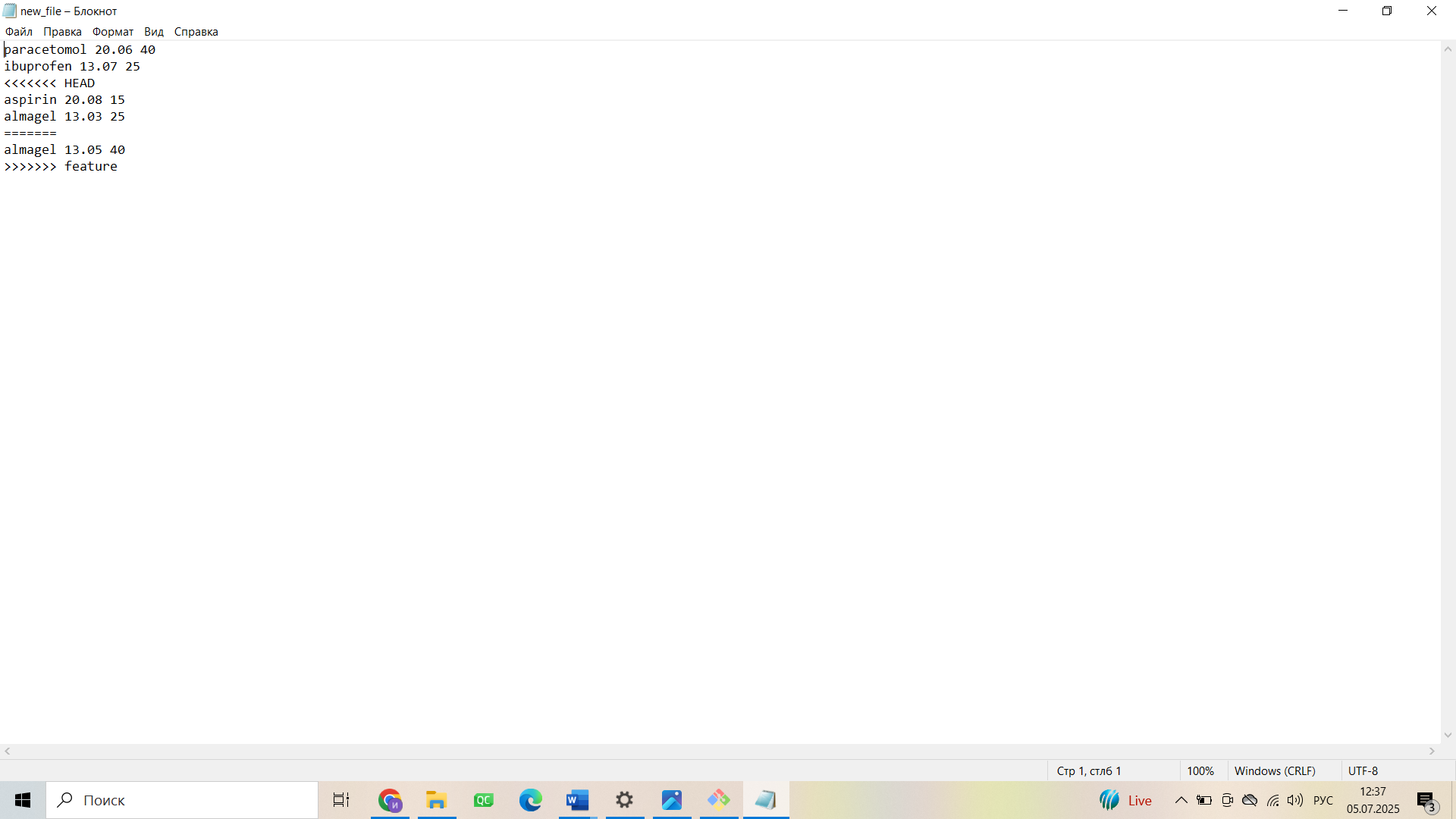
**

## Разрешение конфликтов

При слиянии веток может возникнуть следующая ситуация: некоторый файл проекта был изменён в обеих ветках, причём изменения разнятся. В таком случае Git не может самостоятельно решить, какая из версий файла верна, и не может выполнить слияние веток автоматически:



Такую ситуацию и называют конфликтом данных при слиянии веток. Файл, из-за которого происходит конфликт, принимает следующий вид:



В верхней части видим часть файла часть, общую для обеих версий файла, между маркерами (<) и (=) – отличительную часть версии из ветки main, между (=) и (>) – ветки feature.

Git предлагает нам два решения проблемы: привести данный файл вручную прямо в блокноте к нужному виду и вызвать команду **git add**, а затем **git commit** для сохранения изменений, либо прервать слияние командой **git merge --abort**.

## 1.3 Выводы к главе 1

В рамках теоретической части учебной практики были рассмотрены концепция, устройство, классификация систем контроля версий.

На примере системы Git изучены основные принципы работы с системами контроля версий. Освоены установка и настройка Git; работа на онлайн-хостинге GitHub, создание удалённых репозиториев на платформе; базовые команды для работы с файлами и историей изменений проекта. Рассмотрена концепция ветвления и алгоритмы слияния веток, решения конфликтов данных при слиянии.