

Система контролю версій

Git





Лекція 3. Система контролю версій Git

План

1. Базові поняття
2. Створення репозиторію
3. Операції з репозиторіями
4. Галуження проекту
5. GitHub

1. Базові поняття

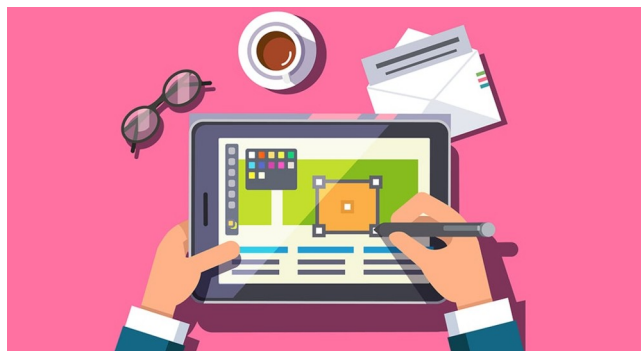
Система керування версіями (VCS – Version Control System) – це автоматизований засіб, який фіксує історію змін у файлах проекту (кодовій базі) протягом часу і дає можливість при необхідності повернутися до певних версій потрібних файлів.

Сучасні VCS дозволяють:

- повернути файли до стану, в якому вони були до змін;
- повернути проект до початкового стану;
- побачити зміни;
- побачити, які зміни призвели до проблем, хто їх вносив і коли;
- і багато іншого.

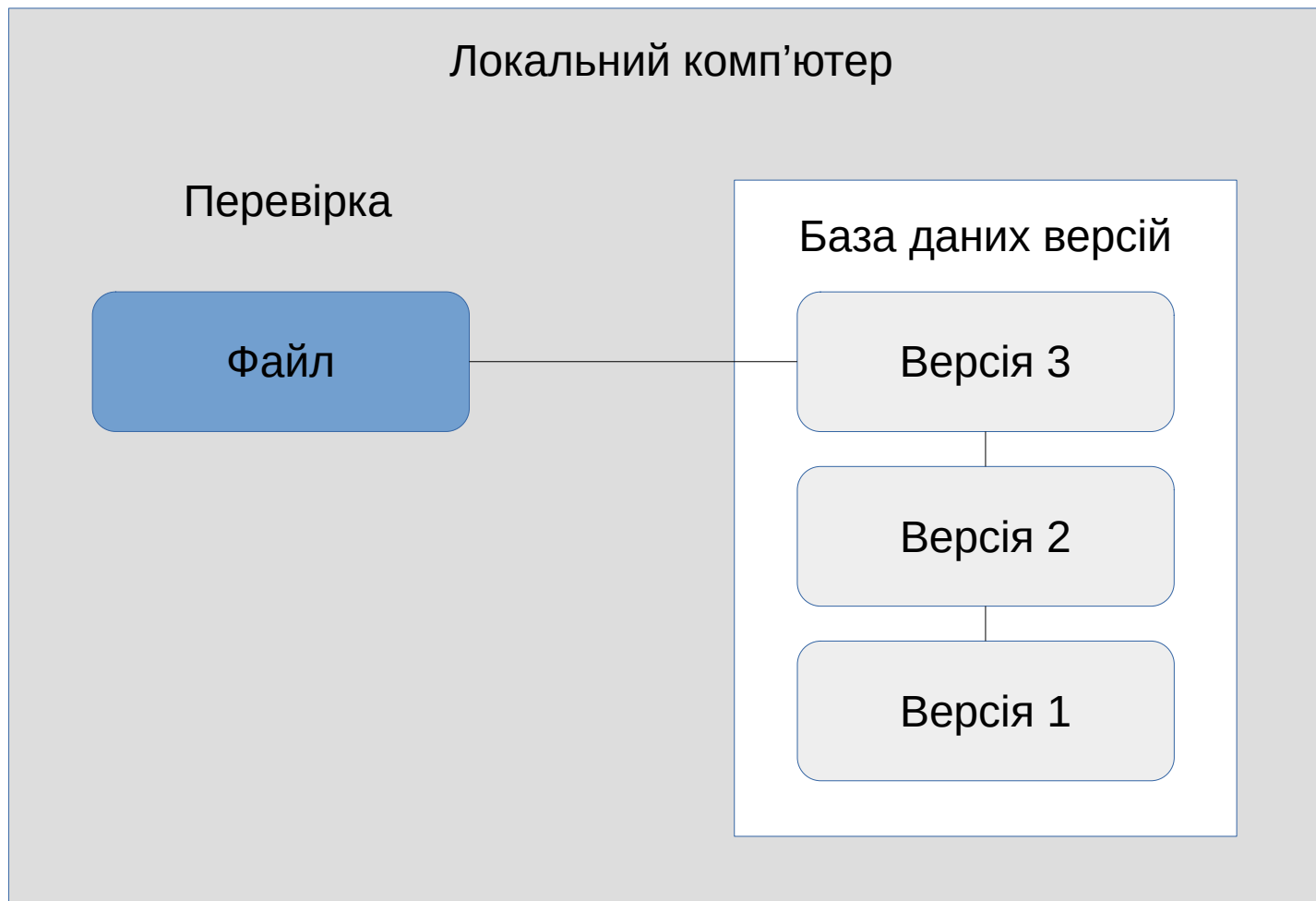
Використання VCS дає дозволяє автоматично полагодити проект, якщо ньому щось зламалося або були втрачені певні файли.

Використання VCS дозволяє зберігати всі проміжні версії проекту (наприклад, макети веб-сторінок або зображень), що часто буває конче потрібним на практиці.



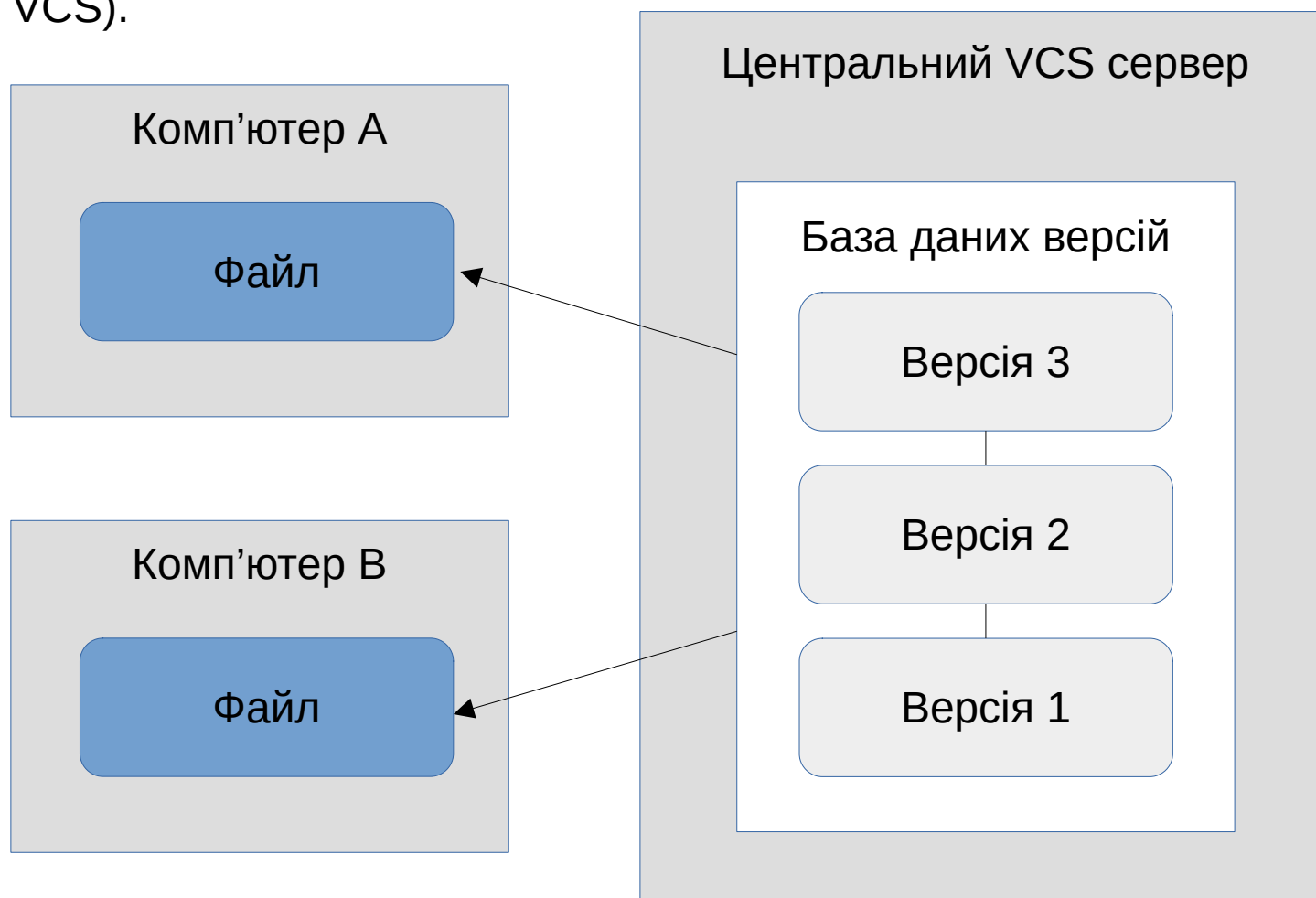
1. Базові поняття

Локальні системи контролю версіями (наприклад, RCS) зберігають у спеціальній базі даних всі зміни у файлах, що знаходяться під контролем версій.



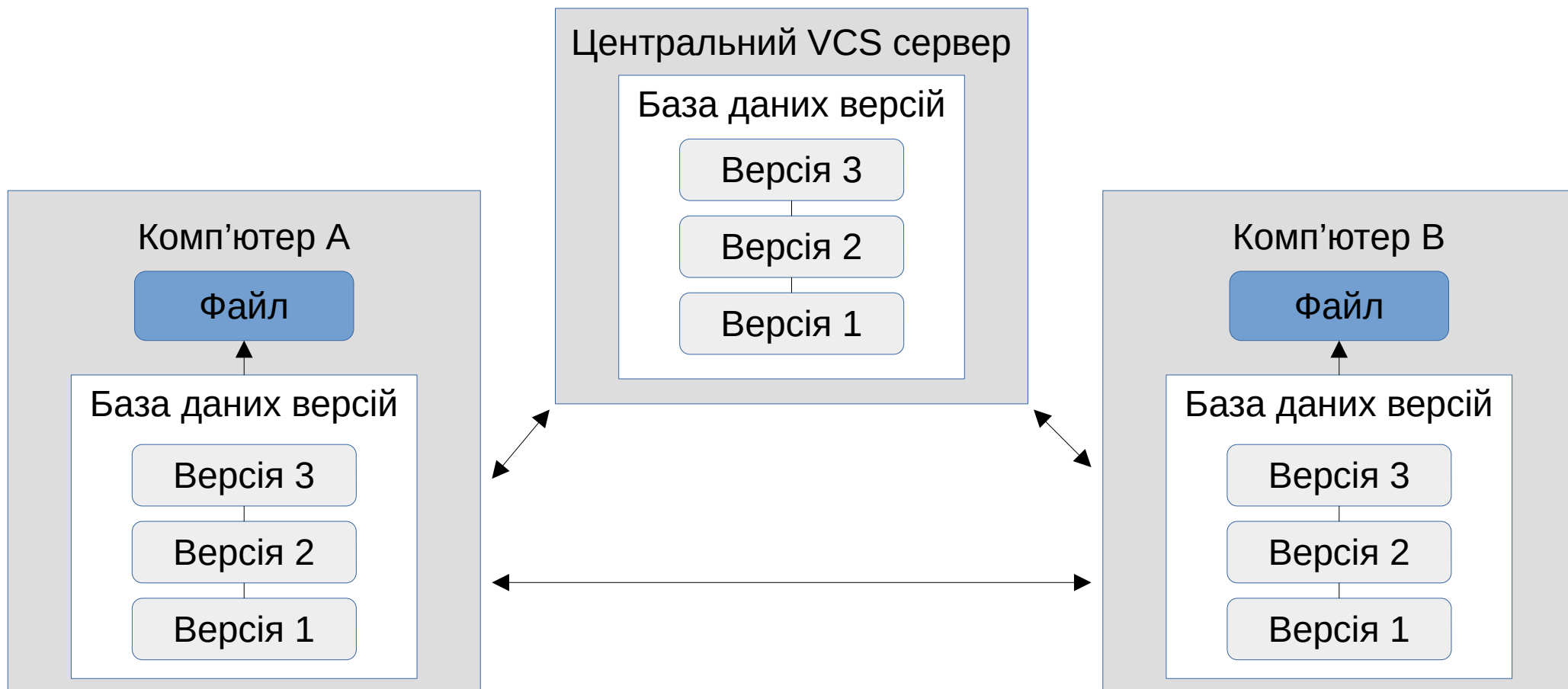
1. Базові поняття

Централізовані системи контролю версіями (CVS, Subversion, Perforce) використовуються для співпраці між різними розробниками і мають багато переваг у порівнянні з локальними. «Вузьким» місцем таких систем є потенційна можливість втрати даних при виході з ладу центрального сервера або бази даних версій (це стосується і локальних VCS).



1. Базові поняття

Децентралізовані системи контролю версіями – DVCS (Git, Mercurial, Bazaar, Darcs) позбавлені недоліків локальних і централізованих VCS, оскільки клієнти отримують повну копію сховища.



1. Базові поняття

Протягом **1991 – 2002** років розробники ядра **Linux** зберігали його початковий код у вигляді великої кількості патчів та архівів, що було дуже незручним.

З **2002** року проект ядра Linux почав використовувати пропрієтарну децентралізовану систему керування версіями **BitKeeper**, проте у **2005** році її власники скасували безкоштовне використання свого продукту, що підштовхнуло автора Linux **Лінуса Торвальдса** до розробки власної системи VCS, яка отримала назву Git.

При розробці Git ставилися наступні цілі:

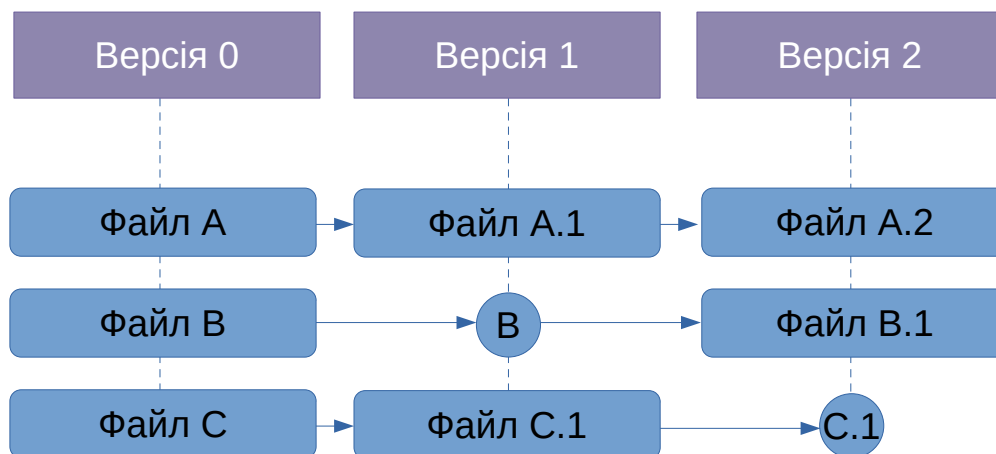
- швидкість;
- проста архітектура;
- сильна підтримка для нелінійного розвитку (тисячі паралельних гілок);
- децентралізація;
- можливість ефективно управляти великими проектами.

З моменту своєї появи у **2005** році, Git постійно розвивався і на сьогодні, завдяки своїй швидкості та ефективності при роботі з великими проектами, став однією з найбільш популярних VCS.

1. Базові поняття

Основною відмінністю Git від інших DVCS (CVS, Subversion, Perforce, Bazaar тощо) є спосіб роботи зі своїми даними. Більшість інших систем зберігають інформацію у вигляді списку змін у файлах: вони представляють інформацію як певний набір файлів і змін, зроблених у кожному файлі у часі.

Підхід Git до зберігання даних схожий на набір знімків мініатюрної файлової системи: кожного разу, коли користувач робить збереження змін (фіксацію), тобто зберігає стан свого проекту в Git, система запам'ятовує як виглядає кожен файл у цей момент, та зберігає посилання на цей знімок. Для збільшення ефективності, якщо файли не були змінені, Git не запам'ятовує ці файли знову, а лише створює посилання на попередню версію файлу, який вже збережений.



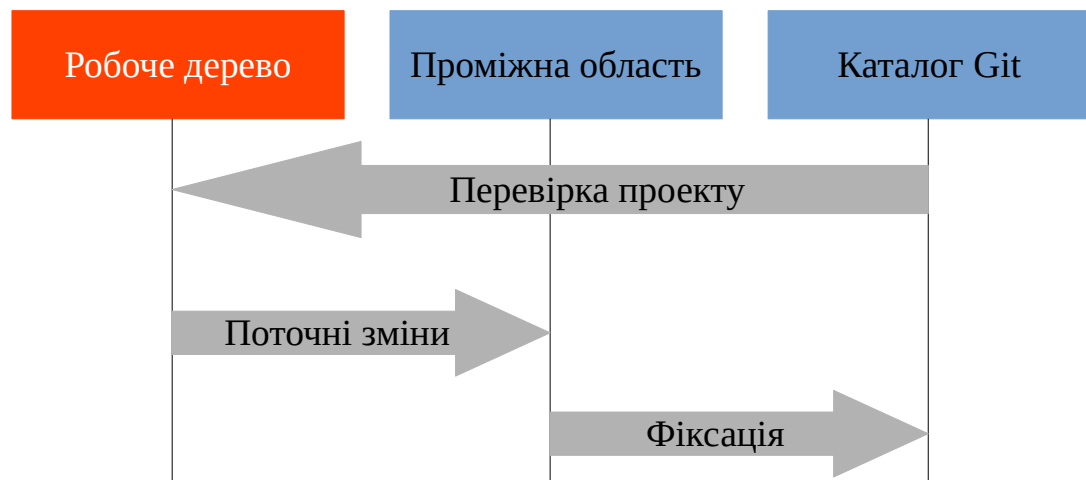
Зберігання даних у вигляді знімків проекту у часі

1. Базові поняття

Файли у Git можуть знаходитися у трьох основних станах:

- 1) **зміненому** (modified) – коли у файлі вже відбулися певні зміни, але вони ще не були зафіксовані;
- 2) **індексований** (staged) – коли змінений файл у його поточній версії був позначений для наступної фіксації;
- 3) **зафіксованому** (committed) – коли файл вже збережено у локальній базі.

Тобто проект в Git складається з трьох основних розділів: **робочого дерева** (working tree), **проміжної області** (staging area) і **каталогу Git** (Git directory) або репозиторію.



Робоче дерево, проміжна область і каталог Git

1. Базові поняття

Робоче дерево (робочий каталог) – це знімок поточної версії проекту. Файли розпаковуються зі збереженої бази даних у каталозі Git і розміщуються на диску, для того щоб їх можна було змінювати та використовувати.

Проміжна область – це спеціальний файл, що міститься у Git-каталозі користувача і зберігає інформацію про те, які зміни будуть включені до наступної фіксації (коміту). Цей файл часто називають «індексом».

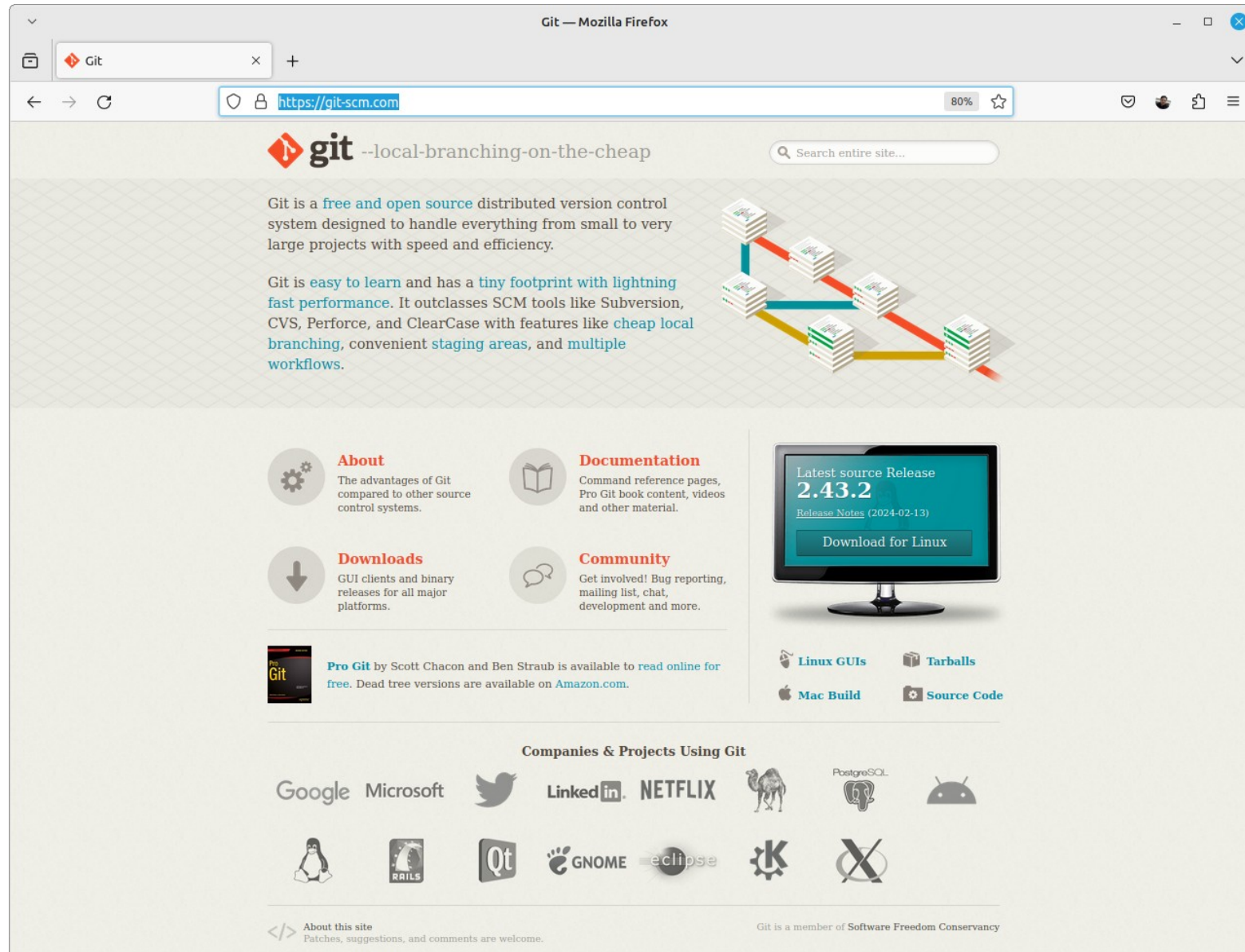
Каталог Git – це сховище, де Git зберігає метадані та базу даних об'єктів для поточного проекту. Це найважливіша частина Git, і саме вона копіюється, коли користувач клонує репозиторій з іншого комп'ютера.

Процес розробки із використанням Git виглядає таким чином. Користувач:

- вносить зміни у файли проекту у робочому дереві;
- вибірково встановлює лише ті зміни, які хоче включити до свого наступного коміту, що додає лише ці зміни до проміжної області;
- виконує фіксацію, яка запам'ятовує файли такими, якими вони є в проміжній області; ці файли (знімок) постійно зберігаються у каталозі Git.

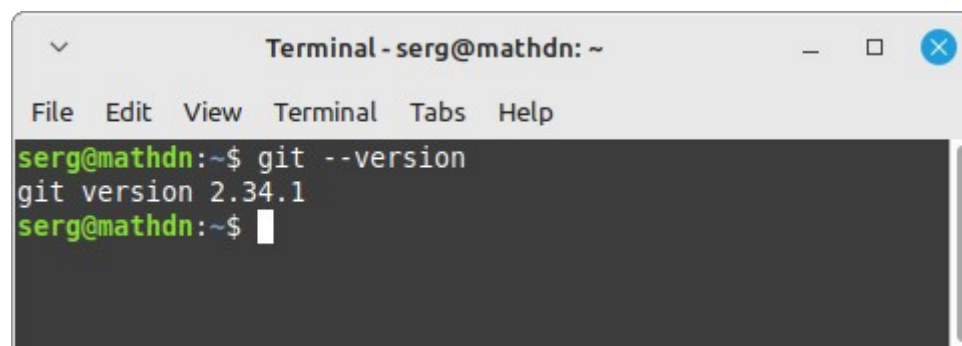
1. Базові поняття

Офіційна веб-сторінка Git: <https://git-scm.com/>



1. Базові поняття

Після встановлення Git його актуальну версію можна перевірити за допомогою команди **git --version**, яку слід ввести у командному рядку терміналу (хоча існують і утиліти для роботи з Git, які мають графічний інтерфейс користувача).

A screenshot of a terminal window titled "Terminal - serg@mathdn: ~". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content shows the command "serg@mathdn:~\$ git --version" being entered, followed by the output "git version 2.34.1". The prompt "serg@mathdn:~\$" is shown again on the next line with a cursor.

```
Terminal - serg@mathdn: ~
File Edit View Terminal Tabs Help
serg@mathdn:~$ git --version
git version 2.34.1
serg@mathdn:~$
```

Після інсталяції Git слід налаштувати середовище роботи. Це потрібно зробити лише один раз – при оновленні версії Git налаштування збережуться, але, при необхідності, користувач може змінити їх у будь-який момент, виконавши такі самі команди ще раз. Перше, що слід зробити після встановлення Git, це налаштувати **логін** та **адресу електронної пошти користувача**, оскільки кожен коміт у Git повинен містити таку інформацію: **git config --global user.name "CurrentUserLogin"** та **git config --global user.email "current.user.mail@example.com"**.

Для перевірки наявних налаштувань можна скористатися командою **git config --list**.

2. Створення репозиторію

Репозиторій Git можна отримати одним з двох способів: 1) перетворити локальну теку, яка ще не знаходиться під контролем версій, на репозиторій Git; 2) клонувати вже наявне сховище Git з іншого каталогу (у тому числі й віддаленого). Після цього користувач отримує на локальній машині готове до роботи Git-сховище.

Для створення репозиторію для вже наявного проекту слід перейти в його теку і виконати команду **git init**. Після виконання цієї команди у поточному каталозі буде створено нову теку **.git**, в якій зберігаються всі необхідні для функціонування поточного репозиторію файли.

Для додавання файлів у репозиторій слід скористатися командою **git add**, параметром якої є назва (або маска) файлу. Наприклад, **git add *.c** додасть до індексу Git всі файли з початковим кодом мови програмування C.

Якщо вказати команду **git add .**, то в репозиторій будуть додані всі файли й каталоги, що знаходяться в поточній теці. Для виконання коміту слід скористатися командою **git commit**, наприклад, у такому форматі: **git commit -m 'Перша версія проекту'**, де рядок у лапках визначає обов'язковий коментар для поточної фіксації проекту.

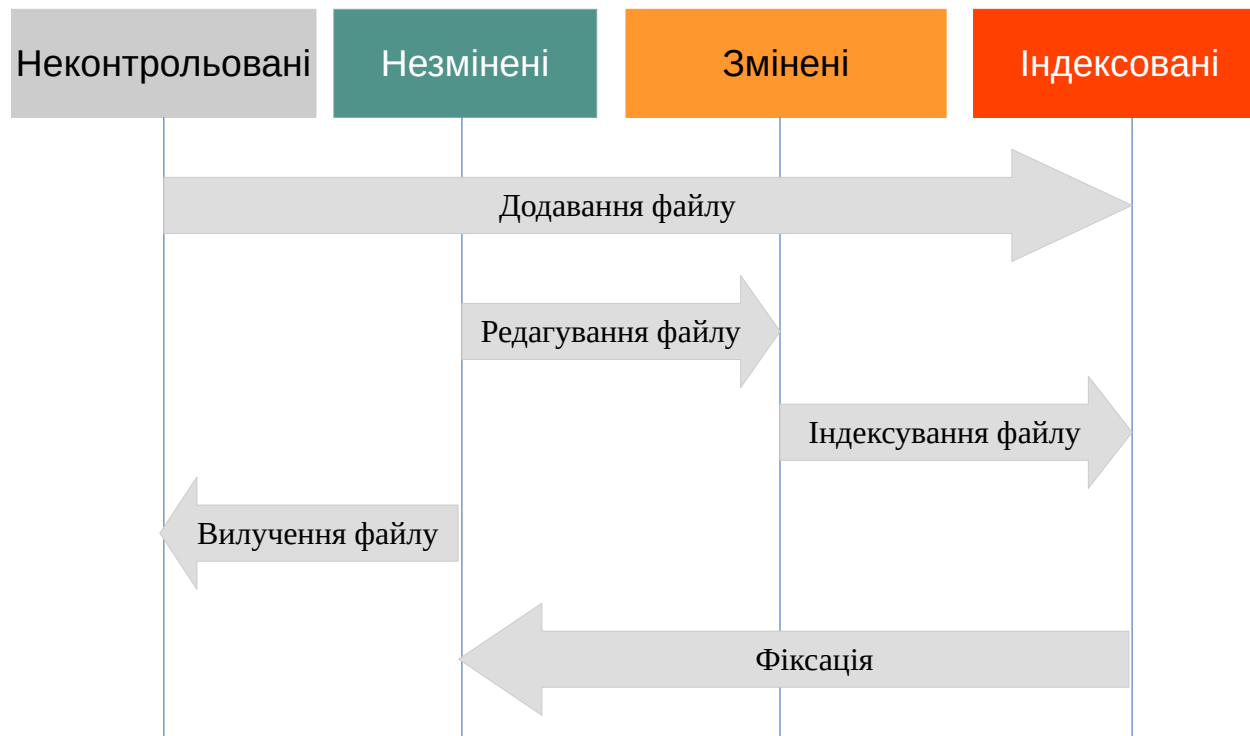
Для отримання копії наявного Git-репозиторія слід використати команду **git clone <url>**. Наприклад, **git clone https://github.com/pmp-library/pmp-library** створить клон деякого репозиторію, що знаходиться в мережі за відповідним посиланням.

3. Операції з репозиторіями

Кожен файл робочого каталогу може бути в одному з двох станів:

- **контрольований** (tracked) – знаходиться в останньому знімку;
- **неконтрольований** (untracked) – з'явився в репозиторії після останнього коміту (ще не існує у індексі поточного проекту).

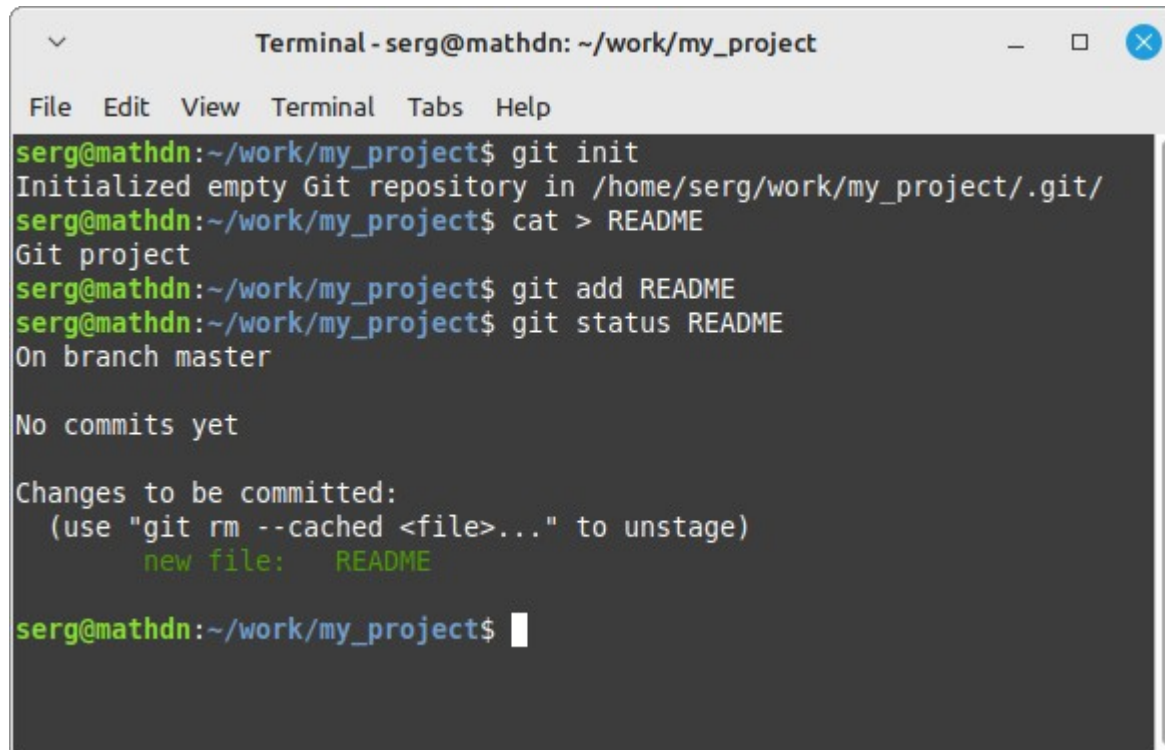
При редагуванні файлів Git бачить, що вони були змінені після останнього коміту. Впродовж роботи користувач вибірково індексує ці змінені файли та потім зберігає всі індексовані зміни, після чого цикл повторюється.



Життєвий цикл статусу файлів у Git репозиторії

3. Операції з репозиторіями

Для перевірки статусу файлів репозиторію використовується команда **git status**. Правилем гарного тону при використанні Git є наявність в кожному репозиторії файлу з його описом. Додати такий файл, наприклад, можна командою **git add README**. Тоді команда **git status** виведе наступну інформацію.

A terminal window titled "Terminal - serg@mathdn: ~/work/my_project" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the following commands and output:

```
serg@mathdn:~/work/my_project$ git init
Initialized empty Git repository in /home/serg/work/my_project/.git/
serg@mathdn:~/work/my_project$ cat > README
Git project
serg@mathdn:~/work/my_project$ git add README
serg@mathdn:~/work/my_project$ git status README
On branch master

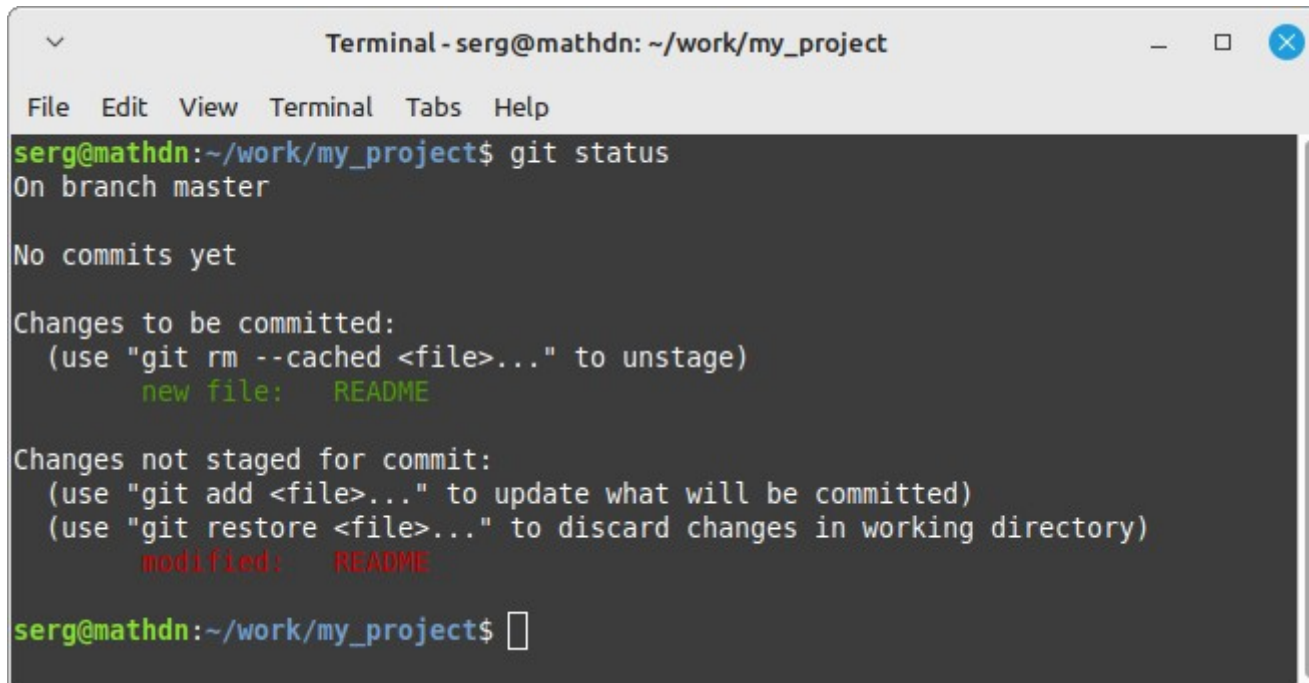
No commits yet

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

serg@mathdn:~/work/my_project$
```

3. Операції з репозиторіями

Якщо тепер у файл README внести зміни і ще раз виконати команду **git status**, то результат буде наступний.

A terminal window titled "Terminal - serg@mathdn: ~/work/my_project" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the command "serg@mathdn:~/work/my_project\$ git status" and its output. The output indicates the current branch is master, there are no commits yet, and there are changes to be committed (new file: README) and changes not staged for commit (modified: README).

```
Terminal - serg@mathdn: ~/work/my_project
File Edit View Terminal Tabs Help
serg@mathdn:~/work/my_project$ git status
On branch master

No commits yet

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

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:   README

serg@mathdn:~/work/my_project$
```

Для індексації раніше зміненого файлу можна ще раз скористатися командою **git add**, наприклад, `git add README` додасть до індексу Git щойно змінений файл README. Тоді, якщо виконати команду **git status** ще раз, результат буде такий, як і на слайді № 16.

3. Операції з репозиторіями

На практиці часто виникає ситуація, коли користувач хоче, щоб певний клас файлів автоматично не індексувався системою Git (наприклад, це може стосуватися згенерованих компілятором файлів). Для цього створюється файл **.gitignore**, який містить відповідні шаблони тимчасових файлів. Так, для проектів на мовах програмування C/C++ він може мати такий вміст (фрагмент):

```
# Compiled Object files
```

```
*.o
```

```
*.obj
```

```
# Compiled Dynamic libraries
```

```
*.so
```

```
*.dll
```

```
# Compiled Static libraries
```

```
*.lib
```

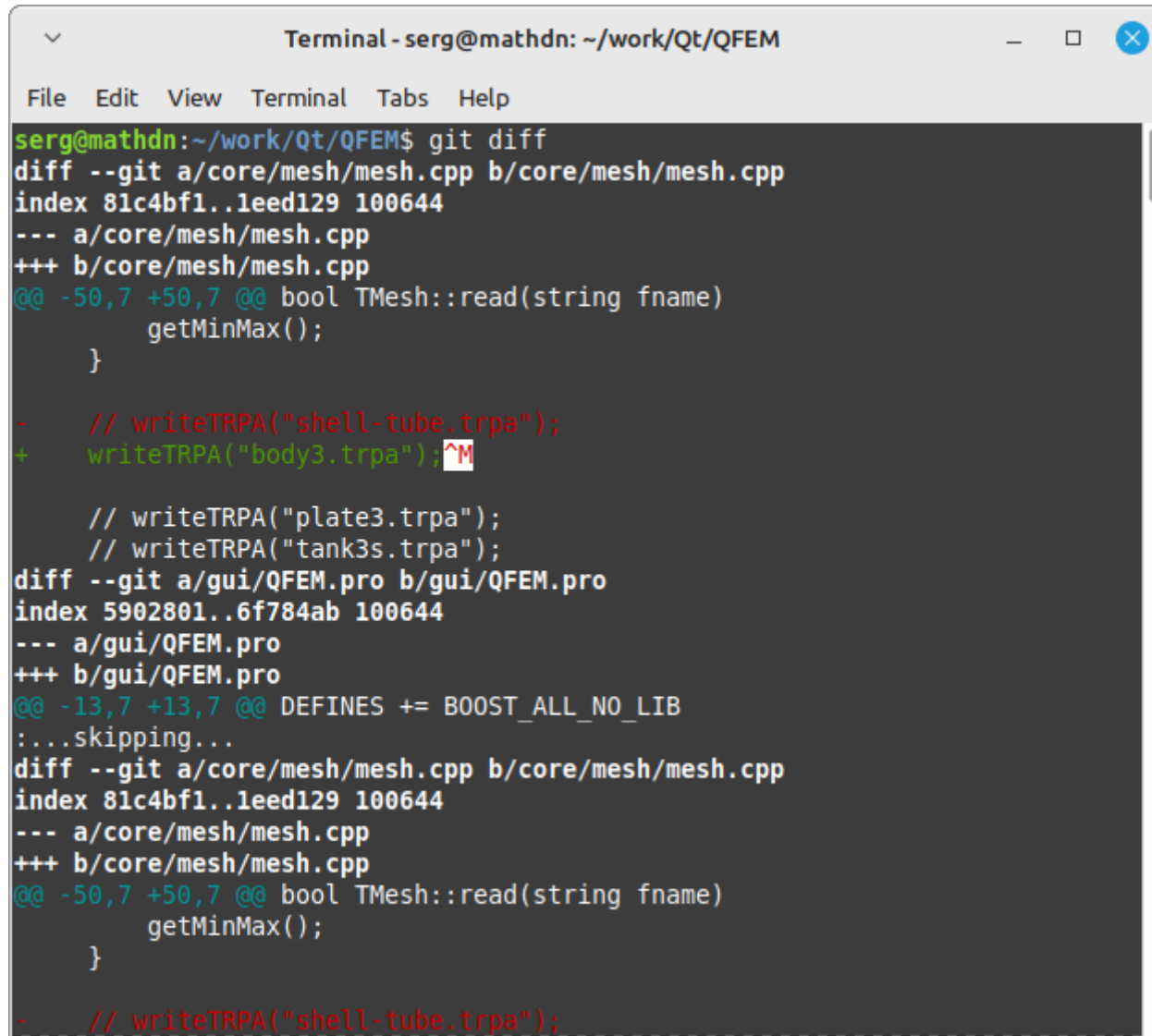
```
# Executables
```

```
*.exe
```

Примітка: на сайті <https://github.com/github/gitignore> можна переглянути приклади оформлення таких файлів.

3. Операції з репозиторіями

Для перегляду інформації про поточні зміни в репозиторії використовується команда **git diff**.

A terminal window titled "Terminal - serg@mathdn: ~/work/Qt/QFEM" showing the output of the command "git diff". The output displays changes between two versions of the repository. It shows a diff for "a/core/mesh/mesh.cpp" and "b/core/mesh/mesh.cpp", highlighting a new line of code: "writeTRPA(\"body3.trpa\");". It also shows a diff for "a/gui/QFEM.pro" and "b/gui/QFEM.pro", highlighting a change in the "DEFINES" section: "BOOST_ALL_NO_LIB". The terminal window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help".

```
serg@mathdn:~/work/Qt/QFEM$ git diff
diff --git a/core/mesh/mesh.cpp b/core/mesh/mesh.cpp
index 81c4bf1..1eed129 100644
--- a/core/mesh/mesh.cpp
+++ b/core/mesh/mesh.cpp
@@ -50,7 +50,7 @@ bool TMesh::read(string fname)
     getMinMax();
 }

- // writeTRPA("shell-tube.trpa");
+ writeTRPA("body3.trpa");

     // writeTRPA("plate3.trpa");
     // writeTRPA("tank3s.trpa");
diff --git a/gui/QFEM.pro b/gui/QFEM.pro
index 5902801..6f784ab 100644
--- a/gui/QFEM.pro
+++ b/gui/QFEM.pro
@@ -13,7 +13,7 @@ DEFINES += BOOST_ALL_NO_LIB
....skipping...
diff --git a/core/mesh/mesh.cpp b/core/mesh/mesh.cpp
index 81c4bf1..1eed129 100644
--- a/core/mesh/mesh.cpp
+++ b/core/mesh/mesh.cpp
@@ -50,7 +50,7 @@ bool TMesh::read(string fname)
     getMinMax();
 }

- // writeTRPA("shell-tube.trpa");
```

3. Операції з репозиторіями

Якщо розробник вважає, що індекс знаходиться у потрібному йому стані, він може зробити фіксацію всіх поточних коригувань.

Найпростіший спосіб створити коміт – виконати команду **git commit**. Ця команда у такому форматі автоматично відкриє текстовий редактор (Vim – за замовчанням), у якому користувач повинен написати обов'язковий коментар до поточної фіксації. Більш вживаним варіантом виконання цієї команди є використання відповідного параметру, який визначає коментар до поточної фіксації, наприклад: **git commit -m "Stage 100500: Fix some errors"**.

Для видалення файлу з індексу використовується команда **git rm -f <file_name>**, де параметр `file_name` визначає назву файлу, який слід видалити.

Для перейменування файлів у репозиторії використовується команда **git mv <old_file_name> <new_file_name>**.

Важливо: будь-які неіндексовані файли, тобто такі, для яких користувач не виконав команду **git add** після їх редагування, не потраплять до коміту.

4. Галуження проекту

Практично всі DVCS підтримують можливість створення окремих гілок (branches) поточного проекту.

Галуження – це відмежування від основної лінії розробки проекту для виконання своєї частини роботи та уникнення потенційних конфліктів з основною лінією. Особливістю Git є те, що з точки зору швидкодії, галуження тут є дуже швидкою на відміну від інших систем керування версіями операцією.

При створенні нового репозиторію після першого коміту автоматично формується головна гілка проекту, яка зазвичай має назву **master**. Для створення нової гілки поточного проекту використовується команда **git branch <branch_name>**, де **branch_name** – назва нової гілки.

Наприклад, команда **git branch testing** створює нову гілку проекту з ім'ям **testing**. Для переключення на потрібну гілку, слід виконати команду **git checkout <branch_name>**. Наприклад, для переключення на гілку **testing** слід використати команду **git checkout testing**.

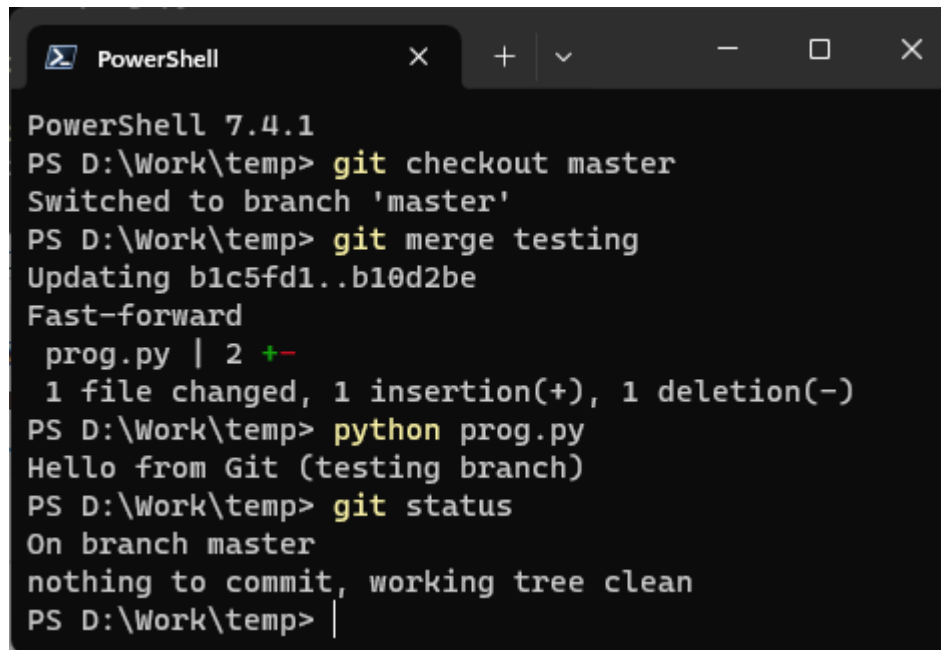
4. Галуження проекту

Приклад галуження

```
PowerShell 7.4.1
PS D:\Work\temp> echo "print('Hello from Git')" > prog.py
PS D:\Work\temp> git init
Initialized empty Git repository in D:/Work/temp/.git/
PS D:\Work\temp> git add .
PS D:\Work\temp> git commit -m "first commit"
[master (root-commit) b1c5fd1] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 prog.py
PS D:\Work\temp> git branch testing
PS D:\Work\temp> git checkout testing
Switched to branch 'testing'
PS D:\Work\temp> echo "print('Hello from Git (testing branch)')" > prog.py
PS D:\Work\temp> git add prog.py
PS D:\Work\temp> git commit -m "testing branch"
[testing b10d2be] testing branch
 1 file changed, 1 insertion(+), 1 deletion(-)
PS D:\Work\temp> git checkout master
Switched to branch 'master'
PS D:\Work\temp> python prog.py
Hello from Git
PS D:\Work\temp> git checkout testing
Switched to branch 'testing'
PS D:\Work\temp> python prog.py
Hello from Git (testing branch)
PS D:\Work\temp> |
```

4. Галуження проекту

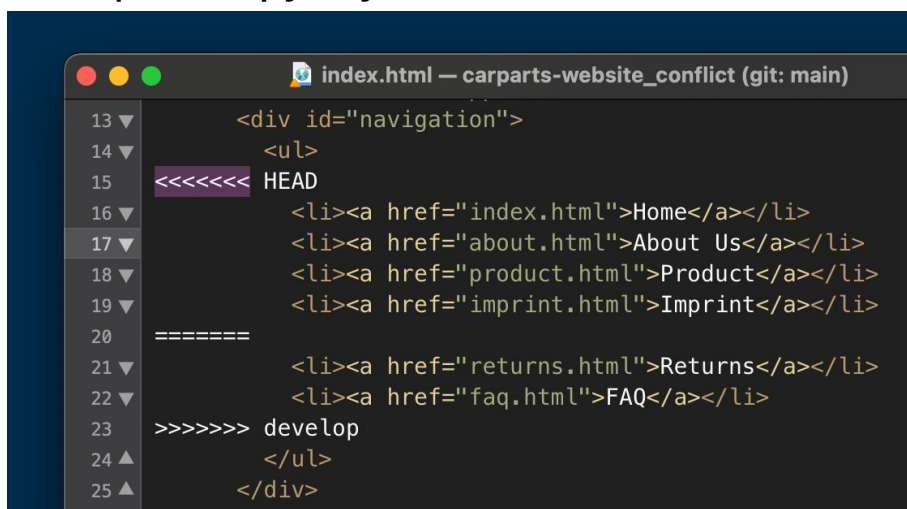
Якщо користувач вирішив об'єднати гілки проекту, він повинен скористуватися командою **git merge <branch_name>**. Для цього слід перейти у робочу гілку (з якою відбувається злиття) і виконати цю команду.



```
PowerShell 7.4.1
PS D:\Work\temp> git checkout master
Switched to branch 'master'
PS D:\Work\temp> git merge testing
Updating b1c5fd1..b10d2be
Fast-forward
 prog.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
PS D:\Work\temp> python prog.py
Hello from Git (testing branch)
PS D:\Work\temp> git status
On branch master
nothing to commit, working tree clean
PS D:\Work\temp> |
```

4. Галуження проекту

Доволі часто при злитті різних гілок можуть виникати проблеми. Якщо, наприклад, у двох гілках була змінена по різному одна й та ж сама частина певного файлу, що підлягає злиттю, то Git не зможе виконати цю операцію. Новий коміт не створюється, поки користувач не вирішить конфлікт вручну.



```
13 <div id="navigation">
14 <ul>
15 <<<<<< HEAD
16 <li><a href="index.html">Home</a></li>
17 <li><a href="about.html">About Us</a></li>
18 <li><a href="product.html">Product</a></li>
19 <li><a href="imprint.html">Imprint</a></li>
20 =====
21 <li><a href="returns.html">Returns</a></li>
22 <li><a href="faq.html">FAQ</a></li>
23 >>>>>> develop
24 </ul>
25 </div>
```

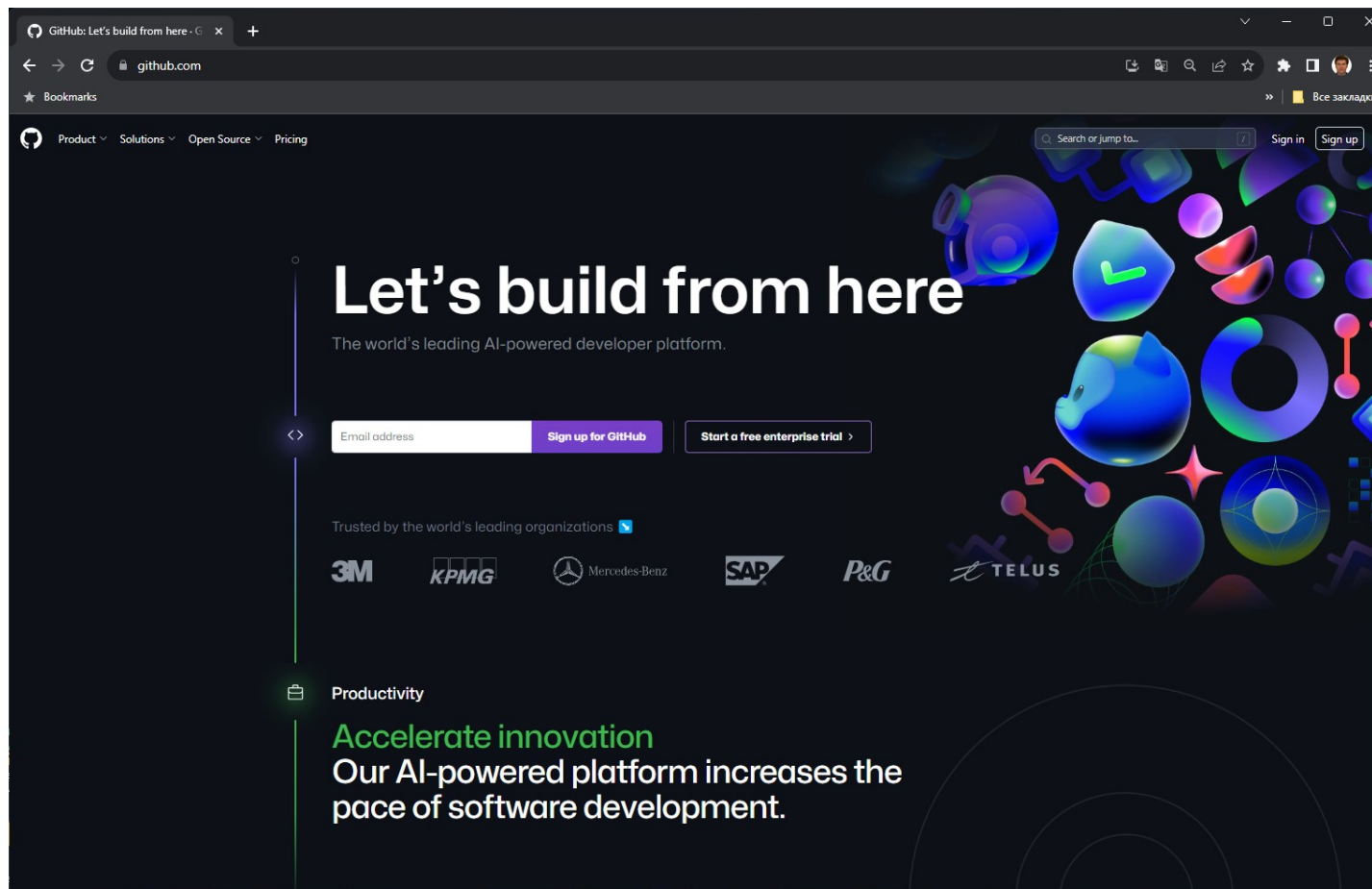
Подивитися, які конкретно файли не пройшли злиття, можна за допомогою команди **git status**.

Для вирішення конфлікту користувач повинен або вибрати одну з наявних у файлі змін, або якимось чином ці зміни об'єднати на власний розсуд.

Якщо деяка гілка вже непотрібна, її можна вилучити командою **git branch -d <branch_name>**. Для цього слід попередньо зробити поточною гілку master.

5. GitHub

Для організації співпраці колективів розробників використовуються віддалені сховища Git, які організовуються на деякому сервері. Всі розробники при цьому отримують спільний доступ до нього і можуть, як викладати в нього свої зміни, так і забирати корективи, внесені іншими програмістами. Найбільш відомим таким прикладом є **GitHub** (<https://github.com>) – найбільший хостинг для сховищ Git, який є центром співпраці великої кількості розробників та проектів по всьому світу.



5. GitHub

Приклад проекту, що зберігається на GitHub

The screenshot displays the GitHub repository page for **SeregaGomen/QFEM**. The repository is public and has 2 stars and 2 watchers. The repository structure is as follows:

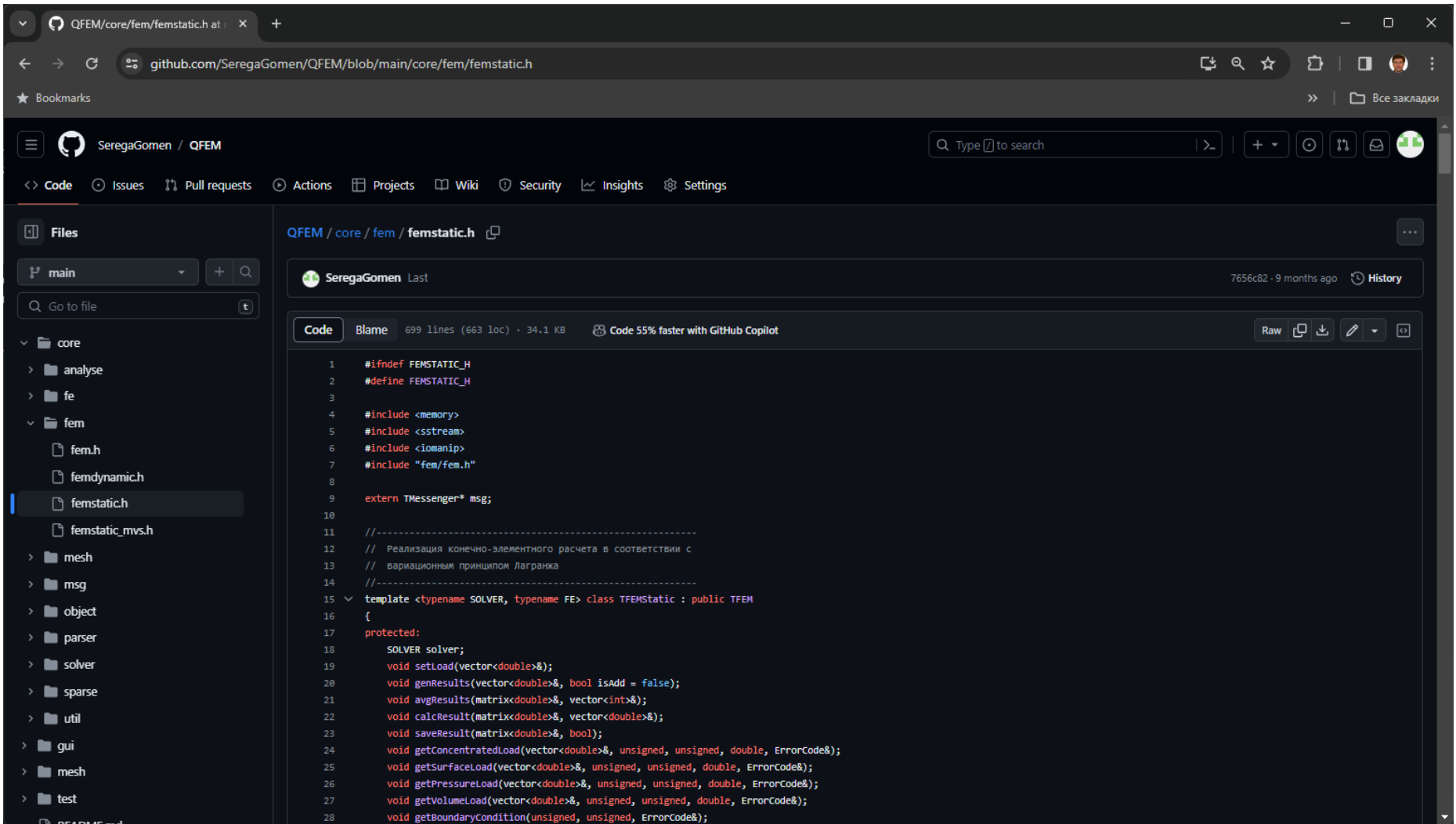
File/Folder	Last Commit	Time Ago
core	Last	2 months ago
gui	Last	2 months ago
mesh	Last	last year
test	Last	last year
README.md	Initial commit	2 years ago

The README file is titled **QFEM** and describes it as a **Simple FEM Solver**. The right sidebar shows the following statistics:

- About:** QFEM - Simple FEM Solver
- Readme:** Readme
- Activity:** Activity
- Stars:** 2 stars
- Watching:** 2 watching
- Forks:** 0 forks
- Releases:** No releases published. [Create a new release](#)
- Packages:** No packages published. [Publish your first package](#)
- Languages:** C++ 92.7%, GLSL 6.6%, QMake 0.7%

5. GitHub

Перегляд початкового коду на GitHub



The screenshot shows a web browser displaying the GitHub repository for SeregaGomen/QFEM. The page is titled "QFEM / core / fem / femstatic.h". The left sidebar shows the file structure, with "femstatic.h" selected. The main content area displays the source code for "femstatic.h", which is a C++ header file. The code includes preprocessor directives for conditional compilation, standard library headers, and a namespace definition. It also contains a template class definition for "TFEMStatic".

```
1  #ifndef FEMSTATIC_H
2  #define FEMSTATIC_H
3
4  #include <memory>
5  #include <sstream>
6  #include <iomanip>
7  #include "fem/fem.h"
8
9  extern TMessenger* msg;
10
11  //-----
12  // Реализация конечно-элементного расчета в соответствии с
13  // вариационным принципом Лагранжа
14  //-----
15  template <typename SOLVER, typename FE> class TFEMStatic : public TFEM
16  {
17  protected:
18      SOLVER solver;
19      void setLoad(vector<double>&);
20      void genResults(vector<double>&, bool isAdd = false);
21      void avgResults(matrix<double>&, vector<int>&);
22      void calcResult(matrix<double>&, vector<double>&);
23      void saveResult(matrix<double>&, bool);
24      void getConcentratedLoad(vector<double>&, unsigned, unsigned, double, ErrorCode&);
25      void getSurfaceload(vector<double>&, unsigned, unsigned, double, ErrorCode&);
26      void getPressureLoad(vector<double>&, unsigned, unsigned, double, ErrorCode&);
27      void getVolumeLoad(vector<double>&, unsigned, unsigned, double, ErrorCode&);
28      void getBoundaryCondition(unsigned, unsigned, ErrorCode&);
```