



Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №1
із дисципліни *«Технології розроблення програмного забезпечення»*
Команди Git

Виконала
студентка групи ІА–34
Кузьменко Вероніка Сергіївна

Перевірив:
Мягкий М.Ю.

ЗМІСТ

| | |
|----------------------------|----|
| ТЕОРЕТИЧНІ ВІДОМОСТІ | 3 |
| ХІД РОБОТИ..... | 14 |
| ВИСНОВОК | 16 |

Тема: Системи контролю версій. Розподілена система контролю версій «Git».

Мета: Навчитися виконувати основні операції в роботі з децентралізованими системами контролю версій на прикладі роботи з сучасною системою Git.

Теоретичні відомості

Система управління версіями (від англ. Version Control System або Source Control System) – програмне забезпечення яке призначено допомогти команді розробників керувати змінами в вихідному коді під час роботи. Система керування версіями дозволяє додавати зміни в файлах в репозиторій і таким чином після кожної фіксації змін мانی нову ревізію файлів. Це дозволяє повертатися до попередніх версій коду для аналізу внесених змін або пошуку, які зміни привели до появи помилки. Таким чином можна знайти хто, коли і які зміни зробив в коді, а також чому ці зміни були зроблені.

Такі системи найбільш широко використовуються при розробці програмного забезпечення для зберігання вихідних кодів програми, що розробляється. Однак вони можуть з успіхом застосовуватися і в інших областях, в яких ведеться робота з великою кількістю електронних документів, що безперервно змінюються. Зокрема, системи керування версіями застосовуються у САПР, зазвичай у складі систем керування даними про виріб (PDM). Керування версіями використовується у інструментах конфігураційного керування (Software Configuration Management Tools).

Основні типи систем контролю версій:

Локальні системи контролю версій – найпростіший тип, який зберігає зміни у вигляді копій файлів на одному комп'ютері. Це базовий підхід, який підходить лише для індивідуальних проєктів.

Централізовані системи контролю версій (CVCS) – усі версії файлів зберігаються на центральному сервері, а розробники отримують доступ до останньої версії файлів, з якою можуть працювати. Однак, якщо сервер недоступний, то й працювати з файлами складніше. Приклад такої системи: SVN (Subversion).

Розподілені системи контролю версій (DVCS) – такі системи, як Git та Mercurial, дозволяють кожному учаснику зберігати повну копію всього проєкту, включаючи історію змін. Користувачі можуть працювати локально, і лише потім синхронізувати свої зміни з іншими. Це забезпечує вищу надійність, швидкість і гнучкість.

Переваги систем контролю версій:

Збереження історії змін – можна повернутися до будь-якої попередньої версії файлу.

Можливість командної роботи – декілька людей можуть працювати над одним проєктом одночасно, зменшуючи ймовірність конфліктів у коді.

Відновлення після помилок – якщо виникла проблема, можна легко повернутися до попередньої стабільної версії.

Тестування та експериментування – можна створювати окремі гілки для тестування нових функцій, не впливаючи на основний проєкт.

Система контролю версій є важливою частиною сучасного розробницького процесу, оскільки вона дозволяє ефективно керувати змінами, співпрацювати у команді та уникати втрат інформації.

Git – це система контролю версій (VCS, Version Control System), яка дозволяє відслідковувати зміни у файлах, керувати кодом і спільно працювати над проєктами. Git широко використовується у розробці програмного забезпечення для управління кодовою базою, оскільки забезпечує збереження історії змін та дозволяє кільком розробникам працювати над проєктом одночасно.

Робота з Git

Робота з Git може виконуватися з командного рядка, а також за допомогою візуальних оболонок. Командний рядок використовується програмістами, як можливість виконання всіх доступних команд, а також можливості складання складних макросів. Візуальні оболонки як правило дають більш наглядне представлення репозиторію у вигляді дерева, та більш зручний спосіб роботи з

репозиторієм, але, дуже часто доступний не весь набір команд Git, а лише саме ті, що найчастіше використовуються.

Прикладами візуальних оболонок для роботи з Git є Git Extension, SourceTree, GitKraken, GitHub Desktop та інші.

Основна ідея Git, як і будь-якої іншої розподіленої системи контролю версій – кожен розробник має власний репозиторій, куди складаються зміни (версії) файлів, та синхронізація між розробниками виконується за допомогою синхронізації репозиторіїв.

Відповідно, є ряд основних команд для роботи:

1. Клонувати репозиторій (`git clone`) – отримати копію репозиторію на локальну машину для подальшої роботи з ним;
2. Синхронізація репозиторіїв (`git fetch` або `git pull`) – отримання змін із віддаленого (вихідного, центрального, або будь-якого іншого такого ж) репозиторію;
3. Фіксація змін в репозиторій (`git commit`) – фіксація виконаних змін в програмному коді в локальний репозиторій розробника;
4. Синхронізація репозиторіїв (`git push`) – переслати зміни – `push` – передача власних змін до віддаленого репозиторію – Записати зміни – `commit` – створення нової версії;
5. Оновитись до версії – `update` – оновитись до певної версії, що є у репозиторії.
6. Об'єднання гілок (`git merge`) – об'єднання вказаною гілки в поточну (часто ще називається «злиттям»).

Таким чином, якщо розглядати основний робочий процес програміста в команді, то він виглядає наступним чином: На початку роботи з проєктом виконується клонування, після цього, в рамках виконання поставленої задачі, створюється бранч і всі зміни в коді, зроблені в рамках цієї задачі фіксуються в репозиторії (періодично виконується синхронізація з основним репозиторієм). Далі, коли задача виконана, то виконується об'єднання гілки з основною гілкою і фінальна синхронізація з центральним репозиторієм.

Основні особливості Git:

Відслідковування версій: Git дозволяє відновити попередні версії файлів або весь проєкт, якщо потрібно повернутися до стабільного стану.

Розгалуження та злиття: Git дозволяє створювати різні гілки для роботи над новими функціями або виправленням помилок, що не впливає на основний код. Потім зміни можна об'єднати з основною гілкою.

Розподілена структура: Кожен розробник зберігає повну копію проєкту на своєму комп'ютері, що дозволяє працювати локально і зберігати історію змін.

Висока швидкість: Завдяки локальному збереженню проєкту Git швидко виконує операції з файлами.

git init

Команда `git init` створює новий порожній Git-репозиторій у поточній папці, що дозволяє відслідковувати зміни у файлах проєкту. Вона створює приховану папку `.git`, де зберігатимуться метадані та історія проєкту.

git add

Команда `git add` додає файли у відстежуваний (staging) стан, готуючи їх до коміту. Це означає, що ви вибираєте, які файли або зміни будуть збережені у наступному коміті.

Приклади використання:

Додати конкретний файл: `git add <filename>`

Додати всі файли в поточній папці: `git add .`

git commit

Команда `git commit` зберігає зміни, що знаходяться в стадії, у репозиторії, створюючи новий знімок проєкту. Після коміту можна повернутися до цієї версії у будь-який момент.

Приклади використання:

`git commit -m "Опис змін"`

git status

`git status` відображає стан файлів у репозиторії: показує, які файли додані до стадійованих, а які - змінені та не збережені. Це корисно для відстеження, які файли готові до коміту, а які ще потрібно підготувати.

git branch

`git branch` керує гілками у репозиторії, дозволяючи створювати нові гілки або видаляти існуючі. Гілки допомагають розділяти роботу над різними функціями або виправленнями, щоб основний код залишався стабільним.

Приклади використання:

Перегляд всіх гілок: `git branch`

Створення нової гілки: `git branch <branch-name>`

git merge

`git merge` об'єднує зміни з різних гілок, об'єднуючи поточну гілку з іншою. Ця команда зазвичай використовується після роботи в окремій гілці для об'єднання її з основною.

Приклад використання:

`git merge <branch-name>`

git push

Команда `git push` завантажує локальні коміти на віддалений сервер (наприклад, GitHub або GitLab), щоб зробити їх доступними для інших учасників команди.

Приклад використання:

`git push origin <branch-name>`

git pull

Команда `git pull` завантажує зміни з віддаленого репозиторію на локальний комп'ютер і автоматично об'єднує їх з локальною гілкою.

Приклад використання:

```
git pull origin <branch-name>
```

git log

Показує історію комітів у репозиторії.

git checkout

Команда `git checkout` використовується для перемикання між гілками та відновлення файлів у їхній попередній стан. Вона дозволяє перейти на іншу гілку, відновити файл з історії комітів, або повернутися до певного коміту.

Основні способи використання `git checkout`:

Перемикання між гілками: Щоб перейти на іншу гілку, використовуйте:

```
git checkout <branch-name>
```

Це перемкне поточну робочу директорію на гілку `<branch-name>`, щоб ви могли працювати з її версією файлів.

Створення та перемикання на нову гілку: Можна створити нову гілку та одразу перейти на неї, додавши прапорець `-b`:

```
git checkout -b <new-branch-name>
```

Це створить нову гілку `<new-branch-name>` і автоматично перемкне вас на неї.

Відновлення конкретного файлу з попереднього коміту: Якщо ви хочете відновити файл до його версії з певного коміту, можна використати:

```
git checkout <commit-hash> -- <filename>
```

Це відновить файл `<filename>` до стану, у якому він був на момент коміту з указаним `<commit-hash>`.

Перехід на певний коміт у режимі detached HEAD: Використовуючи `checkout`, можна перемістити HEAD на конкретний коміт, щоб переглянути його зміст (наприклад, для тестування):

```
git checkout <commit-hash>
```


Це переключить вас на коміт із зазначеним <commit-hash> у режимі detached HEAD, що означає, що ви не перебуваєте на жодній гілці, і всі зміни, які ви зробите, не будуть збережені в жодній гілці, доки не створите нову.

Команда `git checkout` була частково замінена на команду `git switch` для зміни гілок і `git restore` для відновлення файлів, що робить її застосування трохи менш необхідним у нових версіях Git.

git clone

`git clone` створює копію існуючого віддаленого репозиторію на вашому комп'ютері. Вона завантажує весь репозиторій, включаючи історію комітів, файли, гілки тощо.

Приклад використання:

```
git clone <repository-url>
```

git reset

Команда `git reset` повертає зміни в історії, видаляючи коміти або знімаючи файли зі стадії. Вона має різні режими (`--soft`, `--mixed`, `--hard`), які змінюють дію команди.

Приклади використання:

Зняти файл зі стадії: `git reset <filename>`

Повернути останній коміт (зберігши зміни): `git reset --soft HEAD~1`

git stash

`git stash` зберігає тимчасові зміни, які ще не готові до коміту, і дозволяє повернутися до чистого робочого стану. Це зручно, коли потрібно переключитися на іншу гілку, але ви ще не закінчили роботу над поточною.

Приклади використання:

Зберегти зміни: `git stash`

Повернути зміни з stash: `git stash pop`

git fetch

Команда `git fetch` завантажує оновлення з віддаленого репозиторію, не об'єднуючи їх з локальною гілкою. Це дозволяє переглянути зміни, які були додані до віддаленого репозиторію, перед об'єднанням їх з локальною роботою.

Приклад використання:

```
git fetch origin
```

git rebase

`git rebase` змінює історію комітів, переміщаючи поточну гілку на вершину іншої гілки. Ця команда корисна для збереження "чистої" історії комітів, коли потрібно об'єднати зміни з основною гілкою без створення додаткових комітів злиття.

Приклад використання:

```
git rebase <branch-name>
```

Типи Rebase

- **Інтерактивний rebase:**

Дозволяє редагувати, об'єднувати, змінювати порядок або видаляти коміти під час rebase. Використовується з прапором `-i`.

```
git rebase -i <commit_hash>
```

При виконанні цієї команди відкриється редактор, у якому можна вказати дії для кожного коміту (наприклад, `pick`, `squash`, `edit`).

- **Автоматичний rebase при pull:**

Виконує rebase замість merge при отриманні змін з віддаленого репозиторію.

```
git pull --rebase
```

git cherry-pick

git cherry-pick дозволяє вибірково взяти один або декілька комітів з іншої гілки і застосувати їх до поточної гілки. Це корисно, коли потрібно перенести конкретні зміни з однієї гілки в іншу.

Приклад використання:

```
git cherry-pick <commit-hash>
```

git revert

На відміну від git reset, команда git revert створює новий коміт, який скасовує зміни, зроблені в зазначеному коміті. Це корисно, коли потрібно скасувати зміни в історії, не видаляючи коміт.

Приклад використання:

```
git revert <commit-hash>
```

git tag

Команда git tag створює тег для певного коміту, що зазвичай використовується для позначення певних версій (наприклад, v1.0, v2.0). Теги корисні для позначення релізів або інших важливих етапів проєкту.

Приклади використання:

Створення легкого тега: git tag <tag-name>

Створення анотованого тега (із повідомленням): git tag -a <tag-name> -m "Опис тега"

Перегляд усіх тегів: git tag

git diff

git diff показує відмінності між різними версіями файлів, наприклад, між змінами в робочій директорії та стадійованими змінами або між комітами.

Приклади використання:

Показати зміни в робочій директорії: git diff

Показати зміни між стадійованими та закоміченими файлами: git diff --staged

Показати зміни між двома комітами: `git diff <commit1-hash> <commit2-hash>`

git archive

Команда `git archive` створює ZIP або TAR архів із вмістом конкретної гілки або коміту. Це корисно, коли потрібно створити архів поточної версії проєкту.

Приклад використання:

```
git archive --format=zip --output=<filename.zip> <branch-name>
```

git reset

Ця команда може мати розширені опції, які використовуються для більш глибокого скасування змін:

- `--soft` — повертає коміт, залишаючи зміни на стадії.
- `--mixed` — знімає зміни зі стадії, але зберігає їх у робочій директорії.
- `--hard` — видаляє зміни як з коміту, так і з робочої директорії, скасовуючи їх повністю.

Приклад використання:

```
git reset --hard HEAD~1
```

git rm

`git rm` видаляє файли з робочої директорії та з індексу Git, готуючи їх до видалення у наступному коміті.

Приклад використання:

```
git rm <filename>
```

git ls-files

Ця команда показує список файлів, які відстежуються Git-ом. Можна використовувати для перевірки стану індексу.

Приклад використання:

```
git ls-files
```

git show

Команда `git show` показує деталі коміту (зміни, автора, дату) або об'єкта (наприклад, тегу). Це корисно для перегляду інформації про конкретний коміт.

Приклад використання:

```
git show <commit-hash>
```

git blame

`git blame` показує, хто і коли зробив певні зміни в рядках файлу. Ця команда корисна для відстеження історії правок по рядках.

Приклад використання:

```
git blame <filename>
```

git log --graph --oneline --all

Ця команда поєднує кілька параметрів команди `git log`, щоб вивести наочну графічну історію гілок у короткому форматі. Вона корисна для огляду всіх гілок і комітів у зручному для читання вигляді.

Приклад використання:

```
git log --graph --oneline --all
```

git clean

Команда `git clean` видаляє неіндексовані файли або директорії з робочої директорії (тобто файли, які не відстежуються Git-ом).

Приклад використання:

```
git clean -f
```

Прапорець `-f` означає "force" і обов'язковий для підтвердження видалення.

git config

Команда `git config` налаштовує параметри Git. Вона використовується для встановлення вашого імені, електронної пошти та інших налаштувань, які застосовуються локально або глобально.

Приклади використання:

Встановити ім'я користувача: `git config --global user.name "Your Name"`

Встановити електронну пошту: `git config --global user.email "you@example.com"`

Перегляд усіх налаштувань: `git config --list`

git bisect

Команда `git bisect` використовується для бінарного пошуку коміту, який ввів помилку в проєкт. Замість перевірки кожного коміту вручну, `git bisect` дозволяє швидко знайти проблемний коміт, особливо у великій історії комітів. Це дуже зручно, коли виникає проблема, але ви не знаєте, коли вона з'явилася.

Як працює `git bisect`

1. Git ділить історію комітів навпіл і перевіряє коміти в середній точці.
2. Ви відзначаєте, чи є помилка на цій точці чи ні.
3. Git продовжує ділити коміти навпіл на основі ваших відповідей.
4. Процес триває, поки не буде знайдено перший коміт, де з'явилася проблема.

Хід роботи

1. Створено локальний репозиторій 'trpz-lab1'

```
[veronikakuzmenko@Noutbuk-Veronika ~ % git init trpz-lab1
підказка: Використання "master" як назви початкової гілки. Назву гілки за за
мовчуванням
підказка: може бути змінено. Щоб налаштувати назву початкової гілки для вико
ристання у всіх
підказка: ваших нових сховищах, що приховає це попередження, викличте
підказка:
підказка:      git config --global init.defaultBranch <назва>.
підказка:
підказка: Замість "master" зазвичай використовують такі назви, як "main", "t
runk" та
підказка: "development". Щойно створену гілку можна перейменувати за допомог
ою цієї команди:
підказка:
підказка:      git branch -m <назва>
Ініціалізовано порожнє Git сховище в /Users/veronikakuzmenko/trpz-lab1/.git/
```

2. Додано та зафіксовано файл 'hello.txt' з текстом 'test'

```
veronikakuzmenko@Noutbuk-Veronika ~ % cd trpz-lab1
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % echo "test" > hello.txt
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git add hello.txt
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git commit -m "Hello added"
[master (кореневий коміт) 7d15021] Hello added
1 file changed, 1 insertion(+)
create mode 100644 hello.txt
```

3. Додано нову директорію 'child_dir' з файлом 'inner.txt'

```
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % mkdir child_dir
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % cd child_dir
veronikakuzmenko@Noutbuk-Veronika child_dir % echo "inner" > inner.txt
```

4. Зафіксовано додавання директорії з файлом

```
veronikakuzmenko@Noutbuk-Veronika child_dir % git commit -m "Add child directory with content"
[master 02526c6] Add child directory with content
1 file changed, 1 insertion(+)
create mode 100644 child_dir/inner.txt
```

5. Створено та перейдено на гілку 'feature_branch'

```
veronikakuzmenko@Noutbuk-Veronika child_dir % git branch feature_branch
veronikakuzmenko@Noutbuk-Veronika child_dir % git checkout feature_branch
Переключено на гілку "feature_branch"
```

6. Видалено додану директорію та зафіксовано зміни

```
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % rm -r child_dir
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git add .
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git commit -m "Remove child_dir"
[feature_branch a46c02d] Remove child_dir
1 file changed, 1 deletion(-)
delete mode 100644 child_dir/inner.txt
```

7. Злило зміни з основною гілкою

```
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git checkout master
Переключено на гілку "master"
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git merge feature_branch
Оновлення 02526c6..a46c02d
Fast-forward
 child_dir/inner.txt | 1 -
1 file changed, 1 deletion(-)
delete mode 100644 child_dir/inner.txt
veronikakuzmenko@Noutbuk-Veronika trpz-lab1 %
```

8. Виведено історію на екран

```
[veronikakuzmenko@Noutbuk-Veronika trpz-lab1 % git log  
commit a46c02d42f6b34f9277a8b73830103b0c19bedd6 (HEAD -> master, feature_branch)  
Author: nectakin <kvs171005@gmail.com>  
Date: Thu Sep 11 15:37:48 2025 +0300  
  
    Remove child_dir  
  
commit 02526c63469371853d84f3dcafb7de0c421b457d  
Author: nectakin <kvs171005@gmail.com>  
Date: Thu Sep 11 15:31:10 2025 +0300  
  
    Add child directory with content  
  
commit 7d1502199e9c709634869841ba40681fe5812041  
Author: nectakin <kvs171005@gmail.com>  
Date: Thu Sep 11 15:27:29 2025 +0300  
  
    Hello added
```

Висновок

У процесі виконання даної лабораторної роботи я ознайомилася з основними принципами систем контролю версій та на практиці попрацювала з розподіленою системою Git. Було створено локальний репозиторій, додано файли під контроль версій, виконано коміти, роботу з гілками та відправку змін на GitHub. Таким чином, мети роботи досягнуто - я здобула практичні навички роботи з Git, які є необхідними для ефективної організації власних проєктів і командної розробки.