Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота № 1

з дисципліни «Технології розроблення програмного забезпечення»

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

Виконав:
студент групи - ІА-32
Воробйов Кирило Андрійович

Перевірив:
Мягкий Михайло Юрійович

Київ 2025

Тема: Системи контроля 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.

Зміст

Теоретичні відомості

1.2.1. Призначення систем управління версіями

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 [1]. Система керування версіями дозволяє додавати зміни в файлах в репозиторій і таким чином після кожної фіксації змін мани 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).

1.2.2. Історія розвитку систем контролю версій

Умовно, розвиток систем контролю версій можна розбити на наступні етапи: ранній етап, етап централізованих систем, етап децентралізації та етап хмарних платформ.

Ранній етап

На цьому етапі основна увага приділялася роботі з окремими файлами у локальному середовищі.

Найпершою системою контролю версій була система «скопіювати і вставити», коли більшість проєктів просто копіювалася з місця на місце зі зміною назва (проєкт_1; проєкт_новий; проєкт_найновіший і т.д.), як правило у вигляді zip архіву або подібних (arj, tar). Звичайно, такі маніпуляції над файловою системою навряд чи можна назвати хоч скільки повноцінною системою контролю версій (або системою взагалі). Для вирішення цих проблем 1982 року з'являється RCS.

RCS

Однією з основних нововведень RCS було використання дельт для зберігання змін (тобто зберігаються ті рядки, які змінилися, а не весь файл). Однак він мав низку недоліків.

Насамперед він був тільки для текстових файлів. Не було центрального репозиторію; кожен версіонований файл мав власний

репозиторій як rcs файлу поруч із самим файлом. Тобто якщо на проєкті було 100 файлів, поруч лягало
100 rcs файлів. У кращому випадку ці 100 файлів утворювалися в директорії RCS (при правильному налаштуванні). Найменування версій і гілок було неможливим.

Етап централізованих систем

На початку 90-х почалася епоха централізованих систем контролю версій. У цей період розробники почали переходити до централізованих систем, що дозволяли працювати кільком користувачам одночасно через сервер

Одина із перших найпопулярніших систем (і досі використовувана) система контролю версій – CVS. Цю епоху можна охарактеризувати досить сформованим уявленням про системи контролю версій, їх можливості, появою центральних репозиторіїв (та синхронізації дій команди).

SVN

SVN – у порівнянні з CVS це був наступний крок. Надійна та швидкодіюча систему контролю версій, яка зараз розробляється в рамках проєкту Apache Software Foundation. Вона реалізована за технологією клієнт-сервер та відрізняється неймовірною простотою – дві кнопки (commit, update). Порівняно з CVS, це удосконалена централізована система з кращим управлінням комітами та резервними копіями.

Незважаючи на це, SVN дуже погано вміє створювати та зливати гілки та погано вирішує конфліктні ситуації з версіями. Але, в багатьох проєктах до цих пір використовується SVN.

Етап децентралізації

Децентралізовані системи усунули залежність від центрального сервера та дозволили кожному розробнику мати повну копію репозиторію.

У 1992 році з'явився один з основних представників світу систем розподіленого контролю версій. ClearCase був однозначно попереду свого часу і для багатьох він досі є однією з найпотужніших систем контролю версій будьколи створених.

Дана система дозволяла користуватися віртуальною файловою системою для зберігання та отримання змін; мала широкий діапазон повноважень щодо зміни, впровадження у процес розробки (аудит збірок товару, версії, зливання змін, динамічні уявлення); запускалася на безлічі різних систем.

У 2005 році було створено дві знакові системи контролю версій Git та Mercurial. Вони стали революційними системами, які забезпечили швидкість, надійність і гнучкість роботи. Вони мають багато ідентичних команд, хоча «під капотом» вони мають різні підходи до реалізації. Досить довго вони конкурували одна з одною, але починаючи з 2018 Git поступово виходить на лідерську позицію серед безкоштовних систем контролю версій.

Git

Лінус Торвальдс, т.зв. Батько Лінукса, розробив і впровадив першу версію Гіт для надання можливості розробникам ядра Лінукс проводити контроль версій не тільки в BitKeeper.

Гіт є системою розподіленого контролю версій, коли кожен розробник має власний репозиторій, куди він вносить зміни [2]. Далі система гіт синхронізує репозиторії із центральним репозиторієм. Це

дозволяє проводити роботу незалежно від центрального репозиторію (на відміну від SVN, коли версіонування передбачало наявність зв'язку з центральним сервером), перекладає складності ведення гілок та склеювання змін більше на плечі системи, ніж розробників та ін.

Зміни зберігаються у вигляді наборів змін (changeset), що отримує унікальний ідентифікатор (хеш-сума на основі самих змін).

Mercurial

Mercurial був створений як і Git після оголошення про те, що BitKeeper більше не буде безкоштовним для всіх. Багато в чому схожий на Git, Mercurial також використовує ідею наборів змін, але на відміну від Git, зберігає їх у не у вигляді вузла в графі, а вигляді плоского набору файлів і папок, званих revlog.

Етап хмарних платформ

Приблизно з 2010 року і до цих пір також можна виділити етап хмарних платформ, основним лозунгом яких є «Інтеграція та автоматизація».

У сучасну епоху акцент робиться на інтеграції систем контролю версій із хмарними платформами та автоматизації розробки. І в більшості випадків такою системою контролю версій є Git.

Можна виділити такі ключові хмарні платформи на основі Git: GitHub, GitLab, Bitbucket. Вони підтримують CI/CD, спільну роботу та інтеграції, інструменти для DevOps, аналітики та автоматичного тестування.

Таким чином, основною характеристикою цього етапу є інтеграція систем контролю версій в хмарні сервіси для глобальної співпраці, які

додатково підтримують розширену функціональність для автоматизації процесів та інтеграції з іншими сервісами.

1.2.3. 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 контроля версій – кожен розробник має власний репозиторій, куди складаються зміни (версії) файлів, та синхронізація між розробниками виконується за допомогою синхронізації репозиторіїв. Процес роботи виглядає так, як зображено на рисунку 1.1.

Відповідно, є ряд основних команд для роботи [2]:

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;

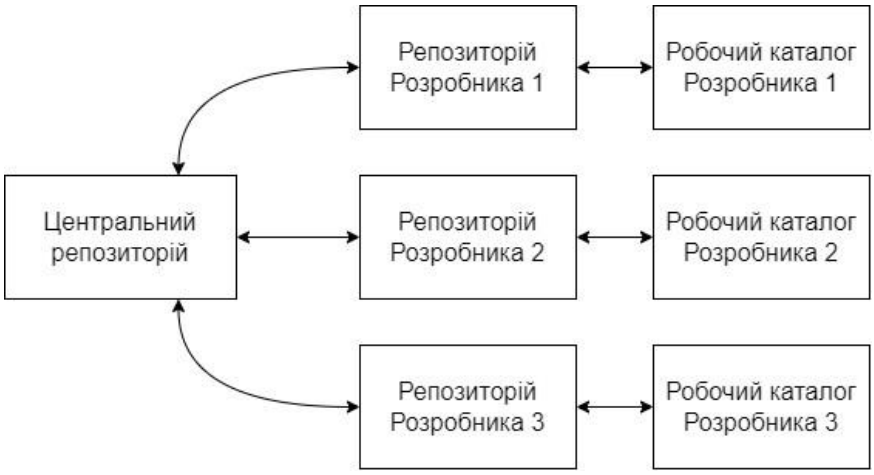


Рисунок 1.1. Схема процесу роботи з Git

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 репозиторіем). Далі, коли задача

виконана, то виконується об'єднання гілки з основною гілкою і фінальна синхронізація з центральним репозиторієм.

1.2.4 Додаткові матеріали для самостійного опрацювання

1. https://learngitbranching.js.org/ – інтерактивний посібник по командах гіт.

Можете пройти кілька перших уроків, чого може бути достатньо для опанування базових команд, але по бажанню можна і все пройти.

2. https://www.codecademy.com/learn/learn-git – ще один інтерактивний курс
3. https://git-scm.com/doc – документація
4. Допомога від самого Git, наприклад, команда в консолі:

```
git help branch
```

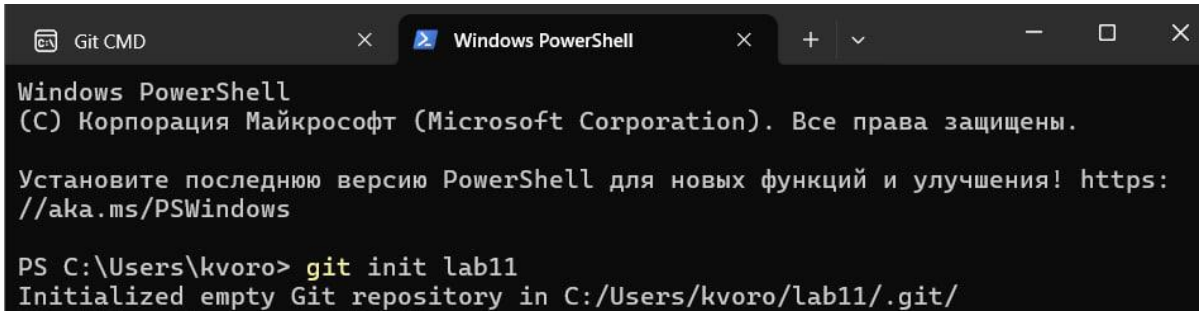
або:

```
git branch -h
```

Хід роботи

1.3. Виконання завдання

1. Створити локальний репозиторій.



2. Додати довільний файл з довільним текстом (в даному випадку текст – test).

3. Зафіксувати додавання файлу.

```
PS C:\Users\kvoro\lab11> git add hello.txt
PS C:\Users\kvoro\lab11> git commit -m "hello added"
[master (root-commit) e7a5fb1] hello added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 hello.txt
```

4. Додати нову директорію з довільним ім'ям.

```
PS C:\Users\kvoro\lab11> mkdir child_dir


    Каталог: C:\Users\kvoro\lab11


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        12.09.2025     21:25                child_dir
```

5. Додати файл у директорію.

```
PS C:\Users\kvoro\lab11> cd child_dir
PS C:\Users\kvoro\lab11\child_dir> echo "inner" > inner.txt
```

6. Зафіксувати додавання директорії із файлом.

```
PS C:\Users\kvoro\lab11\child_dir> cd ..
PS C:\Users\kvoro\lab11> git add child_dir
PS C:\Users\kvoro\lab11> git commit -m "add child diretory with content"
[master 3d79c41] add child diretory with content
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 child_dir/inner.txt
```

7. Створити гілку і перейти на неї.

```
PS C:\Users\kvoro\lab11> git branch feature_branch
PS C:\Users\kvoro\lab11> git checkout feature_branch
Switched to branch 'feature_branch'
PS C:\Users\kvoro\lab11> rmdir child_dir -r
```

8. Видалити додану директорію і зафіксувати зміни.

```
PS C:\Users\kvoro\lab11> rmdir child_dir -r
PS C:\Users\kvoro\lab11> git add .
PS C:\Users\kvoro\lab11> git commit -m "remove child-dir"
[feature_branch f87d155] remove child-dir
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 child_dir/inner.txt
```

9. Злити зміни з основною гілкою.

```
PS C:\Users\kvoro\lab11> git checkout master
Switched to branch 'master'
PS C:\Users\kvoro\lab11> git merge feature_branch
Updating 3d79c41..f87d155
Fast-forward
 child_dir/inner.txt | Bin 16 -> 0 bytes
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 child_dir/inner.txt
```

10. Вивести історію на екран.

```
PS C:\Users\kvoro\lab11> git log
commit f87d1557e3962c6a60cc02a67ecf016c6d99fad9 (HEAD -> master, feature_bra
nch)
Author: KyryloVorobiovIA-32 <kvorobiov17@gmail.com>
Date:   Fri Sep 12 21:28:53 2025 +0200

    remove child-dir

commit 3d79c41c88b112ee52a521cc55381e2f58df8b6d
Author: KyryloVorobiovIA-32 <kvorobiov17@gmail.com>
Date:   Fri Sep 12 21:27:06 2025 +0200

    add child diretory with content

commit e7a5fb1fc020d8adb333f1d836d5a6774bfd7ee3
Author: KyryloVorobiovIA-32 <kvorobiov17@gmail.com>
Date:   Fri Sep 12 21:24:46 2025 +0200

    hello added
PS C:\Users\kvoro\lab11>
```

Отже, кінцевим результатом лабораторної роботи повинні стати:

1. Створений локальний репозиторій.
2. Створений і закомічений в репозиторій файл.
3. Створена в проєкті директорія.
4. В репозиторії існує дві гілки.
5. Гілка зі змінами злита з основною гілкою

Висновки

Висновки: Під час лабораторної роботи ми навчилися виконувати основні операції в системі контролю версій Git, працювати з гілками, комітами та злиттям змін.

Питання до лабораторної роботи

1. Що таке система контролю версій (СКВ)?
СКВ — це програмний інструмент, який дозволяє відслідковувати зміни у файлах, зберігати історію їхніх версій та координувати роботу кількох розробників над одним проєктом.

2. Відмінності між розподіленою та централізованою СКВ:
- **Централізована (CVS, Subversion, Perforce):**
  - Є один центральний сервер.
  - Всі зміни зберігаються лише на ньому.
  - Для роботи потрібне постійне підключення до сервера.
- **Розподілена (Git, Mercurial):**
  - Кожен розробник має повну копію репозиторію (з історією).
  - Можна працювати офлайн.
  - Легке створення гілок і швидка робота з ними.

3. Різниця між stage та commit в Git:
- **Stage (індексація):** підготовка вибраних файлів до збереження (через git add).
- **Commit:** фіксація змін у локальному репозиторії з повідомленням (через git commit).

4. Як створити гілку в Git?

```
git branch new-branch
```

```
git checkout -b new-branch
```

5. Як створити або скопіювати репозиторій Git з віддаленого серверу?

```
git clone <url>
```

6. Що таке конфлікт злиття, як створити конфлікт, як вирішити конфлікт?

- **Конфлікт злиття** — ситуація, коли в одному файлі однакові рядки змінені по-різному в різних гілках.
- **Створити:** зробити зміни в одному файлі в двох гілках і спробувати злити (git merge).

- **Вирішити:** відкрити файл, вручну залишити потрібний варіант, потім:

git add <файл>

git commit

7. Коли використовуються merge, rebase, cherry-pick?

- **merge** — об’єднання двох гілок, зберігає історію.
- **rebase** — "переписування" історії: зміни однієї гілки застосовуються поверх іншої.
- **cherry-pick** — взяти один конкретний коміт із іншої гілки й застосувати у свою.

8. Як переглянути історію змін Git репозиторію в консолі?

git log

9. Як створити гілку в Git не використовуючи команду git branch?

git checkout -b new-branch

10. Як підготувати всі зміни в поточній папці до коміту?

git add .

11. Як підготувати всі зміни в дочірній папці до коміту?

git add path/to/folder/

12. Як переглянути перелік наявних гілок в репозиторії?

git branch

13. Як видалити гілку?

git branch -d branch_name # якщо гілка вже злита

git branch -D branch_name # примусове видалення

14. Які є способи створення гілки та в чому між ними різниця?

1. git branch new-branch — створює гілку, але не перемикає на неї.

2. git checkout -b new-branch — створює гілку і одразу перемикає на неї.

3. git switch -c new-branch — сучасний аналог checkout -b, більш читабельний.