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

Лабораторна робота №1
«Технології розроблення програмного забезпечення»
«Системи контроля версій»

Виконав:
Студент групи ІА-34
Марченко А.О.
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.

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 будь-коли 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;

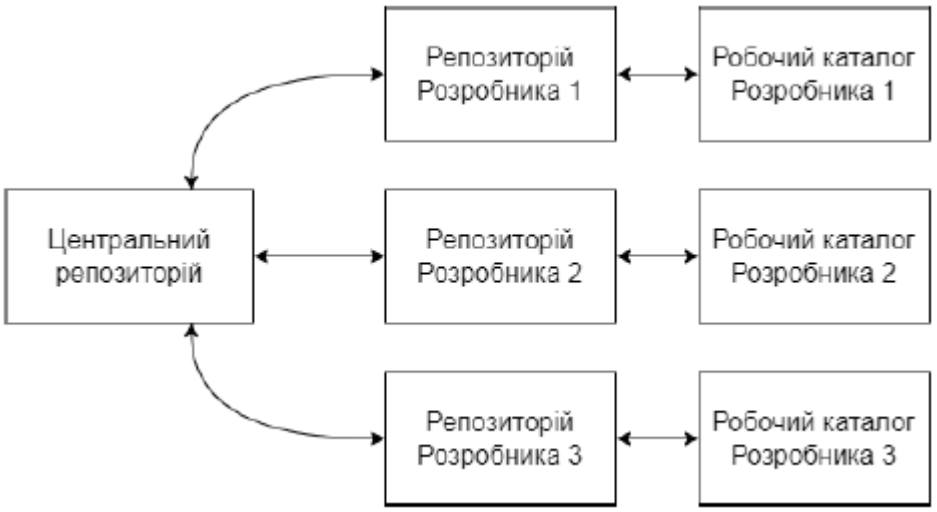


Рисунок 1.1. Схема процесу роботи з Git

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

Хід роботи

1. Створити локальний репозиторій.

```
Artem@ART MINGW64 /c/TRPZ
$ cd lab_1

Artem@ART MINGW64 /c/TRPZ/lab_1
$ git init
Initialized empty Git repository in C:/TRPZ/lab_1/.
git/
```

2.Переконатися що є гілка master

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git a
dd" to track)
```

3.Додавання та коміт першого файлу

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (master)
$ echo "Hello Wolrd!">text.txt

Artem@ART MINGW64 /c/TRPZ/lab_1 (master)
$ git add .
warning: in the working copy of 'text.txt', LF will be replac
ed by CRLF the next time Git touches it

Artem@ART MINGW64 /c/TRPZ/lab_1 (master)
$ git commit -m "frist commit"
[master (root-commit) e19b6fb] frist commit
 1 file changed, 1 insertion(+)
 create mode 100644 text.txt
```

4.Створення 2-х гілок двома способами

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (master)
$ git branch first_version

Artem@ART MINGW64 /c/TRPZ/lab_1 (master)
$ git checkout -b second_version
Switched to a new branch 'second_version'

Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ git  branch
  first_version
  master
* second_version
```

5.Створення 2-х файлів

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ echo "1">f1.txt

Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ echo "2">f2.txt
```

6.Додавання в stage

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ git add f1.txt
warning: in the working copy of 'f1.txt', LF will be replaced
 by CRLF the next time Git touches it
```

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ git add f2.txt
warning: in the working copy of 'f2.txt', LF will be replaced
 by CRLF the next time Git touches it
```

7. Комміт файл f1.txt на гілку first_version і файл f2.txt на гілку second_version

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ git commit f2.txt -m "1212"
warning: in the working copy of 'f2.txt', LF will be replaced
 by CRLF the next time Git touches it
[second_version 04809fe] 1212
 1 file changed, 1 insertion(+)
 create mode 100644 f2.txt

Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ git status
On branch second_version
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   f1.txt


Artem@ART MINGW64 /c/TRPZ/lab_1 (second_version)
$ git checkout first_version
Switched to branch 'first_version'
A       f1.txt

Artem@ART MINGW64 /c/TRPZ/lab_1 (first_version)
$ git commit f1.txt -m "2222"
warning: in the working copy of 'f1.txt', LF will be replaced
 by CRLF the next time Git touches it
[first_version 07d186e] 2222
 1 file changed, 1 insertion(+)
 create mode 100644 f1.txt
```

8.Дерево коммітів і гілок

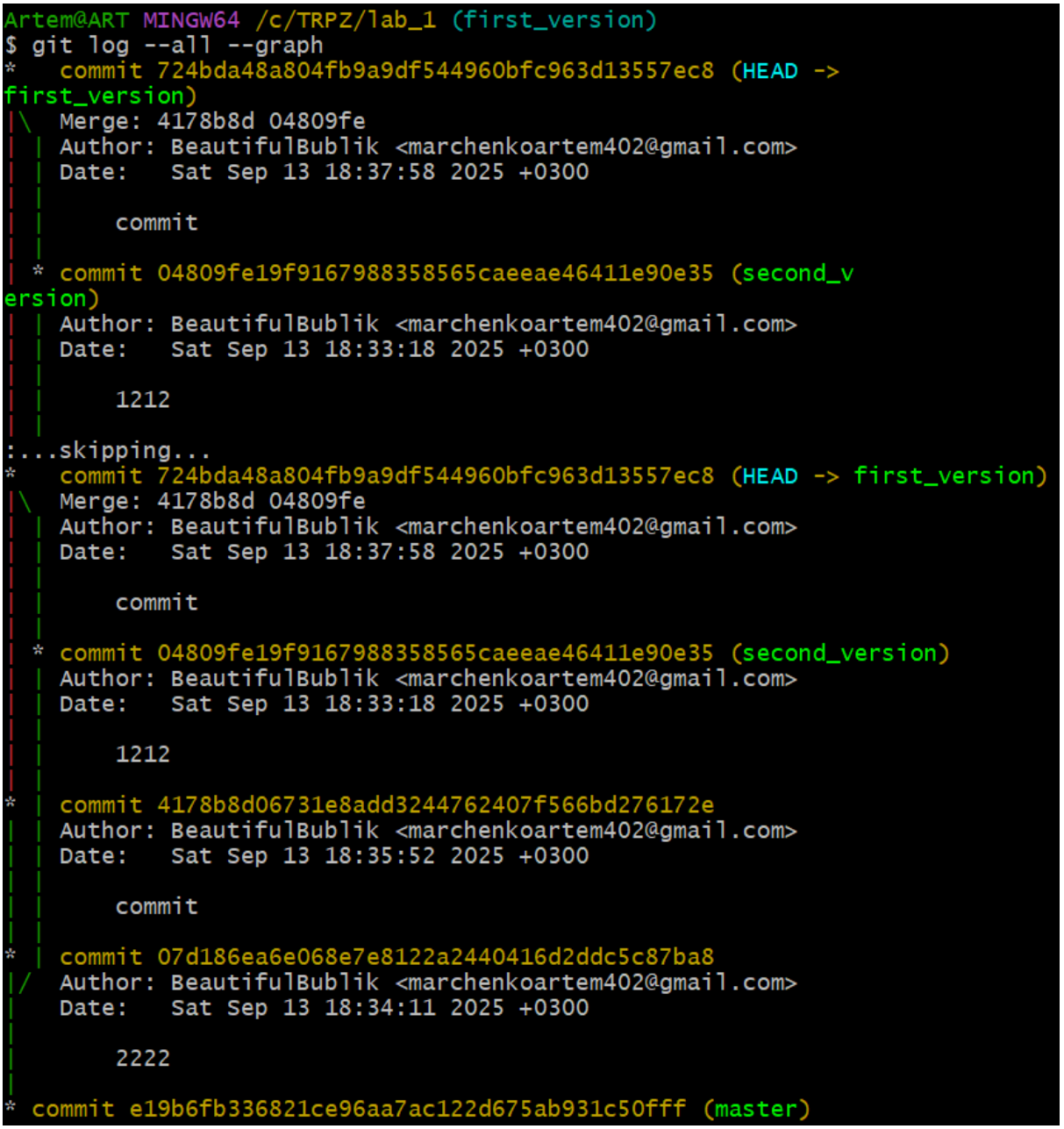
```
Artem@ART MINGW64 /c/TRPZ/lab_1 (first_version)
$ git log --all --graph
* commit 07d186ea6e068e7e8122a2440416d2ddc5c87ba8 (HEAD -> fi
rst_version)
| Author: BeautifulBublik <marchenkoartem402@gmail.com>
| Date:   Sat Sep 13 18:34:11 2025 +0300
|
|     2222
|
| * commit 04809fe19f9167988358565caeeae46411e90e35 (second_v
ersion)
|/  Author: BeautifulBublik <marchenkoartem402@gmail.com>
|   Date:   Sat Sep 13 18:33:18 2025 +0300
|
|       1212
|
* commit e19b6fb336821ce96aa7ac122d675ab931c50fff (master)
:...skipping...
* commit 07d186ea6e068e7e8122a2440416d2ddc5c87ba8 (HEAD -> first_version)
| Author: BeautifulBublik <marchenkoartem402@gmail.com>
| Date:   Sat Sep 13 18:34:11 2025 +0300
|
|     2222
|
| * commit 04809fe19f9167988358565caeeae46411e90e35 (second_version)
|/  Author: BeautifulBublik <marchenkoartem402@gmail.com>
|   Date:   Sat Sep 13 18:33:18 2025 +0300
|
|       1212
|
* commit e19b6fb336821ce96aa7ac122d675ab931c50fff (master)
  Author: BeautifulBublik <marchenkoartem402@gmail.com>
  Date:   Sat Sep 13 18:25:36 2025 +0300

      frist commit
```

9. Зміна файлу f2.txt та злиття з гілкою second_version і виникнення конфлікту

```
Artem@ART MINGW64 /c/TRPZ/lab_1 (first_version)
$ echo "a">f2.txt

Artem@ART MINGW64 /c/TRPZ/lab_1 (first_version)
$ git add .
warning: in the working copy of 'f2.txt', LF will be replaced
 by CRLF the next time Git touches it

Artem@ART MINGW64 /c/TRPZ/lab_1 (first_version)
$ git commit -m "commit"
[first_version 4178b8d] commit
 1 file changed, 1 insertion(+)
 create mode 100644 f2.txt

Artem@ART MINGW64 /c/TRPZ/lab_1 (first_version)
$ git merge second_version
Auto-merging f2.txt
CONFLICT (add/add): Merge conflict in f2.txt
Automatic merge failed; fix conflicts and then commit the res
ult.
```

(Вирішення: залишити в файлі потрібні змінни і закомітити)

10.Граф коммітів зі злиттям



Висновок:У ході виконання роботи було створено , що дозволило на практиці закріпити основні принципи роботи з системою контролю версій. Було продемонстровано базові операції: створення версій , робота з гілками (їх створення та злиття), виконання комітів та вирішення можливих конфліктів. У результаті можна зробити висновок, що Git забезпечує ефективне управління змінами, зручну організацію спільної роботи та надійне збереження історії проєкту.