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
Технології розробки програмного забезпечення

«Знайомство з Git»

Виконав:
студент групи ІА-33
Мартинюк Ю.Р.

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.

1.1. Завдання

- Ознайомитись із короткими теоретичними відомостями.
- Створити Git репозиторій.
- Клонувати Git репозиторій.
- Продемонструвати базову роботу з репозиторієм: створення версій, додавання тегів, робіт з гілками (створення та злиття), робота з комітами, вирішення конфліктів, а також робота з віддаленим репозиторієм.

1.2. 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;

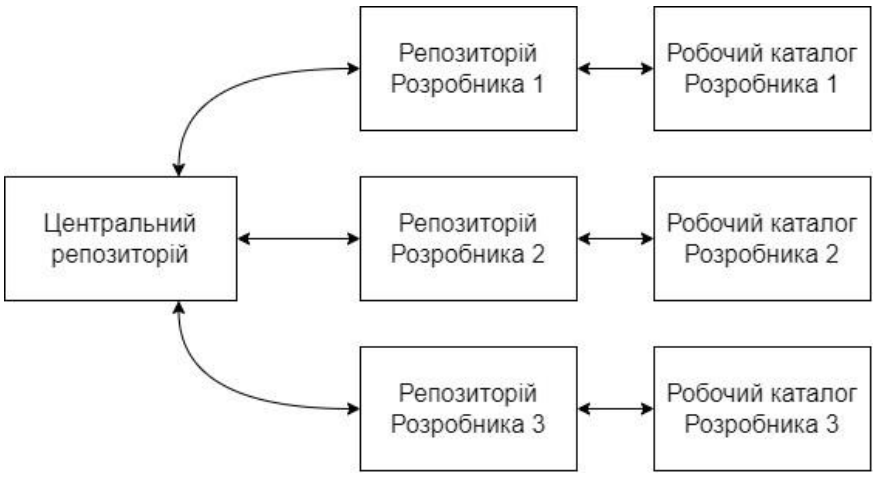

Рисунок 1.1. Схема процесу роботи з Git

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 репозиторієм). 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

Створюємо директорію за допомогою mkdir
Переходимо до директорії за допомогою cd:

```
E:\>cd gitlab1
```

Ініціалізуємо git репозиторій:

```
E:\gitlab1>git init
Initialized empty Git repository in E:/gitlab1/.git/
```

Робимо пустий коміт для можливості створення нових гілок:

```
E:\gitlab1>git commit --allow-empty
[master (root-commit) edb4ade] init
```

Створюємо гілки b1-b3 трьома різними способами:

```
E:\gitlab1>git branch b1

E:\gitlab1>git checkout -b b2
Switched to a new branch 'b2'

E:\gitlab1>git checkout master
Switched to branch 'master'

E:\gitlab1>git switch -c b3
Switched to a new branch 'b3'
```

Створюємо три файли f1-f3:

```
E:\gitlab1>echo "123" > f1.txt

E:\gitlab1>echo "qwer" > f2.txt

E:\gitlab1>echo "asd" > f3.txt
```

Перемикаємось на master:

```
E:\gitlab1>git checkout master
Switched to branch 'master'
```

Додаємо один файл f1.txt та робимо коміт:

```
E:\gitlab1>git add f1.txt

E:\gitlab1>git commit -m "f1"
[master 444b48a] f1
 1 file changed, 1 insertion(+)
 create mode 100644 f1.txt
```

Перемикаємось на b1, додаємо файл f2.txt та робимо коміт:

```
E:\gitlab1>git checkout b1
Switched to branch 'b1'

E:\gitlab1>git add f2.txt

E:\gitlab1>git commit -m "f2"
[b1 3334c2a] f2
 1 file changed, 1 insertion(+)
 create mode 100644 f2.txt
```

Перемикаємось на b2, додаємо файл f3.txt та робимо коміт:

```
E:\gitlab1>git checkout b2
Switched to branch 'b2'
```

```
E:\gitlab1>git add f3.txt

E:\gitlab1>git commit -m "f3"
[b2 c4f71ec] f3
 1 file changed, 1 insertion(+)
 create mode 100644 f3.txt
```

Створюємо, додаємо та комітимо файл f2.txt на гілці b2:

```
E:\gitlab1>echo "asffkj" > f2.txt

E:\gitlab1>git add f2.txt

E:\gitlab1>git commit -m "klsdjf"
[b2 ce4f4e6] klsdjf
 1 file changed, 1 insertion(+)
 create mode 100644 f2.txt
```

Мерджимо b1 в гілку b2, отримуємо конфлікт, так як у b1 та b2 є файли f2.txt з різним наповненням:

```
E:\gitlab1>git merge b1
Auto-merging f2.txt
CONFLICT (add/add): Merge conflict in f2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Редагуємо файл f2.txt, тим самим вирішуючи конфлікт, додаємо цей відредагований файл та продовжуємо merge

```
E:\gitlab1>code .

E:\gitlab1>git add f2.txt

E:\gitlab1>git merge --continue
[b2 d953e03] Merge branch 'b1' into b2
```

Через git log --graph дивимось, які коміти та які гілки було злито:

```
E:\gitlab1>git log --graph
*   commit d953e0337cffa2de5d4abe314fe18184c99a211c (HEAD -> b2)
|\  Merge: ce4f4e6 3334c2a
| | Author: MartyniukYurii <145149346+masonabor@users.noreply.github.com>
| | Date:   Sat Sep 6 13:41:02 2025 +0300
| |
| |     Merge branch 'b1' into b2
| |
| * commit 3334c2a29e88d57750f9f11cebf67d10494360fa (b1)
| | Author: MartyniukYurii <145149346+masonabor@users.noreply.github.com>
| | Date:   Sat Sep 6 13:35:30 2025 +0300
| |
| |     f2
| |
* | commit ce4f4e604bc98652d4ff6868063b7bbf6777238b
| | Author: MartyniukYurii <145149346+masonabor@users.noreply.github.com>
| | Date:   Sat Sep 6 13:39:25 2025 +0300
| |
| |     klsdjf
| |
* | commit c4f71ecc848799ddb855f139528aa2e65b9955e7
|/  Author: MartyniukYurii <145149346+masonabor@users.noreply.github.com>
|   Date:   Sat Sep 6 13:36:05 2025 +0300
|
|       f3
|
* commit edb4ade77d817136e481d661b0eb9bc8f724a33f (b3)
  Author: MartyniukYurii <145149346+masonabor@users.noreply.github.com>
  Date:   Sat Sep 6 13:28:04 2025 +0300

      init
```

Висновок: У ході виконання даної лабораторної роботи я навчився основам роботи з системою контролю версій git, навчився створювати гілки, робити коміти, зливати гілки в одну та вирішувати конфлікти при злитті.