Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота № 1

Технології розроблення програмного забезпечення

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

Виконав:
студент групи ІА–33:
Хитрова НА

Перевірив:
Мягкий МЮ

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 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.

Хід роботи

```
PS C:\Users\admin> mkdir gitTest


    Directory: C:\Users\admin


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        05/12/2025     23:19                gitTest
```

Створюємо нову директорію з ім’ям gitTest

```
PS C:\Users\admin> cd gitTest/
```

Переходимо у щойно створену папку gitTest

```
PS C:\Users\admin\gitTest> git init
Initialized empty Git repository in C:/Users/admin/gitTest/.git/
```

Ініціалізуємо порожній Git-репозиторій у поточному каталозі. Створюється прихована папка .git, де Git зберігає всю історію.

```
PS C:\Users\admin\gitTest> git branch br1
fatal: not a valid object name: 'master'
```

Створює нову гілку br1, але тільки якщо існує хоча б один коміт, якого в нас поки що немає.

```
PS C:\Users\admin\gitTest>  git commit -m "init commit" --allow-empty
[master (root-commit) d178f40] init commit
```

Створюємо порожній коміт (без файлів). Потрібний, щоб Git мав "першу точку" для роботи з гілками.

```
PS C:\Users\admin\gitTest> git branch br1
```

Створюємо гілку br1, тепер уже успішно.

```
PS C:\Users\admin\gitTest> git checkout -b br2
Switched to a new branch 'br2'
```

Створюємо нову гілку br2 і відразу перемикаються на неї.

```
PS C:\Users\admin\gitTest> git switch -c br3
Switched to a new branch 'br3'
```

Створюємо гілку br3 і перемикаються на неї.

```
PS C:\Users\admin\gitTest> git branch -v
  br1    d178f40 init commit
  br2    d178f40 init commit
* br3    d178f40 init commit
  master d178f40 init commit
PS C:\Users\admin\gitTest>
```

Виводимо список усіх локальних гілок, а також останній коміт у кожній гілці, тобто бачимо, що існує 4 локальні гілки та те, що ми зараз знаходимося у гілці br3

```
PS C:\Users\admin\gitTest> git co master
Switched to branch 'master'
```

Переходимо на гілку master

```
PS C:\Users\admin\gitTest> git clone C:\Users\admin\gitTest gitTestClone
Cloning into 'gitTestClone'...
done.
```

Клонування локального репозиторію

```
PS C:\Users\admin\gitTest> touch f3.txt


    Directory: C:\Users\admin\gitTest


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         05/12/2025     23:47              0 f3.txt
```

Створили порожній файл f3.txt. PowerShell показав інформацію про новий файл.

```
PS C:\Users\admin\gitTest> ls


    Directory: C:\Users\admin\gitTest


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         05/12/2025     23:44                gitTestClone
-a----         05/12/2025     23:37              0 f1.txt
-a----         05/12/2025     23:47              0 f2.txt
-a----         05/12/2025     23:47              0 f3.txt
```

Бачимо вкладені файли, та скопійований Git-репозиторій після того, як ми зробили git clone.

```
PS C:\Users\admin\gitTest> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        f1.txt
        f2.txt
        f3.txt
        gitTestClone/

nothing added to commit but untracked files present (use "git add" to track)
```

На гілці master є 4 непроіндексовані об’єкти. 3 файли та ціла папка gitTestClone (яка теж є Git-репозиторієм).

```
PS C:\Users\admin\gitTest> git branch
  br1
  br2
  br3
* master
```

Знаходимося на гілці master.

```
PS C:\Users\admin\gitTest> git log --graph
* commit d178f404b7d2b9b95b0db9fbfdacedeef9bdafee (HEAD -> master, br3, br2,
 br1)
  Author: ch1fir <nattokakhit@gmail.com>
  Date:   Fri Dec 5 23:20:14 2025 +0100

      init commit
```

Перевірка історії, є тільки один коміт.

```
PS C:\Users\admin\gitTest> git checkout br3
A       f1.txt
A       f2.txt
A       f3.txt
A       gitTestClone
Switched to branch 'br3'
```

Перехід на br3 з уже проіндексованими файлами

```
PS C:\Users\admin\gitTest> git status
On branch br3
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   f1.txt
        new file:   f2.txt
        new file:   f3.txt
        new file:   gitTestClone
```

4 об’єкти знаходяться готові до коміту.

```
PS C:\Users\admin\gitTest> echo "one more line" >> f1.txt
```

До файлу f1.txt додається рядок one more line.

```
PS C:\Users\admin\gitTest> git status
On branch br3
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   f1.txt
        new file:   f2.txt
        new file:   f3.txt
        new file:   gitTestClone

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:   f1.txt
```

У staging вже лежить версія f1.txt (порожня), яку ми додали раніше. У робочій директорії файл змінився. Git бачить, що одна версія файлу в staging (порожня), інша – на диску (з текстом). Тому f1.txt одночасно у списку "Changes to be committed" (стара staged версія) і в "Changes not staged for commit" (нова версія в робочій директорії)

```
PS C:\Users\admin\gitTest> git add f1.txt
PS C:\Users\admin\gitTest> git status
On branch br3
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   f1.txt
        new file:   f2.txt
        new file:   f3.txt
        new file:   gitTestClone
```

Оновлення staging для f1.txt. Тепер у staging лежить вже оновлений f1.txt.

```
PS C:\Users\admin\gitTest> git commit -m "Add extra change into f1 on br3"
[br3 71a492d] Add extra change into f1 on br3
 4 files changed, 1 insertion(+)
 create mode 100644 f1.txt
 create mode 100644 f2.txt
 create mode 100644 f3.txt
 create mode 160000 gitTestClone
```

Створився новий коміт 71a492d на гілці br3. У нього увійшли 4 об’єкти. Тепер гілка br3 тепер відрізняється від master, br1, br2, в ній є новий коміт із файлами.

```
PS C:\Users\admin\gitTest> git log --graph --oneline --all
* 71a492d (HEAD -> br3) Add extra change into f1 on br3
* d178f40 (master, br2, br1) init commit
```

Перегляд історії всіх гілок, тепер в історії тепер 2 коміти.

```
PS C:\Users\admin\gitTest> git checkout br2
warning: unable to rmdir 'gitTestClone': Directory not empty
Switched to branch 'br2'
PS C:\Users\admin\gitTest> ls


    Directory: C:\Users\admin\gitTest


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        05/12/2025     23:44                gitTestClone


PS C:\Users\admin\gitTest> echo "VERSION FROM BR2" > f2.txt
PS C:\Users\admin\gitTest> git add f2.txt
PS C:\Users\admin\gitTest> git commit -m "Change f2 in br2"
[br2 5e61fff] Change f2 in br2
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 f2.txt
PS C:\Users\admin\gitTest>
```

Ми перейшли на br2, перевірили директорію, створили файл f2.txt зі своїм текстом для br2, додали файл в індекс і закомітили.

```
PS C:\Users\admin\gitTest> git checkout br3
Switched to branch 'br3'
PS C:\Users\admin\gitTest> echo "VERSION FROM BR3" > f2.txt
PS C:\Users\admin\gitTest> git add f2.txt
PS C:\Users\admin\gitTest> git commit -m "Change f2 in br3"
[br3 5fa93c4] Change f2 in br3
 1 file changed, 0 insertions(+), 0 deletions(-)
PS C:\Users\admin\gitTest>
```

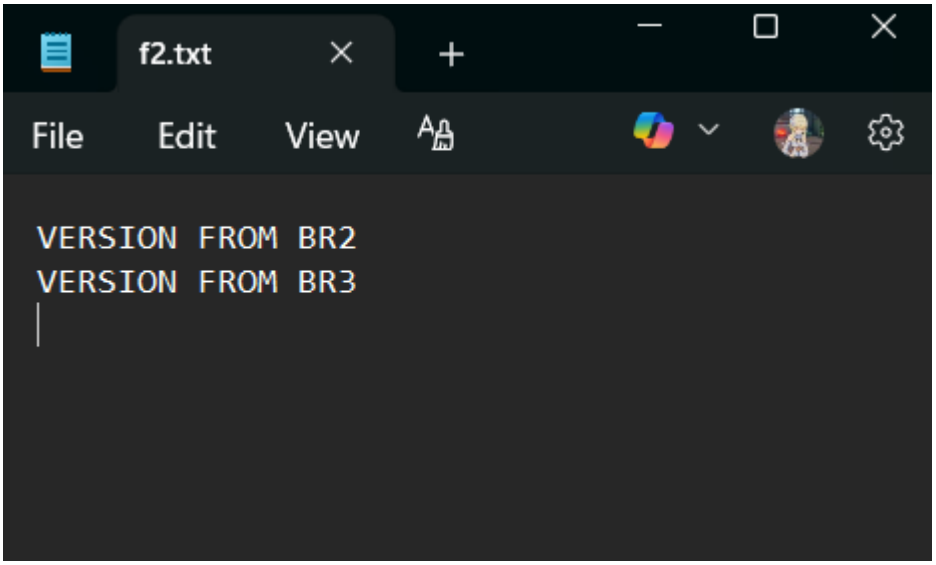
Аналогічно робимо зміни в br3.

```
PS C:\Users\admin\gitTest> git checkout br2
warning: unable to rmdir 'gitTestClone': Directory not empty
Switched to branch 'br2'
PS C:\Users\admin\gitTest> git merge br3
warning: Cannot merge binary files: f2.txt (HEAD vs. br3)
Auto-merging f2.txt
CONFLICT (add/add): Merge conflict in f2.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Merge з конфліктом. Git не може автоматично обрати, який варіант f2.txt правильний з br2 чи з br3.

```
PS C:\Users\admin\gitTest> notepad f2.txt
```

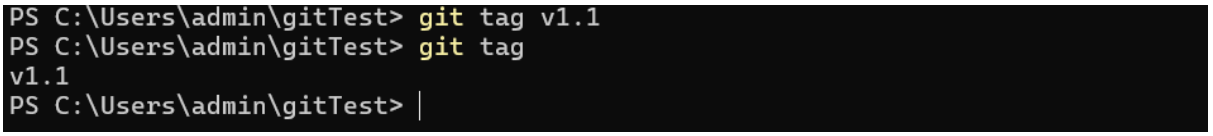
Вирішуємо конфлікт вручну. Відкриємо файл в блокноті.



Редагуємо файл, об’єднуємо два варіанти

Фінальна стадія merge-коміту. Файл .git/COMMIT_EDITMSG. У верхній частині Merge branch 'br3' into br2 автоматично згенероване Git повідомлення про злиття.

Успішно merge завершено.



Додаємо тег та передивляємо всі наявні теги.

```
PS C:\Users\admin\gitTest> git checkout -b feature-x
Switched to a new branch 'feature-x'
PS C:\Users\admin\gitTest> echo "new feature" > feature.txt
PS C:\Users\admin\gitTest> git add feature.txt
PS C:\Users\admin\gitTest> git commit -m "Add feature X"
[feature-x 26c5c52] Add feature X
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 feature.txt
PS C:\Users\admin\gitTest> git log --graph --oneline --all
* 26c5c52 (HEAD -> feature-x) Add feature X
*   a75dd1c (tag: v1.1, br2) Merge branch 'br3' into br2
|\
| * 5fa93c4 (br3) Change f2 in br3
| * 71a492d Add extra change into f1 on br3
* | 5e61fff Change f2 in br2
|/
* d178f40 (master, br1) init commit
PS C:\Users\admin\gitTest>
```

Створюємо ще одну гілку для демонстрації розвитку проекту.

```
PS C:\Users\admin\gitTest> git checkout master
warning: unable to rmdir 'gitTestClone': Directory not empty
Switched to branch 'master'
PS C:\Users\admin\gitTest> git merge feature-x
Updating d178f40..26c5c52
Fast-forward
 f1.txt       | Bin 0 -> 32 bytes
 f2.txt       | Bin 0 -> 74 bytes
 f3.txt       |   0
 feature.txt  | Bin 0 -> 28 bytes
 gitTestClone |   1 +
 5 files changed, 1 insertion(+)
 create mode 100644 f1.txt
 create mode 100644 f2.txt
 create mode 100644 f3.txt
 create mode 100644 feature.txt
 create mode 160000 gitTestClone
PS C:\Users\admin\gitTest>
```

Merge цієї нової гілки у master

Робота з віддаленим репозиторієм:
https://github.com/ch1fir/trpz-lab-1.git

```
PS C:\Users\admin\gitTest> git remote add origin https://github.com/ch1fir/t
rpz-lab-1.git
```

Підключили “origin”

```
PS C:\Users\admin\gitTest> git remote -v
origin  https://github.com/ch1fir/trpz-lab-1.git (fetch)
origin  https://github.com/ch1fir/trpz-lab-1.git (push)
```

Перевіряємо

```
PS C:\Users\admin\gitTest> git push -u origin --all
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 16 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (18/18), 1.49 KiB | 305.00 KiB/s, done.
Total 18 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/ch1fir/trpz-lab-1.git
 * [new branch]      br1 -> br1
 * [new branch]      br2 -> br2
 * [new branch]      br3 -> br3
 * [new branch]      feature-x -> feature-x
 * [new branch]      master -> master
branch 'br1' set up to track 'origin/br1'.
branch 'br2' set up to track 'origin/br2'.
branch 'br3' set up to track 'origin/br3'.
branch 'feature-x' set up to track 'origin/feature-x'.
branch 'master' set up to track 'origin/master'.
PS C:\Users\admin\gitTest>
```

Надіслали все на GitHub

```
PS C:\Users\admin\gitTest> git push --tags
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/ch1fir/trpz-lab-1.git
 * [new tag]         v1.1 -> v1.1
PS C:\Users\admin\gitTest>
```

Надіслали теги

```
PS C:\Users\admin\gitTest> git log --graph --oneline --decorate --all
* 26c5c52 (HEAD -> master, origin/master, origin/feature-x, feature-x) Add f
eature X
*   a75dd1c (tag: v1.1, origin/br2, br2) Merge branch 'br3' into br2
|\
| * 5fa93c4 (origin/br3, br3) Change f2 in br3
| * 71a492d Add extra change into f1 on br3
* | 5e61fff Change f2 in br2
|/
* d178f40 (origin/br1, br1) init commit
```

Демонстрація перегляду всіх комітів.

Висновок: У ході виконання лабораторної роботи ми ознайомилися з основними принципами роботи з децентралізованою системою контролю версій Git та виконали практичні дії, що демонструють стандартний робочий процес у реальному програмному проекті. Було створено локальний репозиторій, додані файли та сформовано перші коміти. Я навчився створювати нові гілки, перемикатися між ними та виконувати їх злиття. Під час роботи було отримано реальний приклад виникнення конфлікту при злитті гілок, який був успішно вирішений шляхом ручного редагування файлу та завершення merge-коміту.

1.4. Питання до лабораторної роботи

1. Що таке система контролю версій (СКВ)?

СКВ – це програма, яка зберігає історію змін файлів, дозволяє повертатися до старих версій, працювати в команді над одним проектом і безпечно об’єднувати зміни.

2. Поясніть відмінності між розподіленою та централізованою СКВ.

У централізованій СКВ (як SVN) є один центральний сервер з усією історією, а клієнти отримують лише потрібні версії. У розподіленій (як Git) кожен розробник має повну копію репозиторію з усією історією, може комітити локально й синхронізуватися з іншими.

3. Поясніть різницю між stage та commit в Git.

Stage (індекс) – це підготовча зона, куди ми складаємо зміни, які хочемо зберегти. Commit – це вже зафіксований «знімок» усіх змін, що були додані в stage на цей момент.

4. Як створити гілку в Git?

git branch new-branch – створює гілку

git checkout -b new-branch або git switch -c new-branch – створює і одразу перемикає на неї.

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

Щоб скопіювати існуючий віддалений репозиторій, використовують:

git clone <url> – Git завантажує всю історію та створює локальну копію.

Створити новий локальний репозиторій можна через git init, а потім прив’язати remote (git remote add origin <url>).

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

Конфлікт злиття виникає, коли Git не може автоматично об’єднати зміни (наприклад, змінені одні й ті самі рядки у двох гілках). Створити його можна, якщо в одній і тій самій частині файлу зробити різні зміни в двох гілках і виконати git merge. Вирішується конфлікт вручну: відредагувати файл, залишивши правильний варіант, потім git add <file> і зробити новий commit.

7. В яких ситуаціях використовуються команди: merge, rebase, cherry-pick?

merge – коли треба об’єднати дві гілки, зберігаючи їхню історію.

rebase – коли хочуть переписати історію гілки, підлаштувавши її під іншу (робити історію лінійною).

cherry-pick – коли треба взяти один конкретний commit з іншої гілки і застосувати його в поточній.

8. Як переглянути історію змін Git репозиторію в консолі?

git log – показує список комітів

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

Можна одразу створити і перейти в гілку:
git checkout -b new-branch
або
git switch -c new-branch.
Тут гілка створюється без окремого виклику git branch.

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

У поточній папці всі зміни (нові, змінені, видалені файли) можна додати в stage командою git add .

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

Треба вказати шлях до цієї папки:
git add path/to/subfolder
Тоді в stage потраплять лише зміни всередині цієї дочірньої папки.

12. Як переглянути перелік наявних гілок в репозиторії?

Список локальних гілок показує команда:
git branch
Усі гілки (локальні + віддалені) можна подивитись через:
git branch -a

13. Як видалити гілку?

Локальну гілку видаляють так:
git branch -d <branch-name> – звичайне видалення,
git branch -D <branch-name> – примусове.
Віддалену через:
git push origin --delete <branch-name>

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

Основні способи:
git branch new-branch – просто створює гілку від поточного коміту, але не перемикає на неї.
git checkout -b new-branch / git switch -c new-branch – створює гілку і відразу переходить в неї.
git branch new-branch <commit> – створює гілку, що починається з конкретного коміту або тега. Різниця в тому, чи перемикається Git на нову гілку автоматично і з якого саме коміту вона стартує.