Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

# 

# 

**GIT ТА ЙОГО ВИКОРИСТАННЯ**

**Текстова частина до курсової роботи**

**за спеціальністю „Програмна інженерія” 6.050103**

**Керівник курсової роботи**

к.ф.-м.н., доц. Глибовець А. М.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(підпис)* “\_\_\_\_” \_\_\_\_\_\_\_\_\_\_ 2015 р.

Виконала студентка

Яремко С. А.

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_ 2015 р.

Київ 2015

**Тема:** Git та його використання

**Календарний план виконання роботи:**

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Назва етапу курсової роботи | Термін виконання етапу | Примітка |
| 1. | Отримання теми курсової роботи. | 09.10.2014 |  |
| 2. | Пошук та збір тематичних матеріалів. | 15.11.2014 |  |
| 3. | Ознайомлення з відповідними матеріалами та обдумування структури роботи. | 15.12.2014 |  |
| 4. | Написання вступу та плану роботи. | 25.12.2014 |  |
| 5. | Вивчення відповідних матеріалів, що необхідні для написання першого розділу роботи, та безпосереднє його написання. | 10.02.2015 |  |
| 5. | Опрацювання необхідних матеріалів та написання другого розділу роботи. | 15.03.2015 |  |
| 6. | Вивчення потрібних джерел та завершення роботи над третім розділом роботи. | 20.04.2015 |  |
| 7. | Коректне оформлення роботи відповідно до вимог написання курсової роботи. | 28.04.2015 |  |
| 8. | Створення презентації та написання доповіді для захисту роботи. | 5.05.2015 |  |
| 9. | Подання та аналіз попередньої версії роботи з керівником. | 7.05.2015 |  |
| 10. | Корегування роботи згідно із зауваженнями керівника. | 18.05.2015 |  |
| 11. | Захист курсової роботи. | 20.05.2015 |  |

СтудентЯремко С. А.

Керівник Глибовець А. М.

**“\_\_\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

ЗМІСТ

**Анотація** ……………….......................................................................................4

***ВСТУП***...................................................................................................................5

**РОЗДІЛ 1: Системи керування версіями**……………………….…...............7

1. . Призначення та типи систем керування версіями…………………….7
2. . Огляд різних систем керування версіями……………………………..11
3. . Загальний опис системи керування версіями Git…………………….16
4. . Порівняння Git з іншими системами керування версіями…………...19

**РОЗДІЛ 2: Структура та основи роботи з Git** ..............................................25

1. . Спосіб збереження даних в Git ………………………………………...25
2. . Налаштування та початок роботи з Git………………………………...28
3. . Робота з коммітами…………………………………….. ……………....32
4. . Перегляд історії проекту………………………………………………..41
5. . Галуження в Git………………………………………………………….42

**РОЗДІЛ 3: Робота з Git на сервері**………........................................................50

1. . Огляд мережних протоколів……............................................................50
2. . Створення та робота з віддаленим репозиторієм……..……...………..52
3. . Опис Git хостинг-сайту GitHub………………. ………………….........56

**Висновки** ..............................................................................................................59 **Список використаної літератури**.....................................................................61

**Глосарій** …………………………………………………………………………64

Анотація

У курсовій роботі з’ясовано переваги використання розподіленої системи керування версіями Git та визначено основні її можливості та способи застосування. Було обґрунтовано доцільність використання систем керування версіями та більшу ефективність використання розподілених систем керування версіями у порівнянні з централізованими. У роботі було досліджено основні переваги та недоліки використання Git, надавши її порівняння з іншими найбільш використовуваними системами, описано основні команди та принципи використання Git. Також надано опис роботи з Git на сервері та розглянуто засади роботи з веб-сервісом GitHub.

**Ключові слова:** система керування версіями, розподілена система керування версіями, централізована система керування версіями, Git, зміни, розробник, розробка, файл, проект, версія, робочий каталог, область підготовлених файлів, репозиторій, віддалений репозиторій, локальний репозиторій, чистий репозиторій, комміт, гілка, віддалена гілка, мітка, галуження, злиття, команда, історія проекту, клонування, хеш-код, сервер, веб-сервіс, хостинг-сайт, протокол, GitHub.

Вступ

**Актуальність теми.** У процесі розробки програмного забезпечення виникла потреба у збереженні та контролі різних версій проекту для підвищення продуктивності роботи, а також більш зручної та ефективної організації спільної роботи групи розробників над певними проектами. Розподілена система керування версіями Git стала одним з найбільш ефективних рішень цієї проблеми у порівняння з іншими системами.

Проект, першочергово розроблений для управління розробкою ядра Linux, є одним з найбільш зручних, надійних, швидких та високопродуктивних систем керування версіями завдяки іншому підходу до збереження файлів як зліпка їхніх станів у певний момент часу, швидкому та надійному галуженню, а в результаті – надання можливості паралельної роботи над вирішенням декількох завдань та внесенням змін без впливу на основний код.

За допомогою цих та інших особливостей система керування версіями Git забезпечує високий рівень контролю над процесом створення програмного забезпечення, що є здійсненим певною групою розробників чи окремим розробником, ефективність, зручність та швидкість командної розробки програмного забезпечення, що є вкрай важливими аспектами у процесі роботи над проектом.

**Мета дослідження** – з’ясувати переваги застосування розподіленої системи керування версіями Git, визначити її основні можливості та способи використання.

Досягнення мети передбачає виконання низки **завдань**:

* обґрунтування доцільності використання систем керування версіями;
* опис найбільш поширених систем керування версіями;
* обґрунтування більшої продуктивності використання розподіленої системи керування версіями у порівняння з централізованою;
* дослідження основних переваг та недоліків Git у порівнянні з іншими системами керування версіями;
* розгляд способу збереження даних в Git, а також принципів роботи та основних команд;
* опис роботи з Git на сервері;
* розгляд основних засад роботи з веб-сервісом GitHub.

**Об’єкт дослідження** – розподілена система керування версіями файлів та командної роботи Git.

**Предметом дослідження** є переваги та недоліки системи керування версіями Git у порівнянні з іншими системами, спосіб збереження даних, основні команди, принципи роботи з Git та його використання.

Логіка дослідження зумовила таку структуру курсової роботи: вступ, 3 розділи, висновки, список використаної літератури.

У першому розділі обгрунтовано доцільність використання систем керування версіями, наведено опис основних із них. Також надана загальна характеристика розподіленої системи керування версіями Git та її переваги й недоліки у порівнянні з іншими системами.

Другий розділ присвячено розгляду підходу Git до збереження даних, надано детальний опис основних команд та можливостей Git.

У розділі три розглянуто роботу з Git на сервері, зокрема описано Git хостинг-сайт GitHub та його використання для роботи з Git.

1. Системи керування версіями
   1. Призначення та типи систем керування версіями

Загальновідомим є той факт, що ІТ-індустрія набула справді широкого розмаху. На ринку з’являється все більше компаній, що розробляють програмне забезпечення. Оскільки дуже часто потрібно працювати над справді масштабними проектами, то розробка проводиться одночасно декількома програмістами, кожен з яких працює над певною частиною проекту, виконуючи своє завдання. Безумовно, такий підхід забезпечує більшу продуктивність та швидкість роботи над проектом, проте виникає проблема контролю над процесом виконання кожного завдання, а також проекту в цілому, стеження за змінами, що вносить кожен розробник, та узгодження їх діяльності. Вирішенням цих проблем стало використання такого інструмента як систем керування версіями, що забезпечують відстеження, документування та контроль над поступовим внесенням змін в проект, що здійснює кожним з розробників.

Також використання систем керування версіями є доцільним й для окремих розробників, що працюють над масштабними проектами для самостійного контролю стадій розробки та для відображення перебігу робочого процесу для замовника. Це забезпечене тим, що кожна версія коду, збережена в системі керування версіями має власну назву, а також є збереженим час зміни та інформація про автора зміни. Використання таких систем при розробці проекту дозволяє краще організувати діяльність, підвищити швидкість та ефективність праці, а також запобігає можливості втрати файлів проекту чи самого проекту в цілому.

Отже, система керування версіями - це система, що веде протокол змін, внесених у файли проекту протягом усієї роботи, для забезпечення можливості повернення при потребі до попередніх версій цих файлів, а також дозволяє кільком розробникам одночасно, без завад одне одному, проводити роботу над спільними проектами, об’єднуючи зміни, зроблені кожним із них [1].

Місце, де є збережені усі версії файлів проекту, називається репозиторієм. У кожному репозиторії можна створювати паралельні лінії розробки - гілки, що забезпечують можливість збереження експериментальних, незавершених та повністю робочих версій проекту.

Використання систем керування версіями при роботі над великими проектами, яку здійснює один чи декілька розробників, надає ряд таких можливостей:

* резервне копіювання і відновлення - збереження файлів у процесі редагування, що надає можливість повернення до будь-якої збереженої версії проекту;
* повернення до "останньої вдалої" версії при внесенні невдалих змін у файли проекту;
* при виявленні помилок можливість повернення до давніших версій проекту і перегляд змін, що були внесенні в даний проект у цій версії;
* отримання інформація про те, хто востаннє змінював щось, що могло призвести до проблеми;
* отримання інформація про те, хто вирішив проблему та коли;
* додавання пояснень до змін та їх документування;
* перегляд змін, зроблених протягом усього часу;
* розгалуження і злиття - створення копії проекту в окремій області, її модифікація та тестування без внесення змін до основної гілки проекту з подальшою можливістю об’єднання з основним кодом;
* контроль доступу користувачів до файлів з можливістю його обмеження;
* порівняння та знаходження відмінностей у файлах проекту;
* синхронізація – надання можливості обміну файлами та отримання інформації щодо останньої версії проекту [1, 2, 3].

Розрізняють системи керування версіями трьох типів: локальні, централізовані та розподілені системи.

Велика кількість розробників для збереження та контролю версій створюють копії файлів у іншій директорії, додаючи дату в назву директорії.

Такий підхід є простим, а отже досить поширеним, проте він не забезпечує бажаного контролю робочого процесу над проектом, оскільки можна помилитися директорією і записати не ті файли або помилково скопіювати і перезаписати інші файли.

Для вирішення цією проблему уже давно була створена локальна система керування версіями з простою базою даних, в якій є збережені усі зміни необхідних файлів.

Однією з найбільш поширених систем керування версіями цього типу стала RCS, яку й досі встановлюють на багатьох комп'ютерах. Вона базується на збереженні наборів різниць між файлами, патчів, від однієї зміни файлів до іншої, які збережені в спеціальному форматі на диску. Це дозволяє відновити будь-який файл на будь-який момент часу за допомогою послідовного накладання патчів.

Суттєвим недоліком цієї системи є можливість втрати повної історії змін проекту, якщо вони збережені в одному місці [4].

### Проте швидко виникла проблема співпраці з іншими, вирішенням якої стало створення централізованих систем керування версіями.

### У цих системах усі файли проекту та клієнти, що отримують копію останньої версії усіх файлів цього проекту є збережені на одному центральному сервері.

### Така система має ряд переваг:

* наявність загальної нумерації версій;
* збереження усіх даних на одному сервері;
* можливість реалізації функції блокування файлів;
* можливість керування доступом до файлів;

Проте вона містить й ряд значних недоліків:

* наявність потреби в постійному мережному з'єднанні для оновлення робочої копії чи збереження змін;
* наявність вразливого місця - централізованого сервера, при пошкодженні якого та відсутності резервної копії проекту відбувається втрата абсолютно всієї історії проекту, за винятком кількох робочих версій файлів проекту, збережених на робочих машинах користувачів, а також неможливості взаємодії розробників та збереження нових версій при вимкнені сервера чи відсутності мережі;
* спочатку проведення злиття користувачами існуючих ревізій у свою роботу, а потім лише створення комміта.

До централізованих систем контролю версій належать такі: CVS, Subversion, Perforce [1, 2, 4].

### Оскільки при використані ЦСКВ для розробки проектів виникає проблема можливості втрати усього проекту при виході з ладу центрального сервера, а також є відсутньою можливість паралельного проведення декількох різних типів робочих процесів була створена порівняно нова розподілена (децентралізована) система керування версіями. ЇЇ вагомою відмінністю від ЦСКВ є те, що при копіювання клієнти отримують не лише останню версію усіх файлів проекту, а й копію усього репозиторія. Це ліквідує можливість втрати усього проекту при виході з ладу централізованого сервера, через який проводилася уся робота. Будь-який клієнтський репозиторій може бути скопійований назад на сервер, оскільки кожного разу, коли клієнт копіює нову версію файлів, створюється копія усієї історії проекту.

Більше того, у переважній кількості таких систем є надана можливість одночасної співпраці з декількома віддаленими репозиторіями, тобто можна розв’язувати різні завдання з різними групами людей в рамках одного проекту паралельно.

Отже, чітко окреслимо переваги РСКВ:

* наявність можливості для кожного розробника працювати з власним репозиторієм;
* наявність можливості для кожного розробника створювати власні гілки;
* можливість багаторазового клонування репозиторія будь-ким;
* можливість існування декількох «центральних» репозиторіїв;
* прийняття рішення щодо злиття гілок здійснюється лише керівником проекту;
* відсутність потреби у мережному з’єднанні для здійснення більшості операцій;
* існування переліку операцій для синхронізації репозиторіїв — передачі або отримання змін з інших репозиторіїв;
* остаточне відокремлення створення коммітів від злиття гілок.

Та відзначимо ряд недоліків розподіленої системи контролю версій:

* відсутність можливості контролю доступу до файлів;
* відсутність загальної нумерація версій файлів;
* потреба у більшій кількості дискового простору;
* відсутність можливості блокування файлів;

До розподілених систем контролю версій належать, наприклад, такі: Git, Mercurial, Bazaar, Darcs [1, 2, 4].

1.2 Огляд різних систем керування версіями

На даний момент існує велика кількість різних систем керування версіями. Одними з найбільш поширених є такі: Git, CVS, Subversion, Perforce, Mercurial та Bazaar. Розглянемо основні можливості кожної з них.

CVS – централізована система керування версіями, що вперше була випущена 1986 року і яка, на відміну від однієї з перших таких систем RCS, має можливість працювати не лише з окремим файлами, а й з цілими проектами. Написана нам мові програмування С.

Основний принцип роботи CVS полягає у тому, що остання версія проекту знаходить у репозиторії сервера. Клієнти за допомогою мережі підключаються до сервера та порівнюють локальну версію проекту з тією, що знаходиться у репозиторії сервера. При наявності відмінностей відбувається загрузка клієнтом цих відмінностей з репозиторія сервера у свій локальний проект. Після внесення потрібних змін та вирішення при потребі конфліктів відбувається загрузка змін в репозиторій сервера.

Використання CVS є доречним при розробці невеликих проектів, що не потребують створення паралельних версій з подальшим їх злиттям.

До переваг цієї системи належать такі:

* можливість одночасної роботи декількох розробників над одним проектом;
* можливість контролю роботи не над одним файлом, а над усім проектом;
* передавання лише змін, а не цілих файлів при їхній загрузці із репозиторія;
* велика кількість зручних графічних інтерфейсів;
* автоматичне встановлення у переважній більшості операційних систем Linux;

До недоліків можна віднести:

* втрата всієї історії змін, що стосується фалу чи директорії, при перенесенні чи перейменуванні цього файлу чи директорії;
* виникнення труднощів при роботі з декількома паралельними гілками одного проекту;
* обмежена кількість шрифтів, що підтримуються;
* передавання на сервер всього файлу, а не лише змін, що були внесені клієнтом;
* необхідність наявності доступу до мережі для зв’язку з репозиторієм [5, 6].

Subversion - це створена у 2000 році сучасна централізована система керування версіями, що вирішує основні проблеми CVS, а саме: перейменування та переміщення файлів і каталогів. Написана на мові програмування С.

Принцип роботи Subversion є дуже схожим до принципу роботи CVS: клієнти копіюють зміни із репозиторія сервера, з’єднуючи їх із власним локальним проектом, а після внесення потрібних змін та вирішення конфліктів, що виправляють вручну, результат знову зберігають у репозиторій сервера.

Для з’єднання із сервером потрібний доступ до мережі.

Простий процес встановлення та підготовки до роботи, широкі можливості цієї системи забезпечують їй одне із провідних місць серед інших систем контролю версій. Для роботи з невеликими проектами, що не потребують створення нових гілок і великої кількості версій, використання Subversion  є цілком доречним.

Переваги Subversion:

* підтримка можливостей CVS та схожість системи команд;
* наявність різних графічних інтерфейсів та зручна роботу через консоль;
* відстеження історії змін файлів, каталогів й після їхнього перейменування чи переміщення;
* висока продуктивність роботи як і з текстовими, так і з бінарними файлами;
* можливість створення дзеркальних копій репозиторія;
* наявність двох типів репозиторія: бази даних та набору звичайних файлів;
* зручний механізм створення міток та гілок;

Недоліки системи:

* зберігання повної копії репозиторія на локальному комп’ютері в прихованих файлах потребує великого об’єму пам’яті;
* виникнення проблем з перейменованими файлами, що були перейменовані одним клієнтом на локальному комп’ютері і одночасно змінені іншим клієнтом та завантажені в репозиторій;
* слабка підтримка процесу злиття гілок;
* виникнення труднощів із повним видаленням інформації про файли, що потрапили в репозиторій [5, 7].

Mercurial – це розподілена система керування версіями, що була розроблена для швидкої та надійної роботи з дуже великими репозиторіями Меттом Макалом одночасно з Git і була вперше випущена у 2005 році. Написана на мові програмування Python та C.

Система першочергово була написана для Linux, а пізніші адаптована для роботи під Windows, Mac OS X і більшість Unix-систем. Перше за все це консольна програма.

Для контролю цілісності даних в репозиторії використовується алгоритм хешування SHA-1, для співпраці між клієнтами використовуються протоколи HTTP, HTTPS чи SSH, дані передаються в стислому вигляді. Також у цій розподіленій системі добре підтримується можливість створення гілок із можливість подальшим їхнім злиттям.

До переваг системи можна віднести такі:

* швидка робота з репозиторієм незалежно від кількості даних у ньому;
* компактне зберігання даних в проіндексованому і стислому вигляді;
* надання можливості одночасної роботи над проектом необмеженій кількості розробників завдяки розподільній моделі;
* швидкість перевірки цілісності репозиторія;
* кросплатформенна підтримка;
* можливість одночасної роботи з декількома гілками проекту;
* простота користування;
* можливість конвертування репозиторіїв інших систем контролю версій, таких як: CVS, Subversion, Git, Darcs, GNU Arch, Bazaar та інших.

Недоліки системи:

* можливість співпадіння хеш-кодів для ревізій із різним вмістом;
* орієнтація на роботу в консолі [5, 8].

Bazaar – це розподілена система керування версіями, що є розроблена при підтримці компанії Canonical Ltd і вперше була випущена у 2007 році. Написана на мові програмування Python, C та Pyrex.

Bazaar, на відміну від Git і Mercurial, є орієнтована на зручний користувацький інтерфейс.

Команди Bazaar дуже схожі на команди CVS,  SVN чи Git.

Велика увага у цій системі є надана роботі з великою кількістю гілок та подальшому їх злиттю. Є можливість роботи з репозиторіями інших систем контролю версія, таких як Subversion чи Git, розпочато роботу над підтримкою Mercurial. Також  Bazaar підтримує роботу як з сервером так і без нього.

Переваги Bazaar:

* + кросплатформенна підтримка;
  + зручний користувацький графічний інтерфейс;
  + швидкість роботи з гілками, наявність засобів для автоматичного злиття змін, можливості перейменування файлі та директорій;
  + можливість роботи з репозиторіями інших систем контролю версій;
  + висока гнучкість як при централізованому, так і при розподіленому використанні;
  + компактне зберігання історії.

Недоліки Bazaar:

* нижча швидкість робота, аніж у Git чи Mercurial;
* необхідність встановлення великої кількості плагінів для повноцінного функціонування системи [5, 9] .

Perforce – це комерційна централізована система керування версіями, що була розроблена компанією Perforce Software і вперше випущена у 1995 році. Ця система дає надає можливість одночасної роботи над декількома проектами за допомогою створення для кожного проекту окремого репозиторія. Perforce написана на С та С++.

Адаптована для роботи під систем Unix, Mac OS X, Microsoft Windows.

Є надана можливість роботи як за допомогою консолі, так і графічного інтерфейсу.

До переваг Perforce належить:

* + кросплатформенна підтримка;
  + можливість одночасної роботи над декількома проектами;
  + можливість створення та легкого керування гілками.

Основним недоліком системи є те, що вона є комерційним продуктом, що є значною перешкодою для розповсюдження даної системи. Здебільшого Perforce використовують великі комерційні компанії, для яких важливою є не лише функціональність, але й сучасна технічна підтримка[5, 10].

1.3 Загальний опис системи керування версіями Git

Створення розподіленої системи керування версіями Git пов’язана з роботою над справді великим відкритим проектом розробки ядра Linux. Проект започаткував творець Linux Лінус Торвальдс для управління розробкою ядра. З 1991 по 2002 рік зміни в код проекту розробки ядра вносили за допомогою патчів та архівованих версій. У 2002 році проект перейшов на РСКВ BitKeeper, проте коли у 2005 році стосунки з компанією, що розробляла BitKeeper, погіршилися, право безкоштовного користування продуктом BitKeeper було скасовано. Це й підштовхнуло розробників Linux створити свою систему керування версіями, беручи до уваги досвід використання BitKeeper.

Нова розподілена система керування версіями мала бути швидкою, підтримувати паралельні розробки, тобто роботу над декількома завданнями одночасно, мати простий дизайн та можливість ефективної роботи з справді великими проектами.

У 2005 році була розроблена розподілена система керування версіями файлі та спільної роботи із ними Git, що є дуже швидкою, справді ефективною для великих проектів та володіє чудовою системою розгалуження, що забезпечує можливість паралельної розробки проекту.

На відміну від переважної більшості систем керування версіями Git ставиться до збереження даних не як до набору файлів і змін, зроблених для кожного з цих файлів в часі, а як до набору зліпків невеликої файлової системи.

Це означає, що при збереженні чергової версії певних файлів проекту Git зберігає зліпок того, як виглядають всі файли проекту на даний момент. Якщо ж певні файли не були зміненими, то здійснюється посилання на раніше збережений файл. Це є однією з основних відмінностей Git від інших систем керування версіями.

Для виконання більшості операцій в Git доступ до мережі є непотрібним, оскільки вся історія проекту є збережена локально на диску комп’ютера. Це також забезпечує практично миттєве виконання усіх операцій, на відміну від централізованих систем, де для виконання кожної операції потрібно проводити звернення до сервера за допомогою мережі.

Також при використанні Git непомітна втрата певних даних є неможливою, оскільки перед збереження кожного файлу Git обчислює контрольну суму, тобто створює рядок, що складається з шістнадцяткових знаків (0-9 і a-f), що називається SHA-1 хеш та обчислюється на основі вмісту файлу або структури каталогу, що є збереженим в Git.

В базі даних Git зберігає всі копії файлів не за назвами, а за хешами їхнього вмісту.

Більше того у Git видалення даних є дуже складним і втрата уже зафіксованих даних є практично неможливою, особливо якщо здійснюється регулярне відправлення змін у віддалений репозиторій.

У Git файли можуть перебувати в одному з трьох станів: зміненому, підготовленому та зафіксованому. Таким чином, у проекті, в якому використовується Git, є наявні 3 частини: робочий каталог (working directory) , область підготовлених файлів (staging area) та каталог Git (Git directory).

До змінених файлів належать ті, що знаходяться у робочому каталозі, але ще не додані до область підготовлених файлів, а отже й не були зафіксовані, тобто не додані безпосередньо до репозиторіяt. До підготовлених файлів належать ті, що були змінені та додані до області підготовлених файлів, тобто є відміченими для включення в наступний комміт. І нарешті до зафіксованих файлів належать такі, які вже були збережені у локальному репозиторії.

Робочий каталог – це одна з копій певної версії проекту. Ці файли є взяти із стиснутої бази даних Git і поміщені на диск для подальшого їх перегляду та редагування.

Область підготовлених файлів – це звичайний файл, що зазвичай є збереженим в каталозі Git і містить інформацію про те, які файли повинні увійти до наступного комміта.

Каталог Git - це найбільш важлива частина Git, місце, де є збережена уся історія проекту, і саме вона копіюється при клонуванні репозиторія з іншого комп’ютера.

Беручи до уваги структуру Git, робочий процес у Git складається з таких базових кроків :

* Зміна файлів у робочому каталозі;
* Додавання файлів, що мають увійти до наступного комміту в область підготовлених файлів;
* Здійснення комміта, після чого файли з області підготовлених файлів є збережені одним коммітом у каталозі Git [4, 11].

1.4 Порівняння Git з іншими системами керування версіями

Для вичерпного опису усіх можливостей, переваг та недолків, підтвердження та більшої аргументації ефективності Git доцільно навести також порівняння цієї системи керування версіями з системами, загальний опис яких був поданий у попередньому підрозділі.

Порівнюючи розподілену систему керування версіями Git з централізованими системами CVS, Subversion та Perforce, одразу можна виділити переваги та деякі недоліки Git, пов’язані з його приналежністю до іншого типу систем контрою версій.

У CVS, Perforce та Subverion повна історія змін проекту є збережені в одному місці - в репозиторії сервера, відповідно є велика загроза втрати даних при виходу з ладу центрального сервера, в той час як у Git кожен користувач при клонуванні репозиторія отримує повну історію проекту локально на своєму комп’ютері, що попереджує можливість легкої втрати даних.

Також локальне розміщення репозиторія Git забезпечує значно вищу швидкість виконання майже усіх операцій у порівнянні з Subverion, Perforce та CVS, оскільки вони виконуються без мережних затримок:

* швидке виконання операції перегляду історії файлу;
* швидке отримання іншої версії файлу;
* швидке злиття гілок;
* швидке збереження змін у репозиторії.

Лише операції клонування віддаленого репозиторія і обмін змінами між локальним та віддаленим репозиторієм потребують зв’язку з мережею, коли у Subversion, Perforce та CVS для виконання кожної операції потрібен доступ до мережі. Цей факт також унеможливлює збереження змін, внесених у файли, в репозиторій при відсутності мережі.

Також у Git, на відміну від Subversion, Perforce та CVS, у яких користувачі спочатку мають провести злиття існуючих версій у свою роботу, а потім лише створити комміт, створення коммітів та проведення злиття є цілком окремими операціями.

Проте з іншого боку розподіленість Git може призвести до несанкціонованого розповсюдження проекту, тому що вона надає можливість створення великої кількості його копій. Також у Git відсутня можливість контролю доступу до файлів.

Тепер розглянемо більш детальне порівняння Git із CVS, Subversion та Perforce.

Git є значно ефективнішою системою керування версіями, аніж CVS, що є підтвердженим значним списком її властивостей.

Перш за все, в Git комміти на відміну від CVS є атомарними. Це означає, що після невдало виконаної операції ніяких змін до стану проекту не буде внесено. У Git частково підтримується перейменування або перенесення файлів чи директорії із збереженням їхньої історії. На відміну від CVS, система Git надає можливість злиття змін, внесених до файлу на одній гілці, у той самий перейменований файл на іншій гілці чи навпаки, проте якщо файл був перейменований на обох гілках, виникне конфлікт, який повинен користувач вирішити вручну; зберігає інформацію про те, які зміни були злиті між якими гілками, а також при об’єднанні гілок зливаються лише ті зміни, що присутні в одній гілці, проте яких немає в іншій. Також у Git робота з гілками є значно легшою, а збереження бінарних файлів ефективніше. У CVS, на відміну від Git, не можна створювати віддалених копій репозиторія та приймати зміни з віддаленого репозиторія, об’єднувати групи зв’язаних змін в один логічний блок, тобто обирати файли, які увійдуть у наступний комміт [6, 12].

Subversion та Git є одними з найбільш поширених систем керування версіями кожна у своїй категорії. Проте Git має значний ряд переваг порівняно з Subversion, що в першу чергу пояснюється їхньою належністю до різних типів систем керування версіями.

Створення та злиття гілок в Git працює значно швидше та ефективніше, аніж в Subversion. В Subversion створення гілки відбувається за допомогою копіювання файлу. В Git створення гілки — це створення нового вузла дерева зі своїми батьками, у цій системі **гілка -** це рухомий вказівник на один із коміттів, тому галуження не є ресурсномістким процесом.

Незважаючи на те, що в Subversion також легко створювати гілки, можуть з’явитися проблеми при виникненні конфліктів у процесі злиття, які потрібно буде вирішувати вручну у змінених файлах.

Вагомими перевагами Git у порівнянні з Subversion є й те, що Git-репозиторій приблизно в 30 разів компактніший репозиторіїв Subversion, а також, як уже було згадано у попередніх розділах, Git зберігає «зліпки» файлів у стиснутому вигляді, в той час як Subversion зберігає лише зміни у файлах; Git забезпечує цілісність файлів завдяки хеша SHA-1 і надає можливість отримувати патчі від інших незалежних розробників через e-mail.

Проте одним із основних недоліків Git у порівнянні з Subversion є те, що ця систем в першу чергу інструмент для роботи за допомогою командної стрічки і не має достатньої кількості зручних графічних інтерфейсів користувача, в тому числі й для операційної системи Windows. Також номери коміттів у Subversion є набагато коротшими та зрозумілішими, нумерація починається від одиниця, в той час як у Git з цією метою використано SHA-1 хеш[7, 13].

Git та Mercurial вважаються двома найкращими представниками розподілених систем керування версіями. Проте їх справді важко порівнювати, оскільки вони обидві поділяють базову ідею розподіленого управління кодом.

До переваг Git у порівнянні з Mercurial можна віднести його вищу швидкість роботи, гнучкість, можливість бути адаптованою до різних технологічних процесів, а також розмір репозиторія в Git є меншим, аніж в Mercurial.

Оскільки Git зберігає зліпки файлів, а не набір різниць між ними, виконання операції знаходження різниць між коммітами виконується динамічніше, аніж в Mercurial.

Git підтримує злиття коммітів з необмеженою кількістю батьків, коли Mercurial дозволяє злиття для коммітів лише з двома батьками.

Також у цих двох систем різний погляд на гілки. Для Mercurial гілка - це якась відмітка, яка прикріплюється до комміта назавжди, є глобальною та унікальною, а у Git гілки, по суті, є лише покажчиками на комміт, що можуть передаватися окремо.

Оскільки Git не орієнтований на розрахунок відмінностей між файлам, то він є ефективнішим у роботі з бінарними файлами. Однак Mercurial, на відміну від Git, краще підтримує Windows, її набагато легше освоїти й ця система краще підтримує можливість перейменування або перенесення файлів чи директорії із збереженням їхньої історії. Також документація Git є неповною, що слугує однією з причин виникнення труднощів при освоєнні Git.

Отже, для більш масштабних проектів з великою кількістю розробників та високою динамікою роботи доцільним буде використання саме Git для контролю робочого процесу, проте для швидкого освоєння використання систем керування версіями та початку роботи з ними кращим все ж надати перевагу Mercurial [8, 11, 14].

Порівнюючи Git та Bazaar, можна стверджувати, що вагомим перевагами Git є значно вища швидкість виконання не лише локальних операцій, але й мережевих, та легша і більш ефективна робота з гілками. Git є більш потужною завдяки бездоганному наборі функцій, які вона надає. Також розмір репозиторія Git є меншим, аніж у Bazaar, й Git є більш ефективним з точки зору збереження даних завдяки їх стиснення у репозиторії. У Git є неможлива непомітна втрата даних за допомогою SHA-1 хеша.

Bazaar, так само як і Mercurial, є системою керування версіями файлового рівня, що зберігає різні версії файлів. Проте Git є зосередженим не на файлах, а безпосередньо на їх вмісті, тобто відбувається збереження лише змінених файлів, а файли з однаковим вмістом зберігаються лише один раз. Також коли для Git є приблизно 8 хостингів для збереження проектів з відкритим вихідним кодом, то для Bazaar їх є лише 3.

Проте Bazaar краще забезпечує централізований робочий процес. А також Bazaar, на відміну від Git і Mercurial, є більщ орієнтована на зручний користувацький інтерфейсі і , як і Mercurial, легша для освоєння [9, 12, 13 ].

У порівнянні з Perforce Git надає можливість вибору файлів, що будуть додані до наступного комміта, можливість клонування проектів з чужих віддалених репозиторіїв та забезпечує кращу роботу з гілками. Проте Perforce більш ефективно працює з бінарними файлами [5, 10, 11].

Отже, розподілена система керування версіями забезпечує найбільш ефективну, зручну та безпечну роботу над проектом. А також беручи до уваги подане вище порівняння Git з різними системами керування версіями, можна чітко визначити ряд основних переваг та недоліків розподіленої системи контролю версій Git.

Основні переваги Git:

* швидкість, гнучкість, потужність;
* попередження можливості втрати всієї історії проекту у зв’язку з виходом з ладу центрального сервера, оскільки при клонуванні репозиторія кожен користувач отримує повну історію проекту локально на своєму комп’ютері;
* швидке виконання практично усіх операцій, що зумовлене локальним розміщенням репозииторія. Лише операції клонування віддаленого репозиторія і злиття локального та віддаленого репозиторіїв потребують зв’язку з мережею. Це забезпечує можливість повноцінної роботи над проектом: створення коммітів, гілок, проведення їх злиття без доступу до мережі.
* компактність Git-репозиторія у порівнянні з іншими системами контролю версій, що зумовлене стисненням файлів, розміщених у репозиторії.
* неможливість легкої втрати уже зафіксованих файлів у репозиторії. Практично кожна операції Git лише додає дані, видалення даних є вкрай рідким випадком. Непомітна втрата певних даних є також неможливою через використання SHA-1 хеша, що обчислюється на основі вмісту файлу або структури каталогу, що зберігається Git.
* можливість виправлення помилок, повернення коммітів до попереднього стану, відновлення видалених коммітів;
* можливість об’єднання зв’язаних змін в один логічний блок,тобто вибору файлів, що входитимуть до наступного комміта, за допомогою їх додавання до області підготовлених файлів(staging area);
* швидке та легке галуження. Якщо в Subversion потрібно пам’ятати, з якою останню версію проводилося злиття, щоб згенерувати правильну команду злиття, то Git робить це автоматично, що попереджує помилки при злитті двох гілок разом. Також інформація про те, злиття яких гілок, коли і ким було проведене, зберігається в історії проекту.

В Git г**ілка** — це іменований комміт, тому галуження не є ресурсномістким процесом.

Основні недоліки Git:

* недостатня кількості зручних графічних інтерфейсів користувача;
* неповнота документації Git;
* значна втрата часу на формування всієї історії конкретного файлу;
* відсутність спеціалізованої команди для перейменування чи перенесення файлу в іншу директорію;
* нездатність системи відстежувати порожні каталоги [15, 16, 17, 18].

2 Структура та основи роботи з Git

2.1 Спосіб збереження даних в Git

Git - це контентно-адресна система, тобто основою цієї системи керування версіями є репозиторій - певна сукупність коммітів, кожен з яких є знімком усіх файлів проекту у певний момент часу, що зберігає дані за принципом «ключ-значення».

У Git є 4 типи об’єктів: об’єкти-блоби, об’єкти-дерева, об’єкти-комміти та об’єкти - мітки.

Фундаментальною одиницею в Git є об’єкти-блоби - сукупність байтів.

Після створення порожнього репозиторія у папці проекту створюється .git директорія, у якій і зберігатиметься уся історія проекту. Git зберігає вміст кожного файлу, що є стиснутим за допомогою zlib, в директорії .git/objects**.**

Вміст знаходиться в файлі, назвою якого є контрольна сума, SHA-1 хеш, що є обчислена на основі заголовку та вмісту файлу. Перші два знаки використані як назва директорії у .git/objects, а інші 38 символів безпосередньо позначають назву файлу. Саме цей файл і є об’єктом-блобом. Після створення об’єкта-блобу, до області підготовлених файлів додається посилання на цей об’єкт. Хеш файлу можна отримати за допомогою команди git hash-object (назва файлу).

Отже, файли в Git зберігаються не за назвою, а за хешем, що є згенерованим на базі їх вмісту. Такий спосіб збереження даних й називають контентно-адресним методом. Однією з переваг цього методу є те, що при намаганні збереження іншого файлу, вміст якого збігається з вмістом уже доданого, за допомогою генерування хеша буде виявлено, що файл з таким вмістом уже існує і відповідно збереження нового файлу не відбудеться. Також Git зберігає стиснуті версії файлів, що допомагає зберегти місце на диску.

Оскільки в Git файли зберігаються не за назвою, а за вмістом, заголовки файлів зберігаються не в об’єктах-блобах, а в об’єктах-деревах.

Дерево в Git – це об’єкт, який може містити сукупність блобів, а також піддерев. Кожний елемент дерева містить режим доступу до файлу, тип файлу, хеш, який відповідає блобу чи піддереву, а також назву файлу. Кожне дерево має свій хеш.

Після введення команди git commit Git сворює об’єкт-комміт, який має власний хеш та є збереженим в репозиторії Git.

Кожен об’єкт-комміт має обов’язково містити такі елементи:

* хеш об’єкта-дерева, який містить усі файли, що є в області підготовлених файлів на момент створення комміта;
* ім’я та е-mail користувача, що створює комміт, а також дата і час створення;
* повідомлення, в якому надана інформація про зміст змін в проекті, які зафіксовані даним коммітом.

Об’єкт-комміт містить хеші батьківських коммітів, що й забезпечує наявність історії проекту в репозиторії. Тобто кожен новий об’єкт-комміт містить хеш щойно створеного об’єкта-дерева, що відображає стан області підготовлених файлів на момент створення комміта та хеші усіх попередніх дерев та блобів. Лише перший комміт проекту не міститиме хеша батьківського комміта.

Також об’єкт-комміт, окрім імені та е-mail користувача, що створює комміт, може містити ім’я та е-mail автора змін. Це можливо у тому випадку, якщо людина, що внесла зміни до проекту, дає доступ до цих змін особі, що є відповідальною за внесення змін до репозиторія проекту [ 4, 18, 19].

Git надає можливість присвоєння об’єкту-комміту не лише хеша, але й певної зрозумілої назви, що і є об’єктом-міткою.

Розрізняють два типи міток в Git: легкі (**lightweight tags**) та анотовані (**annotated tags**).

Легку мітку створюють за допомогою команди git tag (назва мітки)-lw. Вона вказує на поточний комміт і не є окремим об’єктом-міткою Git, оскільки вона є лише посиланням на поточний комміт.

Анотована мітка й є об’єктом-міткою, тобто окремим об’єктом, що містить посилання на певний комміт. Об’єкти-мітки містять не лише хеші коммітів, на які вони вказують, але й повідомлення про мітку, ім’я та e-mail автора мітки, а також дату та час її створення.

Анотовані мітки можуть бути створені як для конкретного комміта, що здійснюється за допомогою команди git tag –a (назва мітки) -m “повідомлення” (хеш комміта), так і для поточного комміта за допомогою команди git tag –a (назва мітки) –m “повідомлення” [4, 20].

Для перегляду списку усіх міток застосовують команду git tag, а для перегляду певної мітки разом із коммітом, на який вона вказує, потрібно застосувати команду git show (назва мітки) (рисунок 2.1).

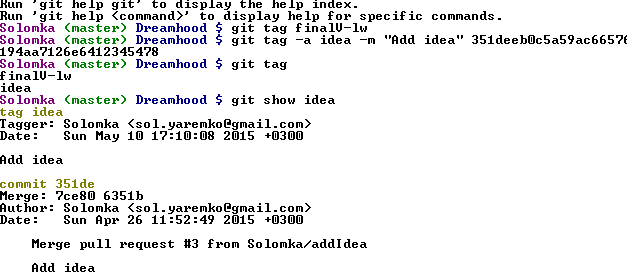


Рисунок 2.1 – Робота з мітками

Усі 3 основні типи об’єктів в Git: блоби, дерева та комміти є збережені як окремі файли в каталозі git/object.

2.2 Налаштування та початок роботи з Git

Перш ніж розпочати роботу з Git потрібно налаштувати середовище для роботи за допомогою утиліти git config, що дозволяє встановлювати параметри, які контролюють усі аспекти роботи з Git. Ці налаштування є збережені в звичайному текстовому файлі, що може бути розміщений в одній із трьох директорій: ~/.gitconfig, /etc/gitconfig, .git/config . Встановлювати параметри в кофігураційному файлі Git можна за допомогою команди git config --{local, global, system}( параметр) ( значення).

Якщо застосувати цю команду, перебуваючи всередині певного репозиторія, то Git застосує --local і змінить налаштування Git лише для роботи з цим репозиторієм в файлі .git/config.

За замовчуванням Git застосовує параметр --global і змінює налаштування Git для конкретного користувача, тоді файл зберігається в директорії ~/.gitconfig.

Вказавши параметр --system, зміни щодо налаштування Git зберігатимуться у файлі /etc/gitconfig і будуть дійсними для всіх користувачів системи та їх репозиторіїв.

Git читає всі ці параметри в такому порядку: system, global, local. Зміни, внесені за допомогою кожного наступного параметра, замінюють зміни, внесені за допомогою попереднього.

Також перед початком роботи потрібно надати Git інформацію про своє ім’я та e-mail, оскільки вони будуть використані у кожному комміті. За це відповідають команди git config --global user.name "назва користувача" та

git config --global user.email (e-mail користувача) (рисунок 2.2).

При потребі зміни налаштування Git для конкретного репозиторія, наприклад, використання іншого імені чи e-mail користувача, потрібно, перебуваючи в ньому, внести відповідні зміни за допомогою команди git config, не вказуючи параметр --global.

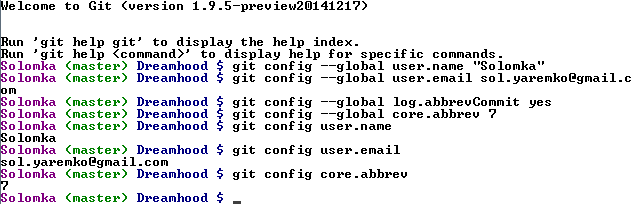
Git отримує інформацію про імя та e-mail користувача із середовища, тому при роботі на різних комп’ютерах достатньо прописати змінну середовища EMAIL, щоб працювати над проектом як відповідний користувач.

Також можна прописати такі змінні середовища, як GIT\_AUTHOR\_NAME та  GIT\_COMMITTER\_EMAIL, що надаватимуть Git інформацію про автора змін та автора коммітів.

При відсутності –m “повідомлення” при створенні комміта Git відкриє автоматично текстовий редактор для створення цього повідомлення. Інформацію про текстовй редактор Git отримує з однієї із змінних середовища GIT\_EDITOR, EDITOR чи VISUAL, які прописує користувач, або інформацію про текстовий редактор можна надати Git за допомогою команди git config --global core.editor ” назва текстового редактора ”.

Для уникнення вводу усього сорока символьного імені комміта при зверненні до нього, можна налаштувати Git таким чином, щоб він ідентифікував кожен комміт за декількома першими цифрами. За це налаштування відповідають команди git config --global log.abbrevCommit yes та git config --global core.abbrev (число) (рисунок 2.2). Це число має бути менше 40 - кількість символів, за допомогою яких можна звернутися до певного комміта. Щоб побачити повну назву комміта, а не скорочену, потрібно ввести параметр git log --no-abbrev-commit.

Перевірити увесь список налаштувань можна за допомогою команди git config --list, для перевірки конкретного налаштуванні із списку потрібно застосувати команду git config (налаштування) (рисунок 2.2) [4, 19].

Рисунок 2.2 – Налаштування середовища для роботи з Git

Для отримання допомоги при використанні Git можна скористатися однієї із трьох команд: git help (команда) чи git (команда) --help. Цими командами можна користуватися навіть при відсутності Інтернету [4].

В Git існує два основних підходи до створення репозиторія. Першим з них є імпортування в Git уже існуючого проетку, а другим - клонування існуючого репозиторія з сервера.

Для створення репозиторія Git в існуючому проекті потрібно скористатися командою git init (рисунок 2.3). Ця команда у каталозі проекту створює підкаталог .git, який містить усі необхідні файли git-репозиторія.

У щойно створеному каталозі .git повинні мітситися такі файли та каталоги: HEAD, branches/, config, description, hooks/, index, info/, objects/, refs/.

У файлі config містяться усі налаштування проекту, в каталозі info знаходиться файл з глобальним фільтром, що ігнорує файли, які користувач не хоче розміщувати в .gitignore, в каталозі hooks розміщені клієнтські та серверні перехоплювачі.

Найважливішими ж складовими щойно створеного репозиторія є файли HEAD , index та каталоги objects і refs. В каталозі objects знаходиться фактично база даних, в refs –посилання на об’єкти- комміти в цій базі даних. Файл HEAD вказує на поточну гілку проекту, а файл index зберігає інформацію області підготовлених файлів.

Проте після виконання цієї команди існуючі файли проекту ще не знаходяться під версійним контролем. Для цього потрібно спочатку додати усі файли до області підготовлених файлів за допомогою команди git add (назва файлу), або для додавання одразу усіх файлів проекту виконати команду git add . , а потім зробити перший знімок стану проекту (рисунок 2.3).

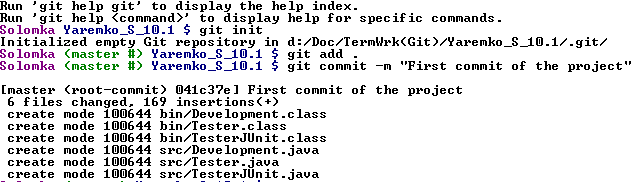


Рисунок 2.3 – Створення репозиторія Git в існуючому проекті

Для отримання копії існуючого репозиторія Git, потрібно скористатися командою git clone (URL репозиторія) (рисунок 2.4 ). Після здійснення цієї команди користувач отримує копію усіх даних, розміщених на сервері, а саме усі версії кожного файлу з історії проекту [4].

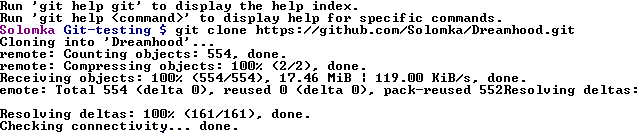


Рисунок 2.4 – Створення копії існуючого репозиторія Git

2.3 Робота з коммітами

Усі файли у робочому каталозі можуть перебувати у двох станах: ті, що відслідковуються Git, і ті, що ні. Файли, що відслідковуються Git – це ті, що входили до складу останнього комміта Вони можуть бути зміненими, незміненимим та доданими до області підготовдених файлів. При внесенні змін до певних файлів, Git розглядатиме їх, як змінені, у порівнянні з їх станом у останньому комміті.

Для з’ясування, які файли були змінені, а які ні, використвують команду git status (рисунок 2.11). Усі файли, які не були внесені до області підготовлених файлів за допомогою команди git add будуть вважатися такими, що не відслідковуються Git, тобто “ untracked files”, а файли, що відслідковуються Git, але були змінені у порівнянні з їх станом у останньому комміті, будуть позначені, як “modified”. Для додавання зміненого файлу до наступного комміта, потрібно спочатку додати його до області підготовлених файлів, таким чином переспрямувати посилання на новий об’єкт - змінений файл, а потім виконати команду git commit. Проте, якщо спочатку файл був доданий до області підготовлений файлів, а потім внесено зміни до нього у робочій директорій і здійснено команду git commit, то до комміта цей файл увійде із тими змінами, що були внесені до нього на момент здійснення команди git add, а не з тими змінами, що були винесені після виконання цієї команди. Для збереження останніх змін у комміті у такому випадку, потрібно ще раз здійснити додавання файлу до області підготовлених файлів.

При наявності файлів, які не потрібно додавати до репозиторія і бачити у списку файлів, що не відслідковуються, у Git наявна можливість створення спеціального файла .gitignore із списком шаблонів, що відповідають таким файлам. Найчастіше до них належать ті, що генеруються автоматично [4, 19].

Для отримання більш детальної інформації, про зміни, що були внесені до файлів проекту використовують команду git diff. Ця команда надає інформацію про те, які рядки були видалені, а які додані до файлу.

Команда git diff порівнює вміст файлу у робочій директорії з його вмістом у області підготовлених файлів (рисунок 2.5). Для порівняння стану файлу, внесеного до області підготовлених файлів, з його станом у останньому комміті, використовують команду git diff – staged. Команда git diff (назва комміта) надає інформацію про зміни у робочій директорії в порівнянні з певним іменованим коммітом. Команда git diff (хеш-код) (хеш-код) показує відмінності між двома коммітами, деревами чи блобами за їх відповідними хеш-кодами.

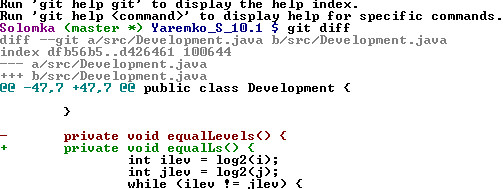


Рисунок 2.5 – Застосування команди git diff

Для додавання до області підготовлених файлів файлу з частиною винесених змін до нього у робочій директорії, використовують команду git diff –p . Використання цієї команди є доречним, коли до файлу були внесені зміни, які варто подати у двох різних коммітах, щоб при перегляді історії проекту, зміни, що були внсені до нього кожним коммітом, були більш зрозумілі та структуризовані [4, 19].

Також команда git add –u відповідає за додавання до області підготовлених файлів усіх змінених та видалених файлів у робочому каталозі, але не нових файлів, а команду git add –A використовують при заміни робочого дерева іншою версією коду для додавання його до робочого дерева та області підготовлених файлів. При використанні цієї команди до області підготовлених файлів додаються також нові файли [19].

Для видалення файлів із Git, потрібно видалити його із області підготовлених файлів за допомогою команди git rm (назва файлу), а потім зробити комміт проекту. Після здійснення цієї команди файл буде видаленим також з робочого каталогу (рисунок 2.6). При видаленні файлу з робочого каталогу після застосування команди git status цей файл буде відображений, як змінений, але не доданий до області підготовлених файлів. Лише після застосування команди

git rm (назва файлу) він буде вилученим з області підготовлених файлів. Якщо файл був уже зміненим та доданим до області підготовлених файлів, то для його видалення потрібно застосувати додатковий параметри –f: git rm –f (назва файлу).

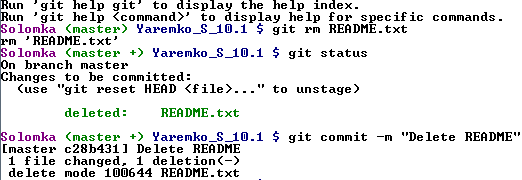


Рисунок 2.6 – Видалення файлів в Git

Для видалення файлу з області підготовлених фалів, залишивши його в робочому каталозі, потрібно застосувати команду git rm - -cached (назва файлу) [4].

При перейменовуванні файлу Git не зберігає ніякої інформації про те, що відбулося його перейменування, тому для того, щоб зберегти цю інформацію в історії проекту при створенні комміта потрібно залишити повідомлення про виконання цієї зміни. Для здійснення перейменування в Git застосовують команду git mv (стара назва файлу) (нова назва файлу), після якої потрібно зробити новий комміт проекту для збереження зміни. Якщо після виконання команди git mv (стара назва файлу) (нова назва файлу), викликати команду git status, то файл буде помічений як перейменований: renamed (рисунок 2.7). Насправді, у Git немає окремої функції перейменування. Викликаючи команду git mv (стара назва файлу) (нова назва файлу), виконуться ті самі дії, що й при послідовному виклику таких команд: git rm (стара назва файлу), git add (нова назва файлу) [4, 19].

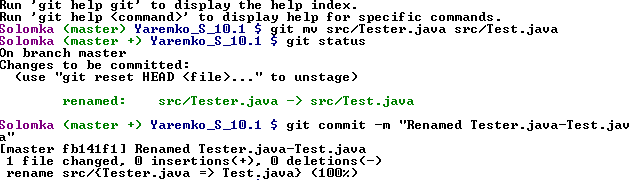


Рисунок 2.7 – Перейменування файлів в Git

Для фіксації змін, внесених до проекту, використовують команду git commit. Варто зазначити, що до нового комміта зі змінами ввійдуть лише ті файли, що були попередньо додані до області підготовлених файлів. Після виклику цієї команди, у текстовому редакторі буде подана інформація про змінені файли, які ввійдуть до складу комміта. Це повідомлення можна замінити на власне. Після закриття текстового редактора буде створено новий комміт.

Також комміт можна стоврити за допомогою команди git commit –m ”повідомлення”, таким чином відразу вказати повідомлення, не відкриваючи текстового редактора.

Після стоврення комміта Git буде подана інформація про те, на якій гілці був створений комміт, контрольна сума комміта, скільки файлів було змінено у порівнянні з минулим коммітом та інформація про те, скільки було додано, а скідьки видалено стрічок (див. рисунок 2.3).

За допомогою команди git commit –a можна не добавляти файл, який уже відслідковується Git, до області підготовлених файлів перед створенням комміта, Git це виконає автоматично.

При необхідності додати ще один файл до уже створеного комміта, потрібно виконати послідовно такі команди: git add (назва файлу), git commit

--amend. В результаті їх виконання до останнього комміта буде доданий ще один файл. Після введення цієї команди відкриється графічний редактор із старим повідомлення до комміта, яке можна буде відредагувати (рисунок 2.8) [4, 19].

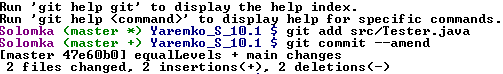


Рисунок 2.8 – Застосування команди git commit --amend

Команда git commit --amend дає можливість зміни лише останнього комміта. Для зміни декількох коммітів з історії проекту потрібно застосувати команду git rebase -i HEAD~n, де n – кількість останніх коммітів, які потрібно змінити (рисунок 2.9). Після виконання цієї команди у текстовому редакторі буде подано список коммітів у порядку, протилежному виводу команди git log, тобто останній зроблений комміт буде й останнім у списку (рисунок 2.10). Для редагування коммітів потрібно замінити першу колонку виводу для кожного комміта у текстовому редакторі на одну із таких команди:

* pick - залишити комміт без змін;
* reword - змінити лише повідомлення відповідного комміта;
* еdit - змінити вміст, а також за бажанням повідомлення комміта. Після виклику цієї команди, Git надає користувачу можливість внесення бажаних змін до проекту, а потім виконання команди git commit --amend для додавання цих змін до відповідного комміта із можливістю редагування повідомлення. Після цього потрібно викликати команду git rebase --continue для завершення здійснення редагування.
* squash - об’єднання декількох коммітів в один . Для цього потрібно до першого комміта застосувати команду pick, а до решти – squash. Git об’єднає повідомлення всіх коммітів в одне повідомлення з наданням можливості редагування.
* fixup - об’єднання декількох коммітів в один. Для цього потрібно позначити перший комміт як pick, а всі решта – fixup. Git видаляє повідомлення поточного комміта при створенні спільного повідомлення.

Якщо внесення змін до певного комміта спричинило виникнення конфлікту, то після його вирішення потрібно застосувати спочатку команду git add, а потім git rebase --continue для продовження процесу внесення змін до ряду коммітів (рисунок 2.9). Для переривання цього процесу і повернення усіх коммітів до поточного стану потрібно застосувати команду git rebase --abort [19, 21].

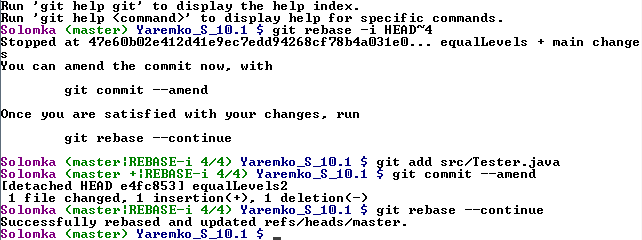


Рисунок 2.9 – Зміна декількох коммітів за допомогою команди

git rebase -i HEAD~n

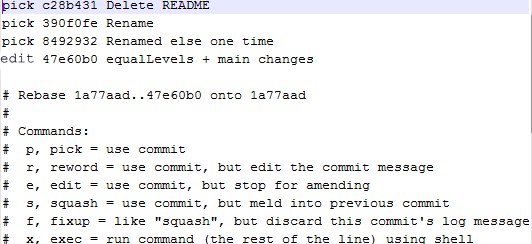


Рисунок 2.10 - Список коммітів у текстовому редакторі після застосування команди git rebase -i HEAD~n

За повернення області підготовлених файлів до стану останнього комміта відповідає команда git reset HEAD. Вона відміняє всі зміни, що були внесені в область підготовлених файлів за допомогою команди git add. Після здійснення цієї команди, Git надає інформацію про змінені файли, які не додані до області підготовлених файлів. Можна вказати назву конкретного файлу, який потрібно повернути до стану останнього комміта в області підготовлених файлів і таким чином стан інших файлів у цій області залишити незмінним [19].

Для відміни змін, що були внесені до файлу в робочому каталозі можна скористатися командою git checkout -- (назва файлу) (рисунок 2.11). Після здійснення цієї команди файл повернеться у той стан, в якому він перебував на момент створення останнього комміта. Проте зміни, що були внесені до цього файлу, уже ніколи не можна буде відновити.

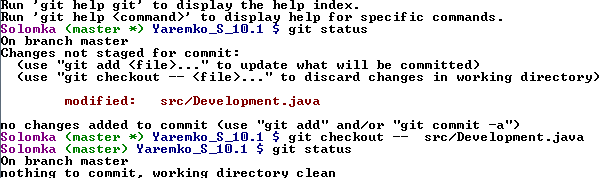


Рисунок 2.11 – Застосування команди git checkout -- (назва файлу)

Для відміни щойно створеного комміта використовується команда git reset HEAD~. Після її виконання усі зміни у файлах, що входили до комміта, стають знову не доданими до області підготовлених файлів (рисунок 2.12) [4, 19].

reset.png

Рисунок 2.12 – Застосування команди git reset HEAD~

При потребі відміни декількох останніх коммітів використовують ту саму команду, що й при відміні останнього, проте вказується кількість останніх коммітів, які потрібно відмінити: git reset HEAD~n, де n – кількість коммітів, що потрібно відмінити. У цьому випадку Git надає декілька додаткових можливостей при використанні певного параметра:

* git reset --mixed хеш-код комміта - область підготовлених файлів приймає стан відповідного комміта, проте усі зміни, що були здійснені у робочому каталозі залишаються;
* git reset --soft хеш-код комміта - область підготовлених файлів і робочий каталог залишаються без змін, проте до них будуть додані всі зміни з історії, починаючи від вказаного комміта до даного;
* git reset --merge хеш-код комміта - усі файли, що не були додані до області підготовлених файлів залишаються без змін, а всі інші файли переходять у стан, в якому вони перебували у відповідній версії проекту.
* git reset --hard хеш-код комміта - усі файли в робочому каталозі та область підготовлених файлів набуваються стану, в якому вони перебували у відповідному комміті.

Також Git надає можливість відміни змін певного комміта за допомогою створення нового без внесення змін до історії проекту. Це можна здійснити за допомогою команди git revert (хеш-код комміта). Після її виконання стан проекту буде повернений до стану комміта, що передував відміненому[19, 21].

2.4 Перегляд історії проекту

Для перегляду всієї історії проекту використовують команду git log, що виводить список коммітів в оберненому хронологічному порядку. Git також має набір спеціальних параметрів для форматування виводу результатау цієї команди. Розглянемо лише декілька з них.

Для виводу змін, що були внесені до проекту кожним коммітом, до команди git log потрібно додати параметр –p, а для отримання лише короткої статистики змін, до цієї команди потрібно додати параметр --stat, що забезпечить вивід для кожного комміта списку змінених файлів, їх кількість, а також кількість доданих та видалених рядків у кожному файлів[22].

Git також надає можливість зміни формату виводу списку коммітів за допомогою параметра --pretty. Залежно від значення, наданому цьому параметру, можна контролювати кількість та оформлення інформації, виведеної Git для кожного комміта. Наприклад, надання цьому параметру значення oneline: git log --pretty=oneline забезпечить вивід кожного комміта одним рядком (рисунок 2. 13). Проте одним із найбільш корисних значень цього параметра є format, який дозволяє користувачу створювати власний формат виводу даних кожного комміта.

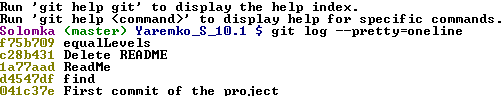


Рисунок 2.13 – Застосування команди git log --pretty=oneline

Для виведення графа – історії галуження та злиття в проекті, потрібно до команди git log додати параметр --graph.

Git надає можливість контролю кількості виведених коммітів, яка задається після команди git log. Також можна вивести список коммітів, створених впродовж певного періоду за допомогою параметрів --since та --until, доданих до команди git log, надавши їм значення відповідного часового проміжку чи дати.

Більше того у Git є можливість фільтрації коммітів за певним критерієм чи декількома критеріями одразу. Для фільтрації коммітів за більше, аніж одним критерієм, потрібно застосувати параметр --all-match. Наприклад, для одночасної фільтрації виводу коммітів певного автора та за певним ключовим словом повідомлення потрібно застосувати комануд: git log --author (ім’я автора) --grep (ключове слово) --all-match [4, 19].

Також за допомогою команди git log –g можна побачити усі операції, що були виконані на кожній гілці проекту, зокрема commit, pull, checkout, merge, amend [19].

Для візуалізації історії коммітів проекту можна застосувати програму gitk, що є доданою до Git, ввівши в командній стрічці gitk. У ній буде відображено увесь список коммітів, вибравши один із них, можна побачити усі зміни, що були внесені до проекту цим коммітом [4].

2.5 Галуження в Git

Однією із ключових переваг Git у порівнянні з іншими системами керування версіями є висока швидкість виконання операції галуження. Це пов’язано з тим, що гілка в Git є простим файлом з сорока символьною контрольною сумою комміта, на який він вказує.

Гілка в Git - це рухомий указник, який завжди вказує на останній комміт поточної гілки. При створенні нового репозиторія та початку роботи з ним Git надає гілку за замовчуванням master. Створення нової гілки - це створення нового указника на останній комміт проекту**.** Спеціальний указник HEAD вказує на поточну гілку проекту.

За створення нової гілки відповідає команда git branch (назва гілки), проте для початку роботи з нею потрібно застосувати команду git checkout (назва гілки), що відповідає за переміщення указника HEAD на відповідну гілку (рисунок 2.14). Після виконання цих двох команд усі подальші комміти проекту будуть збережені на новій гілці, а поточна гілка master вказуватиме на останній комміт перед здійсненням команди git checkout, що забезпечує можливість роботи розробника над декількома різними завданнями одночасно [4].

Для створення та одразу початку роботи з новою гілкою існує команда git checkout –b (назва гілки). При потребі здійснення відгалуження з комміта, відмінного від поточного, користуються командою команда git checkout –b (назва гілки) (хеш-код потрібного комміта). Також команда git branch –m (стара назва) (нова назва) відповідає за перейменування гілки, а команда git branch -d (назва гілки) - за її видалення (рисунок 2.14).

Для перегляду списку усіх гілок репозиторія застосовують команду git branch, а команда git branch --all виводить список усіх гілок проекту як локальних, так і віддалених (рисунок 2.14). Команди git branch --merged та git branch --no-merged відповідають за надання списку гілок, що були об’єднані чи відповідно не об'єднані з поточною. Для видалення гілки, яка не була злита з ніякою іншою, застосовують команду git branch -D (назва гілки).

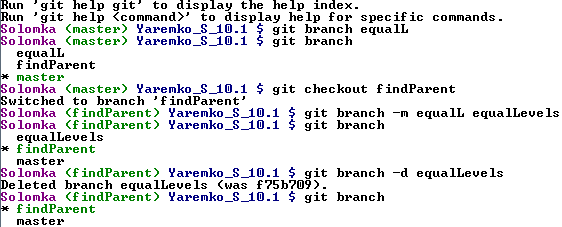
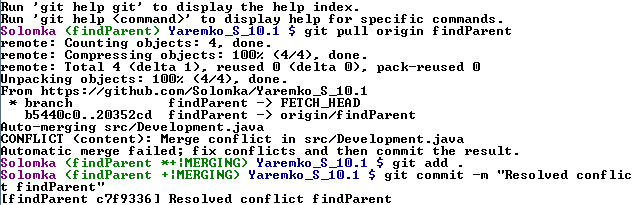


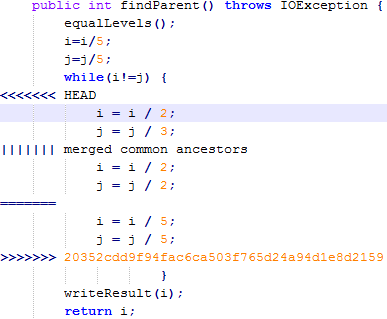
Рисунок 2.14 – Робота з гілками в Git

Для злиття двох гілок спочатку потрібно перейти на гілку, в яку потрібно додати зміни з іншої, а потім виконати команду git merge (назва гілки), після чого видалити останню з них за допомогою команди git branch –d (назва гілки), оскільки тепер два указники вказують на один і той самий комміт [19].

При злитті двох коммітів, один з яких є прямим нащадком іншого, відбувається лише переміщення указника на відповідний комміт, оскільки відсутні зміни для об’єднання. Цей процес називається «fast - forward merging». В іншому випадку відбувається створення нового комміта, який має більше одного предка і який називається коммітом - злиттям. Система Git самостійно визначає найкращого спільного предка для двох коммітів [18, 19].

При спробі злиття двох гілок, а також при оновленні гілки змінами із віддаленої, у яких були внесені різні зміни до однієї частини файлу, Git повідомляє про виникнення конфлікту, за наявності якого злиття гілок є неможливим. Команда git status надає інформацію про файли, у яких виник цей конфлікт, а команда git diff показує конкретні відмінності у цих файлах. Також Git додає до відповідних файлів маркери, за допомогою яких можна легко вручну вирішити конфлікти, після чого виконати команди git add і git commit для закінчення процесу злиття (рисунок 2.15).

а)



б)

Рисунок 2.15 – Конфлікти при об’єднанні змін в Git:

а - застосування команд для вирішення конфлітку, б - файл із відповідними маркерами для вирішення конфлікту

Для того, щоб залишити варіант файлу з однієї із двох гілок при їхньому злитті, потрібно застосувати команду git checkout (назва відповідної гілки) (назва файлу), після чого виконати команду git add (назва зміненого файлу) для його долучення до області підготовлених файлів та остаточного завершення процесу злиття.

При відміні процесу злиття і уникнення потреби вирішення конфлікту існує команда git merge --abort. За відміну злиття після його здійснення відповідає команда git reset --hard ORIG\_HEAD[ 18, 20].

Для підвищення продуктивності роботи над великими проектами варто притримуватися певного підходу до роботи з Git. Гілка master – повинна містити лише стабільний готовий код програми. Для розробки та тестування використовувати гілку develop, що містить зміни, що пройшли тести і не мають помилок. ЇЇ використовують для об’єднання вирішених задач з тематичних гілок, кожна з яких є тимчасовою і створеною винятково для вирішення якогось конкретного завдання.

Також у Git виділяють віддалені гілки - відсилки на стан гілок у віддалених репозиторіях, що надають інформацію про останній комміт, на який вказувала гілка при останньому з’єднанні з віддаленим репозторієм. Назва цих гілок має такий вигляд: (назва віддаленого репозиторія)/(назва гілки).

При внесенні змін до локального репозиторія відбувається переміщення лише локальної гілки, віддалена гілка змінюється лише при під'єднанні до віддаленого репозиторія [4].

Для синхронізації змін на віддаленому та локальному репозиторіях застосовують команду git fetch (назва віддалного репозитрія), що забирає усі комміти з віддаленого репозиторія, яких немає у локальному, і оновлює позицію указника (назва віддаленого репозиторія)/(назва гілки). Після виконання цієї команди користувач отримує лише незмінний указник на віддалену гілку, для їхнього додавання до робочої гілки проекту потрібно застосувати команду git merge (репозиторій)/(гілка).

При потребі надсилання на віддалений сервер нової гілки використовують команду git push (віддалений репозиторій) (назва гілки).

Для створення власної нової гілки на основі віддаленої потрібно здійснити команду git checkout –b (назва гілки) (назва віддаленої гілки ). Такі гілки називаються відстежуючими гілками. При виконанні команд git push та git pull, перебуваючи на цих гілках, Git надсилатиме зміни автоматично в відповідний віддалений репозиторій та на потрібну гілку.

Для виведення списку усіх відстежуючи гілок та відповідних їм гілок у віддаленому репозиторії існує команда: git branch –vv.

Команда git push (віддалений репозиторій): (гілка) дає можливість видалення гілки у віддаленому репозиторії [4, 19, 20].

Для уникнення виникнення конфліктів при злитті тематичних гілок з основною, здійснюють спочатку злиття основної гілки в тематичну. Для цього потрібно створити додаткову гілку, що вказує на останній комміт основної гілки та використати команду git cherry-pick (хеш-коди коммітів тематичної гілки), що виконує копіювання відповідних коммітів в іншу гілку. Після цього потрібно переспрямувати першу тематичну гілку на створену додаткову за допомогою команди git reset - -hard (назва додаткової гілки), перебуваючи на відповідній тематичній гілці, а потім видалити додаткову гілку , виконавши команду git branch –D (назва додаткової гілки).

Оскільки цей процес є справді складним, у Git існує зручніший спосіб уникнення вирішення конфліктів при об’єднанні змін двох гілок, що називається переміщенням (rebase). Команда git rebase (назва гілки) копіює усі зміни, що були зроблені на одній гілці в іншу гілку. Після виконання цієї команди достатньо перейти на гілку, у яку додавалися зміни, та виконати команду git merge, після чого дві гілки вказуватимуть на один комміт - останній комміт проекту. Кінцевий результат об’єднання та переміщення буде однаковим, проте зміниться історія проекту. При переміщенні, на відміну від об’єднання, всі зміни в історії проекту будуть відображені як послідовно внесені, коли насправді вони вносилися паралельно.

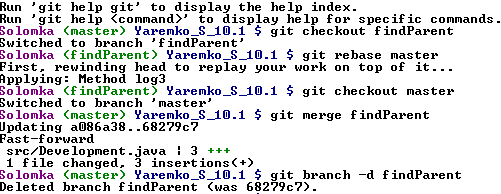


Рисунок 2.16 – Здійснення переміщення в Git за допомогою команди

git rebase

При виникненні потреби переміщення змін з тематичної гілки, яка є відгалуженням іншої тематичної, в основну застосовують команду git rebase - -onto (гілка, на яку здійснюється переміщення) (перша тематична гілка) (друга тематична гілка). Також команда git rebase (основна гілка) (тематична гілка) надає можливість переміщення змін з тематичної гілка на основну без попереднього переходу на неї.

Після здійснення команди rebase можна видалити гілки, з яких переміщувалися зміни, за допомогою команди git branch –d (назва гілки) із збереженням усієї історії проекту (див. рисунок 2.16).

Проте не можна робити переміщення коммітів, що були надіслані у віддалений репозиторій, оскільки це може значно ускладнити паралельну роботу інших розробників над цим проектом [4].

Ще однієї зручною командою Git є git stash. Якщо в процесі роботи над проектом виникає потреба оновити локальний репозиторій змінами з віддаленого, або розпочати роботу на іншій гілці, але при цьому не робити комміт незавершеної роботи над проектом, доцільним є використання цієї команди, оскільки вона приховує усі внесені зміни та повертає проект до стану останнього комміта. Це дає можливість виконати команду git pull чи git checkout , а потім продовжити роботу над змінами проекту, виконавши команду git stash apply (рисунок 2.17).

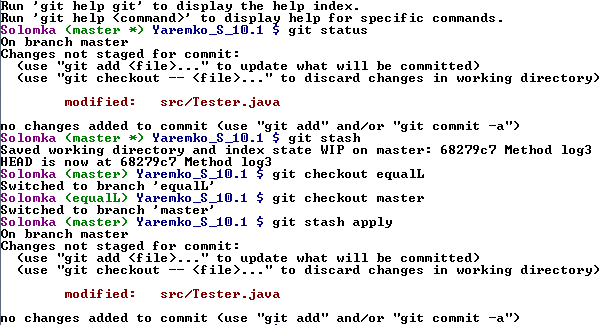


Рисунок 2.17 – Застосування команди git stash

Також є певний список параметрів, які можна застосовувати до команди git stash:

* git stash list – вивід списку змін;
* git stash show – вивід останніх змін;
* git stash drop – видалення останніх змін у списку;
* git stash pop – застосування змін до поточної версії проекту та видалення останніх змін зі списку;
* git stash clear – видалення усіх змін зі списку;

Отже, Git забезпечує ефективну спільну роботу над різними завданнями проекту одночасно без ризику внесення змін, що можуть порушити коректну роботу програми, завдяки зручному та швидкому процесу роботи з гілками, а також зручний контроль змін за допомогою гнучкої роботи з коммітами та можливості форматованого виводу історії проекту [20, 21].

3 Робота з Git на сервері

3.1 Огляд мережних протоколів

Для організації ефективної спільної роботи над проектом в Git необхідним є створення віддаленого репозиторія, за допомогою якого розробники здійснюватимуть спільне внесення змін до проекту.

Віддалений Git-репозиторій - це репозиторій без робочого каталогу, що містить лише .git каталог локального репозиторія проекту, оскільки його використовують лише для обміну змінами між розробниками.

Перш ніж розпочати роботу з репозиторієм на сервері, потрібно вибрати протокол, за допомогою якого буде здійснено доступ до віддаленого репозиторія.

Git працює з чотирма мережними протоколами: локальним, SSH, Git та HTTP.

Локальний протокол є базовим. При його використанні віддаленим репозиторієм є інший каталог на диску. Цей протокол використовують, якщо усі члени команди мають доступ до однієї файлової ситсеми.

Використання локального протоколу є доцільним, якщо є спільна файлова система. Також отримання змін з робочого репозиторія розробника є значно швидшим, аніж їх отримання з віддаленого сервера, після їх додавання туди іншим розробником. Проте спільний доступ зазвичай важче налаштовувати і отримати із різних місць, аніж мережний доступ, а також локальний репозиторій є швидким лише у тому випадку, якщо є швидкий доступ до даних.

Найбільш широко використовуваним протоколом є SSH-протокол, оскільки його легко налаштовувати і використовувати. Також це єдиний протокол, що надає право на читання та запис даних.

Для клонування репозиторія, використовуючи SSH-протокол, вказують ssh:// в URL цього репозиторія: git clone ssh://user@(сервер)/(назва проекту).git, або ж git clone user@(сервер):(назва проекту).git .

Використання SSH-протоколу є необхідним, коли потрібний авторизований доступ для запису змін у віддалений репозиторій. Також доступ за допомогою цього протоку є безпечним, оскільки усі дані, що передаються по авторизованих каналах, є зашифрованими, також перед передачею усі дані стають максимально компактними. Проте використання SSH-протоколу не дозволяє анонімний доступ до проекту. Також використання цього протоколу є недоцільним для проектів з відкритим вихідним кодом, оскільки навіть для роботи в режимі лише для читання, користувачі повинні мати доступ через SSH-протокол.

Git - протокол є одним із найбільш швидких протоколів, проте одним із найважчих для налаштування через необхідність запуску Git-демона для проекту. Цей протокол використовує той самий механізм передачі даних, що й SSH-протокол, але без шифрування даних та авторизованого доступу. Саме тому варто використовувати його разом із SSH-протоколом для доступу до проекту. Розробникам, що мають право на запис змін, надають доступ до проекту через SSH-протокол, а тим, хто має право лише на читання даних, через Git-протокол.

HTTP/HTTPS протокол є також широко використовуваним та легким для налаштування. Для цього необхідно помістити репозиторій без робочого каталогу в каталог із HTTP документами і налаштувати перехолювач post-update, що забезпечує коректне виконання операцій клонування віддаленого репозиторія та синхронізації змін у віддаленому та локальному репозиторія через HTTP. Перевагою використання цього протоколу є й те, що він надає доступ до проекту лише в режимі для читання з можливістю авторизації клієнтів. Також використання HTTP - протоколу не потребує багато ресурсів сервера. Проте одним із недоліків цього протоколу є його достатньо низька ефективність для клієнта. Операції клонування та синхронізації змін виконуються значно довше, а навантаження на мережу є значно більшим, аніж при використанні будь-якого іншого протоколу [4].

3.2 Створення та робота віддаленим репозиторієм

Для розміщення локального репозиторія на сервері, спочатку необхідно склонувати цей репозиторій у новий «чистий», що не містить робочого каталогу, оскільки зміни до репозиторія додаватимуть за допомогою каманди git push, а на за допомогою внесення змін у робочому каталозі та створенні відповідних коммітів. Це можна здійснити, виконавши команду git clone --bare (назва проекту) (назва проекту).git. Назва каталогу з таким репозиторієм закінчується на .git.

Маючи доступ через SSH-протокол до налаштованого сервера, розмістити «чистий» репозиторій в певному каталозі на сервері можна за допомогою команди scp -r (назва проекту).git user@(сервер): (каталог). Після виконання цієї команди інші розробники матимуть змогу скопіювати цей репозиторій за допомогою команди: git clone user@(сервер):(каталог)/ (назва проекту).git.

Для автоматичного надання права на запис, тобто відправлення змін до репозиторія для групи розробників, потрібно в каталозі проекту на сервері виконати команду: git init --bare --shared.

Щоб надати кожному розробнику комнди доступ до сервера через SSH-протокол і можливість внесення змін до репозиторія на цьому сервері, потрібно створити для кожного з них обліковий запис. Проте більш ефективним способом надання доступу є створення одного користувача «git» на сервері та додавання відкритих SSH-ключів розробників, які кожен із них повинен згенерувати, викликавши програму ssh-keygen, до файлу ~/.ssh/authorized\_keys користувача «git». У такому випадку усі розробники матимуть доступ до сервера через одного користувача «git».

При потребі створення пустого «чистого» репозиторія на сервері, куди розробники зможуть надіслати першу версію проекту, необхідно виконати команду git --bare init у відповідному каталозі на сервері. Після здійснення цієї команди розробники зможуть надсилати версії проекту у цей репозиторій, додавши його як віддалений за допомогою команди git remote add origin git@(назва сервера):(каталог)/(назва проекту).git.

Отже, для того, щоб склонувати віддалений репозиторій, незалежно від його розміщення потрібно виконати команду git clone (URL віддаленого репозиторія) (див. рисунок 2.4). Після виконання цієї команди, розробник отримує повну копію репозиторія, відбувається створення посилань на локальні копії віддалених гілок, про роботу з якими було згадано у попередньому розділі. А також відбувається створення та спрямування указника HEAD на початкову гілку, що є копією активної гілки віддаленого репозиторія [4].

Проте якщо виникає потреба для багатьох розробників, що мають доступ до однієї файлової системи, склонувати один репозиторій, або одному розробнику для вирішення певної задачі потрібно часто клонувати один репозиторій, а його клонування займає значний відрізок часу, Git надає можливість створення однієї локальної копії віддаленого репозиторія з подальшим її клонуванням. Для здійснення цього спочатку потрібно виконати команду git clone --bare (URL репозиторія) refrep, що забезпечить створення «чистого» репозиторія – копії віддаленого, який надалі використовуватимуть як репозиторій - посилання. Після цього здійснюється клонування віддаленого репозиторія ще раз, але із вказуванням refrep як посилання: git clone - -reference refrep (URL репозиторія) (рисунок 3.1). Інші розробники, що мають доступ до цієї ж файлової системи, можуть використовувати цю команду для копіювання репозиторія, що буде виконана миттєво, оскільки усі необхідні об’єкти уже скопійовані у репозиторії - посиланні. Проте, після внесення змін до віддаленого репозиторія, даний локальний репозиторій - посилання потрібно оновлювати відповідними змінами, виконавши команду git fetch - -all [19, 20].

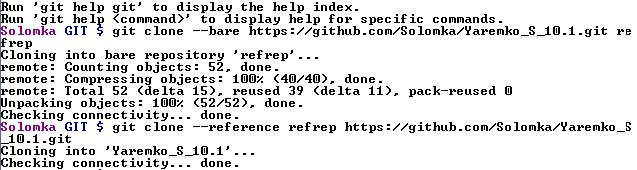


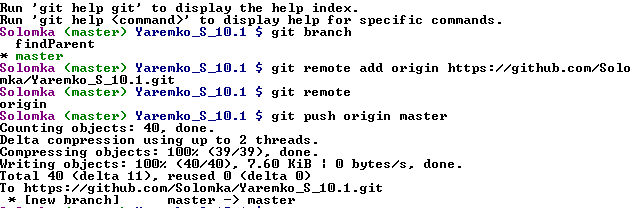
Рисунок 3.1 – Процес створення локальної копії віддаленого репозиторія з подальшим її клонуванням

Для перегляду списку віддалених репозиторіїв, що існують для даного проекту, потрібно виконати команду git remote, а за додавання віддаленого репозиторія до проекту відповідає команда git remote add (назва віддаленого репозиторія) (URL віддаленого репозиторія) (рисунок 3.2) [4].

Після внесення потрібних змін до проекту та створення коммітів, додати їх до віддаленого репозиторія можна за допомогою команди git push (назва віддаленого репозиторія )(назва гілки) (рисунок 3.2). Виконавши цю команду, Git оновлює потрібні віддалені відсилки та надсилає потрібні об’єкти для їх оновлення. Як уже було сказано у попередньому розділі, для оновлення стану локального репозиторія відповідно до віддаленого, потрібно скористатися командою git fetch (назва віддалного репозитрія), що забирає усі комміти з віддаленого репозиторія, яких немає у локальному, і оновлює позиції указників (назва віддаленого репозиторія)/(назва гілки).

Рисунок 3.2 – Робота з віддаленим репозиторієм в Git

Для забрання усіх змін з віддаленого репозиторія, яких немає у локальному, та додавання їх до робочої гілки проекту, потрібно виконати команду git pull (назва репозиторія) (назва гілки) (див. рисунок 2.15 а) ). Ця командам виконує послідовно дві команди: git fetch (назва репозиторія ), що оновлює позиції указників на локальні копії віддалених гілок, та git merge (репозиторій)/(гілка), що безпосередньо здійснює додавання нових віддалених гілок до робочої гілки [ 20].

 У Git є також можливість налаштування анонімного доступу на читання до репозиторія, запустивши веб-сервер та вказавши кореневим каталогом для документів каталог, у якому розміщений репозиторій, і налаштувавши перехоплювач post-update, можливість налаштування веб-візуалізатора за допомогою CGI ( «спільний інтерфейс шлюзу») сценарія, що має назву GitWeb. Також у випадку наявності великої кількості користувачів для управління файлом authorized\_keys та реалізації простішого контролю доступу є доцільним використання Gitosis - набору сценаріїв, що відповідають за контроль доступу до репозиторія. Встановлення ж системи Gitolite надасть можливість надати права доступу не лише до всього репозиторія, а й до окремих гілок та міток [4].

3.3 Опис Git хостинг – сайту GitHub

Для того, щоб не встановлювати власний Git-сервер, можна скористатися можливістю розміщення Git-проектів на зовнішніх хостинг-сайтах. Розміщення проектів на хостинг-сайтах має значну кількість переваг у порівнянні з власними Git - серверами, оскільки, на них значно швидше налаштувати та запустити проект, зникає потреба у підтримці сервера та дані мають вищий рівень безпеки.

Сьогодні є велика кількість хостинг-сайтів для розміщення Git репозиторіїв, наприклад, Assembla, Beanstalk, Bitbucket та GitHub. Проте одним із найбільших [веб-сервісів](http://uk.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D1%96%D1%81) для спільної розробки [програмного забезпечення](http://uk.wikipedia.org/wiki/%D0%9F%D0%97) сьогодні є GitHub, у якому нараховано більше 9 мільйонів користувачів, 21 мільйона репозиторіїв та який підтримує більше 200 мов програмування. Цей сервіс надає можливість створення як безкоштовних проектів з відкритим вихідним кодом, так і платних індивідуальних з різними тарифними планами. Він є розроблений на [Ruby on Rails](http://uk.wikipedia.org/wiki/Ruby_on_Rails) і [Erlang](http://uk.wikipedia.org/wiki/Erlang) компанією GitHub, Inc, базується на системі керування версіями Git та був вперше випущеним у квітні 2008 року [4,16, 23].

Окрім розміщення кодів, сервіс надає також можливість розробникам спілкуватися між собою та коментувати зміни одне одного.

Кожному проекту надано персональну сторінку з можливістю зміни вмісту кожного файлу проекту та системою відстеження помилок. Також реалізовано підсвічування синтаксису для більшості мов програмування і надано можливість отримання коду не лише через Git, але й за допомогою завантаження архівного файлу проекту [23].

Перш ніж розпочати користування сервісом потрібно створити власний обліковий запис, ввівши ім’я користувача, електронну почту та пароль.

Для створення нового репозиторія потрібно вказати його назву, за бажанням опис та вибрати тип репозиторія: приватний чи публічний. Для створення нового проекту GitHub надає інструкцію щодо його створення. Проте якщо існує локальний репозиторій, щоб його помістити на GitHub достатньо додати url створеного пустого репозиторія на GitHub як віддалений для локального проетку і надіслати туди основну гілку проекту master. Після цього проект є розміщеним на GitHub. Також існує можливість імпортування в Git публічного Subvertion-проекту.

GitHub надає можливість іншим розробникам вносити власні зміни до проекту за допомогою їх долучення до учасників проекту, якщо вони мають облікові записи на цьому сервісі [4].

На сторінці проекту можна переглядати його історію, вносити до нього зміни та робити комміти цих змін, а також здійснювати створення нових гілок, їх перегляд, злиття та видалення. Внесення змін інших розробників до проекту здійснюють за допомогою створення pull request – спеціального запиту про долучення змін до гілки проекту. Pull request дозволяє порівняти зміни двох гілок та вирішити: варто додавати певні зміни до проекту чи ні. Також створення цього запиту є доречним для початку обговорення коммітів [4, 24].

У вкладці Network можна побачити список людей, що копіювали проект та додавали власні зміни до нього. У вкладці Wiki можна написати документацію чи певні нотатки до проекту, а у вкладці Issue можна залишати корисну інформацію, яка варта увагу, наприклад, повідомлення про наявність помилки, незавершеної роботи, або залишати питання. Вкладки Graphs та Puls надають статистику вкладу учасників у проект, а вкладка Code показує вміст кореневого каталогу проекту, автоматично виводить вміст файлу README, якщо він був створений, та інформацію про останній комміт [4, 24].

Для роботи з проектом, на внесення змін до якого користувач не має права, GitHub надає можливість його копіювання для цього користувача за допомогою натискання кнопки Fork, після здійснення чого користувач отримує увесь репозиторій проекту у власному обліковому записі. Проте на таких проектах вказано, що вони є відгалуженнями інших проектів [4].

Отже, Git забезпечує ефективний та швидкий обмін змінами при спільній роботі над проектом за допомогою можливості створення віддалених репозиторіїв на власному сервері чи Git хостинг-сайтах, одним з яких є GitHub, що надає можливість користувачам швидко створювати нові репозиторії та працювати з ними, використовуючи Git, як із віддаленими репозиторіями проектів. А також забезпечує легку та ефективну спільну роботу розробників, що можуть пропонувати власні зміни, обговорювати та об’єднувати їх із основною гілкою проекту.

Висновки

Отже, для забезпечення зручної та високопродуктивної роботи розробника чи групи розробників над проектом доцільним є використання саме розподіленої системи керування версіями, що забезпечує можливість проведення роботи над вирішенням декількох завдань одночасно, а також попереджує можливість втрати проекту при виході із ладу централізованого сервера.

Одним із представників цього типу систем керування версіями є Git. Завдяки збереженню усіх файлів проекту за хешами їхнього вмісту, що забезпечує їх цілісність, практично неможливої легкої втрати зафіксованих файлів та проекту в цілому, збереженню зліпків файлів у стиснутому вигляді, а відповідно компактності репозиторія, швидкого виконання практично всіх операцій без доступу до мережі, ця система керування версіями на даний момент є найбільш ефективною, надійною, швидкою та потужною.

Git має легку та швидку систему розгалуження, що забезпечує можливість зручної спільної роботи над проектом. Оскільки галуження не є ресурсномістким процесом, створення та злиття гілок стає невід’ємною складовою частиною роботи з Git, що значно полегшує спільну роботу над проектом, забезпечує можливість паралельного вирішення декількох завдань та попереджує порушення працездатності програми шляхом внесення помилкових змін до основної гілки.

Git, у порівнянні з деякими іншими системами керування версіями, надає можливість вибору файлів із списку змінених у процесі роботи над проектом, які ввійдуть до наступного комміта, що забезпечує об’єднання зв’язаних змін в один логічний блок, можливість внесення змін до раніше створеного комміта чи низки коммітів та здійснення інших маніпуляцій із змінами, що забезпечують зручну організацію перебігу робочого процесу, а також найбільш підхожого для користувача форматування виводу історії проекту.

Також створення віддалених репозиторіїв у файловій системі, до якої група розробників має спільний доступ, на власному сервері чи на Git хостинг-сайтах забезпечує легку та ефективну спільну роботу групи розробників, що можуть пропонувати власні зміни до проекту, обговорювати їх та додавати до основної гілки цього проекту.

Список використаної літератури

1. A Visual Guide to Version Control / Better Explained. –[Електронний ресурс]. – Режим доступу: <http://betterexplained.com/articles/a-visual-guide-to-version-control/>.

# [Auvray](http://www.infoq.com/author/Sebastien-Auvray) І. Distributed Version Control Systems: A Not-So-Quick Guide Through / [Sebastien Auvray](http://www.infoq.com/author/Sebastien-Auvray) // InfoQ. – [Електронний ресурс]. – Режим доступу: http://www.infoq.com/articles/dvcs-guide.

1. Sink E. Version Control by Example. – [Електронний ресурс]. – Режим доступу: <http://ericsink.com/vcbe/html/intro.html>.
2. Chacon S. Pro Git / Scott Chacon, Ben Straub. – Apress, 2009. – [Електронний ресурс]. – Режим доступу: <http://git-scm.com/book/en/v2>.
3. Обзор систем контроля версий // HIT - Информационный сайт о высоких технологиях. – [Електронний ресурс]. – Режим доступу: <http://all-ht.ru/inf/prog/p_0_1.html>.
4. Concurrent Versions System / Операційна система GNU. – [Електронний ресурс]. – Режим доступу: [www.nongnu.org/cvs](http://www.nongnu.org/cvs).
5. Apache Subversion. – [Електронний ресурс]. – Режим доступу: <https://subversion.apache.org/>.
6. Mercurial source control management / Mercurial. – [Електронний ресурс]. – Режим доступу: <http://mercurial.selenic.com/>.
7. Bazaar - The Adaptive Version Control System.– [Електронний ресурс]. – Режим доступу:<http://bazaar.canonical.com/en/>.
8. A Better Way to Build Complex Products / Рerforcе. – [Електронний ресурс]. – Режим доступу: <http://www.perforce.com/>.
9. Chacon S. Pro Git / Scott Chacon, Ben Straub. – Apress, 2009. – [Електронний ресурс]. – Режим доступу: <http://git-scm.com>.
10. Stansberry G. Essential Guide To Regular Expressions: Tools and Tutorials /Glen Stansberry. –[Електронний ресурс]. – Режим доступу: <http://www.smashingmagazine.com/2008/09/18/the-top-7-open-source-version-control-systems/>.
11. The New Breed of Version Control Systems / LAMP: THE OPEN SOURCE WEB PLATFORM. – [Електронний ресурс]. – Режим доступу: <http://www.onlamp.com/pub/a/onlamp/2004/01/29/scm_overview.html>.
12. Analysis of Git and Mercurial / User support for Google Project Hosting.– [Електронний ресурс]. – Режим доступу: <https://code.google.com/p/support/wiki/DVCSAnalysis>.
13. Почему Git лучше, чем X. – [Електронний ресурс]. – Режим доступу: <http://rezvanov.info/whygitisbetter/>.
14. [**Виспянський Ігор**](https://disqus.com/home/forums/vispyanskiy/). Трохи про Git для тих, хто використовує Subversion. – [Електронний ресурс]. – Режим доступу: <http://vispyanskiy.name/trohy-pro-git-dlya-tyh-hto-vykorystovuje-subversion>.

## Learn Version Control with Git / TOWER-- Version control with Git. –[Електронний ресурс]. – Режим доступу: http://www.git-tower.com/learn/git/ebook/command-line/appendix/why-git.

1. The Git Expert // GitGuys. –[Електронний ресурс]. – Режим доступу: <http://www.gitguys.com>.
2. Silverman Richard E. Git Pocket Guide // OSCON. – [Електронний ресурс]. – Режим доступу: <http://chimera.labs.oreilly.com/books/1230000000561/index.html>.
3. Wynn J. A Hacker’s Guide to Git / Joseph Wynn. – [Електронний ресурс]. – Режим доступу: <http://wildlyinaccurate.com/a-hackers-guide-to-git/>.
4. Wiegley J. Git from the bottom up / John Wiegley, 2009. – [Електронний ресурс]. – Режим доступу: <http://ftp.newartisans.com/pub/git.from.bottom.up.pdf>.
5. Chacon S. Git Community Book / Scott Chacon. – [Електронний ресурс]. – Режим доступу: <http://issues.liferay.com/secure/attachment/40633/git-book.pdf>.
6. The easiest way to use GitHub on Windows / Work in Visual Studio and deploy GitHub Enterprise on Azure. – [Електронний ресурс]. – Режим доступу: <https://github.com>.

# GitHubGuides / Hello World. – [Електронний ресурс]. – Режим доступу: https://guides.github.com/activities/hello-world/#commit.

Глосарій

ЛСКВ - локальна система керування версіями;

ЦСКВ - централізована система керування версіями;

РСКВ - розподілена система керування версіями;

RCS ( від англ. Revision Control System ) – система керування переглядами версій - одна з перших реалізацій [системи керування версіями](http://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%B5%D1%80%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D1%96%D1%8F%D0%BC%D0%B8);

CVS ( від англ. Concurrent Versions System ) - система одночасних версій – одна з централізованих систем керування версіями;

SHA-1 ( від англ. Secure Hash Algorithm 1 ) - алгоритм криптографічного [хешування](http://uk.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F);

URL ( від англ. Uniform Resource Locator ) - єдиний вказівник на ресурс - стандартизована адреса певного ресурсу;

 HTTP ( від англ. Hyper Text TransferProtocol ) - протокол передачі [гіпер-текстових документів](http://uk.wikipedia.org/wiki/%D0%93%D1%96%D0%BF%D0%B5%D1%80%D1%82%D0%B5%D0%BA%D1%81%D1%82);

HTTPS  (від англ. HyperText Transfer Protocol Secure) - розширення протоколу [HTTP](https://ru.wikipedia.org/wiki/HTTP), що підтримує шифрування даних.

SSH  (від англ. Secure Shell) - безпечна оболонка - один зі мережних протоколів, що надає можливість проведення віддаленого управління комп’ютером та тунелювання ТСР – з’єднань.