РОЗДІЛ 6. СИСТЕМИ КЕРУВАННЯ ВЕРСІЯМА GITHUB.

6.1. Загальні задачі систем керування версіями

Множина вимог до ІТ проєктів ознак якісного ПЗ: надійність, стійкість, модульність, безпечність, продуктивність, масштабованість, зручність застосування, тестування та супровід, і багато інших.

При цьому проєкт має включати: гарну документацію, включаючи журнали змін; деталізовані узгодження та стандарти; версіонування; автоматизовані тести та ін. Одним з зручних та ефективних засобів покращення робочих процесів стали системи верифікування версій, що на сьогодні надають гнучкі інструменти, які не диктують вам як потрібно реалізувати та документувати зміни, яку стратегію злиття використовувати, на яких етапах та стадіях фіксувати результати, які інструменти застосовувати, яким стандартам commit-ів (фіксування змін) слідувати та що потрібно рецензувати. Все це зорієнтовано вашими творчими підходами.

6.2. Що таке система контролю версій

Контроль версії - це практика відстеження змін програмного коду та управління ним. Системи контролю версії — це програмні інструменти, які допомагають командам розробників керувати змінами в вихідному коді з течією часу. В умовах праці вони допомагають команді розробників працювати швидше і ефективно. Системи контролю версії найбільш корисні для команд **DevOps**, оскільки допомагають скоротити час розробки та збільшити кількість успішних розгортань.

Програмне забезпечення контролю версій відстежує всі зміни, що вносяться в код, у спеціальній базі даних. При виявленні помилки розробники можуть повернутися назад і порівняти з попередніми версіями коду для виправлення помилок, зводячи до мінімуму проблеми для всіх учасників команди.

Система контролю версій (СКВ від англ. Version Control System, VCS) - це спеціалізоване місце зберігання коду, система, що записує зміни у файл, чи набір файлів протягом часу і дозволяє повернутися пізніше до певної версії [15].

Інше формулювання звучить так:

СКВ - це програмне забезпечення, яке допомагає розробникам програм співпрацювати та вести повну історію своєї роботи. Воно може зберігати зміни файлів та модифікації вихідного коду. Щоразу, коли користувач вносить зміни в проєкт, СКВ приймає стан проєкту та зберігає їх. Ці різні збережені стани проєкту відомі як версії.

Системи **VCS** інколи називають інструментом **SCM** (source code management - управління початковим кодом) або **RCS** (revision control system - система управління редакціями).

Усі системи контролю версій по суті вирішують 4 задачі.

1. **Доступ до коду.** Вихідники коду зберігаються у віддаленому репозиторії (сховищі даних), куди звертаються розробники, щоб забрати актуальну версію файлів або внести зміни. Так вибудовується командна технологія.
2. **log-ування змін у коді.** Відстеження **commit-**ів (внесення змін до коду) допомагає знайти хто, що і коли змінював, вирішити конфлікти при модифікуванні одних і тих же файлів, відкотитися на будь-який попередній стан.
3. **Розгалуження розробки.** Програмісти паралельно ведуть розробку нового функціоналу у окремих гілках, не торкаючись працездатності старого.
4. **Підтримка версії продуктів.** При випуску оновлень програмних продуктів ми позначаємо релізні версії, наприклад, за допомогою **tag-**ів, щоб зафіксувати їх у цьому стані, для дебагу або ретроспективи.

На сьогодні існує розподіл СКВ на три головні категорії:

- **Локальні;**
- **клієнт-серверні (централізовані);**
- **розподілені (децентралізовані).**

В свою чергу СКВ поділяють на **вільні (відкриті) та закриті**.

6.3. Локальні системи контролю версій

Кожен користувач як метод контролю версій застосовує копіювання файлів в окремі каталоги, але такий підхід є недосконалим, оскільки не дозволяє відслідковувати всі важливі зміни та відповідно уникнути різник помилок в роботі з файлами, що погіршує якість та продуктивність праці і особливо в складних проєктах.

Для того, щоб вирішити цю проблему, програмісти давно розробили локальні СКВ (рис. 16.1) з простою базою даних, яка зберігає записи про всі зміни в файлах, здійснюючи тим самим контроль ревізій.

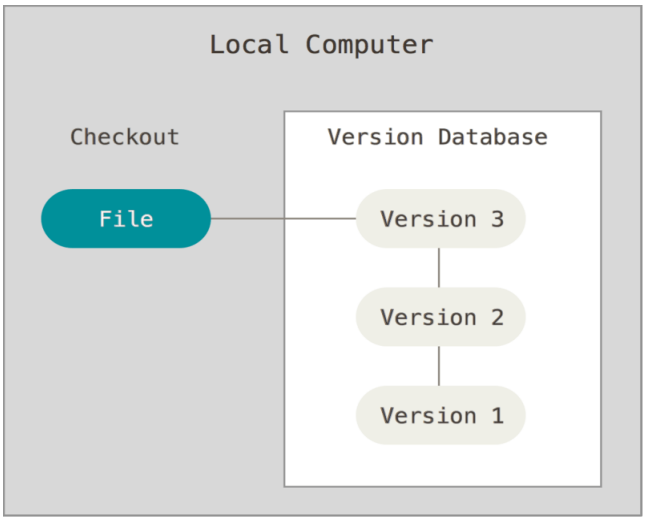

Рис. 16.1 Локальний контроль версій

6.4. Централізовані системи контролю версій

Для вирішення проблеми необхідності тісної взаємодії з іншими розробниками проєкту, були розроблені централізовані системи контролю версій (ЦСКВ - рис. 16.2). Такі системи, як **CVS, Subversion** і **Perforce**, використовують єдиний сервер, що містить всі версії файлів, і кілька клієнтів, які отримують файли з цього централізованого сховища. Застосування ЦСКВ було стандартом багато років.

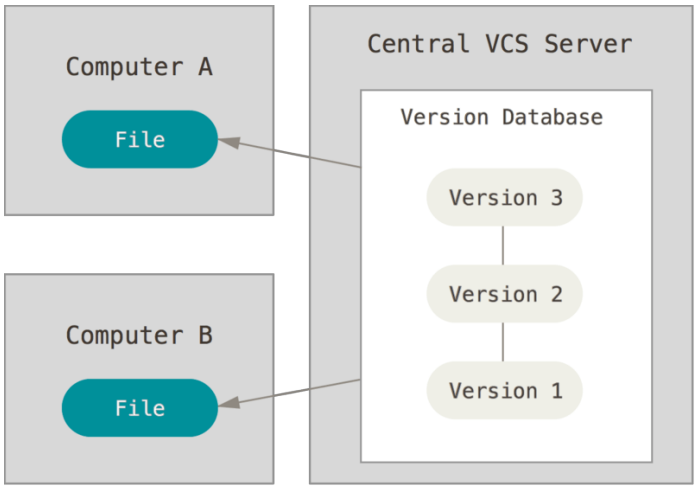

Рис. 16.2 Централізований контроль версій

Централізована СКВ має переваги над іншими СКВ, особливо над локальним. Наприклад, всі розробники проєкту певною мірою знають, чим займається кожен із них. Адміністратори мають повний контроль над тим, хто і що може робити і також

набагато простіше адмініструвати ЦСКВ, ніж оперувати локальними базами даних на кожному клієнті.

Мінуси – єдина точка відмови представлена централізованим сервером.

Якщо цей сервер вийде з ладу на годину, протягом цього часу ніхто не зможе використовувати контроль версій для збереження змін, над якими працює, а також ніхто не зможе обмінюватися цими змінами з іншими розробниками. Якщо жорсткий диск, на якому зберігається центральна БД, пошкоджений, а своєчасні бекапи відсутні, ви втратите весь проєкт разом з історією проєкту, не враховуючи одиничних знімків репозиторію, які збереглися на локальних машинах розробників.

Локальні СКВ страждають від тієї самої проблеми: коли вся історія проєкту зберігається в одному місці, ви ризикуєте втратити все.

6.5. Розподілені системи контролю версій

Розподілені СКВ (РСКВ - рис. 6.1) вже долають головні недоліки локальних та централізованих СКВ. У РСКВ (таких як **Git, Mercurial, Bazaar** або **Darcs**) клієнти не просто завантажують знімок всіх файлів (стан файлів на певний момент часу) - вони повністю копіюють репозиторій. У цьому випадку, якщо один із серверів, через який розробники обмінювалися даними, помре, будь-який репозиторій клієнта може бути скопійований на інший сервер для продовження роботи. Кожна копія репозиторію є повним бекапом всіх даних.

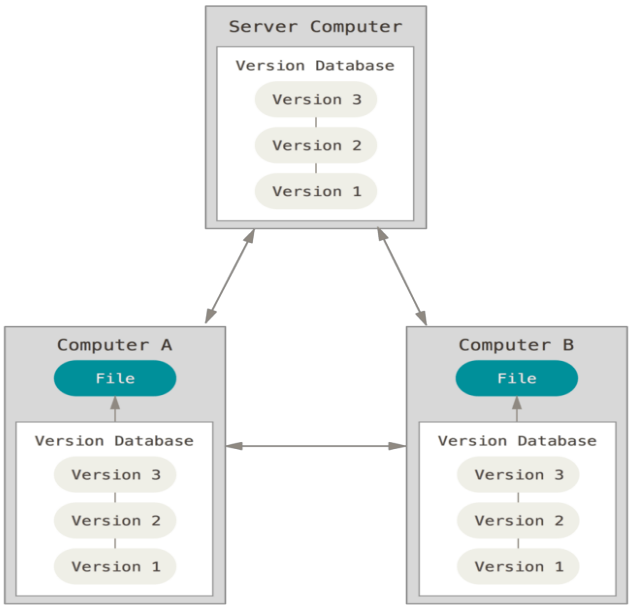

Рис. 16.3 Розподілений контроль версій

Ще однією з переваг РСКВ є те, що більшість з таких систем можуть одночасно взаємодіяти з декількома віддаленими репозиторіями. Завдяки цьому ви можете працювати з різними групами людей, застосовуючи різні підходи одночасно в рамках одного проєкту. Це дозволяє застосовувати відразу кілька підходів у розробці, наприклад, ієрархічні моделі, що не можливо в централізованих системах.

6.6. Архітектура Git

Git є розподіленою системою. Центрального репозиторію не існує: всі копії створюються рівними, що різко відрізняється від **VCS** другого покоління, де робота заснована на додаванні та вилученні файлів із центрального репозиторію. Це означає, що розробники можуть обмінюватись змінами один з одним безпосередньо перед об'єднанням своїх змін в офіційну гілку.

Крім того, розробники можуть вносити свої зміни до локальної копії репозиторію без відома інших репозиторіїв. Це дозволяє **commit**-и без підключення до мережі або інтернету. Розробники можуть працювати локально в автономному режимі, доки не будуть готові поділитися своєю роботою з іншими. У цей момент зміни відправляються до інших репозиторіїв для перевірки, тестування або розгортання.

Коли файл додається для відстеження **Git**, він стискається за допомогою алгоритму стиснення ***zlib***. Результат хешується за допомогою хеш-функції **SHA-1**. Це дає унікальний хеш, який відповідає конкретному вмісту в цьому файлі. **Git** зберігає його в базі об'єктів, що знаходиться в прихованій папці ***.git/objects***. Ім'я файлу – це згенерований хеш, а файл містить стислий контент. Дані файли називаються **blob-ами** і створюються щоразу при додаванні до репозиторію нового файлу (або зміненої версії існуючого файлу).

Git реалізує проміжний індекс (***staging index***), який виступає як проміжна область для змін, які готуються до **commit**. У міру підготовки нових змін на їхній стислий вміст посилаються в спеціальному індексному файлі, який набуває форми об'єкта дерево. Дерево – це об'єкт **Git**, який пов'язує **blob-и** з їхніми реальними іменами файлів, дозволами на доступ до файлів та посиланнями на інші дерева і таким чином представляє стан певного набору файлів та каталогів. Коли всі відповідні зміни підготовлені для **commit**, індексне дерево можна зафіксувати у репозиторії, який створює об'єкт **commit** у базі даних об'єктів **Git. Commit** посилається на дерево заголовків для конкретної версії, а також на автора **commit**, адресу електронної пошти,

дату та повідомлення **commit**. Кожен **commit** також зберігає посилання на свій батьківський **commit**(-и), і так згодом створюється історія розвитку проєкту.

Як уже згадувалося, всі об'єкти **Git** – **blob-и, tree (дерева) та commit**-и – стискуються, хешуються та зберігаються в базі даних об'єктів на основі їх хешів. Вони називаються вільними об'єктами (**Loose Objects**). Тут не використовуються ніякі **diff**-и для економії місця, що робить **Git** дуже швидким, оскільки повний вміст кожної версії файлу є вільним об'єктом. Однак деякі операції, такі як передача **commit**-ів у віддалений репозиторій, зберігання дуже великої кількості об'єктів або ручний запуск команди складання сміття **Git** викликають переупаковку об'єктів у пакетні файли. У процесі упаковки обчислюються зворотні **diff**-и, які стискаються для виключення надмірності та зменшення розміру. В результаті створюються файли ***.pack*** з вмістом об'єкта, а для кожного з них створюється файл ***.idx*** (або індекс) з посиланням на упаковані об'єкти та їхнє розташування в пакетному файлі.

Коли гілки переміщуються у віддалені сховища або витягуються з них, мережі передаються ці пакетні файли. Під час витягування або вилучення гілок файли пакета розпаковуються для створення вільних об'єктів у репозиторії об'єктів.

Підходи в роботі систем контролю версій

Підхід **Git** до зберігання даних більше схожий на набір знімків мініатюрної файлової системи [15]. Щоразу, коли ви виконуєте параметр «**commit**», тобто зберігаєте стан свого проєкту в Git, система запам'ятовує, як виглядає кожен файл у цей момент, і зберігає посилання на цей знімок (рис. 16.4). Для збільшення ефективності, якщо файли не були змінені, **Git** не запам'ятовує ці файли знову, а лише створює посилання на попередню версію файлу, який вже збережений. Git представляє свої дані, як, скажімо, потік знімків.

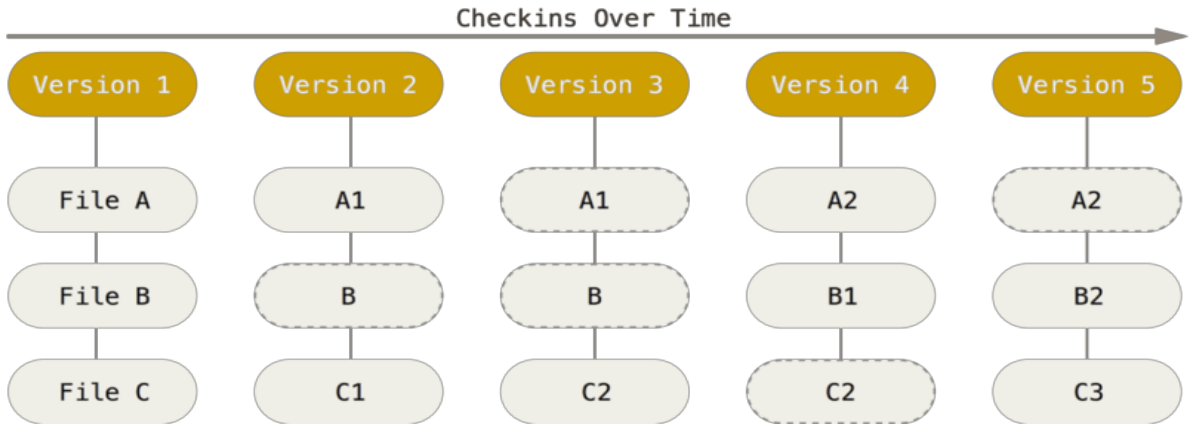

Рис. 16.4 Зберігання даних як знімків проєкту у часі

Це дуже важлива відмінність між **Git** і від будь-якої іншої СКВ. **Git** переосмислює практично всі аспекти контролю версій, скопійованих з попереднього покоління більшістю інших систем. Це робить **Git** більш схожим на мініатюрну файлову систему з напрочуд потужними утилітами, надбудованими над нею, ніж просто на СКВ. Такий підхід надає суттєвих переваг підходам використаним в **Git**.

Виконання операцій локально.

Загальна проблема у роботі більшості СКВ - необхідність постійної підтримки мережевого зв'язку, що призводить до постійних затримок. З метою подолання цієї проблеми більшість операцій **Git** виконує з локальним розташуванням файлів і ресурсів – системі не потрібна жодна інформація з інших комп'ютерів у вашій мережі. Оскільки вся історія проєкту зберігається прямо на вашому локальному диску, більшість операцій здаються мало не миттєвими.

Для прикладу, щоб переглянути історію проєкту, **Git** не потрібно з'єднуватися з сервером для її отримання та відображення — система просто зчитує дані безпосередньо з локальної бази даних. Це означає, що ви побачите історію проєкту практично миттєво. Якщо вам необхідно переглянути зміни, зроблені між поточною версією файлу та версією, створеною місяць тому, **Git** може знайти файл місячної давності та локально обчислити зміни, замість того, щоб вимагати віддалений сервер виконати цю операцію, або замість отримання старої версії файлу з сервера та виконання операції локально.

Це також означає, що є лише невелика кількість дій, які ви не зможете виконати, якщо ви знаходитесь офф-лайн або не маєте доступу до VPN на даний момент. Ви можете без будь-яких проблем працювати з вашою локальною копією: коли буде можливість підключитися до мережі, всі зміни можна буде синхронізувати.

Цілісність у **Git** забезпечується обчисленням хеш-суми, і потім відбувається збереження [15]. Всі подальші звернення до збережених об'єктів відбувається за цією хеш-сумою, що виключає будь-які випадкові зміни вмісту файлів чи каталогів. Ця функціональність вбудована в **Git** на низькому рівні і є невід'ємною частиною його філософії. Ви не втратите інформацію під час її передачі та не отримаєте пошкоджений файл без відома **Git**.

SHA-1 хеш - механізм, яким користується **Git** для обчислення хеш-сум, становить рядок довжиною 40 шістнадцяткових символів (0-9 і a-f). Він обчислюється на основі вмісту файлу або структури каталогу.

Зразок **SHA-1** хеш:

24b9da6552252987aa493b52f8696cd6d3b00373

Git зберігає всі об'єкти в свою базу даних не за ім’ям, а за хеш-сумою вмісту об'єкта.

Робота **Git** практично заснована на збереженні будь-яких дій додаванням нових даних в базу **Git**. Дуже складно змусити систему видалити дані або зробити щось, що не можна згодом скасувати. Ви можете втратити або зіпсувати свої зміни, поки вони не зафіксовані, але після того, як ви зафіксуєте знімок у **Git**, буде дуже складно щось втратити, особливо, якщо ви регулярно синхронізуєте свою базу з іншим репозиторієм.

Вищевказані особливості **Git** розкривають множину переваг, суттєво полегшуючи роботу з проєктами, дозволяючи серйозно експериментувати не боячись серйозних проблем.

Три стани Git

У **Git –** файли можуть бути в трьох основних станах – рис. 16.5 [15]:

- **змінений (modified),**
- **індексований (staged),**
- **зафіксований (committed).**

До **змінених** відносяться файли, які змінилися, але ще не були зафіксовані.

Індексований - це змінений файл у його поточній версії, позначений для включення в наступний «**commit**».

Зафіксований означає, що файл збережений у вашій локальній базі.

Три робочі секції проєкту Git

Проєкт **Git** поділяється на три робочі секції [15]**:**

- **робоча копія (*working tree*),**
- **область індексування (*staging area*),**
- **каталог Git (*Git directory*).**

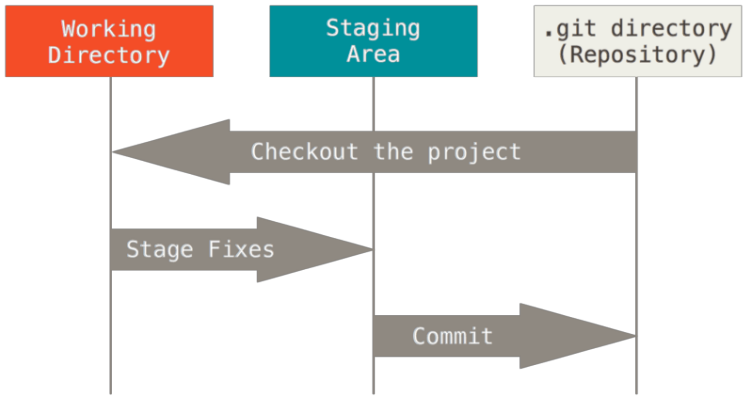

Рис. 16.5 Робоча копія, область індексування та каталог **Git**

Робоча копія є знімком однієї версії проєкту. Ці файли виймаються зі стиснутої бази даних у каталозі **Git** і поміщаються на диск, щоб їх можна було використовувати чи редагувати.

Область індексування - це файл(-и), що зазвичай знаходиться в каталозі **Git**, в ньому міститься інформація про те, що потрапить в наступний «**commit**». Її технічна назва мовою **Git — індекс**, але фраза **«область індексування»** також працює.

Каталог Git - це місце, де **Git** зберігає метадані і базу об'єктів вашого проєкту. Це найважливіша частина **Git** і це частина, яка копіюється при клонуванні репозиторію з іншого комп'ютера.

Базовий підхід у роботі з Git:

1. Ви змінюєте файли вашої робочої копії.
2. Вибірково додаєте до індексу лише ті зміни, які мають потрапити до наступного «**commit**», додаючи тим самим знімки лише цих змін до індексу.
3. Коли ви робите **«commit»**, використовуються файли з індексу як є, і цей знімок зберігається у вашому каталозі **Git**.

Якщо певна версія файлу є у каталозі **Git**, то ця версія вважається **зафіксованою (committed).** Якщо файл був змінений і доданий до індексу, значить, він **індексований (staged).** І якщо файл був змінений з моменту останнього розпаковування з репозиторію, але не був доданий до індексу, він вважається **зміненим** (**modified**).

6.7. Директорія Git та робоча директорія

В **“директорії git”** зберігається вся історія **Git** та мета-інформація вашого проєкту - включаючи всі об'єкти (**commit**, дерева, **blob**-и, **tag-**и), всі покажчики на різні гілки та багато іншого [16].

На кожен проєкт є тільки одна директорія **Git**, і це директорія (за замовчуванням, але не обов'язково) **“.git”** в корені вашого проєкту. Якщо ви подивитеся на вміст цієї директорії, то побачите всі ваші важливі файли (можуть бути і інші файли/директорії):

```
$>tree -L 1
.
|-- HEAD         # вказівник на вашу активну гілку
|-- config       # ваші персональні налаштування
|-- description  # опис проєкту
```

|-- hooks/ # pre/post action hooks (скрипти (надалі хуки) які можуть викликатися git командами)

|-- index # індексний файл

|-- logs/ # історія гілок проєкту (де вони розташовані)

*|-- objects/ # ваші об'єкти (**commit**, дерева, **blob**-и, **tag-**и)*

`-- refs/ # вказівники на ваші гілки розробки

Робоча директорія це просто тимчасове місце, де ви можете модифікувати файли, а потім виконати **commit**.

6.8. Об'єктна модель Git

Історія проєкту зберігається у особливо організованих файлах, що посилаються один на одного за допомогою 40-значного "імені об'єкта" **SHA1** – криптографічної хеш-функції [17].

Це дозволяє:

- **Git** може швидко визначити чи ідентичні два об'єкти чи ні, просто порівнюючи їх імена.
- Так як імена об'єктів обчислюються однаково у всіх репозиторіях, то об'єкти з однаковим вмістом у двох репозиторіях завжди зберігаються під однаковими іменами.
- **Git** може знаходити помилки, коли читає об'єкт, для цього потрібно просто порівняти хеш значення вмісту об'єкта з його ім'ям.

Об'єкти

Основою побудови **Git** є чотири типи об'єктів - **"блоб" (blob)**, **"дерево" (tree)**, "**commit**", та **"таг" (tag)**. Кожен об'єкт має три частини – **тип, розмір та зміст**. Таку структуру порівнюють з надбудовою над традиційною файловою системою.

"blob" використовується щоб зберігати вміст файлу - зазвичай це просто файл.

"tree" це щось на зразок директорії - воно посилається на групу інших дерев та/або **blob-**ів (тобто файлів та директорій).

"Commit" вказує на окреме дерево, за суттю відзначає дерево фіксуючи в історії яким чином воно виглядає в момент виконання **commit**. Він містить мета-інформацію фіксуючи момент часу та автора змін, внесених з останнього **commit**, покажчик на попередній **commit** тощо.

"tag" це спосіб маркувати певним чином певний **commit**. Зазвичай це використовується щоб маркувати (по суті дати якесь ім'я, що легко запам'ятовується) певні **commit** є специфічними, щоб згодом було легше їх знайти.

Об’єкт типу «Блоб» (blob).

Blob зберігає зміст файлу (рис. 16.6).

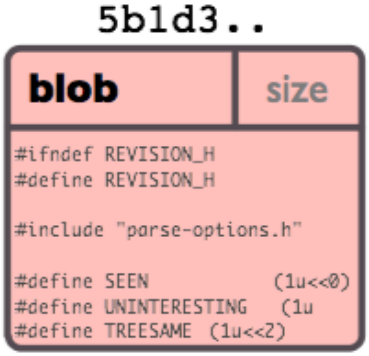

Рис. 16.6 Представлення об’єкту типу Блоб

Об'єкт **"blob"** це порція бінарних даних. **Blob** ні на що не посилається у нього немає жодних атрибутів, не має навіть імені файлу.

Оскільки **blob** повністю визначається його власним вмістом, то якщо два файли в директорії або навіть у різних версіях репозиторію мають однаковий вміст, вони будуть розділяти той самий **blob** об'єкт. Об'єкт повністю залежить від розташування в дереві каталогів, і перейменування файлу не змінить об'єкт із яким файл пов'язаний.

Зміст **blob** можливо переглянути командою **git show**. Наприклад:

$ git show 6ff87c4664

Note that the only valid version of the GPL as far as this project
is concerned is _this_ particular version of the license (ie v2, not
v2.2 or v3.x or whatever), unless explicitly otherwise stated.

...

Об’єкт «Дерево» (Tree)

Tree це простий об'єкт який містить у собі групу покажчиків на **blob**-и та інші дерева (рис. 16.7) - представляє вміст директорій чи піддиректорій.

Рис. 16.7 Представлення об’єкту дерево

Щоб оглянути зміст дерева необхідно використати команду **git ls-tree (**можливо використовувати **git show**, але вона надає менший обсяг інформації про зміст дерева). Наприклад за відомим значенням **SHA** дерева:

$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3 COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745 Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200 GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7dcd470b INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1 Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52 README
...

Об'єкт **tree** містить список записів, що складаються з виду, типу об'єкту, значення **SHA1**, і імені відповідно. Записи відсортовані на ім'я. Так виглядає вміст однієї директорії **tree**.

Посилання на об'єкт у дереві може бути як **blob** (файлом по суті), так і **tree** (піддиректорією). Оскільки імена всіх об'єктів, **tree** та **blob**, збігаються з **SHA** хеш-значенням їхнього вмісту, то **SHA** значення двох **tree** будуть ідентичні тільки якщо їх вміст (включаючи, рекурсивно, вміст усіх піддиректорій) ідентичний.

Об'єкт commit

Об'єкт "**commit**" пов'язує фізичний стан **tree** з описом того, яким чином ми прийшли до нього і чому (рис. 16.8).

Рис. 16.8 Представлення об'єкту **commit**

Щоб дослідити **commit** можна застосувати параметр **--pretty=raw** з **git show** або **git log**. Наприклад:

$ git show -s --pretty=raw 2be7fcb476

commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700
Fix misspelling of 'suppress' in docs
Signed-off-by: Junio C Hamano <gitster@pobox.com>

Commit визначається наступними головними значеннями:

- **Дерево (tree)**: ім'я **SHA1** об'єкта **tree** (як визначено нижче), що представляє вміст директорії в певний момент часу.
- **Батько(и)**: **SHA1** ім'я деякого числа **commit**, які являють собою попередній крок(и) в історії проєкту. Приклад вище має одного з батьків; хоча **commit**-и злиття можуть мати більше одного батька. **Commit** без батьків називається **"root (кореневий)" commit**, і є початковим станом проєкту. Кожен проєкт повинен мати принаймні один кореневий **commit**. Проєкт може також мати багато коренів, проте це не загальний випадок (і не обов'язково хороша ідея).
- **Автор:** Ім'я розробника відповідального за ці зміни, разом із датою.
- **Commit**-ер: ім'я розробника, який створив цей **commit**, разом з датою цієї події. Воно (ім'я) може відрізнятись від імені автора; наприклад у випадку, якщо автор написав патч і відправив його електронною поштою іншому розробнику який наклав патч і виконав **commit**.
- **Коментар:** описує цей **commit**.

Об'єктна модель

Розглянемо приклад поєднання 3х вище розглянутих головних об'єкти (**blob**, **tree** та **commit**).

Для проєкту директорії зразкової структури (рис. 16.9):

```
$>tree
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb

2 directories, 3 files
```

Рис. 16.9 Директорія зразкової структури

За умови, якщо виконано **commit** в репозиторій **Git** отримаєм наступну відображення (рис. 16.10).

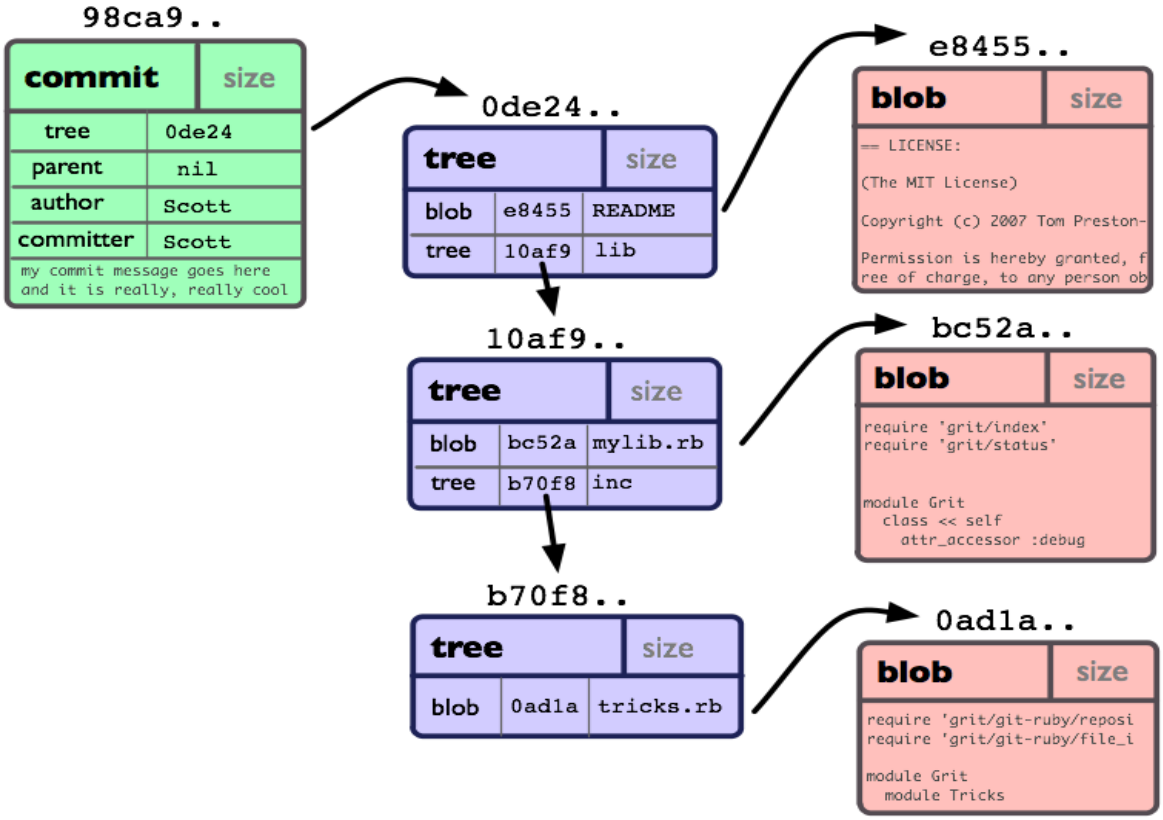

Рис. 16.10 Представлення зразка директорії після **commit**

Рис. 16.11 Представлення об’єкту таг

Об’єкт «маг» (tag)

Об'єкт **tag** містить **ім'я об'єкту** (називається просто - “об'єкт”), **тип об'єкту**, **ім'я tag**, або розробника ("таггер") який створив **tag**, і **повідомлення**, яке може

містити підпис (рис. 16.11). Оглянути об'єкт **tag** можна командою **git cat-file**. Наприклад:

$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Haemano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----

Питання до розділу 6

1. Що таке системи контролю версій?
2. Які задачі виконують системи контролю версій?
3. Які категорії систем контролю версій ви знаєте? Опишіть їх.
4. Які покоління систем контролю версій ви знаєте? Опишіть їх.
5. Що ви знаєте про третє покоління систем контролю версій?
6. Як побудована Git?
7. Які підходи в побудові роботи СКВ ви знаєте? Яка відмінність Git від інших СКВ?
8. Як забезпечується цілісність Git?
9. Як організована директорія Git?
10. Об'єктна модель Git? Опишіть її.
11. Опишіть об'єкт blob.
12. Опишіть об'єкт tree.
13. Опишіть об'єкт commit.
14. Опишіть об'єкт tag.