РОЗДІЛ 8. КРОКИ РЕАЛІЗАЦІЇ ІТ-ПРОЄКТІВ В GITHUB

8.1. Фіксація в Git

Можливості проєктів розширюють розгалуження, що підтримують більшість СКВ. **Git** заохочує процес роботи, при якому розгалуження та злиття виконується часто: операція створення гілки виконується майже миттєво, перемикання між гілками, як правило, також швидко. Розгалуження, як функціональність надає унікальний та потужний інструмент, який може суттєво підвищити ефективність змінити звичний процес розробки [15].

Раніш було розглянуто, що **Git** зберігає дані як послідовність знімків.

Об'єкт фіксації **Git** зберігає **вказівник на знімок змісту**, який ви додали. Цей об'єкт також містить:

- ім'я,
- поштову адресу автора,
- набране вами повідомлення,
- вказівники на фіксацію або фіксації, що були прямо до цієї фіксації (батько чи батьки): нуль для першої фіксації, одна фіксація для нормальної фіксації, та декілька фіксацій для фіксацій, що вони є результатом злиття двох чи більше гілок.

Приклад: припустимо, що у вас є тека з трьома файлами, які ви додали та зафіксували. Додання файлів обчислює контрольну суму для кожного (**SHA-1** хеш про котрий ми згадували раніш), зберігає версію файлу в сховищі **Git** (**Git** називає їх **blob ами**), та додає їх контрольні суми до області додавання:

$ git add README test.rb LICENSE

$ git commit -m 'The initial commit of my project'

Коли ви створили фіксацію за допомогою **git commit** (рис. 18.1), **Git** обчислив контрольну суму кожної теки (у цьому випадку, тільки кореневої теки) та зберігає ці об'єкти дерева в сховищі **Git.** Потім **Git** створює об'єкт фіксації, що зберігає метадані та вказівник на корінь дерева проєкту, щоб він міг відтворити цей знімок, коли потрібно.

Ваше **Git** сховище тепер зберігає п'ять об'єктів: по одному **blob** зі змістом на кожен з трьох файлів, одне **tree**, що перелічує зміст теки та вказує, які файли зберігаються у яких **blob**, та одну фіксацію, що вказує на корінь дерева, та зберігає метадані фіксації.

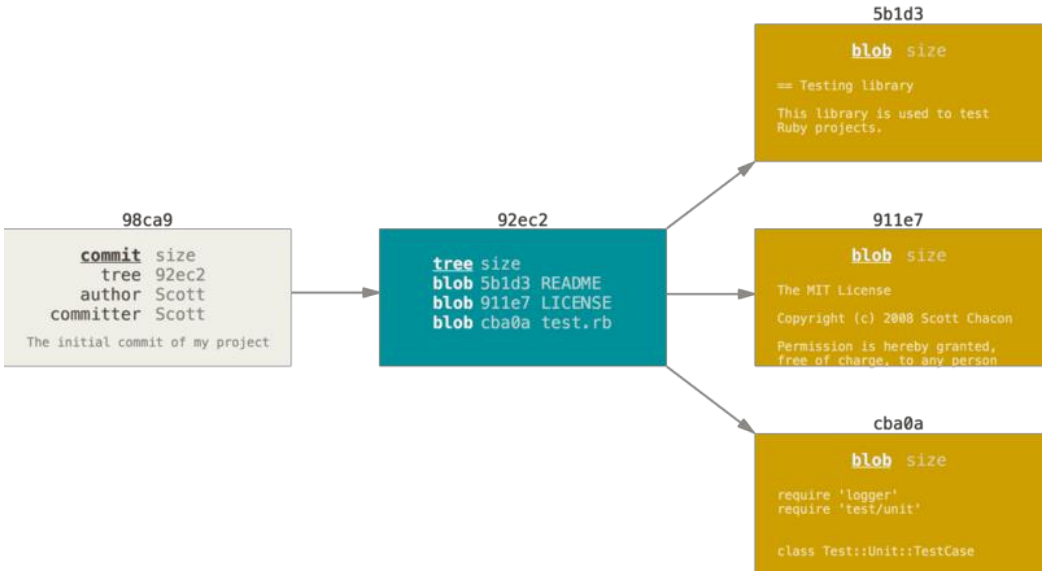

Рис. 18.1 Фіксація як дерево

Ваше **Git** сховище тепер зберігає п’ять об’єктів: по одному **blob** зі змістом на кожен з трьох файлів, одне **tree**, що перелічує зміст теки та вказує, які файли зберігаються у яких **blob**, та одну фіксацію, що вказує на корінь дерева, та зберігає метадані фіксації.

Якщо ви зробите якісь зміни та зафіксуєте знову, наступна фіксація буде зберігати вказівник на попередню.

Гілка в **Git** це просто легкий вказівник, що може пересуватись, на одну з цих фіксацій. Загальноприйнятим ім’ям першої гілки в **Git** є **master**. Коли ви почнете робити фіксації, вам надається гілка **master**, що вказує на останню зроблену фіксацію. Щоразу, коли ви фіксуєте, вона переміщується вперед автоматично (рис. 18.2).

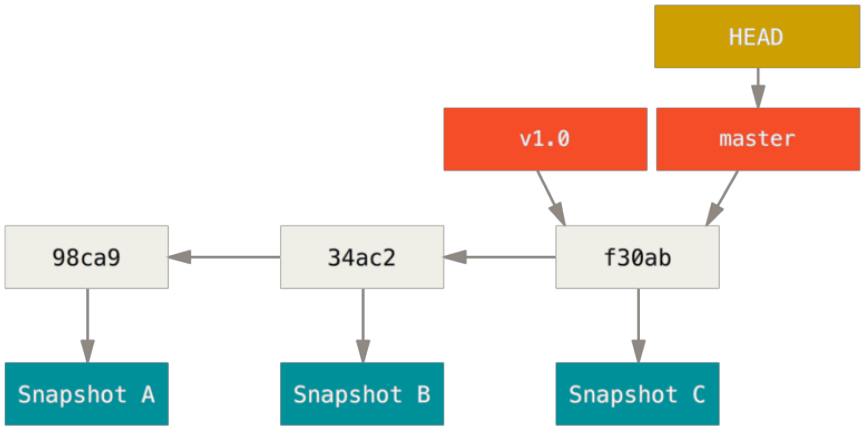

Рис. 18.2 Гілка та її історія фіксацій

8.2. Робота з гілками

Створення нової гілки виконується командою:
$ git branch testing
Наприклад нова гілка під назвою **testing** (рис. 18.3)

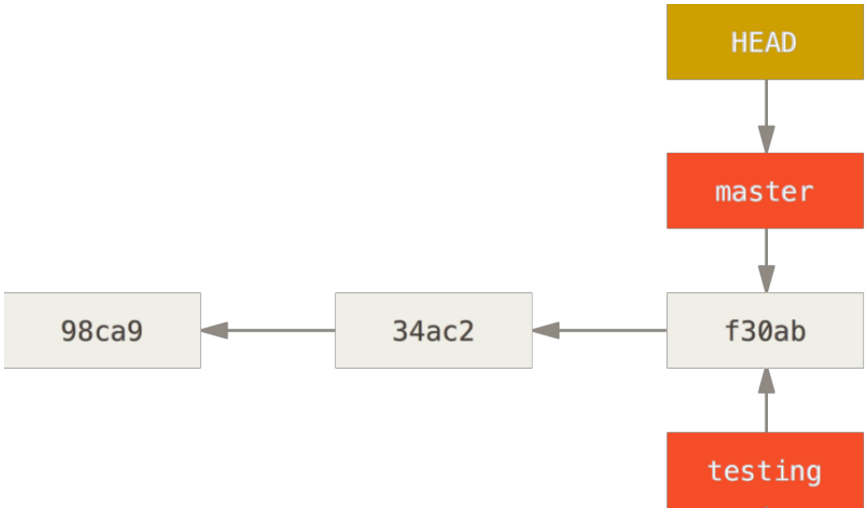

Рис. 18.3 Дві гілки вказують на одну послідовність фіксацій. HEAD вказує на поточну гілку

Перехід на існуючу гілку, виконують командою **git checkout** (рис. 18.4).
$ git checkout testing

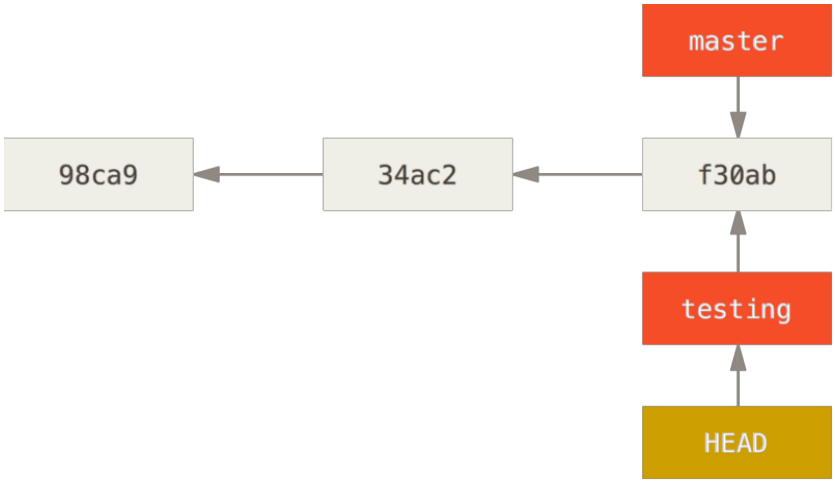

Рис. 18.4 HEAD вказує на поточну гілку.

При фіксації поточна гілка пересувається вперед (**testing**) (рис. 18.5).

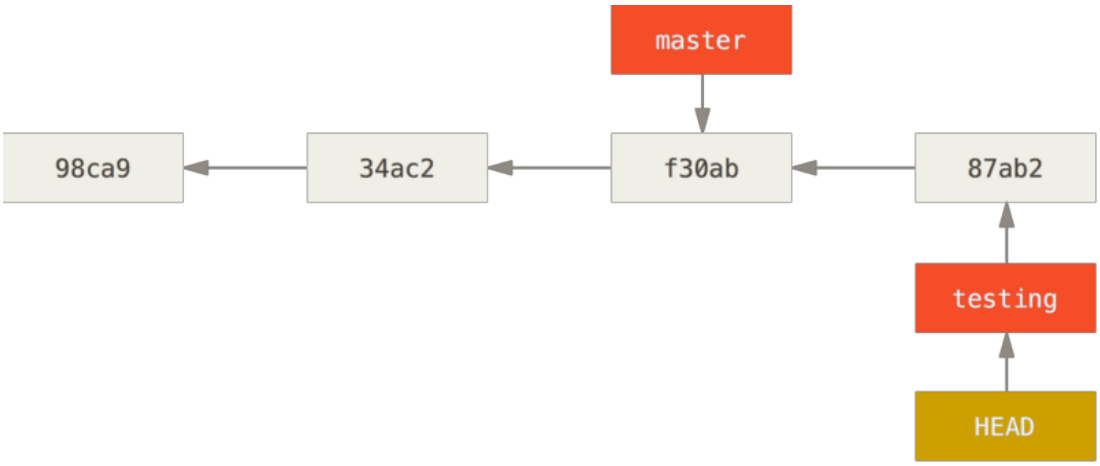

Рис. 18.5 Гілка HEAD пересувається уперед при фіксації

Переключення до гілки **master** виконують командою **git checkout** (рис. 18.6)**:**
$ git checkout master

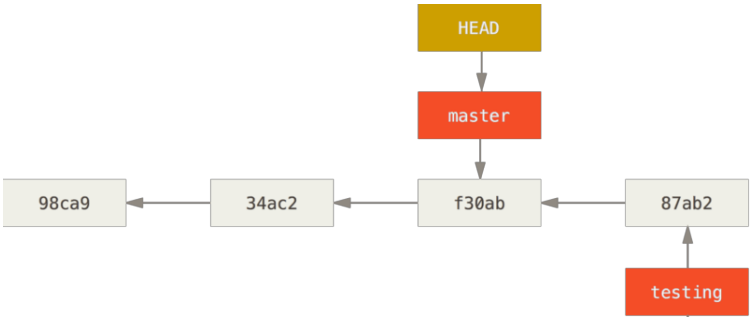

Рис. 18.6 HEAD пересувається, коли ви отримуєте (checkout)

Якщо знову зафіксувати то отримуємо розбіг (**diverged**) (рис 18.7) історії проєкту.

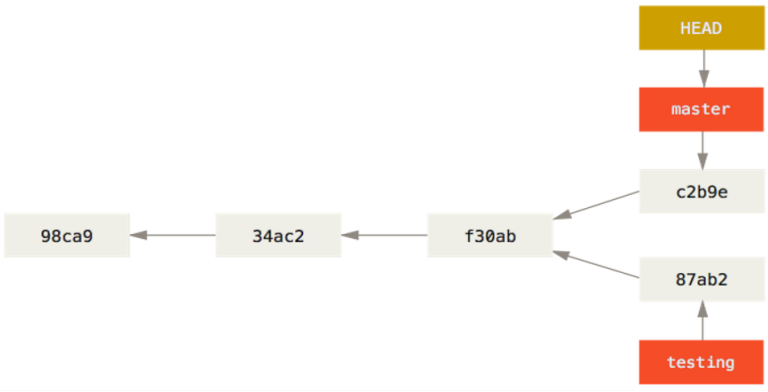

Рис. 18.7 Розбіг історії проєкту

Перевірити історію ваших **commit** ви, можете командою **git log**.

Перевага **Git** над іншими СКВ у простій та зручній роботі з гілками. Гілка в **Git** — це насправді простий файл, що містить 50 символів контрольної суми **SHA-1 commit**, на який вказує. Гілки легко створювати та знищувати. Створити гілку так же швидко, як записати 41 байт до файлу (40 символів та символ нового рядка).

Ми маємо певну розгалужену структуру проєкту, наприклад зображену на рис. 18.8.

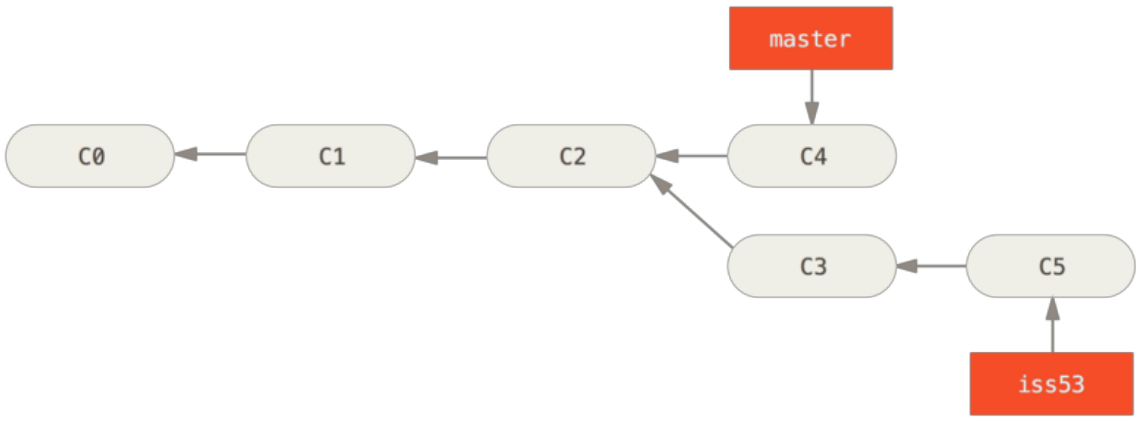

Рис. 18.8 Структура проєкту з розгалуженням на гілку iss53

Якщо в процесі роботи над проєктом – над гілкою **iss53**, ми вирішили що роботу завершено і можна злити з гілкою **master**. Для цього потрібно перейти на робочу гілку та виконати команду **git merge:**

$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)

В цьому випадку **Git** робить просте три-точкове злиття (рис. 18.9), користуючись двома знімками, що вказують на гілки та третім знімком - їх спільним предком.

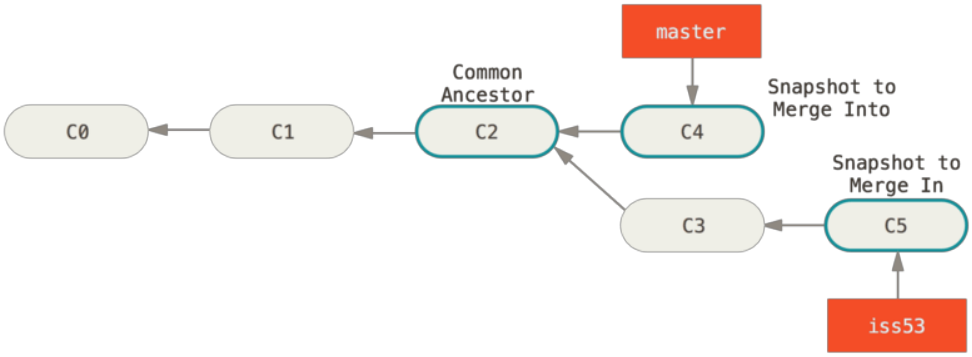

Рис. 18.9 Три відбитки типового злиття

Історія змін двох гілок почала відрізнятися в якийсь момент. Так як **commit** поточної гілки не є прямим нащадком гілки, в яку ви зливаєте зміни, **Git** мусить трохи попрацювати. В цьому випадку **Git** робить просте три-точкове злиття, користуючись двома знімками, що вказують на гілки та третім знімком - їх спільним предком. Такий **commit** називають **commit** злиття (**merge commit**) (рис. 18.9). Його особливістю є те, що він має більше одного батьківського **commit**.

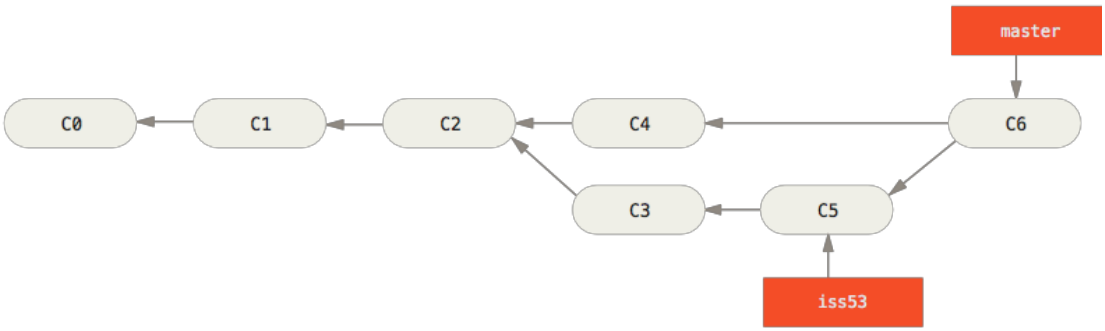

Рис. 18.9 **Commit** злиття

Тепер, коли ваші зміни зафіксовані, гілка **iss53** вам більше не потрібна. Її можна видалити:

$ git branch -d iss53

Git сам визначає найбільш підходящого спільного спадкоємця, якого брати за основу зливання. Це відрізняє **Git** від старіших систем таких як **CVS** чи **Subversion** (до версії 1.5), де розробник, що виконує зливання, сам повинен вказувати що брати за основу зливання.

8.3. Кроки реалізації іт-проєктів в GitHub

З розвитком систем контролю версій з'явилися різні стилі розробки, що дозволяють програмістам простіше знаходити баги, писати код паралельно з колегами і підвищувати частоту релізів. Сьогодні більшість програмістів використовують для створення якісного програмного забезпечення одну із двох моделей розробки**: Git flow** або **магістральну розробку**.

Робочий процес Git-flow, який був популяризований першим, є суворішою моделлю розробки, в якій схвалювати зміни основного коду можуть лише певні люди. Це дозволяє забезпечити якість коду та звести кількість багів до мінімуму.

Магістральна розробка – більш відкрита модель, оскільки доступ до основного коду мають усі розробники. Це дозволяє командам швидко виконувати ітерації та впроваджувати CI/CD.

*У розробці ПЗ, **CI/CD** або **CICD** — це комбінація безперервної інтеграції (**continuous integration**) та безперервного розгортання (**continuous delivery** або **continuous deployment**) програмного забезпечення у процесі розробки.*

CI/CD поєднує розробку, тестування та розгортання додатків.

*На даний момент **DevOps-**програмісти прагнуть застосовувати CI/CD практично для всіх завдань*

Магістральна розробка - це метод контролю версій, при якому розробники поєднують невеликі часті оновлення з центральною магістраллю або основною гілкою. Це звичайна практика серед команд DevOps та один із етапів життєвого циклу DevOps, оскільки цей метод дозволяє оптимізувати етапи злиття та інтеграції. Магістральна розробка є обов'язковою практикою CI/CD. В даному випадку розробники можуть створювати короткострокові гілки з декількома невеликими комітами, на відміну від інших стратегій із функціональними гілками, які надовго залишаються у роботі. У міру зростання складності бази коду та розміру команди модель магістральної розробки допомагає підтримувати надходження релізів до робочого середовища.

Git-flow – альтернативна модель розгалуження Git, в якій використовуються довгострокові функціональні гілки та декілька основних гілок. У Git-flow використовується більше гілок, їх термін життя довше, а комміти в них більші, ніж у моделі магістральної розробки. Відповідно до цієї моделі розробники створюють функціональну гілку і відкладають її злиття з головною магістральною гілкою до завершення роботи над функцією. Такі довгострокові функціональні гілки вимагають

тіснішої взаємодії розробників при злитті, оскільки створюють підвищений ризик відхилення від магістральної гілки та виконання конфліктуючих оновлень.

Модель Git-flow також має на увазі окремі напрямки основних гілок для розробки, термінових виправлень, функцій та релізів. Для злиття коммітів між цими гілками застосовуються різні стратегії. Оскільки більша кількість гілок створює складності при взаємодії та управлінні, у таких системах часто виникає потреба у додаткових нарадах щодо планування та перевірок з боку команди.

Вести магістральну розробку набагато простіше, оскільки як джерело виправлень і релізів виступає основна гілка. Дана модель розробки передбачає, що основна гілка завжди стабільна, не має помилок і готова до розгортання.

Магістральна технологія потрібна для безперервної інтеграції. Якщо процеси складання та тестування автоматизовані, але розробники довго працюють із ізольованими функціональними гілками, які рідко інтегруються у загальну гілку, потенціал безперервної інтеграції залишається нереалізованим.

Магістральна технологія полегшує інтеграцію коду. Коли розробники завершують новий етап роботи, вони повинні виконати злиття нового коду з головною гілкою. Однак злиття змін з магістральною гілкою не виконується доти, доки не буде підтверджено їх успішне складання. На цьому етапі можуть виникати конфлікти, якщо з початку нової роботи в магістральну гілку були внесені зміни. У міру розширення команд розробників та масштабування бази коду ці конфлікти стають дедалі складнішими. Таке трапляється, коли розробники створюють окремі гілки з відхиленнями від вихідної гілки, а інші розробники одночасно виконують злиття коду, що перекривається. На щастя, магістральна технологія скорочує такі конфлікти.

Основні переваги магістрального підходу:

- Неперервна інтеграція коду.
- Неперервна перевірка коду.
- Послідовні релізи коду в робоче середовище.
- Магістральна розробка та CI/CD

Модель магістральної розробки передбачає наявність репозиторію зі стабільним потоком комітів, які надходять у основну гілку. Безперервна інтеграція досягається шляхом додавання набору автоматизованих тестів та моніторингу покриття коду для цього потоку комітів. Після злиття нового коду із магістраллю виконується перевірка його якості. Для цього запускаються автоматизовані тести з інтеграції та покриття коду.

При частих та невеликих коммітах у системах магістральної розробки перевірка коду стає більш ефективною. При малому розмірі гілок розробники можуть швидко

переглядати та перевіряти невеликі зміни. Це набагато простіше, ніж працювати з довгостроковою функціональною гілкою, в якій перевіряльнику доводиться читати кілька сторінок коду або вручну перевіряти велику область змін коду.

Командам слід виконувати часті щоденні злиття з основною гілкою. Ця модель розробки передбачає, що магістральна гілка залишається зеленою, тобто готова до розгортання на будь-якому коміті. Автоматизовані тести, злиття коду та його перевірка дозволяють гарантувати, що код проєкту, що розробляється за моделлю з магістральною гілкою, готовий до розгортання у робочому середовищі у будь-який час. Завдяки цьому команда отримує можливість виконувати часті розгортання у робочому середовищі та ставити подальші цілі щодо щоденних релізів.

Магістральна розробка та CI/CD

У міру зростання популярності безперервної інтеграції та безперервного постачання моделі розгалуження були вдосконалені та оптимізовані, що призвело до поширення розробки на основі магістральної гілки. Сьогодні магістральна розробка є обов'язковою умовою безперервної інтеграції. У системах безперервної інтеграції розробники застосовують магістральну розробку разом із автоматичними тестами, виконуваними після кожного комміту в магістральну гілку. Це гарантує, що проєкт залишається працездатним будь-якої миті.

Магістральна розробка дозволяє командам швидко та послідовно випускати код.

Прийоми та методи, які допоможуть скоротити робочий цикл команди та оптимізувати графік релізів:

- Розробка невеликими пакетами.
- Прапорці функції.
- Впровадження комплексного автоматизованого тестування.
- Впровадження асинхронної перевірки коду.
- Впровадження в репозиторії коду програми не більше трьох активних гілок.
- Впровадження злиття в магістральну гілку не рідше одного разу на день.
- Скорочення кількості заборон на зміну коду та етапи інтеграції.
- Виконання складання швидко та тестування негайно.

Магістральна розробка підтримує швидкий цикл постачання коду до робочого середовища. Якщо порівнювати таку модель розробки з музикою, вийде швидке стакато: короткі, лаконічні ноти, які швидко змінюють одна одну, і ці ноти — комміти в репозиторій. Малий розмір коммітів та гілок забезпечує прискорений темп злиття та розгортання.

Невеликі зміни в результаті кількох комітів або модифікація декількох рядків коду зводять до мінімуму когнітивні витрати. Командам набагато простіше вести змістовні обговорення та швидко приймати рішення при розгляді обмеженої області коду, а не великої кількості змін.

Прапорці можливостей чудово доповнюють модель магістральної розробки, дозволяючи розробникам поміщати нові зміни до неактивного шляху коду та активувати його пізніше. Це дає можливість відмовитися від створення функціональної гілки в окремому репозиторії і натомість робити комміти для нової функції безпосередньо в основну гілку, поміщаючи їх у шлях, позначений прапорцем можливості.

Функціональні прапори підтримують принцип оновлення невеликими пакетами. Замість створення функціональної гілки та очікування складання повної специфікації, розробники можуть виконати коміт у магістральну гілку, вводячи прапор функції, та потім будуть відправляти нові комміти магістральної гілки, у яких специфікація функції реалізується у межах прапора.

Автоматичне тестування необхідне для будь-якого сучасного проєкту розробки з методології CI/CD. Існують різні типи автоматичних тестів, що виконуються на різних етапах конвеєра релізів. Короткострокові модульні та інтеграційні тести виконуються у процесі розробки та після злиття коду. Більш тривалі та комплексні наскрізні тести виконуються на пізніших етапах конвеєра на основі всього розділу проіндексованих файлів або робочого середовища.

Автоматичні тести запускаються для невеликих пакетів зі злиття розробниками нових коммітів, підтримуючи процес магістральної розробки. Набір автоматичних тестів перевіряє код на наявність будь-яких проблем та автоматично підтверджує чи відхиляє його. Це дозволяє розробникам швидко створювати комміти та автоматично тестувати їх, щоб дізнатися, чи не виникли нові проблеми.

При магістральній розробці перевірка коду має виконуватися негайно, а не плануватися на майбутнє в асинхронної системі. Автоматичні тести утворюють рівень запобіжної перевірки коду. Коли розробники будуть готові перевірити запит **pull** від учасника команди, вони зможуть спочатку подивитися, чи позитивний результат автоматичних тестів та чи розширилося покриття коду. Так перевіряльники можуть переконатися, що новий код відповідає певним специфікаціям. Після цього перевірку можна звести до оптимізації.

Після злиття гілки її рекомендується видалити. Репозиторій з великою кількістю активних гілок має низку неприємних побічних ефектів. Командам може бути корисно дізнаватися з активних гілок, яка робота ведеться, проте за наявності

застарілих та неактивних гілок ця можливість зникає. Деякі розробники використовують інтерфейси Git, які при завантаженні великої кількості віддалених гілок починають працювати повільно.

Високопродуктивні команди, що застосовують модель магістральної розробки, повинні закривати та об'єднувати будь-які відкриті та готові до злиття гілки хоча б раз на день. Це допомагає підтримувати періодичність та задає графік відстеження релізів. Потім наприкінці дня команда може відзначити магістральну гілку як коміт релізу. Це має корисний побічний ефект у вигляді генерації щоденного гнучкого інкременту релізу.

У командах, що наслідують принципи AGILE та практикують безперервну інтеграцію та безперервне постачання, не повинно виникати потреби у планових заборонах на зміни коду або у перервах на етапі інтеграції, хоча організація може використовувати ці механізми з інших причин. У CI/CD принцип безперервності передбачає, що оновлення виконуються постійно. Команди, які ведуть магістральну розробку, повинні по можливості уникати заборон на зміни коду, які зупиняють усі процеси, та планувати свою роботу так, щоб конвеєр релізів не зупинявся.

Для підтримки високої частоти релізів необхідно оптимізувати час складання та тестування. Інструменти збирання в CI/CD повинні в міру необхідності застосовувати шари кешування, щоб уникнути дорогих обчислень на основі статичних ресурсів. Тести мають бути оптимізовані так, щоб не виконувати зайвих звернень до сторонніх служб.

Магістральна розробка сьогодні є стандартом для високопродуктивних команд розробки, оскільки такий підхід задає та підтримує високу частоту релізів на основі спрощеної стратегії розгалуження Git. Крім того, магістральна розробка забезпечує командам розробки більшу гнучкість та контроль над процесом постачання програмного забезпечення кінцевому користувачеві.

8.3. Робочий процес Git-flow Workflow

Git-flow - це застаріла версія робочого процесу Git, яка свого часу стала принципово новою стратегією управління гілками в Git. Популярність Git-flow стала знижуватися під впливом магістральних робочих процесів, які сьогодні вважаються кращими для сучасних схем безперервної розробки ПЗ і застосування DevOps. Крім того, Git-flow не надто зручно застосовувати у процесах CI/CD.

Git-flow – альтернативна модель розгалуження **Git**, в якій використовуються функціональні гілки та декілька основних гілок. Ця модель була вперше опублікована

та популяризована Вінсентом Дрісеном на сайті **nvie**. У порівнянні з моделлю магістральної розробки, в **Git-flow** використовується більше гілок, кожна з яких існує довше, а коміти зазвичай більші. Відповідно до цієї моделі розробники створюють функціональну гілку і відкладають її злиття з головною магістральною гілкою до завершення роботи над функцією. Такі довгострокові функціональні гілки вимагають тісної взаємодії розробників при злитті та створюють підвищений ризик відхилення від магістральної гілки. В них також можуть бути конфліктні оновлення.

Git-flow можна використовувати для проєктів, у яких заплановано цикл релізів та реалізується характерна для **DevOps** методика безперервного постачання. У цьому процесі використовуються поняття та команди, які були запропоновані в рамках робочого процесу з функціональними гілками. Однак **Git-flow** привносить нові специфічні ролі для різних гілок та визначає характер та частоту взаємодії між ними. Крім функціональних гілок у рамках цього робочого процесу використовуються окремі гілки для підготовки, підтримки та реєстрації релізів. При цьому ви, як і раніше, можете користуватися перевагами робочого процесу з функціональними гілками, такими як запити pull, ізольовані експерименти та ефективна командна взаємодія.

Git-flow – це методика роботи з **Git;** що визначає, які види гілок необхідні проєкту та як виконувати злиття між ними. Набір інструментів git-flow є окремою утилітою командного рядка, яка вимагає установки.

Установку у системах OS X можна виконати командою **brew install git-flow**. Якщо ви використовуєте Windows, вам потрібно завантажити та встановити **git-flow**. Після встановлення рішення **git-flow** необхідно виконати команду **git-flow init**, щоб використовувати його у проєкті. Цей набір інструментів відіграє роль обгортки Git. Команда **git-flow init** є розширенням стандартної команди **git init** і не вносить змін до репозиторій, крім створення гілок.

Гілка розробки та головна гілка в Git-flow

Git-flow реєструє історію проєкту починаючи з двох гілок – головної (**main),** що зберігає офіційну історію релізу та гілки розробки **(develop).** Для зручності рекомендується надавати всім коммітам у гілці **main** номер версії.

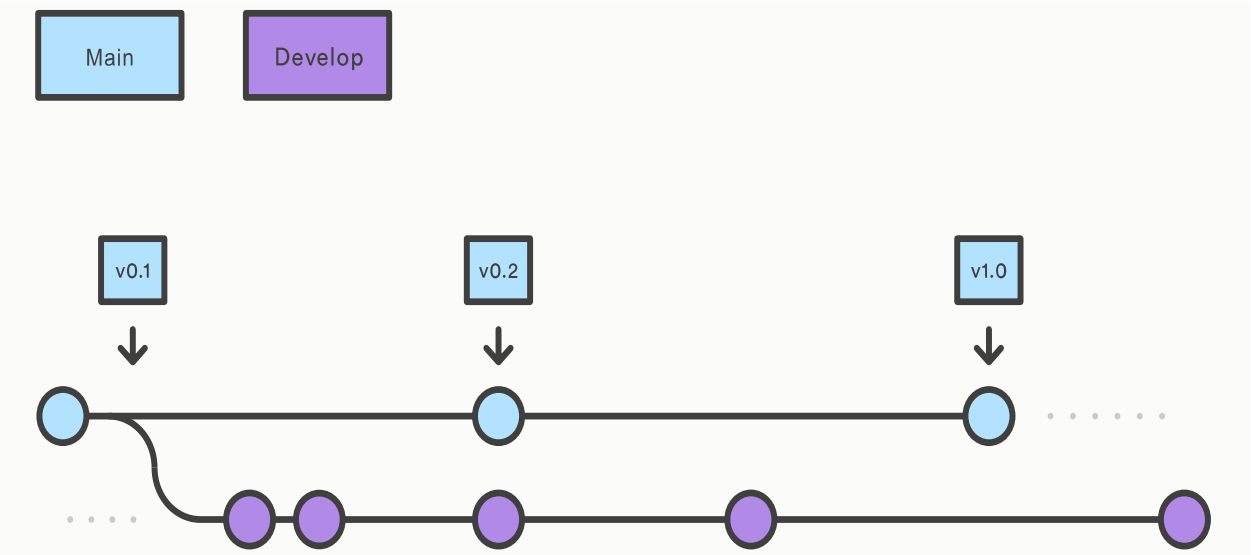

Рис. 18.10 Дві головні гілки початку проєкту

Перший крок робочого процесу полягає у створенні гілки **develop** від стандартної гілки **main** (рис. 18.10). Розробнику буде простіше створити порожню гілку **develop** локально та відправити її на сервер.

```
git branch develop
git push -u origin develop
```

Гілка **develop** буде зберігати повну версію проєкту, а **main** скорочену.

При використанні бібліотеки розширення git-flow, для створення гілки **development** можна виконати команду **git flow init** в існуючих репозиторіях: При використанні бібліотеки розширення **git-flow**, для створення гілки **development** можна виконати команду **git flow init** в існуючих репозиторіях.

Функційні гілки (feature)

Під кожну нову функцію виділяють власну гілку, яку можна відправити в центральний репозиторій для створення резервної копії команди або спільної роботи. Гілки функції створюються не на основі **main**, а на основі **development**. Коли робота над функцією завершується, відповідна вітка зливається з гілкою **development**. Функції не слід відправляти безпосередньо у гілку **main**.

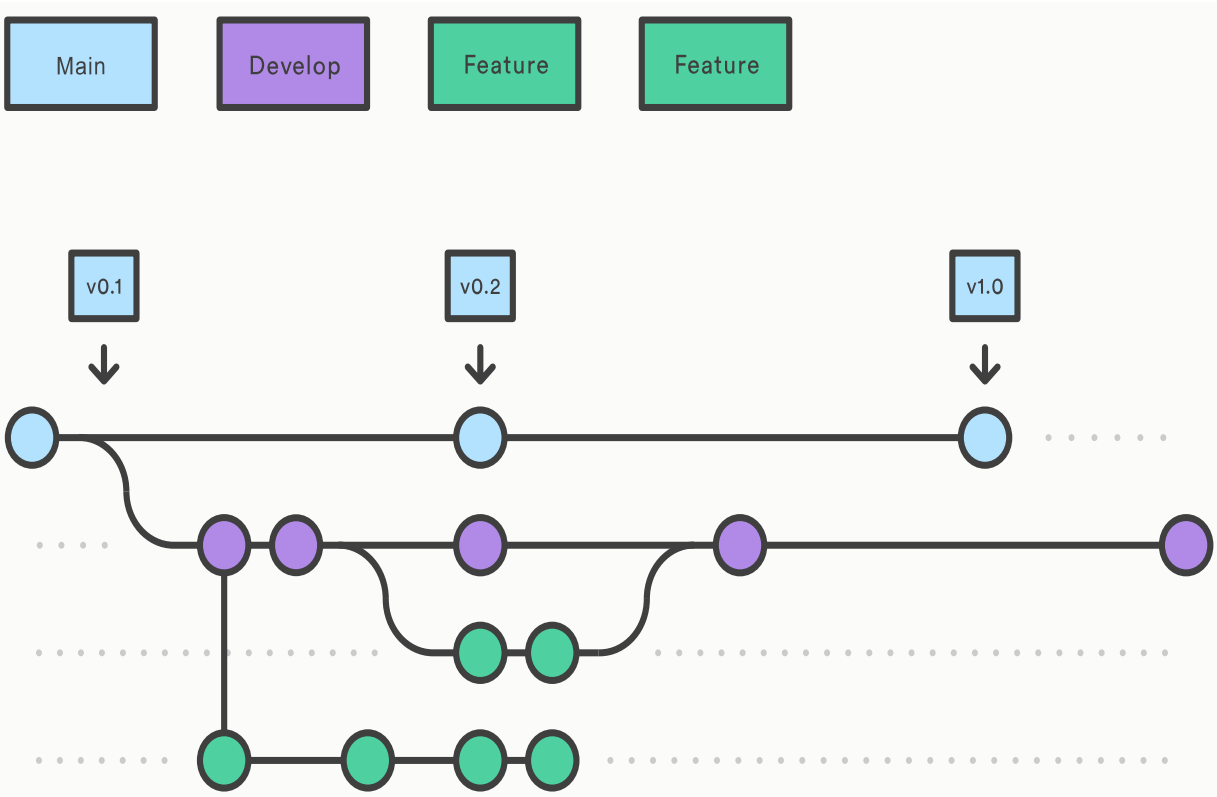

Рис. 18.11 Функційні гілки

Зверніть увагу, що комбінація гілок **feature** з гілкою **develop** фактично є робочим процесом з функціональними гілками (рис. 18.11). Але робочий процес **Git-flow** на цьому не закінчується. Як правило, гілки **feature** створюються на основі останньої гілки **develop**.

Створення функційної гілки

Без застосування розширення **Git-flow**.

```
git checkout develop
git checkout -b feature_branch
```

З застосуванням розширення **Git-flow**.

```
git flow feature start feature_branch
```

Після роботи над функцією слід об'єднати гілку **feature_branch** з **develop**.

Без застосування розширення **Git-flow**.

```
git checkout develop
git merge feature_branch
```

З застосуванням розширення **Git-flow**.

```
git flow feature finish feature_branch
```

Гілки випуску (release)

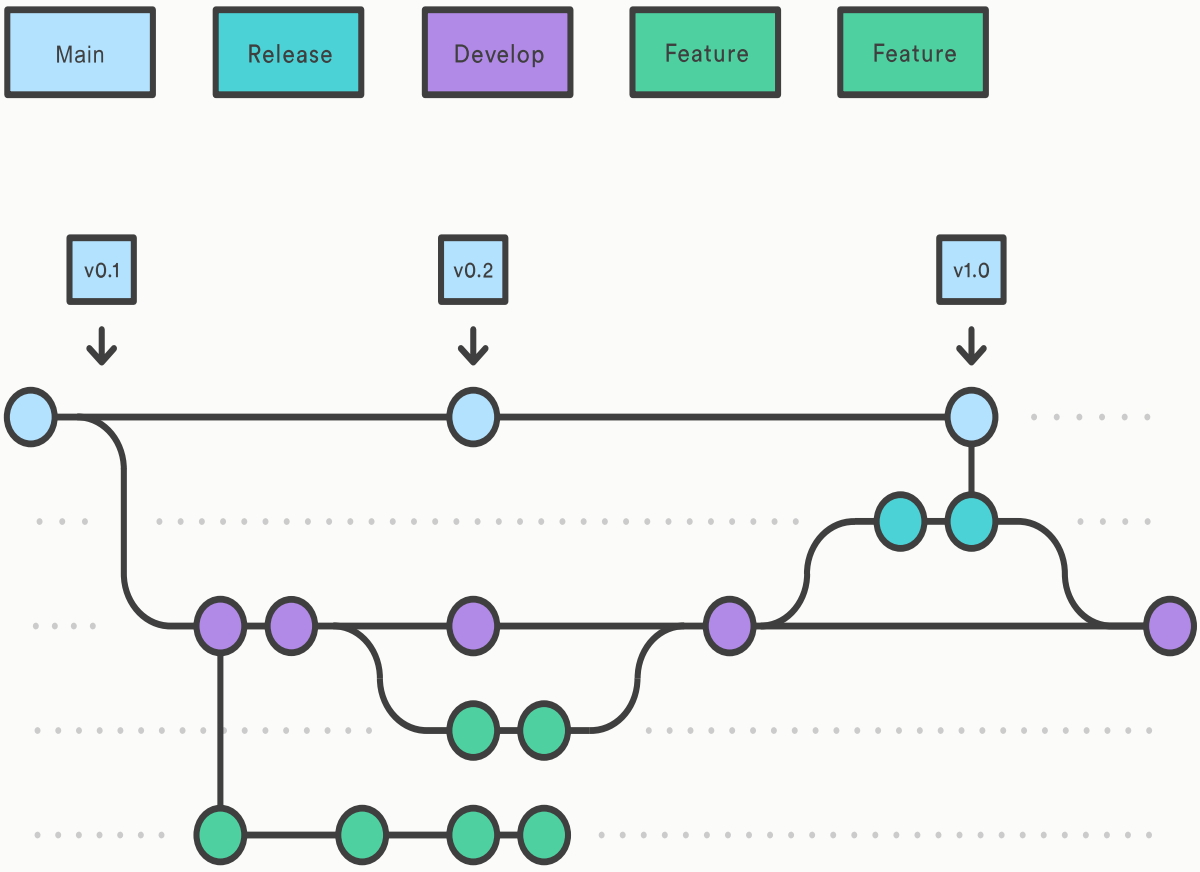

Рис. 18.12 Гілки випуску

Коли у гілці **develop** виявляється достатньо функцій для випуску (або наближається призначена дата релізу), від гілки **develop** створюється гілка **release** (рис. 18.12). Створення цієї галузі запускає наступний цикл релізу, і з цього моменту нові функції додати більше не можна — допускається лише виправлення багів, створення документації та вирішення інших завдань, пов'язаних з релізом. Коли підготовка до постачання завершується, гілка **release** зливається з **main** і присвоюється

номер версії. Крім того, потрібно виконати її злиття з гілкою **develop**, в якій з моменту створення гілки релізу могли виникнути зміни.

Завдяки тому, що для підготовки випусків використовується спеціальна гілка, одна команда може доопрацьовувати поточний випуск, тоді як інша команда продовжує роботу над функціями наступного. Це також дозволяє розмежувати етапи розробки (наприклад, можна легко присвятити тиждень підготовці до версії 4.0 і дійсно побачити це в структурі репозиторію).

Створення гілок **release** це ще одна проста операція розгалуження. Як і гілки **feature**, гілки **release** засновані на гілці **develop**. Створити нову гілку **release** можна за допомогою наступних команд.

Без застосування розширення **Git-flow**.

```
git checkout develop
git checkout -b release/0.1.0
```

З застосуванням розширення **Git-flow**.

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
```

Коли підготовка до постачання завершується, реліз зливається з гілками **main** і **develop**, а гілка **release** видаляється. Важливо злити її з гілкою **develop**, оскільки у гілку **release** могли додати критичні оновлення, які мають бути доступні для нових функцій. Якщо у вашій організації приділяють особливу увагу перевірці коду, це ідеальне місце для виконання запиту **pull**.

Для завершення роботи у гілці **release** використовуйте такі команди:

Без застосування розширення **Git-flow**.

```
git checkout main
git merge release/0.1.0
```

З застосуванням розширення **Git-flow**.

```
git flow release finish '0.1.0'
```

Гілки виправлення (hotfix)

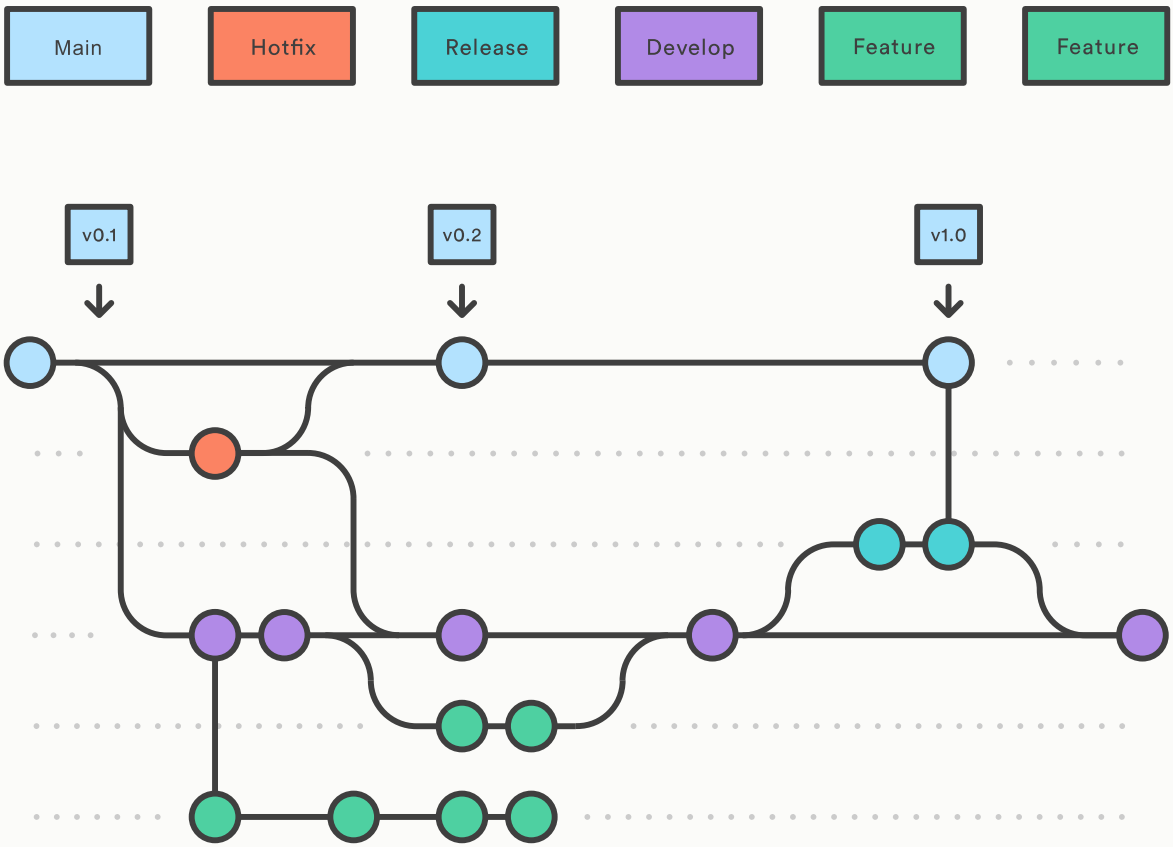

Рис. 18.13 Гілки виправлення

Гілки супроводу або виправлення (**hotfix**) використовуються для швидкого внесення виправлень у робочі релізи (рис. 18.3). Гілки **hotfix** дуже схожі на гілки **release** і **feature**. Відмінність полягає в тому, що вони створюються на основі **main**, а не **develop**. Це єдина гілка, яку потрібно обов'язково створювати безпосередньо від **main**. Як тільки виправлення завершено, цю гілку слід злити з **main** і **develop** (або поточною гілкою **release**), а гілці **main** присвоїти оновлений номер версії.

Завдяки спеціальній гілці для виправлення помилок команда може усувати проблеми, не перериваючи решту робочого процесу і не чекаючи наступного циклу релізу. Гілки супроводу можна розглядати як спеціальні гілки **release**, які працюють безпосередньо з **main**. Гілку **hotfix** можна створити за допомогою наступних команд.

Без застосування розширення **Git-flow**.

```
git checkout main
git checkout -b hotfix_branch
```

З застосуванням розширення **Git-flow**.

```
$ git flow hotfix start hotfix_branch
```

Після завершення роботи з гілкою **hotfix** її зливають з **main** і **develop** (як і у випадку з гілкою **release**).

```
git checkout main
git merge hotfix_branch
git checkout develop
git merge hotfix_branch
git branch -D hotfix_branch
```

```
$ git flow hotfix finish hotfix_branch
```

Приклад

Розглянемо повний цикл роботи з функціональною гілкою (передбачається, що ми маємо репозиторій з гілкою **main**).

```
git checkout main
git checkout -b develop
git checkout -b feature_branch
# work happens on feature branch
git checkout develop
git merge feature_branch
git checkout main
git merge develop
git branch -d feature_branch
```

Крім роботи з гілками **feature** та **release**, продемонструємо роботу з гілкою **hotfix**:

```
git checkout main
git checkout -b hotfix_branch
# work is done commits are added to the hotfix_branch
git checkout develop
git merge hotfix_branch
git checkout main
git merge hotfix_branch
```

Ключові ідеї, які потрібно запам'ятати про **Git-flow**:

- Ця модель відмінно підходить для організації робочого процесу на основі релізів.
- Робота за моделлю **Git-flow** передбачає створення спеціальної гілки для виправлення помилок у робочому релізі.

Послідовність дій під час роботи за моделлю Git-flow:

1. З гілки **main** створюється гілка **develop**.
2. З гілки **develop** створюється гілка **release**.
3. З гілки **develop** створюються гілки **feature**.
4. Коли робота над гілкою **feature** завершується, вона зливається у гілку **develop**.
5. Коли робота над гілкою **release** завершується, вона зливається з гілками **develop** та **main**.
6. Якщо у гілці **main** виявляється проблема, з **main** створюється гілка **hotfix**.
7. Коли робота над гілкою **hotfix** завершується, вона зливається з гілками **develop** та **main**.

Питання до розділу 8

1. Зміст об’єкту фіксації Git?
2. Опишіть фіксацію об’єктів в Git/
3. Створення нової гілки в Git?
4. Переключення гілок в Git?
5. Зливання гілок в Git?
6. Які підходи до ведення в системі керування версіями проєкту ви знаєте?
7. Що представляє магістральний підхід?
8. Що представляє **Git-flow** підхід?
9. В чому переваги магістрального підходу?

10. Які ви знаєте рекомендації до магістрального підходу в веденні проєкту?
11. Які гілки ведення проєкту в **Git-flow** ви знаєте?
12. В чому головні особливості для функційних, релізних та гілках виправлення в проєкті?
13. Перелічіть послідовність дій під час роботи з **Git-flow**?