МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп’ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_______ Олександр ЛИТВИНЕНКО
“_____”____________ _____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Програмний засіб контролю версій для корпоративної платформи бізнес аналітики

Виконавець: _______________________________ Ігор КУЛИБКО

Керівник: _______________________________ Надія МАРЧЕНКО

Нормоконтролер: _______________________________ Євгеній ТУПОТА

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп’ютерних наук та технологій

Кафедра комп’ютеризованих систем правління

Спеціальність 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування»

Форма навчання денна

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____________ Олександр ЛИТВИНЕНКО

«_______»_____________________2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Кулибка Ігоря Віталійовича

1. Тема кваліфікаційної роботи «Програмний засіб контролю версій для корпоративної системи бізнес аналітики»

затверджена наказом ректора від «28» 08 2023 р. №1494/ ст

2. Термін виконання роботи (проєкту): з 02.10.2023 по 31.12.2023

3. Вихідні дані до роботи (проєкту): мова програмування *C#*, середовище розробки *Visual Studio 2022*, бібліотеки *libgit2sharp, Windows.Forms*

4. Зміст пояснювальної записки: вступ, аналіз потреб та вимог, розробка програмного засобу контролю версій для корпоративної платформи бізнес аналітики

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1) Схема алгоритму роботи системи контролю версій до внесення в історію

2) Схема алгоритму виявлення конфліктів під час комміту

3) Схема алгоритму комміту в систему контролю версій

4) Схема алгоритму мерджа в системі контролю версій

5) Схема алгоритму детекції та вирішення конфліктів

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Отримати тему роботи	02.10.2023	
2	Зібрати і ознайомитися з інформацією згідно з темою роботи	03.10.2023 – 06.10.2023	
3	Обрати середовище розробки для виконання поставленої задачі	09.10.2023 – 11.10.2023	
4	Ознайомитися з принципами систем контролю версій	12.10.2023 – 17.10.2023	
5	Проаналізувати потреби для систем контрою версій	18.10.2023 – 23.10.2023	
6	Розробити необхідні компоненти програмного продукту	24.10.2023 – 17.11.2023	
7	Написати функціональну частину програми	18.11.2023 – 01.12.2023	
8	Виконати тестування розробленого програмного продукту	04.12.2023 – 08.12.2023	
9	Оформити графічний матеріал	11.12.2023 – 15.12.2023	
10	Оформити пояснювальну записку	18.12.2023 – 22.12.2023	
11	Оформити презентацію по темі	25.12.2023 – 29.12.2023	

7. Дата видачі завдання: «02» 10. 2023 р.

Керівник кваліфікаційної роботи ____________________ Надія МАРЧЕНКО

Завдання прийняв до виконання ____________________ Ігор КУЛИБКО

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи «Програмний засіб контролю версій для корпоративної платформи бізнес аналітики»: 98 с., 27 рис., 33 літературних джерела, 1 додаток.

Ключові слова: СИСТЕМА КОНТРОЛЮ ВЕРСІЙ, МЕРДЖ, ПОРІВНЯННЯ, GIT

Об’єкт дослідження – опанування контролю версій в системі бізнес аналітики.

Предмет дослідження – програмний засіб контролю версій в системі бізнес аналітики.

Мета кваліфікаційної роботи – розробка програмного засобу для контролю версій документів корпоративної платформи бізнес аналітики.

Методи дослідження: аналіз існуючих програмних засобів для контролю версій, порівняння, аналізу історії, злиття.

Технічні та програмні засоби: мова програмування *C#,* фреймворки *libgit2sharp, Windows.Forms*.

Отримані результати: програмний засіб контролю версій для корпоративної платформи бізнес аналітики.

Наукова новизна: створення механізму контролю версій для файлів корпоративної платформи бізнес аналітики, із функціоналом повноцінного перегляду модифікацій, додавання, видалення елементів та їх налаштувань.

ЗМІСТ

ВСТУП 6
РОЗДІЛ 1 АНАЛІЗ ДОЦІЛЬНОСТІ СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ 8
1.1. Завдання та роль систем контролю версій в корпоративних платформах ... 8
1.2. Переваги та недоліки системи контролю версій 13
1.3. Аналіз потреб до системи контролю версій 23
1.4. Висновки до розділу 26
РОЗДІЛ 2 ОЦІНКА ОСНОВНИХ ВИМОГ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ 28
2.1. Аналіз функціональних вимог 28
2.2. Аналіз нефункціональних вимог 45
2.3. Визначення критеріїв ефективності 52
2.4. Вибір мови програмування 60
2.5. Вибір фреймворку 61
2.6. Вибір середовища розробки 63
2.7. Висновки до розділу 64
РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ КОНТРОЛЮ ВЕРСІЙ ДЛЯ КОРПОРАТВНОЇ ПЛАТФОРМИ БІЗНЕС АНАЛІТИКИ 67
3.1. Розробка загального алгоритму робочого процесу 67
3.2. Структура мерджа та комміту системи контролю версій 75
3.3. Структура вирішення конфліктів 78
3.4. Інструкція із налаштування та демонстрація інтерфейсу 82
3.5. Випробовування програмного засобу 86
3.6. Висновки до розділу 90
ВИСНОВКИ 93
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ 96
ДОДАТОК А 99

ВСТУП

У сучасному світі корпоративної бізнес-аналітики змінюється характер інформаційних процесів і набирає насичення змістовний аспект організаційного розвитку. У зв'язку з цим роль програмного забезпечення у рішеннях, спрямованих на реалізацію бізнес-аналітичних задач, стає все більш значущою. Параметри якості і ефективності програмних засобів, що використовуються в бізнес-аналітиці корпоративних платформ, мають безпосередній вплив на результати діяльності підприємства і дозволяють оптимізувати поточні процеси.

Серед програмних інструментів для корпоративних бізнес-аналітичних систем значне місце посідають засоби контролю версій. Робота з версіями даних та коду є однією з фундаментальних потреб в керуванні проектами, планування ресурсів підприємства та контролю якості.

Ця кваліфікаційна робота присвячена розробці програмного засобу контролю версій для корпоративної платформи бізнес-аналітики, що є актуальною і важливою темою з огляду на наростання запитів щодо ефективності та розширення функціоналу аналітичних систем. Адекватний контроль версій, досяжний через програмне забезпечення, дозволяє забезпечити зберігання, обробку, аналіз та відтворення інформації в межах корпоративної платформи. Це виключає конфлікти, випадкові зміни, втрату даних та передбачає гнучкість відновлення попередніх версій за різних сценаріїв використанні. Крім того, забезпечується безпека інформації, менеджмент даних та можливість аналізу еволюції проектів за різний час.

Програмний засіб контролю версій, що буде розроблений в рамках цієї кваліфікаційної роботи, матиме значне практичне застосування. Він допоможе у підвищенні продуктивності корпоративних аналітичних систем та у розвитку проектів з використанням надійних, гнучких та ефективних рішень для управління ресурсами підприємства.

Важливість розробки такого програмного засобу полягає в його практичному застосуванні для підвищення продуктивності корпоративних аналітичних систем та

розвитку проектів. Це сприятиме використанню надійних та ефективних рішень для управління ресурсами підприємства.

Отже, розробка програмного засобу контролю версій для корпоративної платформи бізнес-аналітики є актуальним завданням, оскільки вона сприятиме оптимізації бізнес-процесів, поліпшенню роботи корпоративних аналітичних систем та забезпеченню високої якості рішень на підприємствах.

РОЗДІЛ 1
АНАЛІЗ ДОЦІЛЬНОСТІ СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ

1.1. Завдання та роль систем контролю версій в корпоративних платформах

В сучасному динамічному бізнес-середовищі, організації постійно прагнуть знайти нові та ефективні інструменти для оптимізації своєї роботи, особливо в контексті інформаційних технологій та аналітики. Відповідно до зростання потреби у сучасних та ефективних інструментах для управління інформаційно-аналітичними процесами, система контролю версій може відігравати роль стратегічного активу, що стимулює організації реагувати на зміни основних показників управління. Таким чином, система контролю версій визначається й розглядається як доцільний інструмент для поліпшення корпоративної платформи бізнес-аналітики.

По-перше, система контролю версій є важливим інструментом для підвищення оперативності, стабільності та безпеки розробки аналітики та її використання. Вона дозволяє відстежувати зміни, що вносяться окремими співробітниками, а також забезпечити доступ до актуальної історії змін та різних версій проектів. Така цілісна згрупування інформації може підвищити рівень контролю над розвитком проектів, поліпшити комунікацію між аналітиками з різних відділів та віддалених членів команди. Водночас, компанія може контролювати відродження та інтеграцію систем, що забезпечує інформаційну безпеку та законність авторства розвитку файлів.

По-друге, система контролю версій може надати можливість об'єктивної оцінки внесених змін у технічні, аналітичні та методологічні аспекти проекту. Це сприяє підтримці належного рівня якості роботи, підвищенню надійності та прогнозованості результатів роботи аналітиків. Така можливість також відіграє важливу роль у контексті управління інтелектуальною власністю, забезпечуючи прозорість авторства та правовий захист знань і досвіду співробітників. Це забезпечує розвиток підприємства, котрий ґрунтується на послідовному збереженні та використанні накопиченого досвіду та знань протягом років роботи.

По-третє, система контролю версій є необхідним інструментом для підтримки стійкої інфраструктури та безперервної інтеграції у контексті стрімкого розвитку інформаційних технологій та пов'язаних з ними змін у конфігурації. Оскільки аналітичні методики та технології швидко розвиваються, система контролю версій дозволяє помітно скоротити час, потрібний для розробки, впровадження та адаптації до змін. Система контролю версій перешкоджає втраті великої кількості часу та зусиль співробітників до усвідомлення процесів, відстеження змін і сумісності завдань. Завдяки єдності проектів буде забезпечено послідовний розвиток аналітичної платформи.

Отже, проведений аналіз дає підстави стверджувати, що впровадження системи контролю версій у корпоративній платформі бізнес-аналітики є доцільним напрямком розробки. Така система завдяки своєму впровадженню надасть низку суттєвих переваг, серед яких: забезпечення ефективної співпраці команди, підвищення оперативності і контролю над проектами, поліпшення аналітичних процесів та полегшення інтеграції аналітичної платформи у вищих рівнях корпоративного управління. Система контролю версій може сприяти підвищенню конкурентоспроможності на ринку так і розвитку підприємства на інноваційні компоненти.

Система контролю версій – це інфраструктура, що відповідає за відстеження, зберігання та управління змінами у різних компонентах проекту, включаючи код програм, документацію, конфігурації та інше. Ця система дозволяє ефективно координувати роботу команди розробників, фіксувати зміни та згодом повернутися до попередніх версій в разі необхідності. Ось більш детальний опис функціонування типової системи контролю версій:

Ініціалізація репозиторію: Процес розпочинається із створення центрального сховища (репозиторію), у якому будуть зберігатися усі файли проекту та їх історія змін. Репозиторій може бути заснований на локальному сервері, хмарному сервісі або іншому рішенні користувачів.

Створення робочої копії: Розробники взаємодіють за допомогою репозиторія шляхом створення локальних робочих копій проекту на своїх комп'ютерах. Кожен

розробник працює над своїми завданнями, вносячи зміни в код, документи або конфігурації. У системах розділених на клієнт-сервер та розподілених системах контролю версій (наприклад *Git*) будуть існувати деякі відмінності у методах роботи.

Фіксація змін: Після розробки нових функцій або внесення будь-яких змін, розробник вносить зміни у свою локальну робочу копію, потім створює та завантажує коміт в репозиторій з повідомленням про виконані зміни.

Синхронізація змін: Регулярна синхронізація змін з репозиторієм є ключовою частиною процесу. Розробники завантажують зміни з центрального репозиторія на свої локальні робочі копії щовечора, щоб координувати свою роботу з іншими членами команди.

Загально-подана політика: В системах контролю версій можуть бути також й інструменти для рецензування та оцінки коду перед зливанням з головною (основною) гілкою. Це допомагає уникнути помилок та конфліктів з основною гілкою.

Зливання версій та управління гілками: Часто в розробці використовуються гілки, які допомагають розподілити різні функціональності та забезпечити їх паралельну розробку злиттям версій в загальнодоступній структурі процесу.

Резервування та відновлення даних: Система контролю версій забезпечує створення резервних копій всіх версій коду, документів та конфігурацій, що забезпечує безпеку даних та можливість їх відновлення після ситуацій відмов або втрати.

Впровадження інформаційних технологій в різні сферами діяльності людини посилює потребу в сучасних, ефективних та гнучких технологіях управління інформаційно-аналітичними процесами. Одним з таких інноваційних рішень є впровадження систем контролю версій в корпоративні міжгалузеві платформи. Системи контролю версій - це механізми, що надають можливість відстежувати зміни в коді програми або будь-яких інших даних, а також контролювати і управляти цими змінами.

Основне завдання системи контролю версій полягає в фіксації змін у коді, документації, конфігураціях або інших компонентах проекту, дозволяючи відновити проект до будь-якого попереднього стану, як показано на рис. 1.1.

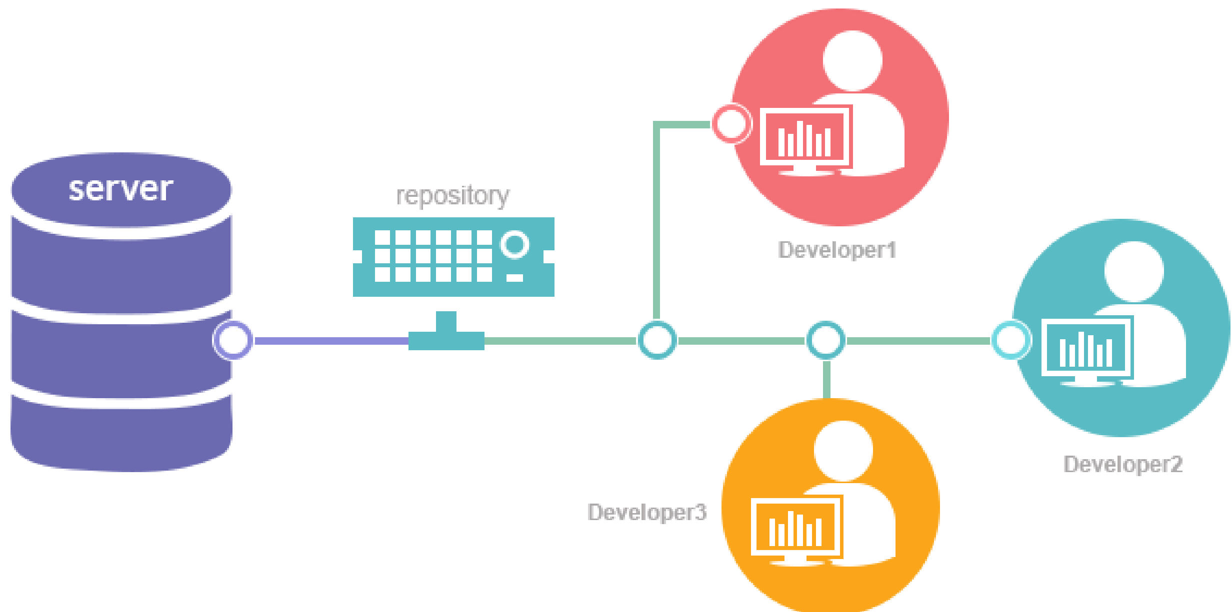


Рис. 1.1. Ієрархія розподіленої системи, як служби проміжного рівня

Системи контролю версій дозволяють забезпечити відновлення попередньої версії проекту в разі помилок у роботі програми, некоректних рішень або будь-яких інших недоліків поточної версії. Даний функціонал важливий для команд розробників та аналітиків, оскільки він спрощує і прискорює знаходження й вирішення помилок, а також відновлення належних параметрів проекту. Система контролю версій допомагає забезпечувати сумісність усіх версій в розробці або підтримці, гарантує відсутність конфліктів, що показано на рис. 1.2, між окремими змінами в програмі. Сумісність версій є основою стабільної роботи контрольованого проекту, забезпечуючи довіру до даних та результатів оракулів корпоративної платформи.

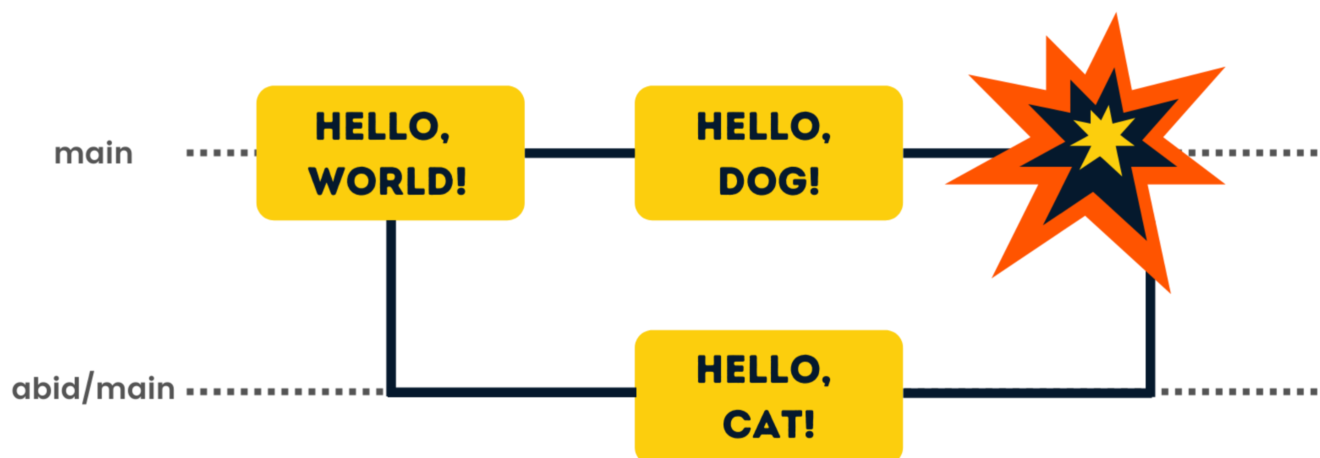


Рис. 1.2. Відстеження конфліктів

Однією із переваг системи контролю версій є можливість відстеження авторства змін. Кожному співробітнику може бути присвоєно авторство конкретних модифікацій. Це дозволяє створювати прозорий та ефективний робочий процес із взаємодією команди, а також виявляти й усувати слабкі сторони.

Збереження резервних копій проекту: Надійність проекту в умовах постійних змін є важливим аспектом для роботи корпоративної платформи, і система контролю версій допомагає у цьому. Це реалізується за рахунок створення та зберігання резервних копій проекту водночас з різними версіями файлів. В результаті, у випадку відмови або втрати даних будь-якої з версій, користувачі матимуть можливість відновити потрібний стан системи.

Серед інших важливих завдань систем контролю версій можна виділити: рецензування коду, автоматизацію процесів, створення гілок та злиття версій, налаштування індивідуальних налаштувань параметрів платформи та їх синхронізації, ефективне планування ресурсів, просунуте виправлення та оптимізація ділових інформаційних систем.

У контексті бізнес-аналітики, система контролю версій дозволяє слідкувати за ходом змін у концептуальних, методологічних та технічних аспектах аналітичних процесів. Вона забезпечує контроль за якістю введених змін, надає інструменти для

аналізу ефективності внесених змін. Основні завдання системи контролю версій у бізнес-аналітиці - це забезпечення безперебійного функціонування аналітичних систем, поліпшення якості прийняття рішень на основі аналітичних даних, підвищення ефективності управлінських рішень.

Насамперед, система контролю версій забезпечує збереження історії змін аналітичних процесів та змістовних компонентів ділової інформації. Це дозволяє аналітикам здійснювати об'єктивне оцінювання ефективності змін в рамках проекту, а також повертатися до попередніх версій коду або конфігурацій при виявленні помилок.

Отже, впровадження систем контролю версій у корпоративні платформи може підвищити ефективність роботи організації, забезпечивши цілісність, доступність та сумісність аналітичних даних. Крім того, можна стверджувати, що системи контролю версій відіграють критичну роль в корпоративних платформах, сприяючи успішній роботі команди розробників, захисті даних та інтелектуальної власності, а також забезпеченню стабільності та ефективності розвитку програмного продукту. Впровадження таких систем у свою корпоративну платформу є інвестицією в майбутнє успішної організації.

1.2. Переваги та недоліки системи контролю версій

Системи контролю версій в корпоративних платформах відіграють ключову роль, організуючи робочий процес команди розробників та впливаючи на ефективність розробки програмного забезпечення. Однак, необхідно ретельно проаналізувати переваги та недоліки, щоб як найточніше інтегрувати такі системи в корпоративному оточенні.

Значні переваги систем контролю версій охоплюють підвищення ефективності командної співпраці, контроль над змінами, відтворення результатів та автоматизація процесів. Ці аспекти роблять з систем контролю версій незамінний інструмент в умовах корпоративного розвитку програмного забезпечення (рис. 1.3).

Однак, деякі недоліки таких систем також варто відмітити - технічні складності впровадження для користувачів, які поки що мало знайомі з такими інструментами. Це може призвести до потреби підвищення рівня кваліфікації співробітників, що може зайняти певний період часу та ресурсів.

Рис. 1.3. Переваги системи контролю версій

Системи контролю версій надають ряд переваг для корпоративних платформ та їх команд розробників, а саме:

– Співпраця є однією з найважливіших переваг систем контролю версій в корпоративних платформах. За допомогою таких систем, процес співпраці між командами розробників стає більш ефективним та організованим. Розглянемо детальніше, як відбувається цей процес у спільному робочому середовищі: Централізований доступ до ресурсів: система контролю версій забезпечує спільний доступ до репозиторія, що містить всі файли і код проекту. Це спрощує обмін інформацією між командами та відслідковує зміни протягом всього процесу розробки. Управління гілками: системи контролю версій дозволяють створювати гілки для розробки окремих функцій, виправлень або експериментів, які можуть відбуватися паралельно з роботою над основною гілкою на проекті. Такий підхід допомагає забезпечити прозорість роботи та підтримує налагоджену взаємодію між командами. Управління конфліктами: системи контролю версій автоматично

виявляють конфлікти при злитті різних версій коду або файлів, спрощуючи їх вирішення та дозволяючи командам зосередитись на пріоритетних виправленнях та удосконаленнях. Рівень видимості: системи контролю версій дозволяють командам та менеджерам стежити за прогресом протягом всього процесу розробки, слідкувати за вкладом кожного члена команди та оцінювати рівень продуктивності. Забезпечення автоматизованих повідомлень: системи контролю версій можуть надсилати відповідні повідомлення і сповіщення стосовно змін, внесених у проект, що дає можливість відповідно реагувати на нові оновлення або ризики.

– Відтворення результатів є ключовою складовою систем контролю версій, яка відіграє важливу роль у розробці програмних продуктів та презентації результатів в корпоративному середовищі. Даний аспект передбачає можливість відновити та аналізувати історію версій у процесі розвитку програми або проекту, включаючи виявлення відповідальності кожного учасника змін та основи для внесення цих змін. Розглянемо детальніше, яким чином цей процес допомагає покращити взаємодію між командами та забезпечити постійне поліпшення розробки: Зберігання історії змін: система контролю версій відстежує і зберігає всі зміни, внесені до репозиторію файлів і коду. Це дозволяє згодом ретельно проаналізувати ці зміни та зрозуміти розвиток програми або проекту з часом. Відновлення до попередніх версій: у разі виникнення проблем або помилок у проекті, системи контролю версій надають можливість повернутися до попередніх версій коду, що допомагає швидко виправити проблеми та забезпечити стабільність продукту. Прийняття рішень та контроль якості: аналіз історії версій дає можливість керівникам проектів та відповідальним особам оцінювати якість та внесок роботи розробників у процесі створення програми. Це сприяє виявленню слабких сторін, прийняттю керованих рішень та забезпеченню сталого рівня якості продукту.
– Гнучкість: можливість створення гілок це основний функціонал, який надається системами контролю версій, додаючи гнучкість до процесу

розробки програмного продукту. Гілка - це окрема копія коду, яку можна модифікувати, не впливаючи на основний код (так звану "майстер"-гілку). Отже, детальніше розглянемо, як створення гілок дає гнучкість у процесі розробки програмного продукту: Паралельна розробка: створення гілок допомагає командам розробляти нові функції, виправляти помилки або працювати над експериментальними покращеннями паралельно з основною гілкою. Це підвищує ефективність роботи, оскільки необхідно було б чекати, поки внесені зміни будуть включені в майстер-гілку, перш ніж приступити до наступного завдання. Мінімізація ризиків: створення гілок забезпечує безпеку основного коду від помилок і неякісних змін. У разі виникнення проблеми в окремій гілці, програма як цілість залишається незмінною, що гарантує стабільність розробки. Оптимізація співпраці: розвиток проекту в окремих гілках дає можливість командам зосередити свою роботу на конкретних задачах, що сприяє кращій організованості роботи. Зливання змін: після завершення роботи над окремою гілкою і переконаності в готовності її змін, відбувається зливання цієї гілки з головною. Цей процес включає перевірку можливих конфліктів та вирішення їх перед фінальним зливанням. Отже, гнучкість, що надається системами контролю версій через можливість створення гілок, дає розробникам можливість працювати над різними функціями або виправленнями одночасно, не ризикуючи порушенням стабільності основного коду.

– Автоматизація є одним з ключових аспектів при використанні систем контролю версій для розробки програмного продукту. Автоматизоване стеження за внесеними змінами допомагає проводити постійний контроль якості коду та виявляти проблеми на ранніх стадіях, збільшуючи продуктивність команд та забезпечуючи розвиток сталої та надійної програми. Ось кілька аспектів автоматизації у контексті систем контролю версій: Автоматичне відстеження змін: системи контролю версій здійснюють автоматичне відстеження змін, що внесені кожним учасником команди, з можливістю автоматичного збереження змін у репозиторій

проекту. Виявлення конфліктів: під час зливання гілок система автоматично виявляє конфлікти, які можуть виникнути між змінами, внесеними у двох паралельних гілках. Такий аналіз сприяє швидкому вирішенню проблем та недопущенню неякісних змін. Автоматична інтеграція: системи контролю версій можуть співпрацювати з автоматизованою системою збірки та інтеграції, такою як *Jenkins*, *Travis CI* або *GitLab CI/CD*, що дозволяє автоматично зібрати код та перевірити його працездатність після внесення змін у репозиторій. Тестування коду: автоматизація дозволяє інтегрувати системи контролю версій з інструментами автоматичного тестування, що прискорює процес пошуку та виправлення помилок, забезпечуючи високу якість коду у процесі розробки. Автоматичне розгортання: інтеграція систем контролю версій з інструментами автоматичного розгортання, такими як *Docker*, *Kubernetes* чи *Google Cloud Build*, сприяє прискоренню процесу розгортання продукту на різних стадіях розробки, від тестування до постачання. Коротко кажучи, автоматизація у системах контролю версій спрямована на полегшення діагностики проблем, підтримки постійного контролю якості коду та забезпечення ефективної роботи над програмним продуктом. Автоматичне відстеження змін, інтеграція з інструментами роботи, автоматичне тестування коду та розгортання - все це сприяє підтримці ефективного робочого процесу, стабільності програмного продукту та співпраці команди.

– Забезпечення безпеки даних: системи контролю версій володіють інструментами та можливостями, що допомагають підтримувати безпеку даних та проектів шляхом встановлення контролю за доступом, слідкування дій користувачів та аналізу змін в історії версій. Розглянемо детальніше кожний з цих механізмів: Обмеження доступу до репозиторіїв: системи контролю версій дозволяють налаштовувати рівні доступу до репозиторіїв для різних користувачів чи груп користувачів. Це забезпечує, що лише уповноважені користувачі можуть вносити зміни до коду, створювати гілки та зливати їх з основною гілкою. Слідкування за діями користувачів: системи

контролю версій ретельно фіксують усі зміни, внесені до проекту, і асоціюють їх з відповідними користувачами або коммітами. Це дозволяє аналізувати історію змін та слідкувати за діями у проекті, а також забезпечує відповідальність учасників. Аудит і виявлення ризиків: завдяки можливості відслідковувати зміни та аналізувати історію версій, системи контролю версій допомагають проводити аудит за внесеними змінами, розпізнавати ризикові дії та виявляти можливі проблеми ще до їх виникнення у продукті. Безпека версіонування: системи контролю версій створюють копії репозиторіїв та коду, що допомагає відновити проект, якщо станеться непередбачувані втрати даних або задачі. Це гарантує надійне збереження даних проектів та зменшує ризика витоку інформації. Шифрування та авторизація: багато систем контролю версій підтримують шифрування, яке допомагає гарантувати конфіденційність та цілісність даних, переданих через мережу. Також авторизаційні механізми забезпечують, що тільки автентифіковані користувачі можуть отримати доступ до репозиторіїв та змінювати їх. Враховуючи всі ці аспекти, системи контролю версій забезпечують надійні та ефективні механізми захисту, які гарантують безпеку даних та проектів, обмежують доступ до відповідних репозиторіїв і дозволяють аналізувати зміни в історії версій.

– Віддалена робота: У сучасному світі, коли все більше команд переходить до гнучкої або віддаленої роботи, системи контролю версій можуть допомогти значно спростити співпрацю між співробітниками, незалежно від їх розташування. Основні переваги віддаленої роботи з використанням систем контролю версій включають: Віддалений доступ: Кожен член команди може легко отримати доступ до коду та зберігати зміни, працюючи зі своєї локації, що робить співпрацю над проектами максимально гнучкою і комфортною. Синхронізація та співпраця: Використовуючи системи контролю версій, віддалені команди можуть легко синхронізувати свої зміни, співпрацювати над розв'язанням проблем та обмінюватися знаннями за допомогою чіткого відслідковування змін та комунікації. Асинхронна робота: Розробники

можуть працювати за своїм власним графіком, а системи контролю версій дозволяють співробітникам зручно вносити свої зміни, навіть якщо інші члени команди не працюють в той же час, забезпечуючи більшої гнучкості та продуктивності.

– Масштабованість: Одна з основних переваг систем контролю версій - їхня масштабованість. Це означає, що системи можуть автоматично адаптуватися до змін у команді, проекті чи навіть у компанії в цілому: Легкість включення нових учасників: Системи контролю версій дозволяють новим учасникам проекту швидко отримати доступ до всього існуючого коду та історії змін, що робить процес інтеграції нових розробників зручним і неперервним. Адаптація до розвитку компанії: Розробка продуктів та проектів може масштабуватися від невеликих команд до великих міжнародних корпорацій. Системи контролю версій надають необхідні можливості та зручності, щоб підтримувати стабільну роботу над проектами незалежно від розміру команди або компанії.
– Відповідальність та розуміння: Система контролю версій допомагає створювати культуру відповідальності та прозорості в команді, змушуючи кожного учасника розуміти, як внесені ними зміни впливають на проект в цілому, що сприяє створенню стабільного та якісного проекту.

Однак, поряд з перевагами, системи контролю версій можуть мати деякі недоліки, які необхідно враховувати.

– Технічні складності та рівень входження: Системи контролю версій можуть бути технічно складними для користувачів, які вперше стикаються з ними. Це може створити певні труднощі та затримки у процесі навчання та адаптації. Для опанування системою контролю версій можуть знадобитися такі кроки: Навчання: Запровадження процесу навчання та підтримка учасників команди допоможуть полегшити процес опанування нової системи. Матеріали для навчання мають бути доступні, щоб користувачі змогли швидко оволодіти основами роботи з системою. Технічна документація: Забезпечення доступу до відповідної технічної документації

та прикладів може сприяти швидшому опануванню системи. Дублювання знань та досвіду в команді допоможе надійно підтримувати систему контролю версій. Менторство: Залучення досвідчених користувачів систем контролю версій для менторства менш досвідченим членам команди може покращити процес адаптації та допомогти уникнути поширених помилок або підводних каменів під час використання системи.

– Час впровадження: Впровадження нової системи контролю версій може зайняти значний час та ресурси, особливо для вже існуючих проектів. Це може включати такі аспекти: Міграція даних: Процес міграції весь коду, конфігурації та історії змін з існуючої системи контролю версій до нової може бути трудомістким. Важливо скласти чіткий план міграції та виділити достатньо ресурсів для його виконання. Зміна робочих процесів: Впровадження нової системи контролю версій може вимагати зміни існуючих робочих процесів і методів взаємодії між членами команди. Це може включати налаштування системи, інтеграцію з іншими інструментами та адаптацію команди до нових робочих процесів. Оцінка вартості та вигоди: Впровадження нової системи може забезпечити переваги на довгостроковій перспективі, але потребує інвестицій у час та ресурси на початкових етапах. Необхідно ретельно оцінити вартість впровадження та можливі вигоди, щоб забезпечити успішне оновлення системи контролю версій.

Щоб усунути недоліки системи контролю версій, корпоративним платформам рекомендується виявити наступні можливості для покращення:

– Навчання і підтримка: Для полегшення оволодіння системою контролю версій та підвищення ефективності роботи команди рекомендується забезпечити доступ до навчальних матеріалів та провести тренінги. Це може охоплювати: Внутрішні семінари та воркшопи: Організація навчальних заходів з метою ознайомлення користувачів з базовими та розширеними функціями системи контролю версій. Відео або онлайн-курси: Забезпечення доступу до відеоматеріалів або онлайн-навчання для індивідуального чи групового навчання. Постійна підтримка: Відведення внутрішніх або

зовнішніх ресурсів для підтримки користувачів у разі виникнення питань чи проблем, пов'язаних з системою контролю версій.

– Інтеграція з автоматизованими тестами та іншими інструментами розробки: Інтеграція системи контролю версій з різними інструментами розробки може привести до оптимізації робочих процесів. Це може включати: Використання системи контролю версій як єдиної платформи для управління джерельним кодом, автоматизованими тестами та розгортанням. Автоматизація перевірки налаштувань та змін коду для забезпечення відповідності встановленим стандартам перед зливанням до основної гілки. Полегшення відстеження роботи над кодом завдяки автоматизації коментарів, анотацій та пов'язування змін з відповідними задачами або проблемами.
– Створення документації та встановлення найкращих практик: Організація, структурування та ведення бази знань, створення документації та методичних матеріалів до системи контролю версій важливе для успішної реалізації проектів на різних етапах: Забезпечення обширної та доступної документації: Створення прозорого, структурованого та повного опису роботи з системою контролю версій для команди. Встановлення найкращих практик: Використання найкращих практик для ефективної роботи з системою (наприклад, правила для назв комітів, управління гілками, процедури злиття тощо) (рис. 1.4).
– Вибір оптимальної системи контролю версій: При виборі найбільш підходящої системи контролю версій слід ретельно проаналізувати потреби компанії та специфіку розробки продуктів, а також враховувати ряд аспектів: Гнучкість: Висока гнучкість системи перестраховує рівень підтримки масштабування проектів, упровадження нових технічних рішень та процесів. Стабільність: Оцінка доступності підтримки, безпеки та стабільності роботи системи контролю версій. Сумісність з іншими інструментами розробки: розгляд можливості інтеграції обраної системи контролю версій з іншими інструментами та платформами, які використовуються в компанії.

Враховуючи всі згадані фактори та вимоги до системи контролю версій, можна забезпечити максимально ефективну, гнучку та стабільну роботу команди розробників на всіх етапах розробки та розгортання продуктів.

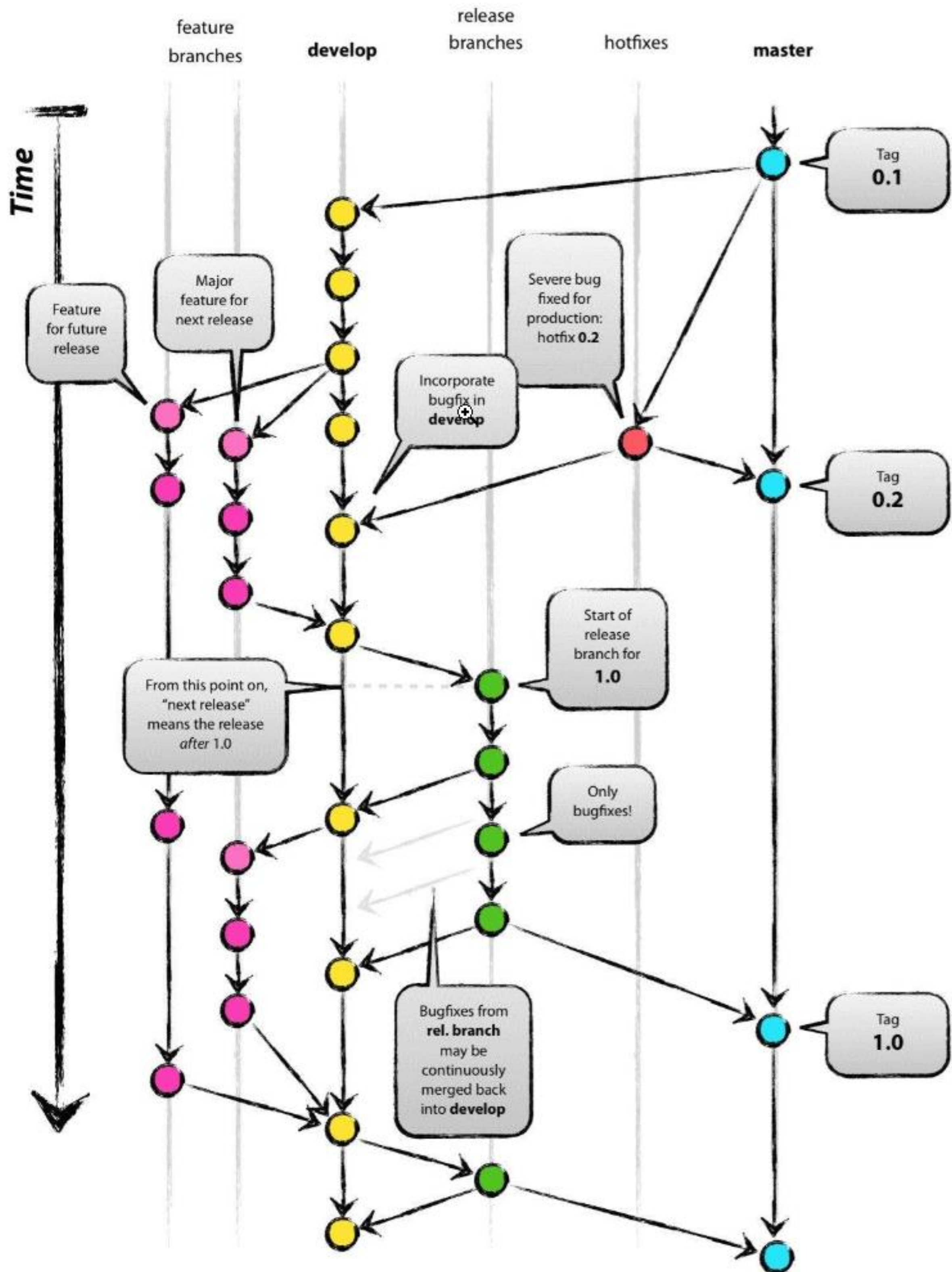


Рис. 1.4. Розгалуження *Git* і модифікований потік *Git*

1.3. Аналіз потреб до системи контролю версій

При впровадженні системи контролю версій в робочий процес команди, важливо забезпечити швидкість та зручність, адаптовані до проекту та специфіки праці команди. Особливість цього процесу лежить в ефективному опануванні системи розробниками, спрощенні навчання за рахунок структурованих настанов і рекомендацій, наявності інтуїтивного керування та адаптації до індивідуальних потреб користувачів.

Система контролю версій повинна також інтегруватись з існуючими інструментами розробки, включаючи середовища розробки, системи управління проектами та інструменти збірки та розгортання. Це дозволить команді використовувати один доступний інтерфейс для керування проектом, забезпечуючи швидке опанування нової системи. Зручність установки та налаштувань системи контролю версій також відіграє важливу роль, дозволяючи команді легко та швидко адаптуватися до нової системи розробки.

В оцінці швидкості та зручності впровадження системи контролю версій слід також опиратися на відгуки користувачів та рекомендації спільноти про їх досвід роботи із конкретною системою. Відгуки користувачів можуть виявитися корисними для ідентифікації складності та успіхів, з якими стикаються інші розробники під час адаптації до визначеної системи контролю версій.

У сукупності, при виборі та впровадженні системи контролю версій, важливо ретельно оцінити швидкість і зручність пропонованих рішень, щоб забезпечити плавний перехід команди до нової системи, надавши їм можливість продуктивно й ефективно працювати над проектами із програмного забезпечення.

Простота та доступність навчання є ключовими факторами успішного впровадження системи контролю версій. Розглядаючи цей аспект, необхідно враховувати представлення та доступність методичних матеріалів, що можуть бути різноманітними від навчальних посібників і відеоуроків до вебінарів і онлайн-курсів.

Такі матеріали потрібні для того, щоб розробники мали змогу швидко отримати необхідні знання і навички для ефективної роботи зі системою. Внутрішньо

організовані навчальні курси, вебінари або семінари, де розробники можуть задавати питання та обмінюватися досвідом, можуть значно надати розробникам екстра переваги.

Також життєво важливим є те, щоб навчальні матеріали були оновлені і актуальні. Технології швидко розвиваються, і матеріали, які застаріли уже за рік, можуть бути некоректними. Важливо переконатися, що система контролю версій, яку ви обираєте, має сильну спільноту, яка підтримує її і постійно оновлює і створює нові навчальні матеріали.

Крім того, методичні матеріали, такі як довідники, повинні бути легкодоступними безпосередньо з середовища системи контролю версій. Корисно мати можливість швидко звернутися до документації або пошукового механізму всередині системи, що значно полегшує процес засвоєння.

Отже, простота та доступність навчання в системі контролю версій визначають не тільки швидкість засвоєння нової системи розробниками, а й можуть впливати на продуктивність розробки в цілому.

Висока стабільність та надійність системи контролю версій є важливими аспектами успішного впровадження та використання на повсякденній основі. Команді розробників потрібна система, що забезпечує безперебійну роботу з кодом та документацією, мінімізуючи ризики від втрати даних, конфліктів версій та інших технічних проблем. Основні фактори, що впливають на стабільність та надійність системи контролю версій, включають відмінні механізми резервного копіювання та відновлення даних у разі системних збоїв, а також можливість швидкого відновлення коду та документації з резервних копій. Система контролю версій повинна також пропонувати гнучкі розгалуження та злиття коду, що надає змогу працювати над різними версіями коду одночасно та злити їх, вирішуючи будь-які конфлікти і відслідковуючи проблеми зі стабільністю. Забезпечення автоматичного тестування та виявлення помилок допомагає розробникам знайти та виправити код на ранніх стадіях та забезпечує виконання певної функціональності відповідно до очікувань.

Інтеграція з іншими інструментами та системами, такими як середовища розробки, системи управління проектами, віртуальні машини, системи автоматичної

збірки та розгортання, також має важливе значення для стабільності та надійності системи контролю версій. Налагоджені процеси взаємодії з зовнішніми системами позитивно позначаються на продуктивності та прозорості роботи команди. Враховуючи вищезгадані аспекти, стабільність та надійність системи контролю версій віддають належне успішному впровадженню і більш продуктивному використанню на практиці, сприяючи ефективному розвитку проектів навіть у найперспективніших та вимогливих умовах.

Система контролю версій служить суттєвою підтримкою командам, допомагаючи їм одночасно співпрацювати над проектами. Це включає забезпечення безперебійних процесів розробки, де кілька розробників можуть працювати над одним та тим же проектом без створення конфліктів або перекриття змін. Це досягається через розгалуження коду і врахування внесених змін, які потім можуть бути злиті назад в головну гілку коду, будуванням конфліктів і внесенням необхідних змін. Триваючи повсякденну роботу над проектами, система контролю версій також має відслідковувати дії користувачів і збирати звіти для аналізу дій команди. Ця інформація є незамінною для розуміння динаміки роботи команди, виявлення можливих проблемних точок і, в результаті, оптимізації робочих процесів та загального управління проектом.

Важливою складовою частиною кожної системи контролю версій є здібність обробляти і використовувати анотації та метадані. Вони дають можливість відслідковувати зміни, проводити більш комплексний аналіз коду та підтримувати структурування проектів (рис. 1.5). Анотації можуть допомогти відслідковувати, чому були зроблені певні зміни, хто їх виконав, і коли вони були виконані, тоді як метадані можуть допомогти у впорядкуванні та пошуку інформації.

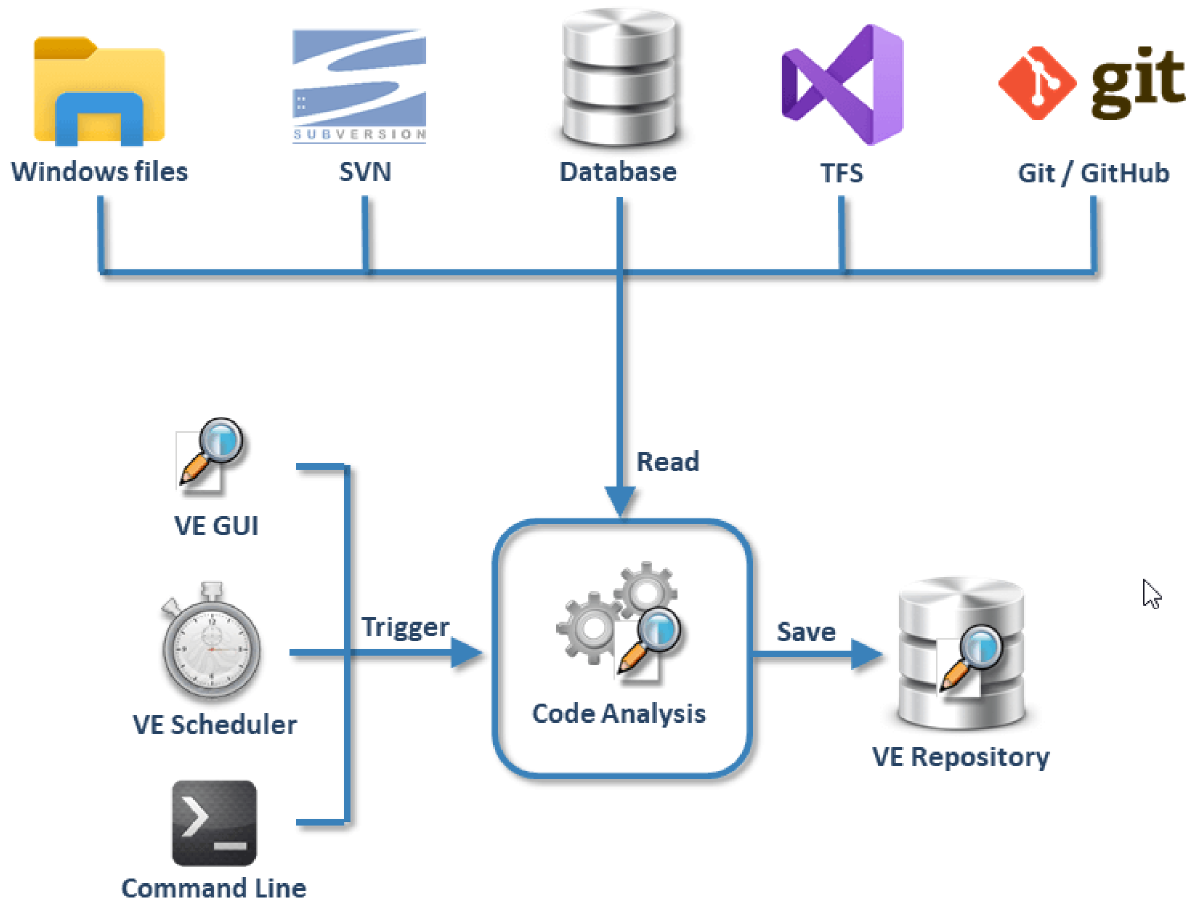


Рис. 1.5. Способи аналізу змін

1.4. Висновки до розділу

У даному аналізі я дослідив завдання, роль та важливість систем контролю версій в корпоративних платформах, розглянув переваги, недоліки і потреби таких систем для команди розробників. Системи контролю версій відіграють важливу роль у корпоративних платформах, дозволяючи розробникам стежити за змінами коду, успішно співпрацювати над проектами та забезпечувати стабільність та надійність виконання програмного забезпечення.

Впровадження системи контролю версій дозволяє підвищити ефективність розробки, уникнути конфліктів та забезпечити можливість відновлення коду в разі втрати даних. Системи контролю версій мають свої переваги та недоліки в корпоративних платформах. Серед переваг, можна виділити безперебійну роботу з кодом і документацією, покращення співпраці та ефективності у команді, зниження

ризику конфліктів або втрати даних. Однак, можуть виникати проблеми щодо впровадження та навчання користувачів, обмежена можливість інтеграції з іншими системами та потенційні питання щодо адаптації до індивідуальних вимог і потреб користувачів.

Аналіз потреб до системи контролю версій дозволив виявити ряд ключових вимог. Система контролю версій повинна бути швидкою, зручною та застосовною залежно від проекту та специфіки праці команди. Вона також повинна забезпечувати ефективне опанування системи, простоту та доступність навчання, а також інтуїтивне керування та адаптацію до індивідуальних потреб користувачів. Важливо також враховувати інтеграції з комп'ютерами, інструментами розробки та сервісами, надійність та стабільність впровадження системи, а також можливість одночасної роботи над проектами у команді.

На підставі проведеного аналізу можна зробити висновок про те, що вибір відповідної системи контролю версій є важливим аспектом, що впливає на продуктивність роботи команди та успішність проекту загалом. Врахування ключових факторів, як-от швидкість, зручність, інтуїтивність, надійність та інтеграція з існуючими інструментами, може сприяти впровадженню найбільш підходящої системи контролю версій та покращенню безперебійної роботи команди розробників.

РОЗДІЛ 2
ОЦІНКА ОСНОВНИХ ВИМОГ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

Оцінка основних вимог та визначення метрик системи контролю версій є важливим процесом, оскільки сприяє підвищенню ефективності роботи системи і її адаптивності до конкретних особливостей робочого процесу команди розробників. Враховуючи вимоги від функціональності, продуктивності, масштабованості та безпеки системи, можна забезпечити оптимальну роботу над різними проектами та відмінну співпрацю між учасниками команди розробників.

2.1. Аналіз функціональних вимог

Функціональні вимоги, які стосуються ключових особливостей системи контролю версій, забезпечують необхідні інструменти, щоб команда розробників могла ефективно працювати над проектами, справляючись з різними викликами робочого процесу. Розгляд конкретних аспектів функціональності допомагає розуміти, як система може відповідати ключовим потребам команди управління кодом.

Можливість зберігання, порівняння та відновлення версій файлів є основою успішної роботи з кодом. Вона допомагає розробникам відслідковувати зміни у програмному коді, контролювати стан коду на різних етапах проекту і вносити покращення за потреби. Це забезпечує стабільність роботи програмного продукту, роблячи її передбачуваною та віддає контрольні можливості розробникам.

Механізми співпраці, такі як мердж змін та вирішення конфліктів, дозволяють команді розробників працювати над кодом паралельно. Це забезпечує гнучкість у внесенні змін і сприяє швидкому вирішенню проблем і непередбачених ситуацій. Це створює природне середовище для співпраці, здатне прискорити вирішення загальних задач.

Врахування різних моделей гілок та репозиторіїв допомагає ефективно організувати програмний код. Використання відповідних систем люблених гілок

дозволяє розробникам прозоро розробляти проект, без непотрібного змішування стабільного та експериментального коду. Це полегшує планування нових функцій та сприяє успішному виконанні проектів різних розмірів та складності.

Таким чином, забезпечення належної функціональності системи контролю версій закладає фундамент успіху команди розробників: від слідкування за змінами до забезпечення гнучкості у внесенні змін та організації коду. Врахування функціональних вимог дозволяє створити стабільну базу для роботи команди та розвитку програмного продукту.

Функціональні вимоги до системи контролю версій визначають основні можливості та характеристики, необхідні для забезпечення ефективного управління кодом та співпраці команди розробників. Основні функціональні вимоги можуть включати:

Зберігання версій файлів є однією з основних функціональних вимог до системи контролю версій, оскільки вона лежить в основі управління кодом та ефективної співпраці команди розробників. Ця функція включає наступні аспекти:

– Створення версій файлів: Кожен раз, коли розробник внесе зміни в програмний код і зафіксує ці зміни (коміт), система контролю версій автоматично створює нову версію файлу. Це забезпечує детальний запис всіх змін, внесених у код, і дозволяє цьому коду бути після завершення більш структурованим та керованим.
– Зберігання файлів і метаданих: Система зберігає файли та їх версії разом з метаданими, які включають інформацію про автора, час створення, пов'язані коментарі та інші деталі, що дозволяють розробникам отримати повний контекст кожної зміни.
– Відслідковування змін: Система контролю версій дозволяє відслідковувати зміни між різними версіями файлів, що показано на прикладі системи контролю версій у *Visual Studio* на рис. 2.1, допомагаючи розробникам бачити, хто, коли і в який спосіб змінював код. Це поліпшує прозорість та історії коду, що є важливим фактором для забезпечення успішної співпраці між членами команди.

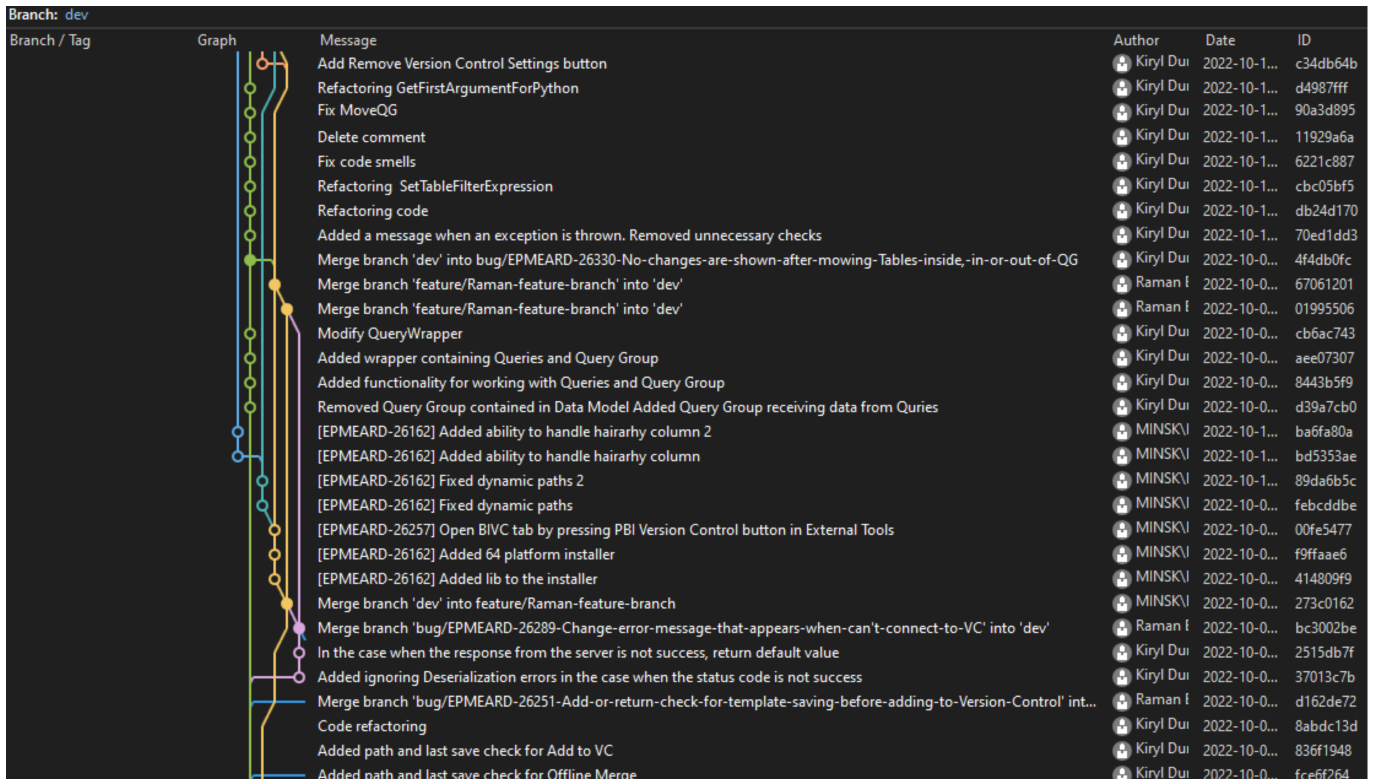


Рис. 2.1. Відслідковування змін у *Visual Studio*

– Управління історією: З можливістю зберігання версій файлів система слід пропонувати основні інструменти для управління історією змін, такі як приклади комітів, реверт редагувань або машина часу для "діагностики" коду.

– Сумісність із різними типами файлів: Оскільки програмне забезпечення може містити не тільки кодові файли, але й різні типи ресурсів (наприклад, документація, конфігураційні файли та мультимедіа), система контролю версій повинна бути сумісна з різними форматами файлів та забезпечувати зберігання версій файлів незалежно від їх типу.

Таким чином, можливість зберігати версії файлів є поряд денною потребою для успішної роботи команд. Таким чином, можливість зберігати версії файлів є поряд денною потребою для успішної роботи команди розробників, оскільки вона забезпечує надійну фундаментальну структуру для ефективного управління кодом, має гнучкість для різних робочих процесів та стимулює сумісну співпрацю розробників, оскільки вона забезпечує надійну фундаментальну структуру для

ефективного управління кодом, має гнучкість для різних робочих процесів та стимулює сумісну співпрацю.

Система контролю версій дозволяє відслідковувати зміни між різними версіями файлів, допомагаючи розробникам бачити, хто, коли і в який спосіб змінював код. Це поліпшує прозорість та історії коду, що є важливим фактором для забезпечення успішної співпраці між членами команди.

– Пошук версій файлів: Системи контролю версій надають можливість швидко знайти попередні версії файлів під час відкату коду. Це включає можливість шукати *version history*, фільтрувати по авторові, даті та змішаним критеріям.
– Оцінка варіантів відновлення: Розробники можуть переглядати різні версії файлів перед відновленням та порівнювати внесені зміни, щоб обрати потрібну версію файлу значуще відмінну від непотрібної.
– Відновлення файлів: Коли відбір файлів розглядається, системи контролю версій дозволяють відновити необхідну версію файлів, створити відокремлену гілку процесу або ж замінити код на стару версію в головній гілці.
– Налагодження проблем і достовірності: Коли певна версія файлів відновлена, розробники можуть налагоджувати проблеми або оцінювати надійність коду. Відновлення старої версії коду може допомогти виявити проблемні зміни та налагодити їх на поточному етапі розробки.
– Започатковані зміни: З метою відновлення втраченого точної версії файлів або виправлення подальших внесених помилок, команда розробників може в коміті контролю версій ініціювати започатковані зміни на поточному етапі розробки.
– Врахування інших викликів: Відновлення версій файлів також може бути використане для адресації випадкового вилучення даних іншими розробниками, зміни специфікацій або тестування різних версій залежностей.

Результатом можливості відновлення версій файлів стає підтримання відповідного стану коду, повертання до попередніх стадій коду при виявленні проблем або помилок, обмеження втраченої роботи та підвищення продуктивності команди розробників.

Порівняння версій файлів дозволяє розуміти зміни між різними версіями коду та перевіряти зміни перед їх впровадженням, що показано на інтерфейсі порівняння на рис.2.2.

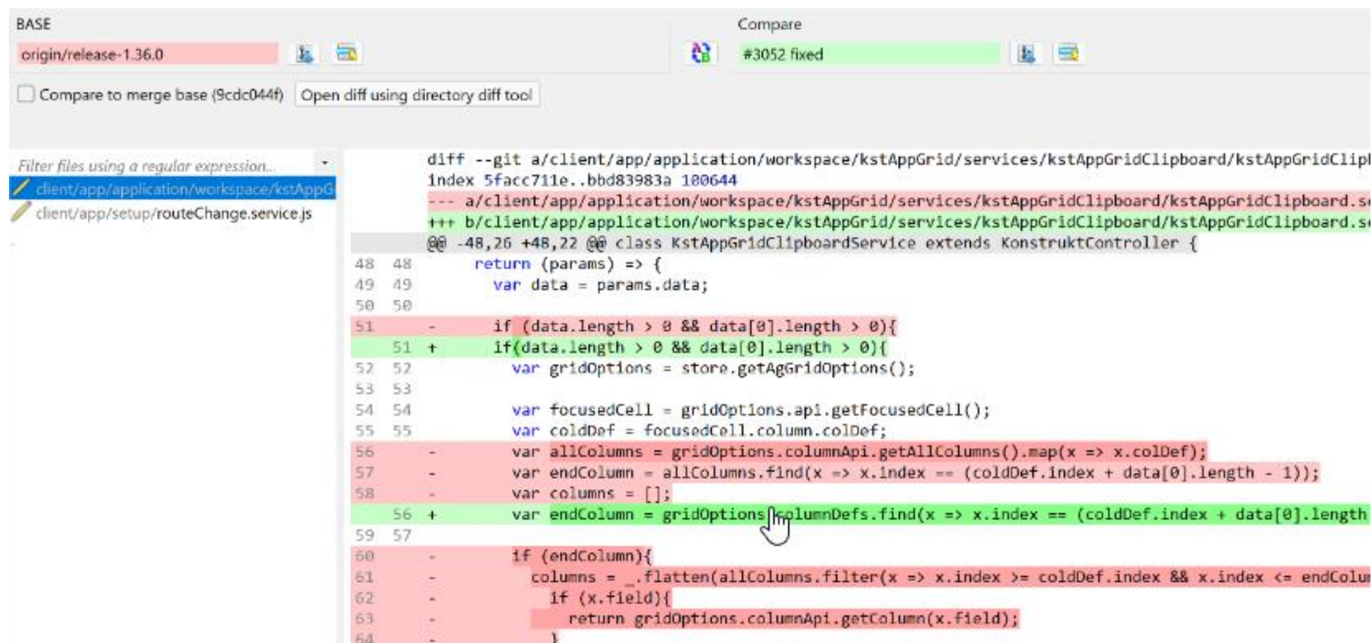


Рис. 2.2. Інтерфейс порівняння змін в *Git*

Використання інструментів порівняння версій коду може поліпшити якість коду, мінімізувати помилки та підвищити продуктивність команди. Детальніше розглянемо аспекти порівняння версій файлів:

– Обрання версій для порівняння: Системи контролю версій дозволяють користувачам обирати версії файлів для порівняння, наприклад, порівняти головну гілку з гілкою, що містить нову функцію, або порівняти дві попередні версії для аналізу змін.

– Візуалізація змін: Інструменти порівняння версій файлів представляють зміни візуально, зазвичай показуючи два файли поряд з підсвіченими рядками або блоками коду, які були додані, змінені або видалені. Це полегшує розуміння змін та швидке виявлення можливих конфліктів.

- Підтримка метаданих: Інструменти порівняння версій також можуть включати додаткові метадані, такі як коментарі до коміту, час зміни, авторські дані, що можуть допомогти у процесі розуміння змін і процесу прийняття рішень щодо впровадження.
- Інтерактивність: Інструменти порівняння версій можуть надавати можливість інтерактивного порівняння та виправлення змін, дозволяючи об'єднувати обидві версії коду та вирішувати конфлікти перед успадкуванням змін.
- Вивчення шаблонів: Досвідчені розробники можуть використовувати інструменти порівняння версій файлів для вивчення шаблонів у змінах коду, що може підвищити якість коду та зменшити кількість відкатів (*reverts*) або інших проблем.
- Відновлення помилок: У разі виявлення проблем під час процесу порівняння версій, розробники можуть швидко встановити, коли певна помилка була внесена, і відновити потрібну версію коду, щоб виправити проблему.

Забезпечення ефективних інструментів для порівняння версій файлів дозволяє команді розробників максимально забезпечити якість коду, розуміти зміни в коді та уникати помилок при успадкуванні змін. Такі інструменти є необхідними для успішної роботи з програмним кодом та забезпечення ефективної співпраці команди.

Гілки та моделі репозиторіїв: Можливість створювати гілки для окремих функцій, завдань або версій, дозволяючи команді розробників працювати над різними частинами коду одночасно, що продемонстровано на рис. 2.3.

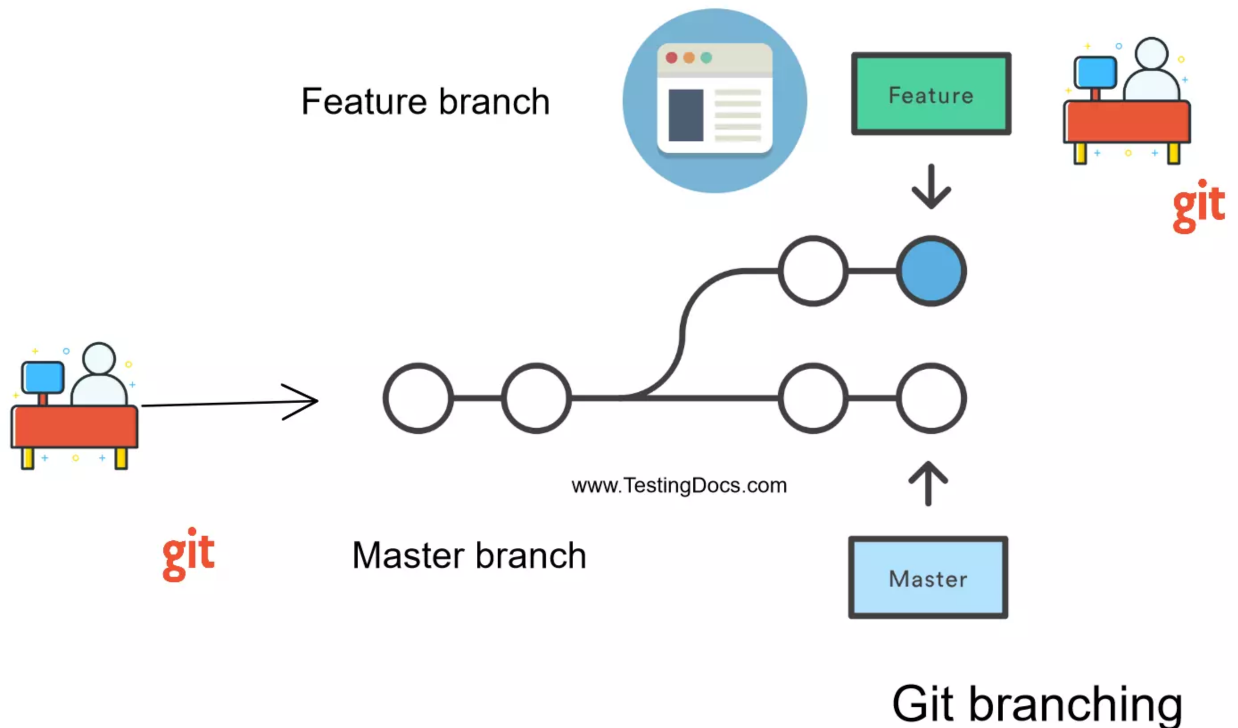


Рис. 2.3. Приклад створення гілок

Також слід мати можливість вибору між централізованою та децентралізованою моделлю репозиторіїв залежно від представлених вимог і структури команди.

- Створення гілок: Гілки допомагають розробникам працювати над різними частинами коду окремо, відриваючись від головної гілки. Вони особливо корисні коли зміни критичні та великі або потребують відстеження. У кінці часто створені гілки можуть бути злиті назад з головною гілкою.
- Робота з гілками: Гілки допомагають розробникам працювати над окремими завданнями, функціями або релізами без виникаючих конфліктів між колегами. Може бути декілька гілок, які можуть працювати паралельно та поєднатися за відсутності помилок.
- Вибір між централізованою та децентралізованою моделлю: Залежно від розміру команди, структури, переваг і потреб команди, можна вибрати між централізованою та децентралізованою моделлю репозиторіїв. Централізована модель (наприклад, *SVN*): У цьому випадку існує один центральний сервер, на якому зберігається повна історія версій, і усі

розробники мають доступ до нього. Цей підхід може бути корисним для невеликих команд, але може стати бар'єром для більших команд, оскільки у ході роботи можуть виникнути проблеми під час успадковування змін. Децентралізована модель (наприклад, *Git*): В даному типі репозиторія кожен розробник має власну своєрідну копію коду і історії версій, при цьому також мають можливість обмінюватися змінами між собою. Підхід допомагає більш швидко підготувати зміни та знову завантажити їх одночасно з роботою над різними фрагментами, сприяючи ефективності та взаємодії розробників.

– Зливання гілок: Процес зливання дозволяє комбінувати різні версії коду в одну послідовність, інкорпоруйте найкращі аспекти кожної версії. На даний момент треба проявити увагу до можливих конфліктів та їх вирішення.
– Робота з моделлю репозиторія: Вибираючи підхід до моделі репозиторіїв, важливо розглянути різному рівню співпраці, поданому досвідом розробників, розміром, структурою та технічними можливостями команди.

Забезпечення можливості створення гілок та вибору між централізованою та децентралізованою моделлю репозиторіїв дозволяє команді розробників працювати гнучкіше та продуктивніше, адаптуючи систему контролю версій до своїх індивідуальних потреб.

Мердж та вирішення конфліктів є важливими процесами в системі контролю версій, оскільки це дозволяє команді розробників успішно об'єднувати зміни та вирішувати різні конфлікти, своєчасно реєструючи прогрес у роботі. Основні аспекти мерджів та вирішення конфліктів включають:

– Контроль гілок: Перед мерджем потрібно впевнитися, що ви контролюєте стан гілок та правильно обираєте гілки для об'єднання коду. Це забезпечує, що об’єднання відбуватиметься правильно, максимально зменшуючи конфлікти.
– Автоматичне об'єднання: Більшість систем контролю версій намагаються спростити процес мерджу, надаючи автоматичні страховки, якщо зливання може відбутися без конфліктів. Це забезпечує швидкі та чіткі зливання коду.

– Розпізнавання конфліктів: Якщо система виявить конфлікти між змінами у різних гілках, вона повідомить розробників про це, допомагаючи їм швидко зосередитися на проблемних ділянках коду.
– Вирішення конфліктів: Розробники мають виважено та систематично вирішувати конфлікти, аналізуючи різні версії коду та вибираючи найкращі. Це може включати вручну внесення змін, застосування стратегій об'єднання частин або заміни коду.
– Перевірка змін: Після вирішення конфліктів та проведення мерджу, розробникам слід перевірити об'єднаний код та переконатися, що всі зміни працюють корректно, а код відповідає очікуванням.
– Збереження результатів: Коли конфлікти вирішені, а мердж успішно завершено, результати об'єднання коду слід зберегти в системі контролю версій, з наданням відповідної інформації про мердж (коментарі, метадані тощо).
– Підтримка командної роботи: Системи контролю версій зазвичай надають можливості співпраці при вирішенні конфліктів мерджу, такі як можливість додавати коментарі або спільно редагувати код.

Забезпечення ефективних інструментів мерджу та вирішення конфліктів в системі контролю версій сприяє гладкому та продуктивному процесу розробки, мінімізації проблем і віддаленій роботі команди розробників.

Інтеграції з іншими інструментами (рис. 2.4.): Інтеграція системи контролю версій з іншими інструментами, такими як *IDE*, системами збірки та розгортання, системами завдань та сповіщень тощо, може сприяти підвищенню ефективності командного робочого процесу.

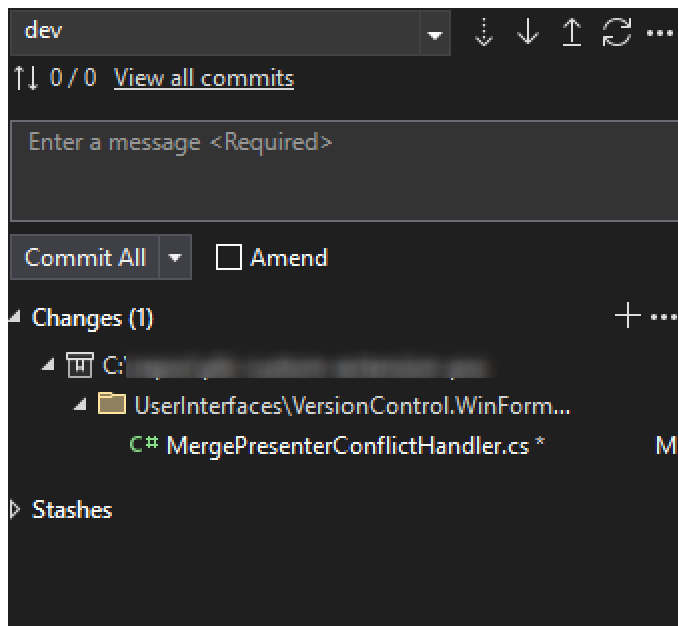


Рис. 2.4. *Git* інтегровано у *Visual Studio*

– Інтеграція з середовищем розробки: Інтеграція контролю версій з *IDE*, такими як *Visual Studio*, *Eclipse* або *IntelliJ IDEA*, забезпечує розробникам можливість з комфортом працювати з кодом, здійснювати коміти, стежити за змінами та взаємодіяти з іншими розробниками без прямого переходу до системи контролю версій.
– Інтеграція з системами збірки та розгортання: При інтеграції з інструментами збірки та розгортання, такими як *Jenkins*, *TeamCity* або *Travis CI*, команда розробників може автоматично будувати та перевіряти код після кожного коміта або мерджа, що забезпечує відмінну якість коду та поліпшення процесу розгортання.
– Інтеграція з системами завдань та сповіщень: Інтеграція контролю версій з системами завдань та сповіщень, наприклад *JIRA*, *Trello* або *Asana*, допомагає команді розробників ефективно прослідковувати і відстежувати зміни, створювати і назначати завдання, відображати проект розробки і отримувати сповіщення про актуальні зміни в коді.
– Інтеграція з системами повідомлень та спілкування: Інтеграція з інструментами спілкування, такими як *Slack*, *Microsoft Teams* або *Discord*, дозволяє команді розробників спілкуватися, обмірковувати зміни,

розповідати про стан проекту, отримувати сповіщення про оновлення репозиторію та покращувати взаємодію всередині проекту.

- Інтеграція з системами рев'ю коду та співпраці: Інтеграція з інструментами рев'ю коду, як *GitHub*, *BitBucket* або *GitLab*, дозволяє проводити оцінку коду, співпрацю, присвоювання завдань та відстеження проекту через спільну систему.
- Інтеграції з хмарними сервісами та сховищами: Системи контролю версій також можуть інтегруватися з хмарними сервісами та сховищами, як *GitHub*, *Google Cloud* та *Amazon AWS*. Це полегшує доступ до репозиторіїв, автоматизацію збірки та розгортання, резервне копіювання даних та безперервну інтеграцію.

Таким чином, інтеграція системи контролю версій з різними інструментами та сервісами забезпечує гнучкість, сприяє оптимізації робочого процесу, поліпшення співпраці між членами команди та підвищення ефективності розробки програмного забезпечення.

Керування доступом та автентифікація користувачів є критично важливими властивостями систем контролю версій, оскільки це допомагає забезпечити безпеку коду і даних, а також дозволяє ефективно розподіляти ресурси та обов'язки в команді розробників. Давайте детальніше розглянемо ці аспекти:

- Автентифікація користувачів: Автентифікація є першим кроком для керування доступом. Вона вимагає від користувачів перевірки своєї особи, найчастіше за допомогою імені користувача та пароля, або інших методів, таких як відбитки пальців, аутентифікація за допомогою смс, або апаратні токени. Це допомагає забезпечити, що тільки авторизовані користувачі мають доступ до репозиторію.
- Ролями засноване керування доступом (*RBAC*): В контексті систем контролю версій, *RBAC* зазвичай включає в себе визначення ролей, таких як "адміністратор", "розробник" або "гість", і надання цим ролям дозволів на виконання певних дій, таких як зміна, видалення або створення гілок.

- Управління доступом на рівні репозиторію: Деякі системи контролю версій надають можливість обмеження доступу до певних частин репозиторію або до самих репозиторіїв, як це реалізовано, наприклад, на *GitHub*. Це може бути корисним для захисту конфіденційних даних або роботи з великими командами.
- Аудит та логування: Логування дій користувачів та аудит систем контролю версій є важливими для відслідковування змін, виявлення несанкціонованих дій та виявлення та вирішення проблем.
- Керування *SSH* ключами: Можливість керування *SSH* ключами є важливим аспектом безпеки у більшості систем контролю версій. *SSH* ключі забезпечують безпечне з'єднання між робочою станцією розробника та репозиторієм і використовуються для автентифікації користувачів.
- Двофакторна автентифікація: Для додаткової безпеки, деякі системи контролю версій мають опцію двофакторної автентифікації, що вимагає двох форм перевірки особи користувача перед наданням доступу до репозиторію.

Обраний підхід до керування доступом та автентифікації користувачів значною мірою залежить від розміру команди, набору технічних компетенцій та вимог безпеки. Ці інструменти та процеси допомагають командам забезпечити захист репозиторію від несанкціонованого доступу, контролювати активність користувачів та ефективно управляти обсягом роботи.

Інструменти для співпраці та комунікації в системах контролю версій відіграють важливу роль, бо вони підвищують рівень взаєморозуміння між членами команди, сприяють в застосуванні гарних практик розробки, підтримують продуктивність та якість розробки. Ось декілька ключових характеристик:

- Обмін коментарями: Інструменти обміну коментарями дозволяють розробникам обговорювати зміни (рис. 2.5), які вони роблять у коді, вказувати помилки та пропонувати вдосконалення. Вони можуть включати прямі коментарі до рядків коду, загальні коментарі до комітів або мерджів, або навіть листівки для глобальних обговорень.

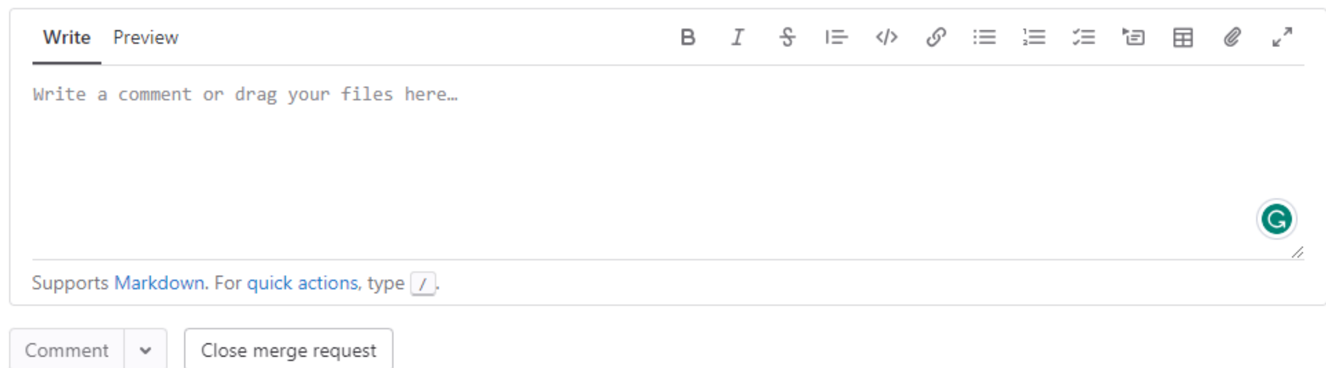


Рис. 2.5. Секція коментування в *Git*

– Розбір коду: Розбір коду, або рев'ю коду, - це процес, за допомогою якого інші члени команди перевіряють код на наявність помилок. Інструменти рев'ю коду мають бути легкими в користуванні та мати можливість легко інтегруватися з основною системою контролю версій.
– Прямий обмін інформацією: Інструменти для прямого обміну інформацією, такі як чати, форуми або системи повідомлень, можуть зробити комунікацію більш відкритою та ефективною. Вони забезпечують швидкий доступ до дискусій, забезпечують напрям для вирішення проблем і підтримують загальну організованість команди.
– Документація: Документація допомагає розробникам розуміти код, відслідковувати зміни та пізнавати найкращі практики. Вона може включати кодові коментарі, описи процесів, технічні специфікації, схеми архітектури та інше. Документація може бути доступна прямо в системі контролю версій, або як окремий ресурс (такий як вікі-сторінки або веб-сайт).
– Хмарні платформи співпраці: Додатково до вищенаведеного, команди, що працюють віддалено, можуть отримати перевагу від використання хмарних платформ для співпраці, таких як *Google Docs* або *Microsoft 365*, що дозволяє їм працювати над спільними документами, таблицями або презентаціями.

Організована та ефективна співпраця та комунікація є ключовими факторами успіху в будь-якому проекті розробки програмного забезпечення. Система контролю версій, яка надає високоякісні інструменти співпраці та комунікації, може сприяти підвищенню ефективності команди а свою чергу імплементацію проекту.

Процес реплікації та резервного копіювання даних виявляється невід'ємним елементом сучасних стратегій управління інформацією, який впливає на стабільність та безпеку функціонування інформаційних систем. Необхідність вирішення проблеми втрати даних через технічні збої та відмову обладнання визначає важливість впровадження ефективних заходів з реплікації та резервного копіювання. Реплікація, забезпечуючи створення дублікатів даних на різних серверах чи системах зберігання, не лише усуває ризик втрати інформації у разі збою апаратного забезпечення, але й гарантує безперервний доступ до даних навіть у випадку відмови конкретного компонента. Це робить інформаційні системи більш стійкими до різноманітних негараздів, забезпечуючи неперервну доступність для користувачів.

Резервне копіювання, з свого боку, є ключовим елементом стратегії відновлення даних після серйозних випадків втрати, таких як природні катастрофи, хакерські атаки або інші великомасштабні події. Здатність оперативно відновлювати інформацію з резервних копій забезпечує не тільки відновлення даних, але й відновлення діяльності організації без значних перерв чи збитків. Цей комплексний підхід до управління даними надає додаткову впевненість користувачам та проектним командам. Вони можуть мати впевненість, що їхні дані захищені від втрати та доступні в будь-який момент, навіть у екстремальних умовах. Це сприяє не лише забезпеченню надійності інформаційних систем, а й робить їх більш конкурентоспроможними та готовими до вирішення викликів у сучасному світі технологій.

Масштабованість систем контролю версій виявляється ключовим аспектом, який впливає на їхню ефективність та привабливість для розробників, незалежно від розміру проекту чи обсягу команди. Отримані результати підтверджують, що можливість безперервної роботи та стабільної функціональності систем контролю версій важлива не лише для великих проектів, але і для менших команд та навіть індивідуальних розробників. Здатність системи ефективно масштабуватися враховує не тільки обсяг репозиторіїв, але й кількість користувачів, які залучені до процесу розробки. Це важливо для забезпечення плавної та ефективної роботи, адже збільшення кількості учасників проекту може викликати значний навантаження на

систему контролю версій. Такий підхід гарантує, що інфраструктура залишається стійкою та швидко реагує на зростання вимог до неї.

Зазначене висловлює необхідність пильно враховувати питання масштабованості під час вибору та налаштування системи контролю версій. Важливо не лише підтримувати її поточний обсяг робіт, але й передбачати можливі майбутні зростання, щоб уникнути технічних обмежень та забезпечити безперебійну роботу розробки в будь-який момент. Забезпечуючи масштабованість, система контролю версій стає надійною та привабливою для широкого спектру користувачів, сприяючи ефективному управлінню версіями коду та успішному веденню розробок у будь-якому масштабі. Високий рівень автентифікації та керування доступом виявляється фундаментальним елементом для досягнення повного контролю над розробкою та забезпечення безпеки даних. Забезпечуючи ефективний механізм перевірки особистості та докладний контроль над рівнем доступу, ці заходи стають гарантією надійності та конфіденційності в процесі розробки.

Автентифікація гарантує, що лише визначені розробники мають доступ до системи та проектів, утримуючи несанкціонованих користувачів від потенційно небезпечних зон. Керуванням доступом встановлюються чіткі правила, що регулюють, які ресурси доступні кожному користувачеві, забезпечуючи тим самим не лише захист від несанкціонованого доступу, але й виключаючи випадкові помилки чи втрату даних. Ці заходи забезпечують розробникам необхідний ступінь свободи та впевненості в тому, що їхні проекти та робота захищені від потенційних загроз. З високим рівнем автентифікації та керування доступом розробники можуть фокусуватися на творчості та вдосконаленні коду, знаючи, що їхні усі зусилля підтримуються ефективними заходами забезпечення безпеки. Такий підхід допомагає створити надійне та стійке середовище для розробки, що є важливим чинником в сучасному цифровому світі.

Інтеграція з іншими інструментами виявляється невід'ємним елементом сучасного процесу розробки, спрямованим на максимізацію продуктивності та якості роботи команд розробників. Забезпечуючи злагоджену взаємодію між різними інструментами, цей підхід дозволяє розробникам використовувати всі переваги своїх

улюблених інструментів, створюючи таким чином ефективне та ергономічне робоче середовище. Інтеграція з інтегрованими середовищами розробки дозволяє невідчутно переходити між фазами розробки та використовувати усі можливості обраного інструменту, що підвищує ефективність та зручність для розробників. Взаємодія з системами збірки та розгортання сприяє автоматизації процесів тестування , забезпечуючи плавний і швидкий цикл розробки. Інтеграція з системами управління завданнями сприяє відслідковуванню прогресу та організації завдань, підвищуючи координацію в команді. Крім того, взаємодія з інструментами спілкування забезпечує зручний обмін інформацією та сприяє взаєморозумінню у команді розробників, що важливо для успішної співпраці. Цей комплексний підхід до інтеграції підвищує ефективність розробки, роблячи процес більш гнучким, швидким та спрямованим на досягнення високоякісних результатів.

Таким чином, інтеграція з іншими інструментами стає критичним елементом сучасного розробницького середовища, сприяючи розвитку індивідуального потенціалу кожного розробника та підвищенню колективної продуктивності. Це не просто технічний аспект, але і стратегічна можливість для команд вдосконалювати свою роботу, пристосовуватися до змін та досягати відмінних результатів у світі постійної інновації та розвитку. Автоматизація, охоплюючи процеси автоматичної збірки, тестування та розгортання, виявляється невід'ємною складовою для досягнення високої ефективності та високої якості продукту в розробницькому середовищі. Отримані результати підтверджують, що автоматизація сприяє покращенню робочого процесу та забезпечує численні переваги для команд розробників.

Автоматична збірка дозволяє виконувати процес компіляції та будування програмного продукту без втручання розробників, забезпечуючи їм можливість фокусуватися на самому процесі розробки. Автоматизоване тестування забезпечує ретельну перевірку функціоналу та допомагає виявляти та виправляти помилки на ранніх етапах розробки, що призводить до вищої якості програмного продукту. Розгортання автоматизованої системи забезпечує швидке та надійне розгортання нових версій продукту, знижуючи час випуску та ризик людських помилок. Такий

підхід не лише полегшує життя розробників, але і сприяє створенню більш пристосованого, ефективного та надійного робочого середовища. Заощаджений час та зниження ймовірності помилок при автоматизації виливаються в підвищену ефективність розробки. Автоматизація визнається ключовим елементом сучасного розробницького процесу, що допомагає досягти якісних та конкурентоспроможних результатів у інформаційному середовищі.

У підсумку, можливості співпраці та комунікації в межах системи контролю версій виявляються необхідним стимулом для активної участі всіх учасників у процесі розробки. Забезпечуючи зручний механізм обговорення та обміну ідеями, ці можливості дозволяють розробникам ефективно співпрацювати, надаючи можливість детально обговорювати зміни, ділитися враженнями і вчиться на власних помилках до того, як вони можуть стати суттєвими труднощами.

Крім того, сприяючи розподіленому характеру роботи, системи контролю версій створюють віртуальне робоче оточення, що сприяє взаємному впливу та взаєморозумінню в команді. Це робить розробку більш гнучкою та реактивною до змін, а також сприяє високій якості та швидкості розробки. Отже, системи контролю версій виступають не лише як інструмент для відстеження змін у коді, але і як платформа для створення сприятливого середовища для співпраці та комунікації. Цей підхід підкреслює важливість колективної роботи та взаємодії у сучасному розробницькому процесі, сприяючи високому рівню продуктивності та якості вироблених рішень. Функціональні вимоги визначають основні функції та операції, які має виконувати система чи програма. Вони визначають, як користувачі мають взаємодіяти з системою та які конкретні результати чи стани повинні бути досягнуті при використанні. Функціональні вимоги визначають область відповідальності системи, чітко формалізуючи очікувані можливості та обмеження. Це сприяє розумінню завдань, які система повинна вирішувати, і створює основу для подальшого проектування та розробки. Функціональні вимоги виступають як орієнтир для команди розробників, забезпечуючи рамки для створення продукту, який відповідає вимогам та очікуванням користувачів.

2.2. Аналіз нефункціональних вимог

У контексті системи контролю версій, аналіз нефункціональних вимог відіграє розшаровану роль у визначенні параметрів, які впливають на її загальну ефективність та надійність. Нефункціональні вимоги ставлять завдання, що не обмежуються лише функціональністю системи, але визначають її поведінку в різних умовах.

Визначення продуктивності є критичним елементом для ефективності та прийнятної користувацької досвіду в системах контролю версій. У цьому контексті, продуктивність визначається як здатність системи забезпечувати оперативну та швидку реакцію на запити користувачів, особливо при обробці великого обсягу даних та виконанні операцій з контролю версій. Із зростанням обсягу розроблюваного коду, кількості користувачів та розмірів проектів, продуктивність системи контролю версій стає ключовою вимогою. Це включає в себе аналіз часу відгуку системи, тобто часу, необхідного для виконання певної операції або відповіді на запит користувача. Додатково, важливо визначити, як система пристосовується до обсягу операцій та змін в розроблюваному коді. Це може включати аналіз швидкості виконання операцій, таких як коміти, злиття гілок, отримання або відправлення змін, та інших рутинних завдань. Під продуктивністю також розуміється швидкість доступу до інформації. Система повинна надавати ефективний та швидкий доступ до історії змін, відомостей про авторів, або інших атрибутів коду. Це важливо для швидкого виявлення та виправлення помилок, а також для ефективного спілкування між учасниками проекту. Враховуючи ці аспекти, аналіз продуктивності дозволяє визначити та оптимізувати ті елементи системи, які визначають її швидкодію та відповідність високим стандартам користувацького досвіду. Підходи до оптимізації можуть включати в себе кешування результатів, оптимізацію запитів до бази даних, розпаралелювання операцій та використання передових алгоритмів оптимізації коду.

В контексті системи контролю версій, забезпечення безпеки та конфіденційності є критичним завданням, оскільки вона має обробляти та зберігати важливі дані, пов'язані з розробкою проектів. Аналіз нефункціональних вимог у

цьому контексті орієнтується на визначення ключових елементів безпеки та заходів для забезпечення конфіденційності та інтеграцію даних.

Система аутентифікації та авторизації: Важливо визначити ефективну систему аутентифікації, яка гарантує, що лише вповноважені користувачі можуть отримати доступ до системи. Авторизація повинна бути гнучкою та базуватися на ролях, щоб обмежити доступ до конфіденційної інформації тільки для тих, кому це дозволено.

– Двофакторна аутентифікація: Використання не тільки пароля, але й додаткового елементу (наприклад, коду, отриманого на мобільний телефон) для підтвердження ідентифікації користувача.
– Біометрична ідентифікація: Впровадження можливості використання біометричних даних (відбитки пальців, розпізнавання обличчя) для аутентифікації, що зростає рівень безпеки.
– Визначення різних рівнів доступу: Розроблення системи, що надає різні рівні доступу в залежності від ролі користувача. Наприклад, адміністратор має повний доступ, тоді як розробники можуть мати обмежений доступ до певних функцій.
– Динамічна зміна ролей: Можливість динамічно змінювати ролі користувачів в залежності від їх функціональних обов'язків або проектів.
– Моніторинг активності користувачів: Система повинна реалізувати механізми моніторингу та реєстрації активності користувачів для виявлення підозрілих або несправедливих дій.
– Блокування облікових записів: Автоматичне блокування облікових записів після зазначеної кількості неуспішних спроб входу для уникнення потенційних атак.
– Хешування паролів: Застосування алгоритмів хешування для збереження паролів в захешованому вигляді, уникнення зберігання чіткого тексту паролів в базі даних.
– Вимоги до паролів: Встановлення вимог щодо складності паролів (довжина, використання букв, цифр та символів).

– Автоматичне виходження після періоду не активності: Визначення часового обмеження не активності користувача, після якого відбувається автоматичний вихід з системи для уникнення несанкціонованого доступу у випадку залишеного відкритого сеансу.
– Використання токенів: Застосування технік, таких як використання токенів для підтвердження авторизації та уникнення атак типу "перехоплення сесії".

Вивчення та розробка таких механізмів аутентифікації та авторизації допомагає створити систему контролю версій, яка ефективно обмежує доступ до конфіденційної інформації лише для уповноважених користувачів, забезпечуючи високий рівень без

Застосування механізмів шифрування для захисту даних під час передачі та зберігання. Шифрування допомагає уникнути можливого перехоплення або несанкціонованого доступу до конфіденційної інформації.

– Симетричне шифрування: Використання одного ключа для як шифрування, так і розшифрування даних. Це забезпечує швидке шифрування, але вимагає безпечної передачі ключа.
– Асиметричне шифрування: Використання пари ключів - публічного та приватного. Публічний ключ використовується для шифрування, а приватний - для розшифрування. Це забезпечує безпеку передачі даних, оскільки публічний ключ може бути вільно розголошений.
– Протоколи *HTTPS/SSL/TLS*: Застосування протоколів шифрування для захисту даних під час їх передачі через мережу, забезпечуючи конфіденційність та цілісність даних.
– Захист від атак *MITM*: Використання механізмів, що унеможливлюють атаки типу "перехоплення посередника", такі як перевірка сертифікатів та обов'язкове використання шифрування.
– Шифрування на рівні бази даних: Використання функціоналу шифрування, який надається самою базою даних для забезпечення конфіденційності даних, що зберігаються.

– Файлове шифрування: Захист конфіденційних файлів шляхом їх шифрування перед зберіганням, з використанням ключа, що генерується із використанням інформації облікового запису користувача.
– Захищене зберігання ключів: Використання безпечних засобів для зберігання та управління ключами шифрування, щоб уникнути можливого їхнього витоку.
– Регенерація ключів: Регулярна зміна ключів для запобігання атакам та збереження високого рівня безпеки.
– Журналювання подій шифрування: Введення системи аудиту та моніторингу, щоб виявляти можливі атаки або невдалий доступ до ключів шифрування.
– Виявлення змін шифрування: Механізми для виявлення неправомірних змін налаштувань шифрування, що можуть вказувати на потенційні порушення безпеки.

Шифрування даних в системі контролю версій грає важливу роль у забезпеченні конфіденційності та недоступності для несанкціонованих осіб, зменшуючи ризики перехоплення та зберігаючи цілісність важливої інформації.

Система повинна мати точний контроль доступу, щоб керувати тим, які користувачі мають доступ до різних частин системи та які дозволи вони мають. Це включає в себе визначення рівнів доступу до різних гілок коду, історії змін та інших ресурсів.

– Адміністраторський рівень: Повний доступ до всіх функцій та ресурсів системи, включаючи право керування користувачами та встановлення політик безпеки.
– Розробницький рівень: Доступ до інструментів для роботи з гілками коду, внесення змін та відслідковування історії змін у відповідності до проектних завдань.
– Читацький рівень: Обмежений доступ для перегляду вмісту, але без можливості внесення змін. Призначений для сторонніх учасників, які повинні ознайомлюватися з кодом.

– Визначення прав доступу на рівні гілок: Користувачі можуть мати різні права для різних гілок коду. Наприклад, для основної гілки можуть встановлюватися обмеження на запис, а для гілок розробки - більше свободи.
– Створення захищених гілок: Можливість обмежити доступ до певних гілок лише конкретним користувачам або групам.
– Доступ до конкретних версій: Визначення, які користувачі можуть переглядати та взаємодіяти з певними версіями коду.
– Локальний та віддалений доступ: Обмеження доступу до історії змін залежно від того, чи користувач працює локально або віддалено.
– Доступ до багтрекера: Регулювання прав доступу до інструментів для відслідковування та управління багами та задачами.
– Керування доступом до документації: Забезпечення можливості обмеженого доступу до документації та ресурсів, які супроводжують проект.
– Визначення ролей користувачів: Надання різних ролей користувачам в системі, таких як адміністратор, розробник, *QA*-інженер і т.д.
– Присвоєння ролей на основі проектів: Визначення, які користувачі мають доступ до певних ресурсів, в залежності від їх ролі в конкретному проекті.
– Журнал подій доступу: Ведення журналу подій для реєстрації всіх дій користувачів, які взаємодіють з системою контролю версій.
– Виявлення неправомірного доступу: Встановлення механізмів для виявлення та повідомлення про неправомірний доступ або спроби вторгнення.

Контроль доступу в системі контролю версій дозволяє точно налаштовувати рівні доступу для різних користувачів та груп, забезпечуючи ефективне управління інформаційними ресурсами та збереження конфіденційності проекту.

Введення системи моніторингу та аудиту для виявлення потенційних загроз безпеці, а також відстеження дій користувачів. Це дозволяє оперативно реагувати на можливі атаки або несправедливі дії, а також забезпечує можливість розслідування подій.

– Запис основних подій: Включення в журнал подій усіх дій користувачів, зокрема входу в систему, внесення змін, видалення файлів та інші ключові операції.
– Ідентифікація аномальних подій: Встановлення механізмів для виявлення аномальних або підозрілих дій, що можуть свідчити про можливі загрози безпеці.
– Автоматизований аналіз даних: Використання інструментів для автоматичного аналізу журналу подій з метою виявлення відхилень від звичайної діяльності.
– Агрегація та зведення даних: Система повинна забезпечити можливість агрегувати та зведення дані з різних джерел для виявлення загальних трендів та патернів.
– Системи виявлення вторгнень (*IDS*): Розгортання систем, які моніторять мережевий трафік та активності системи для виявлення аномалій та потенційних атак.
– Повідомлення про підозрілі дії: Автоматичне сповіщення адміністратора чи відділу безпеки про підозрілі дії для швидкого реагування.
– Відстеження змін в рівнях доступу: Моніторинг змін у рівнях доступу та ролях користувачів для виявлення неправомірних дій чи втрати привілеїв.
– Виявлення спроби несанкціонованого доступу: Моніторинг неуспішних спроб входу та інших подій, що можуть вказувати на спроби несанкціонованого доступу.
– Автоматичні реакції на аномальні дії: Налаштування системи для автоматичних реакцій на підозрілі дії, такі як блокування облікових записів чи призупинення певних функцій.
– Сповіщення та інструкції для адміністраторів: Надсилання повідомлень та інструкцій адміністраторам щодо дій, які потребують їхньої уваги.
– Аналіз вразливостей та ризиків: Вивчення вразливостей та ризиків, виявлених під час моніторингу, для подальшого вжиття заходів.

- Створення звітів та аналіз причин інцидентів: Формування звітів для аналізу причин подій та вдосконалення системи безпеки.

Моніторинг та аудит безпеки в системі контролю версій забезпечують ефективний захист від потенційних загроз, оперативну реакцію на аномалії та можливість проведення ретельного розслідування подій для підвищення рівня безпеки системи.

Фізична безпека серверів та інфраструктури: Забезпечення фізичної безпеки серверів, де зберігається інформація про проекти, щоб уникнути можливості фізичного доступу несанкціонованих осіб.

- Обрання безпечного місця розташування: Вибір фізично безпечного приміщення для розміщення серверів, такого як захищений дата-центр або приміщення з обмеженим доступом.
- Фізичні бар'єри: Встановлення фізичних бар'єрів, таких як двері з електронним замком, що вимагають авторизації для доступу.
- Безпека периметру: Застосування систем контролю доступу, таких як відеоспостереження та сигналізація, для недопущення несанкціонованого вторгнення.
- Контроль температури та вологості: Забезпечення оптимальних умов зберігання обладнання для попередження перегріву та корозії.
- Захист від стихійних лих: Вживання заходів безпеки для запобігання пошкодження обладнання в результаті стихійних лих, таких як повітряні фільтри, водовідведення та інше.
- Ідентифікація користувачів: Використання систем ідентифікації (картки доступу, біометричні системи) для підтвердження авторизації осіб, які мають доступ до серверних приміщень.
- Обмеження фізичного доступу: Визначення персоналу, який має доступ до серверів, і обмеження цього доступу за необхідності.
- Відеоспостереження: Розгортання систем відеоспостереження для постійного моніторингу приміщень та реєстрації будь-яких неправомірних дій.

– Датчики вторгнень: Встановлення датчиків вторгнень для виявлення руху та неправомірної активності в областях, де розміщені сервери.
– Фізичні заходи безпеки для серверів: Замок на рівні обладнання, піддони для захисту від затоплення, антивандальні корпуси та інші заходи для захисту самого обладнання.
– Резервне живлення: Забезпечення систем живлення з можливістю автоматичного переключення на резервне живлення в разі відмови основного джерела.
– Авторизований доступ для технічного персоналу: Забезпечення тільки авторизованому технічному персоналу доступу до серверів для обслуговування та технічної підтримки.
– Ідентифікація та реєстрація відвідувачів: Встановлення процедур ідентифікації та реєстрації для всіх осіб, які звертаються до серверних приміщень.

Фізична безпека серверів та інфраструктури грає важливу роль у захисті від фізичного доступу несанкціонованих осіб, запобігаючи можливим загрозам та забезпечуючи надійність інформації про проекти.

2.3. Визначення критеріїв ефективності

У контексті сучасного програмування та розробки програмного забезпечення, система контролю версій стала невід'ємною частиною робочого процесу. Вона використовується для збереження та відстеження змін у вихідному коді проекту, дозволяючи розробникам спільно працювати, вносити зміни та відновлювати попередні версії програмного продукту. Визначення критеріїв ефективності для системи контролю версій стає стратегічно важливим завданням. Головна мета полягає в тому, щоб забезпечити оперативність та надійність управління версіями коду та його історією. Це не тільки сприяє покращенню якості розробки, але і спрощує спільну роботу команди, забезпечуючи кращий контроль над проектом. Потреба у швидкому визначенні та відновленні версій коду, мінімізація часу на виконання

операцій, таких як коміти та злиття гілок, стає критичною для ефективного використання системи контролю версій. Враховуючи обсяги даних та численність операцій у розробці, оптимізація продуктивності стає ключовим аспектом, щоб забезпечити зручну та ефективну роботу розробників. Гнучкість та легкість використання інтерфейсу системи контролю версій визначають, наскільки швидко нові члени команди можуть адаптуватися та пристосовуватися до роботи з системою. Це важливо для швидкого включення нових розробників у процес роботи над проектом. Надійність та стабільність системи умовно пов'язані зі стійкістю до завантажень та здатністю утримувати продуктивність при великому навантаженні та одночасних операціях. Забезпечення цих характеристик дозволяє уникнути простоїв та забезпечити плавну роботу команди розробників.

Визначення критеріїв ефективності для системи контролю версій також охоплює питання успішності операцій мерджу гілок та порівняння змін. Швидкість та точність цих операцій визначають, наскільки ефективно команда може об'єднувати свої робочі гілки та виявляти різниці між різними версіями коду. Таким чином, визначення критеріїв ефективності у системі контролю версій стає фундаментальним етапом у забезпеченні оптимального функціонування розробницького процесу та досягненні високої якості програмного продукту, розглянемо більш детально. Визначення критеріїв ефективності у контексті системи контролю версій має на меті створення умов для швидкого, надійного та оптимального управління версіями коду та його історією. Це визначення розглядається як стратегічна задача, яка впливає на різні аспекти розробки програмного забезпечення.

Оптимізація продуктивності в рамках системи контролю версій є невід'ємною складовою, яка визначає її ефективність та користь для розробників. Головна ідея полягає в тому, щоб забезпечити максимально швидкі та ефективні операції при взаємодії з системою, незалежно від обсягу та складності розроблюваних проектів. Ця оптимізація охоплює мінімізацію часу виконання ключових операцій, таких як коміти, пул-реквести та злиття гілок. Кожна з цих операцій є критично важливою для колективного та індивідуального внесення змін у код проекту. Швидкі та ефективні операції дозволяють розробникам зосередитися на суттєвих завданнях, замість того,

щоб витрачати час на очікування завершення дій. При оптимізації продуктивності важливо враховувати не лише час виконання окремих операцій, але й загальний рівень швидкодії системи при роботі з великою кількістю користувачів та обсягом даних. Це забезпечує консистентність роботи системи навіть у висконавантажених умовах та великих проектах. Додатково, оптимізація продуктивності сприяє поліпшенню загального досвіду користувача. Розробники відчувають зручність та швидкість взаємодії з інтерфейсом системи контролю версій, що сприяє підвищенню їхньої продуктивності та задоволеності від роботи. Ключовою частиною оптимізації є розробка та впровадження ефективних алгоритмів, що регулюють роботу системи в режимі реального часу. Такий підхід дозволяє не тільки мінімізувати час виконання операцій, а й гарантує стійкість та надійність системи навіть при інтенсивному використанні. Загалом, оптимізація продуктивності у системі контролю версій є фундаментальною для забезпечення ефективності розробки та забезпечення зручності взаємодії користувачів з цією системою. Вона виступає як ключовий фактор у забезпеченні швидкого та ефективного управління версіями коду, що, в свою чергу, визначає успішність розробки проектів та досягнення їхніх цілей.

Ефективне керування обсягами даних у системі контролю версій визначається як ключовий аспект, оскільки він впливає на продуктивність та функціональність системи при роботі з об'ємними проектами та великою кількістю файлів. Задачею є оптимізація швидкості операцій під час роботи з об'ємними даними. Це включає в себе такі ключові операції, як зберігання, відновлення та взаємодія з великим обсягом інформації. Система повинна ефективно керувати цими операціями, забезпечуючи високу швидкість доступу та виконання завдань навіть при значних розмірах даних. Особливо важливою є здатність системи адаптуватися до сучасних вимог до розробки, де об'єм та складність проектів постійно зростають. Розробники повинні мати можливість зручно та продуктивно працювати з великими обсягами даних, не відчуваючи при цьому значних затримок чи обмежень. Оптимізація швидкості роботи з файлами та проектами забезпечує не лише комфорт для розробників, але і покращує загальну ефективність розробки. Швидке зберігання та отримання інформації сприяє швидшій реакції на зміни, прискорює процеси тестування та внесення змін, що в свою

чергу сприяє швидкому впровадженню нового функціоналу та виправленню помилок. У сучасному світі, де великі обсяги даних стали нормою, ефективне керування обсягами даних у системі контролю версій стає стратегічною необхідністю. Це не тільки забезпечує оптимальну роботу системи для поточних проектів, але і готує її до майбутніх викликів та змін в області розробки програмного забезпечення.

Гнучкість та легкість використання інтерфейсу представляють собою критично важливі аспекти системи контролю версій, оскільки вони безпосередньо впливають на здатність розробників різних рівнів навичок ефективно взаємодіяти з цією системою. Гнучкість системи визначає її здатність адаптуватися до різних стилів та потреб користувачів. У контексті системи контролю версій це означає, що розробники мають можливість вибору оптимальних стратегій роботи з кодом відповідно до особливостей їхнього проекту. Гнучкість дозволяє адаптувати робочий процес до конкретних вимог команди розробників та природи проекту. Легкість використання інтерфейсу визначається його доступністю та зручністю для користувача. Інтерфейс системи контролю версій повинен бути інтуїтивно зрозумілим, навіть для тих, хто має мінімальний досвід роботи з подібними інструментами. Забезпечення простоти та зрозумілості інтерфейсу сприяє ефективній взаємодії розробників з системою та вирішує питання навчання нових користувачів. Інтуїтивність інтерфейсу є ключовим елементом, що полегшує вступ нових користувачів у роботу з системою контролю версій. Прозорість та легкість в освоєнні робочого процесу допомагає новачкам швидко адаптуватися та починати активну участь у розробці. Окрім цього, легкий та гнучкий інтерфейс зменшує можливість помилок та збільшує продуктивність розробників. Розміщення основних функцій та опцій у зручних місцях та надання доступу до них інтуїтивно допомагає розробникам швидше та точніше виконувати необхідні завдання. Отже, гнучкість та легкість використання інтерфейсу у системі контролю версій визначають якість взаємодії розробників з інструментарієм та впливають на ефективність робочого процесу. Забезпечуючи комфорт та зручність, ці аспекти сприяють продуктивності та успіху управління версіями коду.

Надійність та стабільність системи контролю версій грають визначальну роль у забезпеченні безперебійної та ефективної роботи розробників навіть у високонавантажених умовах та під час великої кількості одночасних операцій. У світі розробки програмного забезпечення, де швидкість та точність рішень мають величезне значення, надійність системи контролю версій є невід'ємним фактором. Вона визначає, наскільки ефективно система втручається у робочий процес розробки та управляє історією змін. Надійність означає, що система здатна працювати стабільно, навіть під великим навантаженням. Це важливо для уникнення витрат часу та зусиль, пов'язаних із відновленням робочого стану системи після можливих збоїв чи завантажень. Забезпечення безперебійності роботи системи призводить до зменшення можливості простоїв та максимізації продуктивності розробників. Стабільність системи є ключовим аспектом у контексті управління версіями коду та історією проектів. Вона гарантує, що розробники можуть довіряти системі навіть у ситуаціях великого обсягу одночасних операцій, таких як масові зміни коду, паралельні гілки розробки та інші складні дії. У високотехнологічному світі швидкі зміни та постійне вдосконалення проектів вимагають системи контролю версій, яка не тільки відповідає на виклики сучасності, але й забезпечує стабільність та високий рівень надійності. Від цього залежить не лише ефективність розробки, але й успіх вдосконалення та розширення програмних продуктів.

Таким чином, визначення критеріїв ефективності у системі контролю версій виявляється завданням, що охоплює різноманітні аспекти, спрямовані на створення оптимальних умов для роботи розробників та максимальної оптимізації їхнього внеску у проекти. Важливо зазначити, що ефективність у контексті системи контролю версій включає не лише швидкість виконання операцій та зручність користування, але й адаптацію до зростаючих вимог до розробки програмного забезпечення. Це означає, що система повинна бути гнучкою та готовою пристосовуватися до різних потреб розробників та вимог конкретних проектів.

Оптимізація продуктивності, визначена через мінімізацію часу виконання операцій та забезпечення ефективного керування обсягами даних, виступає як основна мета. Вона не лише забезпечує розробникам швидкий та надійний доступ до

інформації, але і створює умови для оптимальної роботи над великими та складними проектами. Безпека та конфіденційність стають критичними аспектами, оскільки система контролю версій операції з важливою інформацією. Важливість аналізу нефункціональних вимог у цьому контексті полягає в забезпеченні ефективної системи аутентифікації та авторизації, а також застосуванні шифрування та точного контролю доступу для уникнення несанкціонованого доступу та збереження конфіденційності проектів. Важливим елементом є інтеграція механізмів моніторингу та аудиту безпеки для виявлення можливих загроз та відстеження дій користувачів. Це не лише забезпечує безпеку системи, але й надає можливість оперативно реагувати на події та вживати заходів у випадку атак чи навіть недбалості користувачів. Фізична безпека серверів та інфраструктури є ще однією важливою складовою, оскільки вона спрямована на уникнення фізичного доступу несанкціонованих осіб до серверів, де зберігається інформація про проекти. Отже, визначення критеріїв ефективності в системі контролю версій включає в себе широкий спектр аспектів, які спільно працюють для створення оптимальних умов для розробників та забезпечення найвищого рівня функціональності, безпеки та надійності.

Розглянемо ключові критерії ефективності системи контролю версій, оскільки це важливий етап у визначенні її здатності надавати оптимальні умови для розробників та максимізації продуктивності при управлінні версіями коду та історією проектів. Аналіз цих критеріїв дозволяє зрозуміти, наскільки ефективно система вирішує ключові завдання, такі як оптимізація продуктивності, ефективне керування обсягами даних, гнучкість та легкість використання, надійність та стабільність, а також її успішність у виконанні мерджів та порівнянні змін.

Швидкодія, яка є одним із ключових критеріїв ефективності системи контролю версій, визначається її здатністю мінімізувати час виконання різноманітних операцій, таких як коміти, пул-реквести та злиття гілок. Цей аспект грає важливу роль у забезпеченні оптимального робочого процесу розробників, адже час, який вони витрачають на рутинні операції, напряму впливає на їхню продуктивність та швидкість розвитку проекту. Мінімізація часу виконання комітів є важливою,

оскільки це дозволяє розробникам швидко внести свої зміни до кодової бази, зберігаючи при цьому стабільність та консистентність. Подібно до цього, важливо оптимізувати час, необхідний для створення та об'єднання пул-реквестів, щоб уникнути затримок у відгуку на зміни в коді. Злиття гілок також вимагає мінімізації часових затрат. Швидке та ефективне злиття гілок дозволяє розробникам спрощувати робочий процес, підтримуючи при цьому логічну та послідовну структуру коду. Це важливо для уникнення конфліктів та забезпечення плавного розвитку проекту. Таким чином, швидкодія в контексті системи контролю версій стає важливим фактором, що визначає ефективність та зручність використання інструменту для розробників, дозволяючи їм швидко та ефективно керувати версіями коду та сприяючи швидкому розвитку програмного продукту.

Продуктивність при роботі з гілками є важливим аспектом системи контролю версій, оскільки вона визначає, наскільки ефективно користувач може працювати з різними гілками коду та здійснювати їхнє злиття. Оптимізація часових параметрів для операцій, пов'язаних із створенням та злиттям гілок, визначається необхідністю мінімізації часу, що витрачається на ці процеси. Швидкість створення гілок грає важливу роль у роботі розробників, оскільки це дозволяє їм ефективно розділяти та ізолювати різні частини коду для роботи над конкретними функціями чи вдосконаленням. Минулі версії коду можуть бути легко відновлені, а нові можливості додаються швидше завдяки оперативній роботі з гілками. Оптимізація процесу злиття гілок також визначається потребою мінімізувати час, необхідний для об'єднання внесених змін. Швидке та ефективне злиття гілок є ключовим аспектом для підтримання структури коду та уникнення конфліктів. Це дозволяє розробникам швидше впроваджувати новий функціонал та ефективно спільно працювати над проектом. Таким чином, продуктивність при роботі з гілками визначається не лише швидкістю виконання операцій, але й їхньою ефективністю у забезпеченні легкості та швидкості роботи розробників з окремими гілками коду, що сприяє швидшому розвитку та підтримці проекту.

Відновлення після відмов є ключовим елементом ефективної системи контролю версій, оскільки воно визначає, наскільки оперативно та ефективно можна відновити

функціональність системи після можливих відмов або непередбачених ситуацій. Мінімізація часу відновлення є важливою складовою для забезпечення неперервної роботи розробки та збереження стабільності проектів. Час відновлення після відмов стає критичним у випадках, коли система контролю версій стикається із ситуаціями, що порушують її функціональність. Це може бути спричинене конфліктами при злитті гілок, технічними неполадками чи іншими негативними сценаріями. Швидке та ефективне відновлення дозволяє зменшити час, протягом якого розробники не можуть працювати повноцінно, тим самим запобігаючи втратам часу та ресурсів. Масштабованість, як інший аспект, є важливим фактором для систем контролю версій, оскільки вони повинні ефективно працювати при збільшенні обсягу даних та користувачів. Це означає, що система повинна бути готовою обробляти зростаюче навантаження та масштабуватися з розвитком проектів. Інфраструктура системи контролю версій повинна бути налаштована так, щоб ефективно взаємодіяти з різними об'ємами даних та витримувати зростаючу кількість користувачів без втрати продуктивності. Отже, відновлення після відмов і масштабованість стають ключовими аспектами для забезпечення стійкості та ефективності системи контролю версій в умовах непередбачуваних ситуацій та росту проектів.

Відстеження змін та історії у системі контролю версій відіграють вирішальну роль у розробці проектів, забезпечуючи важливий інструмент для розуміння та контролю за розвитком кодової бази. Це важливо не лише для індивідуальних розробників, але й для всієї команди, щоб забезпечити структурованість та розуміння того, як відбувались зміни в часі. Швидкість та точність відтворення історії змін є критичними аспектами, оскільки вони визначають, наскільки швидко та ефективно можна відновити конкретну версію коду та зрозуміти, які саме зміни були внесені в попередніх варіантах. Це дозволяє розробникам легко визначати, коли та які зміни були внесені, а також виявляти можливі конфлікти чи неполадки. Успішність мерджа та порівняння змін є ще однією важливою аспектом в системі контролю версій. Це стосується операцій, пов'язаних з об'єднанням різних гілок коду. Мерджі вимагають точності та швидкості, оскільки неправильно виконані мерджі можуть призвести до конфліктів та помилок у коді. Додатково, швидкі та ефективні операції порівняння

змін дозволяють розробникам легко виявляти різниці між версіями коду, що є важливим для визначення впливу конкретних змін на проект. Таким чином, відстеження змін та історії, разом із успішністю мерджа та порівнянням змін, формують важливу складову системи контролю версій, сприяючи стабільності та якості розробки програмного забезпечення.

Успішність операцій мерджу та порівняння змін у системі контролю версій визначається часом виконання цих операцій та точністю їх результатів. Ці аспекти мають ключове значення, оскільки вони безпосередньо впливають на продуктивність розробників та якість кінцевого коду. Час виконання операцій мерджу гілок є важливим параметром, оскільки він визначає, наскільки швидко різні гілки коду можуть бути об'єднані. Мінімізація часу мерджу є критичною, особливо у великих проектах, де об'єм змін може бути значним. Швидкі та ефективні мерджі сприяють не тільки швидкості розробки, але і уникненню можливих конфліктів та збереженню стабільності кодової бази. Ефективність порівняння змін також має велике значення. Розробники часто потребують зручного інструмента для виявлення різниць між версіями коду. Швидкість та точність операцій порівняння дозволяють легко визначати, які саме зміни були внесені між версіями, враховуючи різні контекстуальні варіанти. Таким чином, успішність мерджу та порівняння змін визначається їхньою швидкістю та точністю, забезпечуючи надійні та продуктивні інструменти для розробників під час роботи з різними гілками коду та виявлення відмінностей у версіях коду.

2.4. Вибір мови програмування

Вибір мови програмування, а саме *C#* (рис. 2.6), для нашого проєкту, є стратегічним рішенням, яке базується на конкретних потребах та вимогах. Співвідношення ряду факторів вибору робить *C#* оптимальним вибором.

Рис. 2.6. Логотип мови *C#*

Інтеграція з екосистемою Microsoft визначається не лише можливістю безпроблемного взаємодії з іншими технологіями, але й доступністю висококласних інструментів розробки, таких як *Visual Studio*. Це створює комфортне середовище для розробки та підтримки проєкту.

Багатозадачність та асинхронність *C#* виявляються важливими аспектами для оптимізації продуктивності. Здатність ефективно керувати асинхронними операціями та працювати в багатозадачному середовищі дозволяє оптимально використовувати ресурси та забезпечує швидку відповідь.

Такий вибір мови програмування для проєкту дозволяє забезпечити не лише технічну ефективність, але й враховує зручність роботи із середовищем розробки.

2.5. Вибір фреймворку

LibGit2Sharp (рис. 2.7) - це потужний фреймворк для мови програмування *C#*, який надає зручний інтерфейс для взаємодії з системою контролю версій *Git*. Фреймворк базується на низькорівневій бібліотеці *libgit2*, яка написана на мові програмування *C*. Основна мета *LibGit2Sharp* - це забезпечити простий та ефективний спосіб роботи з *Git*-репозиторіями з допомогою мови програмування *C#*.

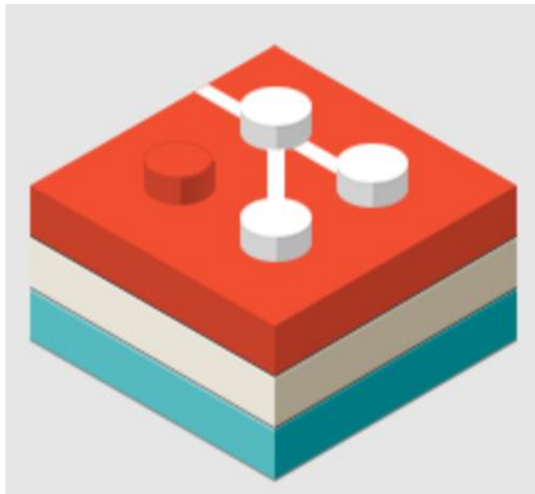


Рис. 2.7. Логотип *LibGit2Sharp*

Основні можливості *LibGit2Sharp* включають:

– Операції з репозиторіями: Вона дозволяє створювати, клонувати, відкривати і закривати *Git*-репозиторії. Ви можете отримати інформацію про репозиторій та його конфігурацію.
– Керування гілками: *LibGit2Sharp* надає можливості роботи з гілками - створювати, видаляти, переходити між ними та об'єднувати їх.
– Робота з коммітами: Ви можете створювати коміти, переглядати вміст конкретного комміту, отримувати інформацію про коміти та видаляти їх.
– Взаємодія з гітом на рівні файлів: Зможете додавати, видаляти і коммітити зміни в файлах.
– Робота з тегами і віддаленими репозиторіями: Підтримка тегів, а також взаємодія з віддаленими репозиторіями (забирати та відправляти зміни).
– Робота зі списками файлів і деревами: Взаємодія зі структурою файлів та деревами в репозиторії.
– Високорівневий *C# API*: *LibGit2Sharp* надає високорівневий інтерфейс для використання в мові програмування *C#*. Завдяки цьому, розробники можуть взаємодіяти з *Git*-репозиторіями, не вдаючись до прямих викликів *Git*-команд.

– Операції з *Git*: Бібліотека підтримує багато операцій *Git*, таких як створення та коммітування змін, робота з гілками, робота з тегами, взаємодія з віддаленими репозиторіями, та інші.
– Крос-платформенність: Оскільки *LibGit2Sharp* базується на крос-платформенній бібліотеці *libgit2*, вона може бути використана на різних операційних системах, включаючи *Windows*, *macOS* і *Linux*.
– Активна спільнота та підтримка: Бібліотека користується підтримкою від активної спільноти розробників, що забезпечує актуальність та вирішення можливих проблем.

Загалом, *LibGit2Sharp* є зручним та необхідним інструментом для створення застосунку, який буде побудований на мові програмування *C#* і буде взаємодіяти з *Git*-репозиторіями.

2.6. Вибір середовища розробки

Вибір середовища розробки, такого як *Visual Studio 2022* (рис. 2.8), може бути обумовлений рядом факторів, які включають в себе зручність використання, потужність інструментарію, підтримку мов програмування, інтеграцію з іншими інструментами.

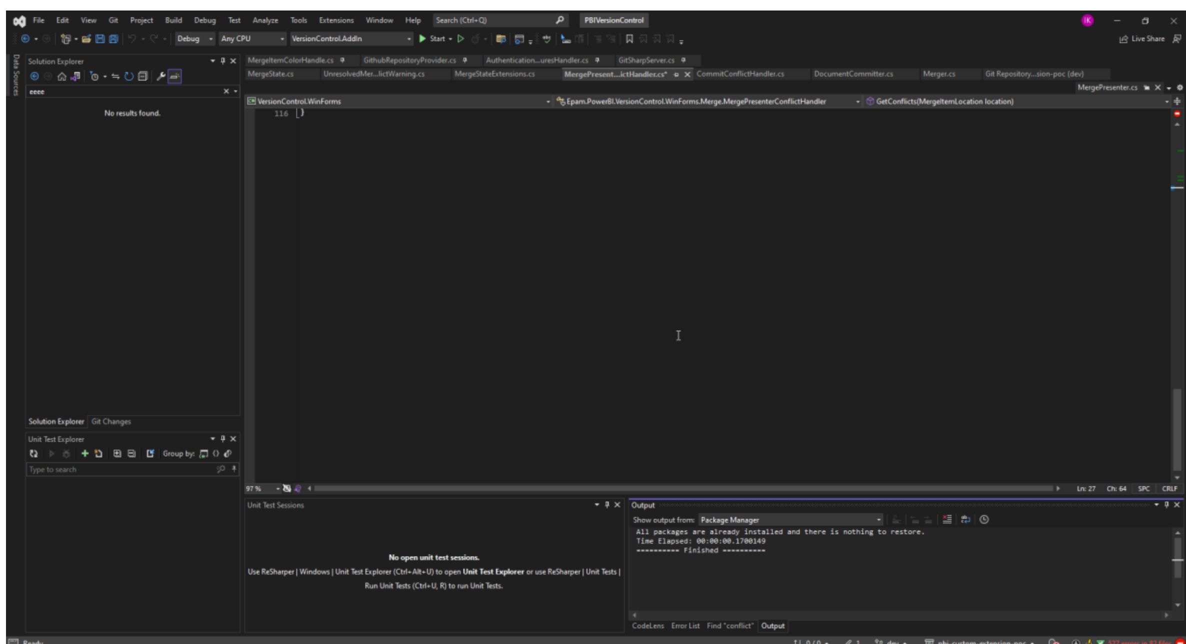


Рис. 2.8. Інтерфейс *Visual Studio 2022*

Переваги *Visual Studio 2022*:

- Широкі можливості розробки: *Visual Studio 2022* - це інтегроване середовище розробки *(IDE*), яке підтримує широкий спектр мов програмування, включаючи *C#*, *C++*, *Visual Basic*, *F#* та інші. Це дозволяє розробникам працювати над різноманітними типами проектів.
- Потужний інструментарій: *Visual Studio* надає розгорнутий набір інструментів для розробки, тестування та налагодження коду. Це включає в себе інтегровану систему контролю версій, засоби для автоматизованого тестування, засоби для налагодження та профілювання коду.
- Підтримка нових технологій: *Visual Studio 2022* постійно оновлюється для підтримки нових технологій та платформ. Він має підтримку для розробки крос-платформених додатків, веб-розробки, розробки хмарних додатків та інших актуальних технологій.
- Інтеграція з *Azure* та іншими сервісами: *Visual Studio* гарно інтегрований з обліковим записом *Microsoft* та послугами *Azure*. Це полегшує публікацію, тестування та моніторинг додатків в хмарному середовищі.
- Спільнота підтримка: *Visual Studio* користується активною спільнотою розробників, що означає, що ви можете знайти багато ресурсів, форумів, додатків та розширень, що полегшують роботу та вирішення проблем.
- Інновації у версії 2022: *Visual Studio 2022* приніс низку інновацій, таких як підтримка *.NET 6*, новий редактор коду, покращений візуальний дизайнер та інші покращення продуктивності.

Враховуючи ці фактори, *Visual Studio 2022* стала вибором для реалізації проєкту у середовищі *.NET* і на платформі *Microsoft*.

2.7. Висновки до розділу

Визначення продуктивності виявляється як ключова нефункціональна вимога, оскільки вона визначає ефективність роботи системи при обробці великого обсягу даних та операцій. Аналіз цього аспекту дозволяє визначити, наскільки система

впорається з навантаженням та забезпечить зручну роботу користувачів. Безпека та конфіденційність визнаються як надзвичайно важливі вимоги, оскільки система контролю версій опрацьовує важливу інформацію проектів. Основний акцент робиться на визначенні системи аутентифікації, авторизації та механізмів шифрування для захисту конфіденційності та уникнення несанкціонованого доступу. У висновках щодо системи аутентифікації та авторизації виявлено важливість ефективного контролю доступу, щоб гарантувати, що лише вповноважені користувачі отримують доступ до системи. Авторизація повинна бути гнучкою та базуватися на ролях для ефективного обмеження доступу до конфіденційної інформації. Щодо шифрування даних, визначено, що використання сучасних механізмів шифрування визначається як необхідний елемент для захисту інформації під час передачі та зберігання. Це допомагає уникнути можливого перехоплення та несанкціонованого доступу до конфіденційної інформації. Контроль доступу до різних ресурсів системи контролю версій визначено як обов'язкова вимога. Точний контроль доступу забезпечує управління тим, які користувачі мають доступ до різних частин системи та які дозволи вони мають. Нарешті, висновки щодо моніторингу та аудиту безпеки вказують на важливість введення системи моніторингу та аудиту для виявлення можливих загроз безпеці та відстеження дій користувачів. Це дозволяє оперативно реагувати на можливі атаки або несправедливі дії, а також забезпечує можливість розслідування подій для підвищення рівня безпеки системи. Узагальнюючи, врахування цих нефункціональних вимог дозволяє створити систему контролю версій, яка не лише ефективно керує кодом проектів, але й забезпечує його безпеку, конфіденційність та надійність у всіх аспектах функціонування.

Розглянуті критерії, такі як швидкодія, продуктивність при роботі з гілками, відновлення після відмов, масштабованість, сумісність та інтеграція, відстеження змін та історії, успішність мерджа та порівняння змін, дозволяють максимально оптимізувати роботу з системою контролю версій. Визначення критеріїв ефективності в системі контролю версій є критичним завданням, оскільки це допомагає забезпечити найкращі умови для розробників та оптимізувати їхню роботу над проектами. Шляхом мінімізації часу виконання операцій, оптимізації роботи з

гілками, ефективного відновлення після відмов та забезпечення масштабованості системи, досягається високий рівень продуктивності та задоволення потреб розробників. Успішність мерджу та точність порівняння змін стають вирішальними величинами у забезпеченні стабільності кодової бази та швидкості розробки. Це стає основою для високоякісного управління версіями коду та підтримки розробки в умовах зростаючої складності проектів. Таким чином, аналіз цих критеріїв є важливим кроком у покращенні ефективності та використанні системи контролю версій, забезпечуючи надійність, продуктивність та зручність для розробників у кожному етапі їхньої роботи.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ КОНТРОЛЮ ВЕРСІЙ ДЛЯ КОРПОРАТИВНОЇ ПЛАТФОРМИ БІЗНЕС АНАЛІТИКИ

Розробку програмного засобу контролю версій для корпоративної платформи бізнес аналітики було розділено на кілька етапів, в яких виконується розробка:

1) загального алгоритму робочого процесу системи контролю версій ;

2) алгоритму мерджа, комміту;

3) алгоритму вирішення конфліктів та мерджа ;

4) виявлення мерджів та комітів.

Розроблений програмний засіб був протестований для перевірки його належного функціонування.

3.1. Розробка загального алгоритму робочого процесу

Система контролю версій повинна мати в собі весь необхідний функціонал, щоб задовільнити потреби користувачів для забезпечення керування версіями документі, відображення внесених змін, зливання змін, виявлення та перестереження конфліктів, зберігання та відновлення інформації. На рис. 3.1 показано алгоритм роботи системи контролю версій до внесення в історію. На рис. 3.2 показано алгоритм виявлення конфліктів під час комміту.

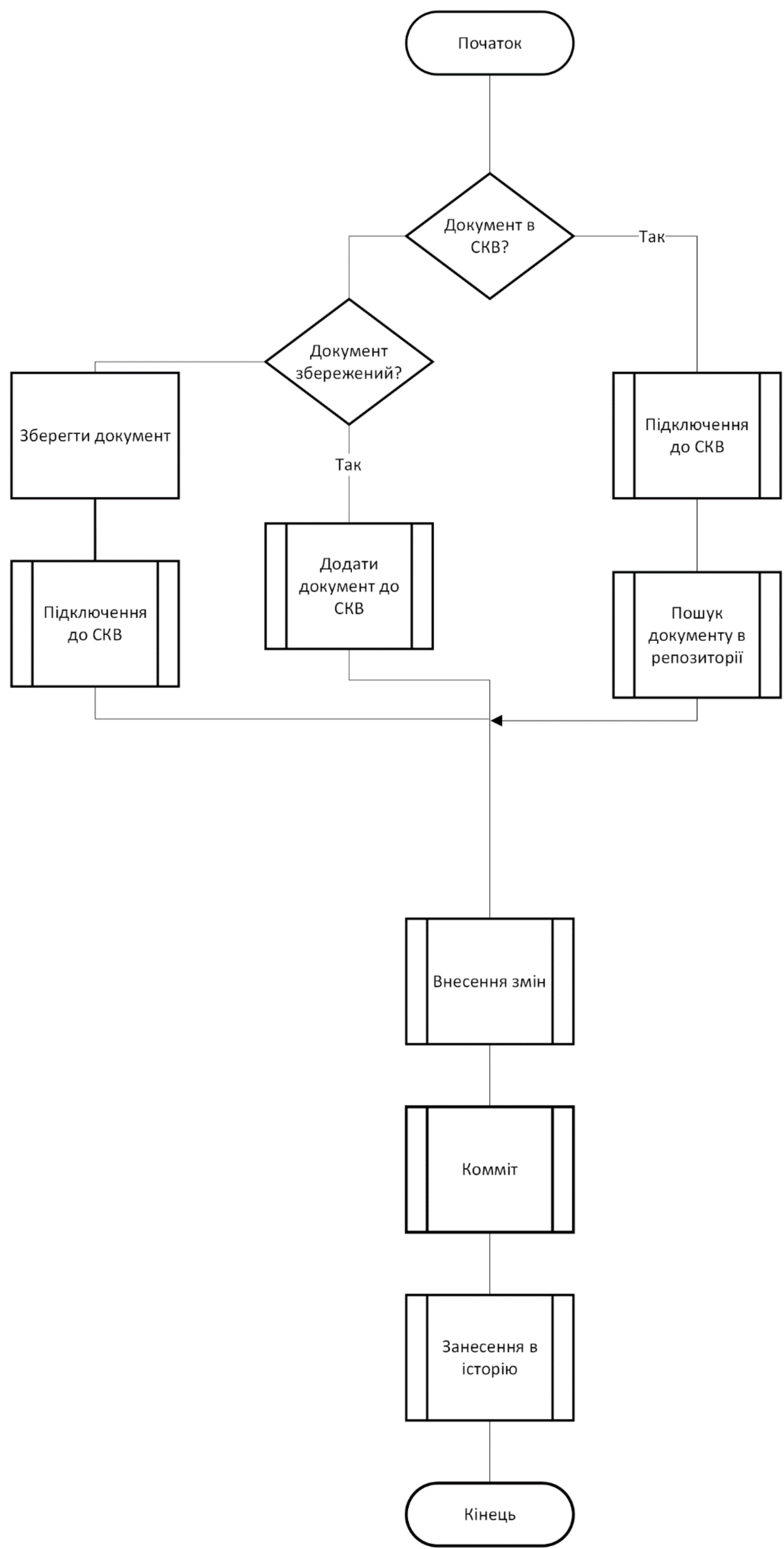


Рис. 3.1. Схема алгоритму роботи системи контролю версій до внесення в історію

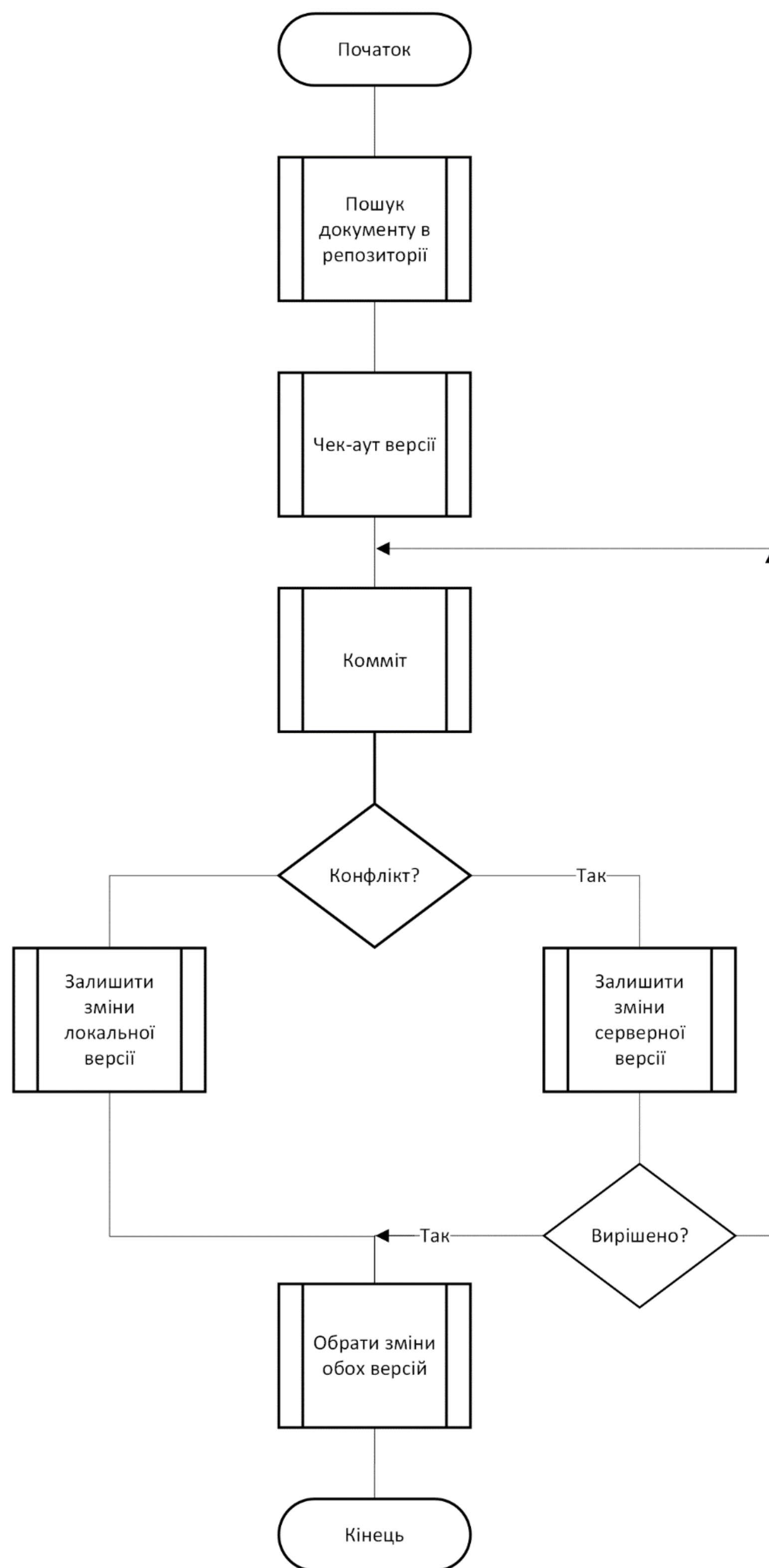


Рис. 3.2. Схема алгоритму виявлення конфліктів під час комміту

Нижче будуть описані основні складові, які будуть входити у систему контролю версій:

Репозиторій в системі контролю версій відіграє роль невід'ємної частини управління розвитком проекту, надаючи централізоване місце для зберігання історії змін. Він служить основним засобом відстеження, контролю та спільної роботи над кодовою базою.

- Репозиторій фіксує кожен етап розвитку проекту, будуючи повну історію змін. Кожен коміт, який вноситься розробником, стає важливою точкою в цій історії, зафіксовуючи конкретні зміни в коді, документації або будь-яких інших ресурсах проекту.
- Розробники можуть не лише переглядати поточний стан проекту, але і детально аналізувати, як відбувалися зміни протягом часу. Кожен коміт містить інформацію про те, хто вніс зміни, коли це сталося, та конкретні відмінності внесених змін.
- Зберігання повної історії у репозиторії дозволяє розробникам ефективно відновлювати попередні версії проекту. Це корисно в ситуаціях, коли необхідно повернутися до попереднього стану системи або визначити, які зміни внеслись між певними версіями.
- Репозиторій фіксує не лише основні зміни в коді, але і включає різноманітні елементи розробки, такі як створення та злиття гілок, видалення чи переміщення файлів. Це надає повну картину еволюції проекту та дозволяє здійснювати контроль над будь-яким аспектом його розвитку.
- Репозиторій надає можливість зберігати всі версії файлів та коду проекту, які були внесені під час різних комітів. Це означає, що кожен стан проекту може бути точно відтворений, а розробники можуть переходити між різними версіями в зручний для них спосіб.
- Управління версіями дозволяє розробникам точно визначати, які зміни були внесені між певними версіями. Завдяки цьому, можна аналізувати історію змін, визначати, як впливали певні коміти на функціональність та виправляти помилки.
- Розробники можуть створювати гілки та працювати над окремими версіями проекту паралельно. Це дозволяє відокремити експериментальні функції або

виправлення помилок від основної робочої версії, а потім злити їх при необхідності.

– Управління версіями забезпечує страхування від непередбачуваних проблем. Якщо виникає необхідність відновлення попередньої версії проекту, розробники можуть з легкістю вибрати необхідний комміт та повернутися до попереднього стану системи.
– Гілки дозволяють розробникам ізолювати різні функціональності або експериментальні зміни в окремих гілках, не впливаючи при цьому на основний код проекту. Це створює контрольоване середовище для тестування нового функціоналу чи вирішення конкретних завдань.
– Гілки надають можливість паралельного розвитку різних аспектів проекту. Оскільки розробники можуть працювати над своїми завданнями в окремих гілках, це сприяє швидшому розвитку та тестуванню різних функціональностей.
– Гілки часто використовуються для виконання експериментів або виправлення помилок. Розробники можуть створювати гілки для спроб різних підходів до розв'язання проблеми, не ризикуючи стабільність основної версії проекту.
– Після успішного тестування чи вирішення завдань розробники можуть злити зміни з гілок розробки назад в основну гілку, збагачуючи таким чином основний код новими функціональностями чи виправленнями. Цей процес злиття може бути контрольованим та піддається перевірці.
– Комміт є фундаментальним етапом у системі контролю версій. Розробники вносять зміни у код та файлову структуру проекту, відображаючи поточний стан робочого простору. Це може включати додавання нового коду, видалення файлів, редагування наявного коду, чи будь-які інші модифікації.
– Коміт фіксує ці зміни та створює нову версію проекту, яка зберігається в репозиторії. Це дозволяє відслідковувати та переходити між різними станами проекту, що важливо для контролю та відновлення попередніх версій.

– Розробники створюють гілки для розробки конкретної функціональності чи розвитку, щоб уникнути взаємного впливу з іншими гілками та основною гілкою. Це дозволяє розробникам паралельно працювати над різними аспектами проекту без конфліктів.
– Гілки дозволяють ізолювати розвиток конкретної функціональності чи експериментальні зміни, забезпечуючи безпечне тестування та виправлення помилок без впливу на основний код проекту.
– Після завершення роботи розробники можуть злити зміни з гілок назад в основну гілку, об'єднуючи розробку та збагачуючи основний код новими функціональностями. Цей процес злиття може бути контрольованим та включати вирішення конфліктів.
– Коли розробник завершує роботу над гілкою та готується злити її з основною гілкою, він переконується, що всі зміни в гілці відповідають вимогам проекту та не конфліктують з основним кодом. Це може включати вирішення конфліктів, якщо вони виникають.
– Розробник запускає процес злиття, внаслідок чого зміни з гілки об'єднуються з основною гілкою. Цей процес може бути автоматизованим або вимагати ручного підтвердження вирішення конфліктів.
– Після злиття розробники зазвичай проводять тестування, щоб переконатися, що нові зміни не порушують функціональність та взаємодію з існуючим кодом. Цей етап важливий для забезпечення стабільності проекту.
– Якщо в результаті злиття виявляються серйозні проблеми чи конфлікти, розробники можуть відмінити злиття та повернутися до попереднього стану гілки. Це забезпечує безпеку та вплив на основний код проекту.
– Репозиторій дозволяє розробникам переглядати історію змін, включаючи деталі про те, хто та коли вніс конкретні зміни. Це допомагає відстежувати внески різних учасників у розвиток проекту.
– Розробники можуть порівнювати різні версії коду, визначати зміни та розуміти, як вони вплинули на проект. Це корисно для аналізу розвитку проекту та виявлення причин можливих проблем чи неполадок.

Алгоритми обробки змін в системі контролю версій грають ключову роль у забезпеченні точності, ефективності та надійності управління змінами в коді проекту. Ці алгоритми визначають способи, якими зміни в коді взаємодіють із системою та як вони зберігаються у репозиторії

- Алгоритми виявлення змін визначають, як система реєструє зміни, які вносяться у код. Це включає в себе виявлення нових файлів, вилучення, переміщення та модифікації існуючих файлів.
- Система повинна точно ідентифікувати, які саме зміни були внесені, щоб забезпечити відслідковування кожного елементу коду.
- Алгоритми обробки конфліктів визначають, як система вирішує ситуації, коли дві або більше гілок роблять зміни у тому ж самому місці коду. Ефективне вирішення конфліктів допомагає уникнути порушень та зберегти цілісність проекту.
- Алгоритми повинні оптимізувати збереження змін, забезпечуючи компактність та швидкий доступ до інформації. Це особливо важливо при роботі з великими обсягами даних.

Система аутентифікації та авторизації в системі контролю версій є важливою складовою для забезпечення безпеки, конфіденційності та цілісності кодової бази проекту. Давайте розглянемо цей аспект більш детально:

- Ідентифікація користувача: Система аутентифікації перевіряє ідентифікаційні дані користувача, такі як ім'я користувача та пароль, щоб переконатися, що він дійсно той, за кого себе видає.
- Двофакторна аутентифікація: Забезпечується можливість використання додаткових шарів аутентифікації, наприклад, кодів, які генеруються мобільним додатком.
- Права доступу: Система авторизації визначає, до яких ресурсів та операцій має доступ користувач після успішної аутентифікації. Рівні доступу: Встановлюються різні рівні доступу для різних користувачів чи груп, забезпечуючи принцип найменших привілеїв.

– Логування дій: Система аудиту може вести журнал дій користувачів, зберігаючи записи про кожну авторизацію та взаємодію з репозиторієм. Виявлення аномалій: Механізми аудиту можуть слідкувати за аномальними діями чи спробами несанкціонованого доступу.
– Термін дії сесій: Визначається час, протягом якого користувач має можливість взаємодіяти з системою без повторної аутентифікації.

Далі розглянемо алгоритм роботи системи контролю версій, який лежить в основі робочого процесу.

Для коректної роботи із репозиторієм та системою контролю версій, користувачу необхідно додати *SSH*-ключ в *GitHub*. *SSH*-ключ є безпечним методом аутентифікації у системі, який використовується в мережевих протоколах, зокрема, в протоколі *SSH* (*Secure Shell*). Він дозволяє користувачам входити в систему, не вводячи пароль, що робить процес аутентифікації більш безпечним та зручним. *SSH*-ключі працюють за принципом використання криптографічних пар ключів для перевірки ідентичності. У процесі створення *SSH*-ключа генерується пара ключів - приватний та публічний. Приватний ключ зберігається на комп'ютері користувача, а публічний ключ передається на сервер. Коли користувач намагається підключитися до сервера за допомогою *SSH*, відбувається такий процес:

1) Користувач надсилає свій публічний ключ на сервер.
2) Сервер перевіряє цей ключ і порівнює його з тим, що вже зберігається у нього на сервері.
3) Якщо ключі співпадають, користувач отримує доступ.

Коли необхідний документ потрібно додати в систему контролю версій, то цей документ обов’язково повинен бути збережений на комп’ютері. Якщо документ не збережений, то застосунок покаже помилку про те що документ не збережено на комп’ютері.

Якщо документ збережений на комп’ютері, то наступним кроком буде додавання документ до системи контролю версій.

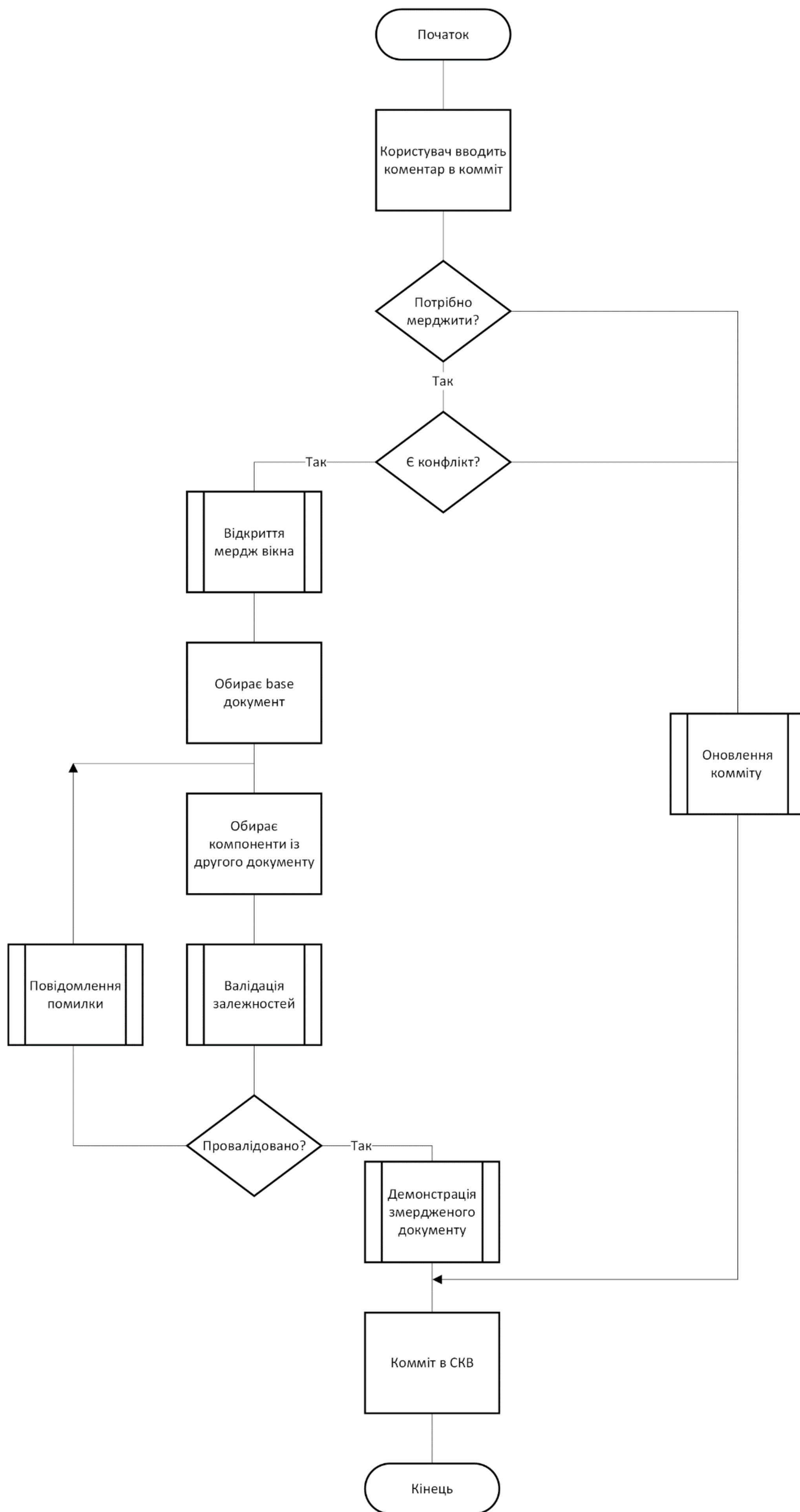
Якщо документ відсутній у системі, то його необхідно зберегти, повторити попередню операцію.

Якщо користувачу потрібно, то він робить необхідні модифікації в документі, доданому на початку.

Виконується комміт, пошук конфліктів, та перехід на стадію злиття, де можна обрати які саме зміни необхідні для злиття.

3.2. Структура мерджа та комміту системи контролю версій

Основною частиною системи контролю версій є функціонал який передбачає коміт змін до системи контролю версій, та мердж змін із документу який знаходиться на сервері з документом який було відредаговано. Даний функціонал реалізовано за допомогою бібліотеки *LibGit2Sharp*. На рис. 3.3 показано алгоритм комміту в систему контролю версій.



Початок
Користувач вводить коментар в комміт
Потрібно мерджити?
Так
Є конфлікт?
Так
Відкриття мердж вікна
Обирає base документ
Обирає компоненти із другого документу
Повідомлення помилки
Валідація залежностей
Провалідовано?
Так
Демонстрація змердженого документу
Оновлення комміту
Комміт в СКВ
Кінець

Рис. 3.3. Схема алгоритму комміту в систему контролю версій

Мердж виконується за допомогою функції *NeedToMerge* в класі *Merger*. Дана функція перевіряє *item* – документ в якому можуть бути присутні зміни: додано або видалено візуалізацію, обрано фільтр, використано сортування, модифіковано відображення або параметри. На рис. 3.4 показано алгоритм мерджа в системі контролю версій.

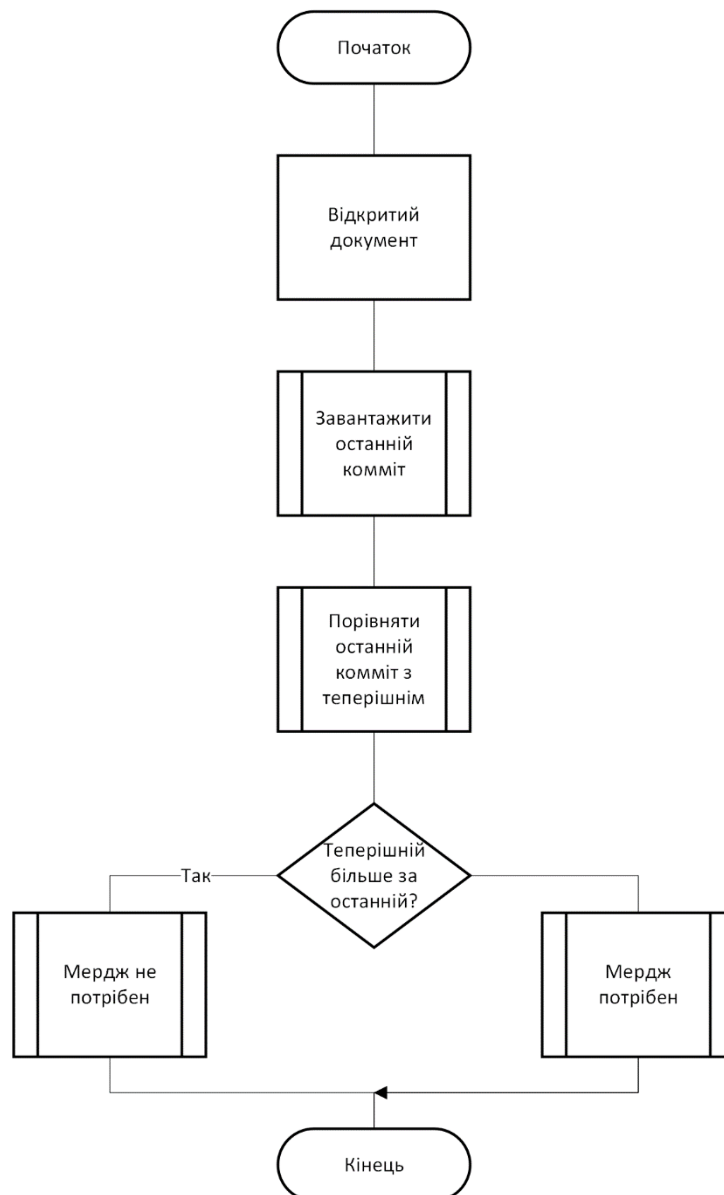


Рис. 3.3. Схема алгоритму мерджа в системі контролю версій

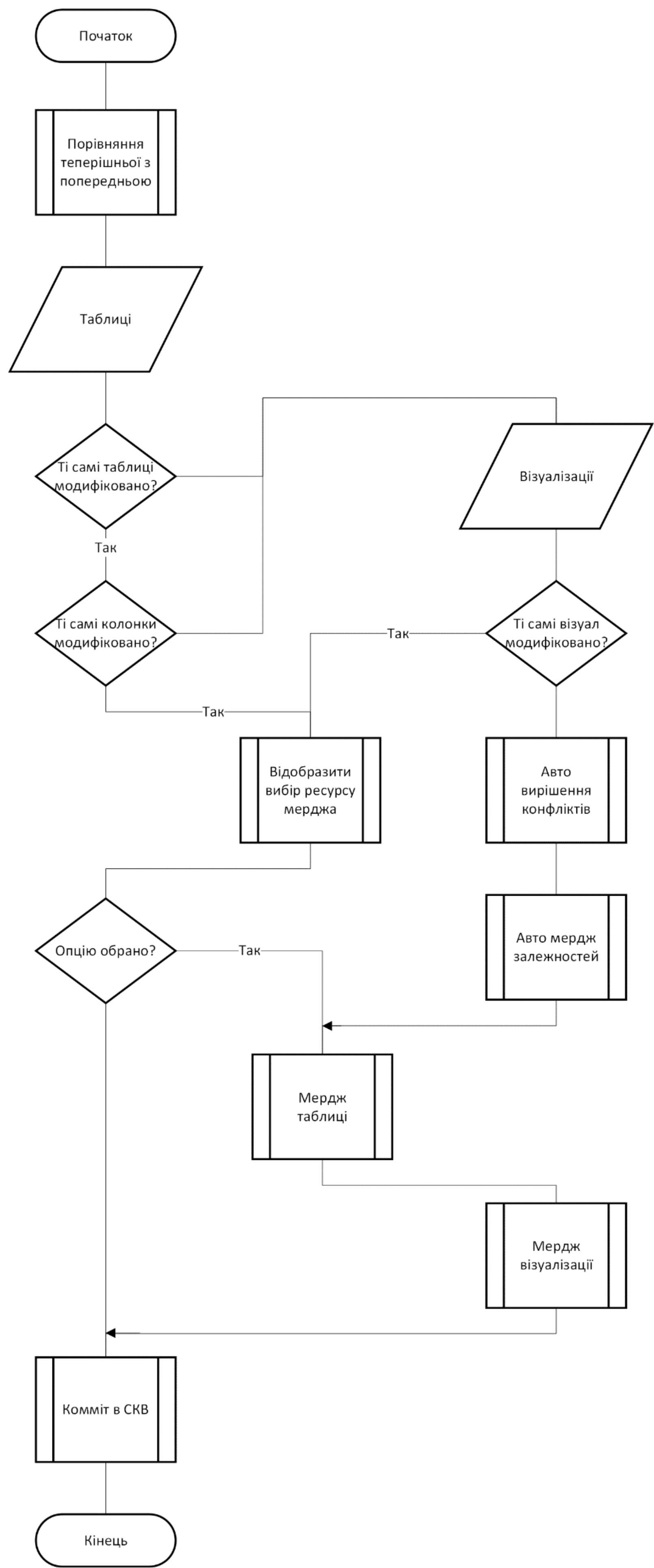
Зміни в документі перевіряються за допомогою методів *IsNotChanged*, *IsAdded*, *IsDeleted* та *IsModified*.

Комміт в систему контролю версій виконується за допомогою методу *Commit* в класі *DocumentCommitter*.

– Метод *ReportProgressAndCheckCancel*: відображає повідомлення про прогрес та перевіряє, чи не відмінено операцію. Назва методу вказує на те, що він викликається для відображення прогресу під час збереження шаблону.
– Метод *SaveTemplate*: відповідає за збереження шаблону
– Метод *RefreshHistory*: Оновлення історії комітів, отримується історія комітів, і якщо є конфлікти, викликається метод *HasConflicts*, і якщо є конфлікти, виклик завершується.
– Метод *RunInTimeBlock*: стосується редагування локального репозиторію та виконання комміту змін. Ключові слова *Edit* і *Commit* вказують на роботу з локальним репозиторієм.
– Метод *saveAfterCommit*: перевіряється результат комміту, оновлюється інформація та, якщо вказано, виконується знову збереження шаблону.

3.3. Структура вирішення конфліктів

Алгоритм детекції та вирішення конфліктів (рис. 3.4) спрямована на ефективне виявлення та обробку ситуацій, де різні версії одного і того ж елемента конфліктують під час злиття. Спочатку використовується метод *GetConflicts*, який дозволяє отримати список конфліктів у вказаному місці злиття. Після отримання цього списку визначається, чи є конфлікти для подальшого вирішення.



Початок
Порівняння теперішньої з попередньою
Таблиці
Ті самі таблиці модифіковано?
Так
Ті самі колонки модифіковано?
Візуалізації
Так
Ті самі візуал модифіковано?
Так
Відобразити вибір ресурсу мерджа
Авто вирішення конфліктів
Опцію обрано?
Так
Авто мердж залежностей
Мердж таблиці
Мердж візуалізації
Комміт в СКВ
Кінець

Рис. 3.4. Схема алгоритму детекції та вирішення конфліктів в системі контролю версій

Метод *SetCheckConflictedItems* встановлює прапорці для елементів з конфліктами, готуючи їх до подальшого розгляду. Для цього використовуються внутрішні методи, які враховують конфліктуючі елементи та їх стан, і встановлюють прапорці залежно від умов.

Процес вирішення конфліктів включає в себе позначення елементів, які потребують розгляду та визначення прапорців для їх розв'язання. Для цього використовується метод *CheckConflictedItems*, який враховує конфліктуючі елементи та їх протилежні елементи, якщо такі існують.

Загалом, ця структура дозволяє ефективно визначати, позначати та підготовляти конфліктуючі елементи для подальшого розгляду та розв'язання, спрощуючи процес управління конфліктами при злитті версій в системі контролю версій.

Метод *HasCheckableConflict*:

– Цей метод перевіряє наявність конфліктів в заданому місці злиття (локальному чи віддаленому). Він використовує метод *GetConflicts* для отримання конфліктів та повертає *true*, якщо є хоча б один конфлікт.

Метод *SetCheckConflictedItems*:

– Цей метод встановлює прапорці для елементів, що мають конфлікти, для подальшого їх розгляду. Залежно від місця злиття (локального чи віддаленого), він визначає конфліктуючі елементи та викликає *CheckConflictedItems* для встановлення прапорців.

Метод *CheckConflictedItems* :

– Цей метод встановлює прапорці для елементів, які мають конфлікти, а також їх протилежних елементів (якщо вони існують). Використовується для підготовки елементів до подальшого розгляду та рішення конфліктів.

Метод *IsConflicted*:

– Цей метод перевіряє, чи є даний об'єкт конфліктним. Враховується стан елемента та його роль у злитті.

Метод *SetCheckToResolveConflict*:

– Цей метод встановлює прапорець для елемента, щоб вказати його готовність до розв'язання конфлікту. Якщо елемент вимагає розгляду та редагування, то прапорець встановлюється безпосередньо для цього елемента; в іншому випадку, встановлюється для першого перевіреного батьківського елемента.

Метод *GetConflictedItems*:

– Цей метод отримує всі елементи, які мають конфлікти в заданому місці злиття (локальному чи віддаленому). Використовується для підготовки елементів до розгляду та розв'язання конфліктів.

Система контролю версій успішно реалізує збереження та відстеження змін на рівні окремих сторінок. Включаючи такі елементи, як таблиця, крос-таблиця та різноманітні графіки, система забезпечує точність та ефективність в управлінні різноманітними візуалізаціями.

Visualizations: Наша система успішно покриває візуалізації, такі як таблиця, крос-таблиця, графічні графіки, бра-графік, водоспадний графік, діаграма місяця, комбінована діаграма, пирогова діаграма, точкова діаграма, карта, тремблей, діаграма теплової карти, *KPI*-діаграма, паралельна координатна діаграма, коробковий графік. Кожен з цих типів візуалізацій може бути збережений та відстежений за допомогою системи контролю версій.

Table: Система здійснює контроль та збереження змін в таблиці, зокрема для заголовку, опису, даних таблиці, маркування, обмеження даних з використанням маркування, налаштувань фільтрації даних, зовнішнього вигляду, шрифтів, кольорів, сортування, легенди, показу/приховання елементів та стовпців.

Cross Table: Реалізовано збереження та відстеження змін для крос-таблиці, охоплюючи різні аспекти, такі як заголовок, опис, дані таблиці, маркування, обмеження даних через маркування, налаштування фільтрації даних, зовнішній вигляд, шрифти, кольори, сортування, легенда, показ/приховання елементів, стовпців, форматування, осі (горизонтальна, вертикальна, значення комірок), підмножини.

Bar Chart: Успішно здійснено збереження та відстеження змін для стовпчатої діаграми, охоплюючи такі елементи, як заголовок, опис, дані таблиці, маркування, обмеження даних через маркування, налаштування фільтрації даних, зовнішній вигляд, шрифти, кольори, сортування, легенда, показ/приховання елементів, осі категорій, осі значень, написи, вспливаючі підказки, трильси (рядки та стовпці, панелі), лінії та криві, полоси помилок, підмножини.

Document Properties / *Generic Fields*: Система успішно покриває роботу з загальними полями документа, включаючи налаштування та відстеження змін.

Document Properties / *Library Settings*: Реалізовано збереження та відстеження змін в бібліотечних налаштуваннях документа за допомогою системи контролю версій.

Document Properties / *Markings*: Система контролю версій забезпечує збереження та відстеження змін в маркуваннях документа.

Document Properties / *Filtering Schemes*: Здійснено контроль та збереження змін у схемах фільтрації документа.

Document Properties / *Properties*: Система покриває властивості документа, забезпечуючи ефективний контроль та збереження їхньої історії.

3.4. Інструкція із налаштування та демонстрація інтерфейсу

Використання та налаштування системи контролю версій, зокрема *Git*, вимагає виконання обов'язкових налаштувань для забезпечення правильної роботи. Перед з'єднанням із сервером *Git* необхідно визначити основні параметри. Ці налаштування можна виконати через підменю *Version Control Settings* в розділі *Git Settings* можна побачити на рис. 3.5.

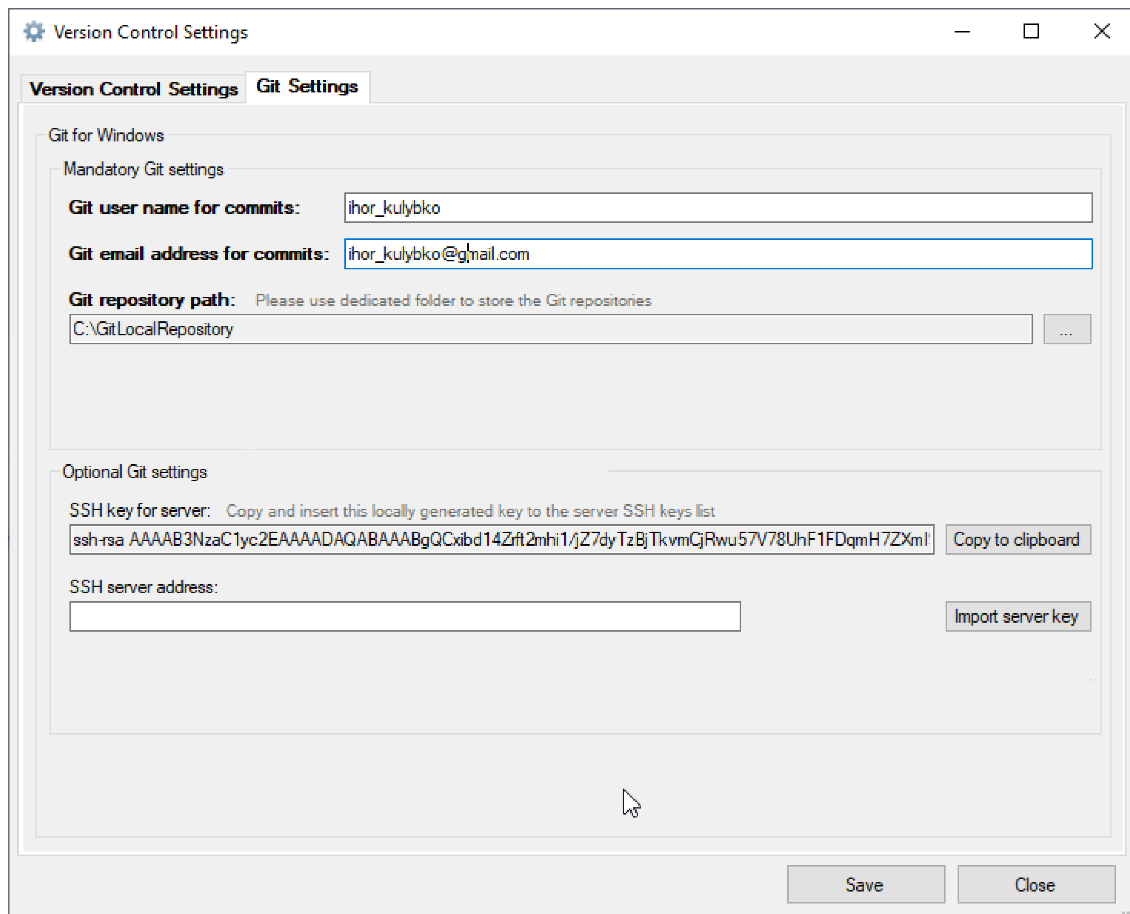


Рис. 3.5. Вікно налаштувань *Git*

Ключові параметри, які потрібно заповнити, включають ім'я користувача та електронну адресу. Кожен комміт в *Git* вимагає вказання цих особистих даних. Також необхідно вказати шлях до репозиторію *Git*. Оскільки операції *Git* спочатку виконуються локально, усі дані репозиторію повинні бути завантажені на локальний диск перед використанням будь-якої функціональності *Git*.

Також користувачу необхідно додати свій *SSH* ключ на сервіс *Git* для подальшого використання цього для авторизації на сервісі, що показано на рис. 3.6.

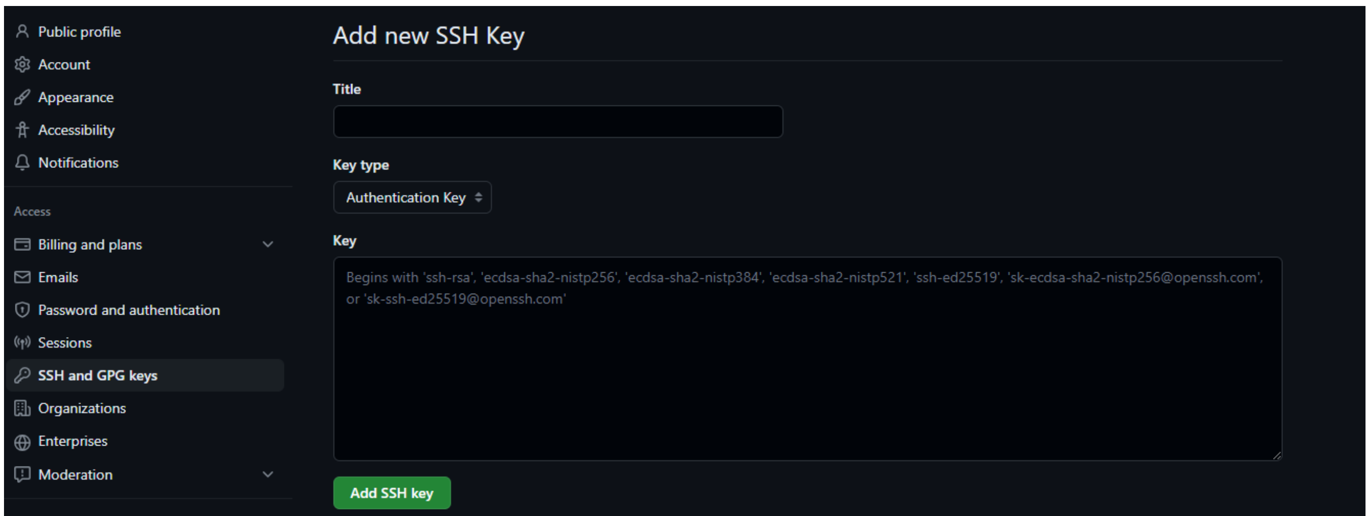


Рис. 3.6. Шлях для імпорту *SSH* ключа

Далі користувачу необхідно наступні поля в налаштування з'єднання, що показано на рис. 3.7:

- *Setting name*: Ім'я налаштування з'єднання
- *Server URL*: Посилання на сервер *Git* із вказанням необхідного репозиторію
- *Token*: Серверний *SSH* ключ

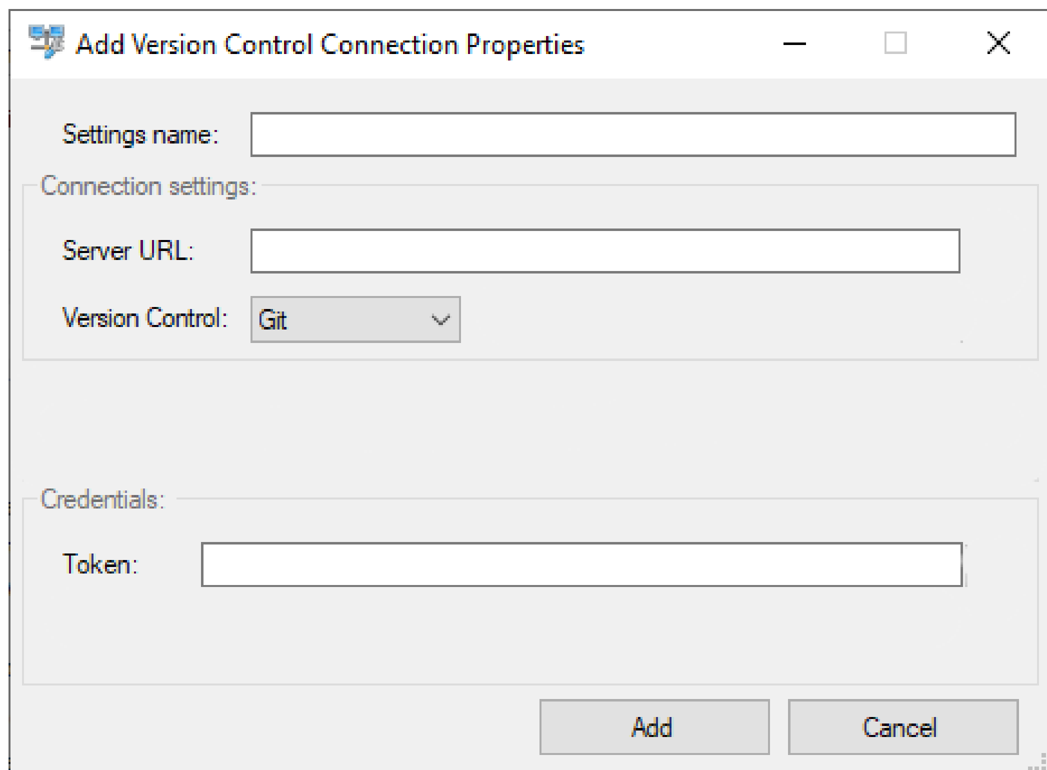


Рис. 3.7. Налаштування з'єднання *Git*

Так як дана система контролю версій націлена на роботу із корпоративними сервісами бізнес аналітики, то вона була інтегрована в середовище бізнес аналітики *TIBCO Spotfire*.

TIBCO Spotfire є потужною платформою для візуалізації та аналізу даних, розробленою *TIBCO Software Inc*. Ця платформа призначена для бізнес-аналітики та прийняття рішень, надаючи розширені можливості в галузі візуалізації даних, статистичного аналізу та інтеграції з різноманітними джерелами даних. *TIBCO Spotfire* дозволяє користувачам створювати різноманітні графіки та діаграми для ефективного представлення даних, а вбудовані візуалізаційні засоби сприяють розумінню складних взаємозв'язків у великих наборах даних. Платформа також включає багатий набір аналітичних інструментів, які охоплюють статистичний аналіз, моделювання та аналіз великих даних, спрощуючи вирішення різноманітних завдань бізнес-аналітики. *Spotfire* також може ефективно інтегруватися з різноманітними джерелами даних, забезпечуючи комплексний підхід до роботи з інформацією для зручності та ефективності користувачів. Тому в меню *Tools* і було інтегровано застосунок *Version Control*, що показано на рис. 3.8.

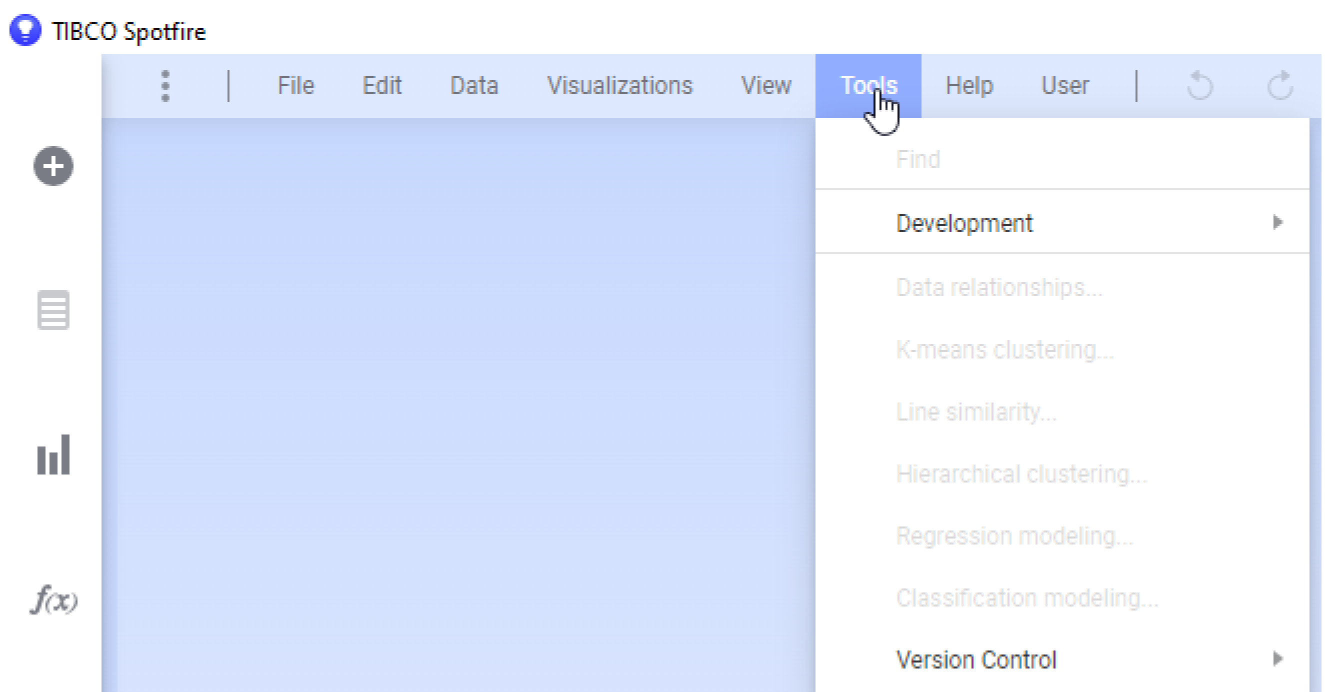


Рис. 3.8. *Version Control* в *Spotfire*

3.5. Випробовування програмного засобу

Для оцінки працездатності застосунку було вирішено провести пробний мердж системи із візуалізацією та вирішенням створених конфліктів між версіями.

Для демонстрації працездатності застосунку буде використано два документи, які були створені на базі корпоративної системи бізнес аналітики *Spotfire*. Два дані документи було додано до репозиторію.

Перший документ отримав назву *base* та в нього було додано та зображено на рис. 3.9:

- Таблицю
- Перехресну-таблицю
- Гістограму

Другий документ отримав назву *remote* та у ньому було модифіковано та зображено на рис. 3.10:

- Таблиця: Додано сортування по місяцю
- Перехресна-таблиця: Видалено
- Гістограма: Доданий розділ даних по кольорам

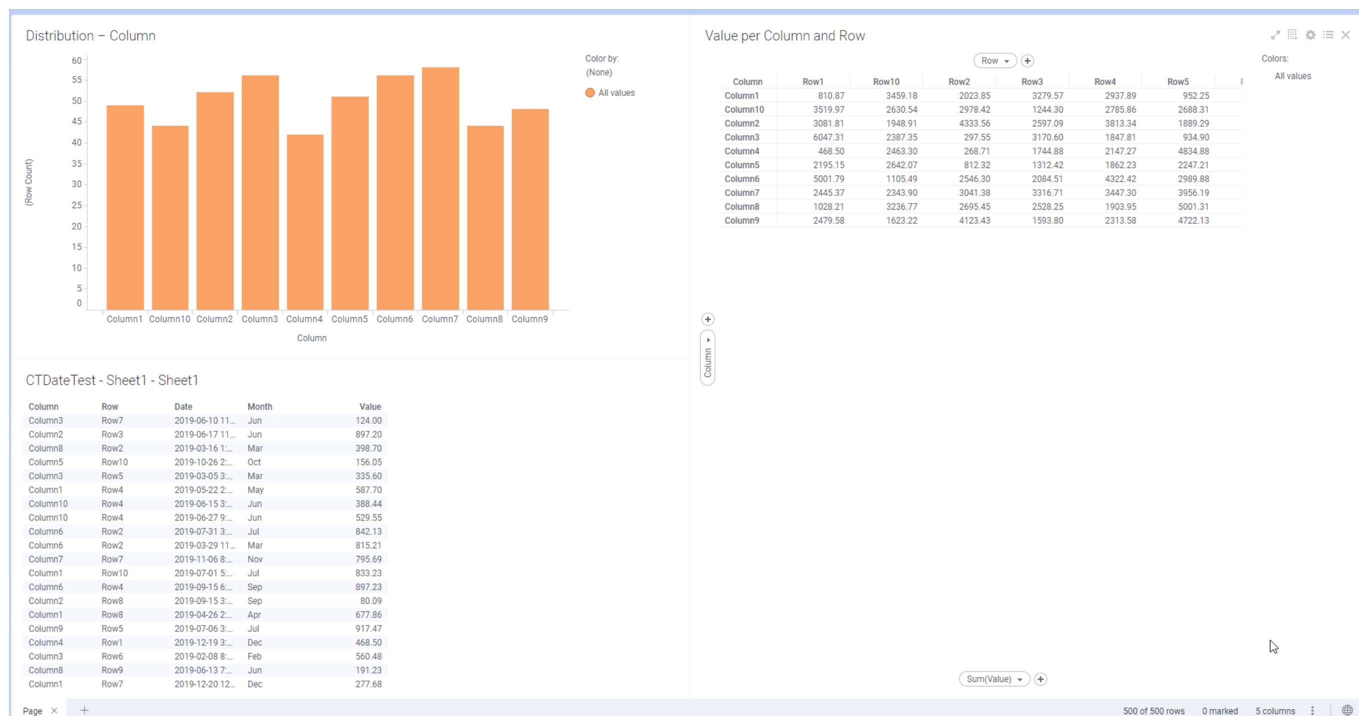


Рис. 3.9. Документ *base*

Рис. 3.10. Документ *remote*

Щоб система контролю версій змогла розпізнати модифікацію, та перейти до вибору необхідних файлів для мерджу, то потрібно в розділі *Tools* натиснути на *Version Control*, далі послідовно будуть викликані вікна пошуку необхідних файлів, спочатку *base* а потім *remote*, що показано на рис. 3.11.

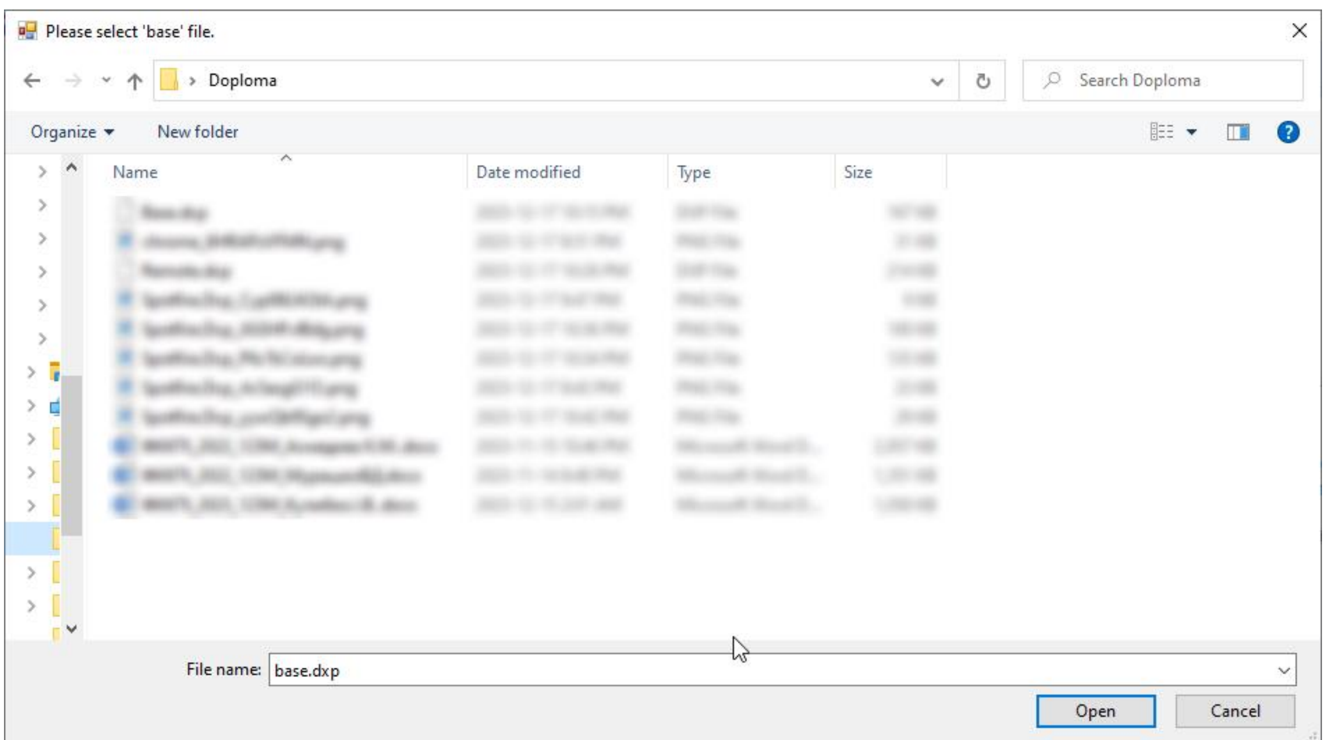


Рис. 3.11. Запит документу *base*

Після того, як два необхідні файли було обрано, з'явиться вікно мерджа де будуть вказані всі відмінності в документах і вони будуть виділені кольором, що показано на рис. 3.12:

– Чорний колір: модифікацій не знайдено
– Зелений колір: додана нова сутність
– Червоний колір: сутність видалена, або конфлікт
– Синій колір: сутність модифікована

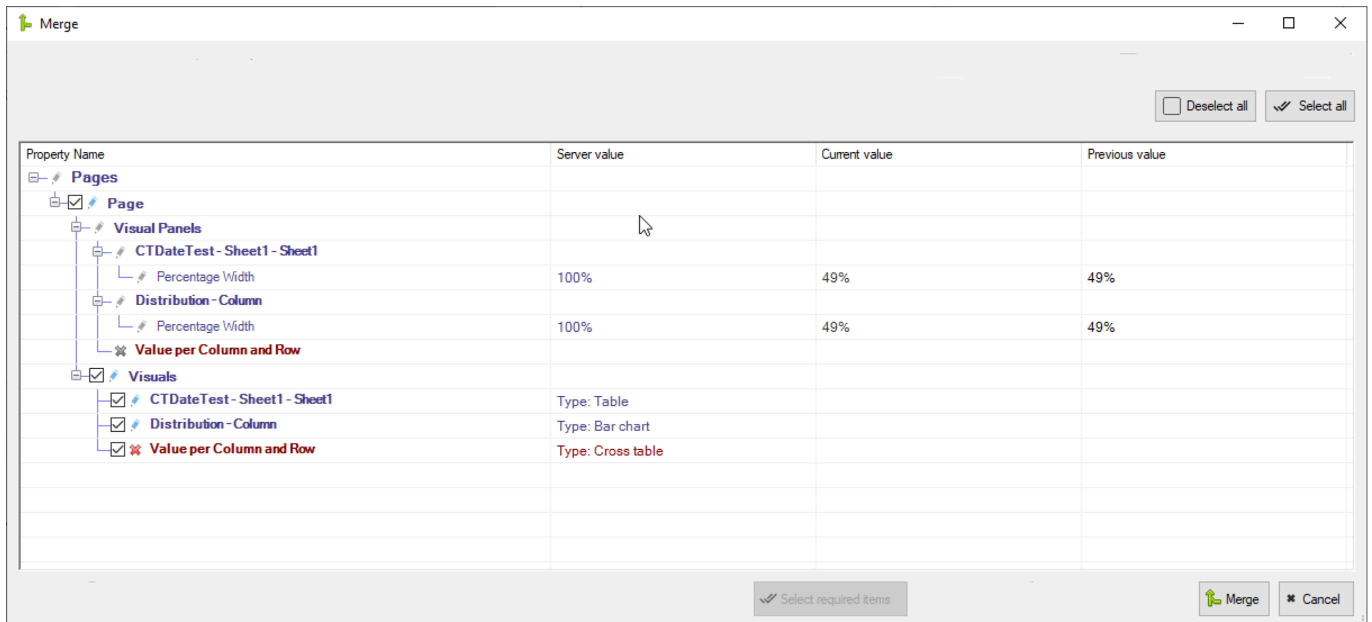
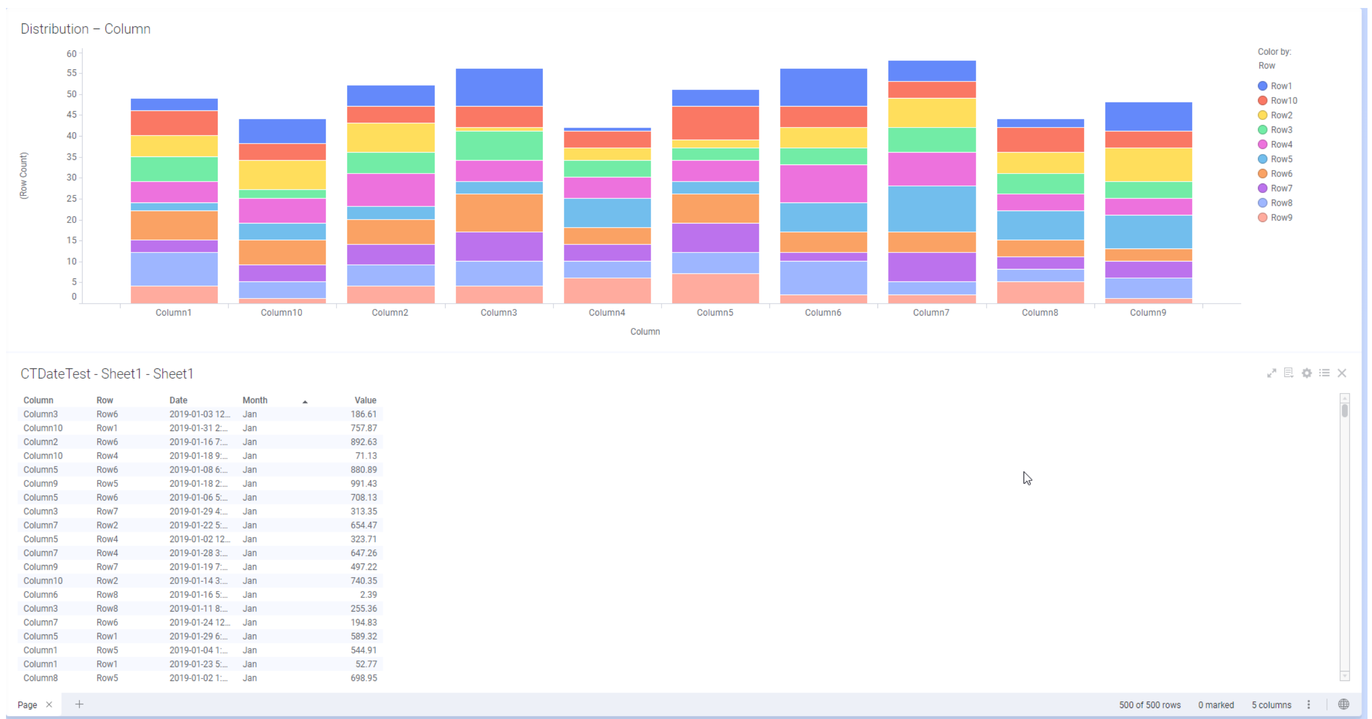


Рис. 3.12. Вікно модифікацій

Далі користувач може обрати всі необхідні зміни які будуть додані в документ після злиття, та натиснути на кнопку *Merge*. Документи будуть злиті в один фінальний варіант, що буде мати вигляд *remote* документу з модифікаціями, що показано на рис. 3.13, та версія даного документу буде оновлена в репозиторії, що показано на рис. 3.14.



Column	Row	Date	Month	Value
Column3	Row6	2019-01-03 12...	Jan	186.61
Column10	Row1	2019-01-31 2:...	Jan	757.87
Column2	Row6	2019-01-16 7:...	Jan	892.63
Column10	Row4	2019-01-18 9:...	Jan	71.13
Column5	Row6	2019-01-08 6:...	Jan	880.89
Column9	Row5	2019-01-18 2:...	Jan	991.43
Column5	Row6	2019-01-06 5:...	Jan	708.13
Column3	Row7	2019-01-29 4:...	Jan	313.35
Column7	Row2	2019-01-22 5:...	Jan	654.47
Column5	Row4	2019-01-02 12...	Jan	323.71
Column7	Row4	2019-01-28 3:...	Jan	647.26
Column9	Row7	2019-01-19 7:...	Jan	497.22
Column10	Row2	2019-01-14 3:...	Jan	740.35
Column6	Row8	2019-01-16 5:...	Jan	2.39
Column3	Row8	2019-01-11 8:...	Jan	255.36
Column7	Row6	2019-01-24 12...	Jan	194.83
Column5	Row1	2019-01-29 6:...	Jan	589.32
Column1	Row5	2019-01-04 1:...	Jan	544.91
Column1	Row1	2019-01-23 5:...	Jan	52.77
Column8	Row5	2019-01-02 1:...	Jan	698.95

Рис. 3.13. Документ *base* після злиття з *remote*

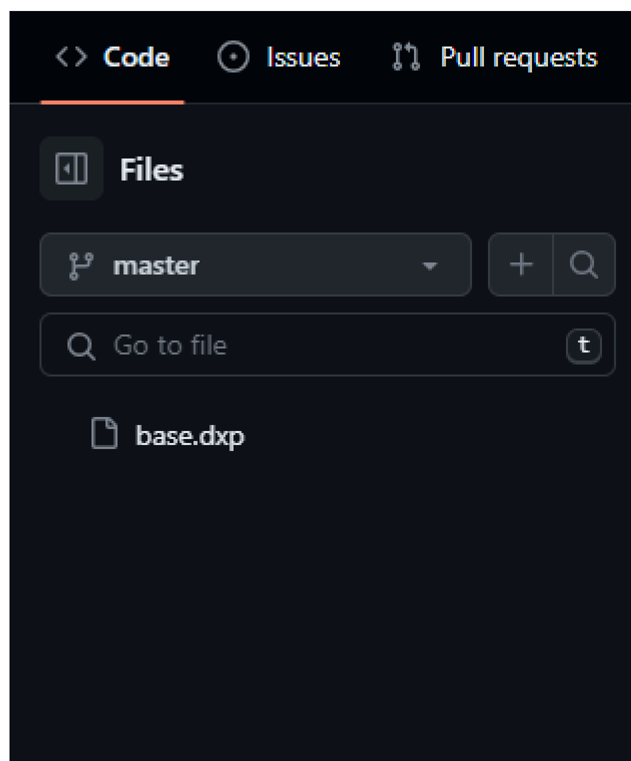


Рис. 3.14. Документ в репозиторії *Git*

Провівши дані випробування, можна вважати, що система контролю версій для корпоративної платформи бізнес аналітики є функціональною, та відповідає заданим вимогам.

3.6. Висновки до розділу

У цьому розділі був розроблений загальний алгоритм робочого процесу для системи контролю версій, що служить важливим інструментом для ефективної роботи розробників та управління різними версіями кодової бази проекту. Розроблений алгоритм враховує основні етапи взаємодії з репозиторієм, включаючи створення гілок, комміти, мерджі та вирішення конфліктів, що забезпечує плавний і структурований робочий процес розробників.

Структура мерджа та комміту в системі контролю версій є критичним елементом, визначаючим як розробники об'єднують та фіксують зміни в репозиторії, що має вирішальне значення для ефективного управління проектами. Кожен етап цього процесу впливає на розвиток кодової бази та забезпечує стабільність та надійність програмного продукту.

Комміт є фундаментальним етапом у системі контролю версій, оскільки він фіксує зміни, внесені розробниками, та створює нову версію, яка зберігається в репозиторії. Ретельно визначені кроки комміту гарантують, що історія розвитку проекту є чіткою, легко зрозумілою та відображає внесені зміни. Включення в комміт інформації про різні етапи процесу розробки, опису функціональності та призначення змін забезпечує структурованість історії, що спрощує спільну роботу розробників.

Мердж, у свою чергу, відображає процес об'єднання різних гілок розробки. Розробники об'єднують свої зміни з основною гілкою, та в ретельно виконаному мерджі ключовим елементом є уникнення конфліктів та збереження цілісності проекту. Якщо мердж виконаний правильно, це забезпечує, що новий код існуючого проекту взаємодіє з внесеними змінами ефективно та безпомилково.

Отже, структура мерджа та комміту не лише дозволяє зберігати структуру проекту та точно відображати історію змін, але і грає ключову роль у спрощенні спільної роботи розробників, забезпечуючи ефективне управління розвитком проекту на всіх етапах.

Структура вирішення конфліктів в системі контролю версій визначає ряд важливих механізмів, спрямованих на ефективне виявлення та усунення конфліктів,

що можуть виникати при об'єднанні змін з різних гілок розробки. Це важливий етап у процесі управління версіями, який робить взаємодію розробників з системою контролю версій більш надійною та зручною.

Механізми вирішення конфліктів включають в себе системи автоматичного виявлення конфліктів та ручного втручання розробника при необхідності. Система аналізує зміни, які внесені в одні й ті ж файли різними користувачами або в різних гілках, та намагається автоматично злити ці зміни. У випадках, коли автоматичне злиття неможливе або призводить до конфліктів, розробнику надається можливість ручного вирішення конфліктів.

Врахування різних сценаріїв конфліктів, таких як конфлікти внесення змін у ті самі рядки, додавання та видалення файлів тощо, забезпечує широкий спектр можливостей для ефективного вирішення суперечливих ситуацій. Детальне вивчення сценаріїв конфліктів дозволяє створити гнучкі та адаптивні інструменти для вирішення проблем, що виникають в процесі спільної роботи.

Отже, структура вирішення конфліктів в системі контролю версій є важливим компонентом, який забезпечує надійність та ефективність управління змінами, дозволяючи розробникам працювати з системою без перешкод та зберігати цілісність кодової бази проекту.

Інструкція із налаштування та демонстрація інтерфейсу дозволяє користувачам ефективно налаштовувати параметри системи контролю версій, забезпечуючи правильну ідентифікацію користувачів, вибір шляхів до репозиторіїв, а також налаштування параметрів безпеки та інших важливих опцій.

Проведені випробування системи контролю версій для корпоративної платформи бізнес аналітики підтвердили її функціональність та відповідність заданим вимогам. Для оцінки працездатності застосунку було використано два документи, які були створені на базі *Spotfire*.

Процес мерджу виявився зручним та інтуїтивно зрозумілим. Користувачеві була надана можливість вибрати необхідні файли для мерджу та чітко визначити відмінності між версіями за допомогою кольорової мітки. Чорний колір вказував на

відсутність модифікацій, зелений - на додані нові сутності, червоний - на видалені або конфліктні сутності, а синій - на модифіковані елементи.

Процес злиття було реалізовано зрозуміло та ефективно. Користувач мав можливість вручну вибрати необхідні зміни та об'єднати їх у фінальний варіант. Після успішного злиття версія документу була оновлена в репозиторії.

Такий підхід дозволяє впевнено стверджувати, що система контролю версій в корпоративній платформі бізнес аналітики є не лише функціональною, але й досить інтуїтивно зрозумілою для користувача, що робить її ефективним інструментом для управління версіями.

ВИСНОВКИ

У ході даної роботи вдалося створити докладний та добре працюючий алгоритм робочого процесу для системи контролю версій, що став необхідним інструментом для оптимізації роботи розробників та ефективного управління різними версіями кодової бази проекту. Цей алгоритм ретельно розглядає всі ключові етапи взаємодії з репозиторієм, зокрема створення та управління гілками, фіксація змін через комміти, злиття гілок (мерджі), а також ефективне вирішення конфліктів.

Загальний алгоритм надає розробникам чіткі та зрозумілі кроки для кожного етапу, що сприяє однаковому розумінню та дотриманню робочого процесу у всьому розробницькому колективі. Врахування всіх можливих сценаріїв взаємодії з репозиторієм робить цей алгоритм гнучким та адаптивним до різноманітних вимог та потреб проекту.

Створення гілок дозволяє розробникам безпечно працювати над своїми завданнями, уникати конфліктів та впроваджувати зміни відокремлено. Комміти, у свою чергу, фіксують зміни, роблячи історію проекту зрозумілою та легко відслідковуваною. Процес мерджу виявився ефективним у злитті різних гілок, забезпечуючи безперервність розвитку проекту.

Однак особливо важливим аспектом виявилося вирішення конфліктів, де структура алгоритму надає розробникам потужні інструменти для виявлення та усунення конфліктів ефективно та безпечно. Це дозволяє розробникам швидко та точно вирішувати суперечливі ситуації та забезпечує надійність системи контролю версій.

Отже, вироблений алгоритм робочого процесу в системі контролю версій виявився не лише функціональним, але й високо ефективним інструментом для впорядкування та управління розробкою проекту.

Структура мерджа та комміту у системі контролю версій виявилася вирішальною для належного та ефективного управління проектами, визначаючи спосіб, яким розробники об'єднують та фіксують зміни в репозиторії. Кожен етап

цього складного процесу має вагомий вплив на розвиток кодової бази та забезпечує стабільність та надійність результуючого програмного продукту.

Комміт, як фундаментальний етап у системі контролю версій, відіграє ключову роль у фіксації змін та створенні нових версій проекту. Це не лише дозволяє зберігати чітку та легко зрозумілу історію розвитку проекту, але й створює базу для докладного вивчення та відстеження внесених змін. Коміт фіксує не лише відповідність кодової бази конкретному моменту, але і відображає в ньому важливі відомості, такі як опис функціональності, призначення змін, інтервали часу та інші деталі, які допомагають зрозуміти сутність та контекст кожної зміни.

Стосовно мерджа, цей етап виявляється критичним для здійснення об'єднання різних гілок розробки та збереження цілісності проекту. Вдало виконаний мердж гарантує, що новий код існуючого проекту взаємодіє з внесеними змінами ефективно та безпомилково. Важливо враховувати можливі конфлікти під час мерджу та активно застосовувати механізми їх вирішення для забезпечення гармонійного об'єднання змін.

Отже, комміт та мердж, взаємодіючі як ключові елементи системи контролю версій, визначають не тільки структуру проекту та його історію, а й забезпечують основу для ефективного співпраці розробників та стабільного розвитку програмного продукту на всіх етапах його життєвого циклу.

Процес мерджу в системі контролю версій виявляється невід'ємною частиною управління розробкою, відображаючи складний процес об'єднання різних гілок розробки. Цей етап грає ключову роль у вирішенні конфліктів та збереженні цілісності проекту, створюючи зручне та ефективне середовище для розробників.

Ретельно виконане злиття змін є важливим аспектом, оскільки воно дозволяє новому коду існуючого проекту взаємодіяти з внесеними змінами ефективно та безпомилково. Важливо враховувати потенційні конфлікти та активно застосовувати механізми їх вирішення для забезпечення гармонійного об'єднання змін, що, в свою чергу, сприяє стабільності та функціональності проекту.

Структура вирішення конфліктів в системі контролю версій виступає важливим компонентом, який забезпечує надійність та ефективність управління змінами. Цей

аспект дозволяє розробникам працювати з системою без перешкод, надаючи ефективні механізми виявлення та вирішення конфліктів, що можуть виникати при об'єднанні змін з різних гілок розробки. Гнучкість та адаптивність цих механізмів дозволяють ефективно вирішувати різноманітні сценарії конфліктів, забезпечуючи безперебійний розвиток проекту. Такий підхід робить взаємодію розробників з системою контролю версій більш надійною та зручною, сприяючи успішному управлінню проектом на всіх його етапах.

Проведені випробування системи контролю версій для корпоративної платформи бізнес аналітики визначили її високу функціональність та повну відповідність вимогам. Зокрема, для оцінки працездатності застосунку використовувалися два документи, які успішно інтегрувалися в репозиторій системи контролю версій під назвами "*base*" та "*remote*".

Процес мерджу виявився не лише зручним, але й інтуїтивно зрозумілим для користувача. Надавалася можливість вибору необхідних файлів для мерджу, а кольорова позначеність чітко вказувала на різниці між версіями: чорний - відсутність модифікацій, зелений - додані нові сутності, червоний - видалені або конфліктні елементи, синій - модифіковані елементи.

Процес злиття (*Merge*) було реалізовано зрозуміло та ефективно, дозволяючи користувачеві вручну визначати необхідні зміни та об'єднувати їх у фінальний варіант. Після успішного злиття версія документу автоматично оновлювалася в репозиторії.

Такий підхід дозволяє впевнено стверджувати, що система контролю версій в корпоративній платформі бізнес аналітики є не лише високофункціональною, але й легко зрозумілою для користувача. Це робить систему ефективним інструментом для управління версіями та спільної роботи над проектами, сприяючи успішному впровадженню та розвитку програмних продуктів.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 3008-95 «Документація. Звіти у сфері науки і техніки. Структура і правила оформлення»
2. ГОСТ 7.11-2004 «Система стандартів по інформації, бібліотечній та видавничій справі. Бібліографічний запис. Скорочення слів та словосполучень на іноземних та європейських мовах»
3. ДСТУ ГОСТ 7.1:2006 «Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання»;
4. ДСТУ 3582: 2013 «Бібліографічний опис скорочення слів і словосполучень в українській мові»;
5. ДСТУ 8302-2015 «Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання».
6. *GitLab*. [Електронний ресурс] – (Дата звернення: 04.09.2023). Режим доступу: *https://gitlab.com/*
7. *GitHub*. [Електронний ресурс] – (Дата звернення: 06.09.2023). Режим доступу: *https://github.com/*
8. *Bitbucket.* [Електронний ресурс] – (Дата звернення: 07.09.2023). Режим доступу: *https://bitbucket.org/*
9. *Atlassian*. [Електронний ресурс] – (Дата звернення: 07.09.2023). Режим доступу: *https://www.atlassian.com/git/*
10. *Mercurial*. [Електронний ресурс] – (Дата звернення: 07.09.2023). Режим доступу: *https://www.mercurial-scm.org/*
11. *Version Control with Git*. [Електронний ресурс] – (Дата звернення: 08.09.2023). Режим доступу: *https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control/*
12. *Revision Control: Tools and Techniques for Software Development*. [Електронний ресурс] – (Дата звернення: 09.09.2023). Режим доступу: *https://www.spiceworks.com/tech/devops/articles/what-is-version-control/*

13. *Using Version Control*. [Електронний ресурс] – (Дата звернення: 09.09.2023). Режим доступу: *https://itrevolution.com/podcast/the-idealcast-episode-10/*
14. *Version Control Systems: A Comparison*. [Електронний ресурс] – (Дата звернення: 09.09.2023). Режим доступу: *https://dannorth.net/cupid-for-joyful-coding/*
15. *The Benefits of Using Version Control*. [Електронний ресурс] – (Дата звернення: 17.09.2023). Режим доступу: *https://www.researchgate.net/*
16. *How to Choose the Right Version Control System*. [Електронний ресурс] – (Дата звернення: 17.09.2023). Режим доступу: *https://hanselminutes.com/882/dead-simple-python-with-jason-c-mcdonalds*
17. *Version Control with Git*. [Електронний ресурс] – (Дата звернення: 17.09.2023). Режим доступу: *https://www.codecademy.com/git*
18. *Pro Git Tutorial*. [Електронний ресурс] – (Дата звернення: 16.09.2023). Режим доступу: *https://www.freecodecamp.org/news/git-and-github-for-beginners/*
19. *Stack Overflow*. [Електронний ресурс] – (Дата звернення: 17.09.2023). Режим доступу: *https://stackoverflow.com/*
20. *Introduction to Git*. [Електронний ресурс] – (Дата звернення: 16.09.2023). Режим доступу: *https://www.khanacademy.org/git*
21. *Git Cheat Sheet*. [Електронний ресурс] – (Дата звернення: 18.09.2023). Режим доступу: *https://www.geeksforgeeks.org/git-cheat-sheet/*
22. *Git for Scientists*. [Електронний ресурс] – (Дата звернення: 18.09.2023). Режим доступу: *https://swcarpentry.github.io/git-novice/*
23. *Performance Evaluation of C#*. [Електронний ресурс] – (Дата звернення: 07.10.2023). Режим доступу: *https://www.mdpi.com/Csharp*
24. *C# 11 and .NET 7: What's New for Developers*. [Електронний ресурс] – (Дата звернення: 10.10.2023). Режим доступу: *https://medium.com/entech-solutions/new-features-in-c-11-net-7-with-interactive-examples-c8e4e5ea1e5aa*
25. *Building a REST API with ASP.NET Core 6 and C#*. [Електронний ресурс] – (Дата звернення: 10.10.2023). Режим доступу: *https://medium.com/net-core/build-a-restful-web-api-with-asp-net-core-6-30747197e229/*

26. *Clean Code in C#.* [Електронний ресурс] – (Дата звернення: 10.10.2023). Режим доступу: *https://methodpoet.com/clean-code/*

27. *Mastering Dependency Injection in C#*. [Електронний ресурс] – (Дата звернення: 12.10.2023). Режим доступу: *https://eightify.app/summary/gaming/clean-architecture-mastering-dependency-injection*

28. *Securing C# Applications with Modern Techniques*. [Електронний ресурс] – (Дата звернення: 12.10.2023). Режим доступу: *https://www.bytehide.com/blog/5-simple-security-tips-for-your-net-applications*

29. *The Internet of Things with ESP32*. [Електронний ресурс] – (Дата звернення: 12.10.2023). Режим доступу: *http://esp32.net/*

30. *Testing C# Applications with xUnit and NUnit*. [Електронний ресурс] – (Дата звернення: 13.10.2023). Режим доступу: *https://levelup.gitconnected.com/the-battle-of-c-testing-frameworks-nunit-vs-xunit-vs-mstest-87b60d69da18*

31. *Design Patterns for C# Developers*. [Електронний ресурс] – (Дата звернення: 13.10.2023). Режим доступу: *https://www.dofactory.com/net/design-patterns*

32. *Optimizing C# Code for Performance*. [Електронний ресурс] – (Дата звернення: 13.10.2023). Режим доступу: *https://www.bytehide.com/blog/performance-optimization-tips-csharp*

33. *Developing Web APIs with ASP.NET Core MVC and C#*. [Електронний ресурс] – (Дата звернення: 14.10.2023). Режим доступу: *https://www.toptal.com/asp-dot-net/asp-net-web-api-tutorial*

ДОДАТОК А

Програмний код засобу контролю версій для корпоративної платформи бізнес аналітики

Файл *GitSharpServ.cs:*

```
internal sealed class CustomGitServer : ICustomGitServer, ICustomLogger
{
    private readonly string _localRepositoryPath;
    private readonly Repository _repository;
    private string _repositoryName;

    private IServerCredential _serverCredentials;

    public CustomGitServer(string localRepositoryPath)
    {
        this._repository = new Repository(localRepositoryPath);
        this._localRepositoryPath = localRepositoryPath;
    }

    public void InitializeServer(string repositoryName, IServerCredential
credentials)
    {
        this._repositoryName = repositoryName;
        this._serverCredentials = credentials;
    }

    ~CustomGitServer()
    {
        this.CleanUp(false);
```

```
    }

    public IBranchItem GetCurrentBranch
    {
        get
        {
            return this.GetBranches(b =>
b.IsCurrentRepositoryHead).FirstOrDefault();
        }
    }

    public string CurrentSha => this._repository?.Head?.Tip?.Sha ?? String.Empty;

    /// <inheritdoc />
    public string LogCategory => nameof(CustomGitServer);

    public RemoteCollection GitRemotes => this._repository.Network.Remotes;

    public void AddFileToRepository(string filePath)
    {
        Guardian.ThrowIfNullOrEmpty(filePath, nameof(filePath));
        var cleanedPath = filePath.TrimStart('/');
        this._repository.Index.Add(cleanedPath);
    }

    public bool DoesBranchExist(string branchName)
    {
        return this.GetAllBranches().FirstOrDefault(b =>
b.DisplayName.InvariantEquals(branchName)) != null;
    }
```

```
public void PerformCheckout(CheckoutParameters checkoutParameters)
{
    var remotePath = checkoutParameters.RemotePath;
    if (String.IsNullOrEmpty(remotePath))
    {
        remotePath = "/";
    }

    var cleanRemotePath = IoHelper.CleanFileName(remotePath);
    string checkoutRelativePath = null;

    if (!remotePath.Contains("/") && !remotePath.Contains("\\") &&
        this._repositoryName.InvariantEquals(cleanRemotePath))
    {
        remotePath = String.Empty;
        checkoutRelativePath = this._repositoryName;
    }

    if (!String.IsNullOrEmpty(remotePath))
    {
        remotePath = remotePath.TrimStart("/");
        checkoutRelativePath = remotePath;
    }

    if (String.IsNullOrEmpty(checkoutRelativePath))
    {
        return;
    }
```

```
        var checkoutOptions = new CheckoutOptions { CheckoutModifiers =
CheckoutModifiers.Force };
        this._repository.CheckoutPaths(checkoutParameters.ChangeSetId, new[] {
checkoutRelativePath }, checkoutOptions);

        this.CopyRepoContent(checkoutParameters, remotePath);
    }

    /// <summary>
    /// Commits and pushes changes to remote git repository with the specified
message.
    /// </summary>
    /// <param name="commitMessage">The commit message.</param>
    public string PerformCommit(string commitMessage)
    {
        var signature = this.GetSignature();

        if (signature == null)
            throw new ArgumentNullException(nameof(signature));

        try
        {
            this._repository?.Commit(commitMessage, signature, signature);
        }
        catch (EmptyCommitException ex)
        {
            throw new EmptyCommitException("No changes; nothing to commit.",
ex.GetUserFriendlyException());
        }
```

```
        return this._repository?.Head.Tip.Id.Sha;
    }

    public int RemoveFile(string filePath)
    {
        if (String.IsNullOrEmpty(filePath))
        {
            return 0;
        }

        var cleanedPath = filePath.TrimStart('/');
        var fullPath = Path.Combine(this._repository.Info.WorkingDirectory,
cleanedPath);

        try
        {
            if (IoHelper.IsDirectory(fullPath) == true)
            {
                IoHelper.EmptyDirectory(fullPath, false);
            }
            else if (File.Exists(fullPath))
            {
                File.Delete(fullPath);
            }

            var status = this._repository.RetrieveStatus();
            var entries = status.Where(e => e.FilePath.StartsWith(cleanedPath));

            foreach (var statusEntry in entries)
            {
```

```
                this._repository.Index.Remove(statusEntry.FilePath);
            }

            return 1;
        }
        catch (Exception)
        {
            return 0;
        }
    }
}
```

Файл *MergeItemsColor.cs:*

```
public class MergeItemColorHandler
{
    internal static readonly Color AddedColor = Color.Green;
    internal static readonly Color ConflictColor = Color.Red;
    internal static readonly Color DefaultColor = Color.FromArgb(56, 56, 56);
    internal static readonly Color DeletedColor = Color.Maroon;
    internal static readonly Color ModifiedColor = Color.DarkSlateBlue;

    public MergeItemColorHandler()
    {
    }

    public Dictionary<MergeState, Color> ColorsForMergeStates { get; } = new
Dictionary<MergeState, Color>
    {
        { MergeState.Unchanged, DefaultColor },
        { MergeState.Added, AddedColor },
```

```
        { MergeState.Modified, ModifiedColor },
        { MergeState.Deleted, DeletedColor },
        { MergeState.Conflict, ConflictColor }
    };

    public List<KeyValuePair<MergeState, Color>> GetOppositeStates(MergeState
state, Color color)
    {
        return ColorsForMergeStates
            .Where(item => !item.Key.Equals(state) && item.Value.Equals(color))
            .ToList();
    }

    public Color GetColorByMergeState(IMergeItem mergeItem)
    {
        return GetColorByMergeState(mergeItem.State);
    }

    public Color GetColorByMergeState(MergeState state)
    {
        var color = ColorsForMergeStates[MergeState.Unchanged];

        if (state == MergeState.Added)
        {
            return ColorsForMergeStates[MergeState.Added];
        }

        if (state.IsDeleted())
        {
            color = ColorsForMergeStates[MergeState.Deleted];
```

```
        }

        if (state.IsModified() || state.IsUnknown())
        {
            color = ColorsForMergeStates[MergeState.Modified];
        }

        if (state.IsConflicted())
        {
            color = ColorsForMergeStates[MergeState.Conflict];
        }

        return color;
    }

    public void SetColorByMergeState(MergeState state, Color color)
    {
        if (!ColorsForMergeStates.ContainsKey(state))
        {
            ColorsForMergeStates.Add(state, DefaultColor);
        }

        ColorsForMergeStates[state] = color;
    }

    internal void AddOrUpdateColors(IEnumerable<KeyValuePair<MergeState,
Color>> colors)
    {
        if (colors is null)
        {
```

```
            return;
        }

        foreach (var keyValuePair in colors)
        {
            if (!ColorsForMergeStates.ContainsKey(keyValuePair.Key))
            {
                ColorsForMergeStates.Add(keyValuePair.Key, keyValuePair.Value);
            }
            else
            {
                ColorsForMergeStates[keyValuePair.Key] = keyValuePair.Value;
            }
        }
    }

    internal void UpdateColors(IEnumerable<KeyValuePair<MergeState, Color>>
colors)
    {
        if (colors is null)
        {
            return;
        }

        foreach (var keyValuePair in colors)
        {
            if (ColorsForMergeStates.ContainsKey(keyValuePair.Key))
            {
                ColorsForMergeStates[keyValuePair.Key] = keyValuePair.Value;
            }
```

```
        }
    }
}
```

Файл *Merger.cs:*

```
private readonly MergeOperationExecutor _mergeOperationExecutor;
private List<IMergeItem> _mergedItems;

public Merger(IMergeItemStorage itemStorage)
{
    this.ItemStorage = itemStorage;
    this._mergeOperationExecutor = new MergeOperationExecutor(this);
}

public event EventHandler<MergeOperationEventArgs> OperationProcessed
{
    add => this.ProcessEventHandler += value;
    remove => this.ProcessEventHandler -= value;
}

public static Func<IMergeItem, bool> ShouldMergeAddition { get; set; }

public int ItemCount => this.GetOrderedMergeItems().Count;

public EventHandler<MergeOperationEventArgs> ProcessEventHandler { get;
private set; }

/// <inheritdoc />
public IMergeItemStorage ItemStorage { get; }
```

```
public void ExecuteMerge()
{
    var mergeItems = this.GetOrderedMergeItems();

    while (mergeItems.Count > 0)
    {
        var currentItem = mergeItems.First();
        var localParent = this.GetLocalParentItem(currentItem);

        this._mergeOperationExecutor.ProcessItem(currentItem, localParent);

        mergeItems.Remove(currentItem);
    }
}

private static bool ShouldMergeOperation(IMergeItem item)
{
    if (item.IsNotChanged())
    {
        return false;
    }

    if (item.HasParent(mergeItem => item.Checked, MergeState.Added,
MergeState.Deleted))
    {
        return false;
    }

    if (item.HasTopParent())
    {
```

```
        return false;
    }

    if (!item.IsCheckable())
    {
        return false;
    }

    if (item.IsAdded() || item.IsModified())
    {
        if (ShouldMergeAdditionMethod(item) && item.Checked)
        {
            return true;
        }

        if (item.HasParent(parent => parent.Checked))
        {
            return false;
        }
    }

    return item.Checked || item.IsDeleted() && item.HasParent(parent =>
parent.Checked);
}

private static bool ShouldMergeAdditionMethod(IMergeItem mergeItem)
{
    return ShouldMergeAddition != null &&
ShouldMergeAddition.Invoke(mergeItem);
}
```

```
private IMergeItem GetLocalParentItem(IMergeItem item)
{
    var parentItem = item.Parent;
    if (parentItem == null || item.IsElementOfReadOnlyCollection())
    {
        return null;
    }

    this.ItemStorage.LocalStorage.TryGetValue(parentItem, out var parentInLocal);

    return parentInLocal;
}

private List<IMergeItem> GetOrderedMergeItems()
{
    if (this._mergedItems != null)
    {
        return this._mergedItems;
    }

    var resultRoot = this.ItemStorage.ResultStorage.TopItem;
    this._mergedItems = resultRoot.GetChildren(ShouldMergeOperation).ToList();

    var comparer = new MergeItemComparer(SortOrder.Ascending, true)
    {
        OperationOrder = OperationOrderGetter.Instance,
        ExtOperationOrder = ExtOperationOrderGetter.Instance,
        PriorityDeleteOrder = PriorityDeleteOrderGetter.Instance
    };
```

```
    comparer.Initialize();

    this._mergedItems.Sort(comparer);

    return this._mergedItems;
}
```

Файл *ConflictLicator.cs:*

```
private readonly ISpotfireVersionController _remoteController;
private readonly string _targetCommitId;
private const string TimingKey = nameof(CommitConflictHandler);

private readonly string _latestCommitId;
private readonly ISpotfireVersionController _versionController;

protected CommitConflictHandler(Report reportDocument)
    : base(reportDocument)
{
}

public CommitConflictHandler(ISpotfireVersionController versionCtrl, string
latestCommitId)
    : this(versionCtrl.Document)
{
    this._versionController = versionCtrl;
    this._latestCommitId = latestCommitId;
}

public CommitConflictHandler(ISpotfireVersionController versionCtrl,
ISpotfireVersionController remoteCtrl, string latestCommitId, string
targetCommitId)
    : this(versionCtrl, latestCommitId)
{
    this._remoteController = remoteCtrl;
    this._targetCommitId = targetCommitId;
```

```
}

protected override async Task<Guid> LoadBaseReport()
{
   return await this.LoadReport(MergeItemLocation.Base,
this._versionController.VersionControlNode.Version ?? this._latestCommitId);
}

protected override async Task<Guid> LoadReport(MergeItemLocation location,
string version = null)
{
   Report reportDocument = null;

   if (!string.IsNullOrEmpty(version) || location == MergeItemLocation.Remote
&& this._remoteController != null)
   {
      DocumentGetter documentGetter;
      if (this._remoteController != null && location ==
MergeItemLocation.Remote)
      {
         documentGetter = new DocumentGetter(this._remoteController.Factory,
this.ReportDocument);
         documentGetter.Initialize(this._remoteController.VersionController);
      }
      else
      {
         documentGetter = new DocumentGetter(this._versionController.Factory,
this.ReportDocument);
         documentGetter.Initialize();
      }

      try
      {
         var filePath = documentGetter.CheckOut(version);

         ProgressService.ExecuteSubtask($"Deserialize
{location.GetName().ToLowerInvariant()} report...");
```

```
            reportDocument = await this.Generator.GetDocumentFromFile(filePath,
"BASE", true);
        }
        finally
        {
            documentGetter.Dispose();
        }
    }

    if (reportDocument == null)
    {
        return Guid.Empty;
    }

    var reportWrapper = new ReportWrapper(reportDocument);
    var parser = new MergeTreeBuilder(this.MergeItemsStorage);

    ProgressService.ExecuteSubtask($"Parsing objects from the
{location.GetName().ToLowerInvariant()} report");

    parser.ParseObjectTree(reportWrapper, reportWrapper.DocumentNodeId,
location);

    return reportWrapper.DocumentNodeId;
}

protected override async Task<Guid> LoadRemoteReport()
{
    var commitId = this._latestCommitId;
    if (this._remoteController != null)
    {
        commitId = this._targetCommitId;
    }

    return await this.LoadReport(MergeItemLocation.Remote, commitId);
}
```