МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Литвиненко О.Є.

«___» _____2020 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"

Тема: Онлайн система контролю версій *web*-ресурсів

Виконавець: Довгалюк Д.О.

Керівник: Ткаченко В.Г.

Нормоконтролер: Тупота Є.В.

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютеризованих систем управління
Освітнього ступеня магістр
Спеціальність 123 "Комп'ютерна інженерія"
(шифр, найменування)

Спеціалізація 123.02 "Системне програмування"
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ Литвиненко О. Є.

«____»_____2020 р.

ЗАВДАННЯ
на виконання дипломної роботи (проекту)

Довгалюка Дениса Олеговича
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема роботи: Онлайн система контролю версій *web*-ресурсів
затверджена наказом ректора від " 27 " серпня 2020 року № 1203 /ст.
2. Термін виконання роботи: з 05.10.2020 до 31.12.2020
3. Вихідні дані до роботи: аналізатор версій програмного коду

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

1) принципи оцінювання якості кода в програмній інженерії;

2) методи оцінки якості програмного коду;

3) розробка програмного забезпечення для оцінки якості коду.

5. Перелік обов'язкового графічного матеріалу:

1) діаграма змін у програмній системі, орієнтованій на кінцевих користувачів;
2) візуалізація принципу роботи системи контролю версій;
3) діаграма взаємодії модуля Репозиторій в системі контролю версій;
4) вікно результатів команди порівняння версій;
5) схема алгоритму обробки запиту на пошук у системі контролю версій.

6. Календарний план

№ п/п	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1			
2			
3			
4			
5			
6			

7. Дата видачі завдання 05.10.2020

Керівник Ткаченко В.Г.
(підпис)

Завдання прийняв до виконання Довгалюк Д.О.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Онлайн система контролю версій *web*-ресурсів”: 83 с., 37 рис., *29* літературних джерел.

WEB-ПРОЕКТ, *WEB*-РЕСУРС, КОНТРОЛЬ ВЕРСІЙ, ПРОГРАМНИЙ ПРОЕКТ, ОНОВЛЕННЯ КОДУ, ПРОГРАМНИЙ КОД

Мета дослідження – розробити систему контролю версій програмного забезпечення на віддалених серверах.

Об’єкт дослідження – оновлення програмного забезпечення.

Предмет дослідження – онлайн система контролю версій *web*-ресурсів.

Встановлено, що запропонований метод оновлення версій програмного забезпечення *web*-ресурсів побудовано за парадигмою сучасних *SCV* і має сенс продовжувати роботу в даному напрямку.

Результати дипломної роботи рекомендується використовувати при командній розробці програмних модулів веб-проектів. Розвиток проекту передбачається за рахунок розширення параметрів оновлення ПЗ.

Публікація: Довгалюк Д.О. Онлайн система контролю версій *web*-ресурсів// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). – К.: НАУ, 2020. – С. 35.

ЗМІСТ

ВСТУП

На сьогоднішній день існує дві групи *VCS:* Розподілені і централізовані.

Представлені системи є клієнт-серверним програмним забезпеченням, що означає, що дані проекти знаходяться в єдиному зразку на сервері.

CVS (система одночасних версій) – одна з перших систем, яка набула великої популярності серед багатьох розробників. Сьогодні представлений програмний продукт більш не розвивається, оскільки має кілька важливих недоліків: неможливо змінити ім'я файлу, погане зберігання даних, відсутній контроль цілісності.

Subversion (SVN) – ця система контролю версій замінила описаний вище *CVS* в 2004 році і по сьогоднішній день активно використовується розробниками. Незважаючи на велику кількість переваг *CVS*, у *SVN* є деякі недоліки: неможливо видалити дані зі сховищ, проблеми зі зміною імені, труднощі в злитті гілок.

За допомогою розподілених систем контролю версій кожен розробник може зберегти копію проекту. У них також є загальне центральне сховище, яке вже містить зміни, відправлені зі збережених копій розробників, і вони вже синхронізується. Коли користувачі працюють з розподіленими системами контролю версій, вони зазвичай синхронізують свою копію з центральним репозиторієм і вносять будь – які зміни в свій локальний репозиторій.

Є кілька переваг таких систем:

– автономність програміста при роботі над проектами.

– підвищена надійність.

– гнучкість всієї системи.

Всі ці переваги отримані завдяки локальної копії центрального сховища. Ми можемо виділити найбільш відомі *DVCS* – це *Git* і *Mercurial.*

Mercurial> Представляє собою вільну систему, в якій не існує центральне сховище. Заради комфортного використання існує спеціальне консольне програмне забезпечення під назвою *Hg.* Представлені *VCS* володіє всіма

основними функціями: об’єднання, розгалуження, синхронізація. Дана система виконана на мові програмування пітон, за рахунок чого може використовуватися на всіх сучасних ОС.

Git – являє собою розподілену систему контролю версій, призначена для використання на ОС *Linux.* Ми також можемо виділити кілька популярних компаній, які використовують *VCS Git – Qt*, Лінукс, Андроїд. *VCS* за своїм стандартного функціоналу досить схожий на *Mercurial*, який описаний вище, але має ряд переваг (продуктивність) і досить популярний серед розробників. *Git* є лідером системи контролю версій.

Проблемою предметної області є відсутність цілісності. Інформація зберігається в великій кількості директорій, що сприяє її пошкодженню. База даних гарантує захист та однозначність інформації.

Зберігання ПЗ дозволяє організувати структуроване розміщення актуальних версій пакетів ПЗ в сховище даних. Пакети ПЗ розміщуються у відповідних розділах Системи контролю версій і зберігаються в них до моменту зміни свого статусу.

Статус пакета ПЗ може бути змінений адміністратором або серверної частиною Системи контролю версій. При цьому пакет ПЗ не видаляється з розділу Системи контролю версій, він переводиться до архівного стан з наступним переміщенням в відповідний розділ Системи контролю версій. Актуалізований пакет ПЗ заміщає розташований пакет ПЗ в розділі Системи контролю версій, а вихідний перекладається в архівне стан з наступним переміщенням в необхідний розділ Системи контролю версій.

Управління правами доступу дозволяє ідентифікувати суб’єкта і надати йому доступ до заданих об’єктів Системи контролю версій. Компонент управління правами доступу забезпечує довірена взаємодія з системою аутентифікації і авторизації користувачів, а також проводит журналізацію їх дії при роботі з Системою контролю версій.

Тому найкращою альтернативою для реалізації системи контролю версій являється створення программного засобу, який дозволив би швидко і ефективно

отримувати ПЗ декільком користувачам одночасно, а також забезпечив однозначність зберігаємих даних. Таким програмним засоб є база даних.

Розглянемо події, які можуть відбутися в системі контролю версій:

– надходять нові пакети. Пакети реєструються в *trunk.* Реєстрація проводиться шляхом занесення в система контролю версій таких даних: назва проекта, розробники, ліцензія, підпроекти, залежності проекта, версія, вихідні файли проекта;

– надходять зміни до проекта. Після того, як проект був зареєстрований, в директорію /*branches* надходять зміни та виправлення до проекта. Заносяться такі дані: назва проекта, розробники, ліцензія, підпроекти, залежності проекта, версія змін, вихідні файли проекта. Зміни мають відрізнятися номером версії від основного проекта та інших змін;

– користувач створює запит на отримання программного забезпечення. Користувач створює запит який складається з назви та версії проекта. Користувач отримує список залежностей відповідного проекта. Відповідні вихідні файли або їх *ftp* адресу;

– реєстрація розробника. Розробник надає дані про себе. Отримує логін та пароль, який дозволяє створювати власні проекти та надсилати зміни до інших;

– надавати звіт. Здійснюється пошук ПЗ за датою створення, тематикою;

– надавати дистрибутив користувачу. Користувач створює запит на отримання дистрибутиву, а не окремого пакету. Отримує *ftp*-адресу архіва дистрибутиву.

База даних, яка відповідатиме всім перерахованим вимогам значно спростить та пришвидшить задачу надання програмного забезпечення, оскільки людині потрібно буде головним чином лише вводити свої запити з клавіатури (або використовуючи мишу), програма автоматично оброблятиме їх і видаватиме всю необхідну інформацію.

РОЗДІЛ 1
ПРИЗНАЧЕННЯ ТА ОРГАНІЗАЦІЙНА СТРУКТУРА СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Системою контролю версій – це програмний продукт, який запам’ятовує всі модифікації даних і, при необхідності, дозволяє виконати відкат. А також додатковою функцією є можливість визначити, хто вніс різні зміни в документ.

Всі знайомі з проблемою при роботі з проектом, коли виникає можливість додати зміни, але для цього необхідно створити величезну кількість папок з різними мітками. Через деякий час кількість папок стає великим, особливо погано стежити за документами, якщо над ними працює ціла команда.

Для того, щоб усунути такі проблеми, більшість програмістів стали використовувати систему контролю версій, за допомогою неї зручно стежити за проектом як командно, так і індивідуально. З представленої системою програміст здатний відстежувати різні модифікації документів, додавати і об’єднувати гілки проектів, а також виконувати скидання документа до певних моментів. Репозиторій вважається головним визначенням. *VCS* – це спеціальне виділене сховище, на якому зберігається інформація про файли, також за допомогою сховища можливо спостерігати за модифікацією даних.

1.1. Аналіз методів управління системами контролю версій

На сьогоднішній день існує дві групи *VCS:* розподілені і централізовані.

1.1.1. Централізовані системами контролю версій

Системи є клієнт-серверним програмним забезпеченням, що означає, що дані проекти знаходяться в єдиному зразку на сервері. Ви маєте доступ до файлів через спеціальний програмний продукт, наприклад, ви можете вибрати *CVS*, *Subversion.*

*CVS (*система одночасних версій*)* – одна з перших систем, яка набула великої популярності серед багатьох розробників. Сьогодні представлений програмний продукт більш не розвивається, оскільки має кілька важливих недоліків: неможливо змінити ім'я файлу, погане зберігання даних, відсутній контроль цілісності.

Subversion (SVN) – ця система контролю версій замінила описаний вище *CVS* в 2004 році і по сьогоднішній день активно використовується розробниками. Незважаючи на велику кількість переваг *CVS*, у *SVN* є деякі недоліки: неможливо видалити дані зі сховищ, проблеми зі зміною імені, труднощі в злитті гілок (рис. 1.1).

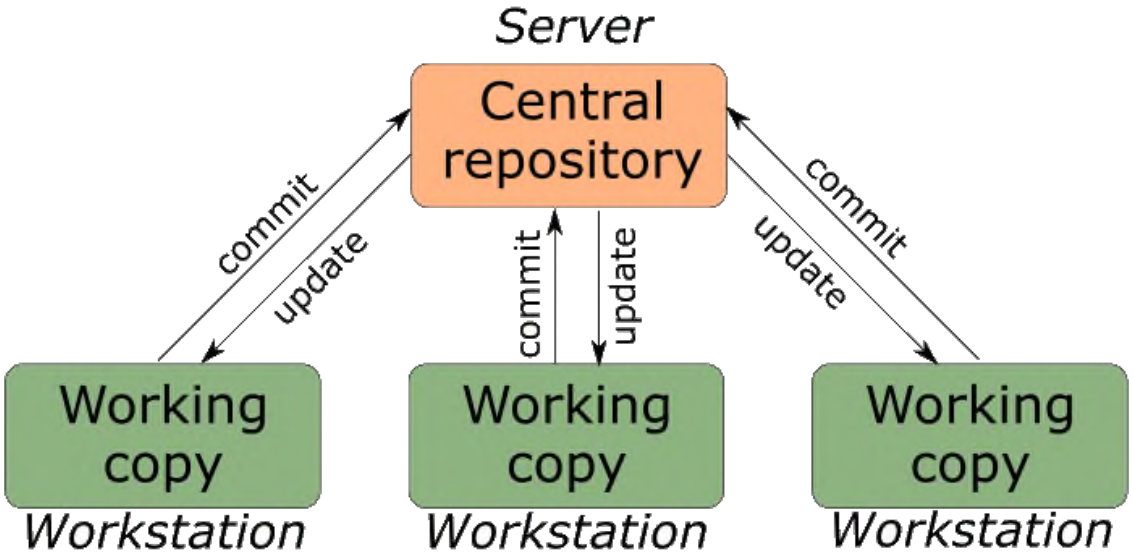

Рис. 1.1. Принцип побудови централізованих систем контролю версій

1.1.2. Розподілені системи контролю версій

За допомогою цих систем контролю версій кожен розробник може зберегти копію проекту. У них також є загальне центральне сховище, яке вже містить зміни, відправлені зі збережених копій розробників, і вони вже синхронізується. Коли користувачі працюють з розподіленими системами контролю версій, вони зазвичай синхронізують свою копію з центральним репозиторієм і вносять будь-які зміни в свій локальний репозиторій.

Є кілька переваг таких систем:

– автономність програміста при роботі над проектами;

– підвищена надійність;

– гнучкість всієї системи.

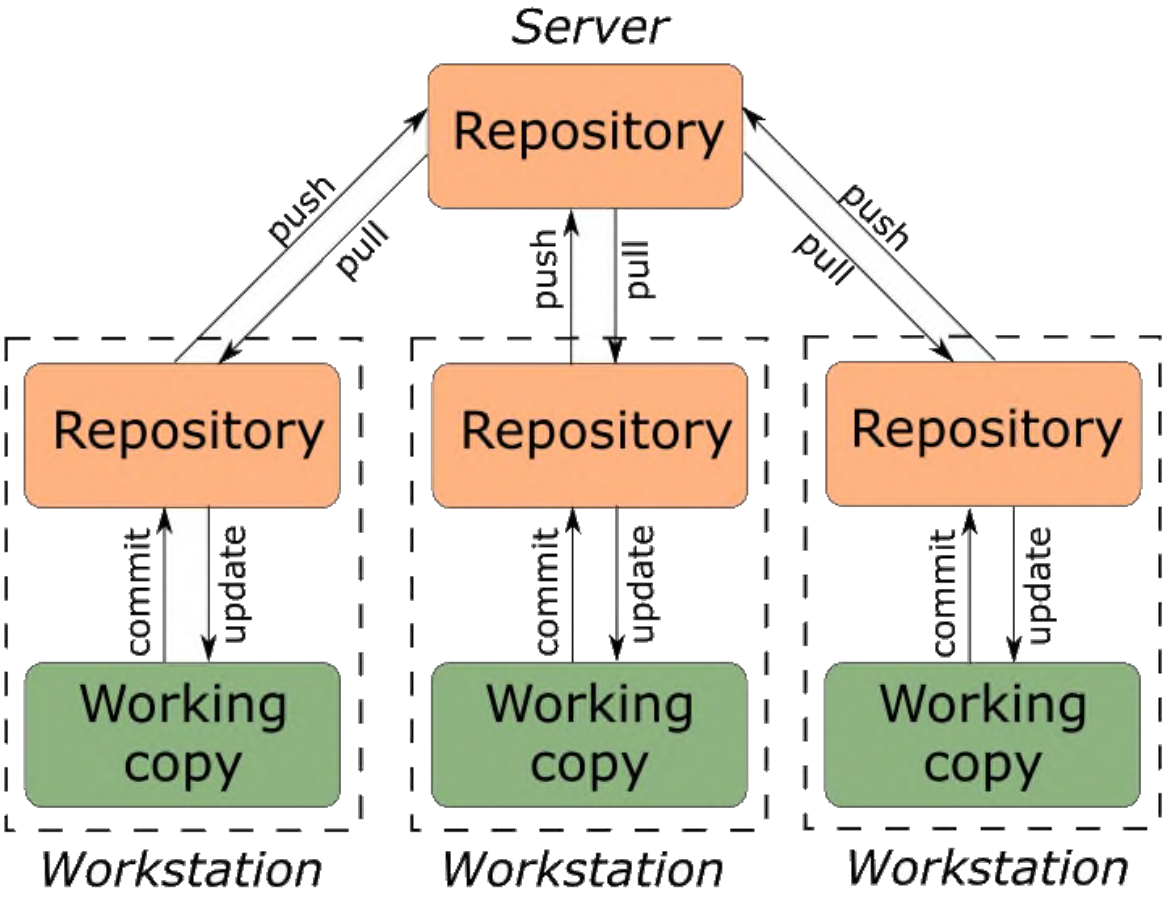

Рис. 1.2. Принцип побудови розподілених систем контролю версій

1.2. Аналіз систем контролю версій

Системи контролю версій стали невід'ємною частиною життя не тільки розробників програмного забезпечення, але і всіх людей, які зіткнулися з проблемою управління інтенсивно змінюється інформацією, і бажаючих полегшити собі життя. Внаслідок цього, з'явилася велика кількість різних продуктів, що пропонують широкі можливості і надають великі інструменти для управління версіями. У цій статті будуть коротко розглянуті найбільш популярні з них, наведені їх переваги та недоліки.

Для порівняння були обрані найбільш поширені системи контролю версій: *RCS, CVS, Subversion, Aegis, Monoton, Git, Bazaar, Arch, Perforce, Mercurial, TFS.*

1.2.1. *RCS* – система управління переглядами версій

RCS (*Revision Control System* – система управління переглядами версій), розробленої в 1985 році. Вона прийшла на зміну популярній в той час системи контролю версій *SCCS* (*Source Code Control System* – система управління вихідним кодом).

На даний момент *RCS* активно витісняється більш потужною системою контролю версій *CVS*, але все ще – досить популярна, і є частиною проекту *GNU*.

RCS дозволяє працювати тільки з окремими файлами, створюючи для кожного історію змін. Для текстових файлів зберігаються не всі версії файлу, а тільки остання версія і все зміна, внесені в неї. *RCS* також може відстежувати зміни в бінарних файлах, але при цьому кожна зміна зберігається у вигляді окремої версії файлу.

Коли зміни в файл вносить один з користувачів, для всіх інших цей файл залишається заблокованим. Вони не можуть запросити його зі сховищ для редагування, поки перший користувач не закінчить роботу і не зафіксує зміни.

Розглянемо основні переваги та недоліки системи контролю версій *RCS*.

Переваги:

1. *RCS* – проста у використанні і добре підходить для ознайомлення з принципами роботи систем контролю версій;

2. Добре підходить для резервного копіювання окремих файлів, які не потребують частої зміни групою користувачів;

3. Широко поширена і предустановленна в більшості вільно розповсюджуваних операційних системах;

Недоліки:

1. Чи відстежує зміни лише окремих файлів, що не дозволяє використовувати її для управління версіями великих проектів;

2. Не дозволяє одночасно вносити зміни в один і той же файл кількома користувачами;

3. Низька функціональність, в порівнянні з сучасними системами контролю версій.

Система контролю версій *RCS* надає занадто слабкий набір інструментів для управління розробляються проектами і підходить хіба що для ознайомлення з технологією контролю версій або ведення невеликої історії відкатів окремих файлів.

1.2.2. *CVS* – система управління паралельними версіями

Система управління паралельними версіями (*Concurrent Versions System)* – логічний розвиток системи управління переглядами версій (*RCS)*, яка використовує її стандарти і алгоритми з управління версіями, але значно більш функціональна, і дозволяє працювати не тільки з окремими файлами, а й з цілими проектами.

CVS заснована на технології клієнт-сервер, що взаємодіють по мережі. Клієнт і сервер також можуть розташовуватися на одній машині, якщо над проектом працює тільки одна людина, або потрібно вести локальний контроль версій.

Робота *CVS* організована таким чином. Остання версія і всі зроблені зміни зберігаються в репозиторії сервера. Клієнти, підключаючись до сервера, перевіряють відмінності локальної версії від останньої версії, збереженої в репозиторії, і, якщо є відмінності, завантажують їх у свій локальний проект. При необхідності вирішують конфлікти і вносять необхідні зміни в розроблюваний продукт. Після цього всі зміни завантажуються в репозиторій сервера. *CVS*, при необхідності, дозволяє відкочуватися на потрібну версію розроблюваного проекту і вести управління декількома проектами одночасно.

Наведемо основні переваги та недоліки системи управління паралельними версіями.

Переваги:

1. Кілька клієнтів можуть одночасно працювати над одним і тим же проектом;

2. Дозволяє управляти не одним файлом, а цілими проектами;

3. Володіє величезною кількістю зручних графічних інтерфейсів, здатних задовольнити практично будь-який, навіть найвимогливіший смак;

4. Широко поширена і поставляється за замовчуванням з більшістю операційних систем *Linux;*

5. При завантаженні тестових файлів з репозиторію передаються тільки зміни, а не весь файл цілком.

Недоліки:

1. При переміщенні або перейменування файлу або директорії втрачаються всі, прив'язані до цього файлу або директорії, зміни;

2. Складнощі при веденні декількох паралельних гілок одного і того ж проекту;

3. Обмежена підтримка шрифтів;

4. Для кожної зміни бінарного файлу зберігається вся версія файлу, а не тільки внесена зміна.

5. З клієнта на сервер змінений файл завжди передається повністю.

6. Ресурсоємні операції, так як вимагають частого звернення до сховища, і зберігаються копії мають деяку надмірність.

Незважаючи на те, що *CVS* застаріла і володіє серйозними недоліками, вона все ще є однією з найпопулярніших систем контролю версій і відмінно підходить для управління невеликими проектами, які не потребують створення декількох паралельних версій, які треба періодично об'єднувати. *CVS* можна порекомендувати, як проміжний крок в освоєнні роботи систем контролю версій, що веде до більш потужним і сучасним видам таких програм.

1.2.3. Система керування версіями *Subversion.*

Subversion – ця централізована система управління версіями, створена в 2000 році і заснована на технології клієнт-сервер. Вона має всі переваги *CVS* і вирішує основні її проблеми (перейменування і переміщення файлів і каталогів, робота з двійковими файлами і т.д.). Часто її називають по імені клієнтської частини – *SVN*.

Принцип роботи з *Subversion* дуже схожий на роботу з *CVS*. Клієнти копіюють зміни зі сховищ і об'єднують їх з локальним проектом користувача. Якщо виникають конфлікти локальних змін і змін, збережених в репозиторії, то

такі ситуації вирішуються вручну. Потім в локальний проект вносяться зміни, і отриманий результат зберігається в репозиторії.

При роботі з файлами, що не дозволяють об'єднувати зміни, може використовуватися наступний принцип:

1. Файл скачується з сховища та блокується (забороняється його скачування з репозиторію);

2. Вносяться необхідні зміни;

3. Завантажується файл в репозиторій і розблокується (дозволяється його скачування з репозиторію іншим клієнтам).

Багато в чому, через простоту і схожості в управлінні з *CVS*, але в основному, через свою широкої функціональності, *Subversion* з успіхом конкурує з *CVS* і навіть успішно її витісняє.

Однак, і у *Subversion є* недоліки. Розглянемо її слабкі та сильні сторони для порівняння з іншими системами управління версіями.

Переваги:

1. Система команд, подібна до *CVS;*

2. Підтримується більшість можливостей *CVS;*

3. Різноманітні графічні інтерфейси і зручна робота з консолі;

4. Відстежується історія зміни файлів і каталогів навіть після їх перейменування і переміщення;

5. Висока ефективність роботи, як з текстовими, так і з бінарними файлами;

6. Вбудована підтримка в багато інтегровані засоби розробки, такі як *KDevelop*, *Zend Studio* і багато інших;

7. Можливість створення дзеркальних копій сховища;

8. Два типу сховища – база даних або набір звичайних файлів;

9. Можливість доступу до сховища через *Apache* з використанням протоколу *WebDAV;*

10. Наявність зручного механізму створення міток і гілок проектів;

11. Можна з кожним файлом і Директорією зв'язати певний набір властивостей, що полегшує взаємодію з системою контролю версії;

12. Широке поширення дозволяє швидко вирішити більшість проблем, що виникають, звернувшись до даних, накопичених Інтернет-спільнотою.

Недоліки:

1. Повна копія сховища зберігається на локальному комп'ютері в прихованих файлах, що вимагає досить великого обсягу пам'яті;

2. Існують проблеми з перейменуванням файлів, якщо перейменований локально файл одним клієнтом був в цей же час змінений іншим клієнтом і завантажений в репозиторій;

3. Слабо підтримуються операції злиття гілок проекту;

4. Складнощі з повним видаленням інформації про файлах потрапили в репозиторій, так як в ньому завжди залишається інформація про попередні зміни файлу, і не передбачене ніяких штатних засобів для повного видалення даних про фото зі сховищ.

Subversion – сучасна система контролю версій, що володіє широким набором інструментів, що дозволяють задовольнити будь-які потреби для управління версіями проекту за допомогою централізованої системи контролю. В Інтернеті безліч ресурсів присвячено особливостям *Subversion*, що дозволяє швидко і якісно вирішувати всі виникаючі в ході роботи проблеми.

Простота установки, підготовки до роботи і широкі можливості дозволяють ставити *subversion* на одну з лідируючих позицій в конкурентній гонці систем контролю версій.

1.2.3. Система управління версіями *Aegis*

Aegis, створена Пітером Міллером в 1991 році, *є* першою альтернативою централізованим системам управління версіями. Всі операції в ній виробляються через файлову систему *Unix*. На жаль, в *Aegis* немає вбудованої підтримки роботи по мережі, але взаємодії можна здійснювати, з використанням таких протоколів, як *NFS, HTTP, FTP.*

Основна особливість *Aegis* – це спосіб контролю вносяться в репозиторій змін.

По-перше, перед занесенням будь-яких змін, вони повинні обов’язково пройти ряд тестів. І якщо нововведення в вихідний код програми не проходять тести, то потрібно або додавати нові тести, або виправляти можливі помилки у вихідному коді.

По-друге, перед внесенням змін в основну гілку розроблюваного проекту, вони повинні бути схвалені оглядачем.

По-третє, передбачена ієрархія доступу до сховища, заснована на системі прав доступу *Unix*-подібних операційних систем до файлів.

Все це робить використання системи контролю версій *Aegis* надійним, але вкрай складним, і навіть добре опрацьована документація не сильно це полегшує.

Виділимо основні переваги та недоліки системи контролю версій *Aegis.*

Переваги:

1. Надійний контроль коректності завантажуються змін;

2. Можливість надавати різні рівні доступу до фалам сховища, що дає пристойний рівень безпеки;

3. Якісна документація;

4. Можливість перейменовувати файли, збережені в репозиторії, без втрати історії змін;

5. Можливість роботи з локальним репозиторієм, якщо відсутній мережевий доступ до головного сховища.

Недоліки:

1. Відсутність вбудованої підтримки мережевої взаємодії;

2. Складність настройки і роботи з репозиторієм;

3. Слабкі графічні інтерфейси.

Складність роботи *Aegis* може відштовхнути користувачів від використання систем контролю версій, тому її не можна рекомендувати для ознайомлення або ведення невеликих програмних проектів. Однак, вона має ряд переваг, які можуть бути корисні в деяких специфічних ситуаціях, особливо, коли потрібно жорсткий контроль за якістю розроблюваного програмного забезпечення.

1.2.4. Система управління версіями *Monotone*

Monotone – ще одна децентралізована система управління версіями, розроблена Грейдон Хоем. У ній кожен клієнт сам відповідає за синхронізацію версій продукту, що розробляється з іншими клієнтами.

Робота з цією системою контролю версій – досить проста, а багато команд – схожі з командами, що використовуються в *Subversion* і *CVS*. Відмінності, в основному, полягають в організації злиття гілок проектів різних розробників.

Робота з *Monotone* будується наступним чином. В першу чергу, створюється база даних проекту *SQLite*, і генеруються ключі з використанням алгоритму хешування *SHA1* (*Secure Hash Algorithm 1)*.

Потім, по ходу коригування проекту користувачем, всі зміни зберігаються в цій базі даних, аналогічно збереженню змін в репозиторії інших систем контролю версій.

Для синхронізації проекту з іншими користувачами необхідно:

– експортувати ключ (хеш – код останньої версії проекту) і отримати аналогічні ключі від інших клієнтів.

– зберегти всім клієнтам отримані ключі в зв'язці ключів своїх локальних проектах *Monotone*.

– тепер кожен, зареєстрований таким чином користувач, може синхронізувати розробку зі своїми колегами, використовуючи простий набір команд.

Узагальнимо переваги і недоліки системи контролю версій *Monotone*.

Переваги:

1. Простий і зрозумілий набір команд, схожий з командами *Subversion* і *CVS;*

2. Підтримує перейменування і переміщення файлів і директорій;

3. Якісна документація, значно полегшує використання системи контролю версій.

Недоліки:

1. Низька швидкість роботи;

2. Відсутність потужних графічних оболонок;

3. Можливі (але надзвичайно низькі) збігу хеш – коду відмінних за змістом ревізій.

Monotone – це потужний і зручний інструмент для управління версіями розроблюваного проекту. Набір команд – продуманий і інтуїтивно зрозумілий, особливо, він буде зручний для користувачів, які звикли до роботи с *Subversion* і *CVS.* Прекрасно оформлена і повна документація дозволить швидко освоїтися і використовувати всі можливості *Subversion* на повну потужність.

Однак, відносно низька швидкість роботи і відсутність потужних графічних оболонок, можливо, зробить роботу з великими проектами кілька скрутною. Тому, якщо вам потрібна система контролю версій для підтримки складних і об'ємних продуктів, варто звернути увагу на *Git* або *Mercurial.*

1.2.5. Система управління версіями *Git*

З лютого 2002 року для розробки ядра *Linux'*а більшістю програмістів стала використовуватися система контролю версій *BitKeeper.* Досить довгий час з нею не виникало проблем, але в 2005 році Ларі МакВоем (розробник *BitKeeper'*а*)* відкликав безкоштовну версію програми.

Розробляти проект масштабу *Linux* без потужної і надійної системи контролю версій – неможливо. Одним з кандидатів і найбільш підходящим проектом виявилася система контролю версій *Monotine*, але Торвальдса Лінуса не влаштувала її швидкість роботи. Так як особливості організації *Monatone* не дозволяли значно збільшити швидкість обробки даних, то 3 квітня 2005 року Лінус приступив до розробки власної системи контролю версій – *Git.*

Практично одночасно з Лінусом (на три дні пізніше), до розробки нової системи контролю версій приступив і Метт маку. Свій проект Метт назвав *Mercurial*, але про це пізніше, а зараз повернемося до розподіленої системи контролю версій *Git.*

Git – це гнучка, розподілена (без єдиного сервера) система контролю версій, що дає масу можливостей не тільки розробникам програмних продуктів, але і письменникам для зміни, доповнення та відстеження зміни «рукописів» і

сюжетних ліній, і вчителям для коригування та розвитку курсу лекцій, і адміністраторам для ведення документації, і для багатьох інших напрямків, що вимагають управління історією змін.

У кожного розробника, який використовує *Git*, є свій локальний репозиторій, що дозволяє локально керувати версіями. Потім, збереженими в локальний репозиторій даними, можна обмінюватися з іншими користувачами.

Часто при роботі з *Git* створюють центральний репозиторій, з яким інші розробники синхронізуються. Приклад організації системи з центральним репозиторієм – це проект розробки ядра *Linux'a* (*http*://*www.kernel.org).*

В цьому випадку всі учасники проекту ведуть свої локальні розробки і безперешкодно викачують оновлення з центрального сховища. Коли необхідні роботи окремими учасниками проекту виконані і налагоджені, вони, після посвідчення власником центрального сховища в коректності і актуальності виконаної роботи, завантажують свої зміни в центральний репозиторій.

Наявність локальних репозиторієм також значно підвищує надійність зберігання даних, так як, якщо один з репозиторіїв вийде з ладу, дані можуть бути легко відновлені з інших репозиторіїв.

Робота над версіями проекту в *Git* може вестися в декількох гілках, які потім можуть з легкістю повністю або частково об'єднуватися, знищуватися, відкочуватися і розростатися в усі нові і нові гілки проекту.

Можна довго обговорювати можливості *Git'*а, але для стислості і більш простого сприйняття наведемо основні переваги і недоліки цієї системи управління версіями.

Переваги:

1. Надійна система порівняння ревізій і перевірки коректності даних, засновані на алгоритмі хешування *SHA1* (*Secure Hash Algorithm 1)*;

2. Гнучка система розгалуження проектів і злиття гілок між собою;

3. Наявність локального сховища, що містить повну інформацію про всі зміни, дозволяє вести повноцінний локальний контроль версій і заливати в головний репозиторій тільки повністю пройшли перевірку зміни;

4. Висока продуктивність і швидкість роботи;

5. Зручний і інтуїтивно зрозумілий набір команд;

6. Безліч графічних оболонок, що дозволяють швидко і якісно вести роботи з *Git'*ом;

7. Можливість робити контрольні точки, в яких дані зберігаються без дельта компресії, а повністю. Це дозволяє зменшити швидкість відновлення даних, так як за основу береться найближча контрольна точка, і відновлення йде від неї. Якби контрольні точки були відсутні, то відновлення великих проектів могло б займати години;

8. Широка поширеність, легка доступність і якісна документація;

9. Гнучкість системи дозволяє зручно її налаштовувати і навіть створювати спеціалізовані контролю системи або призначені для користувача інтерфейси на базі *git;*

10. Універсальний мережевий доступ з використанням протоколів *http*, *ftp*, *rsync*, *ssh* і ін.

Недоліки:

1. *Unix* – орієнтованість. На даний момент відсутня зріла реалізація *Git*, сумісна з іншими операційними системами;

2. Можливі (але надзвичайно низькі) збігу хеш – коду відмінних за змістом ревізій;

3. Не відстежується зміна окремих файлів, а тільки всього проекту цілком, що може бути незручно при роботі з великими проектами, що містять безліч незв'язаних файлів;

4. При початковому (першому) створенні сховища та синхронізації його з іншими розробниками, буде потрібно досить тривалий час для скачування даних, особливо, якщо проект великий, так як потрібно скопіювати на локальний комп'ютер весь репозиторій.

Git – гнучка, зручна і потужна система контролю версій, здатна задовольнити абсолютна більшість користувачів. Існуючі недоліки поступово віддаляються і не приносять серйозних проблем користувачам. Якщо ви ведете великий проект, територіально віддалений, і тим більше, якщо часто доводиться розробляти програмне забезпечення, не маючи доступу до інших розробникам

(наприклад, ви не хочете втрачати час при перельоті з країни в країну або під час поїздки на роботу), можна робити будь-які зміни і зберігати їх в локальному репозиторії, відкочуватися, перемикатися між гілками і т.д.). *Git* – один з лідерів систем контролю версій.

1.2.6. Система управління версіями *Mercurial*

Розподілена система контролю версій *Mercurial* розроблялася Меттом Макалу паралельно з системою контролю версій *Git*, створеної Торвальдс Лінус.

Спочатку, вона була створена для ефективного управління великими проектами під *Linux'*ом, а тому була орієнтована на швидку і надійну роботу з великими репозиторіями. На даний момент *mercurial* адаптований для роботи під *Windows*, *Mac OS X* і більшість *Unix* систем.

Велика частина системи контролю версій написана на мові *Python*, і тільки окремі ділянки програми, що вимагають найбільшої швидкодії, написані на мові Сі.

Ідентифікація ревізій відбувається на основі алгоритму хешування *SHA1* (*Secure Hash Algorithm 1)*, проте, також передбачена можливість присвоєння ревізій індивідуальних номерів.

Так само, як і в *git'*е, підтримується можливість створення гілок проекту з подальшим їх злиттям.

Для взаємодії між клієнтами використовуються протоколи *HTTP*, *HTTPS* або *SSH*.

Набір команд – простий і інтуїтивно зрозумілий, багато в чому схожий з командами *subversion.* Так само є ряд графічних оболонок і доступ до сховища через веб-інтерфейс. Важливим є і наявність утиліт, що дозволяють імпортувати репозиторії багатьох інших систем контролю версій.

Розглянемо основні переваги та недоліки *Mercurial.*

Переваги:

1. Швидка обробка даних;

2. Кросплатформенних підтримка;

3. Можливість роботи з декількома гілками проекту;

4. Простота в обіг;

5. Можливість конвертації репозиторіїв інших систем підтримки версій, таких як *CVS*, *Subversion*, *Git*, *Darcs*, *GNU Arch*, *Bazaar* і ін.

Недоліки:

1. Можливі (але надзвичайно низькі) вірогідності збігу хеш-коду відмінних за змістом ревізій;

2. Орієнтований на роботу в консолі.

Простий і відточений інтерфейс, і набір команд, можливість імпортувати репозиторії з інших систем контролю версій, – зроблять перехід на *Mercurial* і навчання основним особливостями безболісним і швидким. Навряд чи це займе більше кількох днів.

Надійність і швидкість роботи дозволяють використовувати його для контролю версій величезних проектів. Все це робить *mercurial* гідним конкурентом *git'*а.

1.2.7. Система керування версіями *Bazaar*

Bazaar – розподілена, що вільно розповсюджується система контролю версій, що розробляється за підтримки компанії *Canonical Ltd.* Написана на мові *Python* і працює під управлінням операційних систем *Linux*, *Mac OS X* і *Windows*.

На відміну від *Git* і *Mercurial*, створюваних для контролю версій ядра операційної системи *Linux*, а тому орієнтованих на максимальну швидкодію при роботі з величезним числом файлів, *Bazaar* орієнтувався на зручний і дружній інтерфейс користувача. Оптимізація швидкості роботи вироблялося вже на другому етапі, коли перші версії програми вже з'явилися.

Як і в багатьох інших системах контролю версій, система команд *Bazaar'a* – дуже схожа на команди *CVS* або *Subversion*, що, втім, не дивно, так як забезпечує зручний, простий і інтуїтивно зрозумілий інтерфейс взаємодії з програмою.

Приємно, що велика увага приділяється роботі з гілками проектів (створення, об'єднання гілок і т.д.), що дуже важливо при розробці серйозних

проектів і дозволяє проводити доопрацювання і експерименти без загрози втрати основної версії програмного забезпечення.

Великий плюс цієї системи контролю версій дає можливість роботи з репозиторіями інших систем контролю версій, таких як *Subversion* або *Git.*

Коротко наведемо найбільш суттєві переваги і недоліки цієї системи контролю версій.

Переваги:

1. Кросплатформенних підтримка;

2. Зручний і інтуїтивно зрозумілий інтерфейс;

3. Проста робота з гілками проекту;

4. Можливість роботи з репозиторіями інших систем контролю версій;

5. Чудова документація;

6. Зручний графічний інтерфейс;

7. Надзвичайна гнучкість, що дозволяє підлаштується під потреби конкретного користувача.

Недоліки:

1. Більш низька швидкість роботи, в порівнянні з *git* і *mercurial*, але ця ситуація поступово виправляється;

2. Для повноцінного функціонування необхідно встановлювати досить велика кількість плагінів, що дозволяють повністю розкрити всі можливості системи контролю версій.

Bazaar – зручна система контролю версій з приємним інтерфейсом. Добре підходить для користувачів, яких відштовхує перспектива роботи з командним рядком. Безліч додаткових опцій і розширень дозволить налаштувати програму під свої потреби. Схожість системи команд з *Git* і *Subversion*, і можливість роботи безпосередньо з їх репозиторіями, – зробить перехід на *Bazaar* швидким і безболісним. Про успішність базару говорить і той факт, що їй користуються розробники *Ubuntu Linux*.

1.3. Висновки до розділу

Великий вибір систем контролю версій дозволяє задовольнити будь-які вимоги і організувати роботу так, як вам необхідно. Однак, серед усього розмаїття систем є явні лідери. Так, якщо необхідно управляти величезним проектом, що складається з десятків тисяч файлів і над яким роботу ведуть тисячі чоловік, то краще за все вибір зупинити на *Git* або *Mercurial. Я*кщо для вас головне – зручний інтерфейс, а розробляється проект – не дуже великий, то для вас краща система *Bazaar*.

Для програмістів-одинаків або невеликих проектів, які не потребують розгалуження і створення безлічі версій, найкраще підійде *Subversion.*

Але, в остаточному підсумку, вибір – це справа смаку, так як зараз існує безліч систем контролю версій, які дадуть вам все необхідне. Так що вибирайте і не пошкодуєте. Системи контролю версій – це абсолютно необхідне програмне забезпечення для кожного розробника і не тільки.

На основі приведеного вище аналізу можна зробити висновки щодо необхідності формування реєстру програм на машинах користувачів і перевірки їх оповіщень.

Даний реєстр повинен бути розміщений на серверах з доступам через загальну або закриту мережу.

РОЗДІЛ 2
ПІДХІД ДО СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ ЧЕРЕЗ ПАРАДИГМУ ПРОГРАМНИХ ЕКОСИСТЕМ

Розробка програмного забезпечення веде до залучення сукупності аспектів, що впливають на всі етапи процесу розробки програмного забезпечення, як на організаційно-економічну, так і на мови програмування та середовища розробки. З кожним новим кроком до еволюції розвитку виникають нові фактори успіху, а отже, і нові проблеми зростають.

2.1. Співвідношення між екологією та програмним забезпеченням

Виберіть три основні причини застосування екологічного підходу до досліджень програмного забезпечення. Перші два стосуються концепцій сталого розвитку та визначають вплив програмних рішень та їх виготовлення на навколишнє середовище. Третя причина заснована на необхідності управління програмним забезпеченням, як системою, яка організована в контексті реального світу.

Перша причина заснована на припущенні, що програмні рішення в системах і технологіях та їх виробництво в галузі програмного забезпечення, як і в інших технічних секторах, впливають на навколишнє середовище. Такий вплив може бути шкідливим і не контролюватися.

Друга причина заснована на загальновідомому парадоксі розвитку науково-технічного прогресу, суть якого полягає в тому, що необхідно здійснити ліквідацію наслідків.

У такому випадку галузеве програмне забезпечення, як і будь-яка інша галузь людської діяльності, базується на точному та розумному операторі з наступних цілей – створити якомога менше відходів та якомога більше перевантажуватись цим, чим воно було не обов `язково

Третя причина заснована на спостереженні за тим, наскільки ефективне планування розробки та обслуговування програмного забезпечення вимагає розуміння не тільки їх місця в реальному світі, незалежно від встановленої ними взаємодії з реальним світом, як у межах єдиного продукту, але і безпосереднього в межах елементів продукту, але також і на дослідженнях з розширенням учасників та їх типами взаємодії, розумінням місця програмного забезпечення в історичному контексті.

Розглянемо застосування екологічних концепцій та принципів у програмному забезпеченні.

Дослідження використання термінів та принципів екології у галузі програмного забезпечення показують, що їх розподіл має три основні напрямки:

– Перший, пов'язаний із поширенням на програмному забезпеченні загальних принципів та вимог екологічного виробництва та використання засобів;

– Другий, що забезпечує економію ресурсів та виготовлення програмного забезпечення без відходів;

– По-третє, це незмінне використання науки для опису процесів, що відбуваються у розробці та використання програмного забезпечення

2.2. Програмна екосистема та система контролю версій як екосистема

Програмне забезпечення, використовуючи екологічний підхід, можна вивчати у двох аспектах – програмне забезпечення як частина екосистеми та програмне забезпечення, як екосистема.

Програмне забезпечення характеризується такими властивостями:

1. Зміна (розробка) – незамінний атрибут програмного забезпечення, завдяки наявності зворотних муфт та еволюції програм, пов'язаних із законами;

2. Наявність метасистеми, яка включає суб'єктів та дії, процеси та організацію, містить значну кількість відповідей, стабілізуючих внутрішні механізми, що впливають на процеси планування, контролю та підвищення їх

ефективності, ефективного планування та обслуговування, вимагаючи розуміння ролі в цій системі, а також взаємодії між елементами.

У цьому випадку розглянута програма, згідно з класифікацією *SPE*, є *E*-програмою, а метасистема – це їх зовнішнє середовище – реальний світ (рис 2.1).

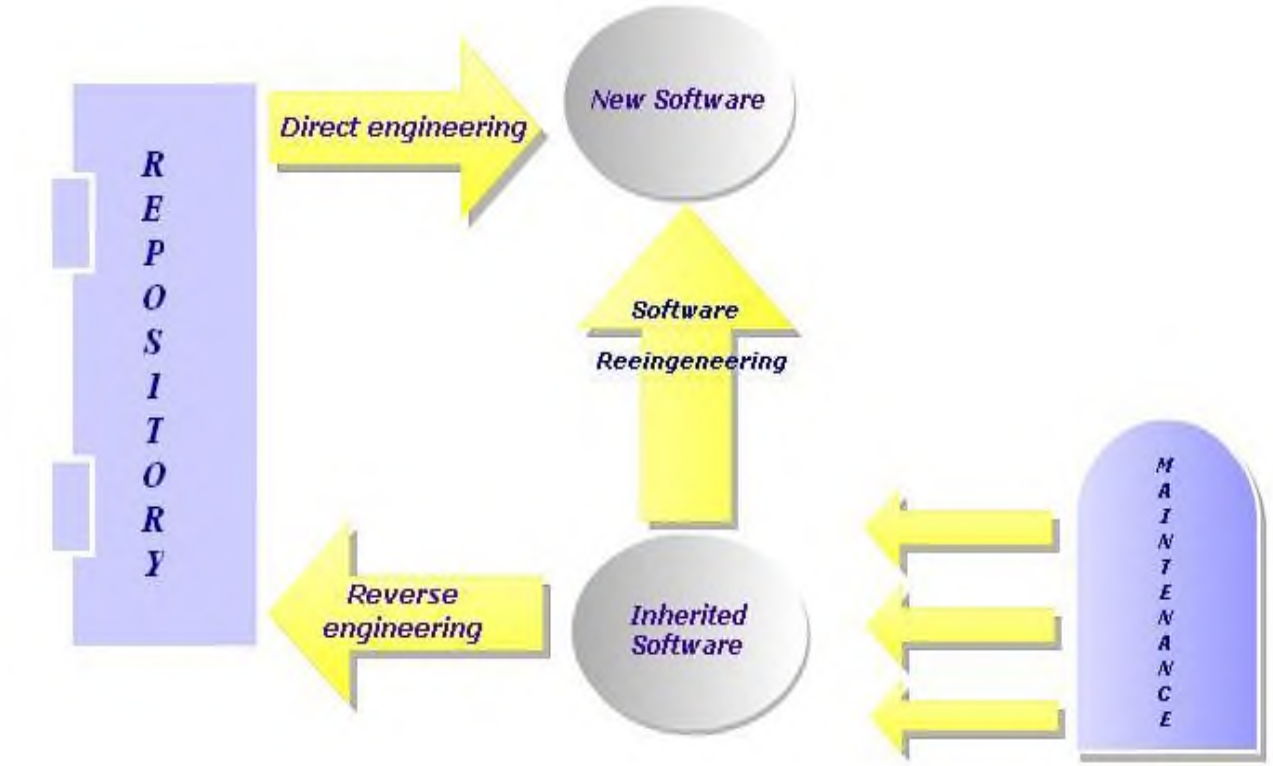

Рис. 2.1. Електронна програма в реальному світі

3. Якісне засвоєння оцінки програмного забезпечення – як частини екосистем;

4. Розуміння причин впливу на програмне забезпечення, зміни компонентів, джерел та наслідків;

5. Прогноз стабільності програмного забезпечення як частин екосистеми та прийнятність змін.

Як вже зазначалося вище, екологічний підхід до програмного забезпечення можна розглянути двома способами: враховуючи екосистему програмного забезпечення та враховуючи програмне забезпечення, як екосистему.

2.3. Визначення програмних екосистем

Система являє собою сукупність взаємодіючих або взаємозалежних компонентів системи формування єдиного цілого.

Загальні характеристики мають більшість систем, включаючи:

– системи мають структуру, визначену компонентами та їх складом;

– системи мають поведінку, яка включає введення, обробку та матеріальний вихід, енергію, інформацію або дані;

– системи мають узгодженість: різні частини системи мають між собою функціональні, а також структурні відносини.

– системи можуть мати деякі функції або групи функцій

Реалізація терміна "екосистема", який успадковує всі вищезазначені характеристики систем, у свою чергу збільшує їх можливості вивчення програмного забезпечення в продуктах та підтримці навколишнього середовища, відкриваючи їх у нових ракурсах: перспектива екосистеми, історичні перспективи та сталий розвиток перспективи.

Згідно з визначенням [5] стабільність має здатність тривати тривалий час. В екології цей термін описує, як біологічні системи залишаються різноманітними та продуктивними з часом. Тривалі та здорові заболочені місця та ліси є прикладами стабільності біологічних систем. Для людей стабільність – це потенціал для тривалого підтримання доброго самопочуття, яке має екологічний, економічний та соціальний аспекти. Стабільність безпосередньо взаємодіє з економікою через соціальні та екологічні наслідки ділової діяльності. Стабільність економіки включає екологічну економіку, де інтегровані соціальні, культурні, пов’язані з охороною здоров’я та грошово-кредитними та фінансовими аспектами.

Перехід до сталого розвитку – це також соціальна проблема, яка включає міжнародне та державне право, містобудування та транспорт, місцевий та індивідуальний спосіб життя та етичні потреби. Методи живуть більш стійко, можуть приймати різні форми від реорганізації умов життя (наприклад, екомістечок, екомуніципалітетів та сталого розвитку міст), завищення секторів

економіки (пермакультура, зелене будівництво, стабільне сільське господарство) або функціонування методи (підтримувана архітектура), розробка нових технологій (зелені технології, перезапущені джерела енергії) перед перестроюванням на окремий спосіб життя, що економить природні ресурси.

Представлення показників стабільності, а також економіки та суспільства обмежені обмежувальними умовами навколишнього середовища, представлено на рисунку 2.2.

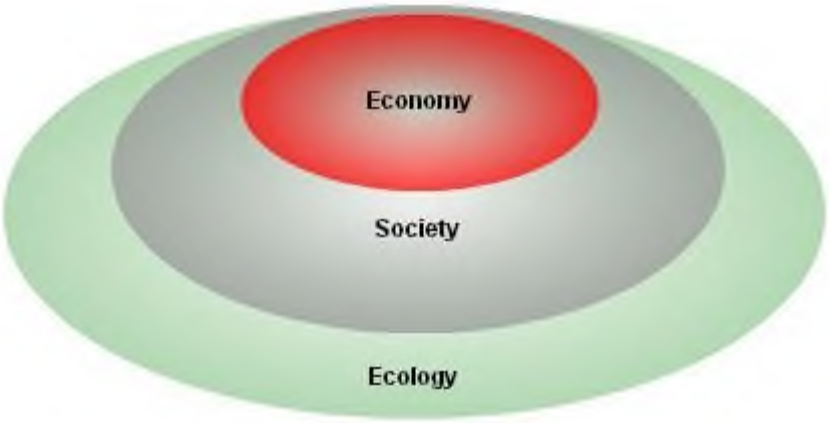

Рис. 2.2 Уявлення про стабільність, а також про економіку та суспільство обмежені екологічними межами

Стійкий розвиток (СР) – це зразок використання ресурсів, який спрямований на задоволення потреб людини, збереження навколишнього середовища, щоб ці потреби могли бути задоволені не лише нині, а й у майбутньому, наступних поколінь. Цей термін був використаний Комісією Брундтланд, яка визначила найбільш часто цитоване визначення сталого розвитку як розвиток, який «задовольняє потреби нинішнього покоління без шкоди для можливості майбутніх поколінь задовольняти власні потреби» [5].

Сталий розвиток пов’язує безпечне навантаження на природні системи із соціальними проблемами, з якими стикається людство. У 1970-х роках слово «стабільність» використовувалося для визначення, як економіка «в рівновазі з основними системами екологічного забезпечення». Екологи зазначили в «межі зростання» та представили альтернативи «економіки сталого стану» з метою вирішення екологічних проблем.

Зона сталого розвитку можна концептуально розділити на три компоненти: підтримка екологічної стабільності, економічної стабільності та соціально-політичної стабільності (рис. 2.3) [8].

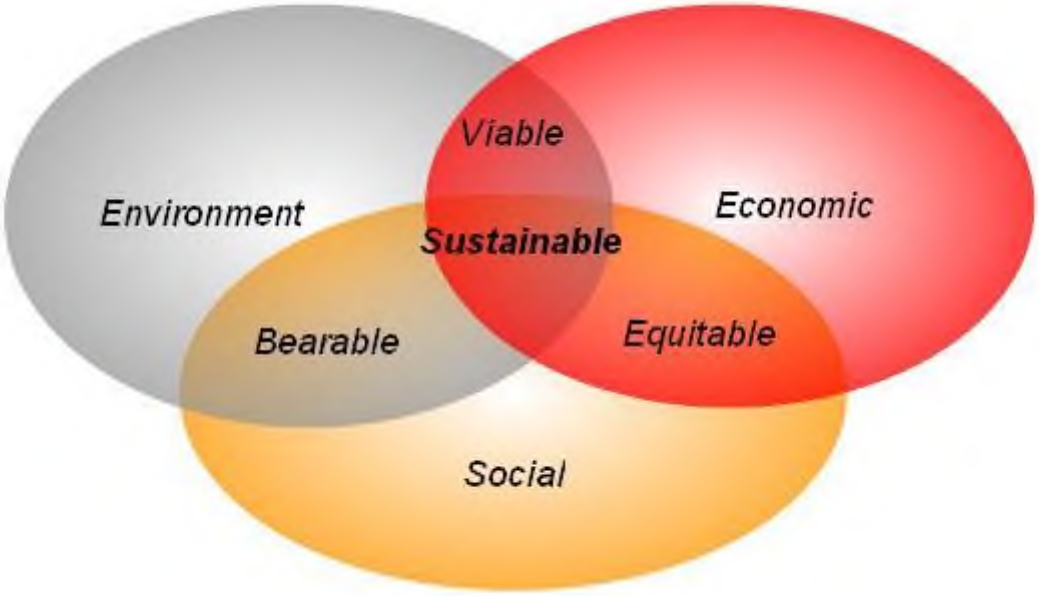

Рис. 2.3 Схема сталого розвитку: об’єднання трьох компонентів

Таким чином, можна зробити висновок, що підхід екосистем до програмного забезпечення є екологічною основою концепції систем [6].

У довгостроковій перспективі екосистеми щодо реального світу – екосистеми, однією з об’єктів яких є програмне рішення, обмін енергією та питання функціональних та соціальних комунікацій процесу навчання.

Можна зауважити, що соціальний аспект також успадкований від концепції системи в підході до екосистеми і є однією із складових стратегії сталого розвитку (рис. 2.4).

2.4. Проблеми взаємодії в програмних екосистемах

Давайте розглянемо аспекти взаємодії, їх роль та вплив на програмні екосистеми.

Створення програмного забезпечення – це багатогранна діяльність. Існує теза, що створення програмного забезпечення є синонімом програмування.

Приблизно схоже на те, що створення фільмів є синонімом кінофільму аудіо та монтажу відео. Як створення фільмів, так і створення програм включає безліч додаткових дій, таких як: оцінка можливостей та прийняття інвестиційного рішення (подібно до продюсера фільму), розробка деталей, які будуть забезпечені функціями та можливостями (сценарист), кваліфікація та вдосконалення ідей із реальними користувачами (допитувані фокус-групи), створення архітектури (раскадровка – сценарний відділ кіностудії), контроль численних груп програмістів (режисер), реалізація програмного забезпечення за допомогою програмування (кінематографія та монтаж), тестування (друк попередній перегляд),

Давайте розглянемо загальні питання, які суттєво впливають на кінцевий успіх у створенні програмного забезпечення. Слідом за ним розглядається повна організація процесу створення програмного забезпечення, включаючи послідовний, повторюваний та публічний підходи.

Чудові результати в галузі основної електроніки, магнітного накопичувача та технологій волоконної оптики показані в оперативному усуненні будь-яких фізичних перешкод, яких може досягти програмне забезпечення.

Розробка програмного забезпечення стає дедалі вільнішою для концентрації на отриманні максимальних переваг від програмного забезпечення для людей, організацій та суспільства. Безумовно, це включає економічні інтереси постачальника програмного забезпечення, але є також безліч інших причин та можливостей.

Залишатися на піку передових досягнень у галузі розробки програмного забезпечення та контролю над цією розробкою є важливим для постачальника, що є цілком зрозумілим.

Крім цього, у світлі сили, яку спостерігає закон Мура, успіх розробки програмного забезпечення вимагає уважного відношення у чотирьох питаннях [1]:

– прийняття уваги користувача та оператора. Програмне забезпечення, що пропонує велику цінність для загальних користувачів, приносить вигоди для

постачальників та ще більше переваг для користувачів та суспільства. Вирішальне значення для успіху мають як оператори, так і користувачі;

– прийняття до уваги того, що вже існує. На практиці прогрес істотно досягається додаванням замість повної заміни низьких технологій та програм. Оптова зміна інфраструктури або програми (або одночасно всіх разом взяті) часто не є варіантом того, що вищезазначений ризик, тим більше порушень, а відповідно і бажання клієнта продуктивно використовувати попередні інвестиції. Тому постачальники програмного забезпечення повинні пропонувати рішення, які будуть життєздатними в контексті користувача, що, в свою чергу, означає сумісність із спадщиною та програмним обладнанням. Зміни повинні бути поетапними, рухатись вперед, не спричиняючи надмірного капіталу та інвестицій у розвиток або довільного виконання статусу-кво;

– прийняття до уваги функціонування ринку. Ринкові операції з програмним забезпеченням та додатковим виробництвом (наприклад, обладнання та мережа), а також відповіді постачальників суттєво впливають на результати. Необхідна стійка увага та спостереження за конкурентоспроможними фірмами, що практично завжди є розумним рішенням багатьох проблем. Постійне спостереження за конкурентоспроможними фірмами є невід'ємною частиною, оскільки один постачальник рідко коли може запропонувати повне рішення завдання. Інші важливі операції ринку включають розповсюдження до нової епохи Інтернету (даний розділ), стандартизацію, організацію промисловості та, з часом, розділи інтелектуальної власності та контролю за правами, а також різні фактори, пов'язані з економікою;

– прийняття уваги колективного благополуччя. Розробники повинні вміти розпізнавати можливі позитивні (і негативні) ефекти програмного забезпечення як на наш колективний добробут, так і на суспільство в цілому, а також працювати, підкреслюючи позитивне і пом'якшуючи негативне. Проблеми, пов'язані з соціальною сферою: недоторканість приватного життя, правоохоронних органів, державна безпека, право інтелектуальної власності та підтримка конкурентного ринку. Це в подробицях свідчить не тільки про хороші стосунки громадян, але і одночасно є хорошим бізнесом. Неуважне ставлення до

вивчення цих питань може завдати прямих економічних збитків постачальникам, що, у свою чергу, провокує на дії політиків, які, швидше за все, не приносять прихильності галузі.

Створення нового програмного рішення часто включає численні команди експертів, що застосовують додаткові навички з великими витратами на координацію.

Особливо при застосуванні програмного забезпечення розробка успішного продукту вимагає тісної координації з користувачами та можливими клієнтами (включаючи менеджерів та операторів) та ефективного контролю численних команд програмістів. Таким чином, ефективне створення програмного забезпечення – це організаційне завдання та завдання контролю, що стосуються технічного питання.

Звернемо свою увагу на суспільний (соціальний) розвиток. У цій моделі слабко пов’язана команда розробників програмного забезпечення – це люди, які співпрацюють над програмним проектом, використовуючи Інтернет та спільні інструменти в режимі онлайн. Часто ці розробники не належать до певної організації або навіть просто є звичайними волонтерами.

Основою соціального розвитку є вихідний код, доступний певному колу споживачів; з часом, звичайно, код стає доступним кожному, хто його цікавить. Більшість комерційних представників програмного забезпечення не надає права доступу до кодів стороннім особам, які не входять до складу даної організації, оскільки це прирівнюється до комерційної таємниці. Створення доступного вихідного коду є новим, по суті, протилежним методом у традиційній комерційній практиці.

Термін з відкритим вихідним кодом описує конкретний тип, заснований на спільноті розробників, і пов’язані з ним умови ліцензування, як торговельна марка *Open Source Initiative (OSI)*, яка з ентузіазмом описує відкритий код наступним чином: дуже просто. Коли програмісти читають, перерозподіляють і змінюють вихідний код програмної частини, – розробляється програмне забезпечення. Люди вдосконалюють його, адаптують і виправляють помилки. І це може трапитися зі швидкістю, яка, якщо ви звикнете повільно стандартна

розробка програмного забезпечення, здається дивним". Існує багато можливих умов ліцензування, таких самих, як різні спонукання зробити вихідний код доступним.

Отже, з вищесказаного можна підсумувати: соціальний аспект глибоко забезпечує і підтримує стійкий розвиток програмного забезпечення [1].

Розглядаючи розвиток, заснований на співтоваристві, слід зазначити мережевий ефект.

Для більшості вартість програмного забезпечення залежить не тільки від функцій інтер'єру та можливостей програмного забезпечення, а й від ряду інших рішень, що відповідають сумісності «батьків, що приймають»: чим більше «батьків, що приймають», тим вище значення. Це називається ефектом мережі або зовнішніх мереж, і вона відіграє важливу роль на деяких ринках програмного забезпечення.

Зовнішній вигляд – ситуація, коли дії однієї зі сторін впливають на дії іншої, без грошової компенсації (наприклад, забруднення повітря). Зовнішні ефекти мережі можуть бути як позитивними (тут наведена пріоритетна форма), так і негативними. При негативних зовнішніх факторах цінність фактично зменшується, а не збільшується при значній кількості "батьків, які приймають".

Позитивні зовнішні мережі в технологічних продуктах випускаються у двох чудових формах, як показано на рисунку 2.4. Припускаємо, що в руках різних «батьків, що приймають» є кілька примірників товару. Коли ці випадки не мають ніякого відношення, немає мережі і немає мережевого ефекту. Коли вони певною мірою залежать, мережа закінчується. (Це не означає реальну мережу зв'язку, що їх з'єднує, і, швидше за все, просто мережу додаткових залежностей.) У більш сильному прямому мережевому ефекті різні копії продукту відразу додають одна одну, і цінність кожного окремого «батьківського одержувача» зазвичай збільшується із збільшенням розмір мережі. Зокрема, перший батьківський одержувач не може отримати будь-яке значення [1].

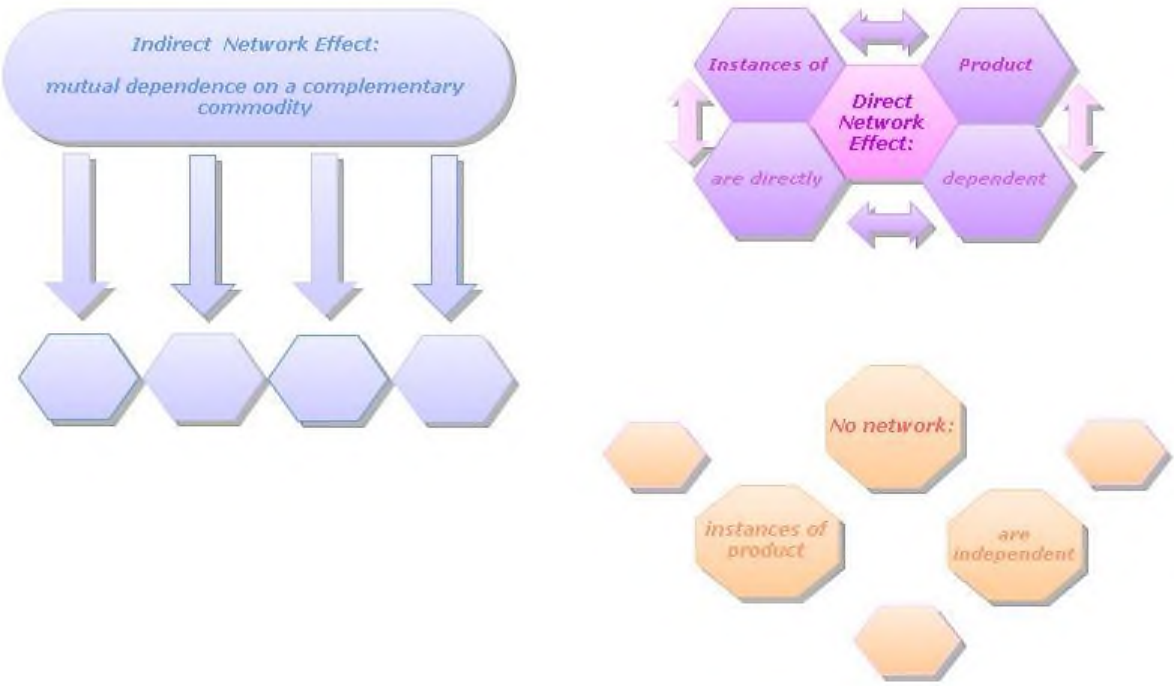

Рис. 2.4. Мережевий ефект в програмних екосистемах

Тепер давайте підключимо екосистеми програмного забезпечення та стабільність зростання. Наступна схема відображає основну ідею на високому рівні.

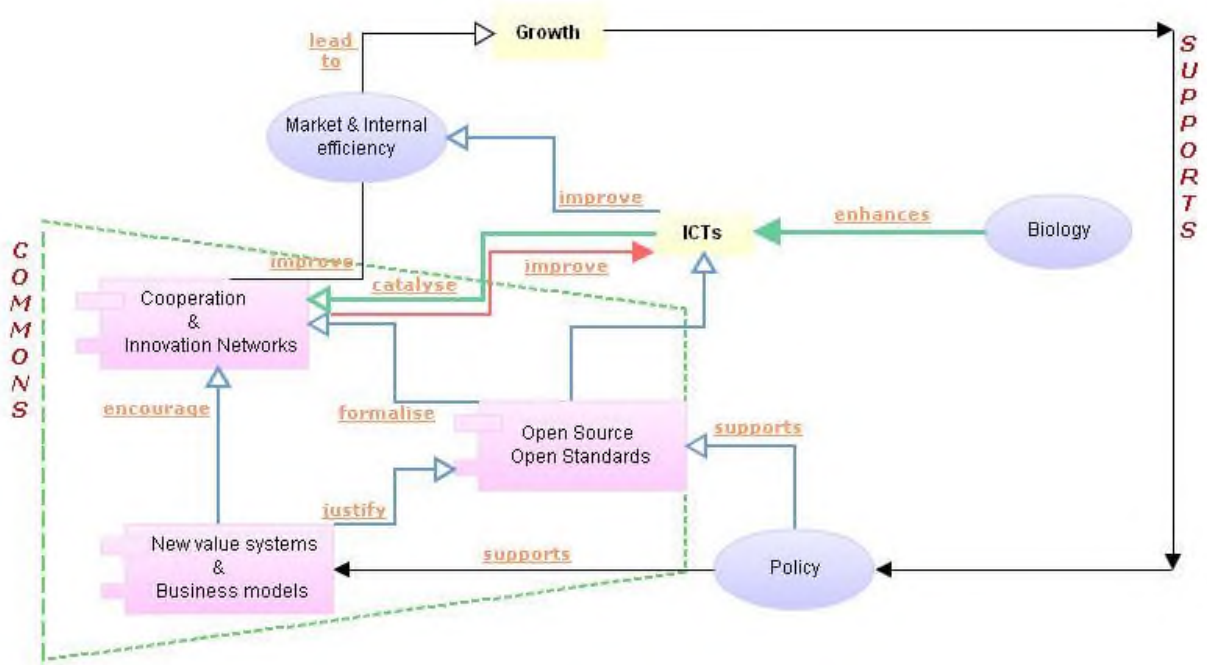

Рис. 2.5 Структурна схема зростання ІКТ

У випадку програмних екосистем, успіх або занепад будуть дуже очевидними, оскільки екосистеми або виживають і виростають за період фінансування проекту, або загинуть, можливо, замінившись якоюсь іншою формою технології, яка може бути власною та змусити фактичні стандарти замінити спочатку надані. Що стосується стратегії, технічно рання адаптація напруги та фінансування можуть залишити себе відкритими для ризику, але ми нарешті підтримали та "програли" та "перемогли". Однак підтримка ранніх спонсорів екосистеми з фінансуванням з метою розвитку громади до досягнення комерційного успіху приводить до обґрунтування підтримки існуючої екосистеми, пропонуючи альтернативну модель [6].

Давайте розглянемо просування ланцюжка створення вартості від наукових досліджень до кінцевого користувача.

Розуміння різноманітності способів сприйняття витрат і перетворення їх у кожну точку цієї схеми – це все процес, який з часом має розвинутися. На практичному розгляді пройдений тест підтверджує цінність нових і встановлених ідей дослідження, що являє собою повторюваний процес, що включає перехресну передачу через самих різних учасників проекту в проектах та ініціативах. Основу емпіричних досліджень склав Цифровий кластер екосистем (Кластер цифрових екосистем), і на його основі визначено багато ключових сфер, що викликають занепокоєння, як важливих для всіх учасників проекту – від МСП, великим корпораціям та академічним установам.

Основна мета прикладного програмного забезпечення полягає у задоволенні потреб кінцевих користувачів, будь то окремі особи, групи осіб, організації (наприклад, компанії, університети, уряди) групи організацій (наприклад, торгівля), громади на інтереси або суспільства в цілому (наприклад, розваги, політика). Прикладне програмне забезпечення відрізняється від інфраструктурного програмного забезпечення тим, що друге негайно подається розробникам додатків і лише у другій черзі для користувачів. Більш повне визначення програми наступне: це ряд функцій і можливостей (також їх називають функціями), що утворює цілісне ціле, має якості інтеграції (наприклад,

модель колективної безпеки) і призначений для характерної сфери діяльність людини. Ця область, у своїх екстремальних значеннях,

Важливо зазначити, що користувачі часто не є клієнтами продуктів програмного забезпечення. У певних випадках вони є і тим, і іншим, як з операційною системою та додатками, придбаними для домашнього додатку. Зазвичай в організації є посередники, які отримують та розгортають програмне забезпечення та контролюють його від імені користувачів як клієнтів. Те саме стосується додатків, які отримують доступ до мережі.

Хоча розробка програмного забезпечення має репутацію ізоляції та асоціальних дій, це упередження все більше і більше стає застарілим та неточним, особливо у випадку прикладного програмного забезпечення. Успішне прикладне програмне забезпечення закінчується, коли розробники, які його розподіляють, безпосередньо ставляться до контексту та думають про майбутніх користувачів. Розробник прикладного програмного забезпечення повинен працювати у тісній співпраці з майбутніми користувачами, а не просто уявляти та розуміти їх суть. Уявіть, будинок, ландшафтні архітектори, котрий гардеробні інтер'єрів не можуть широко взаємодіяти з майбутніми орендарями, котрі повинні жити зі своїми рішеннями невизначений час. Їм це буде неможливо, і будь-який забудовник, що працює в ізоляції, такий план роботи не буде успішно виконаний [1].

Інший метод класифікації програм характеризується як група користувачів, яку вони підтримують. Окремі програми вдосконалюють ефективність або продуктивність. Групові додатки обслуговують групу людей і підкреслюються традиціями співпраці та базою даних. Потім є додатки, які обслуговують організації (групи, що виконують довгострокові, колективні цілі.) Заявка на спільноту дозволяє численним групам людей, які поділяють загальну програму чи інтереси, взаємодіяти або координуватися, навіть таким чином, що вони, ймовірно, не знати один одного або динамічні зміни членства. Корпоративні програми підтримують організаційні завдання (комерційні, освітні чи державні) та основні процеси, що відбуваються в організації. Вони бувають двох типів: програми бізнес-процесів, які автоматизують повторювані завдання, та

адміністративні програми, які підтримують прийняття спеціальних рішень. Застосування електронної комерції дозволяють двом або більше організаціям виконувати колективні функції, особливо там, де в процес входить продаж товарів або послуг. Застосування електронної комерції прирівнюється до корпоративних додатків, як. групова заявка прирівнюється до індивідуальної заявки.

В якості нової галузі IT були додані нові програми, які зазвичай включають можливості як нові, так і попередні технології (як це проілюстровано базою даних та публікаціями на рисунку 2.6) [1]. Таким чином, сьогодні багато програм чітко поєднують обробку, масове зберігання та передачу різними методами, і це стає все більш і більш актуальним для програм у майбутньому.

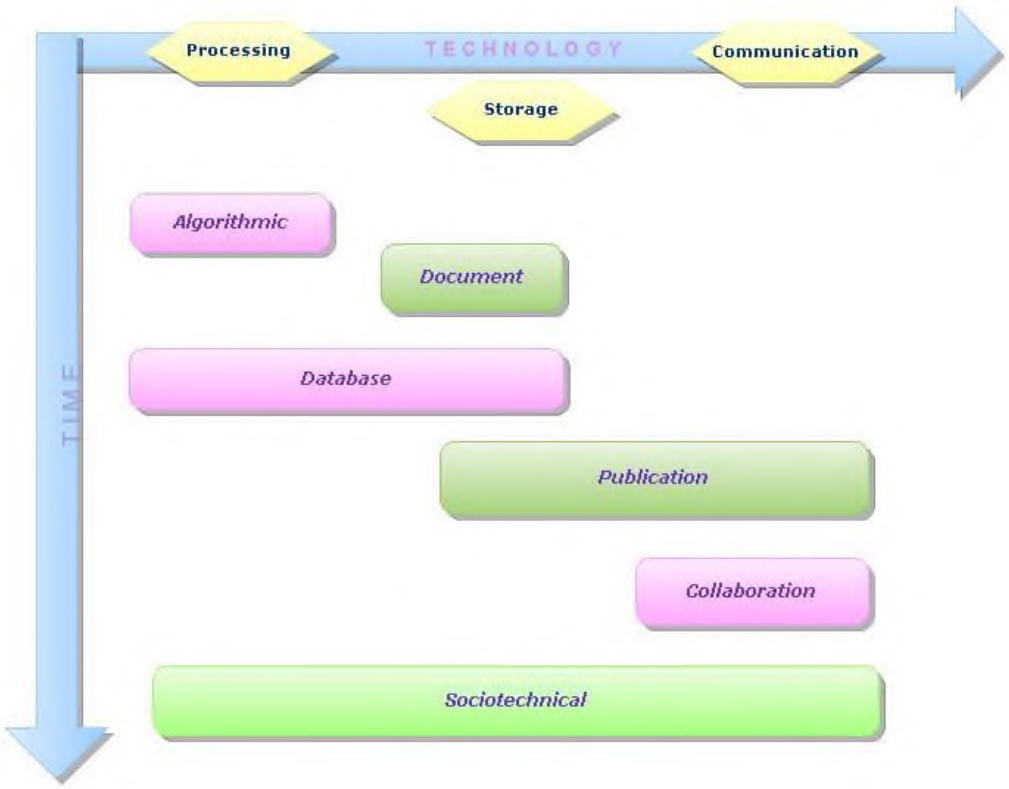

Рис. 2.6. Класифікація програмного забезпечення як зовнішнього вигляду та використання основних технологій [1]

Звернемо увагу на таке поняття, як соціально-технічний аспект застосування.

Традиція об’єднує групи людей, які працюють над:

– колективні завдання,

– *IT (*обробка, зберігання та передача),

– обсяг інформації та знань (останнє в основному стосується людей),

– багато інших фізичних елементів (матеріали, товари та доставка).

Усі традиції застосування інтегровані в ці програми, які часто мають складний характер. У буквальному сенсі це кульмінація традицій застосування, принаймні за оцінкою на сьогодні.

У соціально-технічних додатках *IT* є лише частиною соціально-технічної системи, яка включає людей та їх організації. *IT* компонент не можна поступово визначати або вивчати в ізоляції від інших елементів; він вбудований як невід'ємна частина системи, оскільки фірмовий посуд є невід'ємною частиною обладнання. Аспект таких додатків вимагає розподілу функцій між технологіями та людьми, організаціями людей та технологій та відповідним інтерфейсом між технологіями, людьми та організацією. Проект соціально-технічних додатків повинен бути більшим, ніж попередні програми – міждисциплінарні дії, включаючи технологів з організаційними експертами та соціологами [1].

Існує кілька більш амбіційних підходів до визначення соціально-технічних застосувань:

– автоматизація. Візьміть існуючий процес та автоматизуйте його. Зазвичай він включає етапи переходу від процесу фізичної роботи або заміни паперу до автоматизованих механізмів, але без істотних змін до основних процесів$

– реінжиніринг. Не змінюючи фундаментальних цілей або функціональних можливостей, повернувшись до баз, зосередьтесь на універсальних цілях і запитайте, як би ці цілі були кращими, розподіляються між людьми та їх організаціями та *IT,* і це буде найкращою формою взаємодії серед них елементів. Розглянути і розробити всі елементи соціально-технічних систем з нуля без суттєвої зміни цілей.

– інновації. Зверніться до нових цілей та функціональних можливостей, які неможливі без IT.

Розглядаючи питання кореляції, необхідно виділити відмінності як у наукових інтересах, так і в галузевих інтересах, і в різних точках зору. Малі та

середні підприємства (МСП) шукали бізнес-рішень більше, ніж певні технологічні підприємства, і це велика проблема підтримки та представлення досліджень, яка повинна проводитися з точки зору бізнес-застосувань та переваг, а не необроблених технологічних досягнення. У чіткому підході "нехай галузь контролює дослідження" існує небезпека, що галузь буде зосереджена лише на короткотермінових цілях і працюватиме в межах відомих парадигм. Важлива роль наукового співробітника полягає у вказівці на виробничі речі та забезпеченні того, що раніше ніколи і про що не мріялося.

Науці не потрібно боятися вести галузь. Однак наука заради себе – це більше, ніж область академічних закладів.

Проекти повинні базуватися на реалістичній можливості досягнення певного бізнесу, покращуючи як ці інновації, так і аспекти технічного обслуговування повинні бути важливою частиною загального плану проекту. У чисельних наукових дослідженнях екосистем результати вимагають обережного підходу. Центральна метафора "екосистеми" переважає натхнення з біології, але є також точки контакту з фізикою та математикою. Результати науки, які часто сприймаються як цікаві, але "занадто інтелектуальні", також важкі для розуміння та управління учасниками проекту різних дисциплін дослідження.

У галузі екосистемного програмного забезпечення не запатентована інтелектуальна власність не спотворює конкуренції і повинна розглядатися як нейтральна щодо конкурентоспроможності (за винятком тих компаній, які прагнуть домінувати на ринку). Таким чином, чисельні екосистеми можуть включати програмування програм, демонстрації та навіть підтримку на ранніх стадіях спільнот користувачів за умови, що вони відкриті та абсолютно нейтральні. Існують такі технології, як Інтернет, де лише в результаті раннього та очевидного використання відбувається реальне технічне обслуговування. Таким чином, раннє використання та знання можуть стати фактичною передумовою для швидкого та всебічного прогресу технологій. [6].

Наступне питання – це обмін знаннями в цифрових екосистемах. Оскільки впровадження революційної технології включає безліч різних гравців, деякі з яких не можуть бути включені в технологічний аспект, такі як законодавці,

комунікація цих гравців та діалог є важливими для перегляду на рівні інфраструктури для вироблення шляху до скоординованого впровадження нової технології, замість зіткнення з ненавмисними бар'єрами. Хоча чисельні екосистеми матимуть спільноти користувачів, може бути вигідним забезпечити більш розподілені методи для обміну знаннями серед широкого загалу, які впливають.

У числових екосистемах загальні знання, загальні моделі та навчальні одиниці розглядаються як форма людського капіталу, яка була накопичена, сформована та побудована в цих екосистемах. Обмін ідеями дозволяє проводити спільне програмування додатків, але також дозволяє проводити побудову сумісних прагнень та загального сприйняття реальності, формуючи загальну ідентичність даних та полегшуючи зобов'язання на рівні "громади". Технологія вивчення відкритого вихідного коду, в рамках якої учасники МСП можуть ділитися та розвивати ідеї, послуги щодо складу та потенційних бізнес-сценаріїв, формує постійно зростаючий ресурс, який вбудований в інфраструктуру чисельної екосистеми [6].

2.5. Побудова моделі прикладно-орієнтованої програмної екосистеми

Переглядаючи таку екосистему, можна виділити такі об'єкти та предмети:

– платформа додатків (програмне забезпечення) – власне програмне забезпечення, розроблене компанією;

– розробник – компанія, яка ініціювала розробку програмного забезпечення, має права, здійснює підтримку;

– зовнішні розробники – суб'єкти, які можуть бути партнером розробника програмного забезпечення у створенні програмного забезпечення або можуть створювати окремі блоки, які розширюють основні функціональні можливості програмного забезпечення.

– клієнт – користувач програмного забезпечення.

– основна функціональність – функціональність, що надається додатком.

– додаткова функціональність – функціональність, додана до ядра.

– інтерфейси, середовища розробки, засоби розробки – ресурси для реалізації доповнень до основної програми.

– додаткові вимоги – вимоги, що стимулюють розвиток додаткової функціональності.

Розберемо компоненти та відношення між ними, як і в попередньому випадку:

$$S = (C, R),$$

де C – непустий набір компонентів,

R – непустий набір відношень.

Тоді C і R можна представити наступним чином:

C = (Платформа додатків (програмне забезпечення), Компанія розробник, Зовнішні розробники, Клієнт, Основна функціональність, Додаткова функціональність, Інтерфейси, Середовища розробки, Інструменти розробки, Додаткові вимоги), $C \subset$ C,

R = (взаємодії, розширення, підтримка, створення, надання, розуміння, використання), $R \subset$ C.

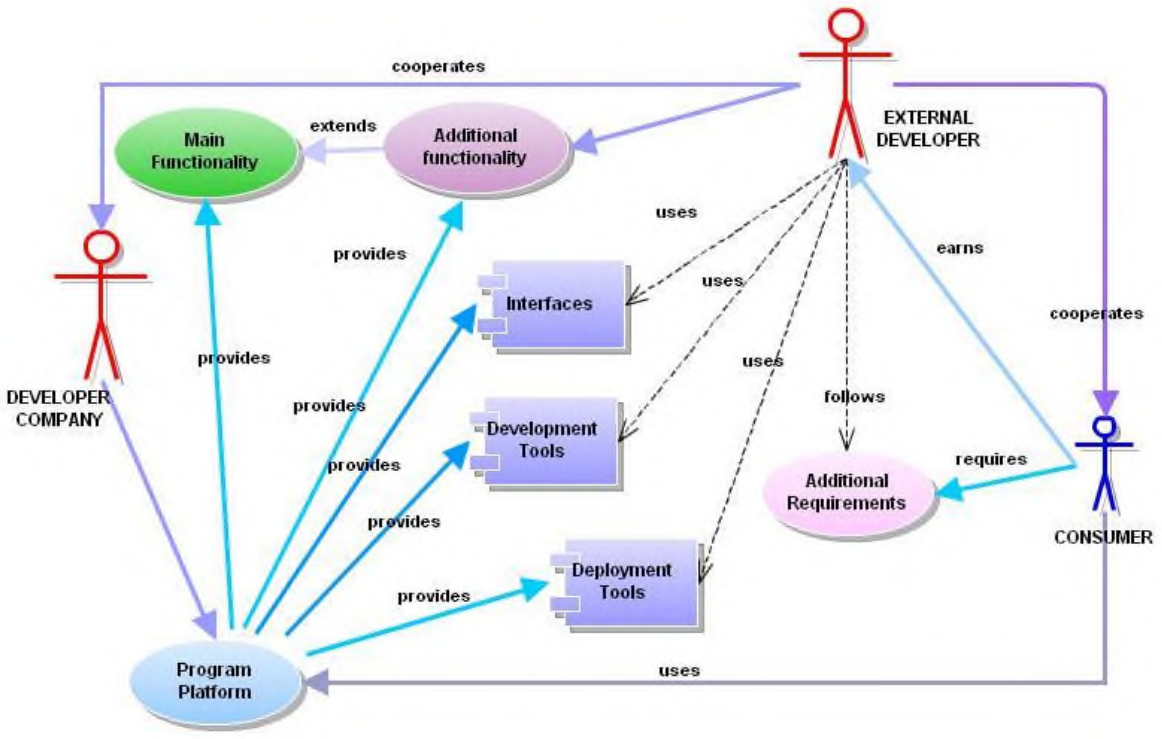

Рис. 2.7 Екосистема програмного забезпечення, орієнтованого на додатки

Ця модель екосистеми програмного забезпечення не є великою за розміром, але дає можливість для зміни підходу при перегляді цієї екосистеми. Насправді можна додати користувача актора, як і в попередній моделі, але це призводить до надмірності.

У цій екосистемі з'являються нові об'єкти. Це інтерфейси, інструменти розробки, середовище розробки. Зовнішній вигляд цих об'єктів забезпечується особливостями, описаними в попередніх розділах.

Є також деякі проблеми, з якими можна зіткнутися:

– основною проблемою є боротьба компаній розробників платформи додатків проти конфлікту між стратегією продукту і стратегією платформи. Спочатку компанія досягає першого головного успіху, який фокусує всю компанію на товарній стратегії. Якщо потрібно перейти до стратегії, спрямованої на платформи, між обома напрямками існують відмінності. З точки зору стратегії спрямованості продукту, стратегія платформи не контролює нове значення для клієнтів, оскільки в основному дозволяє непрямим розробникам визначати ці значення.

– свобода змін *API*, користувальницьких інтерфейсів, моделей даних тощо значно обмежена стратегією платформи, і особливо спочатку компанії-розробники платформи можуть сприяти появі помилок, оскільки відносний пріоритет стратегії платформи все ще недостатньо надійний.

Принаймні, не підтверджена інформація з блогів та статей новин передбачає, що створення діючої бізнес-моделі для непрямих розробників доводить непомітну мету, яку важко досягти для більшості екосистем програмного забезпечення. Для екосистем прикладних програм є додаткова проблема – платформа для додатків повинна бути достатньо корисною для клієнтів, щоб її розпізнавали. Через це кількість клієнтів, які активно сподіваються розширити платформу за допомогою додаткових функціональних можливостей, може бути обмежена [7].

2.6. Побудова моделі специфічної для кінцевого користувача екосистеми

Ця екосистема програмного забезпечення є трохи більш конкретною, ніж інші. Обираємо такі об’єкти та предмети:

– програмне рішення – основне програмне забезпечення;

– доменна мова – мова, надана програмним забезпеченням;

– *User*_1 – користувач програмного забезпечення, який самостійно приймає рішення для власних потреб.

– *User*_2 – користувач програмного забезпечення, який використовує рішення, надане *User*_1.

– додатки клієнтів – сукупність рішень, створених користувачами.

– спільнота клієнтів – група користувачів, яка використовує початкове програмне забезпечення, і створюється власною.

Визначимо програмну систему наступним чином:

$$S = (C, R),$$

де C – непустий набір компонентів,

R – непустий набір відношень.

Тоді C і R можна представити наступним чином:

C = {програмне рішення, мова домену, *User_1*, *User_2*, програми клієнтів, спільнота клієнтів}, $C \subset$ C;

R = {Підтримка, створення, аксесуар, використання}, $R \subset$ C.

Спільнота клієнтів, програми клієнтів – це підмножини, так що Спільнота клієнтів $a \subset$ C, Застосування клієнтів $\subset C$.

Припустимо труднощі, з якими можна зіткнутися в цій екосистемі:

– найбільша проблема, безумовно, полягає в моделюванні предметного домену програми з урахуванням того, що кінцевий користувач може бути збитий з додатком з мінімальними зусиллями та зробити його створення інтуїтивно зрозумілим і вимагаючи мінімальних;

– багато предметних доменів, просто занадто складно створити предметно-орієнтовану мову і підключену конфігурацію, яка відповідає вимогам досить великої кількості кінцевих користувачів.

– територія, в якій створюється екосистема кінцевого споживача, вимагає достатньої стійкості до неї, що дає можливість низьких витрат на обслуговування [7].

2.7. Поведінка програмної екосистеми

Тепер давайте спостерігатимемо за поведінкою екосистем на випадок різних змін.

Ми беремо екосистему, засновану на операційних системах, як зазначено на рис. 2.8. (Увімкнення *OEM)*, ми також припускаємо, що певний тип пристрою вже не виготовляється виробником, але все ще необхідний на заводі (рис. 2.9).

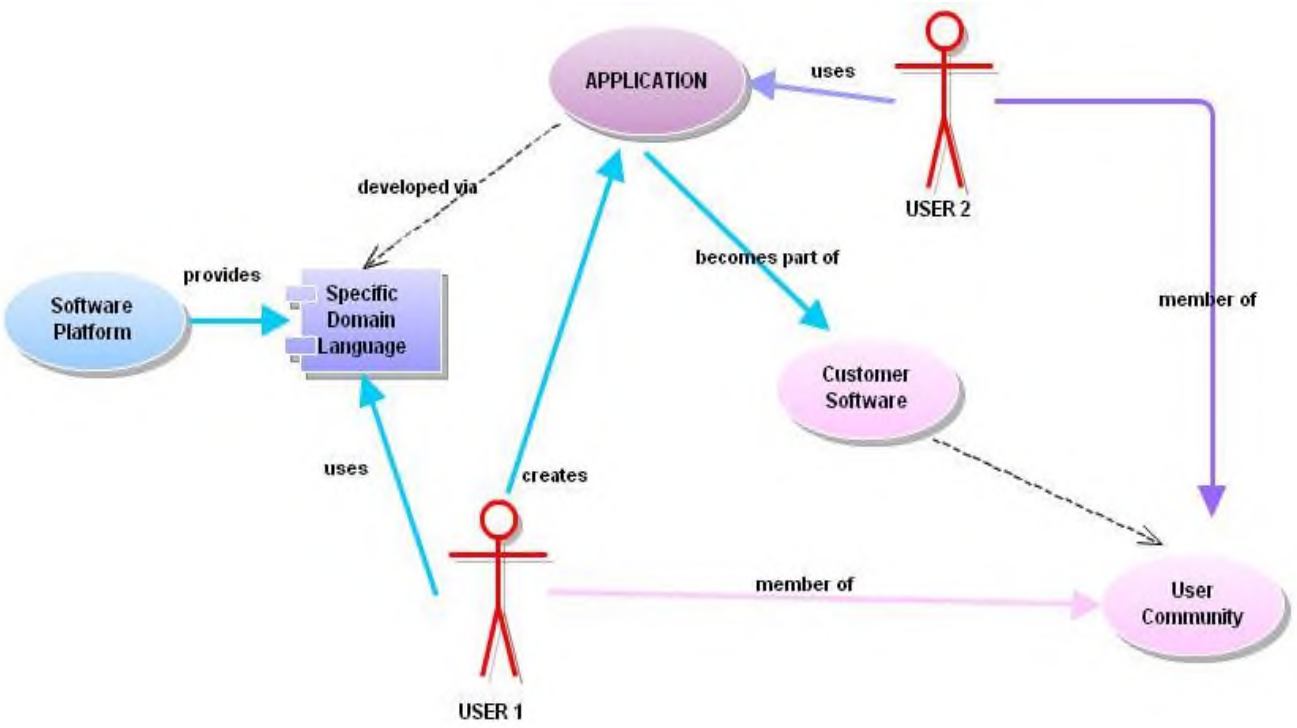

Рис. 2.8. Екосистема програмного забезпечення, орієнтована на кінцевих користувачів

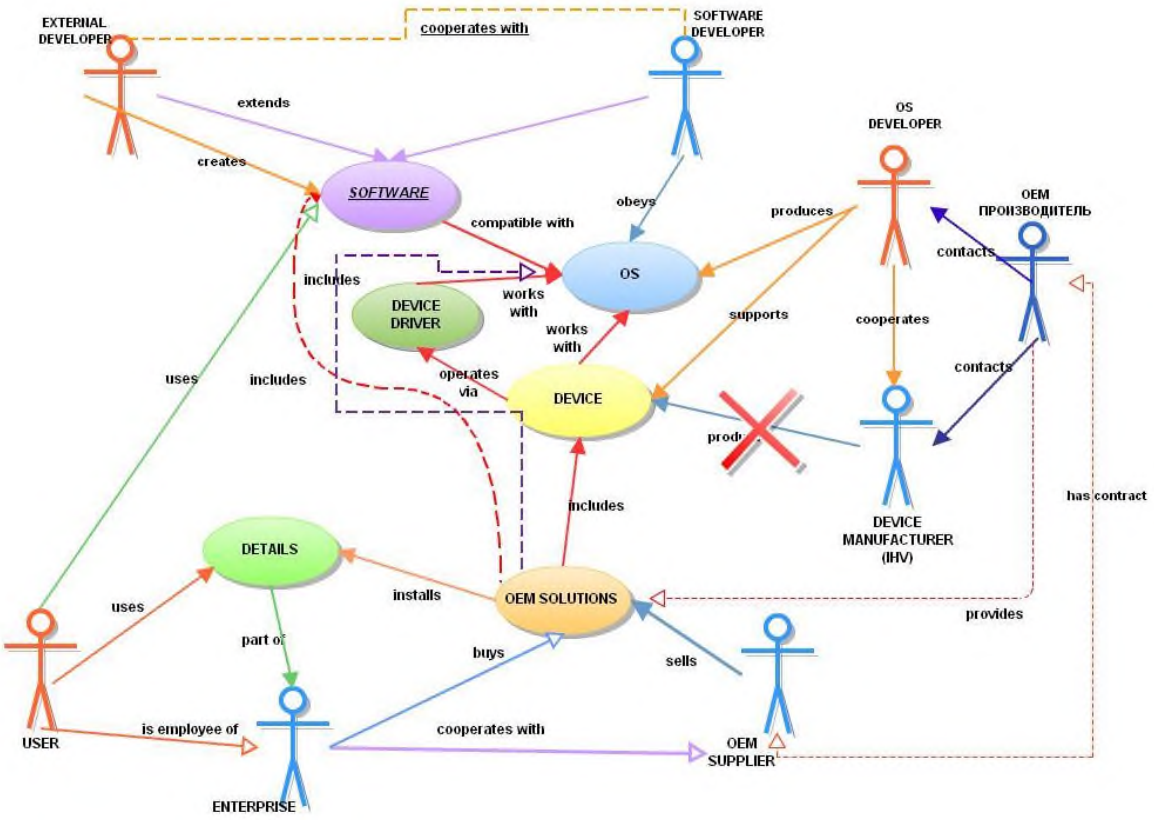

Рис. 2.9. Програмна екосистема з відсутністю певного типу пристроїв

Давайте подивимось, що відбувається далі. Якщо немає доступних затребуваних пристроїв, *OEM* вже не може надавати деякі аспекти *OEM*-рішень. Отже, завод їх більше не купуватиме. Оскільки програмне забезпечення є частиною рішення для *OEM*, воно стає недоступним (рис. 2.10).

Крім того, можливо, що фабрика порушує умови контракту з постачальником постачальників (рис. 2.11). Це – лише один із можливих сценаріїв, але він відображає вплив однієї зміни в одній частині екосистеми на інші.

Тепер давайте спостерігатимемо за екосистемою, яка базується на прикладних програмах (рис. 2.12).

Інженер-розробник випускає нову версію програмного продукту, але не надає інструментів для розробки та розширення програмного забезпечення (рис. 2.13).

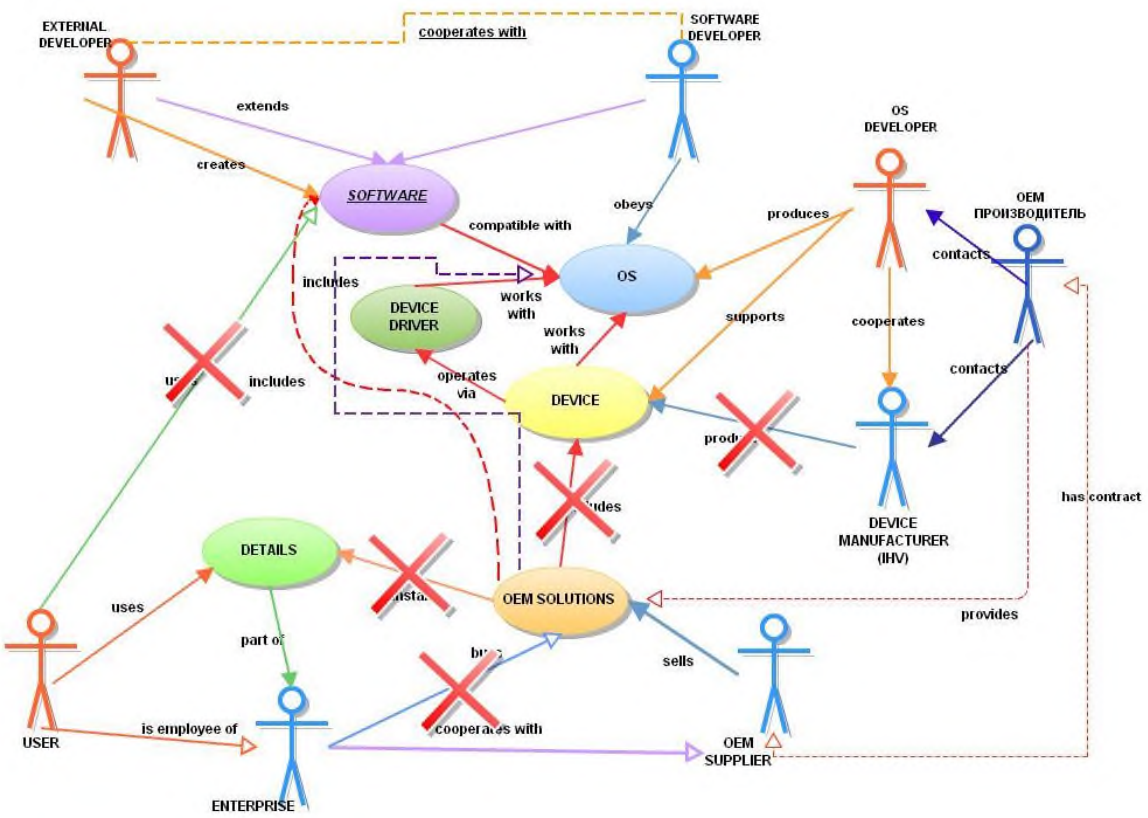

Рис 2.10. Вплив відбувся на програмну екосистему

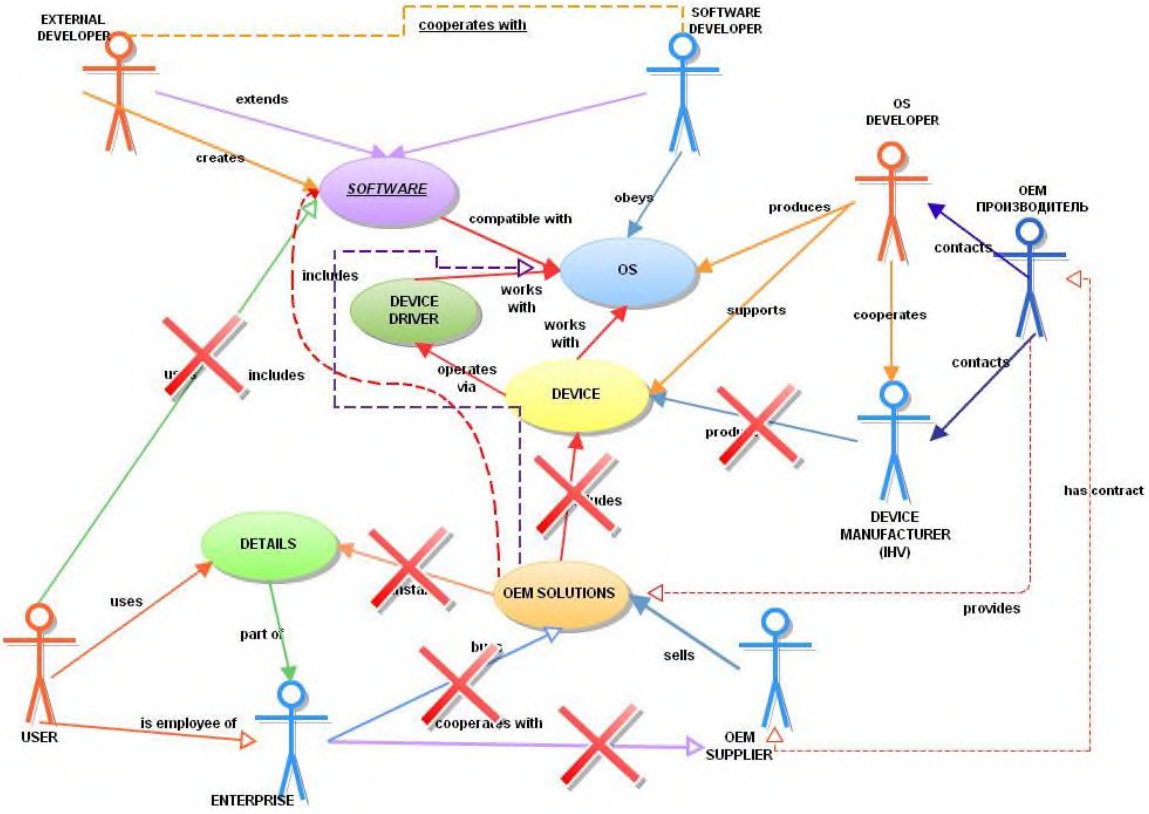

Рис. 2.11 Програмна екосистема з розірваним контрактом з постачальником *OEM*

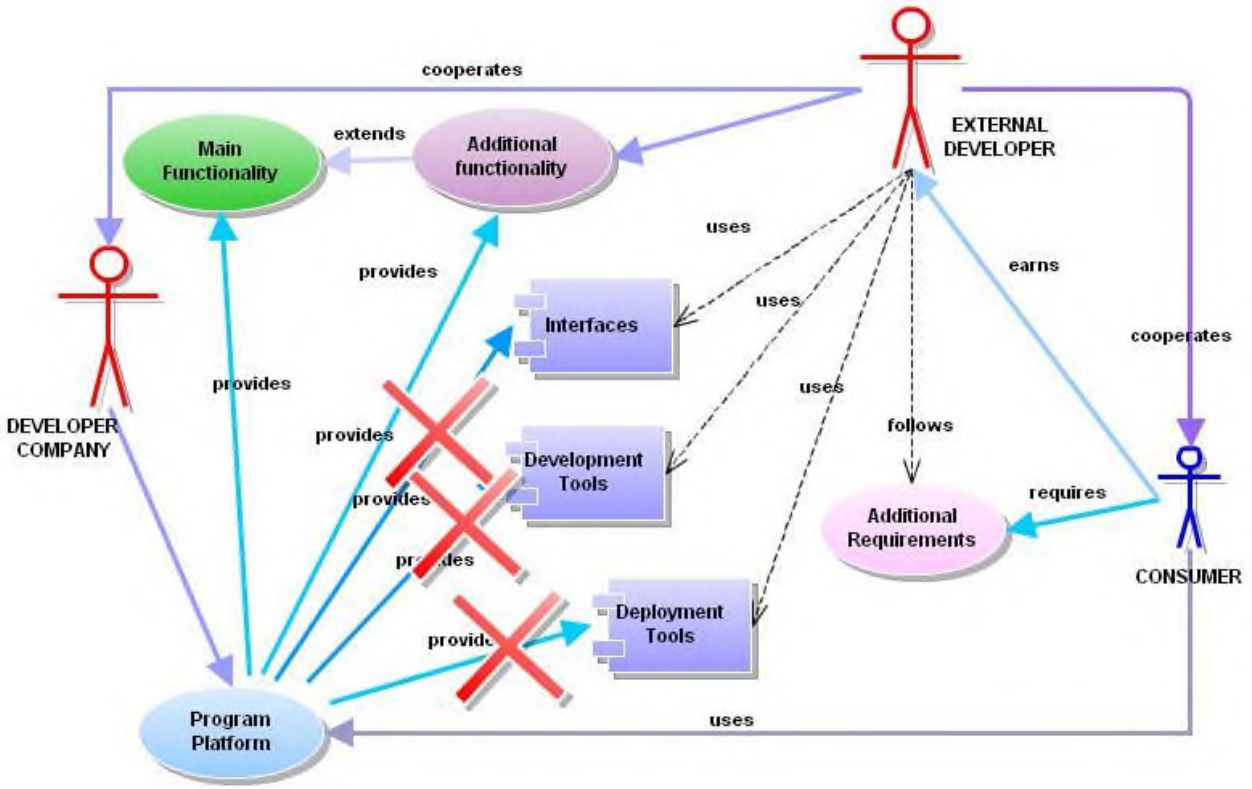

Рис. 2.12. Програмний продукт без інструментів для розширення

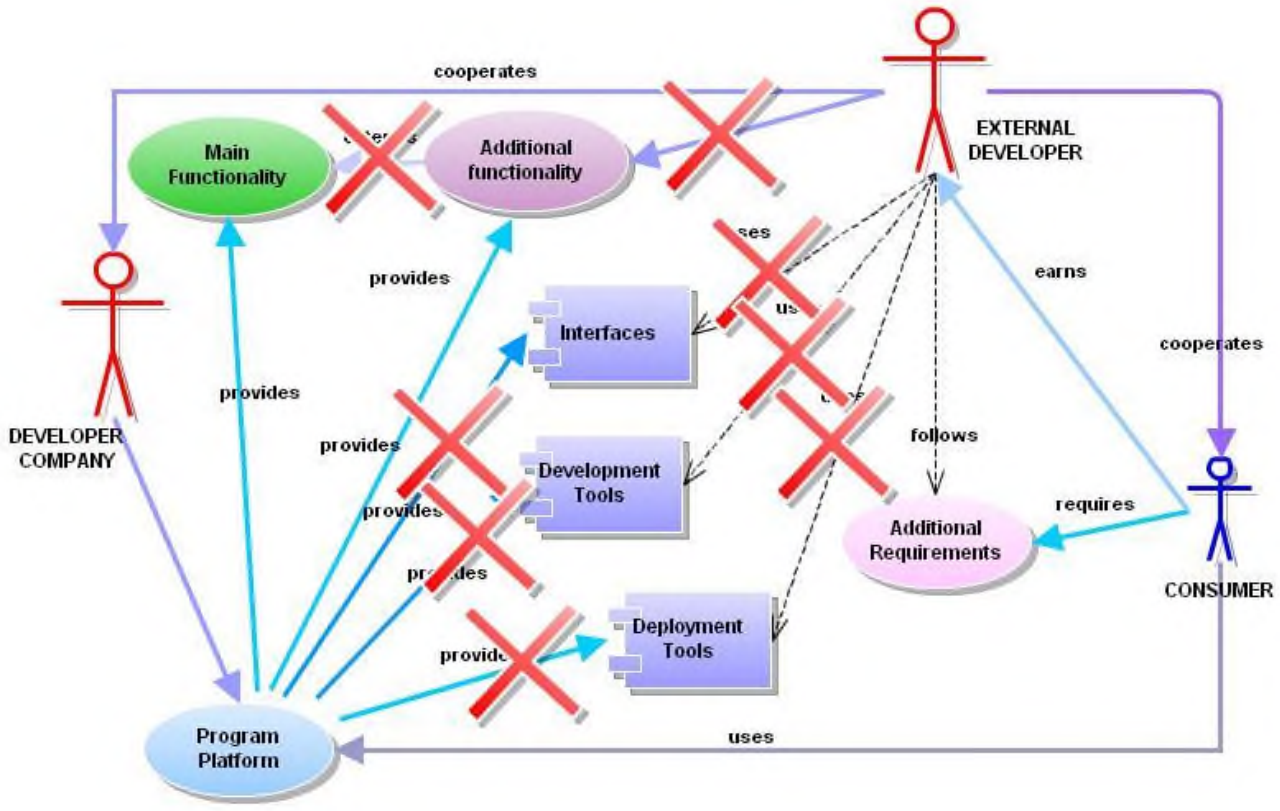

Рис. 2.13. Змінено програмно-орієнтовану екосистему

Спостерігатимемо за екосистемою кінцевого споживача (рис. 2.14). Припустимо, що користувач, який створив та супроводжував додаток, перестав це робити (рис. 2.15). Як наслідок, такий користувацький додаток буде позбавлений використання інших користувачів.

Тепер можна побачити приклад реакцій поведінки екосистеми на зміни, які вимагають оновлення версій програмного забезпечення. Але загалом стабільність системи залишається на високому рівні.

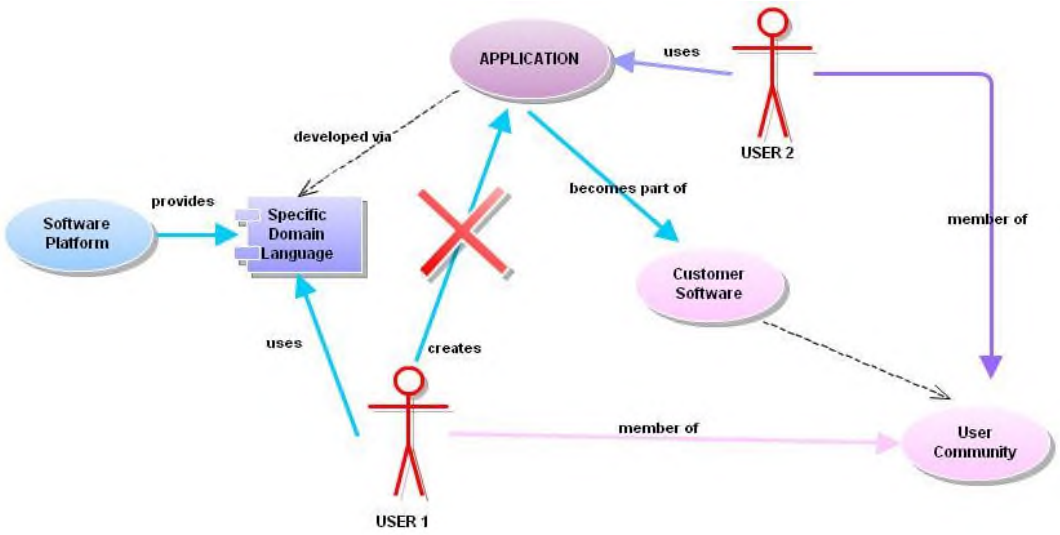

Рис. 2.14. Зміни відбулися в екосистемі програмного забезпечення для кінцевих користувачів

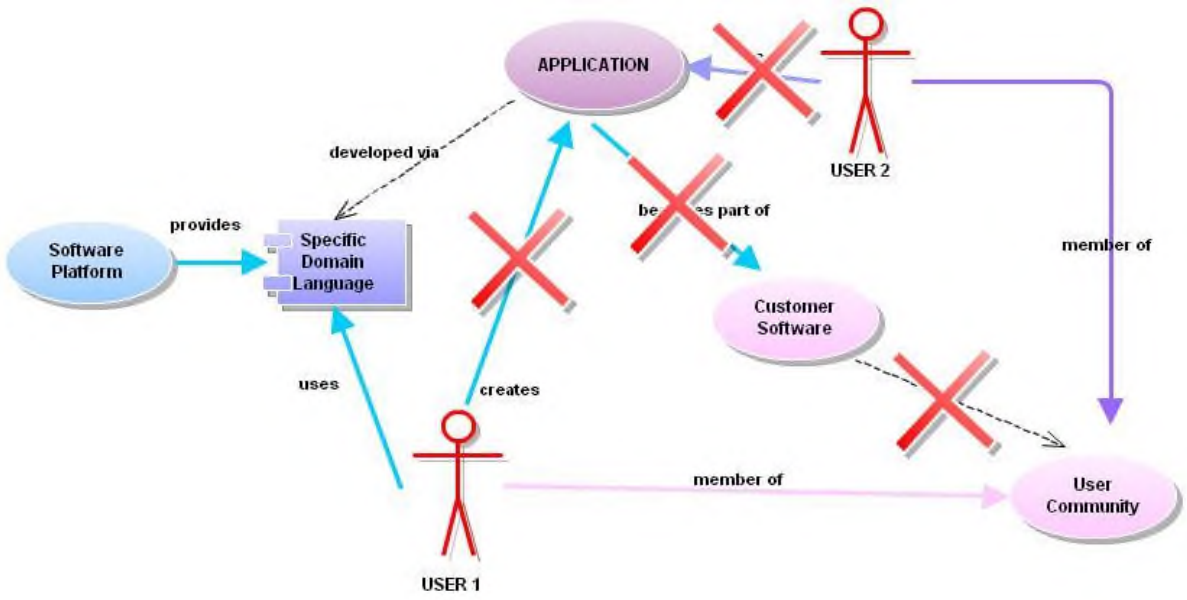

Рис. 2.15. Результати змін у програмній системі, орієнтованій на кінцевих користувачів

2.8. Висновки до розділу

У цьому розділі було оцінено методи вивчення систем контролю версій через парадигму екосистем: Програмна екосистема розглядалась на рівнях:

– на основі операційної системи;

– на підставі заяви;

– на основі кінцевого користувача.

Екосистеми змодельовано за допомогою мови *UML*.

Поведінка цих моделей моделюється в тих випадках, коли зміни однієї частини екосистеми відображалися в інших частинах.

Отримані моделі дозволяють проводити подальше вивчення екологічного підходу до галузі програмного забезпечення для розкриття питання необхідності ведення версій ПЗ при розробці.

РОЗДІЛ 3

РОЗРОБКА СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ ДЛЯ ОНОВЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Описання принципів роботи розробленої системи контролю версій

Базу даних системи контролю версій ПЗ реалізовано в вигляді *web*-додатку.Відкрити базу даних можна скориставшись посиланням *http*://217.77.222.218. Реалізований принцип роботи системи контролю версій представлено на рисунку 3.1.

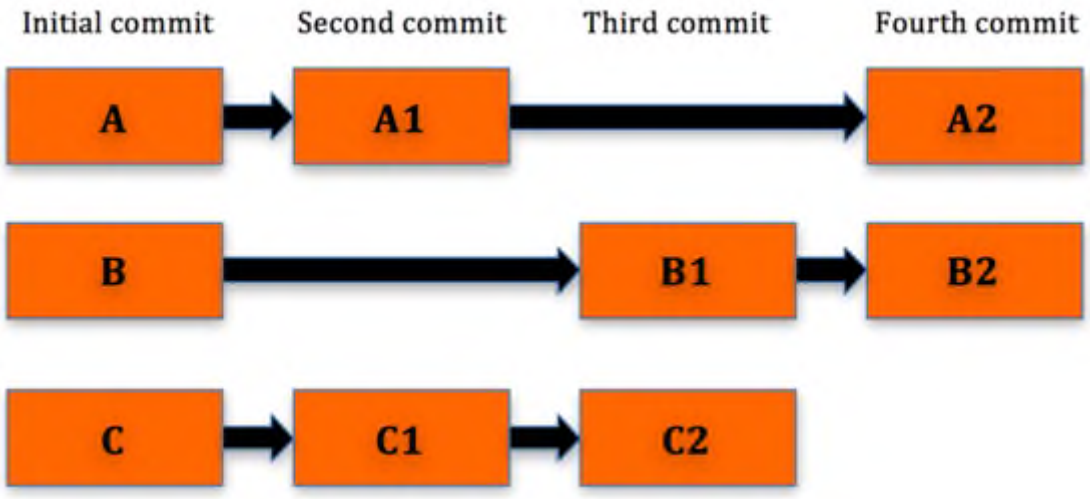

Рис. 3.1. Принцип роботи системи контролю версій

За допомогою меню можна обирати необхідні операції: “Додати/видалити ПЗ”, “Додати/видалити дистрибутив”, “Додати/видалити автора”, “Пошук ПЗ за датою”, “Пошук ПЗ за автором”, “Пошук ПЗ за назвою”, “Пошук автора”, “Звіт”.

Якщо потрібно додати до бази інформацію про новий дистрибутив потрібно натиснути в меню кнопку “Додати/видалити дистрибутив”.

Роботи системи може бути представлена наступною діаграмою (рис. 3.2):

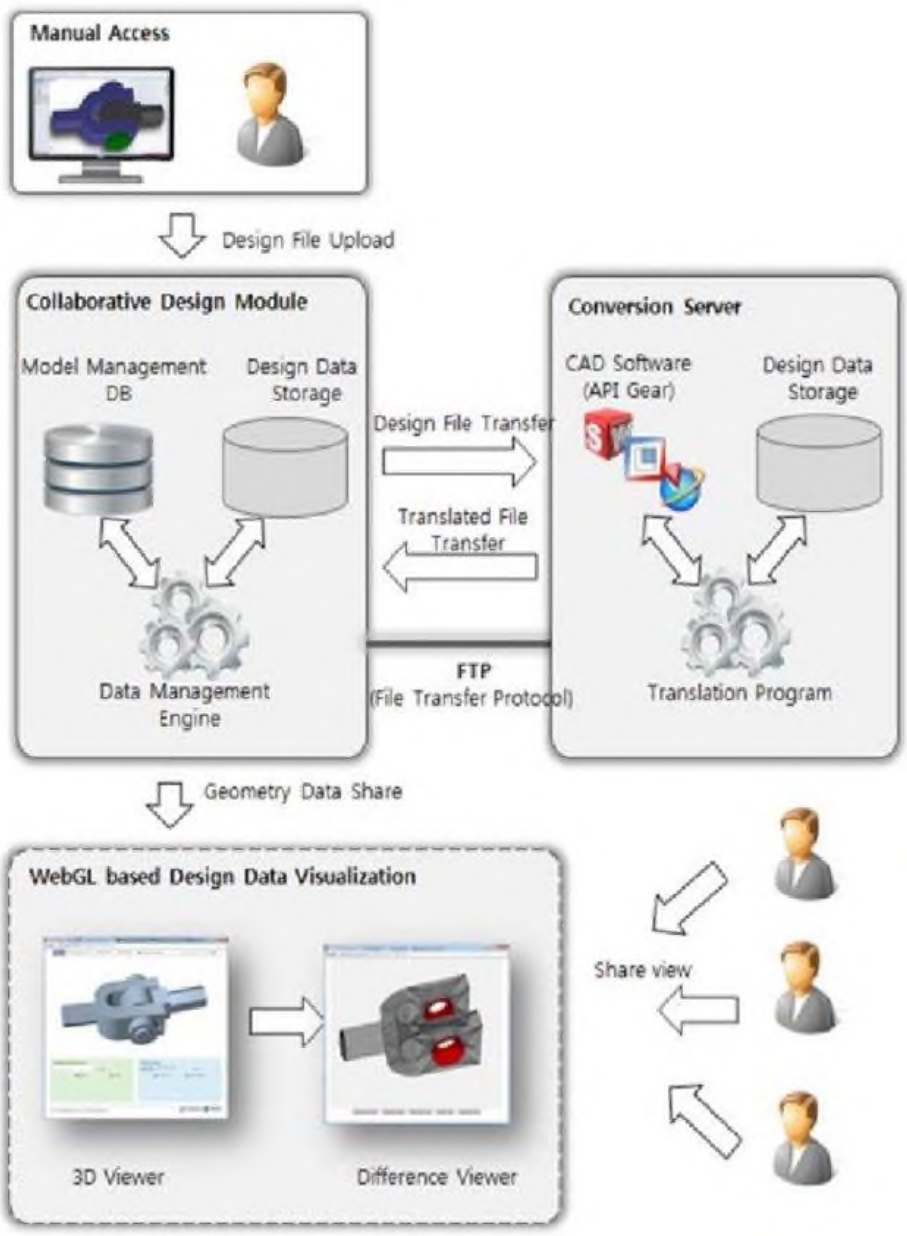

Рис. 3.2. Діаграма роботи системи контролю версій

У вище приведеному вікні потрібно визначати назву продукту, його версію, *ftp*-адреса тощо. Ці данні направляються до дистрибутиву для додавання. Або вибрати мишкою дистрибутив з таблиці “Наявні дистрибутиви” та натиснути кнопку “Видалити відмічені” для видалення.

Посередником між рівнями області визначення і розподілу даних (*domain and data mapping layers),* використовуючи інтерфейс, схожий з колекціями для доступу до об’єктів області визначення.

Система зі складною моделлю області визначення може бути спрощена за допомогою додаткового рівня, наприклад *Data Mapper*, який би ізолював об’єкти від коду доступу до БД. У таких системах може бути корисним додавання ще одного шару абстракції поверх шару розподілу даних (*Data Mapper)*, в якому б був зібраний код створення запитів. Це стає ще більш важливим, коли в області визначення безліч класів або при складних, важких запитах. У таких випадках додавання цього рівня особливо допомагає скоротити дублювання коду запитів.

Патерн *Repository* (рис. 3.3.) є посередником між шаром області визначення і шаром розподілу даних, працюючи, як звичайна колекція об’єктів області визначення. Об’єкти-клієнти створюють опис запиту декларативно і направляють їх до об’єкта-сховища (*Repository)* для обробки. Об’єкти можуть бути додані або видалені з сховища, як ніби вони формують просту колекцію об’єктів. А код розподілу даних, прихований в об’єкті *Repository*, подбає про відповідних операціях в непомітно для розробника. У двох словах, патерн *Repository* инкапсулирует об’єкти, представлениие в сховище даних і операції, вироблені над ними, надаючи більш об’єктно-орієнтоване уявлення реальних даних. *Repository* також має на меті досягнення повного поділу і односторонньої залежності між рівнями області визначення і розподілу даних.

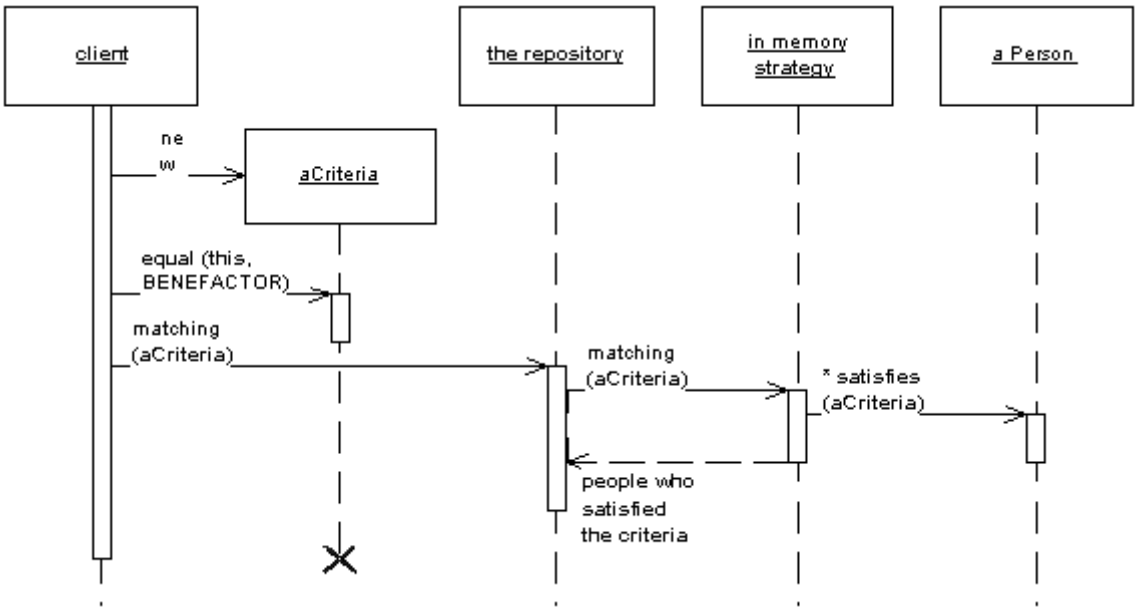

Рис. 3.3. Діаграма взаємодії модуля Репозиторій
в системі контролю версій

3.2. Розгортання та підключення системи контролю версій в ОС *Ubuntu*

Репозиторії містять велику кількість програм, однак існують програми, відсутні в репозиторіях *Ubuntu*, і можливо, Ви хотіли б їх використовувати. Існує багато сторонніх репозиторіїв, підключивши які Ви отримаєте доступ до додаткового ПЗ. Зробити це можна як за допомогою графічного інтерфейсу, так і в консолі.

Деякі репозиторії крім потрібних Вам пакетів можуть містити експериментальні збірки різного системного ПО, в тому числі і ядер *linux*. Оскільки версія цих експериментальних пакетів як правило вище, ніж встановлена у Вас, Менеджер оновлень може спробувати «оновити» систему з цих репозиторіїв, що в свою чергу може пошкодити Вашу систему. Тому уважно читайте опис підключається сховища та інформацію в Менеджері оновлень.

За допомогою графічного інтерфейсу

Для підключення сховища виконайте наступні кроки.

Відкрийте Центр додатків. Відкрийте пункт меню Правка → Джерела додатків. У вікні виберіть вкладку «Інше ПЗ», натисніть кнопку «Додати».

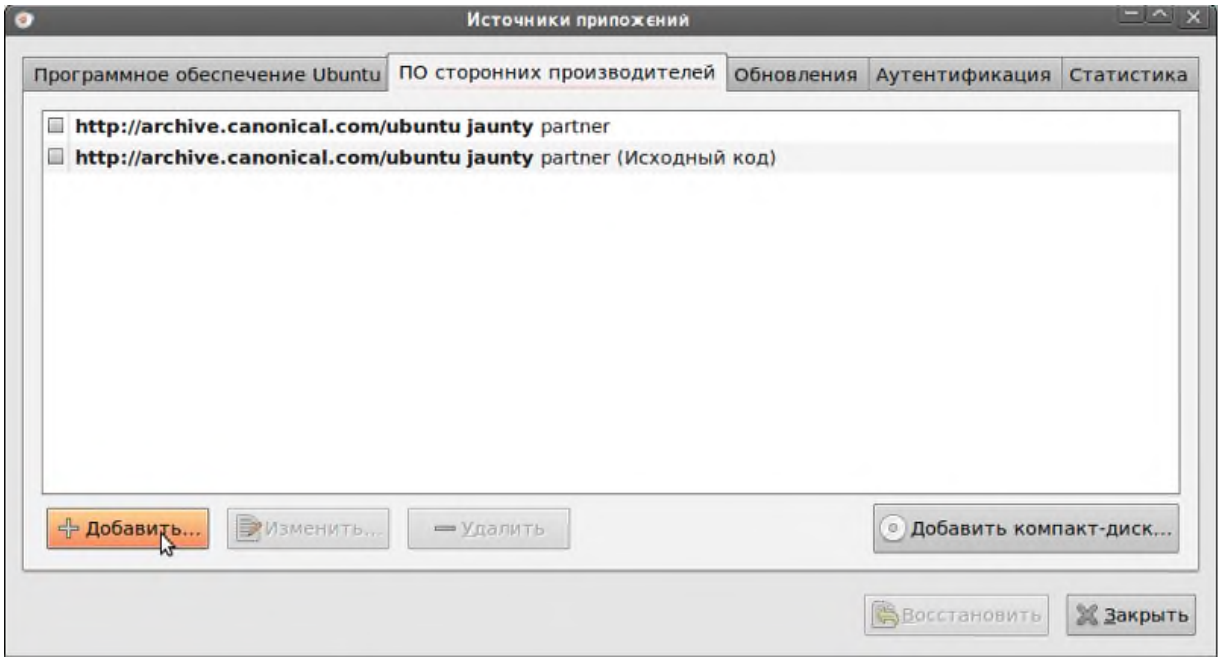

Рис. 3.4. Кроки встановлення СКВ. Джерела додатків

У вікні заповніть поле «Рядок *APT:*» і натисніть кнопку «Додати джерело» (рис. 3.5).

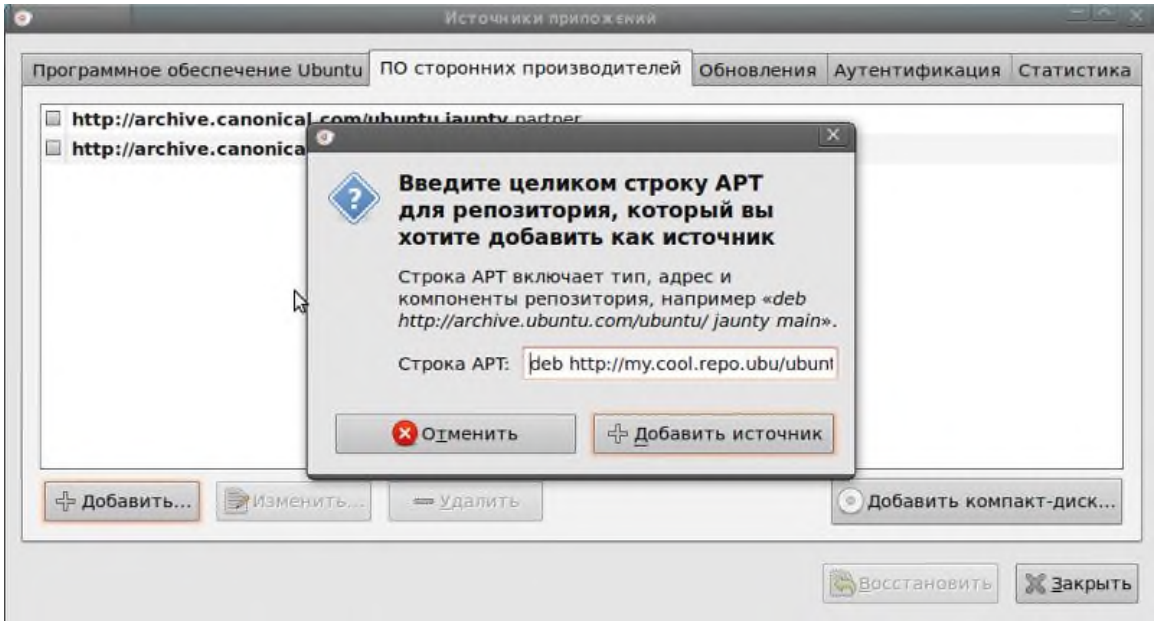

Рис. 3.5. Кроки встановлення СКВ. Рядок APT

Джерело буде додано і включений, натисніть кнопку «Закрити» (рис. 3.6).

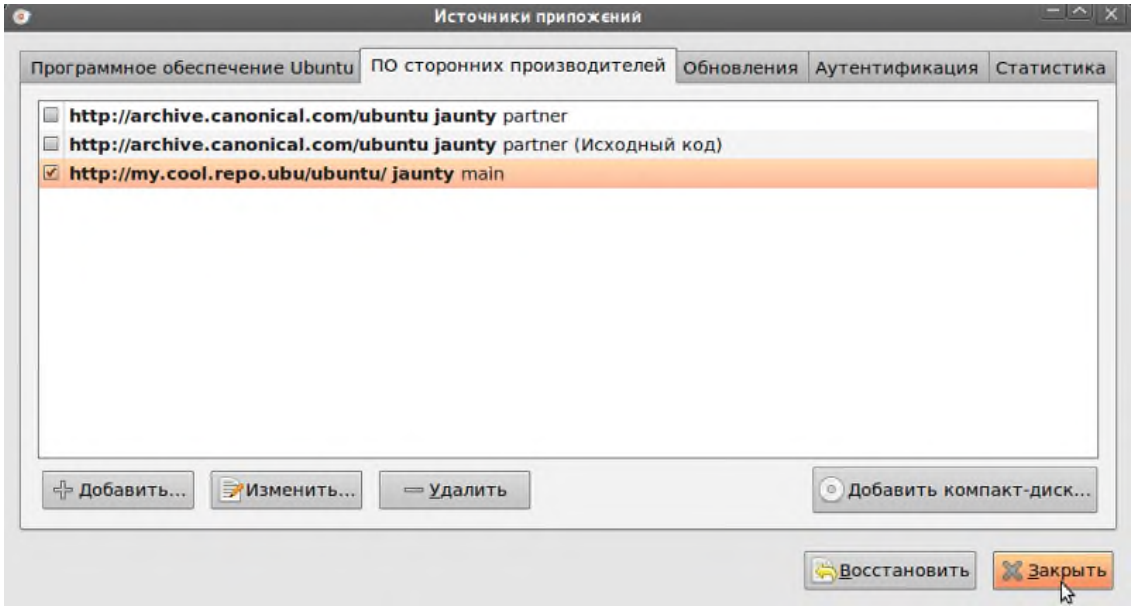

Рис. 3.6. Кроки встановлення СКВ. Закінчення встановлення джерела додатків

Оскільки було підключене нове джерело програмного забезпечення, необхідно оновити інформацію про пакети. З’явиться вікно, з пропозицією це зробити. Необіхідно натиснути «Оновити» (рис. 3.7).

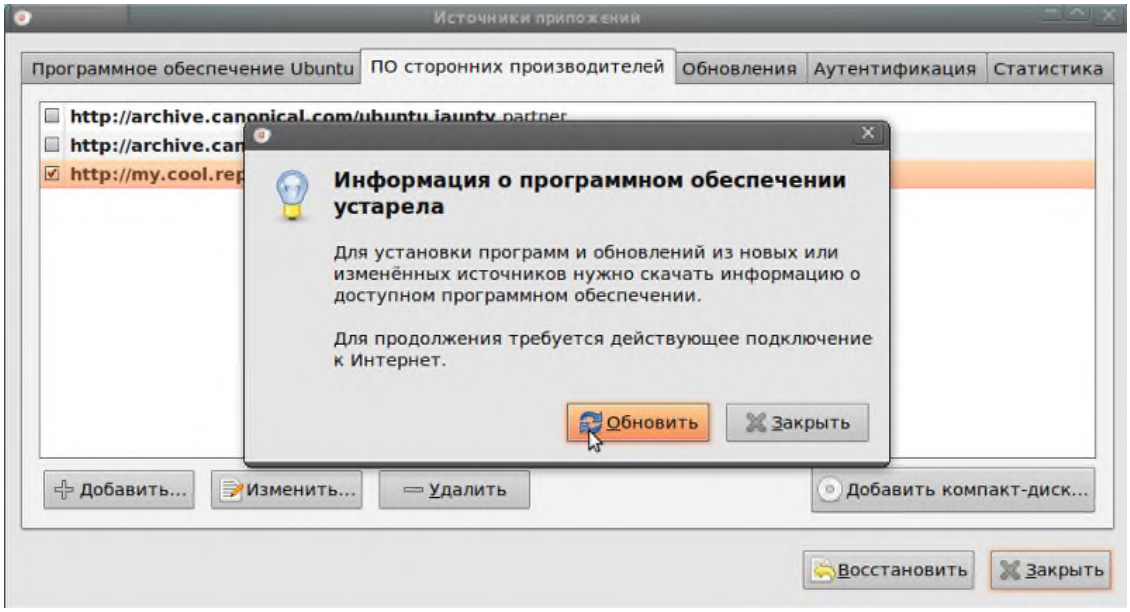

Рис. 3.7. Кроки встановлення СКВ. Оновлення джерела додатків

Після оновлення інформації про пакети вікно «Джерела додатків» закриється, і швидше за все ви отримаєте помилку про непідписаному джерелі додатків, тим не менш, ви зможете встановлювати пакети, що містяться в свежеподключенном репозиторії стандартними засобами. Для усунення помилки непідписаного сховища см. Пункт про захист репозиторіїв нижче.

За допомогою консолі (рекомендований спосіб)

В системі *Ubuntu* 10.04 додавати репозиторій можна однією командою, ось приклад для *ppa*-сховища:

sudo apt-add-repository ppa: ripps818 / coreavc

Системний список репозиторіїв міститься в файлі */etc/apt/sources.list*. Для того, щоб додати репозиторій – відредагуйте цей файл, наприклад так:

sudo nano /etc/apt/sources.list

Де першим рядком йде додається нами репозиторій.

Збережіть файл і закрийте редактор. Для *nano* потрібно натиснути *Ctrl + X*, підтвердити збереження змін – *Y* і переконавшись, що ім'я файлу */etc/apt/sources.list* натиснути *Enter*.

Далі слід оновити список пакетів. Для цього виконайте:

sudo apt-get update

Тепер можна встановлювати пакети з нового сховища, правда, для комфортної роботи вам доведеться так само імпортувати в систему ключ сховища, тому що у вас постійно буде з'являтися таке попередження:

Оскільки репозиторії здебільшого розташовані в інтернеті, існує ймовірність підміни сховища зловмисником на свій, що містить модифіковані пакети. Таким чином, користувач може встановити собі модифікований пакет і тим самим поставити безпеку своєї системи під загрозу. Багато репозиторії мають захист від підміни. Такий захист реалізована за допомогою звірки цифрових підписів сховища та клієнта. У разі, коли репозиторій має цифровий підпис, а призначений для користувача комп'ютер містить відкритий ключ для цього сховища – такий репозиторій вважається довіреною.

В *Ubuntu* за замовчуванням довіреними є репозиторії на настановних дисках і основні інтернет репозиторії – *archive.ubuntu.com.* При наявність на комп'ютері користувача кількох підключених репозиторіїв, перевага віддається довіреною.

При підключенні сховища, захищеного цифровим підписом Вам потрібно завантажити (зазвичай з ресурсу, що розповідає про цей репозиторій, або з сервера ключів, що є кращим в будь-якому випадку) відкритий ключ і додати його в систему. Іноді для скачування надається доступний для установки пакет, який в свою чергу при своїй установці сам прописує ключ сховища. Якщо ви завантажуєте ключ з сайту сховища, то ви отримаєте звичайний файл з розширенням *.key*, *.gpg* або іншим. Додати його в систему можна так:

sudo apt-key add repo.key

За допомогою графічного інтерфейсу – запустіть «Джерела додатків» (Система → Адміністрування → Джерела додатків), перейдіть на вкладку «Аутентифікація» і натисніть на кнопку «Імпортувати файл ключа» – відкриється діалог вибору файлу. Виберіть файл ключа і натиснути *OK* (рис. 3.8).

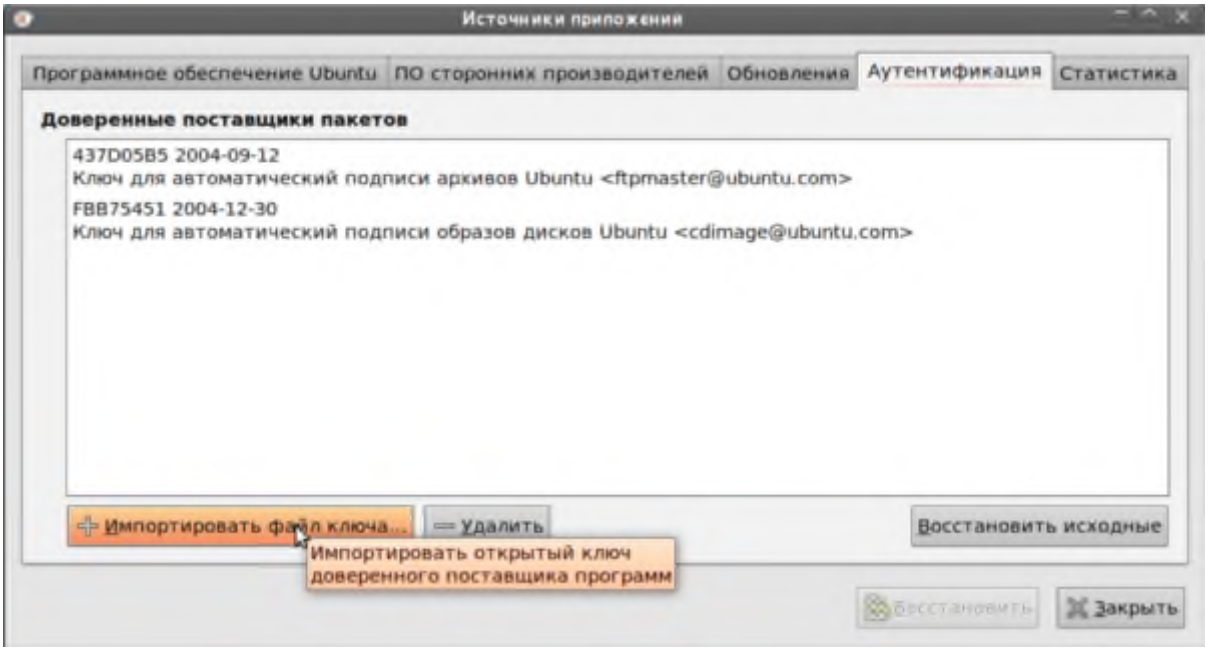

Рис. 3.8. Захист репозиторію ПЗ через графічний інтерфейс ОС

Однак набагато більш привабливим є додавання ключа зі спеціального захищеного сервера. Зазвичай, коли заходить мова про ключі, дається його незрозумілий з першого погляду буквено-цифровий ідентифікатор виду 123*ABCDEFG*456 (рядок з довільних цифр і букв латинського алфавіту в верхньому регістрі). Це – унікальне ім'я (ідентифікатор) ключа. Іноді ключ описується рядком виду 1024*R* / 123*ABCD*, тоді ідентифікатором є частина після слеша. Так ось, ключі переважно зберігаються на спеціальних серверах, звідки будь-хто може їх отримати. Ключі для репозиторіїв *Ubuntu* прийнято зберігати на *keyserver.ubuntu.com.* Для отримання і імпортування в систему ключа з сервера необхідно виконати команду:

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 0*x12*345678

В даній команді замість *keyserver.ubuntu.com* можна підставити адресу іншого сервера ключів, а замість 12345678 необхідно написати ідентифікатор потрібного вам ключа.

3.3. Основні вікна керування системою контролю версій

Розроблений інтерфейс подібний до системи *Git.* Даний інтерфейс в основному направлений на програмістів, яким постійно необхідно взаємодіяти зі співробітниками, вирішувати типові завдання контролю саме коду, для людей, які звикли працювати в *Unix-like* системах, використовуючи термінал, найкраще підійде консольний вид додатків. Вони так само прості в зверненні, трохи швидше і функціональніша, але їм доведеться приділити час, для того щоб розібратися в використанні.

Правий клік мишкою на папці викликає контекстне меню системи (рис. 3.9).

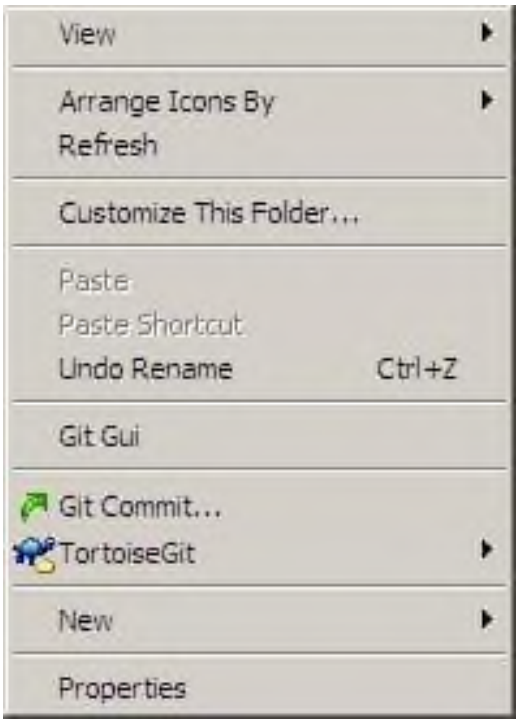

Рис. 3.9. Контекстне меню системи

Правий клік мишкою на додатку викликає контекстне меню додатку (рис. 3.10).

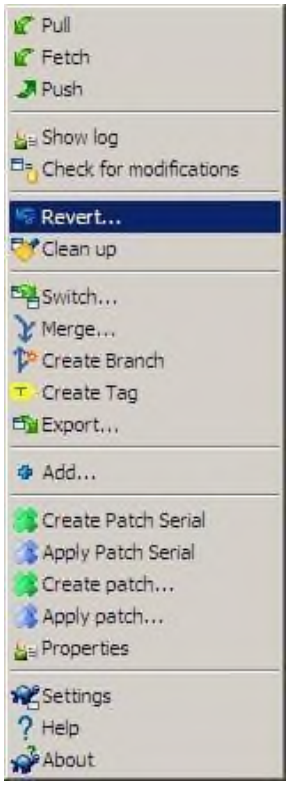

Рис. 3.10. Контекстне меню додатку

Правий клік мишкою на додатку або на каталогу, який містить код, що інтерпритується, викликається розширене контекстне меню додатку (рис. 3.11).

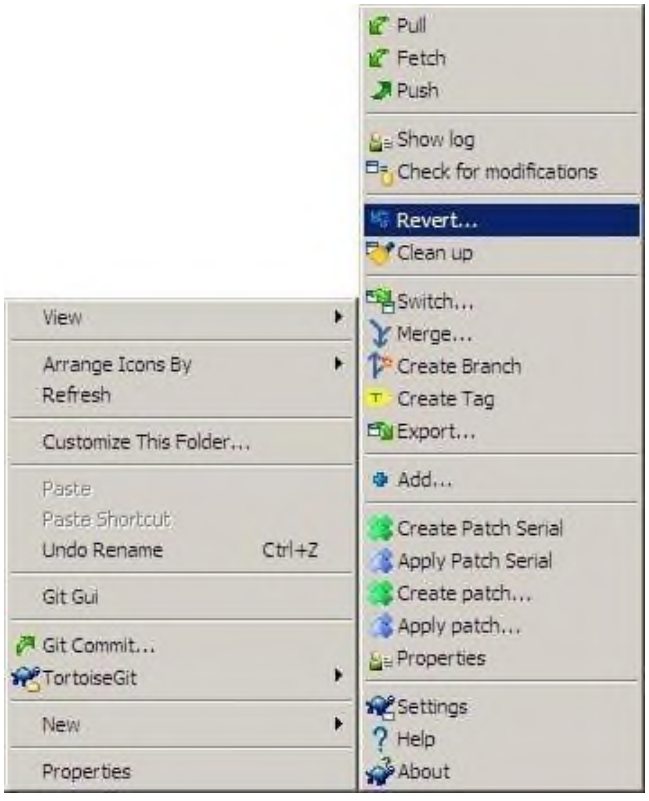

Рис. 3.11. Контекстне меню для додатку, написаного інтерпритованою момоб програмування

В системі передбачено функцію порівняння версій (рис. 3.12). Результат порівняння зрівняння версій представлено на рисунку 3.13.

Також в системі передбачено консольний інтерфейс (рис. 3.14).

3.4. Перетворення лінійки продуктів на відкриту платформу з елементами контролю версій

Як тільки компанія вирішить перетворити ряд продуктів на платформу для зовнішніх інженерів-розробників, знову слід прийняти деякі рішення. По-перше, якнайбільше можливо, рішення з'ясовується, наскільки компанія застосовує спрямований підхід проти ненаправленого підходу до вибірки заявки та партнера (рис. 3.15).

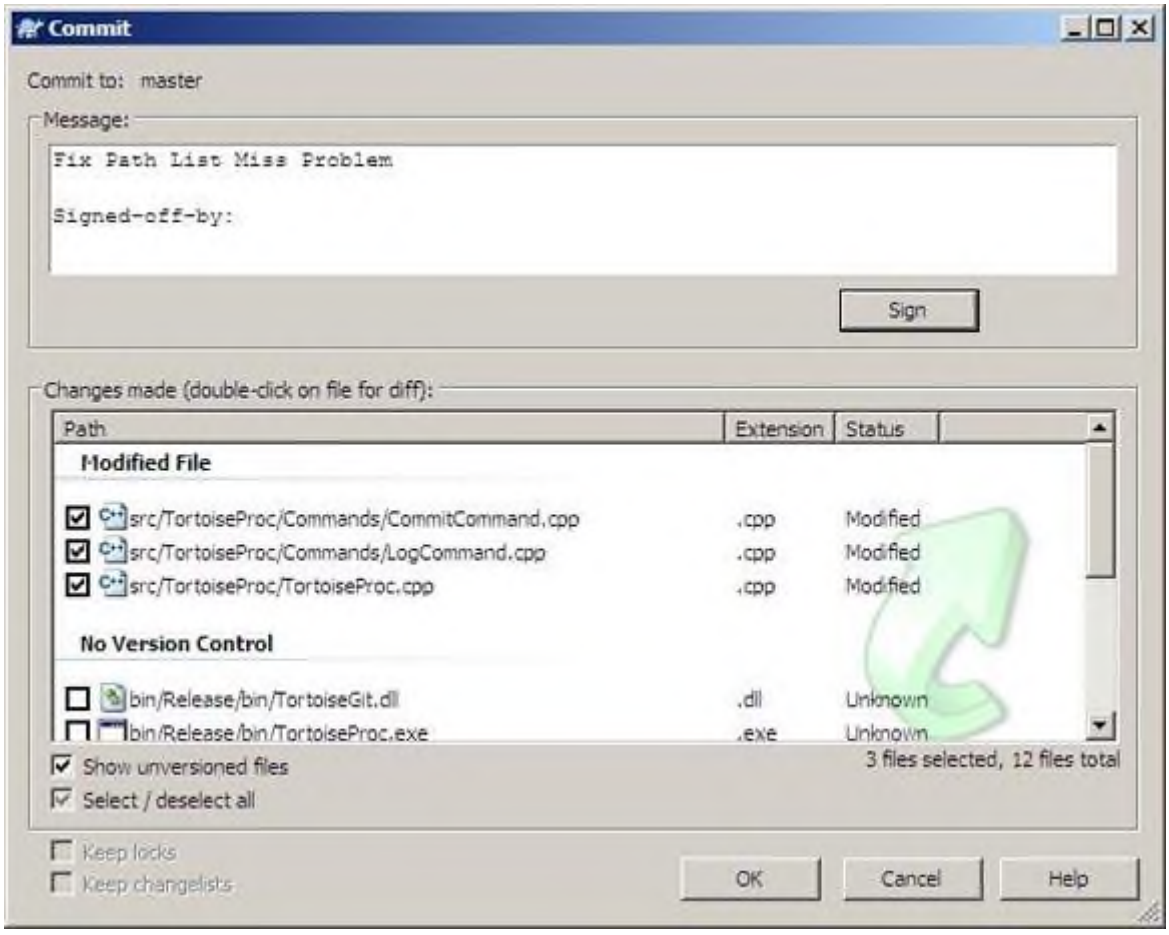

Рис. 3.12. Вікно порівняння версій (*Merge)*

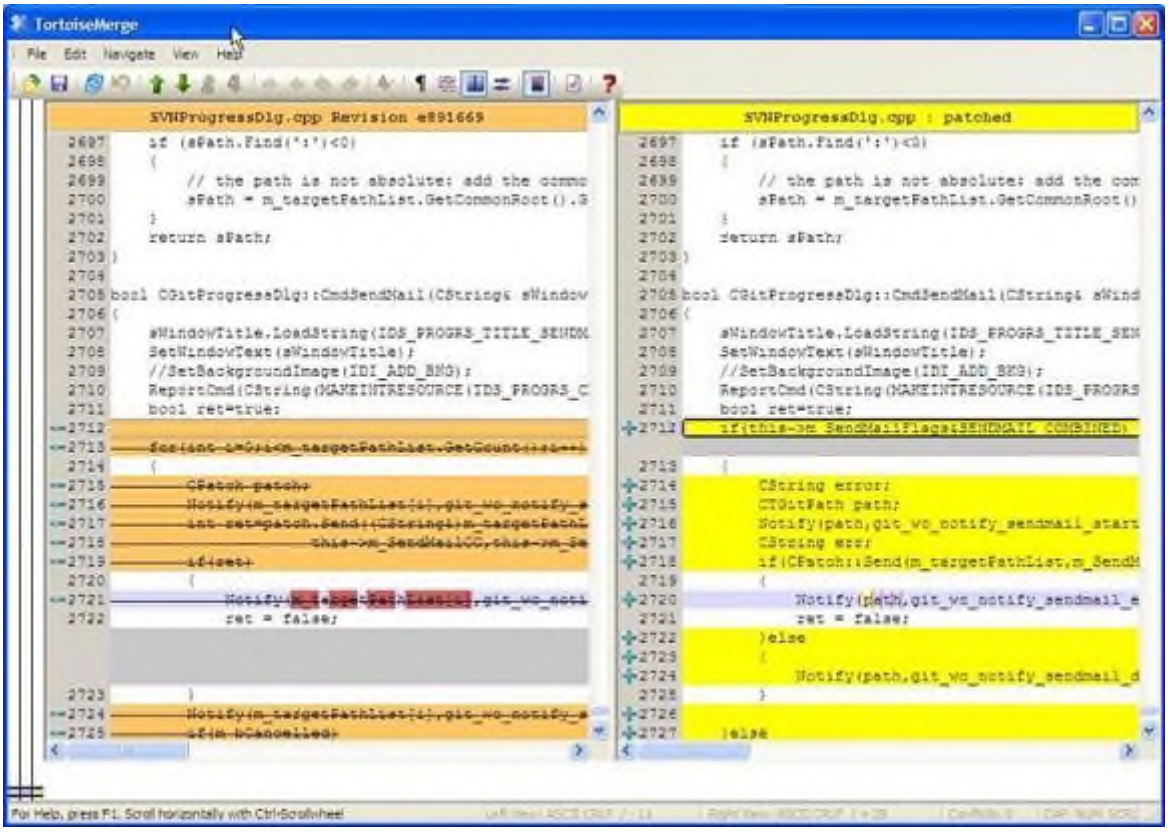

Рис. 3.13. Результат команди порівняння версій

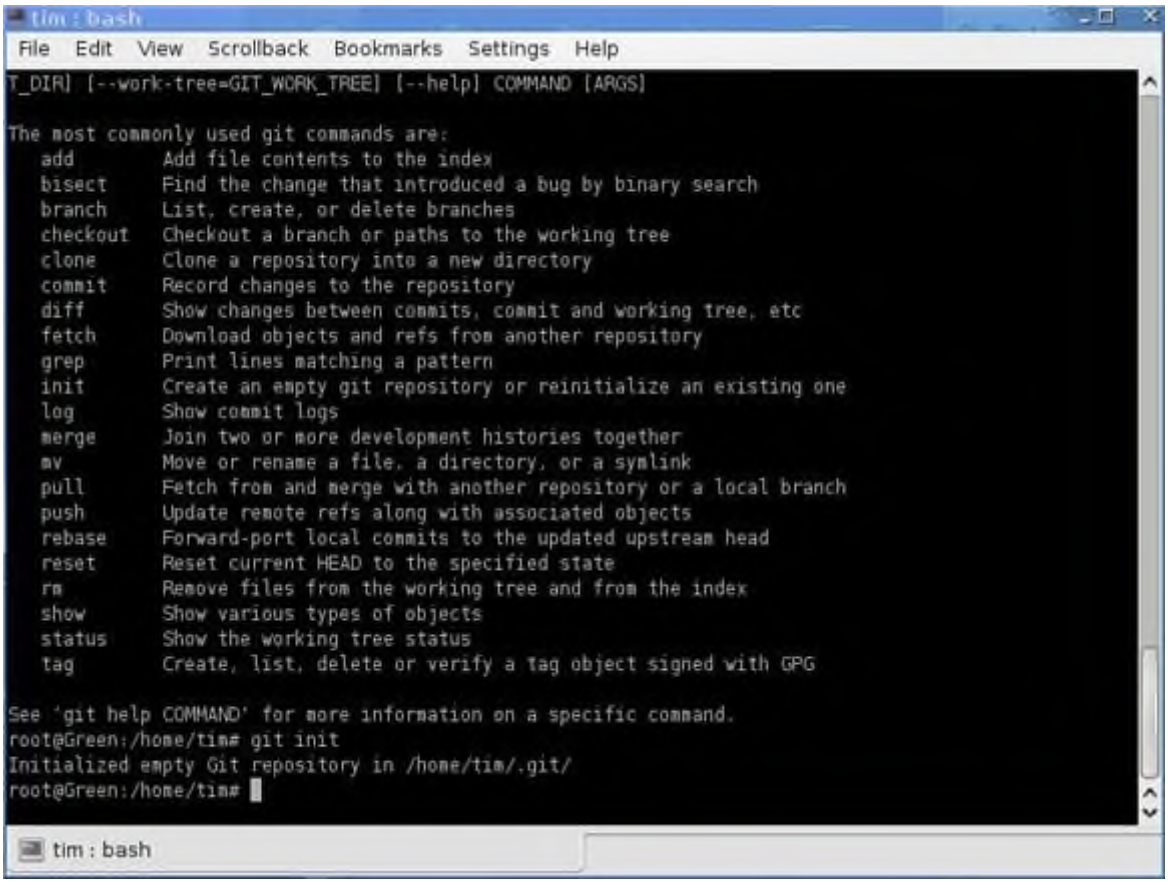

Рис. 3.14. Консольний інтерфейс

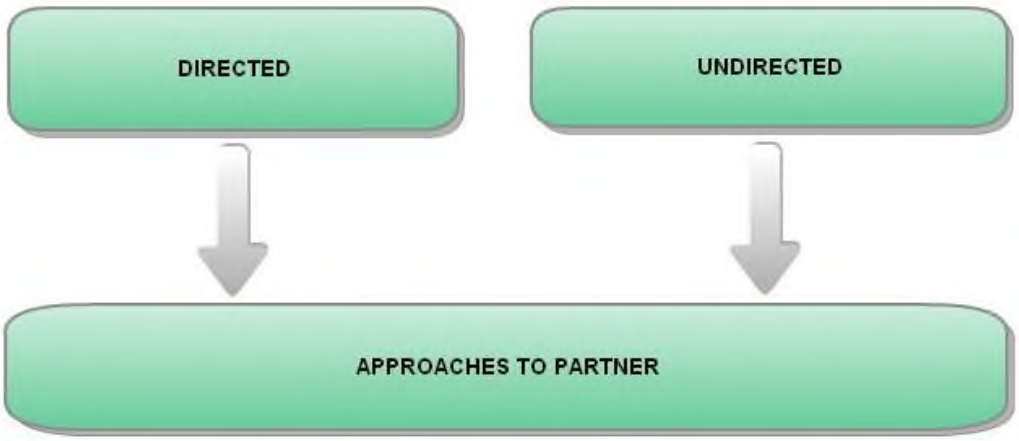

Рис. 3.15. Підходи до програмного партнера

Спрямований підхід передбачає, що компанія-розробник платформи визначила певні сфери функціональності, які не бажають розвиватися самостійно, але хочуть запропонувати своїм клієнтам. У цьому випадку компанія вибирає партнерів, які в змозі надати рішення для обраних областей та узгоджують контракт з партнером. Угода зазвичай включає форму розподілу доходів, і компанія-партнер отримує доступ до платформи та продуктів, що розробляються всередині. Зазначаємо, що угода, в якій партнеру компенсується плата за НДДКР, а потім передаються права інтелектуальної власності компанії розробника платформи, *є* традиційною розробкою іноземного програмного забезпечення (рис. 3.16). Спрямований підхід дозволяє компанії-платформі широко виуористовувати зовнішні ресурси для забезпечення внутрішніх потреб.

Ненаправлений підхід можна представити, як протилежний інший кінець спектру: компанія платформи не пропонує платформу як основу для програмування програм зовнішнього розвитку інженерів, не обмежуючи ні розробку програмного забезпечення, ні створення (рис. 3.17). Зовнішні інженери-розробники можуть розробляти рішення, які конкурують між собою, як і компанія-платформа. Хоча це може здатися парадоксальним, але мати зовнішніх партнерів у розробці програмного забезпечення, конкурувати безпосередньо з компанією-платформою, і основним припущенням є те, що конкуренція є найкращим механізмом для вибору найкращих рішень для користувачів [9].

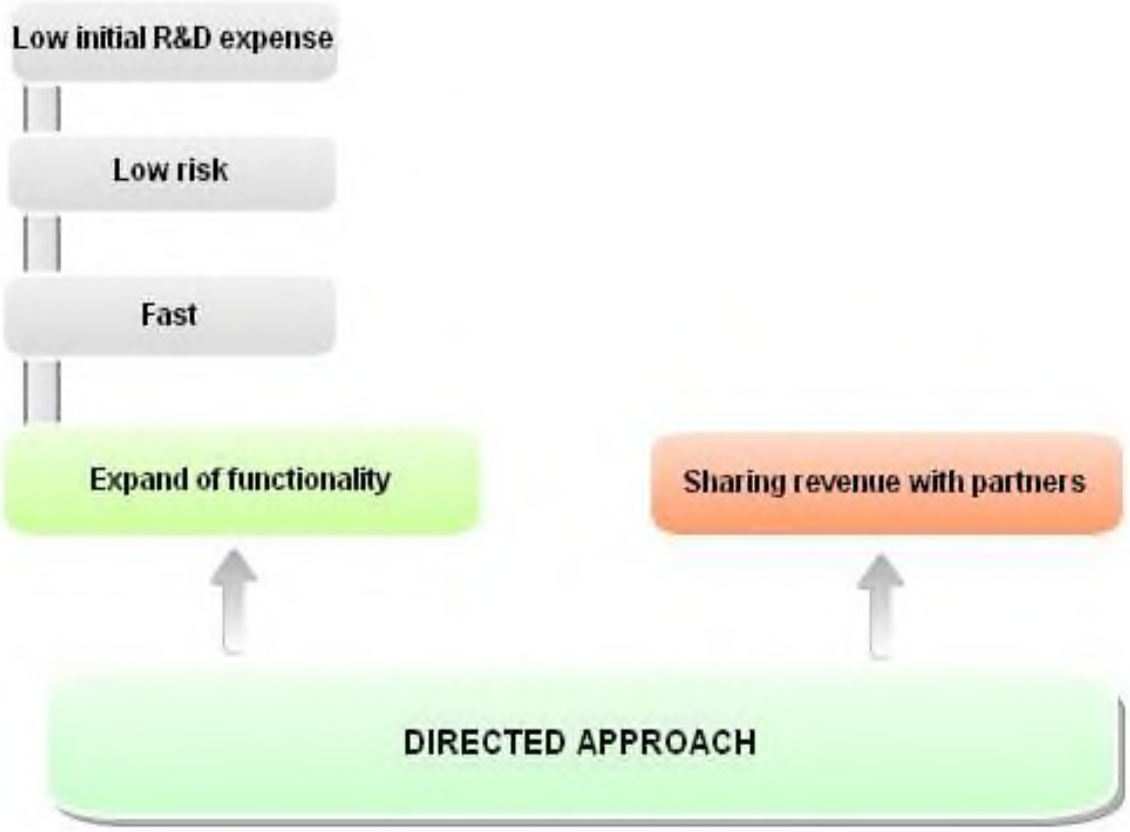

Рис. 3.16. Спрямований підхід до вибору партнера

Рис. 3.17. Непрямий підхід до вибору партнерів

Спрямований та ненаправлений підхід – це дві крайні цінності, які ігнорують важливий аспект: вони виникають за наявності неодноразових рівнів інженерів-розробників. Для типового перетворення лінійки продуктів в екосистему програмного забезпечення зазвичай достатньо чотирьох рівнів інженерів-розробників (рис. 3.18).

Інженери з внутрішнього розвитку: Незважаючи на те, що компанія вирішила відкрити платформу для зовнішніх розробників, ця платформа все ще служить для внутрішніх розробників у командах продуктів. Оскільки команди продукту та платформи є частиною однієї організації, рівень довіри та простота передачі – це ті, що можуть запропонувати глибокий доступ та розуміння платформи внутрішнім розробникам.

Інженери стратегічного розвитку: інженери стратегічного розвитку – це ті компанії, які мають довгострокові відносини і які, очевидно, були обрані для партнерства. Ці інженери-розробники, як правило, під керівництвом моделі, розширюють стратегічні сфери функціональності.

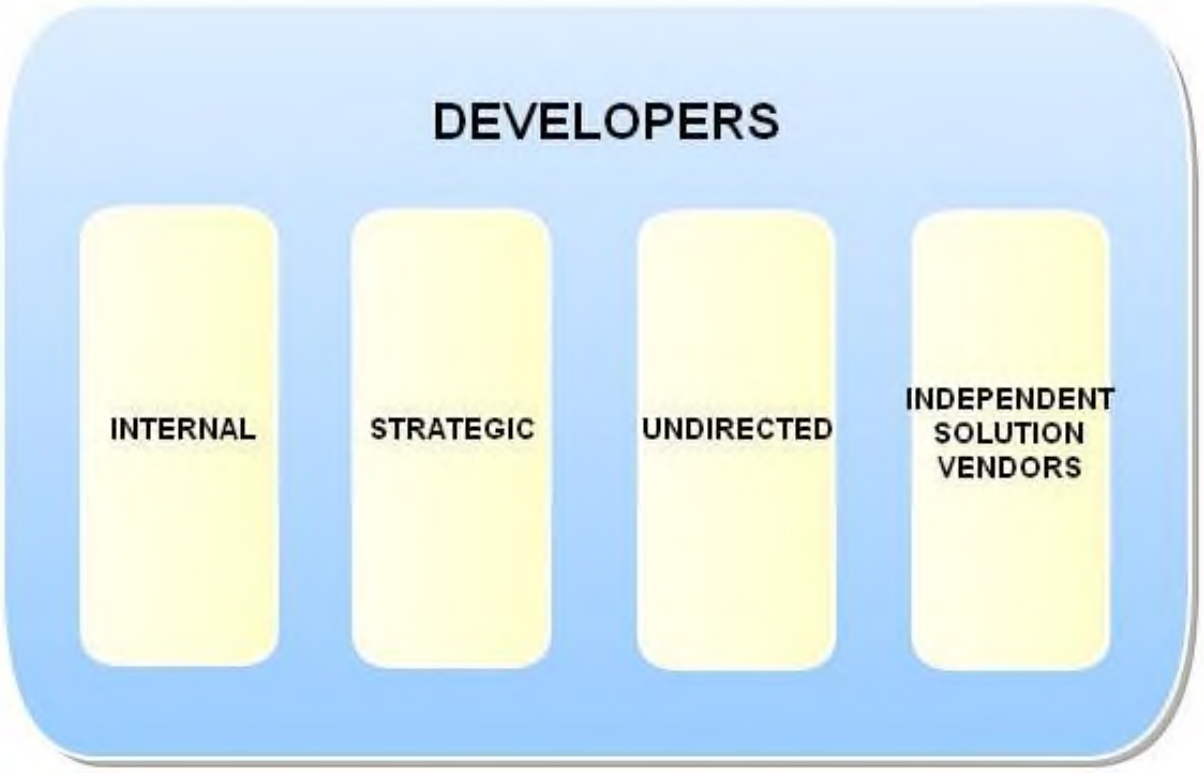

Рис. 3.18. Чотири рівні розробників

Через характер відносин, інженери стратегічного розвитку мають глибокий доступ до платформи, осягають довгострокову мапу комунікацій для платформи та повну лінійку продуктів, а їх програмне забезпечення з певною перевіркою перевіряє допустимість і, отже, вони мають доступ для читання та доступ до запису до приватних даних користувачів.

Не спрямовані інженери-розробники: Ці інженери-розробники створюють власні рішення, націлюються на ринок та продають своїм користувачам базову платформу для розширення функціональних можливостей, що надаються компанією інженерів платформи та стратегічного розвитку. Оскільки часто важко забезпечити відсутність випадкового або навіть навмисно шкідливого коду в рішеннях, що надаються непрямими інженерами-розробниками, ці інженери-розробники мають обмежений доступ до функціональності продукту та платформи. Хоча не спрямовані інженери-розробники є непередбачуваними і часом складними в управлінні, зазвичай це група, яка в стані використовувати платформу повністю непередбачуваними способами і, отже, забезпечити важливий імпульс інновації в повній екосистемі.

Незалежні постачальники рішення: клас закриття, не обов'язково унікальний для програмної екосистеми, але також є актуальним у цьому випадку. Незалежні постачальники рішень (*ISV)* пропонують клієнту певну інтеграцію однієї із значної кількості продуктів, що надаються компанією лінійки продуктів, з іншими ІТ-рішеннями, щоб створити програму, яка оптимально вирішує певні вимоги унікального клієнта. Підтримка добрих відносин з *ISV* є важливою для постачальника платформи, оскільки важливим джерелом майбутніх вимог є надання клієнтам певних рішень, які передаються між неодноразовими користувачами. Узагальненість між запитами споживачів, які не забезпечуються платформою, та продуктами на базі платформи, робить все можливе для розробки програмного забезпечення [9].

3.5. Взаємовідносини між розробниками

Одна з проблем переходу компанії з лінійки продуктів на програмну екосистему полягає у створенні та розвитку відносин із зовнішніми інженерами-розробниками. Оскільки компанія досягла успіху, продаючи продукцію, функціональність якої надзвичайно турбується про клієнтів, існує природний напрямок, який дозволяє зовнішнім інженерам-розробникам брати участь у процесі і, певною мірою, навіть мати частину взаємовідносин з клієнтами.

По-друге, компанія, розробник платформи, повинна розробляти платформу та включати нові функції. Одним з основних джерел нових функціональних можливостей є рішення, розроблені зовнішніми інженерами-розробниками. Якщо цей процес компанія інженер-розробник платформи обережно не працює, функціональний союз буде спірним і називає негативну репутацію цієї компанії. З іншого боку, компанія, розробник платформи, не може уникнути розвитку цієї платформи, не вдаючись до ризику.

Найкращий підхід – це публічно оприлюднити довгостроковий сюжет платформи, який визначає наміри компанії та дозволяє зовнішнім інженерам-розробникам рухатися до більш диференційованої функціональності та невдалості з тієї області, куди компанія-інженер-розробник платформ-літаків рухається (рис. 3.19).

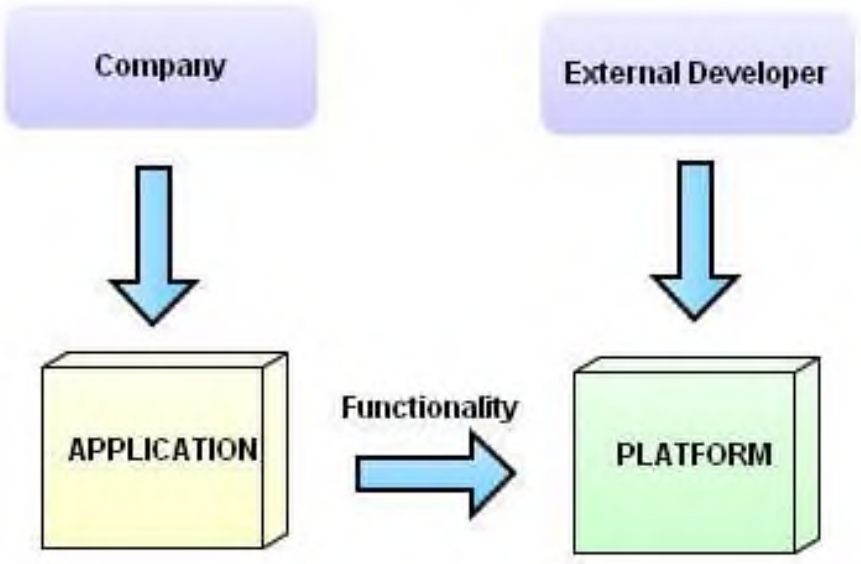

Рис. 3.19. Візуалізація відносин розробника з ПЗ

3.6. Реалізація механізму координації

Можна спостерігати перехід від лінійки продуктів до підходу до екосистеми програмного забезпечення, як перехід від підходу спрямованості на продукт до лінійки продуктів. Післядії фактично впливають на кожну функцію в компанії, як було сказано в попередньому розділі. Однак спосіб, яким компанія розробляє програмне забезпечення, підпорядковується навіть більшим, ніж іншим функціям організації. Післядії можна розділити на класи за трьома основними напрямками, тобто механізмами координації, технічною швидкістю та складом продукції. [9]

Перша основна зміна між внутрішньою розробкою програмного забезпечення та підходом екосистеми програмного забезпечення полягає в тому, що зовнішні групи інженерів-розробників не можуть піддаватися стандартизованим моделям процесів, інструментів та способів роботи. Це означає, що традиційна зрілість процесу наближається.

З іншого боку, кількість підпорядкованих партій та складність відносин між цими партіями ускладнює повну модель взаємодії до таких пор, що традиційна централізована модель буде важкою в управлінні. Складність та послідовні витрати на координацію – це ті, що через конкурентоспроможність підходу можна поставити під сумнів.

Ця тривожність стосується не лише процесів розробки програмного забезпечення програмного забезпечення, але також і безперервних процесів, включаючи управління запитами та планування ініціатив. Прийняття екосистеми програмного забезпечення називає процеси, оптимізовані для того, щоб більше не працювати у внутрішньоорганізаційних цілях, або, принаймні, бути набагато менш ефективними, і, отже, слід знайти альтернативи.

Вирішення цієї проблеми на високому рівні полягає у переході від централізованого до децентралізованого підходу. Традиційна розробка програмного забезпечення програмного забезпечення, як правило, більшою мірою передбачається централізованими механізмами, наприклад, круглим управлінням вимогами, розвитком архітектури, інтеграцією, забезпеченням якості та СКМ. На цих етапах усі активи, які були ввімкнені у великій системі, як команди, відповідальні за ці активи, повинні бути синхронізовані та координовані. Очевидно, що логічне рішення полягає в децентралізації цих актів і переході до композиційного підходу до розробки програмного забезпечення (рис. 3.20).

Однак наслідки децентралізованого, орієнтованого на цілісність, підходу до розробки програмного забезпечення є глибоким. Перша істотна зміна – координація, яка все ще необхідна системі, досягається за допомогою архітектури програмного забезпечення, а не процесів розробки програмного забезпечення. Повний розподіл системних, базових проектних рішень та інтерфейсів, визначених для систем, забезпечують контекст, в якому окремі команди можуть розробляти рішення, не потребуючи співпраці з іншими командами (рис. 3.21).

Рис. 3.20. Централізований підхід

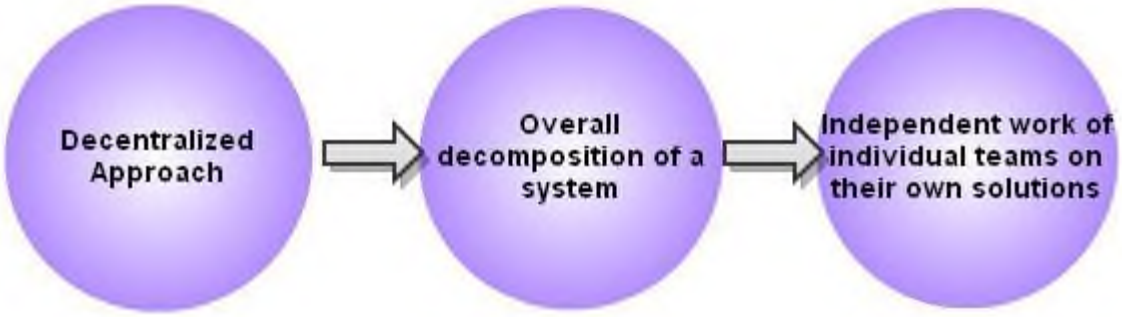

Рис.3.21. Децентралізований підхід

Архітектура забезпечує формалізацію правил функціональної сумісності, і, отже, команди можуть, в значній мірі, працювати самостійно [9].

Друга зміна – централізоване управління вимогами та планування замінюються за зростанням, керуються командою, сюжетами та уточненням вимог. Звичайно, така ситуація виникає між інженером-розробником платформи та зовнішніми інженерами-розробниками, але деякі компанії застосовують той самий підхід внутрішньо. Це означає, що кожна компонентна команда оголошує власний графік і вимоги, які звільняються в кінці наступного ітераційного циклу. Нова функціональність може впливати на надані та необхідні інтерфейси між компонентами, вимагаючи, щоб компонентна команда була адресована іншим командам для локального обговорення інтерфейсів. [9]

Заключним аспектом, який обговорюється в цьому розділі, *є* зміна ролі управління, що налаштовує *ON* та гарантії якості. Замість того, щоб зосередитись на повній функціональності системи, ці функції стосуються цілісності та вимог

до сумісності повернень. Це дозволяє проводити децентралізоване забезпечення якості та мінімізує силу, яку слід витратити. Друга головна перевага концентрації на сумісності з поверненням полягає в тому, що вона значно спрощує управління версіями: коли нова версія збирається, все більше і більше старих версій можна видалити, оскільки повертається сумісність. Звичайно, щоб досягти останнього, інтерфейси між компонентами повинні бути вузькими та враховувати мінімальне доповнення, наприклад, використовуючи підходи стилю веб-сервісу. [9]

Друга основна зміна у підході до розробки програмного забезпечення полягає в тому, що парадокс між швидкістю повної можливості розробки програмного забезпечення програмного забезпечення та проблемою частих вичерпань платформи потрібно ретельно оперувати.

Однією з проблем у лінійках програмного продукту з розширеним контекстом є зменшення частоти вичерпань платформи. Кількість і складність залежностей між різними компонентами та їх підключеними командами, як правило, така, що загальна сила для забезпечення якості вимагає гідних енергетичних витрат.

У попередньому розділі ми обговорювали важливість децентралізації та деякі випадки, коли маленькі команди можуть працювати дуже ефективно, навіть в умовах великих і складних систем.

Для платформ, де відсутня економічна модель для частих ітерацій усіх рівнів архітектури, слід враховувати різну частоту вичерпання для різних рівнів стека [9].

Третім і останнім основним напрямком впливу на методи розробки програмного забезпечення для організаційного переходу від лінійки програмного продукту до програмної екосистеми є зміна у володінні цілісністю продукту. Екосистема програмного забезпечення складається з платформи, створених продуктів та додатків, створених на платформі, яка розширює продукти за допомогою функціональних можливостей, розроблених зовнішніми інженерами-розробниками. Однак прямим наслідком цього підходу є те, що сторона, що робить функціонал повного рішення, вже не є компанією лінійки продуктів, а натомість – користувачем. Користувач визначає рішення, яке найкращим чином

підходить під його вимоги. Хоча це здається тривіальним у теорії, практично існує багато конфігурацій, які неможливо перевірити, і, отже, архітектура та повні вказівки повинні забезпечувати цілісність обраної функціональності. Друга проблема в цьому просторі підтримує непереборний досвід взаємодії з користувачем. Оскільки користувачі роблять власне рішення, що складається з елементів, створених різними сторонами, повний користувальницький досвід може легко постраждати, що призведе до менш привабливої пропозиції для клієнта. Компанія-інженер-розробник платформи повинна слугувати базовим досвідом для користувачів, що викликає занепокоєння. Хоча не існує рішення, яке б повністю вирішило цю проблему, компанія, розробник платформи, може надати важливу підтримку для зведення до нуля загальних проблем [9]. Оскільки користувачі роблять власне рішення, що складається з елементів, створених різними сторонами, повний користувальницький досвід може легко постраждати, що призведе до менш привабливої пропозиції для клієнта. Компанія-інженер-розробник платформи повинна слугувати базовим досвідом для користувачів, що викликає занепокоєння. Хоча не існує рішення, яке б повністю вирішило цю проблему, компанія, розробник платформи, може надати важливу підтримку для зведення до нуля загальних проблем. [9] Оскільки користувачі роблять власне рішення, що складається з елементів, створених різними сторонами, повний користувацький досвід може легко постраждати, що призведе до менш привабливої пропозиції для клієнта. Компанія-інженер-розробник платформи повинна слугувати базовим досвідом для користувачів, що викликає занепокоєння. Хоча не існує рішення, яке б повністю вирішило цю проблему, компанія, розробник платформи, може надати важливу підтримку для зведення до нуля загальних проблем. [9] компанія, розробник платформи, може надати суттєву підтримку для зведення до нуля загальних проблем. [9] компанія, розробник платформи, може надати суттєву підтримку для зведення до нуля загальних проблем (рис. 3.22).

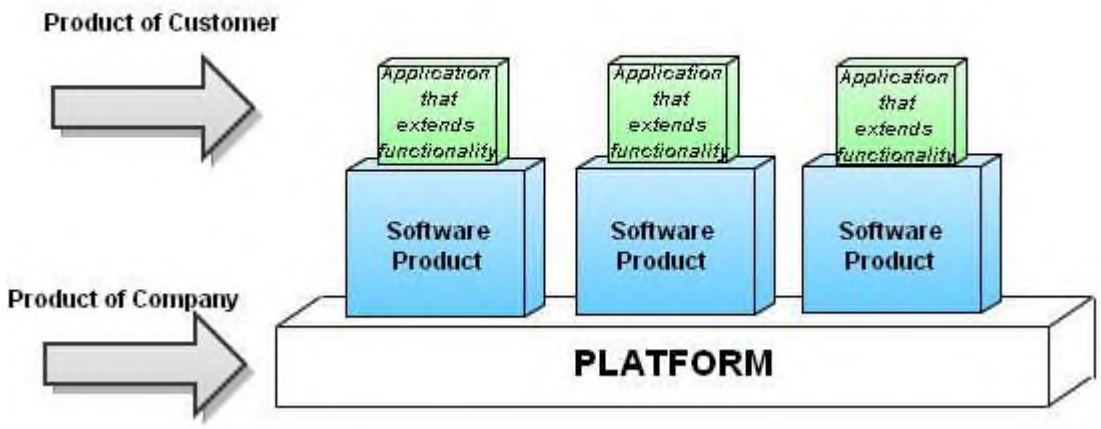

Рис. 3.22. Склад продукту

3.7. Висновки до розділу

В третьому розділі розкрито етапи розробки системи контролю доступу і управління системою контролю версій програмного забезпечення через корпоративну мережу (на прикладі банківської комп'ютерної мережі). Було описано мову програмування, яка використовувалась для розробки системи. Визначено необхідне апаратне забезпечення. Описано реалізацію алгоритмів системи. Розкрито функції модуля адміністрування та представлено загальний інтерфейс системи.

Також у цьому розділі реалізовано перехід від лінійки продуктів до програмної екосистеми.

Після відбору вибірки відповідного типу екосистеми програмного забезпечення було згадано питання трансформації лінійки продуктів у відкриту платформу, взаємних відносин інженерів-розробників та наслідків для розробки програмного забезпечення.

ВИСНОВКИ

В дипломній роботі в повній мірі виконано поставлену задачу – підвищено ефективність роботи системи контролю версій *web*-ресурсів в межах проектів компанії, за раунок зменшення часу на пошук і розгортання програмного забезпечення.

У цій дипломній роботі було використано підхід до *web*-ресурсів, як до програмної екосистеми і аналізувалась через поняття екосистеми програмного забезпечення, його походження, використання та причина концепції. Екосистема програмного забезпечення складається з набору програмних продуктів, фабрик, суб'єктів та комунікацій між ними, які дозволяють, підтримують та автоматизують дії та операції суб'єктів у пов'язаній соціальній чи діловій екосистемі.

Було визначено три типи програмних екосистем:

– на основі операційних систем;

– на основі прикладних програм;

– на основі кінцевого користувача.

Під час впровадження роботи ми створили ці екосистеми та дослідили їх на практиці. При зміні структури екосистеми ми проаналізували її поведінку. Це дозволяє робити прогнози і надалі будувати системи прогнозування екосистем програмного забезпечення.

Аналіз контролю версій програмних екосистем дозволяє компаніям-розробникам програмного забезпечення, які переходять від лінійки продуктів до програмних екосистем, вибрати та виправити відповідні шляхи, що, в свою чергу, дозволяє досягти успіху на ринку програмного забезпечення та задовольнити потреби користувачів.

Також ця робота може стати основою для подальшого вивчення контролю версій програмних екосистем в контексті екологічного підходу до програмного забезпечення.

Обговорювались два способи, за допомогою яких великий організатор програмних екосистем може управляти кластерами в своїх екосистемах, за допомогою моделей партнерства та членства. Ці моделі можуть бути використані для досягнення різних цілей, залежно від їх бізнес-моделі. Асоційовані моделі складаються з набору зобов'язань між власником моделі та учасником. Хоча моделі партнерства та членства відрізняються одна від одної, загальна їх структура однакова. Застосовуючи науку про дизайн, було створено концептуальний огляд, який охоплює структуру асоційованих моделей. У кожному зобов'язанні в рамках такої моделі учасник виконує одну або кілька ролей із заздалегідь визначеним набором переваг, вимог та витрат.

Це дослідження є важливим для повного розуміння того, що таке екосистеми та як ними можна керувати. Різні підходи та результуюча точка показують наслідки залежності в різних випадках взаємодії, але головна ідея фактора високого коефіцієнта успіху полягає в тому, що сильний взаємозв'язок та наявність усіх необхідних компонентів екосистем забезпечують необхідний результат при мінімальних витратах часу. Дослідження, якщо воно визначене, *є* основою для всебічного моделювання кожної системи, яка зацікавлена в обсязі домену. За допомогою простих інструментів та повної графічної моделі легко спостерігати за відсутніми посиланнями та акторами з їхніми послідовними елементами, а у разі потреби – створювати нові підсистеми, рекламувати нові компоненти та спостерігати за результатами.

У повсякденному житті ми не звертаємо особливої уваги на згадане вище питання. Швидше ми використовуємо програмне забезпечення, не замислюючись про загальну його структуру та спосіб його представлення нам, кінцевим користувачам. Ось чому у декількох компаній зменшується дохід, оскільки користувачі здебільшого не забезпечують грошовий фон для власних обраних програм, для розробників та постачальників, які вкладають свої здібності у презентацію кінцевого готового продукту. Екосистеми програмного забезпечення – це не лише взаємоописана система опису, це також основа для оживлення життєво важливих процесів.