Ю. А. Кузнецова, І. Б. Туркін

ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2017

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Ю. А. Кузнецова, І. Б. Туркін

ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

Харків «ХАІ» 2017

УДК 004.4
К63

Рецензенти: д-р техн. наук, проф. Н. В. Шаронова;
д-р техн. наук, проф. М. М. Корабльов

Кузнецова, Ю. А.
К63 Інформаційні технології розроблення програмного забезпечення [Електронний ресурс] : навч. посіб. / Ю. А. Кузнецова, І. Б. Туркін. – Харків. : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. Ін-т», 2017. – 151 с.

Наведено основні відомості про системи контролю версій, подано необхідний словник термінів для його застосування під час командного розроблення програмного забезпечення й використання веб-сервісів для хостингу проектів і їхнього спільного розроблення, що основані на системі контролю версій (Mercurial, Git або ін.). Викладено узагальнену архітектуру систем контролю версій. Підкреслено необхідність планування проекту та відстеження змін у ньому, а також злиття версій вихідного коду проекту. Описано перелік переваг безперервної інтеграції проекту, а також застосування серверів збирання. Показано необхідність застосування Unit-тестування, а також патернів впровадження залежностей класів. Висвітлено основні відомості про делегати, анонімні методи, лямбда-вирази, а також переваги використання Moq-об'єктів. Запропоновано приклади виділення тестових випадків. Показано на прикладах переваги версіонування програмних продуктів і створення інсталяторів. Наведено основні відомості про сертифікати прикладних програм, центри сертифікації, а також показано на прикладах підпис інсталятора сертифікатом.

Для студентів спеціальності 121 «Інженерія програмного забезпечення». Також може бути корисним студентам, аспірантам і технічним фахівцям, які прагнуть одержати базові знання про командне розроблення промислового програмного забезпечення з використанням систем контролю версій.

Іл. 96. Табл. 9. Бібліогр.: 46 назв

УДК 004.4

© Кузнецова Ю. А., Туркін І. Б., 2017
© Національний аерокосмічний університет ім. М. Є. Жуковського «Харківський авіаційний інститут», 2017

ЗМІСТ

ВСТУП...4
СЛОВНИК ТЕРМІНІВ...6
1 СИСТЕМИ УПРАВЛІННЯ ВЕРСІЯМИ...9
1.1 Огляд систем контролю версій...55
1.2 Система управління версіями для Visual Studio 67
2 БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ ПРОЕКТУ 74
3 ТЕСТУВАННЯ ЯК ЕТАП БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ 84
4 UNIT-ТЕСТУВАННЯ. ПАТЕРНИ ВПРОВАДЖЕННЯ ЗАЛЕЖНОСТЕЙ КЛАСІВ 89
5 UNIT-ТЕСТУВАННЯ У VISUALSTUDIO. АНАЛІЗ ПОКРИТТЯ КОДУ 97
6 ДЕЛЕГАТИ, АНОНІМНІ МЕТОДИ, ЛЯМБДА-ВИРАЗИ. ВИКОРИСТАННЯ MOQ-ОБ'ЄКТІВ...115
7 ПРИКЛАД ВИДІЛЕННЯ ТЕСТОВИХ ВИПАДКІВ............................120
8 ВЕРСІОНУВАННЯ ПРОДУКТІВ. СТВОРЕННЯ ІНСТАЛЯТОРІВ 131
9 СЕРТИФІКАТИ ДОДАТКІВ. ЦЕНТРИ СЕРТИФІКАЦІЇ. ПІДПИС...........142
БІБЛІОГРАФІЧНИЙ СПИСОК..147

ВСТУП

Ситуація, в якій електронний документ за час свого існування зазнає ряд змін, досить типова. При цьому часто буває важливо мати не тільки останню версію, але і кілька попередніх. У найпростішому випадку можна просто зберігати кілька варіантів документа, нумеруючи їх відповідним чином. Такий спосіб є неефективним (доводиться зберігати кілька практично ідентичних копій), потребує підвищеної уваги і дисципліни і часто призводить до помилок, тому було розроблено засоби для автоматизації цієї роботи.

Традиційні системи управління версіями використовують централізовану модель, коли є єдине сховище документів, кероване спеціальним сервером, який і виконує велику частину функцій з управління версіями. Користувач, який працює з документами, повинен спочатку отримати потрібну йому версію документа зі сховища; зазвичай створюється локальна копія документа, так звана «робоча копія». Може бути отримана остання версія або будь-яка з попередніх, яку можна вибрати за номером версії або датою створення, іноді й за іншими ознаками. Після того, як у документ внесено потрібні зміни, нову версію поміщають у сховище. На відміну від простого збереження файла, попередню версію не стирають, а також залишають у сховищі, і вона може бути звідти отримана у будь-який час. Сервер може використовувати дельта-компресію – такий спосіб зберігання документів, при якому зберігаються тільки змінення між послідовними версіями, що дозволяє зменшити обсяг збережених даних. Оскільки зазвичай найбільш затребуваною є остання версія файла, система може при збереженні нової версії зберігати її цілком, замінюючи у сховищі останню, раніше збережену версію, на різницю між цією і останньою версією. Деякі системи (наприклад, ClearCase) підтримують збереження версій обох видів: більшість версій зберігається у вигляді дельт, але періодично (за спеціальною командою адміністратора) зберігаються версії усіх файлів у повному вигляді; такий підхід забезпечує максимально повне відновлення історії у разі пошкодження репозитарія (сховища).

Іноді нова версія створюється непомітно для користувача (прозоро) або прикладною програмою, що має вбудовану підтримку такої функції, або за рахунок використання спеціальної файлової системи. У цьому випадку користувач просто працює з файлом, як завжди, і при збереженні файла автоматично створюється нова версія.

Часто буває так, що над одним проектом одночасно працюють кілька людей. Якщо дві людини змінюють один і той же файл, то одна з них може випадково скасувати зміни, внесені іншою. Системи управління версіями відстежують такі конфлікти і пропонують засоби їх вирішення. Більшість систем може автоматично об'єднати (злити) зміни, зроблені

різними розробниками. Однак таке автоматичне об'єднання змін зазвичай є можливим тільки для текстових файлів і за умови, що змінювалися різні (неперетинні) частини цього файла. Таке обмеження пов'язане з тим, що більшість систем управління версіями орієнтовано на підтримку процесу розроблення програмного забезпечення, а вихідні коди програм зберігаються в текстових файлах. Якщо автоматичне об'єднання виконати не вдалося, система може запропонувати вирішити проблему вручну.

Часто виконати злиття неможливо ні в автоматичному, ні в ручному режимі, наприклад, якщо формат файла невідомий або дуже складний. Деякі системи управління версіями дають можливість заблокувати файл у сховищі. Блокування не дозволяє іншим користувачам отримати робочу копію або перешкоджає зміні робочої копії файла (наприклад, засобами файлової системи) і забезпечує, таким чином, винятковий доступ тільки тому користувачеві, який працює з документом.

Кожна система управління версіями (VCS – Version Control System) має свої специфічні особливості в наборі команд, порядку роботи користувачів і адмініструванні. Проте загальний порядок роботи для більшості VCS є абсолютно стереотипним. Тут передбачається, що проект, яким би він не був, вже існує, і на сервері розміщено його репозитарій, до якого розробник отримує доступ.

СЛОВНИК ТЕРМІНІВ

Конфлікт (conflict) – ситуація, коли кілька користувачів змінили одну і ту ж ділянку документа. Конфлікт виявляється, коли один користувач зафіксував свої зміни, а другий намагається зафіксувати, і система сама не може коректно злити конфліктуючі зміни. Оскільки програма може бути недостатньо розумною для того, щоб визначити, яка зміна є «коректною», другому користувачеві потрібно самому вирішити конфлікт (resolve).

Основна версія (head) – найсвіжіша версія для гілки / стовбура, що знаходиться в сховищі. Скільки гілок, стільки основних версій.

Злиття (merge, integration) – об'єднання незалежних змін в єдину версію документа. Здійснюється, коли двоє людей змінили один і той же файл або при перенесенні змін з однієї гілки на іншу.

Перенесення точки розгалуження (rebase) (версії, від якої починається гілка) на більш пізню версію основної гілки. Наприклад, після випуску версії 1.0 проекту в стовбурі триває доопрацювання (виправлення помилок, доороблення наявного набору функцій), одночасно починається робота над новою функціональністю в новій галузі. Через деякий час в основній гілці відбувається випуск версії 1.1 (з виправленнями); тепер бажано, щоб гілка розроблення нової функціональності включала зміни, що відбулися в стовбурі. Взагалі, це можна зробити базовими засобами, за допомогою злиття (merge), виділивши набір змін між версіями 1.0 і 1.1 і злив його в гілку. Але при наявності в системі підтримки перебазування гілки ця операція робиться простіше, однією командою: за командою rebase (з параметрами: гілкою і новою базовою версією) система самостійно визначає потрібні набори змін і виробляє їх злиття, після чого для гілки базовою стає версія 1.1; при подальшому злитті гілки зі стовбуром система не розглядає повторно зміни, внесені між версіями 1.0 і 1.1, оскільки гілка логічно вважається виділеною після версії 1.1.

Сховище документів (Repository, Depot) – місце, де система управління версіями зберігає всі документи разом з історією їх змінення та іншою службовою інформацією.

Версія документа (Revision). Системи управління версіями розрізняють версії за номерами, які надаються автоматично.

Відкладання змін (Shelving). Надана деякими системами можливість створити набір змін (changeset) і зберегти його на сервері без фіксації (commit'а). Відкладений набір змін доступний для читання іншим учасникам проекту, але до спеціальної команди не входить в основну гілку. Підтримка відкладання змін дає можливість користувачам зберігати незавершені роботи на сервері, не створюючи для цього окремих гілок.

Мітка (Tag, Label) – це символічне ім'я для групи документів, яке можна надати певній версії документа. Мітка описує не тільки набір імен

файлів, але і версію кожного файла. Версії занесених до мітки документів можуть відноситися до різних моментів часу.

Стовбур (Trunk, Mainline, Master) – головна галузь розроблення проекту. Політика роботи зі стовбуром може відрізнятися від проекту до проекту, але в цілому вона така: більшість змін вноситься у стовбур; якщо потрібна серйозна зміна, здатна призвести до нестабільності; створюється гілка, яка зливається зі стовбуром, коли нововведення буде достатньою мірою випробувано; перед випуском чергової версії створюється «релізна» гілка, в яку вносяться тільки виправлення.

Синхронізація (Update, Sync.) робочої копії до деякого заданого стану сховища. Найчастіше ця дія означає оновлення робочої копії до актуального стану сховища. Однак, за необхідності, можна синхронізувати робочу копію і до попереднього його стану, ніж поточний.

Робоча (локальна) копія документів **(Working copy).**

У документації Visual Studio використовується ряд термінів для опису функцій і понять системи управління версіями [1]. У таблиці 1.1 подано деякі загальні терміни.

Таблиця 1.1 – Словник термінів

Термін	Значення
Основна версія	Серверна версія файла, з якої виймається локальна версія
Прив'язка	Інформація, яка зіставляє робочу папку рішення або проекту на диску у відповідній папці в базі даних
Розгалуження	Процес створення нової версії або гілки для спільно використовуваного файла або проекту, що знаходиться під управлінням системи управління версіями. Коли гілка створена, дві версії, що знаходяться під управлінням системи управління версіями, будуть мати загальний журнал до певної точки, який відрізняється від журналів після цієї точки
Конфлікт	Наявність двох або декількох відмінних змін на тому самому рядку коду, коли два або більше розробників витягли і змінили один і той же файл
Підключення	Поточна лінія передачі даних між системою управління версіями (наприклад, Visual Studio) і сервером бази даних системи управління версіями
База даних	Сховище, де розміщуються основні копії, журнал, структури проектів і призначена для користувача інформація. Проект завжди міститься в одній базі даних. Кілька проектів можуть зберігатися в одній базі даних і можуть використовувати кілька баз даних. Інші терміни, які використовують для бази даних, – це репозитарій і сховище

Закінчення таблиці 1.1

Термін	Значення
Журнал	Використовується для запису змін у файлі відразу ж після його додавання в систему управління версіями. За допомогою управління версіями можна повернутися в будь-яку точку в журналі файла і відновити його в тому вигляді, в якому він був у цій точці
Мітка	Ім'я, визначене користувачем, яке приєднане до конкретної версії елемента з керуванням версіями
Локальна копія	Файл у робочій папці користувача, в якому зберігаються зміни, до тих пір, поки не проводиться повернення. Локальна копія іноді визначається як робоча копія
Головна копія	Найчастіше – це повернута версія файла з контролем версій, на противагу до локальної копії файла в робочій папці користувача. Іншими термінами для головної копії є серверна версія і версія бази даних
Злиття	Процес об'єднання в новій версії файла відмінностей у двох або більше змінених версіях файла. Злиття можна застосовувати до різних версій одного і того ж файла або до змін, зроблених у тому ж самому файлі
Загальний файл	Файл, який має версії, що знаходяться в декількох розташуваннях системи управління версіями. Інші терміни для загального файла – це копія і ярлик
Корінь рішення	Порожня папка в базі даних, яка містить всі елементи рішення з керуванням версіями. За замовчуванням це папка <імя_решенія> .root
Супер-уніфікований корінь	Віртуальний контейнер, в якому розміщуються всі проекти і файли в рішенні з керуванням версіями. Наприклад, [SUR]: \ – це суперуніфікований корінь рішення з керуванням версіями, який містить проекти, розташовані в [SUR]: \ C: \ Solution \ ProjOne і [SUR]: \ D: \ ProjTwo
Уніфікований корінь	Шлях до батьківського каталогу для всіх робочих папок і файлів у рішенні або проект з керуванням версіями. Наприклад, C: \ Solution – це уніфікований корінь рішення з керуванням версіями, який містить файли, розміщені в C: \ Solution, C: \ Solution \ ProjOne і C: \ Solution \ ProjTwo
Робоча папка	Сховище, де зберігаються локальні копії елементів з керуванням версіями (зазвичай на комп'ютері користувача). Інший термін для робочої папки – робоча область

1 СИСТЕМИ УПРАВЛІННЯ ВЕРСІЯМИ

Визначення

Система управління версіями (від англ. *Version Control System, VCS*, або *Revision Control System*) – програмне забезпечення для полегшення роботи зі змінною інформацією. Система управління версіями дозволяє зберігати кілька версій одного і того ж документа, за необхідності повертатися до попередніх версій, визначати, хто і коли зробив ту чи іншу зміну, і багато іншого.

Також під **системою управління (контролю) версій (СКВ)** розуміється механізм збереження проміжних станів коду розроблюваного програмного забезпечення, тобто за допомогою цієї системи програміст може управляти своїми файлами в часі: дивитися історію змін файлів і каталогів, повертатися до попередніх версій коду, об'єднувати кілька версій файла.

Можливості

Системи управління версіями дають можливість:

– створювати різні варіанти одного документа, т. н. гілки, із загальною історією змін до точки розгалуження і з різними історіями – після неї;

– дізнатися, хто і коли додав або змінив конкретний набір рядків у файлі;

– вести журнал змін, в який користувачі можуть записувати пояснення про те, що і чому вони змінили в цій версії;

– контролювати права доступу користувачів, дозволяючи або забороняючи читання або зміну даних, залежно від того, хто запитує цю дію.

Область застосування систем управління версіями

Основною областю застосування контролю версій є колективне розроблення чогось (найчастіше це програми, але область застосування програмуванням не обмежується). Однак і для розробника-одинаки контроль версій може бути корисним.

Такі системи широко використовуються при розробленні програмного забезпечення для зберігання вихідних кодів програми, що розробляється. Однак їх можна з успіхом застосовувати і в інших областях, в яких ведеться робота з великою кількістю безперервно змінюваних електронних документів. Зокрема, системи управління версіями застосовуються в САПР, зазвичай, у системах управління даними про виріб (PDM). Управління версіями використовується в інструментах конфігураційного управління (*Software Configuration Management Tools*).

Основні поняття

Сховище (Repository) – це місце централізованого зберігання даних. Сховище розташовано на сервері та має вигляд дерева файлів.

Робоча копія (Working Copy) – це копія сховища (або його частини) на робочому місці користувача.

Виправлення (Revision) – це стан сховища у певні моменти часу. Процес створення правки називається «Фіксація». Кожна правка має номер, який присвоюється при фіксації.

Гілка (Branch) – це пов'язана послідовність ревізій (changeset), що є окремим напрямком розроблення; цей напрямок розроблень, що існує незалежно від інших напрямків, але має спільну з ними історію, фактично являє собою копію проекту (або його частини) у певний момент часу і сукупності фіксацій змін. Найчастіше гілки використовуються для зберігання різних релізів проекту. Крім цього, гілки застосовуються для ізоляції групи правок, які можуть порушити працездатність усього коду.

На рисунку 1.1 наведено приклад розгалуження в Tortoise HG – графічний фронтенд для системи контролю версій Mercurial (клієнт для Mercurial).

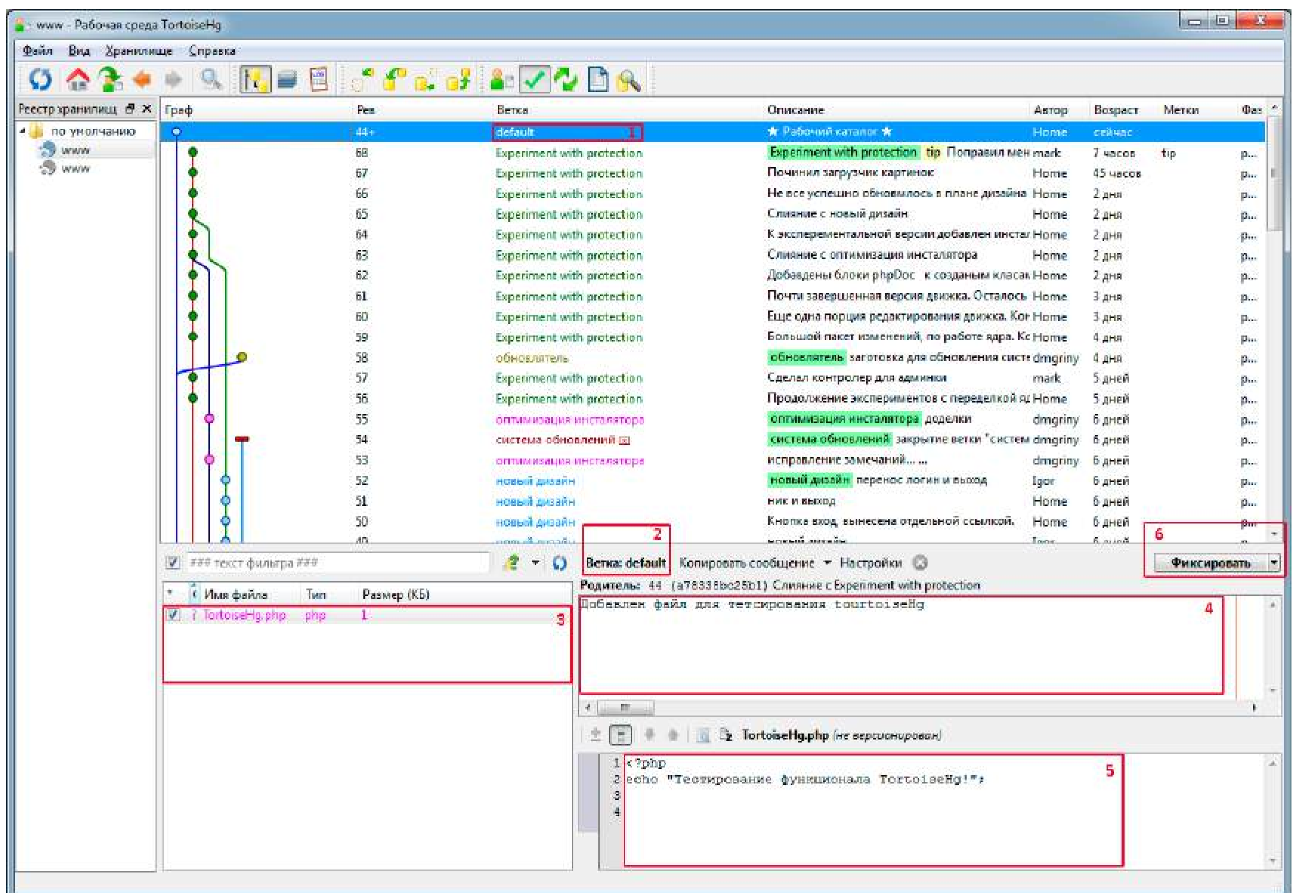


Рисунок 1.1 – Приклад розгалуження в Tortoise HG

Робочі копії. Робоча копія, наприклад у СКВ Subversion, являє собою звичайне дерево каталогів на локальному комп'ютері, що містить набір файлів. Є можливість на свій розсуд редагувати ці файли, і, якщо це вихідні коди, то можна звичайним способом скомпілювати з них програму. Таким чином, Ваша робоча копія – це Ваш особистий робочий простір.

Робоча копія містить кілька додаткових файлів, що створені та обслуговуються Subversion, які допомагають їй при виконанні цих команд. Зокрема, кожен каталог Вашої робочої копії містить підкаталог з ім'ям .svn, який називається службовим каталогом робочої копії. Файли у службовому каталозі допомагають Subversion визначити, які файли робочої копії містять неопубліковані зміни, а які файли застаріли по відношенню до файлів інших учасників.

У сховищі може перебувати кілька проектів. У цьому випадку офіційно рекомендується в кореневому каталозі сховища створювати для кожного проекту свій підкаталог, в який мають входити три підкаталоги:

/ trunk – основна гілка розроблення. У ній ведеться більша частина розробки: впроваджуються нові функції, виправляються помилки;

/ branches – різні гілки. Тут можуть зберігатися попередні стабільні релізи[1] або розташовуватися набори виправлень, що вирішують якусь певну задачу. Загалом, в цьому каталозі зберігаються всі гілки, крім / trunk;

/ tags – різні мітки. Міткою називається пойменована правка. Фізично являє собою просту копію частини сховища у певний момент часу. На відміну від гілки, в мітці не можна виконувати фіксацію,тобто змінювати її. Найбільш часто мітки використовуються для позначення релізів проекту.

Початок роботи з проектом

Першою дією, яку повинен виконати розробник, є отримання робочої копії проекту або тієї його частини, з якою доведеться працювати. Ця дія, яка виконується за допомогою стандартної команди вилучення версії (checkout або clone) або спеціальної команди, фактично робить те ж саме діяння. Розробник задає версію, яка має бути скопійована. За замовчуванням зазвичай копіюється остання (або вибрана адміністратором як основна) версія.

За командою вилучення встановлюється з'єднання із сервером, і проект (або його частина – один з каталогів з підкаталогами) у вигляді дерева каталогів і файлів копіюється на комп'ютер розробника.

1 Реліз, або RTM (англ. Release to manufacturing - промислове видання) – видання продукту, готового до тиражування. Це стабільна версія програми, що пройшла всі попередні стадії, в яких виправлені основні помилки, але існує ймовірність появи нових, раніше не помічених, помилок. RTM передує загальній доступності (GA), коли продукт випущений для громадськості. Реліз ПЗ (від англ. Release – випуск) – випуск програми, коду або бібліотеки – готового для використання продукту. Зазвичай він містить усі оновлення, виправлення і є версією, готовою для використання кінцевим споживачем.

Звичайною практикою є дублювання робочої копії: крім основного каталогу з проектом на локальний диск (або в окремий, спеціально вибраний каталог, або в системні підкаталоги основного дерева проекту) додатково записується ще одна його копія. Працюючи з проектом, розробник змінює лише файли основної робочої копії. Друга локальна копія зберігається як еталон, дозволяючи в будь-який момент без звернення до сервера визначити, які зміни внесені в конкретний файл або проект в цілому і від якої версії «відгалуджилась» робоча копія. Як правило, будь-яка спроба ручної зміни цієї копії призводить до помилок у роботі програмного забезпечення VCS.

Цикл при використанні СКВ

При деяких варіаціях, що визначаються особливостями системи і деталями прийнятого бізнес-процесу, звичайний цикл роботи розробника протягом робочого дня має такий вигляд.

Оновлення робочої копії. Цей крок необхідний для приведення робочої копії до актуального стану.

У міру внесення змін до основної версії проекту робоча копія на комп'ютері розробника старіє: розбіжність її з основною версією проекту збільшується. Це підвищує ризик виникнення конфліктних змін. Тому зручно підтримувати робочу копію в стані, максимально наближеному до поточної основної версії. Для цього розробник виконує операцію поновлення робочої копії (update) наскільки можливо часто (реальна частота оновлень визначається частотою внесення змін, що залежить від активності розроблення і числа розробників, а також від часу, витраченого на кожне оновлення – якщо його багато, розробник змушений обмежувати частоту оновлень, щоб не втрачати час).

Внесення змін. Цей крок є основним в роботі. На цьому кроці вносяться зміни у файли і / або в структуру файлів проекту.

Розробник модифікує проект, змінюючи вхідні в нього файли в робочій копії відповідно до проектного завдання. Ця робота проводиться локально і не вимагає звернень до сервера VCS.

Аналіз змін. Цей крок служить для контролю зроблених змін перед фіксацією їх у сховищі.

Злиття змін, виконаних іншими, зі своєю робочою копією. Цей крок потрібен для гарантії, що немає конфліктів між локальними змінами і змінами інших розробників

Фіксація змін. Цей крок служить для збереження змін робочої копії у сховищі.

Завершивши черговий етап роботи над завданням, розробник фіксує (commit) свої зміни і передає їх на сервер (або в основну гілку, якщо робота над завданням повністю завершена, або в окрему гілку розроблення цього завдання). VCS може вимагати від розробника перед фіксацією обов'язково виконати оновлення робочої копії. За наявності в

системі підтримки відкладених змін (shelving) зміни можуть бути передані на сервер без фіксації. Якщо затверджена політика роботи в VCS це дозволяє, то фіксація змін може проводитися не щодня, а лише по завершенні роботи над завданням; в цьому випадку до завершення роботи всі пов'язані із завданням зміни зберігаються тільки в локальній робочій копії розробника.

Таким чином, зміни, зроблені однією людиною, стають доступними для усіх учасників проекту.

Базові принципи розроблення ПЗ у VCS

Порядок використання системи управління версіями в кожному конкретному випадку визначається технічними регламентами та правилами, прийнятими в конкретній фірмі або організації, що розробляє проект. Проте, загальні принципи правильного використання VCS нечисленні та єдині для будь-яких розробок і систем управління версіями:

– будь-які робочі, тестові або демонстраційні версії проекту збирають тільки зі сховищ системи. «Персональні» збирання, що включають ще незафіксовані зміни, можуть робити тільки розробники для цілей проміжного тестування. Таким чином, гарантується, що репозитарій містить усе необхідне для створення робочої версії проекту;

– поточна версія головної гілки завжди коректна. Не допускається фіксація в головній гілці неповних або таких, що не пройшли хоча б попереднього тестування змін. У будь-який момент збирання проекту, проведеного з поточної версії, має бути успішним;

– будь-яка значуща зміна має бути оформлена як окрема гілка. Проміжні результати роботи розробника фіксуються у цій гілці. Після завершення роботи над зміною гілка об'єднується зі стовбуром. Винятки допускаються лише для дрібних змін, робота над якими ведеться одним розробником протягом не більше ніж одного робочого дня;

– версії проекту позначаються тегами. Виділена і позначена тегом версія більш ніколи не змінюється.

Архітектура систем контролю версій

З однієї сторони схеми зображено сховище Subversion (рисунок 1.2), в якому зберігається інформація з версіями. На протилежній стороні показано програму-клієнт Subversion, яка управляє локальними відбитками різних фрагментів цих даних (також званими «робочими копіями»). Між цими сторонами прокладено різні маршрути, що проходять через різні прошарки доступу до сховища. Деякі з цих маршрутів використовують комп'ютерні мережі й мережні сервери, щоб

досягти сховища, у той час як інші маршрути мережі не потребують і ведуть до сховища безпосередньо [2].

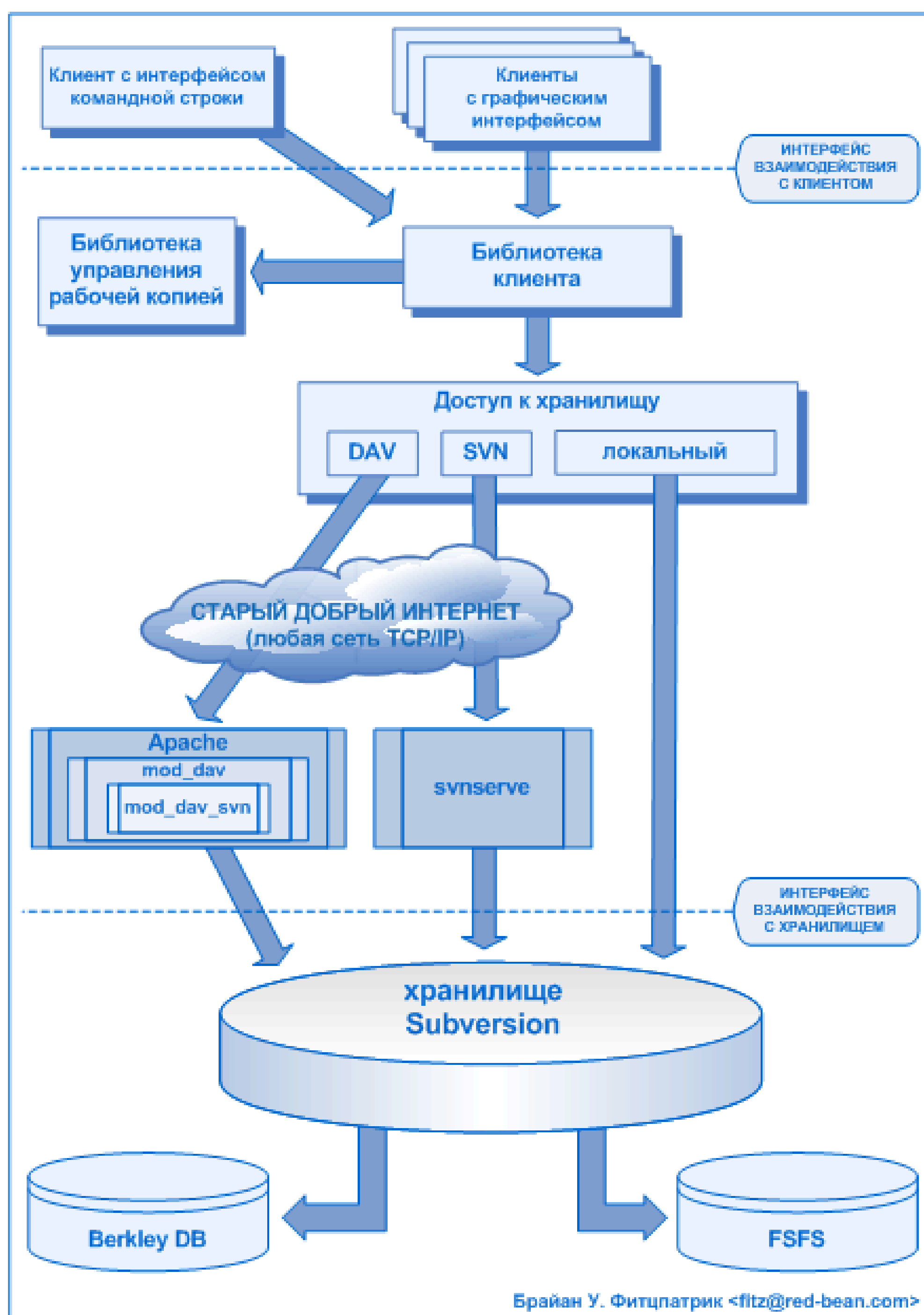


Рисунок 1.2 – Архітектура СКВ Subversion

Subversion є централізованою системою для спільного використання інформації. За своїм призначенням – це сховище є центром зберігання даних.

Subversion запам'ятовує кожну внесену зміну: будь-яку зміну будь-якого файла, так само як зміни в самому дереві каталогів, такі, як додавання, видалення та розпізнавання файлів і каталогів.

При читанні даних зі сховища клієнт зазвичай бачить тільки останню версію дерева файлів. Але клієнт також має можливість переглянути попередні стани файлової системи.

Клієнти підключаються до сховища і читають або змінюють ці файли (рисунок 1.3). Записуючи дані, клієнт робить інформацію доступною для інших; читаючи дані, клієнт отримує інформацію від інших [3].

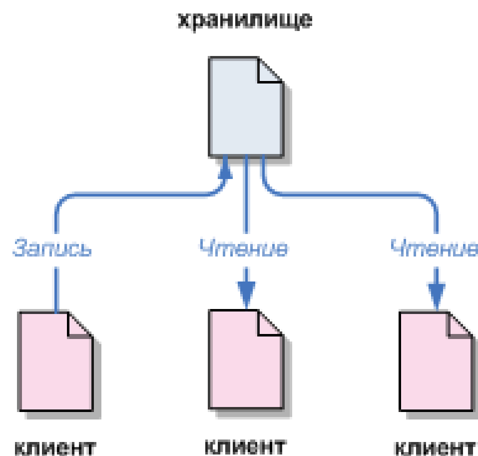


Рисунок 1.3 – Типова клієнт-серверна система

Моделі версіонування [4]

Основним завданням системи управління версіями є забезпечення спільного редагування та використання інформації.

Проблема втрати змін. Приклад. Припустимо таку ситуацію: в компанії є два співробітника, що працюють над одним проектом: Ігор і Світлана. Вони одночасно вирішили поправити один і той же файл. Якщо Ігор збереже свої зміни першим, тоді Світлана (зберігшись декількома секундами пізніше) може ненавмисно їх переписати своєю новою версією файла. Незважаючи на те, що версія Ігоря не може бути втрачена назавжди (бо система запам'ятовує всі зміни), внесених Ігорем правок не буде в новій версії файла Світлани, адже вона їх ніколи не бачила. Робота Ігоря фактично втрачена – або, принаймні, відсутня в останній версії файла (рисунок 1.4) [5]. А такої ситуації слід уникати.

1 Модель «Блокування-Зміна-Розблокування»

Для того, щоб кілька авторів не заважало працювати один одному, багато систем управління версіями застосовують модель «Блокування-Зміна-Розблокування». Ця модель забороняє одночасне редагування файла декількома користувачами. Ексклюзивність доступу гарантується блокуваннями. У такій системі сховище дозволяє вносити зміни у файл тільки одній людині за раз.

Приклад. До того, як Ігор зможе внести зміни у файл, він повинен спочатку його заблокувати. Блокування файла подібно взяттю книги в бібліотеці: якщо Ігор заблокував файл, Світлана не зможе змінити його – сховище відхилить запит на блокування файла. Все, що вона зможе, –

це прочитати файл і чекати, поки Ігор закінчить свою роботу і зніме блокування. Після того, як Ігор розблокує файл, Світлана зможе заблокувати його і внести свої зміни (рисунок 1.5) [6].

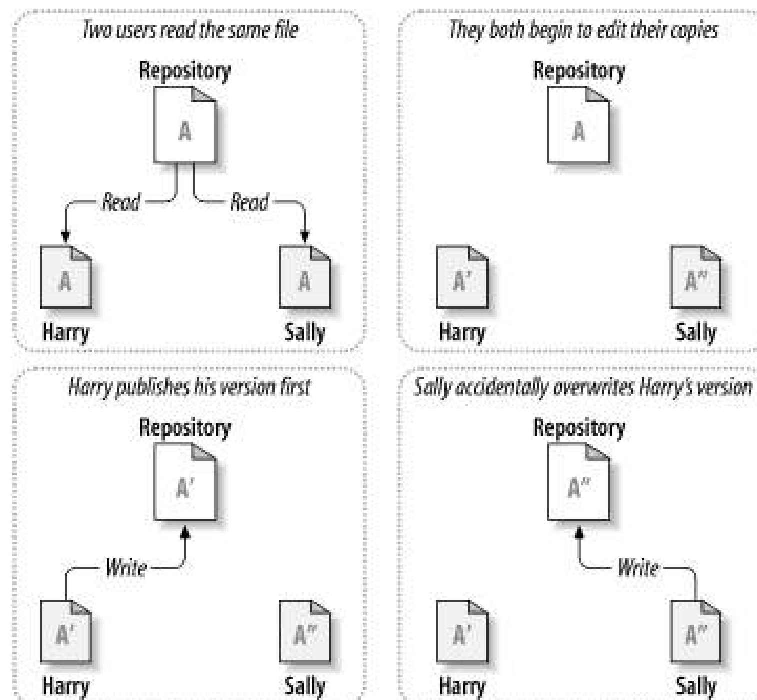


Рисунок 1.4 – Графічний приклад втрати змін

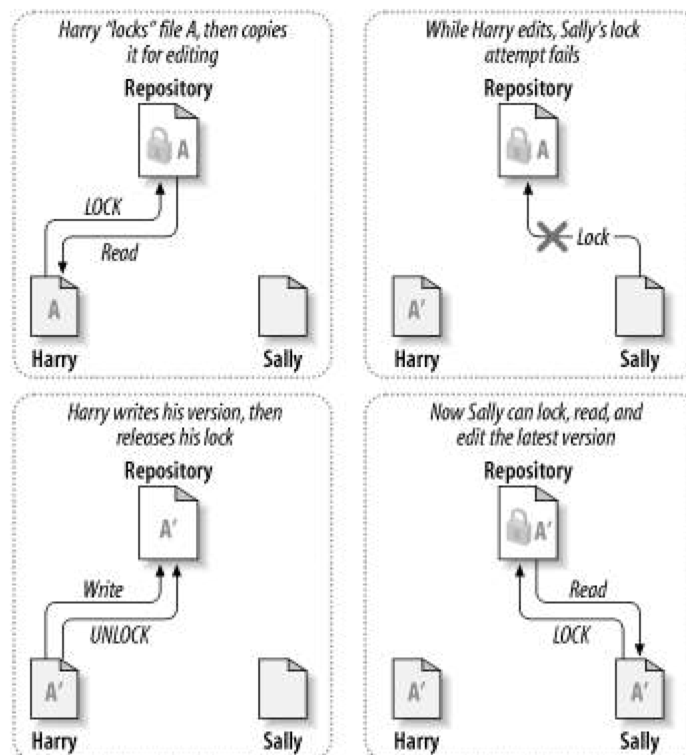


Рисунок 1.5 – Модель «Блокування-Зміна-Розблокування»

Проблема з моделлю «Блокування-Зміна-Розблокування» полягає в її жорстокості – ця модель може створити такі незручності користувачам [7]:

– блокування може спричинити адміністративні проблеми. Іноді Ігор, заблокувавши файл, забуває про це. Тим часом, Світлана все ще чекає, коли вона зможе почати редагування файла. А потім Ігор іде у відпустку. Тепер Світлана для того, щоб зняти блокування, спричинене Ігорем, мусить звернутися до адміністратора. Ситуація призводить до непотрібної втрати часу;

– блокування може спричинити зайву черговість. Якщо Ігор редагує початок великого файла, а Світлана хоче підправити кінець? Ці зміни взагалі не перетинаються. Вони могли б легко працювати одночасно, і ніякої шкоди це б не принесло (припускаючи коректне злиття змін);

– блокування може викликати помилкове відчуття безпеки. Припустимо, що Ігор заблокував і редагує файл А, в той час, як Світлана заблокувала і редагує файл Б. Але якщо файли А і Б залежать один від одного, то зміни, зроблені в кожному файлі, не сумісні. Раптово А і Б перестають працювати разом. Блокуюча система безсила в запобіганні проблеми – замість цього вона забезпечила помилкове відчуття безпеки. Ігор зі Свєтою запросто можуть вирішити, що, блокуючи свій файл, кожен починає безпечне ізольоване завдання, і це перешкоджає завчасному обговоренню їхніх несумісних змін.

2 Модель «Копіювання-Зміна-Злиття»

Subversion, CVS та інші системи управління версіями використовують модель «Копіювання-Зміна-Злиття» замість блокування. У цій моделі клієнт кожного користувача зчитує зі сховища проект і створює персональну робочу копію – локальне відображення файлів і каталогів сховища. Після цього користувачі працюють, одночасно змінюючи свої особисті копії. Зрештою, особисті копії зливаються в нову, фінальну версію. Зазвичай система управління версіями виконує злиття автоматично, але, природно, в загальному випадку необхідна присутність людини.

Приклад. Ігор і Світлана створили свої робочі копії одного і того ж проекту. Вони працюють одночасно і вносять зміни у файл А у своїх робочих копіях. Світлана першою зберігає свої зміни у сховищі. Потім, коли Ігор намагається зберегти свої, централізоване сховище вільно інформує його, що його файл А застарів. Тобто, файл А у сховищі був змінений з тих пір, як Ігор отримав його. Тоді Ігор виконує злиття (merge) будь-яких змін сховища зі своєю робочою копією. Ймовірно, що зміни Світлани не перетинаються з його власними, і, оскільки тепер його

робоча копія містить обидва набори змін, він записує її назад у сховище (рисунки 1.6 – 1.7) [8].

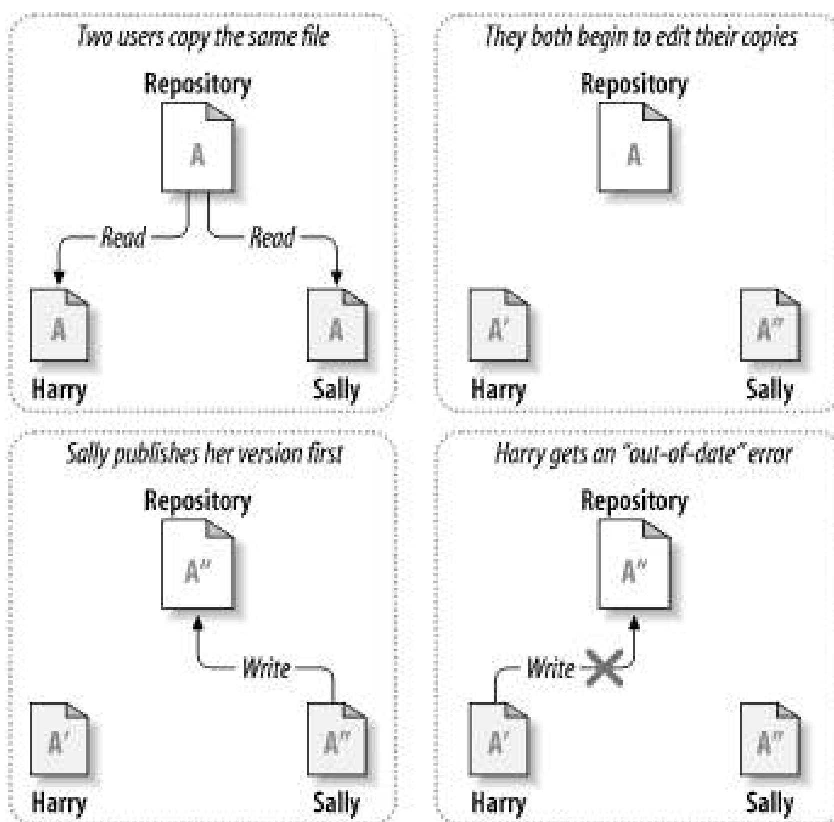


Рисунок 1.6 – Модель «Копіювання-Зміна-Злиття»

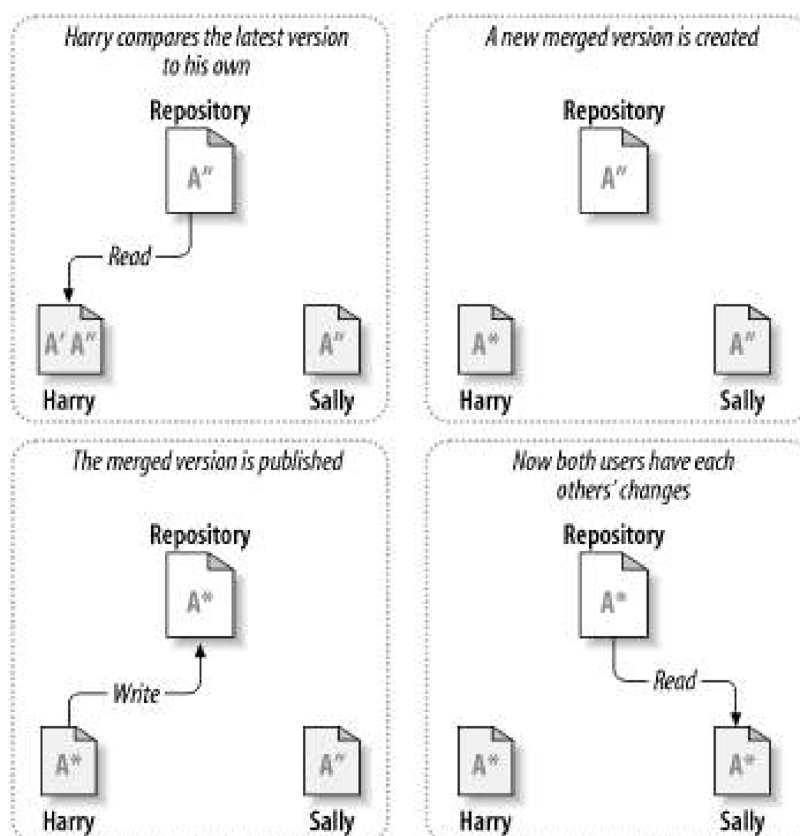


Рисунок 1.7 – Модель «Копіювання-Зміна-Злиття». Продовження

У цій моделі кожен призначений для користувача клієнт звертається до сховища проекту і створює персональну робочу копію – локальне відображення файлів і каталогів сховища. Після цього користувачі працюють одночасно і незалежно один від одного, змінюючи свої особисті копії. Зрештою, особисті копії зливаються в нову, підсумкову версію.

Операція svn commit публікує зміни будь-якої кількості файлів і каталогів за одну атомарну операцію. У своїй робочій копії Ви можете змінювати вміст файлів, створювати, видаляти, перейменовувати і копіювати файли і каталоги, а потім зафіксувати всі зміни за одну атомарному транзакцію.

Правки

Під «атомарною транзакцією» розуміється таке: або в сховище вносяться всі зміни повністю, або вони не вносяться взагалі. Subversion поводиться так, беручи до уваги можливі програмні збої, системні збої, проблеми з мережею, а також неправильні дії користувача.

Кожен раз, коли відбувається фіксація, створюється новий стан файлової системи, який називається правкою. Кожна правка отримує унікальний номер, на одиницю більший номера попередньої правки (рисунок 1.8). Початкова правка щойно створеного сховища отримує номер 0 і не містить нічого, крім порожнього кореневого каталогу.

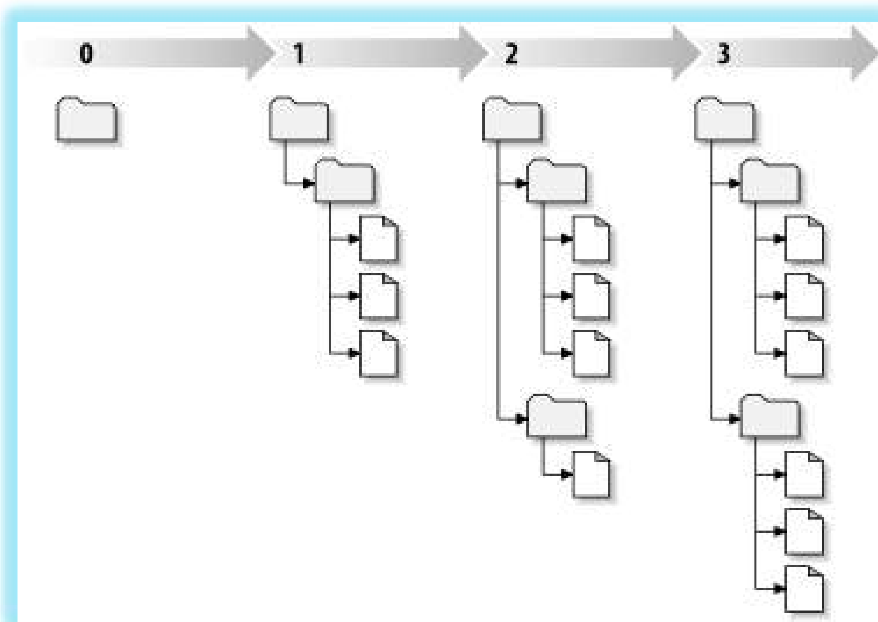


Рисунок 1.8 – Приклад створення правок

Відстеження сховища робочими копіями

У службовому каталозі .svn/ для кожного файла робочого каталогу Subversion записує інформацію про дві найважливіші властивості:

– на якій правці оснований Ваш робочий файл (це називається «робоча правка файла»);

– про часову мітку, що визначає, коли робоча копія останній раз оновлювалася зі сховища.

Використовуючи цю інформацію при з'єднанні зі сховищем, Subversion може вказати, в якому з наступних чотирьох станів знаходиться робочий файл:

– не змінився і не застарів;

– змінювався локально і не застарів;

– не змінювався і застарів;

– змінювався локально і застарів.

Стан файла «Не змінювався і не застарів»

Файл не змінився в робочому каталозі, а в сховищі також не фіксувалися зміни цього файла з часу створення його робочої правки. Команди svn commit і svn update ніяких операцій робити не будуть.

Стан файла «Змінювався локально і не застарів»

Файл був змінений в робочій копії, але в сховищі не фіксувалися зміни цього файла останнього оновлення робочої копії. Є локальні зміни, які не були зафіксовані у сховищі, тому svn commit виконає фіксацію Ваших змін, а svn update не зробить нічого.

Стан файла «Не змінювався і застарів»

У робочому каталозі файл не змінювався, але був змінений у сховищі. Необхідно оновити програмне забезпечення файла для того, щоб він відповідав поточній опублікованій правці. Команда svn commit не зробить нічого, а svn update оновить Вашу робочу копію файла відповідно до останніх змін.

Стан файла «Змінювався локально і застарів»

Файл був змінений як у робочому каталозі, так і у сховищі. svn commit зазнає невдачі, видавши помилку «out-of-date». Файл необхідно спочатку відновити; svn update спробує об'єднати локальні зміни з опублікованими. Якщо Subversion не зможе виконати об'єднання самостійно, вона запропонує користувачеві вирішити конфлікт вручну.

Архітектура системи контролю версій. Збереження даних

До системи контролю версій зазвичай пред'являються три ключові функціональні вимоги, а саме:

– зберігання даних;

– відстеження змін вмісту (зберігання історії змін, що включає метадані злиття);

– поширення даних та історії змін між розробниками.

Примітка. Третя вимога з цього списку не є функціональною вимогою для всіх систем контролю версій.

Таблиця merge_contents може стати проблемою для будь-якого адміністратора баз даних, який обслуговує реплікацію злиттям (merge) досить багато часу. Проблема полягає в тому, що merge_contents буде постійно зростати, якщо цим не управляти. Merge реплікація дуже

інтенсивно використовує merge_contents, що і спричиняє проблеми, оскільки її розмір неухильно зростає. Це зазвичай проявляється у вигляді взаємоблокувань та істотного уповільнення обміну змін до реплікації. Відбувається це тому, що час пошуку шуканих рядків у таблиці метаданих збільшується разом зі збільшенням її розміру.

Реплікація – механізм синхронізації вмісту декількох копій об'єкта [9]. Приклад можливого використання механізму реплікації наведений на рисунку 1.9 [10, 11].

Метадані – структуровані дані, що являють собою характеристики описуваних сутностей для цілей їх ідентифікації, пошуку, оцінки, управління ними [12]. Це набір допустимих структурованих описів, які доступні в явному вигляді й призначення яких може допомогти знайти об'єкт. Термін використовується в контексті пошуку об'єктів, сутностей, ресурсів.

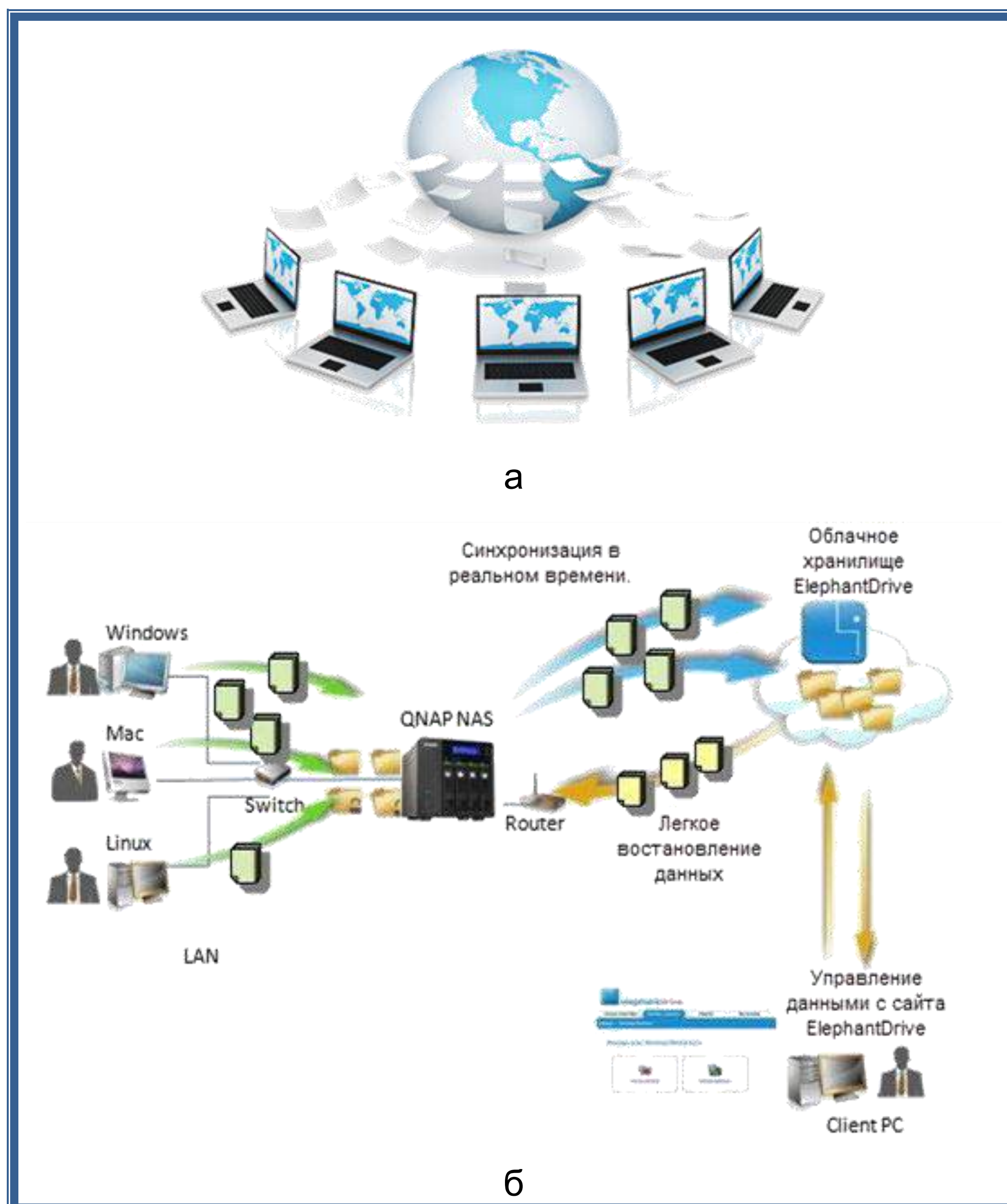


Рисунок 1.9 – Приклад можливого використання механізму реплікації

Зберігання даних у формі ациклічного орієнтованого графа

Найчастіше використовуваними архітектурними рішеннями для зберігання даних у світі систем контролю версій є набори змін на основі відмінностей файлів або подання даних у формі ациклічного орієнтованого графа (DAG) (рисунок 1.10) [13].

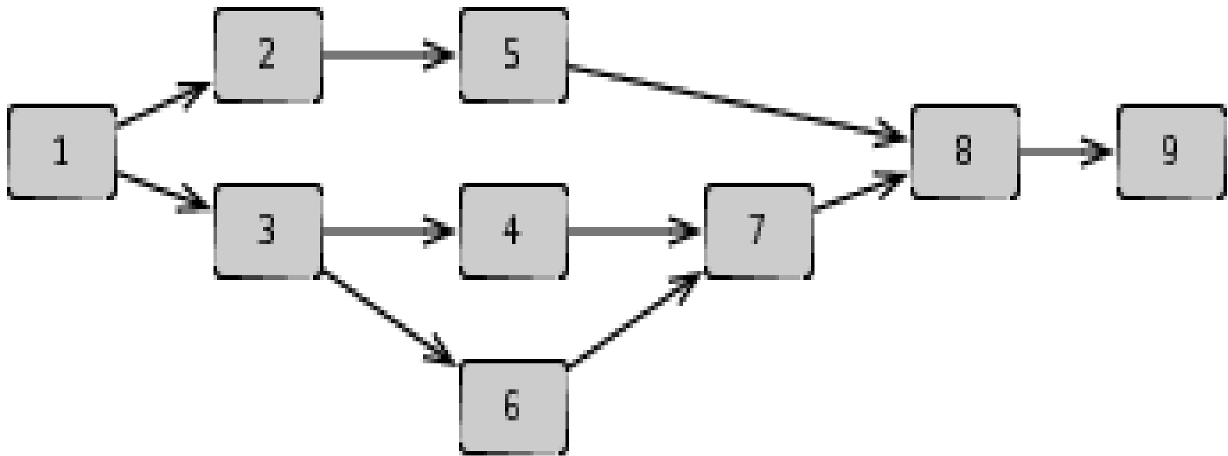


Рисунок 1.10 – Приклад зберігання даних у формі DAG

У цій схемі є три спеціальних типа версій коду: коренева версія, яка не має предків (репозитарій може мати кілька коренів), об'єднувальна ревізія (у ній кілька предків) і головна версія, у якій немає нащадків. Кожне сховище починається з порожньої кореневої версії коду і далі триває від неї по декількох лініях змін, закінчуючись однією або декількома головними версіями. Якщо два користувача незалежно один від одного закоммітили свої зміни і один з них хоче отримати зміни коду від іншого, то йому доведеться явно об'єднати зміни, внесені іншим користувачем, у нову версію, яку він потім коммітить як об'єднувальну версію.

Зберігання даних на основі відмінностей файлів

Набори змін на основі відмінностей файлів інкапсулюють відмінності між двома версіями даних довільного типу, доповнюючи їх деякою кількістю метаданих.

Під поданням даних у формі ациклічного орієнтованого графа мають на увазі використання об'єктів, які формують ієрархію, що «віддзеркалює» вміст дерева файлової системи у вигляді копії даних, що додаються (при цьому об'єкти, які не змінюються, в рамках дерева використовуються повторно тоді, коли це можливо).

Архітектура системи контролю версій. Поширення даних

Системи контролю версій здійснюють поширення даних робочої копії між учасниками процесу розроблення проекту одним з трьох способів:

1. Виключно локально: метод можна застосовувати для систем контролю версій, до яких не пред'являється третьої функціональної вимоги з поданого вище списку.

2. З використанням центрального сервера: в цьому випадку всі змінені дані мають бути передані через один певний репозитарій для занесення даних про їхні зміни в історію (рисунок 1.11) [14].

3. З використанням розподіленої моделі: в цьому випадку репозитарії завжди будуть відкриті для учасників процесу розроблення, які зможуть "помістити" дані в них, але операції зміни даних можуть також здійснюватися локально, і в цьому випадку дані будуть поміщені в репозитарій пізніше, що дозволяє виконувати роботу в умовах відсутності з'єднання з мережею (рисунок 1.12) [15].

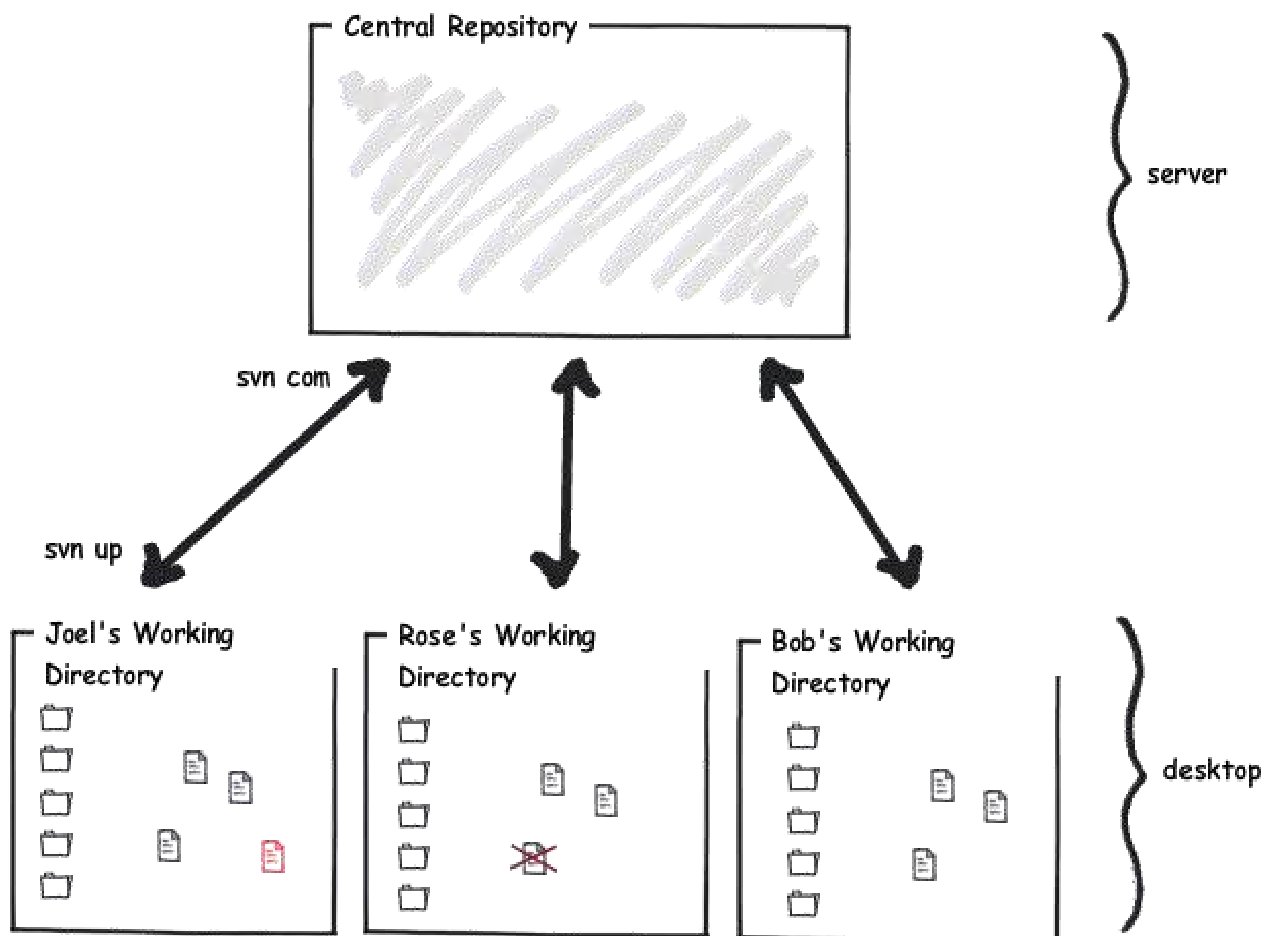


Рисунок 1.11 – Поширення даних з використанням центрального сервера

Розподілені системи управління версіями також відомі як англ. Distributed Version Control System, DVCS. Такі системи використовують розподілену модель замість традиційної клієнт-серверної. Вони, в загальному випадку, не потребують централізованого сховища: вся історія зміни документів зберігається на кожному комп'ютері, в локальному сховищі, і за необхідності окремі фрагменти історії локального сховища синхронізуються з аналогічним сховищем на іншому

комп'ютері. У деяких таких системах локальне сховище розташовується безпосередньо в каталогах робочої копії.

Коли користувач такої системи виконує звичайні дії, такі, як витяг певної версії документа, створення нової версії тощо, він працює зі своєю локальною копією сховища. У міру внесення змін, сховища, що належать до різних розробників, починають різнитися, і виникає необхідність у їх синхронізації. Така синхронізація може здійснюватися за допомогою обміну патчами[2] або так званими наборами змін (англ. Change sets) між користувачами.

Описана модель логічно близька до створення окремої гілки для кожного розробника в класичній системі управління версіями (у деяких розподілених системах перед початком роботи з локальним сховищем потрібно створити нову гілку). Відмінність полягає в тому, що до моменту синхронізації інші розробники цієї гілки не бачать. Поки розробник змінює лише свою гілку, його робота не впливає на інших учасників проекту і навпаки. По завершенні відокремленої частини роботи, внесені до гілки зміни зливають з основною (загальною) гілкою. Як при злитті гілок, так і при синхронізації різних сховищ можливі конфлікти версій. На цей випадок у всіх системах передбачено ті чи інші методи виявлення і вирішення конфліктів злиття.

З точки зору користувача, розподілена система відрізняється необхідністю створювати локальний репозитарій і наявністю в командній мові двох додаткових команд: команди отримання сховища від віддаленого комп'ютера (pull) і передачі свого сховища на віддалений комп'ютер (push). Перша команда виконує злиття змін віддаленого і локального репозитаріїв з внесенням результату до локального репозитарію; друга – навпаки, виконує злиття змін двох репозитаріїв з розміщенням результату у віддаленому репозитарії. Як правило, команди злиття у розподілених системах дозволяють вибрати, які набори змін будуть передаватися в інший репозитарій або вилучатись з нього, виправляти конфлікти злиття безпосередньо у ході операції або після її невдалого завершення повторювати або відновлювати незакінчене злиття. Зазвичай передача своїх змін до чужого репозитарію (push) завершується вдало тільки за умови відсутності конфліктів. Якщо конфлікти виникають, користувач повинен спочатку злити версії у своєму репозитарії (виконати pull), і лише потім передавати їх іншим.

---

[2] Патч (англ. – латочка, заплатка) – інформація, призначена для автоматизованого внесення певних змін у комп'ютерні файли. Це виправлення може потребувати застосовуватися до вже встановленої програми або до її вихідного коду. Сюди входить виправлення помилок, зміна зовнішнього вигляду, поліпшення ергономічності або продуктивності програм, а також будь-які інші зміни, які розробник побажав зробити. Розмір патчів може варіюватися від декількох кілобайт до сотень мегабайт. При цьому слова «патч», «латочка» зазвичай використовуються для позначення невеликих виправлень, великі ж патчі, що серйозно міняють або оновлюють програму, часто називаються «service pack» або «software updates» [15].

Зазвичай рекомендується організовувати роботу з системою так, щоб користувачі завжди або переважно виконували злиття у себе в репозитарії. Тобто, на відміну від централізованих систем, де користувачі передають свої зміни на центральний сервер, коли вважають за потрібне, в розподілених системах більш природним є порядок, коли злиття версій ініціює той, кому потрібно отримати його результат (наприклад, розробник, керуючий складальним сервером).

Основні переваги розподілених систем - їх гнучкість і значно більша (у порівнянні з централізованими системами) автономія окремого робочого місця. Кожен комп'ютер розробника є, фактично, самостійним і повнофункціональним сервером. З таких комп'ютерів можна побудувати довільну за структурою і рівнем складності систему, задавши (як технічними, так і адміністративними заходами) бажаний порядок синхронізації. При цьому кожен розробник може вести роботу незалежно, так, як йому зручно, змінюючи і зберігаючи проміжні версії документів, користуючись усіма можливостями системи (в тому числі доступом до історії змін) навіть за відсутності підключення до мережі з сервером. Зв'язок з сервером або іншими розробниками потрібний виключно для проведення синхронізації, при цьому обмін наборами змін може здійснюватися за різними схемами.

До недоліків розподілених систем можна віднести збільшення необхідного обсягу дискової пам'яті: на кожному комп'ютері доводиться зберігати повну історію версій, тоді як у централізованій системі на комп'ютері розробника зазвичай зберігається лише робоча копія, тобто зріз сховища на якийсь момент часу і внесені зміни. Менш очевидним, але неприємним недоліком є те, що в розподіленій системі практично неможливо реалізувати деякі види функціональності, що надаються централізованими системами. Це:

• *Блокування файла або групи файлів* (для зберігання ознаки блокування потрібен загальнодоступний і такий центральний сервер, що постійно знаходиться в онлайні). Це змушує застосовувати спеціальні адміністративні заходи, якщо доводиться працювати з бінарними файлами, непридатними для автоматичного злиття.

• *Стеження за певним файлом або групою файлів* (зміни файлів відбуваються на різних серверах, злиття і виділення гілок відбуваються локально, про зміни стає відомо тільки при синхронізації, причому не всім розробникам, а тільки тим, хто в цій синхронізації бере участь).

• *Єдина наскрізна нумерація версій системи і / або файлів*, в якій номер версії монотонно зростає (така нумерація також вимагає наявності головного сервера, що задає номери версій для всіх інших). У розподілених системах доводиться обходитися локальними позначеннями версій і застосовувати теги, призначення яких

визначається угодою між розробниками або корпоративними стандартами фірми.

• *Локальна робота користувача* з окремою, невеликою за обсягом вибіркою із значного за розміром і внутрішньої складності сховища на віддаленому сервері.

Можна виділити такі типові ситуації, в яких використання розподіленої системи дає помітні переваги:

• *Періодична синхронізація декількох комп'ютерів під управлінням одного розробника* (робочого комп'ютера, домашнього комп'ютера, ноутбука і так далі). Використання розподіленої системи позбавляє від необхідності виділяти один з комп'ютерів як сервер. Синхронізація за необхідності виконується при «пересадці» розробника з одного пристрою на інший.

• *Спільна робота над проектом невеликої територіально розподіленої групи розробників без виділення загальних ресурсів.* Як і в попередньому випадку, реалізується схема роботи без головного сервера, а актуальність репозитаріїв підтримується періодичними синхронізаціями за схемою «кожен з кожним».

• *Великий розподілений проект, учасники якого можуть довгий час працювати кожен над своєю частиною, при цьому не мають постійного підключення до мережі.* Такий проект може використовувати централізований сервер, з яким синхронізуються копії всіх його учасників. Можливі й більш складні варіанти - наприклад, зі створенням груп для роботи за окремими напрямами всередині більшого проекту. При цьому можуть бути виділені окремі «групові» сервери для синхронізації роботи груп, тоді процес остаточного злиття зміни стає деревовидним: спочатку окремі розробники синхронізують зміни на групових серверах, потім оновлені репозитарії груп синхронізуються з головним сервером. Можлива робота і без «групових» серверів, тоді розробники однієї групи синхронізують зміни між собою, після чого будь-який з них (наприклад, керівник групи) передає зміни на центральний сервер.

У традиційній «офісній» розробці проектів, коли група розробників відносно невелика і цілком знаходиться на одній території, в межах єдиної локальної комп'ютерної мережі, з постійно доступними серверами, централізована система може виявитися кращим вибором завдяки своїй більш жорсткій структурі та наявності функціональності, відсутньої в розподілених системах (наприклад, вже згаданого блокування). Можливість фіксувати зміни без їх злиття у центральну гілку в таких умовах легко реалізується шляхом виділення незавершених робіт в окремі гілки розроблення.

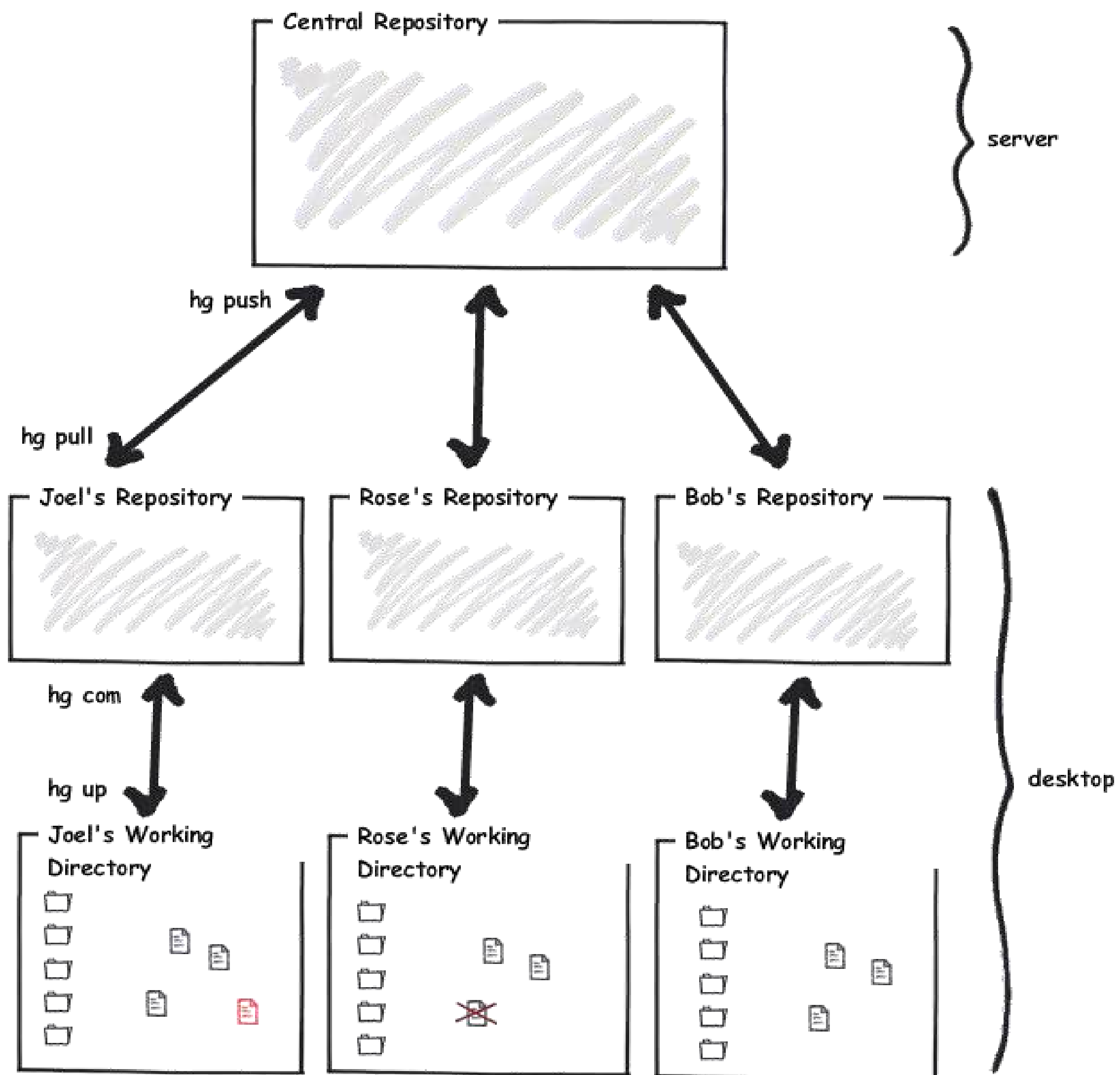


Рисунок 1.12 – Поширення даних з використанням розподіленої моделі

Планування проекту та відстеження змін

1. Перш за все, у всіх централізованих системах контролю версій фіксування змін (операція commit) і їх публікація за своєю суттю є одним і тим же, оскільки історія сховища зберігається централізовано в одному місці. Це означає, що відправка змін без доступу до мережі неможлива.

2. По-друге, доступ до репозитаріїв у централізованих системах завжди вимагає передачі даних по мережі на сервер, що робить такі системи відносно повільними порівняно з локальним доступом у розподілених системах.

3. По-третє, описані раніше системи мали певні недоліки при відстеженні злиттів (хоча в деяких з них з тих пір їх підтримка покращилася). Для великих груп розробників, що працюють одночасно, важливо, щоб система контролю версій записувала, які зміни були включені в нові версії коду, щоб нічого не загубилося, і наступні злиття могли використовувати цю інформацію.

4. По-четверте, централізація, яка потрібна традиційним системам контролю версій, здається штучною і висуває вимогу наявності єдиного простору для інтеграції.

5. Прихильники розподілених систем контролю версій стверджують, що розподілені системи дозволяють більш органічно організувати роботу: розробники можуть передавати та інтегрувати зміни так, як того вимагає проект у кожен момент часу.

Створення команди з використанням Bitbucket

Bitbucket – це хостинг для Mercurial і Git репозитаріїв. Найближчий аналог і прямий конкурент – github. За популярністю Bitbucket відстає, проте у нього є пара помітних фіч[3] у порівнянні з github – це підтримка Mercurial і можливість створювати скільки завгодно приватних репозитаріїв на безкоштовному акаунті, однак дати доступ можна максимум п'ятьом користувачам (у github взагалі немає приватних репозитаріїв для безкоштовних акаунтів).

Зараз команда Bitbucket оновила свій сервіс. Повністю оновився дизайн і додалося багато нових функцій. Короткий огляд нововведень зроблений на основі запису в офіційному блозі команди розробників.

На рисунку 1.13 наведений скріншот створення команди розробників з використанням Bitbucket.

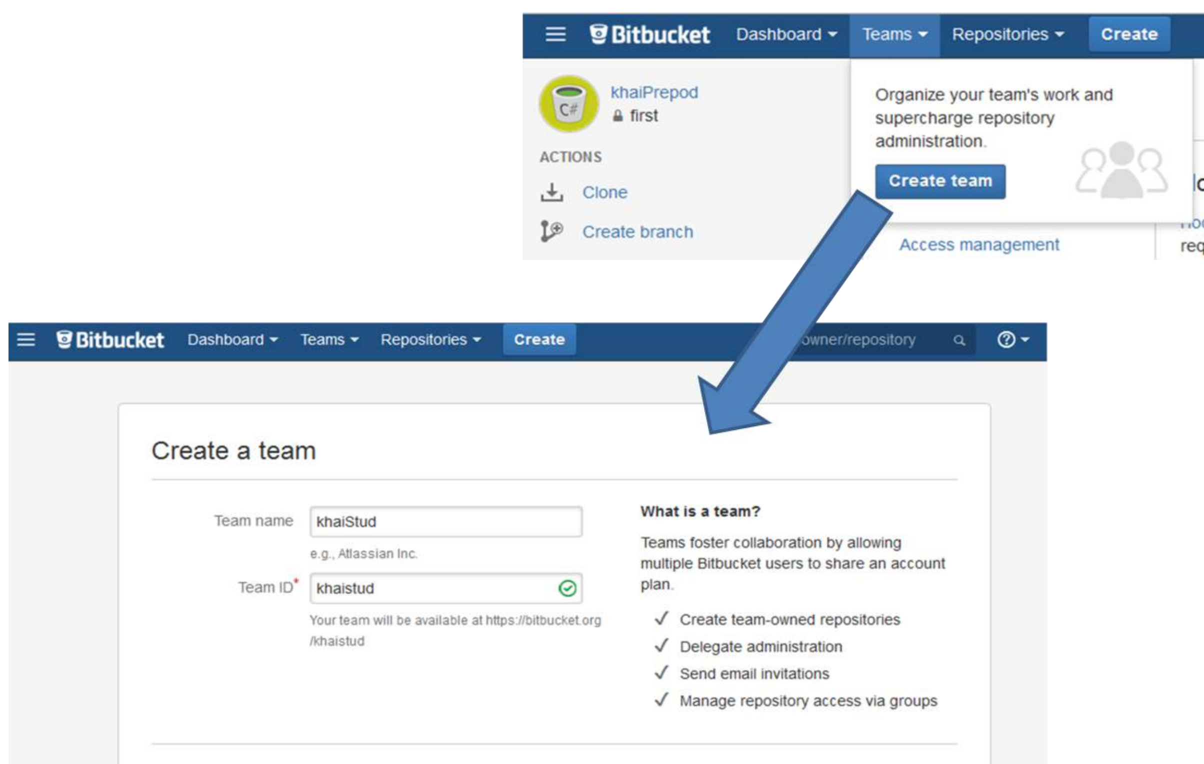


Рисунок 1.13 – Приклад створення команди розробників

[3] Фіча (англ. Feature) – в жаргоні програмістів, геймерів й інших користувачів комп'ютерів це специфічна особливість, можливість або здатність програми або проекту до виконання будь-якої функції. Це слово зазвичай використовується тільки в усному мовленні. У мережі Інтернет поширено вираз «це не баг, це фіча», що позначає, що якась особливість програми розцінюється зазвичай як програмна помилка, і в той же час є її корисною можливістю.

Додавання нових учасників проекту до команди

На рисунку 1.14 зображений скріншот додавання нових учасників проекту до команди з використанням Bitbucket.

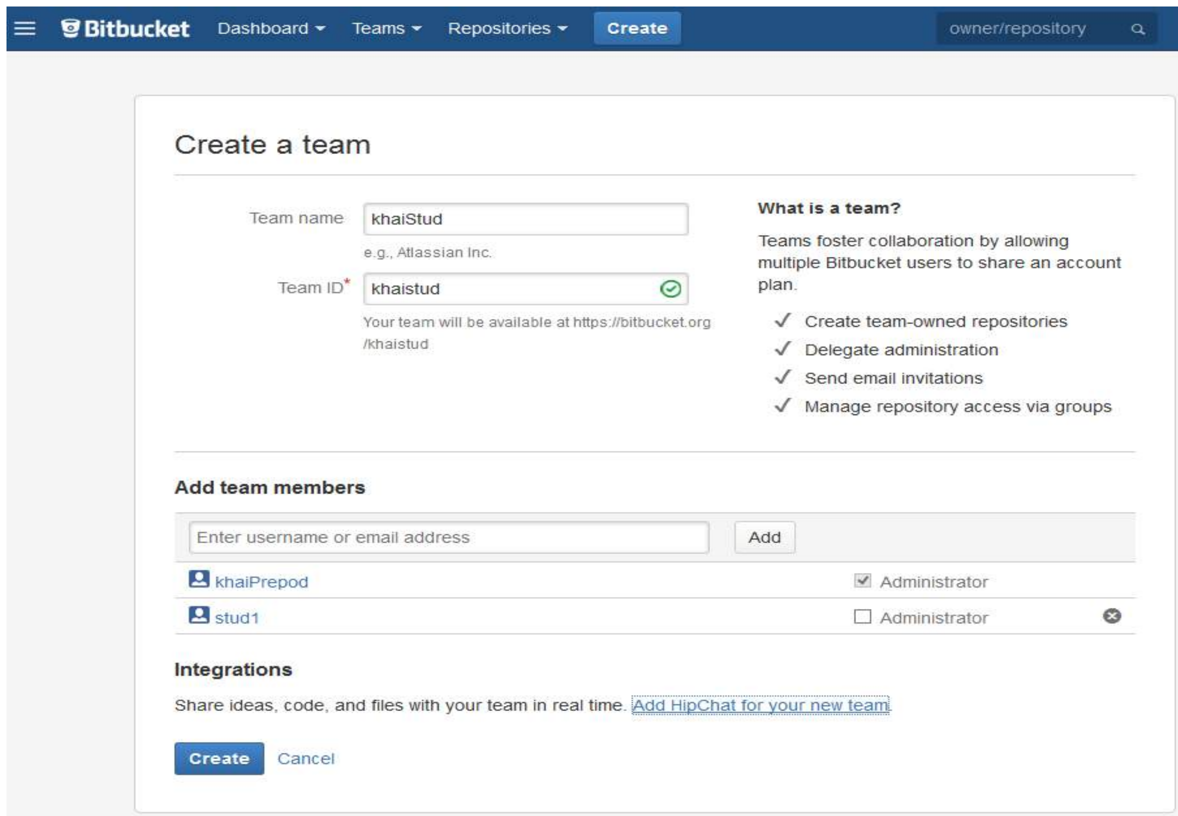


Рисунок 1.14 – Приклад додавання нових учасників проекту до команди

Створення репозитарію для команди (рисунок 1.15).

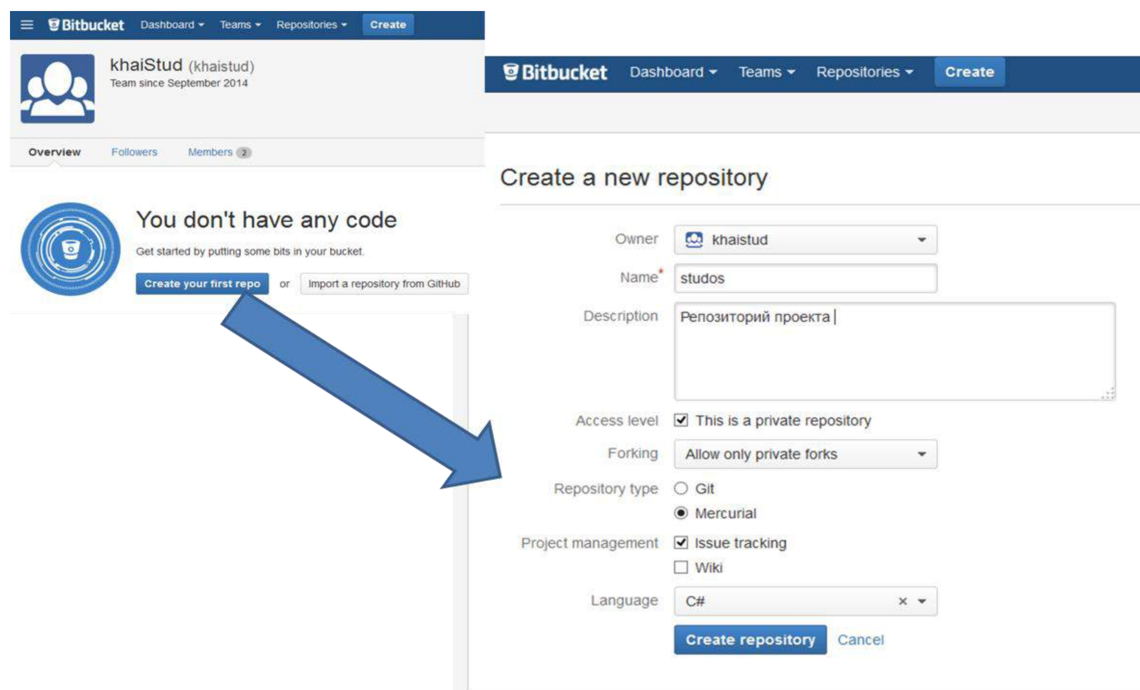


Рисунок 1.15 – Приклад створення репозитарію для команди

Клонування створеного репозитарію (створення робочої копії) (рисунок 1.16).

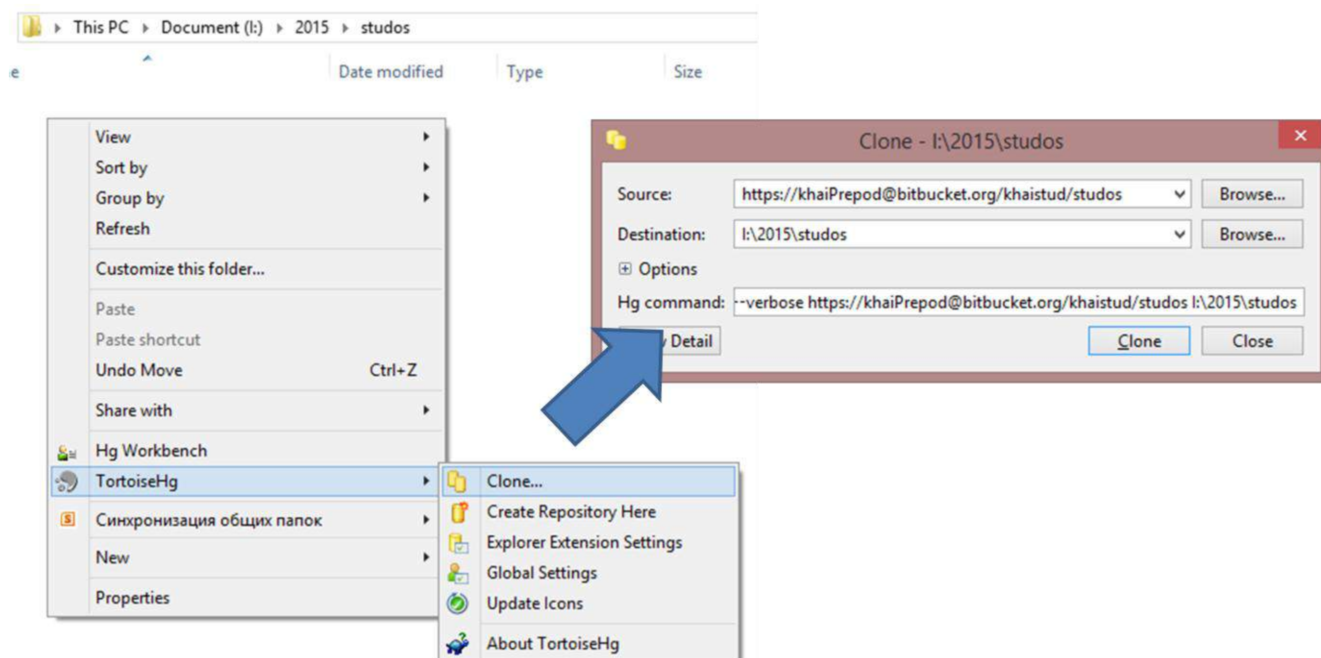


Рисунок 1.16 – Приклад клонування створеного репозитарію (створення робочої копії) у Tortoise HG

Додавання нових файлів до проекту (рисунки 1.17 – 1.18).

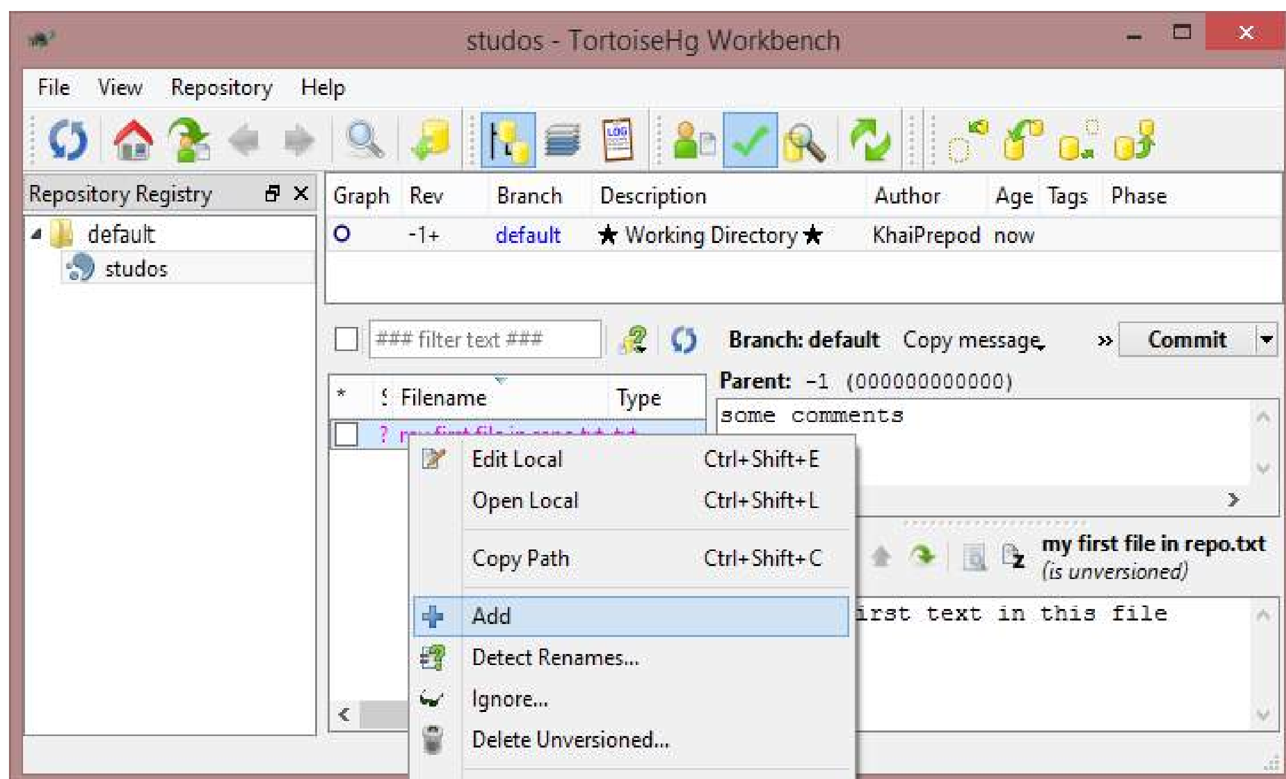


Рисунок 1.17 – Приклад додавання нових файлів до проекту у Tortoise HG WorkBench

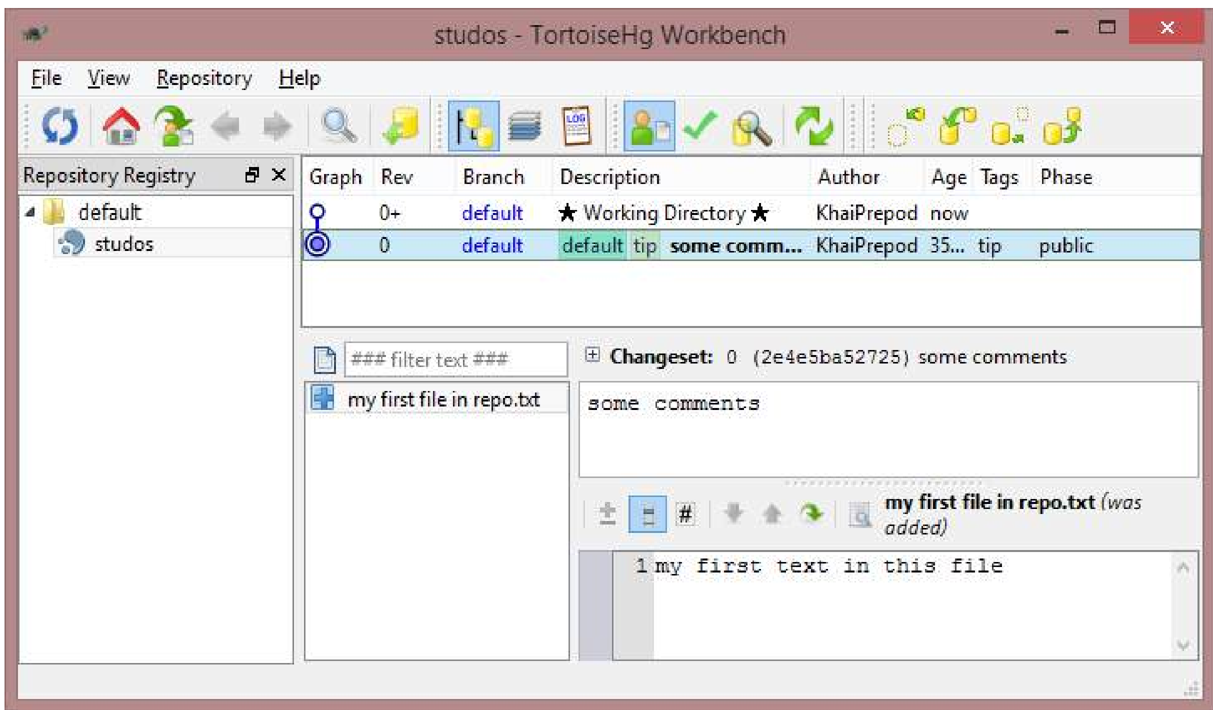


Рисунок 1.18 – Приклад додавання нових файлів до проекту у Tortoise HG WorkBench. Продовження

Налаштування прив'язки завдань до коментарів коммітів (рисунок 1.19).

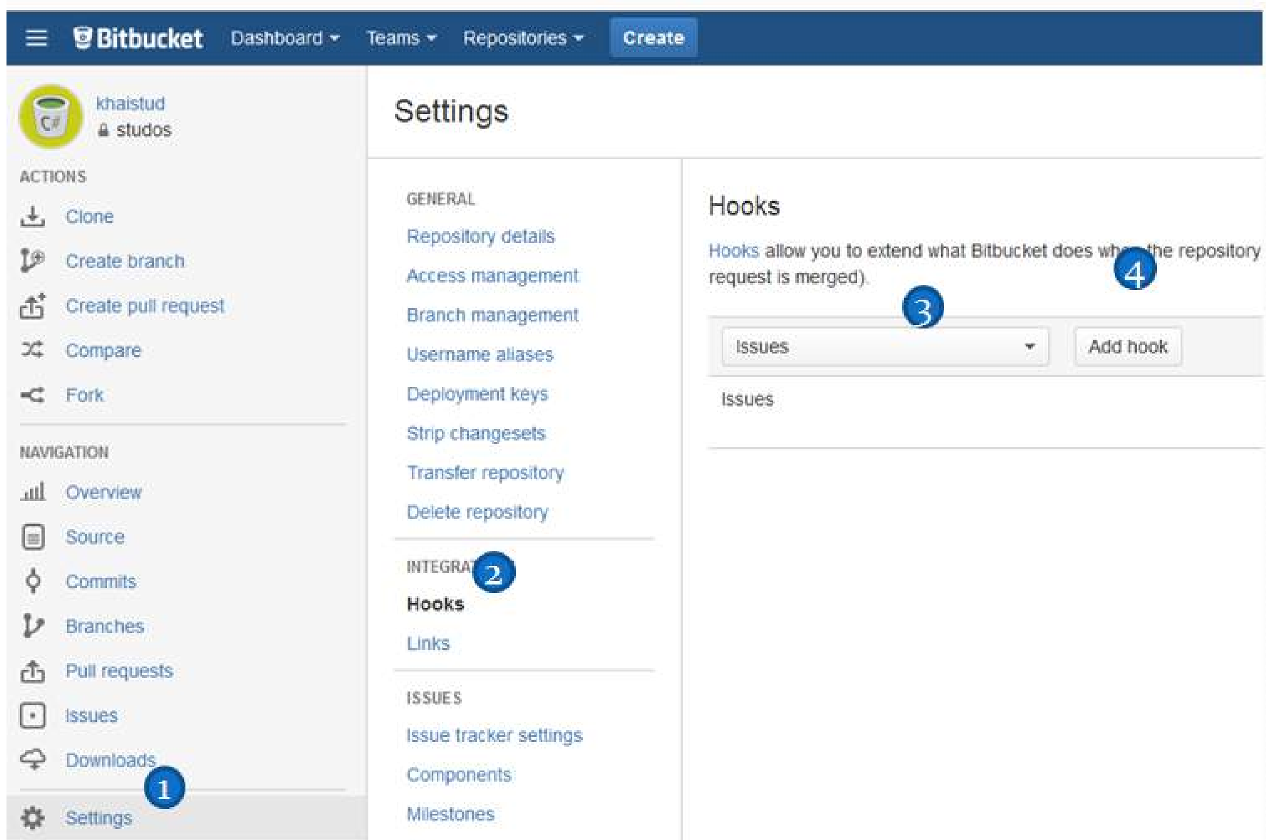


Рисунок 1.19 – Налаштування прив'язки завдань до коментарів коммітів з використанням Bitbucket

Список задач за проектом (рисунок 1.20).

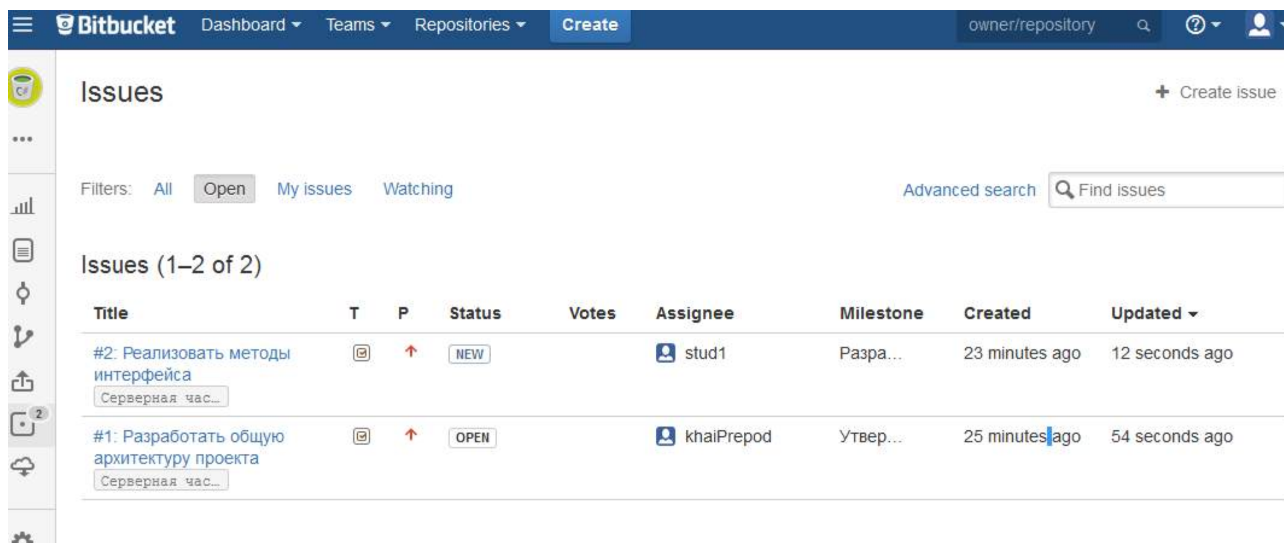


Рисунок 1.20 – Приклад створення списку задач (тікетів) на Bitbucket

Перегляд набору змін (рисунок 1.21).

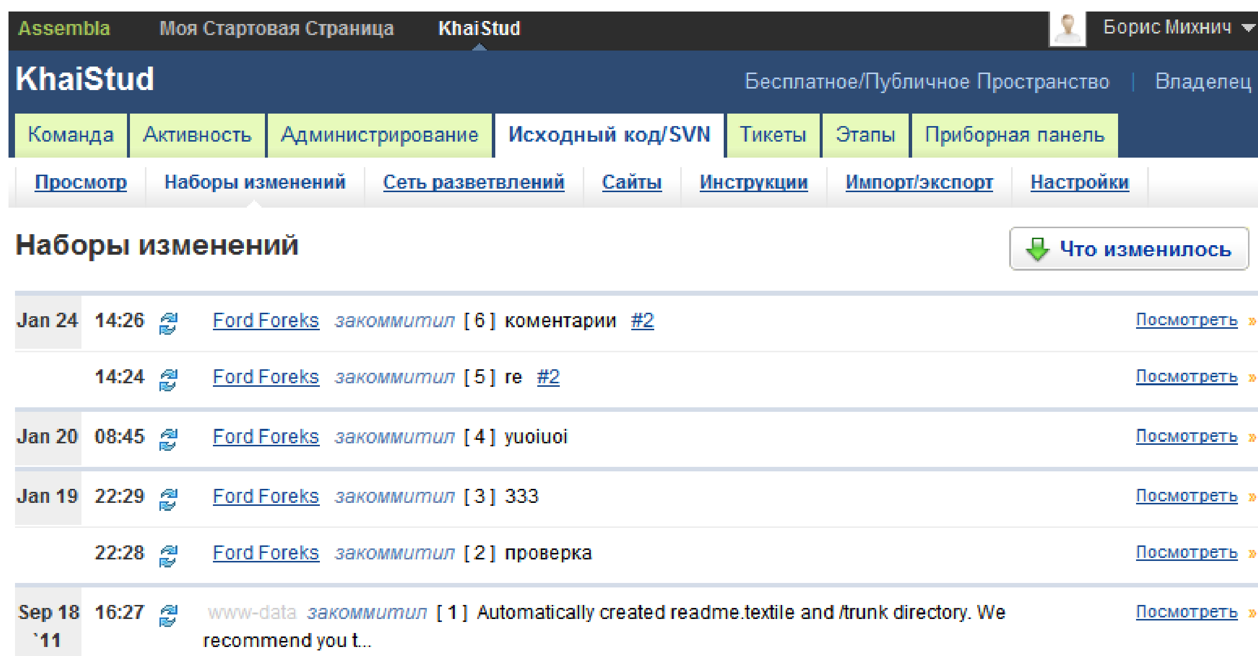


Рисунок 1.21 – Приклад перегляду набору змін на Assembla

Посилання на тікети поряд з коммітами (рисунок 1.22).

Посилання на тікети поряд з коммітами на Bitbucket і в Tortoise HG WorkBench (рисунок 1.23).

Віхи проекту (Milestones).

Віха, або Milestone (Майлстоун), – термін, що використовується в управлінні проектами, контрольна точка, значущий, ключовий момент, (наприклад, перехід на нову стадію, новий етап у ході виконання проекту).

Історично, Майлстоун – це кам'яні плити із зазначенням відстаней до населених пунктів, які розставлялися уздовж доріг і служили орієнтиром мандрівникам.

Майлстоун (milestone) – метафора, якою в software development позначають проміжний етап розроблення проекту.

Для ссылки на выполняемую задачу, которой сопоставлен Ticket (Карточка) используется добавление номера тикета к комментариям коммита:

Комментарий о проделанной работе #2

Ваше сообщение

Номер задачи

Рисунок 1.22 – Посилання на тікети поряд з коммітами

Посилання на тікети поряд з коммітами наведено на рисунку 1.23.

Просмотр набора изменений:

Номер задачи

Рисунок 1.23 – Посилання на тікети поряд з коммітами
на Bitbucket і в Tortoise HG WorkBench

Процес розроблення розбивається Майлстоун, згідно зі спеціальним планом, з метою мінімізувати ризики і встигнути в задані терміни. Кожен Майлстоун має на увазі список завдань, які повинні бути вирішені та втілені у проект до дедлайну. Якщо вирішити всі завдання не встигають, відбувається зрив Майлстоун. У цьому випадку або виділяють додатковий час на реалізацію і змінюють терміни, або відмовляються від проблемного завдання зовсім.

Майлстоун фіксує всі прийняті рішення, щоб у розробників не виникало спокуси переробляти все до нескінченності.

Додавання нових віх до проекту (рисунок 1.24).

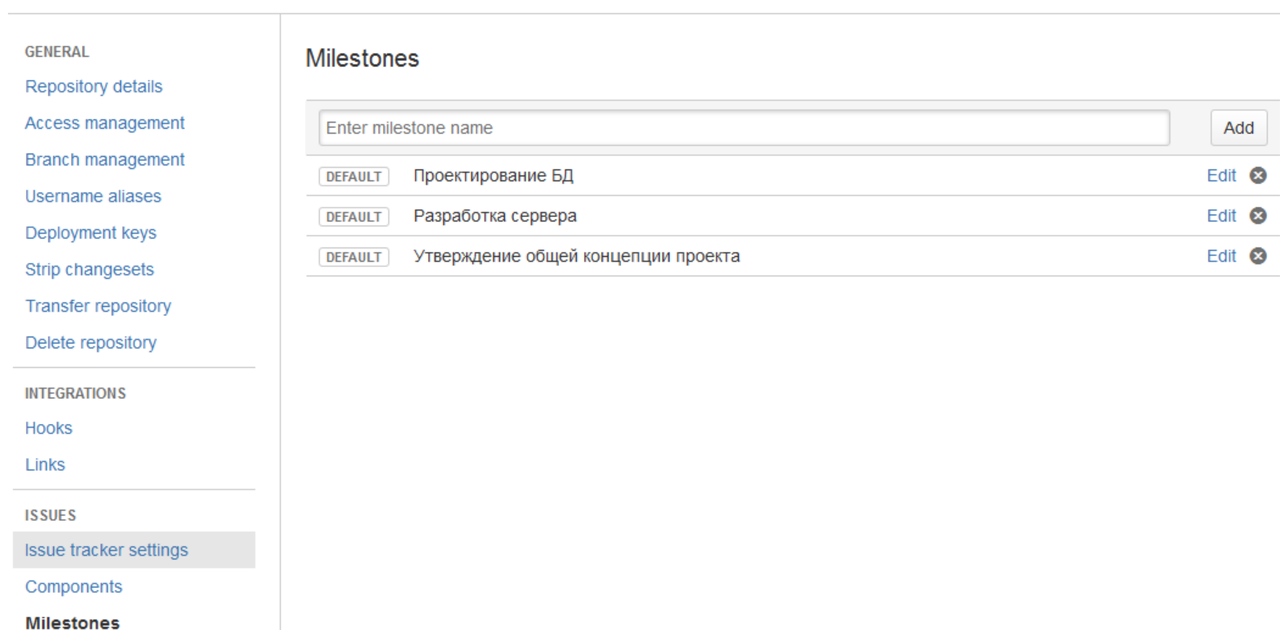


Рисунок 1.24 – Додавання нових віх до проекту

Злиття версій

Злиття. Основні поняття

Гілка (branch) – напрям розроблення, незалежний від інших. Гілка є копією частини (як правило, одного каталогу) сховища, в яку можна вносити свої зміни, які не впливають на інші гілки. Документи в різних гілках мають однакову історію до точки розгалуження і різні – після неї.

Основна версія (head) – найсвіжіша версія для гілки / стовбура, що знаходиться в сховищі. Скільки гілок, стільки основних версій.

Виправлення (Revision) – це стан сховища в певні моменти часу. Процес створення правки називається «Фіксація». Кожна правка має номер, який присвоюється при фіксації.

Злиття (merge, integration) – об'єднання незалежних змін в єдину версію документа. Здійснюється, коли двоє людей змінили один і той же файл або при перенесенні змін з однієї гілки в іншу.

Конфлікт (conflict) – ситуація, коли кілька користувачів зробили зміни однієї й тієї ж ділянки документа. Конфлікт виявляється, коли один

користувач зафіксував свої зміни, а другий намагається зафіксувати, і система сама не може коректно злити конфліктуючі зміни. Оскільки програма може бути недостатньо розумною для того, щоб визначити, яка зміна є «коректною», другому користувачеві потрібно самому вирішити конфлікт (**resolve**).

Злиття. Об'єднання змін

Три види операцій, виконуваних у системі управління версіями, можуть приводити до необхідності об'єднання змін.

Оновлення робочої копії (зміни, зроблені в основній версії, зливаються з локальними).

Фіксація змін (локальні зміни зливаються зі змінами, вже зафіксованими в основній версії).

Злиття гілок (зміни, зроблені в одній гілці розроблення, зливаються зі змінами, зробленими в іншій).

У всіх випадках ситуація принципово однакова і має такі характерні риси:

1. Раніше була зроблена копія дерева файлів і каталогів сховища або його частини.

2. Згодом і в оригінальне дерево, і в копію були незалежно внесені деякі зміни.

3. Потрібно об'єднати зміни в оригіналі й копії таким чином, щоб не порушити логічну зв'язність проекту і не втратити дані.

Цілком очевидно, що при невиконанні умови (2) (тобто якщо зміни були внесені лише в оригінал або тільки в копію) об'єднання елементарно – достатньо скопіювати змінену частину туди, де змін не було. В іншому випадку злиття змін перетворюється в нетривіальну задачу, яка у багатьох випадках потребує втручання розробника.

Злиття. Основні принципи

Механізм автоматичного злиття змін працює, грунтуючись на таких принципах:

Зміни можуть полягати у модифікації вмісту файла, створенні нового файла або каталогу, видаленні або перейменуванні раніше існуючого файла або каталогу в проекті.

Якщо дві зміни відносяться до різних і не пов'язаних між собою файлів і / або каталогів, вони завжди можуть бути об'єднані автоматично. Їх об'єднання полягає в тому, що зміни, зроблені в кожній версії проекту, копіюються в об'єднувальну версію.

Створення, видалення та перейменування файлів у каталогах проекту можуть бути об'єднані автоматично, якщо тільки вони не конфліктують між собою. У цьому випадку зміни, зроблені в кожній версії проекту, копіюються в об'єднувальну версію.

Злиття. Конфлікти

- Видалення і зміна одного і того ж файла або каталогу.

– Видалення і перейменування одного і того ж файла або каталогу (у разі, якщо система підтримує операцію перейменування).

– Створення в різних версіях файла з одним і тим же ім'ям і різним вмістом.

Зміни в межах одного текстового файла, зроблені в різних версіях, можуть бути об'єднані, якщо вони знаходяться в різних місцях цього файла і не перетинаються. У цьому випадку в об'єднану версію вносяться всі зроблені зміни.

Зміни в межах одного файла, якщо він не є текстовим, завжди є такими, що конфліктують, і не можуть бути об'єднані автоматично.

У всіх випадках базовою версією для злиття є версія, в якій було зроблено поділ версій, що зливаються. Якщо це операція фіксації змін, то базовою буде версія останнього оновлення перед фіксацією, якщо оновлення – то версія попереднього поновлення, якщо злиття гілок – то версія, в якій була створена відповідна гілка. Відповідно, наборами змін, що зіставляються, будуть набори змін, зроблені від базової до поточної версій у всіх поєднуваних варіантах.

Абсолютна більшість сучасних систем управління версіями орієнтована, у першу чергу, на проекти розроблення програмного забезпечення, в яких основним видом вмісту файла є текст. Відповідно, механізми автоматичного злиття змін орієнтуються на оброблення текстових файлів, тобто файлів, що містять текст, який складається з рядків букв і цифр, пробілів і табуляцій, розділених символами переведення рядка.

При визначенні припустимості злиття змін у межах одного і того ж текстового файла працює типовий механізм порядкового порівняння текстів (прикладом його реалізації є системна утиліта GNU diff), який порівнює об'єднувальні версії з базовою та будує список змін, тобто доданих, віддалених і замінених наборів рядків . Мінімальною одиницею даних для цього алгоритму є рядок, навіть найменша відміна робить рядки різними. З урахуванням того, що символи-роздільники в більшості випадків не несуть змістовного навантаження, механізм злиття може ігнорувати ці символи при порівнянні рядків.

Ті знайдені набори змінених рядків, які не перетинаються між собою, вважаються сумісними, і їх злиття відбувається автоматично. Якщо у файлах, які зливаються, є зміни, що зачіпають один і той же рядок файла, це призводить до конфлікту. Такі файли можуть бути об'єднані тільки вручну. Будь-які файли, окрім текстових, за VCS, є бінарними і не допускають автоматичного злиття.

Конфлікти під час комміту

У таблиці 1.1 подано дві версії програмного коду. У першій версії обробник події buttonHello_Click (objectsender, EventArgse) виводить

повідомлення «Привіт», у другій версії обробник події buttonBye_Click (objectsender, EventArgse) виводить повідомлення «Поки».

Таблиця 1.1 – Версії коду програми

Перша версія коду	Друга версія коду

namespaceWindowsFormsApplication
{
public partial class Form1 : Form
{
public Form1()
{
InitializeComponent();
}
private void buttonHello_Click(object sender, EventArgs e)
{
MessageBox.Show("Hello!!!");
}
}
}
	namespace WindowsFormsApplication
	{
	public partial class Form1 : Form
	{
	public Form1()
	{
	InitializeComponent();
	}
	private void buttonBye_Click(object sender, EventArgs e)
	{
	MessageBox.Show("Goodbye!");
	}
	}
	}

На рисунках 1.25 – 1.26 наведено результати виконання обох програм.

Рисунок 1.25 – Виведення на екран повідомлення «Привіт»

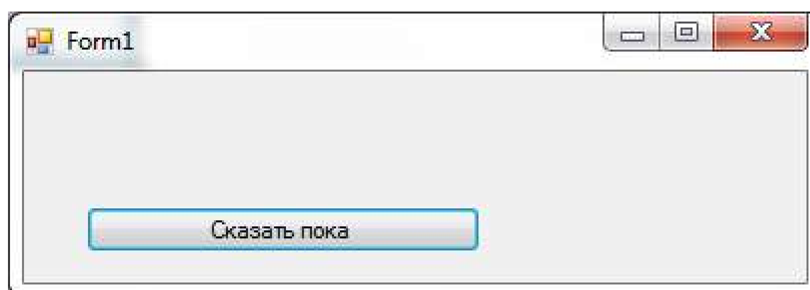


Рисунок 1.26 – Виведення на екран повідомлення «Пока»

Комміт зміненої першої версії зображено на рисунку 1.27.

Рисунок 1.27 – Комміт зміненої першої версії

Комміт пройшов успішно (рисунок 1.28).

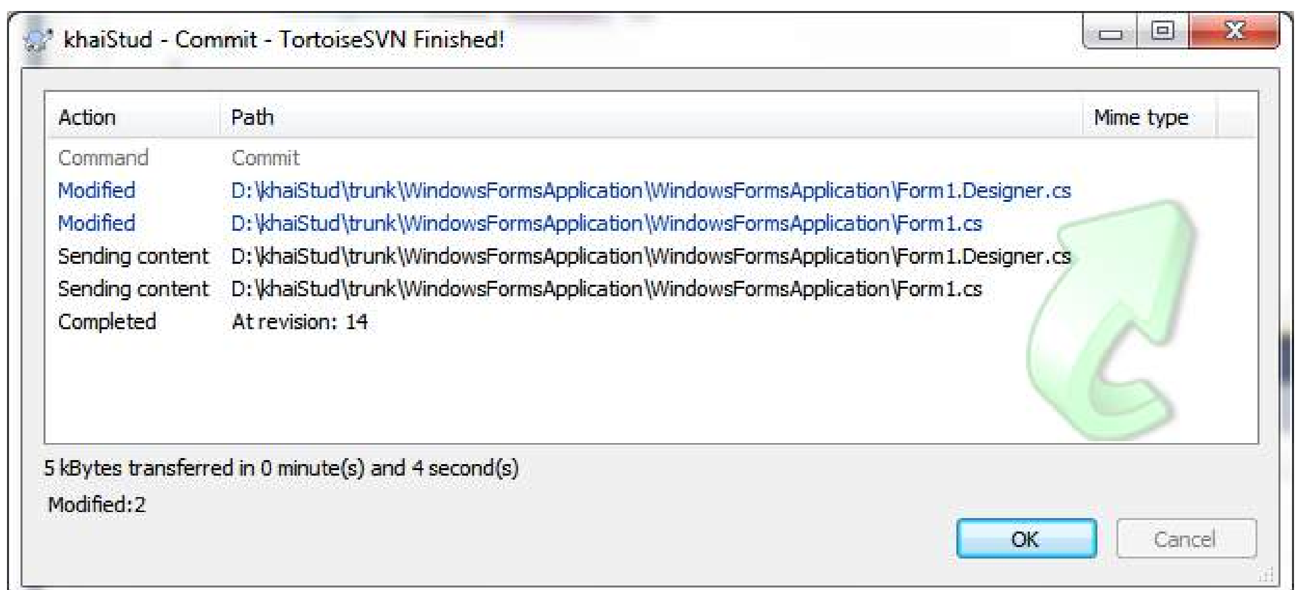


Рисунок 1.28 – Скріншот успішно пройденого комміту

Комміт зміненої другої версії (рисунок 1.29).

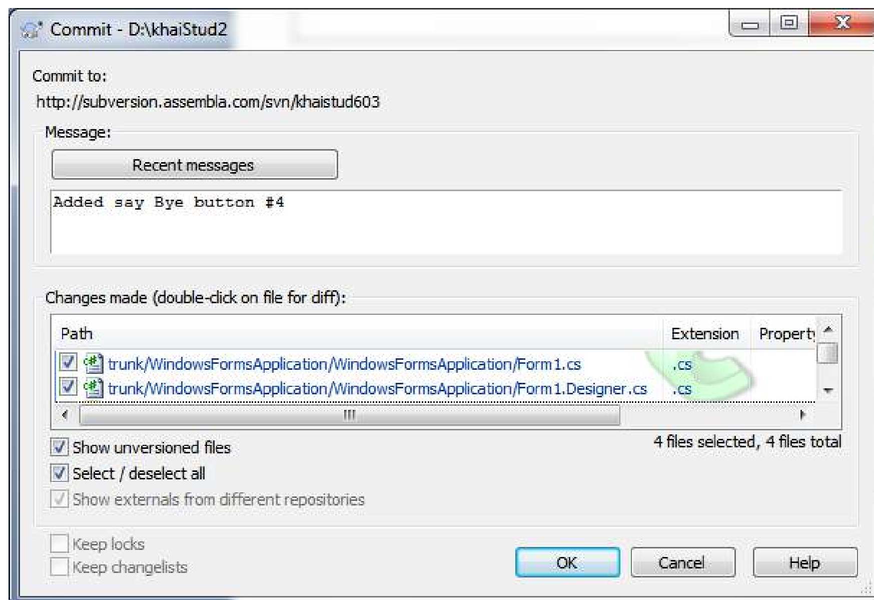


Рисунок 1.29 – Комміт зміненої другої версії

Змінена перша версія застаріла, тому що в сховищі вже знаходиться інша версія (рисунок 1.30).

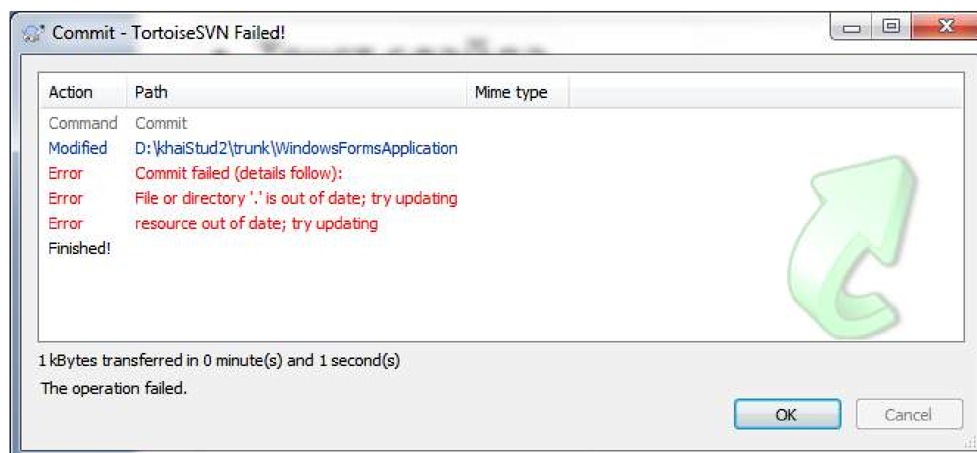


Рисунок 1.30 – Скріншот пройденого з помилками комміту

Update (оновлення) (рисунок 1.31).

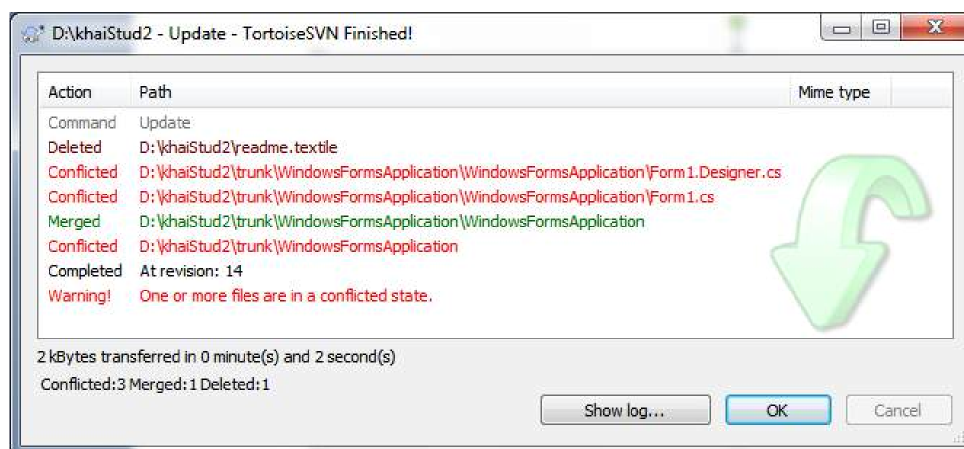


Рисунок 1.31 – Результат виконання операції оновлення

Злиття (Merge) (рисунок 1.32).

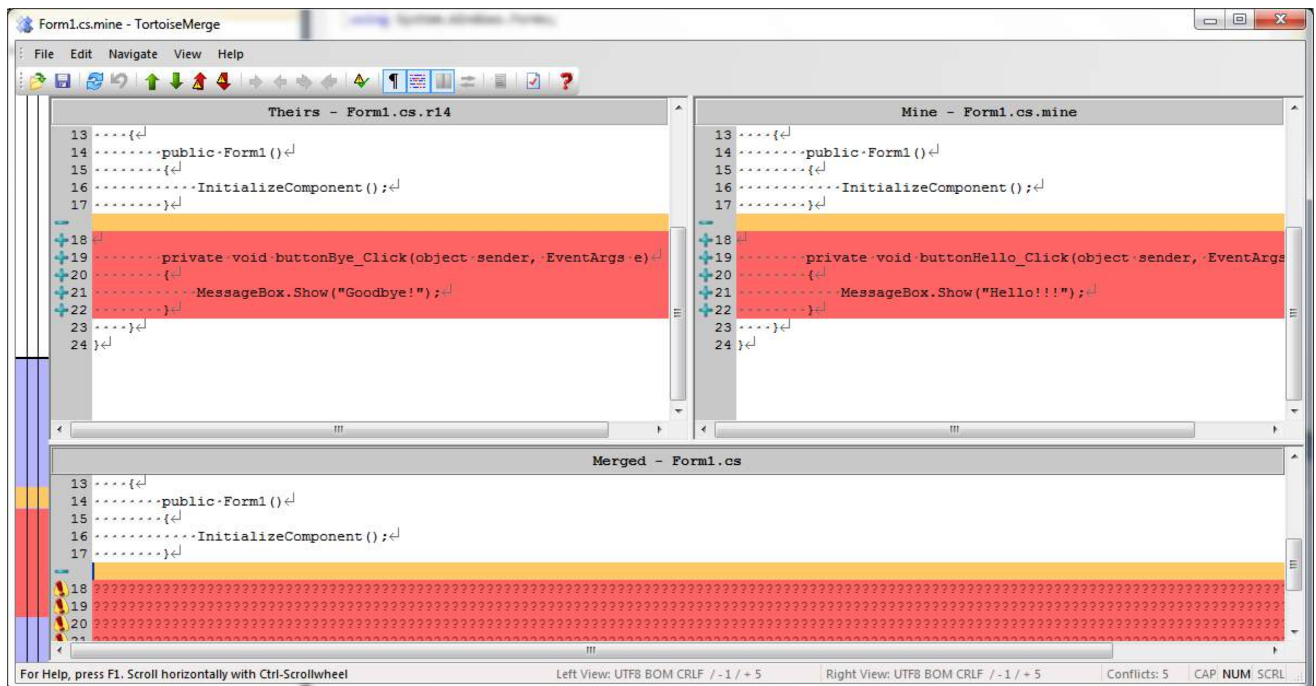


Рисунок 1.32 – Приклад визначення відмінностей у версіях коду

Результат злиття і підтвердження (рисунок 1.33).

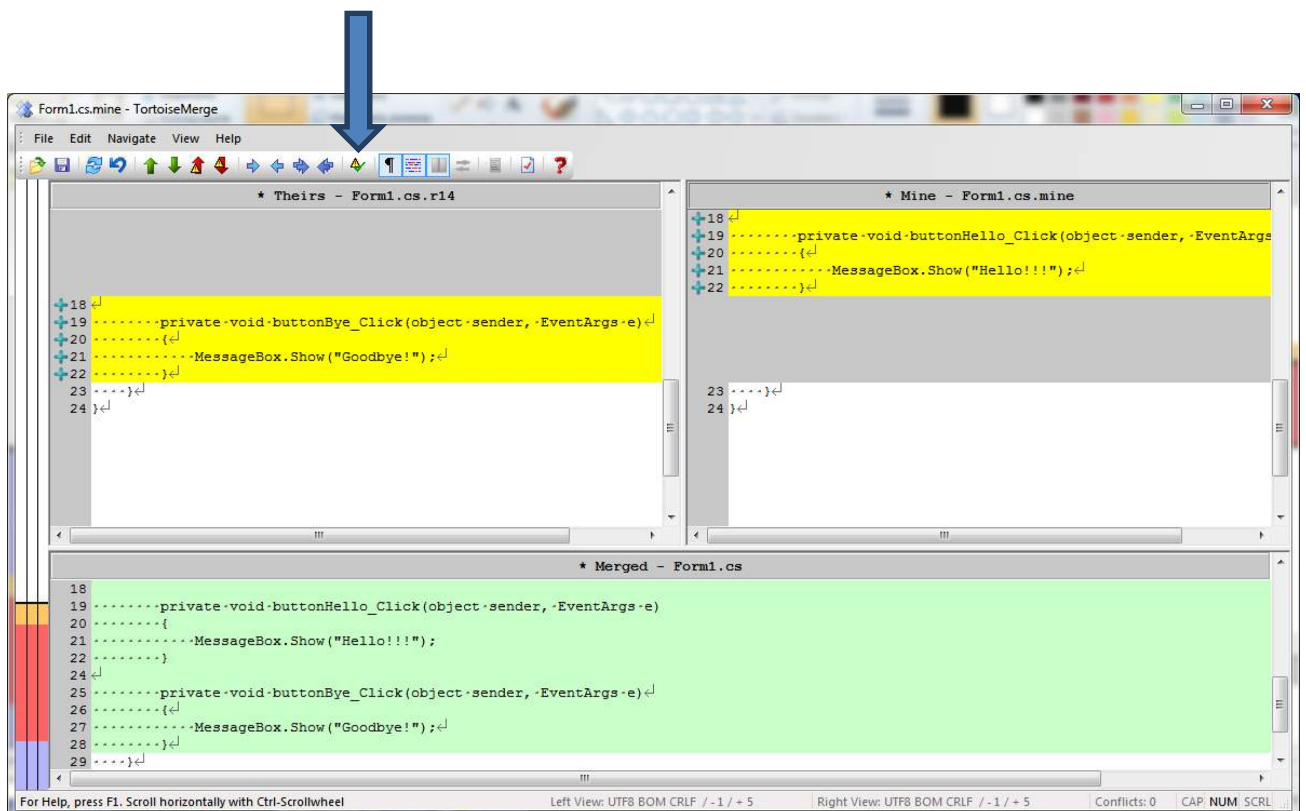


Рисунок 1.33 – Результат злиття і підтвердження

Змінений проект та історія набору змін (рисунок 1.34).
Використання kdiff3 для злиття версій файлів (рисунок 1.35).

Microsoft Visual Studio

D:\khaiStud2\trunk\WindowsFormsApplication\WindowsFormsApplication\Form1.cs

This file has been modified outside of the source editor.
Do you want to reload it?

Yes | Yes to All | No | No to All

Form1

Сказать привет

Сказать пока

KhaiStud

Команда | Активность | Администрирование | Исходный код/SVN | Тикеты | Этапы | Приборна

Просмотр | Наборы изменений | Сеть разветвлений | Сайты | Инструкции | Импорт/экспорт

Наборы изменений

Jan 24	23:08	Ford Foreks *закоммитил* [15] Added say Bye button #4
	22:51	Ford Foreks *закоммитил* [14] Added say Hello button #3

Рисунок 1.34 – Змінений проект та історія набору змін

Використання kdiff3 для злиття версій файлів (рисунок 1.35).

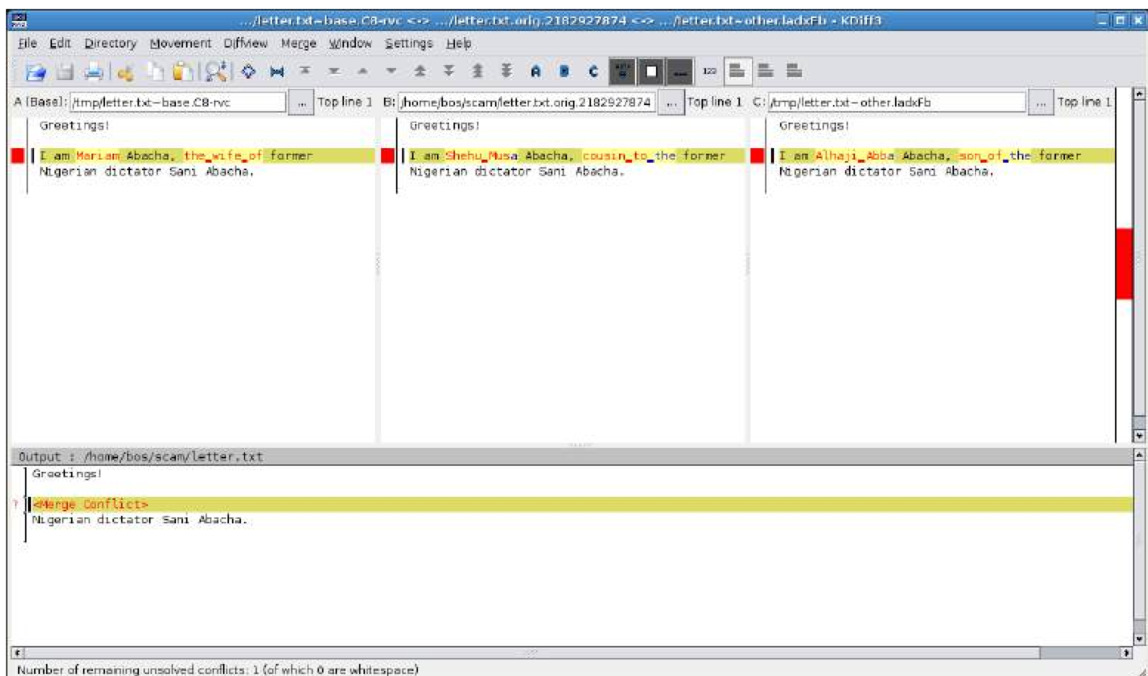


Рисунок 1.35 – Використання kdiff3 для злиття версій файлів

Конфлікти та їх вирішення

Ситуацію, коли при злитті декількох версій зроблені в них зміни перетинаються між собою, називають конфліктом. При конфлікті змін система управління версіями не може автоматично створити об'єднаний проект і змушена звертатися до розробника. Як вже згадувалось раніше, конфлікти можуть виникати на етапах фіксації змін, поновлення або злиття гілок. У всіх випадках при виявленні конфлікту відповідна операція припиняється до його дозволу.

Для вирішення конфлікту система, в загальному випадку, пропонує розробнику три варіанти конфліктуючих файлів: базовий, локальний і серверний. Конфліктуючі зміни або показуються розробнику в спеціальному програмному модулі об'єднання змінень (в цьому випадку там демонструються варіанти, що зливаються, і динамічно змінюваний залежно від команд користувача об'єднаний варіант файла), або просто позначаються спеціальною розміткою прямо в тексті об'єднаного файла (тоді розробник повинен сам сформувати бажаний текст у спірних місцях і зберегти його).

Конфлікти у файловій системі вирішуються простіше: там може конфліктувати тільки видалення файла з однією з інших операцій, а порядок файлів у каталозі не має значення, тобто розробнику залишається лише вибрати, яку операцію потрібно зберегти у злитій версії [16].

Блокування

Механізм блокування дозволяє одному з розробників захопити в монопольне використання файл або групу файлів для внесення до них змін. На той час, поки файл заблокований, він залишається доступним усім іншим розробникам тільки на читання, і будь-яка спроба внести до нього зміни «відкидається» сервером. Технічно блокування може бути організованим по-різному. Типовим для сучасних систем є такий механізм [17].

• Файли, для роботи з якими потрібне блокування, позначаються спеціальним прапором «блокується». Така позначка може ставитися автоматично при додаванні файла до проекту, зазвичай для цього попередньо створюється список масок імен файлів, які при додаванні повинні ставати такими, що блокуються.

• Якщо файл позначений як такий, що блокується, то при вилученні робочої копії з сервера він отримує у локальній файловій системі атрибут «тільки для читання», що перешкоджає його випадковому редагуванню.

• Розробник, який бажає змінити файл, викликає спеціальну команду блокування (lock) із зазначенням імені цього файла. В результаті роботи цієї команди відбуваються такі дії:

1. Сервер перевіряє, чи не заблокований вже файл іншим розробником; якщо це так, то команда блокування завершується з помилкою «файл заблокований іншим користувачем», і розробник, що

викликав її, повинен очікувати, поки інший користувач не зніме своє блокування.

2. Файл на сервері позначається як «заблокований», зі збереженням ідентифікатора розробника, що заблокував його, і часу блокування.

3. Якщо блокування на сервері пройшло вдало, на локальній файловій системі з файла робочої копії знімається атрибут «тільки для читання», що дозволяє почати його редагувати [18].

• Розробник працює із заблокованим файлом. Якщо в процесі роботи з'ясовується, що файл змінювати не потрібно, розробник може викликати команду розблокування (unlock, release lock). Всі зміни файла будуть скасовані, локальний файл повернеться в стан «тільки для читання», з файла на сервері буде знято атрибут «заблокований», а інші розробники отримають можливість змінювати цей файл.

• По завершенні роботи з файлом, що є заблокованим, розробник фіксує зміни. Зазвичай блокування при цьому знімається автоматично, хоча в деяких системах блокування потрібно знімати вручну після фіксації або вказувати в команді фіксації змін відповідний параметр. Так чи інакше, при цьому файл після змін втрачає прапор «заблокований» і може бути змінений іншими розробниками.

Масове використання блокувань, коли всі або більшість файлів у проекті є такими, що блокуються, і для будь-яких змін необхідно заблокувати відповідний набір файлів, називається ще стратегією «блокованого вилучення» [19]. Ранні системи управління версіями підтримували виключно цю стратегію, запобігаючи таким способом появі конфліктів на корені. В сучасних VCS кращим є використання неблокуючих вилучень, блокування же вважаються, швидше, «неминучим злом», яке потрібно, по можливості, обмежувати. Недоліки використання блокувань очевидні [20]:

• Блокування просто заважають продуктивній роботі, оскільки змушують чекати звільнення блокованих файлів, хоча в більшості випадків навіть спільні зміни одних і тих же файлів, які робляться в ході різних за змістом робіт, не перетинаються і поєднуються при злитті автоматично.

• Частота виникнення конфліктів і складність їх вирішення в більшості випадків не настільки великі, щоб створити серйозні труднощі. Виникнення ж серйозного конфлікту змін найчастіше сигналізує або про суттєве розходження в думках різних розробників щодо дизайну одного і того ж фрагмента, або про неправильну організацію роботи (коли два або більше розробників роблять одне і те ж).

• Блокування створюють адміністративні проблеми. Типовий приклад: розробник може забути зняти блокування з зайнятих їм файлів, йдучи у відпустку. Для вирішення подібних проблем доводиться застосовувати адміністративні заходи, в тому числі включати в систему технічні засоби для скидання неправильних блокувань, але з-за їхньої наявності на приведення системи до ладу витрачається час.

З іншого боку, в деяких випадках використання блокувань цілком виправдано. Очевидним прикладом є організація роботи з бінарними файлами, для яких немає інструментальних засобів злиття змін або таке злиття принципово неможливе (як, наприклад, для файлів зображень). Якщо автоматичне злиття неможливе, то при звичайному порядку роботи будь-яка паралельна зміна подібних файлів буде призводити до конфлікту. В цьому випадку набагато зручніше зробити файл таким, що блокується, для того, щоб гарантувати, що будь-які зміни до нього будуть вноситися тільки послідовно.

Розгалуження

Робити дрібні виправлення у проекті можна шляхом безпосередньої правки робочої копії та подальшої фіксації змін прямо в головній гілці (у стовбурі) на сервері. Однак при виконанні об'ємних робіт такий порядок стає незручним: відсутність фіксації проміжних змін на сервері не дозволяє працювати над чим-небудь у груповому режимі, крім того, підвищується ризик втрати змін при локальних аваріях і втрачається можливість аналізу і повернення до попередніх варіантів коду в межах цієї роботи. Тому для таких змін звичайною практикою є створення гілок (branch), тобто «відгалудження» від стовбура в якійсь версії нового варіанту проекту або його частини, розроблення в якому ведеться паралельно зі змінами в основній версії. Гілка створюється спеціальною командою. Робоча копія гілки може бути створена заново звичайним способом (командою вилучення робочої копії, із зазначенням адреси або ідентифікатора гілки), або шляхом перемикання наявної робочої копії на задану гілку [21].

Базовий робочий цикл при використанні гілок залишається таким самим, як і в загальному випадку: розробник періодично оновлює робочу копію (якщо з гілкою працює більше однієї людини) і фіксує в ній свою щоденну роботу. Іноді гілка розробки так і залишається самостійною (коли зміни породжують новий варіант проекту, який далі розвивається окремо від основного), але частіше за все, коли робота, для якої створена гілка, виконана, гілка реінтегрується в стовбур (основну гілку). Це може робитися командою злиття (зазвичай merge), або шляхом створення патча (patch), що містить внесені в ході розроблення гілки зміни і застосування цього патча до поточної основної версії проекту.

Відгалуження і мітки

- Однією з можливостей систем управління версіями є здатність виділити зміни в окрему лінію розроблення. Ця лінія відома як відгалуження. Відгалуження часто використовуються для випробування нових можливостей без порушення основної лінії розроблення помилками компіляції і дефектами. Коли нові можливості стануть досить стійкими, тоді гілка розроблення зливається з основною гілкою (стволом).
- Іншою можливістю систем управління версіями є здатність позначати приватні ревізії (наприклад, версію випуску), тому Ви зможете

в будь-який час відтворити конкретне збирання або оточення. Цей процес відомий як створення мітки.

Злиття діапазону ревізій [22]

- Цей метод застосовується у разі, коли Ви зробили одну або кілька ревізій у відгалуженні (або в основному стовбурі) і бажаєте перенести ці зміни і в інше відгалуження.
- Виходить, що в цій ситуації Ви наказували Subversion виконати такі дії: «Обчисліть зміни, необхідні, щоб [ВІД] ревізії 1 відгалуження А перейти [ДО] ревізії 7 відгалуження А, і застосуйте ці зміни до моєї робочої копії (що відноситься до стовбура або відгалуження Б)».
- Якщо Ви залишили діапазон ревізій порожнім, то Subversion використовує функції відстеження злиття для обчислення правильного діапазону ревізій. Це відомо як возз'єднувальне або автоматичне злиття.
- Іншою можливістю систем управління версіями є здатність позначати приватні ревізії (наприклад, версію випуску), так що Ви зможете в будь-який час відтворити конкретне збирання або оточення. Цей процес відомий як створення мітки.

На рисунку 1.36 зображено скріншот злиття діапазону ревізій.

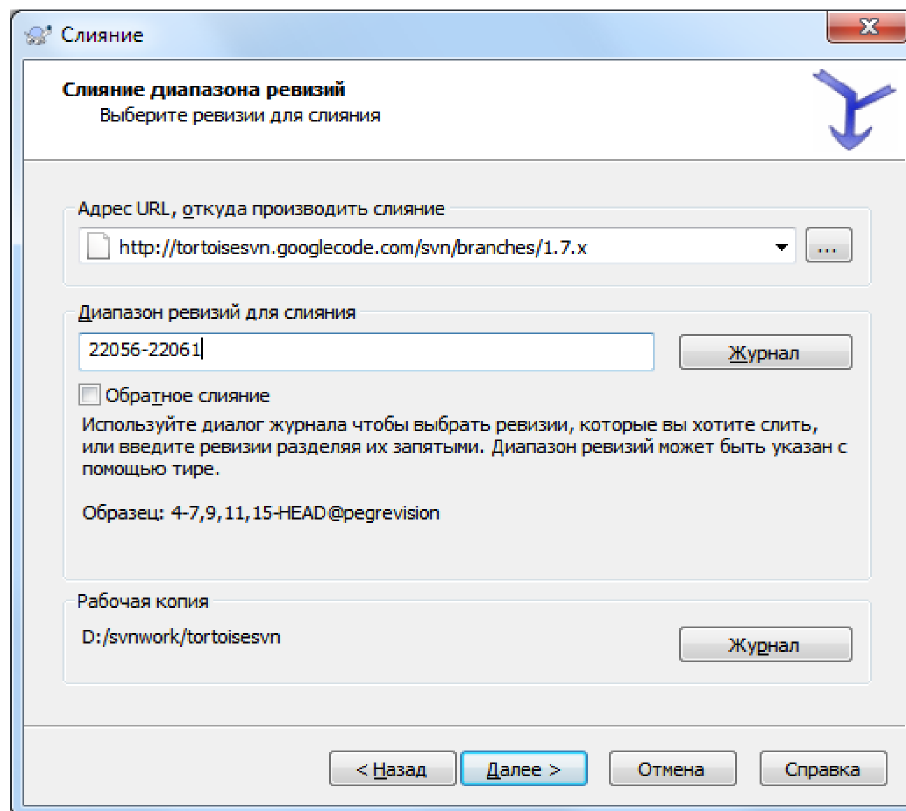


Рисунок 1.36 – Злиття діапазону ревізій

- Це більш загальний випадок методу возз'єднання. Ви наказуєте Subversion виконати такі дії: «Обчисліть зміни, необхідні, щоб [ВІД] провідної ревізії стовбура перейти [ДО] провідної ревізії відгалуження, і застосуйте ці зміни до моєї робочої копії (що відноситься до стовбура)».

У кінцевому підсумку стовбур буде мати такий самий вигляд, як і відгалуження.

- Якщо Ваш сервер / сховище не підтримують відстеження злиттів, то це є єдиним способом здійснення злиття відгалуження назад у стовбур. Інший типовий випадок використання виникає при використанні відгалужень для коду сторонніх виробників (vendor branches), коли Вам необхідно провести злиття змін, що виникають після появи нової версії стороннього коду, з Вашим кодом у стовбурі. На рисунку 1.36 зображено скріншот злиття двох різних дерев.

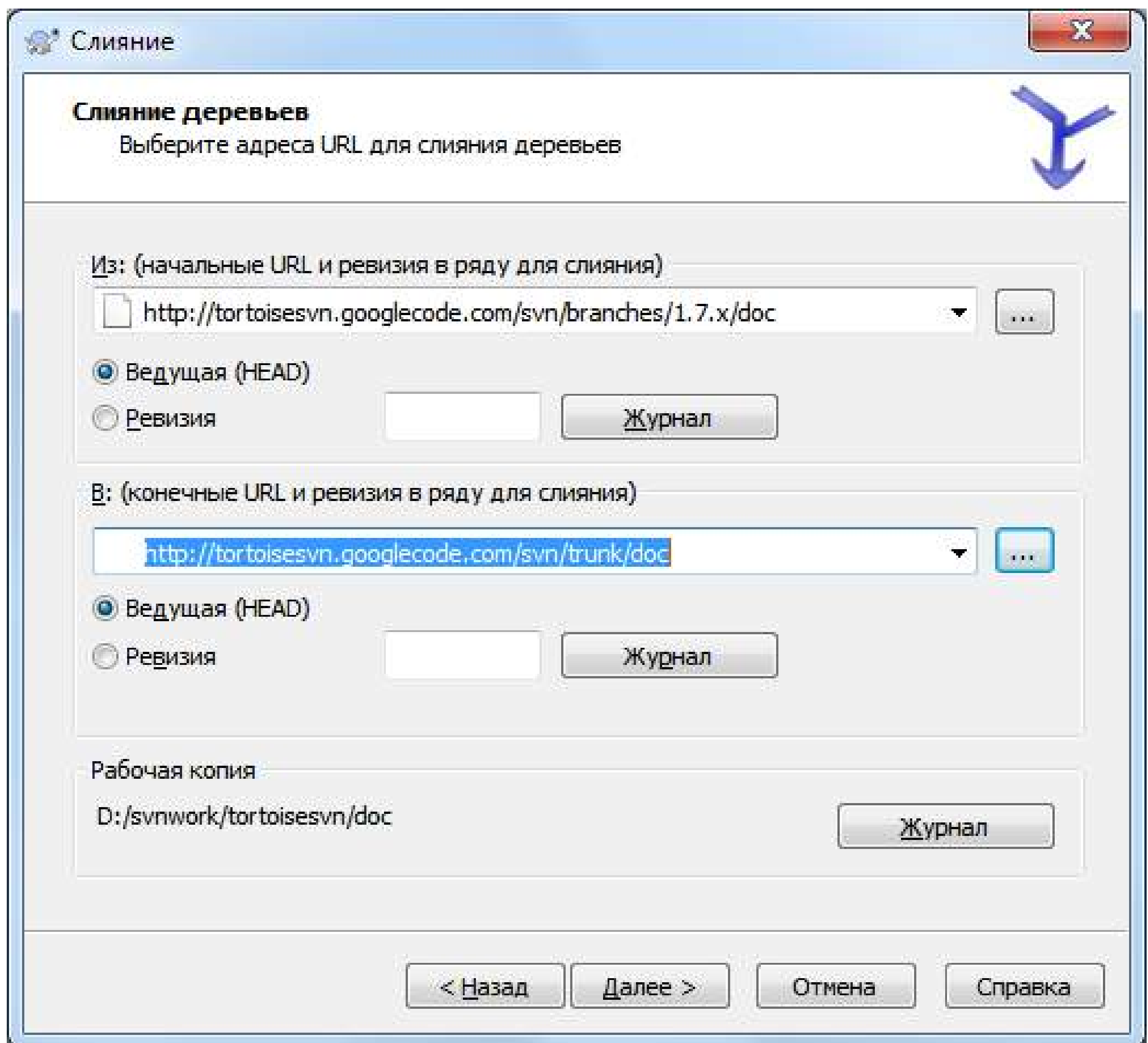


Рисунок 1.37 – Злиття двох різних дерев

Створення відгалуження або мітки

- *Провідна ревізія у сховище (HEAD).* У нове відгалуження копіюється провідна ревізія безпосередньо у сховище. Не треба

передавати ніяких даних з Вашої робочої копії, і відгалуження створюється дуже швидко.

- *Зазначена ревізія у сховище.* Нове відгалуження копіюється безпосередньо у сховище, але Ви можете вибрати ранішу ревізію. Це корисно, якщо Ви забули зробити мітку, коли Ви випускали проект минулого тижня. Якщо Ви не пам'ятаєте номер ревізії, натисніть кнопку праворуч для відображення журналу ревізій і виберіть номер ревізії там. І в цьому випадку дані з Вашої робочої копії не будуть передані іншим розробникам, відгалуження створюється дуже швидко.
- *Робоча копія.* Нове відгалуження буде ідентичною копією Вашої локальної робочої копії. Якщо Ви оновили деякі файли до старих ревізій у Вашій робочій копії, або якщо Ви зробили локальні зміни, саме це і увійде в копію. Природно, що цей різновид складних команд може призводити до передачі даних з Вашої робочої копії назад у сховище, якщо їх там поки немає.

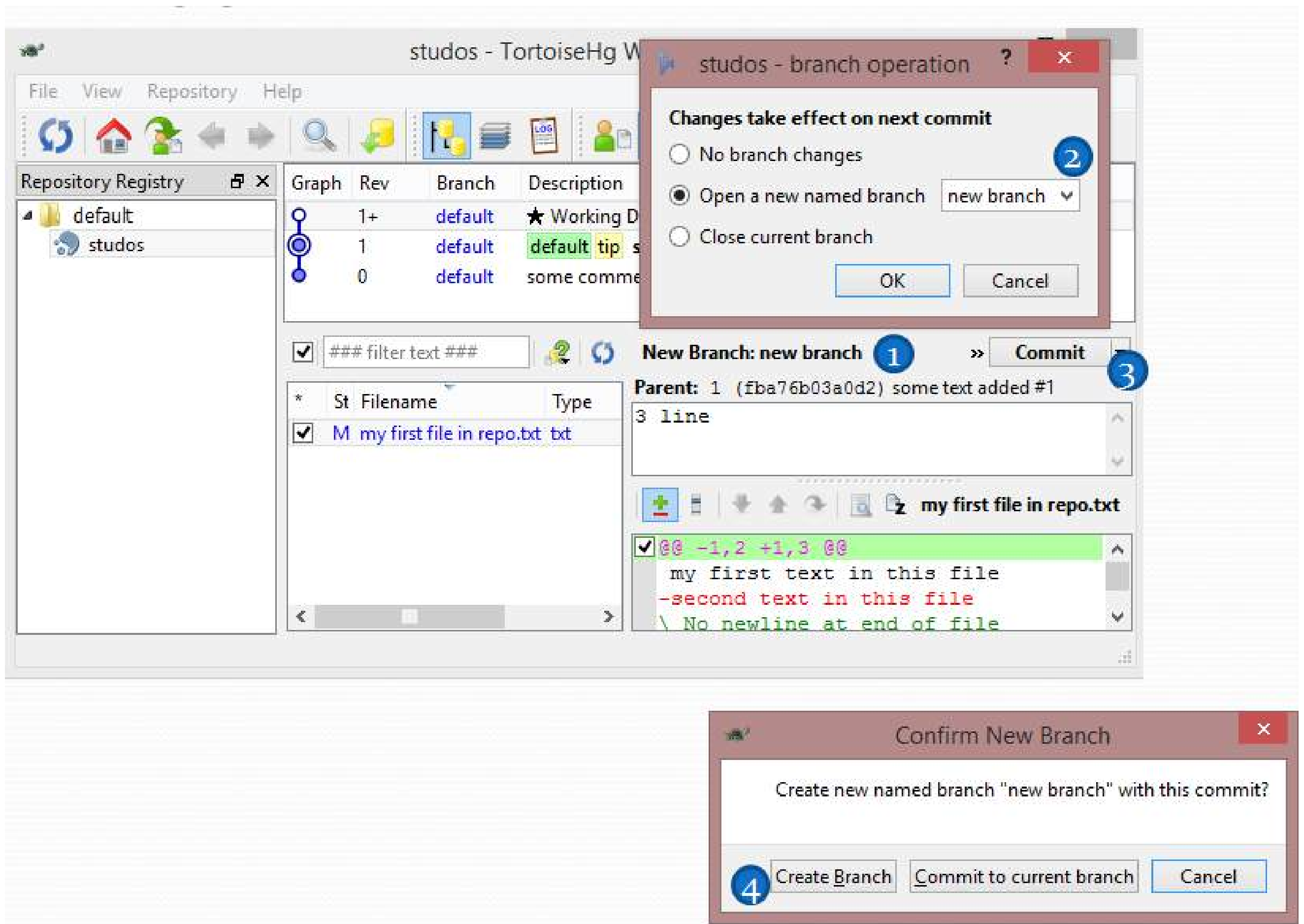


Рисунок 1.38 – Приклад створення розгалудження у Tortoise HG WorkBench

Переходи (переключення) між гілками проекту (рисунок 1.39).

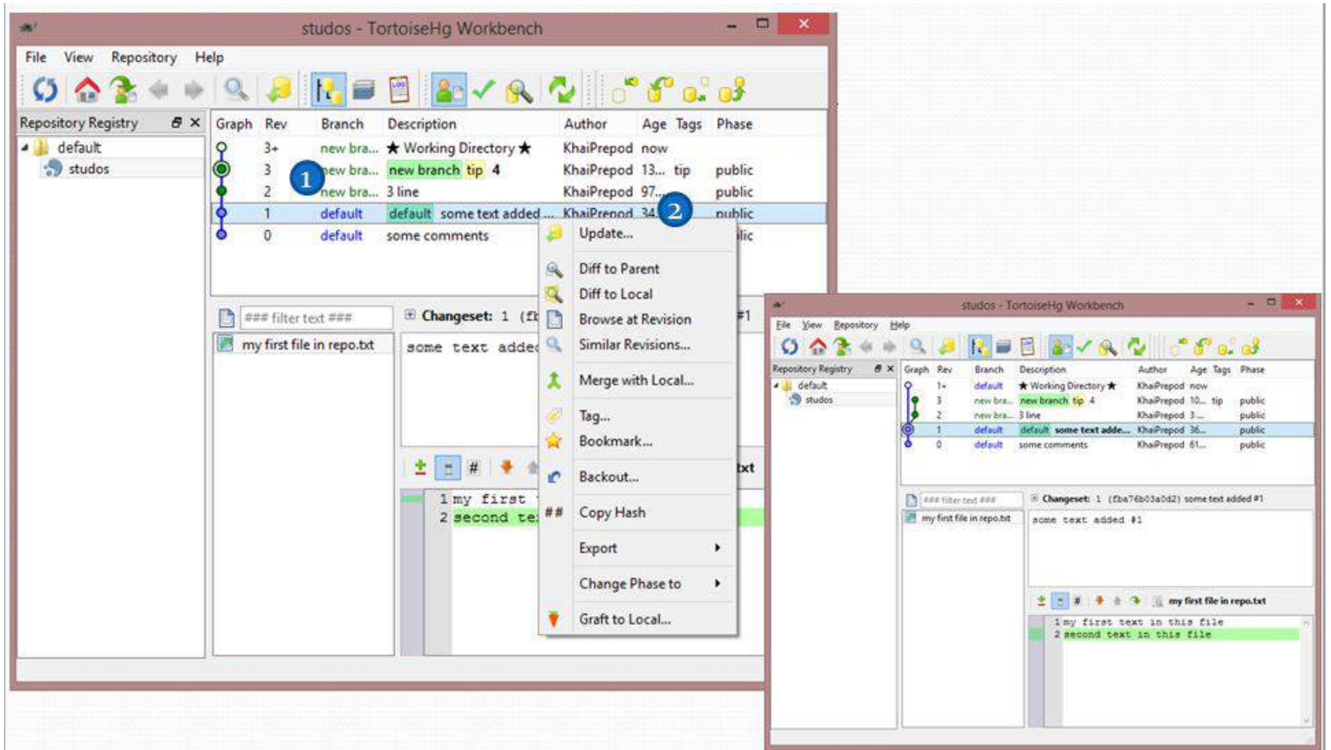


Рисунок 1.39 – Приклад переходу (переключення) між гілками проекту в Tortoise HG WorkBench

Злиття гілок (рисунок 1.40).

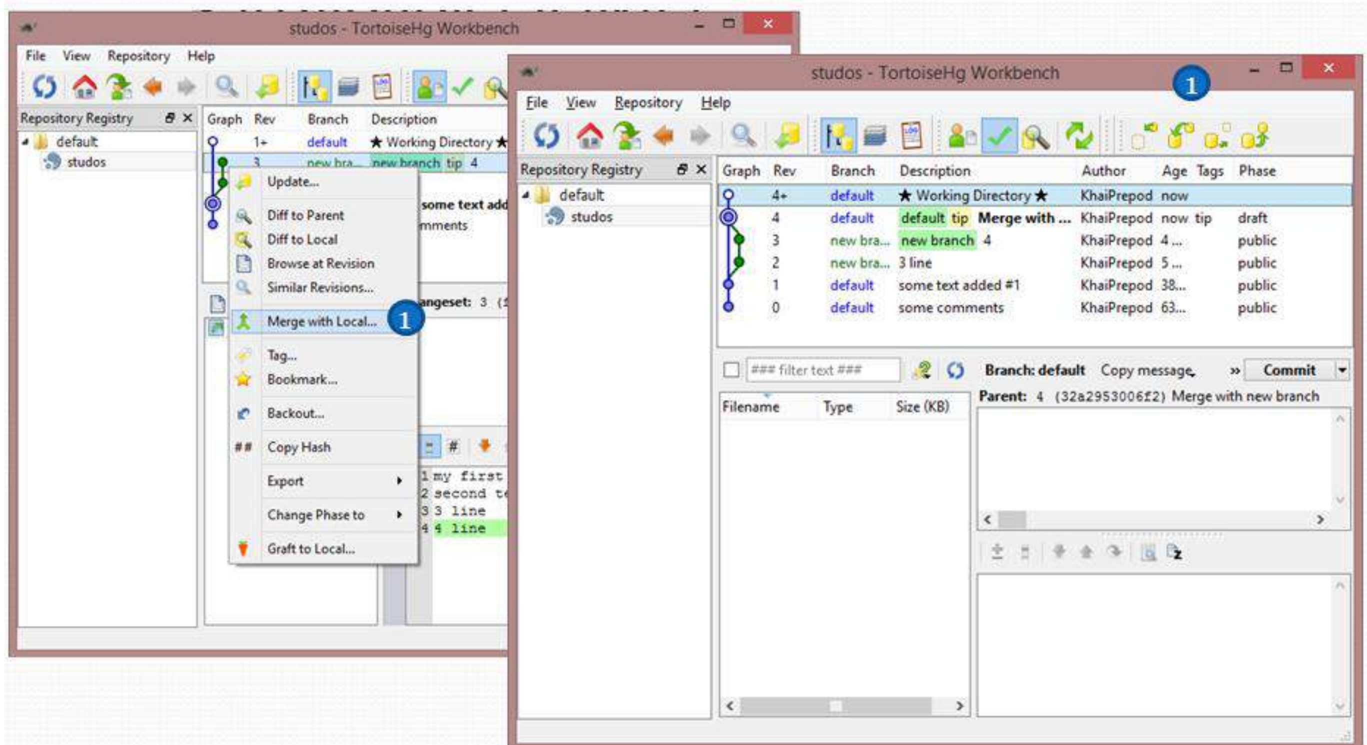


Рисунок 1.40 – Приклад злиття гілок у Tortoise HG WorkBench

Висновки до першої глави

Мотивація

- Вихідний код змінюється і змінюється «нелінійно»:
  - відкат невдалих змін;
  - паралельні гілки розроблення;
  - командна робота:
    - конфліктуючі зміни.
- У найпростішому випадку ще можна:
  - зберігати архіви усіх версій;
  - обмінюватися кодом поштою або якимось іншим способом.
- Але навіщо? Адже є системи контролю версій (Version Control Systems – VCS).

Що дають VCS

- Безпека:
  - транзакційність;
  - авторизація доступу;
  - легко робити backup / restore сховища.
- Організація гілок розроблення:
  - розгалуження історії версій;
  - маркування версій.
- Ефективність:
  - простота операцій над історією;
  - інтеграція:
    - в IDE («з коробки» або у вигляді плагінів);
    - в OS (командний рядок, інтеграція в Windows Explorer).

Область

- VCS добре підходять для:
  - зберігання історії правок;
  - текстових форматів даних.
- Погано підходять для:
  - баг-трекінгу, зліпків баз даних, управління конфігураціями;
  - зберігання бінарних форматів даних.

Базовий сценарій використання VCS

1. Отримати локальну «робочу копію» коду зі сховищ.
2. Внести зміни.
3. Варіативність: виконати злиття (merge) змін з новими правками в репозитарії.
4. Зафіксувати зміни у репозитарії.

Види VCS

- **Централізовані:**
  - більш давніший вигляд VCS;
  - використовувалися ще у 70-ті роки.
- **Розподілені:**

– «нова течія»;
– перші системи – 90-ті, початок 2000-х;
– масове поширення – з 2005 року.

Централізовані VCS

– Єдине сховище версій – центральний репозитарій.

– Розробник працює з локальною копією і відправляє зміни у центральний репозитарій.

– Репозитарій видно всім (у кого є доступ), і обмін кодом – тільки через нього.

– **Приклади:** SVN, Perforce, MS TFS, ClearCase.

Розподілені VCS

– Кожен розробник володіє копією сховища:
  – фактично, своїм локальним «сервером» VCS;
  – копії легко створювати: простіше експериментувати з кодом.
– Передавати зміни можна між будь-якою парою сховищ;
– Немає «головного» сховища:
  – Можна будувати більш складну модель «обороту» змін:
    – персональні репозитарії розробників;
    – командні та цільові репозитарії (для тестування, для гілок розроблення, автоматичних збирань, і т. п.).
– **Приклади**: Git, Mercurial, Bazaar.

CVCS vs. DVCS:

– На DVCS можна виконувати те ж саме, що і на CVCS;
– Покращення в DVCS:
  – простіше виконувати злиття гілок;
  – вся історія зберігається локально:
    – можна працювати офлайн;
    – робота в цілому виконується швидше.
– Більш гнучка модель обміну змінами.

SVN vs. git&hg:

– svn-файли по всьому дереву каталогів;
– git і hg зберігають свої дані в окремій папці;
– простіше робити експерименти;

– клонування локального сховища (у svn довелось би робити гілки, які потім можна видалити).

Нюанси DVCS:

– розробники звикли до VCS, отже потрібно перелаштовуватися;
– у CVCS нижче «поріг входження»;

– для роботи з DVCS треба краще розуміти концепції контролю версій;

– у світі CVCS є фаворит – SVN, у DVCS поки не виявлено «переможця».

Робота з централізованим репозитарієм на прикладі Subversion (SVN)
Коротко про SVN:

- З'явилася у 2004 році як заміна CVS;
- Широко використовується в Open source:
  - Apache, GCC, Python, Ruby, Boost, ... ;
  - репозитарій SourceForge.net, GoogleCode, etc.
- Одна з найпопулярніших SVC.

Термінологія

– Репозитарій (repository) – сховище документів, місце, де SVN зберігає документи, історію їх зміни і службову інформацію;

– Ревізія (revision) – версія документа, використовується автоматична нумерація;

– Робоча копія (working copy) – локальна копія документів.

Типовий процес роботи

1. Отримання робочої копії:
- svn checkout – створення (отримання);
- snv update – оновлення робочої копії до заданої ревізії, за замовчуванням – до новішої;
- svn blame – інформація про зміни та їх авторів.
2. Зміна робочої копії:
- svn add | mkdir | delete | move | copy;
- перегляд стану і змін файлів;
- svn info | status | diff;
- svn revert – відкат змін.
3. За необхідності – svn update і злиття змін.
4. Фіксація змін – svn commit.

Розгалуження

- Створення гілки – svn copy над репозитарієм;
- Робота з гілками:
  - svn switch – перемикання робочої копії на іншу гілку;
  - svn merge – злиття змін однієї гілки з іншою гілкою.

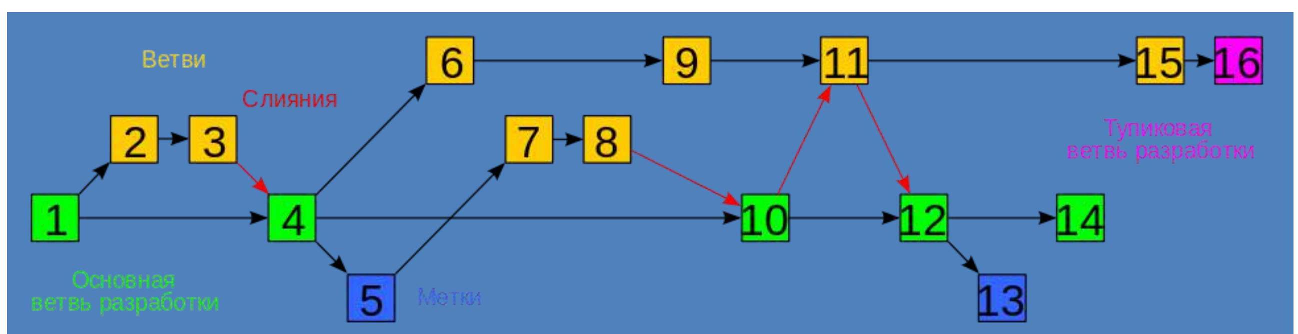


Рисунок 1.41 – Приклад розгалуження в СКВ SVN

На рисунках 1.42 – 1.43 наведено результати використання інструментального засобу KDiff для виконання об'єднань змін вручну при

виникненні конфліктів, у разі, якщо ВКВ не може автоматично виконати вирішення конфліктів при злитті об'єднань.

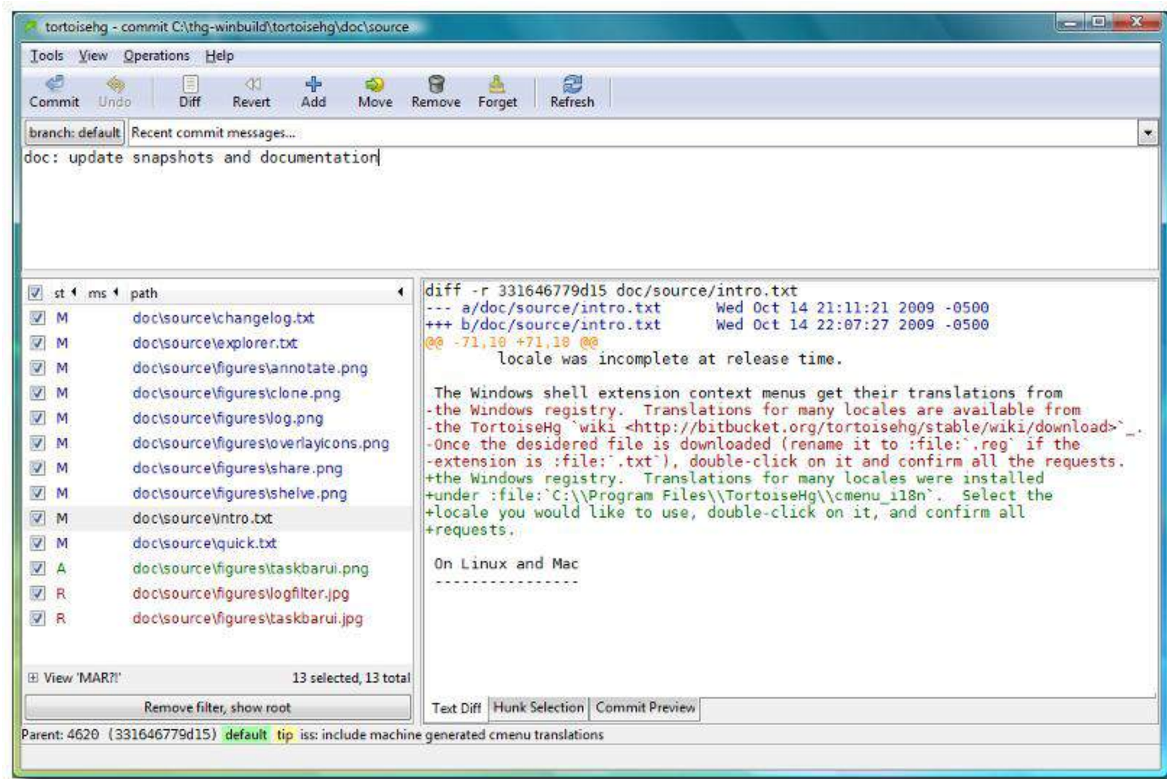


Рисунок 1.42 – Приклад вибору і додавання файлів з використанням інструментального засобу KDiff

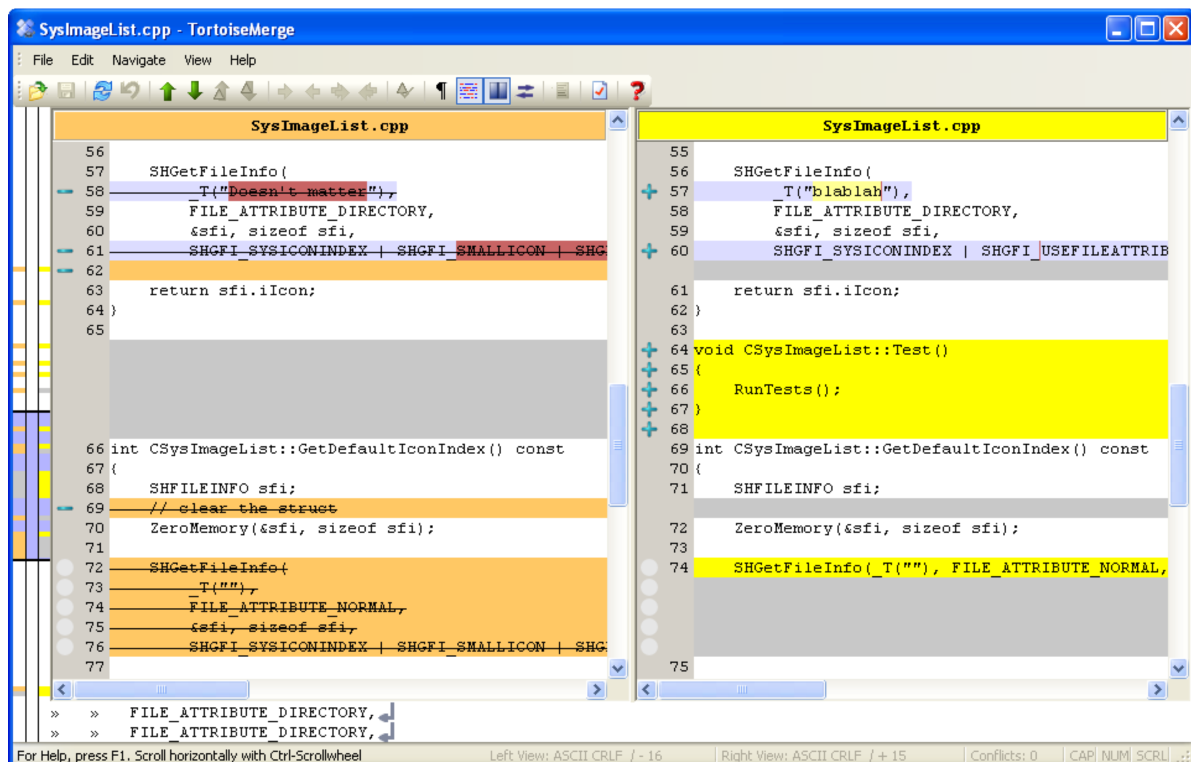


Рисунок 1.43 – Приклад об'єднання змін з використанням інструментального засобу KDiff

Робота з розподіленим репозитарієм Mercurial (hg)
Коротко про Mercurial

- Створювалася як конкурент git;
- Досить популярна VCS:
  - багато проектів на Python / Java;
  - приклади: OpenJDK, Mozilla, NetBeans, ...
- Синтаксис команд близький до SVN;
- В основному написана мовою Python.

Робочий процес

1. Створення сховища:
- hg init – «з нуля» у поточній директорії:
- hg addremove – додавання наявних файлів до репозитарію;
- hg commit – зафіксувати початкову версію;
- hg clone – копія сховища.
2. Зміни:
- отримання змін з віддаленого сховища: hg pull;
- перехід до ревізії: hg update;
- стан сховища: hg status, hg diff, hg cat, hg log;
- фіксація змін: hg commit;
- відкат змін: hg revert [--all].
3. Проштовхування змін у віддалений репозитарій:
- перегляд змін, які відправляють: hg outgoing;
- hg push;
- при конфліктах: hg pull + hg merge + hg push.

Робота з розподіленим репозитарієм git
Коротко про git

- Створювався для сховища ядра Linux;
- Найпопулярніша DSVC:
  - особливо в Open-source проектах, у Linux- і Ruby-спільнотах;
  - використовують: Linux Kernel, GitHub (в т. ч. Ruby on Rails).

Концепція

- Залежно від стану, файли зберігаються:
  - Committed files – у Repository;
  - Modified (редаговані) – у Working Directory;
  - Staged (відмічені для включення в комерц) – у Staging Area.
- Зберігає «знімки» версій файлів, а не зміни між версіями.

Робочий процес

1. Створення сховища:
- git init – «з нуля» в поточній директорії;
- git add. – додавання наявних файлів у Staged;
- git clone – копія сховища.
2. Зміна:

– Нова гілка:
  – git branch my-branch – створити;
  – git checkout my-branch – переключитися на гілку.
– Індексація змін у гілці: git add | rm;
– Стан проекту (файлів): git status;
– Фіксація змін: git commit;
– Відкат змін:
  – "unstage" – git reset;
  – і відкат до робочої версії – git checkout.

3. Злиття гілок:

– перемикання на «базову» гілку: git checkout master-branch;
– отримання останніх змін: git pull;
– злиття: git merge my-branch.

4. Проштовхування змін у віддалений репозитарій:

– git push.

Посилання

1. SVN:
   - http://ru.wikipedia.org/wiki/Subversion
   - Офсайт: http://subversion.apache.org/
   - Клієнт TortoiseSVN: http://tortoisesvn.net/
   - Коротка інструкція: http://www.source-team.com/svnfordummies
   - Ще інструкція: http://habrahabr.ru/post/150799/
2. Mercurial (Hg):
   - Офсайт: http://mercurial.selenic.com/ - там же можна отримати версію разом з GUI-клієнтом TortoiseHg
   - Великий набір матеріалів: http://mercurial.ru/
   - Керівництво по Hg от Joel Spolsky: http://hginit.com/
   - Цикл статей-перекладів "Hg Init" російською мовою: частина 1, частина 2, 3, 4, 5, 6 (якщо не знайомі з SVN, можна почати з ч.2)
   - Робота з TortoiseHg: http://dreamhelg.ru/2009/02/tortoisehg/
3. Git:
   - Офсайт: http://git-scm.com/
   - Документація російською мовою: http://git-scm.com/book/ru/
   - Короткий вступ: http://www.rsdn.ru/article/tools/Git.xml

1.1 Огляд 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 змінюваною інформацією і бажають полегшити собі життя. Внаслідок цього з'явилася велика кількість різних продуктів, що пропонують широкі можливості та надають обширні інструменти для управління версіями. У рамках цього розділу коротко розглянуті найбільш популярні з них і наведені їхні переваги і недоліки.

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

RCS – система управління переглядами версій (www.gnu.org/software/rcs/rcs.html)

Огляд доцільно почати з однієї з перших систем контролю версій – RCS (Revision Control System – система управління переглядами версій), розробленої у 1985 році. Вона прийшла на зміну популярної в той час системи контролю версій SCCS (Source Code Control System – системи управління вихідним кодом).

Зараз 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 файла, а тільки остання версія і всі зміни, внесені до неї. RCS також може відстежувати зміни в бінарних файлах, але при цьому кожна зміна зберігається у вигляді окремої версії файла.

Коли зміни у 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 й 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 встановлена у більшості вільно розповсюджуваних операційних системах.

Недоліки:

1. 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, у порівнянні з сучасними системами контролю версій.

Висновки:

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

CVS – система управління паралельними версіями [(www.nongnu.org/cvs)](http://www.nongnu.org/cvs)

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 основана на технології клієнт-сервер, що взаємодіють у 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 репозитарії сервера. Клієнти, підключаючись до сервера, перевіряють відмінності локальної версії від останньої версії, збереженої в репозитарії, і, якщо є відмінності, завантажують їх у свій локальний проект. За необхідності вирішують конфлікти і вносять необхідні зміни у розроблюваний продукт. Після цього всі зміни завантажуються в репозитарій сервера. CVS, за необхідності, дозволяє відкочуватися на потрібну версію розроблюваного проекту і вести управління декількома проектами одночасно.

Наведемо основні переваги і 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 репозитарію передаються тільки зміни, а не весь файл цілком.

Недоліки:

1. При переміщенні або перейменуванні файла або директорії втрачаються всі, прив'язані до цього файла або директорії, зміни.

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

3. Обмежена підтримка шрифтів.

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

5. Від клієнта до серверу змінений файл завжди передається повністю.

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

Висновки:

Незважаючи на те, що CVS застаріла і має серйозні недоліки, вона все ще є однією з найпопулярніших систем контролю версій і відмінно підходить для управління невеликими проектами, які не потребують створення декількох паралельних версій, що треба періодично об'єднувати. CVS можна порекомендувати, як проміжний крок в освоєнні роботи систем контролю версій, що веде до більш потужних і сучасних видів таких програм.

Система управління версіями Subversion (www.subversion.tigris.org)

Subversion – ця централізована система управління версіями, створена в 2000 році та основана на технології клієнт-сервер. Вона має усі переваги CVS і вирішує основні її проблеми (перейменування і переміщення файлів і каталогів, робота з двійковими файлами і т. д.). Часто її називають згідно з іменем 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 у репозитарії, то такі ситуації вирішуються вручну. Потім у локальний проект вносяться зміни, і отриманий результат зберігається в репозитарії.

Під час роботи з файлами, що не дозволяють об'єднувати зміни, використовують такий принцип:

1. Файл завантажують зі сховищ і блокують (забороняється його скачування з репозитарію).

2. Вносять необхідні зміни.

3. Завантажують файл до репозитарію і розблоковують (дозволяється його скачування з репозитарію іншим клієнтам).

Багато в чому, через простоту і схожість в управлінні з CVS, але в основному, через свою широку функціональність, Subversion з успіхом конкурує з CVS і навіть успішно її витісняє.

Однак і у Subversion є недоліки. Давайте 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 та переміщення.

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

6. Вбудована підтримка в багатоінтегровані засоби розроблення, такі як KDevelop, Zend Studio і багато інших.

7. Можливість створення дзеркальних копій сховища.

8. Два типи 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 зіставити певний набір властивостей, що полегшує взаємодію з системою контролю версії.

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

Недоліки:

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

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

3. Слабо підтримуються операції злиття гілок проекту.

4. Складнощі з повним видаленням інформації про файли потрапили до репозитарію, оскільки в ньому завжди залишається інформація про попередні зміни файла і не передбачено ніяких штатних засобів для повного видалення даних про фото зі сховищ.

Висновки:

Subversion – сучасна система контролю версій, що володіє широким набором інструментів, які 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 у конкурентній гонці систем контролю версій.

Система управління версіями Aegis **(www.aegis.sourceforge.net)**

Aegis, створена Пітером Міллером у 1991 році, є першою альтернативою централізованим системам управління версіями. Всі операції в ній проводяться через файлову систему Unix. На жаль, у Aegis немає вбудованої підтримки роботи у мережі, але взаємодії можна здійснювати, використовуючи такі протоколи, як NFS, HTTP, FTP.

Основна особливість Aegis – це спосіб контролю змін, що вносяться до репозитарію.

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

По-друге, перед внесенням змін до основної гілки розроблюваного проекту вони повинні бути схвалені оглядачем.

По-третє, передбачена ієрархія доступу до сховища основана 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 дуже це полегшує.

Виділимо основні переваги і недоліки системи контролю версій Aegis.

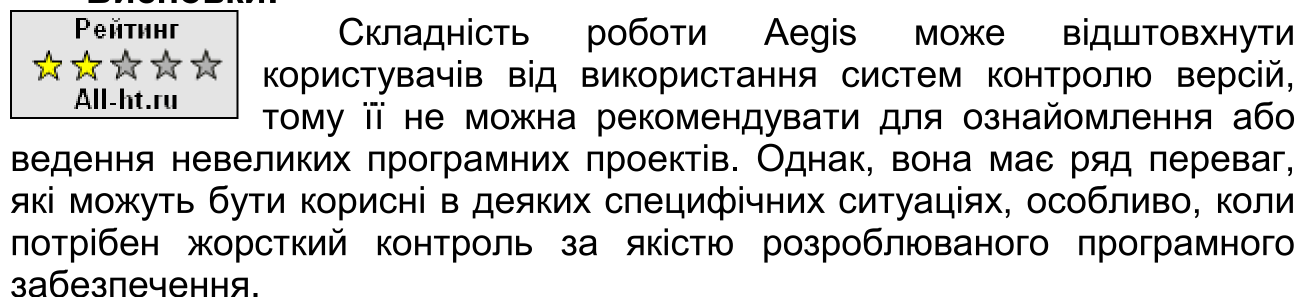
Переваги:

1. Надійний контроль коректності завантажуваних змін.
2. Можливість надавати різні рівні доступу до файлів сховища, що дає пристойний рівень безпеки.
3. Якісна документація.
4. Можливість перейменовувати файли, які збережені у репозитарії, без втрати історії змін.
5. Можливість роботи з локальним репозитарієм, якщо відсутній мережний доступ до головного сховища.

Недоліки:

1. Відсутність вбудованої підтримки мережної взаємодії.
2. Складність налаштування репозитарію і роботи з ним.
3. «Слабкі» графічні інтерфейси.

Висновки:



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 потрібен жорсткий контроль за якістю розроблюваного програмного забезпечення.

Система управління версіями Monotone (monotone.ca)

Monotone – ще одна децентралізована система управління версіями, розроблена Грейдоном Хоєм. 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 таким чином. У першу чергу, створюється база даних проекту SQLite, і генеруються ключі з використанням алгоритму хешування SHA1 (Secure Hash Algorithm 1).

Потім, у процесі коригування проекту користувачем, усі зміни зберігаються в цій базі даних, аналогічно збереженню змін в репозитарії 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 у зв'язці ключів своїх локальних проектів Monotone;
- тепер кожен, зареєстрований таким чином, користувач може синхронізувати розроблення зі своїми колегами, використовуючи простий набір команд.

Узагальнемо 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) збіги хеш-коду відмінних за змістом ревізій.

Висновки:

Monotone – це потужний і зручний інструмент для управління версіями розроблюваного проекту. Набір команд – продуманий і інтуїтивно зрозумілий, особливо, він буде зручним для користувачів, які звикли до роботи с Subversion і CVS. Чудово 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, зроблять роботу з великими проектами декілька скрутною. Тому, якщо Вам потрібна система контролю версій для підтримки складних і об'ємних продуктів, варто звернути увагу на Git або Mercurial.

Система управління версіями Git (www.git-scm.com)

З лютого 2002 року для розроблення ядра Linux'a більшість програмістів почала використовувати систему контролю версій BitKeeper. Досить довгий час з нею не виникало проблем, але в 2005 році Ларі МакВоем (розробник BitKeeper'a) відкликав безкоштовну версію програми.

Розробляти проект масштабу Linux без потужної і надійної системи контролю версій – неможливо. Однією з кандидатів і найбільш підходящим проектом виявилася система контролю версій Monotine, але Торвальдса Лінуса не влаштувала її швидкість роботи. Оскільки особливості організації Monatone не дозволяли значно збільшити

швидкість оброблення даних, то 3 квітня 2005 року Лінус приступив до розроблення власної системи контролю версій – Git.

Практично одночасно з Лінусом (на три дні пізніше), почав розроблення нової системи контролю версій і Метт Макал. 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 і відстеження зміни «рукописів» і сюжетних ліній, і вчителям для коригування і розвитку курсу лекцій, і адміністраторам для ведення документації, і для багатьох інших напрямків, що потребують управління історією змін.

У кожного розробника, який використовує Git, є свій локальний репозитарій, що дозволяє локально керувати версіями. Потім, зі збереженими у локальному репозитарії даними, можна обмінюватися з іншими користувачами.

Часто при роботі з Git створюють центральний репозитарій, з яким інші розробники синхронізуються. Приклад організації системи з центральним репозитарієм – це проект розроблення ядра Linux'а (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 й налагоджені, вони, після посвідчення власником центрального сховища в коректності й актуальності виконаної роботи, завантажують свої зміни в центральний репозитарій.

Наявність локальних репозитаріїв також значно підвищує надійність зберігання даних, оскільки, якщо один з репозитаріїв вийде з ладу, дані можуть бути легко відновлені з інших репозитаріїв.

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

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

Переваги:

1. Надійна система порівняння ревізій і перевірки коректності даних, основана 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» у головний репозитарій тільки ті зміни, що повністю пройшли перевірку.

4. Висока продуктивність і швидкість роботи.

5. Зручний та 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 системи контролю або призначені для користувача інтерфейси на базі git.

10. Універсальний мережний доступ з використанням протоколів http, ftp, rsync, ssh та ін.

Недоліки:

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

2. Можливі (але надзвичайно низькі) збіги 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, які містять безліч незв'язаних файлів.

4. При початковому (першому) створенні сховища і синхронізації його з іншими розробниками потрібен досить тривалий час для скачування даних, особливо, якщо проект великий, оскільки потрібно скопіювати на локальний комп'ютер весь репозитарій.

Висновки:

Git – гнучка, зручна і потужна система контролю версій, здатна задовольнити абсолютну більшість користувачів. Існуючі недоліки поступово видаляються і не приносять серйозних проблем користувачам. Якщо Ви ведете великий проект, територіально віддалений, і тим паче, якщо часто доводиться розробляти програмне забезпечення, не маючи доступу до інших розробників (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 сховищі, «відкочуватися», перемикатися між гілками і т. д.). Git – один з лідерів систем контролю версій.

Система управління версіями Mercurial (mercurial.selenic.com)

Розподілена система контролю версій Mercurial розроблялася Меттом Макалом паралельно з системою контролю версій Git, створеною Торвальдсом Лінусом.

Спочатку вона була створена для ефективного управління великими проектами під Linux'ом, а тому була орієнтована на швидку і надійну роботу з великими репозитаріями. На цей час 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'і, підтримується можливість створення гілок проекту з подальшим їх злиттям.

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

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

Розглянемо основні переваги і недоліки Mercurial.

Переваги:

1. Швидке оброблення даних.
2. Кросплатформна підтримка.
3. Можливість роботи з декількома гілками проекту.
4. Простота в обігу.
5. Можливість конвертації репозитаріїв інших систем підтримки версій, таких як CVS, Subversion, Git, Darcs, GNU Arch, Bazaar й ін.

Недоліки:

1. Можливі (але надзвичайно низькі) збіги хеш-коду відмінних за змістом ревізій.
2. Орієнтований на роботу в консолі.

Висновки:

Простий і відточений інтерфейс і набір команд, можливість імпортувати репозитарії з інших систем контролю версій – зроблять перехід на Mercurial і навчання основним особливостям «безболісними» і швидкими. Навряд чи це займе більше декількох днів.

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

Система управління версіями Bazaar (bazaar.canonical.com)

Bazaar – розподілена система контролю версій, що вільно розповсюджується і 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 величезною кількістю файлів, Bazaar орієнтована на зручний і дружній інтерфейс користувача. Оптимізація швидкості

роботи проводилася вже на другому етапі, коли перші версії програми вже з'явилися.

Як і в багатьох інших системах контролю версій, система команд Bazaar'а – дуже схожа на команди CVS або Subversion, що, втім, не дивно, так як забезпечує зручний, простий та 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 репозитаріями інших систем контролю версій, таких, як Subversion або Git.

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

Переваги:

1. Кросплатформенна підтримка.
2. Зручний та інтуїтивно зрозумілий інтерфейс.
3. Проста робота з гілками проекту.
4. Можливість роботи з репозитаріями інших систем контролю версій.
5. Система добре задокументована.
6. Зручний графічний інтерфейс.
7. Надзвичайна гнучкість, що дозволяє підлаштовуватися 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 велику кількість плагінів, що дозволяють повністю розкрити всі можливості системи контролю версій.

Висновки:

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 їхніми репозитаріями – зробить перехід на Bazaar швидким і «безболісним». Про успішність цієї системи свідчить і той факт, що нею користуються розробники Ubuntu Linux.

Система управління версіями Arch.

Arch – розподілена система контролю версій, створена Томом Лордом. Спочатку вона створювалася для вирішення проблем CVS, що розробникам цілком вдалося.

Arch здійснює атомарні операції по збереженню змін в репозитарії, тобто виключає ситуацію скачування сховища, коли частина змін завантажена, а частина ще не встигла завантажитися.

Підтримуються можливості розгалуження версій проекту і об'єднання окремих гілок, перейменування і переміщення файлів і каталогів із збереженням історії змін і багато інших приємних можливостей.

Не потребує спеціального сервісу для мережного сховища та може використовувати такі протоколи, як FTP, SFTP або WebDAV і так далі.

Але, на жаль, підтримується тільки UNIX-системами, проте, переведення Arch під інші операційні системи не повинно викликати особливих труднощів.

Неможливо відзначити якісь принципово кращі якості, порівняно з іншими розподіленими системами контролю версій, такими, як Git, Mercurial, Bazaar, оскільки, якщо є вибір, то краще використовувати щось більш потужне і поширене.

Система управління версіями Perforce (www.perforce.com)

Продовжимо огляд систем контролю версій і перейдемо до комерційних програм. Почнемо з централізованої системи контролю версій – Perforce, розробленої компанією Perforce Software.

Система Perforce має клієнт-серверну організацію і дозволяє одночасно керувати кількома проектами, створюючи для кожного проекту свій репозитарій.

Perforce – кросплатформна система. Існують версії, здатні працювати під управлінням операційних систем Unix, Mac OS X, Microsoft Windows.

Для роботи з системою контролю версій можна використовувати, як консоль, так і спеціально розроблений графічний інтерфейс.

Серйозну перевагу Perforce'у дає можливість інтегруватися з безліччю засобів розроблення програмного забезпечення і такими застосунками, як Autodesk 3D Studio Max, Maya, Adobe Photoshop, Microsoft Office, Eclipse, emacs і багатьма іншими.

Підтримка можливості створення гілок версій проекту, гнучко управляти ними, об'єднувати, відкочуватися на попередні ревізії – робить Perforce цілком конкурентно здатною системою і сприяє її поширенню. Однак, це продукт – комерційний, що дещо звужує область його застосування і стримує поширення. В основному, він використовується у великих комерційних компаніях, для яких важлива не тільки функціональність, але і своєчасна технічна підтримка.

Система управління версіями Team Foundation Server (msdn.microsoft.com/en-us/library/ms364061.aspx)

Власне кажучи, не можна назвати Team Foundation Server (TFC) просто системою контролю версій – це якесь комплексне рішення, до складу якого входить і система управління версіями, і система збирання даних, і побудови звітів й інші корисні функції.

Керований проект при роботі з TFC – це гілки вихідного коду проекту, набори звітів і призначені для користувача елементи. При створенні проекту заздалегідь вибирають його параметри, які можна вибирати самостійно або використовувати шаблони. Шаблони дозволяють визначити шлях розвитку проекту, зробити його гнучким або жорстко формалізованим, закласти стратегію розвитку, врахувати необхідні заготовки (шаблони) документів і звітів.

TFC легко інтегрується з Microsoft Excel і Microsoft Project, що значно полегшує створення і відстеження елементів контрольованих проектів.

Як система контролю версій, TFC дозволяє:

- спільно коригувати файли проекту;
- вирішувати конфлікти;
- створювати гілки проектів, а потім об'єднувати їх;
- керувати доступом до сховища;
- відкочуватися на попередні версії;
- створювати відкладені зміни – зміни, які, безпосередньо, не додано до головного сховища, але їх можуть бачити інші користувачі, причому завантажити ці зміни можна, тільки отримавши спеціальний дозвіл від власника змін;
- позначати окремі версії файлів у репозитарії і групувати їх.

Для збереження даних і репозитаріїв розроблюваних проектів використовуються бази даних SQL Server 2005.

TFC – потужний і зручний інструмент, що дозволяє не тільки управляти версіями вихідного коду, але і повністю організовувати весь цикл розроблення проекту – від написання програм до їх документації. Однак, ця потужна і складна система більше підходить для ведення великих проектів, які потребують складного і досконалого управління розробкою. Якщо у Вас – невелика розробка, то має сенс використовувати менш потужний інструмент, а ще краще вільно розповсюджуваний, так як це Вам заощадить час, гроші й нерви.

Узагальнення

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 якими роботу ведуть тисячі людей, то, краще за все, вибір зупинити на Git або Mercurial. Якщо для Вас головне – зручний інтерфейс, а розробляється проект – не дуже великий, то для Вас найкращою системою є 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», оскільки зараз існує безліч систем контролю версій, які надають Вам усе необхідне. Отож вибирайте і не пошкодуєте. Системи контролю версій – це абсолютно необхідне програмне забезпечення для кожного розробника і не тільки.

1.2 Система управління версіями для Visual Studio

Не покидаючи середовища розроблення, можна легко управляти індивідуальними і командними проектами, використовуючи можливості системи управління версіями Microsoft Visual Studio. Система управління версіями Visual Studio дозволяє виконувати такі операції:

• Управління доступом до бази даних. Система управління версіями Visual Studio підтримує як загальний, так і винятковий доступ до файлів, а також механізми злиття файлів.

• Вилучення послідовних версій елементів з керуванням версіями. Більшість пакетів систем управління версіями, що розміщуються в Visual Studio, зберігає дані про відмінності між версіями елемента з керуванням версіями.

• Підтримка докладних відомостей журналу на елементах з керуванням версіями. Багато з пакетів системи управління версіями забезпечують механізми для збереження і вилучення журналу елемента, наприклад, створення дати і часу.

• Спільна робота над проектами і рішеннями. Можливо спільне використання файлів для декількох проектів і рішень для загального доступу до елементів з керуванням версіями. Зміни в елементі із загальним доступом відображаються у всіх проектах і рішеннях.

• Автоматизація часто повторюваних операцій системи управління версіями. Наприклад, пакет системи управління версіями, що розміщується в Visual Studio, може визначати інтерфейс командного рядка, що підтримує основні можливості системи управління версіями. Цей інтерфейс можна використовувати в пакетних файлах для автоматизації регулярно виконуваних завдань системи управління версіями.

• Відновлення після випадкових вилучень. Система управління версіями Visual Studio підтримує відновлення самої останньої повернутої версії файла.

• Економія дискового простору як в пакеті системи управління версіями, так і на відповідному сервері.

Система Visual Studio підтримує систему управління версіями, використовуючи рівень протоколу Visual Studio Integration Protocol (VSIP) на своєму інтегрованому рівні розроблення (Integrated Development Environment, IDE). Протокол VSIP може підтримувати безліч пакетів систем управління версіями, які зазвичай реалізуються як модулі, що написані для відповідних протоколів. Прикладом модуля управління версіями, який підключається, є модуль SourceSafe LAN, підтримуваний системою Visual SourceSafe. Докладні відомості про цей модуль можна прочитати у Довідці системи Visual SourceSafe.

Примітка. Система Visual Studio посилається на пакети систем управління версіями як на модулі, хоча вони можуть бути реалізовані у вигляді інших типів програмних модулів.

Система управління версіями Visual Studio є тільки середовищем для модулів систем управління версіями сторонніх постачальників. Тому її функціональні можливості активуються лише за допомогою установлення модуля. Щоб використовувати модуль системи управління версіями стороннього виробника, зазвичай необхідно встановити додаток (застосунок) стороннього виробника і / або плагіни системи управління версіями на клієнтському і серверному комп'ютерах веб-вузла. Після завершення їх установлення відповідно до інструкцій стороннього постачальника функціональність цих модулів буде доступна за допомогою системи Visual Studio. Доступні операції різняться залежно від модуля управління версіями, який підключається. Для отримання детальної інформації про роботу конкретного пакета слід ознайомитися з документацією стороннього постачальника.

Основна підтримка системи управління версіями в Visual Studio включає установку параметрів модуля управління версіями, який підключають, і параметрів середовища, перемикання модулів, доступ до бази даних, а також управління версіями і роботу з проектами, рішеннями і файлами Visual Studio і відповідними метаданими. Система управління версіями в Visual Studio також підтримує протоколи для управління доступом до бази даних, наприклад, робочу процедуру "Блокування-Зміна-Розблокування" (Lock-Modify-Unlock), при якій користувач, охочий змінити файл, повинен отримати його в монопольному режимі.

Важливо враховувати, що необхідно використовувати механізми системи управління версіями в Visual Studio, щоб взаємодіяти з тим модулем системи управління версіями, що підключається. Не використовуйте інші клієнтські програми сторонніх виробників, що поставляють Plug-in, такі, наприклад, як провідник Visual SourceSafe. Правильне використання механізмів системи управління версіями в Visual Studio гарантує додавання в систему управління версіями тільки належних файлів, а також виконання оновлення файлів проектів і рішень Visual Studio з урахуванням конкретних особливостей плагіна.

Конфігурація і перемикання модулів системи управління версіями

Система управління версіями Visual Studio підтримує конфігурацію і комутацію модулів за допомогою елемента «Система управління версіями в діалоговому вікні» -> «Параметри». Цей елемент доступний шляхом вибору команди «Параметри» в меню «Сервіс системи» Visual Studio. Діалогове вікно «Параметри» використовуватимуть для вибору додатків, які потрібно застосувати для системи управління версіями, а також для налаштування параметрів середовища для цього модуля.

Перед тим, як користувач і його команда зможуть скористатися перевагами функцій системи управління версіями в IDE системи Visual Studio, необхідно виконати такі дії:

• визначити, чи доступний будь-який з модулів системи управління версіями;

• якщо необхідний модуль системи управління версіями не встановлено на комп'ютері, встановіть продукт стороннього виробника, який підтримує модуль, і перезапустіть Visual Studio, щоб зареєструвати його;

• створити базу даних системи управління версіями відповідно до функціональних можливостей конкретного модуля;

• надіслати посилання на розташування бази даних усім членам команди.

Доступ до бази даних

Основні команди доступу до бази даних, наприклад, «Вилучити» і «Додати в систему управління версіями», доступні в меню «Файл» системи Visual Studio. Однак ці команди активуються тільки після вибору необхідного користувачеві модуля управління версіями, що підключають. Коли використовується одна з основних команд доступу до бази даних, вибраний модуль активує відповідні функціональні можливості та / або середу стороннього виробника для виконання певної операції.

Деякі операції доступу активні тільки при використанні вибраного модуля, тоді як інші операції доступні тільки тоді, коли також вибраний проект, рішення або файл Visual Studio в оглядачі рішень системи Visual Studio. Наприклад, коли модуль вибрано, можна використовувати команду «Додати в систему управління версіями». Однак, щоб використовувати команду «Повернути», необхідно мати елемент, вибраний в оглядачі рішень.

Оброблення файла системою управління версіями

У систему управління версіями Visual Studio можна додати такі файли:

• Файли рішень (* .sln).

• Файли проекту, наприклад, файли * .csproj і * .vbproj.

• Файли конфігурації додатків на базі XML, що використовуються для управління поведінкою проекту Visual Studio під час виконання.

До файлів, які не можна додати до системи управління версіями, відносяться такі:

• Файли для користувача параметрів рішення (* .suo).

• Файли для користувача параметрів проекту, наприклад, файли * .csproj.user і * .vbproj.user.

• Інформаційні веб-файли, наприклад, * .csproj.webinfo і * .vbproj.webinfo, які керують розташуванням віртуального кореневого каталогу веб-проекту.

• Вихідні файли проекту, наприклад, файли * .dll і * .exe.

Поширення змін простору імен

Система управління версіями Visual Studio підтримує поширення змін простору імен у модулях системи управління версіями, що підключаються. Поширення змін застосовується для операцій видалення, перейменування і переміщення. Якщо запитується операція, для якої дозволяється поширення зміни, модуль системи управління версіями змінює робочу копію елемента з керуванням версіями, основну копію в базі даних і копії всіх інших користувачів, коли відбувається повернення елемента, й інші користувачі отримують його.

Як система управління версіями виконує оброблення рішень і проектів

Коли рішення або проект додається до системи управління версіями, модуль системи управління версіями повинен спочатку ідентифікувати єдиний кореневий каталог для елемента, що додається. Цей кореневий каталог визначає шлях до батьківського каталогу для всіх робочих папок і файлів, що складають рішення або проект.

Єдиний кореневий каталог часто відповідає фізичному шляху до диска. Однак якщо рішення містить файли або проекти, що розміщуються на декількох дисках, відсутня фізична папка, на яку може бути відображений єдиний кореневий каталог. Рішення може перекривати диски, але це неможливо для єдиного кореневого каталогу системи управління версіями. Для вирішення цієї ситуації система управління версіями Visual Studio підтримує поняття супер-єдиного кореня. Цей тип кореня є віртуальним контейнером, у рамках якого розміщуються всі проекти і файли в рішенні з керуванням версіями.

Коли проводиться додавання рішення за допомогою підключуваного модуля управління версіями з розширеними можливостями, модуль створює в базі даних порожню кореневу папку рішення. Ця папка буде містити всі елементи в рішенні з керуванням версіями. За замовчуванням це папка <імя_решенія> .root.

Примітка. Коли в систему управління версіями додається одиничний проект, папка .root не створюється.

Використання кореня рішення забезпечує такі переваги:

• Зменшення числа запитів користувача. Корінь рішення мінімізує для вирішення можливе число прив'язок системи управління версіями і, таким чином, зводить до мінімуму число запитів користувачеві, коли проводиться додавання рішення до системи управління версіями і виконуються інші завдання.

• Інкапсуляція проекту. Корінь рішення гарантує, що всі проекти в рішенні можуть легко ідентифікуватися як відповідні один до одного, навіть у тому випадку, коли один або кілька проектів розміщуються в різних розділах або комп'ютерах.

Можливе відключення створення папки <ім’я_рішення> .root, але це не рекомендується робити. Додаткові відомості див. у розділі «Практичне керівництво. Скасування створення папки <ім’я_рішення> .root».

Рішення у Visual Studio є або рішеннями з правильним форматом, або рішеннями, в яких відсутній такий формат. Рішення з правильним форматом – це рішення, ієрархічна структура якого на диску відповідає його структурі в оглядачі рішень. Усі проекти рішення з правильним форматом зберігаються у вкладених папках папки рішення на диску. Якщо в систему управління версіями додається рішення з правильним форматом, модуль системи управління версіями створює нижче папку * .root, щоб зберігати для вирішення основні копії файла рішення (* .sln) і файлів для користувача параметрів рішення (* .suo). І нарешті, модуль системи управління версіями створює папку нижче папки .sln для кожного додаткового проекту в базі даних системи управління версіями.

Якщо рішення не має правильного формату, модуль системи управління версіями створює папку для вирішення і його початкового проекту. Потім у папці рішення створюються паралельно папки для кожного з додаткових проектів.

Подання рішення або проекту

Система Visual Studio забезпечує три відмінних подання для вирішення або проекту з управлінням версіями: конструювання, система управління версіями і фізичне подання (русс. «представление»). Багато із завдань системи управління версіями виконуються простіше, коли є однозначна відповідність між окремими елементами цих уявлень. Однак, якщо проводиться створення рішень і проектів та їх додавання в систему управління версіями за допомогою стандартних параметрів системи Visual Studio, ці рішення і проекти не будуть обов'язково організовані на диску таким же чином, як в оглядачі рішень і в базі даних.

Подання конструювання для вирішення або проекту, що є в оглядачі рішень, є логічним зображенням вмісту рішення або проекту. Подання конструювання є зазвичай впорядкованим і логічним. Непотрібні файли приховані, а файли з багатьох різних фізичних розташувань об'єднуються в одному контейнері проекту.

Подання системи управління версіями для вирішення або проекту, яке відображається в автономному додатку, такому як провідник Visual SourceSafe Explorer, також є логічним поданням про вирішення або проекту. Однак уявлення системи управління версіями не є обов'язковим для відображення логічного подання.

Фізичне представлення (подання) рішення або проекту, яке відображається в провіднику Windows чи відображає ієрархічну структуру логічного подання або подання системи управління версіями.

Наведені далі рекомендації можуть виявитися корисними для того, щоб домогтися точної організаційної відповідності між поданням

конструювання, поданням системи управління версіями і фізичним втіленням рішень і проектів з керуванням версіями:

• Створюйте порожні рішення, а потім додавайте до них проекти. Це допомагає підтримувати в пам'яті логічні зв'язки типу "батько-нащадок" між рішенням і його проектами. Коли згодом рішення додається до системи управління версіями, як уявлення системи управління версіями, так і уявлення проектування, будуть відображати ієрархію рішення на диску.

• Дайте кожному рішенню унікальну і описову назву, відмінну від імені кожного з проектів, що містяться в ньому.

• Уникайте додавання довідкових файлів у рішення або проект з керуванням версіями.

• Якщо це можливо, збережіть всі файли в рішенні або проект на одному диску.

Підключення і прив'язки системи управління версіями

Система Visual Studio визначає підключення як наявність реального зв'язку за даними між Visual Studio і сервером бази даних. Коли рішення або проект додається до системи управління версіями, модуль системи управління версіями копіює елементи і весь їхній вміст з диска в базу даних. Для кожної папки, що містить файл рішення або проекту, створюється одна папка системи управління версіями. Після додавання елемента модуль системи управління версіями пов'язує локальну робочу копію рішення або проекту з її версією в базі даних.

Кожне рішення з керуванням версіями має, принаймні, одну прив'язку системи управління версіями. Однак елемент може мати декілька прив'язок і вимагати наявності декількох підключень до бази даних. Кількість прив'язок і підключень залежить від того, як спочатку створюється рішення, і, чи зберігаються всі його проекти і файли в одному і тому ж розділі.

Як приклад прив'язок і підключень уявімо собі рішення з правильним форматом і з управлінням версіями, яке містить кілька проектів, у вигляді будинку з кількома кімнатами. При будівництві будинку можна встановити одну високошвидкісну лінію передачі даних з однієї кімнати на вулицю. Позаду брандмауера встановлюється маршрутизатор, щоб розподіляти подачу даних в інші кімнати. Через Інтернет-провайдера проводиться оплата за підключення будинку до Інтернету.

Можна уявити прив'язку системи управління версіями як використання однієї лінії передачі даних, створеної для будинку. Коли користувач відкриває рішення з керуванням версіями, створюється підключення через цю прив'язку. Підключення здійснює підтвердження встановлення зв'язку між робочою копією рішення на диску і основною копією рішення в базі даних.

Якщо рішення з керуванням версіями не має правильного формату, можна уявити його собі як будинок, в якому кожна кімната безпосередньо

підключена до Інтернету. Витрати на Інтернет істотно вище, ніж в будинку з одним підключенням, витрати при поточному обслуговуванні більше, а також більш складним і трудомістським є перемикання між різними постачальниками послуг Інтернету.

В ідеальному випадку рішення і його проекти спільно використовують одну прив'язку до системи управління версіями. Рішення з однією прив'язкою є більш керованими, ніж рішення з декількома прив'язками. Простіше виконуються такі дії:

• відключення від системи управління версіями, щоб працювати в автономному режимі;

• підключення до бази даних після повторного підключення до мережі;

• виконання розгалуження за один крок.

Можна створити рішення з декількома проектами з використанням однієї прив'язки шляхом створення порожнього рішення перед додаванням в нього проектів. Крім того, це можна зробити, вибравши параметр «Створити каталог для рішення» в діалоговому вікні «Створити проект», коли створюється пара "рішення-проект".

Якщо за один крок створюється пара "рішення-проект" і не вибрано «Створити каталог для рішення» в діалоговому вікні «Створити проект» (відключений за замовчуванням), буде створена друга прив'язка при додаванні до рішення другого проекту. Одна прив'язка створюється для початкового проекту та рішення. Додаткові прив'язки створюються для кожного додаткового проекту.

2 БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ ПРОЕКТУ

Безперервна інтеграція (Continuous Integration, CI) – це методика розроблення проекту, яка визначає процес розроблення ПЗ, як нерозривний ланцюжок дій, що повторюються щодо розроблення і впровадження [23].

Основа підходу спрямована на нівелювання проблем, пов'язаних з інтеграцією майбутнього програмного продукту на сервер збирання проектів. На рисунку 2.1 зображено процес безперервної інтеграції [24].

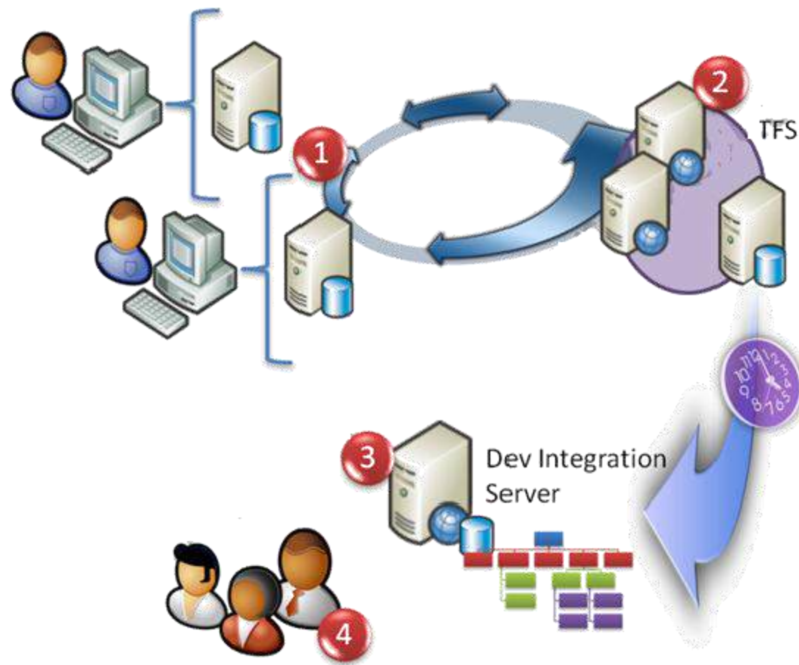


Рисунок 2.1 – Схема процесу безперервної інтеграції

Переваги CI:

• наявність робочої версії проекту в будь-який момент часу, який готовий до розгортання (deployment) на production сервер;

• негайний вплив зафіксованих змін на проект;

• автоматична публікація зібраного проекту, готового для тестування;

• автоматичне інкриментування версій збирань проекту.

Невід'ємною частиною continuous integration є приймальні тестування, unit-тестування, функціональне тестування в автоматичному режимі як метод контролю коректності роботи програми після змін.

Вимоги до проекту

Для здійснення безперервної інтеграції необхідне виконання таких вимог:

• Тексти програм і все, що необхідно для побудови та тестування проекту, зберігається в репозитарії системи управління версіями.

• Операції копіювання зі сховищ, збирання та тестування всього проекту автоматизовані та легко викликаються із зовнішньої програми.

ContinuousIntegration підходить для застосування в сімействі гнучких (Agile) методологій.

Організація процесу безперервної інтеграції

На виділеному сервері організується служба, до завдання якої входять:

1) отримання початкового коду зі сховищ;
2) збирання проекту;
3) виконання тестів;
4) розгортання готового проекту;
5) відправлення звітів.

Локальне збирання проекту може здійснюватися:

• за зовнішнім запитом;
• за розкладом;
• за фактом поновлення сховища та за іншими критеріями.

Основні принципи CI

Кожна зміна має інтегруватися

Слово "continuous" у термінології Continuous Integration означає «безперервний / безперервний». Це означає, що в ідеалі збирання Вашого проекту повинне відбуватись буквально весь час.

На практиці досить часто реалізують обидва процеси і безперервну інтеграцію і нічні збирання – рідкіснішу інтеграцію. У дуже великих проектах цієї вимоги іноді неможливо дотриматися, але інтеграція щодоби – це межа, за яку не варто виходити.

Швидке збирання проекту

«Збірка повинна йти швидко» – точніше, не більше 10 хвилин. Якщо після одного невеликого комміту Ваш інтеграційний сервер буде йти у двогодинне «сопіння» на збирання, тестування і розгортання проекту, від цього буде мало користі.

У разі, якщо всі етапи процесу ніяк не вдається втиснути у прийнятні часові рамки, можна розділити його на кілька частин. При кожному комміті виконувати лише саме збирання проекту і мінімальний набір тестів (smoke tests), щоб зменшити час. А ночами проводити повний цикл інтеграції, результати якого команда буде аналізувати з ранку.

Проведення автоматичного тестування

Тести необхідно включати в continuous integration-процес.

Основними двома обмежувачами на кількість тестів буде:

1) час *інтеграції* – збирання як і раніше має залишатися швидким, основне тестування можна перенести «на ніч»;

2*) доцільність тестів*.

Саме присутність тестів – одна з відмінностей інтеграції від натискання кнопки Build у Вашій IDE.

Інтеграція на виділеному сервері

Організовувати процес необхідно на спеціально виділеній машині. Така машина за своєю конфігурацією і набором прикладних програм

повинна максимально відповідати оточенню, в якому проект буде розгорнуто (production environment).

При цьому це не має бути машина розробника або когось ще, це повинна бути виділена машина (можна – віртуальна). Адже часто проект, зібраний на машині одного розробника, не можуть зібрати на машині іншого. Виділення машини для цілей інтеграції дозволяє зменшити ризик, пов'язаний зі зміною програмного і апаратного забезпечення.

Сервер збирань проекту

Сервер збирань TFBuild. Приклад (рисунок 2.2).

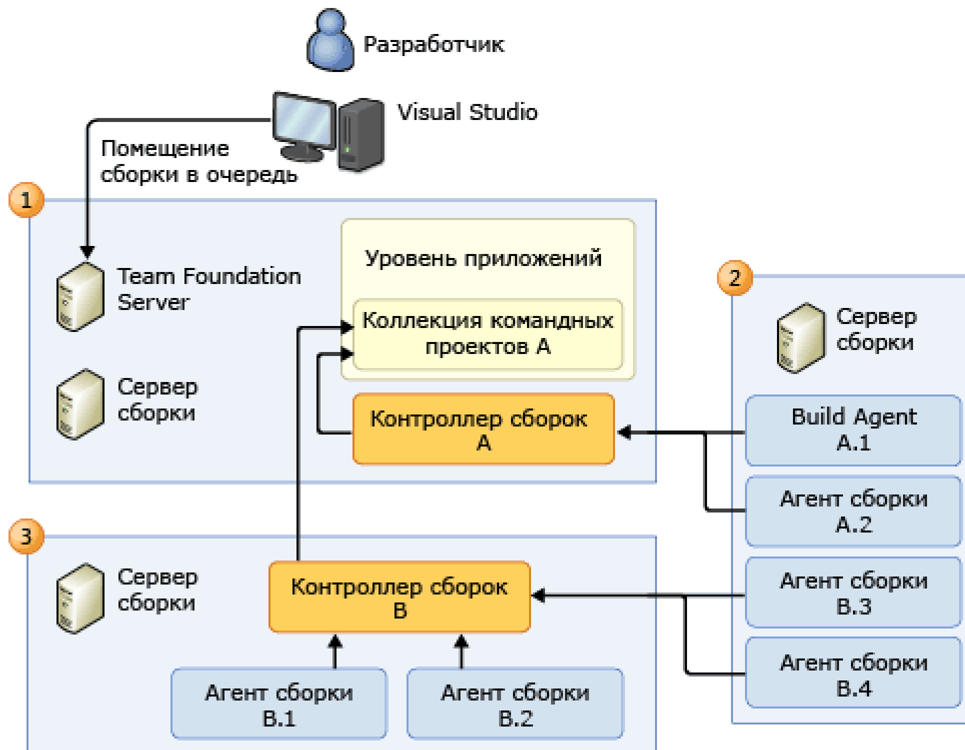


Рисунок 2.2 – Сервер збирання TFBuild

Для кожної колекції командних проектів виділяється окремий сервер збирання. Насправді, незважаючи на те, що налаштування і зміна сервера збирання, а також управління його роботою здійснюються безпосередньо на тому комп'ютері, де виконується *Служба побудови TeamFoundation*, дані конфігурації зберігаються в колекції командних проектів.

На сервері збирань можна запустити:

– один контролер побудови;

– один або кілька агентів побудови;

– один контролер побудови, а також один або кілька агентів побудови.

Розгортання сервера збирань

Сервер збирань розгортається при установці служби збирання TeamFoundation.

Можна підключити сервер TFBuild до свого локального сервера рівня додатків VisualStudioTeamFoundationServer.

Якщо виконується налаштування служби збирання, поки виконаний вхід у систему як члена групи *Адміністратори колекції проектів*, установка автоматично додає обліковий запис служби збирання у групу облікових записів служб збирання колекції проектів, тому немає необхідності робити це вручну.

Можна замінити існуючий сервер збирань, скопіювавши його конфігурацію на новий сервер збирання.

Можна налаштувати спеціальний сервер збирання на будь-якому клієнтському або серверному комп'ютері з відповідними ресурсами процесора і жорсткого диска. Наприклад, окремий розробник, у якого є ще один комп'ютер, може налаштувати його як сервер збирання.

Сервер збирань можна розгорнути на фізичному комп'ютері або на віртуальній машині.

Схема організації сервера інтеграції (рисунок 2.3) [25].

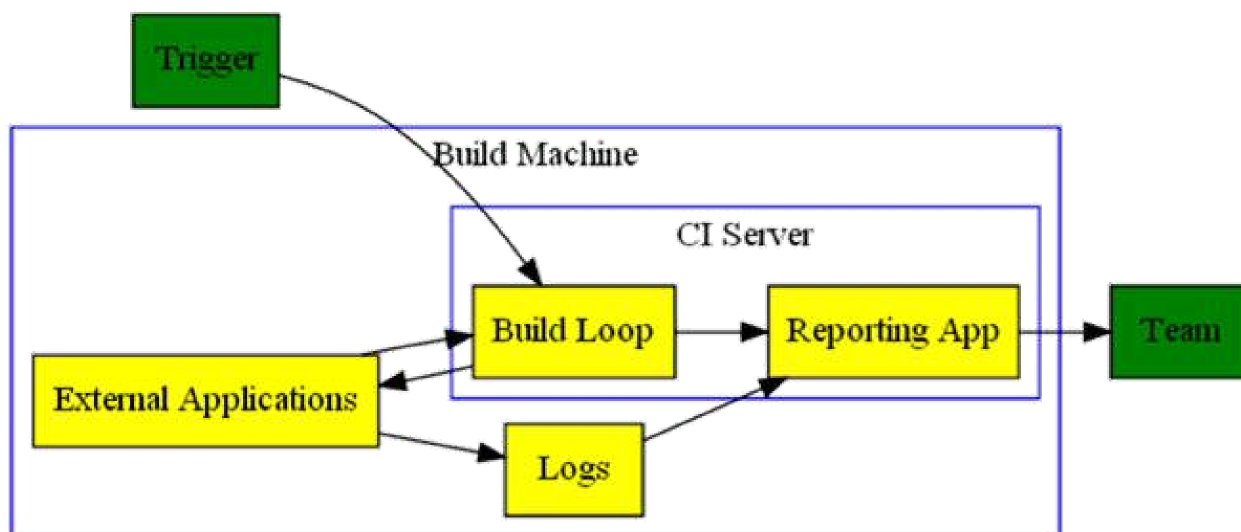


Рисунок 2.3 – Схема організації сервера інтеграції

Збирання проекту за розкладом

У разі збирання за розкладом (англ. Daily build – укр. «щоденне збирання») вони, як правило, проводяться кожної ночі в автоматичному режимі – нічні збирання (nightly build) (щоб до початку робочого дня були готові результати тестування).

Для розрізнення додатково вводиться система нумерації збирань – зазвичай, кожне збирання нумерується натуральним числом, яке збільшується з кожним новим збиранням.

Вихідні тексти та інші вихідні дані при взятті їх зі сховищ системи контролю версій позначаються номером збирання. Завдяки цьому, точно таке ж збирання може бути точно відтворене в майбутньому – досить взяти вихідні дані як потрібну мітку і запустити процес знову. Це дає можливість повторно випускати навіть дуже старі версії програми з невеликими виправленнями.

Переваги:

– проблеми інтеграції виявляються і виправляються швидко, що обходиться дешевше;

– негайний прогін модульних тестів для останніх редагувань;

– постійна наявність поточної стабільної версії разом з продуктами збирання – для тестування, демонстрації й т. п.

– негайний ефект від неповного або непрацюючого коду привчає розробників до роботи в ітеративному режимі з більш коротким циклом.

Недоліки:

– витрати на підтримку роботи безперервної інтеграції;

– потенційна необхідність у спеціальному навчальному сервері для потреби безперервної інтеграції;

– негайний ефект від неповного або непрацюючого коду відучує розробників від виконання періодичних резервних включень коду до репозитарію.

У разі використання системи управління версіями вихідного коду з підтримкою розгалуження, ця проблема може вирішуватися створенням окремої «гілки» (англ. Branch) проекту для внесення великих змін (код, розроблення якого до працездатного варіанта займе кілька днів, але бажано частіше резервне копіювання у репозитарій). Після закінчення розроблення та індивідуального тестування такої гілки вона може бути об'єднана (англ. Merge) з основним кодом або «стволом» (англ. Trunk) проекту.

Автоматизація збирання

Автоматизація збирання – етап написання скриптів або автоматизація широкого спектру завдань, що застосовується розробниками в їхній повсякденній діяльності [26].

Включає в себе такі дії, як:

- компіляція вихідного коду в бінарний код;
- збирання бінарного коду;
- виконання тестів;
- розгортання програми на виробничій платформі;
- написання супровідної документації або опис змін нової версії.

«Просунута» автоматизація збирання

«Просунута» автоматизація збирання дає можливість віддаленому користувачеві управляти **обробленням розподілених збирань** і / або **розподіленим обробленням збирань**.

Під терміном **«Розподілені збирання»** розуміють, що виклики компілятора і лінковщика можуть передаватися безлічі комп'ютерів для прискорення швидкості збірки. Цей термін часто плутають з терміном «розподілене оброблення».

Розподілене оброблення означає, що кожен етап процесу може бути адресований різним машинам для виконання ними цього кроку.

Розподілений процес збирання повинен мати певну логіку, щоб правильно визначити залежності у вихідному коді для того, щоб виконати етапи компіляції і компонування на різних машинах. Рішення автоматизації збирання має бути здатне **керувати** цими **залежностями**, щоб виконувати розподілені збирання.

Автоматизація збирання, яка здатна розсортовувати взаємозв'язки залежностей вихідного коду, також може бути налаштована на виконання дій компіляції і компонування в режимі паралельного виконання. Це означає, що компілятори і лінковщики можуть бути викликані в багатопотоковому режимі на машині, яка сконфігурована з урахуванням наявності більш одного процесорного ядра.

«Просунута» автоматизація збірки. Переваги:

- покращання якості продукту;
- прискорення процесу компіляції і компонування;
- позбавлення від зайвих дій;
- мінімізація «поганих (некоректних) збирань»;
- позбавлення від прив'язки до конкретної людини;
- ведення історії збирань і релізів для розбору випусків;
- економія часу і грошей завдяки причинам, зазначеним вище.

«Просунута» автоматизація збирань. Типи [27]:

Автоматизація за запитом (On-Demand automation): запуск користувачем скрипта в командному рядку.

Запланована автоматизація (Scheduled automation): безперервна інтеграція, яка відбувається у вигляді нічних збирань.

Умовна автоматизація (Triggered automation): безперервна інтеграція, яка виконує складання при кожному підтвердженні зміни коду (commit) у системі управління версіями.

Вимоги до систем збирання:

1. Часті або нічні збирання для своєчасного виявлення проблем.
2. Підтримка управління залежностями вихідного коду (Source Code Dependency Management).
3. Оброблення різницевого збирання.
4. Повідомлення при збігу вихідного коду (після складання) з наявними бінарними файлами.
5. Прискорення збирання.
6. Звіт про результати компіляції й компонування.

Додаткові вимоги:

1. Створення опису змін (release notes) та іншої супутньої документації (наприклад, керівництва).
2. Звіт про статус збирання.
3. Звіт про успішне / неуспішне проходження тестів.

4. Підсумовування доданих / змінених / віддалених особливостей у кожному новому збиранні.

Проекти у TeamCity

На рисунку 2.4 зображено проекти сервера безперервної інтеграції TeamCity.

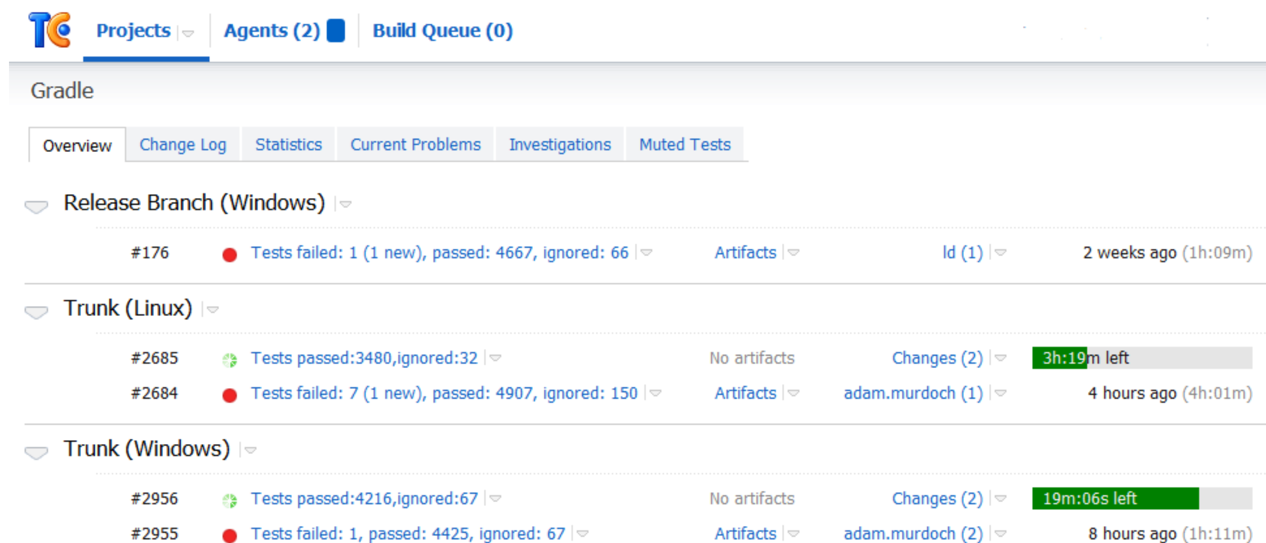


Рисунок 2.4 – Скріншот проектів у TeamCity

Етапи безперервної інтеграції:

Trigger – обов'язковий;
Update – обов'язковий;
Analyse – не обов'язковий;
Build – обов'язковий;
UnitTest – вкрай бажаний;
Deploy – потрібен для розслідування обставин;
Test – не обов'язковий, але вкрай бажаний;
Archive – бажаний;
Report – обов'язковий.

Trigger – Запуск збирання проекту

Цикл інтеграції починається зі спрацьовування тригера. Це може бути одна з таких подій:

- зміна в системі контролю версій;
- зміна у файловій системі;
- певний момент часу;
- збирання іншого проекту;
- натиснута «червона» кнопка;
- зміна на веб-сервері.

Характерним прикладом буде випадок, коли один із розробників робить комміт у систему контролю версій. Для інтеграційного сервера це означає, що у вихідному коді проекту відбулися зміни і необхідно провести збирання для перевірки того, що ці зміни нічого не зіпсували і узгоджуються з раніше зробленими.

Update – оновлення версії коду

На цьому етапі CI сервер робить update своєї локальної копії вихідного коду проекту. В процесі update з'ясовуються зміни в коді (і не тільки), які відбулися з моменту останньої інтеграції. З'ясування змін необхідно для того, щоб у разі збою можна було легко з'ясувати причину і знайти відповідального.

Лог змін TeamCity (рисунок 2.5).

Projects | Agents (2) | Build Queue (0) — Welcome, Guest user

Gradle

Overview | Change Log | Statistics | Current Problems | Investigations | Muted Tests

Show changes by: --- All users --- containing path: Filter Number of rows: 50

Found 5963 changes Show builds Permalink

20 Jan 12 20:11	Separated some new coverage from the DaemonLifecycleSpec into a separate spec.	szczepiq	2 files	jetbrains.git: 7581a9e7d3f1dd...
21 Jan 12 15:16	GRADLE-1815 - replaced use of deprecated method	ld	1 file	jetbrains.git: 42a979b3c08706...
21 Jan 12 06:58	- Changed LoopbackDependencyResolver.locate() to first resolve the module for the specified artifact, then download the artifact. - Removed ArtifactToFileResolver.	adam.murdoch	6 files	jetbrains.git: a8094477b5cd27...
21 Jan 12 06:08	- Changed ProjectDependencyResolver, ClientModuleResolver and UserResolverChain to implement ModuleVersionResolver.getArtifact(). - Changed ClientModuleResolver to first resolve the dependency, and then overlay the module descriptor later. - Added ClientModuleDependencyDescriptor, and use this instead of ClientModuleRegistry.	adam.murdoch	18 files	jetbrains.git: 40c45d63f865bc...
21 Jan 12 05:23	GRADLE-1815 - replaced use of deprecated method	ld	1 file	jetbrains.git: 7850e5c6a80b53...
21 Jan 12 05:21	GRADLE-1815 - integration test for using a closure as the file tree base dir	ld	1 file	jetbrains.git: 7b6f60bcdd5ff8...
21 Jan 12 05:13	GRADLE-2057 - intégration test for fileTree(Object, Closure)	ld	1 file	jetbrains.git: d853db1e8cfefd...
21 Jan 12 05:12	fix javadoc errors	ld	2 files	jetbrains.git: a0c19e57af45bc...
21 Jan 12 05:06	replace use of deprecated method in integration test	ld	1 file	jetbrains.git: ff6057f0ad0ce7...

Рисунок 2.5 – Скріншот логу змін у TeamCity

Analyse – попередній аналіз

Після того, як «свіжу» версію проекту витягнено з системи контролю версій, але збирання ще не розпочато, можна провести статичний аналіз коду. Існує безліч автоматичних засобів для різних мов програмування, що дозволяють провести такий аналіз. Зазвичай вимірюються такі характеристики коду:

- наявність типових помилок;
- статичні характеристики коду: складність, розмір, інше;

• відповідність прийнятим стандартам кодування.
Результати аналізу включають до звітів.

Build – збирання

Один з основних етапів процесу – це збирання проекту. Тут відбувається компіляція (трансляція) вихідних кодів у виконані файли або якийсь інший результат. Оскільки сервер інтеграції – це спеціально виділена машина зі строго визначеною конфігурацією, результат тільки цієї збірки можна вважати кінцевим. Більше ніяких дій під назвою «Проект збирається на моїй машині!».

UnitTest – модульні тести

Модульні тести з самого початку – автоматизовані, їх включення до процесу інтеграції вкрай бажано. Оскільки часто у розробників немає часу або бажання запускати такі тести до того, як зміни відправлені до системи контролю версій, додаткове їх виконання ніколи не буде зайвим. Додаткову інформацію можна отримати, вимірюючи покриття модульних тестів. Ця метрика допоможе краще контролювати якість продукту, що випускається.

Успішні тести (рисунок 2.6).

Gradle › Trunk (Linux) › ● #2684 (21 Jan 12 09:04)

Overview | Changes (1) | Tests | Build Log | Artifacts — « ● #2683 | All history | Last recorded build

Total test count: **5064** (7 failed, 150 ignored); total duration: **12h:17m** — Download all tests in CSV — Permalink

View: tests — containing: — with: any status — Filter — Show: 20 items

Status	Test	Duration	Order#
OK	IvyRemoteDependencyResolutionIntegrationTest. can resolve and cache dependencies from multiple HTTP Ivy repositories (org.gradle.integtests.resolve.ivy)	28s,103ms	5004
OK	ArchiveIntegrationTest. canCreateAnEmptyJar (org.gradle.integtests)	28s,023ms	3916
OK	IdeaProjectIntegrationTest. allows configuring the language level (org.gradle.plugins.ide.idea)	28s,007ms	3513
OK	CppLibPluginGoodBehaviourTest. plugin can build with empty project (org.gradle.plugins.cpp)	27s,988ms	3390
OK	UserGuideSamplesIntegrationTest. configurationHandlingAllFiles (org.gradle.integtests.samples)	27s,968ms	4746
OK	ArchiveIntegrationTest. canCreateAJarArchiveWithDefaultManifest (org.gradle.integtests)	27s,890ms	3927
OK	CommandLineIntegrationTest. checkDefaultGradleUserHome (org.gradle.integtests)	27s,889ms	3818
OK	ArchiveIntegrationTest. canCreateATgzArchive (org.gradle.integtests)	27s,744ms	3923
OK	CopyTaskIntegrationTest. emptyDirsAreCopiedByDefault (org.gradle.integtests)	27s,723ms	4147
OK	ArchiveIntegrationTest. canCreateATbzArchive (org.gradle.integtests)	27s,709ms	3925
OK	CopyTaskIntegrationTest. testCopyWithCopyspec (org.gradle.integtests)	27s,692ms	4145
OK	IdeaPluginGoodBehaviourTest. plugin does not force creation of build dir during configuration (org.gradle.plugins.ide.idea)	27s,677ms	3461
OK	CppExePluginGoodBehaviourTest. plugin can build with empty project (org.gradle.plugins.cpp)	27s,606ms	3389
OK	ArchiveIntegrationTest. metaInfSpecsAreIndependentOfOtherSpec (org.gradle.integtests)	27s,553ms	3929
OK	JavaConfigurabilityIntegrationTest. customized java args are reflected in the inputArguments and the build model [current -> current] (org.gradle.integtests.tooling.m8)	27s,512ms	4325
OK	IdeaIntegrationTest. allowsCustomOutputFolders (org.gradle.plugins.ide.idea)	27s,466ms	3503

Рисунок 2.6 – Скріншот успішних тестів у TeamCity

«Провалені» тести (рисунок 2.7).

Projects | Agents (2) | Build Queue (0) Welcome, Guest user

Gradle › Trunk (Linux) › #2684 (21 Jan 12 09:04)

Overview | Changes (1) | Tests | Build Log | Artifacts « #2683 | All history | Last recorded build

Total test count: 5064 (7 failed, 150 ignored); total duration: 12h:17m Download all tests in CSV Permalink

View: tests containing: with: failed status Filter [reset]

Found 7 matching tests (7 failed); duration: 1m:56s

Status	Test	Duration	Order#
Failure	ConcurrentToolingApiIntegrationTest. handles the same target gradle version concurrently [current -> current] (org.gradle.integtests.tooling.m8)	30s,961ms	4613
Failure	ConcurrentToolingApiIntegrationTest. handles different target gradle versions concurrently [current -> current] (org.gradle.integtests.tooling.m8)	30s,221ms	4614
Failure	UserGuideSamplesIntegrationTest. fileTrees (org.gradle.integtests.samples)	15s,195ms	4872
Failure	**ConsumingStandardInputIntegrationTest. consumes input when building model [current -> current]** (org.gradle.integtests.tooling.m8)	10s,124ms	4345
Failure	ConsumingStandardInputIntegrationTest. does not consume input when not explicitly provided [current -> current] (org.gradle.integtests.tooling.m8)	10s,081ms	4351
Failure	ConsumingStandardInputIntegrationTest. works well if the standard input configured with null [current -> current] (org.gradle.integtests.tooling.m8)	10s,079ms	4348
Failure	ConsumingStandardInputIntegrationTest. consumes input when running tasks [current -> current] (org.gradle.integtests.tooling.m8)	10s,055ms	4353

Рисунок 2.7 – Скріншот «провалених» тестів у TeamCity

Deploy – розгортання

У разі веб-додатку розгортання передбачає викладання на веб-сервер (сервер додатків) і запуск готового проекту. Для GUI-додатків – це (пере-) установка в системі або генерація інсталятора.

Етап розгортання повинен проходити якомога більш «чисто». При цьому для подальшого тестування часто необхідно привести додаток (застосунок) до певного «стандартного» стану:

1) «залити» дамп бази;
2) налаштувати в стандартному режимі;
3) прибрати «сліди» попередньої діяльності додатка.

3 ТЕСТУВАННЯ ЯК ЕТАП БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ

Тестування програмного забезпечення – процес дослідження, випробування програмного продукту, що має дві різні цілі [28]:

• продемонструвати розробникам і замовникам, що програма відповідає вимогам;

• виявити ситуації, в яких поведінка програми є неправильною, небажаною або такою, що не відповідає специфікації.

Тестування. Види

Існує кілька ознак, за якими прийнято проводити класифікацію видів тестування. Зазвичай виділяють такі [29]:

За об'єктом тестування:

• функціональне тестування;
• тестування продуктивності:
• тестування навантаження;
• стрес-тестування;
• тестування стабільності.
• конфігураційне тестування;
• юзабіліті-тестування;
• тестування інтерфейсу користувача;
• тестування безпеки;
• тестування локалізації;
• тестування сумісності.

За знанням системи (за доступом до системи):

• тестування «чорної скриньки»;
• тестування «білої скриньки»;
• тестування «сірої скриньки».

За ступенем автоматизації:

• ручне тестування;
• автоматизоване тестування;
• автоматичне тестування.

За ступенем ізольованості компонентів:

• модульне тестування;
• інтеграційне тестування;
• системне тестування;

За часом проведення тестування:

• альфа-тестування;
• димове тестування (англ. Smoke testing);
• тестування нової функції (new feature testing);
• підтверджувальне тестування;
• регресійне тестування;
• приймальне тестування;
• бета-тестування.

За ознакою позитивності сценаріїв:

• позитивне тестування;
• негативне тестування.

За ступенем підготовленості до тестування:

• тестування за документацією (формальне тестування);
• інтуїтивне тестування (англ. Ad hoc testing).

Покриття коду

Покриття коду – міра, яка використовується при тестуванні програмного забезпечення. Вона показує відсоток, наскільки вихідний код програми було протестовано.

Техніка покриття коду була однією з перших методик, винайдених для систематичного тестування програмного забезпечення.

Існує кілька різних способів вимірювання покриття, основні з них:

покриття операторів – кожний рядок вихідного коду було виконано і протестовано;

покриття умов – кожну точку рішення (обчислення щодо істинності або хибності виразу) було виконано і протестовано;

покриття шляхів – чи всі можливі шляхи через задану частину коду було виконано і протестовано;

покриття функцій – кожну функцію програми було виконано;

покриття вхід / вихід – чи були виконані виклики функцій і повернення з них;

покриття значень параметрів – чи були перевірені типові та граничні значення параметрів.

Тестовий випадок (англ.test case) в розробленні програмного забезпечення – це набір умов, при яких тестувальник визначатиме, чи задовольняється заздалегідь визначена вимога. Щоб визначити, чи вимога повністю виконується, може знадобитися багато варіантів тестування. Часто варіанти тестування групують у тестові набори.

Для того, щоб повністю протестувати, чи всі вимоги в додатку виконуються, необхідно, щоб було, *принаймні, два варіанти тестування для кожної вимоги* (якщо вимога не має додаткових вимог).

У ситуації, якщо вимога має додаткові вимоги, на кожну додаткову вимогу також має бути щонайменше два варіанти тестування.

Варіант тестування зазвичай *складається з однієї стадії, але іноді з послідовності кроків,* щоб перевірити поведінку / функціональність, особливості застосування. Зазвичай очікуваний результат або очікуваний вихід встановлено.

Додаткова інформація, яка може бути включена до варіанта тестування:

• унікальний ідентифікатор варіанта тестування;
• короткий опис варіанта тестування;
• стадія тесту або порядок виконання;
• вимоги;

- глибина тесту;
- категорія тесту;
- автор;
- кнопка-прапорець для вказівки, автоматизований тест.

Test-регресійне тестування

Після того, як додаток «розгорнуто», необхідно його протестувати. Тут мають на увазі автоматичні функціональні тести, інакше кажучи, на цьому етапі проводиться регресійне тестування.

Регресійне тестування – це вид тестування, спрямований на перевірку змін, зроблених у додатку або у навколишньому середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, веб-сервер або сервер додатка), для підтвердження того факту, що існуюча раніше функціональність працює, як і раніше.

Після проходження регресійних тестів можна вважати, що інтеграція пройшла успішно, і в проект не внесено правок, які можуть привести до його непрацездатності (тут все залежить від Вашого набору тестів, модульних і функціональних). В іншому випадку – інтеграція не є успішною – код містить помилки і потрібно його виправлення / доопрацювання.

Сем Канер, наприклад, описав три основних типи регресивного тестування [30]:

Регресія багів (Bug regression) – спроба довести, що помилка, яку виправляли, насправді не виправлена.

Регресія старих багів (Old bugs regression) – спроба довести, що нещодавня зміна коду або даних «зламала» виправлення старих помилок, тобто старі баги стали знову відтворюватися.

Регресія побічного ефекту (Side effect regression) – спроба довести, що нещодавня зміна коду або даних «зламала» інші частини, які розробляються.

Archive – збереження проекту

Після того, як досягнута максимальна впевненість в якості вихідного коду, необхідно зберегти його. Це можна зробити, наприклад, за допомогою міток у системі контролю версій. Також необхідно зберегти бінарні файли проекту. Вони можуть знадобитися, якщо потрібно буде відтворити помилку в конкретній версії і для ручного тестування.

Репозитарії артефактів (рисунок 2.8).

Рисунок 2.8 – Репозитарії артефактів у TeamCity

Репозитарії артефактів є виділеними сховищами результатів збирання, влаштованими так, щоб спростити пошук потрібного артефакту (за іменем, версією, типом артефакту). Вони дозволяють групі розробників користуватися результатами роботи один одного без необхідності мати копії вихідних кодів їхніх модулів.

Report – Етап генерації й публікації звітів

Звіти містять такі складові:

• причина збирання – наприклад, зміни в репозитарії;

• зміни у вихідних кодах – тут можливі два варіанти: зміни від останнього збирання або від останнього успішного збирання;

• звіти зі статичного аналізу коду – всі результати, які є;

• лог збирання;

• лог модульних тестів – які тести пройшли і, що важливіше, які не пройшли;

• лог регресійних тестів – аналогічно модульним тестам;

• статистика збирань проекту:

• загальне число вдалих / провальних збирань;

• розподіл вдалих / провальних збирань у часі;

• статистика результатів статичного аналізу коду;

• усі інші метрики, що використовуються і збираються в проекті.

Для правильної організації цього етапу важливо розуміти, кого і як необхідно сповіщати про результати інтеграції. Тут треба вибрати між двома крайнощами – сповіщати завжди або ніколи.

Зразкове рішення цього завдання буде таким:

• **розробники** – мінімум при збої інтеграції. Звичайно, можна оповіщати і завжди, це залежить від частості збирань.

• **тестувальники** – якщо вони входять до складу команди, то сповіщати тоді ж, коли і розробників, адже іноді помилки можуть бути і в тестах. Якщо практикується незалежне тестування, то взагалі не сповіщати їх або сповіщати по завершенні інтеграції.

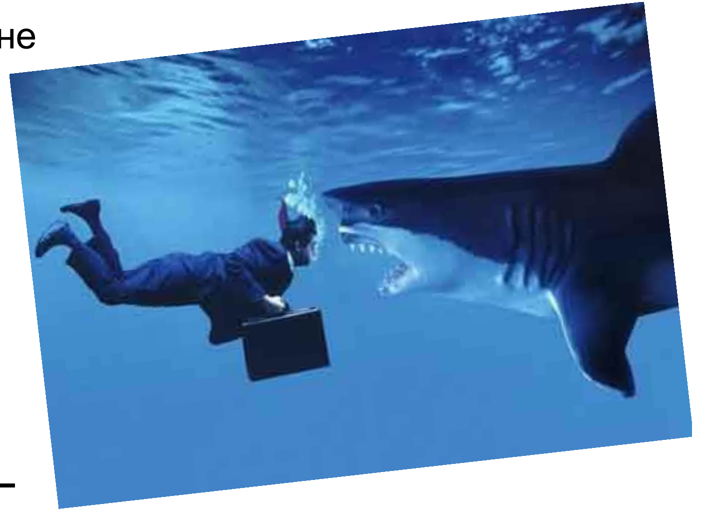
• **менеджер проекту** – суто за бажанням.

«Фатальні» етапи CI:

Build – проект неможливо зібрати.

UnitTest – модульні тести не пройшли або покриття впало нижче заданого рівня.

Test – регресивні тести не пройшли або покриття впало нижче заданого рівня.



Іноді до них приєднують етап **Analyse** – якщо в коді виявлено невідповідність стандартам кодування, то це є помилкою.

Continuous Improvement – безперервне вдосконалення

Після того, як налагоджений процес безперервної інтеграції, може здатися, що справа зроблена: сервер працює, «білди» збираються, пошта йде, і все добре, поки немає ніяких надзвичайних подій (наприклад, поломки сервера). Але це не так.

Сам процес потребує постійного налагодження, налаштування. Якщо спочатку у Вас не було ніяких тестів, то їх потрібно зробити.

Після – Ви захочете збирати інформацію про покриття Вашого застосування тестами, потім зміну такого покриття в часі, можливо, якісь ще специфічні метрики.

На рисунку 3.1 показано схему безперервного вдосконалення.

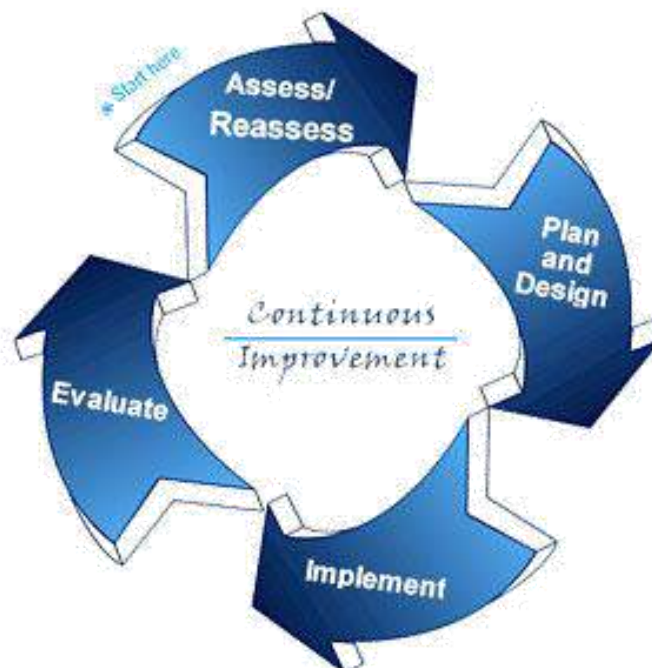


Рисунок 3.1 – Схема безперервного вдосконалення

4 UNIT-ТЕСТУВАННЯ. ПАТЕРНИ ВПРОВАДЖЕННЯ ЗАЛЕЖНОСТЕЙ КЛАСІВ

Що таке юніт-тести?

Модульні тести спочатку є автоматизованими, їх включення в процес інтеграції вкрай бажано. Оскільки часто у розробників немає часу або бажання запускати такі тести до того, як зміни відправлені в систему контролю версій, додаткове їх виконання ніколи не буде зайвим. Додаткову інформацію можна отримати, вимірюючи покриття модульних тестів. Ця метрика допоможе краще контролювати якість продукту, що випускається.

Які переваги дає використання юніт-тестів?

Використання юніт-тестів:

- *регресійне тестування;*
- *розроблення через тестування;*
- *рефакторинг;*
- *Мок-об'єкти.*

Типові відмовлення від використання Unit-тестування.

Інструментарій.

Висновок.

Тест-дизайн

Тест-дизайн – це етап процесу тестування ПО, на якому проектуються і створюються тестові випадки (тест-кейси), відповідно до визначених раніше критеріїв якості та цілей тестування.

Ролі, відповідальні за тест-дизайн:

Тест-аналітик – визначає "ЩО тестувати?"

Тест-дизайнер – визначає "ЯК тестувати?"

План робіт над тест-дизайном:

- аналіз наявних проектних артефактів: документація (специфікації, вимоги, плани), моделі, виконуваний код і т. д.;
- написання специфікації за тест-дизайном (Test Design Specification);
- проектування і створення тестових випадків (Test Cases).

Модульне тестування

Модульне тестування, або юніт-тестування (англ. Unit testing), – процес у програмуванні, що дозволяє перевіряти на коректність окремі модулі вихідного коду програми.

Модульні тести створюються для кожної нетривіальної функції або методу.

Такий підхід дозволяє досить швидко перевірити, чи не призвела чергова зміна коду до регресії, тобто до появи помилок у вже відтестованих місцях програми, а також полегшує виявлення і усунення таких помилок.

Ідея юніт-тестування

Модульні тести створюються для кожної нетривіальної функції або методу.

Мета – ізолювати окремі частини програми і показати, що окремо ці частини працездатні.

Тестування «білої скриньки».

Цей тип тестування зазвичай виконується програмістами.

Переваги юніт-тестування:

Заохочення змін.

Дозволяє програмістам проводити рефакторинг, надає впевненості, що модуль, як і раніше, працює коректно (регресійні тестування).

Це заохочує програмістів до зміни коду, оскільки досить легко перевірити, що код працює і без змін.

Спрощення інтеграції.

Допомагає усунути сумніви з приводу окремих модулів і може бути використано для підходу до тестування «знизу вгору»: спочатку тестуються окремі частини програми, потім програма в цілому.

Документування коду.

Юніт-тести можна розглядати як «живий документ» для класу, що тестується.

Клієнти, які не знають, як використовувати певний клас, можуть використовувати юніт-тест як приклад.

Відокремлення інтерфейсу від реалізації.

Оскільки деякі класи можуть використовувати інші класи, тестування окремого класу часто поширюється на пов'язані з ним класи.

Приклад.

Клас користується базою даних; в ході написання тесту програміст виявляє, що тесту доводиться взаємодіяти з базою. Це помилка, оскільки тест не повинен виходити за межі класу. В результаті розробник абстрагується від з'єднання з БД і реалізує цей інтерфейс, використовуючи свій власний mock-об'єкт. Це приводить до менш пов'язаного коду, мінімізуючи залежності в системі.

Переваги, окрім тестування.

Тести – це не інструмент, щоб правильно писати код. Тести - це середовище, в якому проходить тільки правильний код (або настільки правильний, наскільки точними є тести).

Тести впливають на дизайн Вашого коду.

Тести зменшують паніку.

У міру усунення кожної маленької проблеми загальна стабільність системи поліпшується.

Концепція модульних тестів

Unit testing (юніт-тестування, або модульне тестування) полягає в ізольованій перевірці кожного окремого елемента шляхом запуску тестів у штучному середовищі.

Для цього необхідно використовувати драйвери і заглушки. Оцінюючи кожен елемент ізольовано і підтверджуючи коректність його роботи, точно встановити проблему значно простіше, ніж якби елемент був частиною системи.

Елементи Unit-тестування:

Unit (Елемент) – найменший компонент, який можна скомпілювати.

Драйвери – модулі тестів, які запускають тестований елемент.

Заглушки – замінюють відсутні компоненти, які викликаються елементом.

Заглушка (stub)

Фіктивна підпрограма (клас, або окремий метод класу), що імітує одну або кілька функцій відсутнього модуля програмного виробу. Зазвичай має точку входу і точку виходу. Використовується, як правило, при низхідному тестуванні.

Функції заглушки:

- повертаються до елемента, не виконуючи ніяких інших дій;
- відображають трасувальні повідомлення та іноді пропонують тестувальнику продовжити тестування;
- повертають постійне значення або пропонують тестеру самому ввести значення, що повертається;
- здійснюють спрощену реалізацію компоненти, якої не вистачає;
- імітують виняткові або аварійні умови.

Ізоляція в тестуванні

Ідея ізоляції є однією з центральних у модульному тестуванні.

Ізоляція дозволяє тестувати класи, що розробляються, в момент, коли інші підсистеми ще не створені, коли деякі сервіси, які будуть використовуватися в продукційному середовищі, ще не доступні й т. д.

Мок-об'єкти

Для ізолювання використовують патерн Mock_Object (у вигляді заглушок, створених вручну, або таких, що генеруються автоматично).

Мок-об'єкти передаються в тестований код і повністю емулюють роботу іншого класу, який мав би використовуватися в тестованому коді під час реального використання.

Впровадження Мок-об'єктів у код

Базові принципи, на яких грунтується впровадження мок-об'єктів у тестовий код, – це інверсія залежностей (Dependency Inversion) або впровадження залежностей (Dependency Injection) [31, 32].

Загальна суть застосування принципів інверсії залежностей зводиться до таких дій: взаємодії всередині системи починаються створюватись на основі інтерфейсів, а не класів. Класи якби відмовляються від знань, кому саме вони делегують обов'язки. Для них стає важливим лише те, щоб делеговані «вміли» щось робити, хто вони саме такі при цьому – неважливо.

Інверсія управління (Inversion of Control, IoC) – принцип ООП, що використовується для зменшення пов'язаності в комп'ютерних програмах [32].

Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. І ті, й інші повинні залежати від абстракції.

Абстракції не мають залежати від деталей. Деталі повинні залежати від абстракцій.

Техніки реалізації Інверсії управління:

- фабричний метод (англ. Factory pattern);
- service locator (англ. Service locator pattern);
- впровадження залежності (англ. Dependency injection):
- через метод класу (англ. Setter injection);
- через конструктор (англ. Constructor injection);
- через інтерфейс впровадження (англ. Interface injection).

Фабричний метод

Фабричний метод (англ. Factory Method) – породжує шаблон проектування, надає підкласам інтерфейс для створення екземплярів деякого класу. У момент створення спадкоємці можуть визначити, який клас створювати [33].

Іншими словами, Фабрика делегує створення об'єктів спадкоємцям батьківського класу. Це дозволяє використовувати в коді програми неспецифічні класи, а маніпулювати абстрактними об'єктами на більш високому рівні.

```
using System;
using System.Collections.Generic;

namespace Factory
{
   abstract class Product {
      abstract public string GetType();
   }
    class ConcreteProductA: Product {
      public string GetType(){ return "ConcreteProductA"; }
   }
```

```
   class ConcreteProductB: Product {
      public string GetType(){ return "ConcreteProductB"; }
   }

   abstract class Creator{
      public abstract Product FactoryMethod();
   }

   class ConcreteCreatorA: Creator{
      public override Product FactoryMethod(){ return new
ConcreteProductA(); }
   }

   class ConcreteCreatorB: Creator{
      public override Product FactoryMethod(){ return new
ConcreteProductB(); }
   }
}

public class MainApp
   {
      public static void Main()
      {
        // an array of creators
        Creator[] creators =
 {new ConcreteCreatorA(), new ConcreteCreatorB()};
        // iterate over creators and create products
        foreach (Creator creator in creators)
   {
           Product product = creator.FactoryMethod();
           Console.WriteLine("Created {0}", product.GetType());
        }
        // Wait for user
        Console.Read();
      }
   }
```

Service locator. Патерн Service locator є сховищем сервісних об'єктів. Фактично це деякого роду асоціативний масив з екземплярами об'єктів-реалізацій, які визначаються за ключем, – типу або імені інтерфейсу [34].

```
public interface IServiceLocator
{
   T GetService<T>();
```

```
}
class ServiceLocator : IServiceLocator
{
   // словник відповідності типів і реализації
   private IDictionary<object, object> services;

   internal ServiceLocator()
   {
       services = new Dictionary<object, object>();
       // заповнюємо словник (заповнення словника можна здійснювати
      //за першим викликом GetService() ➔ LazyInit)
       this.services.Add(typeof(IServiceA), new ServiceA());
       this.services.Add(typeof(IServiceB), new ServiceB());
       this.services.Add(typeof(IServiceC), new ServiceC());
   }
}
```

Використання (передбачається, що ServiceLocator реалізує шаблон Singleton):

```
IServiceLocator locator = new ServiceLocator();
IServiceA myServiceA = locator.GetService<IServiceA>();

public T GetService<T>()
   {
      try
      {
         return (T)services[typeof(T)];
      }
      catch (KeyNotFoundException)
      {
         throw new OutOfRangeException
   («Невідомий тип класу");
      }
   }
```

Передача залежного об'єкта через конструктор
Constructor Injection

«Ін'єкція за допомогою конструктора» використовує конструктор для асоціювання об'єкта з конкретними реалізаціями абстракцій. При використанні цього типу інверсії залежностей необхідні об'єкти передаються до конструктора як аргументи.

Одна з переваг Constructor injection полягає в тому, що всі залежності класу прописуються явно. Тобто, поглянувши на один лише конструктор класу, ми вже розуміємо, якими ресурсами решти системи він користується.

Передача залежного об'єкта через set-метод
Setter Injection

Ін'єкція за допомогою set-методу вимагає визначення окремого set-методу для кожного з об'єктів, що підлягають ін'єкції. Від попереднього типу ін'єкції вона відрізняється місцем ін'єктування.

Передача залежного об'єкта через інтерфейс
Interface Injection

Interface injection використовує інтерфейси для зв'язування об'єктів.

По-перше, задаються інтерфейси, які визначають методи для зв'язування. Один інтерфейс на кожну залежність.

```
interface injectReader
{
 void  injectReader(IReader obj);
}

interface injectWriter
{
   void  injectWriter(IWriter obj);
}
```

Залежний об'єкт повинен реалізовувати всі ці інтерфейси.

```
class Copier:  injectReader, injectWriter...

  public void injectReader(IReader obj)
  {
      reader = obj;
  }

  public void  injectWriter(IWriter obj)
  {
      writer = obj;
  }
}
```

Визначається також єдиний інтерфейс для всіх сервісів:

```
interface Injector
{
  public void inject(object obj);
}
```

Кожен сервіс реалізує цей інтерфейс таким чином, щоб впровадити себе в залежний об'єкт:

```
class keyboardReader: Injector...
  public void inject(object obj)
  {
    if ( obj is injectReader )
    {
      throw new typeException(obj);
    }
    obj.injectReader(this);
  }
}
```

Таким чином, сервіси самі впроваджують себе в залежний об'єкт за допомогою встановленого інтерфейсу:

```
reader = new keyboardReader();
writer = new stdoutWriter();
copier = new copier();
reader->inject(copier);
writer->inject(copier);
```

5 UNIT-ТЕСТУВАННЯ У VISUALSTUDIO. АНАЛІЗ ПОКРИТТЯ КОДУ

Вимоги до тестованого проекту:

- одиниця, що підлягає тестуванню (проект – бібліотека класів);
- зв'язки між тестованими класами можуть бути розірвані й підмінені;
- тестовані класи реалізують інтерфейси.

Устрій тестів

Модульний тест являє собою клас, що міститься в окремому тестовому проекті спеціального типу і позначений атрибутом [TestClass].

Всі методи, які є тестами, позначаються атрибутом [TestMethod].

Також є атрибути методів для ініціалізації [TestInitialize] і очищення [TestCleanup] тестового оточення.

TestClassAttribute

- TestClassAttribute: під час створення модульного тесту TestClassAttribute включається в тестовий файл, щоб показати, що цей конкретний клас може включати методи, помічені атрибутом [TestMethod ()]. Без атрибута TestClassAttribute методи тесту пропускаються.
- Тестовий клас може успадковувати методи від іншого тестового класу, що входить в ту ж збірку. Це означає, що можна створювати методи тесту в базовому тестовому класі, а потім використовувати їх у похідних тестових класах.

Структура тестового методу (рисунок 5.1).

```
[TestMethod()]
public void CalculatorTest()
{
    ICalc target = new Calc();
    int actual;

    actual = target .Sum(2,3);

    int expected = 5;

    Assert.AreEqual(expected, actual);
}
```

Инициализация тестов

Вызов тестируемых методов

Проверка результата

Рисунок 5.1 – Структура тестового методу

Додавання бібліотеки класів до Solution

В Solution Explorer вибрати пункт меню Add-> New Project… -> Windows -> ClassLibrary (рисунки 5.2 – 5.3).

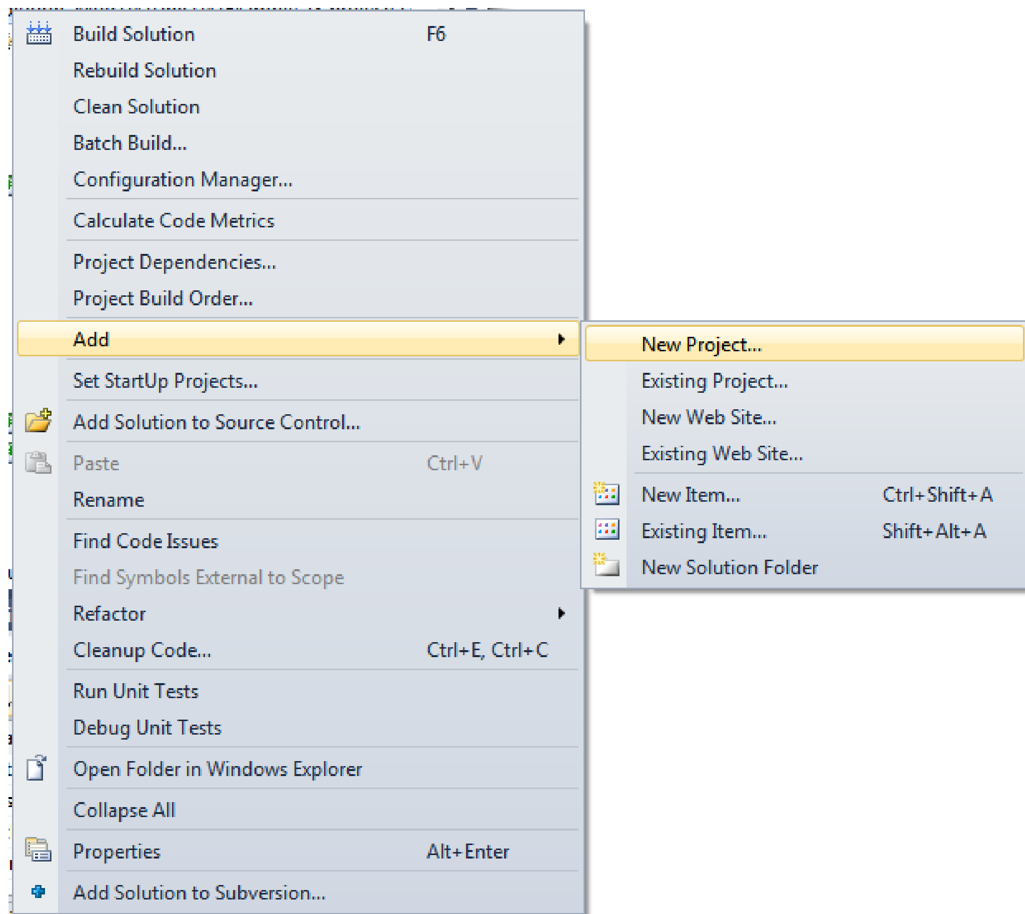


Рисунок 5.2 – Скріншот додавання нового проекту до рішення (Solution)

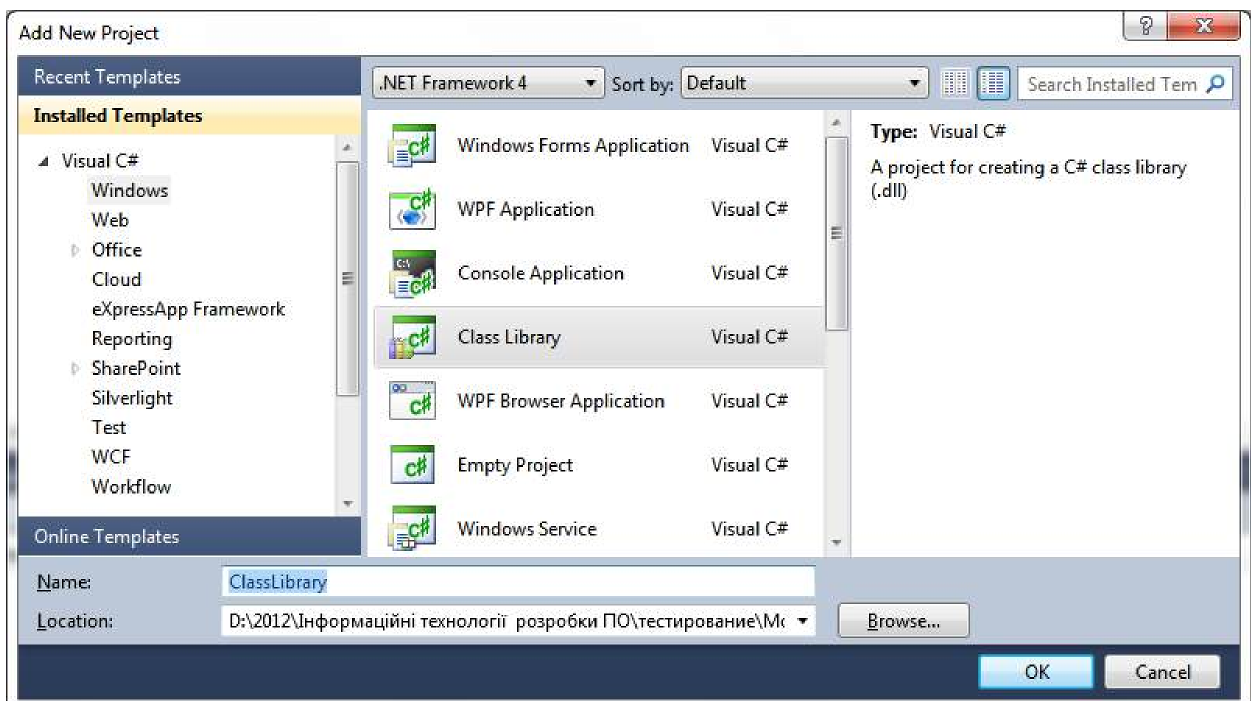


Рисунок 5.3 – Скріншот вибору типу проекту

Клас *MailWrapper*

Зберігає інформацію про повідомлення, що відправляється, і не несе ніякої бізнес-логіки. => Особливого тестування не потребує.

```
public class MailWrapper
   {
      public MailWrapper(string from, string to, string login, string psw, string message)
      {
         From = from;
         To = to;
         Login = login;
         Psw = psw;
         Message = message;
      }
      public string From { get; private set; }
      public string To { get; private set; }
      public string Login { get; private set; }
      public string Psw { get; private set; }
      public string Message { get; private set; }
   }
```

Для ефективної підміни використовуємо патерн впровадження залежностей через конструктор – Constructor Injection.

Для цього оголосимо інтерфейс класу, від якого спостерігається залежність:

```
public interface IMailHelper
   {
      bool SendMail (MailWrapper mailWrapper);
   }
```

Цей інтерфейс буде реалізований класом MailHelper і використаний як оголошення типу об'єкта, який приймає конструктор класу MailSpammer.

Метод SendMail отримує об'єкт типу MailWrapper (обгортка листа) і відправляє цей лист адресату.

Опис класу MailHelper:

```
public class MailHelper: IMailHelper
   {
      public bool SendMail (MailWrapper mailWrapper)
      {
         MessageBox.Show (string.Format ( "Повідомлення надіслано одержувачу {0}: {1}", mailWrapper.To, mailWrapper.Message));
         return true;
      }
   }
```

Цей клас реалізує інтерфейс IMailHelper і здійснює відправлення повідомлення з мережного протоколу.

Для спрощення прикладу відправка повідомлення з мережного протоколу замінена повідомленням *MessageBox.Show*.

Опис класу *MailSpammer:*

```
public class MailSpammer
  {
     IMailHelper _mailHelper;
  public MailSpammer(IMailHelper mailHelper)
     {
        _mailHelper = mailHelper;
     }
     public int SendManyLetters(List<MailWrapper> letters)
     {
        int countSend=0;
        foreach (var letter in letters)
           if (_mailHelper.SendMail(letter))
              countSend++;
        return countSend;
     }
  }
```

Клас MailSpammer у конструкторі приймає посилання на клас, який реалізує інтерфейс IMailHelper.

А метод SendManyLetters здійснює розсилку отриманого в параметрах списку листів і підрахунок успішно відправлених листів.

На рисунку 5.4 зображено діаграму взаємодії об'єктів класів MailSpammer і MailHelper.

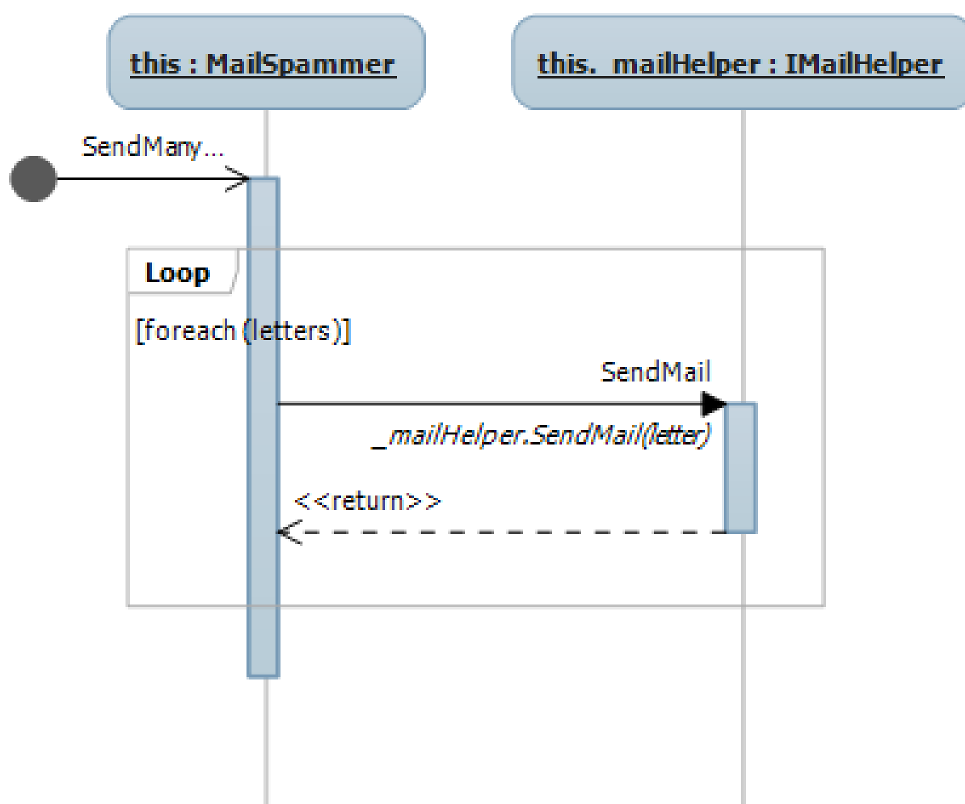


Рисунок 5.4 – Діаграма взаємодії об'єктів класів MailSpammer і MailHelper

Додавання Unit-test

В Solution Explorer вибрати пункт меню Add-> New Project… -> Windows -> Test Project.

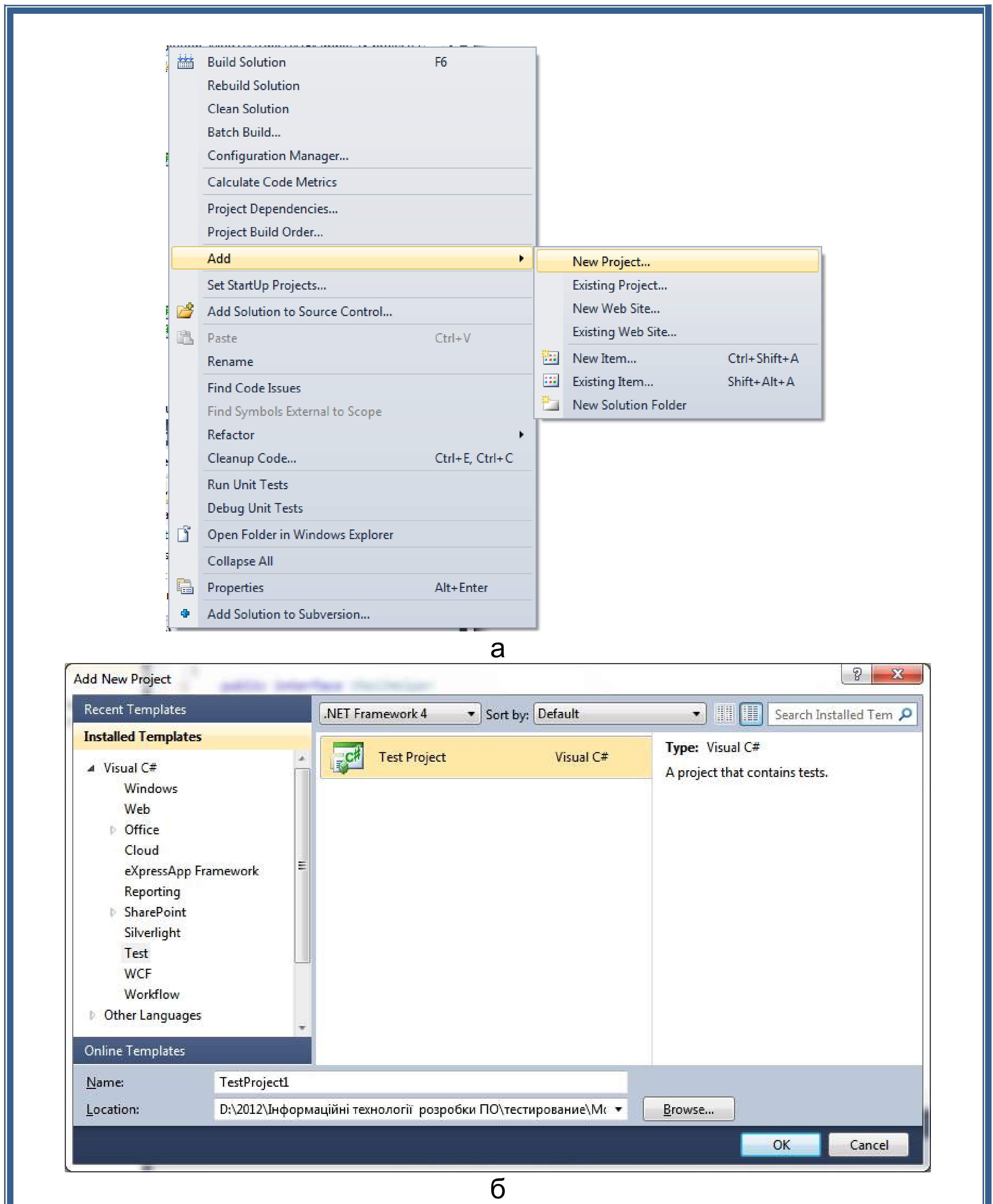


Рисунок 5.5 – Скріншот додавання тестового проекту:
а – додавання нового проекту до рішення (Solution);
б – вибір типу проекту

Додавання Unit-test альтернативним способом

Необхідно виконати таку послідовність дій:

1. Відкриваємо клас, на який необхідно згенерувати модульні тести.
2. Правою кнопкою миші тиснемо на імені класу і в контекстному меню вибираємо пункт Create Unit Tests ... (рисунок 5.6).
3. Вказуємо методи, на які необхідно згенерувати тести (рисунок 5.7).

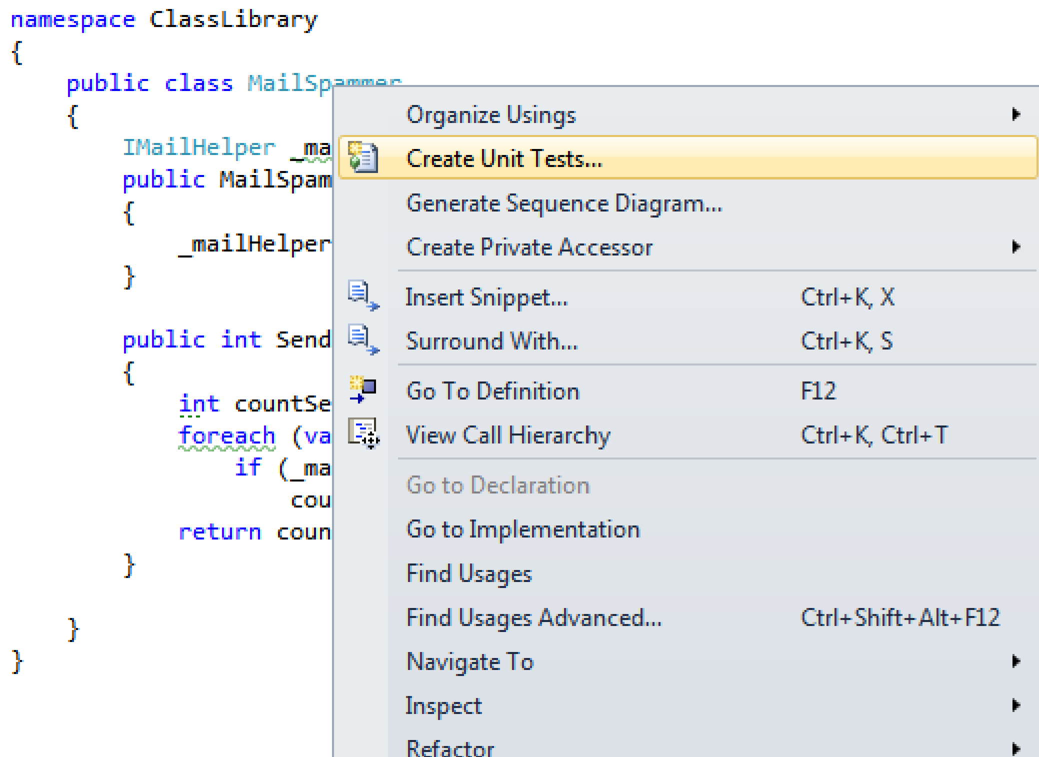


Рисунок 5.6 – Контекстне меню створення модульних тестів

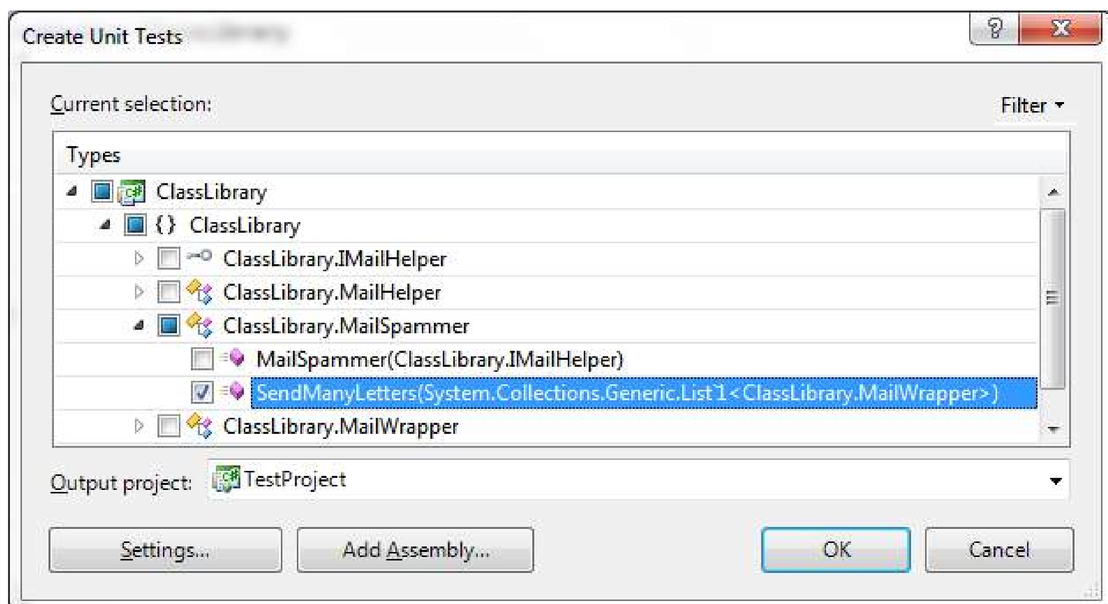


Рисунок 5.7 – Вікно вибору методів, на які необхідно згенерувати тести

Структура Solution (рішення) (рисунок 5.8).

На цьому етапі ми маємо Solution, до складу якого входять такі проекти:

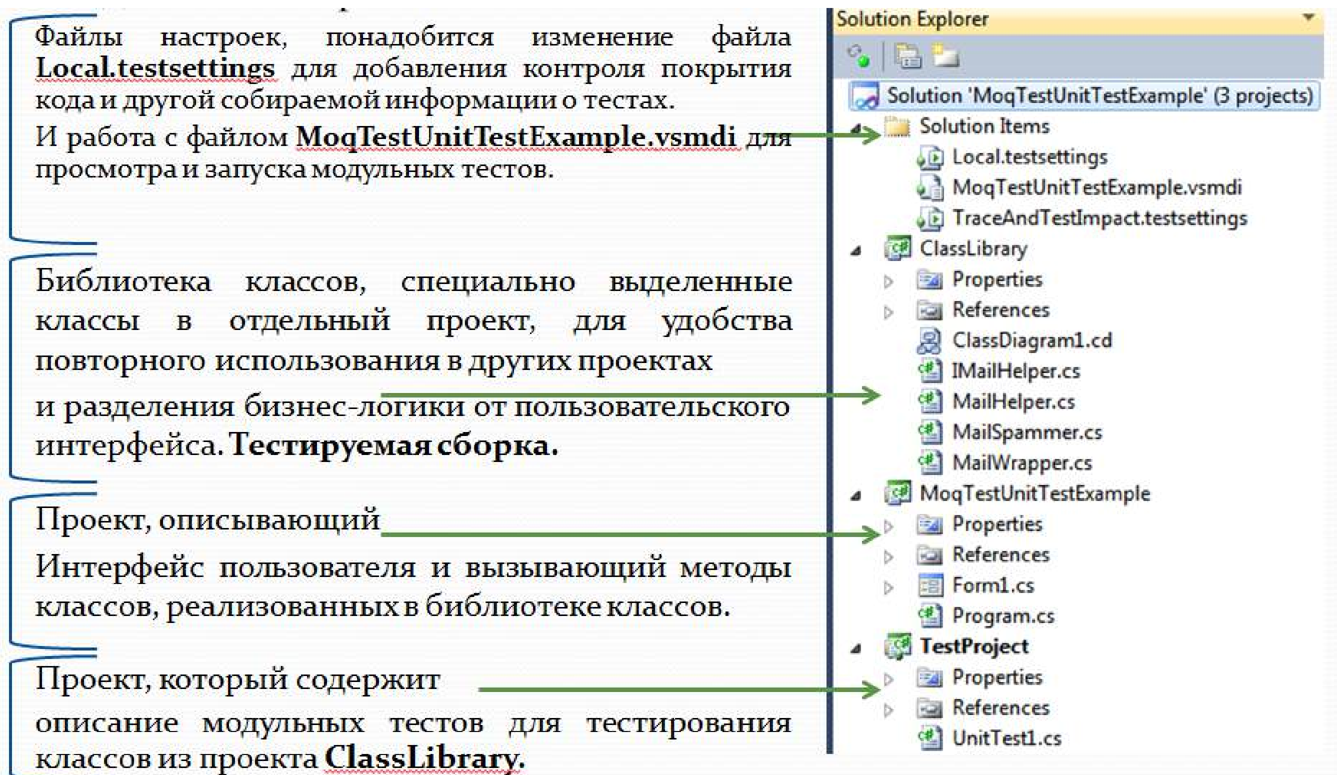


Рисунок 5.8 – Структура Solution (рішення)

Unit-test project за замовчуванням:

```
namespace TestProject
{
   [TestClass()]
   public class MailSpammerTest
   {
      private TestContext testContextInstance;
      [TestMethod()]
      public void SendManyLettersTest()
      {
         IMailHelper mailHelper = null; // TODO: Initialize to an appropriate value
         MailSpammer target = new MailSpammer(mailHelper); // TODO: Initialize to an appropriate
         List<MailWrapper> letters = null; // TODO: Initialize to an appropriate value
         int expected = 0; // TODO: Initialize to an appropriate value
         int actual;
         actual = target.SendManyLetters(letters);
         Assert.AreEqual(expected, actual);
         Assert.Inconclusive("Verify the correctness of this test method.");
      }
   }}
```

Автосгенерований код, зайве видалено.

Засоби для перевірки умов Assert [36]

Класи **Assert** простору імен UnitTestingFramework служать для перевірки певних функціональних можливостей.

Метод модульного тесту використовує код методу в коді розробки, але видає результати відносно поведінки коду тільки в тому випадку, якщо включені оператори Assert.

Принцип роботи Assert

Якщо порівнювані дані (числа, рядки, колекції, об'єкти) виявилися не еквівалентними, то сімейство класів Assert генерує виняток **AssertFailedException**, який відловлюється середовищем запуску модульних тестів, і тест позначається як не пройдений.

При порівнянні можна вказувати коментар, який потрапить до винятку і до результату тестів.

Оброблення користувальницьких винятків

Якщо метод, який тестується, в результаті виконання тесту повинен згенерувати очікуваний і призначений для користувача виняток, тестовий метод також позначається атрибутами ExpectedException із зазначенням типу очікуваного винятку.

[TestMethod] [ExpectedException (typeof (SomeException), "Коментар до проваленого тесту")]

Види класів Assert:

Assert

У методі тесту можна викликати будь-яке число методів класу Assert, таких як Assert.AreEqual (). Клас Assert містить багато методів для вибору, і багато з цих методів мають кілька перевантажень.

CollectionAssert

Клас CollectionAssert призначений для порівняння колекцій об'єктів і перевірки стану однієї або декількох колекцій.

StringAssert

Клас StringAssert служить для порівняння рядків. Цей клас містить різні корисні методи, такі як StringAssert.Contains, StringAssert.Matches і StringAssert.StartsWith.

AssertFailedException

Виняток AssertFailedException виникає у разі невиконання тесту. Причиною неможливості виконання тесту може бути витікання часу очікування, непередбачене виключення або оператор Assert, що створює результат "Помилка".

AssertInconclusiveException

Виняток AssertInconclusiveException виникає при кожному результаті тесту з невизначеним результатом. Як правило, оператор Assert.Inconclusive додається до тесту, над яким ще ведеться робота, для позначення його неготовності до виконання.

Assert.AreEqual (Of T) метод (T, T, String)

Перевіряє, що два зазначених елемента даних універсального типу дорівнюють один одному, використовуючи оператор рівності. Твердження не виконується, якщо вони не дорівнюють один одному. Якщо твердження не виконується, виводить повідомлення, зазначене в останньому параметрі. Також можливі такі параметри, що визначають дельту (похибку) порівняння – порівняння із зазначеною точністю, і режими порівняння рядків – з урахуванням регістра і т. п.

Параметри

Expected Тип: T.

Перший елемент даних універсального типу для порівняння – це дані універсального типу, які очікуються процесом модульного тесту.

Actual Тип: T.

Другий елемент даних універсального типу для порівняння – це дані універсального типу, які створює процес модульного тесту.

Message Тип: System.String.

Повідомлення, що відображається в разі, якщо твердження не виконується. Це повідомлення можна переглянути в результатах модульного тесту.

Assert.AreEqual

```
[TestMethod ()]
public void CalculatorTest ()
{
ICalc target = new Calc ();
int actual;
actual = target .Sum (2,3);
int expected = 5;
Assert.AreEqual (expected, actual);
}
```

StringAssert.AreEqual

Contains (String, String, String)

Перевіряє, чи містить перший рядок другий рядок. Якщо твердження не виконується, виводить повідомлення.

DoesNotMatch (String, Regex, String)

Matches (String, Regex, String)

Перевіряє, що зазначений рядок не відповідає (або відповідає) регулярному виразу. Якщо твердження не виконується, виводить повідомлення в заданому форматі.

EndsWith (String, String, String)

StartsWith (String, String, String)

Перевіряє, закінчується / чи починається перший рядок другим рядком. Якщо твердження не виконується, виводить повідомлення. Цей метод враховує регістр.

CollectionAssert.AreEqual

AreEqual (ICollection, ICollection, String)

Перевіряє дві зазначені колекції на рівність.

Твердження не виконується, якщо колекції не дорівнюють одна одній.

AreEquivalent (ICollection, ICollection, String)

Перевіряє дві зазначені колекції на еквівалентність.

Твердження не виконується, якщо колекції не еквівалентні.

AllItemsAreInstancesOfType (ICollection, Type, String)

Перевіряє, чи є всі елементи в зазначеній колекції екземплярами заданого типу.

Твердження не виконується, якщо в колекції існує хоча б один елемент, в ієрархії успадкування якого зазначений тип не виявляється.

AllItemsAreUnique (ICollection, String)

Перевіряє, чи всі елементи в зазначеній колекції унікальні.

Твердження не виконується, якщо будь-які два елементи в колекції дорівнюють один одному.

Contains (ICollection, Object, String)

Перевіряє, чи містить зазначена колекція заданий елемент.

Твердження не виконується, якщо заданий елемент в цій колекції не виявляється.

IsSubsetOf (ICollection, ICollection, String)

Перевіряє, чи є перша колекція підмножиною другої колекції.

Модифікований тестовий метод:

```
[TestMethod ()]
      public void SendManyLettersTest ()
      {
         IMailHelper mailHelper = new MailHelper ();
         MailSpammer target = new MailSpammer (mailHelper);
         const int count = 5;
         var letters = new List <MailWrapper> ();
         int expected = 0;
         var rand = new Random (123456);
         for (int i = 0; i <count; i ++)
         {
// Генерація випадкового листа, можна довантажувати з БД
            MailWrapper mailWrapper = new MailWrapper ( "user" + i, "user" + i * 2, "name" + i, "psw" + rand.Next (), "Лист" + i);
            letters.Add (mailWrapper);
            expected ++;
         }
         var actual = target.SendManyLetters (letters);
         Assert.AreEqual (expected, actual, "Кількість відправлених листів не збігається");
      }
```

Ініціалізація і очищення тестового оточення:

- **[AssemblyInitialize ()]** виконання коду до виконання першого тесту в збірці проекту.
- **[AssemblyCleanUp ()]** виконання коду після виконання останнього тесту в збірці проекту.
- **[ClassInitialize ()]** виконання коду до виконання першого тесту в класі.
- **[ClassCleanUp ()]** виконання коду після завершення виконання всіх тестів у класі.
- **[TestInitialize ()]** виконання коду до виконання кожного тесту.
- **[TestCleanUp ()]** виконання коду після завершення виконання кожного тесту.

Тести, керовані даними (Data Driven Tests)

Нехай є CSV-файл, в якому зберігаються значення у вигляді матриці. Числа в перших двох стовпчиках типу Int є вихідними даними, в останньому стовпчику – результат їх підсумовування. Вибираємо цей файл і переглядаємо його вміст (рисунки 5.9 – 5.11).

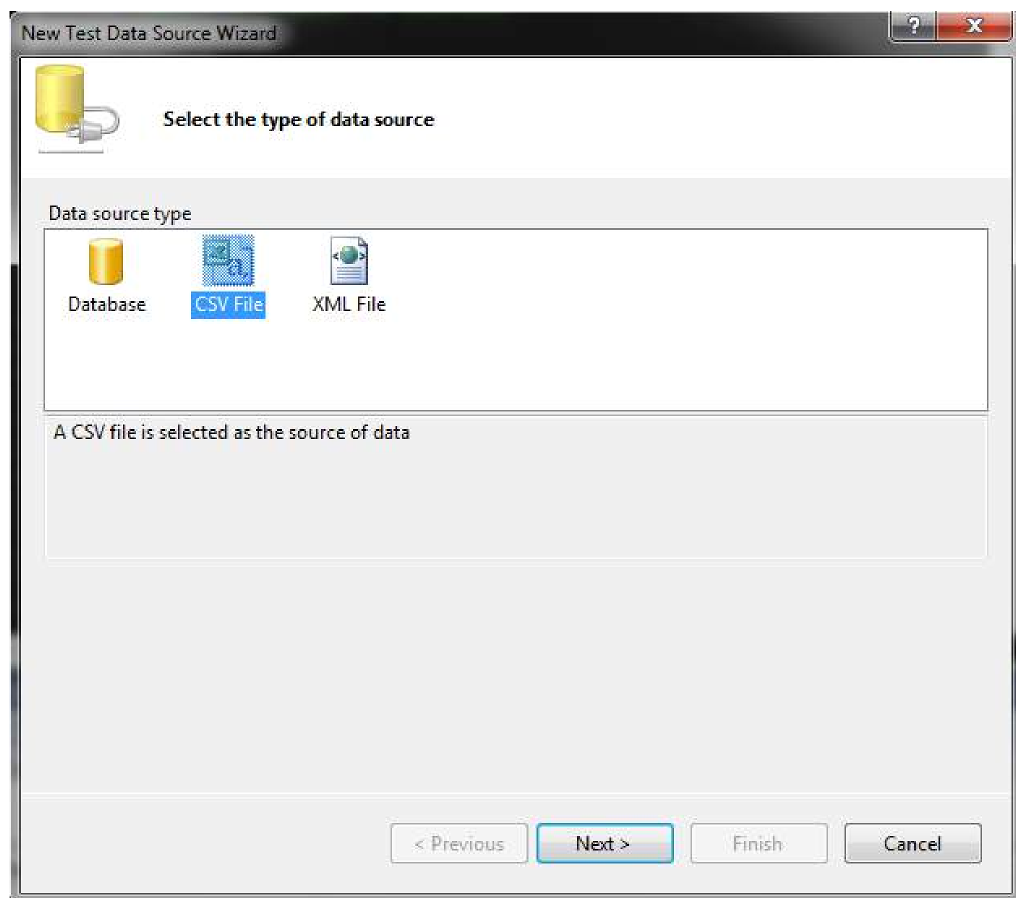


Рисунок 5.9 – Вибір типу джерела ресурсів

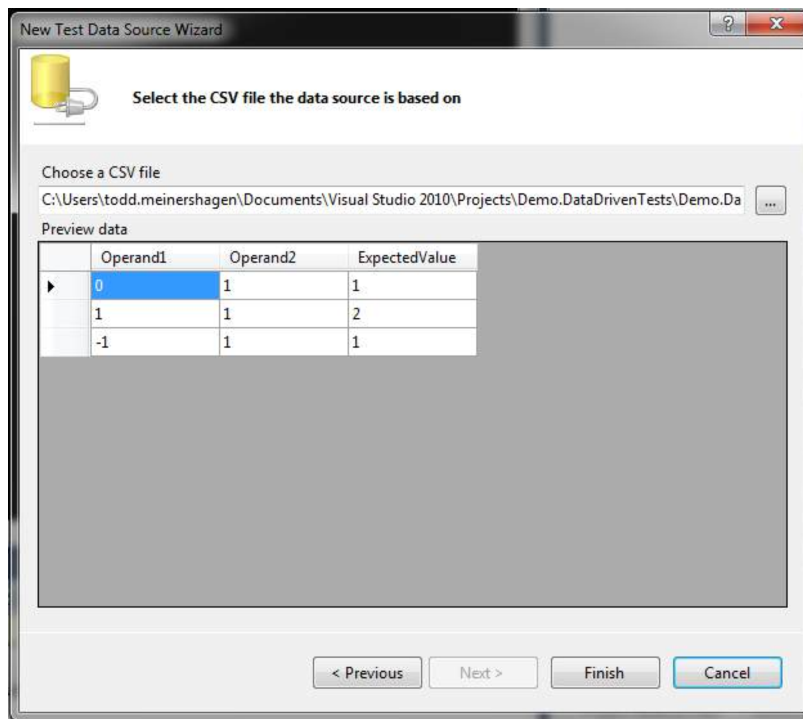


Рисунок 5.10 – Вибір джерела ресурсів

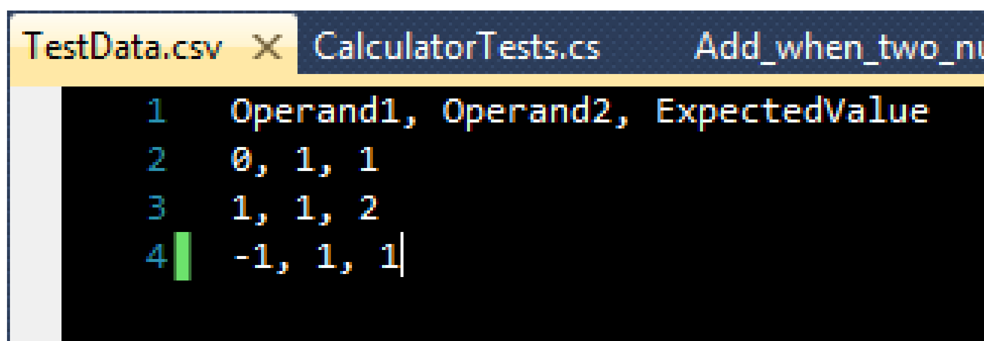


Рисунок 5.11 – Перегляд вмісту CSV-файла

На рисунках 5.12 – 5.13 показано вибір файла, що містить модульний тест, і його властивості.

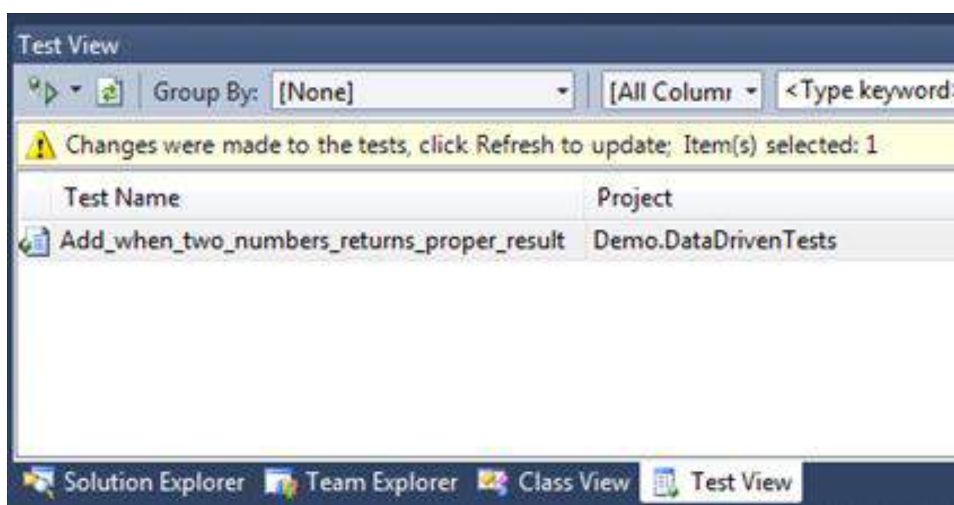


Рисунок 5.12 – Результат вибору модульного тесту

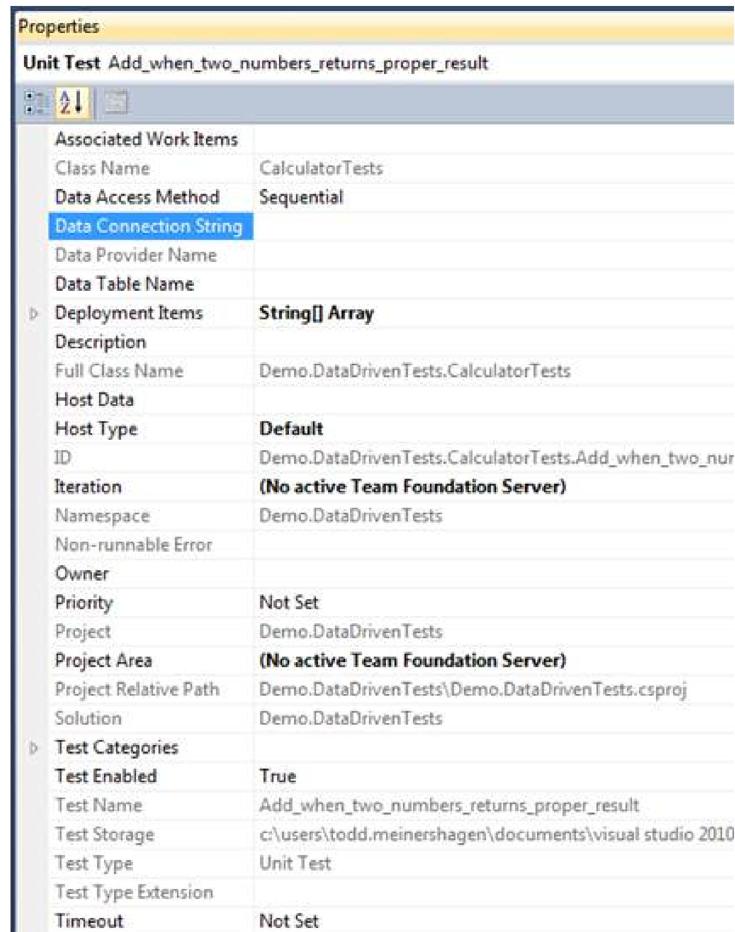


Рисунок 5.13 – Вікно властивостей модульного тесту

На рисунку 5.14 поданий результат заповнення необхідних властивостей модульного тесту.

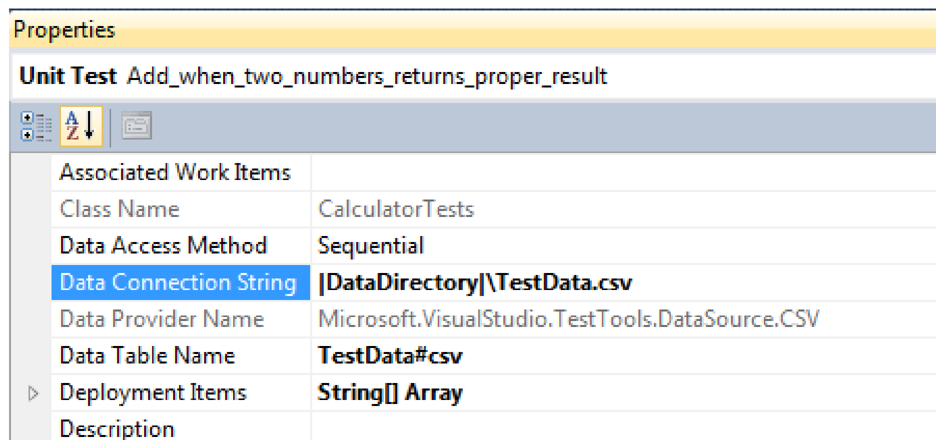


Рисунок 5.14 – Результат заповнення необхідних властивостей модульного тесту:

[**DataSource**("Microsoft.VisualStudio.TestTools.DataSource.CSV", "|DataDirectory|\\TestData.csv", "TestData#csv", DataAccessMethod.Sequential), **DeploymentItem**("Demo.DataDrivenTests\\TestData.csv"), **DeploymentItem**("TestData.csv"), TestMethod]

Приклад модульного тесту, керованого даними:

```
private TestContext  TestContext;
[TestMethod()]
public void CalculatorTest()
{
ICalc target = new Calc();
int actual;
int a = System.Convert.ToInt32(TestContext.DataRow["Operand1"]);
int b = System.Convert.ToInt32(TestContext.DataRow["Operand2"]);
actual = target .Sum(a,b);
int expected = System.Convert.ToInt32(TestContext.DataRow[" ExpectedValue "]);

Assert.AreEqual(expected, actual);
}
```

Запуск модульних тестів на виконання

Для запуску тестів вибираємо в Solution Explorer пункт **[імя_солюшіна] .vsmdi** (рисунок 5.15).

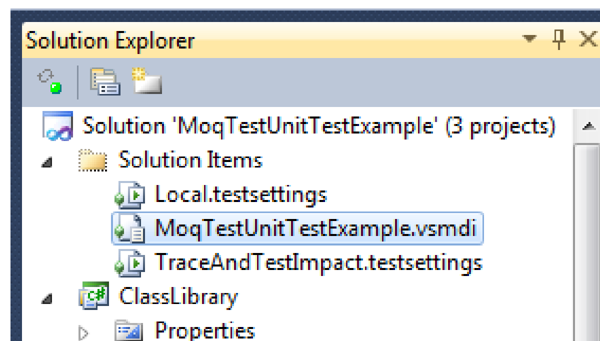


Рисунок 5.15 – Вікно вибору файла Solution Explorer: [имя_солюшина].vsmdi

Отримуємо вікно зі списком модульних тестів за все рішення (рисунок 5.16). Сюди потрапляють усі методи класів, які позначені атрибутами TestClass, методи з атрибутами [TestMethod]:

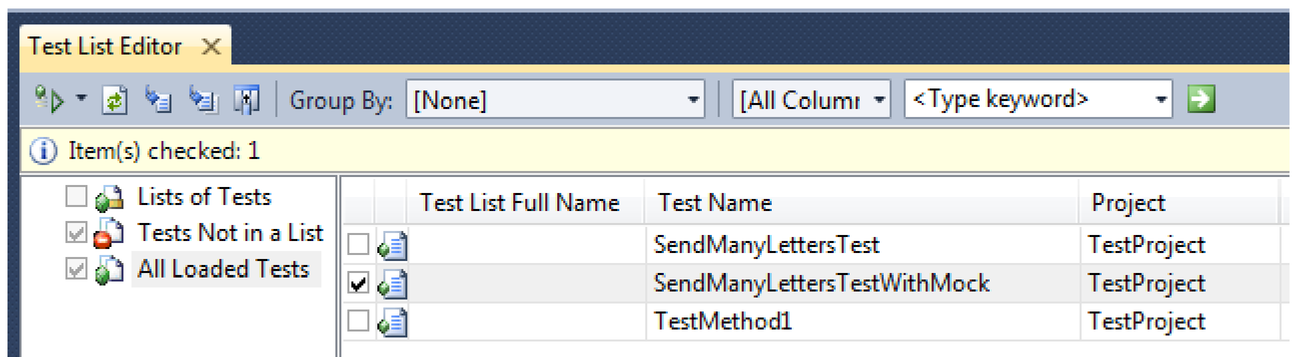


Рисунок 5.16 – Вікно зі списком модульних тестів

Для запуску тесту позначаємо його галочкою і вибираємо пункт Run Checked Tests (рисунок 5.17).

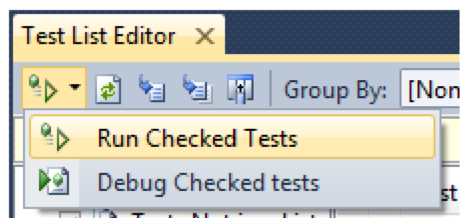


Рисунок 5.17 – Вікно запуску модульних тестів

Результати виконання тестів:

1. В очікуванні ... (рисунок 5.18).

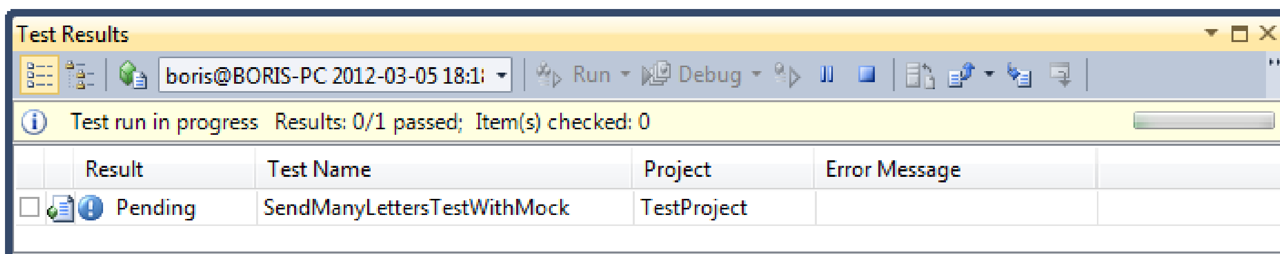


Рисунок 5.18 – Результати виконання модульних тестів. В очікуванні виконання

2. У процесі… (рисунок 5.19).

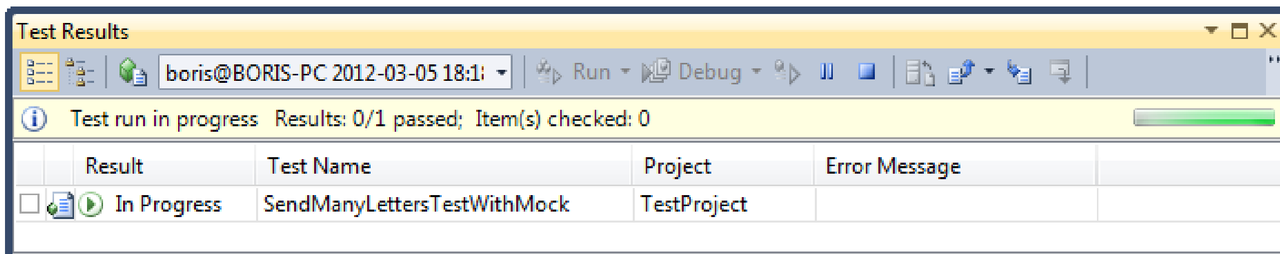


Рисунок 5.19 – Результати виконання модульних тестів. У процесі виконання

3. Виконано успішно (рисунок 5.20).

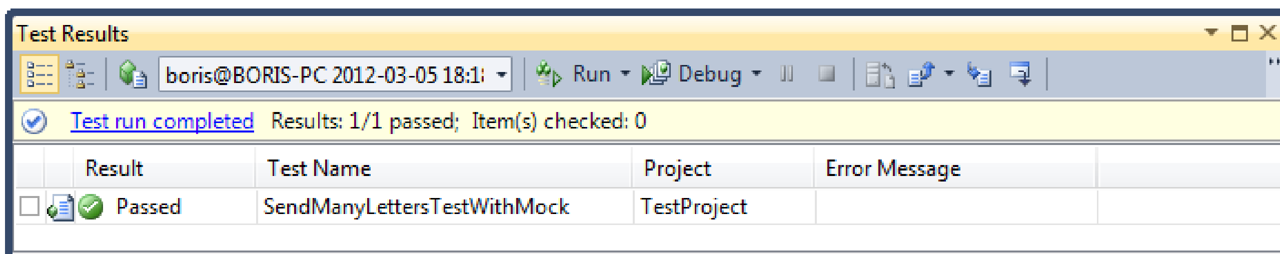


Рисунок 5.20 – Результати виконання модульних тестів. Виконано успішно

4. Виконано з помилкою (рисунок 5.21).

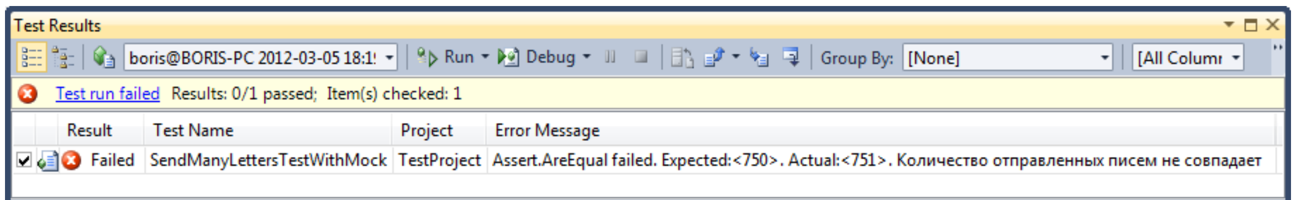


Рисунок 5.21 – Результати виконання модульних тестів. Виконано з помилкою

Оцінка покриття коду тестами

Покриття коду – міра, яка використовується при тестуванні програмного забезпечення. Вона показує відсоток, наскільки вихідний код програми було протестовано.

Існує кілька різних способів вимірювання покриття, основні з них:

Покриття операторів – кожний рядок вихідного коду було виконано і протестовано.

Покриття умов – кожна точка рішення (обчислення виразу істинне або хибне) була виконана і протестована.

Покриття шляхів – чи всі можливі шляхи через задану частину коду були виконані і протестовані.

Покриття методів – чи кожен метод класів був виконаний.

Покриття вхід / вихід – чи виклики функцій і повернення з них були виконані.

Налаштування опцій для оцінки покриття коду

Для налаштування опції аналізу покриття коду тестами вибираємо у Solution Explorer пункт Local.testsettings (рисунок 5.22).

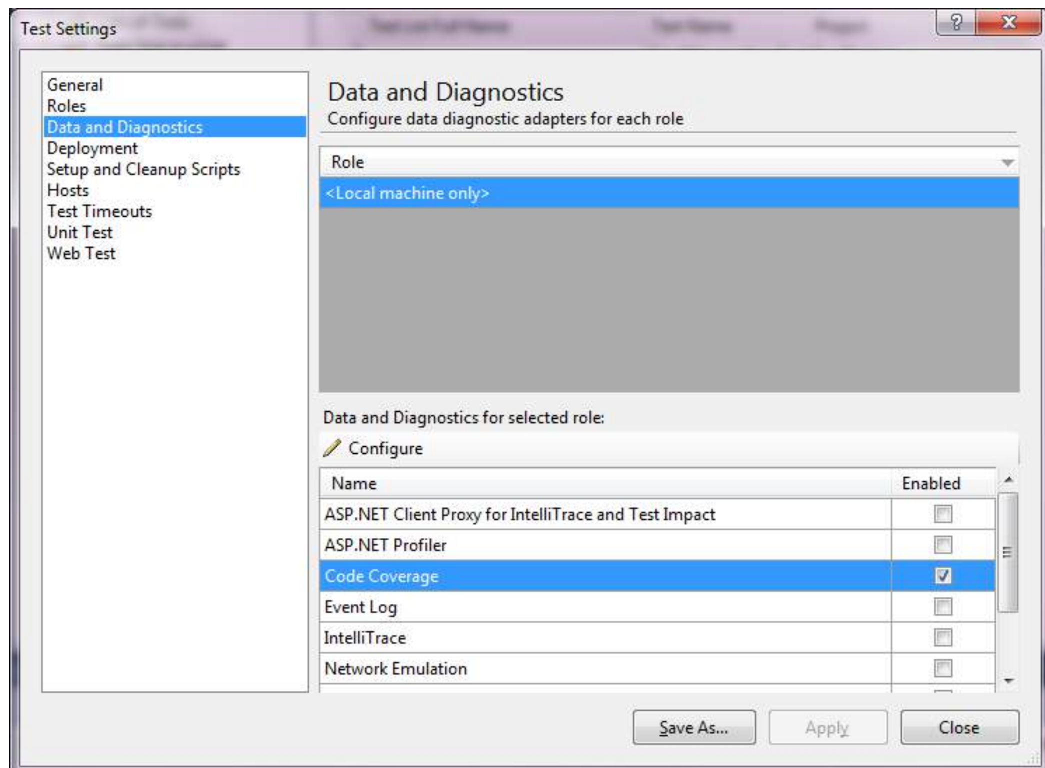


Рисунок 5.22 – Вікно вибору Local.testsettings

У розділі **Data and Diagnostics** зробити активною конфігурацію настроювання покриття коду, поставивши галочку в колонці *Enabled* пункту **Code Coverage.**

Далі запустити детальне настроювання подвійним кліком миші по цьому пункту (рисунок 5.23).

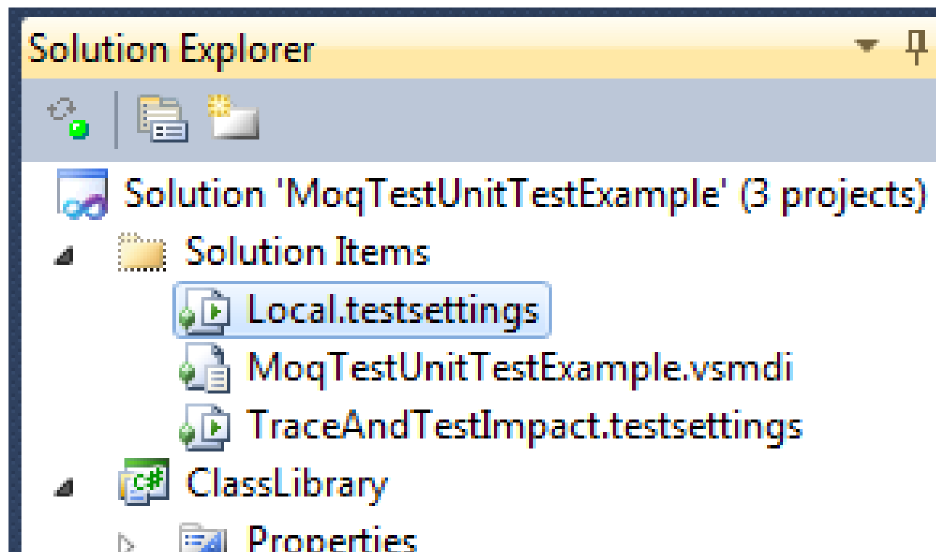


Рисунок 5.23 – Вікно запуску детального настроювання

Вказати, для яких компонент необхідно аналізувати покриття коду тестами.

У розглянутому прикладі таким компонентом є бібліотека класів (**ClassLibrary**), створена раніше (рисунок 5.24).

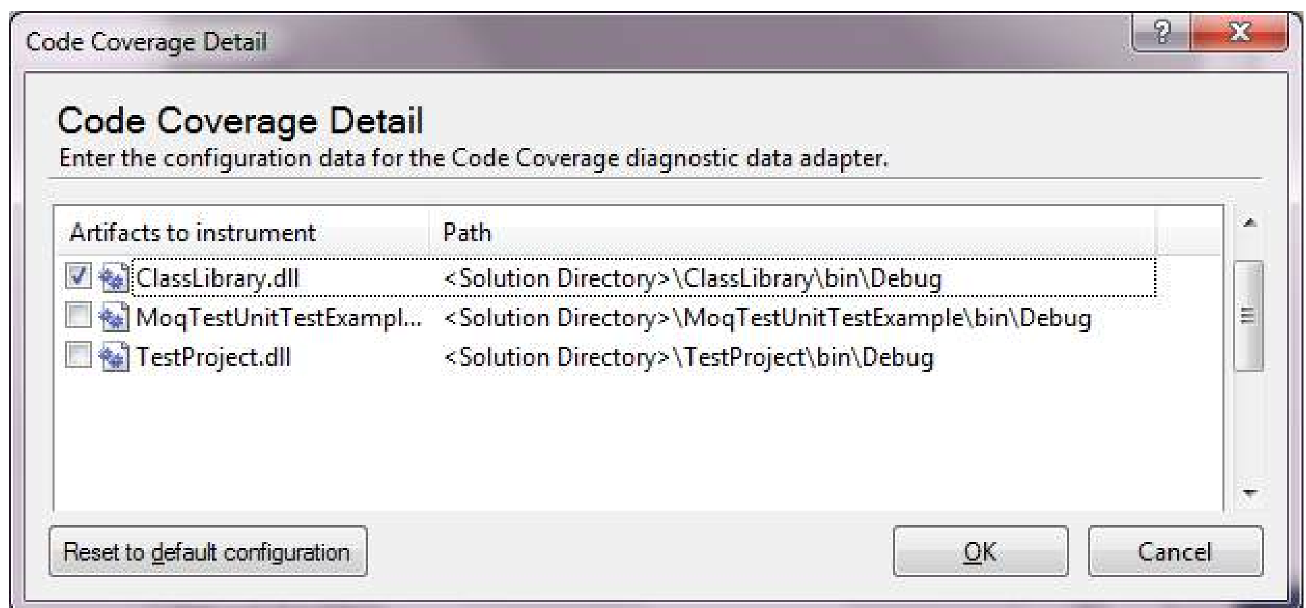


Рисунок 5.24 – Вікно вибору компонент
для аналізу покриття коду тестами

Аналіз покриття коду (результат, наведений на рисунку 5.26).

Для перегляду результату аналізу покриття коду повторно запускаємо тести на виконання і в результаті тесту вибираємо пункт Code Coverage Results (рисунок 5.25).

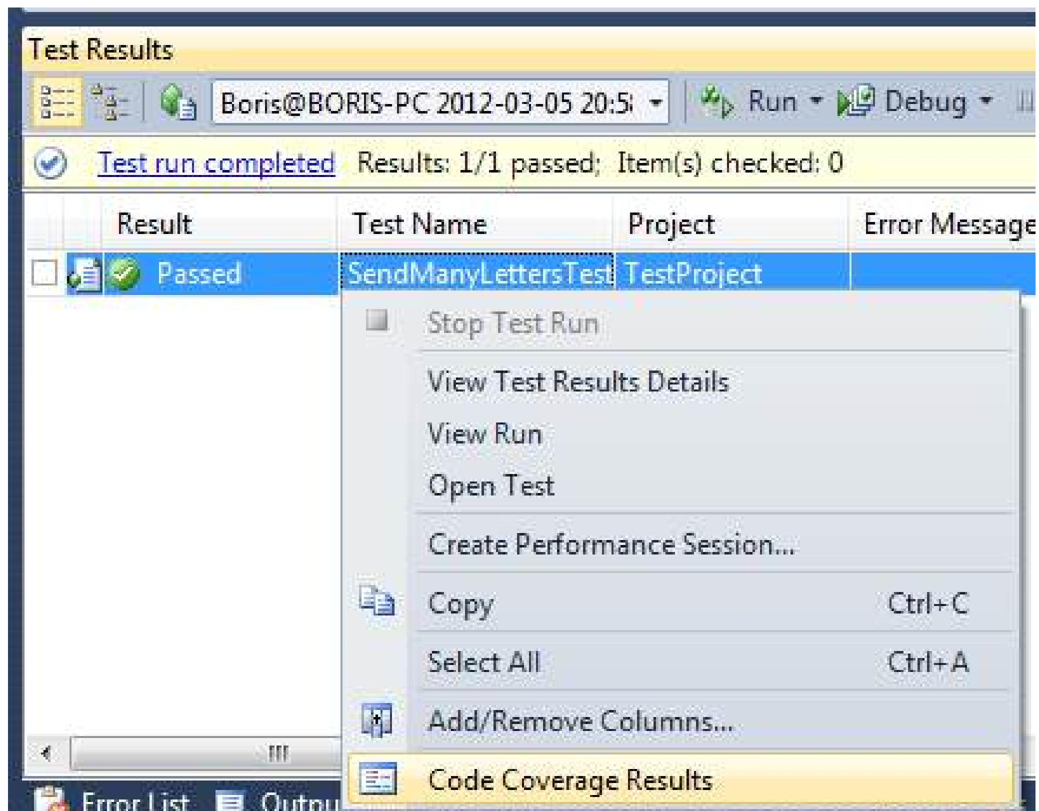


Рисунок 5.25 – Вибір пункту Code Coverage Results

Code Coverage Results

Boris@BORIS-PC 2012-03-05 20:58:25

Hierarchy	Not Covered (Blocks)	Not Covered (% Bloc...	Covered (Blocks)	Covered (% Blocks)
Boris@BORIS-PC 2012-03-05 20:58:25	6	24,00%	19	76,00%
ClassLibrary.dll	6	24,00%	19	76,00%
{ } ClassLibrary	6	24,00%	19	76,00%
MailHelper	6	100,00%	0	0,00%
SendMail(class ClassLibrary.MailWra...	6	100,00%	0	0,00%
MailSpammer	0	0,00%	12	100,00%
.ctor(class ClassLibrary.IMailHelper)	0	0,00%	2	100,00%
SendManyLetters(class System.Colle...	0	0,00%	10	100,00%
MailWrapper	0	0,00%	7	100,00%
.ctor(string,string,string,string,string)	0	0,00%	7	100,00%

Рисунок 5.26 – Результат аналізу покриття коду. Code Coverage Results

6 ДЕЛЕГАТИ, АНОНІМНІ МЕТОДИ, ЛЯМБДА-ВИРАЗИ. ВИКОРИСТАННЯ MOQ-ОБ'ЄКТІВ

Делегати, анонімні методи, лямбда-вирази

Лямбда-вираз – це анонімна функція, за допомогою якої можна створювати типи делегатів або дерев виразів. Користуючись лямбда-виразами, можна також написати локальні функції, а потім їх передавати в інші функції як аргументи або повертати з них [функцій] як значення.

Лямбда-оператор

Для створення лямбда-виразів можна визначити вхідні параметри (якщо такі є) зліва від лямбда-оператора **=>**.

Лямбда-вираз з виразом з правого боку оператора **=>** називається виразом лямбда.

Наприклад, лямбда-вираз **x => x * x** приймає параметр з ім'ям x і повертає значення x, зведене у квадрат.

Приклад делегата:

```
delegate int del(int i);
static void Main(string[] args)
{
   del myDelegate = x => x * x;
   int j = myDelegate(5); //j = 25
}
```

Параметри лямбда-виразів

(input parameters) => expression

Якщо лямбда має тільки один вхідний параметр, дужки можна не ставити, у всіх інших випадках вони є обов'язковими. Якщо вхідних параметрів два і більше, то їх розділяють комами і ставлять у дужки:

(x, y) => x + y.

Типи вхідних параметрів

Типи вхідних параметрів вказувати не обов'язково, але іноді компілятору буває важко або навіть неможливо вивести типи вхідних параметрів. У цьому випадку типи можна вказати в явному вигляді:

(Int x, string s) => s.Length> x

Відсутність вхідних параметрів задається порожніми дужками:

() => SomeMethod ()

Лямбда-оператор

(Input parameters) => **{**statement;**}**

Лямбда-оператор (або операційна лямбда) нагадує вислів-лямбду, за винятком того, що оператор (або оператори) ставлять у фігурні дужки:

```
(a,b)=> {
  if (a>b) return a-b;
  return a+b;
}
```

Ізоляція тестованого коду (рисунок 6.1).

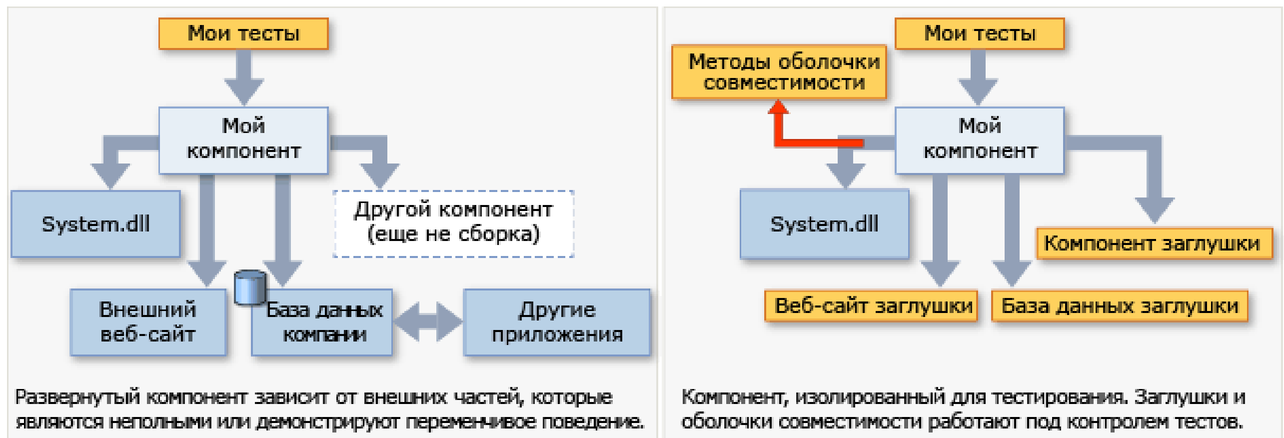


Рисунок 6.1 – Приклад ізоляції тестованого коду

Залежності класів. Приклад (рисунок 6.2).

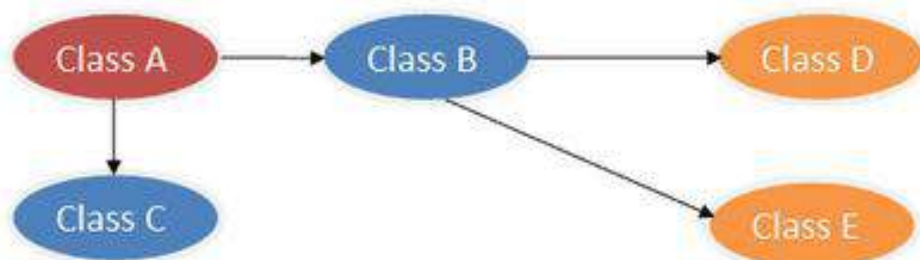


Рисунок 6.2 – Схема залежностей класів

Як видно з діаграми класів, клас A залежить від класу B і класу C, клас B залежить від класу D і класу E.

Що потрібно зробити для модульного тестування класу A?

Перш за все, потрібно ізолювати клас A (рисунок 6.3). Необхідно здійснити підміну екземплярів класів B і C, які дадуть можливість перевірити поведінку екземпляра класу A.

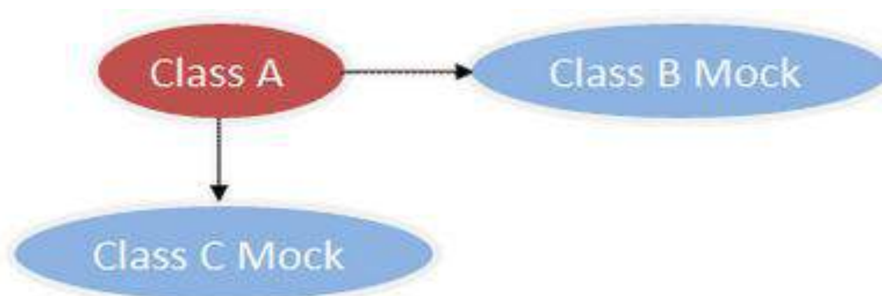


Рисунок 6.3 – Схема ізоляції класу A (підміна екземплярів класів B і C)

Приклад реалізації залежності класів (рисунок 6.4).

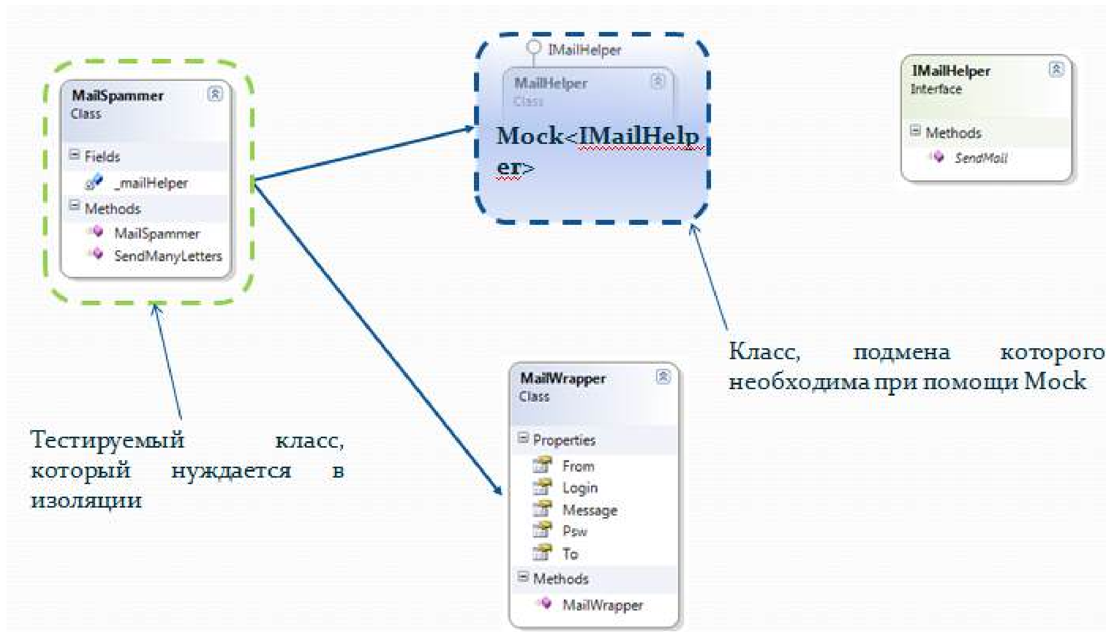


Рисунок 6.4 – Приклад реалізації залежності класів

Додавання Moq-фреймворка до проекту:

1. Завантажуємо останню версію бібліотеки з сайту: http://code.google.com/p/moq/downloads/list

2. Розпаковуємо архів і додаємо до тестового проекту посилання на бібліотеку (Add Reference ...) \\ Moq.4.0.10827 \ NET40 \ Moq.dll (рисунки 6.5 і 6.6).

3. Бажано скопіювати цю бібліотеку в папку проекту і після цього додати на неї посилання (рисунок 6.7).

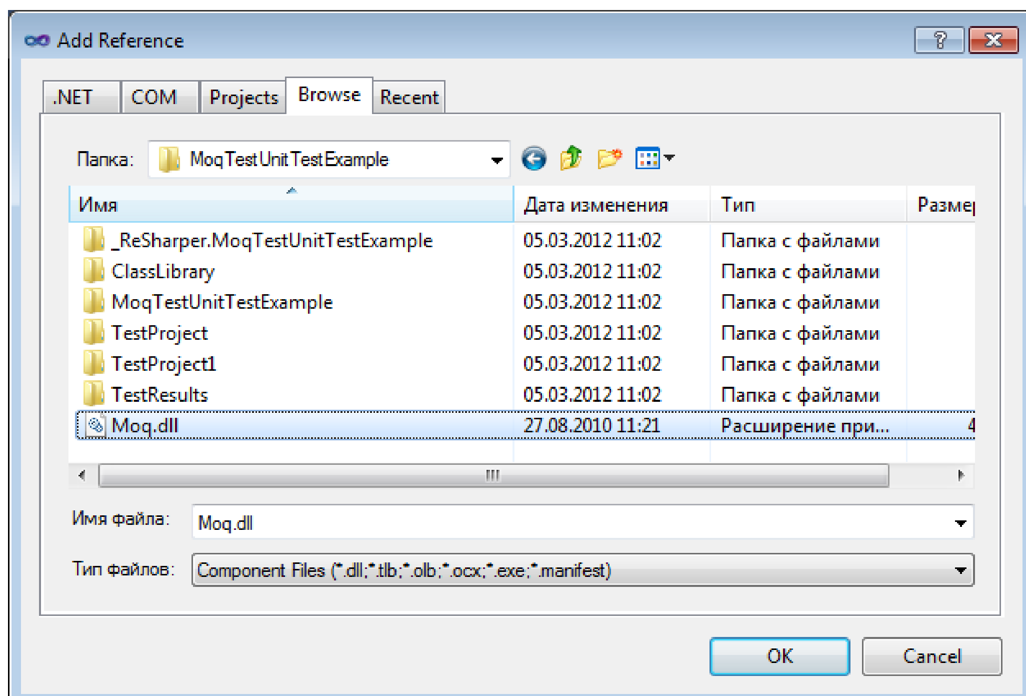


Рисунок 6.5 – Вікно вибору Moq-бібліотеки

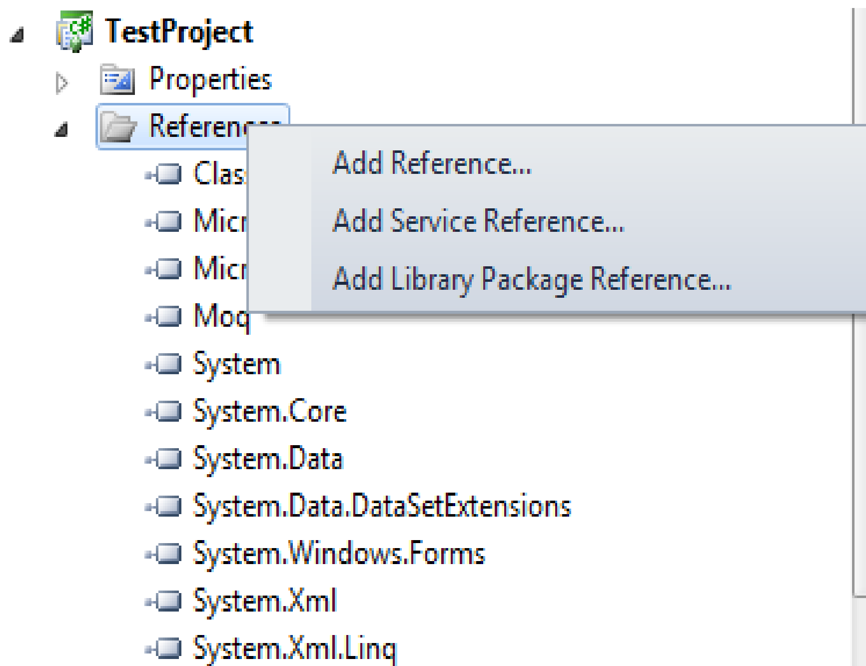


Рисунок 6.6 – Контекстне меню додавання до тестового проекту посилання на бібліотеку

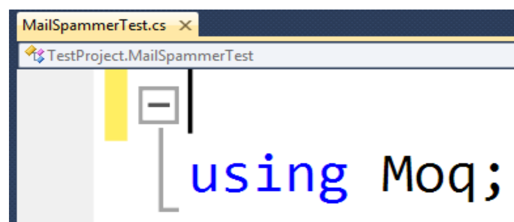


Рисунок 6.7 – Додавання посилання на Moq-бібліотеку

Заміна об'єкта

У попередньому лістингу тестів створювався реальний клас, який відправляє листи:

IMailHelper mailHelper = new MailHelper ();

Для виключення взаємодії з реальними поштовими серверами зробимо заміну:

var mailHelper = new Mock <IMailHelper> ();

Створимо параметризований інтерфейсом *IMailHelper* Mock-об'єкт.

Впровадження Mock-об'єкта в залежний клас буде мати такий вигляд:

var mailSpammer = new MailSpammer (mailHelper.Object);

Визначення поведінки об'єкта, що потребує заміни

Mock-об'єкт необхідно «навчити» реагувати на вхідні параметри методів реалізованого інтерфейсу, тобто задати поведінку методів і властивостей об'єкта, що потребує заміни. Для цього використовується такий синтаксис:

mailHelper.Setup (m => m.SendMail (mailWrapper)). Returns (true);

Ця конструкція означає:

«Встановити методу SendMail при отриманні об'єкта mailWrapper значення, що повертається, true».

При цьому створення об'єкта *mailWrapper* має відбутися до встановлення цього правила, оскільки порівняння еквівалентності об'єктів відбуватиметься за посиланням (адресою виділеної пам'яті).

Перепишемо тести. Нехай кожний другий лист буде відправлено з помилкою, у цьому випадку метод *SendMail* повертає *false*.

Unit-test з використанням Mock:

```
[TestMethod ()]
      public void SendManyLettersTestWithMock ()
      {
         int count = 1500;
         var letters = new List <MailWrapper> ();
         var mailHelper = new Mock <IMailHelper> ();
         int expected = 0;
         var rand = new Random (123456);
         for (int i = 0; i <count; i ++)
         {
               MailWrapper mailWrapper = new MailWrapper ( "user" + i,
"user" + i * 2, "name" + i, "psw" + rand.Next (), "Лист" + i);
            letters.Add (mailWrapper);
            if (i% 2 == 0)
            {
               expected ++;
               mailHelper.Setup (m => m.SendMail (mailWrapper)).
Returns (true);
            }
            else
            {
               mailHelper.Setup (m => m.SendMail (mailWrapper)).
Returns (false);
            }
         }
         var mailSpammer = new MailSpammer (mailHelper.Object);
         var actual = mailSpammer.SendManyLetters (letters);
         Assert.AreEqual (expected, actual, "Кількість відправлених
листів не збігається");
      }
```

7 ПРИКЛАД ВИДІЛЕННЯ ТЕСТОВИХ ВИПАДКІВ

Припустимо, що система розсилки масових інформаційних повідомлень випускникам кафедри повинна формувати список розсилки на основі зазначеного файла Excel і застосовувати до нього специфічні фільтри (дата народження, стать, спеціальність і т. п.).

На рисунках 7.1 і 7.2 зображені вікна для розсилання інформаційних листів і завантаження одержувачів відповідно.

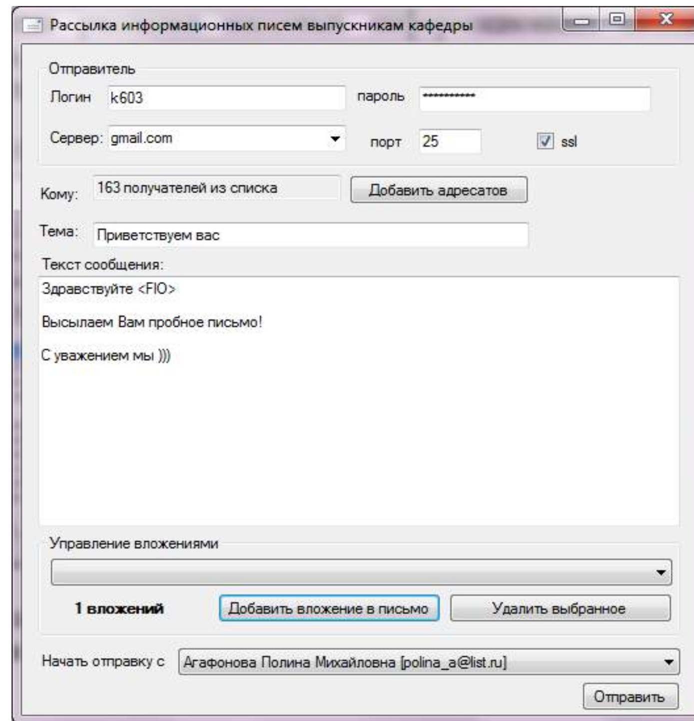


Рисунок 7.1 – Вікно розсилання інформаційних повідомлень випускникам кафедри

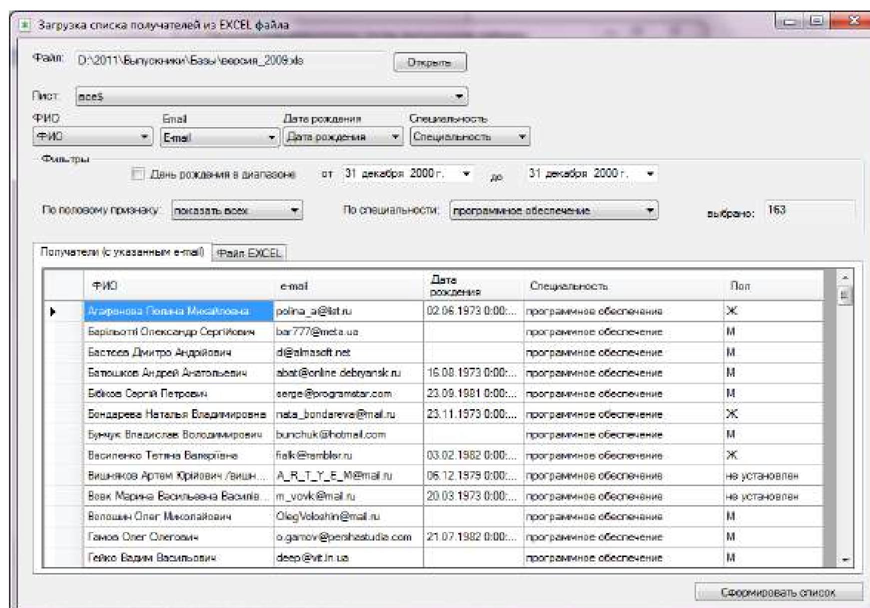


Рисунок 7.2 – Вікно завантаження списку одержувачів

Тестові вимоги

Необхідно визначити тестовану область системи – вказати частину проектної документації, вихідних текстів, виконуваного коду, що піддаються тестуванню із зазначенням типу тестування (автоматизовані тести, формальні інспекції, тестування на моделях, напівнатурні випробування і т. п.). Необхідно зафіксувати тестовані й нетестовані аспекти.

Тестування проводиться з урахуванням погляду кінцевого користувача. Розроблені тести можуть бути використані для приймального тестування.

Тестовані аспекти

Згідно з цим планом виконують функціональне тестування ПЗ у режимах генерації списку одержувачів, формування повідомлень і відправки листів.

Також передбачається провести модульне тестування класу *Recipient*, який відповідає за зберігання і перетворення інформації про одержувача листів.

Нетестовані аспекти

Згідно з цим планом не передбачається виконувати:

– функціональне тестування системи в режимі аутентифікації користувача і перевірки кількості реально відправлених листів;

– нефункціональне тестування, у тому числі тестування навантаження, тестування продуктивності, тести щодо зручності викоритсання (usability);

– тестування буде проводитись з використанням системи автоматизованого тестування *TestComplete*, а також інтегрованих у середовище розроблення *Microsoft Visual Studio* юніт-тестів.

Опис варіантів використання

Описати ролі користувачів залежно від рівня доступу і дій, які вони можуть виконувати в системі. В процесі тестування описані тут варіанти використання дозволяють простіше оцінити точність реалізації вимог користувачів і провести покрокову перевірку цих вимог.

Стратегія використання прецедентів при визначенні вимог викликає необхідність додаткового запитання "Що користувачі чекають від системи?", а також задавати запитання "Що система повинна зробити для учасників форуму?". Такий підхід дозволяє шукати функції, які потрібні багатьом користувачам, і виключати ті можливості, які не можуть допомогти користувачам виконувати свої повсякденні завдання [35].

Для більш повного покриття тестових випадків розглянемо діаграму варіантів використання, яку наведено на рисунку 7.3 для системи розсилки масових інформаційних повідомлень випускникам факультету.

У тестованого програмного забезпечення можна виділити три основних режими функціонування: робота з файлами адрес розсилки, формування наповнення розсилки і, безпосередньо, відправлення листів зазначеним адресатам. І два допоміжних режими – авторизація і ведення звіту відправлень (рисунок 7.3).

Рисунок 7.3 – Діаграма варіантів використання ПЗ

Об'єкти тестування для кожної ролі користувача

Діаграма містить список тих об'єктів (Use-Case-ів, функціональних вимог, нефункціональних вимог), які були визначені як об'єкти тестування. Цей список показує те, що буде протестовано.

В результаті проведеного аналізу діаграми варіантів використання виділили такі варіанти використання, які вимагають додаткового тестування:

- Авторизація.
- Формування списку розсилки:
  - вибір файла БД;
  - вибір стовпців;
  - додавання фільтрів.
- Введення заголовка повідомлення.
- Введення тексту повідомлення.

- Робота з вкладеннями.
- Відправка листів.
- Автопідстановка шаблонів.
- Відправка, починаючи з зазначеного одержувача.
- Формування і збереження звіту про відправку.
- Клас Recipient.

Стратегія тестування

Головними питаннями тестової стратегії є техніки тестування, які будуть використовуватися, і критерій, за яким можна було б визначити, що тестування завершено. Стратегію тестування визначають, враховуючи вид тестування.

Необхідно дати опис підходів до тестування (unit-тестів, тестування за допомогою Test Complete, системи відстежування помилок (bug tracking) та інших спеціальних засобів тестування), які будуть застосовуватися в ході проведення тестування – у рамках одного тестового плану слід дотримуватися загальних методик тестування системи. Описати, наскільки детально будуть тестуватися система та її компоненти.

Рівні тестування:

Системне тестування, з урахуванням погляду кінцевого користувача, для тестування варіантів використання ПЗ. Стратегія тестування – методом «чорної скриньки».

Модульне тестування (для перевірки методів розробленого класу Recipient) з точки зору розробника-програміста, який використовує цей клас. Стратегія тестування – методом «білої скриньки».

Спеціальні засоби тестування:

Тестування проводять за допомогою *TestComplete* і вбудованих у *Microsoft Visual Studio Unit-тестів*. Деякі тести будуть проводитися вручну.

Вимоги до оточення:

Для виконання тестів потрібен встановлений *TestComplete* версії 7 або вище. Також для виконання деяких тестів необхідний доступ до інтернету з відкритими портами для відправки електронної пошти. Для функціонування тестованої системи необхідний встановлений *Microsoft Office 2007* і *.NET Framework 3.5* і вище.

Розроблення позитивних і негативних тестових випадків (test case)

Основна вимога до контрольного прикладу – опис перевірки чітко визначеної самостійної частини функціональності (або властивостей) програмного забезпечення і очікуваних результатів. У загальному випадку, один контрольний приклад відповідає одному варіанту використання [36].

Розроблення контрольних прикладів тестування (test case)

Тестовий випадок (Test Case) описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації тестованої функції або її частини.

Виділимо тестові випадки на основі аналізу об'єктів тестування для системного тестування (виділені на основі аналізу діаграми use-case):

1. Перевірка авторизації користувача – логін і пароль користувача повинні бути введені. При підключенні до smtp-сервера відповідь має бути позитивною.

2. Перевірка формування списку розсилки – контролюється коректне завантаження і аналіз Excel-файла списку одержувачів, автоматичний вибір стовпців за заголовком, а також зміна заголовків, перевірка установлення фільтрів.

3. Перевірка контролю введення заголовка повідомлення – заголовок повідомлення не повинен бути порожнім.

4. Перевірка введення тексту повідомлення – текст повідомлення не повинен бути порожнім і може містити керуючі символи переходу на новий рядок, а також шаблони, які будуть замінені при автопідстановці.

5. Перевірки управління вкладеннями – контролюється можливість додавати і видаляти файли вкладення до сформованої розсилки.

6. Перевірка відправлення листів – тестується можливість відправлення листів через зазначений smtp-сервер вибраним одержувачам (для успішного проходження цього ТЕСТ-кейса необхідне інтернет-підключення з відкритими портами).

7. Контроль автопідстановки шаблонів – перевірка автоматичної заміни шаблонів у тексті листа на відповідні значення зі списку одержувачів (підстановка імен чоловічого / жіночого роду).

8. Контроль відправки листів із зазначеного одержувача – перевірка можливості вказати зі списку одержувачів адресу, з якої необхідно продовжити відправку, а також контроль того, що відправка почалася саме з цього одержувача.

9. Перевірка формування і збереження звіту про відправку – звіт про відправку має містити детальну інформацію про розсилку, яку здійснюють – дату і час, текст повідомлення, адреси отримувачів, результат успішності відправлення листа кожному з адресатів.

10. Модульне тестування класу Recipient. Перевірка функціонування всіх методів цього класу. Основну увагу приділити трьом властивостям: sex, FIO і emails. Решта властивостей просто не повинні самостійно змінювати свого значення. Властивості sex, FIO і emails змінюються за такими правилами:

1. При тестуванні властивості sex проводиться аналіз «по батькові» за закінченням запису. Якщо властивість FIO закінчується на * вич, то властивість sex повертає рядок «М», якщо властивість FIO закінчується на * вна, то властивість sex повертає рядок «Ж», в іншому випадку повертає рядок "не встановлено".

2. При тестуванні властивості FIO необхідно перевірити, щоб встановлена властивість завжди повертала рядок, усі слова в якому починаються з великої літери.

3. При тестуванні властивості emails необхідно перевірити, щоб рядок, розділений роздільниками типу точки, точки з комою або пробілами, розбивався на масив рядків адрес одержувача.

Опис коректних і некоректних тестових даних для кожного тестового випадку

Тестові випадки поділяються, за очікуваним результатом, на позитивні й негативні.

Позитивний тестовий випадок використовує тільки коректні дані й перевіряє, чи додаток правильно виконав функцію, що викликається.

Негативний тестовий випадок оперує як коректними, так і некоректними даними (мінімум – один некоректний параметр), і ставить за мету перевірку виняткових ситуацій (спрацьовування валідаторів), а також перевіряє, що функція, яка викликається додатком, не виконується при спрацьовуванні валідатора.

Опис коректних і некоректних тестових даних для кожного контрольного прикладу тестування

У таблиці 7.1 подано тестовий набір коректних і некоректних даних для кожного тестового випадку.

Таблиця 7.1 – Тестовий набір даних

Контрольний приклад	Коректні дані	Некоректні дані
А.1. Перевірка авторизації користувача	*Логін: «Vasya»* *Пароль: «Passw»*	*Логін: пусте поле або не зареєстрований користувач* *Пароль: пусте поле*
А.2. Перевірка формування списку розсилки	Існуючий файл Excel, який містить у собі заголовки і більше, ніж один запис	Некоректне ім'я файла, порожній файл або зазначені заголовки не відповідають змісту
А.3. Перевірка контролю введення заголовка повідомлення	*Заголовок:* «План заходів святкування дня ХАІ»	Тема: Нічого не введено
А.4. Помилки під час введення тексту повідомлення	*Тіло повідомлення:* Привіт <FIO> Надсилаємо Вам пробного листа! З повагою, ми)))	*Тіло повідомлення: нічого не введено*

Закінчення таблиці 7.1

Контрольний приклад	Коректні дані	Некоректні дані
А.5. Перевірки управління вкладеннями	Вказуються існуючі файли, до яких є права доступу на читання	Вказані неіснуючі файли або файли з закритим рівнем доступу
А.6. Перевірка відправки листів	Проведена авторизація, сформований список одержувачів, заповнені заголовок і тіло повідомлення	Будь-яке з перелічених полів не заповнено або введені не коректні дані авторизації
А.7. Контроль автопідстановки шаблонів	*Тіло повідомлення:* Привіт <FIO> Надсилаємо Вам пробний лист! З повагою, ми)))	*Тіло повідомлення:* Привіт <тут хочу, щоб було його ПІБ> Надсилаємо Вам пробного листа! З повагою, ми)))
А.8. Контроль відправки листів з зазначеного одержувача	Вибраний будь-який одержувач зі сформованого списку одержувачів	–
А.9. Перевірка формування і збереження звіту про відправку	Для збереження вказана локальна папка і введено коректне ім'я файла	Шлях до файла звіту заданий некоректно чи немає прав доступу до збереження у цій папці
А.10. Перевірка статевої ознаки запису по закінченню прізвища	a. Поле FIO приймає значення рядка, який закінчується на «-вич» або на «-вна»	a. Поле FIO приймає значення рядка, яка не закінчується на «-вич» або на «-вна»
	b. Поле FIO містить у собі хоча б одне слово	b. Поле FIO пусте
	c. Поле EMail містить одну або кілька адрес електронної пошти, розділених комою, крапкою з комою або пробілами	c. Поле EMail пусте або містить тільки роздільники

Умови, за яких кожен тест-кейс повинен бути перевірений

Кожен тест-кейс повинен мати три частини:

Передумови – список дій, які приводять систему до стану, придатного для проведення цього тесту; або список умов, виконання яких свідчить про те, що система знаходиться у придатному для проведення основного тесту стані.

Описи тесту – список дій, які переводять систему з одного стану до іншого, для отримання результату, на підставі якого можна зробити висновок про задоволення реалізації поставлених вимог.

Післяумови – список дій, які переводять систему в первинний стан (стан до проведення тесту – initial state).

Критерії припинення тестування

Визначити метрики вимірювання повноти тестованого функціоналу системи:

- тестове покриття;
- деталізація тест-кейсів;
- час проходження тест-кейса;

Існує два широко застосовуваних підходи до оцінювання і вимірювання тестового покриття:

- покриття вимог;
- покриття коду.

Приклади критеріїв закінчення тестування:

- результати тестування задовольняють критерії якості продукту;
- вимоги до кількості відкритих багів виконано;
- витримка певного періоду без зміни вихідного коду програми **Code Freeze (CF)**;
- витримка певного періоду без відкриття нових багів **Zero Bug Bounce (ZBB)**.

Умови, за яких кожен тест-кейс має бути перевірений (див. таблиці 7.2 – 7.5).

Таблиця 7.2 – Позитивний тестовий випадок для тесту «Перевірка формування списку розсилки»

Назва	А.2. Перевірка формування списку розсилки	
Функція	Формування списку одержувачів	
Дія	**Очікуваний результат**	**Результат тесту:**
Передумова:		
Підготуйте файл зі списком одержувачів	Excel файл адресатів, розташований на Вашому локальному комп'ютері	
Запустіть систему розсилки	ПЗ запущено. Відкрито головну форму	
Кроки тесту:		
Натисніть на кнопку «Додати адресатів»	Відкрилася форма завантаження списку одержувачів	
Вкажіть підготовлений EXCEL файл, який містить список одержувачів з адресами і ПІБ	Список одержувачів завантажився у таблицю	
Перевірте відповідність зазначених заголовків і колонок у файлі	Усі заголовки було визначено правильно	
У випадаючому списку фільтрів спеціальностей встановіть фільтр за фахом «Програмна інженерія»	Фільтр встановлений. У таблиці одержувачів залишилися передплатники, відповідні цьому фільтру	
Встановіть прапорець «День народження в діапазоні»	Прапорець встановився, а поля введення дати почали бути активними	
Вкажіть діапазон дня народження від 21 березня 2011 року до 27 березня 2011 року	Фільтр «дата народження» встановлено. У таблиці одержувачів залишилися тільки ті, у кого день народження припадає з 21-го до 27-го березня включно, рік ігнорується	
Натисніть кнопку «Сформувати список»	Форма завантаження одержувачів закрилася. Відбулося повернення до головної форми. На головній формі у полі «Кому» зазначена кількість відфільтрованих одержувачів	
Постумова:		
Потрібно закрити ПЗ	ПЗ було закрито без помилок	

Таблиця 7.3 – Позитивний тестовий випадок для тесту «Контроль статевої ознаки запису по закінченню прізвища»

Назва	А.10.a Перевірка статевої ознаки запису згідно з закінченням прізвища	
Функція	Статева приналежність	
Дія	**Очікуваний результат**	**Результат тесту:**
Передумова:		
Створіть об'єкт Recipient target = new Recipient();	Об'єкт створено. Виключень не виникало	
Кроки тесту:		
Задайте властивість FIO об'єкта target target.FIO = "Петро Іванович";	Поле встановлено. Виключень не виникало	
Проаналізуйте значення властивості sex у об'єкта target	target.sex прийняло значення «М»	
Постумова:		
Вкажіть порожнє посилання на об'єкт target = null	Виключень не виникало	

Таблиця 7.4 – Позитивний тестовий випадок для тесту «Контроль нормалізації поля ПІБ»

Назва	А.10.b Контроль нормалізації поля ПІБ	
Функція	Нормалізація ПІБ	
Дія	**Очікуваний результат**	**Результат тесту:**
Передумова:		
Створіть об'єкт Recipient target = new Recipient();	Об'єкт створено. Виключень не виникало	
Кроки тесту:		
Задайте властивість FIO об'єкта target target.FIO = " пЕтро ІванОвич";	Поле встановлено. Виключень не виникало	
Проаналізуйте значення властивості sex у об'єкта target	target.FIO прийняло значення «Петро Іванович»	
Постумова:		
Вкажіть порожнє посилання на об'єкт target=null	Виключень не виникало	

Таблиця 7.5 – Позитивний тестовий випадок для тесту «Контроль поділу електронних адрес»

Назва	А.10.с Контроль поділу електронних адрес	
Функція	Поділ електронних адрес	
Дія	**Очікуваний результат**	**Результат тесту:**
Передумова:		
Створіть об’єкт Recipient target = new Recipient();	Об’єкт створено. Виключень не виникало	
Кроки тесту:		
Задайте властивість FIO об'єкта target target.Email = "monkey@microsoft.com; vasya@rambler.ru, zozo@test.net";	Поле встановлено. Виключень не виникало	
Проаналізуйте значення властивості sex у об'єкта target	target.emails прийняло значення {"monkey@microsoft.com”, “vasya@rambler.ru”, “zozo@test.net”}	
Постумова:		
Вкажіть порожнє посилання на об'єкт target=null	Виключень не виникало	

8 ВЕРСІОНУВАННЯ ПРОДУКТІВ. СТВОРЕННЯ ІНСТАЛЯТОРІВ

Логічна структура пакета

Інсталяційний пакет описує установлення одного продукту і має свій GUID. Продукт складається з **компонент** (*components*) (теж мають свої GUIDи), згрупованих у **можливості** (*features*) [37].

Компонент (component) – мінімальна неподільна установча одиниця, що являє собою групу файлів, значень реєстру, створюваних папок й інших елементів, об'єднаних загальною назвою (ім'ям компоненти), які або встановлюються разом, або не встановлюються. Компоненти приховані від кінцевого користувача. Кожна компонента має ключовий шлях (key path) – наприклад, ім'я свого головного файла, за яким визначається наявність цієї компоненти на комп'ютері користувача.

Можливість (feature) – це ієрархічна група компонент і / або інших можливостей. Коли при встановленні компонент з’являється діалог вибору встановлюваних частин програми, користувач управляє вибором саме можливостей. Вибір цих можливостей дає змогу встановити усі компоненти, які входять до складу інсталяційного пакета.

Фізична структура пакета

Файл .msi являє собою складовий документ OLE (OLE compound document – у тому ж форматі-контейнері зберігаються документи Microsoft Word, Excel і т. д.), в якому міститься невелика реляційна база даних – набір з декількох десятків взаємозалежних таблиць, що містять різну інформацію про продукт і процеси установлення.

Окрім бази, структура файла .msi передбачає розміщення в ньому користувальницьких сценаріїв і допоміжних DLL, якщо такі потрібні для встановлення, а також самих встановлюваних файлів, запакованих у форматі .cab. Файли можна розміщувати і окремо від пакета, в запакованому або розпакованому вигляді (зі збереженням структури каталогів).

Перегляд фізичної структури пакета (база даних файла msi)

За допомогою програми ORCA можна переглядати і редагувати таблиці БД інсталяційного пакета. На скріншоті відкрито таблицю властивостей, яка містить дві колонки – Property і Value (рисунок 8.1).

Приклад програмного вилучення інформації з БД файла msi наведено у лістингу коду (рисунок 8.2) [38].

TortoiseSVN-1.7.1.22161-x64-svn-1.7.1.msi - Orca

File Edit Tables Transform Tools View Help

Tables
Directory
DrLocator
Environment
Error
EventMapping
Feature
FeatureComponents
File
Icon
InstallExecuteSequence
InstallUISequence
LaunchCondition
ListBox
Media
ModuleComponents
ModuleSignature
MsiFileHash
Property
RadioButton
RegLocator
Registry
RemoveFile
Shortcut
Signature
TextStyle
UIText
Upgrade
_Validation

Property	Value
InstallMode	Complete
UpgradeCode	{6113DD64-FD87-491A-9D05-0D1749DF7E18}
BannerBitmap	bannrbmp
WixUIRMOption	UseRM
IAgree	No
Manufacturer	TortoiseSVN
ProductCode	{3C5380EC-1D8B-45D2-B38A-4544DD0036D9}
ProductLanguage	1033
ProductName	TortoiseSVN 1.7.1.22161 (64 bit)
ProductVersion	1.7.22161
ALLUSERS	1
VSDUIANDADVERTISED	This advertised application will not be installed because it might
ARPCOMMENTS	Windows Shell Integration For SubVersion Source Control, v1.7.1.
ARPCONTACT	Stefan Kueng
ARPHELPLINK	http://tortoisesvn.net/
ARPPRODUCTICON	TSVNIcon
ARPNOMODIFY	1
ARPNOREPAIR	1
ARPURLINFOABOUT	http://tortoisesvn.net/
ButtonText_Back	< &Back
ButtonText_Browse	Br&owse
ButtonText_Cancel	Cancel
ButtonText_Exit	&Exit
ButtonText_Finish	&Finish
ButtonText_Ignore	&Ignore
ButtonText_Install	&Install
ButtonText_Next	&Next >

Tables: 41 Property - 56 rows No column is selected.

Рисунок 8.1 – Таблиця властивостей файла *.msi

```
public static string GetMsiProperty(string msiFile, string property)
{
    Record record = null;
    View view = null;
    Database database = null;
    try
    {
        // Create an Installer instance
        var classType = Type.GetTypeFromProgID("WindowsInstaller.Installer");
        var installerObj = Activator.CreateInstance(classType);
        var installer = installerObj as Installer;
        if (installer == null) return null;
        // Open the msi file for reading 0 - Read, 1 - Read/Write
        database = installer.OpenDatabase(msiFile, 0);

        // Fetch the requested property
        var sql = String.Format("SELECT `Value` FROM `Property` WHERE `Property` = '{0}'", property);
        view = database.OpenView(sql);
        view.Execute();

        record = view.Fetch(); // Read in the fetched record
        if (record != null)
            return record.StringData[1];
        return null;
    }
    finally
    {
        if (record != null)
        {
            Marshal.FinalReleaseComObject(record);
        }
        if (view != null)
        {
            view.Close();
            Marshal.FinalReleaseComObject(view);
        }
        if (database != null) Marshal.FinalReleaseComObject(database);
    }
}
```

Рисунок 8.2 – Приклад програмного вилучення інформації з БД файла msi

Процес установлення

Процес установлення складається з таких етапів [39]:

- збір інформації;
- виконання процесу встановлення;
- можливий відкат (у разі помилки або скасування установки користувачем).

Збір інформації (immediate mode)

На етапі збирання інформації Windows Installer збирає інструкції (або шляхом взаємодії з користувачем, або програмним шляхом), щоб встановити або видалити одну або кілька можливостей, що входять до продукту. Ці інструкції надалі формують на основі бази даних внутрішній сценарій, який детально описує наступний етап виконання.

Цей етап називають також безпосереднім режимом (immediate mode).

Виконання (deferred mode)

До початку цього етапу інсталятор генерує внутрішній сценарій, призначений для виконання без втручання користувача. Цей сценарій запускається інсталятором у привілейованому режимі (конкретно – під аккаунтом LocalSystem).

Привілейований режим потрібен через те, що інсталяція могла б бути запущена користувачем, що не володіє необхідними правами для зміни системних параметрів і файлів (хоча право встановити програму йому було надано).

Цей етап іноді називається відкладеним режимом (Deferred mode).

Відкат

Якщо будь-яка з дій, визначених у сценарії, закінчується невдачею, або установка в процесі скасовується користувачем, всі дії, виконані до цього місця, відкочуються, повертаючи систему в стан, який був до установки.

Відкат забезпечується наявністю для кожної дії, яка вносить зміни до системи, зворотньої до неї [дії]. Вводячи в пакет нестандартні дії, програміст також повинен створити зворотні до них дії для належного функціонування відкату.

Дії

Кожен етап установлення складається з послідовності дій (actions), записаної в базі даних. Діям привласнені номери, що визначають порядок їх виконання, а іноді – і умови, при яких дії виконуються або не виконуються.

Велика частина дій – це стандартні дії, характерні для типового процесу збору інформації та установки. Всі ці дії задокументовані. Окрім них, користувач може визначити і свої дії (custom actions).

Дії, визначені користувачем, можуть бути або написані однією із скриптових мов, вбудованих в операційну систему (JScript або VBScript, також і Eclips, побічна мова від C ++), або розміщуватися у спеціально створеній DLL (написаній такими мовами, як C, C ++ і т. д.). Файли з цими діями вміщуються всередину файла .msi і вилучаються звідти на початку запуску інсталяції.

Характеристика програмного продукту в системі

Існує **три важливих GUID**, які визначають Ваш інсталяційний пакет і які особливо важливі для оновлень продукту [40]:

Код пакету (Package Code) зазвичай має бути своїм для кожного .msi-файла.

Код продукту (Product Code) ідентифікує групу пакетів, з яких тільки один може бути встановлений в кожен момент часу. Наприклад, два пакети можуть мати один і той же код продукту, якщо вони є локалізаціями на різних мовах.

Код поновлення (Upgrade Code) ідентифікує групу пакетів, усі з яких можуть бути оновлені до одного нового продукту. Зазвичай, код поновлення збігається для всіх версій, які не можуть бути встановлені одночасно (side-by-side), тим самим дозволяючи попереднім версіям визначати наявність більш нових і відмовлятися від установки при цьому, а також нових версій визначати наявність старіших і автоматично оновлювати їх.

Версії продукту ProductVersion

Версія продукту наведена у вигляді типу даних, що містить допустимий рядок версії у вигляді:

xxxxx.xxxxx.xxxxx.xxxxx

де x – це цифра.

Максимальна версія продукту може бути:

255.65535.65535.65535.

major.minor.revision.build

1-е число (major) – привід зібрати з користувачів нову порцію грошей.

2-е число (minor) – номер доробки того, за що з користувачів вже взяли порції грошей, але з різних причин з передачі цього не зробили.

Всі інші числа служать для визначення набору відомих помилок, знайдених у користувача.

Таким чином, значення слів revision, buid, private слід шукати в системі контролю версій (зазвичай це номер гілки і номер зрізу).

Мажорне оновлення – Major upgrade [41]

Якщо Ви хочете надати користувачам можливість оновлюватися з однієї версії на іншу без обов'язкової деінсталяції (т. н. Major upgrade), то

кожна версія повинна мати власний код продукту, тому зазвичай новий випуск з новим номером версії є підставою для зміни коду продукту.

Такий вид оновлень передбачає повний цикл випуску продукту, включаючи маркетинг.

ProductVersion змінюється за межами R & D (Research and development – науково-дослідні, дослідно-конструкторські та технологічні роботи НДДКР).

Наприклад, маркетинговий відділ може прийняти рішення, що наступна версія буде '11' – для відповідності конкуренту, в той час, як поточна версія 2:

2.0.0.263 ➔11.0.0.285

Мінорне оновлення – Minor upgrade [42]

Мінорне оновлення можна використовувати в разі додавання нових функцій і компонентів до продукту, але не можна реорганізувати дерево можливостей компонент.

Мінорне оновлення змінює ProductVersion послідовно:

2.0.0.263 ➔2.1.0.270

Також мінорне оновлення змінює код версії продукту.

Мале оновлення – Small update [43]

Мале оновлення вносить зміни в один або кілька файлів програми, які є досить незначними, щоб виправдати зміни коду продукту. Мале оновлення також часто називають «швидкими інженерними виправленнями» (quick fix engineering – QFE). Мале оновлення не робить реорганізації дерева функцій-компонент.

Типове мале оновлення змінює тільки один або два файли або записи реєстру. У зв'язку з цим код пакета установлення також слід змінити.

Код продукту ніколи не змінюється з малим оновленням. Якщо необхідно провести відмінність між продуктами без зміни коду продукту, використовують мале оновлення.

ProductVersion змінюється: 2.0.0.263 ➔2.0.1.267

Створення GUID

GUID (Globally Unique Identifier) – статистично унікальний 128-бітний ідентифікатор.

Для створення GUID у VisualStudio заходимо в меню Tools-> CreateGUID.

Якщо необхідно створити GUID програмно, то використовуємо метод класу: Guid.NewGuid ().

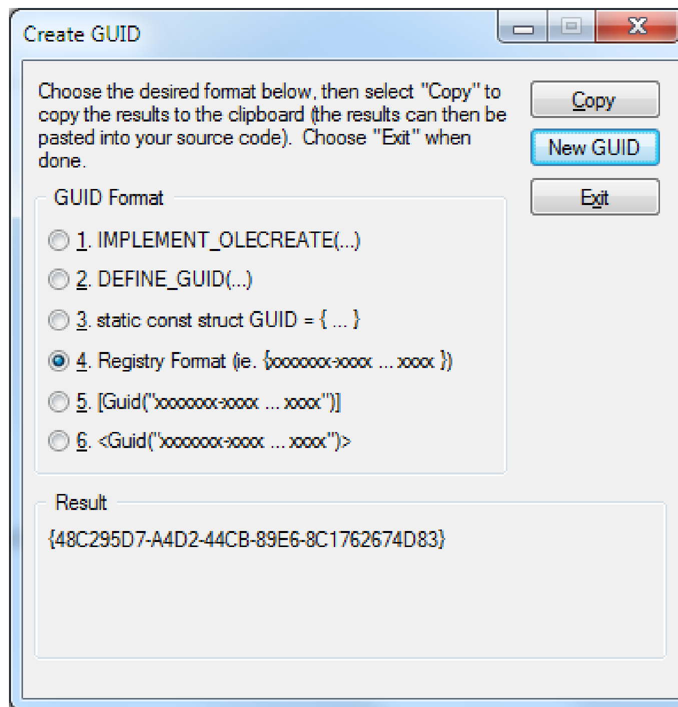


Рисунок 8.3 – Вікно створення GUID у VisualStudio

Створення інсталяційного пакета за допомогою WiX

The Windows Installer Xml (WiX) toolset – набір інструментів, що надають можливість створювати інсталяційні пакети Windows Installer (.MSI і .MSM) на основі XML-описів.

Windows Installer Xml був випущений компанією Microsoft у квітні 2004 року за ліцензією CPL і розміщений на сайті SourceForge.net (вихідні файли зараз «переїхали» на codeplex.com). WiX став першим проектом, який випустила компанія за відкритою ліцензією.

Склад пакета WIX:

candle

Компілятор / препроцесор – отримує об'єктні модулі за вихідними XML-документами.

light

Компонувальник – збирає готовий інсталяційний пакет з об'єктних модулів й інших ресурсів.

lit

Бібліотекар – дозволяє зібрати з декількох об'єктних модулів один бібліотечний файл.

dark

Декомпілятор – за інсталяційним пакетом (.MSI) отримує відповідний XML-документ.

tallow / heat

Інструмент, що дає можливість за каталогом файлів отримати їхній XML-опис, придатний для використання в WiX. У WiX 3.0 і вище подібну функціональність надає утиліта heat.

Встановлення WiX. Беремо з сайту дистрибутив (рисунок 8.4): http://wix.sourceforge.net/downloadv36.html

Рисунок 8.4 – Дистрибутив Windows Installer Xml

Генерація списку файлів через консоль (рисунок 8.5).

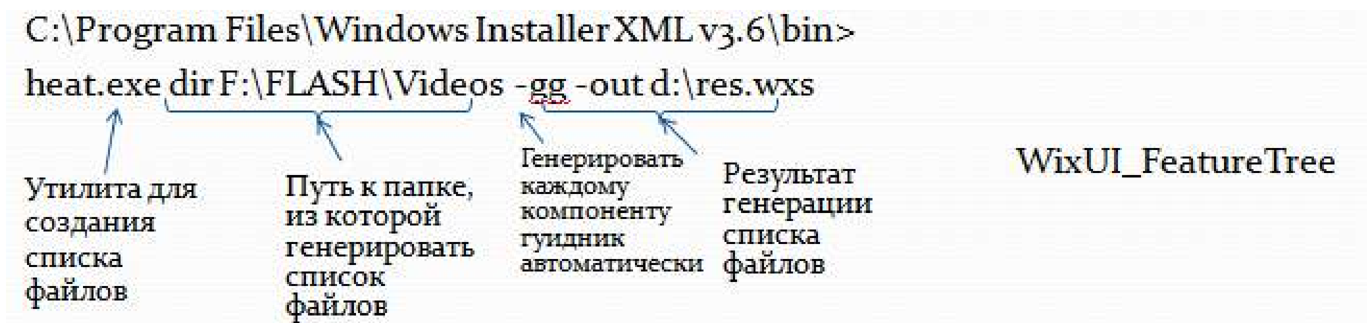


Рисунок 8.5 – Генерація списку файлів через консоль

На рисунку 8.6 зображено вміст файла res.wxs.

```
<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Fragment>
        <DirectoryRef Id="TARGETDIR">
            <Directory Id="dir11E4A76B9721F543C62ABA564D6730A9" Name="Videos" />
        </DirectoryRef>
    </Fragment>
    <Fragment>
        <DirectoryRef Id="dir11E4A76B9721F543C62ABA564D6730A9">
            <Component Id="cmp25716ECCE384A516FCA7B27F4F42C3C7" Guid="{281438A9-595B-428C-B63C-43514A2B5C04}">
                <File Id="fil58F1143815CEF28DAC91D29FEA6F5DFA" KeyPath="yes" Source="SourceDir\15082008.3gp" />
            </Component>
        </DirectoryRef>
    </Fragment>
    <Fragment>
        <DirectoryRef Id="dir11E4A76B9721F543C62ABA564D6730A9">
            <Component Id="cmp258BF176DCC25610C7D9D6ECC3C9A735" Guid="{33B80C74-3924-4B1A-9470-AE3A8B74DCB0}">
                <File Id="fil881D3E078137ECA1CE506FDADED29961" KeyPath="yes" Source="SourceDir\05072008.3gp" />
            </Component>
        </DirectoryRef>
    </Fragment>
    <Fragment>
        <DirectoryRef Id="dir11E4A76B9721F543C62ABA564D6730A9">
            <Component Id="cmp317389B210B191A0D9BBFE07E0016863" Guid="{8C238B47-5A03-40EE-B6C5-E8C5B9397196}">
                <File Id="fil324132CFA8F38694A11D446EC098FCC2" KeyPath="yes" Source="SourceDir\12092008.3gp" />
            </Component>
        </DirectoryRef>
    </Fragment>
    <Fragment>
        <DirectoryRef Id="dir11E4A76B9721F543C62ABA564D6730A9">
            <Component Id="cmpFF718EE6C6ECE3B19B085DFEC6FE3B0E" Guid="{624960C7-A56E-4345-AB56-C1AEC3EA59C4}">
                <File Id="fil500E67AF821DBEDB245D0AFC8DFB3E17" KeyPath="yes" Source="SourceDir\12092008(001).3gp" />
            </Component>
        </DirectoryRef>
    </Fragment>
</Wix>
```

Рисунок 8.6 – Перегляд остаточного файла res.wxs

У таблиці 8.1 наведено приклади використання мажорного, мінорного і малого оновлень.

У таблиці 8.2 подано коди установчого пакета, які змінюються за мажорним, мінорним і малим оновленнями відповідно.

Таблиця 8.1 – Мажорне чи мінорне оновлення, або ж мале оновлення [43]

Вимоги до оновлення	Використан-ня мажорного оновлення	Використан-ня мінорного оновлення	Використан-ня малого оновлення	Примітка
Змінити ім'я пакета .msi	Так	Ні	Ні	Файл за замовчуванням береться з властивості «Ім'я продукту», за умови, що файл .msi не стискується у виконуваний файл Setup.exe
Дозволити прикінцевим користувачам встановлювати більш ранні версії та останню версію на тій самій машині	Так	Ні	Ні	
Додати нову можливість	Так	У деяких випадках	У деяких випадках	Якщо нова можливість (subfeature) складається тільки з нових компонентів, Ви можете використовувати мажорне оновлення, мінорне оновлення або мале оновлення. Якщо новий subfeature складається з існуючих компонентів, необхідно використовувати мажорне (велике) оновлення

Продовження таблиці 8.1

Вимоги до оновлення	Використання мажорного оновлення	Використання мінорного оновлення	Використання малого оновлення	Примітка
Перемістити або видалити компонент з дерева продукту	Так	Ні	Ні	
Додати новий компонент до нової функції	Так	Так	Так	
Додати новий компонент до існуючої функції	Так	Так, if the version of Windows Installer is 2.0 or later	Так, if the version of Windows Installer is 2.0 or later	Установник Windows 1.x вимагає нових компонентів у пакеті оновлення, які будуть розміщені з новими можливостями для мінорних і малих оновлень; він також потребує спеціального оброблення командного рядка
Перемістити або видалити компонент з дерева продукту	Так	Ні	Ні	
Замінити код компонента на існуючий компонент	Так	Ні	Ні	
Змінити ключовий файл компонента	Так	Ні	Ні	

Закінчення таблиці 8.1

Вимоги до оновлення	Використання мажорного оновлення	Використання мінорного оновлення	Використання малого оновлення	Примітка
Додати, видалити або змінити будь-який з наступних компонентів: файли, ключі реєстру або ярлики	Так	Так	Так	Якщо файл, ключ реєстру або ярлик є більш, ніж одним компонентом, і компонент має дві або більше широкі можливості, необхідно використовувати велике (мажорне) оновлення

Таблиця 8.2 – Коди, які потрібно змінити для різних типів оновлень [37, 38]

	Код пакета	Версія продукту	Код продукту	Код оновлення
Мале оновлення	X			
Мінорне оновлення	X	X		
Мажорне оновлення	X	X	X	

• **Код пакета (Package Code)** – це «Summary Information Stream» (який є невід'ємною частиною файла MSI), код пакета ідентифікує конкретну базу даних. Код пакета не є властивістю установника Windows. Будь-які дві .msi бази даних з ідентичними кодами пакета повинні мати ідентичний зміст. Таким чином, Ви повинні змінювати код пакета для кожної збірки проекту (збирання, складання).

• **Версія продукту (ProductVersion)** – це властивість установника Windows, який містить версію продукту. Зверніть увагу на те, що інсталятор Windows використовує тільки перших три поля об'єкта ProductVersion (версія продукту) для порівнювання версій. Наприклад, для версії продукту 1.2.3.4, 4 ігнорується (зверніть увагу на те, що це правильно для порівняння значень ProductVersion, а не для версій файлів).

• **Код продукту (Product Code)** – це властивість установника Windows, який містить GUID продукту. Установник Windows обробляє два продукти з різними кодами GUIDs, які пов'язані між собою, навіть, якщо значення властивості ProductName однакові.

• **Код оновлення (Upgrade Code)** – це властивість установника Windows, який містить GUID, що являє собою сімейство продуктів. Коди оновлення повинні бути узгодженими у різних версіях і мовах сімейства родинних продуктів для цілей використання патчу (for patching purposes). Ви можете встановити UpgradeCode (код оновлення) для поновлення з метою модернізації.

Для будь-якого типу оновлення Ви повинні змінити різні комбінації коду пакета, версії продукту і коду продукту, щоб ідентифікувати продукт, який встановлюється. У таблиці 8.2 було наведено, коли кожен код має бути змінений для різних типів оновлень.

9 СЕРТИФІКАТИ ДОДАТКІВ. ЦЕНТРИ СЕРТИФІКАЦІЇ. ПІДПИС

Secure Socket Layer сертифікат

SSL – скорочення від Secure Socket Layer – це стандартна інтернет технологія безпеки, яка використовується для того, щоб забезпечити зашифроване з'єднання між веб-сервером (сайтом) і браузером. SSL-сертифікат дозволяє нам використовувати https-протокол. Це безпечне з'єднання, яке гарантує, що інформація, яка передається від Вашого браузера на сервер, залишається приватною [44].

Особливості SSL сертифікатів

SSL сертифікати – найпоширеніший на цей момент тип сертифікатів у Інтернеті. Найчастіше їх використовують в інтернет-магазинах, тобто на сайтах, де є функція замовлення і де клієнт вводить свої персональні дані. Для того, щоб ці дані в момент передачі з браузера на сервер неможливо було перехопити, використовується спеціальний протокол HTTPS, який шифрує всі дані, що передаються.

Code Signing сертифікати – це сертифікат, яким підписується програмне забезпечення або скрипти; він підтверджує автора програми і гарантує, що код не був змінений після того, як було накладено цифровий підпис. Також їх ще називають сертифікатами розробника.

Особливості сертифіката розробника [45]

Сертифікати розробника надають кілька можливостей:

1) це механізм цифрового підпису, який підтверджує, що програма, якою Ви користуєтесь, дійсно випущена тією чи іншою компанією, тобто гарантує справжність джерела;

2) гарантується цілісність вмісту, тобто, що з моменту підписання програмний продукт не був пошкоджений або змінений.

Центри сертифікації

Це організації, які володіють правом видачі цифрових сертифікатів. Вони проводять перевірку даних, що містяться в CSR, перед видачею сертифіката.

Перелік найпопулярніших центрів сертифікації

Comodo – працює з 1998 року, штаб-квартира в Jersey City, New Jersey, США.

Geotrust – заснований у 2001 р., у 2006 р. проданий Verisign, штаб-квартира Mountain View, California, США.

Symantec – колишній Verisign, до складу якого входить і Geotrust.

Thawte – заснований у 1995 р., проданий Verisign у 1999 р.

Trustwave – працює з 1995 р., штаб-квартира Chicago, Illinois, США.

Найпростіший і безкоштовний спосіб – це використовувати так званий самопідписний сертифікат (self-signed), який можна згенерувати прямо на веб-сервері.

У всіх найпопулярніших панелях управління хостингом (Cpanel, ISPmanager, Directadmin) ця можливість доступна за замовчуванням, тому технічні процеси створення сертифіката можна не брати до уваги.

«Плюс» самопідписного сертифіката – це його ціна, точніше, її відсутність, оскільки Ви не платите ні копійки за такий сертифікат. А ось з «мінусів» – це те, що на такий сертифікат усі браузери будуть видавати помилку з попередженням, що сайт не підтверджений.

Різниця між самопідписним безкоштовним і платними сертифікатами, виданими центром сертифікації, і полягає в тому, що дані в сертифікаті перевірені центром сертифікації, і під час використання такого сертифіката на сайті або при запуску програми користувач не побачить помилку про справжність сертифіката.

Починаючи з Windows XP SP2, при установленні програмного забезпечення або драйверів без такого цифрового підпису Ви отримаєте попередження, що у цієї програми «Невідомий видавець» і запускати її не рекомендується (рисунок 9.1).

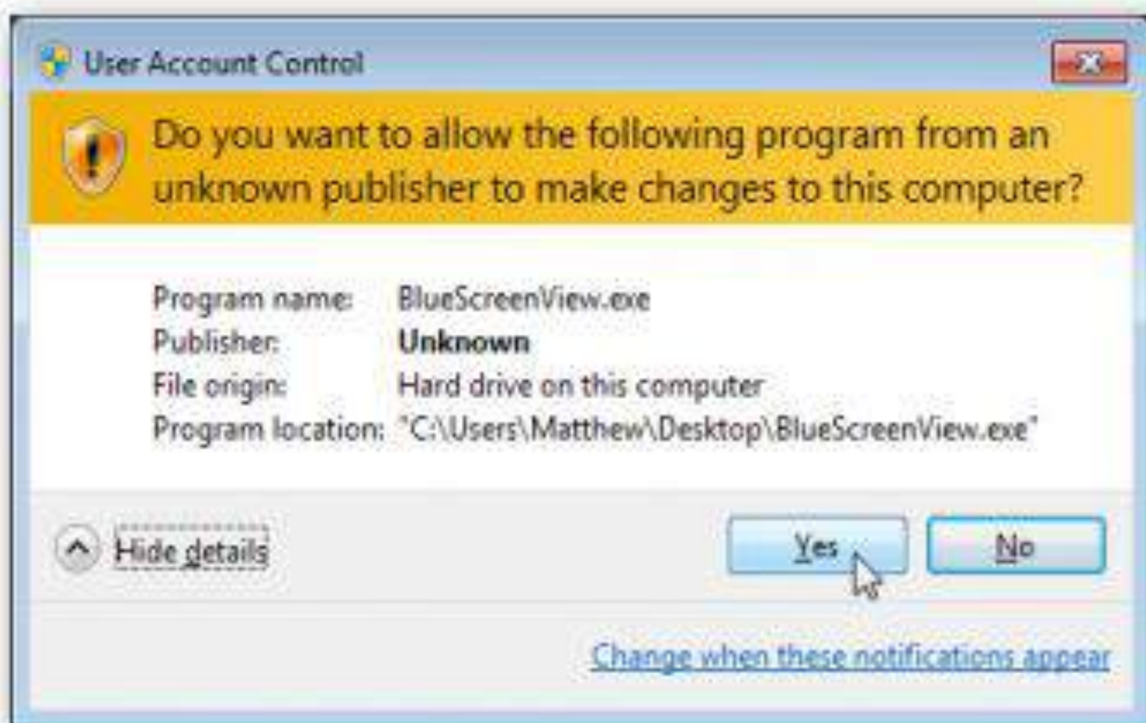


Рисунок 9.1 – Скріншот вікна з повідомленням, яке попереджає про можливість використання програми «Невідомого видавця»

Процес підпису коду (рисунок 9.2) [46].

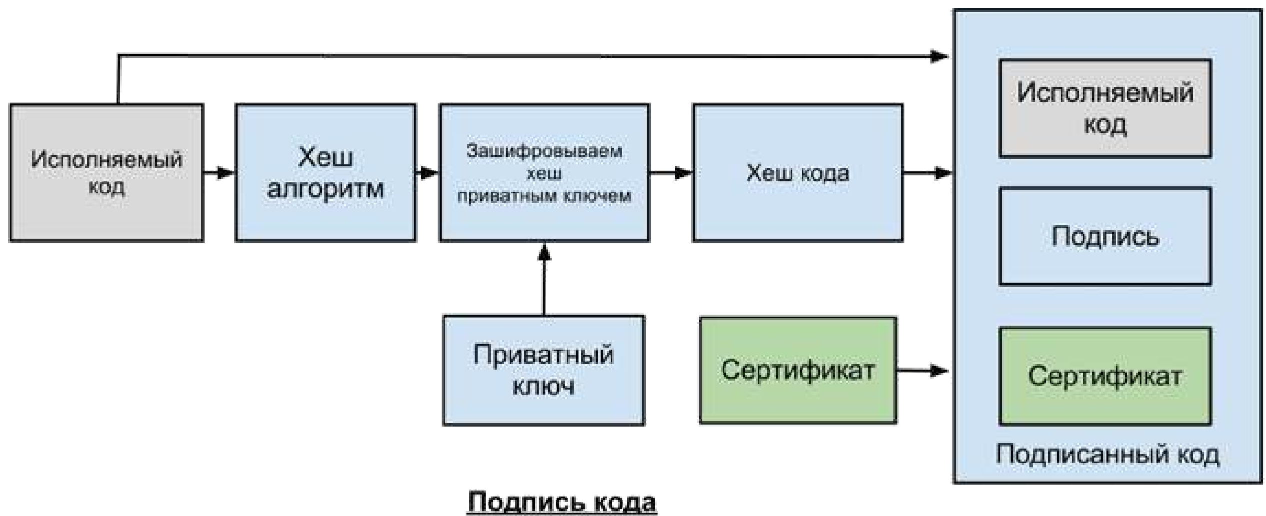


Рисунок 9.2 – Схема процесу підпису коду

1. Видавець (розробник) запитує Code Signing сертифікат у центрі сертифікації.

2. Скориставшись SIGNCODE.EXE або іншою утилітою для підпису коду, видавець створює хеш-код з використанням алгоритмів MD5 або SHA.

3. Видавець кодує хеш за допомогою приватного ключа.

4. Видавець створює пакет, який містить: код, зашифрований хеш і сертифікат видавця.

Процес перевірки підписаного коду (рисунок 9.3).

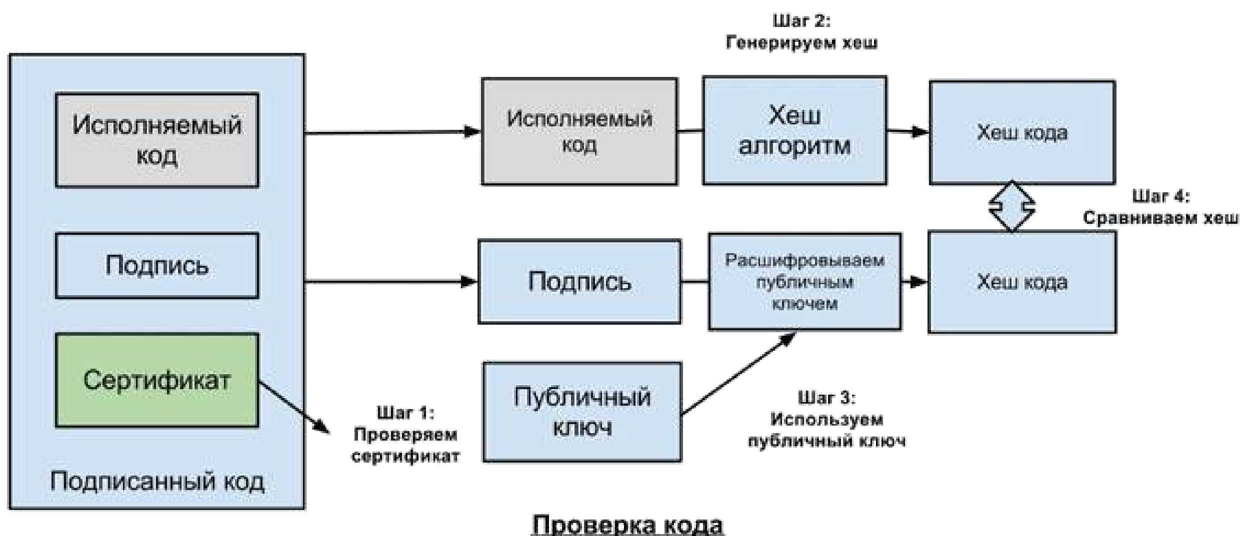


Рисунок 9.3 – Схема процесу перевірки підписаного коду

1. Користувач завантажує або встановлює підписане ПЗ, і платформа або система користувача перевіряє сертифікат видавця, який підписаний кореневим приватним ключем центру сертифікації.

2. Система запускає код, використовуючи той самий алгоритм створення хешу, що і видавець, і створює новий хеш.

3. Використовуючи публічний ключ видавця, який міститься у сертифікаті, система розшифровує зашифрований хеш.

4. Система порівнює між собою два хеші.

Timestamp, або тимчасова мітка

Timestamp, або тимчасова мітка, використовується для вказівки часу, коли цифровий підпис було зроблено. Якщо ця позначка присутня, то додаток, який перевіряє підпис, перевірить, чи був сертифікат, пов'язаний з підписом, дійсним на момент підпису. Якщо ж такої мітки немає і термін сертифіката вже закінчився, то підпис буде вважатися недійсним.

Приклад:

Сертифікат дійсний з 01.01. 2008.

Сертифікат дійсний до 31.12.2010.

Підпис зроблено – 04.07.2009.

Підпис перевірено – 30.04.2012.

Із тимчасовою міткою (timestamp) підпис пройде перевірку, оскільки на момент підпису сертифікат був дійсний. Без такої мітки сертифікат не пройде перевірки, оскільки на момент перевірки у сертифіката вже закінчився термін.

Тобто ця *позначка (мітка) дозволяє використовувати підписаний код, навіть після закінчення терміну сертифіката.*

Створення самопідписного сертифіката для підпису коду:

1) завантажуємо Windows SDK, встановлюємо його і «слідуємо» за ним в його папку;

2) makecert.exe -cy end -pe -r -n "CN= ***Ім'я Вашого Сертифіката***" -sky Signature –sv ***path_to***\key.pvk ***path_to***\key.cer;

3) pvk2pfx.exe -pvk ***path_to***\key.pvk -spc ***path_to***\key.cer -pfx ***path_to***\key.pfx;

4) імпортуємо ключ key.pfx у Ваше сховище сертифікатів або на Вашу смарткарту.

http://msdn.microsoft.com/en-en/library/aa386968%28v=vs.85%29.aspx

Створення самопідписного сертифіката для підпису коду (Visual Studio): заходимо у властивості проекту та створюємо тестовий сертифікат (рисунок 9.4).

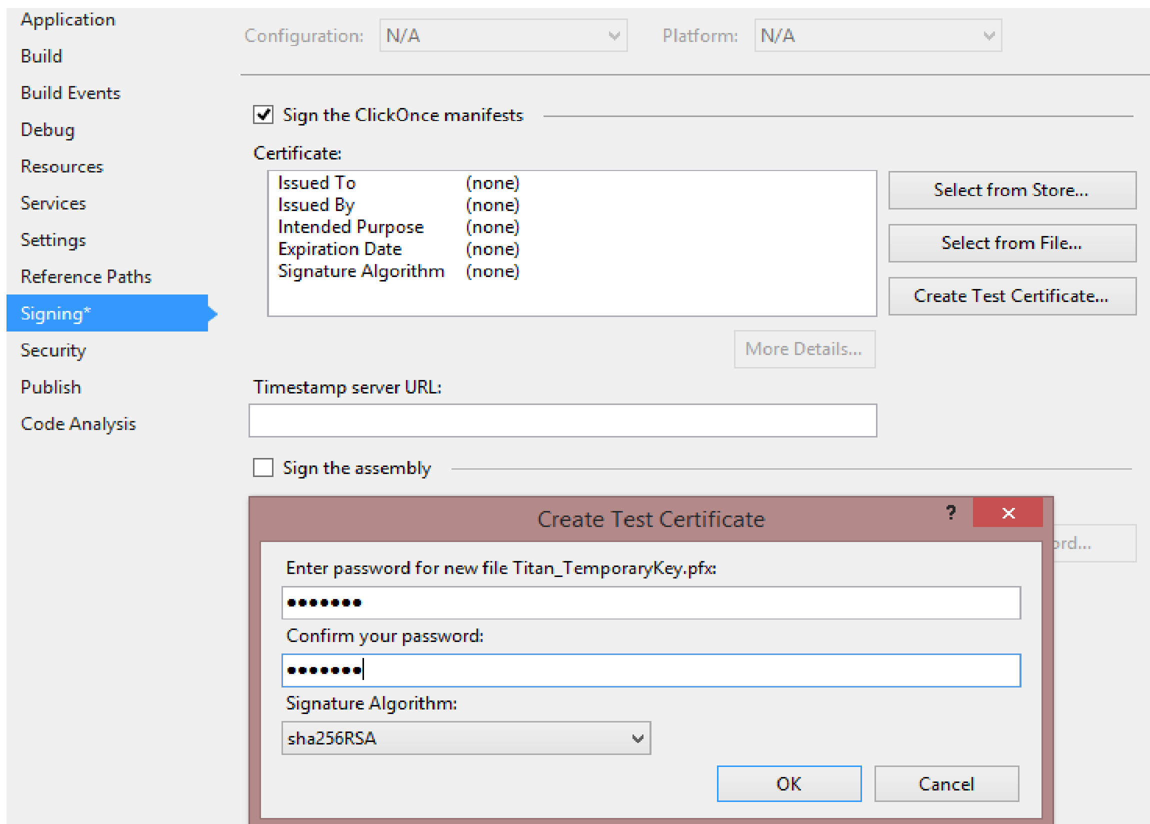


Рисунок 9.4 – Створення самопідписного сертифіката
для підпису коду (Visual Studio)

Підпис додатка за допомогою SignTools

Sign Tool – це програма командного рядка, яка виконує цифровий підпис файлів, перевіряє підписи файлів або створює штампи часу для файлів.

signtool [command] [options] [file_name | ...]

http://msdn.microsoft.com/ru-ru/library/8s9b9yaz%28v=vs.110%29.aspx

Приклад підписання файла

Наступна команда дозволяє підписати файл цифровим підписом за допомогою сертифіката, що зберігається у PFX-файлі, захищеному паролем:

signtool sign /f MyCert.pfx /p MyPassword MyFile.exe

Приклад підписання файла з таймштампом:

Наступна команда підписує цифровим підписом і ставить штамп часу на файлі.

Сертифікат, який використовується для підписання файла, зберігається у файлі PFX:

signtool sign /f MyCert.pfx /t http://timestamp.verisign.com/scripts/timstamp.dll MyFile.exe

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Chacon, Scott. Pro Git [Internet source] / Scott Chacon. – It is licensed under the Creative Commons Attribution-Non Commercial-Share Alike 3.0 license. – URL: http://tinyurl.com/amazonprogit – 29.10.2017.

2. Ben Collins-Sussman. Version Control with Subversion. For Subversion 1.7 (Compiled from r5206) [Internet source] / Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato. – It is licensed under the Creative Commons Attribution License. – USA, 2011. – 433 p. – URL: http://creativecommons.org/licenses/by/2.0/ – 29.10.2017.

3. Бен Коллинз-Сассмэн. Идеальная IT-компания. Как из гиков собрать команду программистов [Текст] / Бен Коллинз-Сассмэн, Брайан Фитцпатрик. – ООО Изд-во «Питер», 2014. – 230 с.

4. Mercurial: The Definitive Guide by Bryan O'Sullivan [Internet source]. – URL: http://hgbook.red-bean.com/read/ – 29.10.2017.

5. [Internet source]. – URL: https://hgbook.bacher09.org/html/index.html – 29.10.2017.

6. [Internet source]. – URL: https://dou.ua/lenta/articles/mercurial-step-by-step-dvcs-intro/ – 29.10.2017.

7. Ben, Lynn. Волшебство Git (Руководство выпущено под GNU General Public License 3-й версии. Исходный текст находится в хранилище Git и может быть получен командой: $ git clone git://repo.or.cz/gitmagic.git #) [Internet source] / Lynn Ben. – URL: http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/index.html

8. Шторгина, Е. С. Системы контроля версий. Сетевые технологии. Технологии Интернет : Лекция 2 [Internet source] / Е. С. Шторгина. – 82 с. – URL: http://its.kpi.ua/subjects/34/Documents/%D0%9B%D0%B5%D0%BA_%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9.pdf – 29.10.2017.

9. [Internet source]. – URL: http://tortoisehg.bitbucket.org/screenshots.html#mq – 29.10.2017.

10. [Internet source]. – URL: http://tortoisehg.bitbucket.org/ – 29.10.2017.

11. [Internet source]. – URL: https://habrahabr.ru/post/154255/ – 29.10.2017.

12. [Internet source]. – URL: http://rus-linux.net/MyLDP/BOOKS/Architecture-Open-Source-Applications/Vol-2/git-02.html – 29.10.2017.

13. Иртегов, Д. В. Лекции по инструментам управления конфигурацией [Internet source] / Д. В. Иртегов. – URL: http://parallels.nsu.ru/~fat/subversion.ppt – 29.10.2017.

14. [Internet source]. – URL: https://books.google.com.ua/books?id=kZ5nAAAAQBAJ&pg=PA75&lpg=PA75&dq=%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5+%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85+%D0%B2+%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B5+%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F+%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9&source=bl&ots=iCnBVHOYMg&sig=u9TdcTMnkUhnl6qmL6BOSdt9CHE&hl=ru&sa=X&ved=0ahUKEwjyvtbC_t3QAhXJhiwKHS0DBiYQ6AEIKzAD#v=onepage&q=%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85%20%D0%B2%20%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B5%20%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F%20%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9&f=false – 29.10.2017.

15. [Internet source]. – URL: https://ru.wikipedia.org/wiki/Subversion – 29.10.2017.

16. [Internet source]. – URL: https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8 – 29.10.2017.

17. Управление версиями в Subversion [Internet source]. – URL: http://svnbook.red-bean.com/ – 29.10.2017.

18. [Internet source]. – URL: http://www.inteks.ru/PM/ – 29.10.2017.

19. [Internet source]. – URL: https://confluence.atlassian.com/bamboo/mercurial-289277014.html – 29.10.2017.

20. [Internet source]. – URL: https://developer.mozilla.org/en-US/docs/Mercurial/Using_Mercuria – 29.10.2017.

21. Обзор систем контроля версий [Internet source]. – URL: http://all-ht.ru/inf/prog/p_0_1.html – 29.10.2017.

22. Рейтинг систем контроля версий [Internet source]. – URL: http://tagline.ru/version-control-systems-rating/ – 29.10.2017.

23. Сравнение систем контроля версий [Internet source]. – URL: http://mitra.ru/events/news/website_3.html – 29.10.2017.

24. Ройс, У. Управление проектами по созданию программного обеспечения. Унифицированный подход [Текст] / У. Ройс. – Лори, 2002. – 431 с.

25. An American National Standart. ANSI/PMI 99-001-2008. A Guide to the Project Management Body of Knowledge. – USA : Project Management Institute, Inc., 2005. – 506 p.

26. Лаврищева, Е. М. Методы и средства инженерии программного обеспечения : учеб. пособ. [Текст] / Е. М. Лаврищева, В. А. Петрухин. – М. : МФТИ, 2006. – 304 с.

27. Бек, К. Экстремальное программирование [Текст] / К. Бек. – Спб. : Изд-во «Питер», 2003. – 224 с.

28. Брукс, Ф. Мифический человеко-месяц, или как создаются программные системы : пер. с англ. [Текст] / Ф. Брукс. – Символ-Плюс, 2007. – 304 с.

29. Липаев, В. В. Программная инженерия. Методологические основы : учеб. пособ. [Текст] / В.В. Липаев. – М. : Теис, 2006. – 608 с.

30. Йордан, Э. Путь камикадзе. Как разработчику программного обеспечения выжить в безнадежном проекте [Текст] / Э. Йордан. – М. : Лори, 2003. – 256 с.

31. Терехов, А. Н. Технология программирования : учеб. пособ. [Internet source] / А. Н. Терехов. – М. : Интернет-Ун-т Информ. Технологий, 2006. – 152 с. – URL: http://www.math.spbu.ru/user/ant/all_articles/057_Terekhov_Technology_programming.pdf – 29.10.2017.

32. Савин, Р. Тестирование Дот Ком, или Пособие по жёсткому обращению с багами в интернет-стартапах [Текст] / Р. Савин. – М. : Дело, 2007. – 312 с.

33. Куликов, С. Тестирование программного обеспечения. Базовый курс [Internet source] / С. Куликов. – EPAM Systems, 2016. – 289 с. – URL: http://svyatoslav.biz/software_testing_book/ – 29.10.2017.

34. Орлик, С. Программная инженерия. Тестирование программного обеспечения [Internet source] / С. Орлик. – 2004 – 2005. – 16 с. – URL: https://software-testing.ru/files/se/3-4-software_engineering_testing.pdf – 29.10.2017.

35. Симан, Марк. Внедрение зависимостей [Текст] / Марк Симан. – Спб. : Изд-во «Питер», 2013. – 624 с.

36. Орехов, А. А. Технологии и инструментальные средства тестирования программных систем : учеб. пособие / А. А. Орехов, И. А. Слизовская, Д. А. Кочкарь. – Харьков : Харьков. авиац. ин-т, 2009. – 62 с.

37. [Internet source]. – URL: https://msdn.microsoft.com/ru-ru/library/stxxt3fd(v=vs.90).aspx – 29.10.2017.

38. Нумерация версий [Internet source]. – URL: https://habrahabr.ru/post/119400/ – 29.10.2017.

39. Microsoft Corporation. Анализ требований и создание архитектуры решений на основе Microsoft .NET. Учебный курс MCSD / Пер. с англ. – М. : Издательско-торговый дом «Русская Редакция», 2004. – 416 с.

40. Guckenheimer, S. Software Engineering with Microsoft Visual Studio Team System [Text] / S. Guckenheimer, J.Perez. – Addison-Wesley Professional, 2006. – 255 p.

41. [Internet source]. – URL:http://wixtoolset.org/documentation/manual/v3/howtos/updates/major_upgrade.html – 29.10.2017.

42. [Internet source]. – URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa370037(v=vs.85).aspx – 29.10.2017.

43. [Internet source]. – URL: https://www.safaribooksonline.com/library/view/wix-36-a/9781782160427/ch13s04.html – 29.10.2017.

44. [Internet source]. – URL: http://helpnet.flexerasoftware.com/installshield22helplib/helplibrary/MajorMinorSmall.htm – 2015 – 29.10.2017.

45. Ericsson, A. B. Secure Socket Layer [Internet source] / A. B. Ericsson. – ERLANG, September 25, 2017. – 38 p. – URL: http://erlang.org/doc/apps/ssl/ssl.pdf – 29.10.2017.

46. Липаев, В. В. Сертификация программных средств [Текст]: учеб. пособ. / В. В. Липаев. – М. : СИНТЕГ, 2010. – 348 с.

Навчальне видання

Кузнецова Юлія Анатоліївна
Туркін Ігор Борисович

ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Редактор Є. О. Александрова

Зв. план, 2017
Підписано до видання 20.11.2017
Ум. друк. арк. 8,3. Обл.-вид. арк. 9,37. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
“Харківський авіаційний інститут”
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб’єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30 03.2001