Лекція 6

Конфігураційне управління

Анотація: Поняття конфігураційного управління. Управління версіями. Поняття "гілки" проекту. Управління збірками. Засоби версійного контролю. Одиниці конфігураційного управління. Поняття baseline.

Проблема

Всім відомо, що на великих промислових підприємствах, в магазинах, книжкових видавництвах і ін. існують склади. Основне завдання складу – забезпечити зберігання і доступ до матеріальних активів: товарів, виробів, книг і ін. Тобто, різних матеріальних активів стає так багато, що необхідно мати спеціальну службу для їх обліку. Виявляється, що не достатньо складати, наприклад, всі наявні в книговидавництві книги в спеціальну кімнату і видавати їх власникам тиражу, коли вони за ними прийдуть. Книг виявляється дуже багато, а процедура видачі тиражу є не зовсім тривіальною. Потрібно, щоб власник приніс велику кількість супровідних документів, і всі вони повинні бути перевірені перед видаванням книг. А на самому складі необхідно підтримувати порядок, щоб було можливо швидко знайти потрібні книги (як показує досвід, вони можуть там досить довго знаходитися). Ще складніша процедура роботи з книгами в бібліотеці – там додаються ще каталоги, розподілені книжкові сховища, необхідність підтримувати хороший стан книг, а також контролювати повернення їх в бібліотеку після певного терміну. Аналогічним чином працює склад на будь-якому заводі, фабриці і так далі.

Розглянемо тепер проект розробки програмного забезпечення. Що в ньому є аналогом матеріальних активів на звичайному виробництві? Безумовно, не столи і стільці, якими користуються розробники. І навіть не комп'ютери, запчастини до них і інше устаткування. Обліку і контролю вимагають **файли** проекту. У програмному проекті їх дуже багато – сотні і тисячі, навіть для відносно невеликих проектів. Адже створити новий файл дуже легко. Багато технологій програмування підтримують стиль, коли, наприклад, для кожного класу створюється свій окремий файл.

Файл – це віртуальна інформаційна одиниця. У чому головна відмінність файлу від матеріальних одиниць обліку? У тому, що у файлу може бути **версія**, і не одна, і породити ці версії дуже легко – досить скопіювати даний файл в інше місце на диску. Тоді як матеріальні предмети існують на складі самі по собі, і для них немає поняття версії. Так, може бути декілька однотипних предметів, різних заготовок виробу різного ступеня готовності. Але все це не те... А версія файлу – це дуже непростий об'єкт. Чим одна версія відрізняється від іншої? Декількома строчками тексту або повністю оновленим змістом? І яка з двох і більшої кількості версій є головнішою або кращою? До цього додається ще і те,

що багато робочих продуктів можуть складатися з набору файлів, і кожен з них може мати декілька версій. Як зібрати коректну версію продукту?

Інколи в програмному проекті починають відбуватися містичні і загадкові події.

- Програма, яку ретельно тестували, на показових випробуваннях не працює
- Функціональність, про яку довго просив замовник і яка була, нарешті, додана у продукт, і у новій версії урочисто надіслана замовникові, таємничим чином зникла з продукту.
- На комп'ютері розробника програма працює, а у замовника – ні...

Пояснення є досить простим – вся справа у версіях файлів. Там, де все добре, присутні файли однієї версії, а там, де все погано – іншої. Але біда в тому, що "версія всього продукту" – це абстрактне поняття. В дійсності є версії окремих файлів. Один або декілька файлів в постачанні продукту мають не ту версію – все, справи погані. Необхідно управляти версіями файлів, інакше подібна містика може стати величезною проблемою.

Така «містика» серйозно гальмує внутрішню роботу. То розробники і тестери працюють з різними версіями системи, то підсумкове збирання системи вимагає спеціальних зусиль всього колективу. Більш того, можливі неприємності на рівні управління. Різні курйозні ситуації, коли заявлена функціональність відсутня або не працює (знову не ті файли надіслали!), можуть сильно погіршувати відносини із замовником. Незадоволений замовник може зажадати навіть грошової компенсації за те, що помилки, що виникають, дуже довго виправляються. А як може бути інакше, коли розробники не можуть відтворити і виправити помилку, оскільки точно не знають, з яких саме початкових текстів була зібрана дана версія!

Отже, стає зрозуміло, що в програмних проектах необхідна спеціальна діяльність з підтримки файлових активів проекту у порядку. Ця діяльність і називається **конфігураційним управлінням**.

Виділимо два основні завдання в конфігураційному управлінні – **управління версіями** і **управління збірками**. Перше відповідає за управління версіями файлів і виконується в проекті на основі спеціальних програмних пакетів – **засобів версійного контролю**. Існує велика кількість таких засобів – Microsoft Visual Source Safe, IBM ClearCase, CVN, Subversion і ін. **Управління збірками** – це автоматизований процес трансформації початкових текстів ПЗ в пакет виконуваних модулів, що враховує численні налагоджування проекту, налагоджування компіляції, і інтегроване у процес автоматичне тестування. Ця процедура є могутнім засобом інтеграції проекту, основою ітеративної розробки.

Одиниці конфігураційного управління

Які об’єкти підлягають контролю у межах конфігураційного управління? Це будь-які файли, які є в проекті? Ні, не будь-які, а лише ті, які змінюються. Наприклад, файли з придбаним для проекту ПЗ повинні спокійно покоїтися на CD-дисках або в локальній мережі. Книги, документи із зовнішніми стандартами, які використовуються в проекті (наприклад, в телекомунікаціях дуже багато різних стандартів на мережеві інтерфейси і ін.) також повинні просто зберігатися там, де кожен охочий їх може взяти. Як правило, такої інформації в проекті небагато, але, зрозуміло, вона повинна бути в порядку. Заради цього спеціальний вид діяльності в проекті не потрібний.

Отже, конфігураційне управління має справу із змінними в процесі продуктами, що складаються з наборів файлів. Такі продукти прийнято називати **одиницями конфігураційного управління** (configuration management items). Ось приклади:

1. призначена для користувача документація;
2. проектна документація;
3. початкові тексти ПЗ;
4. пакети тестів;
5. інсталяційні пакети ПЗ;
6. тестові звіти.

У кожної одиниці конфігураційного управління повинні бути присутні:

1. Структура – набір файлів. Наприклад, призначена для користувача документація в .html повинна включати індекс-файл і набір html-файлів, а також набір винесених рисунків (.gif або .jpeg-файли). Ця структура повинна бути добре визначена і відслідковуватися підчас конфігураційного управління: жоден файл не має бути втраченим і повинен існувати, не мусять мати однакову версію, але мусять мати коректні посилання один на одного і так далі.

2. Відповідальна особа і, можливо, група тих, хто їх розробляє, а також мати ширшу і менш відповідальну групу тих, хто користується цією інформацією. Наприклад, визначеним програмним компонентом можуть в проекті користуватися багато розробників, але відповідати за його розробку, виправлення помилок і ін. повинен хтось один.

3. Практика конфігураційного управління – хто і в якому режимі, а також в яке місце викладає нову версію елементу конфігураційного управління в засіб управління версіями, правила іменування і коментування елементу в цій версії, подальші маніпуляції там з ним і ін. Більш високорівневі правила це такі, які зв'язані, наприклад, з правилами

зміни тестів і тестових пакетів на разі зміни коду. Проте, десь тут лежить вододіл між конфігураційним управлінням і іншими видами діяльності в проекті.

4. Автоматична процедура контролю цілісності елементу – як приклад, збірка для початкових текстів програм. Є не у всіх елементів, наприклад, може не бути у документації, тестових пакетів.

Елементи конфігураційного управління можуть утворювати ієрархію. Приклад наведено на Рис. 6.1.

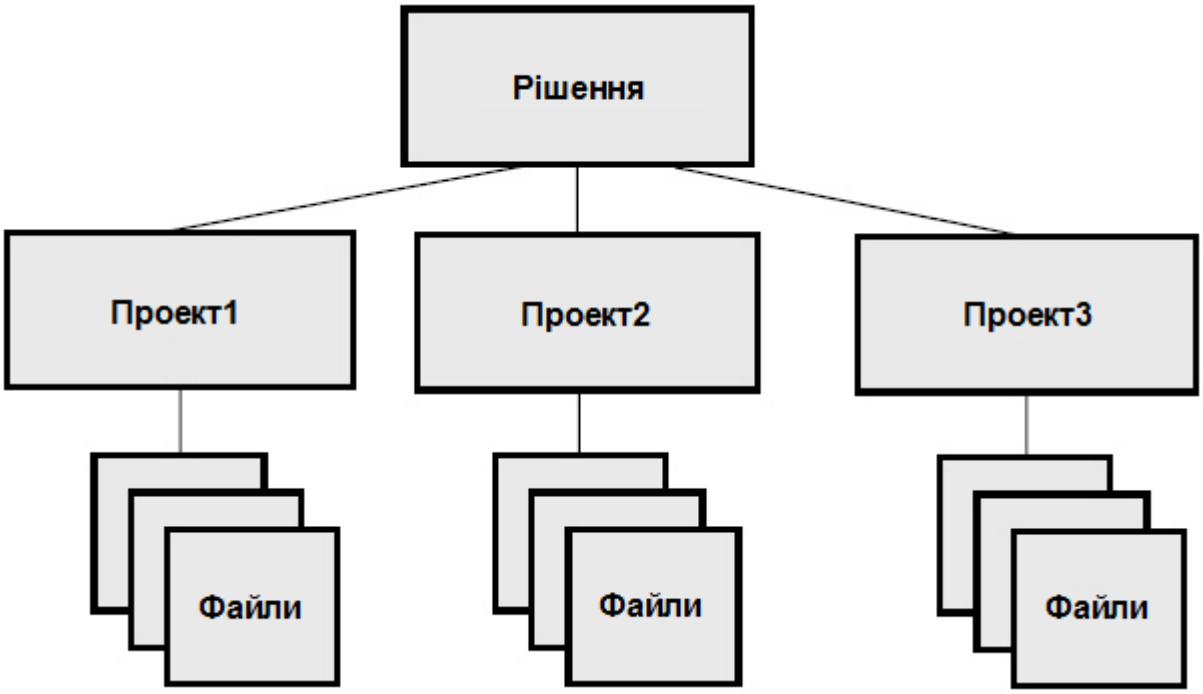

Рис. 6.1. Ієрархія елементів конфігураційного управління

Управління версіями

Управління версіями файлів. Оскільки програмісти мають справу з величезною кількістю файлів, багато файлів водночас можуть бути необхідні декільком людям і важливо, щоб всі вони постійно складали єдину, як мінімум, версію продукту, що компілюється, необхідно, щоб була налагоджена робота з файлами з початковим кодом. Також може бути налагоджена робота і з іншими типами файлів. У цій ситуації файли виявляються самими молодшими (за ієрархією включення) елементами конфігураційного управління.

Управління версіями складених конфігураційних об'єктів. Поняття "гілки" проекту. Одночасно може існувати декілька версій системи – і в сенсі для різних замовників і ін. (так би мовити, у великому, справжньому сенсі), і в сенсі одного проекту, одного замовника, але як різний набір початкових текстів. І в тому і в іншому випадку в засобі

управління версіями утворюються різні **гілки**. Зупинимося трохи докладніше на другому випадку.

Кожна гілка містить повний образ початкового коду і інших артефактів, що знаходяться в системі контролю версій. Кожна гілка може розвиватися незалежно, а може в певних точках інтегруватися з іншими гілками. В процесі інтеграції зміни, які були виконані в одній з гілок, напівавтоматично переносяться в іншу. Як приклад можна розглянути таку структуру розділення проекту на гілки.

· ***V1.0*** – гілка, що відповідає випущеному релізу. Внесення змін до таких гілок заборонені і вони зберігають образ коду системи на момент випуску релізу.

· ***Fix V1.0.1*** – гілка, що відповідає випущеному пакету виправлень до певної версії. Подібні гілки відгалужуються від початкової версії, а не від основної гілки і заморожуються відразу після виходу пакету виправлень.

· ***Upcoming (V1.1)*** – гілка, що відповідає релізу, який підготовлений до випуску і знаходиться у стадії стабілізації. Для таких гілок, як правило, діють строгіші правила і робота в них ведеться формалізованіше.

· ***Mainline*** – гілка, що відповідає основному напряму розвитку проекту. У міру дозрівання саме від цієї гілки відходять гілки підготовлюваних релізів.

· ***WCF Experiment*** – гілка, яка створена для перевірки деякого технічного рішення, переходу на нову технологію, або внесення великого пакету змін, що потенційно порушують працездатність коду на тривалий час. Такі гілки, як правило, стають доступними лише для певного круга розробників і убиваються по завершенню робіт після інтеграції з основною гілкою.

Управління збірками

Отже, чому ж процедура компіляції і створення .exe, .dll файлів за файлами кодів проекту – така важлива процедура? Тому що вона багато разів в день виконується кожним розробником на його власному комп'ютері, з його власною версією проекту. При цьому відрізняється:

· набір підпроектів, що збираються розробником; він може збирати не весь проект, а лише якусь його частина; інша частина або їм не використовується зовсім, або не збирається заново дуже давно, а по факту вона давно змінилася;

· відрізняються параметри компіляції.

При цьому якщо не збирати регулярно підсумкову версію проекту, то загальна інтеграція може виявити багато різних проблем:

· невідповідність один одному різних частин проекту;

· наявність специфічних помилок, що виникли через те, що окремі проекти розроблялися без урахування параметрів компіляції (зокрема, перехід в Visual Studio з debug на release версію часто супроводжується появою численних проблем).

У зв'язку з цим процедуру збірки проекту часто автоматизують, тобто виконують не з середовища розробки, а із спеціального скрипта – build-скрипта. Цей скрипт використовується тоді, коли розробникові потрібна повна збірка всього проекту. А також він використовується в процедурі безперервної інтеграції (continues integration) – тобто регулярній збірці всього проекту (як правило – щоночі). Як правило, процедура безперервної інтеграції включає і регресійне тестування, і часто – створення інсталяційних пакетів. Загальна схема автоматизованої збірки представлена на Рис. 6.2.

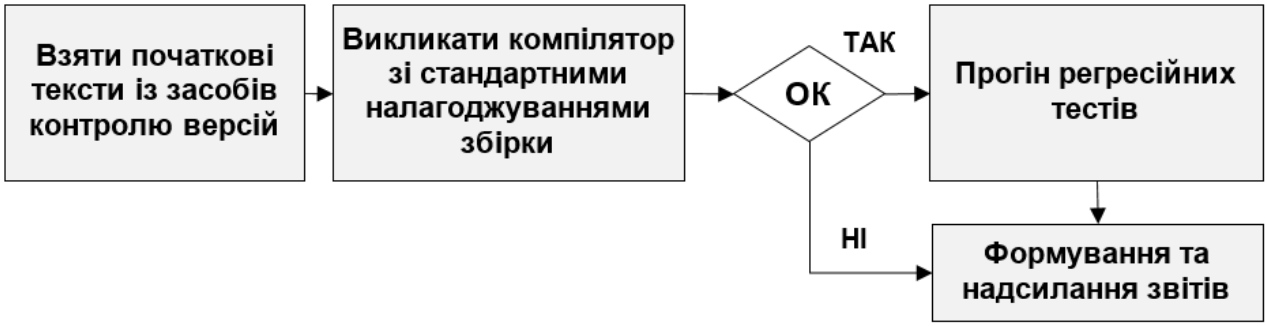

Рис. 6.2 Загальна схема автоматизованої збірки

Тестувальники повинні тестувати, за можливістю, підсумкову і цілісну версію продукту, так що результати регулярної збірки виявляються дуже затребувані. Крім того, наявність базової, актуальної, цілісної версії продукту дозволяє організувати розробку в ітеративно-інкрементальному стилі, тобто на основі внесення змін. Такий стиль розробки має назву baseline-метод.

Поняття baseline

Baseline – це базова, остання цілісна версія деякого продукту розробки, наприклад, документації, програмного коду і так далі. Мається на увазі, що розробка йде не суцільним потоком, а з фіксацією проміжних результатів у вигляді поточної офіційної версії активу, що розробляється. Ухвалення такої версії супроводжується додатковими діями відносно оформлення, згладжування, тестування, включення лише закінчених фрагментів і так далі. Цей результат можна подивитися, віддати тестувальникам, передати замовникові і так далі. Baseline є хорошим засобом синхронізації групової роботи.

Baseline може бути зовсім простою – гілкою в засобі управління версіями, де розробники зберігають поточну версію своїх початкових код. Єдиною вимогою в цьому випадку може бути лише компіляція проекту в цілому. Але підтримка baseline може бути формально складною процедурою, як показано на Рис. 6.3.

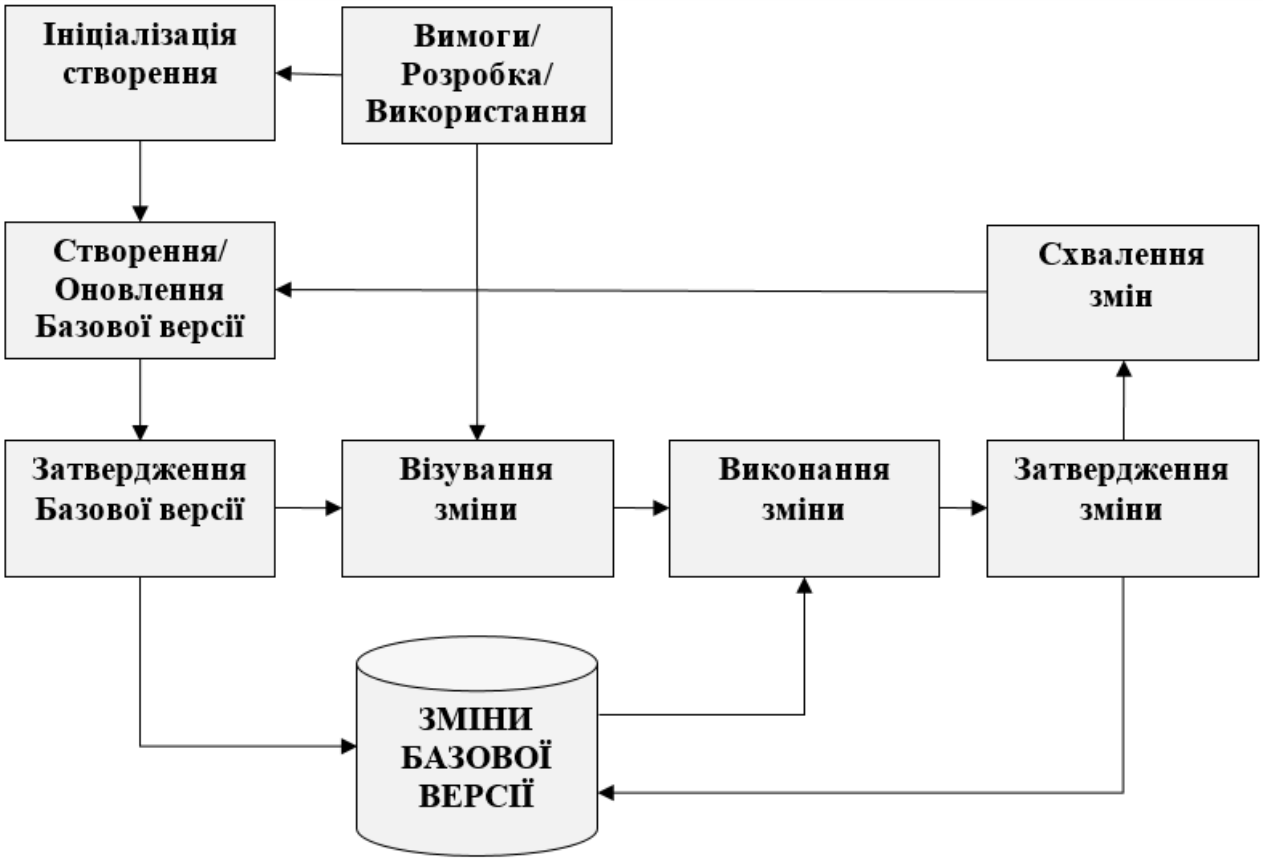

Рис. 6.3 Процедура формальної підтримки baseline

Baseline може також підтримуватися безперервною інтеграцією.

Важливо, що Baseline (особливо у випадку з програмними активами) не повинна встановлюватися дуже рано. Спочатку потрібно написати якусь кількість коду, щоб було що інтегрувати. Крім того, спочатку багато уваги приділяється розробці основних архітектурних рішень, і цілісна версія виявляється не затребуваною. Але починаючи з якогось моменту вона просто необхідна. Який цей момент – вирішувати членам команди. Нарешті, існують проекти, де автоматична збірка не потрібна зовсім – це прості проекти, що розробляються невеликою кількістю учасників, де немає великого кількість початкових текстів програм, проектів, складних параметрів компіляції.