Міністерство освіти і науки України

Національний технічний університет України

Київський Політехнічний Інститут

Навчально-науковий комплекс

“Інститут прикладного системного аналізу”

Кафедра математичних методів системного аналізу

**Звіт про проведення переддипломної практики**

***Виконав:***

студент групи КА-73

Даниленко А.С.

***Місце проходження практики:***

кафедра ММСА

***Керівник практики від вузу:***

Тимощук О.Л.

***Керівник практики на установі:***

Дідковська М.В.

2011 рік

**Індивідуальне завдання**

- Проаналізувати класичні підходи до якості програмного забезпечення

- Проаналізувати класичні підходи та методи регресійного тестування

- Проаналізувати основні принципи роботи систем керування версіями

**Зміст**

[1. Тестування програмного забезпечення 4](#_Toc291799744)

[1.1 Види тестування 5](#_Toc291799745)

[2. Регресійне тестування 7](#_Toc291799746)

[2.1 Визначення необхідної множини тестів 8](#_Toc291799748)

[2.2 Умови керованості і особливості реалізації регресійного тестування 10](#_Toc291799749)

[2.3 Класифікація тестів при відборі 13](#_Toc291799750)

[2.4 Класифікація методів вибору теcтів 16](#_Toc291799751)

[3. Системи керування версіями 19](#_Toc291799752)

[3.1 Традиційні системи контролю версій 21](#_Toc291799754)

[3.2 Розподілені системи контролю версій 22](#_Toc291799755)

[4. Список використаної літератури 24](#_Toc291799756)

# Тестування програмного забезпечення

Тестування програмного забезпечення — це процес, що використовується для виміру якості розроблюваного [програмного забезпечення](http://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F). Зазвичай, поняття якості обмежується такими поняттями, як коректність, повнота, безпечність, але може містити більше технічних вимог, які описані в стандарті ISO 9126[?]. Тестування - це процес технічного дослідження, який виконується на вимогу замовників, і призначений для вияву інформації про якість продукту відносно контексту, в якому він має використовуватись.[?] До цього процесу входить виконання [програми](http://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%27%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B0) з метою знайдення помилок.

Існуючі на сьогоднішній день методи тестування програмного забезпечення не дозволяють однозначно і повністю виявити всі дефекти і встановити коректність функціонування програми що аналізується, тому всі існуючі методи тестування діють в рамках формального процесу перевірки досліджуваного або розроблюваного програмного забезпечення.

Такий процес формальної перевірки або верифікації може довести, що дефекти відсутні з точки зору використовуваного методу. (Тобто немає ніякої можливості точно встановити або гарантувати відсутність дефектів у програмному продукті з урахуванням людського фактора, присутнього на всіх етапах життєвого циклу ПЗ).

## Види тестування

Існує кілька ознак, за якими прийнято робити класифікацію видів тестування. Зазвичай виділяють наступні[?]:

* За об'єктом тестування:
  + Функціональне тестування (functional testing)
  + Тестування продуктивності (performance testing)
  + Тестування навантаження (load testing)
  + Стрес-тестування (stress testing)
  + Тестування стабільності (stability / endurance / soak testing)
  + Тестування зручності використання (usability testing)
  + Тестування інтерфейсу користувача (UI testing)
  + Тестування безпеки (security testing)
  + Тестування локалізації (localization testing)
  + Тестування сумісності (compatibility testing)
* За знанням системи:
  + Тестування чорного ящика (black box)
  + Тестування білого ящика (white box)
  + Тестування сірого ящика (grey box)
* За ступенем автоматизації:
  + Ручне тестування (manual testing)
  + Автоматизоване тестування (automated testing)
  + Напівавтоматизоване тестування (semiautomated testing)
* За ступенем ізольованості компонентів:
  + Компонентне (модульне) тестування (component / unit testing)
  + Інтеграційне тестування (integration testing)
  + Системне тестування (system / end-to-end testing)
* За часом проведення тестування:
  + Тестування нової функціональності (new feature testing)
  + Регресійне тестування (regression testing)
  + Альфа-тестування (alpha testing)
  + Бета-тестування (beta testing)
  + Тестування при здачі (acceptance testing)
  + Тестування під час приймання (smoke testing)
* За ознакою позитивності сценаріїв:
  + Позитивне тестування (positive testing)
  + Негативне тестування (negative testing)
* За ступенем підготовленості до тестування:
  + Тестування з документації (formal testing)
  + Інтуїтивне тестування (ad hoc testing)

# Регресійне тестування

Регресійне тестування (regression testing) – збірна назва для всіх видів тестування програмного забезпечення, спрямованих на виявлення помилок у вже протестованих ділянках вихідного коду[?]. Такі помилки, що виникають коли після внесення змін до програми перестає працювати те, що повинно було продовжувати працювати, - називають регресійними помилками (regression defects).

Регресійне тестування включає[?]: перевірку виправленого знайденого раніше дефекту, перевірку, що виправлений раніше і верифікований дефект не відтворюється в системі знову, а також перевірку того, що не порушилася працездатність працюючої раніше функціональності, якщо її код міг бути задітий при виправленні деяких дефектів в іншій функціональності. Звичайно використовувані методи регресійного тестування включають повторні прогони попередніх тестів, а також перевірки, чи не потрапили регресійні помилки в чергову версію в результаті злиття коду.

Регресійне тестування – дорогий[?], але необхідний єтап тестування програмного забезпечення, спрямований на повторну перевірку коректності зміненої програми. Тому важливим завданням регресійного тестування є також зменшення вартості і скорочення часу виконання тестів.



## Визначення необхідної множини тестів

Нехай - множина з ***N*** тестів, що використовується при первинній розробці програми ***P***, а - підмножина регресійних тестів для тестування нової версії програми ***P'***. Інформація про покриття коду, яке забезпечується , дозволяє вказати блоки ***P'***, що потребують додаткового тестування, для чого може знадобитися повторний запуск деяких тестів з множини , або навіть створення - набору нових тестів для ***P'*** - і оновлення . Правильне виявлення множини є дуже важкою задачею.

Завдання відбору тестів з набору для заданої програми ***P*** і зміненої версії цієї програми полягає у виборі підмножини для повторного запуску на зміненій програмі , де . Так як вихідні дані ***P*** і для тестів з множини завідомо однакові, немає необхідності виконувати жоден з тестів на ***P'***. У загальному випадку, за відсутності динамічної інформації про виконання ***P*** і не існує методу обчислення множини для довільних множин ***P*** , і . Це випливає з відсутності загального вирішення проблеми зупинки[?], що складається в неможливості створення в загальному випадку алгоритму, що дає відповідь на питання, чи завершується коли-небудь довільна програма ***P*** для заданих значень вхідних даних. На практиці створення можливо тільки шляхом виконання на кожного регресійного тесту, чого і хочеться уникнути.

Реалістичний варіант вирішення завдання вибіркового регресійного тестування полягає в отриманні корисної інформації за результатами виконання ***P*** і об'єднання цієї інформації з даними статичного аналізу для отримання множини у вигляді апроксимації . Цей підхід застосовується у всіх відомих вибіркових методах регресійного тестування, заснованих на аналізі коду. Множина повинно включати всі тести з , що активують змінений код, і не включати ніяких інших тестів, тобто тест входить до тоді і тільки тоді, коли задіє код ***P*** в точці, де в код був видалений або змінений, або де був доданий новий код.

Якщо певний тест задіює в ***P*** той самий код, що і в , вихідні дані ***P*** і для відрізнятися не будуть. З цього випливає, що якщо , повинен задіяти певний код, змінений в по відношенню до , тобто повинно виконуватися відношення . З іншого боку, оскільки не кожне виконання зміненого коду відбивається на вихідних значеннях тесту, можуть існувати деякі такі , що . Таким чином, містить цілком і може використовуватися в якості його альтернативи без шкоди для якості програмного продукту.

## Умови керованості і особливості реалізації регресійного тестування

Протягом життєвого циклу програми період супроводу триває найдовше. Коли змінена програма тестується набором тестів , ми зберігаємо без змін по відношенню до тестування вихідної програми всі фактори, які могли б впливати на вихід програми. Тому атрибути конфігурації, в якій програма тестувалася останній раз (наприклад, план тестування, тести і елементи що покриваються ), підлягають управлінню конфігурацією. Практика тестування зміненої версії програми в тих же умовах, в яких тестувалася вихідна програма , називається керованим регресійним тестуванням. При некерованому регресійному тестуванні деякі властивості методів регресійного тестування можуть змінюватися, наприклад, безпечний метод відбору тестів може перестати бути безпечним. У свою чергу, для забезпечення керованості регресійного тестування необхідно виконання низки умов:

* Як при модульному, так і при інтеграційному регресійному тестуванні в якості модулів, що викликаються модулем, що тестується безпосередньо чи опосередковано, мають використовуватися реальні модулі системи. Це легко здійснити, оскільки на етапі регресійного тестування всі модулі доступні в завершеному вигляді.
* Інформація про зміни коректна. Інформація про зміни вказує на змінені модулі та розділи специфікації вимог, не маючи на увазі при цьому коректність самих змін. Крім того, при зміні специфікації вимог необхідне посилене регресійне тестування змінених функцій цієї специфікації, а також всіх функцій, які могли бути порушені з необережності. Єдиним випадком коли ми змушені покластися на правильність зміненого технічного завдання, є зміна технічного завдання для всієї системи або для модуля верхнього (у графі викликів) рівня, за умови, що крім технічного завдання, не існує ніякої додаткової документації і / або будь-якої інший інформації, за якою можна було б судити про помилку в технічному завданні.
* У програмі немає помилок, крім тих, які могли виникнути через її зміни.
* Тести, що застосовувалися для тестування попередніх версій програмного продукту, доступні, при цьому протокол прогону тестів складається з вхідних даних, вихідних даних і траєкторії. Траєкторія являє собою шлях у керуючому графові програми, проходження якого викликається використанням деякого набору вхідних даних. Її можна застосовувати для оцінки структурного покриття, забезпечуваного набором тестів.

Для проведення регресійного тестування з використанням існуючого набору тестів необхідно зберігати інформацію про результати виконання тестів на попередніх етапах тестування.

Перерахуємо деякі особливості реалізації регресійного тестування.

* Деякі ділянки коду програми не одержують керування при виконанні деяких тестів.
* Якщо ділянка коду реалізує вимогу, але змінений фрагмент коду не отримує управління при виконанні тесту, то він і не може впливати на значення вихідних даних програми при виконанні даного тесту.
* Навіть якщо ділянка коду, що реалізує вимоги, отримує управління при виконанні тесту, це далеко не завжди відбивається на вихідних даних програми при виконанні даного тесту. Дійсно, якщо змінюється перший блок програми, наприклад, шляхом додавання ініціалізації змінної, всі шляхи у програмі також змінюються, і, як наслідок, вимагають повторного тестування. Проте може так трапитися, що тільки на невеликому підмножині шляхів дійсно використовується ця ініціалізована змінна.
* Не кожен тест, перевіряючий код, що знаходиться на одному путі зі зміненим кодом, обов'язково покриває цей змінений код.
* Код, що знаходиться на одному путі зі зміненим кодом, може не впливати на значення вихідних даних змінених модулів програми.
* Не завжди кожен оператор програми впливає на кожен елемент її вихідних даних.

Припустимо, що зміни у програмі обмежуються одним оператором. Якщо при виконанні якого-небудь тесту на вихідній програмі цей оператор ніколи не отримує управління, можна з упевненістю сказати, що він не отримає управління і в ході виконання тесту на новій програмі, а результати тестування нової та старої програм будуть збігатися. Отже, немає необхідності виконувати цей тест на новій програмі. Зазначений метод легко можна узагальнити для випадку декількох змін: якщо тест не задіє жодного зміненого оператора, і його вхідні дані не змінилися, код, що виконується їм у зміненій програмі, буде в точності таким же, як в первинній версії. Такий тест не виявляє відмінностей між двома версіями системи, отже, немає необхідності проганяти його повторно. Якщо тест не зачіпає жодного оператора виведення, поведінка якого залежить від змінених операторів, це означає, що, незважаючи на зміни в програмі, всі оператори, які одержують керування при виконанні цього тесту, не змінять висновок системи по відношенню до попередньої версії. Таким чином, немає необхідності повторно проганяти і тести такого роду.

Отже, необхідно орієнтуватися на вибір тільки тих тестів, які покривають змінений код, що впливає, у свою чергу, на вихідні дані програми. Такий підхід гарантує, що будуть обрані тільки тести, які виявляють зміни, і метод буде точним.

## Класифікація тестів при відборі

Створення наборів регресійних тестів рекомендується починати з множини вихідних тестів. При заданому критерії регресійного тестування всі вихідні тести ***t*** поділяються на чотири підмножини:

* Множина тестів, придатних для повторного використання. Це тести, які вже запускалися і придатні до використання, але зачіпають тільки елементи програми, що не зазнали змін. При повторному виконанні вихідні дані таких тестів співпадуть з вихідними даними, отриманими на вихідній програмі. Отже, такі тести не вимагають перезапуску.
* Множина тестів, які потребують повторного запуску. До них відносяться тести, які вже запускали, але вимагають перезапуску, оскільки зачіпають, принаймні, один змінений елемент, який підлягає повторному тестуванню. При повторному виконанні такі тести можуть давати результат, відмінний від результату, показаного на вихідній програмі. Множина тестів, які потребують повторного запуску, забезпечує хороше покриття структурних елементів навіть за наявності нових функціональних можливостей.
* Множина застарілих тестів. Ці тести, більше не можуть бути застосовні до зміненої програмі та непридатні для подальшого тестування, оскільки вони зачіпають тільки елементи, які були видалені при зміні програми. Їх можна видалити з набору регресійних тестів.
* Нові тести, які ще не запускалися і можуть бути використані для тестування.

Новий

Придатний для повторного використання

Потребує повторного запуску

Застарілий

Рис. 1 Життєвий цикл тесту

Рис. 1 дає уявлення про життєвий цикл тесту. Відразу після створення тест вводиться в структуру бази даних як новий. Після виконання новий тест переходить у категорію тестів, придатних для повторного використання або застарілих. Якщо виконання тесту сприяло збільшенню поточної ступеня покриття коду, тест позначається як придатний для повторного використання. В іншому випадку він позначається як застарілий і відкидається. Існуючі тести, повторно запущені після внесення зміни в код, також класифікуються заново як придатні для повторного використання або застарілі в залежності від тестових траєкторій і використовуваного критерію тестування.

Класифікація тестів по відношенню до змін в коді вимагає аналізу наслідків змін. Тести, що активують код, що був змінений, можуть вимагати повторного запуску або виявитися застарілими. Щоб тест був включений у клас тестів, які потребують повторного запуску, він повинен покривати зміни в коді, а також має сприяти збільшенню ступеня покриття зміненого коду по використовуваному критерію. Покритим тестом елементом, що був змінений, може бути траєкторія, вихідні значення, або і те, й інше. Щоб тест був включений у клас тестів, придатних для повторного використання, він має вносити внесок у збільшення ступеня покриття коду і не вимагати повторного запуску.

Ступінь покриття коду визначається для тестів, придатних для повторного використання, оскільки до цього класу відносяться тести, які не потребують повторного запуску, які сприяють збільшенню ступеня покриття до бажаної величини. Якщо є компонент програми, не задіяний придатними для повторного використання тестами, то замість них вибираються і виконуються з метою збільшення ступеня покриття тести, що вимагають повторного запуску. Після запуску такий тест стає придатним для повторного використання або застарілим. Якщо тестів, які потребують повторного запуску, більше не залишилося, а необхідна ступінь покриття коду ще не досягнута, породжуються додаткові тести і тестування повторюється.

Остаточний набір тестів збирається з тестів, придатних для повторного використання, тестів, які потребують повторного запуску, і нових тестів. Нарешті, застарілі й надлишкові тести видаляються з набору тестів, оскільки надлишкові тести не перевіряють нові функціональні можливості і не збільшують покриття.

## Класифікація методів вибору теcтів

Для перевірки коректності різних підходів до регресійного тестування використовується модель оцінки методів регресійного тестування. Основними об'єктами розгляду стали повнота, точність, ефективність та універсальність.

Повнота відображає міру відбору тестів з , на яких результат виконання зміненої програми відмінний від результату виконання вихідної програми, внаслідок чого можуть бути виявлені помилки в . Метод, повний на 100%, називається безпечним.

Точність - міра здатності методу уникати вибору тестів з , на яких результат виконання зміненої програми не буде відрізнятися від результату її первинною версією, тобто тестів, нездатних виявляти помилки в . Припустимо, що набір містить регресійних тестів. З них для тестів поведінка та результати виконання старої програми відрізняються від поведінки і результатів виконання нової програми . Набір тестів містить тестів, отриманих з використанням методу відбору регресійних тестів . З цих тестів для тестів поведінка і розрізняється. Точність щодо , , і , виражена у відсотках, визначається виразом , тоді як відповідний відсоток вибраних тестів визначається виразом , якщо або дорівнює ***100%*** , якщо .

Виходячи з наведеного визначення, точність множини тестів - це відношення числа тестів даної множини, на яких результати виконання нової та старої програм розрізняються, до загального числа тестів множини. Точність є важливим атрибутом методу регресійного тестування. Неточний метод має тенденцію відбирати тести, які не повинні були бути обрані. Чим менш точний метод, тим ближче обсяг обраного набору тестів до обсягу вихідного набору тестів.

Ефективність - оцінка обчислювальної вартості стратегії вибіркового регресійного тестування, тобто вартості реалізації її вимог щодо часу і пам'яті, а також можливості автоматизації. Відносною ефективністю називається ефективність методу тестування за умови наявності не більше однієї помилки в програмі, що тестується. Абсолютною ефективністю називається ефективність методу в реальних умовах, коли оцінка кількості помилок в програмі не обмежена.

Універсальність відображає міру здатності методу до застосування в досить широкому діапазоні ситуацій, що зустрічаються на практиці.

Для програми , її зміненої версії і набору тестів для потрібно, щоб методика вибіркового повторного тестування задовольняла наступним критеріям оцінки:

Критерій 1. Безпека. Методика вибіркового повторного тестування повинна бути безпечною, тобто повинна вибирати всі тести з , які потенційно можуть виявляти помилки (усі тести, чия поведінка на і може бути різним). Безпечна методика повинна розглядати наслідки додавання, видалення і зміни коду. При додаванні нового коду в у можуть вже міститися тести, що покривають цей новий код. Такі тести необхідно виявляти і враховувати при відборі.

Критерій 2. Точність. Стратегія повторного прогону всіх тестів є безпечною, але неточною. На додаток до вибору всіх тестів, потенційно здатних виявляти помилки, вона також вибирає тести, які в жодному разі не можуть демонструвати змінену поведінку. В ідеалі, методика вибіркового повторного тестування повинна бути точною, тобто повинна вибирати тільки тести зі зміненим поведінкою. Однак для довільно взятого тесту, не запускаючи його, неможливо визначити, чи зміниться його поведінку. Отже, в кращому випадку ми можемо розраховувати лише на деяке збільшення точності.

Всілякі існуючі вибіркові методи регресійного тестування розрізняються не в останню чергу вибором об'єкта або об'єктів, для яких виконується аналіз покриття та аналіз змін. Наприклад, при аналізі на рівні функції при зміні будь-якого оператора функції вся функція вважається зміненою; при аналізі на рівні окремих операторів ми можемо виключити частину тестів, що містять виклик функції, але не активують змінений оператор. Вибір об'єктів для аналізу покриття відбивається на рівні подробиці аналізу, а значить, і на його точності та ефективності. Абсолютні величини точності та кількості обраних тестів для заданих набору тестів і безлічі змін повинні розглядатися тільки разом зі зменшенням розміру набору тестів. Невеликий відсоток вибраних тестів може бути прийнятним, тільки якщо рівень точності залишається досить високим.

Критерій 3. Ефективність. Методика вибіркового повторного тестування повинна бути ефективною, тобто повинна допускати автоматизацію і виконуватися достатньо швидко для практичного застосування в умовах обмеженого часу регресійного тестування. Методика повинна також передбачати зберігання інформації про хід виконання тестів в мінімально можливому обсязі.

Критерій 4. Універсальність. Методика вибіркового повторного тестування повинна бути універсальною, тобто застосовної до всіх мов і мовних конструкцій, ефективною для реальних програм і здатною до обробки як завгодно складних змін коду.

У загальному випадку існує певний компроміс між безпекою, точністю і ефективністю. При відборі тестів аналіз необхідно провести за час, менший, ніж потрібно для виконання та перевірки результатів тестів з **T**, що не увійшли до . З урахуванням цього обмеження рішенням задачі регресійного тестування буде безпечний метод з хорошим балансом дешевизни і високої точності.

# Системи керування версіями

Система управління ревізіями, також відома як система контролю версій або система управління – є система управління змінами в документах, програмах та іншій інформації що зберігається у вигляді файлів на комп'ютері[?]. Найбільш часто використовується в розробці програмного забезпечення, де група людей може змінювати одні і ті ж файли. Зміни, як правило, позначаються певною кодовою цифрою або літерою, що називається "номер ревізії". Наприклад, початковий набір файлів називають "ревізія 1". Коли перша зміна зроблена, результуюча множина називається "ревізія 2", і так далі. Кожна ревізія пов'язана з відміткою про час зміни та особу, що вносила зміну. Ревізії можна порівнювати, відновлювати більш стару ревізію, і з деякими типами файлів – зливати дві ревізії створені різними людьми у одну – об’єднану.

Системи контролю версій найбільш часто використовуються як самостійні програми, але контроль версій також вбудований у різні види програмного забезпечення, такі як текстові процесори (наприклад, Microsoft Word, OpenOffice.org Writer, KWord, Pages), процесори електронних таблиць (наприклад, Microsoft Excel, OpenOffice.org Calc, KSpread, Numbers), а також у різні системи управління контентом (наприклад, Drupal, Joomla, WordPress). Вбудований контроль ревізій є ключовою особливістю wiki пакетів програмного забезпечення, таких як MediaWiki, DokuWiki, TWiki т.д. У wiki, контроль версій дає здатність повернутися до сторінки попередньої ревізії, яка має вирішальне значення для редакторів і дозволяє відстежувати правки один одного, виправляти помилки, і захищати суспільні wiki від вандалізму і спаму.

У галузі розробки програмного забезпечення, контроль версій є єдиною можливістю відстежувати і забезпечувати контроль за змінами у вихідному коді. Розробники іноді використовують програмне забезпечення контролю версій для збереження документації і конфігураційних файлів так само як вихідного коду.

Команди розробників одночасно проектують, розробляють та впроваджують декількох версій одного і того ж програмного забезпечення, які будуть розгорнуті в різних місцях, і одночасно працюють над оновленнями(patches). Певні помилки або функції програмного забезпечення часто присутні тільки в деяких версіях (через фіксацію деяких проблем і впровадження інших, програма розвивається). Таким чином, з метою пошуку та виправлення помилок, життєво важливо мати можливість контролювати і запускати різні версії програмного забезпечення, щоб визначити в якій версії виникає проблема.

На найпростішому рівні, розробники могли просто зберегти кілька копій різних версій програми, і позначити їх відповідним чином. Цей простий підхід був використаний на багатьох великих програмних проектах. Хоча цей метод може працювати, це неефективно,так як багато майже ідентичних копії програми повинні бути збережені. Це вимагає гарної дисципліни з боку розробників, і часто призводить до помилок. Отже, були розроблені системи для автоматизації деяких або всіх процесів контролю версій.

Крім того, в розробці програмного забезпечення, в правовій та бізнесовії сферах все частіше однин документ або фрагмент коду редагує вся команда, члени якої можуть бути географічно розділені і можуть переслідувати різні і навіть протилежні інтереси . Складна система контролю версій, яка відстежує і встановлює хто змінював певну частину документу може бути дуже корисною і навіть необхідною.



## Традиційні системи контролю версій

Традиційні системи контролю версій використовують централізовану модель, в якій усі функції контролю версій відбуваються на сервері. Якщо два розробники намагаються змінити той же файл в той же час, без будь-якого методу управління доступом розробники можуть в кінцевому підсумку перезаписати роботу один одного. Централізовані системи контролю версій вирішують цю проблему однією з двох можливих моделей управління : блокування файлів і злиття версій.

**Атомарні операції**

Вчені з комп'ютерних наук говорять о атомарних операціях, якщо система залишається в узгодженому стані, навіть якщо операція перервана. Як правило, найбільш важливим є щоб команда фіксування змін була атомарною у цьому сенсі. Операція фіксування говорить системі контролю версій що ви хочете, щоб група змін, що ви робили стали доступним для всіх користувачів. Не всі системи контролю версій мають можливість атомарних змін; зокрема, широко використовувана система CVS не має цієї функції.

**Метод блокування файлів**

Найпростіший спосіб запобігання "одночасного доступ" передбачає блокування файлів, так що тільки один розробник в той час, має доступ на запис до центрального "сховища" копії певних файлів. Після того як один розробник заблокував файл, інші можуть прочитати цей файл, але ніхто не може змінити цей файл, поки розробник не зафіксує оновлену версію (чи скасує блокування).

Блокування файлів має як переваги і недоліки. Воно може забезпечити деякий захист від важких конфліктів злиття, коли користувач вносить радикальні зміни в багато розділів великого файлу (або групи файлів). Однак, якщо файли залишаються заблокованими занадто довго, інші розробники можуть бути змушені обійти програмне забезпечення контролю версій і змінювати файли локально, що призводить до більш серйозних проблем.

**Метод злиття**

Більшість систем керування версіями дозволяє декільком розробникам редагувати той же файл в той же час. Перший розробник фіксує зміни в центральному репозиторії завжди успішно, при цьому система повинна забезпечити можливості для подальшого злиття змін в центральне сховище, і зберегти зміни першого розробника, коли інші розробники будуть фіксувати свої зміни.

Злиття двох файлів – дуже тонка операція, і, як правило можливе тільки тоді, коли структура файлів, проста, як в текстових файлах. Злиття двох файлів зображень не може призвести до результуючого файлу зображення. Також розробник повинен перевірити результат злиття, щоб переконатися, що зміни сумісні і що операція злиття не вносить свої власні помилки логіки у файли. Метод злиття може передбачати блокування файлів для ексклюзивного доступу на запис, навіть якщо можливість злиття існує.

**Базові точки, етикетки і бирки**

Більшість інструментів контролю версій використовує тільки один з цих подібних методів(базова точка, етикетка, бирка) для позначення дій виявлення знімку проекту в певний час. Ці три назви можна вважати синонімами.

У більшості проектів деякі знімки є більш значущими, ніж інші, такі, як ті, які використовуються для позначення опублікованих релізів, версій, або оновлень.

## Розподілені системи контролю версій

Розподілені системи контролю версій використовують так званий Peer -To-Peer підхід, на відміну від клієнт-серверного підходу централізованих систем. Замість одного, центрального сховища, з яким клієнти синхронізуються, кожний розробник має свою робочу копію у своєму локальному сховищі. Розподілені системи контролю версій проводять синхронізацію шляхом обміну патчами між собою. У результаті вони мають деякі важливі відмінності від централізованої системи:

* Найчастіше використовувані операції (наприклад, фіксація, перегляд історії, і скасування змін) відбуваються швидко, тому що немає необхідності спілкуватися з центральним сервером.
* Мережевий зв'язок необхідний тільки при завантаженні або вивантаженні змін іншим розробникам.
* Кожна робоча копія ефективно функціонує в якості віддаленого резервного серверу збереження коду, історії, і забезпечуює захист від втрати даних.

# Список використаної літератури

1. ISO/IEC 9126 Software engineering — Product quality. — International Standard . — 2001 . — 34 p
2. ISO/IEC 25010 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models. — International Standard . — 2011 . — 34 p
3. Тестування програмного забезпечення // Вікіпедія — вільна енциклопедія. – http://www.uk.wikipedia.org/wiki/Тестування\_програмного\_забезпечення
4. Регрессионное тестирование // Википедия — свободная энциклопедия . – http://www.ru.wikipedia.org/wiki/Регрессионное\_тестирование
5. Регрессионное тестирование: цели и задачи, условия применения, классификация тестов и методов отбора // Интернет-Университет Информационных Технологий .- http://www.intuit.ru/department/se/testing/11/testing\_11.html
6. Лайза Криспин, Джанет Грегори Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд = Agile Testing: A Practical Guide for Testers and Agile Teams. — М.: «Вильямс», 2010. — 464 с.
7. Канер Кем, Фолк Джек, Нгуен Енг Кек Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. — Киев: ДиаСофт, 2001. — 544 с.
8. Калбертсон Роберт, Браун Крис, Кобб Гэри Быстрое тестирование. — М.: «Вильямс», 2002. — 374 с.