Технологія створення програмних продуктів

Види та рівні тестування

Правильність ПЗ. Надійність ПЗ.

Під програмою, часто розуміють *правильну програму*, тобто програму, яка не містить помилок: у програмі є помилка, якщо вона не виконує того, що очікує від неї користувач (при цьому розумність визначає документація - частковим випадком помилки може бути неузгодженість документації з ПЗ).

Надійність ПЗ – здатність ПЗ безвідмовно виконувати визначені функції за заданих умов протягом заданого часу і з достатньо великою вірогідністю.

Під *відмовою* розуміють наявність в ПЗ помилки (надійність ПЗ не виключає наявність помилки). Для оцінки надійності враховують вартість кожного збою.

Природа помилок ПЗ

Інтелектуальні можливості людини для розробки ПЗ:

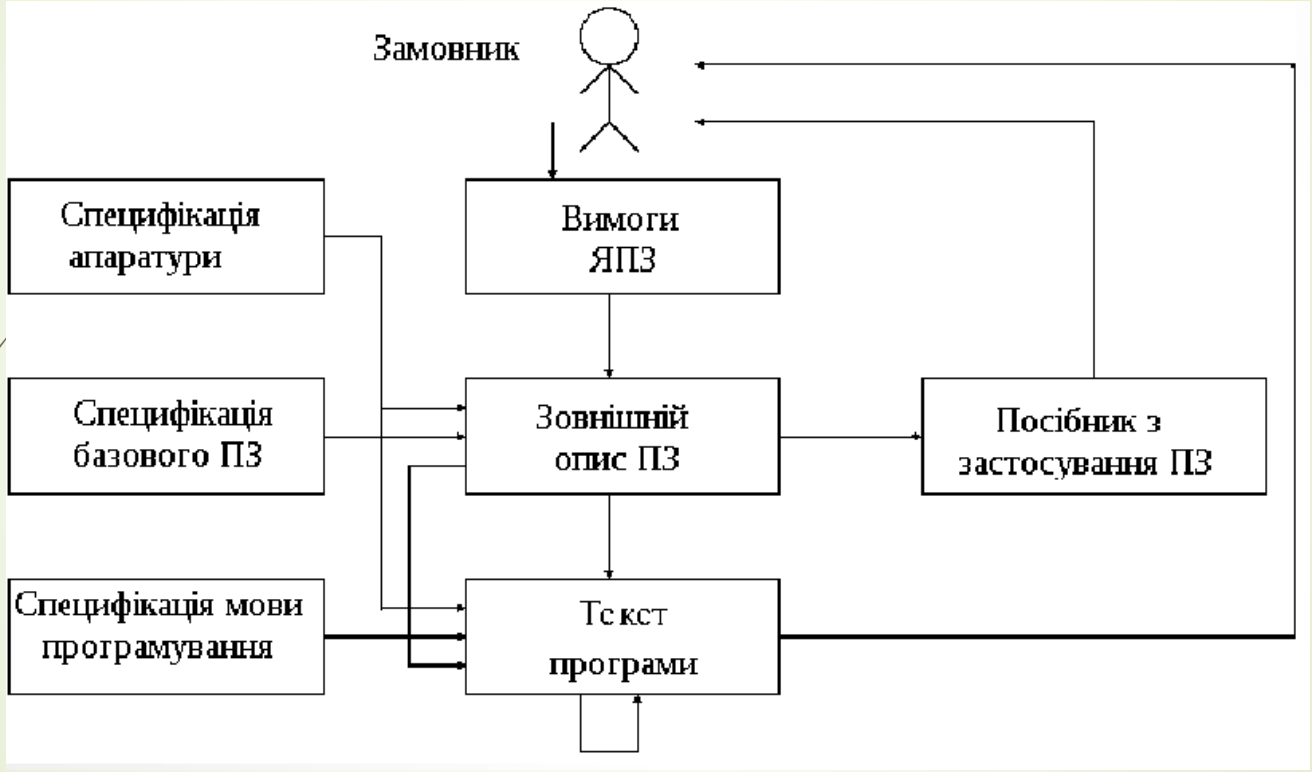
- здатність до перебору (типово до 1000 предметів);
- здатність до абстракції (об'єднання предметів у групи);
- здатність до математичної індукції (узагальнення).

Система – сукупність взаємозв'язаних елементів.

Зрозуміти систему – значить осмислено перебрати всі шляхи взаємодії елементів між собою. Проста система – людина може впевнено перебрати всі зв'язки між елементами n < 7 (зв'язки: 6! = 720 < 1000), складна – система, в якій вона цього зробити не в стані (елементів >7).

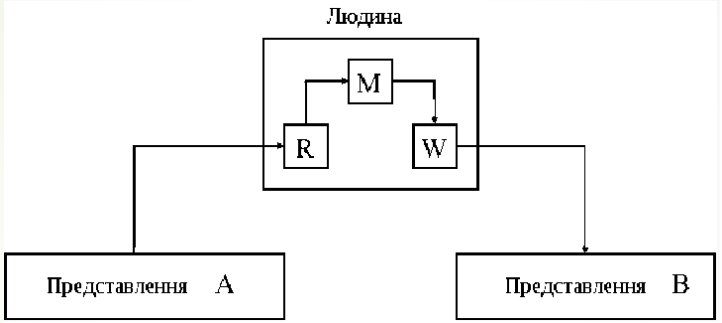
Мала система завжди проста, а велика може бути як простою, так і складною. Наша мета — навчитись робити великі системи простими.

Неправильне перетворення інформації



На кожному з етапів можлива поява помилки через неправильне розуміння вихідного представлення. Виникнувши на одному з етапів проектування або розробки, помилка перетворюється в нові помилки і потрапляє в ПЗ.

Неправильне перетворення інформації



Механізм перетворення здійснюється за 4 етапи:

- людина отримує інформацію з представлення A за допомогою механізму R;
- запам’ятовує отриману інформацію в пам’яті M;
- вибирає з своєї пам’яті інформацию, що перетворюється, і інформацію, що описує процес перетворення, і відсилає результат механізму W;
- за допомогою механізму W людина фіксує нове представлення інформації у вигляді B.

На кожному з цих етапів людина може допустити помилку

Тестування програмного забезпечення (Software Testing) – перевірка відповідності між реальною і очікуваною поведінкою програми, здійснювана на кінцевому наборі тестів, обраному певним чином.

План тестування (Test Plan) – це документ, що описує весь обсяг робіт з тестування, починаючи з опису об'єкта, стратегій, розкладів, критеріїв початку та закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінки ризиків з варіантами їх вирішення.

Тест дизайн (Test Design) – це етап процесу тестування програмного продукту, на якому проектуються і створюються тестові випадки (тест кейси), у відповідності з певними визначеними попередньо критеріями якості та цілями тестування.

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

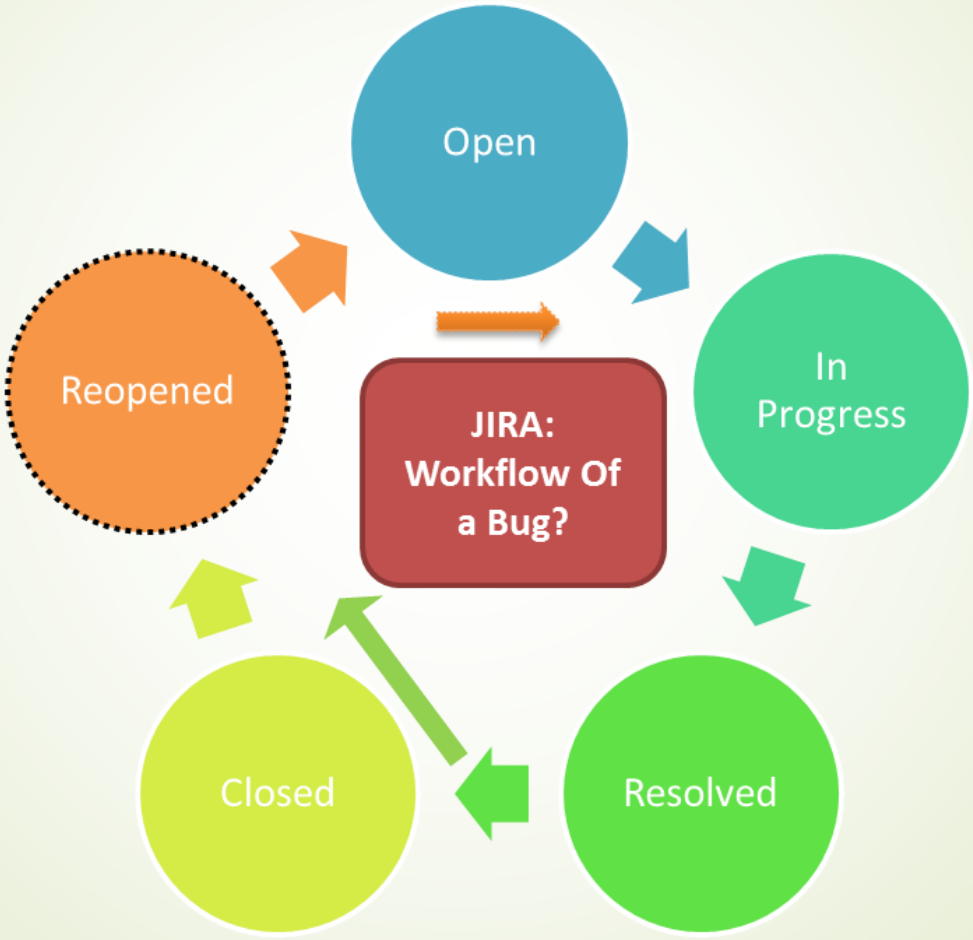
Баг / Дефект репорт (Bug Report) – це документ, що описує ситуацію або послідовність дій, яка призвела до некоректної роботи об'єкта тестування, із зазначенням причин та очікуваного результату.

Тестове покриття (Test Coverage) – це метрика оцінки якості тестування, що представляє із себе щільність покриття тестами вимог або виконуваного коду.

Деталізація тест кейсів (Test Case Specification) – це рівень деталізації опису тестових кроків і необхідного результату, при якому забезпечується розумне співвідношення часу проходження до тестового покриття.

Час проходження тест кейса (Test Case Pass Time) – це час від початку проходження кроків тест кейса до отримання результату тесту.

Життєвий цикл дефекту



Шаблон баг-репорту

“Шапка”	
Короткий опис (Summary)	Короткий опис проблеми, що явно вказує на причину і тип помилкової ситуації.
Проект (Project)	Назва тестованого проекту
Компонент додатку (Component)	Назва частини або функції тестованого продукту
Номер версії (Version)	Версія на якій була знайдена помилка
Серйозність (Severity)	Найбільш поширена п'ятирівнева система градації серйозності дефекту: ○ S1 Блокуючий (Blocker) ○ S2 Критичний (Critical) ○ S3 Значний (Major) ○ S4 Незначний (Minor) ○ S5 Тривіальний (Trivial)
Пріоритет (Priority)	Пріоритет дефекту: ○ P1 Високий (High) ○ P2 Середній (Medium) ○ P3 Низький (Low)
Статус (Status)	Статус бага. Залежить від використовуваної процедури та життєвого циклу бага (bug workflow and life cycle)
Автор (Author)	Автор баг-репорту
Призначений на (Assigned To)	Ім'я співробітника, призначеного на вирішення проблеми

Шаблон баг-репорту

Оточення	
ОС / Сервіс пак і т.д. / Браузера + версія / ...	Інформація про оточення, на якому був знайдений баг: операційна система, сервіс пак, для WEB тестування – назва і версія браузера і т.д.
…	
Опис	
Кроки відтворення (Steps to Reproduce)	Кроки, за якими можна легко відтворити ситуацію, що призвела до помилки.
Фактичний результат (Result)	Результат, отриманий після проходження кроків до відтворення
Очікуваний результат (Expected Result)	Очікуваний правильний результат
Доповнення	
Прикріплений файл (Attachment)	Файл з логами, скріншот або будь-який інший документ, який може допомогти прояснити причину помилки або вказати на спосіб вирішення проблеми
…	

Шаблон баг-репорту

Серйозність (Severity) – це атрибут, що характеризує вплив дефекту на працездатність додатка.

Пріоритет (Priority) – це атрибут, який вказує на черговість виконання завдання або усунення дефекту.

Градація серйозності дефекту (Severity):

- S1 ***Блокуюча*** (Blocker) – помилка, що приводить додаток в неробочий стан, в результаті якого подальша робота з тестованою системою або її ключовими функціями стає неможливою. *Рішення проблеми необхідно для подальшого функціонування системи.*
- S2 ***Критична*** (Critical) – помилка, неправильно працююча ключова логіка, діра в системі безпеки, проблема, яка призвела до тимчасового падіння сервера або приводить в неробочий стан деяку частину системи, без можливості вирішення проблеми, використовуючи інші вхідні точки. *Рішення проблеми необхідно для подальшої роботи з ключовими функціями тестованої системи.*

Шаблон баг-репорту

- S3 ***Значна*** (Major) – частина основної логіки працює некоректно. Помилка некритична або є можливість для роботи з тестованою функцією, використовуючи інші вхідні точки.
- S4 ***Незначна*** (Minor) – помилка, що не порушує логіку тестованої частини додатка або очевидна проблема користувацького інтерфейсу.
- S5 ***Тривіальна*** (Trivial) – помилка, яка не стосується логіки додатка, погано відтворювана, малопомітна засобами інтерфейсу користувача, проблема сторонніх бібліотек або сервісів, проблема, не надає ніякого впливу на загальну якість продукту.

Шаблон баг-репорту

Градація пріоритету дефекту (Priority):

- P1 ***Високий*** (High) – Помилка має бути виправлена якомога швидше, оскільки її наявність є критичною для проекту.
- P2 ***Середній*** (Medium) – Помилка має бути виправлена, її наявність не є критичною, але вимагає обов'язкового рішення.
- P3 ***Низький*** (Low) – Помилка має бути виправлена, її наявність не є критичною, і не потребує термінового вирішення.

Приклад тест кейсу

Перевірити чи після введення назви предмету в поле «Предмети» та прізвище викладача в поле «Викладачі» правильний розклад буде показаний користувачеві.

Кроки:

- Завантажити в браузері http://www.electronics.lnu.edu.ua/.
- Вибрати з меню Довідка «Розклад»
- В поле «Предмети» вводимо назву «Матаналіз»
- В поле «Викладачі» вводимо прізвище «Цаповська»
- Натиснути кнопку «Search»

Очікуваний результат:

Розклад лише з предметом «Матаналіз» та викладачем Цаповська показаний на формі.

Вівторок	Вів	ФеІ-11	Вів	ФеІ-12	Вів	ФеІ-13
3		Матаналіз (лекц) доц. Цаповська Ж.Я. 1		Матаналіз (лекц) доц. Цаповська Ж.Я. 1		Матаналіз (лекц) доц. Цаповська Ж.Я. 1
4	чис.	Матаналіз (пр) доц. Цаповська Ж.Я. 4				
Середа	Сер	ФеІ-11	Сер	ФеІ-12	Сер	ФеІ-13
2		Матаналіз (пр) доц. Цаповська Ж.Я. 3				
4	чис.	Матаналіз (лекц) доц. Цаповська Ж.Я. 1	чис.	Матаналіз (лекц) доц. Цаповська Ж.Я. 1	чис.	Матаналіз (лекц) доц. Цаповська Ж.Я. 1

Приклад звіту про помилку

Після зменшення розміру вікна всі меню стали недоступними для користувача.

Кроки:

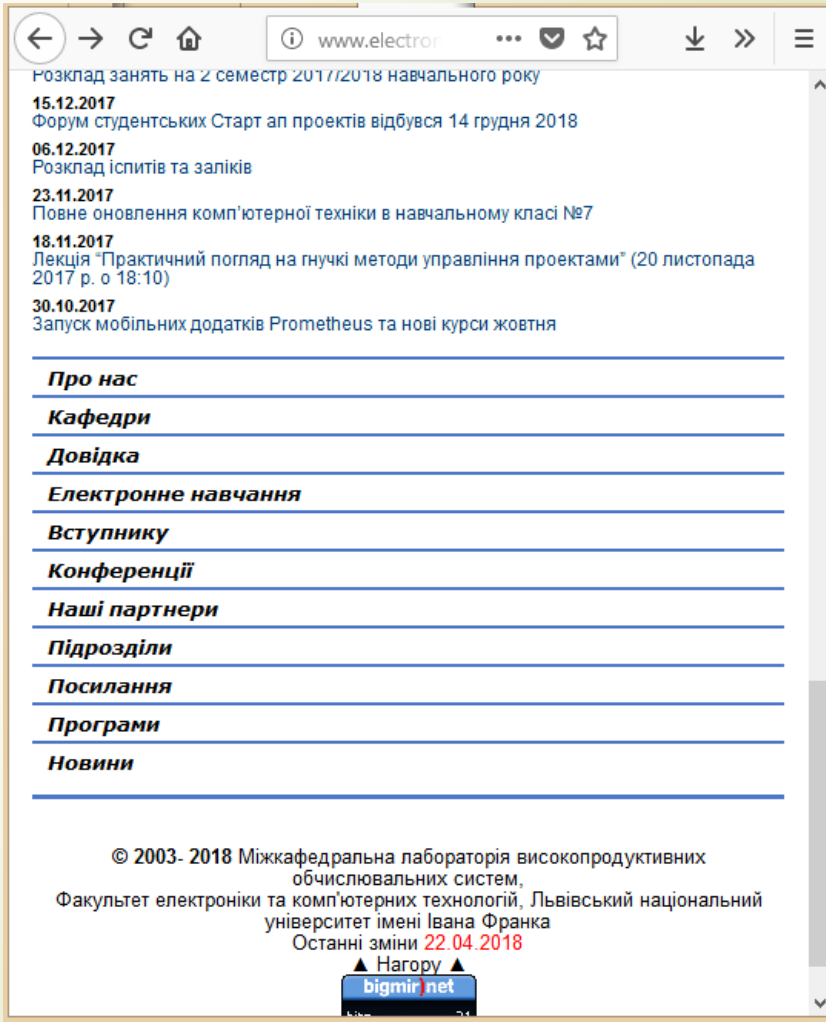
- Завантажити в браузері http://www.electronics.lnu.edu.ua/.
- Зменшити розмір вікна до мобільного вигляду.

Реальний результат:

Сайт не придатний до перегляду. Користувач не може повноцінно користуватись його функціоналом.

Очікуваний результат:

Сайт залишається придатним до перегляду. Користувач може використовувати весь його функціонал.



Види тестування програмного забезпечення

Всі *види тестування програмного забезпечення*, в залежності від переслідуваних цілей, можна умовно розділити на наступні групи:

- Функціональні
- Нефункціональні
- Пов'язані зі змінами

Функціональні тести базуються на функціях і особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх *рівнях тестування*: компонентному або модульному (Component / Unit testing), інтеграційному (Integration testing), системному (System testing) та приймальному (Acceptance testing).

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні різними величинами.

Пов'язані зі змінами види тестування – після проведення необхідних змін, таких як виправлення бага/дефекту, програмне забезпечення повинне бути переатестоване для підтвердження того факту, що проблема була дійсно вирішена.

Функціональні тести: Функціональне тестування

Функціональне тестування (Functional Testing) розглядає заздалегідь зазначену поведінку і ґрунтується на аналізі специфікацій функціональності компонента або системи в цілому. Як правило, ці функції описуються у вимогах, функціональних специфікаціях чи у вигляді випадків використання системи (use cases).

Тестування функціональності може проводиться в двох аспектах:

- вимоги
- використання

Переваги функціонального тестування:

- ✓ імітує фактичне використання системи;

Недоліки функціонального тестування:

- ✓ можливість упущення логічних помилок в програмному забезпеченні;
- ✓ ймовірність надмірного тестування.

Функціональні тести: Тестування безпеки

Тестування безпеки (Security and Access Control Testing) – це стратегія тестування, використовувана для перевірки безпеки системи, пов'язаних із забезпеченням цілісного підходу до захисту додатка від атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних , а також для аналізу ризиків.

Загальна стратегія безпеки ґрунтується на трьох основних принципах:

- конфіденційність;
- цілісність;
- доступність.

Конфіденційність – це приховування певних ресурсів або інформації. Під конфіденційністю можна розуміти обмеження доступу до ресурсу деякої категорії користувачів, або іншими словами, за яких умов авторизований користувач може отримати доступ до певного ресурсу.

Функціональні тести: Тестування безпеки

Існує два основні критерії при визначенні поняття ***цілісності***:

- **Довіра.** Очікується, що ресурс буде змінений тільки відповідним способом певною групою користувачів.
- **Пошкодження та відновлення.** У разі коли дані пошкоджуються або неправильно змінюються авторизованим або не авторизованим користувачем, ви повинні визначити на скільки важливою є процедура відновлення даних.

Доступність являє собою вимоги про те, що ресурси повинні бути доступні авторизованому користувачеві, внутрішньому об'єкту або пристрою. Як правило, чим більш критичний ресурс тим вище рівень доступності повинен бути.

Функціональні тести: Тестування взаємодії

Тестування взаємодії (Interoperability Testing) – це функціональне тестування, що перевіряє здатність додатка взаємодіяти з одним і більше компонентами або системами і включає в себе тестування сумісності (compatibility testing) та виконується на інтеграційному рівні тестування (integration testing).

З розвитком мережевих технологій та інтернету взаємодія різних систем, сервісів і додатків один-з-одним набула значної актуальності, оскільки будь-які пов'язані з цим проблеми можуть привести до падіння авторитету компанії, що, як наслідок, спричинить фінансові втрати. Програмні продукти з хорошими характеристиками взаємодії може бути легко інтегроване з іншими системами, не вимагаючи якихось серйозних модифікацій.

Нефункціональні тести: Навантажувальне тестування

Навантажувальне тестування або ***тестування продуктивності*** – це автоматизоване тестування, яке імітує роботу певної кількості користувачів на якомусь загальному для них ресурсі.

У тестування навантаження входять наступні *види тестування*:

- *Стресове тестування (Stress Testing) – дозволяє перевірити наскільки додаток і система в цілому працездатні в умовах стресу, а також оцінити здатність системи до регенерації;*
- *Об'ємне тестування (Volume Testing):*
  - *вимір часу виконання обраних операцій при певних інтенсивностях виконання цих операцій*
  - *може проводитися визначення кількості користувачів, що одночасно працюють з додатком*
- *Тестування стабільності або надійності (Stability / Reliability Testing) – перевірка працездатності додатка при тривалому (багатогодинному) тестуванні із середнім рівнем навантаження.*

Нефункціональні тести: Тестування встановлення

Тестування встановлення (Installation Testing) націлене на перевірку успішної інсталяції та налаштування, а також оновлення або видалення програмного забезпечення.

Інсталятор – це програма, основні функції якої – Встановлення (Інсталяція), Оновлення та Видалення (Деінсталяція) програмного забезпечення. *Інсталятор має ряд особливостей*, серед яких варто відзначити наступні:

- Глибока взаємодія з операційною системою і залежність від неї (файлова система, реєстр, сервіси та бібліотеки);
- Сумісність як рідних, так і сторонніх бібліотек, компонент або драйверів, з різними платформами;
- Зручність використання: інтуїтивно зрозумілий інтерфейс, навігація, повідомлення та підказки;
- Сумісність налаштувань користувача і документів в різних версіях додатка.

Нефункціональні тести: Тестування встановлення

Список ризиків:

- ризик втрати користувацьких даних
- ризик виведення операційної системи з ладу
- ризик непрацездатності додатка
- ризик некоректної роботи додатку.

Нефункціональні тести: Тестування встановлення

Встановлення (Інсталяція):

- Наявність достатніх для встановлення програми ресурсів таких як: оперативна пам'ять, дисковий простір тощо
- Коректність списку файлів в інсталяційному пакеті
  - при виборі різних типів установки, або параметрів встановлення, список файлів і шляхи до них також можуть відрізнятися
  - відсутність зайвих файлів (проектні файли, не включені в інсталяційний пакет, не повинні потрапити на диск користувача)
- Реєстрація додатка в ОС

Нефункціональні тести: Тестування встановлення

- Реєстрація розширень для роботи з файлами:
  - для нових розширень
  - для вже існуючих розширень
- Права доступу користувача, який ставить додаток:
  - права на роботу з системним реєстром
  - права на доступ до файлів і каталогів, наприклад %Windir%\system32
- Інсталяція декількох додатків за одні візит
- Встановлення одного і того ж додатка в різні робочі директорії однієї робочої станції

Нефункціональні тести: Тестування встановлення

Оновлення:

- Правильність списку файлів, а так само відсутність зайвих файлів:
  - перевірка списку файлів при різних параметрах установки
  - відсутність зайвих файлів
- Зворотна сумісність створюваних даних
  - збереження і коректна робота створених до оновлення даних
  - можливість коректної роботи старих версій додатка з даними, створеними в нових версіях
- Оновлення при запущеному додатку
- Переривання оновлення

Нефункціональні тести: Тестування встановлення

Видалення (Деінсталяція):

- *Коректне видалення програми*:
  - видалення з системного реєстру встановлених в процесі інсталяції бібліотек і службових записів
  - видалення файлів програми
  - видалення / відновлення попередніх файлових асоціацій
  - збереження файлів, створених за час роботи з додатком
- Видалення при запущеному додатку
- Видалення з обмеженим доступом до каталогу програми
- Видалення користувачем без відповідних прав

Нефункціональні тести: Зручність користування

Тестування зручності користування – це метод тестування, спрямований на встановлення ступеня зручності використання, навченості, зрозумілості та привабливості для користувачів розроблювального продукту в контексті заданих умов. Тестування зручності користування дає оцінку рівня зручності використання програми за наступними пунктами:

- продуктивність, ефективність (efficiency) – скільки часу і кроків знадобиться користувачеві для завершення основних завдань додатку, наприклад, розміщення новини, реєстрації, покупки, тощо? (Менше – краще)
- правильність (accuracy) – скільки помилок зробив користувач під час роботи з додатком? (Менше – краще)
- активізація в пам'яті (recall) – як багато користувач пам'ятає про роботу додатка після припинення роботи з ним на тривалий період часу? (Повторне виконання операцій після перерви повинно проходити швидше ніж у нового користувача)
- емоційна реакція (emotional response) – як користувач почувається після завершення завдання - розгублений, отримав стрес? Чи порекомендує користувач систему своїм друзям? (Позитивна реакція – краще)

Нефункціональні тести: Відмова і відновлення

Тестування на відмову і відновлення (Failover and Recovery Testing) перевіряє тестовий продукт з точки зору здатності протистояти і успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками програмного забезпечення, відмовами устаткування або проблемами зв'язку (наприклад, відмова мережі).

Методика подібного тестування полягає в симулювання різних умов збою і наступному вивченні та оцінці реакції захисних систем (чи була досягнута необхідна ступінь відновлення системи після виникнення збою).

Об'єктом тестування в більшості випадків є вельми ймовірні експлуатаційні проблеми, такі як:

- Відмова електрики на комп'ютері-сервері
- Відмова електрики на комп'ютері-клієнті
- Незавершені цикли обробки даних (переривання роботи фільтрів даних, переривання синхронізації).
- Оголошення або внесення в масиви даних неможливих або помилкових елементів.
- Відмова носіїв даних.

У всіх перерахованих вище випадках, по завершенні процедур відновлення, повинен бути досягнутий певний потрібний стан даних:

- Втрата або псування даних в допустимих межах.
- Звіт або система звітів із зазначенням процесів або транзакцій, які не були завершені в результаті збою.

Нефункціональні тести: Конфігураційне тестування

Конфігураційне тестування (Configuration Testing) – спеціальний вид тестування, спрямований на перевірку роботи програмного забезпечення при різних конфігураціях системи (заявлених платформах, підтримуваних драйверах, при різних конфігураціях комп'ютерів тощо).

Залежно від типу проекту конфігураційне тестування може мати різні цілі:

- *Проект по профілізації роботи системи*

Мета Тестування: визначити оптимальну конфігурацію обладнання, що забезпечує необхідні характеристики продуктивності і часу реакції тестованої системи.

- *Проект з міграції системи з однієї платформи на іншу*

Мета Тестування: Перевірити об'єкт тестування на сумісність з оголошеним у специфікації обладнанням, операційними системами та програмними продуктами третіх фірм.

Нефункціональні тести: Конфігураційне тестування

Рівні проведення тестування

На серверному рівні тестується взаємодія програмного продукту з оточенням, в яке воно буде встановлено:

- Апаратні засоби;
- Програмні засоби.

Основний наголос тут робиться на тестування з метою визначення оптимальної конфігурації обладнання, що задовольняє необхідним характеристикам якості (ефективність, портативність, зручність супроводу, надійність). На клієнтському рівні програмні продукти тестуються з позиції його кінцевого користувача і конфігурації його робочої станції:

- Тип, версія і бітність операційної системи
- Тип і версія Web-барузера, у разі якщо тестується Web-додаток
- Тип і модель відео адаптера (при тестуванні ігор це дуже важливо)
- Робота додатка при різних роздільних здатностях екрану
- Версії драйверів, бібліотек тощо.

Пов'язані зі змінами види тестування: Димове

Поняття димове тестування походить від інженерії середовища:

«При введенні в експлуатацію нового обладнання ("заліза") вважалося, що тестування пройшло вдало, якщо з установки не пішов дим.»

Димове тестування застосовується для поверхневої перевірки всіх модулів програми на предмет працездатності та наявності критичних і блокуючих дефектів, що можна швидко знайти. Підвидом димового тестування є ***Тестування збірки*** (Build Verification Test), виконуване на функціональному рівні командою тестування, за результатами якого робиться висновок про те, приймається чи ні встановлена версія програмного забезпечення для тестування, експлуатації або на поставку замовникові.

Пов'язані зі змінами види тестування: Регресійне

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

Як правило, для регресійного тестування використовуються тест кейси, написані на ранніх стадіях розробки і тестування.

Сам по собі термін "регресійного тестування", в залежності від контексту використання може мати різний зміст:

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

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

Регресія побічного ефекту (Side effect regression) – спроба довести, що недавнє зміна коду або даних зламало інші частини розроблювального додатка

Пов'язані зі змінами види тестування: Санітарне

Санітарне тестування або *перевірка узгодженості/справності* (Sanity Testing) – це вузьконапрямлене тестування, достатнє для доказу того, що конкретна функція працює згідно заявлених у специфікації вимог. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини додатка після змін вироблених в ній або навколишньому середовищі. *Зазвичай виконується вручну.*

У деяких джерелах помилково вважають, що санітарний та димове тестування – це одне і теж. На відміну від димового (Smoke testing), санітарне тестування (Sanity testing) направлено вглиб перевірки функції, в той час як димове направлене вшир, для покриття тестами якомога більшого функціоналу в найкоротші терміни.

Рівні Тестування

- Компонентне або Модульне тестування (Component Testing or Unit Testing)
- Інтеграційне тестування (Integration Testing)
- Системне тестування (System Testing)
- Приймальне тестування (Acceptance Testing)

Компонентне або Модульне тестування

Компонентне (модульне) тестування перевіряє функціональність і шукає дефекти в частинах програми, які можуть бути протестовані поокремо (модулі програм, об'єкти, класи, функції і тощо). Зазвичай компонентне (модульне) тестування проводиться викликаючи код, який необхідно перевірити, у середовищах розробки, таких як *фреймворки* (*frameworks – каркаси*) *для модульного тестування* або інструменти для відлагодження. Всі знайдені дефекти, як правило виправляються в коді без формального їх опису в *системі менеджменту багів* (*Bug Tracking System*).

Один з найбільш ефективних підходів до компонентного (модульним) тестування – це підготовка *автоматизованих тестів* до початку основного кодування (розробки) програмного продукту. Це називається *розробка від тестування* (*test–driven development*) або підхід *тестування спочатку* (*test first approach*). Розробка ведеться до тих пір поки всі тести не будуть успішними.

Інтеграційне тестування

Інтеграційне тестування *(Integration Testing)* призначене для перевірки зв'язку між компонентами, а також взаємодії з різними частинами системи (операційною системою, обладнанням або зв'язку між різними системами).

Рівні інтеграційного тестування:

- ***Компонентний інтеграційний рівень*** *(Component Integration testing)*

  Перевіряється взаємодія між компонентами системи після проведення компонентного тестування.

- ***Системний інтеграційний рівень*** *(System Integration Testing)*

  Перевіряється взаємодія між різними системами після проведення системного тестування.

Інтеграційне тестування

Підходи до інтеграційного тестування:

- *Знизу вгору (Bottom Up Integration)*

  Всі низькорівневі модулі, процедури або функції збираються воєдино і потім тестуються. Після чого збирається наступний рівень модулів для проведення інтеграційного тестування.

- *Зверху вниз (Top Down Integration)*

  Спочатку тестуються всі високорівневі модулі, і поступово один за іншим додаються низькорівневі. Всі модулі нижчого рівня симулюються заглушками з аналогічною функціональністю, потім у міру готовності вони замінюються реальними активними компонентами.

- *Великий вибух ("Big Bang" Integration)*

  Всі або практично всі розроблені модулі збираються разом у вигляді закінченої системи або її основної частини, і потім проводиться інтеграційне тестування.

Системне тестування

Основним завданням *системного тестування (System Testing)* є перевірка як функціональних, так і не функціональних вимог в системі в цілому. При цьому виявляються дефекти такі як невірне використання ресурсів системи, непередбачені комбінації даних користувацького рівня, несумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, незручність використання тощо.

Можна виділити два підходи до системного тестування:

- на базі вимог (requirements based)

Для кожної вимоги пишуться тестові випадки (test cases), що перевіряють виконання даної вимоги.

- на базі випадків використання (use case based)

На основі уявлення про способи використання продукту створюються випадки використання системи (Use Cases). Для конкретного випадку використання можна визначити один або більше сценаріїв. На перевірку кожного сценарію пишуться тест кейси (test cases).

Приймальне тестування або приймально-здавальні випробування

Формальний процес тестування, який перевіряє відповідність системи вимогам і проводиться з метою:

- визначення чи задовольняє система приймальним критеріям;
- винесення рішення замовником або іншою уповноваженою особою приймається додаток чи ні.

Приймальне тестування (Acceptance Testing) виконується на підставі набору типових тестових випадків і сценаріїв, розроблених на підставі вимог до даного додатку.

Рішення про проведення приймального тестування приймається, коли:

- продукт досяг необхідного рівня якості;
- замовник ознайомлений з *Планом приймальних Робіт (Product Acceptance Plan)* або іншим документом, де описаний набір дій, пов'язаних з проведенням приймального тестування, дата проведення, відповідальні тощо. Фаза приймального тестування триває до тих пір, поки замовник не виносить рішення про відправлення додатки на доопрацювання або видачі додатка.

Класифікація видів тестування

За об'єктом тестування:

- Функціональне тестування (functional testing)
- Тестування продуктивності (performance testing)
  - ✓ Навантажувальне тестування (load testing)
  - ✓ Стрес-тестування (stress testing)
  - ✓ Тестування стабільності (stability / endurance / soak testing)
- Юзабіліті-тестування (usability testing)
- Тестування інтерфейсу користувача (UI testing)
- Тестування безпеки (security testing)
- Тестування локалізації (localization testing)
- Тестування сумісності (compatibility testing)

Класифікація видів тестування

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)

Класифікація видів тестування

За часом проведення тестування:

- Альфа-тестування (alpha testing)
  - ✓ Тестування при прийомі (smoke testing)
  - ✓ Тестування нової функціональності (new feature testing)
  - ✓ Регресійне тестування (regression testing)
  - ✓ Тестування при здачі (acceptance testing)
- Бета-тестування (beta testing)

За ознакою позитивності сценаріїв:

- Позитивне тестування (positive testing)
- Негативне тестування (negative testing)

За ступенем підготовленості до тестування:

- Тестування з документації (formal testing)
- Тестування ad-hoc або інтуїтивне тестування (ad-hoc testing)

Статичні методи тестування

Статичні методи використовуються при проведенні інспекцій і розгляді специфікацій компонентів без їхнього виконання.

Статичний аналіз направлений на аналіз документів, розроблених на всіх процесах ЖЦ і полягає в інспекції вхідного коду і наскрізного контролю програми.

Інспекція ПС – це статична перевірка відповідності програми заданим специфікаціями, проводиться шляхом аналізу різних представлень результатів проектування (документації, вимог, специфікацій, схем або коду програм) на процесах ЖЦ.

При інспекції програм розглядаються документи робочого проектування на процесах ЖЦ разом з незалежними експертами й учасниками розробки ПС. На початковому процесі проектування інспекція припускає перевірку повноти, цілісності, однозначності, несуперечності і сумісності документів з вимогами до програмної системи. На процесі реалізації системи під інспекцією розуміють аналіз текстів програм на дотримання вимог стандартів і прийнятих керівних документів технології програмування.

Динамічні методи тестування

Динамічні методи тестування використовуються в процесі виконання програм. Вони базуються на графовій структурі, що пов'язує причини помилок з очікуваними реакціями на них. У процесі тестування накопичується інформація про помилки, що використовується при оцінці показників надійності і якості ПС.

Динамічне тестування орієнтоване на перевірку коректності ПС на множині тестів, що проганяються по ПС, з урахуванням зібраних даних на процесах ЖЦ, проведення виміру окремих показників (число відмов, збоїв) тестування для оцінки характеристик якості, зазначених у вимогах, шляхом виконання системи на ЕОМ. Тестування ґрунтується на *систематичних*, *статистичних*, (*імовірнісних*) і *імітаційних* методах.

Систематичні методи тестування поділяються на методи, у яких програма розглядається як «*чорна скринька*» (використовується інформація про розв'язувану задачу), і методи, у яких програма розглядається як «*біла скринька*» з використанням структури програми. Цей вид називають тестуванням з керуванням за даними або керуванням на вході-виході. Ціль – з'ясування обставин, при яких поводження програми не відповідає її специфікації. При цьому ***кількість виявлених помилок у програмі є критерієм якості тестування***.

«Чорна скринька»

Ціль динамічного тестування програм за принципом «чорної скриньки» – виявлення одним тестом максимального числа помилок з використанням невеликої підмножини можливих вхідних даних.

Методи «чорної скриньки» забезпечують:

- еквівалентне розбиття;
- аналіз граничних значень;
- застосування функціональних діаграм, що в поєднанні з реверсивним аналізом дають досить повну інформацію про функціонування тестованої програми.

Еквівалентне розбиття складається з поділу вхідної області даних програми на скінченне число класів еквівалентності так, щоб кожен тест, що є представником деякого класу, був еквівалентний будь-якому іншому тесту цього класу.

Методи тестування за принципом «чорної скриньки» використовуються для тестування функцій, реалізованих у програмі, шляхом перевірки невідповідності між реальною поведінкою функцій і очікуваною поведінкою з урахуванням специфікацій вимог.

«Біла скринька»

Метод «білої скриньки» дозволяє досліджувати внутрішню структуру програми, при чому виявлення всіх помилок у програмі є критерієм вичерпного тестування маршрутів потоків (графа) передач керування, серед яких розглядають:

a) *критерій покриття операторів* – набір тестів у сукупності повинен забезпечити проходження кожного оператора не менше ніж один раз;

b) *критерій тестування областей* (відомий як покриття рішень або переходів) – набір тестів у сукупності повинен забезпечити проходження кожної гілки і виходу, принаймні, один раз.

Тестування за принципом «білої скриньки» орієнтоване на перевірку проходження всіх віток програм за допомогою застосування шляхового й імітаційного тестування.

«Біла скринька»

Шляхове тестування застосовується на рівні модулів і графової моделі програми з вибором тестових ситуацій, підготовки даних і містить у собі тестування наступних елементів:

- операторів, що повинні бути виконані хоча б один раз, без обліку помилок, що можуть залишитися в програмі через велику кількість логічних шляхів і необхідності проходження підмножин цих шляхів;
- шляхів по заданому графу потоків керування для виявлення різних маршрутів передачі керування за допомогою шляхових предикатів, для обчислення якого створюється набір тестових даних, що гарантують проходження всіх шляхів. Проте усі шляхи протестувати неможливо, тому залишаються не виявлені помилки, що можуть виявитися в процесі експлуатації;
- блоків, що розділяють програми на окремі дрібні блоки, які виконуються хоча б один раз або багаторазово при проходженні через шляхи програми, що вміщують сукупність операторів реалізації однієї функції, або на вхідній множині даних, що буде використовуватися при виконанні зазначеного шляху.

Середовища для тестування:
функціональне тестування

- Tricentis Tosca Testsuite
- HP Unified Functional Testing (UFT)
- IBM Rational Functional Tester
- Katalon Studio
- TestPlant eggPlant Functional
- Ranorex

Середовища для тестування:
інтеграційне тестування

- CA Technologies Application Test
- IBM Rational Test Workbench
- Parasoft SOAtest
- SmartBear Ready! API
- Crosscheck Networks SOAPSonar
- Apache JMeter

Середовища для тестування:
тести навантажуання

- Tricentis Flood
- Automation Anywhere Testing Anywhere
- BlazeMeter
- Borland Silk Performer
- CA Technologies Application Test
- HP LoadRunner, Performance Center & StormRunner
- IBM Rational Performance Tester
- SmartBear LoadComplete
- TestPlant eggPlant Performance

Середовища для тестування:
тестування безпеки

- HP Fortify On Demand
- Veracode
- IBM Application Security APPScan

Середовища для тестування:
трекінг помилок

- Bugzilla
- MantisBT
- Atlassian JIRA
- FogCreek FogBugz
- BugAware

Середовища для тестування:
компонентне тестування

- Nunit
- Jmockit
- Junit, TestNG
- Microsoft unit testing Framework
- Emma
- Karma
- Jasmine
- Parasoft

Середовища для тестування:
тестування інтерфейсу

- Ranorex Studio
- Abbot Java GUI Test Framework
- AutoIt UI testing
- eggPlant UI automation testing
- FitNesse
- Selenium
- TestComplete
- Jubula GUI testing tool
- Test Anywhere
- Coded UI
- Unified Functional Testing (UFT)

Верифікація і валідація програм

Верифікація і валідація – це методи аналізу, перевірки специфікацій і правильності виконання програм відповідно до заданих вимог і формального опису програми.

Метод верифікації допомагає зробити висновок про коректність створеної програмної системи при її проектуванні і після завершення її розроблення.

Валідація дозволяє встановити здійснимість заданих вимог шляхом їх перегляду, інспекції і оцінки результатів проектування на процесах ЖЦ для підтвердження того, що здійснюється коректна реалізація вимог, дотримання заданих умов і обмежень до системи.

Верифікації і валідації піддаються:

- компоненти системи, їх інтерфейси (програмні, технічні і інформаційні) і взаємодія об'єктів (протоколи, повідомлення) у розподілених середовищах;
- описи доступу до баз даних, засоби захисту від несанкціонованого доступу до даних різних користувачів;
- документація до системи;
- тести, тестові процедури і вхідні набори даних.

Верифікація і валідація програм

Процес верифікації. Мета процесу – переконатися, що кожен програмний продукт (і/або сервіс) проекту відбиває погоджені вимоги до їхньої реалізації. Цей процес ґрунтується:

- – на стратегії і критеріях верифікації всіх робочих програмних продуктів на ЖЦ;
- – на виконанні дій з верифікації відповідно до стандарту;
- – на усуненні недоліків, виявлених у програмних (робочих, проміжних і кінцевих) продуктах;
- – на узгодженні результатів верифікації з замовником. Процес валідації. Мета процесу
- – переконатися, що специфічні вимоги для програмного продукту виконано, і здійснюється це за допомогою:
- – розробленої стратегії і критеріїв перевірки всіх робочих продуктів;
- – обговорених дій з проведення валідації;
- – демонстрації відповідності розроблених програмних продуктів вимогам замовника і правилам їхнього використання;
- – узгодження із замовником отриманих результатів валідації продукту.

Верифікація об’єктних моделей

*Верифікація **об’єктної моделі*** (ОМ) ґрунтується на специфікації:

- – базових (простих) об'єктів ОМ, атрибутами яких є дані та операції об'єкта – функції над цими даними;
- – об'єктів, які вважаються перевіреними, якщо їх операції використовуються як теореми, що застосовуються над підоб'єктами і не виводять їх з множини станів цих об'єктів.

*Доведення **правильності побудови** ОМ* передбачає:

- – введення додаткових і (або) видалення зайвих атрибутів об'єкта і його інтерфейсів в ОМ, доведення правильності об'єкта ОМ на основі специфікації інтерфейсів і взаємодій з іншими об'єктами;
- – доведення правильності завдання типів для атрибутів об'єкта, тобто правильності того, що вибраний тип реалізує операцію, а множина його значень визначається множиною станів об'єкта.

Верифікація об’єктних моделей

Верифікація інтерфейсів об'єктів ОМ зводиться до доведення правильності передачі типів і кількості даних в параметрах повідомлень про їхні специфікації в мові IDL.

Верифікація моделі розподіленого застосування виконується на основі специфікації SDL (Specification Description Language) моделі перевірки (Model Checking) індуктивних тверджень. Метод перевірки полягає в редукції системи з нескінченним числом станів до системи із скінченного числа станів, а також у доведенні коректності розподіленого застосування за допомогою індуктивних міркувань і системи переходів скінченного автомата.

Основні підходи до верифікації – аксіоматичний і семантичний шлях Model Checking.

Аксіоматичний (за методом Hoare) підхід міститься в описі програми набором аксіом для завдання станів з використанням теорія логіки.

Семантичний підхід ґрунтується на теорії темпоральної логіки Манна для завдання специфікації програм. Аксіоми використовуються для керування семантикою мови специфікації.

Верифікація об’єктних моделей

Основними типами даних специфікації в SDL є наперед визначені і сконструйовані типи даних (масив, послідовність тощо). У мові описуються формули за допомогою предикатів, булевих операцій, кванторів, змінних і модальностей. Семантика їх визначення залежить від можливих послідовностей дій (поведінки), що виконуються специфікацією процесу, а також моменту часу його виконання.

Схема специфікації процесу – це опис умов виконання і діаграм процесів. Вона ініціюється посиланням повідомлення із зовнішнього середовища для виконання. Діаграма процесу складається з описів переходів, станів, набору операцій процесу і переходу до наступного стану. Кожна операція визначає поведінку процесу і спричиняє деяку подію. Логічна формула задає модальність поведінки специфікації і моменти часу. Процес, наданий формальною специфікацією, виконується не детерміновано. Обмін із зовнішнім середовищем відбувається через вхідні і вихідні параметри повідомлень.

Подія. У кожний момент часу виконання процес має деякий стан, який може бути поданий у вигляді знімка, що характеризує деяку подію, яка містить у собі значення змінних, яким відповідають параметри і характеристики станів процесу.

Базові методи доведення правильності програм

Метод Флойда заснований на знаходженні умов для вхідних і вихідних даних і полягає у виборі контрольних точок у програмі, яка доводиться, таким чином, щоб шлях проходження перетинав хоча б одну контрольну точку. Для цих точок формулюються твердження про стан і значення змінних у них (для циклів ці твердження повинні бути істинними при кожному проходженні циклу – інваріанта).

Метод Хоара – це вдосконалений метод Флойда, заснований на аксіоматичному описі семантики мови програмування початкових програм. Кожна аксіома описує зміну значень змінних за допомогою операторів цієї мови. Формалізація операторів переходу і викликів процедур забезпечується за допомогою правил виводу, що містять у собі індуктивні вислови для кожної точки і функції початкової програми.

Метод Маккарті полягає у структурній перевірці функцій, що працюють над структурними типами даних, структур даних і шляхів переходу під час символьного виконання програм. Ця техніка складається з моделювання виконання коду з використанням символів для змінних даних. Тестова програма має вхідний стан, дані і умови її виконання.

Модель доведення програми за твердженнями

Розглядається формальне доведення програми, заданої структурною логічною схемою і сукупністю тверджень, що описуються логічними операторами, комбінаціями змінних (true/false), операціями (кон'юнкція, диз'юнкція тощо) і кванторами загальності й існування. Мета алгоритму програми – побудова для масиву цілих чисел T довжини N (array T[1:N]) еквівалентного масиву T' тієї ж довжини N, що і масив T. Елементи в масиві T' повинні розташовуватися в порядку зростання їхніх значень.

Логічна операція		
Назва	Приклад	Значення
Кон'юнкція	$x \,\&\, y$	x і y
Диз'юнкція	$x * y$	x або y
Суперечність	$\neg x$	не x
Імплікація	$x \rightarrow y$	якщо x то y
Еквівалентність	$x = y$	x рівнозначно y
Квантор загальності	$\forall x\ P(x)$	для всіх x, умова істинна
Квантор існування	$\exists x\ P(x)$	Існує x, для якого $P(x)$істина

Модель доведення програми за твердженнями

Вхідна умова алгоритму задається у вигляді початкового твердження:

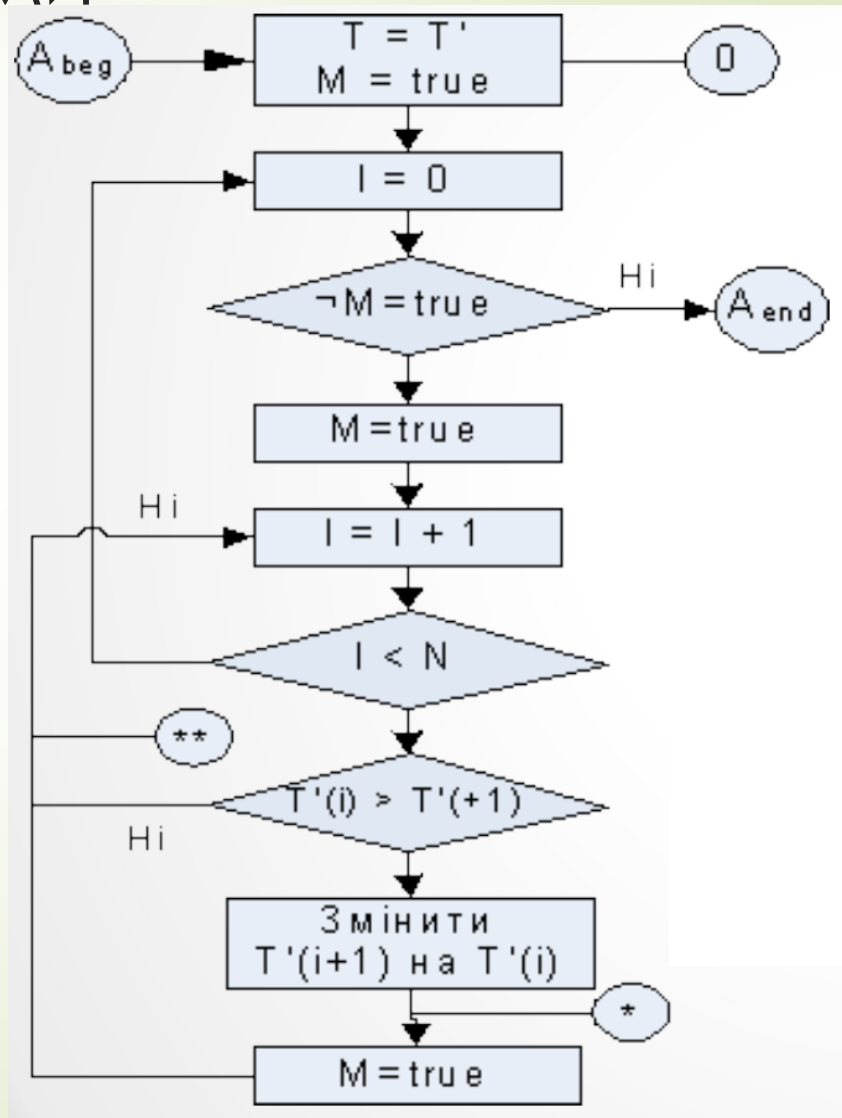
A_{beg}: (T [1:N] – масив цілих) & (T' [1:N] – масив цілих).

Вихідне твердження A_{end} – це кон'юнкція таких умов:

(а) (T– масив цілих) & (T' – масив цілих),

(б) (∀i , якщо i ≤ N, то ∃j (T'(i) £ T' (j))),

(в) (∀i , якщо i < N, то (T' (i) £ T' (i+1)).



Модель доведення програми за твердженнями

Виконання вихідних умов забезпечене порядком і відповідною семантикою операторів перестановки масиву.

Цей алгоритм можна подати у вигляді серії теорем, що доводяться. Починаючи з першого твердження і переходячи від одного перетворення до іншого, визначаємо індуктивний шлях висновку. Якщо одне твердження є правильним, то істинним є й інше. Іншими словами, якщо дано перше твердження A_1 і перша точка перетворення A_2, то перша теорема – $A_1 \rightarrow A_2$. Якщо A_3 – наступна точка перетворення, то другою теоремою буде $A_2 \rightarrow A_3$.

Інакше кажучи, формулюється загальна теорема $A_i \rightarrow A_j$, де A_i та A_j – суміжні точки перетворення. Ця теорема формулюється так: якщо умова істинна в останній точці, то і вихідне твердження $A_k \rightarrow A_{end}$ є істинним.

Далі специфікується твердження типу *if – then*.

Щоб довести, що програма коректна, необхідно послідовно розташувати усі твердження, починаючи з A_1 і закінчуючи A_{end}, цим буде підтверджено істинність вхідної і вихідної умов.

Верифікація композиції компонентів

Метод верифікації композиції компонентів базується на специфікації функцій і часових (temporal) властивостей готових перевірених компонентів (типу reuse) і виконується за допомогою абстракцій моделі перевірки Model Checking.

Загальна компонентна модель (ЗКМ) – це сукупності перевірених специфікацій компонентів, часових властивостей і умов функціонування для асинхронної передачі повідомлень (АПП).

Модель перевірки забезпечує верифікацію програм на надійність шляхом розв'язання наступних задач:

- специфікація компонентів мовою xUML діалекту UML з описом тимчасових властивостей;
- опис функцій reuse компоненти, специфікації інтерфейсу і часових властивостей;
- перевірка властивостей складних компонентів композиційним апаратом.

Верифікація композиції компонентів

Модель компонента в ЗКМ моделі має вигляд:

$$C = (E, I, V, P),$$

де E – початковий опис компонента; I – інтерфейс цього компонента; V – множина змінних, визначених в множині E і пов'язаних з властивостями з множини P; P – часові властивості середовища компонента.

Властивість компонента C включається в P тоді, коли вона перевірена у середовищі і представлена парою $(p, A(p))$ на множині E, де p – властивість компонента C в E, $A(p)$ – множина часових формул з властивостей, визначених на множинах I і V.

Композиція компонентів – це сукупність найпростіших компонентів: (E_0, I_0, V_0, P_0), ..., $(E_{n-i}, I_{n-i}, V_{n-i}, P_{n-i})$, визначених на моделі компонента C.

Верифікація композиції компонентів

Модель обчислень АПП – це обчислювальна модель системи, задана на скінченній множині взаємодіючих процесів, представлених кортежами:

$$P = (X, Q \tilde{N}),$$

де X – множина змінних, кожна з яких має певний тип; P – розширена модель стану; Q – черга повідомлень у порядку надходження; $\tilde{N}$ – множина початкових значень для кожної змінної з X.

Модель стану ПС (Ф, М, Т), де $Ф$ – множина станів; M – множина типів повідомлень; T – набір переходів, визначених на множинах $Ф$ і M.

Асинхронна передача повідомлень АПП викликає чергування переходів станів і дій процесів. Для двох процесів P_1 і P_2 передача повідомлення від P_1 до P_2 містить у собі: тип повідомлення m з множини M для P_2 і відповідні параметри. Коли оператор дії виконується, повідомлення m з параметрами ставиться у чергу до процесу P_2.