**Лекція № 13.** **Сучасний підхід до програмування. Тестування програм.**

**Покоління мов програмування**

1 покоління – 1GL (First Generation Languages).

Початок 1950-х років – мови перших комп’ютерів. Перша мова - асемблер була створена за принципом «одна інструкція – один рядок». Процес програмування передбачав запис усіх алгоритмів безпосередньо машинною мовою.

2 покоління – 2GL.

Кінець 1950-х – початок 1960-х р.р. – розроблено символьний асемблер, в якому з’явилося поняття змінної. Це перша повноцінна мова програмування. Програми застосовуються для створення драйверів устаткування комп’ютерів.

3 покоління – 3GL.

1960-ті р.р. – мови програмування високого рівня. Їх характеристики:

– відносна простота;

– незалежність від конкретного комп’ютера;

– можливість використання потужних синтаксичних конструкцій.

Приклади: Fortran, COBOL, ALGOL, Pascal (Паскаль). Так, наприклад, мова Паскаль була створена в якості навчальної мови програмування в 1968-71 роках швейцарським ученим Ніклаусом Віртом на кафедрі інформатики Стенфордського університету (Цюріх).

4 покоління – 4GL.

Початок 1970-х р.р. до сьогоднішнього часу – мови, призначені для реалізації крупних проектів. Проблемно-орієнтовані мови, що оперують конкретними поняттями вузької галузі. Це вже не мови, а системи програмування чи засоби, орієнтовані на створення великих програмних комплексів. Ці мови чи засоби програмування інтегровані в користувацькі оболонки і мають простий і зручний інтерфейс. Основна сфера їхнього застосування – проектування додатків з використанням баз даних, широке застосування прототипів (структур без даних, екранних форм, звітів), засобів візуального програмування.

Часто відносять: SQL, SGML (HTML, XML), Prolog (для розробки систем штучного інтелекту) та багато інших вузькоспеціалізованих до декларативних мов. Щоправда, низка мов, які відносяться до четвертого покоління, не є мовами програмування як такими. Наприклад SQL є мовою запитів до баз даних, HTML є мовою розмітки гіпертексту, і не є повноцінними мовами програмування, скоріше вони виступають своєрідними спеціалізованими доповненнями до мов програмування. Теж саме стосується XML.

Основна відмінна риса мови четвертого покоління: наближення до людської мови (декларативні мови). Деякі мови мають риси одночасно і третього, і четвертого поколінь.

5 покоління – 5GL.

Мов програмування п'ятого покоління поки що не існує. З середини 1990-х р.р. – до теперішнього часу відбувається розробка систем програмування, призначення яких полягає у перетворенні інструкцій на тексти програм універсальною мовою програмування. Мова п'ятого покоління витіснить чи суттєво посуне мови третього (наприклад, Java) і четвертого покоління (наприклад, SQL) за рахунок значно збільшеної продуктивності праці програміста — у 10-1000 разів. За прогнозами 5GL буде оперувати мета-даними.

## Парадигми програмування

Виділяють чотири основні парадигми програмування: процедурне, функціональне, логічне та об’єктно-орієнтоване програмування.

***Процедурне програмування*.**

Концепція процедурного програмування є історично першою та найбільш близькою до класичного визначення програми. Головний акцент у процедурному програмуванні робиться на обробці, тобто на алгоритмі. В основі програми, побудованої за процедурними принципами, лежить послідовна зміна вхідних даних, поки не буде отриманий результат, причому кожна операція розписується в явному вигляді. Дані, з якими оперує програма, зберігаються в іменованих ділянках оперативної пам'яті, які називаються змінними.  Процедурні мови можна розділити на прості (типу Бейсик, Фортран, Кобол) і блокові (модульні) (типу Паскаль, Модула-2 і С).

***Функціональне програмування***

Виконання програми розглядається як виклик деякої функції, яка, в свою чергу, може викликати інші функції. Найбільш відомим представником цієї парадигми є ЛІСП.

***Логічне програмування***

В основі виконання програми лежить механізм автоматичного доведення теорем на основі логічного виведення. Найбільш відомий представник – Пролог.

***Об'єктно-орієнтоване програмування***

Ця парадигма є на сьогодні найбільш популярною і стрімко розвивається. Об'єктна програма розглядається як сукупність паралельно існуючих сутностей (об'єктів), які взаємодіють між собою. Кожний об'єкт вміє виконувати певні операції та характеризується певною поведінкою. Історично першою об'єктною мовою вважається Smalltalk. Найбільш відомими представниками цієї парадигми є C++, Object Pascal (знаходиться в основі системи візуального програмування Delphi), Java.

***Узагальнене програмування***

Якщо алгоритм можна подати незалежно від деталей його організації, стає можливою парадигма узагальненого програмування. При цьому програміст повинен визначити, які потрібні алгоритми, і параметризувати їх так, щоб вони могли працювати із більшістю потрібних типів і структур даних. Механізм, який забезпечує узагальнене програмування, складається із шаблонних функцій і класів, які параметризуються типами даних.

***Інтегровані системи програмування***

Для того, щоб комп’ютер міг виконувати програму, її потрібно перекласти на машинну мову. Для цього використовують спеціальні програми - транслятори.

Транслятор – це програма, призначена для перекладу тексту програми з однієї мови програмування на іншу. Процес перекладання називається трансляцією.

Розрізняють два типи трансляторів – компілятори та інтерпретатори.

Компілятор – це програма, що призначена для перекладу та наступного запам’ятовування всієї програми, яка написана деякою мовою програмування, у програму в машинних кодах. Перший у світі компілятор був створений у 1951 році. Процес такого перекладання називається компіляцією. Компілятор створює програму в машинних кодах, яка потім виконується. Для повторного виконання програми компілятор вже не потрібен. Досить завантажити з диска в пам’ять комп’ютера скомпільований раніше варіант і виконати його.

Існує інший спосіб поєднання процесів трансляції та виконання програми. Він називається інтерпретацією. Інтерпретатор – це програма, яка призначена для покомандних трансляцій та виконання вихідної програми. Такий процес називається інтерпретацією.

У процесі трансляції відбувається перевірка тексту програми на відповідність до правил мови, яка використовувалася для її опису. Якщо в програмі знайдено помилки, транслятор виводить повідомлення про них на пристрій виведення. Інтерпретатор повідомляє про знайдені помилки після трансляції кожної команди програми. Це значною мірою полегшує процес пошуку та виправлення помилок, але суттєво збільшує час трансляції. Компілятор транслює програму набагато швидше, ніж інтерпретатор, але повідомляє про знайдені помилки лише після завершення компіляції всієї програми. Знайти та виправити помилки в цьому випадку важче. Через це інтерпретатори розраховані, в основному, на мови, що призначені для навчання програмуванню.

Більшість сучасних мов програмування призначені для розробки складних пакетів програм і розраховані на компіляцію. Як правило, програми компілятори і інтерпретатори називаються так само, як і мови програмування.

Інтегроване середовище програмування – це система програмування, що поєднує редактор для зручного введення і редагування програми, транслятор і налагоджувач помилок.

Розглянемо основні етапи обробки компілятором програми мовою С++.

1) Компілятор аналізує, які зовнішні бібліотеки потрібно підключити, розбирає текст програми на складові елементи, перевіряє синтаксичні помилки і в разі їх відсутності формує об'єктний код (у Windows - файл з розширенням .obj). Одержаний на цьому етапі двійковий файл (об'єктний код) не включає в себе об'єктні коди бібліотек, що підключаються.

2) На другому етапі компонувальник підключає до об'єктного коду програми об'єктні коди бібліотек і генерує код програми. Цей етап називається компонуванням або складанням програми. Отриманий на цьому етапі виконуваний код програми можна запускати на виконання.

**Види діяльності зі створення програми**

Процес створення програми в найзагальніших рисах вимагає кількох видів діяльності:

– аналіз задачі й уточнення її постановки;

– проектування програми;

– перевірка програми (тестування);

– передача замовнику (упровадження).

***Аналіз задачі й уточнення її постановки***. Спочатку замовник формулює задачу, яку йому необхідно розв'язати. Задачу зазвичай сформульовано недостатньо точно, навіть може бути, що замовник чітко не уявляє, що саме йому насправді потрібно. Саме тому робота над програмою починається з аналізу задачі й уточнення її постановки. Для цього потрібно заглибитися в предметну область замовника й у діалозі з ним уточнити деякі питання. Зазвичай за результатами аналізу задачі будується спрощена модель предметної області, у термінах якої уточнюється задача. Необхідно також з'ясувати вхідні дані майбутньої програми й результат її роботи над ними. Якщо розв'язання поставленої задачі можливе не за всіх вхідних даних, то слід *визначити поведінку програми на некоректних вхідних даних*. В промисловому виробництві ПЗ замовник формулює свої вимоги, вони розглядаються розробником, аналізуються, перевіряються, узгоджуються. Узгоджені вимоги оформлюються у вигляді специфікації вимог, яка може бути як окремим документом, так і входити в інші, наприклад, до постановки задач. Специфікація вимог є важливим документом на етапі перевірки відповідності ПЗ вимогам замовника.

В умовах навчального процесу замовником виступає викладач. Проте умова задачі все рівно не обов'язково формулюється цілком точно й однозначно. Якщо не провести аналіз задачі й на його підставі не уточнити її постановку, то можна розв'язати не ту задачу. Аналіз задачі дозволяє визначити, якими саме засобами (математичними та програмними) розв'язувати задачу, а уточнена постановка – що саме й у яких ситуаціях має робити програма.

***Проектування програми***. Між написанням твору та програми є певна аналогія. Писати твір починають із плану, який далі розкривають у творі. Так само з програмою: спочатку формулюють її план у вигляді проекту, а потім втілюють цей план у життя– пишуть код (текст) програми. Під час проектування з'ясовують структуру програми, її складові частини та взаємодію між ними. Тут поступово *уточнюють дії* з розв'язання задачі *та їх опис*. З'ясовують і *уточнюють дані*, потрібні для розв'язання задачі. Дуже часто в задачі можна виділити кілька *підзадач* і описати їх розв'язання окремо. Тоді й алгоритм складається зі зв'язаних і узгоджених між собою частин (*допоміжних алгоритмів*), що описують розв'язання підзадач.

Одночасно з проектуванням зазвичай відбувається подальше уточнення постановки задачі й моделі предметної області. На практиці замовник може вирішити дещо змінити умову задачі (причому будь-коли!), і тоді доводиться знов уточнювати постановку та аналізувати задачу. Результатом проектування є модель (проект) програми, що далі поступово перетворюється на текст програми. Ця модель може бути записана, наприклад, у вигляді алгоритму з достатньо абстрактними кроками.

У багатьох програм є чотири основні частини: отримати вхідні дані, обробити некоректні вхідні дані, обробити коректні вхідні дані, вивести результати обробки. Для складніших задач під час проектування потрібно визначати не лише загальний алгоритм роботи програми, але й необхідні структури даних і, можливо, програмні засоби для створення програми та її окремих частин.

***Розробка програми (кодування)***. Коли дії та дані уточнено до вигляду, в якому їх можна виразити мовою програмування, починають розробку програми. Найчастіше програму записують *мовою високого рівня* (інколи окремі її частини – різними мовами).

Розробляючи програму, можна припуститися помилок, що виявляються під час трансляції або виконання програми. Помилки необхідно виявляти й виправляти, тобто налагоджувати програму. ***Налагодження програми*** полягає в тому, що її багаторазово запускають зі *спеціально підібраними* вхідними да-

ними, які допомагають виявити й відшукати помилки, а потім виправити їх.

Проект програми може залежати від можливостей мови програмування та обладнання. У довготривалих проектах трапляється, що під час розробки змінюються програмні й апаратні засоби, замовник уточнює задачу відповідно до нових можливостей свого обладнання, тому всі процеси починаються наново.

***Перевірка програми (тестування)***. У реальних виробничих процесах програму розробляють програмісти, а перевіряють інші спеціалісти – **тестувальники**. Тестування починається, коли програмісти впевнені, що програма (або деяка її частина) є правильною. Задачею тестування є лише встановлення факту відсутності або наявності помилок. Зазвичай спочатку тестувальники виявляють помилки, і цикл "розробка – тестування" повторюється. В умовах навчання ситуація аналогічна, тільки в ролі програміста виступає студент, а тестувальника – спочатку студент, а потім викладач.

***Передача замовнику (упровадження*)**. У найпростішому випадку робота програміста над програмою завершується передачею програми й супроводжувальної документації з її використання замовникові. Для серйозніших програм може знадобитися не просто передати код замовнику, але й установити програму в замовника, зокрема у випадках, коли програма потребує спеціального налаштування параметрів. Інколи потрібно навчити персонал замовника працювати з програмою. Усі ці дії (передавання, установлення, навчання), що дозволяють замовнику користуватися програмою у своїх виробничих процесах, є частиною впровадження програми.

Серйозні програми потребують ***супроводження***. Розробник виправляє помилки, виявлені під час експлуатації програми, модернізує її, передає оновлені варіанти користувачам тощо.

У реальних великих проектах одночасно можуть виконуватися кілька видів робіт. Як тільки постановку задачі більш-менш зрозуміло, можна проектувати програму, обирати програмні засоби для реалізації, писати частини коду й тестувати їх, демонструвати готові частини замовнику, отримувати від нього уточнення й навчати його користуватися програмою. Зазвичай програма та її можливості нарощуються поступово, шляхом багаторазових повернень до аналізу задачі та інших видів роботи.

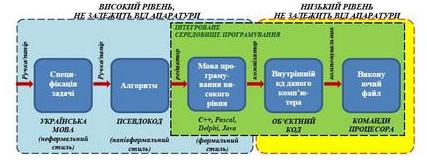
****

Рисунок 1 - Схема розробки програм

**Поняття тестування.**

***Тестування* -** складова процесу розроблення програмного забезпечення (ПЗ), один з методів подальшого поліпшення якості розробленого ПЗ за допомогою виявлення дефектів, що залишилися, не виявлених раніше всіма іншими видами перевірок.

**Тестування -** процес *виконання* програмної системи (або її елементів, зокрема, конкретної програми) з метою *перевірки її відповідності встановленим вимогам* і *виявлення дефектів.*

**Мета тестування: *основна* –** виявити дефекти і перевірити функціональну придатність (тобто виконання всіх функцій)**; *додаткова* –** перевірити зручність застосування та користування, продуктивність та інші характеристики.

Основні підходи до виконання тестування

* деструктивний (негативне, руйнівне тестування)
* конструктивний (позитивне або демонстраційне).

**Основні методи тестування.**

Зазвичай, для проведення тестування застосовуються методи структурного («білий ящик») та функціонального («чорний ящик») тестування.

*При функціональному тестуванні* вихідний код програми не доступний. Суть полягає в перевірці відповідності поведінки програми її зовнішній специфікації. Критерієм повноти тестування вважається перебір всіх можливих значень вхідних даних, що здійснити на практиці надзвичайно важко.

*При структурному тестуванні* текст програми відкритий для аналізу. Суть даного методу полягає в перевірці внутрішньої логіки ПЗ. Повним тестуванням у цьому випадку буде таке, що приведе до перебору всіх можливих варіантів передач керування програми. Число таких шляхів може досягати десятків тисяч, відповідно, виникає питання про створення тестів, що забезпечують дане покриття. Здійснити повне всеохоплююче тестування навіть простої програми вкрай важко, а часом і неможливо в силу обмеженості часу й ресурсів. Отже, необхідно мати певні критерії за якими мають обиратися контрольні приклади та критерії зупинки процесу тестування.

Структурне тестування, або тестування «білого ящика», - це методика аналізу вихідного коду програми. Існує три різновиди структурного тестування: тестування на основі потоку керування програми, на основі потоку даних та мутаційне тестування.

*При використанні тестування на основі потоку керування програми* тестується логіка програми, яка може бути представлена у вигляді графа керування: вершинами є оператори, а гілками - переходи між ними.

*При тестування на основі потоку даних* увага приділяється взаємозв'язкам між змінними. Виділяються вершини, у яких змінна ініціалізується та в яких використовується, і вивчаються переходи й взаємозв'язки між такими вершинами.

*Мутаційне тестування* полягає у внесенні несправностей у вихідний код програми та порівняння роботи вихідної програми та програми мутанта.

Оскільки здійснити вичерпне структурне тестування вкрай важко, необхідно вибрати такі критерії його повноти, які допускали б їхню просту перевірку й полегшували б цілеспрямований підбір тестів.

**Тестування і налагодження програми**

На кожному кроці розробки алгоритму і програми можуть бути допущені помилки. Причини помилок різноманітні: неадекватне розуміння задачі, неправильно складені специфікації, невдале проектування, погане знання мови програмування, механічні помилки тощо. Найсерйознішими є помилки, зроблені на перших кроках. Тому створену програму треба довести до робочого стану – відтестувати та налагодити. Це робиться за допомогою спеціальних тестових даних.  Тестові дані – це вхідні і вихідні дані, підібрані так, щоб для заданих вхідних даних були відомі вихідні. Для вхідних даних програма (модуль), що перевіряється, видає свої вихідні дані. Якщо результат роботи програми не збігається з тестовими  даними, то це значить, що в програмі є помилка; якщо ж збігається, то це не гарантує відсутність помилки -  це значить, що на цих вхідних тестових даних помилка не проявилась. При наявності помилки програму треба налагодити. Для цього необхідно виявити причину  і віднайти місце виникнення помилки, після чого причину і помилку ліквідувати.

Тестування та налагодження програми – складна і трудомістка робота. Необхідно відтестувати роботу кожного модуля, усі передачі керування між модулями. Це найпростіше можна зробити за умови, що проектування і програмування  виконувались з використанням структур керування та структур даних. Для кожного модуля треба перевірити, чи реалізує він ті функції, які сформульовані в специфікації модуля. Тобто, потрібно перевірити, ЩО робить модуль, не розглядаючи того,  ЯК  це робиться. Треба перевірити логіку (алгоритм) модуля:  відтестувати усі операції, усі передачі керування, тобто перевірити усі шляхи потоків інформації. Цим самим перевіряється, як реалізується функція модуля, що визначена в специфікації модуля. Ця робота найпростіше виконується для структурної програми.

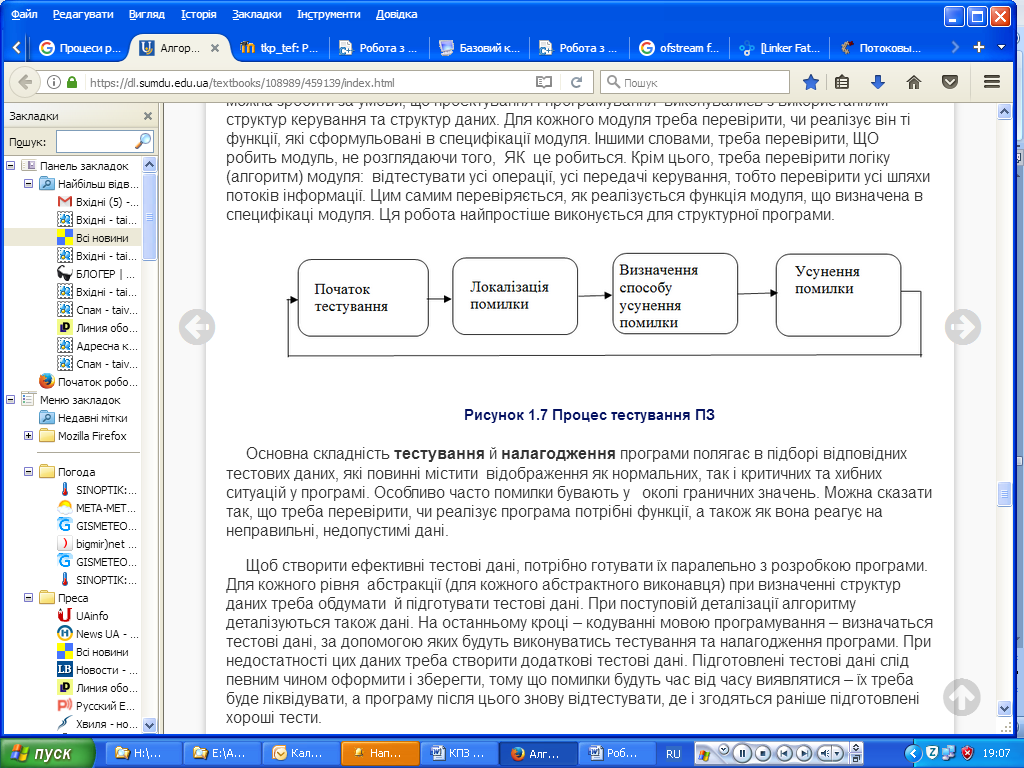


Рисунок 2. Процес тестування ПЗ

Основна складність тестування й налагодження програми полягає в підборі відповідних тестових даних, які повинні містити  відображення як нормальних, так і критичних та хибних ситуацій у програмі, тобто перевірити, чи реалізує програма потрібні функції, а також як вона реагує на неправильні, недопустимі дані. Особливо часто помилки бувають у   околі граничних значень.

Щоб створити ефективні тестові дані, потрібно готувати їх паралельно з розробкою програми. Для кожного рівня  абстракції (для кожного абстрактного виконавця) при визначенні структур даних треба обдумати  й підготувати тестові дані. При поступовій деталізації алгоритму деталізуються також дані. На останньому кроці – кодуванні мовою програмування – визначаться тестові дані, за допомогою яких будуть виконуватись тестування та налагодження програми. При недостатності цих даних треба створити додаткові тестові дані. Підготовлені тестові дані слід певним чином оформити і зберегти, тому що помилки будуть час від часу виявлятися – їх треба буде ліквідувати, а програму після цього знову відтестувати, де і згодяться раніше підготовлені хороші тести.

**Види тестування.**

Тестування правильності. *Мета* – підтвердити, що функції, описані у специфікації вимог до ПЗ, відповідають очікуванням замовника.

Підтвердження правильності ПЗ виконується за допомогою тестів «чорного ящика», що демонструють відповідність вимогам. При виявленні відхилень від специфікації вимог створюється список недоліків. Як правило, відхилення і помилки, виявлені при підтвердженні правильності, вимагають зміни термінів розробки продукту.

Важливим елементом підтвердження правильності є перевірка конфігурації ПЗ. *Конфігурацією ПЗ* називають сукупність усіх елементів інформації, що виробляються в процесі конструювання ПЗ.

*Мінімальна конфігурація ПЗ* включає наступні базові елементи:

1. системну специфікація;
2. план програмного проекту;
3. специфікацію вимог до ПЗ; працюючий або паперовий макет;
4. попереднє керівництво користувача;
5. специфікація проектування;
6. лістинги вихідних текстів програм;
7. план і методику тестування; тестові варіанти і отримані результати;
8. керівництва по роботі та інсталяції;
9. ехе-код виконуваної програми;
10. опис бази даних;
11. керівництво користувача по налаштуванню;
12. документи супроводу; звіти про проблеми ПЗ; запити супроводу; звіти про конструкторських змінах;
13. стандарти та методики конструювання ПЗ.

Перевірка конфігурації гарантує, що всі елементи конфігурації ПЗ правильно розроблені, враховані і достатньо деталізовані для підтримки етапу супроводу в життєвому циклі ПС.

Розробник не може передбачити, як замовник буде реально використовувати ПЗ. Для виявлення помилок, які здатний знайти тільки кінцевий користувач, використовують процес, що включає альфа-і бета-тестування.

Альфа-тестування проводиться замовником в організації розробника. Розробник фіксує всі виявлені замовником помилки і проблеми використання.

Бета-тестування проводиться кінцевим користувачем в організації замовника. Розробник у цьому процесі участі не бере. Фактично, бета-тестування – це реальне застосування ПЗ у середовищі, яке не керується розробником. Замовник сам записує всі виявлені проблеми і повідомляє про них розробнику. Бета-тестування проводиться протягом фіксованого терміну (близько року). За результатами виявлених проблем розробник змінює ПЗ і тим самим готує продукт повністю на базі замовника.

Системне тестування. Даний вид тестування передбачає вихід за рамки області дії програмного проекту і проводиться не тільки програмним розробником. Класична проблема системного тестування – виявлення причини. Вона виникає, коли розробник одного системного елемента звинувачує розробника іншого елемента в причині виникнення дефекту. Для захисту від подібного звинувачення розробник програмного елемента повинен:

1. передбачити засоби обробки помилки, які тестують всі входи інформації від інших елементів системи;
2. провести тести, моделюючі невдалі дані або інші потенційні помилки інтерфейсу ПЗ;
3. записати результати тестів, щоб використовувати їх як доказ невинуватості у випадку «зазначення причини»;
4. взяти участь в плануванні та проектуванні системних тестів, щоб гарантувати адекватне тестування ПЗ.

*Системні тести повинні перевіряти, що всі системні елементи правильно об'єднані і виконують призначені функції.* Розглянемо основні типи системних тестів – тестування відновлення, тестування безпеки, стресове тестування та тестування продуктивності/ працездатності .

Тестування відновлення.

Комп'ютерні системи повинні відновлюватися після відмов і відновлювати обробку в межах заданого часу. У деяких випадках система повинна бути відмовостійкою, тобто відмови обробки не повинні бути причиною припинення роботи системи. В інших випадках системна відмова має бути усунута в межах заданого кванта часу, інакше замовнику наноситься серйозний економічний збиток.

Тестування відновлення використовує самі різні шляхи для того, щоб змусити ПЗ відмовити, і перевіряє повноту виконаного відновлення. При автоматичному відновленні оцінюються правильність повторної ініціалізації, механізми копіювання контрольних точок, відновлення даних, перезапуск. При ручному відновленні оцінюється, чи знаходиться середній час відновлення в допустимих межах.

Тестування безпеки.

Комп'ютерні системи дуже часто є мішенню незаконного проникнення. Під проникненням розуміється широкий діапазон дій: спроби хакерів проникнути в систему зі спортивного інтересу, помста колишніх адміністраторів системи тощо.

Тестування безпеки перевіряє фактичну реакцію захисних механізмів, вбудованих в систему, на проникнення.

У ході тестування безпеки випробувач грає роль взламника. Йому дозволено все:

* спроби дізнатися пароль за допомогою зовнішніх засобів;
* атака на систему за допомогою спеціальних утиліт, які аналізують захист;
* цілеспрямоване введення помилок в надії проникнути в систему в ході відновлення;
* перегляд несекретних даних в надії знайти ключ для входу в систему.

Звичайно, при необмеженому часі і ресурсах вдале тестування безпеки зламає будь-яку систему. Завдання проектувальника системи – зробити ціну проникнення більш високою, ніж ціна одержаної в результаті інформації.

Стресове тестування.

Стресові тести проектуються для нав'язування програмами ненормальних ситуацій. По суті, проектувальник стресового тесту запитує, як сильно можна розхитати систему, перш ніж вона відмовить?

Стресове тестування проводиться при ненормальних запитах на ресурси системи (за кількістю, частотою, розміром-об'ємом).

Приклади:

* швидкість введення даних збільшується прямо пропорційно їх важливості (щоб визначити реакцію вхідних функцій);
* формуються варіанти, що вимагають максимуму пам'яті та інших ресурсів;
* проектуються варіанти, що викликають надмірний пошук даних на диску.

Різновид стресового тестування називається *тестуванням чутливості*. У деяких ситуаціях (зазвичай в математичних алгоритмах) дуже малий діапазон даних, що міститься в межах правильних даних системи, може викликати помилкову обробку або різке зниження продуктивності. Тестування чутливості виявляє комбінації даних, які можуть викликати нестабільність або неправильність обробки.

Тестування продуктивності.

У системах реального часу та вбудованих системах неприпустимо ПЗ, яке реалізує необхідні функції, але не відповідає вимогам продуктивності.

Тестування продуктивності перевіряє швидкість роботи ПЗ в комп'ютерній системі. Продуктивність тестується на всіх кроках процесу тестування. Навіть на рівні елемента при проведенні тестів «білого ящика» може оцінюватися продуктивність індивідуального модуля. Тим не менш, поки всі системні елементи не об'єднаються повністю, не може бути встановлена дійсна продуктивність системи. Іноді тестування продуктивності поєднують зі стресовим тестуванням. При цьому нерідко потрібен спеціальний апаратний і програмний інструментарій. Наприклад, часто потрібно точне вимірювання використовуваного ресурсу (процесорного циклу і т. д.). Зовнішній інструментарій регулярно відстежує інтервали виконання, реєструє події (наприклад, переривання) і машинні стану. За допомогою інструментарію випробувач може виявити стани, які призводять до деградації і можливих відмовлень системи.

**Налагодження програми**.

Налагодження є наслідком успішного тестування. Це означає, що якщо тестовий варіант виявляє помилку, то процес налагодження знищує її.

Отже, процесу налагодження передує виконання тестового варіанту. Його результати оцінюються, реєструється невідповідність між очікуваним і реальним результатами. *Процес налагодження намагається зіставити симптом з причиною, внаслідок чого призводить до виправлення помилки*. Можливі два результати процесу налагодження:

1. причина знайдена, виправлена, знищена;
2. причина не знайдена.

У другому випадку відладчик може припускати причину. Для перевірки цієї причини він просить розробити додатковий тестовий варіант, який допоможе перевірити припущення. Таким чином, запускається ітераційний процес корекції помилки.

Можливі різні *види прояву помилок*:

1. програма завершується нормально, але видає неправильні результати;
2. програма зависає;
3. програма завершується з переривання;
4. програма завершується, видає очікувані результати, але збережені дані зіпсовані (це самий неприємний варіант).

Розрізняють дві групи методів налагодження:

* аналітичні;
* експериментальні.

*Аналітичні методи* базуються на аналізі вихідних даних для тестових прогонів. Експериментальні методи базуються на використанні допоміжних засобів налагодження (налагоджувальні друку, трасування), що дозволяють уточнити характер поведінки програми при тих чи інших вихідних даних.

Загальна стратегія налагодження – зворотне проходження від поміченого симптому помилки до вихідної аномалії (місцем в програмі, де помилка здійснена).

У простому випадку місце прояви симптому і помилковий фрагмент збігаються. Але найчастіше вони далеко відстоять один від одного.

*Мета налагодження* – знайти оператор програми, при виконанні якого правильні аргументи приводять до неправильних результатів. Якщо місце прояви симптому помилки не є шуканої аномалією, то один з аргументів оператора повинен бути невірним. Тому треба перейти до дослідження попереднього оператора, який виробив цей невірний аргумент. У підсумку покрокове зворотне простежування призводить до шуканого помилкового місця.

У різних методах простежування організовується по-різному. В аналітичних методах – на основі логічних висновків про поведінку програми.

Основна перевага аналітичних методів налагодження полягає в тому, що вихідна програма залишається без змін.

В *експериментальних методах* для простежування виконується:

1. Видача значень змінних у зазначених точках.
2. Трасування змінних (видача їх значень при кожній зміні).
3. Трасування потоків управління (імен викликаються процедур, міток, на які передається управління, номерів операторів переходу).

Перевага експериментальних методів налагодження полягає в тому, що основна рутинна робота з аналізу процесу обчислень перекладається на комп'ютер. Багато транслятори мають вбудовані засоби налагодження для отримання інформації про хід виконання програми.

Недолік експериментальних методів налагодження – в програму вносяться зміни, при виключенні яких можуть з'явитися помилки. Втім, деякі системи програмування створюють спеціальний налагоджувальний екземпляр програми, а в основній екземпляр не втручаються.

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Література*

1. Єжова Л. Ф. Алгоритмізація і програмування процедур обробки інформації: Навч.-метод. посіб. для самост. вивч. дисц. К.: КНЕУ, 2000.
2. Вступ до програмування мовою С++.  Організація обчислень: навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставовський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с. с.: іл. ISBN (укр.) . URL:  [http://csc.knu.ua/uk/library/books/belov-24.pdf](https://www.blogger.com/)
3. Зелковиц М., Шоу А., Геннон Дж. Принципы разработки программного обеспечения. — М.: Мир, 1982. — 368 с. URL: [http://computersbooks.net/index.php?id1=4&category=teoriyaprogramirovaniya&author=zelkovic-m&book=1982](https://www.blogger.com/)
4. Куприянова Л.М. Программирование, алгоритмические языки и вычислительная математика: Учеб. пособие. — М.: Финансы и статистика, 1985. —223 с.

*Контрольні запитання для самоперевірки*.

1. Визначте та охарактеризуйте покоління мов програмування.
2. До якого покоління мов програмування належить С++?
3. Визначте та охарактеризуйте основні парадигми мов програмування.
4. Назвіть і опишіть основні види роботи зі створення програми.
5. Чому, створюючи програму, не можна відразу починати писати її текст?
6. Чим відрізняється компіляція від інтерпретації?
7. Надайте визначення тестування та сформулюйте його мету?
8. З якими видами діяльності тісно пов‘язане тестування?
9. В чому полягає функціональне тестування?
10. В чому полягає структурне тестування?
11. Чим різниться Альфа-тестування та Бета-тестування?
12. В чому сутність процесу налагодження програми?

*Контрольні запитання для надання письмових відповідей*.

1. Визначте види прояву помилок, які у вас зустрічалися? Яка була причина їх появи. Надайте фрагмент програмного коду.
2. Який вид та метод тестування Ви використовуєте при налагодженні своїх програм. Відповідь поясніть.