**Лабораторна робота № 10-2. Підготовка тестів для перевірки ПЗ – Підготовка тестів "білого ящика" для перевірки ПЗ**

**Мета:** Навчитися формувати перелік тестів для перевірки за методикою "білого ящика" для перевірки ПЗ

**Методичні рекомендації до виконання лабораторної роботи.**

1. Опрацювати теоретичні відомості.
2. Розглянути надані в теоретичних відомостях види тестування та розподілити та надати конкретні приклади відповідно до вашого проекту. Заповнити таблицю:

|  |  |  |
| --- | --- | --- |
| №№ | Види тестування | Приклади тестування |
|  |  |  |
|  | … |  |
|  |  |  |

1. Робота повинна бути виконана згідно критеріїв оформлення документації та повинна містити

* Назва лабораторної роботи.
* Прізвище, група
* Назва проекту.
* Результати роботи оформлюються у вигляді таблиці:

По закінченню практичну роботу потрібно здати на перевірку викладачеві, надіславши електронною поштою на адресу [**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com) . Якщо викладач знаходить помилки чи неточності, він може повернути роботу на доопрацювання.

Файл з роботою повинен мати назву в такому форматі:

**QA<Номер групи><Номер лекції / практичної / лабораторної>[-<Номер завдання>][літера позначення типу роботи L – лекція, P –практична, R – лабораторна]<Прізвище англійською>**.. Наприклад, **QA4101Р**buts.doc.

Не копіюйте фрагментів з різних інформаційних джерел, подумайте і викладіть свою точку зору. При наявності робіт-"близнюків" відповідь буде зараховуватися першому за часом надсилання.

Тему в заголовку листа записати

**QA <Номер групи>-ЛР<Номер лабораторної>-<Прізвище >**

**Строк виконання цієї роботи ІПЗ-41 – 19.04.2024**

**Контрольні питання**

**1.** Які методи тестування Ви знаєте?

**2.** Опишіть класифікацію методів.

**3.** Як відбувається тестування потоку даних.

**4.** Як відбувається  тестування циклів.

**5.** Охарактеризуйте стратегії тестування.

**6.** Назвіть особливості тестування «білого ящика».

**7.** Які переваги тестування?

**8.** Які недоліки тестування?

**9.** Які Ви знаєтеспособи тестування?

10. Охарактеризуйте етап реалізації і тестування програмного продукту.

11. Назвіть критерії вибору тестів.

12. Перерахуєте властивості тестів.

13. Приведіть критерії надійності програм.

14. У чому полягає оцінка надійності програм?

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

***Види тестування***

Тестування програмного забезпечення включає цілий комплекс дій, аналогічних послідовності процесів розробки програмного забезпечення. У нього входять:

* постановка завдання для тестів;
* проектування тестів;
* написання тестів;
* тестування тестів;
* виконання тестів;
* вивчення результатів тестування.

Найбільш важливим є проектування тестів. Існують різні підходи до проектування тестів.

Перший полягає в тому, що тести проектуються на основі зовнішніх специфікацій програм і модулів або специфікацій сполучення модуля з іншими модулями, програма при цьому розглядається як "чорний ящик". Сенс тесту полягає в тому, щоб перевірити, чи відповідає програма зовнішнім специфікаціям. При цьому зміст модуля не має значення. Такий підхід дістав назву – стратегія "чорного ящика".

Другий підхід – стратегія "білого ящика", заснований на аналізі логіки програми. При такому підході тестування полягає в перевірці кожного шляху, кожній гілці алгоритму. При цьому зовнішня специфікація до уваги не береться.

Жоден з цих підходів не є оптимальним. Реалізація тестування методом "чорного ящика" зводиться до перевірки усіх можливих комбінацій вхідних даних. Неможливо протестувати програму, подаючи на вхід нескінченну безліч значень, тому обмежуються певним набором даних. При цьому виходять з максимальної віддачі тесту в порівнянні з витратами на його створення. Вона вимірюється вірогідністю того, що тест виявить помилки, якщо вони є в програмі. Витрати вимірюються часом і вартістю підготовки, виконання і перевірки результатів тестів.

Тестування методом "білого ящика" також не дає 100%-ної гарантії того, що модуль не містить помилок. Навіть якщо припустити, що виконані тести для усіх гілок алгоритму, не можна з повною упевненістю стверджувати, що програма відповідає її специфікаціям. Наприклад, якщо вимагалося написати програму для обчислення кубічного кореня, а програма фактично обчислює корінь квадратний, то реалізація буде абсолютно неправильною, навіть якщо перевірити усі шляхи. Друга проблема – відсутні шляхи. Якщо програма реалізує специфікації не повністю (наприклад, відсутня така спеціалізована функція, як перевірка на негативне значення вхідних даних програми обчислення квадратного кореня), ніяке тестування існуючих шляхів не виявить такої помилки. І нарешті, проблема залежності результатів тестування від вхідних даних. Наприклад, якщо для визначення рівності трьох чисел програмується вираження виду:

*IF (А + В+С) / 3 = А,*

то воно буде вірним не для усіх значень *А*, *В* і *С* (помилка виникає у тому випадку, коли з двох значень *В* або *С* одне більше, а інше на стільки ж менше *А*). Якщо концентрувати увагу тільки на тестуванні шляхів, немає гарантії, що ця помилка буде виявлена.

Таким чином, повне тестування програми неможливо, тобто ніяке тестування не гарантує повну відсутність помилок в програмі. Тому необхідно проектувати тести так, щоб збільшити вірогідність виявлення помилки в програмі.

**Стратегії тестування, базовані на коді**

*До цих стратегій відносять методи структурного тестування.*

У традиційній класифікації структурні методи відносять до методів «білого ящика». За цими методами структура програми представляється у вигляді *направленого графу*, в якому ідентифікуються потоки управління або потоки даних.

Категорії методів:

* тестування потоку управління;
* тестування потоку даних;
* тестування циклів.

З кожним методом зв’язується критерій покриття**,** який визначає ступінь повноти тестування за цим методом.

У методах структурного тестування ці критерії називаються структурними критеріями покриття.

**Критерії покриття рядків, розгалужень, покриття умов**

В методах тестування потоку управління, дані з вхідного простору вибираються так, щоб забезпечити максимальне покриття коду.

Покриття коду визначає повноту перевірки модуля набором тестів.

*Основні методи наведено нижче:*  
*1. Тестування рядків.* Критерій покриття С0 (покриття рядків).   
*2. Тестування розгалужень.*Критерій покриття С1 (покриття розгалужень).  
*3. Тестування логічних умов*. Критерій покриття С2.

Ці методи використовуються на рівні *модульного тестування* (окремих методів класу).

*1. Тестування рядків.*(Statement Coverage). Вибираються дані, що забезпечують виконання всіх рядків (операторів) програми. Цей метод дає найслабший критерій покриття, так зване «покриття рядків», і прийнятний для програм, *що не містять логічних умов і циклів.*

*2. Тестування розгалужень.* (Decision Coverage). Вибираються дані, що забезпечують виконання шляхів, які виділяються в програмі за допомогою логічних умов, що приймають значення *True* і *False*.

3. *Тестування логічних умов*.

Якщо розгалуження в програмі утворюються в результаті виконання складних логічних умов, дані для їх тестування повинні вибиратися так, щоб перевірити всі значення логічних умов. Наприклад, для повної перевірки поданої нижче логічної умови:

if (A..................else ..................endif

необхідно вже чотири тести (один — для випадку, коли умова виконується, і три — для решти випадків):

1) A < B і С = 1.

2) A < B і С ≠ 1.

3) A ≥ B і C = 1.

4) A ≥ B і С ≠ 1.

Цей метод дає найповніший критерій покриття коду програми, який називається критерієм «*логічні умови»*. Знову таки, для задоволення критерію покриття рядків було б достатньо одного тесту, а для покриття розгалужень – двох.

**Метод покриття операторів**

Метою цього методу тестування є виконання кожного оператора програми хоча би один раз.

**Приклад.**

Якщо для тестування задати значення змінних А = 2, В = 0, Х = 3, буде реалізований шлях *асе*, тобто кожен оператор програми виконається один раз (рис. 1, а). Але якщо внести до алгоритму помилки - замінити в першій умові and на or, а в другій Х > 1 на Х < 1 (рис. 1, б), жодна помилка не буде виявлена (таблиця 1). Крім того, шлях abd взагалі не буде охоплений тестом, і якщо в нім є помилка, вона також не буде виявлена. У таблиці. 1 очікуваний результат визначається по блок-схемі на рис. 1, а, а фактичний - по рис..1, б.

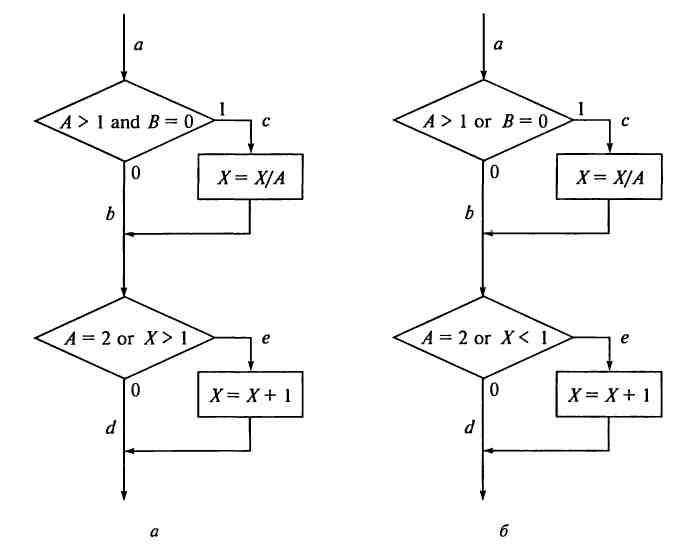


Рисунок 1 Приклад алгоритму програми: а - правильний; б - з помилкою

Як видно з цієї таблиці, жодна з внесених до алгоритму помилок не буде виявлена.

*Таблиця 1* Результат тестування методом покриття операторів

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Очікуваний результат | Фактичний результат | Результат тестування |
| *А = 2, В = 0, Х = 3* | *Х = 2,5* | *Х = 2,5* | Неуспіх |

**Метод покриття рішень (покриття переходів)**

Згідно з методом покриття рішень кожен напрям переходу має бути реалізований, принаймні, один раз. Цей метод включає критерій покриття операторів, оскільки при виконанні усіх напрямів переходів виконаються усі оператори, що знаходяться на цих напрямах.

Для програми, приведеної на мал. 1, покриття рішень може бути виконане двома тестами, що покривають шляхи *{асе, abd},* або *{acd, abe}.* Для цього виберемо наступні вихідні дані: *{А =* 3*, В =* 0*, Х =* 3*}* - в першому випадку і *{А* = 2, *В* = 1, *Х* = 1*}* - в другому. Проте шлях, де Х не міняється, буде перевірений з вірогідністю 50 %: якщо в другій умові замість умови Х > 1 записати Х < 1, то помилка не буде виявлена двома тестами.

Результати тестування приведені в таблицю 2.

*Таблиця 2.* Результат тестування методом покриття рішень

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Очікуваний результат | Фактичний результат | Результат тестування |
| *А = 3, В = 0, Х = 3* | *Х = 1* | *Х = 1* | Неуспіх |
| *А = 2, В = 1, Х = 1* | *Х=2* | *Х = 1,5* | Успішно |

**Метод покриття умов**

Цей метод може дати кращі результати в порівнянні з попередніми. Відповідно до методу покриття умов записується число тестів, достатнє для того, щоб усі можливі результати кожної умови в рішенні виконувалися, принаймні, один раз.

У даному прикладі маємо чотири умови: {А>1, В = 0}, {А = 2, Х > 1}. Отже, вимагається достатнє число тестів, таке, щоб реалізувати ситуації, де А > 1, А ≤ 1, В = 0 і В ≠ 0 в точці а і А = 2, А ≠ 2, Х > 1 і Х ≤ 1 в точці b. Тести, що задовольняють критерію покриття умов (таблиця 3), і відповідні ним шляхи:

а) А = 2, В = 0, Х = 4 *асе*;

б) А = 1, B = 1, X = 0 *abd*.

*Таблиця 3.* Результати тестування методом покриття умов

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Очікуваний результат | Фактичний результат | Результат тестування |
| *А = 2, В = 0, Х = 4* | *Х = 3* | *Х = 3* | Неуспіх |
| *А = 1, В = 1, Х = 0* | *Х = 3* | *Х = 1* | Успішно |

**Метод покриття рішень/умов**

Критерій покриття рішень/умов вимагає такого достатнього набору тестів, щоб усі можливі результати кожної умови виконувалися принаймні один раз, усі результати кожного рішення виконувалися принаймні один раз і, крім того, кожній точці входу передавалося управління принаймні один раз.

Недоліки методу:

* не завжди можна перевірити усі умови;
* неможливо перевірити умови, які приховані іншими умовами;
* недостатня чутливість до помилок в логічних виразах.

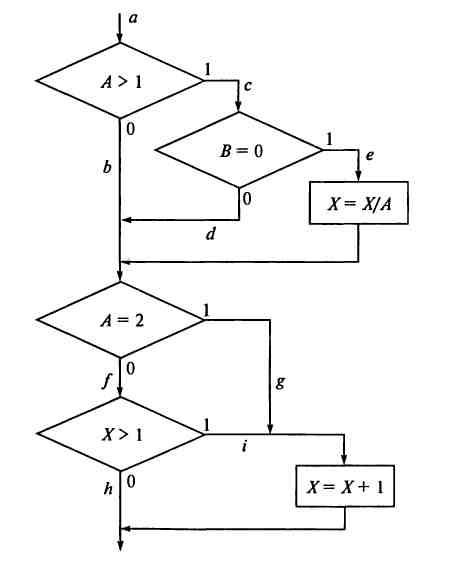
Так, в даному прикладі два тести методу покриття умов

а) А = 2, В = 0, Х = 4 *асе*;

б) А = 1, В = 1, X = 0 *abd*

відповідають і критерію покриття рішень/умов. Це є слідством того, що одні умови приведених рішень приховують інші умови в цих рішеннях. Так, якщо умова А > 1 буде помилковою, транслятор може не перевіряти умови В = 0, оскільки при будь-якому результаті умови В = 0 результат рішення ((А > 1)&(В = 0)) набуде значення неуспіх. Тобто у варіанті на рис. 1 не усі результати усіх умов виконаються в процесі тестування.

Розглянемо реалізацію того ж прикладу на рис. 2. Найбільш повне покриття тестами в цьому випадку здійснюється так, щоб виконувалися усі можливі результати кожного простого рішення. Для цього треба покрити шляхи *aceg* (тест А = 2, В = 0, Х = 4), *acdfh* (тест А = 3, В = 1, Х = 0), *abfh* (тест А = 0, В = 0, Х= 0), *abfi* (тест А = 0, В = 0, Х = 2).

Рисунок 2. Приклад алгоритму програми

Протестувавши алгоритм на рис. 2, неважко переконатися в тому, що критерії покриття умов і критерії покриття рішень/умов недостатньо чутливі до помилок в логічних виразах.

**Метод комбінаторного покриття умов**

Критерій комбінаторного покриття умов задовольняє також і критеріям покриття рішень, покриття умов і покриття рішень/умов.

Цей метод вимагає створення такого числа тестів, щоб усі можливі комбінації результатів умови в кожному рішенні виконувалися принаймні один раз. За цим критерієм в даному прикладі мають бути покриті тестами слідуючи вісім комбінацій:

1. А > 1, В = 0. 5. А = 2, Х > 1.

2. А > 1, В ≠ 0. 6. А = 2, Х ≤ 1.

3. А ≤ 1, В = 0. 7. А ≠ 2, Х > 1.

4. А ≤ 1, В ≠ 0. 8. А ≠ 2, Х ≤ 1.

Для того, щоб протестувати ці комбінації, необов’язково використовувати усі 8 тестів. Фактично вони можуть бути покриті чотирма тестами (таблиця 4):

* *А = 2, В = 0, Х = 4* {покриває 1, 5};
* *А = 2, В = 1, Х = 1* {покриває 2, 6};
* *А = 0,5, В = 0, Х = 2* {покриває 3, 7};
* *А = 1, В = 0, Х = 1* {покриває 4, 8}.

*Таблиця 4.* Результати тестування методом комбінаторного покриття умов

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Очікуваний результат | Фактичний результат | Результат тестування |
| *А = 2, В = 0, Х = 4* | *Х = 3* | *Х = 3* | Неуспіх |
| *А = 2, В =1, Х = 1* | *Х = 2* | *Х = 1,5* | Успішно |
| *А = 0,5, В = 0, Х = 2* | *Х = 3* | *Х = 4* | Успішно |
| *А = 1, В = 0, Х = 1* | *Х = 1* | *Х = 1* | Неуспіх |

**Тестування циклів**

*Тестування циклів*. Тести розробляються для перевірки кожного циклу при граничних значеннях змінних циклу і всередині них.

Цей метод дає критерій покриття, який називається «*всі цикли»*.

***Приклад 1***

using *System;*

namespace C\_Sharp\_Programming

class Cycles

{public static void Main()

{int nl, n2;

nl = 0;

n2 = nl + 1;

while(nl < n2)

{Console WriteLine ("nl, n2 = ", nl, n2 );

n1++;

n2++;

}}}}

У цьому прикладі умова (nl < n2) завжди вірна. Тому вихід з циклу неможливий. Отже, програма увійде до режиму вічного циклу. Такі помилки є критичними, тому слід дуже уважно перевіряти умови виходу з циклу.

Примітка.

Цикл while є найбільш “небезпечним” з точки зору тестування.

***Приклад 2***. Написати і виконати тест для знаходження помилки в методі піднесення до ступеня числа x.

//Варіант 1

// Метод обчислення ступеню n числа, xn(для n>0)

static public double Power(double x, int n)

{double z=1;

for (int i=1;n>=i;i++)

{z = z\*x;

}return z;}

При n<1 метод буде видавати неправильний результат, наприклад:

Power(2,—1) = 2, але ніякої діагностики не виводиться.

Отже, метод не стійкий до помилок у вхідних даних.

// Варіант 2

// Метод обчислення ступеню n числа xn(для n>0)

static public double PowerNonNeg(double x, int n)

{double z=1;

if (n>0)

{for (int i=1;n>=i;i++)

{z = z\*x;}

}

else Console.WriteLine("Помилка! Показник ступеню повинен бути >=0.");

return z;

}

Для цього прикладу розробимо тести для перевірки розгалужень і циклу.

Метод приймає 2 параметри: *double x, int n* і повертає значення xn. Значення n>0. Для спрощення тестування обмежимо n=100.

Крім перевірки вхідних параметрів, корисно знайти обмеження для максимального значення xn .

Написати виклик цього методу з методу main().

За допомогою цього набору тестів ми перевірили всі рядки коду. Критерій С0=100%, розгалуження if (). Критерій C1=100%. Цикл при min, max, проміжних значеннях змінної циклу.

Але цей набір недостатній для виявлення помилки виходу за розрядну сітку при дуже великих значеннях xn.

Модифікуємо метод для того, щоб він був стійким до помилок у вхідних даних.

Введемо обмеження на дані, які будемо аналізувати.

1. Значення числа і ступеня повинні бути цілими.

2. Значення числа, що підноситься до ступеня, повинні знаходитись в діапазоні – [0..999].

3. Значення ступеня повинні знаходитись в діапазоні – [1..100].

Якщо числа, що подаються на вхід, знаходитись за межами вказаних діапазонів, то повинне видаватися повідомлення про помилку.

У метод Main вставимо обробку виключень (блок try{} catch (Exception e))

// Метод обчислення ступеню n числа x

static public double Power(int x, int n)

{double z=1;

for (int i=1;n>=i;i++)

{z = z\*x;

}return z;

}[STAThread]

static void Main(string[] args)

{

int x;

int n;

try

{

Console.WriteLine("Enter x:");

x=Convert.ToInt32(Console.ReadLine());

if ((x>=0) & (x<=999))

{

Console.WriteLine("Enter n:");

n=Convert.ToInt32(Console.ReadLine());

if ((n>=1) & (n<=100))

{

Console.WriteLine("The power n of x is {0}",

Power(x,n));

Console.ReadLine();

}

Else

{

Console.WriteLine("Error : n must be in [1..100]");

Console.ReadLine();

}}

Else

{Console.WriteLine("Error : x must be in [0..999]");

Console.ReadLine();

}}

catch (Exception e)

{

Console.WriteLine("Error : Please enter a numeric argument.");

Console.ReadLine();

}}

Загальні правила проектування тестів.

Тести розробляються спочатку для перевірки правильних вхідних даних і умов. Обов’язково для граничних даних і після цього для неправильних вхідних даних і умов.

**Тестування потоків даних**

У *методі тестування потоку даних* тестові дані вибираються так, щоб відслідкувати шляхи кожної змінної в програмі від призначення значень до самого останнього використання (для всіх змінних). Цей метод вимагає великої кількості тестів, тому на практиці трасуються найбільш критичні значення змінних. Особливий інтерес, наприклад, представляють значення змінних, що беруть участь в операціях ділення, умови і цикли, виконання яких залежить від значень змінних. Метод тестування потоку даних може також застосовуватися для пошуку помилок часу виконання, зв’язаних з використанням пам’яті, які важко знаходяться.

Приклад: написати метод, розробити для нього тести і виконати їх.

Перевіритипри x=0.

static void main()

{

double x,y;

Console.WriteLine("Введіть значення x (double)");

x= double.Parse(Console.ReadLine());

y = Math.Sin(x)/x;

Console.WriteLine ("y = " + y);

}

При x=0 буде помилка ділення на 0.

На практиці в одному модулі (методі, процедурі) зустрічаються і цикли і умовні оператори. Тому розробляються комбіновані набори тестів для покриття всіх критерієв (і циклів і розгалуджень).

*Які типи помилок не можна виявити методами структурного тестування:*

1.     Невідповідність специфікації.

2.     Помилки в інтерфейсі користувача.

3.     Реакція на обробку подій.

4.     Зручність використання.

**Особливості тестування «білого ящика»**

***Відома:*** внутрішня структура програми.

***Досліджуються:*** внутрішні елементи програми і зв'язки між ними.

Об'єктом тестування тут є не зовнішня, а внутрішня поведінка програми. Перевіряється коректність побудови всіх елементів програми і правильність їх взаємодії один з одним. Зазвичай аналізуються керуючі зв'язки елементів, рідше — [інформаційні](http://ua-referat.com/%D0%86%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D1%96%D1%8F) зв'язки. Тестування за принципом "білого ящика" характеризується ступенем, в якому тести виконують або покривають логіку (вихідний [текст](http://ua-referat.com/%D0%A2%D0%B5%D0%BA%D1%81%D1%82)) програми.

Зазвичай тестування «білого ящика» засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреною, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління.

*У цьому випадку формуються тестові варіанти, в яких:*

* Гарантується перевірка всіх незалежних маршрутів програми.
* Знаходяться гілки True, False для всіх логічних рішень.
* Виконуються всі цикли (в межах їх кордонів і діапазонів).
* Аналізується правильність внутрішніх структур даних.

**2.   Переваги та недоліки тестування**

*Недоліки тестування «білого ящика»:*

* Кількість незалежних маршрутів може бути дуже велика.
* Повне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.
* У програмі можуть бути пропущені деякі маршрути.
* Не можна виявити помилки, поява яких залежить від даних.

*Переваги тестування «білого ящика» пов’язані з тим, що принцип «білого ящика» дозволяє врахувати особливості програмних помилок:*

* Кількість помилок мінімально в «центрі» і максимально на «периферії» програми.
* Попередні припущення про ймовірність потоку керування або даних у програмі часто бувають некоректними. У результаті типовим може стати маршрут, модель обчислень за яким опрацьована слабо.
* Під час запису алгоритму програмного забезпечення у вигляді тексту на мові [програмування](http://ua-referat.com/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) можливе внесення типових помилок трансляції ([синтаксичних](http://ua-referat.com/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81) і семантичних).
* Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

Кожна з цих причин є аргументом для проведення тестування за принципом «білого ящика».

**3.   Способи тестування**

***Спосіб тестування базового шляху***

*Тестування базового шляху* — це спосіб, який заснований на принципі «білого ящика». Автор цього способу — Том Мак-Кейб (1976).

*Спосіб тестування базового шляху дає можливість:*

* Отримати оцінку комплексної складності програми.
* Використовувати цю оцінку для визначення необхідної кількості тестових варіантів.

Тестові варіанти розробляються для перевірки базових безліч шляхів (маршрутів) у програмі. Вони гарантують одноразове виконання кожного оператора програми під час тестування.

*Потоковий граф*

Для представлення програми використовується потоковий граф.

*Перерахуємо його особливості:*

* Граф будується відображенням керуючої структури програми. У ході відображення закривають дужки умовних операторів і операторів циклів розглядають як окремих (фіктивних) операторів.
* Вузли (вершини) потокового графу [відповідають](http://ua-referat.com/%D0%92%D1%96%D0%B4%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D1%8C) лінійним ділянкам програми, включають один або кілька операторів програми.
* Дуги потокового графу відображають потік управління у програмі (передачі управління між операторами). Дуга — це орієнтоване ребро.

Розрізняють операторні і предикатні вузла. З операторного вузла виходить одна дуга, а з предикатного — дві дуги.

Предикативні вузли відповідають простим умовам у програмі. Складена умова програми відображається в декількох предикатних вузлах. Складовим називають умова, в якому використовується одна чи кілька булевих операцій.

Замкнуті області, утворені дугами і вузлами, називають регіонами.

Навколишнє середовище граф розглядається як додатковий регіон.

*Цикломатична складність.*

Цикломатична складність — метрика програмного забезпечення, яка забезпечує кількісну оцінку логічної складності програми.

*У способі тестування базового шляху цикломатична складність визначає:*

* Кількість незалежних шляхів у базовій програми.
* Верхню оцінку кількості тестів, яке гарантує одноразове виконання всіх операторів.

Незалежним називається будь-який шлях, який вводить новий оператор обробки або нову умову. У термінах потокового графа незалежний шлях повинен містити дугу, що не входить в раніше певні шляхи.

*Кроки способу тестування базового шляху.*

На основі тексту програми формується потоковий граф:

1. Нумеруються оператори тексту.

2. Проводиться відображення пронумерованого тексту програми у вузли і вершини потокового графа.

Визначається цикломатична складність потокового графа — по кожній з трьох формул.

Готуються тестові варіанти, які ініціюють виконання кожного шляху.

Кожен тестовий варіант формується в наступному вигляді:

*Вихідні дані (ВД):*

*Очікувані результати (ОЧ. РЕЗ.):*

Вихідні дані повинні вибиратися так, щоб предикатні вершини забезпечували потрібні перемикання — запуск тільки тих операторів, які перераховані в конкретному шляху, причому в необхідному порядку.

Реальні результати кожного тестового варіанта порівнюються з очікуваними результатами. Після виконання всіх тестових варіантів гарантується, що всі оператори програми виконані щонайменше один раз.

Важливо відзначити, що деякі незалежні шляху не можуть перевірятися ізольовано. Такі шляхи повинні перевірятися під час тестування іншого шляху (як частина іншого тестового варіанта).

***Способи тестування умов***

Мета цього сімейства способів тестування — будувати тестові варіанти для перевірки логічних умов програми. При цьому бажано забезпечити охоплення операторів з усіх гілок програми.

Спосіб тестування умов орієнтований на тестування кожної умови в програмі. Методика тестування умов мають дві гідності. По-перше, досить просто виконати вимірювання тестового покриття умови. По-друге, тестове покриття умов у програмі — це фундамент для генерації додаткових тестів програми.

Метою тестування умов є визначення не тільки помилок в умовах, але й інших помилок у програмах. Якщо набір тестів для програми A ефективний для виявлення помилок в умовах, що містяться в A, то ймовірно, що це набір також ефективний для виявлення інших помилок в A. Крім того, якщо методика тестування ефективна для виявлення помилок в умовах, то ймовірно, що ця методика буде ефективна для виявлення помилок в програмі.

***Тестування циклів.***

Цикл — найбільш поширена конструкція алгоритмів, використовуваних в ПЗ. Тестування циклів проводиться за принципом «білого ящика», під час перевірки циклів основна [увага](http://ua-referat.com/%D0%A3%D0%B2%D0%B0%D0%B3%D0%B0) звертається на правильність конструкцій циклів.

Розрізняють 4 типи циклів: прості, вкладені, об'єднані, неструктуровані.

*Прості цикл.*

Для перевірки циклів з ​​кількістю повторень n може використовуватися один з наступних наборів тестів:

·       прогін всього циклу;

·       тільки один прохід циклу;

·       два проходи циклу;

·       m проходів циклів, де m <n;

·       n —1, n, n +1 проходів циклу.

*Вкладені цикли*

Зі збільшенням рівня вкладеності циклів кількість можливих шляхів різко зростає. Це призводить до нездійсненної кількості тестів. Для скорочення кількості тестів застосовується спеціальна методика, в якій використовуються такі поняття, як осяжний і вкладені цикли.

*Кроки тестування:*

* Вибирається самий внутрішній цикл. Встановлюються мінімальні значення параметрів всіх інших циклів.
* Для внутрішнього циклу проводяться тести простого циклу. Додаються тести для виключених значень і значень, що виходять за межі робочого діапазону.
* Переходять у наступний за порядком осяжний цикл. Виконують його тестування. При цьому зберігаються мінімальні значення параметрів для всіх зовнішніх (осяжний) циклів і типові значення для всіх вкладених циклів.
* Робота триває до тих пір, поки не будуть протестовані всі цикли.

*Об'єднані цикли*

Якщо кожен із циклів незалежний від інших, то використовується [техніка](http://ua-referat.com/%D1%82%D0%B5%D1%85%D0%BD%D1%96%D0%BA%D0%B0) тестування простих циклів. За наявності залежності (наприклад, кінцеве значення лічильника першого циклу використовується як початкове значення лічильника другого циклу) використовується методика вкладених циклів.

*Неструктуровані цикли*

Неструктуровані цикли тестуванню не підлягають. Цей тип циклів має бути перероблений за допомогою структурованих програмних конструкцій.