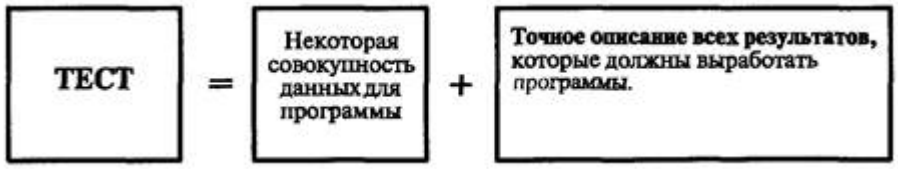
Тема 11 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

11.1. ОСНОВНІ ВІДОМОСТІ

Будь-який замовник хоче отримати надійний програмний виріб, який повністю задовольняє його потреби. Різні рівні надійності забезпечуються різними інженерними підходами до тестування. Іншими словами, за достатні кошти можна досягти рівня надійності як такої фірми або навіть рівня, необхідного для атомної енергетики або космічних досліджень.
Отже, рівень надійності програмних виробів визначається інженерією тестування. Як досягти найвищого рівня надійності? Зрозуміло, що тексти програм, написані однієї організації, можна заново інспектувати, тестувати автономне у складі ядра в інший організації. У самій програмі можна одночасно проводити розрахунки за різними алгоритмами з використанням різних обчислювальних методів і злічувати результати в контрольних точках, використовувати підстановки розрахованих результатів у вихідні рівняння і цим контролювати результати рішення. З викладеного видно, що рівень надійності програмних виробів безпосередньо з витратами як коштів, і часу проекту.
Як відомо, при створенні типового програмного проекту близько 50% загального часу і понад 50—60% загальної вартості витрачається на перевірку (тестування) програми або системи, що розробляється. Крім того, частка вартості тестування у загальній вартості програм має тенденцію зростати зі збільшенням складності програмних виробів та підвищення вимог до їх якості.
Враховуючи це, при виборі способу тестування програм слід чітко виділяти певну (по можливості не дуже велику) кількість правил налагодження, що забезпечують високу якість програмного продукту та знижують витрати на його створення.
Тестування здійснюється шляхом виконання тестів. Сенс тесту програм показано на рис. 11.1.
Аксіоми тестування, висунуті провідними програмістами:
— добрий той тест, для якого висока ймовірність виявлення помилки;
- Головна проблема тестування - вирішити, коли закінчити (зазвичай вирішується просто - закінчуються гроші);
- Неможливо тестувати свою власну програму;
- Необхідна частина тестів - опис вихідних результатів;
- уникайте невідтворюваних тестів;
- готуйте тести як для правильних, так і неправильних даних;
- не тестуйте «з літа»;
- Детально вивчайте результати кожного тесту;
— у міру виявлення дедалі більшої кількості помилок у певному модулі чи програмі, зростає ймовірність виявлення у ній ще більшої кількості помилок;
— тестують програми найкращі уми;
- вважають тестованість головним завданням розробників програми;
- не змінюй програму, щоб полегшити тестування;
- Тестування має починатися з постановки цілей.

Якщо в програмі ставлять коментарі на місці виклику модуля замість використання заглушки, виключають можливість перевірки типів даних, а також часто забувають знімати коментарі. Для пошуку «забутих» коментарів необхідне трудомістке налагодження.
Серед прийомів тестування варто виділити також так званий друк налагодження. Якщо налагоджувальні печатки вилучаються з тексту, супровід ускладнюється. Висновок: ніколи не вилучай налагоджувальні печатки навіть з використанням препроцесора:

```
{$IFDEF DEBUG THEN}
...
{$ELSE}
...
{$ENDIF}
```



*Мал. 11.1.*Сенс тесту програм

Краще опишіть глобальну змінну DebugLevel та програмуйте умовні налагоджувальні друку:

```
var
DebugLevel: слово; {0-немає жодного налагоджувального друку, чим більше значення, тим
докладніше налагоджувальний друк}
DebugFile: text; {файл налагоджувального друку}
DebugLevel: = 4; {завдання рівня налагодження}
if DebugLevel >= 3 then
WriteLn(DebugFile, 'Модуль:', 'MyModule, 'результат:');
```

11.2. ВЛАСТИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Слід виділити такі властивості програмного забезпечення.
Коректність програмного забезпечення — властивість безпомилкової реалізації необхідного алгоритму за відсутності таких факторів, що заважають, як помилки вхідних даних, помилки операторів ЕОМ (людей), збої та відмови ЕОМ.
В інтуїтивному сенсі під коректністю розуміють властивості програми, що свідчать про відсутність у ній помилок, допущених розробником на різних етапах проектування (специфікації, проектування алгоритму та структур даних, кодування). Коректність самої програми розуміють по відношенню до цілей, поставлених перед її розробкою (тобто це відносна властивість).
Стійкість - Властивість здійснювати необхідне перетворення інформації при збереженні вихідних рішень програми в межах допусків, встановлених специфікацією. Стійкість характеризує поведінка програми під впливом неї таких чинників нестійкості, як помилки операторів ЕОМ, і навіть не виявлені помилки програми.
Відновлюваність - властивість програмного забезпечення, що характеризує можливість пристосовуватися до виявлення помилок та їх усунення.
Надійність можна уявити сукупністю наступних характеристик:
- *Цілісністю програмного засобу* (здатністю його до захисту від відмов);
- *живучістю* (здатністю до вхідного контролю даних та їх перевірки під час роботи);
- **Завершеністю** (бездефектністю готового програмного засобу, характеристикою якості його тестування);
- *Працездатністю* (здатністю програмного засобу до відновлення своїх можливостей після збоїв).
Відмінність поняття коректності та надійності програм полягає в наступному:
- надійність характеризує як програму, так і її «оточення» (якість апаратури, кваліфікацію користувача тощо);
— говорячи про надійність програми, зазвичай допускають певну, хоч і малу частку *помилок* у ній і оцінюють *ймовірність* їх появи.

Повернемося до поняття коректності. Вочевидь, що ні всяка синтаксично правильна програма є коректною у зазначеному вище сенсі, т. е. коректність характеризує семантичні властивості програм.
З урахуванням специфіки появи помилок у програмах можна виділити дві сторони поняття коректності:
- коректність як точна відповідність цілям розробки програми (які відображені у специфікації) за умови її завершення або часткова коректність;
— завершення програми, т. е. досягнення програмою у її виконання своєї кінцевої точки.
Залежно від виконання чи невиконання кожного із двох названих властивостей програми розрізняють шість завдань аналізу коректності:
1) доказ часткової коректності;
2) доказ часткової некоректності;
3) доказ завершення програми;
4) доказ не завершення програми;
5) доказ тотальної (повної) коректності (тобто одночасне рішення 1-ї та 3-ї задач);
6) доказ некоректності (вирішення 2-го чи 4-го завдання). Методи доказу часткової коректності програм, як
правило, спираються на аксіоматичний підхід до формалізації семантики мов програмування.
Аксіоматична семантика мови програмування є сукупністю аксіом і правил виведення. За допомогою аксіом задається семантика простих операторів мови (присвоєння, введення-виведення, виклику процедур). З допомогою правил виведення описується семантика складових операторів чи керуючих структур (послідовності, умовного вибору, циклів). Серед цих правил виведення слід зазначити правило виведення для операторів циклу, оскільки воно вимагає знання інваріанту циклу (формули, істинність якої не змінюється за будь-якого проходження циклу).
Найбільш відомим із методів доказу часткової коректності програм є метод індуктивних тверджень, запропонований Флойдом та вдосконалений Хоаром. Один із важливих етапів цього методу – отримання анотованої програми. На цьому етапі для синтаксично правильної програми повинні бути задані твердження мовою логіки предикатів першого порядку: вхідний предикат; вихідний предикат. Ці твердження задаються для вхідної точки циклу та мають характеризувати семантику обчислень у циклі.
Доказ несправжності умов коректності свідчить про неправильність програми або її специфікації, або програми та специфікації.
За впливом на результати обробки інформації до надійності та стійкості програмного забезпечення близька і точність програмного забезпечення, що визначається помилками методу та помилками представлення даних.
Найбільш простими методами оцінки надійності програмного забезпечення є емпіричні моделі, що ґрунтуються на досвіді розробки програм: якщо до початку тестування на 1000 операторів припадає 10 помилок, а прийнятною якістю є 1 помилка, то в ході тестування треба знайти:

$$N \text{ ошибок} = \frac{N \text{ операторов}}{1000}.$$

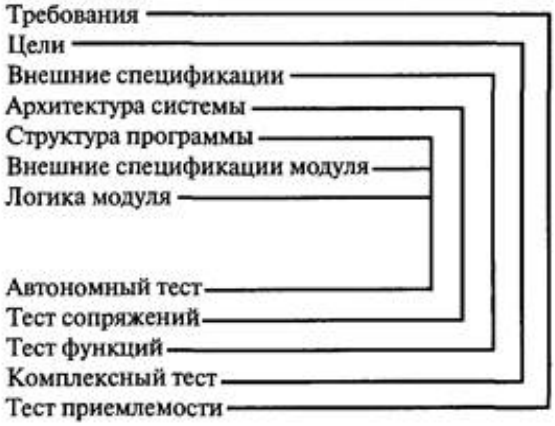
Точніша модель Холстеда: N помилок = N операторів * log2 (N операторів - N операндів),
де N операторів - число операторів у програмі; N операндів - число операндів у програмі.
Емпірична модель фірми IBM:
N помилок = 23 M(10) + 2 M(1),
де M(10) — число модулів із 10 і більше виправленнями; M(1) — число модулів із менше 10 виправленнями.
Якщо в модулі виявлено більше 10 помилок, його програмують заново.
За методом Мілса в програму, що розробляється, вносять заздалегідь відому число помилок. Далі вважають, що темпи виявлення помилок (відомих та невідомих) однакові.

11.3. ЗВ'ЯЗОК ПРОЦЕСІВ ТЕСТУВАННЯ З ПРОЦЕСОМ ПРОЕКТУВАННЯ

З рис. 11.2 видно, що помилки на ранніх етапах проекту вичерпно можуть бути виявлені наприкінці роботи.
Тестування програм охоплює низку видів діяльності:
• постановку задачі;
• проектування тестів;
• написання тестів;
• тестування тестів;
• виконання тестів;
• Вивчення результатів тестування.
Тут найважливішим є проектування тестів.
Отже, тестування це процес виконання програми з метою виявлення помилок.

11.4. ПІДХОДИ ДО ПРОЕКТУВАННЯ ТЕСТІВ

Розглянемо два найбільш протилежні підходи до проектування тестів.
Прихильник першого підходу орієнтується лише на стратегію тестування, яка називається стратегією «чорної скриньки», тестуванням з управлінням за даними або тестуванням з управлінням по входу-виходу. При використанні цієї стратегії програма розглядається як чорна скринька. Тестові дані використовуються лише відповідно до специфікації програми (тобто без урахування знань про її внутрішню структуру). Недосяжний ідеал прихильника першого підходу – перевірити всі можливі комбінації та значення на вході. Зазвичай їх дуже багато навіть для найпростіших алгоритмів. Так, для програми розрахунку середнього арифметичного чотирьох чисел треба готувати 107 тестових даних.



*Мал. 11.2.*Взаємозв'язок процесів проектування та тестування

При першому підході виявлення всіх помилок у програмі є критерієм вичерпного вхідного тестування. Останнє може бути досягнуто, якщо як тестові набори використовувати всі можливі набори вхідних даних. Отже, приходимо до висновку, що для вичерпного тестування програми потрібна нескінченна кількість тестів, а отже, побудова вичерпного вхідного тесту неможлива. Це підтверджується двома аргументами: по-перше, не можна створити тест, який би гарантував відсутність помилок; по-друге, розробка таких тестів суперечить економічним вимогам. Оскільки вичерпне тестування виключається, нашою метою має стати максимізація результативності капіталовкладень у тестування (максимізація числа помилок, які виявляються одним тестом). Для цього необхідно розглядати внутрішню структуру програми і робити деякі розумні, але, звісно, припущення, що не мають повної гарантії достовірності.
Прихильник другого підходу використовує стратегію «білої скриньки», або стратегію тестування, керовану логікою програми, що дозволяє досліджувати внутрішню структуру програми. І тут тестувальник отримує тестові дані шляхом аналізу лише логіки програми; прагне, щоб кожну команду було виконано хоча б один раз. При достатній кваліфікації домагається, щоб кожна команда умовного переходу виконувалася в кожному напрямі хоча б один раз. Цикл повинен виконуватися один раз, жодного разу, максимальну кількість разів. Мета тестування всіх шляхів ззовні також недосяжна. У програмі двох послідовних циклів усередині кожного з них включено розгалуження на десять шляхів, є

1018 шляхів розрахунку. При цьому виконання всіх шляхів розрахунку не гарантує виконання всіх специфікацій. Для довідки: вік Всесвіту 1017с.
Порівняємо спосіб побудови тестів за цієї стратегії з вичерпним вхідним тестуванням стратегії «чорного ящика». Невірно припущення, що достатньо побудувати такий набір тестів, у якому кожен оператор виконується хоча б один раз. Вичерпне вхідне тестування може бути поставлене у відповідність вичерпне тестування маршрутів. Мається на увазі, що програма перевірена повністю, якщо за допомогою тестів вдається здійснити виконання цієї програми по всіх можливих маршрутах потоку (графа) передач управління.
Останнє твердження має два слабкі пункти: по-перше, число маршрутів, що не повторюють один одного, — астрономічне; по-друге, навіть якщо кожен маршрут може бути перевірений, сама програма може містити помилки (наприклад, деякі маршрути пропущено).
Властивість шляху виконуватися правильно для одних даних і неправильно для інших - зване чутливістю до даних, що найчастіше проявляється за рахунок чисельних похибок і похибок усічення методів. Тестування кожного зі всіх маршрутів одним тестом не гарантує виявлення чутливості до даних.
В результаті всіх викладених вище зауважень зазначимо, що ні вичерпне вхідне тестування, ні вичерпне тестування маршрутів не можуть стати корисними стратегіями, тому що вони не реалізуються. Тому реальним шляхом, який дозволить створити хорошу, але, звісно, не абсолютну стратегію, є поєднання тестування програми кількома методами.
Розглянемо приклад тестування оператора
```
if A and B then ...
```
під час використання різних критеріїв повноти тестування.
При критерії покриття умов були б потрібні два тести: A = true, B = false і A = false, B = true. Але в цьому випадку не виконується then-пропозиція оператора if.
Існує ще один критерій, названий покриттям рішень/умов. Він вимагає такого достатнього набору тестів, щоб усі можливі результати кожної умови у вирішенні виконувались принаймні один раз; всі результати кожного рішення виконували також один раз і кожній точці входу передавалося управління, принаймні один раз.
Недоліком критерію покриття рішень/умов є неможливість застосування для виконання всіх результатів всіх умов. Часто таке виконання має місце через те, що певні умови приховані іншими умовами. Наприклад, якщо умова AND є брехня, то жодна з наступних умов у виразі не буде виконана. Аналогічно, якщо умова OR є істиною, то жодна з наступних умов не буде виконана. Отже, критерії покриття умов та покриття рішень/умов недостатньо чутливі до помилок у логічних виразах.
Критерієм, який вирішує ці та деякі інші проблеми, є комбінаторне покриття умов. Він вимагає створення такого числа тестів, щоб усі можливі комбінації результатів умови в кожному рішенні та всі точки входу виконувались принаймні один раз.
У разі циклів кількість тестів задоволення критерію комбінаторного покриття умов зазвичай більше, ніж кількість шляхів.
Легко бачити, що набір тестів, що задовольняє критерію комбінаторного покриття умов, задовольняє також критеріям покриття рішень, покриття умов і покриття рішень/умов.
Таким чином, для програм, що містять лише одну умову на кожне рішення, мінімальним є критерій, набір тестів якого викликає виконання всіх результатів кожного рішення, принаймні один раз; передає керування кожній точці входу (наприклад, оператор CASE).
Для програм, що містять рішення, кожне з яких має більше однієї умови, мінімальний критерій складається з набору тестів, що викликають усі можливі комбінації результатів умов у кожному рішенні та передають управління кожній точці входу програми, принаймні один раз.
Розподіл алгоритму на типові стандартні структури, згідно з проектною процедурою кодування тексту модуля (методу) з п'ятого розділу підручника, дозволяє мінімізувати зусилля програміста, що витрачаються на тестування. Заборона на вкладені структури таки пояснюється зайвими витратами на тестування. Використання ланцюжка простих альтернатив з одним дією чи структури ВИБІР замість вкладених простих АЛЬТЕРНАТИВ значно скорочує кількість тестів!

11.5. ПРОЕКТУВАННЯ ТЕСТІВ ВЕЛИКИХ ПРОГРАМ

Проектування тестів великих програм поки що більшою мірою залишається мистецтвом і меншою мірою є наукою. Щоб побудувати розумну стратегію тестування, треба розумно поєднувати обидва ці крайні підходи і користуватися математичними доказами.
Висхідне тестування. Спочатку автономно тестуються модулі нижніх рівнів, які викликають інших модулів. При цьому досягається така ж їхня висока надійність, як і у вбудованих у компілятор функцій. Потім тестуються модулі вищих рівнів разом із вже перевіреними модулями тощо за схемою ієрархії. При висхідному тестуванні для кожного модуля потрібна провідна програма. Це монітор або драйвер, який подає тести відповідно до специфікацій тестів. Ряд фірм випускає промислові драйвери чи монітори тестів.
Низхідне тестування. У цьому підході ізольовано тестується головний модуль чи група модулів головного ядра. Програма збирається та тестується зверху вниз. Відсутні модулі замінюються заглушками.
Переваги низхідного тестування: цей метод поєднує тестування модуля з тестуванням пар і частково тестує функції модуля. Коли вже починає працювати введення/виведення модуля, зручно готувати тести.
Недоліки низхідного тестування: модуль рідко досконально тестується одразу після його підключення. Для ґрунтовного тестування потрібні витончені заглушки. Часто програмісти відкладають ретельне тестування і забувають про нього. Інший недолік - бажання розпочати програмування ще до кінця проектування. Якщо ядро вже запрограмоване, виникає опір будь-яким його змін, навіть поліпшення структури програми. Зрештою, саме раціоналізація структури програми за рахунок проведення проектних ітерацій сприяє досягненню більшої економії, ніж дасть раннє програмування.
Модифікований низхідний метод. Згідно з цим методом, кожен модуль автономно тестується перед включенням до програми, що збирається зверху донизу.
Метод великого стрибка - Кожен модуль тестується автономно. Після закінчення автономного тестування всіх модулів модулі легко інтегруються в готову програмну систему. Як правило, цей метод небажаний. Однак якщо програма мала і добре спроектована по сполучення, то метод великого стрибка цілком прийнятний.
*Метод сандвіча*є компромісом між низхідним і висхідним підходами. За цим методом реалізація та тестування ведуться одночасно зверху та знизу, і два ці процеси зустрічаються в заздалегідь наміченій тимчасовій точці.
Модифікований метод сандвіча: нижні модулі тестуються строго знизу вгору, а модулі верхніх модулів спочатку тестуються автономно, а потім збираються низхідним методом.

11.6. КРИТЕРІЇ ВИБОРУ НАЙКРАЩОЇ СТРАТЕГІЇ РЕАЛІЗАЦІЇ

Критеріями вибору найкращої стратегії реалізації програми є:
• час до повного складання програми;
• час реалізації скелета програми;
• існуючий інструментарій тестування;
• міра паралелізму ранніх етапів реалізації;
• можливість перевірки будь-яких шляхів програми даними із заглушок;
• складність планування та дотримання графіка реалізації;
• складність тестування.

11.7. СПОСОБИ ТА ВИДИ ТЕСТУВАННЯ ПІДПРОГРАМ. ПРОЕКТУВАННЯ ТЕСТІВ

Існують два способи тестування: публічний та приватний.
Громадське тестування означає, що кожен бажаючий може отримати цей товар (або безкоштовно, або за ціною дисків і доставки).
Приватний спосіб тестування означає, що перевірку продукту проводить обмежене коло тестувальників, які висловили згоду активно працювати з продуктом та оперативно повідомляти про всі виявлені дефекти. Часто використовуються обидва способи: спочатку програма проходить приватне

тестування, а потім коли всі великі проколи вже виявлені, починається її публічне тестування, щоб зібрати відгуки широкого кола користувачів.
Більшість компаній-розробників вимагають, щоб приватний бета-тестувальник підписав «Угоду про нерозголошення» (NDA, Non-Disclosure Agreement). Тим самим він зобов'язується не розголошувати подробиці про продукт, що тестується, і не передавати його копії третім особам. Подібні угоди підписуються з метою збереження комерційної таємниці та щоб уникнути недобросовісної конкуренції. Так, фірма "Microsoft" використовує всі перелічені вище підходи для тестування своїх продуктів. З її сайтів завжди можна безкоштовно завантажити велику кількість продуктів, що проходять публічне тестування бета. Проте їхній появі передує багатомісячний процес приватного тестування. Фірма Microsoft проводить політику відкритого набору приватних бета-тестувальників, при якій кожен бажаючий може повідомити корпорації про своє бажання взяти участь у тестуванні того чи іншого продукту. Насправді коло приватних бета-тестерів Microsoft дуже вузький, і шанс, що вас включать до їхнього числа, невеликий. Тим не менш, він існує, а потрапляють до списку бета-тестерів практично довічно.
Приватне тестування підпрограм починається з етапу контролю, основними різновидами якого є: візуальний, статичний та динамічний.
Візуальний контроль — це перевірка текстів за столом, без використання комп'ютера.
На першому етапі візуального контролю здійснюється читання тексту підпрограми, причому особлива увага приділяється: коментарям та їх відповідності до тексту програми; умовам в операторах умовного вибору та циклу; складним логічним виразам; можливості незавершення ітераційних циклів
Другий етап **візуального контролю** - наскрізний контроль тексту підпрограми (його ручне прокручування на кількох заздалегідь підібраних простих тестах). Поширена думка, що вигіднішим є перекладання більшої частини роботи з контролю програмних засобів на комп'ютер помилково. Основний аргумент на користь цього такий: при роботі на комп'ютері головним чином удосконалюються навички у використанні клавіатури, у той час як програмістська кваліфікація набувається, перш за все, за столом.
Статичний контроль - Це перевірка тексту підпрограми (без виконання) за допомогою інструментальних засобів.
Першою, найбільш відомою формою статичного контролю є синтаксичний контроль програми з допомогою компілятора, у якому перевіряється відповідність тексту програми синтаксичним правилам мови програмування.
Повідомлення компілятора зазвичай поділяються на кілька груп залежно від рівня тяжкості порушення синтаксису мови програмування:
1) інформаційні повідомлення та попередження, при виявленні яких компілятор, як правило, будує коректний об'єктний код і подальша робота з програмою (компонування, виконання) можлива (проте повідомлення цієї групи повинні ретельно аналізуватися, тому що їх поява також може свідчити про помилку у програмі - наприклад, через неправильне розуміння синтаксису мови);
2) повідомлення про помилки, при виявленні яких компілятор намагається їх виправити та будує об'єктний код, але його коректність малоймовірна та подальша робота з ним, швидше за все, неможлива;
3) повідомлення про серйозні помилки, за наявності яких побудований компілятором об'єктний код свідомо некоректний та його подальше використання неможливе;
4) повідомлення про помилки, виявлення яких призвело до припинення синтаксичного контролю та побудови об'єктного коду.
Проте будь-який компілятор пропускає деякі види синтаксичних помилок. Місце виявлення помилки може знаходитись далеко за текстом підпрограми від місця справжньої помилки, а текст повідомлення компілятора може не вказувати на справжню причину помилки. Одна синтаксична помилка може спричинити генерацію компілятором кількох повідомлень про помилки (наприклад, помилка в описі змінної призводить до появи повідомлення про помилку в кожному операторі підпрограми, який використовує цю змінну).
Другою формою **статичного контролю** може бути контроль структурованості тексту підпрограми, тобто перевірка виконання угод та обмежень структурного програмування. Прикладом подібної перевірки може бути виявлення у тексті підпрограми ситуацій, коли цикл утворюється за допомогою оператора безумовного переходу (використання оператора GOTO для переходу до тексту програми).

Для проведення контролю структурованості можуть бути створені спеціальні інструментальні засоби, а за їх відсутності ця форма статичного контролю може поєднуватися з візуальним контролем.
***Третя форма* статичного контролю** - контроль правдоподібності підпрограми, тобто виявлення в її тексті конструкцій, які хоч і синтаксично коректні, але швидше за все містять помилку або свідчать про неї. Основні неправдоподібні ситуації:
• використання у програмі не ініціалізованих змінних (тобто змінних, які отримали початкового значення);
• наявність у програмі описів елементів, змінних, процедур, міток, файлів, які надалі не використовуються в її тексті;
• наявність у тексті підпрограми фрагментів, які ніколи не виконуються;
• наявність у тексті програми змінних, які жодного разу не використовуються для читання після присвоєння їм значень;
• наявність у тексті підпрограми свідомо нескінченних циклів.
Навіть якщо присутність у тексті програми неправдоподібних конструкцій не призводить до її неправильної роботи, виправлення цього фрагмента підвищить ясність та ефективність програми, тобто благотворно позначиться на її якості.
Для можливості контролю правдоподібності в повному обсязі також повинні бути створені спеціальні інструментальні засоби, хоча ряд можливостей контролю правдоподібності є в існуючих налагоджувальних і звичайних компіляторах.
Слід зазначити, що створення інструментальних засобів контролю структурованості та правдоподібності програм може бути спрощено суттєво при застосуванні наступних принципів:
1) проведення додаткових форм статичного контролю після завершення компіляції та тільки для синтаксично коректних програм;
2) максимальне використання результатів компіляції та лінкування програми та, зокрема, інформації, що включається до лістингу компілятора та лінкеру;
3) замість повного синтаксичного розбору тексту програми, що перевіряється, необхідна побудова для неї списку ідентифікаторів та списку операторів із зазначенням усіх їх необхідних ознак.
За відсутності інструментальних засобів контролю правдоподібності ця фаза статичного контролю може об'єднуватися з візуальним контролем.
***Четвертою формою* статичного контролю** програм є їхня верифікація, тобто аналітичний доказ їх коректності.
Незважаючи на достатню складність процесу верифікації програми і на те, що навіть успішно завершена верифікація не дає гарантій якості програми (оскільки помилка може міститися і у верифікації), застосування методів аналітичного доказу правильності дуже корисне для уточнення сенсу програми, що розробляється, а знання цих методів благотворно позначається на кваліфікації програміста.
Динамічний контроль програми- Це перевірка правильності програми при її виконанні на комп'ютері, тобто тестування. Мінімальне автономне тестування підпрограми має забезпечувати проходження всіх шляхів обчислень.
Проектна процедура тестування підпрограми полягає в наступному:
- за зовнішніми специфікаціями модуля підготуйте тести для кожної ситуації та кожної неприпустимої умови;
— перегляньте текст підпрограми, щоб переконатися, що всі умовні переходи виконуватимуться у кожному напрямку; за потреби додайте тести;
- Перевірте текст підпрограми, що тести охоплюють досить багато шляхів; для циклів повинні бути тести без повторення, з одним повторенням та з кількома повтореннями;
— перевірте за текстом підпрограми її чутливість до особливих значень даних (найнебезпечніші числа — це нуль і одиниця), у разі потреби додайте тести.

11.8. ПРОЕКТУВАННЯ КОМПЛЕКСНОГО ТЕСТА

У комплексному тесті мають проводитися такі види тестування:
• працездатності;
• стресів;

• граничного обсягу даних, що вводяться;
• конфігурації різних технічних засобів;
• сумісності;
• захисту;
• необхідної пам'яті;
• продуктивність;
• налаштування;
• надійність;
• засобів відновлення при відмові;
• зручності обслуговування;
• програмної документації;
• психологічних факторів;
• зручність експлуатації.

11.9. ЗАСОБИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

Генератори тестів (automatic unit test) випадково генерують дані.
Статичні аналізатори програм, аналізують вихідний тест та будують діаграми маршрутів; аналізують привласнення даних та роблять спроби побудов даних, що призводять до помилки.
Засоби періоду виконання зазвичай роблять статистичний підрахунок кількості виконання кожного оператора та дозволяють контролювати повноту тестів.

ВИСНОВКИ

• Тестування програм – головне, що визначає найважливішу якість програм – надійність.
• На тестування витрачається основна частина коштів та часу проекту.
• Під час розробки будь-якої програми або системи тестування відбирає більшу частину часу та грошей. З огляду на це необхідно визначити не дуже велику кількість тестів, що забезпечують високу ймовірність виявлення тих чи інших помилок.
• Аксіоми тестування визначають основні цілі та принципи тестування.
• Якщо з тексту виключити налагоджувальні печатки, суттєво ускладниться супровід.
• Існують два крайні підходи до проектування тестів: стратегія «чорної скриньки» та стратегія «білої скриньки». Марно слідувати лише одному підходу. Необхідно будувати стратегію тестування лише з урахуванням поєднання підходів.
• При проектуванні багатомодульних програм використовується висхідне тестування (автономне тестування нижніх модулів, які не викликають інших модулів) та низхідне тестування (застосування заглушок нижніх рівнів). Але у кожного з них є свої переваги та недоліки. Можливі також варіанти:
- модифікований низхідний метод - згідно з цим методом кожен модуль автономно тестується перед включенням до програми, що збирається зверху донизу;
- метод великого стрибка - кожен модуль тестується автономно, далі модулі легко інтегруються в готову програмну систему;
- метод сандвіча - за цим методом реалізація та тестування ведуться одночасно зверху та знизу, і два ці процеси зустрічаються в заздалегідь наміченій тимчасовій точці;
- Модифікований метод сандвіча - нижні модулі тестуються строго знизу вгору, а модулі верхніх модулів спочатку тестуються автономно, а потім збираються низхідним методом. Цей метод апробовано під час створення ОС.
• Існує розподіл тестування на публічне та приватне. Часто використовуються обидва способи: спочатку програма проходить приватне тестування, а потім, коли всі великі проколи вже виявлено, починається її публічне тестування, щоб зібрати відгуки широкого кола користувачів.

Контрольні питання

1. Назвіть основні аксіоми тестування.
2. У чому перевага налагоджувального друку?

3. Які властивості програмного забезпечення мають найбільший вплив на процес виявлення помилок при тестуванні?
4. Навіщо потрібні емпіричні моделі? Як проводиться їхній аналіз?
5. Який зв'язок між процесами тестування та проектування?
6. У чому переваги та недоліки висхідного та низхідного проектування? Відповідь обґрунтувати на прикладі конкретної програми.
7. Який тест максимально швидко виявить зациклювання?
8. Виділіть кілька основних критеріїв під час вибору параметрів тестування.