**Лекція 3. Механізми та методи тестування. Тестування програмного коду.**

**Програмні помилки**

     Програм без помилок не існує. Практика доводить, що винуватцями помилок у програмах найчастіше бувають самі програмісти. Один із загальних законів практичного програмування полягає в тому, що жодна програма не дає бажаних результатів при першій спробі трансляції та виконання. Певне уявлення про справжні причини появи помилок у роботі програми дає таке процентне співвідношення джерел збоїв:

|  |  |
| --- | --- |
| Вхідні дані | 1% |
| Помилки користувача | 5% |
| Апаратура | 1% |
| Системне програмне забезпечення | 3% |
| Розробка системи | 15% |
| Програмування | 75% |

     Існують два типи програмних помилок:

*синтаксичні помилки* - виникають через порушення правил мови програмування. Такі помилки зазвичай виявляються під час компіляції. Можуть бути виключені порівняно легко. Навіть якщо не переглядати текст програми можна бути впевненим, що компілятор на стадії трансляції знайде помилки і видасть відповідні попередження. Фактично пошук помилок здійснює компілятор, а їхнє виправлення - програміст;

*семантичні (логічні) помилки* - ті, що призводять до некоректних обчислень або помилок під час виконання (run-time error). Семантичні помилки усувають зазвичай за допомогою виконання програми з ретельно підібраними перевірочними даними, для яких відома правильна відповідь.

**Завдання і цілі тестування програмного коду**

Тестування програмного коду - процес виконання програмного коду, спрямований на виявлення існуючих в ньому дефектів. Під дефектом тут розуміється ділянку програмного коду, виконання якого за певних умов призводить до несподіваного поведінки системи (тобто поведінки, що не відповідає вимогам). Несподіване поведінку системи може призводити до збоїв у її роботі і відмов, в цьому випадку говорять про істотні дефектах програмного коду. Деякі дефекти викликають незначні проблеми, що не порушують процес функціонування системи, але кілька утрудняють роботу з нею. У цьому випадку говорять про середні або малозначних дефектах.

Завдання тестування при такому підході - *визначення* умов, при яких виявляються дефекти системи, і *протоколювання* цих умов. У завдання тестування зазвичай не входить виявлення конкретних дефектних ділянок програмного коду і ніколи не входить виправлення дефектів - це завдання налагодження, яка виконується за результатами тестування системи.

Мета застосування процедури тестування програмного коду - *мінімізація* кількості дефектів (особливо істотних) в кінцевому продукті. Тестування саме по собі не може гарантувати повної відсутності дефектів у програмному коді системи. Проте, у поєднанні з процесами верифікації та валідації, спрямованими на усунення суперечливості і неповноти проектної документації (зокрема - вимог на систему), грамотно організоване тестування дає гарантію того, що система задовольняє вимогам і веде себе відповідно з ними у всіх передбачених ситуаціях.

При розробці систем підвищеної надійності, наприклад, авіаційних, гарантії надійності досягаються за допомогою чіткої організації процесу тестування, визначення його зв'язку з іншими процесами життєвого *циклу,* введення кількісних характеристик, що дозволяють оцінювати успішність тестування. При цьому чим вище вимоги до надійності системи (її рівень критичності), тим більш жорсткі вимоги пред'являються.

Таким чином, в першу *чергу* потрібно розглядати не конкретні результати тестування конкретної системи, а загальну організацію процесу тестування, використовуючи підхід "добре організований процес дає якісний результат". Такий підхід є загальним для багатьох міжнародних та галузевих стандартів якості. Якість розроблюваної системи при такому підході є наслідком організованого процесу розробки і тестування, а не самостійним некерованим результатом.

Оскільки сучасні програмні системи мають значні розміри, при тестуванні їх програмного коду використовується метод функціональної декомпозиції. Система розбивається на окремі модулі (класи, простору імен і т.п.), що мають певні вимоги щодо функціональності і відповідні інтерфейси. Після цього окремо тестується кожен *модуль* - виконується *модульне тестування.* Потім відбувається *збірка* окремих модулів у більші конфігурації - виконується *інтеграційне тестування,* і нарешті, тестується система в цілому - виконується *системне тестування.* З точки зору програмного коду, модульне, інтеграційне і *системне тестування* мають багато спільного.

У ході модульного тестування кожен *модуль* тестується як на відповідність вимогам, так і на відсутність проблемних ділянок програмного коду, які можуть викликати відмови і збої в роботі системи. Як правило, модулі не працюють поза системою - вони приймають дані від інших модулів, переробляють їх і передають далі. Для того, щоб з одного боку, ізолювати *модуль* від системи і виключити вплив потенційних помилок системи, а з іншого боку - забезпечити *модуль* всіма необхідними даними, використовується ***тестове оточення***. Завдання тестового оточення - створити середовище виконання для модуля, емулювати всі зовнішні інтерфейси, до яких звертається *модуль.*

Типова процедура тестування полягає у підготовці та виконанні тестових прикладів (також званих просто тестами). Кожен тестовий приклад перевіряє одну "ситуацію" в поведінці модуля і складається зі списку значень, переданих на вхід модуля, описи запуску і виконання переробки даних - тестового сценарію і списку значень, які очікуються на виході модуля в разі його коректної поведінки. Тестові сценарії складаються таким чином, щоб виключити звернення до внутрішніх даних модуля, вся взаємодія має відбуватися тільки через його зовнішні інтерфейси.

Виконання тестового прикладу підтримується тестовим оточенням, яке включає в себе програмну реалізацію тестового сценарію. Виконання починається з передачі модулю вхідних даних і запуску сценарію. Реальні *вихідні дані,* отримані від модуля в результаті виконання сценарію, зберігаються і порівнюються з очікуваними. У разі їх збігу тест вважається пройденим, в іншому випадку - НЕ пройденим. Кожний не пройдений тест вказує на дефект або в тестуємому модулі, або в тестовому оточенні, або в описі тесту.

Сукупність описів тестових прикладів становить тест-план - основний документ, який визначає процедуру тестування програмного модуля. Тест-план задає не тільки самі тестові приклади, а й порядок їх слідування, який також може бути важливий.

При тестуванні часто буває необхідно враховувати не тільки вимоги до системи, але і структуру програмного коду модуля, що тестується. У цьому випадку тести складаються таким чином, щоб знайти типові помилки програмістів, викликані невірною інтерпретацією вимог. Застосовуються перевірки граничних умов, перевірки класів еквівалентності. Відсутність у системі можливостей, не заданих вимогами, гарантовано різними оцінками покриття програмного коду тестами, тобто оцінками того, який *відсоток* тих чи інших мовних конструкцій виконаний в результаті виконання всіх тестових прикладів.

**Класи еквівалентності.**

Згідно зі специфікацією усі означені поля мають виконувати певні функції. Класи еквівалентності використовують тоді, коли ми не хочемо виконувати тести, які ми вважаємо однаковими. Але коли отримуємо продукт для тестування, то змушені відтестувати кожне із цих полів хоча б один раз, оскільки невідомо на скільки “еквівалентними” є ці поля. Причина - специфікація може відрізнятися від того як програмісти реалізували специфікацію, і той, хто не був присутній під час їхньої роботи, не можуть про це навіть здогадуватись. Тому класи еквівалентності — це обов’язкові типи користувацького вводу, які потрібно використовувати для тестування кожного із цих полів.

Для початку, кожне з полів, не залежно від специфікації, тестують на можливість вводу в нього: нічого, рядка (1, 5, багато символів, дуже багато символів, спеціальні символи, різні види дужок, символи кінця рядка для різних систем, різні числові типи). Дані цих тестів порівнюють з вимогами у специфікації, і, можливо, для якихось полів роблять висновки про невідповідність. Після цього продовжують тестувати далі з іншими даними та функціональними сценаріями, враховуючи отримані результати, щоб знайти якомога більше граничних умов, і вже за ними зробити якісь висновки щодо “еквівалентності” чи “нееквівалентності” самих полів.

Підхід до розбиття на еквівалентні класи.

1. Розбивати на еквівалентні класи можна все, що завгодно: вимоги, тестові кроки, дані, набори даних, результати тестів, набори тестів, а також самих тестувальників і програмістів.
2. Розбиття на класи тісно пов’язане із граничними умовами (мабуть, якщо ми розбиваємо величину певного типу, таку як число, символьний рядок, розмір та ін.)
3. Накладання різних еквівалентних класів народжує нове розбиття на класи еквівалентності.
4. Класи еквівалентності (граничні умови) однієї величини треба використовувати для визначення класів еквівалентності (граничних умов) іншої пов’язаної величини. Наприклад, якщо введення у числове поле рядків ‘Hello’ та ‘Привіт!!!!1111’ призводить до двох різних повідомлень про помилку (навіть зовсім трішечки різних), то ці рядки (і тести що їх використовують) не можна вважати еквівалентними.

**Таким чином розділ на класи еквівалентності** – це техніка, при який діапазон можливих вхідних значень поділяється на групи значень еквівалентних по впливу на систему, тобто допустимі значення система приймає, а недопустимі – ні. Повинно заборонити введення до поля недопустимих значень. Приклад.

Виділяємо два класи еквівалентності:

1. Допустимі значення: числа от 1 до 1000.
2. Недопустимі значення: числа от -∞ до 0, от 1001 до +∞, а також всі решта літер та символів.

Клас з недопустимими значеннями можна розбити на декілька:

1. Від - ∞  до 0.
2. Від 1001 до + ∞.
3. Спеціальні символи (# @ + — / \_  : ; “ ‘ и т.д.).
4. Літери.

В результаті, класи еквівалентності дозволяють використовувати мінімум 5 тестів для тестування поля введення. Наприклад, в поле ввести такі дані: 46, -37, 1773, Ім’я, $\_=#.

Техніку розділення на класи еквівалентності застосовують для скорочення числа тестів, при цьому зберігаючи прийнятне тестове покриття. Дана техніка підходить також для текстових або інших типів даних.

**Методи тестування. Чорний ящик**

Основна ідея в тестуванні системи як чорного ящика полягає в тому, що всі матеріали доступні тестувальникові, - вимоги на систему, що описують її поведінку, і сама система, працювати з якою він може, тільки подаючи на її входи деякі зовнішні впливи і спостерігаючи на виходах деякий результат. Всі внутрішні особливості реалізації системи приховані від тестувальника, таким чином, система являє собою "чорний ящик", правильність поведінки якого по відношенню до вимог і належить перевірити.

З точки зору програмного коду чорний ящик може представляти з собою набір класів (або модулів) з відомими зовнішніми інтерфейсами, але недоступними вихідними текстами.

Основне завдання тестувальника для даного методу тестування полягає в послідовній перевірці відповідності поведінки системи вимогам. Крім того, тестувальник повинен перевірити роботу системи в критичних ситуаціях: що відбувається у разі подання невірних вхідних значень. В ідеальній ситуації всі варіанти критичних ситуацій повинні бути описані у вимогах на систему і тестувальник залишається тільки придумувати конкретні перевірки цих вимог. Проте в реальності в результаті тестування зазвичай виявляється два типи проблем системи.

1. Невідповідність поведінки системи вимогам.

2. Неадекватна поведінка системи в ситуаціях, не передбачених вимогами.

Звіти по обох типах проблем документуються і передаються розробникам. При цьому проблеми першого типу зазвичай викликають зміну програмного коду, набагато рідше - зміну вимог. Зміна вимог в даному випадку може знадобитися через їх суперечливості (кілька різних вимог описують різні моделі поведінки системи в одній і тій же самій ситуації) або некоректності (вимоги не відповідають дійсності).

Проблеми другого типу однозначно вимагають зміни вимог зважаючи на їх неповноти - у вимогах явно пропущена ситуація, яка веде до неадекватної поведінки системи. При цьому під неадекватною поведінкою може розумітися як повний крах системи, так і взагалі будь-яка поведінка, що не описане в вимогах. Тестування чорної скриньки називають також тестуванням за вимогами, тому що це єдине джерело інформації для побудови тест-плану.

**Скляний (білий) ящик**

При тестуванні системи як скляного ящика тестувальник має доступ не тільки до вимог до системи, її входів і виходів, а й до її внутрішній структурі - бачить її програмний код. Доступність програмного коду розширює можливості тестувальника тим, що він може бачити відповідність вимог ділянкам програмного коду і визначати тим самим, на весь чи програмний код існують вимоги. Програмний код, для якого відсутні вимоги, називають кодом, не вкритим вимогами. Такий код є потенційним джерелом неадекватної поведінки системи. Крім того, прозорість системи дозволяє поглибити аналіз її ділянок, що викликають проблеми - часто одна проблема нейтралізує іншу, і вони ніколи не виникають одночасно.

**Тестування моделей**

Тестування моделей знаходиться трохи осторонь від класичних методів верифікації програмного забезпечення, оскільки об'єктом тестування є не саме система, а її модель, спроектована формальними засобами. Вважається, що коректність, застосовність і відповідність вихідній системі самої моделі можуть бути доведені формальними засобами, тому тестувальник отримує в своє розпорядження досить потужний інструмент аналізу загальної цілісності системи. На моделі можна створити такі ситуації, які неможливо створити в тестовій лабораторії для реальної системи. Працюючи з моделлю програмного коду системи, можна аналізувати його властивості і такі параметри системи, як оптимальність алгоритмів або її стійкість. Проте тестування моделей не набуло широкого поширення саме через труднощі, що виникають при розробці формального опису поведінки системи. Одне з небагатьох виключень - системи зв'язку, алгоритмічний і математичний апарат яких досить добре опрацьований.

**Аналіз програмного коду (інспекції)**

У багатьох ситуаціях тестування поведінки системи в цілому неможливо - окремі ділянки програмного коду можуть ніколи не виконуватися, при цьому вони будуть покриті вимогами. Прикладом таких ділянок коду можуть служити обробники виняткових ситуацій. Якщо, наприклад, два модулі передають один одному числові значення і функції перевірки коректності значень працюють в обох модулях, то функція перевірки модуля-приймача ніколи не буде активізована, тому що всі помилкові значення будуть відсічені ще в передавачі.

У цьому випадку виконується ручний аналіз програмного коду на коректність, званий також перегляданнями або інспекціями коду. Якщо в результаті інспекції виявляються проблемні ділянки, то інформація про це передається розробникам для виправлення нарівні з результатами звичайних тестів.

**Тестове оточення**

Основний обсяг тестування практично будь-якої складної системи зазвичай виконується в автоматичному режимі. Крім того, тестуєма система зазвичай розбивається на окремі модулі, кожен з яких тестується спочатку окремо від інших, потім в комплексі.

Це означає, що для виконання тестування необхідно створити деяке середовище, яке забезпечить *запуск* і виконання модуля, що тестується, передасть йому *вхідні дані,* збере реальні *вихідні дані,* отримані в результаті роботи системи на заданих вхідних даних. Після цього середовище має порівняти реальні *вихідні дані* з очікуваними і на підставі даного порівняння зробити *висновок* про відповідність поведінки модуля заданому ( Рис. 1 ).

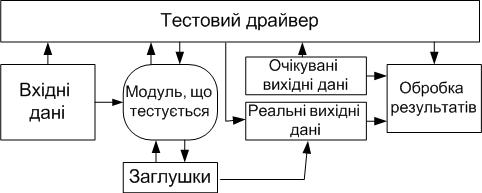


Рис. 1.Узагальнена схема середовища тестування

Тестове оточення також може використовуватися для відчуження окремих модулів системи від всієї системи. Поділ модулів системи на ранніх етапах тестування дозволяє більш точно локалізувати проблеми, що виникають в їх програмному коді. Для підтримки роботи модуля у відриві від системи тестове оточення має моделювати поведінку всіх модулів та функцій, до яких звертається тестований *модуль,* або дані, які від них має отримувати*.*

Оскільки тестове оточення саме є програмою (причому часто реалізованою не тою мовою програмування, якою написана система), воно саме повинно бути протестовано. Метою тестування тестового оточення є *доказ* того, що тестове оточення ніяким чином не спотворює виконання модуля, що тестується і адекватно моделює поведінку системи.

**Драйвери і заглушки**

Тестове оточення для програмного коду на структурних мовах програмування складається з двох компонентів - драйвера, який забезпечує запуск і виконання модуля, що тестується, і заглушок, які моделюють функції, що викликаються з даного модуля. Розробка тестового драйвера являє собою окрему задачу тестування, сам драйвер повинен бути протестований, щоб виключити невірне тестування. Драйвер і заглушки можуть мати різні рівні складності, необхідний рівень складності вибирається залежно від складності модуля, що тестується і рівня тестування. Так, драйвер може виконувати наступні функції:

1. Виклик модуля, що тестується:

2. 1 + передача в модуль, що тестується вхідних значень і прийом результатів;

3. 2 + висновок вихідних значень;

4. 3 + протоколювання процесу тестування і ключових точок програми.

Заглушки можуть виконувати такі функції:

1. Не проводити ніяких дій (такі заглушки потрібні для коректної збірки модуля, що тестується).

2. Виводити повідомлення про те, що заглушка була викликана:

3. 1 + виводити повідомлення зі значеннями параметрів, переданих у функцію;

4. 2 + повертати значення, заздалегідь задане у вхідних параметрах тесту;

5. 3 + виводити значення, заздалегідь задане у вхідних параметрах тесту;

6. 3 + приймати від тестованого ПЗ значення і передавати їх в драйвер.

Для тестування програмного коду, написаного на процедурній мові програмування, використовуються драйвери, що представляють собою програму з точкою входу (наприклад, функцією main ()), функціями запуску модуля, що тестується і функціями збору результатів. Зазвичай драйвер має як мінімум одну функцію - точку входу, якій передається керування при його виклику.

Функції-заглушки можуть поміщатися в той же файл вихідного коду, що й основний текст драйвера. Імена та параметри заглушок повинні збігатися з іменами і параметрами відповідних функцій реальної системи, які заміщує заглушка. Ця вимога важливо не стільки з точки зору коректної складання системи (при складанні тестового драйвера і тестованого ПЗ може використовуватися приведення типів), скільки для того, щоб максимально точно моделювати поведінку реальної системи з передачі даних. Так, наприклад, якщо в реальній системі є функція обчислення квадратного кореня double sqrt (double value); то, з точки зору складання системи, замість типу double може використовуватися і float, але зниження точності може викликати непередбачувані результати в модулі, що тестується.

Як приклад драйвера і заглушок розглянемо реалізацію стека на мові C, причому значення, що поміщаються в стек, зберігаються не в оперативній пам'яті, а поміщаються в ПЗП (постійний запам'ятовувальний пристрій, пам'ять тільки для читання ROM - Read Only Memory) за допомогою окремого модуля, що містить дві функції – записи даних в ПЗП за адресою і читання даних за адресою .

Формат цих функцій наступний:

void NV\_Read (char \* destination, long length, long offset);

void NV\_Write (char \* source, long length, long offset);

Тут destination - адреса області пам'яті, в яку записується значення, лічену з ПЗП, source - адреса області пам'яті, з якої записується значення в ПЗП, length - довжина записуваної області пам'яті, offset - зміщення відносно початкової адреси ПЗП.

Реалізація стека з використанням цих функцій виглядає наступним чином:

long currentOffset;

void initStack ()

{ currentOffset = 0; }

void push (int value)

{

NV\_Write ((int \*) & value, sizeof (int), currentOffset);

currentOffset + = sizeof (int);

}

int pop ()

{

int value;

if (currentOffset> 0)

{

NV\_Read ((int \*) & value, sizeof (int), currentOffset;

currentOffset-= sizeof (int);

} }

При виконанні цього коду на реальній системі відбувається запис в ПЗП, однак, якщо ми хочемо протестувати тільки реалізацію стека, ізолювавши її від реалізації модуля роботи з ПЗП, необхідно використовувати заглушки замість реальних функцій. Для імітації роботи ПЗП можна виділити достатньо велику ділянку оперативної пам'яті, в якій і проводитиметься запис даних, одержуваних заглушкою.

Заглушки для функцій можуть виглядати наступним чином:

char nvrom [1024];

void NV\_Read (char \* destination, long length, long offset)

{

printf ("NV\_Read called \ n");

memcpy (destination, nvrom + offset, length);

}

void NV\_Write (char \* source, long length, long offset);

{

printf ("NV\_Write called \ n");

memcpy (nvrom + offset, source, length);

}

Кожна з заглушок виводить трасувальне повідомлення і переміщує передане значення в пам'ять, емулює ПЗП (функція NV\_Write), або повертає за посиланням значення, яке зберігається в пам'яті, що емулює ПЗП (функція NV\_Read).

Схема взаємодії тестованого ПЗ (функцій роботи зі стеком) з реальним оточенням (основною частиною системи і модулем роботи з ПЗП) і тестовим оточенням (драйвером і заглушками функцій роботи з ПЗП) показана на Рис. 2 і Рис. 3 .

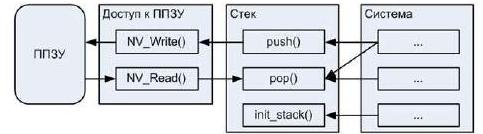


Рис. 2.Схема взаємодії частин реальної системи

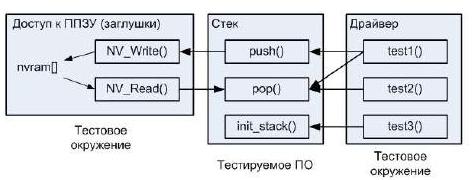


Рис. 3. Схема взаємодії тестового оточення і тестованого ПЗ

**Тестові класи**

Тестове оточення для об'єктно-орієнтованого ПЗ виконує ті ж самі функції, що і для структурних програм (на процедурних мовах). Однак, воно має деякі особливості, пов'язані із застосуванням успадкування та інкапсуляції.

Якщо при тестуванні структурних програм мінімальним тестуємим об'єктом є функція, то в об'єктно-орієнтованим ПЗ мінімальним об'єктом є клас. При застосуванні принципу інкапсуляції всі внутрішні дані класу і деяка частина його методів недоступна ззовні. У цьому випадку тестувальник позбавлений можливості звертатися до своїх тестах до даних класу і довільним чином викликати методи; єдине, що йому доступно - викликати методи зовнішнього інтерфейсу класу.

Існує кілька підходів до тестування класів, кожен з них накладає свої обмеження на структуру драйвера і заглушок.

1. Драйвер створює один або більше об'єктів тестованого класу, всі звернення до об'єктів відбуваються тільки з використанням їх зовнішнього інтерфейсу. Текст драйвера в цьому випадку представляє собою так званий тестуючий клас, який містить по одному методу для кожного тестового прикладу. Процес тестування полягає в послідовному виклику цих методів. Замість заглушок до складу тестового оточення входить програмний код реальної системи, відповідно, відсутня ізоляція тестованого класу. Саме такий підхід до тестування прийнятий зараз у більшості методологій і середовищ розробки. Його класичне назва - *unit testing* (тестування модулів).

2. Аналогічно попередньому підходу, але для всіх класів, які використовує тестований клас, створюються заглушки.

3. Програмний код тестованого класу модифікується таким чином, щоб відкрити доступ до всіх його властивостей і методів. Будова тестового оточення в цьому випадку повністю аналогічно оточенню для тестування структурних програм.

4. Використовуються спеціальні засоби доступу до закритих даних і методам класу на рівні об'єктного або виконуваного коду - скрипти відладчика або accessors в Visual Studio.

Основна перевага перших двох методів: при їх використанні клас працює точно таким же чином, як в реальній системі. Однак у цьому випадку не можна гарантувати, що в процесі тестування буде виконано весь програмний код класу і не залишиться непротестованих методів.

Основний недолік 3-го методу: після зміни вихідних текстів модуля, що тестується не можна дати гарантії того, що клас буде вести себе таким же чином, як і вихідний. Зокрема це пов'язано з тим, що зміна захисту даних класу впливає на спадкування даних і методів іншими класами.

Тестування спадкування - окрема складна задача в об'єктно-орієнтованих системах. Після того, як протестований базовий клас, необхідно тестувати класи-нащадки. Однак, для базового класу не можна створювати заглушки, оскільки в цьому випадку можна припустити можливі проблеми поліморфізму. Якщо клас-нащадок використовує методи базового класу для обробки власних даних, необхідно переконатися в тому, що ці методи працюють. Таким чином, ієрархія класів може тестуватися зверху вниз, починаючи від базового класу. Тестове оточення при цьому може змінюватися для кожної тестованої конфігурації класів.

**Генератори сигналів (подіємо - керований код)**

Значна частина складних програм в даний час використовує ту чи іншу форму взаємодії між процесами. Це обумовлено природною еволюцією підходів до проектування програмних систем, яка послідовно пройшла наступні етапи.

1. Монолітні програми, що містять у своєму коді всі необхідні для своєї роботи інструкції. Обмін даними всередині таких програм проводиться за допомогою передачі параметрів функцій і використання глобальних змінних. При запуску таких програм утворюється один процес, який виконує всю необхідну роботу.

2. Модульні програми, які складаються з окремих програмних модулів з чітко визначеними інтерфейсами викликів. Об'єднання модулів у програму може відбуватися або на етапі складання виконуваного файлу (статична збірка або static linking), або на етапі виконання програми (динамічна збірка або dynamic linking). Перевага модульних програм полягає в досягненні деякого рівня універсальності - один модуль може бути замінений іншим. Однак, модульна програма все одно являє собою один процес, а дані, необхідні для вирішення завдання, передаються всередині процесу як параметри функцій.

3. Програми, що використовують міжпроцесну взаємодію. Такі програми утворюють програмний комплекс, призначений для вирішення загальної задачі. Кожна запущена програма утворює один або більше процесів. Кожен з процесів або використовує для вирішення завдання свої власні дані і обмінюється з іншими процесами тільки результатом своєї роботи, або працює із загальною областю даних, поділюваних між різними процесами. Для вирішення особливо складних завдань процеси можуть бути запущені на різних фізичних комп'ютерах і взаємодіяти через мережу. Перевага використання міжпроцесної взаємодії полягає в ще більшій універсальності - взаємодіючі процеси можуть бути замінені незалежно один від одного при збереженні інтерфейсу взаємодії. Інша перевага полягає в тому, що обчислювальна навантаження розподіляється між процесами. Це дозволяє операційній системі управляти пріоритетами виконання окремих частин програмного комплексу, виділяючи більшу або меншу кількість ресурсів ресурсоємним процесам.

При виконанні багатьох процесів, вирішальних загальну задачу, використовуються кілька типових механізмів взаємодії між ними, спрямованих на вирішення наступних завдань:

 передача даних від одного процесу до іншого;

 спільне використання одних і тих же даних кількома процесами;

 повідомлення про зміну стану процесів.

У всіх цих випадках типова структура кожного процесу являє собою кінцевий автомат з набором станів і переходів між ними. Перебуваючи в певному стані, процес виконує обробку даних, при переході між станами - пересилає дані іншим процесам або приймає дані від них.

Для моделювання кінцевих автоматів використовуються stateflow або SDL-діаграми, акцент в яких робиться відповідно на умовах переходу між станами і пересилає даними.

Мову SDL призначено для розробки подійно - орієнтованих розподілених систем. Ця мова почала розвиватись за сприяння міжнародного комітету ITU ще в 1976 році і є однією з найдавніших розробок в комп'ютерній інженерії. Існує два варіанти цієї мови — текстовий (SDL/PR) та графічний (SDL/GR), синтаксис яких здебільшого співпадає. Застосовувана в моделюванні мова **SDL** (*Specification and Description Language*) — це мова специфікацій та опису. Під **специфікацією** розуміють *точне формальне визначення системи або її частини*, під **описом** — *неформальну специфікацію, яка ілюструє той або інший аспект системи*. Описи використовують на початкових етапах розробки системи або для документування системи. На етапах детального проектування використовують специфікації, за якими передбачається виконання автоматичного генерування програмного коду. Той факт, що для різних етапів розробки системи пропонується одна мова, є суттєвою перевагою SDL, оскільки в такому випадку зникає проблема семантичних розривів.

Так, на Рис. 4 показана схема процесу прийому / передачі даних. Закругленими прямокутниками вказані стану процесу, тонкими стрілками - переходи між станами, великими стрілками - надсилаються дані. Перебуваючи в стані "Старт", процес посилає в зовнішній світ (або процесу, з яким він обмінюється даними) повідомлення про свою готовність до початку сеансу передачі даних. Після отримання від другого процесу підтвердження про готовність починається сеанс обміну даними. У разі надходження повідомлення про кінець даних відбувається завершення сеансу і перехід в стартове стан. У разі надходження невірних даних (наприклад, неправильного формату або з невірною контрольною сумою) процес переходить у стан "Помилка", вийти з якого можливо тільки завершенням і перезапуском процесу.



Рис. 4.Приклад кінцевого автомата процесу прийому-передачі даних

Тестове оточення для такого процесу також повинно мати структуру кінцевого автомата і пересилати дані в тому ж форматі, що і тестований процес. Метою тестування в даному випадку буде показати, що процес обробляє прийняті дані відповідно до вимог, формати переданих даних коректні, а також що процес під час своєї роботи дійсно проходить всі стану кінцевого автомата, моделює його поведінку.

***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. Лавріщева К.М. Програмна інженерія. Електронний підручник: – URL: <http://csc.knu.ua/uk/library/books/lavrishcheva-6.pdf>
2. Сайкс Д. Тестирование объектно-ориентированного программного обеспечения. Практическое пособие / Д.Сайкс, Д.Макгрегор. –К.: Диасофт, 2002. –432 с.
3. Соммервил И. Инженерия программного обеспечения / И.Соммервил.–М. : Издательский дом «Вильямс», 2002. –623 с.
4. Тамре Л. Введение в тестирование программного обеспечения / Л. Тамре. –М.: Издательский дом «Вильямс», 2003. –368 с.
5. Блэк Р. Ключевые процессы тестирования / Р.Блэк. – М.: Лори, 2006. –544 с.
6. Винниченко И.В. Автоматизация процессов тестирования / И.В.Винниченко. – СПб.: Питер, 2005. –208 с.
7. Криспин Л. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд / Л.Криспин, Д.Грегори. –М.: Издательский дом «Вильямс», 2010. –464 с.
8. Майерс Г. Искусство тестирования программ / Г. Майерс. — Пер. с англ.— М.: Финансы и статистика, 1982. — 172 с.
9. Фолк Д. Тестирование программного обеспечения / Д.Фолк, Е.К. Нгуен, С.Канер. – К.: Диасофт, 2003. –400 с.
10. ДСТУ 2873-94 Системи оброблення інформації. Програмування. Терміни та визначення
11. ДСТУ 2853-94 Програмні засоби ЕОМ. Підготовлення і проведення випробувань
12. ДСТУ ISO/IEC 12207:2016 Інженерія систем і програмного забезпечення. Процеси життєвого циклу програмного забезпечення (ISO/IEC 12207:2008, IDT)
13. Интуит. Верификация программного обеспечения. Лекция 2: Тестирование программного кода (методы+окружение). 3.3.1. Драйверы и заглушки. – URL: <https://intuit.ru/studies/professional_skill_improvements/1463/courses/209/lecture/5385?page=2> (дата звернення 24.01.2021)

**Контрольні запитання.**

1. Що мається на увазі під тестуванням програмного продукту?
2. Скільки типів програмних помилок Вам відомі?
3. Які помилки відносяться до синтаксичних?
4. Які помилки відносяться до семантичних?
5. Чим визначається склад і зміст документації, супутньої процесу тестування?
6. За якими ознаками прийнято робити класифікацію видів тестування?
7. Що таке функціональне тестування?
8. Які функціональні тести Вам відомі?
9. Що таке нефункціональне тестування?
10. Які нефункціональні тести Вам відомі?
11. Що таке модульне тестування?
12. Що Вам відомо про тестові скрипти?