Лекція 16. Модульне тестування

1. *Чому тестування важливе. Як написати хороший тест.*
2. *Як додати до рішення проект модульного тестування.*
3. *Структура класу тестів. Методи-тести.*
4. *Різновиди статичних методів класу Assert.*

Інтерес до тестування і до розробки програм через тестування (Test Driven Development, TDD) не обмежується якимось одним середовищем чи мовою програмування. Автоматизоване тестування стало невід'ємною ознакою *технології гнучкої розробки програмного забезпечення* (Agile software development), і кожен програміст, хто турбується про підвищення якості свого програмного забезпечення, мав би взяти його на озброєння.

Ні тестування, ні побудова наборів тестів не є чимось новим – всі знають, що тестування є хорошим способом виявлення помилок. Технологія екстремального програмування (eXtreme Programming) піднесла тестування до рангу однієї з головних практик, наголошуючи на важливості *автоматизованих тестів*, і тим самим перетворила тестування з рутини, яку так не люблять програмісти, на веселе і продуктивне заняття. Спільнота Smalltalk має давні традиції тестування, оскільки покроковий стиль розробки підтримує саме середовище програмування. Зазвичай Smalltalk-розробник писав тести в робочому вікні (workspace), як тільки закінчував програмувати черговий метод, і одразу ж випробовував його. Іноді тест вставляли як коментар на початку того методу, який він мав перевіряти, або включали до класу в якості методів-прикладів тести, що потребували певних початкових налаштувань. Недоліки цих підходів очевидні: тести в робочому вікні недоступні іншим програмістам, які працюють з тим самим кодом і змінюють його. З цього боку краще виглядають коментарі та методи-приклади, проте нема простого способу, щоб відслідковувати їх наявність і запускати автоматично. Не запущений тест не допоможе Вам виявити помилку! Більше того, метод-приклад не повідомить читачеві очікуваний результат. Ви можете запустити такий тест і побачити, іноді несподіваний, результат, але Ви не знатимете чи він правильний, чи підтверджує тест правильну поведінку методу або класу.

Справді, розробники багатьма мовами програмування оцінили важливість модульного тестування, і середовища *xUnit* тепер існують для багатьох мов, включно з Java, Python, Perl, .Net і Oracle.

SUnit має декілька переваг: Ви можете писати тести, що містять визначення правильного результату, і тому придатні для самоперевірки, Ви можете також об'єднувати тести в групи, визначати контекст їхнього виконання і автоматично запускати цілі групи тестів. За допомогою SUnit Ви зможете за невеликий час створити вичерпний набір тестів для будь-якої мети, тому ми закликаємо Вас замість покрокового тестування фрагментів коду в робочому вікні використовувати SUnit та всі його переваги зберігання і автоматичного виконання тестів.

Чому тестування важливе

Багато програмістів, на превеликий жаль, вважають, що написання тестів – це даремна трата часу. Адже це *інші* програмісти помиляються, а *вони* ніколи не пишуть з помилками. Багато хто з нас говорив: «Я б писав тести, якби мав більше часу». Якщо Ви ніколи не робите помилок, і Ваша програма ніколи не змінюватиметься у майбутньому, тоді тестування справді є даремним витрачанням Вашого часу. Проте, це, швидше за все, означає, що Ваша програма або тривіальна, або її не використовуєте ні Ви, ні будь хто інший. Про тести потрібно думати як про інвестицію в майбутнє: корисний сьогодні набір хороших тестів стане просто *незамінним* у майбутньому, коли зазнає змін Ваша програма чи середовище, в якому вона виконується.

Тести виконують декілька завдань одночасно. Перш за все вони слугують документацією тієї функціональності програми, яку покривають. Більше того, така документація є активною: проходження тестів без помилок засвідчує її актуальність. По друге, тести допомагають розробникам переконатися, що щойно зроблені зміни певної

частини коду не порушують функціонування решти системи, або локалізувати помилки в протилежному випадку. І нарешті, якщо Ви пишете тести одночасно з програмою, або навіть перед нею, Ви починаєте думати про функціональність, яку проектуєте, і швидше про те, як це виглядатиме для користувача, ніж як це реалізувати.

Якщо Ви спершу пишете тести, а потім – код, то Ви змушені встановлювати контекст, у якому виконуватиметься проектована функціональність, спосіб взаємодії з кодом користувача й очікувані результати. Спробуйте так програмувати, і Ви побачите, що Ваш код стане кращим.

Культура тестування віддавна притаманна Smalltalk-спільноті, оскільки одразу після написання методу ми пишемо в вікні розробки невеликий тестовий вираз, щоб перевірити метод. Така практика підтримує жорстко встановлений покроковий цикл розробки у Pharo, проте не забезпечує всіх переваг тестування, оскільки тести не зберігаються і не запускаються автоматично. Більше того, часто трапляється так, що контекст тестів «робочого вікна» стає невизначеним, і читач сам повинен інтерпретувати результати тестування та вирішувати, правильні вони, чи ні.

Ми не можемо всесторонньо протестувати жодного справжнього застосунку. Покриття тестами цілої програми є просто неможливим, тому не задавайтеся такою метою. Навіть після застосування досконалого набору тестів окремі помилки можуть закрастися до аплікації і залягти там «на дно», очікуючи слушної нагоди, щоб зруйнувати всю систему. І якщо так справді станеться, скористайтеся моментом на свою користь. Як тільки знайдено не покриту тестами помилку, напишіть тест, що мав би її виявляти, запустіть його і переконайтеся, що він завершується невдачею. Тепер Ви можете перейти до виправлення помилки. Успішне проходження тесту засвідчить, що Ви зробили бажане.

Як написати хороший тест

Уміння писати хороші тести найлегше здобути на практиці. Розглянемо умови, за яких тести будуть найефективнішими.

1. Тести мають бути повторюваними. У Вас має бути можливість запускати їх так часто, як би Ви хотіли. Щоразу тести повинні повертати ті самі результати.
2. Тести повинні виконуватися без втручання людини. У Вас має бути можливість запускати їх будь-коли, навіть, уночі.
3. Тести повинні описувати програму. Завдання кожного тесту – перевірити одну рису деякої частини коду. Тест мав би діяти, як сценарій, прочитавши який і Ви, і будь-хто інший зрозумів би функціональність цієї частини коду.
4. Тести повинні бути стабільними і змінюватися рідше ніж код, який вони перевіряють, адже ніхто не хотів би переписувати весь набір тестів після кожної зміни в програмі. Єдиний спосіб досягнення такої властивості – писати тести, що взаємодіють з тестованим класом через його відкритий інтерфейс. Звичайно, Ви можете писати тести і для допоміжного приватного методу, якщо відчуваєте, що він досить складний і потребує тестування, але будьте готові до того, що такі тести доведеться змінити або повністю вилучити, якщо Ви придумаєте кращу реалізацію методу.

Як наслідок цих властивостей, кількість тестів мала б бути приблизно пропорційна кількості тестованих функцій, методів. Зміна однієї властивості системи не повинна «завалювати» всі тести, а лише обмеженого числа з них. Це важливо, бо невиконання сотні тестів є набагато серйознішим попередженням, ніж невиконання десяти. Проте, не завжди вдається досягти такого ідеалу, зокрема, якщо зміни зачіпають ініціалізацію об’єкта або налаштування тестів, то ймовірно всі тести зазнають невдачі.

Екстремальне програмування пропагує написання тестів перед написанням програми. Здається, що такий підхід суперечить нашим навикам розробників програмного забезпечення. Що ж тут скажеш – спробуйте! Спробуйте, і Ви переконаєтеся, що формулювання тестів перед складанням програми допомагає краще зрозуміти, що власне ми хочемо запрограмувати, сформувати бачення функціональності класу та спроектувати його інтерфейс, впізнати, чи ми вже досягли бажаного. Більше того, розробка через тестування дає нам сміливість швидко рухатися вперед без побоювань забути щось важливе.

Додавання проекту модульного тестування

Припустимо, є деякий *Solution*, що містить один проект з консольною програмою, наприклад, проект *ConsoleApplication*. Нехай до його складу входять такі файли:

- *Functions.h* – оголошення прототипів декількох функцій, оголошення типів;
- *Functions.cpp* – реалізація всього оголошеного у файлі заголовків;
- *Program.cpp* – програма, що використовує ресурси з файла заголовків.

Завдання: написати модульні тести, щоб перевірити правильність роботи оголошених функцій і типів; тести помістити в окремий проект.

Послідовність підготовчих кроків:

1. `Solution [right click] Add > New project… Test / Native Unit Test Project -> Name: UnitTest1 [Ok]` // ім'я проекту стане також іменем простору імен для тестів, тому варто давати осмислене.
2. Вставити директиву `#include "..\ConsoleApplWithTest\Functions.h"` до файла «*unittest1.cpp*», щоб вказати компіляторові, де оголошено прототипи функцій, щоб була змога викликати їх у тестах.
3. Додати до проекту тестів посилання на проект з оголошеними функціями (щоб проект тестів завжди відслідковував усі зміни в оголошенні та визначенні функцій та автоматично їх перекомпільовував їх перед запуском тестів) одним з таких способів:
   a. `UnitTest1 [right click] Add > References… [Add new reference…] Projects √ C:\FullPathTo\ConsoleApplication [Ok]`
   b. `UnitTest1 [right click] Properties… Common Properties / References [Add new reference…] Projects √ C:\FullPathTo\ConsoleApplication [Ok]`
4. Вказати в проекті модульних тестів редакторові зв'язків, де збережено об'єктний код функцій, що підлягають тестуванню: `UnitTest1 [right click] Properties… Configuration Properties / Linker / Input > Additional Dependencies: C:\FullPathTo\ConsoleApplication\Debug\Functions.obj [Ok]`
   Підготовку завершено.

Можливий інший варіант початкових налаштувань. Якщо потрібно відтестувати код великого обсягу: десяток функцій, оголошених у декількох файлах; декілька класів у різних файлах – тоді є сенс скомпілювати його в бібліотеку, яку згодом можна використовувати в різних проектах. Зробимо ті самі припущення: *Solution*, що містить один проект *ConsoleApplication*(*Functions.h*, *Functions.cpp*, *Program.cpp*). Щоб створити бібліотеку, доведеться цей проект поділити на два і відокремити програму (функцію *main*) від решти функцій. *Спосіб 1*: додати до *Solution* порожній проект, перемістити в нього *Functions.h*, *Functions.cpp*, у вікні властивостей проекту встановити `Configuration Properties / General > Configuration Type = Static Library (.lib)`. *Спосіб 2*: додати до *Solution* консольну аплікацію, перемістити в неї *Program.cpp*, у вікні властивостей попереднього проекту встановити `Configuration Properties / General > Configuration Type = Static Library (.lib)`. Два створені проекти потрібно поєднати між собою. Для цього у консольній програмі виконати кроки 2 і 3, описані вище (*include* та *reference*).

Якщо тестують бібліотеку, то підготовчі кроки спрощуються: крок 4 виконувати не потрібно, достатньо в проекті модульних тестів додати посилання на бібліотеку та включити файл заголовків (кроки 2, 3).

TEST / Windows > Test Explorer
Run All

У файлі *unittest1.cpp* додають класи, методи, та всі потрібні засоби для успішного виконання тестованих одиниць коду.

```
TEST_CLASS(TestFunctions) // ім’я класу вказує на його призначення

TEST_METHOD(TestMethodAddEqual) // ім’я методу розкриває суть тесту

// Перевірка, чи рівні два значення в сенсі expected == result
Assert::AreEqual(expected, result);
// додатково AreNotEqual, AreSame (&expected == &result) AreNotSame

// Розпізнавач, діє як AreEqual(true, expression)
Assert::IsTrue(expression);
// Інші розпізнавачі, IsFalse, IsNull, IsNotNull

// Розпізнає, чи лямбда згенерував виняток заданого типу
Assert::ExpectException<BadSize>([a]{ sortSelect(a, -10); });

// атрибути – додаткові можливості структурування тестів:
// їхньої приналежності, призначення авторства тощо
BEGIN_TEST_MODULE_ATTRIBUTE()
     TEST_MODULE_ATTRIBUTE(L"Project", L"Вивчаємо тестування")
END_TEST_MODULE_ATTRIBUTE()

BEGIN_TEST_METHOD_ATTRIBUTE(TestMethodSortZero)
     TEST_DESCRIPTION(L"Перевіримо, чи зреагує функція на розмір = 0")
END_TEST_METHOD_ATTRIBUTE()

// метод виконується один раз перед усіма методами класу
// налаштовує контекст виконання класу
TEST_CLASS_INITIALIZE(ClassInitialize)

// метод виконується один раз після усіх методів класу
// звільняє ресурси, захоплені методом TEST_CLASS_INITIALIZE
TEST_CLASS_CLEANUP(ClassCleanup)

// метод виконується перед кожним методом класу
TEST_METHOD_INITIALIZE(copyArr)

// метод виконується після кожного методу класу
TEST_METHOD_CLEANUP(eraseCopy)

// виводить повідомлення у вікно Output
Logger::WriteMessage("In Class Cleanup");
```

Додаткові ресурси

Документація по інфраструктурі модульного тестування Visual Studio
https://docs.microsoft.com/ru-ru/visualstudio/test/microsoft-visualstudio-testtools-cppunittestframework-api-reference?view=vs-2019
https://docs.microsoft.com/ru-ru/visualstudio/test/unit-test-your-code?view=vs-2019

Добірка навчальних дописів
https://pdsa.com/Blog

Опис альтернативної інфраструктури Google Tests
https://www.codeproject.com/Tips/5247661/Google-Translate-API-Usage-in-Csharp