

МIНIСТЕРСТВО ОСВIТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**Лекція №**\_\_

з дисципліни “ Інформаційна стійкість комп’ютерних технологій та мереж ”

тема “Модульне тестування”

|  |  |  |
| --- | --- | --- |
| Підготував  студент VI курсу  групи КВ-64М  Подольський Сергій Валентинович  (*прізвище, ім’я, по батькові*) |  |  |

Зміст

[Базові поняття та визначення 3](#_Toc308044073)

[Властивості модульного тесту 5](#_Toc308044074)

[Інтеграційне тестування 6](#_Toc308044075)

[Розробка через тестування 7](#_Toc308044076)

[Бібліотеки для модульного тестування 10](#_Toc308044077)

[Усунення зовнішніх залежностей 12](#_Toc308044078)

[Питання для самоконтролю 13](#_Toc308044079)

**Поняття та терміни**: модуль (*unit*), модульний тест (*unit-test*), інтеграційний тест (*integration test*), заміна або фіктивний об’єкт (*fake*), заглушка (*stub*), макет (*mock*), зовнішня залежність (*external dependency*), рефакторинг (*refactoring*).

# Базові поняття та визначення

Модульне тестування не є новою концепцією в розробці програмного забезпечення. Вона виникла з перших днів мови програмування Smalltalk в 1970-х роках і показує себе знову і знову як один з кращих способів, за допомогою яких розробник може поліпшити якість коду, отримуючи більш глибоке розуміння функціональних вимог до класу або методів.

Кент Бек ввів поняття модульного тестування в Smalltalk, це дало поштовх до багатьох інших мов програмування, що робить модульне тестування надзвичайно корисною практикою програмування. Перш ніж ми перейдемо далі, ми повинні дати більш точне визначення модульного тестування.

**Визначення**: *модульний тест – це фрагмент коду (як правило, метод), який викликає інший фрагмент коду і перевіряє правильність деяких припущень після цього. Якщо припущення виявляться неправдивими, модульний тест не пройшов випробування.*

Модуль в термінах класичного ООП – це метод або функція.

Модульне тестування проводиться над системою, що тестується (system under test, SUT).

**Визначення**: *SUT розшифровується як тестована система, та інколи використовується термін CUT (class under test or code under test). Коли ми тестуємо - то ми посилаємося на річ, яку ми перевіряємо, як на SUT.*

Більшість розробників програмного забезпечення вже реалізовували так чи інакше деякі види тестування на свій розсуд. Практично неможливо коли-небудь зустріти розробника, який не перевіряв би свій код перед тим, як передати його.

Можливо, всім доводилося використовувати консольний додаток, який викликає різні методи класу або компоненти, чи, можливо, деякі особливим чином створені графічні інтерфейси, побудовані на WinForm або WebForm UI, які перевіряли функціональність цих класів або компонент. Кінцевим результатом є те, що ви зробили певні у деякій мірі перевірки, що код працює досить добре, щоб передати його комусь іншому.

На рис. 1 зображено, як більшість розробників перевіряють свій код. Інтерфейс користувача може змінитися, але картина, як правило, однакова: за допомогою ручного зовнішнього інструменту щось перевіряється багаторазово, або ж додаток запускається в повному обсязі для перевірки його поведінки вручну.

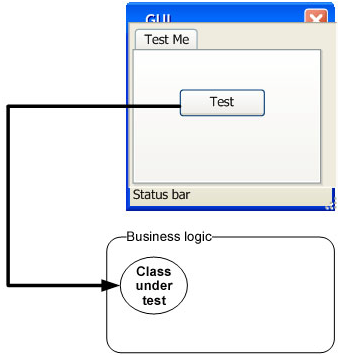


Рис.  1. В класичному тестуванні розробники використовують GUI (графічний інтерфейс користувача), щоб викликати якісь дії класу, який вони хочуть перевірити. Потім вони перевіряють результати.

Ці тести могли бути корисними і вони можуть наблизитися до класичного визначення модульного тестування, але вони далекі від того, як ми визначимо якісний модульний тест в даній лекції. Це підводить нас до перших і найважливіших питань, з якими розробнику доводиться зустрічатися при визначенні якості модульного тестування: що є насправді модульне тестування, а що ні.

# Властивості модульного тесту

Модульний тест повинен мати такі властивостями.

* Він повинен бути автоматизованим і повторюваним.
* Він повинен бути легким у реалізації.
* Як тільки він написаний, він повинен залишатися для подальшого використання.
* Будь-хто повинен мати можливість запустити його.
* Він повинен запускатися лише натисканням однієї кнопки.
* Він повинен швидко виконуватись.

Багато людей плутають здійснення тестування свого програмного забезпечення з концепцією модульного тесту. Для початку, слід задати собі наступні питання щодо тестів, які ви писали до сих пір.

* Чи можу я запускати і отримувати результати від модульного тесту, що я написав, два тижні або кілька місяців чи років тому?
* Чи може хто-небудь з членів моєї команди запустити та отримати результати від модульних тестів, що я написав два місяці тому?
* Чи можу я виконати всі модульні тести, що я написав, не довше, ніж за кілька хвилин?
* Чи можу я виконати всі модульні тести, які я написав, за допомогою одного лише натискання кнопки?
* Чи можу я написати простий модульний тест не довше, ніж за кілька хвилин?

Якщо ви відповіли "ні" на будь-яке з цих питань, є велика ймовірність того, що ви реалізуєте не модульне тестування. Це, безперечно, також якесь тестування, і це так само важливе, як модульне тестування, але воно має свої недоліки у порівнянні з тестами, які дозволять вам відповісти "так" на всі ці питання. "Що я робив до сих пір?", запитаєте ви. Ви робили інтеграційне тестування.

# Інтеграційне тестування

Що відбувається, коли виходить з ладу ваша машина? Як ви дізнаєтеся, в чому проблема, не кажучи вже про те, як її виправити? Двигун складається з багатьох деталей, що працюють разом, кожна побудована на основі інших, щоб допомагати виробляти кінцевий результат: рухомий автомобіль. Якщо автомобіль перестає рухатися, несправність може бути у будь-якої з цих частин, або більше, ніж в однієї. Це все є інтеграція цих частин, що робить автомобіль рухомим. Якщо тест не пройдено, всі частини несправні разом, якщо тест вдається, всі частини справні.

Те ж саме відбувається в програмному забезпеченні. Підхід, яким більшість розробників перевіряють остаточну функціональність, – це за допомогою остаточного функціонального інтерфейсу користувача. Натискання кнопки викликає деякі послідовності дій, різних класів і компонентів, які працюють разом для отримання остаточного результату. Якщо тест не пройдено, всі ці програмні компоненти зазнають невдачі подібно до команди (колективу), і може бути важко зрозуміти, що призвело до краху всієї операції (див. рис. 2).

Ви можете мати багато точок відмови в інтеграційному тесті. Всі модулі повинні працювати разом, і кожен з них може мати несправності, що ускладнює пошук джерела несправності.

**Визначення**: *інтеграційне тестування – це тестування двох або декількох залежних модулів програмного забезпечення в якості групи.*

Тепер, коли ми розглянули важливі властивості, які повинен мати модульний тест, давайте дамо більш повне визначення якісного модульного тесту.

**Визначення**: *модульний тест – це автоматизований фрагмент коду, який викликає метод або клас, що проходить випробування, а потім перевіряє деякі припущення про логічну поведінку цього методу або класу. Модульний тест майже завжди написаний з використанням бібліотеки для модульного тестування (Unit Testing Framework). Він може бути написаний швидко і швидко виконується. Він повністю автоматизований, надійний, легко читається і супроводжується.*



Рис.  2. Багато модулів, що інтегровані разом, можуть мати багато точок несправності

# Розробка через тестування

Як тільки ми знаємо, як писати структуровані, легкі в обслуговуванні і жорсткі тести за допомогою бібліотеки для модульного тестування, наступне питання – коли саме час писати тести. Багато хто вважає, що найкращий час, щоб писати тести для програмного забезпечення, – це після того, як програмне забезпечення було написано, але все більше число людей воліють писати модульні тести перш, ніж код програми буде написаний. Такий підхід називається *спочатку-тест* (*test-first*) або *розробка через тестування* (*Test-Driven Development, TDD*).

На рис. 3 показані відмінності між традиційною розробкою та розробкою через тестування. Зверніть увагу на спіральний характер процесу: написати тест, написати код, рефакторинг, написати наступний тест. Це показує інкрементальний характер TDD: маленькі кроки призводять до високої якості кінцевого результату.

Рис.  3. Традиційна розробка (зліва) та розробка через тестування (справа)

Техніка розробки через тестування досить проста:

1. написати тест, що не виконується, щоб довести, що код або функціональність відсутня в кінцевому продукті. Тест пишеться так, ніби робочий код вже працює, таким чином невиконання тесту означає, що є помилка в коді виробничого призначення. Наприклад, якщо ми хочемо додати нову функцію до деякого класу-калькулятора, який запам’ятовує значення останнього результату, ми б написали тест, який перевіряє, що останнє число – це дійсно число. Тест зазнає невдачі, тому що ми ще не реалізували цю функціональність.
2. Переконайтеся, що тест проходить перевірку успішно, написавши код, який відповідає очікуванням вашого тесту. Він повинен бути написаний якомога простіше.
3. Рефакторинг коду. Якщо тест пройдено, ви маєте вибір: перейти до наступного модульного тесту, або реорганізовувати код, щоб зробити його більш зручним для читання, виконати видалення дублювання коду і так далі. Рефакторинг може бути зроблено після написання кількох тестів або після запису кожного тесту. Це важлива практика, тому що внаслідок неї ваш код стає легше читати і підтримувати, в той же час не порушуючи проходження всіх раніше написаних тестів.

**Визначення**: *рефакторинг – це зміна фрагмента коду без зміни його функціональності. Якщо ви коли-небудь перейменували метод, ви зробили рефакторинг. Якщо ви коли-небудь розбивали великий метод на декілька більш дрібних викликів методів, ви здійснювали рефакторинг коду. Код, як і раніше, робить те ж саме, але його стає легше підтримувати, читати, проводити налагодження і зміни.*

Зазначені кроки виглядають досить просто, але за ними приховано багато мудрості. Здійснена правильним чином розробка через тестування може зробити ваш код якіснішим, зменшити кількість помилок, підвищити вашу упевненість в коді, скоротити час, необхідний для пошуку помилок, поліпшити дизайн (проектування) вашого коду. Якщо розробка через тестування буде виконана неправильно, може статися так, що проект відставатиме від графіку, ви почнете витрачати свій час, знизите мотивацію і знизите якість коду. Це палиця з двома кінцями, і багато людей проходять для цього важкий шлях. Якщо ви пишете тести після написання коду, ви припускаєте, що тест в порядку, тому що він проходить, коли це могло бути зумовлено помилками в ваших тестах. Знаходження помилок у тестах є однією з найнеприємніших речей, які можна собі уявити. TDD є одним з кращих способів запобігання того, щоб тести не доходили до такого стану.

# Бібліотеки для модульного тестування

Розглянемо переваги, які дає вам як розробнику інтегроване середовище розробки (IDE). Бібліотеки для модульного тестування пропонують аналогічні переваги для тестування.

До сьогоднішнього дня у багатьох інтегрованих середовищ розробки для інших середовищ (наприклад, Unix) етапи отримання остаточного двійкового коду на виході були не такими простими, і може знадобитися ручний виклик інших зовнішніх інструментів, щоб виконати частину цього великого завдання. При використанні сучасних IDE, таких як Visual Studio .NET або Eclipse для Java ви робите всі ваші задачі кодування в цьому середовищі, в структурованому вигляді. Ви пишете код, компілюєте його, ви вбудовуєте будь-які ресурси (наприклад, графіку і текст) в нього, і ви створюєте остаточний двійковий код – вся розробка та компіляція з ними не більше, ніж кілька натискань клавіш.

Виконання таких речей повністю вручну буде схильним до помилок і буде віднімати багато часу. Ці проблеми зменшують за рахунок інструментів. Таким же чином, бібліотеки для модульного тестування допомагають розробникам писати тести більш швидко з набором відомих API, виконувати ці тести автоматично і переглядати результати цих тестів простіше.

До цих пір, тести, які ви розробляли, були досить обмежені:

* *Вони не були структуровані*. Потрібно було винаходити «велосипед» кожен раз, коли потрібно перевірити функцію. Один тест може виглядати як консольний додаток, інший використовує користувальницький інтерфейс форми, а інший використовує веб-форми. У вас немає зайвого часу, щоб витрачати його на таке тестування, тести не задовольняють вимогу «простої реалізації».
* *Вони не були повторюваними*. Ні ви, ні члени вашої команди не можуть запускати тести, які ви писали раніше. Це порушує вимогу «повторюваності» і не дає вам можливості знаходити регресійні помилки. За допомогою бібліотеки ви можете легко і автоматично писати тести, які повторюються.
* *Вони не покривали весь ваш код*. Тести не перевіряли весь код, який має важливе значення. Це означає весь код з логікою в ньому, тому що кожен з модулів міг би містити потенційну помилку (геттери (getters) та сеттери (setters) властивостей (properties) класу не враховуються як логіка, якщо у вас немає якоїсь логіки в них). Якби було легше писати тести, ви були б більш схильні писати більше їх і одержувати більш повне покриття.

Коротше кажучи, те, чого вам не вистачало, ­– це бібліотека для написання, запуску, а також перегляду модульних тестів та їх результатів. На рис. 4 показано області розробки програмного забезпечення, де бібліотека модульного тестування має вплив.



Рис.  4. Написання модульних тестів як коду, використовуючи бібліотеки для модульного тестування. Далі тести запускаються з окремого інструменту для модульного тестування.

# Усунення зовнішніх залежностей

**Визначення**: *зовнішня залежність (external dependency) – це об'єкт у вашій системі, з яким взаємодіє ваш код, що тестується, і над яким у вас немає контролю (загальні приклади – файлові системи, потоки, пам'ять, час і так далі).*

У програмуванні використовуються заглушки, щоб обійти проблеми зовнішніх залежностей.

**Визначення**: *заглушка (stub) – це керована заміна існуючих залежностей (або співпрацюючих елементів) в системі. За допомогою заглушки ви можете тестувати свій код без розгляду залежностей безпосередньо.*

**Визначення**: *тестування взаємодії (interaction testing) – це тестування того, як об'єкт посилає дані на вхід або отримує дані від інших об'єктів, як цей об'єкт взаємодіє з іншими об'єктами.*

Різниця між макетами і заглушками важлива, тому що багато сучасних інструментів і бібліотек (а також статті) використовують ці терміни, щоб описати різні речі. Є дуже багато плутанини у тому, що кожен термін означає і багато людей використовують їх як синоніми.

**Визначення**: *макет об'єкта (mock object) – фіктивний об'єкт (fake object) у системі, який вирішує, коли модульний тест проходить успішно, а в якому випадку ні. Це робиться шляхом перевірки, чи взаємодіяв об'єкт, що тестується так, як і очікувалося з фіктивним об'єктом. Зазвичай, використовується не більше, ніж один макет на тест.*

І заглушки, і макети використовуються яз заміни, підробки у системі.

**Визначення**: *фіктивні об’кти, заміни (fakes) – це загальний термін, який може бути використаний для опису або заглушки, або макета об'єкта (рукописного чи іншого), тому що вони обидва схожі на реальний об'єкт. Чи є фіктивний об’єкт заглушкою або макетом, залежить від того, як він використовується в поточному тесті. Якщо він використовується для перевірки взаємодії – це макет об'єкта. В іншому випадку – це заглушка.*

# Питання для самоконтролю

1. Що таке модуль (unit)?
2. Що таке модульний тест (unit test)? Дайте визначення канонічного, якісного модульного тесту.
3. За якими критеріями можна визначити, чи є даний тест модульним? Чим характеризується модульний тест?
4. Дайте визначення інтеграційному тесту. Чим інтеграційний тест відрізняється від модульного тесту?
5. Які переваги мають модульні тести перед іншими видами тестів?
6. Що таке розробка через тестування (test-driven development)? Які переваги вона має перед традиційною розробкою?
7. Для чого використовуються бібліотеки для модульного тестування (unit testing frameworks)?
8. Що таке рефакторинг, для чого він використовується?
9. Дайте визначення зовнішньої залежності (external dependency).
10. Що таке фіктивні об’єкти або заміни (fake objects)? Які фіктивні об’єкти використовуються для усунення зовнішніх залежностей і яким чином їх обирати?
11. Що таке заглушки (stubs)? Дайте визначення.
12. Що таке макет (mock object)? Дайте визначення.
13. Назвіть основні відмінності макетів (mock object) та заглушок (subs). Зробіть їх порівняльну характеристику за властивостями та областями застосування.