

МIНIСТЕРСТВО ОСВIТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**Лабораторна робота №**\_\_

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

[Зміст 2](#_Toc308894324)

[1. Теоретичні відомості 3](#_Toc308894325)

[1.1. Базові поняття та визначення 3](#_Toc308894326)

[1.2. Властивості модульного тесту 5](#_Toc308894327)

[1.3. Інтеграційне тестування 6](#_Toc308894328)

[1.4. Розробка через тестування 8](#_Toc308894329)

[1.5. Бібліотеки (платформи) для модульного тестування 11](#_Toc308894330)

[1.6. Усунення зовнішніх залежностей 13](#_Toc308894331)

[2. Порядок виконання роботи 14](#_Toc308894332)

[3. Завдання на лабораторну роботу 25](#_Toc308894333)

[4. Питання для самоконтролю 28](#_Toc308894334)

**Мета роботи**: вивчення методики написання модульних тестів із застосуванням бібліотек для модульного тестування, здобуття базових навичок рефакторингу з метою підвищення придатності коду для тестування.

**Необхідні програмні засоби**:

* ОС Windows XP + .NET Framework 4.0, Windows 7 або вище;
* Visual Studio 2010 Express Edition або вище;
* NUnit 2.5.10 або вище;
* Rhino Mocks, Typemock Isolator або Moq.

**Зміст роботи**: під час виконання даної роботи студенти набувають практичні навички програмування через тестування, написання модульних тестів, роботи з бібліотеками (платформами) для модульного тестування, а також знайомляться з основними техніками усунення зовнішніх залежностей та виконання рефакторингу.

**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*).

# Теоретичні відомості

## 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.

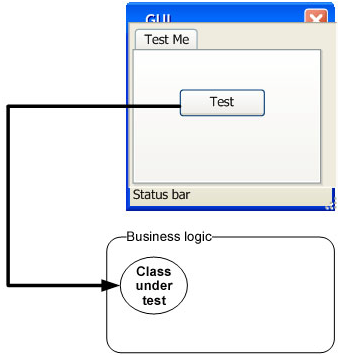


Рис.  1. В класичному тестуванні розробники використовують GUI (графічний інтерфейс користувача), щоб викликати якісь дії класу, який вони хочуть перевірити. Потім вони перевіряють результати.

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 лабораторній роботі. 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).

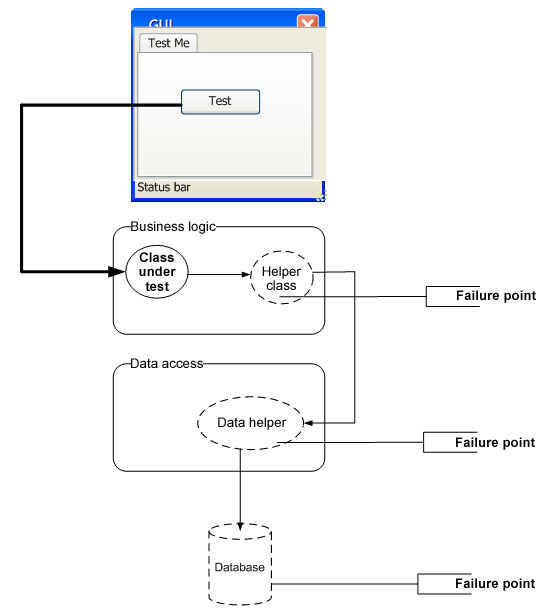


Рис.  2. Багато модулів, що інтегровані разом, можуть мати багато точок несправності

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.*

## Розробка через тестування

Як тільки ми знаємо, як писати структуровані, легкі в обслуговуванні і жорсткі тести за допомогою бібліотеки (платформи) 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.

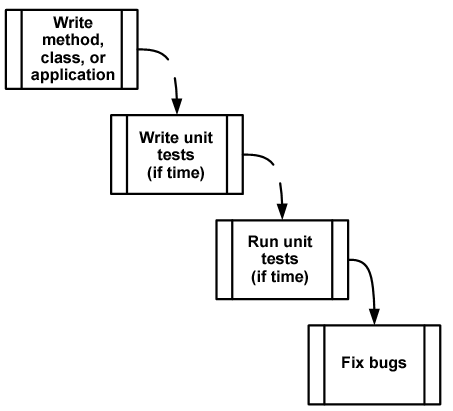
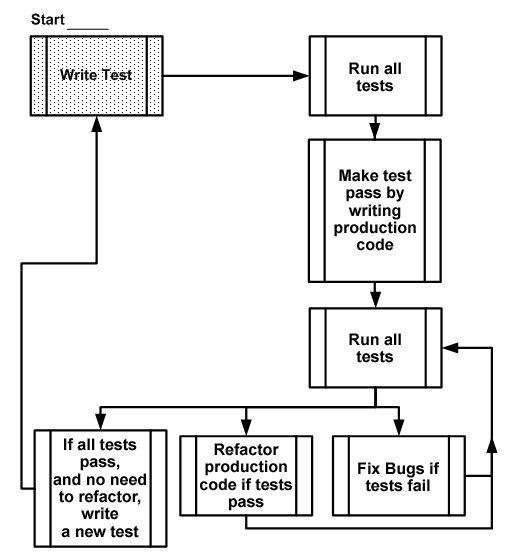
 

Рис.  3. Традиційна розробка (зліва) та розробка через тестування (справа)

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, це може призвести до того, 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 (платформи) для модульного тестування

Розглянемо переваги, які дає вам як розробнику інтегроване середовище розробки (IDE). Бібліотеки (платформи, Frameworks) 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 сих 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 (GUI), 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 (платформи) 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 їх більше число і одержувати більш повне покриття коду.

Коротше кажучи, те, чого вам не вистачало, ­– це бібліотека (платформа, Framework) 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. Проте це неправильно.

**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 фікції, підробки у системі.

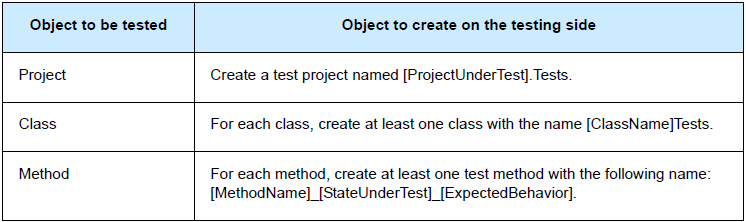
**Визначення**: *фіктивні об’кти, заміни (fakes) – це загальний термін, який може бути використаний для опису або заглушки, або макета об'єкта (рукописного чи іншого), тому що вони обидва схожі на реальний об'єкт. Чи є фіктивний об’єкт заглушкою або макетом, залежить від того, як він використовується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

# Порядок виконання роботи

В межах даної лабораторної роботи пропонується ознайомлення з основами написання модульних тестів на прикладі платформи .NET та використання утиліти і платформи NUnit, яка на сьогоднішній день є де-факто інструментарієм для тестування серед C# розробників.

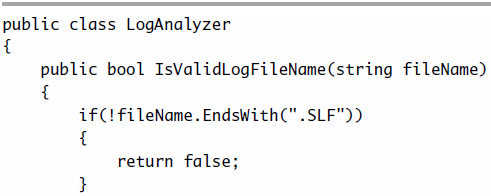
Перш, ніж перейти до безпосереднього написання тестів, розглянемо структуру тестового проекту та коду, що містить класи з тестами. Зазвичай для кожного проекту, що тестується, створюється окремий проект з тестами. Ім’я проекту з тестами складається з імені проекту, що тестується, крапки та слова “Tests”. Аналогічно також іменуються всі класи у даному проекті, тобто для кожного класу проекту, що тестується, створюється клас у тестовому проекті, проте ім’я класу з тестами вже не містить крапку. В кожному класі з тестами створюються методи, які власне будуть виконувати тести. Кожен метод виконує один тест і перевіряє лише одну функціональність. Назва кожного тестового методу складається із назви методу, що тестується, стану, який тестується, а також очікуваного результату. Всі ці складові в назві тестового методу розділяються нижнім підкреслюванням. Таблиця  1 демонструє описану вище техніку іменування. Даний підхід не є обов’язковим, але є загальноприйнятим серед розробників.

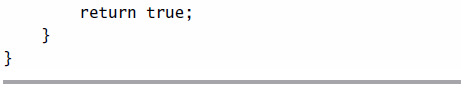
Таблиця  1. Правила іменування тестів



Для того, щоб вказати, що даний проект є тестовим, до нього в Visual Studio додається посилання (Add Reference) на платформу тестування NUnit (NUnit.Framework.dll) та збірку (Assembly) проекту, що тестується. Всі класи, що містять тести, помічаються атрибутом [TestFixture], а методи, що є модульними тестами, – атрибутом [Test]. Далі збірка (Assembly) проекту з тестами відкривається утилітою NUnit через меню File → Open Ptoject…

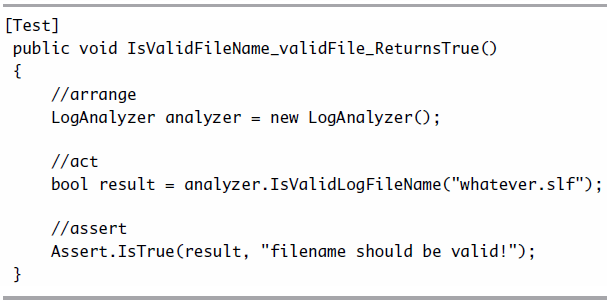
Нехай є деякий простий проект, написаний мовою C#, що виконує аналіз лог-файлів. Складовою частиною такого аналізатору є перевірка (валідація) коректності назви файлу, що аналізується (Лістинг  1).





Лістинг  1. Аналізатор лог-файлів

Напишемо простий модульний тест, для даного коду (Лістинг  2) із використанням утиліти NUnit.

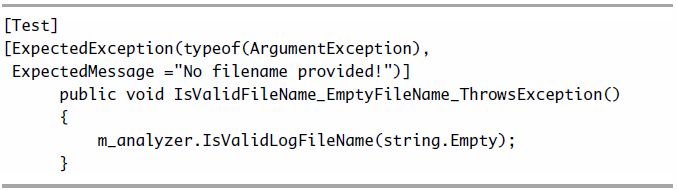


Лістинг  2. Перший модульний тест

Тест зазвичай складається із трьох стандартних секцій: Arrange, Act, Assert.

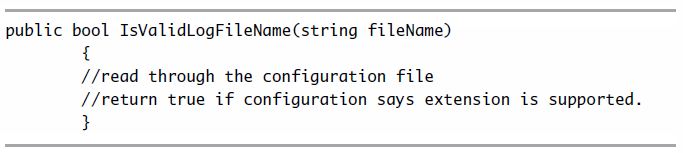
Даний модульний тест передбачає, що при аналізі файлу з коректним іменем файлу метод поверне позитивний результат. У той же час даний модульний тест зазнає невдачі і виявить першу помилку в нашому коді, де аналізатор порівнює розширення файлу, не приводячи його до верхнього регістру.

Модифікуємо сценарій поведінки нашого аналізатору таким чином, що він має генерувати виключну ситуацію у випадку, коли йому передано порожній рядок. Лістинг  3 демонструє модульний тест, який має перевіряти даний функціонал.



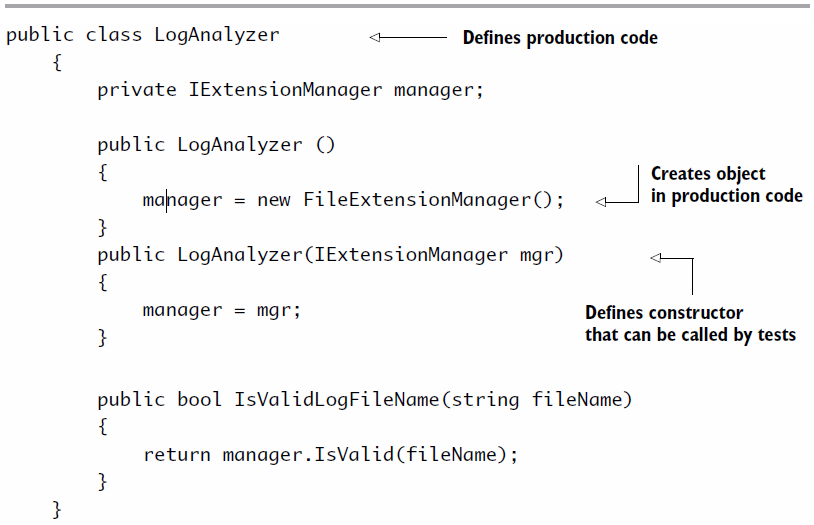
Лістинг  3. Тестування генерації виключної ситуації

Припустимо, що наш модульний тест звертається до файлової системи, щоб визначитися з валідацією файлу (Лістинг  4).



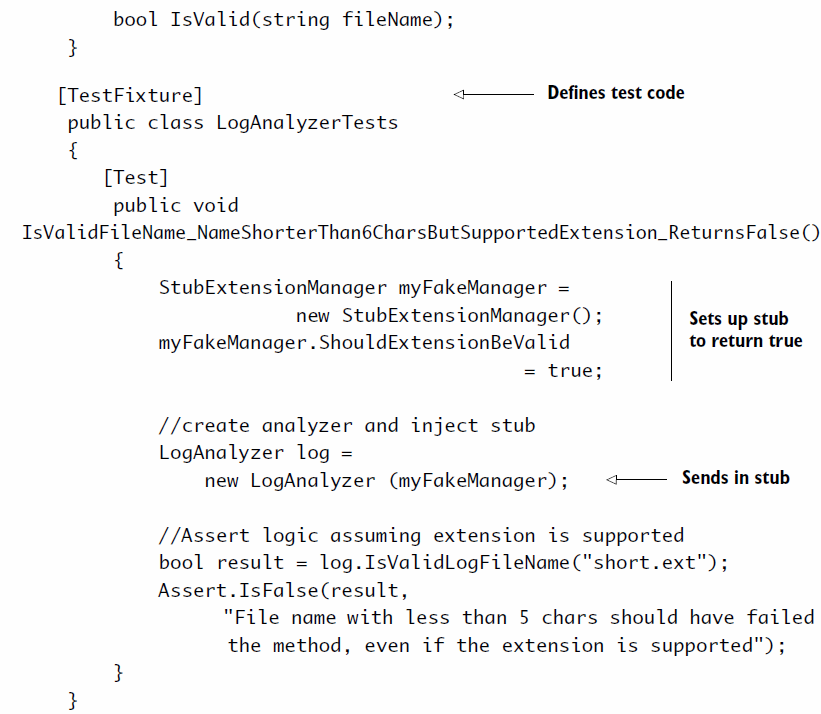
Лістинг  4. Зовнішня залежність від файлової системи

В такому випадку ми вже маємо зовнішню залежність, яку необхідно усунути для можливості написання модульного тесту. Виконаємо рефакторинг нашого аналізатора, ввівши додатковий прошарок у вигляді інтерфейсу в код аналізатора (Лістинг  5). Ін’єкцію залежності будемо проводити через конструктор. При цьому замість реального класу, що працює з файлом, з метою тестування в подальшому ми зможемо вводити заглушку (stub).



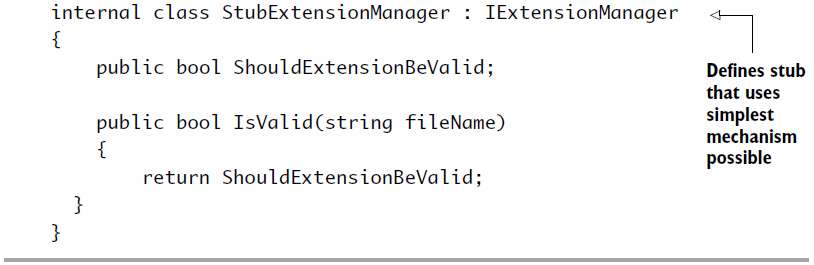
Лістинг  5. Ін’єкція заглушки через конструктор. Клас, що тестується

Лістинг  6 демонструє інтерфейс та тест, в якому використовується заглушка (stub) як клас, що наслідується від даного інтерфейсу. Тест в цьому випадку тестує саме логіку програми і не залежить від файлової системи.

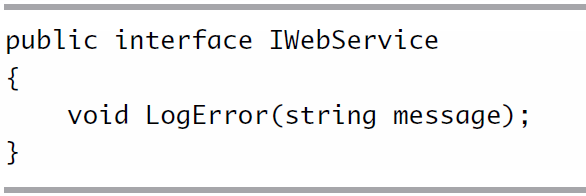
Лістинг  6. Ін’єкція заглушки через конструктор. Модульний тест

Лістинг  7 демонструє наслідування заглушки від того ж інтерфейсу, що й клас файловий менеджер, який підміняється. Слід зауважити при цьому, що заглушка ніколи не приводить тест в червоний стан, а також те, що тест направлений саме проти коду, що тестується, а не проти заглушки.



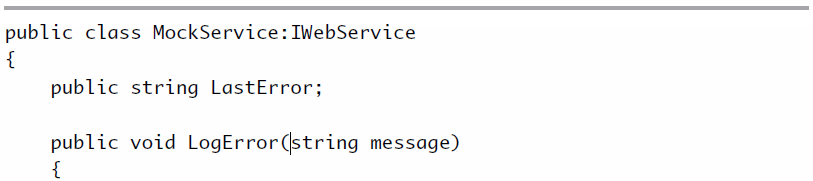
Лістинг  7. Реалізація заглушки

Ускладнимо тепер сценарій поведінки нашого проекту таким чином, щоб у випадку, коли переданий на аналіз файл не є валідним, повідомлення про цю помилку надсилалося до деякого веб-сервісу. У такому випадку нам потрібно тестувати вже взаємодію нашого проекту з веб-сервісом, а саме те, чи коректно надсилається повідомлення до веб-сервісу у разі задання невалідного імені файлу. Лістинг  8 показує, що наш веб-сервіс буде мати один метод, що відправляє повідомлення про помилку.



Лістинг  8. Інтерфейс веб-сервісу

У даному випадку ми вже будемо тестувати взаємодію, тому використаємо макет (mock), а не заглушку (stub) для даного тесту (Лістинг  9). Наш макет буде записувати всю інформацію про взаємодію з ним під час тесту, а потім дана інформація буде верифікуватися на валідність.





Лістинг  9. Макет (mock) веб-сервісу

Власне верифікація буде проводитись у тесті (Лістинг  10) шляхом порівняння даних про взаємодію, що були записані нашим макетом, із даними, які ми очікуємо. Якщо накопичена інформація співпадає з очікуваною, то тест має проходити.



Лістинг  10. Тест із використанням макету (mock)

Покажемо далі, яким чином можна створювати тести, в яких використовуються одночасно і заглушки (mock), і макети (mock). Для цього знову модифікуємо сценарій поведінки нашого проекту. Нехай у випадку, коли ми намагаємось надіслати повідомлення про помилки до веб-сервісу і отримуємо при цьому виключну ситуацію (наприклад, коли веб-сервіс недоступний), нам потрібну повідомити про це адміністратора електронною поштою, тобто надіслати Email. У такому випадку власне веб-сервіс буде вже виступати не макетом, а заглушкою, оскільки ми будемо штучно підробляти його поведінку викиданням виключної ситуації, щоб протестувати, як при цьому буде виконуватись взаємодія із класом, що доставляє листа. Тепер макетом у нас буде підмінятися клас, що надсилає листа, оскільки тестується взаємодія з ним (Рис.  5).

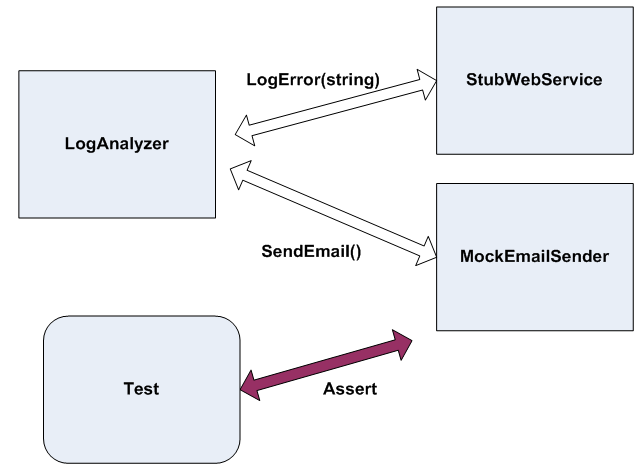
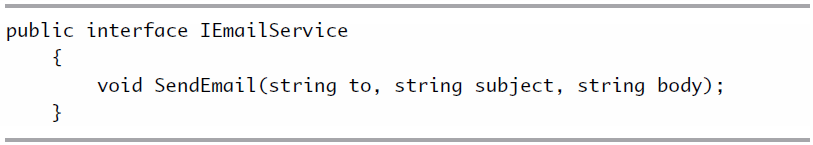
****

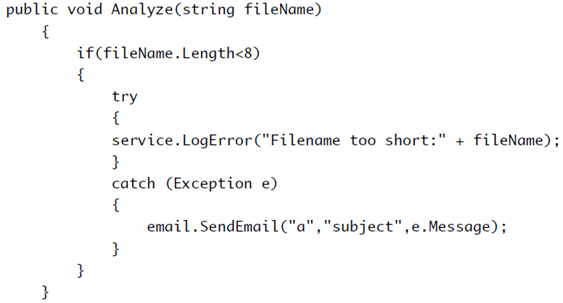
Рис.  5. Схема використання заглушки (stub) і макету (mock) одночасно

Інтерфейс класу, що буде надсилати листа (Лістинг  11), складається з одного методу відправки листа.

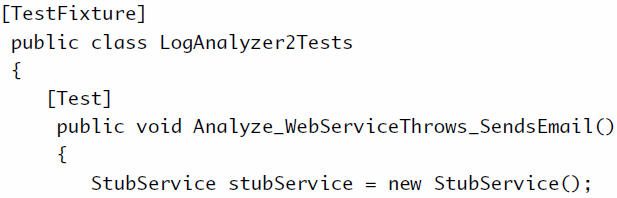


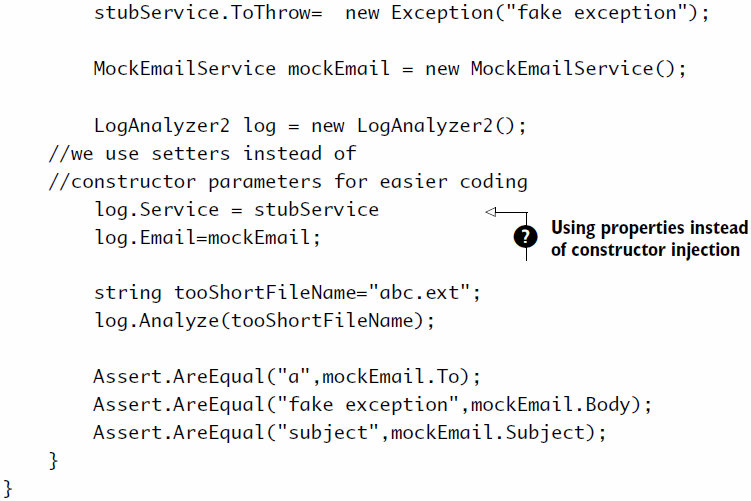
Лістинг  11. Інтерфейс сервісу відправки листа

Далі продемонструємо ускладнений код методу Analyze нашого проекту (Лістинг  12), код тесту, що перевіряє взаємодію із сервісом надсилання листа (Лістинг  13), реалізацію інтерфейсу нашої заглушки (stub) для веб-сервісу (Лістинг  14) та реалізацію інтерфейсу нашого макету (mock) для сервісу відправки листа (Лістинг  15).

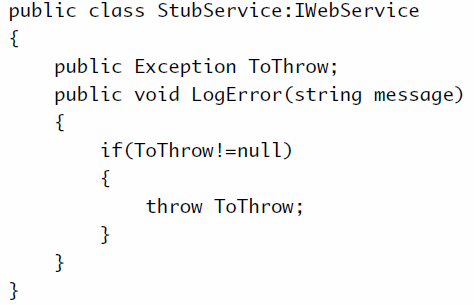


Лістинг  12. Метод, що тестується. Надсилає Email при несправності веб-сервісу

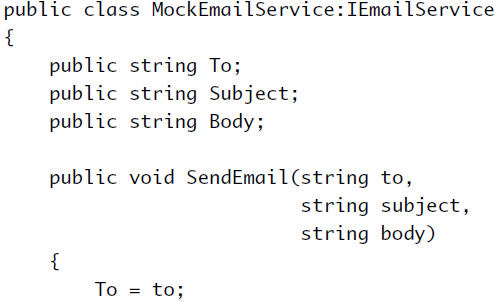


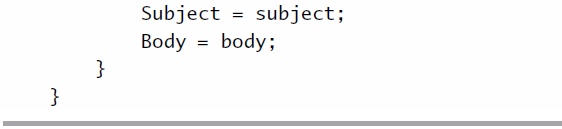


Лістинг  13. Код тесту, що перевіряє взаємодію із сервісом відправки листа



Лістинг  14. Реалізація заглушки (stub), що підміняє веб-сервіс



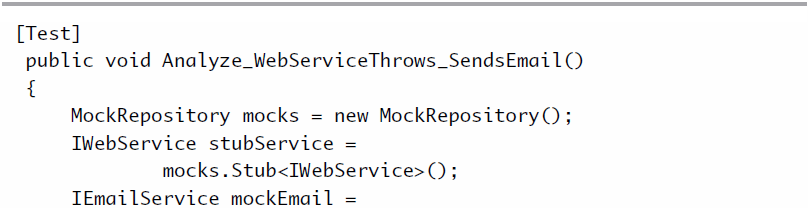


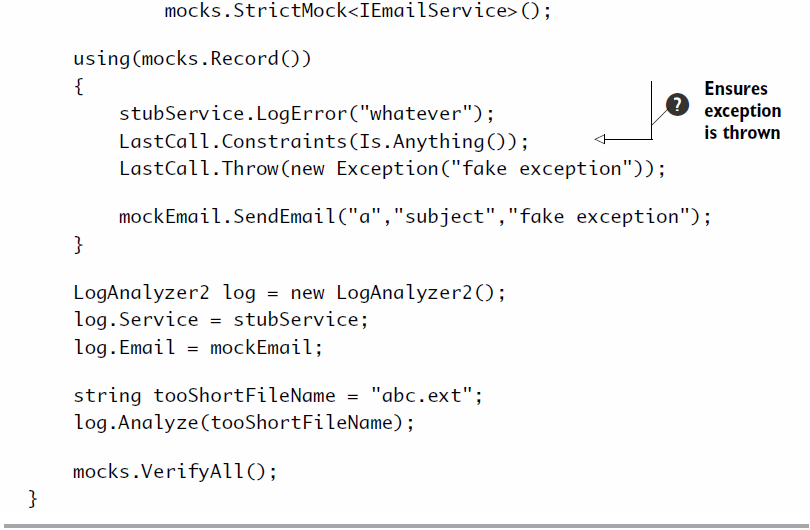
Лістинг  15. Реалізація макету (mock), що підміняє сервіс відправки листа

При написанні модульних тестів слід приймати до уваги одне просте правило: в кожному тесті має бути не більше одного макету (mock). Якщо у вас з якоїсь причини виникає необхідність зробити в одному тесті більше, ніж один макет, то необхідно залишите лише один, а всі інші робити заглушками (stubs). В противному випадку наш тест буде перевіряти вже більше, ніж одну функціональність, а в такому випадку це вже буде не модульний тест за визначенням.

В попередніх прикладах ми створювали фіктивні об’єкти вручну: ми об’являли інтерфейс для екстракції прошарку, а потім вручну писали фіктивну реалізацію цього інтерфейсу. Такий підхід є досить примітивним, оскільки створює багато труднощів у випадках, коли ми маємо більш складну функціональність, що тестується. Нам необхідно писати в декілька разів більше громіздкого коду, реалізовувати всі інтерфейси, їх методи, запам’ятовувати безліч параметрів. Уникнути всього цього можна, використовуючи бібліотеки для модульного тестування (Mock Isolation Frameworks). Ключовою їх відмінністю від раніше розглянутого підходу є те, що в них інтерфейси реалізуються не на етапі компіляції, а динамічно під час виконання (in runtime). Крім того, вони надають дуже багато різного роду інструментів у коді, використання яких робить код більш наглядним та придатним для підтримки.

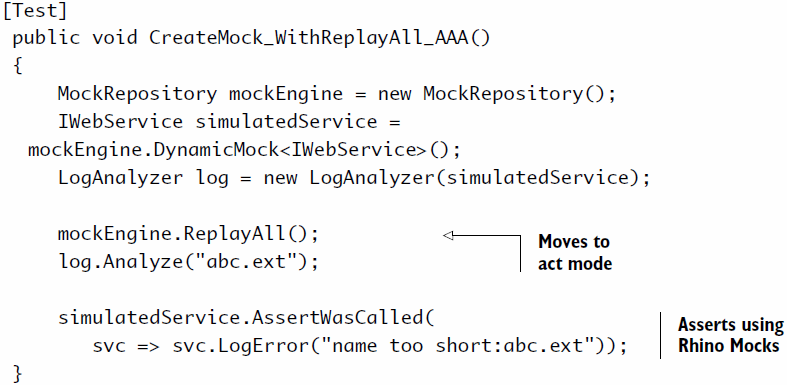
Лістинг  16 демонструє тест із тією ж самою логікою, але написаний із використанням вільно розповсюджуваної бібліотеки Rhino Mocks у якості Mock Isolation Framework. В даному тесті використовується техніка Record and Replay написання тесту. Дана техніка полягає в тому, що у блоці using(mocks.Record()) записується очікувана поведінка взаємодії, яку ми тестуємо. Далі ми виконуємо певні дії і викликаємо метод верифікації, що перевіряє, чи відповідає очікувана поведінка нашої взаємодії реальній поведінці. У даному тесті використано так званий строгий макет (strict mock). При цьому тест падає в тому випадку, якщо не був викликаний очікуваний метод, що вказаний у блоці using(mocks.Record()), або коли був викликаний інший, неочікуваний у даному блоці метод. Щодо детальнішого використання цього та інших видів макетів ознайомтесь із технічною документацією відповідної бібліотеки. Більшість бібліотек для модульного тестування дозволяє також перевіряти, чи була викинута виключна ситуація, чи підписався клас на деяку подію, чи була викликана деяка подія (event triggering testing), вводити обмеження і обробляти параметри методів та багато іншого, що виноситься студентами на самостійне вивчення.





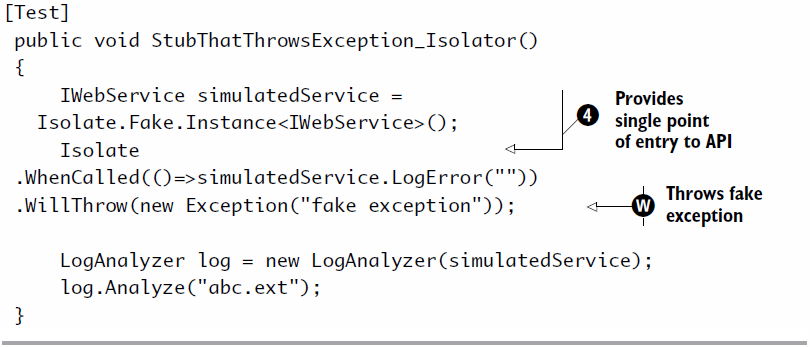
Лістинг  16. Код тесту із використанням Rhino Mocks.

Лістинг  17 демонструє код тесту, написаний із використанням тієї ж бібліотеки Rhino Mocks у стилі Arrange Act Assert. Зазначимо, що останній рядок коду виконує перевірку, чи були викликані очікувані методи, із використанням лямбда-виразу у якості параметру методу. Це дає дуже гнучкі можливості тестування різноманітних сценаріїв поведінки нашого проекту.



Лістинг  17. Тест у стилі Arrange Act Assert синтаксису.

Лістинг  18 демонструє код, що тестує ту ж логіку, але вже із застосуванням бібліотеки Typemock Isolator у якості Mock Isolation Framework.



Лістинг  18. Тест із використанням бібліотеки Typemock Isolator

Typemock Isolator, на відміну від Rhino Mocks, є платною бібліотекою, проте багато розробників вважають її більш зручною, а код, написаний із її використанням більш наглядним.

Рис.  6 демонструє статистику використання деяких найбільш відомих бібліотек Mock Isolation Frameworks, зібрану на локальних Інтернет-суспіствах .NET-розробників. Серед них є як проекти з відкритим вихідним кодом, так і комерційні проекти, які мають пробний період безкоштовного використання. З більшістю із них студентам пропонується ознайомитись самостійно, для кожного з них в мережі Інтернет присутня технічна документація із прикладами застосування.

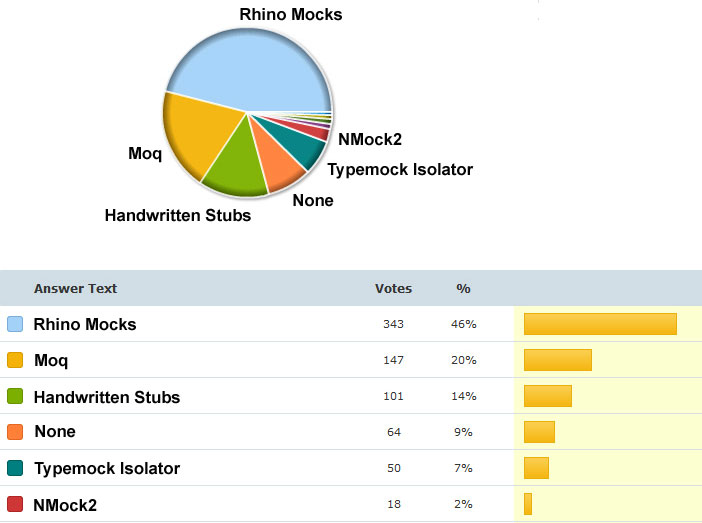


Рис.  6. Isolation Frameworks для .NET

Слід зазначати також, що, не дивлячись на переваги платформ Mock Isolation Framework, їх використання в окремих тестах не завжди є доцільним. Якщо розробнику приходиться писати значно більше коду для тесту і він при цьому втрачає наглядності, то, напевне, такий тест варто краще написати без використання платформ Mock Isolation Framework, оскільки, за визначенням, модульний тест та його зміст повинен мати змогу швидко зрозуміти будь-хто, хто читає ваш код.

Перерахуємо основні переваги Mock Isolation Framework.

* Простіша верифікація параметрів.
* Простіша верифікація декількох викликів методів.
* Простіше створення фіктивних об’єктів.

# Завдання на лабораторну роботу

Реалізувати проект із таблиці 3, заданий за варіантом у таблиці 2, мовою програмування C# .NET. Код проекту має бути максимально покритий модульними тестами. Модульні тести мають бути написані не після реалізації проекту, а в процесі його написання, причому кожен модульний тест має писатися безпосередньо перед тим, як буде реалізована відповідна функція (feature) проекту. Зовнішніми залежностями (external dependency) у даному випадку є файлова система, хост в мережі, сайт або інтерфейс користувача (консоль), які замінюються відповідними фіктивними об'єктами. Всі заглушки (stubs) та макети (mocks) мають створюватися динамічно із використанням Mock Isolation Framework відповідно до номера варіанту за таблицею 2. проект не має містити жодних фіктивних об'єктів (fake), що реалізуються вручну. В кожному проекті має бути продемонстровано два стилі ізоляції: Record and Replay та AAA (Arrange-Act-Assert). Для ін'єкції залежностей продемонструвати ін'єкцію через конструктор, властивість (property), абстрактну фабрику (abstract factory pattern) або будь-який IoC контейнер (Isolation of Control Container). Зробити висновки.

**Примітка**: номер варіанту обирається як остання цифра залікової книжки по модулю дев’ять.

Таблиця  2. Таблиця варіантів

|  |  |  |
| --- | --- | --- |
| **№ варіанту** | **№ проекту** | **Mock Isolation Framework** |
| 0 | I | Rhino Mocks |
| 1 | I | Typemock Isolator |
| 2 | I | Moq |
| 3 | II | Rhino Mocks |
| 4 | II | Typemock Isolator |
| 5 | II | Moq |
| 6 | III | Rhino Mocks |
| 7 | III | Typemock Isolator |
| 8 | III | Moq |

Таблиця  3. Проекти для реалізації

|  |  |
| --- | --- |
| **№** | **Опис проекту** |
| I | Написати простий мережевий командний інтерпретатор, що виконує операції над файлами:   1. команда створення порожнього файлу виду:   cat <file\_name>;   1. команда перейменування (переміщення) файлу виду:   mv <source\_file\_name> | <destination\_file\_name>;   1. команда видалення файлу виду:   rm <file\_name>.  Програма складається із сервера та клієнта. Клієнт являє собою консольний додаток, що працює таким чином:   1. при запуску вводиться IP-адреса та порт серверу, наприклад, 192.168.0.1:8080; 2. після введення адреси сервера та натискання клавіші вводу йде встановлення зв'язку із сервером; 3. виводиться відповідне повідомлення про статус з'єднання; 4. інтерпретатор готовий до вводу команд. Вводиться команда роботи з файлом; 5. виводиться один із статусів виконання команди:    1. done    2. server internal error: <error description>    3. connection error: <error description>   Сервер являє собою консольний додаток, що прослуховує заданий порт. Кількість клієнтів, що можуть під’єднуватись до серверу, необмежена. При надходженні команди над файловою системою сервер намагається її виконати і надсилає клієнту статус результату виконання. |
| II | Написати консольну утиліту, яка виконує обробку матриці. Консольна утиліта приймає такі параметри при запуску (string[] args):   1. ім'я вхідного текстового файлу, який містить матрицю; 2. ім'я вихідного текстового файлу, в який буде записано результат; 3. операція над матрицею:    1. "-d" – визначник матриці;    2. "-t" – транспонування матриці;    3. "-r" – обчислення оберненої матриці.   Якщо третій параметр відсутній, то виконується операція за замовчуванням – вивід (дублювання) вхідної матриці.  У випадку, якщо виникли помилки при відкритті вхідного файлу, у вхідному файлі матриця задана в некоректному форматі, виникла помилка при виконанні операції над матрицею або задані некоректні параметри додатку, програма має записати відповідне повідомлення про помилку замість результату у вихідний файл. У випадку, коли неможливо створити або здійснити запис у вихідний файл, вивести відповідне повідомлення у потік виводу консолі.  Формат файлу матриці такий:   * кожен рядок матриці розділений символом переносу рядка ("\n") або крапкою з комою (";", semicolon); * елементи в межах одного рядка розділені пробілами, табуляціями ("\t") або комами (","). |
| III | Написати калькулятор під Google. Програма являє собою консольний додаток, в який вводиться математичний вираз. Після натиснення клавіші вводу додаток надсилає в пошукову систему Google HTTP-запит, де пошуковою фразою є введений вираз. Після отримання відповіді від пошукової системи Google аналізується отримана HTML-сторінка з метою пошуку в ній відповіді на запит (обчислення математичного виразу) за допомогою регулярного виразу або іншим чином. У випадку, якщо відповідь на запит не надійшла, не вдалося встановити з'єднання або Google не зміг обробити належним чином запит, в консоль виводиться відповідне повідомлення про помилку. |

# 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)? Дайте приклади.
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.