Міністерство освіти і науки України
Сумський державний університет

4560 Методичні вказівки
до практичних занять із дисципліни
«**Реінжиніринг і верифікація програмного забезпечення**»
на тему «Модульне тестування з використанням
бібліотеки JUnit»
для студентів спеціальності 122 *«Комп'ютерні науки»*
всіх форм навчання

Суми
Сумський державний університет
2019

Методичні вказівки до практичних занять із дисципліни «Реінжиніринг і верифікація програмного забезпечення»
на тему «Модульне тестування з використанням бібліотеки JUnit» / укладач В. В. Москаленко. – Суми : Сумський державний університет, 2019. – 28 с.

Кафедра комп’ютерних наук

ЗМІСТ

ВСТУП

Реінжиніринг часто неможливий без виконання масштабних рефакторингів, де під рефакторингом (англ. refactoring) розуміють послідовність невеликих еквівалентних (тобто таких, що зберігають поведінку) перетворень. Більш формалізовано рефакторинг можна визначити як процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки, з метою полегшення розуміння її роботи і функціонального масштабування.

Виконання процесів рефакторингу та реінжинірингу без допущення жодних помилок – справа надзвичайно непроста. Тому на практиці використовують методи валідації й верифікації. Вони необхідні для перевірки програмного забезпечення чи його частин на коректність реалізації поставленого завдання шляхом порівняння з властивостями, що вимагаються. Але варто відзначити, що в процесі верифікації перевіряється правильність роботи функціонала програмного продукту. А в процесі валідації перевіряється відповідність функціонала програмного забезпечення поведінці, яку очікував чи мав на увазі користувач або замовник. Тому інженери-розробники програмного забезпечення більш задіяні в задачах верифікації.

Основною складовою частиною верифікації є тестування, тобто процес виконання програми з метою виявлення помилки. При цьому найбільш поширеним видом тестування систем малого та середнього розмірів є модульне тестування, виконуване для кожного незалежного програмного модуля.

Метою практичної роботи є ознайомлення з основами рефакторингу та модульного тестування як процесів, що передують і супроводжують реінжиніринг програмного забезпечення. У процесі виконання практичної роботи студент повинен набути навичок використання інструментів рефакторингу і розробленням модульних тестів.

1 КОРОТКІ ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1 Мета заняття

Мета заняття полягає в набутті практичних навичок складання модульних тестів, використання бібліотеки JUnit та інструментів рефакторингу в середовищі розроблення NetBeans.

1.2 Поняття рефакторингу

Рефакторинг – це насамперед перетворення брудного коду на чистий. Чистий код характеризується більшою очевидністю для інших програмістів, ніж брудний код. Також чистий код не містить дублювання. Чистий код завжди містить мінімум класів та інших рухомих частин.

Рефакторинг покращує нефункціональні атрибути програмного забезпечення. У будь-якому великому програмному проекті настає час, коли потрібно рефакторити ваш код. Часто корисно рефакторити ваш код багато разів під час розроблення проекту. Під час вашого прогресу ви думаєте про кращі способи структурування вашого коду, а також, можливо – кращі алгоритми або структури даних, використовуваних для підтримання функціональності та швидкості. У цій роботі ми зосередимося на техніках рефакторингу.

Наявність тестового коду полегшує рефакторинг, оскільки ліквідує страх щось зламати, тому тести є ознакою добре продуманого проекту. Якщо ви, як інженер-програміст, боїтеся переробити свій код, це, мабуть, тому, що не написано достатньо тестів, щоб відчувати себе комфортно, змінюючи код вашого проекту.

1.3 Модульне тестування

З огляду на часові та бюджетні міркування неможливо провести вичерпне тестування для кожного набору тестових сценаріїв, особливо якщо їх велика кількість. Нам потрібний простий спосіб або спеціальні методики, що можуть вибирати тестові сценарії інтелектуально з пулу test-case таким чином,

щоб були охоплені всі тестові сценарії. Для цього використовують дві методики: тестування граничного значення і тестування класів еквівалентності.

1.4 Тестування граничного значення

Досвід показує, що тестові сценарії, що вивчають граничні умови, мають переваги порівняно з тестами, які цього не передбачають.

Помилки часто виникають на межі умовних виразів (conditional statements) і структур даних. При тестуванні з аналізом граничних значень тести призначені для тестування на границях вхідних і вихідних областей значень. Розглянемо основні правила розроблення граничних тестів.

Якщо вхід може бути значенням між a і b, ми повинні тестувати значення a і b, а також значення, що знаходяться безпосередньо між a і b.

Якщо модуль програми може мати декілька вхідних змінних (між максимумом та мінімумом), ми повинні тестувати його з максимальною й мінімальною кількістю змінних, а також з кількістю, що знаходяться безпосередньо між мінімумом та максимумом відповідно. Це правило необхідно застосовувати і до вихідних значень.

Для структур даних конкретних розмірів ми повинні тестувати структуру на найменших і найбільших граничних значеннях.

Тестування граничного значення є методом перевірки здорового глузду – програмісти інтуїтивно схильні виконувати цей тип тестування до певної міри під час написання програм. Розглянемо перший приклад тестування граничного значення. Припустимо, у вас дуже важливий інструмент в офісі, він приймає дійсне поле «Ім'я користувача» та «пароль» для роботи з цим інструментом і приймає мінімум 8 символів і максимум 12 символів. Тоді правильний (valid) діапазон дорівнює [8-12], а неправильний (invalid) діапазон дорівнює (-∞,7] та [13, +∞) (рис. 1.1).

Рисунок 1.1 – Ілюстрація діапазонів довжини пароля для тестування граничних значень

Можна записати тестові випадки для значень із Valid partition, Invalid partition і точних (Exact) граничних значень.

– тестовий випадок 1 – розглянемо довжину пароля менше ніж 8;

– тестовий випадок 2 – розглянемо пароль довжиною, що дорівнює 8;

– тестовий випадок 3 – розглянемо пароль довжиною від 9 до 11;

– тестовий випадок 4 – розглянемо пароль довжиною, що дорівнює 12;

– тестовий випадок 5 – розглянемо пароль довжиною, більшою за 12.

Розглянемо другий приклад тестування граничних випадків. Тестові випадки для програми, де область значення вхідного аргументу є набором цілих чисел від 1 до 1 000. Тоді, Valid partition відповідає діапазону [1–1 000], Invalid partition відповідає 0 та діапазону [1 001– +∞) (рис. 1.2).

Рисунок 1.2 – Ілюстрація діапазонів тестування граничного значення

Можна записати тестові випадки для значень із Valid partition, Invalid partition і точних граничних значень :

– тестовий випадок 1 – розглянемо дані тесту точно такі, як вхідні межі вхідного діапазону, тобто значення 1 та 1 000;

– тестовий випадок 2 – розглянемо дані тесту зі значеннями, що знаходяться всередині вхідних діапазонів безпосередньо біля їх межі, тобто значення 0 та 999;

– тестовий випадок 3 – розглянемо дані тесту зі значеннями, що знаходяться за крайніми межами вхідного домену безпосередньо біля їх межі, тобто значення 2 та 1 001.

1.5 Тестування класу еквівалентності

Клас еквівалентності є таким набором вхідних значень, де правильне (неправильне) оброблення будь-якого значення означає правильне (неправильне) оброблення всіх інших значень також.

Тест класів еквівалентності є методом тестування «чорної шухляди». Ми використовуємо специфікацію програми для визначення класів входів. Ідея полягає в тому, що ми групуємо входи в набори правильних (Valid) і неправильних (Invalid) вхідних значень. Тоді ми можемо вибрати одне значення з кожного класу для тестування програми. Це мінімізує кількість тестів, які нам потрібно виконати, але водночас забезпечує високі шанси знайти помилки.

Ми можемо використовувати такі інструкції для вибору класів еквівалентності:

– якщо вхід належить діапазону значень, то ми задаємо один Valid (min < x < max) та два Invalid (x < min та max > x) класи еквівалентності;

– якщо програма набуває певної кількості значень як вхідні дані (наприклад, між 1 і 3 значеннями), ми повинні мати один Valid і два Invalid (немає даних, більше ніж 3 вхідних даних) класи еквівалентності;

– якщо вхід може бути множиною значень, кожне з яких може розглядатися по-різному в програмі, то повинен бути один

valid клас еквівалентності для кожного члена множини та один invalid клас еквівалентності – елемент, що не входить до набору;

– якщо вхід повинен бути певним значенням, ми повинні мати один valid клас еквівалентності, де значення правильне, і один Invalid клас еквівалентності, де значення не є правильним;

– якщо програма не обробляє всіх елементів класу еквівалентності однаковим способом, то клас еквівалентності повинен бути розділений на менші класи еквівалентності.

Давайте розглянемо приклад тестування класу еквівалентності:

– текстове поле допускає лише цифрові символи;

– довжина повинна бути 6-10 символів. Розбиття відповідно до вимог повинне бути таким, як показано на рисунку 1.3.

Рисунок 1.3 – Приклад тестування класу еквівалентності

Під час оцінювання еквівалентного розбиття значення в усіх ділянках є еквівалентними, тому 0–5 еквівалентні, 6–10 еквівалентні та 11–14 еквівалентні.

Під час тестування тести значень 4 та 12 вважаються Invalid, а 7 – Valid. Легко перевірити вхідні діапазони 6–10, але важче тестувати діапазони вхідних даних [2–600]. Тестування буде легким у разі меншої кількості тестових випадків, але ви повинні бути дуже обережними. Припускаючи вхідне значення, дорівнює 7, як Valid, ви припускаєте, що розробник правильно кодував діючий діапазон (6–10).

2 ЗАВДАННЯ ДО ПРАКТИЧНОЇ РОБОТИ

2.1 Попередня підготовка

Тестування програми є невід'ємною частиною циклу розробки, а написання та підтримання одиничних тестів може допомогти забезпечити правильність роботи окремих методів у вихідному коді. Інтегрована підтримка IDE для модуля тестування JUnit дозволяє вам швидко й легко створювати тести JUnit та тестові набори.

Створіть проект Java class library із назвою JUnit-Sample і пакет з назвою sample:

1. Choose File -> New Project з головного меню.
2. Оберіть Java Class Library з Java категорії і натисніть Next.
3. Наберіть JUnit-Sample для цього проекту і задайте розміщення проекту.
4. Зніміть виділення з опції Use Dedicated Folder, якщо вона обрана.
5. Натисніть Finish.
6. У вікні проектів виконайте right-click на вузлі Source Packages у проекті JUnit-Sample та оберіть New -> Java Package з випадного меню.
7. Наберіть sample як ім'я проекту. Натисніть Finish.

Після створення проекту ви формуєте два файли Utils.java та Vectors.java з коду, що дається в додатку, і додаєте їх до вашого проекту JUnit-Sample, наприклад, використовуючи Drag & Drop. Якщо ви подивитеся на вихідний код класів, ви побачите, що у Utils.java є три методи (computeFactorial, concatWords і normalizeWord), і що Vectors.java має два методи – equal та scalarMultiplication. Наступним кроком є створення класів тестів для кожного класу і написання тестів для методів.

2.2 Написання тестового класу для Vectors.java

2.2.1 Правий клік на Vectors.java і вибір Tools -> Create Tests.

2.2.2 Модифікуйте ім'я тестового класу на VectorsJUnit4Test у діалозі Create Tests.

Примітка. Якщо ви зміните назву тестового класу, ви побачите попередження про зміну імені. Назва за замовчуванням базується на назві класу, який ви тестуєте, з тегом, що додано до імені. Наприклад, для класу MyClass.java, назва за замовчуванням класу тесту – MyClassTest.java. У JUnit версії 4 тестування не обов'язково закінчується словом Test. Зазвичай найкраще зберігати ім'я за замовчуванням, але оскільки ви створюєте всі тести JUnit в одному пакеті, імена класів тесту повинні бути унікальними.

2.2.3 Виберіть JUnit у випадному списку Framework.

2.2.4 Зніміть вибір Test Initializer та Test Finalizer. Натисніть OK (рис. 2.1).

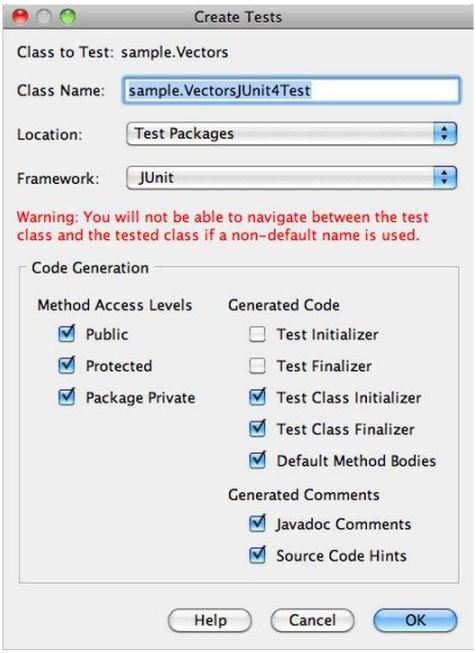

Рисунок 2.1 – Вікно Create Test

2.2.5 Оберіть JUnit 4.x у Select JUnit Version боксі. Натисніть Select (рис. 2.2).

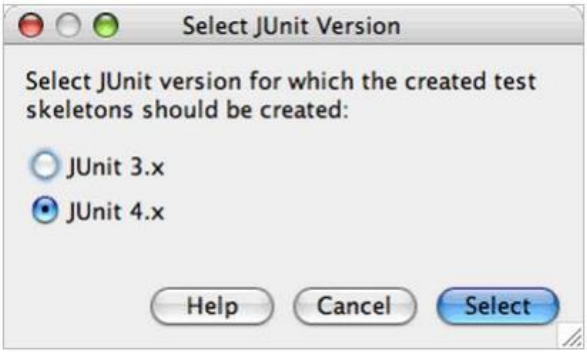

Рисунок 2.2 – Вікно Select JUnit Version

2.2.6 Якщо ви натисните OK, то IDE створить VectorsJUnit4Test.java тестовий клас у пакеті sample у вузлі Test Packages вікна Projects (рис. 2.3).

Рисунок 2.3 – Вікно Projects

Примітка. Для створення тестів потрібний каталог для тестових пакетів. Розміщення за замовчуванням для каталогу тестових пакетів знаходиться на кореневому рівні проекту, але ви можете зазначити інше місце для каталогу в діалоговому вікні «project's Properties dialog».

У VectorsJUnit4Test.java кожний тестовий метод анотований фікстурою @test. IDE згенерувала імена для методів тестування на основі назв методів у Vectors.java, але назва тестового методу не вимагає префікса test. Тіло за замовчуванням кожного згенерованого методу тестування надається виключно як поради та потребує модифікації, щоб стати дійсними тестами. Ви можете скасувати Default Method Bodies за замовчуванням у діалоговому вікні «Create Tests». IDE також створила такі методи ініціалізатора та фіналізатора :

```
@BeforeClass
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}
```

За замовчуванням IDE генерує методи ініціалізації і фіналізації тестового класу JUnit 4 під час їх створення. Анотації @BeforeClass і @AfterClass використовуються для позначення методів, які необхідно запустити перед і після запуску тестового класу. Ви можете видалити методи, оскільки вам не потрібно буде тестувати Vectors.java. Ви можете налаштувати методи, які створюються за замовчуванням, налаштувавши параметри JUnit у вікні параметрів. Зверніть увагу, що в тестах JUnit 4 IDE за замовчуванням додає статичну декларацію імпорту для org.junit.Assert.*.

2.3 Написання тестових методів для Vectors.java

2.3.1 Відкрийте VectorsJUnit4Test.java в редакторі.

2.3.2 Модифікуйте тестові методи для testScalarMultiplication шляхом зміни імені методу, значення для println та видалення згенерованих змінних. Тестовий метод повинен мати такий вигляд:

```
@Test
public void ScalarMultiplicationCheck() {
    System.out.println("* VectorsJUnit4Test: ScalarMultiplicationCheck()");
    assertEquals(expResult, result);
}
```

Примітка. Під час написання тестів не обов'язково змінювати вивід. Ви робите це лише для того, щоб було легше ідентифікувати результати тесту у вікні виводу.

2.3.3 Тепер додайте декілька тверджень (assertions) для тестування методу :

```
@Test
public void ScalarMultiplicationCheck() {
    System.out.println("* VectorsJUnit4Test: ScalarMultiplicationCheck()");
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, 0}));
    assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, 6}));
    assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { 5,-6}));
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, 5}));
    assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, 8}));
}
```

У цьому тестовому методі ви використовуєте метод JUnit assertEquals. Щоб використовувати assertation, ви надаєте актуальні вхідні змінні та очікуваний результат. Щоб пройти тест, тестовий метод повинен успішно повернути всі очікувані результати на основі змінних, що надаються під час запуску тестового методу. Ви повинні додати достатню кількість assertation, щоб охопити різні можливі перестановки.

2.3.4 Змініть ім'я тестового методу testEqual на equalsCheck.

2.3.5 Видаліть згенероване тіло тестового методу equalsCheck.

2.3.6 Додайте наступний println до тестового методу equalsCheck:

```
System.out.println("* VectorsJUnit4Test: equalsCheck()");
```

Тестовий метод тепер повинен мати такий вигляд:

```
@Test
public void equalsCheck() {
    System.out.println("* VectorsJUnit4Test: equalsCheck()");
}
```

2.3.7 Змінити метод equalsCheck, додавши такі assertation:

```
@Test
public void equalsCheck() {
    System.out.println("* VectorsJUnit4Test: equalsCheck()");
    assertTrue(Vectors.equal(new int[] {}, new int[] {}));
    assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));
    assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));
    assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));
    assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));

    assertFalse(Vectors.equal(new int[] {}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));
}
```

2.3.8 Цей тест використовує JUnit assertTrue і assertFalse методи для перевірки різних можливих результатів. assertTrue перевіряє правильнісить умови, а assertFalse – помилковість умови.

2.4 Написання тестового класу для Utils.java

2.4.1 Натисність праву кнопку мишки на Utils.java і оберіть Tools -> Create Tests.

2.4.2 Оберіть JUnit у випадному списку Framework, якщо він не вибраний.

2.4.3 Виберіть Test Initializer і Test Finalizer у випадному меню, якщо їх не вибрано.

2.4.4 Змініть назву класу тесту на UtilsJUnit4Test у діалоговому вікні Create Tests. Натисніть «ОК». Після

натискання кнопки «ОК» IDE створює тестовий файл UtilsJUnit4Test.java у Test Packages->sample-каталозі. Ви можете побачити, що IDE згенерувала методи testComputeFactorial, testConcatWords та testNormalizeWord для методів у Utils.java. IDE також генерує методи ініціалізації та фіналізації для тесту й класу тестів.

2.5 Написання тестових методів для Utils.java

Додайте тестові приклади, які ілюструють деякі загальні тестові елементи JUnit. Також додайте метод println, оскільки деякі методи не друкують жодного виводу у вікні результатів тесту JUnit, щоб зазаначити, що вони були запущені або, що метод пройшов тест. Додавши метод println, ви можете побачити, чи були запущені методи та порядок їх виконання.

2.5.1 Якщо ви створили клас тестування для Utils.java, IDE генерує анотовані методи ініціалізації та фіналізації. Ви можете вибрати будь-яке ім’я для назви методу, оскільки немає необхідної конвенції іменування.

Примітка. Для тестування Utils.java вам не потрібні методи ініціалізації та фіналізації, але вони внесені в цю практичну роботу, щоб продемонструвати, як вони працюють. У JUnit 4 ви можете використовувати анотації для позначення наступних методів як ініціалізатор, або як фіналізатор.

Анотація @BeforeClass позначає метод як метод ініціалізації класу тестування. Метод ініціалізації класу тестування запускається лише один раз, а також перед будь-яким іншим методом у тестовому класі. Наприклад, замість створення з’єднання з базою даних в ініціалізаторі тесту та створення нового з’єднання перед кожним методом перевірки ви можете використати ініціалізатор тестового класу, щоб відкрити з'єднання перед виконанням тестів. Після цього ви можете закрити з’єднання фіналізатором класу тестування.

Test Class Finalizer. Анотація @AfterClass позначає метод як метод завершення тестового класу. Метод завершення тестового

класу запускається лише один раз, після завершення всіх інших методів у тестовому класі.

Анотація @Before позначає метод як метод ініціалізації тесту. Метод ініціалізації тесту запускається перед кожним тестом у класі тесту. Метод ініціалізації тесту не потрібний для запуску тестів, але якщо вам потрібно ініціалізувати деякі змінні, перш ніж запускати тест, використовується метод тестового ініціалізатора.

Анотація @After позначає метод як метод фіналізації тесту. Метод фіналізації тесту виконується після кожного тесту в класі тесту. Для тестування не потрібний метод фіналізації тесту, але вам може знадобитися остаточний інструмент для очищення будь-яких даних, необхідних під час перевірки тестових випадків. Зробіть такі зміни (виділено жирним шрифтом) у UtilsJUnit4Test.java:

```
@BeforeClass
public static void setUpClass() throws Exception {
    System.out.println("* UtilsJUnit4Test: @BeforeClass method");
}

@AfterClass
public static void tearDownClass() throws Exception {
    System.out.println("* UtilsJUnit4Test: @AfterClass method");
}

@Before
public void setUp() {
    System.out.println("* UtilsJUnit4Test: @Before method");
}

@After
public void tearDown() {
    System.out.println("* UtilsJUnit4Test: @After method");
}
```

Під час запуску класу тесту текст, який ви додали, відображається на панелі виводу вікна результатів тесту JUnit.

Якщо ви не додали println, то немає виводу, який свідчить про те, що запущені методи ініціалізації та оброблення.

2.5.2 Тестування з використанням Simple Assertion

Розглянемо простий тестовий приклад для перевірки методу concatWords. Замість того, щоб створити тестовий метод testConcatWords, ви можете додати новий метод тесту з назвою helloWorldCheck, який використовує єдиний simple Assertion для перевірки правильності об'єднує рядків. Метод assertEquals у тестовому прикладі використовує синтакс assertEquals(EXPECTED_RESULT, ACTUAL_RESULT), щоб перевірити, чи очікуваний результат такий самий, як фактичний результат. У цьому разі, якщо вхідні дані методу concatWords є "Hello", ",", "world" і "!", то очікуваний результат буде таким "Hello, world!".

Видаліть згенерований тест testconcatWords. Додайте наступний метод helloWorldCheck до тесту Utils.concatWords:

```
@Test
public void helloWorldCheck() {
    assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world",
"!"));
}
```

Додайте команду println, щоб відобразити текст про тест у вікні результатів тесту JUnit:

```
@Test
public void helloWorldCheck() {
    System.out.println("* UtilsJUnit4Test: test method 1 - helloWorldCheck()");
    assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));
```

2.5.3 Тест, позначений анотацією Timeout, перевіряє надмірно довге виконання методу. Якщо метод займає занадто довгий час, тестовий потік переривається, і проходження тесту вважається неуспішним. Для цього потрібно зазначити часовий інтервал для виконання тесту. Метод тестування викликає метод calculateFactorial у Utils.java. Ви можете припустити, що метод computeFactorial є правильним, але в цьому разі ви перевіряєте

лише те, чи виконання обчислень вкладається в 1 000 мілісекунд. Це здійснюється шляхом переривання тестового потоку після 1 000 мілісекунд. Якщо потік перервано, то метод тесту генерує виключення TimeoutException.

Видаліть згенерований тестовий метод testComputeFactorial. Додайте метод testWithTimeout, який обчислює факторіал випадково сформованого числа:

```
@Test
public void testWithTimeout() {
    final int factorialOf = 1 + (int) (30000 * Math.random());
    System.out.println("computing " + factorialOf + '!');
    System.out.println(factorialOf + "! = " +
            Utils.computeFactorial(factorialOf));
}
```

Додайте такий код, щоб встановити тайм-аут і перервати потік, якщо метод виконується занадто довго:

```
@Test(timeout=1000)
public void testWithTimeout() {
    final int factorialOf = 1 + (int) (30000 * Math.random());
}
```

Ви можете бачити, що тайм-аут встановлено на 1 000 мілісекунд. Додайте такий файл println, щоб надрукувати текст про тест у вікні результатів тесту JUnit:

```
@Test(timeout=1000)
public void testWithTimeout() {
    System.out.println("* UtilsJUnit4Test: test method 2 - test
            WithTimeout()");
    final int factorialOf = 1 + (int) (30000 * Math.random());
    System.out.println("computing " + factorialOf + '!');
}
```

2.5.4 Тест, позначений у анотації як Expected Exception, перевіряє наявність очікуваного виключення. Метод не працює,

якщо він не викидає зазначене очікуване виключення.

У нашому разі ви перевіряєте, чи метод computeFactorial викидає IllegalArgumentException, якщо вхідна змінна є від'ємним числом (–5). Додайте такий метод testExpectedException, який викликає метод computeFactorial зі входом –5.

```
@Test
public void checkExpectedException() {
    final int factorialOf = -5;
    System.out.println(factorialOf + "! = " +
            Utils.computeFactorial(factorialOf));
}
```

Додайте таку властивість до анотації @Test, задавши очікуване виключення IllegalArgumentException

```
@Test(expected=IllegalArgumentException.class)
public void checkExpectedException() {
    final int factorialOf = -5;
    System.out.println(factorialOf + "! = " +
            Utils.computeFactorial(factorialOf));
}
```

Додайте такий println для виводу тексту про тест у вікні JUnit Test Results:

```
@Test (expected=IllegalArgumentException.class)
public void checkExpectedException() {
    System.out.println("* UtilsJUnit4Test: test method 3 - checkEx
            pectedException()");
    final int factorialOf = -5;
    System.out.println(factorialOf + "! = " +
            Utils.computeFactorial(factorialOf));
}
```

2.5.5 Відключення тесту (Disabling a Test) дозволяє тимчасово вимкнути тестовий метод. У JUnit 4 ви можете додати анотацію @Ignore, щоб вимкнути тест.

Видаліть згенерований метод testNormalizeWord. Додайте такий метод тесту до класу тесту

```
@Test
public void temporarilyDisabledTest() throws Exception {
    System.out.println("* UtilsJUnit4Test: test method 4 - checkEx
            pectedException()");
    assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));
}
```

Тестовий метод temporaryDisabledTest буде запускатися, якщо ви запустите тестовий клас. Додайте @Ignore анотацію над тестом @Test, щоб вимкнути тест

```
@Ignore
@Test
public void temporarilyDisabledTest() throws Exception {
    System.out.println("* UtilsJUnit4Test: test method 4 - checkEx
            pectedException()");
    assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));
}
```

Додайте імпорт класу org.junit.Ignore. Тепер, коли ви написали тести, то можете їх запустити й побачити тестовий вивід у вікні JUnit Test Results.

2.6 Запуск окремих тестових класів

Ви можете запускати тести JUnit для всієї програми або окремих файлів і переглядати результати в IDE. Найпростіший спосіб запустити всі одиничні тести для проекту – Run -> Test <PROJECT_NAME> з головного меню. Якщо ви виберете цей метод, IDE виконує всі тестові класи у тестових пакетах. Щоб запустити індивідуальний тестовий клас, клацніть правою кнопкою миші клас тесту у вузлі Test Packages і виберіть Run File.

2.6.1 Клацніть правою кнопкою миші UtilsJUnit4Test.java у вікні Projects.

2.6.2 Виберіть Test File.

2.6.3 Виберіть Window -> IDE Tools -> Test Results, щоб відкрити вікно Test Results. Під час запуску UtilsJUnit4Test.java IDE запускає лише тести у тестовому класі. Якщо клас проходить усі тести, ви побачите щось схоже на таке зображення у вікні JUnit Test Results (рис. 2.4):

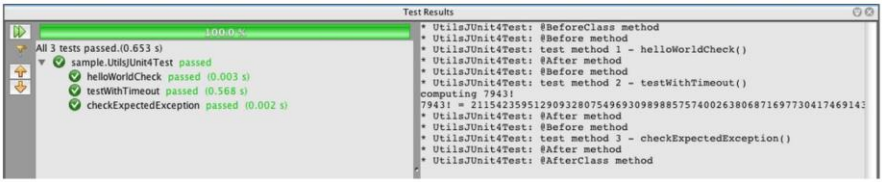

Рисунок 2.4 – Вікно JUnit Test Results

На цьому зображенні ви можете побачити, що IDE запустила тест JUnit для Utils.java і що клас пройшов усі тести. На лівій панелі відображаються результати окремих методів тестування, а на правій панелі відображається тестовий вивід. Якщо ви подивитеся на вивід, то зможете побачити порядок тестування. У полі println, який ви додали до кожного з методів тестування, надруковано ім'я тесту для вікна Test Results і вікна Output.

Ви можете побачити, що в UtilsJUnit4Test метод ініціалізації тестового класу, який анотований із @BeforeClass, запускався до будь-якого іншого методу, і він запускався лише один раз. Метод фіналізації тестового класу, який анотований @AfterClass, був запущений останнім після всіх інших методів у класі. Метод ініціалізації тесту, який анотовано з @Before, був запущений перед кожним тестовим методом.

Елементи керування в лівій частині вікна Test Results дають змогу легко запустити тест знову. Ви можете використовувати фільтр для перемикання між відображенням усіх результатів

тесту або лише невдалими тестами. Стрілки дозволяють перейти до наступної помилки або попередньої відмови.

Якщо ви натиснете правою кнопкою миші на результати тесту у вікні Test Results, то у висхідному меню ви зможете перейти до джерела тесту, запустити тест знову або налагодити тест.

Наступним кроком після створення класів тестування модулів є створення тестових наборів.

2.7 Об'єднання тестів у систему TestSuite

Під час створення тестів для проекту ви зазвичай одержите багато тестових класів. Хоча ви можете запускати тестові класи індивідуально або запускати всі тести в проекті, у багатьох випадках ви захочете запустити підмножину тестів або запустити тести в певному порядку. Ви можете зробити це, створивши один або декілька тестових наборів (test suites). Наприклад, ви можете створювати тестові набори, які перевіряють певні аспекти вашого коду або конкретні умови. Набір тестів в основному являє собою клас із методом, який викликає зазначені тестові випадки, наприклад, конкретні тестові класи, методи тестування в тестових класах та інші тестові набори. Тестовий набір може бути включений як частина тестового класу, але найкращі практики рекомендують створювати окремі тестові набори класів. Коли ви використовуєте IDE для створення тестового набору, за замовчуванням IDE генерує код для виклику всіх тестових класів у тому самому пакеті, що і тестовий набір. Після створення тестового набору ви можете змінити клас, щоб зазначити тести, які ви хочете запустити як частину цього набору.

У пакеті тестування JUnit 4 ви можете зазначити тестові класи, які необхідно включити як значення в анотації @Suite.

2.7.1 Натисніть правою кнопкою миші на вузол проекту в вікні Projects та оберіть New -> Other, щоб відкрити майстер створення нового файла.

2.7.2 Виберіть Test Suite в категорії Unit Tests. Натисніть кнопку Next.

2.7.3 Введіть JUnit4TestSuite для імені файла.

2.7.4 Виберіть пакет sample для створення тестового набору в пакету тестів у каталозі sample.

2.7.5 Зніміть вибір ініціалізатора та фіналізатора тестів. Натисніть «Готово». Коли ви натискаєте «Finish», IDE створює клас тестового набору в пакеті sample і відкриває клас у редакторі. Тестовий набір містить код, подібний до такого.

```
@RunWith(Suite.class)
@Suite.SuiteClasses(value={UtilsJUnit4Test.class, VectorsJUnit4Test.class})
public class JUnit4TestSuite {
}
```

Під час запуску набору тестів IDE запускає класи тесту у тому порядку, в якому вони перелічені.

2.8 Запуск системи тестів TestSuite

Ви запускаєте тестовий набір так само, як виконуєте будь-який індивідуальний тестовий клас.

Розгорніть вузол Test Packages у вікні Projects. Натисніть правою кнопкою миші на клас тестового набору та виберіть Test File. Коли ви запускаєте тестовий набір, IDE запускає тести, включені в набір у тому порядку, в якому вони перелічені. Результати відображаються у вікні результатів тесту JUnit (рис. 2.5).

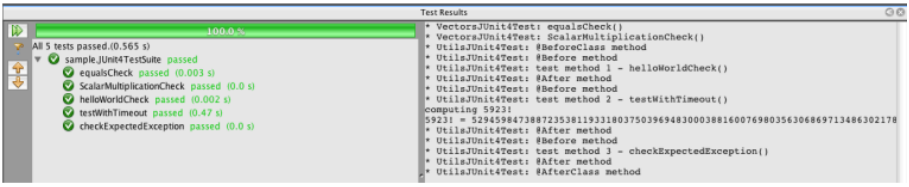

Рисунок 2.5 – Вікно JUnit Test Results

У цьому вікні ви можете побачити результати тесту для набору тестів JUnit 4. Тестовий набір запустив класи тестів

UtilsJUnit4Test та VectorsJUnit4Test як єдиний тест і показав результати тестування на лівій панелі як результат одного тесту. Вивід на правій панелі такий самий, як при тестуванні окремо.

СПИСОК ЛІТЕРАТУРИ

1. https://refactoring.guru/uk.
2. https://sourcemaking.com.

ДОДАТОК
(обов'язковий)
Початковий код програми

```
package sample
import java.lang.reflect.Method;
import java.math.BigInteger;
/**
*
* @author Vyacheslav Moskalenko
*/
public class Utils {
    private Utils() { }

    public static String concatWords(String... words) {
    StringBuilder buf = new StringBuilder();
        for (String word : words) {
            buf.append(word);
        }
        return buf.toString();
    }

    public static String computeFactorial(int number)
            throws IllegalArgumentException {
        if (number < 1) {
            throw new IllegalArgumentException("zero or negative parameter (" +
number + ')');
        }
        BigInteger factorial = new BigInteger("1");
        for (int i = 2; i <= number; i++) {
            factorial = factorial.multiply(new BigInteger(String.valueOf(i)));
        }
        return factorial.toString();
    }

    public static String normalizeWord(String word) {
        try {
            int i;
            Class<?> normalizerClass = Class.forName("java.text.Normalizer");
            Class<?> normalizerFormClass = null;
            Class<?>[] nestedClasses = normalizerClass.getDeclaredClasses();
            for (i = 0; i < nestedClasses.length; i++) {
                Class<?> nestedClass = nestedClasses[i];
                if (nestedClass.getName().equals("java.text.Normalizer$Form")){
                    normalizerFormClass = nestedClass;
                }
            }
            assert normalizerFormClass.isEnum();
            Method methodNormalize = normalizerClass.getDeclaredMethod(
                    "normalize",
                    CharSequence.class,
                    normalizerFormClass
            );
            Object nfcNormalization = null;
            Object[] constants = normalizerFormClass.getEnumConstants();
            for (i = 0; i < constants.length; i++) {
                Object constant = constants[i];
                if (constant.toString().equals("NFC")) {
                    nfcNormalization = constant;
```

Продовження додатку А

```
                }
            }
            return       (String)       methodNormalize.invoke(null,       word,
nfcNormalization);
        } catch (Exception ex) {
            return null;
        }
    }
}

package sample

/**
 * Sample utility class for vector algebra.
 *
 * @author Vyacheslav Moskalenko
 */
public final class Vectors {

    private Vectors() {}

    /**
    * Checks whether the given vectors are equal.
    */
    public static boolean equal(int[] a, int[] b) {
        if ((a == null) || (b == null)) {
            throw new IllegalArgumentException("null argument");
        }
        if (a.length != b.length) {
            return false;
        }
        for (int i = 0; i < a.length; i++) {
            if (a[i] != b[i]) {
                return false;
            }
        }
        return true;
    }

    /**
    * Scalar multiplication of given vectors.
    */
    public static int scalarMultiplication(int[] a, int[] b) {
        if ((a == null) || (b == null)) {
            throw new IllegalArgumentException("null argument");
        }
        if (a.length != b.length) {
            throw new IllegalArgumentException(
                    "different tuple of the vectors ("
                    + a.length + ", " + b.length + ')'
            );
        }
        int sum = 0;
        for (int i = 0; i < a.length; i++) {
            sum += a[i] * b[i];
        }
        return sum;
    }
}
```

Навчальне видання

Методичні вказівки
до практичних занять
із дисципліни «Реінжиніринг і верифікація
програмного забезпечення» на тему «Модульне тестування з
використанням бібліотеки JUnit»
для студентів спеціальності 122 «Комп'ютерні науки»
всіх форм навчання

Відповідальний за випуск А. С. Довбиш
Редактори : Н. З. Клочко, Н. М. Мажуга
Комп'ютерне верстання В. В. Москаленко

Формат 60х84/16. Ум.-друк. арк. 1,63. Обл.-вид. арк. 1,89.

Видавець і виготовлювач
Сумський державний університет,
вул. Римського-Корсакова, 2, м. Суми, 40007
Свідоцтво суб'єкта видавничої справи ДК № 3062 від 17.12.2007.