

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ. ІГОРЯ СІКОРСЬКОГО»

ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра «Безпеки інформаційних і комунікаційних систем»

**Курсова робота**

з дисципліни

«*Технологія забезпечення якості програмних засобів*»

на тему:

**«JUnit - бібліотека для тестування програмного забезпечення для мови Java»**

|  |  |
| --- | --- |
| **Перевірив:**  Викладач  Ткач В. М. | **Виконали:**  студенти групи ФБ-52  Вербицький Дмитро  Ілляшенко Олександр |

Київ 2017

ЗМІСТ

Вступ…………………..………………………………...…….…………..2

Розділ І. Поняття тестування……………………....……....…………….3

* 1. . Рівні тестування………………………....…......….…………..5
     1. Модульне тестування[Ошибка: источник перекрёстной ссылки не найден](#_Toc439758368)[Ошибка: источник перекрёстной ссылки не найден](#_Toc439758372)……..…….....……...……………..5

1.2. [Формування вимог](#_Toc439758374)…………………………….……………...7

1.3. Бібліотека тестування JUnit………...……………...………….8

1.4. Аналіз вимог…………………………...……....………………9

Розділ ІІ. [Проектування програмного комплексу](#_Toc439758378)………....…………..10

2.1. Структура проекту…………………………….....…………...10[Ошибка: источник перекрёстной ссылки не найден](#_Toc439758378)

2.2. Основні можливості. Сценарії та блоки..……....……………11

Висновок………………………………………...…….…………………24

Література…………………………………………...……..…………….25

Додатки………………………………………………………………..…26

**ВСТУП**

Створення власних сайтів, додатків та веб-сервісів грає істотну роль в більшості сфер бізнесу. Більш того, бізнес модель безлічі компаній, таких як Яндекс, Google, Facebook полягає виключно в отриманні прибутку від якісних інтернет-сервісів, реклами, платного контенту тощо.

Веб-додатки можуть приносити величезний прибуток, через це конкуренція в цій сфері дуже висока. Навіть невеликі помилки, допущені при розробці, можуть вплинути на вибір користувача щодо того, якому з конкурентних сервісів йому надати перевагу. Тому, як і в багатьох інших областях, тестування грає дуже важливу роль і в процесі створення веб-додатків. Розробка веб-додатків має на увазі рішення безлічі завдань з безлічі різних областей, таких як: робота з великими обсягами даних, побудова архітектури, реалізація функціоналу, мережева безпека, дизайн та інше.

Оскільки число можливих тестів навіть для нескладних програмних компонентів практично нескінченне - стратегія тестування полягає в тому, щоб провести всі можливі тести з урахуванням наявного часу та ресурсів. Як результат: програмне забезпечення тестується стандартним виконанням програми з метою виявлення багів.

Тестування ПЗ може надавати об'єктивну, незалежну інформацію про якість ПЗ, ризики відмови, як для користувачів так і для замовників.

Об’єктом досліджень у даній курсовій роботі є JUnit testing framework. Предметом дослідженнь є способи використання на практиці та можливості даного фреймворку для тестування.

Основна увага акцентується на розкритті і описі корисних можливостей JUnit. Також досить велика увага приділяється можливостям застосування фреймворку та детальному огляду спеціально розроблених тестових прикладів, на яких можна продемонструвати основні можливості фреймворку. Отож, метою даної курсової роботи є дослідження та систематизація можливостей застосувань фреймворка для тестування ПЗ (програмного забезпечення) JUnit.

Розділ І

**ПОНЯТТЯ ТЕСТУВАННЯ**

*Тестування програмного забезпечення* (англ. Software Testing) — це процес технічного дослідження, призначений для виявлення інформації про якість продукту відносно контексту, в якому він має використовуватись. Техніка тестування також включає як процес пошуку помилок або інших дефектів, так і випробування програмних складових з метою оцінки.

Існує два основних види тестування: ручне і автоматизованне.

*Ручне тестування* — це процес ручної перевірки програмного забезпечення на помилки. Тестувальник має відігравати роль користувача програми й використовувати властивості програми для знаходження помилок у роботі програми. Для професійного тестування тестувальник часто користується написаним планом тестування з варіантами тестування (test cases).

*Автоматизоване тестування програмного забезпечення* — частина процесу тестування на етапі контролю якості в процесі розробки програмного забезпечення. Воно використовує програмні засоби для виконання тестів і перевірки результатів виконання, що допомагає скоротити час тестування і спростити його процес.

Головною ідеєю автоматизованого тестування є повна заміна людської праці. Програміст раз створює програму, яка перевіряє всі підготовлені заздалегідь тестові сценарії. Далі сценарії тільки підтримуються в актуальному робочому стані. Створення таких програм дозволяє мінімізувати процес ручного тестування в життєвому циклу розробки ПО. Хочеться відзначити, що даний підхід економить час співробітників відділу по якості, а також помітно прискорює роботу групи розробників і весь процес розробки в цілому.

Мінусом автоматизації тестування є те, що воно може бути застосовано тільки відносно роботи з тривалими проектами. Створення програми тестування забирає багато часу і витрачати цей час для маленьких проектів просто безглуздо. Як правило, автоматизоване тестування ефективно застосовується на відносно простих сценаріях і перевірках. Автоматизація складних багатокрокових сценаріїв може зайняти дуже багато часу, зважаючи на непередбачені нюанси або технічні проблеми.

Найпоширенішою формою автоматизації є тестування додатків через графічний інтерфейс користувача. Популярність такого виду тестування пояснюється двома факторами: по-перше, додаток тестується тим же способом, яким його буде використовувати людина, по-друге, можна тестувати додаток, не маючи при цьому доступу до вихідного коду. З іншого боку, при зміні інтерфейсу програми необхідно повністю переписати всі тести, які пов'язані з оновленими вікнами, що при великій кількості тестів може відняти значні ресурси.

Не дивлячись на нюанси автоматизованого тестування, даний вид не здає позиції, і компанії впроваджують його в свій робочий процес.

Саме тому в даній курсовій роботі розглядається бібліотека саме для автоматизованого тестування.

**1.1 РІВНІ ТЕСТУВАННЯ**

Тестування розділяється на рівні взаємодії модулів по наступним групам:

• *Модульне тестування* - тестування окремих ділянок коду;

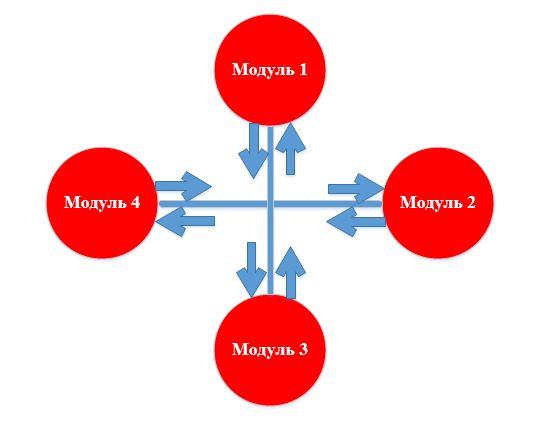
*• Інтеграційне тестування* - тестування взаємодії і спільної роботи компонентів;

*• Системне тестування* - тестування всієї системи в цілому;

*• Приймальне тестування* - підсумкове тестування готової системи на відповідність вимогам.

Метою даного проекту є демонстрація роботи саме модульного тестування

**1.1.1 МОДУЛЬНЕ ТЕСТУВАННЯ**[**Ошибка: источник перекрёстной ссылки не найден**](#_Toc439758368)[**Ошибка: источник перекрёстной ссылки не найден**](#_Toc439758372)

*Модульне тестування* (Unit testing) – ізольоване тестування кожного окремого елемента програмного забезпечення (це можуть бути модулі, класи, об’єкти, функції), в штучно створеному середовищі. Саме потреба в створенні штучного робочого середовища для певного модуля, вимагає від тестувальника знань в автоматизації тестування програмного забезпечення. Перевірка окремих елементів (юнітів) виконується з використанням драйверів і заглушок, і зазвичай виконується програмістами.

*Завдання модульного тестування* – довести, що кожен модуль здатний функціонувати окремо.

Один з найбільш ефективних підходів до компонентного (модульного) тестування – це підготовка автоматизованих тестів до початку основного кодування (розробки) програмного забезпечення. Це називається розробка «через тестування» (test-driven development) або підхід «початкового тестування» (test first approach). При цьому підході створюються і інтегруються невеликі шматки коду, паралельно запускаються тести, написані до початку кодування. Розробка ведеться до тих пір, поки всі тести не будуть успішно пройдені.

Також, модульне тестування має бути менш витратним при пошуку дефектів, ніж інші види тестів і має знижувати час налагодження коду.

*Переваги модульного тестування:*

1) Модульне тестування мотивує писати код максимально оптимізованим, проводити рефакторинг (спрощення коду програми, не зачіпаючи її функціональність), так як за допомогою Unit–тестування можна легко перевірити працездатність даного компонента.

2) Необхідність розмежування реалізації від інтерфейсу (зважаючи на особливості модульного тестування), що дозволяє мінімізувати залежності в системі.

3) Документація Unit–тестів може служити прикладом «живого документа» для кожного класу, що тестується даними способом.

4) Оцінюючи кожен елемент ізольовано, і підтверджуючи коректність його роботи, простіше точно встановити проблему, ніж якби елемент був частиною системи.

5) Модульне тестування допомагає краще зрозуміти роль кожного класу на фоні всієї програмної системи.

Також, при розробці через тестування (TDD), яке активно використовується в екстремальному програмуванні, модульне тестування є одним з основних інструментів.

[**Ошибка: источник перекрёстной ссылки не найден**](#_Toc439758373)

**1.2 ФОРМУВАННЯ ВИМОГ**

Спочатку необхідно точно сформулювати основні ідеї, які визначають функціонал системи автоматизованого функціонального тестування. Правильно складені вимоги до проекту є гарантією того, що він буде виконаний максимально якісно і матиме повністю необхідний функціонал.

Таким чином, до розроблюваної системі автоматизованого функціонального тестування ставляться такі вимоги:

1. Архітектура проекту повинна бути спроектована відповідно до ідей ООП (чіткий поділ на класи, кожен з яких виконує свою певну функцію), що забезпечить систему правильним розподілом на модулі.

2. Простота створення і підтримки тестів навіть для людини, що раніше не працювала в рамках автоматизації тестування, але володіє навичками програмування;

3. Ефективність - система повинна бути оптимізованою, швидкодіючою і здатною здійснювати тестування на різних рівнях від інтеграційного до системного;

4. Кросплатформеність - запускати тести на різних платформах і пристроях, для проведення тестування сумісності;

5. Модульність - система повинна складатися з окремих модулів, які відповідають за окрему задачу розробки тестів;

7. Можливість розширення - функціонал системи повинен бути легко розширюваним.

**1.3 БІБЛІОТЕКА ТЕСТУВАННЯ JUnit**

*JUnit* — бібліотека для тестування програмного забезпечення для мови Java.

Створений Кентом Беком і Еріком Гаммою, JUnit є представником родини фреймворків xUnit для різних мов програмування, яка бере початок у SUnit Кента Бека для Smalltalk. JUnit породив екосистему розширень — JMock, EasyMock, DbUnit, HttpUnit, Selenium тощо.

jUnit застосовується для модульного тестування, яке дозволяє перевіряти на правильність окремі модулі вихідного коду програми. Перевага даного підходу полягає в ізолюванні окремо взятого модуля від інших. При цьому, мета такого методу дозволяє програмісту упевнитися, що модуль, сам по собі, здатний працювати коректно.

*Необхідність використання JUnit*

JUnit дозволяє в будь-який момент швидко переконатися в працездатності коду. Якщо програма не є зовсім простою і включає безліч класів і методів, то для її перевірки може знадобитися чимало часу. Природньо, що даний процес краще автоматизувати. Використання JUnit дозволяє перевірити код програми без значних зусиль і не займає багато часу.

Юніт-тести класів і функцій є свого роду документацією до того, що очікується в результаті їх виконання. І не просто документацією, а документацією яка може автоматично перевіряти код на відповідність пред'явленим функціям.

*Види тестування і місце JUnit тестування в класифікації*

Тестування програмного забезпечення можна розділити на два види:

* тестування чорного ящика;
* тестування білого ящика.

Під час тестування програми, як чорного ящика внутрішня структура додатка до уваги не береться. Все, що має значення - це функціональність, яку програма має забезпечити. При тестуванні програми, як білого ящика до уваги береться внутрішня структура, тобто клас і методи. Юніт тестування по визначенню є тестуванням білого ящика.

**1.4 АНАЛІЗ ВИМОГ**

Проводячи аналіз системних вимог, автори прийшли до наступних висновків:

Для розробки проекту було обране середовище (IDE) IntelliJ IDEA Comunity Edition виробництва компанії JetBrains. Дана IDE є безкоштовною, призначена для розробки проектів на Java і має великі можливості розширення за рахунок додаткових плагінів.

IntelliJ IDEA забезпечує єдиний інтерфейс взаємодії з більшістю систем контролю версій, включаючи Git, SVN, Mercurial, CVS, Perforce і TFS. Дозволяється управляти змінами безпосередньо в IDE, що дуже зручно.

Також IDEA оснащена інструментами для збірки, середовищем виконання тестів, інструментами покриття (coverage tools) і вбудованим термінальним вікном.

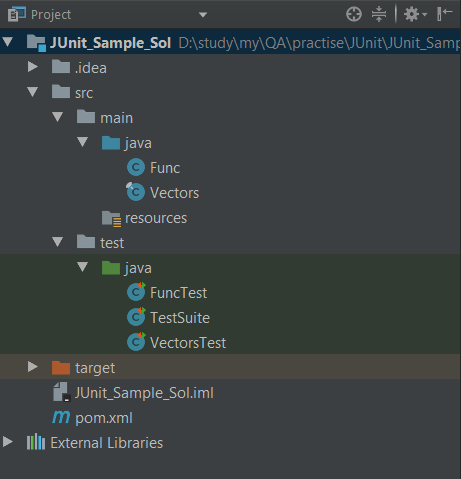
В результаті з'явився набір деяких програмних засобів, кожен з яких відповідає за свою певну задачу. Для Java існує система збірки Maven, призначена для автоматичного складання проектів, виробництва компанії Apache Software Foundation.

Для отримання структури проекту Maven використовує POM мову, яка в свою чергу є підмножиною мови XML. У файлах опису проекту міститься його специфікація, конфігурується, в залежності від інших проектів, індивідуальні фази процесу побудови проекту і список плагінів, що реалізують порядок складання.

Розділ ІІ

[**ПРОЕКТУВАННЯ**](#_Toc439758378) **ПРОГРАМНОГО КОМПЛЕКСУ**

**2.1 СТРУКТУРА ПРОЕКТУ**



**2.2 СЦЕНАРІЇ ТА БЛОКИ**

*pom.xml*

Як в будь-якій системі, в Maven, є свій набір термінів і понять.

Вся структура проекту описується в файлі pom.xml (POM - Project Object Model), який повинен знаходитися в кореневій папці проекту. Кореневий елемент <project>, схема, яка полегшує редагування і перевірку, і версія POM.

Усередині тега project міститься основна і обов'язкова інформація про проект: groupId, artifactId, version.

**<?xml version="1.0" encoding="UTF-8"?>**

**<project xmlns="http://maven.apache.org/POM/4.0.0"**

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>JUnit\_Sample\_Sol</groupId>**

**<artifactId>JUnit\_Sample\_Sol</artifactId>**

**<version>1.0</version>**

Dependencies - це ті бібліотеки, які безпосередньо використовуються у вашому проекті для компіляції коду або його тестування. Вони також мають свої відповідні groupId, artifactId, version. Тег <scope>test</scope> означає, що дана бібліотека Junit використовується виключно виконування тестів.

**<dependencies>**

**<dependency>**

**<groupId>junit</groupId>**

**<artifactId>junit</artifactId>**

**<version>4.12</version>**

**<scope>test</scope>**

**</dependency> </dependencies> </project>**

*Func.java*

**Func.java** містить основні математичні функції, які ми будемо тестувати за допомогою нашої бібліотека JUnit.

Функція **concatWords** буде з’єднує потрібні нам рядки в одне суцільне речення.

**public static String concatWords(String... words) {**

**StringBuilder buf = new StringBuilder();**

**for (String word : words) {**

**buf.append(word);**

**}**

**return buf.toString();**

**}**

Функція **computeFactorial** буде обчислювати факторіал по заданому вхідному параметру. Ми підключили математичну функцію **BigInteger** для обчислення великих значень. В даній функції також присутнє виключення **IllegalArgumentException**, якщо вхідний параметр буде менший ніж 1 нам повідомиться про це відповідним повідомленням.

**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(); }**

Функція **normalizeNumber** буде просто додавати два числа. На прикладі даного методу ми покажем, як можна ігнорувати написані тести, тобто вони виконуватись не будуть. Тому тіло метода і саме обчислення нас не сильно цікавить.

**public static int normalizeNumber(int num1, int num2) {**

**return num1 + num2;**

**}**

**}**

*FuncTest.java*

**FuncTest.java** містить тести, які ми будемо виконувати за допомогою нашої бібліотека JUnit.

Перед початком роботи потрібно підключити всі необхідні класи анотацій та статичні методи, які ми будемо використовувати. По бажанню ми можемо їх імпортувати або купою або по окремості.

**//import org.junit.\*; // for class annotation**

**import org.junit.After;**

**import org.junit.AfterClass;**

**import org.junit.Before;**

**import org.junit.BeforeClass;**

**import org.junit.Ignore;**

**import org.junit.Test;**

**import static org.junit.Assert.\*; // for static method class Assert (include all assert.methods)**

Клас **FuncTest** повинен називатися так само як і назва тесту, він створюється автоматично і містить в собі тіло виконуваних функцій.

**public class FuncTest {**

**public FuncTest() { }**

Сценарії тестів можуть потребувати деяких дій, що передують тесту (наповнення БД, підключення сторонніх файлів) або виконуються після нього (очищення БД, видалення непотрібних елементів). Для цього існують так звані фікстури.

Фікстура - Fixture - стан середовища тестування, яке потрібне для успішного виконання тестового методу. Це може бути набір будь-яких об'єктів, стан бази даних, наявність певних файлів і т. п. Фікстура створюється перед всіма іншими методами в тестовому класі і видаляється після закінчення виконання тестового методу.

Анотація @BeforeClass зазначає метод ініціалізації саме класу тесту. Він виконується тільки перед виконанням будь-яких інших методів в тестовому класі.

**@BeforeClass**

**public static void setUpClass() throws Exception {**

**System.out.println("\* FuncTest: @BeforeClass method");**

**}**

Додавання **println** в кожну функцію дасть можливість відслідковувати процес виконання функцій і порядок їх виконання.

Анотація @AfterClass зазначає метод фіналізації саме класу тесту. Метод фіналізатора тестового класу виконується тільки один раз і тільки після виконання інших методів в тестовому класі.

**@AfterClass**

**public static void tearDownClass() throws Exception {**

**System.out.println("\* FuncTest: @AfterClass method");**

**}**

Анотація @Before зазначає метод ініціалізації тесту. Метод ініціалізації тесту виконується перед кожним тестом в тестовому класі. Для виконання тестів метод ініціалізації тесту не потрібно, його слід використовувати при необхідності ініціалізації деяких змінних до виконання тесту.

**@Before**

**public void setUp() {**

**System.out.println("\* FuncTest: @Before method");**

**}**

Анотація @After зазначає метод фіналізації тесту. Метод фіналізації тесту виконується після кожного тесту в тестовому класі. Метод фіналізації тесту не потрібен для виконання тестів, але фіналізатор може використовуватися для видалення всіх даних, задіяних при виконанні тестів.

**@After**

**public void tearDown() {**

**System.out.println("\* FuncTest: @After method");**

**}**

Для того щоб позначити, що функція є тестом, потрібно позначити її анотацією **@Test.**

Тестування за допомогою простого підтвердження методу з ім'ям helloWorldCheck, що використовує єдине просте твердження для перевірки правильності зчеплення рядків методом.

**@Test**

**public void helloWorldCheck() {**

**System.out.println("\* FuncTest: test method 1 - helloWorldCheck()");**

**assertEquals("Hello, world!", Func.concatWords("Hello", ", ", "world", "!"));**

**}**

Тестування з використанням тайм-ауту. Цей тест демонструє перевірку методу на тривалість виконання. Якщо метод виконується занадто довго, потік виконання тесту переривається, а тест завершується збієм. При перериванні потоку виконання тестовий метод видає TimeoutException.

В даній функції ми будемо перевіряти обчислення факторіалу. Вхідний параметр генерується автоматично, але ми можемо задати його вручну.

**@Test(timeout = 1000)**

**public void testWithTimeout() {**

**System.out.println("\* FuncTest: test method 2 - testWithTimeout()");**

**final int factorialOf = 1 + (int)(100 \* Math.random());**

**System.out.println("computing " + factorialOf + '!');**

**System.out.println(factorialOf + "! = " + Func.computeFactorial(factorialOf));**

**}**

Тестування на «очікуваний виняток». Цей тест використовується для тестування на очікуваний виняток. Метод завершиться збієм, якщо не буде видано очікуваний виняток.

В даній функції ми будемо перевіряти виключення **IllegalArgumentException,**  яке прописали в головній функції **computeFactorial**

**@Test(expected = IllegalArgumentException.class)**

**public void checkExpectedException() {**

**System.out.println("\* FuncTest: test method 3 - checkExpectedException()");**

**final int factorialOf = -5;**

**System.out.println(factorialOf + "! = " + Func.computeFactorial(factorialOf));**

**}**

Відключення тесту. Цей тест включає у себе способи тимчасового відключення тестового методу. В даній функції ми обчислюємо суму для двох чисел, але оскільки перед виконанням тесту стоїть анотація **@Ignore,** даний тест виконуватись не буде.

Також в даному тесті використовується метод який порівнює вхідні значення з очікуваним результатом **assertEquals ,** але більш детально ми його розглянем в наступних прикладах.

**@Ignore**

**@Test**

**public void temporarilyDisabledTest() throws Exception {**

**System.out.println("\* FuncTest: test method 4 - checkExpectedException()");**

**assertEquals(228, Func.normalizeNumber(8, 220));**

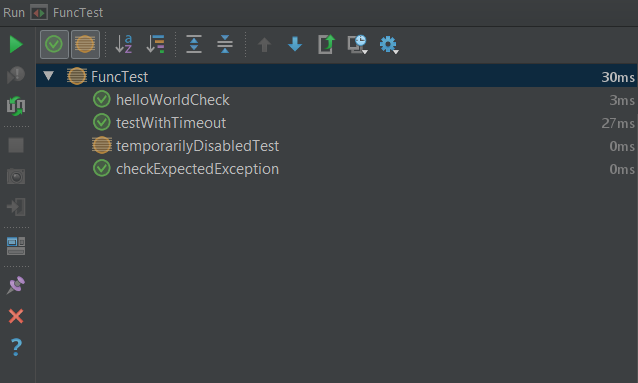
**assertEquals(1337, Func.normalizeNumber(337, 1000));**

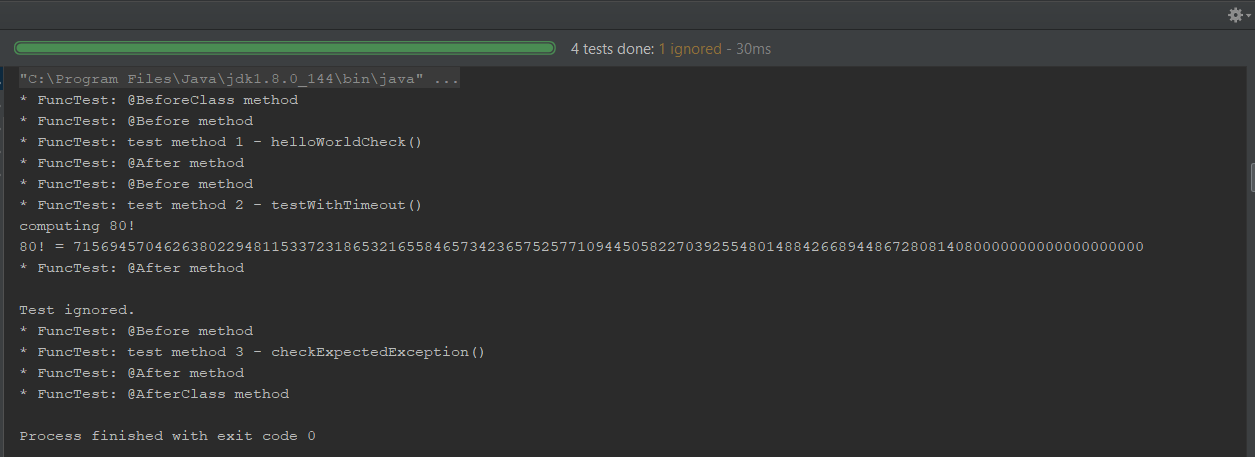
**assertEquals(1488, Func.normalizeNumber(8, 1480));**

**}**

**}**

*Скріншоти виконання FuncTest.java*

****На даному скріншоті ми можемо бачити, які тести виконались, а які були проігноровані.

** Результат виконання всіх класів анотацій показано нижче.

*Vectors.java*

**Vectors.java** містить функції, які ми будемо використовувати для перевірки за допомогою методів класу Assert.

В функції **equal** ми будемо порівнювати на еквівалентність відповідні вектори. Будем перевірятися як кількість символів, так і рівність кожного елементу вектора.

**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;**

**}**

Функції **scalarMultiplication** буде обчислювати скалярний добуток двох векторів.

**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;**

**}**

**}**

*VectorsTest.java*

**VectorsTest.java** містить тести, в яких ми будемо використовувати методи класу Assert.

**public class VectorsTest {**

**public VectorsTest() {**

**}**

**@BeforeClass // метод ініціалізації теста**

**public static void setUpClass() throws Exception { }**

**@AfterClass // метод фіналізатора теста**

**public static void tearDownClass() throws Exception { }**

У цьому тесті використовуються методи JUnit assertTrue і assertFalse для тестування всіх можливих результатів. Для успішного проходження тесту твердження assertTrue повинні бути істинними, а assertFalse – помилковими. Для охоплення можливих перестановок потрібно додати достатню кількість тверджень.

**@Test**

**public void equalsCheck() {**

**System.out.println("\* VectorsTest: 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[]{}));**

**}**

За допомогою методу assertEquals (EXPECTED\_RESULT, ACTUAL\_RESULT) ми перевіряємо всі очікувані результати на основі змінних, введених при виконанні тестового методу

**@Test**

**public void ScalarMultiplicationCheck() {**

**System.out.println("\* VectorsTest: 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}));**

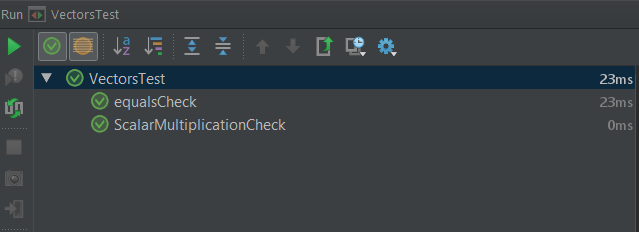
**assertEquals(0, Vectors.scalarMultiplication(new int[]{5, 9}, new int[]{-9, 5}));**

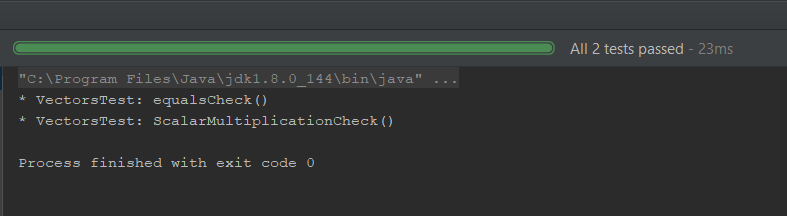
**assertEquals(100, Vectors.scalarMultiplication(new int[]{6, 8}, new int[]{6, 8}));**

**}**

**}**

*Скріншоти виконання VectorsTest.java*

****

****

*TestSuite.java*

**TestSuite.java** містить приклад того, як можна створити набір тестів і виконувати їх одночасно.

Для початку підключимо потрібні нам модулі.

**import org.junit.runner.RunWith;**

**import org.junit.runners.Suite;**

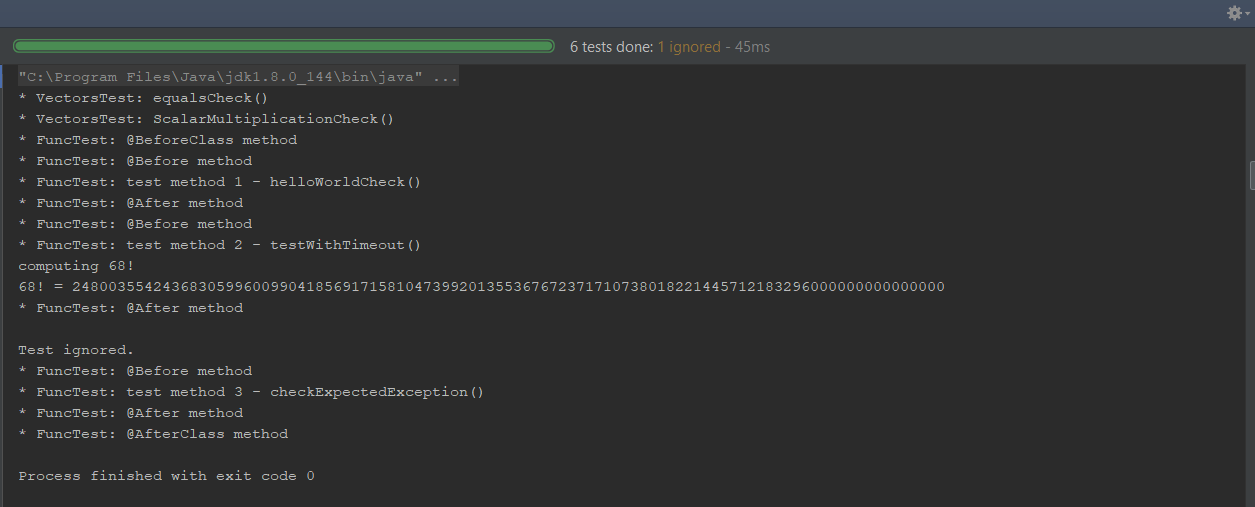
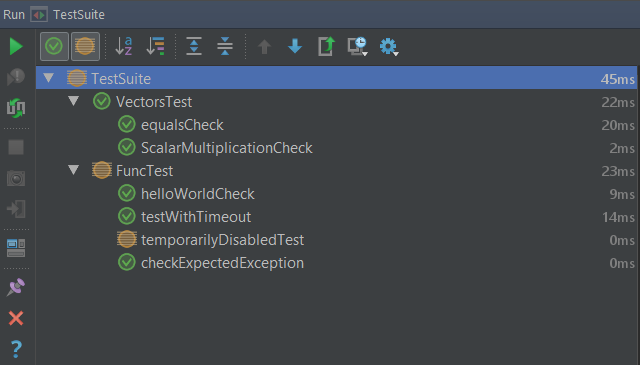
Анотації @RunWith () і @ Suite.SuiteClasses () дозволяють виконати одночасно декілька тестів як один і відобразити результат в одному вікні.

**@RunWith(Suite.class)**

**@Suite.SuiteClasses({VectorsTest.class,UtilsTest.class})**

**public class TestSuite { }**

*Скріншоти виконання TestSuite.java*

****

**ВИСНОВОК**

В першу чергу варто зауважити що ручне і автоматизоване тестування - це взаємодоповнюючі технології. Тому, як в сучасному житті складно випустити якісний продукт без автоматизації, так неможливо обійтися і без ручного тестування.

По-друге хотілося б відзначити, що автоматизовані тести бувають різні і слід вибирати такі тести, які найбільше підходять до даного проекту.

По-третє, в даній курсовій роботі ми дослідили можливості фреймворку для автоматизованого тестування програмного забезпечення JUnit та продемонстрували його використання при тестуванні спеціально розробленого прикладу. Нами було систематизовано та чітко виділено особливості JUnit. Звичайно, продемонструвати всі можливості даної бібліотеки вкрай важко, оскільки її застосовують, в переважній більшості, на справжніх довготривалих проектах, які згодом підтримуються не один реліз. Проте нам вдалося показати всю легкість реалізації, зручність підтримки і модернізації даного фреймворку.

*Таким чином:*

1.Ручне та автоматизоване тестування - це взаємодоповнюючі технології.

2.Автоматизоване тестування вимагає додаткових інвестицій, але дозволяє підвищити якість продукту.

3.Автоматизоване тестування - це розробка (програмування).

4.Автоматизоване тестування гарантує детерміновану перевірку функціональності і може застосовуватися в більшості процесів і на всіх рівнях тестування.

5.Супровід автоматизованих тестів вимагає великих додаткових витрат при активній зміні системи, яка тестується.

6.jUnit є досить вдалим рішенням завдань, пов'язаних з тестуванням java-додатків. Простий і в той же час дуже потужний інструмент для написання unit-тестів.

**ЛІТЕРАТУРА**

1. <https://habrahabr.ru/post/169381/>

(Юнит-тестирование для чайников)

4. <https://uk.wikipedia.org/wiki/JUnit>

(JUnit)

1. <http://lviv.qalight.com.ua/baza-znan/modulne-testuvannya/>

(Модульне тестування (Unit testing))

3. <http://java-online.ru/blog-junit.xhtml>

(Тестирование программы, JUnit)

5. <https://netbeans.org/kb/docs/java/junit-intro_ru.html#Exercise_30>

(Написание тестов JUnit в IDE NetBeans)

6. <https://tproger.ru/translations/unit-tests-purposes/>

(Зачем нужны юнит-тесты)

7. <http://junit.org/junit5/docs/current/user-guide/>

(JUnit User Guide)

8. <https://github.com/SKART1/GEOTRACERT/wiki/JUnit-тесты>

(JUnit тесты)

9. <https://www.youtube.com/watch?v=eZEmWSdhLvA>

(JUnit + TDD)

10. <https://www.youtube.com/watch?v=z9jEVLCF5_w>

(Тестирование кода Java с помощью фреймворка JUnit)

**ДОДАТКИ**

***pom.xml***

**<?xml version="1.0" encoding="UTF-8"?>**

**<project xmlns="http://maven.apache.org/POM/4.0.0"**

**xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">**

**<modelVersion>4.0.0</modelVersion>**

**<groupId>JUnit\_Sample\_Sol</groupId>**

**<artifactId>JUnit\_Sample\_Sol</artifactId>**

**<version>1.0</version>**

**<dependencies>**

**<dependency>**

**<groupId>junit</groupId>**

**<artifactId>junit</artifactId>**

**<version>4.12</version>**

**<scope>test</scope>**

**</dependency>**

**</dependencies>**

**</project>**

***Func.java***

**import java.lang.reflect.Method;**

**import java.math.BigInteger;**

**public class Func {**

**private Func() { }**

**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 int normalizeNumber(int num1, int num2) {**

**return num1 + num2;**

**}**

**}**

***FuncTest.java***

**//import org.junit.\*; // for class annotation**

**import org.junit.After;**

**import org.junit.AfterClass;**

**import org.junit.Before;**

**import org.junit.BeforeClass;**

**import org.junit.Ignore;**

**import org.junit.Test;**

**import static org.junit.Assert.\*; // for static method class Assert (include all assert.methods)**

**public class FuncTest {**

**public FuncTest() { }**

**// Fixtures (DB up and clear, upload some files**

**@BeforeClass // Инициализатор класса тестов**

**public static void setUpClass() throws Exception {**

**System.out.println("\* FuncTest: @BeforeClass method");**

**}**

**@AfterClass // Финализатор класса тестов**

**public static void tearDownClass() throws Exception {**

**System.out.println("\* FuncTest: @AfterClass method");**

**}**

**@Before**

**public void setUp() {**

**System.out.println("\* FuncTest: @Before method");**

**}**

**@After**

**public void tearDown() {**

**System.out.println("\* FuncTest: @After method");**

**}**

**@Test**

**public void helloWorldCheck() {**

**System.out.println("\* FuncTest: test method 1 - helloWorldCheck()");**

**assertEquals("Hello, world!", Func.concatWords("Hello", ", ", "world", "!"));**

**}**

**@Test(timeout = 1000)**

**public void testWithTimeout() {**

**System.out.println("\* FuncTest: test method 2 - testWithTimeout()");**

**final int factorialOf = 1 + (int)(100 \* Math.random());**

**System.out.println("computing " + factorialOf + '!');**

**System.out.println(factorialOf + "! = " + Func.computeFactorial(factorialOf));**

**}**

**@Test(expected = IllegalArgumentException.class)**

**public void checkExpectedException() {**

**System.out.println("\* FuncTest: test method 3 - checkExpectedException()");**

**final int factorialOf = -5;**

**System.out.println(factorialOf + "! = " + Func.computeFactorial(factorialOf));**

**}**

**@Ignore**

**@Test**

**public void temporarilyDisabledTest() throws Exception {**

**System.out.println("\* FuncTest: test method 4 - checkExpectedException()");**

**assertEquals(228, Func.normalizeNumber(8, 220));**

**assertEquals(1337, Func.normalizeNumber(337, 1000));**

**assertEquals(1488, Func.normalizeNumber(8, 1480));**

**}**

**}**

***Vectors.java***

**public final class Vectors {**

**private Vectors() { }**

**// for equal arguments**

**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;**

**}**

**// for IllegalArgumentException**

**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;**

**}**

**}**

***VectorsTest.java***

**import org.junit.AfterClass;**

**import org.junit.BeforeClass;**

**import org.junit.Test;**

**import static org.junit.Assert.\*;**

**public class VectorsTest {**

**public VectorsTest() {**

**}**

**@BeforeClass // метод инициализатора теста**

**public static void setUpClass() throws Exception {**

**}**

**@AfterClass // метод финализатора теста**

**public static void tearDownClass() throws Exception {**

**}**

**@Test**

**public void equalsCheck() {**

**System.out.println("\* VectorsTest: 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[]{}));**

**}**

**@Test**

**public void ScalarMultiplicationCheck() {**

**System.out.println("\* VectorsTest: 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}));**

**}**

**}**

***TestSuit.java***

**import org.junit.runner.RunWith;**

**import org.junit.runners.Suite;**

**@RunWith(Suite.class)**

**@Suite.SuiteClasses({VectorsTest.class,FuncTest.class})**

**public class TestSuite { }**