ВСТУП ДО МОДУЛЬНОГО ТЕСТУВАННЯ. ФРЕЙМВОРК JUNIT

Питання 4.4

Означення модульного тестування

- Тест – оцінка наших знань, доказ концепцій або експертиза даних.
  - Тест класу – експертиза наших знань, що з'ясовує, чи можна перейти на наступний рівень.
  - Для програмного забезпечення це перевірка достовірності (validation) функціональних та нефункціональних вимог до поставки споживачу.
- Модульне тестування означає валідацію чи виконання перевірки логічної відповідності (sanity check) коду.
  - Sanity check – базовий тест для швидкої оцінки того, чи буде результат обчислень істинним.
  - Це проста перевірка, що встановлює відповідність отриманого матеріалу з очікуваним.
- Поширеною практикою модульного тестування коду є використання print-інструкцій в головному методі або шляхом запуску додатку.
  - Жоден з цих підходів не є коректним.
  - Змішування тестованого та тестуючого коду також не є хорошою практикою.
  - Код ускладнюється та може стати складним у підтримці чи спричинити збій системи за умови некоректного конфігурування.

Розробка через тестування

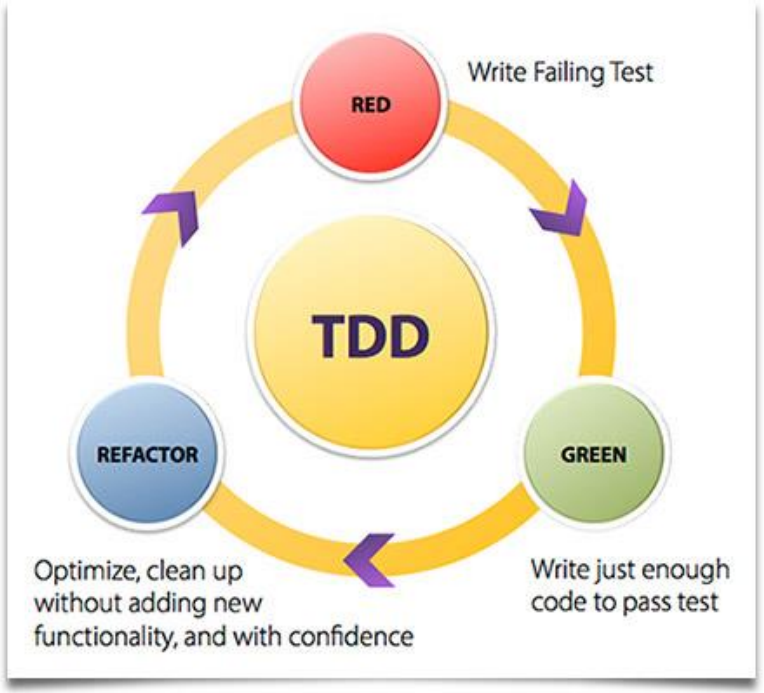
- Модульне тестування – поширена практика в розробці через тестування (**test-driven development, TDD**).
  - TDD – еволюційний підхід до розробки.
  - Пропонує test-fist розробку, коли тестований код пишеться і проходить рефакторинг для задоволення тесту та покращення якості.
- У TDD модульні тести виступають рушієм проектування.
  - Ви пишете код, щоб задовольняти непройдений тест.
  - Таким чином код обмежується лише найнеобхіднішим.
- Тести забезпечують швидку, автоматизовану регресію для рефакторингу та нових покращень.
  - Користувачі не виконують модульних тестів, тому код тестів не постачається з проектом.
  - Код і тести знаходяться в одному пакеті, що дозволяє тесту мати доступ до захищених методів/властивостей.
  - Особливо корисно при роботі з legacy-кодом

Фреймворки тестування

- Код Java може тестуватись, використовуючи code-driven unit testing framework. Для Java доступно декілька таких фреймворків:
  - SpryTest
  - Jtest
  - JUnit
  - TestNG
- JUnit найбільш популярний та поширений, розглядатимемо JUnit 4

Red-green-refactor

- Розробка через тестування базується на test-first concept з екстремального програмування.
- Процедура, яка управляє циклами розробки, називається red-green-refactor:
  - 1. Напишіть тест.
  - 2. Запустіть усі тести.
  - 3. Напишіть код реалізації.
  - 4. Запустіть усі тести.
  - 5. Виконайте рефакторинг.
  - 6. Запустіть усі тести.



Red-green-refactor

- Оскільки тест написано до дійсної реалізації тестованого коду, передбачається його невиконання.
  - Якщо це не так, тест неправильний (хибно-позитивний результат).
  - It describes something that already exists or it was written incorrectly.
  - Такі тести слід видаляти чи піддавати рефакторингу.
- У процесі написання тесті ми знаходимось у червоному стані.
- Після завершення реалізації тесту, всі тести мають проходитись і перейти в зелений стан.
  - Якщо останній тест не проходить, реалізація неправильна чи потребує корекції.
  - Або завершений тест некоректний, або реалізація не відповідає заданим специфікаціям.

Коли тест стає червоним…

- Природна реакція – витратити якомога більше зусиль на поправку коду, щоб всі тести проходились.
  - Проте це неправильно.
  - Якщо fix не зроблено за кілька хвилин, краще revert зміни.
  - Кілька хвилин тому все ж працювало.
- Реалізація, яка порушує роботу, явно неправильна, тому краще повернутись до попередньої точки та знову подумати над коректною реалізацією тесту.
  - Витрачаємо хвилини на неправильну реалізацію, а не набагато більше часу на виправлення того, що з самого початку не було зроблено правильно.
  - Існуюче покриття коду (за виключенням останнього тесту) має бути священним.
- Змінюємо існуючий код через навмисний рефакторинг, а не для виправлення недавно написаного коду.
  - Не робіть реалізацію останнього тесту остаточною, проте постачайте код, якого достатньо для проходження тесту.

Процес рефакторингу

- Пишіть код, як забажаєте, але швидко!
  - Як тільки всі тести стають зеленими, можна сконцентруватись на рефакторингу коду.
  - Робимо код краще та оптимальніше без внесення нових фіч.
- У процесі рефакторингу всі тести мають проходитись.
  - Якщо в процесі рефакторингу один тест провалився, рефакторинг порушує існуючий функціонал і має відмінятись.
  - На цьому етапі ми не представляємо ні нових тестів, ні нового функціоналу.
  - Таким чином забезпечуємо коректність коду та зниження майбутніх витрат на підтримку.
  - Як тільки рефакторинг завершено, процес повторюється.
  - Це нескінченний цикл дуже коротких циклів.

Unit-тест з технічної сторони

- Це клас, який лежить у тестовому ресурсі та призначений лише для тестування логіки, а не використання у production-коді.

```
@Test
public void testMultiply() {
    // Тестируемый класс
    MyClass tester = new MyClass();

    // Проверяемый метод
    assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
}
```

  - У JUnit передбачається, що всі тестовані методи можуть бути виконаними в довільному порядку.
  - Тому тести не повинні залежати один від одного.
  - Для того, щоб вказати, що метод є тестовим, його слід анотувати @Test, після чого цей метод можна буде запускати в окремому потоці для виконання тестування.

Доступні анотації JUnit 4.x

Аннотация	Описание
@Test public void method()	Аннотация **@Test** определяет что метод method() является тестовым.
@Before public void method()	Аннотация **@Before указывает** на то, что метод будет выполнятся перед каждым тестируемым методом @Test.
@After public void method()	Аннотация **@After указывает** на то что метод будет выполнятся после каждого тестируемого метода @Test
@BeforeClass public static void method()	Аннотация **@BeforeClass** указывает на то, что метод будет выполнятся в начале всех тестов, а точней в момент запуска тестов(перед всеми тестами @Test).
@AfterClass public static void method()	Аннотация **@AfterClass** указывает на то, что метод будет выполнятся после всех тестов.
@Ignore	Аннотация **@Ignore** говорит, что метод будет проигнорирован в момент проведения тестирования.
@Test (**expected** = Exception.class)	(**expected** = Exception.class) — указывает на то, что в данном тестовом методе вы преднамеренно ожидается Exception.
@Test (**timeout**=100)	(**timeout**=100) — указывает, что тестируемый метод не должен занимать больше чем 100 миллисекунд.

Перевіряючі методи (основні)

Метод	Описание
fail(String)	Указывает на то что бы тестовый метод завалился при этом выводя текстовое сообщение.
assertTrue([message], boolean condition)	Проверяет, что логическое условие истинно.
assertsEquals([String message], expected, actual)	Проверяет, что два значения совпадают. *Примечание*: для массивов проверяются ссылки, а не содержание массивов.
assertNull([message], object)	Проверяет, что объект является пустым **null**.
assertNotNull([message], object)	Проверяет, что объект не является пустым **null**.
assertSame([String], expected, actual)	Проверяет, что обе переменные относятся к одному объекту.
assertNotSame([String], expected, actual)	Проверяет, что обе переменные относятся к разным объектам.

Правила

- Подоба утиліт для тестів, які додають функціональність до та після виконання тесту.
  - Наприклад, є вбудовані правила щодо встановлення таймауту для теста (Timeout), визначення очікуваних винятків (ExpectedException), роботи з тимчасовими файлами (TemporaryFolder) тощо.
  - Для оголошення правила необхідно створити публічне нестатичне поле типу, породженого від MethodRule, та заанотувати його за допомогою @Rule.

```
@Rule
public final TemporaryFolder folder = new TemporaryFolder();


@Rule
public final Timeout timeout = new Timeout(1000);


@Rule
public final ExpectedException thrown = ExpectedException.none();
```

Запускалки

- Те, як запускається тест, теж можна сконфігурувати за допомогою ***@RunWith***.
  - Вказаний в анотації клас має наслідуватись від Runner.
- ***JUnit38ClassRunner*** призначений для запуску тестів JUnit 3.
  - ***SuiteMethod*** або ***AllTests*** теж призначені для запуску JUnit 3 тестів. На відміну від попередньої, в цю запускалку передається клас зі статичним методом suite(), що повертає тест (послідовність усіх тестів).
- ***JUnit4*** — запускалка за умовчанням, призначена для запуску тестів JUnit 4.
- ***Suite*** — еквівалент попередньої запускалки для JUnit 4 тестів.
  - Для налаштування тестів, що будуть запускатись, використовується анотація *@SuiteClasses*.

```
@Suite.SuiteClasses( { OtherJUnit4Test.class, StringUtilsJUnit4Test.class })
@RunWith(Suite.class)
public class JUnit4TestSuite {
}
```

Запускалки

- ***Enclosed*** — те ж, що і в попередньому варіанті, але замість налаштування за допомогою анотації використовуються всі внутрішні класи.
- ***Categories*** — намагання організувати тести в категорії (групи).
  - Тестам задається категорія за допомогою @Category, потім налаштовуються категорії тестів у сьюті, які запускаються.
- ***Parameterized*** — дозволяє писати параметризовані тести.
  - Для цього у тестуючому класі оголошується статичний метод, який повертає список даних, які потім будуть використані як аргументи конструктора класу.
- ***Theories*** — схожа до попередньої запускалки, проте параметризує тестовий метод, а не конструктор.
  - Дані помічаються за допомогою *@DataPoints* та *@DataPoint*, тестовий метод — за допомогою *@Theory*.

Приклад з категоріями

```
public class StringUtilsJUnit4CategoriesTest extends Assert {
  //...

  @Category(Unit.class)
  @Test
  public void testIsEmpty() {
    //...
  }

  //...
}

@RunWith(Categories.class)
@Categories.IncludeCategory(Unit.class)
@Suite.SuiteClasses( { OtherJUnit4Test.class, StringUtilsJUnit4CategoriesTest.class })
public class JUnit4TestSuite {
}
```

Приклад з *Parameterized*

```java
@RunWith(Parameterized.class)
public class StringUtilsJUnit4ParameterizedTest extends Assert {
  private final CharSequence testData;
  private final boolean expected;

  public StringUtilsJUnit4ParameterizedTest(final CharSequence testData, final boolean expected) {
    this.testData = testData;
    this.expected = expected;
  }

  @Test
  public void testIsEmpty() {
    final boolean actual = StringUtils.isEmpty(testData);
    assertEquals(expected, actual);
  }

  @Parameterized.Parameters
  public static List<Object[]> isEmptyData() {
    return Arrays.asList(new Object[][] {
      { null, true },
      { "", true },
      { " ", false },
      { "some string", false },
    });
  }
}
```

Приклад з *Theories*

```
@RunWith(Theories.class)
public class StringUtilsJUnit4TheoryTest extends Assert {

  @DataPoints
  public static Object[][] isEmptyData = new Object[][] {
      { "", true },
      { " ", false },
      { "some string", false },
  };

  @DataPoint
  public static Object[] nullData = new Object[] { null, true };

  @Theory
  public void testEmpty(final Object... testData) {
    final boolean actual = StringUtils.isEmpty((CharSequence) testData[0]);
    assertEquals(testData[1], actual);
  }
}
```