

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ

**до лабораторних робіт з дисципліни
"Інженерія програмного забезпечення"**

для студентів напрямку 6.050102 "Комп'ютерна інженерія"
всіх форм навчання

І частина

2015

Методичні вказівки до виконання лабораторних робіт з дисципліни "Інженерія програмного забезпечення" для студентів напряму 6.050102 "Комп'ютерна інженерія" всіх форм навчання (І частина) / Укл.: В.В. Шкарупило, М.Б. Ільяшенко, О.В. Польська – Запоріжжя: ЗНТУ, 2015. – 46 с.

Укладачі: В.В. Шкарупило, ст.викладач
М.Б. Ільяшенко, доцент, к.т.н.
О.В. Польська, ст.викладач

Рецензент: Р.К. Кудерметов, доцент, к.т.н.

Відповідальний
за випуск: О.В. Польська, ст.викладач

Затверджено:
на засіданні кафедри
"Комп'ютерні системи та мережі"
Протокол № 10 від 30.06.2015 р.

ЗМІСТ

1 ЛАБОРАТОРНА РОБОТА №1 БІБЛІОТЕКА МОДУЛЬНОГО ТЕСТУВАННЯ <i>JUNIT</i>	4
1.1 Теоретичні відомості.....	4
1.2 Порядок виконання роботи.....	12
1.3 Зміст звіту.....	17
1.4 Контрольні питання.....	18
2 ЛАБОРАТОРНА РОБОТА №2 ЦЕНТРАЛІЗОВАНА СИСТЕМА КЕРУВАННЯ ВЕРСІЯМИ <i>SUBVERSION</i>	19
2.1 Теоретичні відомості.....	19
2.2 Порядок виконання роботи.....	24
2.4 Контрольні питання.....	30
3 ЛАБОРАТОРНА РОБОТА №3 ТЕХНОЛОГІЯ <i>JAX-WS</i>	31
3.1 Теоретичні відомості.....	31
3.2 Порядок виконання роботи.....	41
3.3 Зміст звіту.....	43
3.4 Контрольні питання.....	43
ЛІТЕРАТУРА	44
ДОДАТОК А Рекомендації до виконання завдань згідно варіантів	45
ДОДАТОК Б Java-методи класів <i>string</i> , <i>stringbuilder</i> та <i>integer</i> для роботи з рядками	46

1 ЛАБОРАТОРНА РОБОТА №1

БІБЛІОТЕКА МОДУЛЬНОГО ТЕСТУВАННЯ *JUnit*

Мета роботи – набути навичок роботи з бібліотекою модульного тестування *JUnit*.

1.1 Теоретичні відомості

На сучасному етапі розвитку підходів до розробки програмних систем здійснення при цьому лише навантажувального тестування або тестування продуктивності систем є недостатнім. Причина цього полягає у поширенні об'єктно-орієнтованого підходу при проектуванні та програмуванні. У зв'язку із цим не досить ефективного виявлення помилок виключно у вже розробленому програмному продукті. Це, зокрема, йде всупереч центральній тезі актуального на сьогодні ітераційного підходу до розробки, згідно якої на кожному умовно виділеному етапі процесу розробки має здійснюватися певна перевірка поточних здобутків [1].

При розробці на основі тестування, так званому *TDD*-підходу (*Test-driven Development*), спочатку створюються тести для певних компонентів або модулів системи, а вже потім – самі модулі, що відповідають тестам [2]. Поширеним засобом створення названих тестів є бібліотека модульного тестування *JUnit*. Центральними при цьому є поняття *модульного тесту* (*unit-тесту*) та *тестованого модуля*.

Під *unit-тестом* будемо розуміти *Java*-метод у складі певного *Java*-класу, позначений анотацією `@Test` та призначений для перевірки окремого тестованого модуля – *Java*-методу іншого *Java*-класу, правильність роботи якого ми намагаємося перевірити.

При створенні та використанні модульних тестів будемо користуватися середовищем *NetBeans 8.0 IDE*. При цьому варто пам'ятати, що в проєкті, створюваному в зазначеному середовищі, класи тестів мають групуватися за каталогами, які, в свою чергу, мають знаходитися в каталозі "*Тести*" проєкту.

Ідея *unit*-тестування полягає у виконанні перевірки програмних модулів індукційним шляхом (знизу вгору) – від базових (вихідних) – до похідних (результуючих) модулів. Це дозволяє спростити в процесі розробки внесення змін як в *Java*-методи (при рефакторингу програмного коду), так і у відповідні *unit*-тести.

З використанням бібліотеки *JUnit* можна також здійснювати параметризоване модульне тестування, за якого тестований модуль перевіряється не на окремому входному значенні, а на множині входних даних. Це сприяє підвищенню достовірності одержуваних результатів тестування, за рахунок поліпшення повноти охоплення сценаріїв потенційно можливого застосування окремого модуля.

На основі бібліотеки *JUnit*, окрім непараметризованого та параметризованого *unit*-тестувань, може бути здійснене також і тестування у пакетному режимі – *suite*-тестування. Призначення останнього – автоматизація безпосередньо процесу тестування, що дасть змогу одержувати результуючі дані за результатами перевірок кількох модулів.

В даній лабораторній роботі використовувалась версія бібліотеки *JUnit*–4.11, що підтримується середовищем розробки *NetBeans 8.0*. Використання актуальної бібліотеки (<http://junit.org/>) дасть змогу користуватися актуальними версіями методів та анотацій при виконанні роботи.

Для успішного виконання роботи до *Java*-проекту, створеного в середовищі *NetBeans 8.0*, слід підключити бібліотеку *JUnit*, представлену *jar*-архівом *junit-4.11.jar*. Для цього слід виконати наступне: у властивостях проекту в категорії *Бібліотеки* на вкладці *Компілювати* вказати з використанням елементу керування *Додати файл JAR/папку* вказати шлях до фактичного розміщення архіву *junit-4.11.jar*.

Модульне тестування ґрунтується на наступних принципах [3]:

- окремий модульний тест має виконуватися незалежно від інших *unit*-тестів;
- порядок виявлення помилок в модулях, а також порядок появи повідомлень про помилки мають відповідати послідовності виконання модульних тестів;
- при тестуванні має існувати можливість вибору модульного тесту, що підлягає виконанню;

Розробники бібліотеки *JUnit* (*Kent Beck, Erich Gamma*) керувалися наступними принципами:

- використання *unit*-тестів має бути обґрунтованим потребами процесу розробки;
- модульні тести мають бути актуальними при повторному їх використанні.

Відмітні риси *JUnit*:

- використання анотацій у якості засобів ініціалізації та налаштування тестів (`@After`, `@Before`, `@Test...`);
- використання *assert*-методів для перевірки результатів тестувань;
- інтеграція бібліотеки з популярними комплексами засобів автоматизації створення програмних проєктів (*Ant* та *Maven*), а також із поширеними середовищами розробки (*IDE*, *Integrated Development Environment*) – *Eclipse*, *NetBeans*, *JBuilder* та ін.

Модульне тестування також може здійснюватися і для проєктів, створюваних із використанням інших мов програмування: *JavaScript* – *JUnit*, *C++* – *CPPUnit*, *PHP* – *PHPUnit* та ін.

Призначення анотацій *JUnit* [4]:

- `@After` – вивільнення ресурсів (пам'яті), виділених для використання `@Before`-анотованих методів, по завершенні тестування (анотовані методи мають бути `public void`);

- `@AfterClass` – призначення те саме, що й у випадку `@After`-анотації, але використовується для вивільнення ресурсів, виділених з метою створення передумов до виконання декількох модульних тестів (анотовані методи мають бути `public static void`);

- `@Before` – вказування об'єктів, що підлягають створенню, а також методів, що підлягають використанню, передуючи тестуванню. Анотація `@Before` є засобом створення передумов до здійснення тестування (анотовані методи мають бути `public void`);

- `@BeforeClass` – призначення те саме, що й у випадку `@Before`-анотації, але використовується для створення однакових передумов до декількох модульних тестів – методів, позначених `@Test`-анотаціями (анотовані методи мають бути `public static void`);

- `@Ignore` – скасування `@Test`-анотації. Це означає, що модульний тест, помічений як `@Ignore @Test`, виконанню не підлягає. Анотація `@Ignore` також може бути параметризованою (із зазначенням причини скасування `@Test`-анотації) – наприклад `@Ignore("not ready yet") @Test ...`;

- `@Parameterized.Parameters` – вказування статичного методу, що повертає набір тестових даних, необхідних для проведення параметризованого тестування. При цьому необхідно попередньо

використовувати анотацію `@RunWith(Parameterized.class)`. Параметризоване тестування доцільно виконувати за потреби перевірки тестованого модуля на множині вихідних даних;

- `@Rule` – позначення об’єктів, призначених до використання в модульному тесті;

- `@RunWith` – вказування класу, призначеного для виклику з метою виконання модульного тесту. Наприклад, анотація `@RunWith(Parameterized.class)` сигналізує про здійснення параметризованого тестування, а анотація `@RunWith(Suite.class)` – про тестування в пакетному режимі;

- `@Suite` – виконання модульних тестів в пакетному режимі (з метою автоматизації). Для вказування модульних тестів, що мають бути включені до тестового набору. Класи, що містять елементи тестового набору, вказуються у якості аргументів анотації `@Suite.SuiteClasses`;

- `@Test` – позначення модульних тестів. Анотація `@Test` може бути параметризованою (із ключовим словом `timeout`), де в якості параметру вказується значення часу (в мілісекундах), по завершенні якого тест буде вважатися непройденим (*failed*), тобто параметризація здійснюється з метою встановлення часових обмежень на тестування. Наприклад, параметризованою анотацією `@Test(timeout=1000)` можна задати обмеження на час виконання тесту в 1 с.

Шаблон модульного тесту з використанням анотації `@Test` наведено в лістингу 1.1.

```
//Лістинг 1.1 Структура модульного тесту
//імпортування необхідних JUnit-пакетів
import org.junit.Test;
import static org.junit.Assert.*;
public class A { //клас, що містить
    //модульний тест
    @Test        //позначення нижченаведеного методу
    //як модульного тесту
    public void some_method() {
        //створення екземпляру класу, що містить
        //тестований модуль
        B b = new B();
        //привласнення змінній оціночного (очікуваного)
```

```
//значення-результату роботи тестованого модуля
double estimated_val = 12.3;
//одержання фактичного результату
//роботи тестованого модуля
double result = b.getResult();
//порівняння фактичного та оціночного значень
//шляхом використання assert-методу
assertEquals(estimated_val, result,0);
```

Для лістингу 1.1 результат тестування буде успішним (*passed*), якщо фактичне та оціночне значення рівні. Якщо ні – результат тестування буде ідентифікований як невдалий (*failed*).

Основні класи бібліотеки *JUnit* наведено в табл. 1.1 Деякі *assert*-методи наведено в табл. 1.2.

Таблиця 1.1 – Базові класи бібліотеки *JUnit*

Класи	Призначення
Assert	містить множину <i>assert</i> -методів, призначених для створення <i>unit</i> -тестів;
Runner	надає засоби виконання параметризованого тестування.
Suite	дозволяє здійснювати тестування в пакетному режимі;

Таблиця 1.2 – *assert*-методи бібліотеки *JUnit*

Методи	Аргументи та призначення
static void assertEquals(double expected, double actual, double delta);	expected – очікуване значення; actual – фактичне значення; delta – допустиме відхилення (девіація) між фактичним та оціночним значеннями.
static void assertArrayEquals(char[] expected,char[] actual);	expected – масив оціночних значень; actual – масив фактичних значень.
static void assertTrue(boolean condition);	перевірка істинності припущення;
static void assertFalse(boolean condition);	перевірка хибності припущення.

На практиці, проте, перевірка методів, у тілі яких не виконується створення екземплярів інших класів, є малопоширеною. Процес розробки, як правило, ґрунтується на використанні певних шаблонів проектування, коли важко уникнути створення композитних об'єктів на основі декількох базових класів [5]. Розробники при цьому можуть мати у розпорядженні лише інтерфейси базових класів, що унеможливорює здійснення модульного тестування виключно з використанням бібліотеки *JUnit*. Можливим шляхом вирішення такої проблеми є використання при тестуванні *mock*-об'єктів, що являють собою фіктивні реалізації інтерфейсів.

Для створення *mock*-об'єктів при виконанні роботи використаємо бібліотеку *JMock*. Існують, однак, і чимало альтернативних бібліотек, що базуються на тій самій ідеї, – *RMock*, *Mockito* тощо.

Для користування засобами *JMock* при виконанні роботи до *Java*-проєкту, створеного в середовищі *NetBeans 8.0*, слід підключити наступні *jar*-архіви: *hamcrest-core-1.3.jar*, *hamcrest-library-1.3.jar*, *jmock-2.6.0.jar* та *jmock-junit4-2.6.0.jar*.

Розглянемо приклад. Нехай розробник оперує двома сутностями – інтерфейсом постачальника продукції *Supplier* (лістинг 1.2) та класом складу продукції *Storage* (лістинг 1.3). Розглянемо інтерфейс *Supplier* як інтерфейс базового класу – класу, від результатів роботи методу *getData* якого залежить результат роботи методу *getSupply* класу *Storage*.

```
//Лістинг 1.2. Інтерфейс базового класу
package foo;
public interface Supplier {
//метод одержання інформації відносно продукції
    public String getData();
}
```

```
//Лістинг 1.3. Клас із тестованим модулем getSupply
package foo;
public class Storage {
    public void addSupplier(Supplier obj) {
        this.obj = obj;
    }
    public String getSupply() {
```

```

        return "supplied: " + obj.getData();
    }
    private Supplier obj;
}

```

За умови відсутності в розробника класу, що реалізує інтерфейс `Supplier` (лістинг 1.2), неможливо здійснити *unit*-тестування методу `getSupply` класу `Storage` лістингу 1.3, оскільки для цього необхідно створити екземпляр відсутнього класу.

Наприклад, якщо створити модульний тест методу `getSupply`, наведений в лістингу 1.4, то в результаті запуску на виконання такого тесту (*Shift+F6*) виникне помилка *java.lang.NullPointerException*.

```

//Лістинг 1.4. Модульний тест методу getSupply
package storage;
import foo.*;
import org.junit.Test;
import static org.junit.Assert.*;
public class UnitTester {
    @Test
    public void testStorage() {
        assertEquals(new Storage().getSupply(),
            "supplied: ");
    }
}

```

Названа помилка полягає у неможливості одержати посилання на об'єкт класу, який реалізує інтерфейс `Supplier` – з метою виклику методу `getData`. Аби уникнути цього, можна скористатися *mock*-об'єктом – фіктивним об'єктом класу-реалізації інтерфейсу `Supplier`, який нібито існує (лістинг 1.5).

```

//Лістинг 1.5. Модульний тест методу getSupply на
основі mock- об'єкту
package storage;
import foo.*;
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.*;
import org.junit.runners.*;

```

```

import org.jmock.Mockery;
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;
import org.jmock.integration.junit4.JMock;
@RunWith(JMock.class)
public class StorageTester {
    @Rule public JUnitRuleMockery context =
        new JUnitRuleMockery();
    @Test
    public void testStorage() {
        final Supplier supplier =
            context.mock(Supplier.class);
        Storage storage = new Storage();
        storage.addSupplier(supplier);
        context.checking(new Expectations() {
            {oneOf (supplier).getData();}
        });
        result = storage.getSupply();
        context.assertIsSatisfied();
        assertEquals(result, expected);
    }
    private static final String expected = "supplied: ";
    private String result;
}

```

У лістингу 1.5 анотацією `@RunWith(JMock.class)` вказується, що виконання тесту має здійснюватися на основі методів класу `JMock`. На основі класу `JUnitRuleMockery` одержується контекст – підґрунтя для створення mock-об'єктів згідно інтерфейсу `Supplier` за рахунок використання методу `mock`. За рахунок використання методу `checking` перевіряємо твердження (наше припущення) щодо можливого сценарію застосування створеного mock-об'єкту (виклик методу `getData`). Методом `assertIsSatisfied` перевіряємо істинність припущення, а методом `assertEquals` – рівність результату роботи тестованого модуля `getSupply` очікуваному результату `expected`.

1.2 Порядок виконання роботи

1.2.1 Створення *unit*-тестів та проведення непараметризованого тестування.

1.2.1.1 Створити в *NetBeans 8.0 IDE* новий *Java*-проект: меню *Файл > Створити проект > Java > Java додаток*. В якості назви проекту вказати *Lab1.1*. В проекті створити модульні тести (натиснувши правою клавішею миші на створеному проекті): меню *Файл > Створити файл > Модульні тести > Модульний тест JUnit*. В якості назви пакету вказати *test*, а в якості назви створюваного класу – *Tester*. У результаті виконання зазначених дій створений проект буде містити пакет *test* із класом *Tester* – шаблоном модульного тесту. Вміст класу-шаблону слід відредагувати у відповідності до лістингу 1.6.

```
//Лістинг 1.6. Клас, що містить модульний тест
package test;
import org.junit.Test;
import static org.junit.Assert.*;
public class Tester {
    @Test
    public void test() {
        Calc calc = new Calc(3,3);
        assertEquals(calc.getResult(), 9, 0);
    }
}
```

В лістингу 1.6 наведено модульний тест, представлений методом *test* класу *Tester*, про що свідчить відповідна анотація *@Test*. У тілі методу *test* створюється екземпляр класу *Calc*, що містить тестований метод *getResult*. Шляхом використання *assertEquals*-методу у тілі методу *test* перевіряємо, чи рівний результат роботи методу 9, – для вхідних даних, переданих через конструктор класу *Calc*.

1.2.1.2 Підключити архів *junit-4.11.jar* до створеного проекту.

1.2.1.3 Створити тестований модуль. Для цього необхідно в пакеті вихідних кодів створити пакет *calc*, що буде містити клас *Calc* із тестованим модулем *getResult*. Клас *Calc* наведено в лістингу 1.7.

//Лістинг 1.7. Клас із тестованим модулем

```
package calc;
public class Calc {
    public Calc(int x, int y) {
        a = x;
        b = y;
    }
    public int getResult() {
        return a*b;
    }
    private int a, b;
}
```

1.2.1.4 Імпортувати клас Calc до класу Tester із модульним тестом:

```
import calc.Calc;
```

1.2.1.5 Здійснити непараметризоване *unit*-тестування в середовищі *NetBeans 8.0*. Для цього слід запустити на виконання файл класу Tester (*Shift+F6*).

1.2.1.6 Замінити рядок

```
assertEquals(calc.getResult(), 9, 0);
```

класу Tester рядком

```
assertEquals(calc.getResult(), 8, 1);
```

Після того – рядком

```
assertEquals(calc.getResult(), 10, 0.5);
```

Пояснити вплив значення третього параметру методу `assertEquals` – `delta` – на результати тестування.

1.2.1.7 Створити умови, при яких результати проведення модульного тесту будуть залежати також і від супутніх тестуванню часових витрат. Для цього слід замінити анотацію `@Test` параметризованим аналогом `@Test(timeout=...)`, де замість ... слід задати обмеження (в мілісекундах) на час виконання тесту. Слід також доповнити тіло методу `getResult` класу `Calc` штучно введеною часовою затримкою, шаблон реалізації якої наведено в лістингу 1.8.

```
//Лістинг 1.8. Шаблон часової затримки
try {
```

```

        Thread.sleep(some_delay);
    }
    catch(InterruptedException e) {
        e.printStackTrace();
    }
}

```

В лістингу 1.8 аргумент `some_delay` методу `sleep` являє собою або безпосередньо значення затримки в мілісекундах, або ініціалізовану цим значенням змінну.

1.2.1.8 Поекспериментувати зі значеннями аргументу `sleep`-методу класу `Calc` та `timeout`-аргументу параметризованої `@Test`-анотації для модульного тесту класу `Tester`. Розглянути при цьому три випадки: `some_delay > timeout`; `some_delay < timeout` та `some_delay == timeout`. Прокоментувати одержувані при цьому лістинги про результати *unit*-тестування. Лістинги та коментарі до них навести у звіті про виконання лабораторної роботи.

1.2.1.9 **Варіант 1.** Перевірити рівність двох масивів символьних значень шляхом використання *assert*-методу. Один з масивів має бути полем класу `Calc`, а інший – полем класу `Tester`.

1.2.1.10 **Варіант 2.** Перевірити рівність двох масивів цілих значень шляхом використання *assert*-методу. Один з масивів має бути полем класу `Calc`, а інший – полем класу `Tester`.

1.2.1.11 **Варіант 1.** В пакеті `calc` створити клас, що містить метод обчислення факторіалу від `n` (тестований модуль). В пакеті `test` створити клас, що містить модульний тест для зазначеного методу. Провести *unit*-тестування методу обчислення факторіалу та продемонструвати результати тестування.

1.2.1.12 **Варіант 2.** В пакеті `calc` створити клас, що містить метод одержання чисел Фібоначчі (0,1,1,2,3,5,...). В якості єдиного аргументу методу передається кількість чисел, що слід згенерувати. В пакеті `test` створити клас, що містить модульний тест для зазначеного методу. Провести *unit*-тестування методу одержання чисел Фібоначчі та продемонструвати результати тестування.

1.2.2 Параметризованим *unit*-тестуванням.

1.2.2.1 Провести параметризоване тестування методу встановлення, що задане число є простим – методу `definePrime` класу `Prime` пакету `prime` (лістинг 1.9). Відповідний

параметризований модульний тест наведено в лістингу 1.10. Проведення параметризованого *unit*-тестування замість непараметризованого дозволить перевірити зазначений метод на множині вихідних даних.

```
//Лістинг 1.9. Клас тестованого модуля
package prime;
public class Prime {
    public boolean definePrime(int n) {
        prime = n;
        for(int i=2; i<prime/2; ++i) {
            if(prime%i == 0)
                return false; //число не є простим
        }
        return true; //число є простим
    }
    private int prime;
}
```

```
//Лістинг 1.10. Реалізація параметризованого unit-тесту
package test;
import prime.Prime;
import org.junit.runner.*;
import org.junit.runners.*;
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.Arrays;
import java.util.Collection;
@RunWith(Parameterized.class)
public class PrimeTester {
    public PrimeTester (int n, boolean expected) {
        prime = new Prime();
        this.n = n;
        this.expected = expected;
    }
    @Parameterized.Parameters
    public static Collection getNumbers() {
        return Arrays.asList(new Object[][] {
            {1, true},
            {3, true},
            {6, false},

```

```

        {7, true}
    });
    @Test
    public void test() {
        assertEquals(expected, prime.definePrime(n));
    }
    private int n;
    private boolean expected;
    private Prime prime;
}

```

Зверніть увагу, що у лістингу 1.10 формат елементів списку-результату роботи статичного методу `getNumbers` визначається сигнатурою конструктора класу `PrimeTester`, що містить модульний тест. Формат елементів списку варто трактувати наступним чином: спочатку зазначається ціле число, для якого ми встановлюємо, чи є воно простим – число передається у якості аргументу методу `definePrime` класу `Prime`; потім зазначається наше власне припущення щодо того, чиє число простим (`true` – є, `false` – не є). Множину булевих значень, що фігурують у якості складових елементів списку (тип `Collection`) у відповідності до формату конструктора класу `PrimeTester`, варто розглядати як множину зразкових значень, яким мають дорівнювати результати роботи тестованого методу `definePrime` на множині вихідних значень `{1,3,6,7}`.

1.2.2.2 Провести параметризоване тестування на основі лістингу 1.10, замінивши елемент списку `{6, false}` елементом `{6, true}`. Прокоментувати результат тестування. Коментарі до результатів навести у звіті.

1.2.2.3 Варіант 1. Створити у класі `Prime` метод встановлення парності (непарності) цілого числа. Створити відповідний модульний тест та виконати параметризоване тестування методу.

1.2.2.4 Варіант 2. Створити у класі `Prime` метод обчислення залишку від ділення одного цілого числа на інше. Створити відповідний модульний тест та виконати параметризоване тестування методу.

1.2.2.5 Варіант 1. В пакеті `prime` створити клас `Gray`, в якому реалізувати метод обчислення коду Грея для цілих чисел від 0 до

$2^n - 1$, де $n = 3$ – кількість двійкових розрядів коду. Створити відповідний модульний тест та виконати параметризоване тестування методу.

1.2.2.6 Варіант 2. В пакеті `prime` створити клас `Nsd`, в якому реалізувати метод обчислення найбільшого спільний дільника (НСД) для двох цілих чисел. Створити відповідний модульний тест та виконати параметризоване тестування методу.

Рекомендації до виконання завдань 1.2.2.5 і 1.2.2.6 наведено в додатку А.

1.2.2.7 Здійснити *unit*-тестування методу `getSupply` (лістинг 1.3) на основі лістингу 1.5. Пояснити результат тестування.

1.2.2.8 Створити модульний тест для методу `getSupply`, в якому би не використовувалися *mock*-об'єкти. Для цього потрібно також створити клас, що реалізує інтерфейс `Supplier` (лістинг 1.2).

1.2.3 Тестування у пакетному режимі (*suite*-тестування).

1.2.3.1 На основі модульних тестів (*unit*-тестів), одержаних при виконанні завдань згідно варіантів, здійснити *suite*-тестування. Прокоментувати результати тестування у пакетному режимі.

Зразок класу, що містить *suite*-тест, наведено в лістингу 1.11.

```
//Лістинг 1.11. Клас TestSuite із suite-тестом
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({           // suite-тест
    class_1.class,              // клас із unit-тестами
    class_2.class               // клас із unit-тестами
})
public class TestSuite {}
```

1.3 Зміст звіту

1.3.1 Титульний лист.

1.3.2 Мета роботи.

1.3.3 Лістинги кодів параметризованих *unit*-тестів згідно варіанту та коментарі до них. Лістинг коду результуючого *suite*-тесту.

1.3.4 Відповіді на контрольні питання.

1.4 Контрольні питання

1.4.1 Специфіка *TDD*-підходу до розробки програмних систем.

1.4.2 Призначення та принципи модульного тестування.

1.4.3 Відмінні риси бібліотеки модульного тестування *JUnit*.

1.4.4 Поняття модульного тесту та тестованого модуля.

Послідовність дій при створенні модульного тесту.

1.4.5 Анотації *JUnit*. Призначення *assert*-методів бібліотеки *JUnit*. Приклади використання.

1.4.6 Призначення *mock*-об'єктів.

1.4.7 Прокоментувати результат тестування (failed або passed) стосовно рядка коду лістингу 1.6 `assertEquals(calc.getResult(), 9, 0);`. Чи буде результат таким самим у випадку заміни наведеного рядка рядком `assertEquals(calc.getResult(), 8, 1);`?

1.4.8 Обґрунтувати доцільність використання анотації `@Test(timeout=var)` замість анотації `@Test`.

1.4.9 Призначення параметризованого тестування. Навести приклад.

1.4.10 Доцільність *suite*-тестування. Навести приклад.

2 ЛАБОРАТОРНА РОБОТА №2

ЦЕНТРАЛІЗОВАНА СИСТЕМА КЕРУВАННЯ ВЕРСІЯМИ *SUBVERSION*

Мета роботи – набути навичок роботи з централізованою системою контролю версій *Subversion*.

2.1 Теоретичні відомості

Для організації колективної роботи над програмним проектом розробниками, як правило, використовується певна система контролю версій. Призначення такої системи – дати змогу названому колективу здійснити за потреби повернення до певної попередньої версії проекту. Передумовами до цього можуть бути, зокрема, помилки, виявлені в результаті внесення змін до проекту.

Існує чимало різноманітних систем контролю версій. Найпоширеніші із них – *Subversion* (SVN) та *Git*. Названі системи є демонстраціями двох принципово різних підходів до здійснення контролю версій: *SVN* – централізованого, а *Git* – децентралізованого підходу [6]. Кожен із підходів має як переваги, так і недоліки. *Git*, наприклад, характеризується більшими (у порівнянні із *SVN*) витратами обчислювальних ресурсів (пам'яті), проте його децентралізованість сприяє стійкості до виходу із ладу окремих обчислювальних вузлів мережі із розміщеними на них версіями проекту, використовуваних колективом розробників.

Дана робота присвячена централізованій системі контролю версій *SVN* [7]. Централізованість полягає у виділенні окремого вузла – серверу, на якому розміщується репозитарій проекту. Проект при цьому розглядається як деревовидна структура, елементами якої є файли та каталоги. На клієнтських вузлах при цьому зберігаються копії (робочі копії) репозитарію.

Використання *SVN* дає можливість відслідковувати історію внесення змін до файлів, каталогів розробниками та, за потреби (виникнення помилок), виконувати скасування внесених змін.

Розробником *SVN* є компанія *CollabNet*. В даній роботі використовувалась версія *Subversion 1.8.10*. Актуальну версію можна завантажити за посиланням <https://subversion.apache.org/>.

Базові поняття *SVN*:

– *репозитарій* – центр збереження інформації, розміщений на сервері; контент репозитарію (файли, каталоги) представлений у вигляді деревовидної структури; репозитарій можна розглядати як різновид файлового сервера;

– *робоча копія (PK)* – копія репозитарію, розміщена на клієнтському вузлі. Робочу копію можна розглядати як локальне робоче середовище клієнта.

PK – дерево каталогів із набором файлів [8]. Кожна *PK* містить службовий каталог *.svn*, що містить дані про зміни в *PK*.

Архітектуру *SVN*, що ґрунтується на поняттях *репозитарію* та *PK*, наведено на рис. 2.1.

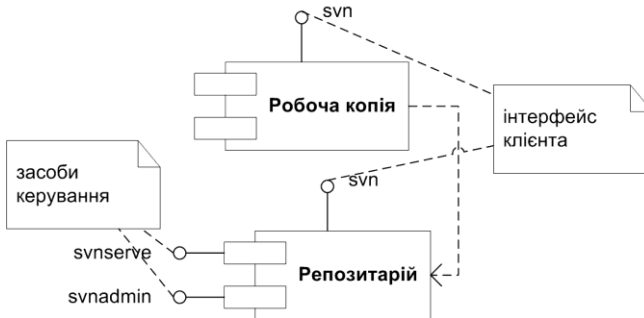


Рисунок 2.1 – Компоненти та утиліти SVN

Ключові утиліти *SVN* – *svn*, *svnserve*, *svnadmin* – представлено на рис. 2.1. Їх призначення роз’яснено в табл. 2.1. Користування наведеними утилітами можливе, наприклад, за рахунок інсталяції збірки *CollabNetSubversion-client-1.7.8-1-Win32*.

Таблиця 2.1 – Утиліти SVN

Утиліти	Призначення
<i>svn</i>	консольний інтерфейс клієнтського командного рядку;
<i>svnserve</i>	відкриття доступу до створеного репозитарію;
<i>svnadmin</i>	створення та налаштування репозитарію;
<i>svnversion</i>	одержання даних про поточну версію <i>PK</i> ;
<i>svnlook</i>	перегляд структури та вмісту репозитарію.

Утиліта *svnserve* – це спеціалізований сервер, що забезпечує доступ до репозитарію через протокол *TCP/IP*.

Взаємодія з репозитарієм можлива із використанням протоколів, наведених в табл. 2.2.

Таблиця 2.2 – Протоколи взаємодії з репозитарієм

Протоколи	Призначення
file:///	доступ до локального репозитарію;
svn://	спеціалізований спрощений протокол доступу до <i>svnserve</i> -репозитарію;
svn+ssh://	взаємодія із шифруванням трафіку;
http://	доступ з використанням протоколу <i>WebDAV</i> ;
https://	доступ на основі <i>SSL</i> -шифрування.

При виконанні роботи будуть використовуватися протоколи *file:///* та *svn://*.

Взаємодія з репозитарієм може здійснюватися згідно однієї з наступних моделей:

– *блокування/модифікація/розблокування* – одночасна взаємодія з репозитарієм декількох клієнтів (розробників) неможлива;

– *копіювання/модифікація/злиття* – на клієнтських вузлах створюються *РК* репозитарію; кожний із розробників працює із власною копією, якими потім модифікується репозитарій. Ця модель буде використовуватися в роботі.

Модель з блокуванням доречно використовувати при роботі із бінарними файлами (звукові, графічні), де неможливо об'єднувати конфліктні зміни [8].

Контроль версій в *SVN* здійснюється на основі віртуальної файлової системи, що дозволяє відслідковувати зміни ієрархічних структур каталогів у часі. Файлам та каталогам ставляться у відповідність множини метаданих (даних про дані), що дозволяє фіксувати відмінності поточної ревізії (версії) проекту від попередньої.

У випадку використання протоколу *svn://* (табл. 2.2) потрібно здійснювати запуск серверу *svnserve* – за допомогою однойменної утиліти. Правила взаємодії з репозитарієм при цьому регламентуються вмістом конфігураційного файлу *svnserve.conf*, розміщеного в каталозі *conf* репозитарію (лістинг 2.1).

```
//Лістинг 2.1. Вміст конфігураційного файлу репозитарію
[general]
anon-access = write
auth-access = write
password-db = passwd
realm = repository No1
```

Модифікація файлу *svnserve.conf* виконується шляхом зміни значень конфігураційних параметрів (табл. 2.3).

Таблиця 2.3 – Параметри конфігураційного файлу *svnserve.conf*

Параметри	Призначення
anon-access	регламентація прав доступу для неавторизованих користувачів: none – доступ заборонено; read – внесення змін заборонено; write – повний доступ.
auth-access	регламентація прав доступу для авторизованих користувачів. Допустимі значення ті самі.
password-db	задає назву файлу, що містить рядки вигляду user_name = password;
realm	унікальний ідентифікатор репозитарію, значенням якого є рядок.

Типовий сценарій роботи з репозитарієм можна представити наступним чином:

- створення репозитарію – `svnadmin create \path`, де `\path` – шлях до каталогу, в якому планується розмістити репозитарій;
- імпортування структури файлів та каталогів до створеного репозитарію – `svn import source destination -m mark`, де `source` – шлях до кореневого каталогу деревовидної структури файлів та каталогів, призначених для імпорту, `destination` – шлях до каталогу з репозитарієм із зазначенням одного з протоколів, наведених в табл. 2.2, `mark` – рядок, що унікальним чином ідентифікує здійснювану операцію;
- перегляд вмісту репозитарію, використовуючи команду `svn list` або `svnlook tree`. У першому випадку вміст буде виведено у

вигляді списку, у другому – у вигляді деревовидної структури. Шаблони використання названих команд наступні: `svn list file:///path` та `svnlook tree \path`, відповідно. Для команди `svn list`, на відміну від команди `svnlook tree`, слід вказувати абсолютний шлях до репозитарію;

- опціональний запуск серверу `svnserve`, що дасть змогу надалі користуватися спеціалізованим протоколом доступу `svn://`. Формат відповідної команди наступний: `svnserve -d -r \path`;

- створення *РК* репозитарію, що дасть змогу організувати взаємодію з останнім згідно моделі *копіювання/модифікація/злиття*. Для цього використовується команда `svn checkout` або команда `svn co`. Наприклад, наступним чином можна створити *РК*, яка міститиметься у каталозі *repo_copy*, з використанням протоколу `svn://`:

```
>>svn co svn://127.0.0.1/E:/.../repo E:\...\repo_copy;
```

- модифікація *РК*;

- внесення змін із *РК* до репозитарію – виконання операції *злиття*, коли до репозитарію додаються нові файли та каталоги, зокрема здійснюється заміщення файлів та каталогів новими ревізіями. Для цього виконується команда `svn commit`, разом із зазначенням рядка-ідентифікатора операції злиття. Наприклад, команда `svn commit -m "1st_commit"` здійснює злиття *РК* та репозитарію, що ідентифікується рядком "1st_commit".

Модифікація *РК* полягає у виконанні наступних дій (таблиця 2.4):

- оновлення *РК* до актуальної версії репозитарію – командою `svn update`. Потреба в цьому ґрунтується на колективному використанні репозитарію – коли певний користувач здійснює `commit`-операцію, інші *РК* перестають бути актуальними;

- внесення змін до *РК* – за рахунок виконання наступних команд: `svn add`, `svn delete`, `svn copy`, та `svn move`;

- перегляд змін в *РК* – команди `svn status` та `svn diff`;

- виправлення помилок, зроблених при внесенні змін до *РК* – команда `svn revert`;

- вирішення локальних конфліктів файлів та каталогів, обумовлених внесенням змін до *РК* – команда `svn resolve`;

Таблиця 2.4 – Деякі команди модифікації *РК*

Команди	Призначення
svn add	додавання нового файлу (каталогу) до <i>РК</i> .
svn diff	порівняння двох версій <i>РК</i> . Наприклад: svn diff -r1:2 – визначити відмінність 1-ї версії від 2-ї. Додані рядки файлу позначатимуться символом +, а видалені – символом –.
svn info	переглянути інформацію про поточну версію <i>РК</i> .
svn revert	скасувати внесені зміни.
svn status	одержання станів файлів та каталогів <i>РК</i> . Станам відповідають такі прапорці: A (<i>added</i>) – файл (каталог) додано; D (<i>deleted</i>) – файл (каталог) видалено; M (<i>modified</i>) – файл (каталог) модифіковано; R (<i>replaced</i>) – файл (каталог) замінено. ...

2.2 Порядок виконання роботи

2.2.1 Робота з репозитарієм та контроль версій.

2.2.1.1 Перейти до власного робочого каталогу, де створити нову директорію *logs* – для збереження лістингів про виконання роботи.

2.2.1.2 Відкрити *Far Manager*, за допомогою якого перейти до власного робочого каталогу, де переглянути інформацію про версію *SVN* та доступні команди:

```
>>svn help > 1.02_svn_help.log
```

Зауваження: >> – символ запиту командного рядка, що не підлягає набору.

2.2.1.3 Знаходячись у власному робочому каталозі, створити новий репозитарій *repo*:

```
>>svnadmin create repo
```

2.2.1.4 Переглянути вміст створеного однойменного каталогу *repo*. Замістити вміст конфігураційного файлу *conf\svnserve.conf* вмістом лістингу 2.1.

2.2.1.5 Імпортувати до репозитарію *JUnit*-проект, створений при виконанні попередньої лабораторної роботи. Для цього слід скористатися протоколом `file:///` згідно наступного шаблону:

```
>>svn import f:\path\to\project file:///f:/path/to/repo
-m "first import" > logs\1.05_import.log
```

2.2.1.6 Переглянути лістинг про результати імпорту проекту до репозитарію у створеному файлі *import_log.txt*. У випадку успіху лістинг міститиме інформацію про файли та каталоги імпортованого проекту, додані до репозитарію. Лістинг має завершуватись рядком *Committed revision 1*, де *1* – початковий номер версії проекту, що надалі буде інкрементуватися.

2.2.1.7 Переглянути вміст репозитарію проекту у вигляді деревовидної структури:

```
>>svnlook tree repo > logs\1.07_svnlook_tree.log
```

2.2.1.8 Переглянути вміст створеного репозитарію у вигляді списку. Для цього слід зазначати повний шлях до каталогу репозитарію, з використанням протоколу `file:///`:

```
>>svn list file:///f:/path/to/repo logs\1.08.log
```

або

```
>>svn list file://localhost/f:/path/to/repo
```

2.2.1.9 Виконати дії попереднього кроку 2.2.1.8 з використанням протоколу `svn://`. замість протоколу `file:///`:

```
>>svn list svn://127.0.0.1/f:/path/to/repo
```

Виникне помилка – неможливість підключення до репозитарію. Аби позбутися її, необхідно спочатку виконати крок 2.2.1.10.

2.2.1.10 Для використання протоколу `svn://`, що надасть змогу користуватися репозитарієм з іншого комп'ютера (наприклад для створення робочої копії) слід попередньо відкрити доступ до репозитарію:

```
>>svnserve -d -r f:\path\to\repo
```

Зауваження: для зазначеної команди `svnserve` зручно відкрити окреме консольне вікно, яке має залишатися відкритим упродовж подальшого виконання роботи.

За успішного відкриття доступу до репозитарію результат виконання завдання 2.2.1.9 буде ідентичним лістингові *1.8_log.txt*.

2.2.1.11 Занести до звіту інформацію про призначення прапорців `-d` та `-r` використаної команди `svnserve`.

2.2.1.12 У власному робочому каталозі на одному рівні ієрархії із каталогом *repo* створити нову директорію *repo_copy* – контейнер робочої копії.

2.2.1.13 Для роботи з репозитарієм згідно моделі *копіювання/модифікація/злиття*, створити робочу копію *repo_copy* репозитарію *repo* шляхом використання команди `svn checkout` або еквівалентної команди `svn co`:

```
>>svn checkout svn://localhost/f:/path/to/repo
f:\path\to\repo_copy > logs\1.13_repo_copy_log.txt
```

За успішного створення *PK* каталог *repo_copy* міститиме *JUnit*-проект, імпортований до репозитарію. У лістингу *1.12_repo_copy_log.txt* міститимуться дані про додані файли і каталоги проекту, на що вказуватимуть прапорці *A*. Завершальний рядок міститиме інформацію про поточну ревізію *PK* – *Checked out revision 1*, номер якої наразі співпадатиме із номером версії репозитарію. Подальші маніпуляції мають здійснюватися безпосередньо над *PK*.

2.2.1.14 Модифікувати конфігураційний файл *svnserve.conf* репозитарію (лістинг 2.1), замінивши значення `write` поля `anon-access` значенням `none`.

2.2.1.15 Спробувати створити нову *PK* *repo_copy_2*. Прокоментувати одержуваний при цьому лістинг.

Зауваження: для успішного виконання завдання 2.2.1.15 необхідно створити обліковий запис у файлі *repo\conf\passwd* у форматі *логін = пароль*.

2.2.1.16 Внести до конфігураційного файлу *svnserve.conf* зворотні зміни.

2.2.1.17 Створити в каталозі *test* робочої копії *repo_copy*, поряд із вже існуючим файлом *test\test\Tester.java*, новий файл модульного тесту *PiTester.java* (лістинг 2.2).

//Лістинг 2.2. Клас модульного тесту

```
package test;
import org.junit.Test;
import calc.*;
import static org.junit.Assert.*;
public class PiTester {
    @Test
```

```

public void test() {
    PiCalc calc = new PiCalc();
    assertEquals(calc.getResult(10), pi, 0);
}
public static final double pi = 3.14159265;
}

```

Призначення тесту – перевірка роботи методу `getResult()` обчислення значення π . Додати створений файл до *PK*:

```
>>svn add PiTester.java
```

Про успіх додавання свідчитиме прапорець *A* поряд із назвою доданого файлу: *A PiTester.java*.

2.2.1.18 Модифікувати файл *Tester.java*, додавши до нього *javadoc*-документацію перед визначенням класу *Tester*:

```

/**
 * unit test container for getResult() method
 */

```

2.2.1.19 Переглянути стан робочої копії *repo_copy*:

```
>>svn st -u -v repo_copy > logs\1.19_svn_st.log
```

У списку-лістингу навпроти доданого та модифікованого файлів *PK* зазначатимуться, відповідно, прапорці *A* та *M (modified)*.

2.2.1.20 Скопіювати вміст файлу *Tester.java* до нового файлу *PiTester.java*.

2.2.1.21 Видалити модульний тест *Tester.java* із *PK*:

```
>>svn del --force Tester.java
```

Індикатором успіху буде прапорець *D* поряд із назвою видаленого файлу, який з'явиться замість прапорця *M*: *D Tester.java*. При цьому файл буде видалено не лише із *PK*, а й взагалі.

2.2.1.22 Відновити втрачений файл *Tester.java*, скориставшись командою `svn revert`:

```
>>svn revert Tester.java
```

Видалений файл буде відновлено до початкового стану, визначеного або командою `svn co`, або останньою командою `svn update`. Це означає, що додана *javadoc*-документація буде відсутня.

2.2.1.23 Знову виконати завдання 2.2.1.21, після чого створити новий файл *Tester.java* із копією вмісту файлу *PiTester.java*, додавши

його до *PK*. При перегляді стану *PK* навпроти файлу *Tester.java* має з'явитися новий прапорець *R*, що означає *заміщений (replaced)*.

2.2.1.24 Здійснити злиття модифікованої *PK* із репозитарієм.

Для цього слід виконати команду `commit`:

```
>>svn commit --force-log -F *.java
```

В результаті злиття буде сповіщено про додавання нового файлу *PiTester.java* із робочої копії *repo_copy* до репозитарію *repo* та про заміщення вже існуючого файлу *Tester.java*. Номер версії репозитарію при цьому інкрементується: *Committed revision 2.*

2.2.1.25 Оновити раніше створену другу робочу копію *repo_copy_2* внесеними до репозитарію модифікаціями, перейшовши до однойменного каталогу та виконавши команду

```
>>svn update
```

У відповідному лістингу, попри інформацію про доданий файл, зазначатиметься також оновлена версія *PK*:

```
Updated to revision 2.
```

2.2.1.26 Переглянути детальнішу інформацію про поточну версію *PK repo_copy_2*:

```
>>svn info
```

2.2.1.27 Відновити попередню версію *PK* та переглянути інформацію про неї:

```
>>svn update -r1
```

```
>>svn info
```

2.2.1.28 Переглянути відмінності між версіями 1 та 2 *PK*. Для цього у власному робочому каталозі слід створити та виконати командний файл *1.28_svn_diff.cmd* із наступним контентом:

```
>>cd repo_copy
```

```
>>svn diff -r1:2 > f:\path\to\...\logs\1.27_diff.log
```

Результатом виконання командного файлу має бути вміст лістингу *1.27_diff.log*, де символом *+* позначатимуться додані рядки *javados*-документації завдання 2.2.1.18.

2.2.1.29 Змінити формат команди `svn diff` створеного командного файлу, у якості аргументу якої зазначити рядок `-r2:1` замість рядку `-r1:2`. Проаналізувати новий результат виконання командного файлу. Коментарі занести до звіту.

2.2.2 Робота з SVN у середовищі *NetBeans IDE*.

2.2.2.1 Запустити *NetBeans IDE 8.0*, де відкрити створений у попередній лабораторній роботі проект *Lab1.1*.

2.2.2.2 Імпортувати до відкритого проекту файл *PiTester.java* репозитарію *repo*. Для цього слід виконати наступне:

– у вкладці *Група* обрати позицію *Получить...* У вікні, що відкриється, слід вказати місцезнаходження створеного репозитарію *repo*:

```
svn://localhost/f:/path/to/repo
```

Зауваження: при цьому слід впевнитися, що сервер *svnserve* запущено і доступ до репозитарію *repo* відкрито.

– натиснувши елемент керування *Далее >*, слід вказати потрібний каталог *test* репозитарію (рис. 2.2); встановити прапорець навпроти рядка *Пропустить "test" и получит только содержимое*; при цьому потрібно також зазначити локальну (цільову) однойменну директорію проекту – для вивантаження вмісту вказаного каталогу репозитарію. Має бути рядок наступного формату:

```
f:\...\Lab1.1\test\test
```

– натиснути *Готово*;

Зауваження: наступні дії слід виконувати в середовищі *Far Manager*.

2.2.2.3 У каталозі *repo_copy* для модульного тесту *\test\test\PiTester.java* лістингу 2.2 створити тестований модуль *\src\calc\PiCalc.java* (лістинг 2.3).

```
//Лістинг 2.3. Клас тестованого модуля getResult()
package calc;
public class PiCalc {
    public double getResult(int n) {
        return 0;}
}
```

2.2.2.4 Додати створений файл-шаблон *PiCalc.java* до *PK repo_copy*, після чого виконати злиття *PK* із репозитарієм.

2.2.2.5 Реалізувати метод *getResult()* згідно варіанту. Аналогічно до завдання 2.2.1.28, одержати лістинг, в якому додані рядки тіла методу позначатимуться символом + (**варіант 1**) або – (**варіант 2**). Провести модульне тестування методу, здійснивши імпорт файлу *PiCalc.java* до середовища *NetBeans*.

Варіант 1. Обчислити π за формулою Дж. Валліса:

$$\pi = 2 \cdot \prod_{n=1}^{\infty} \frac{(2n)^2}{(2n-1) \cdot (2n+1)} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \dots$$

Варіант 1. Обчислити π за формулою Готфріда Лейбніца:

$$\pi = 4 \cdot \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

2.3 Зміст звіту

2.3.1 Титульний лист.

2.3.2 Мета роботи.

2.3.3 Результат виконання завдання 2.2.1.11, коментарі до завдання 2.2.1.15, призначення прапорців завдання 2.2.1.19 та коментарі до завдання 2.2.1.29.

2.3.4 Відповіді на контрольні питання.

2.4 Контрольні питання

2.4.1 Дати визначення поняттям *репозитарій* та *робоча копія*.

2.4.2 Охарактеризувати моделі контролю версій.

2.4.3 Навести та прокоментувати архітектуру *SVN*.

2.4.4 Призначення утиліт *svn.exe*, *svnadmin.exe* та *svnserve.exe*.

2.4.5 Призначення та параметри конфігураційного файлу *svnserve.conf*.

2.4.6 Прокоментувати протоколи доступу до репозитарію.

2.4.7 Команди `svn commit` та `svn checkout`. Призначення. Приклади.

2.4.8 Команда `svn status`. Прапорці станів файлів та каталогів.

2.4.9 Команди `svn list` та `svnlook tree`. Призначення. Приклади.

2.4.10 Послідовність кроків для імпорту файлів та каталогів з репозитарію з використанням *NetBeans IDE*.

3 ЛАБОРАТОРНА РОБОТА №3 ТЕХНОЛОГІЯ JAX-WS

Мета роботи – набути навичок роботи з *API JAX-WS (Java API for XML Web Services)*.

3.1 Теоретичні відомості

Сучасні розподілені програмні системи, призначені для комерційного застосування шляхом реалізації бізнес-процесів підприємств, розробляються, як правило, на основі принципів сервіс-орієнтованої архітектури. Відповідно до цих принципів у якості компонентів названих систем фігурують веб-сервіси або сервіси. Варто відзначити, що на поточному етапі розвитку технологій та засобів створення сервісів та систем на їх основі все суттєвіше виокремлюється тенденція до спрощення та автоматизації безпосередньо процесу розробки сервісів. Актуальним та поширеним засобом для цього є технологія *JAX-WS*, яку також можна розглядати як набір *API (Application Programming Interface)* та анотацій для створення та використання веб-сервісів в *Java* [9, 10]. *JAX-WS* є складовою платформи *Java EE*, відомої раніше як *J2EE*. Ця платформа призначена для створення надійних та масштабованих корпоративних програмних систем.

JAX-WS-сервіс по суті є *Java*-класом, доповненим *JAX-WS*-анотаціями, призначеними спростити процес створення та використання сервісу. Такий клас, у свою чергу, містить *Java*-методи, призначення яких – формування результату виклику сервісу клієнтом з використанням останнім відповідного інтерфейсу. *JAX-WS*-анотації наведено в табл. 3.1.

Таблиця 3.1 – Анотації *JAX-WS*

Анотації	Призначення
@WebService	позначення класу або інтерфейсу <i>Java</i> як сервісу;
@WebMethod	позначення методу, що входить до складу сервісу;
@WebParam	налаштування параметрів методів сервісів;
@WebResult	керування іменами повідомлень, що є результатами роботи методів, помічених як @WebMethod.

Для використання анотацій, наведених в табл. 3.1, необхідно імпортувати однойменні класи пакету `javax.jws`: `WebService`, `WebMethod`, `WebParam` та `WebResult`. Типовий шаблон інтерфейсу *JAX-WS*-сервісу наведено в лістингу 3.1.

```
//Лістинг 3.1. Інтерфейс сервісу
package servicepack;
import javax.jws.WebService;
import javax.jws.WebMethod;
@WebService public interface ServiceInterface {
    @WebMethod public int method1(String s);
    @WebMethod public String method2();}
```

Для пошуку сервісів в *Internet* та організації їх взаємодії використовують мову опису сервісів *WSDL* (*Web Services Description Language*) на основі *XML*. Для організації доступу до створеного сервісу останній необхідно розмістити (*deploy*) на сервері додатків (*Application Server*), прикладами яких, зокрема, є *Apache Tomcat*, *Oracle GlassFish Server*, *JBoss Application Server*. В роботі будемо використовувати сервер *Oracle GlassFish* за наступних причин:

- є складовим компонентом *NetBeans 8.0 IDE*;
- містить вбудовані засоби тестування сервісів.

Сервер додатків варто розглядати як програмну платформу (віртуальну машину), що дозволяє здійснювати керування сервісами.

Для надання сервісів клієнту використовується протокол *SOAP* (*Simple Object Access Protocol*) на основі *XML*, що є засобом організації взаємодії інтерфейсу клієнта із окремим сервісом (рис. 3.1).

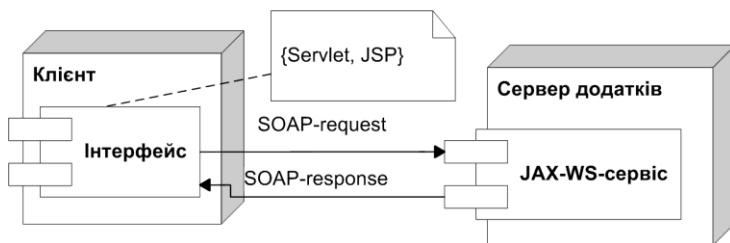


Рисунок 3.1 – Схема взаємодії клієнта із *JAX-WS*-сервісом, розміщеним на сервері додатків

На рис. 3.1 наведено наступне:

- для реалізації інтерфейсу клієнта може бути використана *Servlet*- або *JSP*-технологія (*Java Server Pages*). Ці технології дозволяють одержувати веб-сторінки із статичним та динамічним контентом;

- *JAX-WS*-сервіс розміщуються на сервері додатків (у нашому випадку – на сервері *GlassFish Server 4*);

- обмін інформацією між інтерфейсом клієнта та сервісом здійснюється у форматі *SOAP*-повідомлень. Інтерфейсом клієнта відправляється запит на одержання сервісу (*SOAP-request*), в якому містяться значення аргументів методу сервісу. Результат виклику методу сервісу відправляється у зворотному напрямі у форматі повідомлення *SOAP-response*.

Із посиланням на сказане варто зазначити, що наведені в табл. 3.1 анотації можуть бути параметризованими. Наприклад, використання анотації `@WebService` із атрибутом `serviceName` – `@WebService(serviceName = "someName")` – дозволить замінити у *WSDL*-описі сервісу фактичну назву класу сервісу на псевдонім (*alias*) – значення атрибуту `serviceName`. Використання такої (параметризованої) анотації дозволить в окремих випадках зробити процес користування сервісом більш інтуїтивним. Формат анотації `@WebService` не обмежується лише атрибутом `serviceName`. Через кому, наприклад, або окремо можна (варто) зазначити атрибут `endpointInterface`, що дасть змогу створити посилання з окремого класу *JAX-WS*-сервісу, що містить реалізації методів, на відповідний інтерфейс сервісу. Це означає, що із заданим інтерфейсом сервісу маємо змогу асоціювати різні класи, привласнюючи їм при цьому той самий псевдонім. Наприклад, використання анотації вигляду `@WebService(endpointInterface="servicepack.ServiceInterface")` перед визначенням класу сервісу здійснюється для посилання на інтерфейс `ServiceInterface` сервісу, розміщений в пакеті `servicepack`, згідно якого реалізується відповідний клас.

Псевдоніми можна привласнювати не лише сервісам, а й методам сервісів. Для цього анотацію `@WebMethod` використовують із аргументом `operationName`. Наприклад, якщо в інтерфейсі лістингу 3.1 перед прототипом методу `method1` використати анотацію

`@WebMethod(operationName = "foo")`, названий метод буде наданий клієнту як метод `foo`.

Використання анотацій `@WebParam` та `@WebResult` із параметром `name` дозволить створити псевдоніми для аргументів та результату роботи методу відповідно. При цьому анотацію `@WebResult` слід зазначати безпосередньо перед сигнатурою методу, а анотацію `@WebParam` – перед аргументом методу. Наприклад, для методу `method1` інтерфейсу лістингу 3.1 використання таких анотацій може бути наступним:

```
@WebMethod(operationName = "foo")
    @WebResult(name = "outputString")
    public int method1(@WebParam(name = "inputString")
String s);
```

Значення параметрів `name` анотацій `@WebParam` та `@WebResult` формують назви тегів повідомлень *SOAP-request* та *SOAP-response* відповідно.

Таким чином, відокремлення інтерфейсу сервісу від класу-контейнеру методів сервісу є запорукою зручності реконфігурування останнього, а використання розглянутих анотацій при цьому є засобом досягнення названої зручності.

Для інтерфейсу, наведеного в лістингу 3.1, відповідний клас *JAX-WS*-сервісу можна представити лістингом 3.2.

```
//Лістинг 3.2. Реалізація інтерфейсу ServiceInterface
package servicepack;
import javax.jws.WebService;
import java.util.Random;
@WebService(endpointInterface =
"servicepack.ServiceInterface")
public class SimpleService implements ServiceInterface {
    @Override
    public int method1(String s) {
        return Integer.bitCount(Integer.parseInt(s));
    }
    @Override
    public String method2() {
        return Integer.toBinaryString(
            new Random().nextInt(max));
    }
    private static final int max = 10;}

```

Наступним кроком є розміщення створеного сервісу на сервері додатків *GlassFish Server 4*, що входить до складу *NetBeans 8.0 IDE*. Для цього попередньо слід виконати запуск названого сервера: на вкладці *Службы* в групі *Сервери* обрати елемент списку *GlassFish Server 4*, натиск правою клавішею миші на якому дозволить виконати запуск серверу додатків.

Розміщення *JAX-WS*-сервісу на сервері додатків виконаємо з використанням спеціального класу із *main*-методом (лістинг 3.3). Для цього імпортуємо клас `javax.xml.ws.Endpoint` та використаємо статичний метод цього класу `publish(String address, Object implementor)`, де аргумент `address` – *URI* інтерфейсу сервісу із вказаним портом доступу (8080), а аргумент `implementor` – екземпляр класу сервісу, реалізованого згідно зазначеного у першому аргументі інтерфейсу сервісу.

//Лістинг 3.3. Засіб розміщення сервісу на сервері

```
package servicepack;
import javax.xml.ws.Endpoint;
public class ServicePublisher {
    public static void main(String[] args) {
        Endpoint.publish("http://localhost:8080/
            WebApplication2/ServiceInterface",
            new servicepack.SimpleService());    }}
```

У лістингу 3.3 формат *URI* інтерфейсу наступний: спочатку вказується *IP*-адреса або доменне ім'я, порт доступу (`http://localhost:8080/`); потім зазначається назва веб-проекту *NetBeans IDE* для *JAX-WS*-сервісів (`WebApplication2/`); потім – назва інтерфейсу сервісу (`ServiceInterface`).

У результаті запуску на виконання – *F6* – проекту, що містить інтерфейс та класи, наведені в лістингах 3.1 – 3.3 (при умові, що сервер додатків *GlassFish Server 4* вже запущено), у веб-браузері за адресою `http://localhost:8080/WebApplication2/` з'явиться веб-сторінка із наступним контентом: `TODO write content`. Така сторінка створюється автоматично при створенні веб-проекту *Java: Ctrl+Shift+N > Java Web > Веб-додаток*. Наведений контент є опосередкованим свідченням того, що створений сервіс розміщено на сервері додатків. Для доступу до сервісу (перегляду його *WSDL*-опису у веб-браузері) необхідно модифікувати адресу веб-сторінки

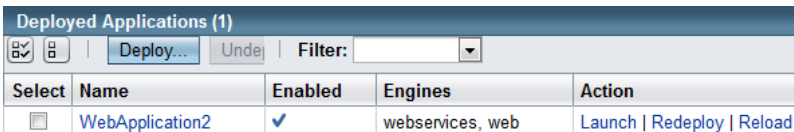
наступним рядком, в якому також слід вказати назву класу сервісу, доповнену ключовим словом *Service*:

– `http://localhost:8080/WebApplication2/SimpleServiceService?wsdl`

Фрагмент результату виконання такого запиту (фрагмент WSDL-опису JAX-WS-сервісу) наведено в лістингу 3.4.

```
//Лістинг 3.4. Фрагмент WSDL-опису сервісу
<definitions targetNamespace="http://servicepack/"
name="SimpleServiceService"> ...
<service name="SimpleServiceService">
<port name="SimpleServicePort"
binding="tns:SimpleServicePortBinding"><soap:address
location="http://localhost:8080/WebApplication2/SimpleServ
iceService"/></port></service></definitions>
```

Аби безпосередньо пересвідчитися, що створений сервіс розміщено на сервері додатків, можна скористатися портом 4848 та командою (адресою) `http://localhost:4848/common/index.jsf` – для переходу до панелі адміністрування серверу. На цій панелі у групі *Deployment* слід обрати елемент *List Deployed Applications*, що дозволяє перейти до панелі керування та перегляду статусу сервісу (рис. 3.2).



Deployed Applications (1)				
Select	Name	Enabled	Engines	Action
☐	WebApplication2	✓	webservices, web	Launch Redeploy Reload

Рисунок 3.2 – Панель керування сервісами

Для тестування сервісу можна скористатися вбудованими засобами серверу додатків *GlassFish Server 4*. Для цього у веб-браузері необхідно ввести наступну команду:

– `http://localhost:8080/WebApplication2/SimpleServiceService?Tester`

Змістове навантаження цієї команди полягає у використанні ключового слова *Tester*. Результатом успішної обробки команди буде динамічно згенерована веб-сторінка, що надає засоби тестування кожного із методів сервісу (рис. 3.3).

SimpleServiceService Web Service Tester

Methods :

```
public abstract void servicepack.ServiceInterface.method1(java.lang.String)
```

```
method1 (  )
```

```
public abstract java.lang.String servicepack.ServiceInterface.method2()
```

```
method2 ()
```

Рисунок 3.3 – Інтерфейс тестування сервісу

На рис. 3.3 у вигляді елементів керування представлено методи сервісу, сигнатури яких визначаються сигнатурами методів інтерфейсу, наведеного в лістингу 3.1. Наприклад, ввівши в якості аргументу методу `method1` сервісу значення 7, одержимо результат роботи методу (3 як кількість бітів, встановлених у значення 1) у форматі *SOAP*-повідомлення (лістинг 3.5).

```
//Лістинг 3.5. SOAP-повідомлення
//SOAP-запит із передачею аргументу
<S:Envelope ... >
  <SOAP-ENV:Header/>
  <S:Body>
    <ns2:method1 xmlns:ns2="http://servicepack/">
      <arg0>7</arg0>
    </ns2:method1>
  </S:Body>
</S:Envelope>
//SOAP-відповідь із результатом роботи методу
<S:Envelope ... >
  <SOAP-ENV:Header/>
  <S:Body>
    <ns2:method1Response
xmlns:ns2="http://servicepack/">
      <return>3</return>
    </ns2:method1Response>
  </S:Body>
</S:Envelope>
```

Альтернативним шляхом тестування сервісів є створення інтерфейсу клієнта. При цьому можна виділити дві варіації:

- створення консольного додатку;
- створення веб-додатку на основі *Servlet*- або *JSP*-технології.

За умови використання *NetBeans 8.0 IDE* процес створення інтерфейсу клієнта суттєво спрощується. Він полягає у наступному:

крок 1 - створюється новий проект – *Ctrl+Shift+N* > (*Java* > *Додаток Java*) або (*Java Web* > *Веб-додаток*);

крок 2 - слідом за натиском правої клавіші на створеному проекті треба обрати опцію *Новий* > *Клієнт веб-служби...* Призначенням такого кроку є одержання WSDL-опису цільового сервісу, згідно якого будуть згенеровані фрагменти коду інтерфейсу клієнта. При цьому можна виділити два шляхи:

– синтез WSDL-опису на основі класу сервісу – обирається опція *Проект*, де вказується клас проекту сервісу, на основі якого автоматично буде синтезовано WSDL-опис;

– одержання вже синтезованого WSDL-опису сервісу – обирається опція *Локальний файл* або опція *WSDL URL*;

крок 3 - у результаті успішного здійснення попереднього кроку у проекті інтерфейсу клієнта (консольного чи веб) буде створено каталог *Посилання на веб-служби* (рис. 3.4);

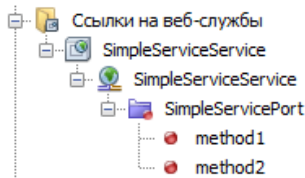


Рисунок 3.4 – Структура синтезованого контенту

крок 4 - переміщення (шляхом перетягування) елементів дерева ієрархії, наведеного на рис. 3.4, до тіла класу (у контексті консольного додатку), що знаходиться в каталозі *Пакети вихідних кодів* проекту інтерфейсу клієнта, дозволить згенерувати так звані *стаби* (заглушки) – методи-засоби виклику методів сервісу, які безпосередньо можна використовувати при створенні інтерфейсу клієнта. Стаб для методу *method1()* наведено в лістингу 3.6;

```
//Лістинг 3.6. Стаб методу сервісу
private static int method1(java.lang.String arg0) {
    servicepack.SimpleServiceService service = new
        servicepack.SimpleServiceService();
    servicepack.ServiceInterface port =
```

```
service.getSimpleServicePort();
return port.method1(arg0);}
```

крок 5 - використання одержаних стабів. Для випадку консольного інтерфейсу це можна здійснити наступним чином:

```
public static void main(String[] args) {
    System.out.println(method1(args[0]));}
```

В результаті виконання кроку 5 одержимо той самий результат, що і при користуванні вбудованими засобами серверу додатків *GlassFish Server 4*.

При створенні веб-інтерфейсу клієнта на основі *JSP*- або *Servlet*-технології є певні нюанси.

Для випадку *JSP*-технології в каталозі *Веб-сторінки* нового веб-проекту слід створити нову *JSP*-сторінку – *Новий > JSP...*, яку можна назвати як *index.jsp*. В цьому файлі буде автоматично згенеровано контекст, наведений в лістингу 3.7. З названого каталогу слід також видалити однойменну *HTML*-сторінку *index.html*.

```
//Лістинг 3.7. Незаповнена JSP-сторінка
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Вміст *JSP*-сторінки, як і у розглянутому випадку створення консольного інтерфейсу, необхідно доповнити аналогічним шляхом – перетягти елемент-представлення потрібного методу з ієрархічної структури (рис. 3.4) до області між тегами *<body>* та *</body>* *JSP*-сторінки. Відповідний стаб матиме вигляд, наведений в лістингу 3.8.

```
//Лістинг 3.8. Додатковий фрагмент коду JSP-сторінки
<%    try {
        servicepack.SimpleServiceService service = new
```

```

    servicepack.SimpleServiceService();
    servicepack.ServiceInterface port =
        service.getSimpleServicePort();
    java.lang.String arg0 = "";
    int result = port.method1(arg0);
    out.println("Result = "+result);
    } catch (Exception ex) {}

```

%>

Далі необхідно ініціалізувати змінну `java.lang.String arg0` значенням аргументу методу `method1` – рядком "7". Результат запуску клієнтського *JSP*-проекту буде тим самим, що і у випадку консольного додатку, але виводитися він буде у веб-браузері.

Для створення *Servlet*-інтерфейсу клієнта також необхідно створити веб-проект *Java: Ctrl+Shift+N > Java Web > Веб-додаток*:

– в каталозі *Пакети вихідних кодів* слід створити новий пакет із довільною назвою, в якому, в свою чергу, створити *Servlet*-клас: *Новий > Сервлет...*;

– до тіла синтезованого *Servlet*-класу слід перетягти елемент-представлення потрібного методу ієрархічної структури (рис. 3.4). Синтезований при цьому додатковий код сервлету наведено в лістингу 3.9;

```

//Лістинг 3.9. Додатковий контент сервлету
@WebServiceRef(wsdlLocation = "WEB-INF/wsdl/localhost_
    8080/WebApplication2/SimpleServiceService.wsdl")
private SimpleServiceService service;
...
private int method1(java.lang.String arg0) {
    servicepack.ServiceInterface port =
        service.getSimpleServicePort();
    return port.method1(arg0);
}

```

– коли шаблонний сервлет буде доповнений кодом, подібним до контенту лістингу 3.9, можна переходити до наступного кроку – виклику синтезованого методу-стабу `method1` в області між рядками `out.println("<body>");` та `out.println("</body>");` сервлету. Це можна зробити, шляхом додавання наступного рядка: `out.println("<h1>Result: " + method1("7") + "</h1>");`;

– запуск одержаного сервлету слід виконувати через опцію *Виконати файл (Shift+F6)*.

3.2 Порядок виконання роботи

3.2.1 Створення *JAX-WS*-сервісу із кількома методами та розміщення сервісу на сервері додатків *GlassFish Server 4*.

3.2.1.1 Створити новий веб-проект *Java: Ctrl+Shift+N > Java Web > Веб-додаток*. В проекті створити інтерфейс сервісу, що містить прототипи трьох методів (на прикладі лістингу 3.1). Прототипи задати згідно варіанту.

Варіант 1:

```
public int method11(String s11);
public String method12(String s12);
public String method13(String s13);
```

Варіант 2:

```
public String method21(String s21);
public boolean method22(String s22);
public String method23(String s23);
```

3.2.1.2 Створити клас *JAX-WS*-сервісу (на прикладі лістингу 3.2), в якому реалізувати методи інтерфейсу згідно варіанту.

Варіант 1:

- а) *method11* – повертає кількість одиниць у бінарному коді, переданому у вигляді рядка;
- б) *method12* – заміщує символи 'a' рядка символами '_';
- в) *method13* – обчислює бінарний код суми десяткових чисел, представлених у вигляді рядка. Наприклад, на основі рядка "123" має бути сформовано рядок "110";

Варіант 2:

- а) *method21* – повертає рядок символів у зворотній послідовності. Наприклад, на основі рядку "abc" має формуватися рядок "cba";
- б) *method22* – на основі числової послідовності, представленої у вигляді рядка, повертає булеве значення *true*, якщо сума двох перших десяткових цифр дорівнює сумі двох останніх десяткових цифр. В іншому випадку метод має повертати значення *false*. Наприклад, для рядка "13922" результатом роботи метода має бути значення *true*;
- в) *method23* – заміщує символну послідовність "12" у складі вхідного рядка послідовністю "(12)". Наприклад, для рядка "ab12cd" результатом роботи методу має бути рядок "ab(12)cd".

Зауваження: для реалізації методів згідно варіанту можна скористатися *Java*-методами, наведеними в *додатку Б*.

3.2.1.3 Створити параметризовані *unit*-тести для реалізованих методів сервісу, що підтверджують правильність роботи останніх, шляхом використання бібліотеки *JUnit*. На основі *unit*-тестів створити *suite*-тест, за допомогою якого виконати перевірку методів.

3.2.1.4 Створити клас (на прикладі лістингу 3.3), призначення якого – розміщення сервісу із трьома методами, реалізованими згідно варіанту, на сервері додатків *GlassFish Server 4*.

3.2.1.5 Протестувати методи *JAX-WS*-сервісу, розміщеного на сервері додатків, з використанням вбудованих засобів *GlassFish Server 4* та ключового слова *Tester*. Переконалися, що результати такого тестування співпадають із результатами, одержаними при проведенні *suite*-тестування.

Зауваження: при формуванні рядку-запиту до веб-браузера назва *Java*-класу сервісу має бути доповнена рядком *Service*, тобто, наприклад, для класу *SimpleService* відповідний рядок-запит матиме наступний вигляд:

```
– http://localhost:8080/projectname/  
SimpleServiceService?Tester
```

3.2.1.6 Переглянути *WSDL*-опис сервісу, що на сервері додатків.

3.2.1.7 Доповнити параметризовану анотацію *@WebService*, зазначену при визначенні класу сервісу, аргументом *serviceName*, ініціалізованим значенням псевдоніму сервісу. Заново розмістити сервіс на сервері додатків, після чого знову переглянути *WSDL*-опис сервісу, але вже з використанням псевдоніму. Рядки *WSDL*-опису, що при цьому зазнали змін, занести до звіту.

Зауваження: при формуванні рядку-запиту до веб-браузера із використанням псевдоніму сервісу вже не потрібно здійснювати конкатенацію псевдоніму із рядком *Service*.

3.2.1.8 В інтерфейсі класу сервісу для кожного із трьох методів замінити непараметризовані анотації *@WebMethod* на анотації із параметром *operationName*. Заново розмістити сервіс на сервері додатків та переглянути *WSDL*-опис сервісу. Рядки *WSDL*-опису, що при цьому зазнали змін, занести до звіту.

3.2.1.9 В інтерфейсі класу сервісу для кожного із трьох методів застосувати параметризовані анотації *@WebParam* та *@WebResult*.

Заново розмістити сервіс на сервері додатків та здійснити тестування методів із використанням ключового слова *Tester*. Занести до звіту рядки (теги) SOAP-повідомлень *SOAP-request* та *SOAP-response*, що зазнали змін у результаті використання зазначених анотацій.

3.2.2 Створення інтерфейсу клієнта для одержання останнім результатів роботи методів JAX-WS-сервісу, розміщеного на сервері додатків *GlassFish Server 4*.

3.2.2.1 Створити консольний інтерфейс клієнта, за допомогою якого викликати кожний із трьох методів сервісу.

3.2.2.2 Створити веб-інтерфейс клієнта, за допомогою якого викликати кожний із трьох методів сервісу.

Варіант 1. Створити інтерфейс за *Servlet*-технологією.

Варіант 2. Створити інтерфейс за *JSP*-технологією.

3.3 Зміст звіту

3.3.1 Титульний лист.

3.3.2 Мета роботи.

3.3.3 Результати виконання завдань 3.2.1.7, 3.2.1.8 та 3.2.1.9 із супутніми коментарями та поясненнями.

3.3.4 Відповіді на контрольні питання.

3.4 Контрольні питання

3.4.1 Технологія JAX-WS. Призначення.

3.4.2 Анотації *@WebService* та *@WebMethod*. Приклад.

3.4.3 Анотації *@WebParam* та *@WebResult*. Приклад.

3.4.4 Схема взаємодії клієнта із JAX-WS-сервісом.

3.4.5 Інтерфейс та класи, створені при виконання роботи. Призначення.

3.4.6 Призначення серверу додатків.

3.4.7 Тестування JAX-WS-сервісу вбудованими засобами серверу додатків *GlassFish Server 4*. Приклад.

3.4.8 Протокол SOAP. Формат SOAP-повідомлень.

3.4.9 Послідовність дій при створенні клієнтського веб-інтерфейсу на основі *Servlet*-технології.

3.4.10 Послідовність дій при створенні клієнтського веб-інтерфейсу на основі *JSP*-технології.

ЛІТЕРАТУРА

1. Ларман К. Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и итеративную разработку / К. Ларман; пер. с англ.; под ред. А.Ю. Шелестова. – [3-е изд.]. – М.: Вильямс, 2013. – 736 с.
2. Freeman S. Growing Object-Oriented Software, Guided by Tests / S. Freeman, N. Pryce. – NY.: Addison-Wesley, 2009. – 384 p.
3. Tahchiev P. JUnit in Action / P. Tahchiev, F. Leme, V. Massol, G. Gregory. – [2nd ed.]. – Stamford, CT, The USA: Manning Publications Co., 2010. – 504 p.
4. Package org.junit [Электронный ресурс] – Режим доступа: <http://junit.sourceforge.net/javadoc/org/junit/package-summary.html>. – Заголовок з екрану.
5. Гранд М. Шаблоны проектирования в Java / М. Гранд; пер. с англ. С. Беликовой. – М.: Новое издание, 2004. – 559 с.
6. Chacon S. Pro Git / Scott Chacon. – NY.: Apress, 2009. – 288 p.
7. Collins-Sussman B. Version Control with Subversion / B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato. – [2nd ed.]. – CA.: O'Reilly Media, Inc, 2008. – 432 p.
8. Управление версиями в Subversion [Электронный ресурс] – Режим доступа: <http://svnbook.red-bean.com/nightly/ru/>. – Заголовок з екрану.
9. Гонсалвес Э. Изучаем Java EE 7 / Э. Гонсалвес; пер. с англ. Е. Зазноба [и др.]. – СПб.: Питер, 2014. – 640 с.
10. Vohra, D. Java 7 JAX-WS Web Services: A practical, focused mini book for creating Web Services in Java 7 / D. Vohra. – Birmingham-Mumbai.: Packt Publishing, 2012. – 64 p.

Додаток А

Рекомендації до виконання завдань згідно варіантів

Таблиця А.1 – Рекомендації до завдання 1.2.2.5

№	Рекомендації
1	метод має повертати значення типу String – бінарний код Грея, представлений у вигляді символьної послідовності, одержуваної на основі цілого числа-аргументу методу;
2	для коду Грея кожне i -е число відрізняється від "сусідніх" $(i-1)$ -го та $(i+1)$ -го чисел значенням лише одного бінарного розряду. Наприклад, послідовність чисел 0, 1, ..., 7 у трьохрозрядному коді Грея буде мати вигляд "000", "001", "011", "010", "110", "111", "101", "100";
3	код Грея обчислюється за формулою $g_i = x_i \oplus x_{i+1}$, де x_i – бінарне представлення числа, що кодується, x_{i+1} – бінарне представлення числа, що кодується, зміщене на один розряд вправо, $\oplus$ – оператор "виключаюче або". Java-реалізацію зазначеної формули можна представити виразом <code>g_i = x_i ^ (x_i >> 1);</code> – для завдання формату коду Грея можна використовувати метод <code>format</code> класу <code>String</code> . Наприклад, щоб обмежити довжину деякої символьної послідовності (рядка) трьома символами для бінарних розрядів, зазначений метод можна використати так: <code>format ("%3s", str);</code>
4	при програмній реалізації методу одержання коду Грея в символьній послідовності, що починається з символу '0', здійснюється автоматичне заміщення останнього символом відступу '_'. Це ускладнює процедуру параметризованого тестування. Для запобігання цьому можна використати метод <code>replace</code> класу <code>String</code> . Наприклад, у результаті виклику методу <code>replace('a', 'b')</code> символ 'a' буде заміщено символом 'b'.

Таблиця А.2 – Рекомендації до завдання 1.2.2.6

№	Рекомендації
1	обчислення НСД можна виконати ітераційно – згідно алгоритму Евкліда: $f(x, y) = \begin{cases} x, & \text{при } y = 0, \\ f(y, x \bmod y), & \text{в інших випадках} \end{cases}$ де x та y – цілі числа.

Додаток Б
Java-методи класів String, StringBuilder та Integer
для роботи з рядками

Таблиця Б.1 – Методи класу String

Методи	Аргументи та призначення
<code>public int length()</code>	одержання кількості символів рядка;
<code>public char CharAt(int index);</code>	одержання символу рядка за індексом <code>index</code> (від 0 до <code>length() - 1</code>);
<code>public String replace(char old, char new);</code>	заміщення символу рядка <code>old</code> новим символом <code>new</code> .

Зручними засобами формування рядків є методи класу `StringBuilder` (табл. Б.2).

Таблиця Б.2 – Методи класу StringBuilder

Методи	Аргументи та призначення
<code>public StringBuilder append(char c);</code>	додавання символу <code>c</code> у кінець рядка;
<code>public StringBuilder delete(int id1, int id2);</code>	видалення елементів екземпляру класу <code>StringBuilder</code> у діапазоні індексів від <code>id1</code> (включно) до <code>id2</code> (виключно).

В табл. Б.3 наведено поширені методи класу `Integer`.

Таблиця Б.3 – Методи класу Integer

Методи	Аргументи та призначення
<code>static Integer valueOf(char c);</code>	одержання цілого числа із його символьного представлення <code>c</code> ;
<code>public static String toBinaryString(int i);</code>	одержання бінарного коду цілого числа <code>i</code> ;
<code>public static String toHexString(int i);</code>	одержання <i>HEX</i> -коду цілого числа <code>i</code> ;