Lucrare de laborator nr. 4

Introducere în testarea unit (Unit Testing)

Testarea unit (testarea modulară) verifică funcționarea părților individuale ale programului în izolare. Aceasta ajută la identificarea erorilor, simplifică depanarea și face codul mai rezistent la modificări. Testele bune trebuie să fie izolate (să nu depindă de factori externi), determinate (să ofere același rezultat la fiecare rulare), automatizate (fără intervenție manuală) și ușor de înteles (pentru a putea fi ușor de menținut).

Acoperirea cu teste (Code Coverage) este un indicator important care arată ce parte a codului este executată în timpul testării. Cu cât acoperirea este mai mare, cu atât este mai mică probabilitatea de erori nedetectate.

Exemplu de teste unitare în Java

Să analizăm un program simplu – un calculator care efectuează adunarea numerelor:

Cod principal (Calculator.java)

```
public class Calculator {
    // Metoda de adunare a două numere
    public int add(int a, int b) {
        return a + b;
    }
}
```

Această clasă conține metoda add, care primește două numere și returnează suma acestora.

Test unit (CalculatorTest.java)

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {
    @Test // Anotarea indică faptul că metoda este un test
    void testAddition() {
        Calculator calculator = new Calculator(); // Creăm obiectul
calculatorului
        int result = calculator.add(2, 3); // Apelăm metoda de
adunare
        assertEquals(5, result, "2 + 3 trebuie să fie 5"); //
Verificăm dacă rezultatul este 5
    }
}
```

Analiza testului:

- Se creează un obiect al clasei `Calculator`.
- Se apelează metoda `add(2, 3)`, care trebuie să returneze 5.
- Se folosește `assertEquals(expected, actual, message)`, care verifică dacă valoarea returnată corespunde celei așteptate. Dacă nu, testul va eșua.

Determinarea acoperirii testelor

Acoperirea codului prin teste arată câte linii de cod, metode sau ramuri ale condițiilor au fost executate în timpul testării. Unul dintre cele mai populare instrumente pentru măsurarea acoperirii este **JaCoCo**.

Integrarea JaCoCo cu Maven

Pentru a adăuga pluginul JaCoCo în fișierul `pom.xml`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.8</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Rularea testelor și generarea raportului de acoperire

Executați testele și generați raportul:

```
mvn test
mvn jacoco:report
```

Raportul va fi disponibil în `target/site/jacoco/index.html`.

JaCoCo analizează:

- **Acoperirea liniilor de cod (Line Coverage)** – câte linii de cod au fost executate.
- **Acoperirea metodelor (Method Coverage)** – câte metode au fost apelate.
- **Acoperirea ramurilor (Branch Coverage)** – câte ramificații logice (`if`, `switch`) au fost parcurse.

Avantajele și dezavantajele testării unitare

Avantaje:

- Detectarea timpurie a erorilor.
- Facilitarea refactorizării codului.
- Documentarea codului prin exemple de utilizare.
- Automatizarea verificării codului.
- Creșterea fiabilității aplicației.

Dezavantaje:

- Necesită timp suplimentar pentru scrierea testelor.
- Necesitatea de a menține testele când codul se modifică.
- Nu înlocuiește testele de integrare.
- Poate oferi un fals sentiment de siguranță.

Sarcina de realizat

Se cere dezvoltarea unui modul software în Java care să implementeze funcționalitatea specificată și să fie acoperit de teste unitare cu un nivel de acoperire de cel puțin 80% conform metricei **Line Coverage** din raportul JaCoCo.

Cerințe pentru cod și testele unitare:

- Respectarea principiilor arhitecturii curate și incapsulării.
- Implementarea unei clase cu funcționalitatea principală.
- Utilizarea JUnit 5 pentru testare.
- Verificarea acoperirii codului folosind JaCoCo.
- Cod ușor de citit și documentat.
- Utilizarea Maven sau Gradle pentru automatizarea testării.

Variante de implementare:

1. **Manager de sarcini (Task Manager)**
    a. Creare, ștergere și listare sarcini.
    b. Schimbarea statusului sarcinilor.
    c. Căutarea sarcinilor după nume.
2. **Calculator de expresii (Expression Calculator)**
    a. Operații matematice de bază (+, -, *, /).
    b. Tratarea împărțirii la zero.
    c. Evaluarea expresiilor (ex. `5 + 3 * 2`).
3. **Validarea parolelor (Password Validator)**
    a. Minim 8 caractere, o cifră, o literă mare și un simbol special.
4. **Convertor de temperatură (Temperature Converter)**
    a. Conversie Celsius → Fahrenheit: `F = C * 9/5 + 32`.
    b. Conversie Fahrenheit → Celsius: `C = (F - 32) * 5/9`.

Tehnologii utilizate:

- **Limbaj:** Java 11+
- **Framework de testare:** JUnit 5
- **Sistem de build:** Maven / Gradle
- **Acoperire cod:** JaCoCo