**Concepte Teoretice**

Testarea unitară reprezintă procesul de verificare a componentelor individuale ale unui software, cunoscute și sub denumirea de "unități", pentru a asigura că fiecare funcționează corect izolat de restul sistemului.

*1.1. Rolul testării unitare*

Testarea unitară este esențială pentru a identifica și remedia erori în stadii incipiente ale dezvoltării software. Prin testarea fiecărei unități (de obicei funcții sau metode individuale), dezvoltatorii se asigură că fiecare componentă își îndeplinește funcțiile așteptate și poate fi integrată în mod corespunzător cu alte părți ale sistemului.

*1.2. Platforme de testare unitară*

Există mai multe platforme și cadre (frameworks) populare pentru testarea unitară, fiecare fiind specifică unui anumit limbaj de programare. Printre cele mai cunoscute se numără:

- **JUnit** (pentru Java)
- **NUnit** (pentru .NET/C#)
- **PyTest** și **unittest** (pentru Python)
- **Mocha** și **Jest** (pentru JavaScript)
- **CppUnit** (pentru C++)

Aceste cadre oferă funcționalități pentru scrierea, organizarea și rularea testelor, precum și pentru raportarea rezultatelor.

*1.3. Caracteristicile testelor unitare*

- *Izolare*: Fiecare test unitar trebuie să fie izolat, fără dependențe de alte teste. Aceasta înseamnă că testele pot fi rulate în orice ordine și în mod independent.
- *Automatizare:* Testele unitare sunt de obicei automatizate, permițându-le să fie rulate rapid și eficient ori de câte ori codul este modificat.
- *Rapiditate:* Testele unitare sunt concepute pentru a fi rapide, deoarece verifică doar funcționalitatea unei mici părți din aplicație.
- *Deterministice*: Testele unitar trebuie să ofere același rezultat de fiecare dată când sunt rulate, dacă nu s-au făcut modificări la codul testat.

*1.4. Avantajele testării unitare*

- *Detectarea precoce a erorilor*: Testarea unitară permite identificarea problemelor imediat ce acestea apar, înainte de integrarea unității în sistemul complet.
- *Îmbunătățirea calității codului*: Prin scrierea de teste unitare, dezvoltatorii tind să scrie cod mai curat și mai bine structurat, facilitând întreținerea ulterioară.
- *Documentație vie:* Testele unitare servesc și ca o formă de documentare, arătând cum ar trebui să funcționeze fiecare componentă a software-ului.

- *Facilitarea refactorizării*: Cu testele unitar în vigoare, codul poate fi refactorizat cu mai multă încredere, deoarece dezvoltatorii pot verifica rapid că noile modificări nu au introdus erori.

*1.5. Dezavantajele testării unitare*

- *Cost inițial ridicat:* Scrierea testelor unitare necesită timp și efort, ceea ce poate încetini inițial procesul de dezvoltare.
- *Întreținerea testelor*: Testele unitar necesită întreținere constantă, în special atunci când codul este modificat frecvent.
- *Acoperire limitată*: Testele unitare verifică doar funcționalitatea individuală a unităților și nu asigură că unitățile funcționează corect atunci când sunt integrate. De aceea, sunt necesare și alte tipuri de teste, cum ar fi testarea de integrare și testarea sistemului.
- *Posibilitatea unui fals sentiment de securitate*: Există riscul ca, dacă testele unitare sunt incomplete sau necorespunzătoare, dezvoltatorii să creadă că codul este mai robust decât este în realitate.

**2.** Exemplu Practic

Context

Să presupunem că avem o clasă simplă numită Calculator, care oferă operații aritmetice de bază: adunare, scădere, înmulțire și împărțire.

```
public class Calculator {

    // Metodă pentru adunare
    public int adunare(int a, int b) {
        return a + b;
    }

    // Metodă pentru scădere
    public int scadere(int a, int b) {
        return a - b;
    }

    // Metodă pentru înmulțire
    public int inmultire(int a, int b) {
        return a * b;
    }

    // Metodă pentru împărțire
    public int impartire(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Nu se poate împărți la zero!");
```

```
            }
            return a / b;
        }
    }
```

### *Teste Unitare folosind JUnit*

Vom scrie teste unitare pentru a verifica funcționalitatea fiecărei metode din clasa *Calculator*.

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

public class CalculatorTest {

    Calculator calculator = new Calculator();

    @Test
    public void testAdunare() {
        // Verificăm dacă adunarea funcționează corect
        assertEquals(5, calculator.adunare(2, 3));
        assertEquals(-1, calculator.adunare(-2, 1));
        assertEquals(0, calculator.adunare(0, 0));
    }

    @Test
    public void testScadere() {
        // Verificăm dacă scăderea funcționează corect
        assertEquals(1, calculator.scadere(3, 2));
        assertEquals(-3, calculator.scadere(-2, 1));
        assertEquals(0, calculator.scadere(2, 2));
    }

    @Test
    public void testInmultire() {
        // Verificăm dacă înmulțirea funcționează corect
        assertEquals(6, calculator.inmultire(2, 3));
        assertEquals(0, calculator.inmultire(0, 5));
        assertEquals(-4, calculator.inmultire(2, -2));
    }

    @Test
```

```
    public void testImpartire() {
        // Verificăm dacă împărțirea funcționează corect
        assertEquals(2, calculator.impartire(4, 2));
        assertEquals(-3, calculator.impartire(9, -3));
        assertThrows(IllegalArgumentException.class, () -> calculator.impartire(5, 0));
    }
}
```

Explicația Testelor

- *testAdunare():* Verifică dacă metoda adunare returnează rezultatul corect pentru diferite perechi de numere.
- *testScadere():* Verifică dacă metoda scadere returnează rezultatul corect pentru operații de scădere.
- *testInmultire():* Verifică corectitudinea metodei inmultire, inclusiv cazul în care unul dintre multiplicatori este zero.
- *testImpartire():* Verifică dacă metoda impartire funcționează corect și dacă lansează o excepție atunci când se încearcă împărțirea la zero.

***3.*** *Activitate practică*

*Materialele Necesare*

- Calculator cu acces la internet.
- Un IDE care suportă testarea unitară (ex. IntelliJ IDEA, Visual Studio).
- Unelte de testare unitară (ex. JUnit, NUnit).
- Proiect software pentru testare.

*Sarcini practice*

- Alegeți un modul dintr-un proiect software și creați teste unitare pentru funcțiile sale.
- Rulați testele unitare și documentați rezultatele obținute.
- Discutați despre izolarea funcțiilor, utilizarea mock-urilor, și acoperirea codului.
- Evaluați robustețea și repetabilitatea testelor.
- Documentați avantajele testării unitare, cum ar fi identificarea timpurie a erorilor și facilitarea refactorizării.
- Analizați dezavantajele, inclusiv timpul de dezvoltare crescut și dificultatea de a testa funcționalități complexe.

**Concluzii**

Testarea unitară este o componentă crucială a procesului de dezvoltare software, care aduce numeroase avantaje în ceea ce privește calitatea și întreținerea codului. Cu toate acestea, trebuie să fie utilizată împreună cu alte tipuri de testare pentru a asigura funcționalitatea și stabilitatea sistemului în ansamblul său.