Testarea sistemelor software

Testare unitară în Java

Echipa A16:
Alexandrescu Marian 464
Gheorghiță Elena Raluca Lorena 464
Ghiță Filip Darius 464
Ionescu Lorena Elena 464
Urlățeanu Alexandru-Ioan 446

Cuprins:

1. **Testare unitară**

Testarea unitară este o practică esențială în domeniul testării software-ului, care constă în examinarea fiecărei componente individuale sau "unități" a unei aplicații în mod independent. Obiectivul acestei metode este să confirme funcționarea corectă a fiecărei unități și respectarea tuturor cerințelor și specificațiilor definite înainte de integrarea lor în aplicația finală.

De obicei, dezvoltatorii scriu testele unitare, axându-se pe verificarea funcționalității la nivelul cel mai mic posibil, cum ar fi o metodă sau o clasă. Aceste teste sunt automate și pot fi rulate rapid și eficient folosind unelte specializate pentru testarea software-ului.

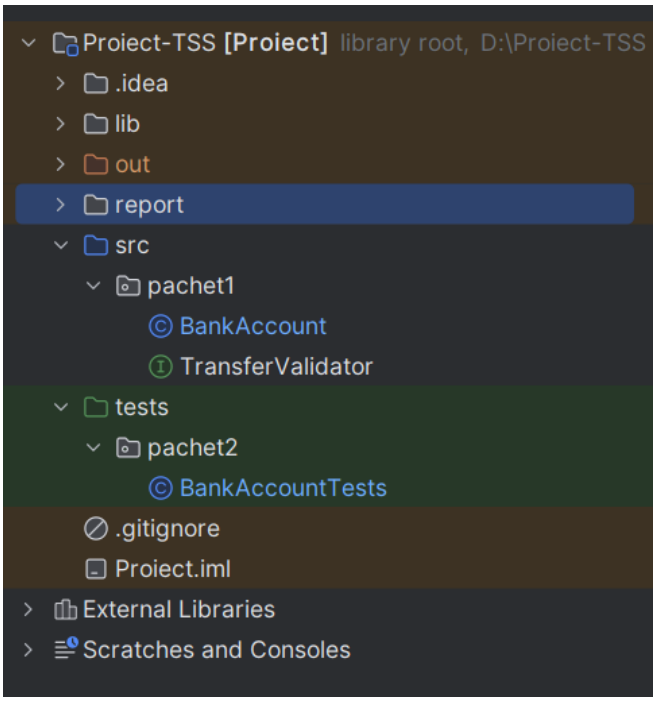
În diverse limbaje de programare, există numeroase framework-uri consacrate pentru testarea unitară, cum ar fi JUnit pentru Java, pytest pentru Python, sau Mocha pentru JavaScript. Aceste framework-uri furnizează dezvoltatorilor un set de funcționalități și metode pentru a scrie și a rula teste unitare, precum și pentru a genera rapoarte detaliate despre rezultatele acestora. Noi am ales să lucrăm cu framework-ul dedicat limbajului de programare Java.

Testarea unitară oferă numeroase beneficii, printre care identificarea și remedierea erorilor în fazele incipiente ale dezvoltării, îmbunătățirea calității și fiabilității software-ului, reducerea costurilor de întreținere și sporirea productivității dezvoltatorilor prin diminuarea timpului dedicat depanării problemelor. Aceasta reprezintă un pilon esențial în garantarea calității software-ului și este integrată în procesul de dezvoltare. Utilizarea unui framework de testare unitară poate contribui la asigurarea consistenței și eficienței în scrierea și rularea testelor, ceea ce conduce la îmbunătățirea calității produsului final și la reducerea timpului alocat pentru rezolvarea defecțiunilor.

Concepte de bază:

- **Test runner**: este un instrument care execută cazurile de testare și oferă un raport despre rezultatele acestora. Acestea sunt folosite pentru a automatiza procesul de rulare a testelor și validarea comportamentului codului.
- **Test case**: se referă la un scenariu sau o condiție specifică în care o bucată de cod este testată pentru a se asigura că se comportă conform așteptărilor. Cazurile de testare sunt scrise de obicei pentru a valida funcționalitatea metodelor, claselor sau componentelor unei aplicații.
- **Assertion**: este o afirmație care verifică dacă o condiție este adevărată sau falsă. În contextul testării, sunt folosite pentru a valida comportamentul așteptat al codului. La scrierea cazurilor de testare, aserțiunile sunt folosite pentru a confirma că rezultatul unei anumite operații se potrivește cu rezultatul așteptat.
- **Mock**: este un obiect simulat folosit pentru a imita comportamentul obiectelor reale, adesea dependențe precum baze de date sau servicii externe (ex: Mockito).

2. **Structura proiectului**



3. **Clasa BankAccount** este clasa principala care reprezintă implementarea unui cont bancar. Conține funcționalitățile de bază precum: autentificare, depozitare, transfer și retragere.
   3.1. Funcția Login reprezintă autentificarea folosind username si password, daca se încearcă de cel puțin trei ori conectarea cu cel puțin una dintre acestea greșite, contul se va bloca.

```
public void login(String username, String password) {
    if (isLocked) {
        transactionLog.add("Login attempted - Account is locked");
        return;
    }

    if ("user".equals(username) && "secret".equals(password)) {
        isAuthenticated = true;
        transactionLog.add("Login successful");
    } else {
        failedLoginAttempts++;
        transactionLog.add("Login attempted");
        if (failedLoginAttempts >= 3) {
            lockAccount();
        }
    }
}
```

3.2. Funcția deposit realizează depunerea unei sume în cont.

```
public void deposit(double amount, String source) {
    if (isAuthenticated && amount > 0) {
        balance += amount;
        transactionLog.add("Deposited: " + amount + " from " + source);
    }
}
```

3.3. Funcția withdraw realizează retragerea unei sume daca exista fondul suficiente.

```
public void withdraw(double amount, String reason) {
    if (isAuthenticated && amount > 0 && balance >= amount) {
        balance -= amount;
        transactionLog.add("Withdrew: " + amount + " for " + reason);
    } else {
        transactionLog.add("Failed withdrawal attempt");
    }
}
```

3.4. Funcția transfer îndeplinește trimiterea unei sume de bani in alt cont bancare, tranzacția este validata prin transferValidator.

```
public void transfer(BankAccount toAccount, double amount) {
    if (isAuthenticated && amount > 0 && balance >= amount &&
            transferValidator.validateTransfer(amount, fromAccount: this, toAccount)) {
        this.balance -= amount;
        toAccount.balance += amount;
        transactionLog.add("Withdrew: " + amount + " for Transfer to " + toAccount.accountNumber);
        toAccount.transactionLog.add("Deposited: " + amount + " from Transfer from " + this.accountNumber);
    } else {
        transactionLog.add("Failed transfer attempt");
    }
}
```

```
package pachet1;

import pachet1.BankAccount;

public interface TransferValidator {
    boolean validateTransfer(double amount, BankAccount fromAccount, BankAccount toAccount);
}
```

4. **Clasa BankAccountTests**

   Cuprinde teste scrise folosind frameworkul JUnit pentru testarea unitara si biblioteca Mockito pentru mock-uri si simularea comportamentului dependințelor.

   Cuprinde teste precum:

   - partiționare în clase de echivalenta
   - analiza valorilor de frontiera
   - acoperire la nivel de instrucțiune, decizie, condiție
   - circuite independente

   4.1. Teste Partiționare în clase de echivalenta

   - **testLoginSuccess** verifică succesul autentificării
   - **testLoginFailure** veridică ce se întâmpla când autentificarea nu se realizează

```
@Test
public void testLoginSuccess() {
    account.login( username: "user",  password: "secret");
    assertTrue(account.isAuthenticated());
    assertEquals(List.of( e1: "Login successful"), account.getTransactionLog());
}

@Test
public void testLoginFailure() {
    account.login( username: "testUsername",  password: "wrongpassword");
    assertFalse(account.isAuthenticated());
    assertEquals( expected: 1, account.getFailedLoginAttempts());

}
```

   - **testDepositWhenAuthenticated** verifică dacă depozitele sunt gestionate corect în starea autentificată
   - **testDepositWhenNotAuthenticated** verifică dacă depozitele sunt gestionate corect atât în starea neautentificată.

```
@Test
public void testDepositWhenAuthenticated() {
    account.login( username: "user",  password: "secret");
    double amount = 100;
    account.deposit(amount,  source: "Salary");
    assertEquals(amount, account.getBalance(),  delta: 0.01);
    assertEquals(List.of("Login successful", "Deposited: " + amount + " from Salary"),
            account.getTransactionLog());
}
@Test
public void testDepositWhenNotAuthenticated() {
    double amount = 100;
    // Authentication not happening before deposit
    account.deposit(amount,  source: "Salary");
    assertEquals( expected: 0, account.getBalance(),  delta: 0.01);
    assertEquals(List.of(), account.getTransactionLog());
}
```

4.2. Analiza valorilor de frontiera

testDepositBoundary, testWithdrawBoundary verifică comportamentul sistemului la valorile minime pozitive și zero, cum ar fi depozitarea unei sume foarte mici sau retragerea întregului sold.

```
@Test
public void testDepositBoundary() {
    account.login( username: "user",  password: "secret");
    account.deposit( amount: 0.01,  source: "Salary"); // Add deposit so
    assertEquals( expected: 0.01, account.getBalance(),  delta: 0.001);
    account.deposit( amount: -0.01,  source: "Salary"); // Deposit of ne
    assertEquals( expected: 0.01, account.getBalance(),  delta: 0.001);
}

@Test
public void testWithdrawBoundary() {
    account.login( username: "user",  password: "secret");
    account.deposit( amount: 100,  source: "Salary"); // Add deposit sou
    account.withdraw( amount: 100,  reason: "Groceries"); // Add withdra
    assertEquals( expected: 0, account.getBalance(),  delta: 0.01);
    account.withdraw( amount: 0.01,  reason: "Rent"); // Add withdraw re
    assertEquals( expected: 0, account.getBalance(),  delta: 0.01); // Ba
}
```

4.3. Acoperire la nivel de instrucțiune, decizie, condiție: toate testele asigură o acoperire bună a instrucțiunilor și deciziilor prin verificarea diferitelor ramuri de cod ale metodelor testate.

- **testLoginSuccess** si **testWithdrawalWithSufficientFunds** teste cu acoperire la nivel de instrucțiune
- **testAccountLockAfterThreeFailedAttempts** acoperă ramurile de cod asociate cu blocarea contului după eșecuri repetate, acoperire la nivel de condiție

```
@Test
public void testAccountLockAfterThreeFailedAttempts() {
    account.login( username: "testUsername",  password: "wrong");
    account.login( username: "testUsername",  password: "wrong");
    account.login( username: "testUsername",  password: "wrong");
    assertFalse(account.isAuthenticated());
    assertTrue(account.isLocked());
    assertEquals(List.of("Login attempted", "Login attempted",
            "Login attempted", "Account locked"), account.getTransactionLog());
}
```

- **testTransferWhenValidatorAllows** și **testTransferWhenValidatorDenies,** testează condițiile în care validatorul permite sau nu transferurile, acoperind deciziile logice.

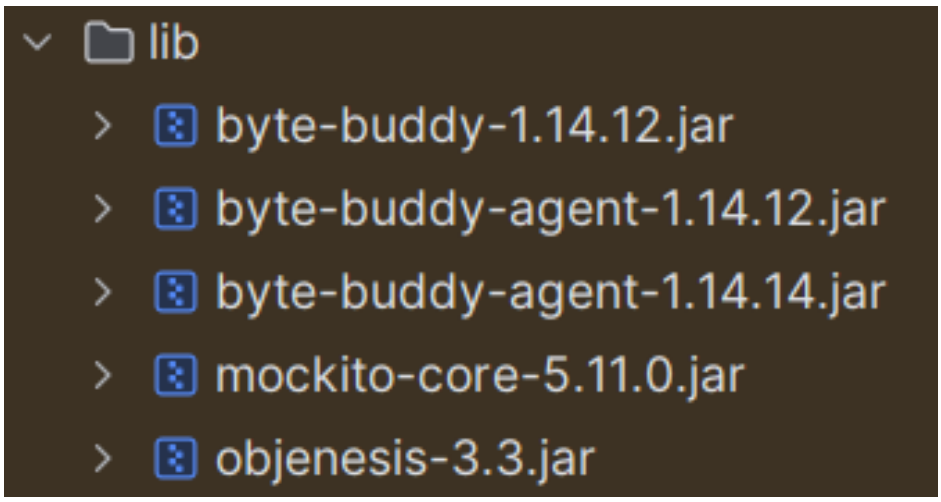
```
@Test
public void testTransferWhenValidatorAllows() {
    when(mockValidator.validateTransfer(anyDouble(), any(BankAccount.class), any(BankAccount.class)))
            .thenReturn( t: true);
    account.login( username: "user",  password: "secret");
    BankAccount anotherAccount = new BankAccount( accountNumber: "654321",  username: "anotherUser", mockValidator);
    anotherAccount.login( username: "anotherUser",  password: "secret");

    account.deposit( amount: 200.0,  source: "Salary");
    account.transfer(anotherAccount,  amount: 100);
    assertEquals( expected: 100, account.getBalance(),  delta: 0.01);
    assertEquals( expected: 100, anotherAccount.getBalance(),  delta: 0.01);
    verify(mockValidator).validateTransfer( amount: 100, account, anotherAccount);
}
```

```
@Test
public void testTransferWhenValidatorDenies() {
    when(mockValidator.validateTransfer(anyDouble(), any(BankAccount.class), any(BankAccount.class)))
            .thenReturn( t: false);
    account.login( username: "user",  password: "secret");
    BankAccount anotherAccount = new BankAccount( accountNumber: "654321",  username: "anotherUser", mockValidator);
    anotherAccount.login( username: "anotherUser",  password: "secret");
    account.deposit( amount: 200.0,  source: "Initial deposit");
    account.transfer(anotherAccount,  amount: 100);
    assertEquals( expected: 200.0, account.getBalance(),  delta: 0.01);
    assertEquals( expected: 0, anotherAccount.getBalance(),  delta: 0.01);
    verify(mockValidator).validateTransfer( amount: 100, account, anotherAccount);
}
```

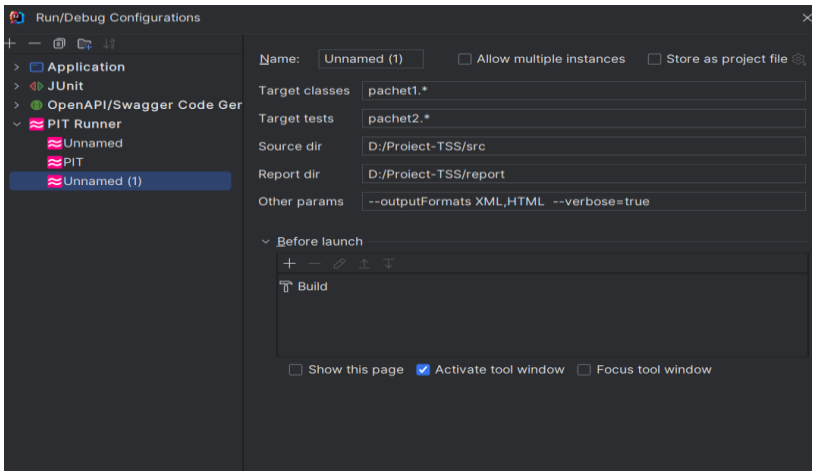
4.4. Circuite independente: s-au realizat folosind mock-uri care permite testarea clasei BankAccount fără a depinde de implementarea reala a validatorului de transferuri.

- **testTransferWhenValidatorAllows** și testTransferWhenValidatorDenies: acestea folosesc mockValidator astfel permite izolarea clasei de logica validării transferurilor si asigura răspunderea corecta a clasei.
- Pentru realizarea acestor teste s-a instalat mock.



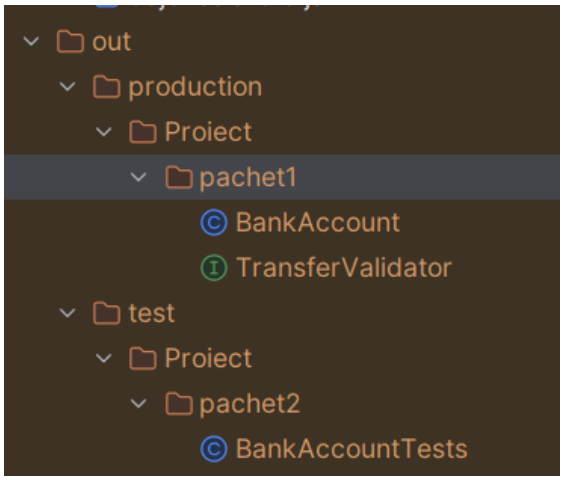
5. **Raport creat de generatorul de mutanți**

Pentru crearea raportului s-a instalat și configurat Pit Runner

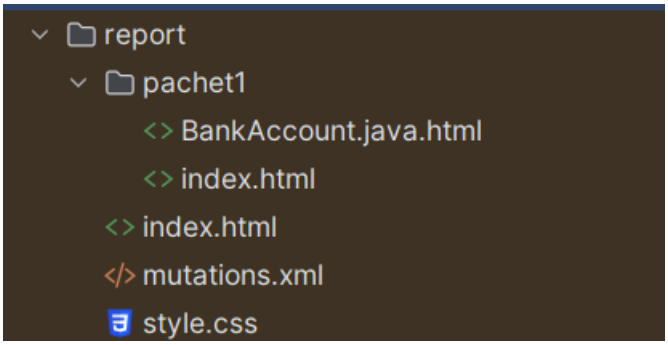


împreuna cu JUnit5

Și a rezultat clase cu mutanții



pe baza cărora s-a produs următorul raport:



6. **Interpretare rezultate Pit Runner**
   Raportul realizat cu Pit Runner oferă o lista de mutanți activi:
   - CONDITIONALS_BOUNDARY - modifica condițiile de frontieră în expresiile condiționale
   - EMPTY_RETURNS - modifica metodelor care returnează valori cu instrucțiuni care returnează valori implicite sau nule
   - FALSE_RETURNS -modifica metodele care returnează valori booleane pentru a returna doar false
   - INCREMENTS -modifica operațiunile de incrementare in decrementare
   - INVERT_NEGS - inversează semnul valorilor numerice
   - MATH - modifica expresiile matematice
   - NEGATE_CONDITIONALS -neaga condițiile
   - NULL_RETURNS - modifica metodele care returnează obiecte pentru a returna null
   - PRIMITIVE_RETURNS - modifică valorile returnate de metodele cu tipuri de date primitive
   - TRUE_RETURNS - modifica metodele care returnează valori booleane pentru a returna doar true.
   - VOID_METHOD_CALLS – elimina apelurile către metodele care nu returnează valori
7. **Teste suplimentare pentru a omori mutanți**
   - **testAccountRemainsLockedAfetMultipleFailedAttempts -** omoară mutanții implicați în autentificarea și blocarea contului, verifică dacă contul a fost blocat după cele trei încercări și asigură că nicio autentificare ulterioară nu este posibilă.

```java
@Test
public void testAccountRemainsLockedAfterMultipleFailedAttempts() {
    // Trei încercări eșuate de logare pentru a bloca contul
    account.login( username: "testUsername",  password: "wrong");
    account.login( username: "testUsername",  password: "wrong");
    account.login( username: "testUsername",  password: "wrong");
    assertTrue( message: "Contul ar trebui să fie blocat după trei încercări eșuate.", account.isLocked());

    // Încercări suplimentare care ar trebui să fie ignorate deoarece contul este blocat
    account.login( username: "user",  password: "secret");
    assertFalse( message: "Autentificarea nu ar trebui să fie posibilă dacă contul este blocat.",
            account.isAuthenticated());
    assertTrue( message: "Contul ar trebui să rămână blocat.", account.isLocked());

    // Verifică logul pentru a asigura că încercările suplimentare sunt înregistrate corespunzător
    List<String> expectedLog = List.of(
            "Login attempted", "Login attempted", "Login attempted", "Account locked",
            "Login attempted - Account is locked"
    );
    assertEquals( message: "Jurnalul tranzacțiilor nu corespunde așteptărilor după încercări " +
            "multiple de autentificare.", expectedLog, account.getTransactionLog());
}
```

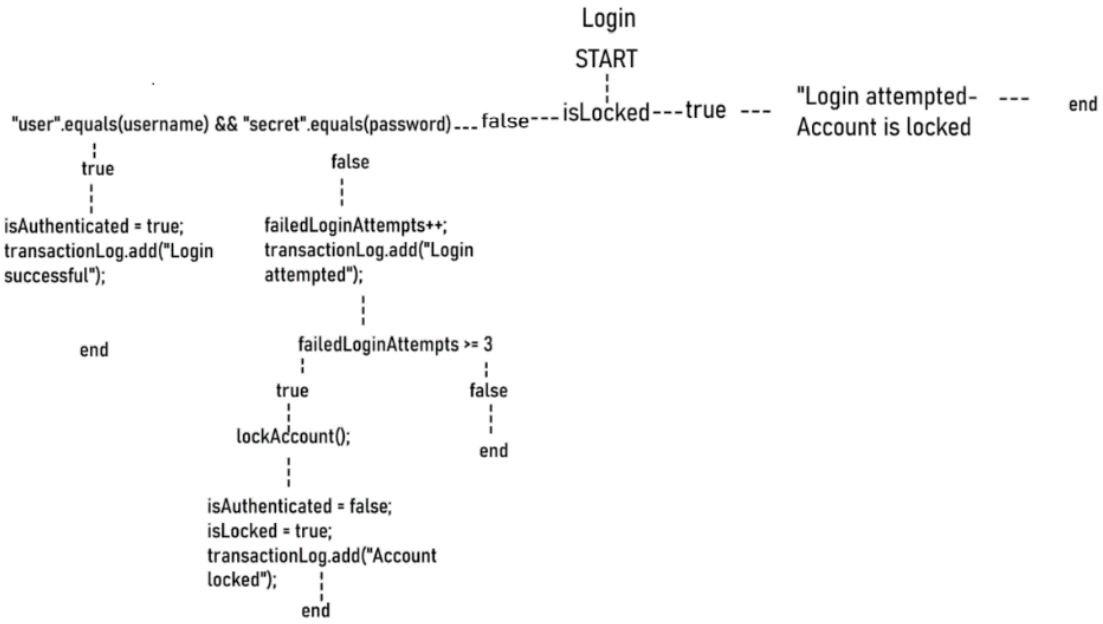
- **testWithdrawExactBalance** - verifică retragerea corectă a întregii sume disponibile, confirmă schimbările soldului care duc la omorârea mutanților care modifică logica matematică

```java
@Test
public void testWithdrawExactBalance() {
    account.login( username: "user",  password: "secret");
    account.deposit( amount: 100,  source: "Initial");
    // Încearcă să retragi exact soldul contului
    account.withdraw( amount: 100,  reason: "Exact Balance");
    assertEquals( message: "Soldul ar trebui să fie 0 după retragerea întregului sold",
            expected: 0, account.getBalance(),  delta: 0.01);
    // Verifică că retragerea este înregistrată corect în jurnalul de tranzacții
    assertTrue( message: "Jurnalul tranzacțiilor ar trebui să includă retragerea efectuată",
            account.getTransactionLog().contains("Withdrew: 100.0 for Exact Balance"));
}
```

Mutations

18	1. replaced int return with 0 for pachet1/BankAccount::getFailedLoginAttempts → KILLED
28	1. negated conditional → KILLED 2. negated conditional → KILLED
32	1. Replaced integer addition with subtraction → KILLED
34	1. negated conditional → KILLED 2. changed conditional boundary → KILLED
35	1. removed call to pachet1/BankAccount::lockAccount → KILLED
40	1. replaced boolean return with true for pachet1/BankAccount::isAuthenticated → KILLED 2. replaced boolean return with false for pachet1/BankAccount::isAuthenticated → KILLED
44	1. replaced boolean return with true for pachet1/BankAccount::isLocked → SURVIVED 2. replaced boolean return with false for pachet1/BankAccount::isLocked → KILLED
60	1. negated conditional → KILLED 2. changed conditional boundary → SURVIVED 3. negated conditional → KILLED
61	1. Replaced double addition with subtraction → KILLED
67	1. negated conditional → KILLED 2. changed conditional boundary → KILLED 3. negated conditional → KILLED 4. changed conditional boundary → SURVIVED 5. negated conditional → KILLED
68	1. Replaced double subtraction with addition → KILLED
76	1. changed conditional boundary → SURVIVED 2. negated conditional → KILLED 3. negated conditional → KILLED 4. negated conditional → KILLED 5. changed conditional boundary → SURVIVED
77	1. negated conditional → KILLED
78	1. Replaced double subtraction with addition → KILLED
79	1. Replaced double addition with subtraction → KILLED
89	1. replaced double return with 0.0d for pachet1/BankAccount::getBalance → KILLED
93	1. replaced return value with Collections.emptyList for pachet1/BankAccount::getTransactionLog → KILLED

8. **CFG pentru funcția login**



9. **Verificarea setului de teste structurale**

S-a folosit Pit Runner care oferă o analiza detaliata asupra acoperirii testelor structurale si a rezultat:

- **Line Coverage** - procentul este de 100%, toate liniile de cod din clasa principala au fost rulate cel puțin o data in timpul testării.
- **Mutation Coverage** - procentul este de 84%, majoritatea mutațiilor introduse de Pit au fost detectate de teste
- **Test Strenght** - procentul este de 84%, testele au o buna capacitatea de a detecta schimbările in comportamentul codului

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	100%	49/49	84%	26/31	84%	26/31

Breakdown by Package

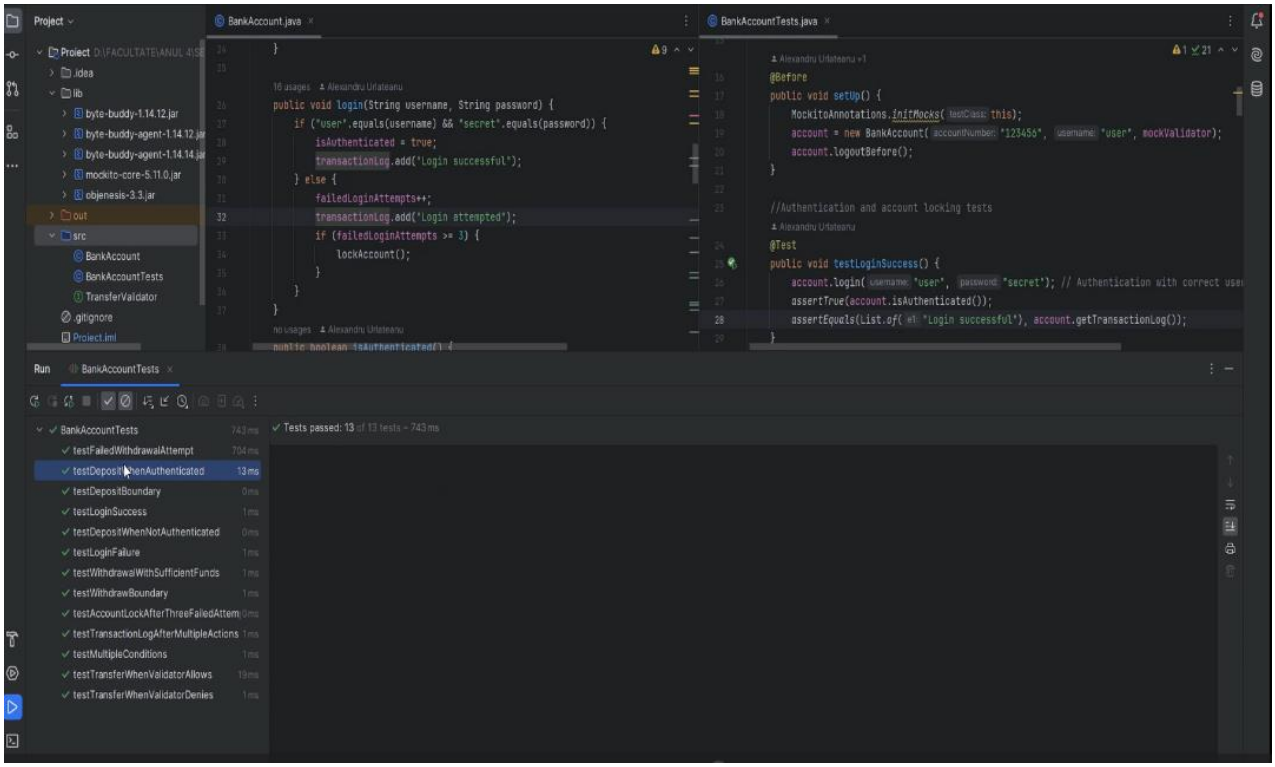
Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
pachet1	1	100%	49/49	84%	26/31	84%	26/31

Report generated by PIT 1.15.8

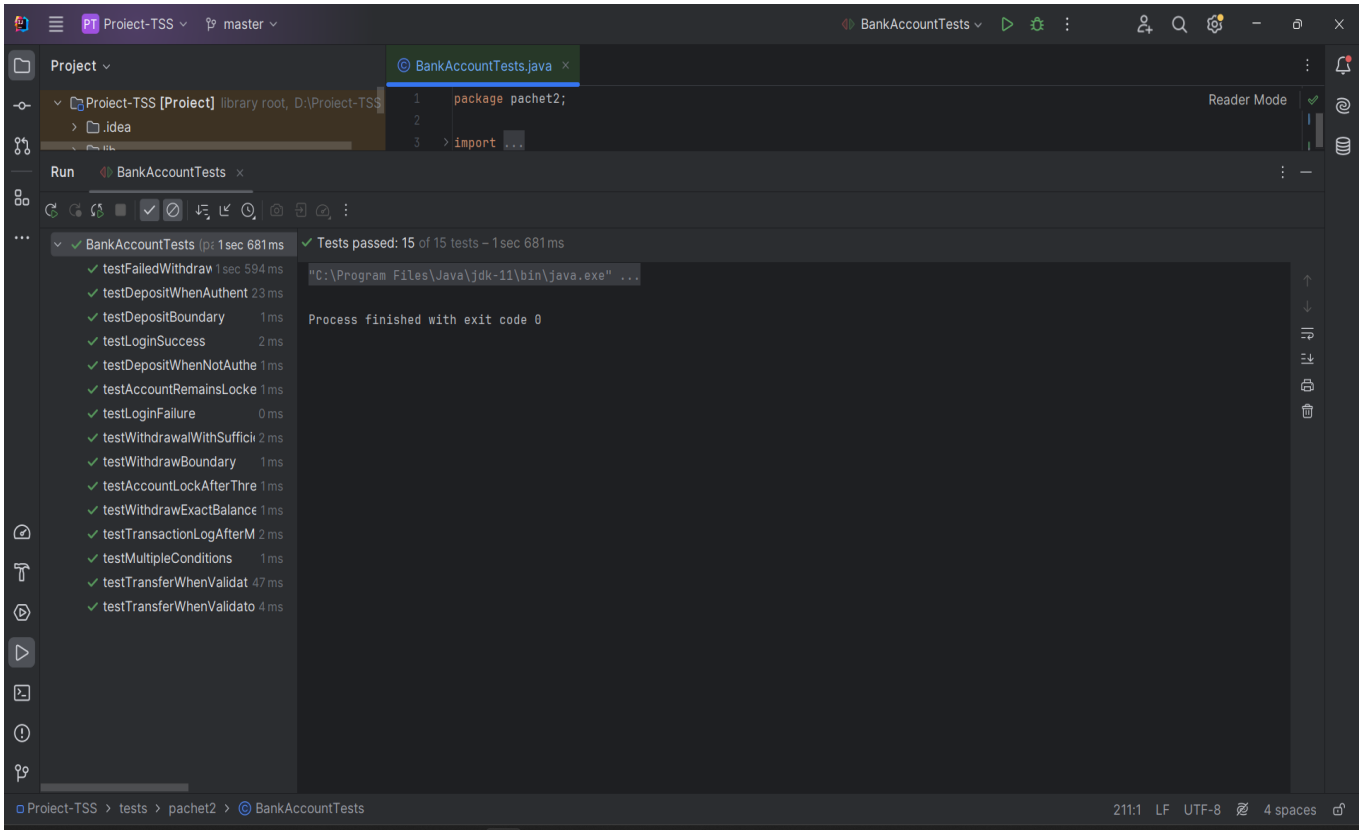
Enhanced functionality available at arcmutate.com

10. **Demonstrație a funcționalității testelor înainte de mutație si după aceasta**

Funcționalitatea testelor înainte de mutație:



Funcționalitatea testelor după mutație:



11. **Raportul utilizării unui tool de AI**

Pentru a realiza testarea cu ajutorul inteligenței artificiale, am folosit ChatGPT. Acesta este un chatbot dezvoltat de OpenAI și este bazat pe modele lingvistice mari (LLM).
În prompt, i-am oferit chat-ului codului pe care să facă teste și i-am zis să scrie teste unitare pentru clasa BankAccount. Acesta ne-a furnizat 12 teste, dintre care 5 teste eșuate și de asemenea lipsește raportul și testele pentru a omorî mutanții.
Pentru o funcționalitate mai bună și pentru a acoperi cerințele echipei i-am oferit drept exemplu testele realizate de către echipă.

```
public class BankAccountTests {

    private BankAccount bankAccount;
    @Mock
    private TransferValidator transferValidator;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
        bankAccount = new BankAccount("123456789", "user",
transferValidator);
    }

    @Test
    void deposit_ValidAmount_IncreasesBalance() {
        bankAccount.login("user", "secret");
        bankAccount.deposit(100.0, "ATM");
        assertEquals(100.0, bankAccount.getBalance());
    }

    @Test
    void withdraw_ValidAmount_DecreasesBalance() {
        bankAccount.login("user", "secret");
        bankAccount.deposit(200.0, "ATM");
        bankAccount.withdraw(50.0, "Supermarket");
        assertEquals(150.0, bankAccount.getBalance());
    }
```

```java
    @Test
    void transfer_ValidAmountAndBalance_TransfersSuccessfully() {
        BankAccount toAccount = new BankAccount("987654321", "receiver",
transferValidator);
        bankAccount.login("user", "secret");
        toAccount.login("receiver", "password");
        bankAccount.deposit(200.0, "ATM");
        bankAccount.transfer(toAccount, 100.0);
        assertEquals(100.0, bankAccount.getBalance());
        assertEquals(100.0, toAccount.getBalance());
    }

    @Test
    void transfer_InsufficientBalance_Fails() {
        BankAccount toAccount = new BankAccount("987654321", "receiver",
transferValidator);
        bankAccount.login("user", "secret");
        toAccount.login("receiver", "password");
        bankAccount.deposit(50.0, "ATM");
        bankAccount.transfer(toAccount, 100.0);
        assertEquals(50.0, bankAccount.getBalance());
        assertEquals(0.0, toAccount.getBalance());
    }

    @Test
    void transfer_InvalidCredentials_Fails() {
        BankAccount toAccount = new BankAccount("987654321", "receiver",
transferValidator);
        bankAccount.login("user", "wrongPassword");
        toAccount.login("receiver", "password");
        bankAccount.deposit(100.0, "ATM");
        bankAccount.transfer(toAccount, 50.0);
        assertEquals(100.0, bankAccount.getBalance());
        assertEquals(0.0, toAccount.getBalance());
    }
```

```
    @Test
    void transactionLog_WithdrawalRecordedInTransactionLog() {
        bankAccount.login("user", "secret");
        bankAccount.deposit(200.0, "ATM");
        bankAccount.withdraw(50.0, "Supermarket");
        assertEquals("Withdrew: 50.0 for Supermarket",
bankAccount.getTransactionLog().get(1));
    }

    @Test
    void transactionLog_TransferRecordedInTransactionLog() {
        BankAccount toAccount = new BankAccount("987654321", "receiver",
transferValidator);
        bankAccount.login("user", "secret");
        toAccount.login("receiver", "password");
        bankAccount.deposit(200.0, "ATM");
        bankAccount.transfer(toAccount, 100.0);
        assertEquals("Withdrew: 100.0 for Transfer to 987654321",
bankAccount.getTransactionLog().get(1));
        assertEquals("Deposited: 100.0 from Transfer from 123456789",
toAccount.getTransactionLog().get(0));
    }
}
```

```
    @Test
    void login_Successful_LoginFlagTrue() {
        bankAccount.login("user", "secret");
        assertTrue(bankAccount.isAuthenticated());
    }

    @Test
    void login_UnsuccessfulAfterThreeAttempts_AccountLocked() {
        bankAccount.login("user", "wrongPassword");
        bankAccount.login("user", "wrongPassword");
        bankAccount.login("user", "wrongPassword");
        assertTrue(bankAccount.isLocked());
    }

    @Test
    void logout_AfterSuccessfulLogin_IsAuthenticatedFalse() {
        bankAccount.login("user", "secret");
        bankAccount.logout();
        assertFalse(bankAccount.isAuthenticated());
    }

    @Test
    void logout_AfterUnsuccessfulLogin_IsAuthenticatedFalse() {
        bankAccount.login("user", "wrongPassword");
        bankAccount.logoutBefore();
        assertFalse(bankAccount.isAuthenticated());
    }

    @Test
    void transactionLog_DepositRecordedInTransactionLog() {
        bankAccount.login("user", "secret");
        bankAccount.deposit(100.0, "ATM");
        assertEquals("Deposited: 100.0 from ATM",
bankAccount.getTransactionLog().get(0));
    }
```

12. **Bibliografie**:

Pentru omorârea mutanților, înțelegere raportului s-au vizualizat astfel de clipuri:
MutationTestingVideo (youtube.com)

Kill All Mutants! (Intro to Mutation Testing) - Dave Aronson (youtube.com)

What is Mutation Testing? | PIT Maven MutationCoverage Example | Tech Primers (youtube.com)

Pentru Mock .jar s-au descărcat de pe:
Maven Central: org.mockito:mockito-core (sonatype.com)
Maven Central: org.objenesis:objenesis (sonatype.com)
Maven Central: net.bytebuddy:byte-buddy-agent (sonatype.com)
Maven Central: net.bytebuddy:byte-buddy-agent (sonatype.com)
Maven Central: net.bytebuddy (sonatype.com)

Pentru JUnit5 .jar s-au descărcat de pe:
Maven Central: org.junit.jupiter:junit-jupiter-engine (sonatype.com)
Maven Central: org.junit.jupiter:junit-jupiter-api (sonatype.com)
https://search.maven.org/artifact/org.junit.platform/junit-platform-engine
https://search.maven.org/artifact/org.junit.platform/junit-platform-commons