Popescu Paullo Robertto Karloss
Grupa 506 IS

Testarea Sistemelor Software

Documentație proiect laborator

- Salary Bonus Calculator –

Cuprins

1. Descrierea Proiectului

1.1 Obiectiv

Programul este implementat în limbajul **Java** și are rolul de a calcula procentul de bonus financiar pentru un angajat, în funcție de trei date de intrare: valoarea vânzărilor, numărul de zile absente și funcția deținută (manager/non-manager). Aplicația validează datele de intrare și aplică un set de reguli de business specifice, fiind verificată riguros prin tehnici de testare **White Box** și **Black Box**.

1.2 Specificația Funcțională

Metoda **`calculateBonus(int sales, int absentDays, boolean isManager)`** primește trei parametri și returnează valoarea bonusului (double), conform următoarelor reguli de business:

1. **Vânzări Puține:** Dacă `sales < 1000`, bonusul este **0**.
2. **Vânzări Medii:** Dacă `sales` este în intervalul `[1000, 5000]`, bonusul este **5%** din vânzări.
3. **Vânzări Ridicate:** Dacă `sales > 5000`:
   - Condiție **Bonus Maxim:** Dacă angajatul are puține absențe (`absentDays < 3`) **SAU** este manager (`isManager == true`), bonusul este **10%**.
   - Condiție Standard: În caz contrar (multe absențe și nu este manager), bonusul este **7%**.

```java
public class SalaryBonusCalculator {  Roberto
    /*...*/
    public double calculateBonus(int sales, int absentDays,  boolean isManager) {
        if (sales < 1000) {
            return 0.0; // no bonus :(
        } else if (sales <= 5000) {
            return sales * 0.05; // bonus 5%
        } else {
            // if sales > 5000
            if (absentDays < 3 || isManager) {
                return sales * 0.10; // bonus 10%
            } else {
                return sales * 0.07; // bonus 7%
            }
        }
    }
}
```

Implementarea metodei calculateBonus() folosind Java.

2. Generarea Datelor de Test și Implementare (JUnit) – Cerința 1

Pornind de la specificațiile funcționale, scenariile de testare au fost derivate prin aplicarea unor metode consacrate, precum **Equivalence Partitioning**, **Boundary Value Analysis (BVA)** și **Cause-Effect Graphing**. Validarea efectivă a acestor scenarii s-a realizat prin scrierea de teste unitare în **JUnit 5**.

2.1 Equivalence Partitioning (EP)

Această tehnică împarte domeniul de intrare în clase de echivalență. Deoarece condiția pentru "Bonus Maxim" este compusă (`absentDays < 3` **SAU** `isManager`), am împărțit această clasă în două cazuri diferite pentru a asigura acoperirea ambelor scenarii care duc la același rezultat.

Clase identificate:

- **CE1 (Invalid/Zero):** `sales < 1000` `// Bonus 0`
- **CE2 (Mediu):** `1000 <= sales <= 5000` `// Bonus 5%`
- **CE3 (Maxim - Performanță):** `sales > 5000` **AND** `absentDays < 3` (Indiferent dacă este sau nu manager) `// Bonus 10%`
- **CE4 (Maxim - Manager):** `sales > 5000` **AND** (`absentDays < 3` **OR** `isManager`) `// Bonus 10%`
- **CE5 (Redus):** `sales > 5000` **AND** (`absentDays >= 3` **AND** `!isManager`) `// Bonus 7%`

ID Test	Clasa	Input (Sales, Absent, Manager)	Output Așteptat	Rezultat JUnit
EP1	CE1	500, 0, false	0.0	Pass
EP2	CE2	2500, 5, false	125.0 (5%)	Pass
EP3	CE3	6000, 1, false	600.0 (10%)	Pass
EP4	CE4	6000, 4, true	600.0 (10%)	Pass
EP5	CE5	6000, 5, false	420.0 (7%)	Pass

```java
public class EquivalencePartitioningTest {  Robertto
    SalaryBonusCalculator calculator = new SalaryBonusCalculator();  5 usages

    // Clase echivalente identificate:
    // 1. Low Sales (<1000)
    // 2. Average Sales (1000 - 5000)
    // 3. Big Sales (>5000) cu performanta buna (absente putine sau manager)
    // 4. Big Sales (>5000) cu performanta slaba (absente multe si non-manager)

    @Test  Robertto
    void testEP_LowSales() {
        // Clasa: < 1000. Alegem valoarea 500.
        assertEquals( expected: 0.0, calculator.calculateBonus( sales: 500,  absentDays: 0,  isManager: false),  delta: 0.01);
    }

    @Test  Robertto
    void testEP_AverageSales() {
        // Clasa: [1000, 5000]. Alegem 2500. Bonus 5% (125).
        assertEquals( expected: 125.0, calculator.calculateBonus( sales: 2500,  absentDays: 5,  isManager: false),  delta: 0.01);
    }

    @Test  Robertto
    void testEP_BigSales_BiggestBonus() {
        // Clasa: > 5000, Absente < 3. Alegem 6000, 1 absenta. Bonus 10% (600).
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 1,  isManager: false),  delta: 0.01);
    }

    @Test  Robertto
    void test_Coverage_ManagerWithTooManyAbsentDays() {
        // Sales > 5000 (ca sa ajungem la ultimul if)
        // AbsentDays = 4 (ca prima conditie "absente < 3" sa fie FALSE)
        // isManager = true (ca sa verificam ramura de TRUE a variabilei isManager)

        // Rezultatul trebuie sa fie 10% (600.0) pentru ca e Manager, chiar daca a lipsit.
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 4,  isManager: true),  delta: 0.01);
    }

    @Test  Robertto
    void testEP_BigSales_LowBonus() {
        // Clasa: > 5000, Absente >= 3, Non-Manager. Alegem 6000, 5 absente. Bonus 7% (420).
        // Adauga 0.01 la final. Asta inseamna ca accepti o diferenta mai mica de 0.01
        // Am folosit delta pentru a compensa erorile de precizie specifice numerelor in virgula mobila (floating-point).
        assertEquals( expected: 420.0, calculator.calculateBonus( sales: 6000,  absentDays: 5,  isManager: false),  delta: 0.01);
    }
}
```

2.2 Boundary Value Analysis (BVA)

BVA se concentrează pe valorile de la limitele claselor, unde erorile sunt cel mai frecvent întâlnite (off-by-one errors). În implementare, am grupat testele limitelor inferioare și superioare în metode logice.

Limite critice analizate:

- **Sales:** 999 (sub limită), 1000 (limita inferioară interval 2), 5000 (limita superioară interval 2), 5001 (limita inferioară interval 3).
- **AbsentDays:** 2 (valid pentru bonus maxim), 3 (invalid pentru bonus maxim).

Metoda de Test (JUnit)	Descriere Limită	Input (sales, absentDays, isManager)	Output Așteptat	Observații / Tip Limită
testBVA_BonusLowerLimit	Imediat sub pragul minim	999, 0, false	0.0	Valoare invalidă pentru bonus (< 1000)
	Exact pe pragul minim	1000, 0, false	50.0 (5%)	Limita inferioară validă (5%)
testBVA_UpperLimitAverage	Limita superioară interval mediu	5000, 0, false	250.0 (5%)	Limita maximă pentru 5%
	Imediat peste pragul mediu	5001, 0, false	500.1 (10%)	Limita inferioară pentru bonus mare (> 5000)
testBVA_AbsentDaysLimit	Maximul de absențe permise	6000, 2, false	600.0 (10%)	Condiția < 3 este TRUE (2 < 3)
	Imediat peste limita de absențe	6000, 3, false	420.0 (7%)	Condiția < 3 devine FALSE (3 nu e < 3)

```java
public class BoundaryValueAnalysisTest {  Robertto
    SalaryBonusCalculator calculator = new SalaryBonusCalculator();  6 usages

    // Granite critice identificate:
    // Sales: 999, 1000 (trecere la 5%), 5000 (maxim 5%), 5001 (trecere la 10/7%)
    // absentDays: 2, 3 (limita pentru conditia < 3)

    @Test  Robertto
    void testBVA_BonusLowerLimit() {
        // 999 -> 0
        assertEquals( expected: 0.0, calculator.calculateBonus( sales: 999,  absentDays: 0,  isManager: false),  delta: 0.01);
        // 1000 -> 5% (50)
        assertEquals( expected: 50.0, calculator.calculateBonus( sales: 1000,  absentDays: 0,  isManager: false),  delta: 0.01);
    }

    @Test  Robertto
    void testBVA_UpperLimitAverage() {
        // 5000 -> 5% (250)
        assertEquals( expected: 250.0, calculator.calculateBonus( sales: 5000,  absentDays: 0,  isManager: false),  delta: 0.01);
        // 5001 -> 10% (500.1) - intram in zona de sus, presupunand absentDays putine
        assertEquals( expected: 500.1, calculator.calculateBonus( sales: 5001,  absentDays: 0,  isManager: false),  delta: 0.01);
    }

    @Test  Robertto
    void testBVA_AbsentDaysLimit() {
        // Vanzari > 5000, Non-Manager
        // 2 absente -> < 3 este True -> 10%
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 2,  isManager: false),  delta: 0.01);
        // 3 absente -> < 3 este False -> 7%
        assertEquals( expected: 420.0, calculator.calculateBonus( sales: 6000,  absentDays: 3,  isManager: false),  delta: 0.01);
    }
}
```

Implementare teste Boundary Value Analysis (BVA) folosing JUnit.

2.3 Cause-Effect Graphing (CEG)

Această metodă modelează relația logică dintre intrări (cauze) și ieșiri (efecte). Pentru logica **OR** din condiția de bonus maxim (`absentDays < 3` **SAU** `isManager`), am testat toate combinațiile posibile de adevăr.

Cauze (C):

- **C1:** `sales < 1000`
- **C2:** `1000 <= sales <= 5000`
- **C3:** `sales > 5000`
- **C4:** `absentDays < 3`
- **C5:** `isManager == true`

Efecte (Ef):

- **Ef1:** Bonus 0 (`Return 0.0`)
- **Ef2:** Bonus 5% (`Return sales * 0.05`)
- **Ef3:** Bonus 10% (`Return sales * 0.10`)
- **Ef4:** Bonus 7% (`Return sales * 0.07`)

Identificarea constrângerilor dintre cauze

În urma analizei domeniului de intrare, au fost identificate constrângeri logice stricte între cauzele care vizează variabila **sales**. Deoarece un număr nu poate aparține simultan mai multor intervale disjuncte, avem:

- **C1 (sales < 1000), C2 (1000 <= sales <= 5000) și C3 (sales > 5000) sunt mutual exclusive.**
    - Aceasta înseamnă că la orice rulare a programului, doar una dintre aceste trei cauze poate fi adevărată.
    - → **Constrângere de tip E (Exclusive)** între grupul {**C1, C2, C3**}. Această constrângere simplifică graful, eliminând combinațiile imposibile (de exemplu, C1 și C2 nu pot fi True în același timp).

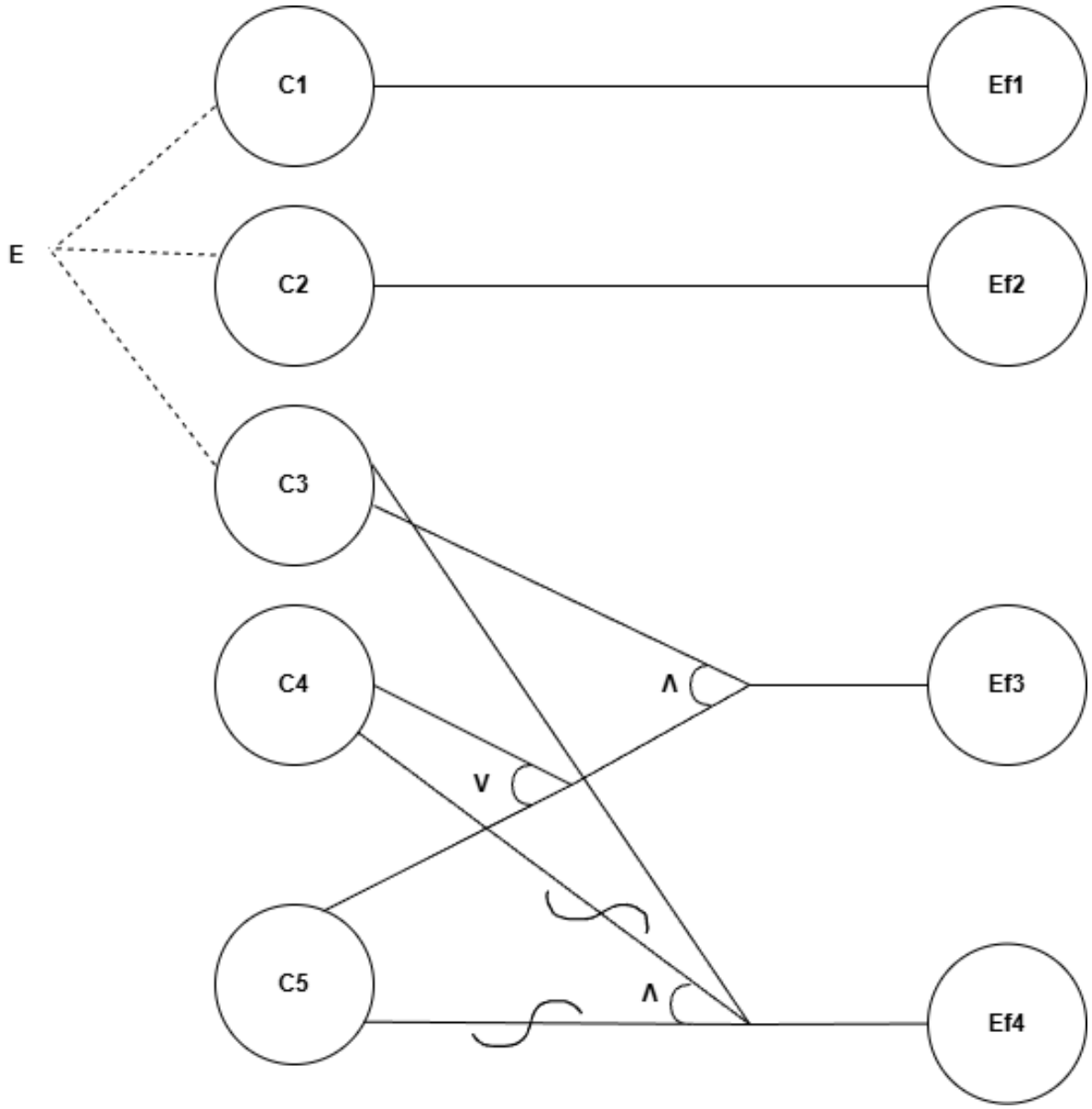
Stabilirea relațiilor cauză–efect

Relațiile logice care leagă cauzele de efecte au fost deduse direct din structura decizională a codului (if-else-if):

- **Regula 1 (Vânzări Puține)** - if (sales < 1000)
    - Dacă **C1** este adevărat, se produce direct **Ef1** (Bonus 0).

- → Relație directă: **C1 → Ef1**.

- **Regula 2 (Vânzări Medii)** - else if (sales <= 5000)
  - Această ramură este evaluată doar dacă C1 este fals. Dacă **C2** este adevărat, se produce **Ef2** (Bonus 5%).
  - → Relație directă: **C2 → Ef2**.
- **Regula 3 (Vânzări Ridicate cu Bonus Maxim)** - if (absentDays < 3 || isManager)
  - Acest efect (**Ef3** - Bonus 10%) apare doar dacă suntem în zona de vânzări mari (**C3**) **ȘI** este îndeplinită condiția compusă (puține absențe **SAU** este manager).
  - Formula logică: C3 AND (C4 OR C5).
  - → Relație complexă: **C3 ∧ (C4 ∨ C5) → Ef3**.
- **Regula 4 (Vânzări Ridicate cu Bonus Redus)** - else
  - Acest efect (**Ef4** - Bonus 7%) apare atunci când suntem în zona de vânzări ridicate (**C3**), dar condiția de bonus maxim nu este îndeplinită (adică angajatul are multe absențe **ȘI** nu este manager).
  - Formula logică: C3 AND (NOT C4 AND NOT C5).
  - → Relație complexă: **C3 ∧ ¬C4 ∧ ¬C5 → Ef4**.



Graful Cauză-Efect asociat.

Pe baza grafului, am generat tabelul de decizie. De remarcat că **Regula logică 3** (Bonus Maxim) a generat **3 cazuri de test distincte** în tabel (Regulile 3, 4 și 5), pentru a acoperi toate combinațiile tabelului de adevăr pentru operatorul **OR** (True/False, False/True, True/True).

Regula (Test)	C1	C2	C3	C4	C5	Efect Așteptat	Corespondență Metodă JUnit
1	1	-	-	-	-	Ef1	testCEG_LowSales
2	0	1	-	-	-	Ef2	testCEG_AverageSales
3	0	0	1	1	0	Ef3	testCEG_BigSales_LowAbsentDays
4	0	0	1	0	1	Ef3	testCEG_BigSales_isManager_TooManyAbsentDays
5	0	0	1	1	1	Ef3	testCEG_BigSales_isManager_LowAbsentDays
6	0	0	1	0	0	Ef4	testCEG_BigSales_LowPerformance

```java
public class CauseEffectGraphingTest {  Robertto
    SalaryBonusCalculator calculator = new SalaryBonusCalculator();  6 usages

    // Cazul 1: C1 -> E1
    @Test  Robertto
    void testCEG_LowSales() {
        // C1 este True (500 < 1000)
        // Restul nu conteaza
        assertEquals( expected: 0.0, calculator.calculateBonus( sales: 500,  absentDays: 0,  isManager: false),  delta: 0.01);
    }

    // Cazul 2: C2 -> E2
    @Test  Robertto
    void testCEG_AverageSales() {
        // C2 este True (2000 este intre 1000-5000)
        // 2000 * 5% = 100
        assertEquals( expected: 100.0, calculator.calculateBonus( sales: 2000,  absentDays: 0,  isManager: false),  delta: 0.01);
    }

    // Cazul 3: C3 + C4 -> E3
    // Vanzari Mari + Absente Putine + Nu e Manager
    @Test  Robertto
    void testCEG_BigSales_LowAbsentDays() {
        // C3 True (>5000), C4 True (<3 abs), C5 False
        // 6000 * 10% = 600
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 1,  isManager: false),  delta: 0.01);
    }
```

Implementare teste Cause-Effect-Graphing (CEG) folosing JUnit.

```java
    // Cazul 4: C3 + (NOT C4) + C5 -> E3
    // Vanzari Mari + Absente Multe + ESTE Manager
    @Test  Robertto
    void testCEG_BigSales_isManager_TooManyAbsentDays() {
        // C3 True, C4 False (5 absente), C5 True (Manager)
        // Logica: Chiar daca are absente, fiind manager ia 10%
        // 6000 * 10% = 600
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 5,  isManager: true),  delta: 0.01);
    }

    // Cazul 5: C3 + C4 + C5 -> E3
    // Vanzari Mari + Absente Putine + ESTE Manager (Bonus dublu asigurat, tot 10% ia)
    @Test  Robertto
    void testCEG_BigSales_isManager_LowAbsentDays() {
        // C3 True, C4 True, C5 True
        // 6000 * 10% = 600
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 1,  isManager: true),  delta: 0.01);
    }

    // Cazul 6: C3 + (NOT C4) + (NOT C5) -> E4
    // Vanzari Mari + Absente Multe + NU e Manager
    @Test  Robertto
    void testCEG_BigSales_LowPerformance() {
        // C3 True, C4 False (5 abs), C5 False
        // Intra pe ramura else finala -> 7%
        // 6000 * 7% = 420
        assertEquals( expected: 420.0, calculator.calculateBonus( sales: 6000,  absentDays: 5,  isManager: false),  delta: 0.01);
    }
```

Implementare teste Cause-Effect-Graphing (CEG) folosing JUnit.

3. Analiza Acoperirii Codului (Code Coverage) – Cerința 2

Pentru evaluarea calității seturilor de teste, s-a utilizat utilitarul **JaCoCo**, integrat în mediul de dezvoltare (Maven). Acesta a generat rapoarte privind **Instruction Coverage** (acoperirea liniilor de cod executate) și **Branch Coverage** (acoperirea deciziilor logice if/else parcurse).

Procesul a fost realizat prin linia de comandă (PowerShell), executând testele secvențial pentru a izola rezultatele fiecărei tehnici de testare (EP, BVA, CEG).

Etapele parcurse au fost următoarele:

Pasul 1: Execuția testelor și generarea fișierelor de date (.exec) Am rulat individual fiecare clasă de test, exportând datele de execuție în fișiere binare distincte în directorul target/.

- Pentru **Equivalence Partitioning (EP):**

```
mvn --% clean test -Dtest=validator.EquivalencePartitioningTest -
Djacoco.destFile=target/jacoco-ep.exec
```

- Pentru **Boundary Value Analysis (BVA):**

```
mvn --% clean test -Dtest=validator.BoundaryValueAnalysisTest -
Djacoco.destFile=target/jacoco-bva.exec
```

- Pentru **Cause-Effect Graphing (CEG):**

```
mvn --% clean test -Dtest=validator.CauseEffectGraphingTest -
Djacoco.destFile=target/jacoco-ceg.exec
```

Pasul 2: Generarea rapoartelor vizuale (HTML) Pe baza fișierelor .exec obținute anterior, am generat rapoartele lizibile pentru a analiza coverage-ul:

```
mvn --% jacoco:report -Djacoco.dataFile="target/jacoco-ep.exec"

mvn --% jacoco:report -Djacoco.dataFile="target/jacoco-bva.exec"

mvn --% jacoco:report -Djacoco.dataFile="target/jacoco-ceg.exec"
```

În urma rulării testelor, raportul rezultat (*target/site/jacoco/index.html*) l-am analizat, iar capturile relevante au fost incluse mai jos pentru **interpretarea datelor (3.1)**.

Rezultatele obținute pentru clasa `SalaryBonusCalculator` sunt centralizate în tabelul următor:

Set de Teste	Instruction Coverage	Branch Coverage	Ramuri Ratate (Missed)
Equivalence Partitioning (EP)	100% (31/31)	100% (8/8)	0
Boundary Value Analysis (BVA)	100% (31/31)	87% (7/8)	1
Cause-Effect Graphing (CEG)	100% (31/31)	100% (8/8)	0

3.1 Interpretarea Rezultatelor

a) Analiza Equivalence Partitioning (EP)

Setul de teste EP a obținut o acoperire perfectă (**100%**). Acest lucru se datorează faptului că partițiile de echivalență au fost identificate corect, incluzând nu doar intervalele numerice pentru vânzări, ci și clasele distincte generate de condiția compusă **OR** (angajat performant vs. manager). Includerea testului *test_Coverage_ManagerWithTooManyAbsentDays* a fost decisivă pentru acoperirea ramurii logice care tratează managerii cu absențe.

Rezultate Equivalence Partitioning (EP):

org.example.validator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
EquivalentMutant		0%		0%	6	6	8	8	2	2	1	1
NonEquivalentMutantKilled		0%		0%	6	6	8	8	2	2	1	1
NonEquivalentMutantNotKIlled		0%		0%	5	5	8	8	2	2	1	1
SalaryBonusCalculator		100%		100%	0	6	0	8	0	2	0	1
Total	91 of 122	25%	22 of 30	26%	17	23	24	32	6	8	3	4

SalaryBonusCalculator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
calculateBonus(int, int, boolean)		100%		100%	0	5	0	7	0	1
SalaryBonusCalculator()		100%		n/a	0	1	0	1	0	1
Total	0 of 31	100%	0 of 8	100%	0	6	0	8	0	2

b) Analiza Boundary Value Analysis (BVA)

Deși BVA a acoperit toate liniile de cod (**100% Instruction Coverage**), acesta a obținut doar **87% Branch Coverage**.

- **Cauza:** Setul BVA s-a concentrat riguros pe limitele numerice (sales: 5000/5001, absentDays: 2/3). Testul pentru limita superioară de absențe a folosit implicit isManager = false.
- **Efect:** Ramura de cod `else if (... || isManager)` a fost evaluată doar parțial. Scenariul în care `absentDays >= 3` dar `isManager == true` nu a fost o "graniță" numerică evidentă, deci nu a fost captat de acest set de teste strict numeric.

Rezultate Boundary Value Analysis (BVA):

org.example.validator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
EquivalentMutant		0%		0%	6	6	8	8	2	2	1	1
NonEquivalentMutantKilled		0%		0%	6	6	8	8	2	2	1	1
NonEquivalentMutantNotKIlled		0%		0%	5	5	8	8	2	2	1	1
SalaryBonusCalculator		100%		87%	1	6	0	8	0	2	0	1
Total	91 of 122	25%	23 of 30	23%	18	23	24	32	6	8	3	4

SalaryBonusCalculator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
calculateBonus(int, int, boolean)		100%		87%	1	5	0	7	0	1
SalaryBonusCalculator()		100%		n/a	0	1	0	1	0	1
Total	0 of 31	100%	1 of 8	87%	1	6	0	8	0	2

c) Analiza Cause-Effect Graphing (CEG)

Setul CEG a atins **100%** pe toate metricile. Deoarece această metodă se bazează pe tabelul de adevăr al logicii booleene, a forțat testarea tuturor combinațiilor True/False pentru condițiile compuse. Aceasta demonstrează superioritatea metodei CEG în validarea logicii condiționale complexe.

Rezultate Cause-Effect Graphing (CEG):

org.example.validator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
EquivalentMutant		0%		0%	6	6	8	8	2	2	1	1
NonEquivalentMutantKilled		0%		0%	6	6	8	8	2	2	1	1
NonEquivalentMutantNotKIlled		0%		0%	5	5	8	8	2	2	1	1
SalaryBonusCalculator		100%		100%	0	6	0	8	0	2	0	1
Total	91 of 122	25%	22 of 30	26%	17	23	24	32	6	8	3	4

SalaryBonusCalculator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
calculateBonus(int, int, boolean)		100%		100%	0	5	0	7	0	1
SalaryBonusCalculator()		100%		n/a	0	1	0	1	0	1
Total	0 of 31	100%	0 of 8	100%	0	6	0	8	0	2

3.2 Concluzii Comparative

Comparând cele trei tehnici, putem observa:

- **Complementaritatea Tehnicilor:** Faptul că EP și CEG au 100% acoperire nu înseamnă că BVA este inutil. BVA este singura metodă capabilă să detecteze erori de tip "off-by-one" (ex: < în loc de <=), erori pe care EP sau CEG le-ar putea rata chiar dacă trec prin acea linie de cod.
- **Eficiența Structurală: Cause-Effect Graphing** s-a dovedit cea mai robustă metodă pentru acoperirea structurală (Branch Coverage), garantând că nicio ramură logică nu rămâne neparcursă.
- Pentru o testare **completă**, este necesară utilizarea combinată: **CEG** pentru logica decizională și **BVA** pentru corectitudinea limitelor numerice.

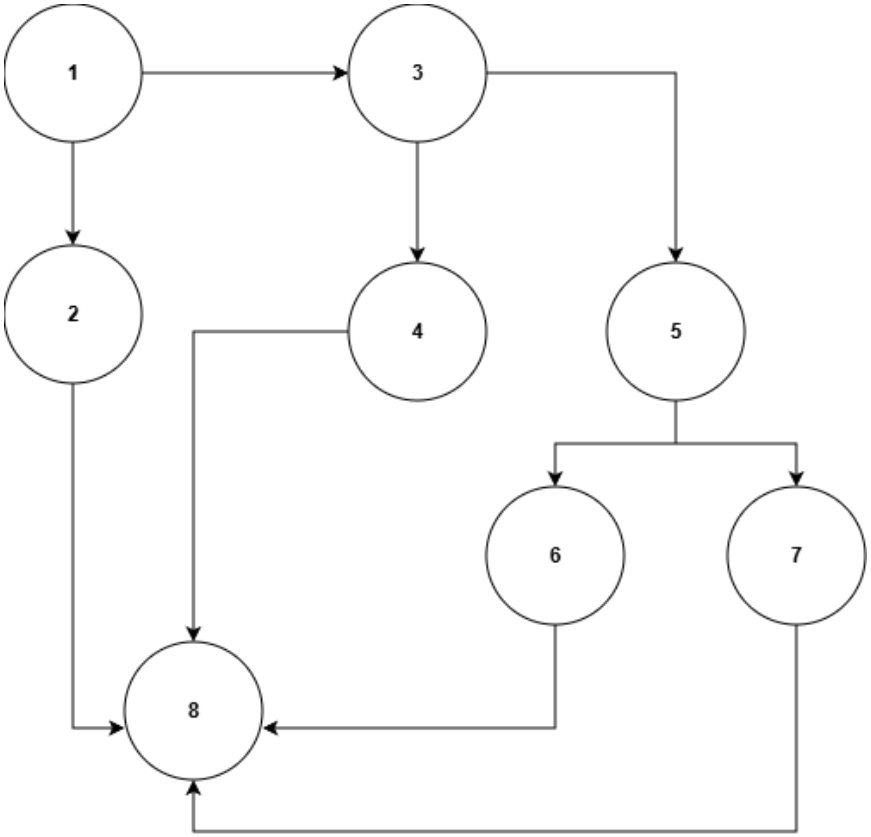
4. Analiza MC/DC și Graful de Control – Cerința 3

4.1 Graful Fluxului de Control (CFG)

Conform cerințelor de specificație, codul sursă a fost transformat într-un graf orientat. S-a optat pentru reprezentarea prin noduri numerotate (cercuri), unde fiecare nod corespunde unei instrucțiuni sau unei decizii din codul sursă.

Tabel de Mapare (Cod → Noduri):

ID_Nod	Instrucțiune	Tip Nod
1	if (sales < 1000)	Decizie simplă (Start)
2	return 0.0;	Instrucțiune Terminală
3	else if (sales <= 5000)	Decizie
4	return sales * 0.05;	Instrucțiune Terminală
5	if (absentDays < 3 || isManager)	Decizie compusă
6	return sales * 0.10;	Instrucțiune Terminală
7	return sales * 0.07;	Instrucțiune Terminală (Else implicit)
8	Exit	Ieșire din program



Graful orientat asociat programului.

Descrierea Fluxului: Graful evidențiază trei puncte de decizie (Nodurile 1, 3 și 5). Execuția pornește de la Nodul 1. În funcție de valoarea variabilei **sales**, fluxul poate termina rapid (Nod 2 sau 4) sau poate ajunge la **Nodul 5**, unde se evaluează condiția complexă pentru bonusul maxim. Toate nodurile terminale converg către Nodul 8 (Exit).

4.2 Modified Condition/Decision Coverage (MC/DC)

Analiza **MC/DC** se concentrează pe **Nodul 5** din graful de mai sus, deoarece acesta conține o decizie compusă formată din doi atomi logici.

Decizia analizată (Nod 5): D = (`absentDays < 3) || (isManager`).

Notăm condițiile atomice:

- **A:** `absentDays < 3`
- **B:** `isManager`
- Expresia este: **A OR B**

Pentru a satisface criteriul **MC/DC**, trebuie să demonstrăm că fiecare condiție atomică (A și B) poate determina independent rezultatul deciziei D, menținând cealaltă condiție fixă.

Setul de Teste (Perechi de Independență):

1. Demonstrarea influenței condiției A (absentDays < 3):

- **Fixăm B = False** (Angajatul nu este manager).
- **Test 1 (A=True):** absentDays = 2 (și isManager=false) → Decizia este **True** (Bonus 10%).
    - *Tranziție în graf:* Nod 5 → Nod 6.
- **Test 2 (A=False):** absentDays = 3 (și isManager=false) → Decizia este **False** (Bonus 7%).
    - *Tranziție în graf:* Nod 5 → Nod 7.
- *Observație:* Deoarece B a rămas constant, schimbarea rezultatului este cauzată exclusiv de modificarea lui A.

2. Demonstrarea influenței condiției B (isManager):

- **Fixăm A = False** (Angajatul are multe absențe, ex: 4).
- **Test 3 (B=False):** isManager = false (și absentDays=4) → Decizia este **False** (Bonus 7%).
    - *Tranziție în graf:* Nod 5 → Nod 7.

- **Test 4 (B=True):** isManager = true (și absentDays=4) → Decizia este **True** (Bonus 10%).
    - *Tranziție în graf:* Nod 5 → Nod 6.
- *Observație:* Deoarece A a rămas constant, schimbarea rezultatului este cauzată exclusiv de modificarea lui B.

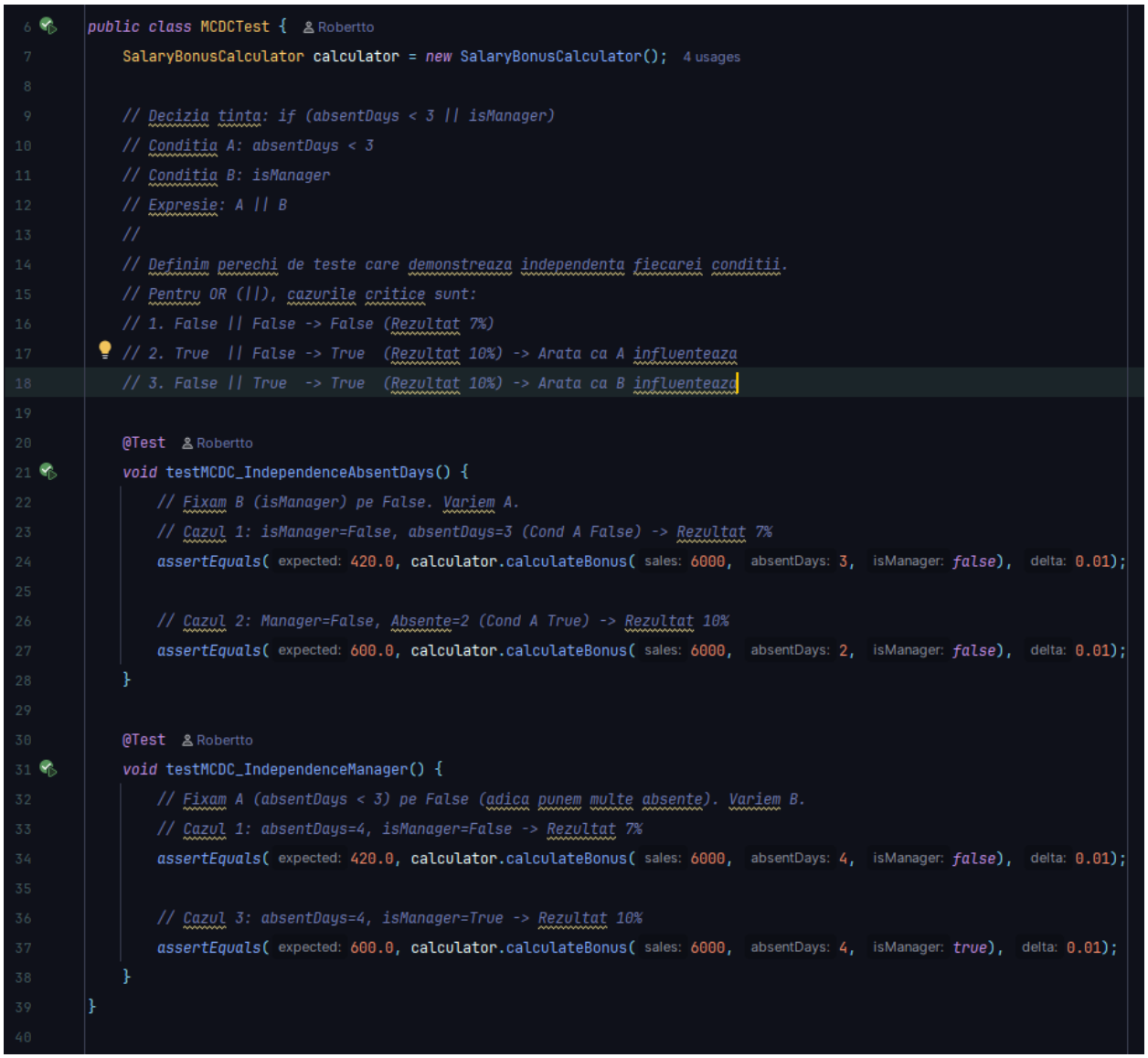
Concluzie: Setul de teste implementat în clasa **MCDCTest** acoperă aceste scenarii, satisfăcând complet criteriul **MC/DC** pentru Nodul 5.

Setul de Teste Rezultat (Sinteză)

Pe baza analizei de mai sus, am identificat următorul set minim de teste care satisface criteriul **MC/DC** pentru funcția **calculateBonus**. Toate testele folosesc o valoare a vânzărilor (**sales = 6000**) care asigură ajungerea fluxului de execuție în **Nodul 5**.

ID_Test	Input: Sales	Input: AbsentDays (A)	Input: IsManager (B)	Rezultat Așteptat	Ramura Acoperită	Scopul Testului
T_MCDC_1	6000	**2** (A=True)	**False**	10% (600.0)	True (Nod 5 → 6)	Independență A
T_MCDC_2	6000	**3** (A=False)	**False**	7% (420.0)	False (Nod 5 → 7)	Independență A & B
T_MCDC_3	6000	**4** (A=False)	**True**	10% (600.0)	True (Nod 5 → 6)	Independență B

Observație: Deși analiza curentă a inclus 4 cazuri de test pentru claritate didactică, **setul minimal de teste** pentru operatorul **OR** (A || B) se reduce la 3 combinații unice: (True, False), (False, True) și (False, False). Trebuie menționat că testul *T_MCDC_2* satisface cerința (False, False) pentru ambele condiții simultan.



```java
public class MCDCTest {  Roberto
    SalaryBonusCalculator calculator = new SalaryBonusCalculator();  4 usages

    // Decizia tinta: if (absentDays < 3 || isManager)
    // Conditia A: absentDays < 3
    // Conditia B: isManager
    // Expresie: A || B
    //
    // Definim perechi de teste care demonstreaza independenta fiecarei conditii.
    // Pentru OR (||), cazurile critice sunt:
    // 1. False || False -> False (Rezultat 7%)
    // 2. True  || False -> True  (Rezultat 10%) -> Arata ca A influenteaza
    // 3. False || True  -> True  (Rezultat 10%) -> Arata ca B influenteaza

    @Test  Roberto
    void testMCDC_IndependenceAbsentDays() {
        // Fixam B (isManager) pe False. Variem A.
        // Cazul 1: isManager=False, absentDays=3 (Cond A False) -> Rezultat 7%
        assertEquals( expected: 420.0, calculator.calculateBonus( sales: 6000,  absentDays: 3,  isManager: false),  delta: 0.01);

        // Cazul 2: Manager=False, Absente=2 (Cond A True) -> Rezultat 10%
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 2,  isManager: false),  delta: 0.01);
    }

    @Test  Roberto
    void testMCDC_IndependenceManager() {
        // Fixam A (absentDays < 3) pe False (adica punem multe absente). Variem B.
        // Cazul 1: absentDays=4, isManager=False -> Rezultat 7%
        assertEquals( expected: 420.0, calculator.calculateBonus( sales: 6000,  absentDays: 4,  isManager: false),  delta: 0.01);

        // Cazul 3: absentDays=4, isManager=True -> Rezultat 10%
        assertEquals( expected: 600.0, calculator.calculateBonus( sales: 6000,  absentDays: 4,  isManager: true),  delta: 0.01);
    }
}
```

Implementare teste MC/DC.

5. Mutant de Ordinul 1 Echivalent – Cerința 4

Definiție: Un mutant echivalent reprezintă o versiune modificată a codului sursă care, deși diferă sintactic de original, păstrează exact aceeași semantică funcțională. Deoarece comportamentul extern al programului nu se modifică, **niciun caz de test existent nu poate detecta ("omorî") acest mutant**; toate testele vor trece cu succes (**GREEN**).

Implementare în Proiect: Pentru a genera un mutant echivalent, am modificat condiția logică referitoare la numărul de absențe din metoda **calculateBonus**. S-a exploatat proprietatea numerelor întregi (tipul int), unde nu există valori fracționare între două numere consecutive.

Comparatie Cod:

- **Cod Original (SalaryBonusCalculator):**

```
// Condiția verifică strict mai mic decât 3
if (absentDays < 3 || isManager) {
    return sales * 0.10;
}
```

- **Cod Mutant (EquivalentMutant):**

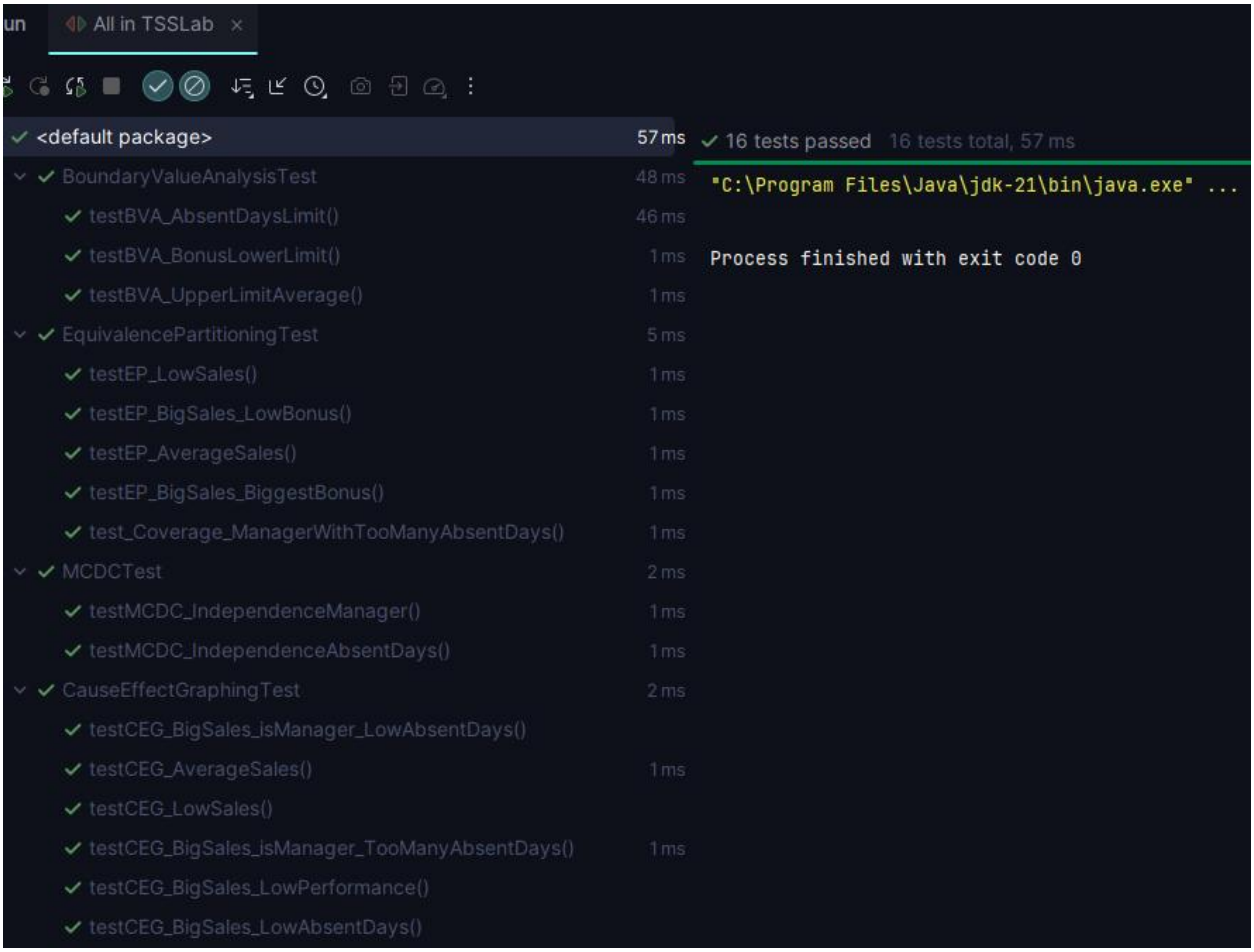
```
// Condiția verifică mai mic sau egal cu 2
if (absentDays <= 2 || isManager) {
    return sales * 0.10;
}
```

Analiza Echivalenței: Variabila absentDays este de tipul int. În mulțimea numerelor întregi, expresia matematică x < 3 este perfect echivalentă cu x <= 2.

- Dacă absentDays este 2: 2 < 3 (True) și 2 <= 2 (True).
- Dacă absentDays este 3: 3 < 3 (False) și 3 <= 2 (False).

Deoarece nu există niciun număr întreg (ex: 2.5) care să satisfacă una dintre condiții și pe cealaltă nu, comportamentul programului rămâne neschimbat pentru orice intrare posibilă.

Rezultatul Rulării Testelor: La rularea suitei complete de teste (EP, BVA, CEG) împotriva clasei EquivalentMutant, toate testele au trecut **(Passed)**. Acest lucru confirmă că mutantul a "supraviețuit", demonstrând că modificarea este într-adevăr echivalentă și nu a introdus niciun defect funcțional detectabil.



Rezultatul rulării testelor implementate folosind JUnit.

6. Analiza Mutanților Ne-echivalenți – Cerința 5

Pentru această etapă, am generat doi mutanți ne-echivalenți (modificări care schimbă logica de business) și am analizat comportamentul acestora în raport cu suita de teste existentă.

6.2 Mutant Ne-echivalent OMORÂT (Killed)

Un mutant este considerat "**Killed**" atunci când cel puțin un test din suită eșuează (devine roșu), semnalând că modificarea din cod este incorectă față de specificație.

- **Cod Mutant (NonEquivalentMutantKilled.java):** Am modificat multiplicatorul pentru calculul bonusului în cazul vânzărilor medii.

```
} else if (sales <= 5000) {
    // Modificare critică: de la 0.05 (5%) la 0.6 (60%)
    return sales * 0.6;
}
```

- Testul care omoară mutantul:
    - Testul *testBVA_UpperLimitAverage* din clasa *BoundaryValueAnalysisTest*.
    - **Input:** sales = 5000.
    - **Rezultat Așteptat (Corect):** 5000 * 0.05 = 250.0.
    - **Rezultat Mutant (Actual):** 5000 * 0.6 = 3000.0.
- **Concluzie:** La rularea testelor, JUnit raportează o eroare de tip AssertionFailedError (Expected: 250.0, Actual: 3000.0). Deoarece testul a picat, mutantul a fost detectat și **omorât**.

6.2 Mutant Ne-echivalent care să nu fie OMORÂT (Not Killed)

Un mutant este "**Not Killed**" atunci când modifică logica programului, dar testul rulat nu reușește să detecteze eroarea (testul rămâne verde). Acest lucru indică o lacună în testare pentru scenariul specific rulat.

- **Cod Mutant (NonEquivalentMutantNotKIlled.java):** Am eliminat verificarea statutului de manager din condiția de bonus maxim.

```
// Cod Original: if (absentDays < 3 || isManager)
// Cod Mutant: Am șters "|| isManager"
if (absentDays < 3) {
    return sales * 0.10;
}
```

- **Scenariul de Testare (Analiză):** Dacă rulăm testul *testCEG_BigSales_LowAbsentDays* din suita Cause-Effect:
    - **Input:** sales = 6000, absentDays = 1, isManager = false.
    - **Logica Mutantului:** Condiția absentDays < 3 este evaluată. Deoarece 1 < 3 este **True**, programul intră pe ramura de bonus 10%.
    - **Rezultat Așteptat:** 600.0.
    - **Rezultat Mutant:** 600.0.
- **Concluzie:** Deși codul este greșit (un manager cu multe absențe ar pierde bonusul în acest cod mutant), testul specificat verifică un angajat care are puține absențe. Prin urmare, modificarea nu afectează rezultatul *acestui* caz de testare. Testul trece (**Green**), iar mutantul **supraviețuiește**.

7. Concluzii

Prin acest proiect, am reușit să testez complet aplicația **SalaryBonusCalculator**, combinând mai multe tehnici pentru a mă asigura că nu există bug-uri ascunse.

Am observat că metodele se completează reciproc: deși **Boundary Value Analysis (BVA)** a avut o acoperire a ramurilor de **87%** (concentrându-se strict pe granițele numerice), celelalte metode precum **Equivalence Partitioning** și **Cause-Effect Graphing** au acoperit restul scenariilor logice. **Prin combinarea tuturor acestor seturi de teste, am obținut o acoperire globală (Code Coverage) de 100%.**

Partea de **MC/DC** a fost crucială pentru a înțelege cum interacționează condițiile complexe (manager vs. absențe), demonstrând că fiecare variabilă contează. Poate cea mai importantă lecție a venit din **testarea prin mutație**: am văzut concret cum un test poate să fie "**GREEN**" (să treacă) chiar dacă există un defect în cod, dacă scenariul de testare nu este suficient de specific.

În final, am obținut un set de teste robust, care nu doar că verifică funcționalitatea curentă, dar este și capabil să prevină erori pe viitor.