Mutaciono testiranje

Kvalitet koda vs. Kvalitet testova

Tradicionalne metrike (pokrivenost koda - Code Coverage) fokusiraju se na to koji delovi koda su izvršeni.

Visoka pokrivenost koda NE garantuje da će testovi zaista detektovati greške u logici.

Mutaciono testiranje pomera fokus sa 'šta je izvršeno' na 'šta je provereno'.

Glavno pitanje: Ako se unese greška u izvorni kod, da li će je naši testovi primetiti?

Šta je mutaciono testiranje?

Sistemska tehnika za procenu i poboljšanje adekvatnosti i efikasnosti serije testova.

Zasniva se na namernom uvođenju malih, veštačkih grešaka (mutacija) u izvorni kod aplikacije.

Nakon modifikacije, pokreće se postojeća serija testova nad izmenjenim kodom.

Cilj: Proveriti sposobnost test-paketa da identifikuje i odbaci (ubije) neispravne verzije programa.

Osnovne pretpostavke (Hipoteze)

Hipoteza o kompetentnom programeru (Competent Programmer Hypothesis):

- Programeri uglavnom pišu kod koji je blizu ispravnog.
- Stvarne greške su najčešće male sintaksne varijacije (npr. pogrešan operator, pogrešna konstanta).

Hipoteza o kompleksnim efektima (Coupling Effect Hypothesis):

- Kompleksne greške su usko povezane (kuplovane) sa jednostavnim greškama.
- Test-set koji uspešno detektuje jednostavne greške biće efikasan i u pronalaženju složenih defekata.

Osnovni pojmovi i terminologija

Mutant (Mutant): Verzija programa koja sadrži tačno jednu malu izmenu u kodu.

Originalni program: Stabilna i ispravna verzija koda nad kojom svi testovi prolaze.

Ubijeni mutant (Killed): Test-serija je uspešno detektovala promenu i barem jedan test je pao.

Preživeli mutant (Survived): Svi testovi su prošli uprkos izmeni u kodu, što ukazuje na nedostatak u test-setu.

Ekvivalentni mutant (Equivalent): Izmena koja menja sintaksu, ali ne i semantiku (ponašanje) programa.

Proces mutacionog testiranja

1. Pokretanje testova nad originalnim kodom (svi testovi moraju da prođu).
2. Generisanje mutanata primenom mutacionih operatora na izvorni kod.
3. Izvršavanje testova nad svakim pojedinačnim mutantom.
4. Analiza ishoda (prebrojavanje ubijenih, preživelih i ekvivalentnih mutanata).
5. Izračunavanje konačnog skora (Mutation Score) i unapređenje testova.

Metrika: Mutacioni skor (Mutation Score)

Mutacioni skor predstavlja glavni pokazatelj kvaliteta vaše test-serije.

Formula: MS = (Ubijeni mutanti / (Ukupan broj mutanti - Ekvivalentni mutanti)) * 100%

Značenje rezultata:

- Što je procenat bliži 100%, to je test-serija robusnija i sigurnija.
- Nizak skor direktno ukazuje na delove koda koji su 'pokriveni', ali nisu suštinski testirani (nedostaju asertacije).

Slaba vs. Jaka mutacija

U mutacionom testiranju, da bi mutant bio proglašen **ubijenim**, moraju se ispuniti tri uzastopna uslova (poznata kao **RIP model** u testiranju softvera):

1) **Reachability** (Dostupnost): Test mora izvršiti mutiranu liniju koda.

2) **Infection** (Infekcija): Izvršenje te linije mora dovesti do promene internog stanja programa (npr. pogrešna vrednost promenljive u memoriji).

3) **Propagation** (Propagacija): Ta interna promena stanja mora se preneti kroz program i manifestovati na samom kraju kao pogrešan izlaz (npr. pogrešan return, bačen izuzetak ili upis u bazu).

Razlika između **jake** i **slabe mutacije** leži isključivo u tome gde i kada alat proverava da li je mutant uspešno napravio razliku.

Jaka mutacija (Strong mutation)

Kako funkcioniše: Prati tradicionalni i najsigurniji pristup. Alat pokreće ceo test i čeka da se on završi. Tek na samom kraju proverava konačni izlaz programa (aserciju u testu).

Uslov za ubistvo: Mutant je ubijen samo ako se infekcija propagira do samog kraja koda i izazove pad testa (neuspešnu aserciju).

Primer: Ako u metodi imamo mutaciju koja promeni lokalnu promenljivu x = 5 u x = 0, ali se ta promenljiva kasnije u kodu nigde više ne koristi niti utiče na konačan return, test će proći. Kod jake mutacije, ovaj mutant preživljava jer nije uticao na krajnji ishod.

Problem: Ekstremno je spora jer zahteva izvršavanje kompletnih testova iznova i iznova za stotine ili hiljade mutanata.

Slaba mutacija (Weak mutation)

Kako funkcioniše: Predstavlja optimizaciju radi brzine. Alat postavlja "osmatrače" (monitore) u memoriju neposredno nakon mutirane linije koda.

Uslov za ubistvo: Mutant se proglašava ubijenim čim se primeti bilo kakva razlika u lokalnom stanju programa, odmah nakon izvršenja te linije. Ne čeka se kraj izvršavanja metode niti se gleda krajnja asercija testa.

Primer: U istom slučaju gde se x = 5 promenilo u x = 0, slaba mutacija odmah detektuje promenu vrednosti u registrima/memoriji i proglašava mutanta ubijenim, bez obzira na to što ta promenljiva kasnije nema nikakav uticaj na izlaz.

Prednost i mana: Mnogo je brža od jake mutacije. Međutim, mana je što može proglasiti mutanta "ubijenim" iako u realnosti vaši testovi uopšte nemaju asercije koje bi tu grešku zaista primetile na krajnjem izlazu aplikacije.

Uvod u PIT (PITest) alat

PIT je vodeći alat za mutaciono testiranje u Java ekosistemu.

Glavne prednosti PIT-a:

- Radi na nivou bajtkoda (.class fajlovi), a ne nad izvornim kodom, što drastično ubrzava proces.
- Odlično se integriše sa popularnim alatima kao što su Maven, Gradle i JUnit.
- Generiše izuzetno čitljive i detaljne HTML izveštaje sa vizuelnim prikazom koda.

Kako PIT optimizuje izvršavanje?

Mutaciono testiranje je tradicionalno izuzetno skupo i sporo.

PIT rešava ovaj problem kroz napredne optimizacije:

- Selekcija testova: Za svakog mutanta pokreće samo one testove koji zaista prolaze kroz tu liniju koda.
- Pametno upravljanje memorijom i paralelizacija.
- Prekid izvršavanja u slučaju beskonačnih petlji (Timeouts) koje mutanti mogu izazvati.

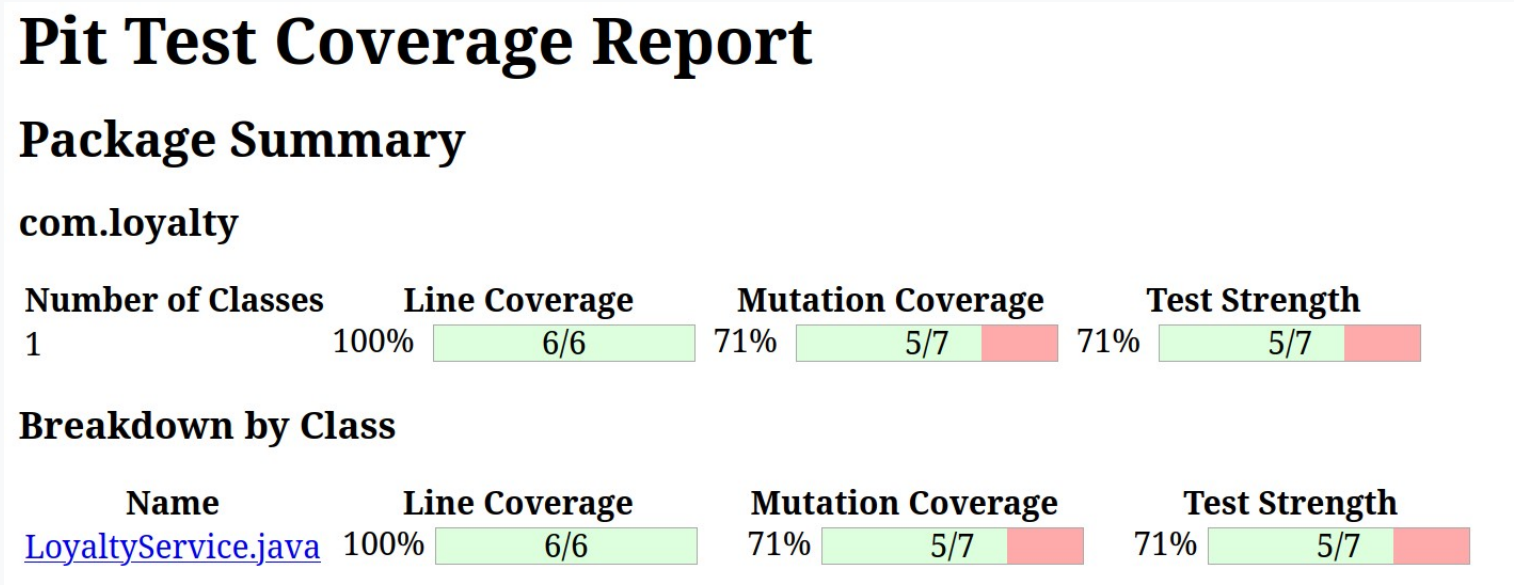
Struktura PIT HTML izveštaja

Nakon izvršavanja, PIT kreira paket HTML strana sa sledećim informacijama:

- Pregled po paketima i klasama sa procentualnim skorom pokrivenosti i mutacije.
- Vizuelno markiran izvorni kod aplikacije:

* Zelene linije: Svi mutanti na toj liniji su uspešno ubijeni.
* Crvene linije: Barem jedan mutant je preživeo.

- Detaljan spisak primenjenih operatora i tačan opis šta je promenjeno u svakom mutantu.



Pit Test Coverage Report

Package Summary

com.loyalty

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
1	100% 6/6	71% 5/7	71% 5/7

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
LoyaltyService.java	100% 6/6	71% 5/7	71% 5/7

LoyaltyService.java

```
1   package com.loyalty;
2
3   public class LoyaltyService {
4
5       public double calculateDiscount(int loyaltyPoints, double currentTotal) {
6   1       double discount = 0.0;
7
8           // Pravilo 1: Ako korisnik ima preko 100 bodova, dobija popust
9   2       if (loyaltyPoints > 100) {
10  2           discount = currentTotal * 0.10; // 10% popusta
11          }
12
13          // Pravilo 2: Inkrementiramo bodove za sledeću kupovinu
14  1       loyaltyPoints++;
15
16  1       return discount;
17      }
18  }
```

Mutations

6	1. Substituted 0.0 with 1.0 → KILLED
9	1. changed conditional boundary → KILLED 2. Substituted 100 with 101 → SURVIVED
10	1. Replaced double multiplication with division → KILLED 2. Substituted 0.1 with 1.0 → KILLED
14	1. Changed increment from 1 to -1 → SURVIVED
16	1. replaced double return with 0.0d for com/loyalty/LoyaltyService::calculateDiscount → KILLED

Active mutators

- CONDITIONALS_BOUNDARY
- INCREMENTS
- INLINE_CONSTS
- MATH
- PRIMITIVE_RETURNS

Mutacioni operatori u Javi

Mutacioni operatori su pravila koja definišu kako se kod menja radi generisanja mutanata.

1. Aritmetički operatori (Math Mutators)

Ovi operatori menjaju matematičke operacije u kodu.

- Zamena sabiranja i oduzimanja: Menja + u - i obrnuto (npr. a + b postaje a - b).
- Zamena množenja i deljenja: Menja * u / i obrnuto (npr. a * b postaje a / b).
- Modul operator: Menja ostatak pri deljenju % u množenje * (ili obrnuto u zavisnosti od verzije).
- Bitski operatori (Bitwise): Menja logičko I, ILI i Ekskluzivno ILI na nivou bitova (&, |, ^) kao i operatore pomeranja bitova (<<, >>, >>>).

Mutacioni operatori u Javi

2. Uslovni i relacioni operatori (Conditionals Mutators)
Ovo je kritična grupa jer programeri najčešće prave greške upravo na granicama uslova (tzv. off-by-one greške).

- Operatori graničnih vrednosti (Conditionals Boundary): Menjaju striktne nejednakosti u nestriktne i obrnuto.
  - < postaje <=
  - <= postaje <
  - > postaje >=
  - >= postaje >
- Uklanjanje uslova (Negate Conditionals): Menja suštinu samog logičkog poređenja.
  - == postaje !=
  - != postaje ==
  - <= postaje >
  - >= postaje <
  - < postaje >=
  - > postaje <=

Mutacioni operatori u Javi

3. Mutatori povratnih vrednosti (Return Values Mutators)

Ovi operatori simuliraju situaciju u kojoj metoda vraća pogrešan rezultat ili "prazan" objekat, testirajući da li sistem koji poziva tu metodu uopšte validira dobijene podatke.

- Primitivni tipovi: Ako metoda vraća ceo broj (int, long), menja povratnu vrednost u 0 (ako je bila različita od nule) ili u 1 (ako je originalno bila nula). Za boolean metode, menja true u false i obrnuto.
- Objektne reference (Empty Returns): Menja povratnu vrednost metode tako da vrati "praznu" instancu klase ukoliko je to moguće (npr. vraća prazan Optional, prazan List, Set, Map ili prazan String umesto stvarnog objekta).
- Null Returns: Prisilno menja povratnu vrednost u null. Odličan mutator za proveru da li vaši testovi hvataju potencijalni NullPointerException.

Mutacioni operatori u Javi

4. Inkrementi i dekrementi (Incremental Mutators)

Mutator lokalnih inkrementa: Menja operatore ++ u -- i obrnuto unutar lokalnih promenljivih (npr. i++ unutar for petlje postaje i--). Ovo je izuzetno opasan mutant jer u petljama gotovo uvek dovodi do beskonačnog izvršavanja (što PIT uspešno detektuje kroz Timeout).

5. Mutatori konstanti (Inline Constant Mutators)

Menjaju hardkodovane vrednosti u kodu (literale) kako bi proverili da li testovi zaista proveravaju tačnost konfiguracije ili fiksnih poslovnih pravila.

- Ako je konstanta 0, menja je u 1.
- Ako je konstanta bilo koji drugi broj (npr. 5), menja je u 0 ili joj dodaje/oduzima 1.
- Menja vrednosti unutar tekstualnih konstanti (String literala).

Mutacioni operatori u Javi

6. Napredni i eksperimentalni operatori (Advanced / Experimental)

U novijim verzijama PIT alata dodati su operatori koji ciljaju specifične Java koncepte i objektno-orijentisani dizajn:

- Uklanjanje poziva metoda (Void Method Calls / Non-Void Method Calls): Potpuno uklanja liniju koda u kojoj se poziva neka druga metoda koja vraća void (npr. uklanja poziv logger.info() ili što je još važnije baza.izmeniStanje()). Verzija za "Non-Void" metode uklanja poziv metode, a umesto njenog rezultata podmeće podrazumevanu vrednost tipa (npr. 0 ili null).
- Konstruktori (Constructor Call Mutator): Menja pozive konstruktora unutar izraza, podmećući null kako bi se proverilo da li se kreiranje objekta adekvatno testira.
- Izuzeci (Remove Exception Handler): Uklanja catch blokove ili kod unutar njih, simulirajući situaciju gde aplikacija ignoriše ili ne uspeva da obradi nastalu grešku.
- Menjanje Switch naredbi (Switch Mutator): Modifikuje labele unutar switch-case blokova ili menja default granu granjanja kako bi proverio da li su pokriveni svi execution path-ovi.

Konkretan primer mutacije (Java)

```
public class LoyaltyService {

    public double calculateDiscount(int loyaltyPoints,
double currentTotal) {
        double discount = 0.0;

        // Ako korisnik ima preko 100 bodova, dobija popust
        if (loyaltyPoints > 100) {
            discount = currentTotal * 0.10; // 10% popusta
        }

        // Inkrementiramo bodove za sledeću kupovinu
        loyaltyPoints++;

        return discount;
    }
}
```

1. Mutator graničnih vrednosti (Conditionals Boundary Mutator)

```
    // Mutant 1
    if (loyaltyPoints >= 100) {
```

2. Aritmetički mutator (Math Mutator)

```
    // Mutant 2
    discount = currentTotal / 0.10;
```

3. Mutator lokalnih inkrementa (Incremental Mutator)

```
    // Mutant 3
    LoyaltyPoints--;
```

4. Mutator konstanti u kodu (Inline Constant Mutator)

```
    // Mutant 4
    double discount = 1.0;
```

5. Primitivni mutator povratnih vrednosti (Return Values Mutator)

```
    // Mutant 5
    return 0.0;
```

JUnit testovi

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

class LoyaltyServiceTest {

    private final LoyaltyService service = new LoyaltyService();

    @Test
    void testNoDiscountForLowPoints() {
        // Korisnik sa malo bodova (50) ne bi trebalo da dobije popust
        double discount = service.calculateDiscount(50, 100.0);
        assertEquals(0.0, discount, 0.001);
    }

    @Test
    void testDiscountForHighPoints() {
        // Korisnik sa dosta bodova (npr. 150) dobija 10% popusta
        double discount = service.calculateDiscount(150, 100.0);
        assertEquals(10.0, discount, 0.001);
    }
```

```
    @Test
    void testExactBoundaryPoints() {
        // Granični slučaj: tačno 100 bodova (prema kodu, popust
je tek PREKO 100)
        double discount = service.calculateDiscount(100, 100.0);
        assertEquals(0.0, discount, 0.001);
    }

    @Test
    void testZeroPointsCondition() {
        // Ekstremni slučaj: 0 bodova
        double discount = service.calculateDiscount(0, 50.0);
        assertEquals(0.0, discount, 0.001);
    }
}
```

Mutaciona matrica

- Ova tabela direktno prikazuje interakciju između naših testova i 5 mutanata koje smo definisali u prethodnom koraku.
- Oznaka K (Killed / Ubijen) znači da test pada na mutiranom kodu, što je poželjno ponašanje jer smo uhvatili grešku.
- Oznaka P (Survived / Preživeo) znači da test prolazi iako je kod pogrešan, što ukazuje na rupu u test-paketu.

Test metod / Naziv testa	Mutant 1 (>= 100)	Mutant 2 (/ 0.10)	Mutant 3 (points--)	Mutant 4 (discount=1.0)	Mutant 5 (return 0.0)
testNoDiscountForLowPoints	P	P	P	K *(vratio 1.0 umesto 0.0)*	P
testDiscountForHighPoints	P	K *(vratio 1000 umesto 10)*	P	P	K *(vratio 0.0 umesto 10)*
testExactBoundaryPoints	K *(vratio 10 umesto 0)*	P	P	K *(vratio 1.0 umesto 0.0)*	P
testZeroPointsCondition	P	P	P	K *(vratio 1.0 umesto 0.0)*	P
Konačni ishod za mutanta:	UBIJEN	UBIJEN	PREŽIVEO	UBIJEN	UBIJEN

Analiza rezultata

- **Mutation Score (Mutacioni skor)**: Iznosi 80% (4 od 5 mutanata su ubijeni).
- **Zašto je Mutant 3 preživeo?** Nijedan test ne može da ga ubije. Promenljiva loyaltyPoints je lokalna (prenosi se po vrednosti), a njena izmenjena vrednost se nakon inkrementiranja/dekrementiranja više nigde ne koristi unutar metode niti utiče na povratnu vrednost. Ovo je odličan primer ekvivalentnog mutanta (ili mutanta u "mrtvom kodu") koji se ne može ubiti spoljnim testiranjem ponašanja.

Problem ekvivalentnih mutanata

Jedan od najvećih teorijskih i praktičnih izazova u mutacionom testiranju.

Kod se sintaksno menja, ali program i dalje proizvodi potpuno iste izlazne rezultate za sve ulaze.

Pošto je ponašanje isto, nijedan test ne može da padne (mutant uvek preživi).

Automatska detekcija ekvivalentnih mutanata je generalno neodlučiv problem.

Zahteva ručnu analizu koda kako bi se isključili iz proračuna adekvatnosti.

Praktični saveti za rad

Nemojte pokretati mutaciono testiranje na celom projektu odjednom ako je baza koda velika.

Fokusirajte se na kritičnu poslovnu logiku gde su asertacije od presudnog značaja.

Koristite PIT filtere za isključivanje klasa koje ne sadrže logiku (DTO, konfiguracije, entiteti).

Preživeli mutanti vam govore gde treba da dodate nove test-slučajeve.