

# JUnit

Seminarski rad u okviru kursa  
Verifikacija softvera  
Matematički fakultet

Lazar Mladenović, 1029/2019  
lazar.s.mladenovic@hotmail.com

10. januar 2020

## Sažetak

Test jedinice koda je deo koda koji je napisao programer koji izvršava određenu funkcionalnost u kodu koji se testira i potvrđuje određeno ponašanje ili stanje. Procenat koda koji se testira testovima jedinice koda obično se naziva pokrivenost testom. Ovaj test cilja malu jedinicu koda, npr. metodu ili klasu. Spoljne zavisnosti trebalo bi ukloniti iz testova jedinice koda, npr. zamenom zavisnosti sa testnom implementacijom ili (mock) objektom kreiranim od strane testnog okvira.

Za Javu je dostupno nekoliko framework-a za testiranje. Najpopularniji su JUnit i TestNG. U ovom tekstu će biti obrađen JUnit Jupiter, poznatiji kao JUnit 5.

## Sadržaj

<b>1</b>	<b>O testiranju jedinica koda</b>	<b>2</b>
1.1	Delovi koda koje bi trebalo testirati . . . . .	2
<b>2</b>	<b>JUnit</b>	<b>2</b>
2.1	Kako definisati JUnit test . . . . .	2
2.2	Konvencije o imenovanju . . . . .	3
<b>3</b>	<b>Pisanje testova</b>	<b>3</b>
3.1	Definisanje test metoda . . . . .	3
3.2	Tagovi . . . . .	3
3.3	Testne klase i metode . . . . .	5
3.4	Tvrđenja . . . . .	6
3.5	Pretpostavke . . . . .	7
3.6	Onemogućavanje testova . . . . .	7
3.7	Uslovno izvršavanje testova . . . . .	8
3.8	Redosled izvršavanja testova . . . . .	9
3.9	Ugnježdeni testovi . . . . .	10
3.10	Ponavljajući testovi . . . . .	11
3.11	Parametrizovani testovi . . . . .	11
3.12	Dinamički testovi . . . . .	13
<b>4</b>	<b>Podrška Eclipse razvojnog okruženja za JUnit 5</b>	<b>13</b>
	<b>Literatura</b>	<b>16</b>

# 1 O testiranju jedinica koda

Jedinični testovi definisani su standardom *IEEE Standard for Software*. Cilj jediničnih testova je dokazivanje da izolovani delovi koda imaju predviđenu funkcionalnost. Ovom vrstom testiranja prolazi se svaki i najmanji deo sistema. Svaki vid komunikacije sa mrežom, bazom podataka, fajl sistemom pa čak i sa drugim klasama, modulima i komponentama sistema se apstrahuje u nekakve fiksirane vrednosti. Prilikom izvršavanja ovih testova dozvoljena je samo komunikacija sa memorijom. Testove piše programer. Ukoliko postoje greške unutar jedinice koda, one bi trebalo da budu otkrivene u ovoj fazi.

## 1.1 Delovi koda koje bi trebalo testirati

Ono što bi trebalo testirati je veoma kontroverzna tema. Neki programeri veruju da svaku liniju u vašem kodu treba testirati.

U svakom slučaju trebate napisati softverske testove za kritične i složene delove vaše aplikacije. Ako uvedete nove funkcije, čvrsti testni paket vas takođe štiti od regresije u postojećem kodu. U zavisnosti od konteksta i programske paradigme, to mogu biti podprogrami, klase, manje ili veće celine formirane od tesno povezanih jedinica.

Uopšteno, sigurno je ignorisati trivijalni kod. Na primer, beskorisno je pisati testove za *getter* i *setter* metode koji prosto dodeljuju vrednosti poljima. Pisanje testova za ove izjave zahteva mnogo vremena i besmisleno je, kao što biste testirali Java virtuelnu mašinu. Sam JVM već ima test slučajeve za to. Ako razvijate aplikacije za krajnjeg korisnika, možete pretpostaviti da dodjela polja funkcioniše na Javi.

# 2 JUnit

JUnit je framework za testiranje Java programa koji koristi određene anotacije da identifikuje metode koje specifikuju test. JUnit je projekat otvorenog koda objavljen na *Github*-u.

Testni kod odvojen je od stvarnog programskog koda, a kod većine IDE-a rezultati ispitivanja su takođe odvojeni od rezultata programa, pružajući čitljivu i prikladnu strukturu.

JUnit 5 je poslednje veliko izdanje JUnit-a.

JUnit 5 sastoji se od niza diskretnih komponenti:

- JUnit Platform - temeljni sloj koji omogućava pokretanje različitih framework-a za testiranje na JVM
- JUnit Jupiter - testni framework JUnit 5 koji je pokrenula JUnit Platform
- JUnit Vintage - TestEngine koji izvršava starije testove

Upotreba je vrlo jednostavna. Iako je JUnit 5 doneo je neke razlike i pogodnosti u odnosu na JUnit 4, JUnit 4 je i dalje široko rasprostranjen. Minimalni JDK za JUnit 4 bio je JDK 5, dok za JUnit 5 treba najmanje JDK 8. Međutim, i dalje možete testirati kodove koji su kompajlirani na ranijim verzijama JDK.

## 2.1 Kako definisati JUnit test

JUnit test je metoda sadržana u klasi koja se koristi samo za testiranje. To se zove *Test klasa*. Da biste definisali da je određena metoda test

metoda, označite je sa `@Test`.

Ova metoda izvršava kod koji se testira. Da biste proverili očekivani rezultat u odnosu na stvarni rezultat, koristite metodu `assert` koja je dao JUnit-a ili neki drugi framework za potvrđivanje. Ovi pozivi metoda obično se nazivaju tvrđenjima. Više o ovome pogledajte u poglavlju 3.4 koje sledi.

U ovim tvrđenjima trebalo bi da navedete smislene poruke. To korisniku olakšava prepoznavanje i otklanjanje problema. Ovo je posebno tačno ako neko drugi pogleda problem, a da pritom nije pisao kod koji se testira, niti kod testa.

## 2.2 Konvencije o imenovanju

Postoji nekoliko potencijalnih konvencija o imenovanju JUnit testova. Rešenje koje ima široku upotrebu je upotreba sufiksa „Test“ na kraju naziva klase.

Opšte pravilo kaže da bi trebalo da naziv testa objašnjava šta test čini. Ako se to obavi pravilno, može se izbeći čitanje stvarne implementacije.

Jedan od mogućih pristupa je upotreba infiksa *”should”* u nazivu metode ispitivanja. Na primer, *”orderShouldBeCreate”* ili *”menuShouldGetActive”*. Ovo daje nagoveštaj šta bi trebalo da se dogodi ako se izvrši metoda testiranja.

## 3 Pisanje testova

### 3.1 Definisane test metode

Naredni primer ilustruje minimum koji je potreban da bi se definisao JUnit Jupiter test. Naredni odeljci ovog poglavlja pružiće dodatne detalje o svim dostupnim funkcijama.

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

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }
}
```

### 3.2 Tagovi

JUnit koristi određene tagove da metode označi kao metode testa i da ih konfiguriše. Sledeća tabela daje pregled najvažnijih tagova u JUnit-u. Svi ovi tagovi mogu se koristiti nad metodama. Ako nije drugačije navedeno, svi osnovni tagovi nalaze se u paketu *org.junit.jupiter.api* u modulu *junit-jupiter-api*.

Tabela 1: Tagovi

Tag	Opis
@Test	Identifikuje metodu kao test metodu
@ParameterizedTest	Označava da je metoda parametrizovani test.
@RepeatedTest	Označava da je test metoda obrazac za ponovljeni test.
@TestFactory	Označava da je metoda skladište za dinamičke testove.
@TestTemplate	Označava da je metoda obrazac za testne slučajeve dizajnirane da se pozivaju više puta u zavisnosti od broja konteksta poziva koji su registrovani provajderi vratili.
@TestMethodOrder	Koristi se da označi redosled izvršavanja testnih metoda.
@TestInstance	Koristi se za konfiguriranje životnog ciklusa testne instance za označenu klasu ispitivanja.
@DisplayName	Definiše proizvoljno ime koje će se prikazati za test klasu ili metod testiranja.
@BeforeEach	Metoda označena ovim tagom se treba izvršiti pre svake @Test, @RepeatedTest, @ParameterizedTest ili @TestFactory metode u datoj klasi, analogno tagu @Before u JUnit 4.
@AfterEach	Metoda označena ovim tagom se treba izvršiti posle svake @Test, @RepeatedTest, @ParameterizedTest ili @TestFactory metode u datoj klasi, analogno tagu @After u JUnit 4.
@BeforeAll	Označava da metodu treba izvršiti pre svih metoda @Test, @RepeatedTest, @ParameterizedTest i @TestFactory u datoj klasi, analogno tagu @BeforeClass u JUnit 4. Ovaj metod mora biti statički.
@AfterAll	Označava da metodu treba izvršiti nakon svih metoda @Test, @RepeatedTest, @ParameterizedTest i @TestFactory u datoj klasi, analogno tagu @BeforeClass u JUnit 4. Ovaj metod mora biti statički.
@Nested	Označava da je označena klasa nestatička ugnježđena klasa testa. Metode @BeforeAll i @AfterAll ne mogu se koristiti direktno u testnoj klasi @Nested ukoliko se ne koristi životni ciklus testne instance za "per-class"
@Disabled	Koristi se da onemogući datu testnu klasu ili metodu. Analogno tagu @Ignore u JUnit 4.

Tag	Opis
@Tag	Koristi se za proglašavanje oznaka za filtriranje testova, bilo na nivou klase ili metode, analogno Categories u JUnit 4. Takve napomene nasleđuju se na nivou klase, <u>ali ne i na nivou metode.</u>
@Timeout	Koristi se za obaranje testa, testnog skupa, ili obrasca ispitivanja ako njihovo izvršenje premašuje određeno trajanje.

### 3.3 Testne klase i metode

Test klasa je bilo koja klasa najvišeg nivoa ili *@Nested* klasa koja sadrži najmanje jednu metodu ispitivanja. Ove klase ne smeju biti apstraktne i moraju imati jedan konstruktor.

Metoda ispitivanja je bilo koja metoda koja je direktno označena tagom *@Test*, *@RepeatedTest*, *@ParameterizedTest*, *@TestFactory* ili *@TestTemplate*.

Metoda životnog ciklusa je bilo koja metoda koja je direktno označena tagom *@BeforeAll*, *@AfterAll*, *@BeforeEach* ili *@AfterEach*.

Sledeća klasa testa pokazuje upotrebu *@Test* metoda i svih podržanih metoda životnog ciklusa.

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }
}
```

```

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }
}

```

### 3.4 Tvrdenja

JUnit Jupiter dolazi sa mnogim metodama tvrdnji koje JUnit 4 ima i dodaje još nekoliko njih. Sve tvrdnje JUnit 5 su statičke metode u klasi *org.junit.jupiter.api.Assertions*.

Tabela 2: Metode za potvrđivanje rezultata ispitivanja

Tvrđenje	Opis
<code>fail([message])</code>	Pušta da metoda ne uspe. Može se koristiti za proveru da određeni deo koda nije dostignut. Parametar poruke nije obavezan.
<code>assertTrue([message,] boolean condition)</code>	Proverava da li je zadati uslov tačan.
<code>assertFalse([message,] boolean condition)</code>	Proverava da li je zadati uslov pogrešan.
<code>assertEquals([message,] expected, actual)</code>	Testira da li su dve vrednosti iste. Napomena: za nizove se proverava referenca, a ne sadržaj nizova.
<code>assertEquals([message,] expected, actual, tolerance)</code>	Ispituje da li se podudaraju <i>float</i> ili <i>double</i> vrednosti. Tolerancija je broj decimala koji mora biti isti.
<code>assertNull([message,] object)</code>	Testira da li je objekat <i>null</i> .
<code>assertNotNull([message,] object)</code>	Testira da li objekat nije <i>null</i> .
<code>assertSame([message,] expected, actual)</code>	Proverava da li se obe varijable odnose na iste objekte.
<code>assertNotSame([message,] expected, actual)</code>	Testira da li se obe varijable odnose na različite objekte.

Tvrđenje	Opis
<code>assertTimeout(Duration timeout, Executable executable)</code>	Tvrđi da se izvršavanje isporučenog izvršnog programa završava pre nego što je dato vremensko ograničenje prekoračeno.
<code>assertThrows(Class&lt;T&gt; expectedType, Executable executable)</code>	Tvrđi da izvršavanje isporučenog izvršnog programa baca izuzetak očekivanog tipa i vraća izuzetak.

Više o tvrđenjima pogledati na [spisku metoda tvrđenja](#).

### 3.5 Pretpostavke

JUnit 5 dolazi sa podskupom metoda pretpostavki koje JUnit 4 pruža i dodaje još nekoliko njih. Sve pretpostavke JUnit Jupitera su statičke metode u klasi *org.junit.jupiter.api.Assumptions*.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import example.util.Calculator;
import org.junit.jupiter.api.Test;

class AssumptionsDemo {

    private final Calculator calculator = new Calculator();

    @Test
    void testOnlyOnCiServer() {
        assumeTrue("CI".equals(System.getenv("ENV")));
        // remainder of test
    }

    @Test
    void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation");
        // remainder of test
    }

    @Test
    void testInAllEnvironments() {
        assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the CI server
                assertEquals(2, calculator.divide(4, 2));
            });

        // perform these assertions in all environments
        assertEquals(42, calculator.multiply(6, 7));
    }
}
```

### 3.6 Onemogućavanje testova

Čitave test klase ili pojedinačne metode ispitivanja mogu se onemogućiti putem *@Disabled* taga.

U prvom primeru je onemogućena cela klasa, a u drugom samo jedna od metoda klase.

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled("Disabled until bug #99 has been fixed")
class DisabledClassDemo {

    @Test
    void testWillBeSkipped() {
    }

}
```

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTestsDemo {

    @Disabled("Disabled until bug #42 has been resolved")
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }

}
```

### 3.7 Uslovno izvršavanje testova

API za proširenje ExecutionCondition u JUnit Jupiter-u omogućava programerima da ili omoguće ili onemoguće test ili skup testova. Najjednostavniji primer takvog stanja je ugrađena DisabledCondition koja podržava *@Disabled* napomenu (pogledajte [Onemogućavanje testova](#)). Pored *@Disabled*, JUnit Jupiter podržava i nekoliko drugih uslova zasnovanih na tagovima u paketu *org.junit.jupiter.api.condition* koji omogućava programerima da deklarativno omoguće ili onemoguće testove.

Test može biti omogućen ili onemogućen na određenom operativnom sistemu putem napomena *@EnabledOnOs* i *@DisabledOnOs*, na određenoj verziji Java Runtime Environment (JRE) putem tagova *@EnabledOnJre* i *@DisabledOnJre* ili na osnovu vrednosti svojstva sistema JVM putem tagova *@EnabledIfSystemProperty* i *@DisabledIfSystemProperty*. Vrednost koja se daje putem atributa *matches* biće interpretirana kao regularni izraz. Moguće je i uslovno izvršavanje u zavisnosti od evaluacije skripta putem tagova *@EnabledIf* ili *@DisabledIf*. Skripte se mogu pisati na JavaScript-u, ili bilo kom drugom skript jeziku za koji postoji podrška za Java Scripting API, definisanom u JSR 223.

```
@Test
@EnabledOnOs({ LINUX, MAC })
void onLinuxOrMac() {
    // ...
}
```



```

@Test
@EnabledOnJre(JAVA_8)
void onlyOnJava8() {
    // ...
}

@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void onlyOn64BitArchitectures() {
    // ...
}

@Test // Static JavaScript expression.
@EnabledIf("2 * 3 == 6")
void willBeExecuted() {
    // ...
}

```

Više o uslovnom izvršavanju pročitati u poglavlju 2.7 Userguide-a za JUnit 5 [2].

### 3.8 Redosled izvršavanja testova

Da biste kontrolisali kojim redosledom će se test metode izvršavati, označite svoju testnu klasu ili testni interfejs tagom *@TestMethodOrder* i odredite željenu implementaciju *MethodOrderer*. Možete implementirati sopstveni *MethodOrderer* ili koristiti jednu od sledećih ugrađenih *MethodOrderer*:

- *Alphanumeric*: sortira metode ispitivanja alfa-numerički na osnovu njihovih imena i formalnih lista parametara.
- *OrderAnnotation*: sortira metode ispitivanja numerički na osnovu vrednosti navedenih preko taga *@Order*.
- *Random*: naručuje metode ispitivanja pseudo-nasumično

```

import org.junit.jupiter.api.MethodOrderer.OrderAnnotation;
import org.junit.jupiter.api.Order;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestMethodOrder;

@TestMethodOrder(OrderAnnotation.class)
class OrderedTestsDemo {

    @Test
    @Order(1)
    void nullValues() {
        // perform assertions against null values
    }

    @Test
    @Order(2)
    void emptyValues() {
        // perform assertions against empty values
    }
}

```

### 3.9 Ugnježdeni testovi

Ugnježdeni testovi, označeni tagom *@Nested* pružaju ispitivaču više mogućnosti da izrazi odnos između nekoliko grupa testova. Detaljan primer dat je u nastavku.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.EmptyStackException;
import java.util.Stack;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("throws EmptyStackException when popped")
        void throwsExceptionWhenPopped() {
            assertThrows(EmptyStackException.class, stack::pop);
        }

        @Test
        @DisplayName("throws EmptyStackException when peeked")
        void throwsExceptionWhenPeeked() {
            assertThrows(EmptyStackException.class, stack::peek);
        }

        @Nested
        @DisplayName("after pushing an element")
        class AfterPushing {

            String anElement = "an element";

            @BeforeEach
            void pushAnElement() {
                stack.push(anElement);
            }

            @Test
            @DisplayName("it is no longer empty")
            void isNotEmpty() {
                assertFalse(stack.isEmpty());
            }
        }
    }
}
```

```

    }

    @Test
    @DisplayName("returns the element when popped and is
empty")
    void returnElementWhenPopped() {
        assertEquals(anElement, stack.pop());
        assertTrue(stack.isEmpty());
    }

    @Test
    @DisplayName("returns the element when peeked but
remains not empty")
    void returnElementWhenPeeked() {
        assertEquals(anElement, stack.peek());
        assertFalse(stack.isEmpty());
    }
}
}
}
}

```

### 3.10 Ponavljajući testovi

JUnit 5 pruža mogućnost ponavljanja testa određeni broj puta dodavanjem metode uz *@RepeatedTest* i određivanje ukupnog broja željenih ponavljanja. Svako pozivanje ponovljenog testa ponaša se poput izvođenja regularnog *@Test* metoda.

```

@RepeatedTest(10)
void repeatedTest() {
    // ...
}

```

Ponovljeni test omogućava dodavanje još nekih informacija prilikom izvršavanja ponovljenog testa. One se dodaju kao parametri uz tag *@RepeatedTest*, a to su:

- *displayName*: ispisuje ime *@RepeatedTest* metoda dobijeno putem taga *@DisplayName*.
- *currentRepetition* - broj do sad izvršenih testova uključujući i onaj koji se izvršava
- *totalRepetitions* - ukupan broj ponavljanja testa

Više o uslovnom izvršavanju pročitati u poglavlju 2.14 Userguide-a za JUnit 5 [2].

### 3.11 Parametrizovani testovi

Parameterizovani testovi omogućavaju pokretanje testa više puta sa različitim argumentima. Deklarisani su baš kao i obični *@Test* metodi, ali umesto pomenutog taga, koriste tag *@ParametrizedTest*. Pored toga, morate deklarirati barem jedan izvor koji će pružiti argumente za svako pozivanje, a zatim koristiti te argumente u test metodi.

Sledeći primer prikazuje parametrizovani test koji koristi tag *@ValueSource* da bi odredio niz stringova kao izvor argumenata.

```

@ParameterizedTest
@ValueSource(strings = { "radar", "ana", "kapak" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}

```

Između tagova `@ParameterizedTest` i `@ValueSource` mogu se naći neki od sledećih tagova koji omogućavaju dodatne tipove argumenta.

- `@NullSource`: pruža jedinstven null argument označenoj metodi `@ParameterizedTest`.
- `@EmptySource`: pruža jedan prazan argument označenoj metodi `@ParameterizedTest` za parametre sledećih tipova: `java.lang.String`, `java.util.List`, `java.util.Set`, `java.util.Map`, primitivni nizovi (npr. `int []`, `char []` itd.), nizovi objekata (npr. `String []`, `Integer []` itd.).
- `@NullAndEmptySource`: kompozitni tag koji kombinuje funkcionalnost `@NullSource` i `@EmptySource`.
- `@EnumSource`: pruža pogodan način korišćenja enumerisanih konstanti. Tag daje opcionalni parametar imena koji vam omogućava da odredite koje konstante će se koristiti.
- `@CsvSource`: omogućava vam da izrazite spisk argumenta kao vrednosti odvojene zarezima (npr., String literal).

```

@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertTrue(argument > 0 && argument < 4);
}

@ParameterizedTest
@NullSource
@EmptySource
@ValueSource(strings = { " ", " ", "\t", "\n" })
void nullEmptyAndBlankStrings(String text) {
    assertTrue(text == null || text.trim().isEmpty());
}

@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithEnumSource(TimeUnit timeUnit) {
    assertNotNull(timeUnit);
}

@ParameterizedTest
@CsvSource({
    "apple, 1",
    "banana, 2",
    "'lemon, lime', 0xF1"
})
void testWithCsvSource(String fruit, int rank) {
    assertNotNull(fruit);
    assertEquals(0, rank);
}

```

Više o uslovnom izvršavanju pročitati u poglavlju 2.15 Userguide-a za JUnit 5 [2].

### 3.12 Dinamički testovi

Za razliku od `@Test` metoda, `@TestFactory` metoda sama po sebi nije test slučaj, već je fabrika za test slučajeve. Dakle, dinamički test je proizvod fabrike. Tehnički gledano, metoda `@TestFactory` vraća jedan *DynamicNode* ili *Stream*, *Collection*, *Iterable*, *Iterator* ili niz instanci *DynamicNode*. Podklase klase *DynamicNode* su *DynamicContainer* i *DynamicTest*.

```
@TestFactory
Stream<DynamicTest> dynamicTestsFromIntStream() {
    // Generates tests for the first 10 even integers.
    return IntStream.iterate(0, n -> n + 2).limit(10)
        .mapToObj(n -> dynamicTest("test" + n, () ->
            assertTrue(n % 2 == 0)));
}
```

Više o uslovnom izvršavanju pročitati u poglavlju 2.17 Userguide-a za JUnit 5 [2].

## 4 Podrška Eclipse razvojnog okruženja za JUnit 5

JUnit testove možete kreirati ručno, ali Eclipse podržava kreiranje JUnit testova putem čarobnjaka. Za potrebe ovog rada korišćen je [Eclipse 2019-09](#).

Za potrebe prikaza korišćić se sledeća klasa.

```
package calculator;

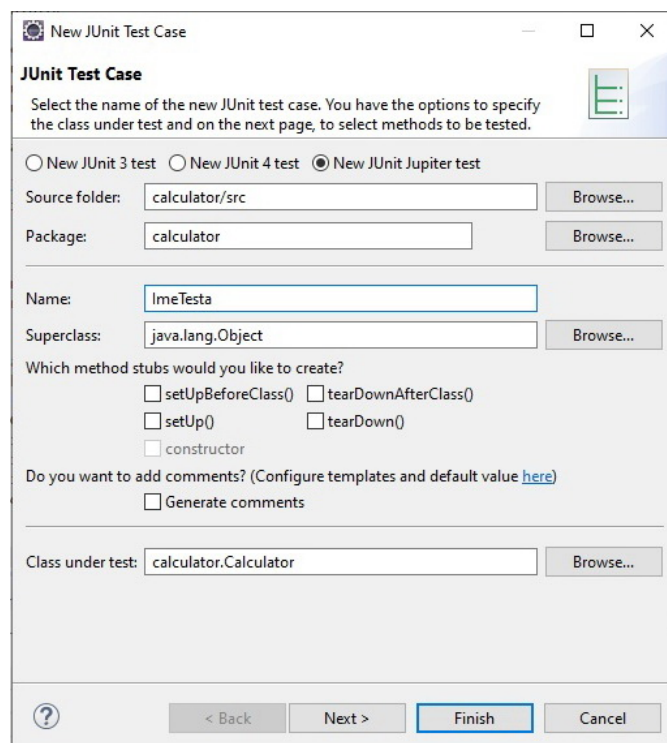
public class Calculator {

    public int add(int a, int b) {
        return a+b;
    }

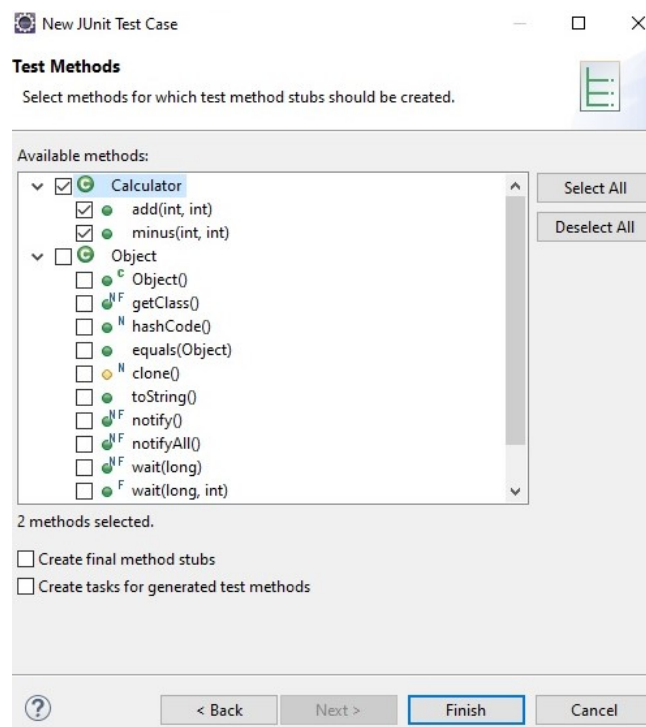
    public int minus(int a, int b) {
        return a-b;
    }
}
```

Na primer, da biste kreirali JUnit test ili testnu klasu za postojeću klasu, kliknite desnim tasterom miša željenu klasu u prikazu Package Explorer-a, i izaberite New>JUnit Test Case.

Alternativno, možete doći do čarobnjaka za kreiranje testova preko File>New>JUnit Test Case.



Zatim ćemo odabrati koje metode iz date klase(a) želimo da obuhvati naš test.



Zatim pišemo testove. Za ovaj primer iskorišćena je sledeća test klasa.

```

package calculator;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

class CalculatorTest {

    @BeforeAll
    static void start() {

    }

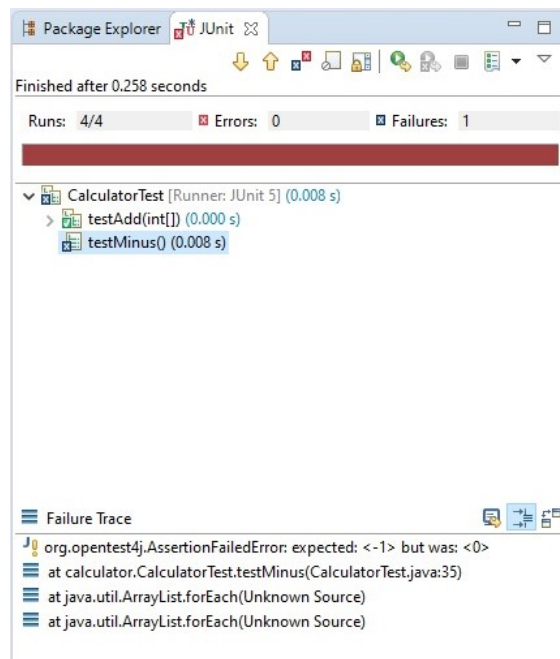
    static int[][] data() {
        return new int[][] { { 1 , 2, 3 }, { 15, 3, 18 }, { 121,
4, 125 } };
    }

    @ParameterizedTest
    @MethodSource(value = "data")
    void testAdd(int[] data) {
        Calculator tester = new Calculator();
        int m1 = data[0];
        int m2 = data[1];
        int expected = data[2];
        assertEquals(expected, tester.add(m1, m2));
    }

    @Test
    void testMinus()
    {
        Calculator tester = new Calculator();
        //Test will fail
        Assertions.assertEquals(-1, tester.minus(2, 2));
    }
}

```

Eclipse IDE također pruža podršku za interaktivno izvršavanje testova. Da biste pokrenuli test, odaberite klasu testa, kliknite desnim tasterom miša na nju i izaberite Run-as JUnit Test. Ovo pokreće JUnit i izvršava sve test metode u ovoj klasi.



Za naš konkretan primer test metoda će proći, dok će druga pasti, što je označeno crvenom linijom.

Srećno sa pisanjem vaših JUnit testova!

## Literatura

- [1] Milena Vujošević Jančić. *Dinamička analiza softvera*. 2019. URL: [http://www.verifikijasoftvera.matf.bg.ac.rs/vs/predavanja/02\\_testiranje/02\\_testiranje.pdf](http://www.verifikijasoftvera.matf.bg.ac.rs/vs/predavanja/02_testiranje/02_testiranje.pdf).
- [2] *JUnit 5 User Guide*. on-line at: <https://junit.org/junit5/docs/current/user-guide/>. 2019.
- [3] Olivera Popović. *Unit Testing in Java with JUnit 5*. on-line at: <https://stackabuse.com/unit-testing-in-java-with-junit-5/>. 2019.
- [4] Lars Vogel. *Unit Testing with JUnit - Tutorial*. on-line at: <https://www.vogella.com/tutorials/JUnit/article.html>. 2016.