

TESTIRANJE SOFTVERA - VEŽBE 03

JUNIT 5

JUNIT 5

- ▶ JUnit je jedan od unit-testing framework-a koji se koristi u Java ekosistemu
- ▶ JUnit 5 je nova verzija koja se sastoji iz 3 modula:
 1. JUnit Platform
 2. JUnit Jupiter
 3. JUnit Vintage
- ▶ **Napomena:** JUnit 5 zahteva minimalno Java JDK 8

DEPENDENCY

The screenshot shows the mvnrepository.com search results for the query 'jupiter'. The page has a sidebar on the left with filters for Repository, Group, Category, and License. The main content area displays a list of search results. The third result, 'JUnit Jupiter (Aggregator)', is highlighted with a red box. To the right of this box, a red text annotation reads: 'Preporuka - automatski povlači sve ostale JUnit Jupiter zavisnosti'. The search results list includes:

- 1. **JUnit Jupiter API** (10,853 usages) - org.junit.jupiter » junit-jupiter-api
- 2. **JUnit Jupiter Engine** (10,659 usages) - org.junit.jupiter » junit-jupiter-engine
- 3. **JUnit Jupiter (Aggregator)** (4,724 usages) - org.junit.jupiter » junit-jupiter
- 4. **JUnit Jupiter Params** (4,157 usages) - org.junit.jupiter » junit-jupiter-params

Dependency koji je potrebno uključiti u pom.xml file
Maven projekta

```
Maven  Gradle  Gradle (Short)  Gradle (Kotlin)  SBT  Ivy  Grape  Leiningen  Builder

<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter -->
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.9.0</version>
  <scope>test</scope>
</dependency>
```

KONVENCIJA IMENOVANJA

- ▶ Build Alati, poput Maven-a koriste paterne kako bi odredili da li je neka Java klasa test klasa ili ne
- ▶ Klase koje će Maven smatrati test klasama:
 1. sve poddirektorijume i sva imena Java datoteka koja počinju sa Test
 2. sve poddirektorijume i sva imena Java datoteka koja se završavaju sa Test
 3. sve poddirektorijume i sva imena Java datoteka koja se završavaju sa Tests
 4. sve poddirektorijume i sva imena Java datoteka koja se završavaju sa TestCase

**/*Test*.java	1
**/*Test.java	2
**/*Tests.java	3
**/*TestCase.java	4

OSNOVNI POJMOVI

- ▶ `@Test` anotacijom označavamo test metode
- ▶ Svaka klasa koja ima bar jednu test metodu se smatra test klasom
- ▶ `@DisplayName` anotacijom definišemo proizvoljno ime test metode ili klase (po default-u će se prilikom izvršavanja testova prikazivati njihovi Java nazivi)
- ▶ Praktičan način za linkovanje ka određenoj korisničkoj priči ili test scenariju koji postoji u sistemu
- ▶ Unutar test klase možemo pokretati pojedinačne test metode (play znak pored definicije metode) ili sve metode jedne test klase (play znak pored definicije klase)
- ▶ **Napomena:** voditi računa o poreklu anotacija

```
3 import org.junit.jupiter.api.*;
4
5 public class FirstTestClass {
6
7     @Test
8     void firstMethod() {
9         System.out.println("This is the first test method");
10    }
11
12    @Test
13    @DisplayName("US1234 - TC12 - this method is the second one")
14    void secondMethod() {
15        System.out.println("This is the second test method");
16    }
17 }
```

LIFECYCLE METODE

- ▶ Test metode često zahtevaju da se određeni deo posla izvrši pre (setup, init), odnosno posle (cleanup, teardown) izvršavanja testova
- ▶ Lifecycle metodom se smatra bilo koja metoda koja je anotirana sa `@BeforeAll`, `@AfterAll`, `@BeforeEach` ili `@AfterEach`
- ▶ `@BeforeAll`, `@AfterAll`
 - ▶ Metoda se izvršava pre odnosno posle bilo koje test metode određene test klase
- ▶ `@BeforeEach`, `@AfterEach`
 - ▶ Metoda se izvršava pre odnosno posle svake pojedinačne test metode određene test klase
- ▶ `@TestInstance(Lifecycle.PER_CLASS)` anotacijom označavamo da će sve metode jedne klase deliti test instancu
- ▶ Na ovaj način je moguće definisati `@BeforeAll` i `@AfterAll` anotacije nad nestatičkim metodama

```

5  @TestInstance(TestInstance.Lifecycle.PER_CLASS)
6  public class FirstTestClass {
7
8      @BeforeAll
9      void beforeAll() {
10         System.out.println("--This is the before All method");
11     }
12
13     @BeforeEach
14     void beforeEach() {
15         System.out.println("----This is the before Each method");
16     }
17
18     @AfterAll
19     void afterAll() {
20         System.out.println("--This is the after All method");
21     }
22
23     @AfterEach
24     void afterEach() {
25         System.out.println("----This is the after Each method");
26     }
27
28     @Test
29     void firstMethod() {
30         System.out.println("This is the first test method");
31     }
32
33     @Test
34     @DisplayName("US1234 - TC12 - this method is the second one")
35     void secondMethod() {
36         System.out.println("This is the second test method");
37     }
38 }

```

PARAMETRIZOVANI TESTOVI

- ▶ Parametrizacija testova omogućavaju višestruko pokretanje testa sa različitim argumentima
- ▶ Da bi se test parametrizovao, potrebno je definisati izvor podaka
 - ▶ `@ValueSource`
 - ▶ `@EnumSource`
 - ▶ `@MethodSource`
 - ▶ `@CsvSource`
 - ▶ `@CsvFileSource`
 - ▶ `@ArgumentsSource`
- ▶ I samu test metodu anotirati sa `@ParametrizedTest`
- ▶ **Napomena:** ova grupa anotacija dolazi iz paketa `org.junit.jupiter.params` (automatski je uključen ako ste koristili Jupiter Aggregator)

VALUE SOURCE

- ▶ `@ValueSource` anotacija omogućava definisanje jednog argumenta test metode
- ▶ Potrebno je definisati niz vrednosti
- ▶ U cilju provere graničnih vrednosti definisane su sledeće anotacije

- ▶ `@NullSource`

- ▶ Pokreće test sa null argumentom
- ▶ Ne može se koristiti za primitivne tipove podataka

- ▶ `@EmptySource`

- ▶ Pokreće test sa empty argumentom
- ▶ Može se koristiti za sledeće tipove `String`, `List`, `Set`, `Map`, primitivne nizove (`int[]`, `char[][]`), nizove objekata (`String[]`, `Integer[][]`)

- ▶ `@NullAndEmptySource`

```

13
14 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
15 >> public class ParameterizedTests {
16
17     @ParameterizedTest(name = "Run: {index} - value: {arguments}")
18     @ValueSource(ints = {1, 5, 6})
19     > void intValues(int theParam) {
20         System.out.println("theParam = " + theParam);
21     }
22
23     @ParameterizedTest
24     @NullAndEmptySource
25     @ValueSource(strings = {"firstString", "secondString"})
26     > void stringValues(String theParam) {
27         System.out.println("theParam = " + theParam);
28     }

```

```

(m) ints default {}
(m) strings default {}
(m) booleans default {}
(m) bytes default {}
(m) chars default {}
(m) classes default {}
(m) doubles default {}
(m) floats default {}
(m) longs default {}
(m) shorts default {}

```


CSV SOURCE

- ▶ `@CsvSource` anotacija omogućava definisanje više argumenta test metode
- ▶ Potrebno je definisati niz vrednosti, pri čemu se svaka torka CSV-a navodi unutar dvostrukih navodnika
- ▶ Pojedinačni parametri se odbijaju zarezom kao defaultnim delimiterom
- ▶ Delimiter može biti promenjen korišćenjem **delimiter** ključa (kao vrednost prima karakter)
- ▶ Broj string vrednosti unutar dvostrukih navodnika mora da se poklapa sa brojem definisanih parametara test metode
- ▶ Ukoliko je potrebno da unutar vrednosti parametrizovanog testa imate zarez, potrebno je da celu vrednost postavite unutar jednostrukih navodnika

```

31  @ParameterizedTest
32  @CsvSource(value = {"steve,rogers", "captain,marvel", "bucky,barnes"})
33  ▶ void csvSource_StringString(String param1, String param2) {
34      System.out.println("param1 = " + param1 + ", param2 = " + param2);
35  }
36
37  @ParameterizedTest
38  @CsvSource(value = {"steve,32,true", "captain,21,false", "bucky,5,true"})
39  ▶ void csvSource_StringIntBoolean(String param1, int param2, boolean param3) {
40      System.out.println("param1 = " + param1 + ", param2 = " + param2 +
41          ", param3 = " + param3);
42  }
43
44  @ParameterizedTest
45  @CsvSource(value = {"captain america,'steve,rogers'", "winter soldier," +
46      "'bucky,barnes'"})
47  ▶ void csvSource_StringWithComa(String param1, String param2) {
48      System.out.println("param1 = " + param1 + ", param2 = " + param2);
49  }
50
51  @ParameterizedTest
52  @CsvSource(value = {"steve?rogers", "bucky?barnes"}, delimiter = '?')
53  ▶ void csvSource_StringWithDiffDelimiter(String param1, String param2) {
54      System.out.println("param1 = " + param1 + ", param2 = " + param2);
55  }

```

CSV FILE SOURCE

- ▶ `@CsvFileSource` anotacija omogućava definisanje parametara test metode na osnovu naziva CSV fajla u kom se nalaze test podaci
- ▶ Svaki red CSV fajla će izazvati pokretanje test metode
- ▶ `files` ključu je moguće proslediti više od jednog CSV fajla
- ▶ Ukoliko CSV fajla sadrži header, moguće ga je ignorisati dodavanjem `numLinesToSkip` ključa
- ▶ Ukoliko test fajl sadrži string vrednosti, prazan string se prosleđuje unutar praznih dvostrukih navodnika, dok se null vrednost prosleđuje ne definisanjem parametra
- ▶ `delimiterString` ključ omogućava definisanje string vrednosti kao delimitera

```

57
58
59
60
61 ▶
62
63
64
65
66
67
68
69
70
71
72
73 ▶
74
75
76
77
78
79
80

@ParameterizedTest
@CsvFileSource(files = {"src/test/resources/params/shoppinglist.csv",
    "src/test/resources/params/shoppinglist2.csv"},
    numLinesToSkip = 1)
void csvFileSource_StringDoubleIntStringString(String name,
    double price,
    int qty, String uom,
    String provider) {
    System.out.println("name = " + name + ", price = " + price +
        ", qty = " + qty + ", uom = " + uom + ", provider = " + provider);
}

@ParameterizedTest
@CsvFileSource(files = "src/test/resources/params/shoppinglist3.csv",
    numLinesToSkip = 1, delimiterString = "___")
void csvFileSource_StringDoubleIntStringStringSpecifiedDelimiter(String name,
    double price,
    int qty,
    String uom,
    String provider) {
    System.out.println("name = " + name + ", price = " + price +
        ", qty = " + qty + ", uom = " + uom + ", provider = " + provider);
}

```

- ```

82 @ParameterizedTest
83 @MethodSource(value = "sourceString")
84 void methodSource_String(String param1) {
85 System.out.println("param1 = " + param1);
86 }
87
88 @ParameterizedTest
89 @MethodSource(value = "sourceStringAsStream")
90 void methodSource_StringStream(String param1) {
91 System.out.println("param1 = " + param1);
92 }
93
94 @ParameterizedTest
95 @MethodSource(value = "sourceList_StringDouble")
96 void methodSource_StringDoubleList(String param1, double param2) {
97 System.out.println("param1 = " + param1 + ", param2 = " + param2);
98 }
99
100 @ParameterizedTest
101 @MethodSource(value = "junit5tests.ParamProvider#sourceStream_StringDouble")
102 void methodSource_StringDoubleStream(String param1, double param2) {
103 System.out.println("param1 = " + param1 + ", param2 = " + param2);
104 }
105
106 1 usage
107 List<String> sourceString() {
108 //processing done here
109 return Arrays.asList("tomato", "carrot", "cabbage");
110 }
111
112 1 usage
113 Stream<String> sourceStringAsStream() {
114 //processing
115 return Stream.of(...values: "beetroot", "apple", "pear");
116 }
117
118 1 usage
119 List<Arguments> sourceList_StringDouble() {
120 //processing
121 return Arrays.asList(arguments("tomato", 2.0),
122 arguments("carrot", 4.5), arguments(
123 "cabbage", 7.8));
124 }

```

# TEST RUN ORDER

- ▶ Podrazumevano, Junit 5 testne klase i metode izvršava korišćenjem algoritma čiji je izlaz deterministički. Ovo osigurava da naredna pokretanja testnog paketa izvršavaju testne klase i metode u istom redosledu, čime se omogućavaju ponovljive izrade
- ▶ Iako pravi jedinični testovi obično ne treba da se oslanjaju na redosled kojim se izvršavaju, postoje slučajevi kada je neophodno primeniti redosled izvršenja specifične metode testa – na primer, kada se pišu integracioni test ili sistemski testovi
- ▶ `@TestMethodOrder` anotacija omogućava promenu podrazumevanog načina izvršavanja testova:
  1. `MethodOrderer.DisplayName`
  2. `MethodOrderer.MethodName`
  3. `MethodOrderer.OrderAnnotation`
    - ▶ Test metode se anotiraju `@Order` anotacijom - manji broj označava veći prioritet
  4. `MethodOrderer.Random`

# ASSUMPTIONS

- ▶ Pretpostavke se koriste za pokretanje testova samo ako su ispunjeni određeni uslovi. Ovo se obično koriste za spoljne uslove koji su potrebni da bi se test pravilno odvijao, ali koji nisu direktno povezani sa onim što se testira.
- ▶ Postoje 3 vrste pretpostavki:
  - ▶ `assumeTrue()`
  - ▶ `assumeFalse()`
  - ▶ `assumingThat()`
- ▶ Ukoliko pretpostavka nije ispunjena, JUnit baca `TestAbortedException` i test metoda biva preskočena

```
10 >> public class AssumptionsTest {
11
12 @ParameterizedTest(name = "Run: {index} - value: {arguments}")
13 @ValueSource(ints = {1, 5, 6})
14 void intValues(int theParam) {
15 assumeTrue(assumption: theParam > 4);
16 System.out.println("theParam = " + theParam);
17 }
18
19 @ParameterizedTest
20 @CsvSource(value = {"steve,rogers", "captain,marvel", "bucky,barnes"})
21 void csvSource_StringString(String param1, String param2) {
22 assumeFalse(param1.equals("steve"), message: "The assumption failed for the " +
23 "following param2: " + param2);
24 System.out.println("param1 = " + param1 + ", param2 = " + param2);
25 }
```

Poruka koja će biti ispisana  
ukoliko test bude preskočen

# DISABLING TEST

- ▶ Test klase i metode je po potrebi moguće isključiti iz izvršavanja korišćenjem `@Disabled` anotacije
- ▶ Testovi anotirani na ovaj način će u krajnjem izveštaju biti preskočeni (ignorirani)
- ▶ Praksa je da se uz svako isključivanje testa navede i razlog
- ▶ Testove je moguće i isključivati i na osnovu različitih uslova, npr:
  - ▶ Tipa operativnog sistema
  - ▶ Sistemskih vrednosti
  - ▶ Vrednosti proizvoljnih metoda
- ▶ **Napomena:** Za sve prikazano postoje i `Enabled` metode

```

12 @ExtendWith(Listener.class)
13 >> public class DisabledEnabledTest {
14
15 @Test
16 @Disabled(value = "Disabled for demo of @Disabled")
17 void firstTest() { System.out.println("This is the first test method"); }
20
21 @Test
22 @DisabledOnOs(value = OS.WINDOWS, disabledReason = "Disabled for demo of " +
23 "@DisabledOnOs")
24 void secondTest() { System.out.println("This is the second test method"); }
27
28 @Test
29 @DisabledIfSystemProperty(named = "env", matches = "staging",
30 disabledReason = "Disabled for demo of @DisabledIfSystemProperty")
31 void thirdTest() { System.out.println("This is the third test method"); }
34
35 @Test
36 @DisabledIf(value = "provider", disabledReason = "Disabled for demo of @DisabledIf")
37 void fourthTest() { System.out.println("This is the fourth test method"); }
40
41 @Test
42 void fifthTest() { System.out.println("This is the fifth test method"); }
45
46 1 usage
47 boolean provider() {
48 return LocalDateTime.now().getDayOfWeek().equals(DayOfWeek.WEDNESDAY);
49 }

```

Moguće je koristiti regex izraze

# REPEATING TEST

- ▶ JUnit Jupiter pruža mogućnost ponavljanja testa određeni broj puta označavanjem metode sa `@RepeatedTest` i navođenjem ukupnog broja željenih ponavljanja
- ▶ Svako pozivanje ponovljenog testa ponaša se kao izvršavanje obične `@Test` metode
- ▶ Pozdrazumevani šablon za imenovanje repetativnih metoda je

```
"repetition {currentRepetition} of
 {totalRepetitions}"
```

- ▶ Moguće je koristiti sledeće promenljive, kako bi se promenio podrazumevani naziv
  - ▶ `{displayName}`
  - ▶ `{currentRepetition}`
  - ▶ `{totalRepetitions}`
- ▶ Ukoliko je potrebno prethodne promenljive koristiti u samim test metodama, potrebno ih je parametrizovati `RepetitionInfo` objektom

```

8 >> public class RepeatedTests {
9
10 @RepeatedTest(5)
11 void firstRepeatedMethod() {
12 System.out.println("We are repeating this test");
13 }
14
15 @RepeatedTest(value = 3, name = "Running repetition: {currentRepetition}." +
16 " Total is: {totalRepetitions}")
17 @DisplayName("This is a repeated test method")
18 void secondRepeatedMethod() {
19 System.out.println("We are repeating a new test");
20 }
21
22 @RepeatedTest(3)
23 @Assumptions({
24 assumption: repetitionInfo.getCurrentRepetition() == 3,
25 () -> System.out.println("This code only runs for repetition " +
26 "3"));
27 }
28 }
29

```



# TAGS

- ▶ Tagovi nam omogućuju da test klase i metode svrstamo u određene grupe
- ▶ Na ovaj način je moguće pokretanje ili isključivanje određene grupe testova na osnovu njihove pripadnosti npr. određenim featurima
- ▶ @Tag ima određena ograničenja:
  - ▶ Vrednost može biti samo string
  - ▶ Ne može biti null vrednost
  - ▶ Ne može biti empty
  - ▶ Ne može sadržati sledeće karaktere: (), &, |, !
  - ▶ Ukoliko je potrebno da test pripada u više grupa, dodaju se nove @Tag anotacije

```
30 @Test
31 @Tag("sanity")
32 ▶ void firstMethod() {
33 System.out.println("This is the first test method");
34 }
35
36 @Test
37 @Tag("sanity")
38 @Tag("acceptance")
39 @DisplayName("US1234 - TC12 - this method is the second one")
40 ▶ void secondMethod() {
41 System.out.println("This is the second test method");
42 }
43
44 @Test
45 @Tag("acceptance")
46 ▶ void thirdMethod() {
47 System.out.println("This is the third test method");
48 }
```



# TIMEOUT

- ▶ `@Timeout` anotacija omogućava da se odredi maksimalno trajanje izvršavanja test metode. Ukoliko izvršavanje metode premaši definisano vreme, test metoda se smatra neuspešnom.
- ▶ Pozdrazumevana jedinica vremena je sekunda, međutim ovo je moguće promeniti postavljanjem `unit` ključa na neku od vrednosti `TimeUnit` enumeracije
- ▶ Pored test metoda, moguće je postavljanje anotacije i iznad `LifeCycle` metoda

```
public class TimeoutTest {

 @BeforeEach
 @Timeout(5)
 void setUp() {
 // fails if execution time exceeds 5 seconds
 }

 @Test
 @Timeout(value = 500, unit = TimeUnit.MILLISECONDS)
 void failsIfExecutionTimeExceeds500Milliseconds() {
 Thread.sleep(600);
 }
}
```

# ASSERTIONS

- ▶ Asertacije predstavljaju ključnu funkcionalnost svakog testing frameworka kojim proveravamo rezultate testa
- ▶ Asertacija je deo kod u kojem se zahteva da određeni izlaz bude istinit
- ▶ Ukoliko ne dođe do poklapanja, test metoda baca `AssertionFailedError`
- ▶ Osnovni oblik asertacije

```
assertEquals(expected, actual)
```

- ▶ Prvi argument je vrednost koju korisnik očekuje kao povratnu vrednost metode koju testira. Može biti dobijena iz specifikacije ili na osnovu ličnog iskustva
- ▶ Drugi argument je stvarna vrednost, koja će uvek biti posledica pozivanja metode koju testiramo
- ▶ Broj asertacija je moguće povećati dodavanjem Hamcrest biblioteke (<https://mvnrepository.com/artifact/org.hamcrest/hamcrest/2.2>)

| Assert statement           | Example                                                                              |
|----------------------------|--------------------------------------------------------------------------------------|
| <code>assertEquals</code>  | <code>assertEquals(4, calculator.multiply(2, 2), "optional failure message");</code> |
| <code>assertTrue</code>    | <code>assertTrue('a' &lt; 'b', () → "optional failure message");</code>              |
| <code>assertFalse</code>   | <code>assertFalse('a' &gt; 'b', () → "optional failure message");</code>             |
| <code>assertNotNull</code> | <code>assertNotNull(yourObject, "optional failure message");</code>                  |
| <code>assertNull</code>    | <code>assertNull(yourObject, "optional failure message");</code>                     |

```

16 @ExtendWith(Listener.class)
17 >> public class AssertionsTest {
18
19 @Test
20 > void assertEqualsTest() {
21 assertEquals(expected: "firstString", actual: "secondString", message: "The String " +
22 "values were not equal");
23 }
24
25 @Test
26 > void assertEqualsListTest() {
27 List<String> expectedValues = Arrays.asList("firstString",
28 "secondString", "thirdString");
29 List<String> actualValues = Arrays.asList("firstString",
30 "secondString");
31 assertEquals(expectedValues, actualValues);
32 }
33
34 @Test
35 > void assertArraysEqualsTest() {
36 int[] expectedValues = {1, 5, 3};
37 int[] actualValues = {1, 2, 3};
38 assertEquals(expectedValues, actualValues);
39 }
40
41 @Test
42 > void assertTrueTest() {
43 assertFalse(condition: false);
44 assertTrue(condition: false, message: "This boolean condition did not evaluate to true");
45 }

```

---

# RUNNING TEST

- ▶ Test klase i metode je moguće pokretati na nekoliko načina

## 1. Iz razvojnog okruženja

- ▶ Klikom na test metodu
- ▶ Klikom na test klasu
- ▶ Pokretanjem paketa sa test klasama
- ▶ Definisanjem test paketa (Suite)

# TEST SUIT

- ▶ Ukoliko je potrebno pokrenuti samo određene testove moguće je to odraditi definisanje test paketa
- ▶ Paketi se kreiraju putem `@Suite` anotacije koja se postavlja iznad definicije klase
- ▶ Test paketi podržavaju i anotacije kojim je moguće uključiti ili isključiti pojedinačne testove
  - ▶ `@SelectClasses`
  - ▶ `@SelectPackages`
  - ▶ `@IncludePackages`
  - ▶ `@ExcludePackages`
  - ▶ `@IncludeClassNamePatterns`
  - ▶ `@ExcludeClassNamePatterns`
  - ▶ `@IncludeTags`
  - ▶ `@ExcludeTags`

```
@Suite
@SelectPackages("junit5tests")
@IncludeClassNamePatterns({"^.*Class?$"})
@IncludeTags("acceptance")
public class TestSuit {
}
```

# RUNNING TEST

- ▶ Test klase i metode je moguće pokretati na nekoliko načina

## 2. Korišćenjem Maven-a

- ▶ Maven po pravilu pokreće samo one test klase koje počinju ili završavaju sa Test(s)
- ▶ Ovo pravilo je moguće promeniti dodavanjem nove konfiguracije

The screenshot shows an IDE interface with three main components: a code editor, a Maven lifecycle tree, and a debug console.

**Code Editor:** Displays the Maven `pom.xml` file. The `maven-surefire-plugin` configuration is highlighted with a red box:

```
</plugin>
<!-- default lifecycle, jar packaging: see https://maven.ap
<plugin>
 <artifactId>maven-resources-plugin</artifactId>
 <version>3.0.2</version>
</plugin>
<plugin>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.8.0</version>
</plugin>
<plugin>
 <artifactId>maven-surefire-plugin</artifactId>
 <version>2.22.1</version>
 <configuration>
 <includes>**/*Test*.java</includes>
 </configuration>
</plugin>
<plugin>
 <artifactId>maven-jar-plugin</artifactId>
 <version>3.0.2</version>
</plugin>
<plugin>
 <artifactId>maven-install-plugin</artifactId>
```

**Maven Lifecycle Tree:** Located on the right, it shows the Maven lifecycle goals. The `test` goal is highlighted with a red box and labeled "1. Test Lifecycle". Above it, the `execute` goal is highlighted with a red box and labeled "2. Execute Maven Goal".

**Debug Console:** At the bottom, it shows the command entered in the debug console: `mvn`, highlighted with a red box and labeled "3. Unos komandi". Below this, a list of recent commands is shown:

- `mvn test -DtestPackage=junit5tests`
- `mvn test -Dtest=FirstTestClass#firstMethod`
- `mvn test -Dtest=FirstTestClass,AssumptionsTest`
- `mvn test -Dtest=FirstTestClass`
- `mvn test -D=FirstTestClass`

---

# SUMMARY

Annotation	Description
@Test	Identifies a method as a test method.
@Disabled("reason")	Disables a test method with an option reason.
@BeforeEach	Executed before each test. Used to prepare the test environment, e.g., initialize the fields in the test class, configure the environment, etc.
@AfterEach	Executed after each test. Used to cleanup the test environment, e.g., delete temporary data, restore defaults, cleanup expensive memory structures.
@DisplayName("<Name>")	<Name> that will be displayed by the test runner. In contrast to method names the name can contain spaces to improve readability.
@RepeatedTest(<Number>)	Similar to @Test but repeats the test a <Number> of times
@BeforeAll	Annotates a static method which is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as <code>static</code> to work with JUnit.
@AfterAll	Annotates a static method which is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as <code>static</code> to work with JUnit.
@TestFactory	Annotates a method which is a Factory for creating dynamic tests
@Nested	Lets you nest inner test classes to group your tests and to have additional @BeforeEach and @AfterEach methods.
@Tag("<TagName>")	Tags a test method, tests in JUnit 5 can be filtered by tag. E.g., run only tests tagged with "fast".
@ExtendWith	Lets you register an Extension class that adds functionality to the tests