Razvoj softvera

dr.sc. Emir Mešković

VI predavanje

Sadržaj predavanja

- Anotacije
- Testiranje

Java anotacije

- prisutne od verzije Java 1.5;
- predstavljaju podatke-instance posebno definiranih tipova koji se mogu pridružiti segmentima Java koda i to:
  - paketima,
  - klasama,
  - metodima i/ili
  - poljima.
- omogućavaju umetanje metapodataka o kodu.
- Upotreba pre-definirane anotacije @Override iz java.lang paketa:

```
public class Abc {
    ...
    @Override
    public String toString() {
        ...
    }
}
```

Meta anotacije

- Pravila zadržavanja metapodataka (Retention):
  - Samo u kodu, odbacuje se nakon kompajliranja (SOURCE)
  - Ostaje kompajliran u class fajlovima (CLASS)
  - Anotacija se može koristiti tokom izvršenja koda putem API-a za introspekciju (RUNTIME)
- Ograničenja na kontekst upotrebe (Target) moguće opcije:
  - CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE i ANNOTATION_TYPE
- @Inherited signalizira da custom anotacija korištena u klasi bi trebala biti naslijeđena u njenim podklasama
- elementi koji koriste anotaciju @Documented bi trebali biti dokumentovani od strane JavaDoc

Kreiranje i upotreba novih anotacija

Definiranje nove anotacije

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Validate {
    boolean flag();
}
```

Korištenje kreirane anotacije

```
@Validate(flag=true)
public class MyClass {
    ...
}
```

Testiranje koda

- Tri nivoa testiranja koda:
  - Acceptance ili System tests:
    - Da li kompletan sistem ispravno radi?
    - Testovi uključuju i vanjske sisteme sa kojim naša aplikacija vrši interakciju.
  - Integration ili Functional tests:
    - Da li naš kod radi spram koda koji ne možemo da mijenjamo?
    - Test interakcije komponenti unutar našeg sistema.
  - Unit tests:
    - Da li naši pojedinačni objekti rade ispravno?
    - Testovi na nivou najmanjih jedinica funkcionalnosti.

Testiranje koda

- Testovi bez obzira na nivo:
  - postavljaju SUT (system under test) u početno stanje;
  - vrše određene manipulacije stanja SUT-a;
  - porede ostvareni rezultat sa očekivanim rezultatom.
- Pojedinačni test može biti
  - *uspješan,* za slučaj da ostvareni rezultat odgovara očekivanom ili
  - *neuspješan,* u svim ostalim varijantama.
- Testovi trebaju biti automatizirani
  - eventualna iznimka mogu biti testovi na sistemskom nivou.

Testiranje koda

- Testovi omogućavaju:
  - potvrdu da napisani kod radi ispravno;
  - sigurne promjene u postojećem kodu (*refaktoriranje)* radi:
    - optimizacije performansi;
    - unapređenja postojećeg dizajna (tj povećanje modularnosti)
  - sigurnost u dodavanju novog koda radi proširivanja funkcionalnosti aplikacije, a bez ugrožavanja već ispunjenih specifikacija.

Unit testing

- Predstavlja najučestali oblik testiranja kojim se utvrđuje ispravnost ponašanja najmanjih jedinica koda.
- Obično operiraju na nivou metoda pojedinačnih klasa.
- Pokrivenost testovima (*coverage*) je mjera kojom se utvrđuje koliki broj linija napisanog koda se provjerava testovima.
- Dva pristupa u pisanju unit testova:
  - Test-first
    - Napisati prvo test za novu, i to obično minimalnu, funkcionalnost.
    - Implementirati kod koji zadovoljava napisani test.
  - Test-after
    - Testovi se pišu nakon implementirane funkcionalnosti.
- Pisanje unit testova, bez obzira na pristup, je odgovornost programera koji implementira funkcionalnost sa novim kodom.

JUnit

- Biblioteka za jedinično testiranje Java koda
  - Open source licenca;
  - Integrira se sa IDE okruženjem tako da omogućava brzo pokretanje:
    - Pojedinačnih testova;
    - Testova na nivou test klase;
    - Testova nastalih integracijom više test klasa (tzv *suite)*
  - Od verzije 4 pojednostavljeno definiranje testova korištenjem anotacija.

Primjer


```
package container;
import org.junit.*;
import static org.junit.Assert.*;
public class TestMyStack {
    MyStack<Integer> stack;
    @Before
    public void setupEmptyStack() {
       stack = new MyStack<Integer>();
    }
    @Test(expected=RuntimeException.class)
    public void popEmptyStackShouldThrow() {
       stack.pop();
    }
    @Test
    public void popAfterPushShouldReturnTheSameNumber() {
       stack.push(2);
       int a = stack.pop();
       assertEquals( 2, a );
    }
    @Test
    public void popingAfterPushingTwiceShouldReturnSameNumbers() {
       stack.push(2);
       stack.push(10);
       assertTrue( stack.pop() == 10 );
       assertTrue( stack.pop() == 2 );
    }
}
```

Primjer


```
package container;

class MyStack<T> {
    private Node<T> head;

    public T pop() {
       if (head == null)
           throw new RuntimeException("Empty stack pop!");
       T temp = head.value;
       head = head.next;
       return temp;
    }

    public boolean empty() {
       return head == null;
    }

    public void push(T o) {
       head = new Node<T>(o,head);
    }

    class Node<E> {
       Node<E> next;
       E value;
       public Node(E v, Node<E> n)
       {
           next = n;
           value = v;
       }
    }
}
```

Primjer


```
package container;
import java.util.ArrayList;
import java.util.List;
import org.junit.*;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
public class TestAlgorithm {
    private MyStack<Integer> source;
    @Before
    public void initialize() {
       source = new MyStack<>();
    }
    @Test
    public void movingFromEmptySourceIsOK() {
       List<Integer> destination = new ArrayList<>();
       Algorithm.move(source,destination);
       assertThat( destination.size(), is(equalTo(0)) );
    }
    @Test
    public void shouldBeAbleToMoveIntsToListOfNumbers() {
       List<Number> destination = new ArrayList<>();
       source.push(5);
       source.push(8);
       Algorithm.move(source,destination);
       assertThat( destination.size(), is(equalTo(2)) );
    }
}
```

Primjer


```
package container;
import java.util.List;

public class Algorithm {
   public static <K> void move(MyStack<K> source, List<? super K> destination)
   {
     while (!source.empty()) {
         K element = source.pop();
         destination.add(element);
     }
   }
}
```

Primjer


```
package container;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({TestMyStack.class, TestAlgorithm.class})
public class TestAlgorithmAndMyStack {

}
```