PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET
METROPOLITAN
BEOGRAD

KI105 - JAVA 3: PROGRAMIRANJE KORISNIČKOG INTERFEJSA

Testiranje softvera sa JUnit

Lekcija 07

PRIRUČNIK ZA STUDENTE

KI105 - JAVA 3: PROGRAMIRANJE KORISNIČKOG INTERFEJSA

Lekcija 07

TESTIRANJE SOFTVERA SA JUNIT

UVOD

JUnit omogućava automatizovano testiranje metoda i klasa.

Cilj ove lekcije je da studente upozna sa testiranje programskih jedinica (klasa i metoda) pomoći JUnit radnog okvira koji omogućava automatizaciju testiranja programa u fazi razvoja.

Poglavlje 1

Jedinično testiranje softvera

JEDINIČNO TESTIRANJE

Jedinično testiranje je testiranje pojedinačnih objekata

Kada smo govorili o klasama i objektima, rekli smo da klasu pišemo uvek tako da ona bude proizvod za sebe, da možemo da tu klasu iskoristimo u bilo kom drugom proizvodu (projektu), i da bez ikakvih problema možemo da je implementiramo i primenimo, da klasa bude jedan jedinstveni proizvod. Ova osobina objektno - orijentisanog programiranja je izuzetno bitna za jedinično testiranje.

Jedinično testiranje je testiranje pojedinačnih objekata. Dakle, imamo test koji testira odgovarajuću klasu, odnosno metode koje nešto rade u okviru te klase. Najbolji alat za testiranje pojedinačnih klasa je **JUnit**. To je Framework, skup alata pomoću kojih možemo da vršimo testiranje. Naravno, testove moramo sami da napišemo. **JUnit** se isporučuje kao Jar biblioteka, sa svim potrebnim alatima za jedinično testiranje, mi tu biblioteku uvrstavamo u naš projekat, i pozivamo odgovarajuće klase iz te biblioteke da bi vršili testiranje. Jedinično testiranje možemo da izvršimo i bez **JUnit** Framework-a tako što bismo napisali novu klasu, koja bi pozivala instance klase koju testiramo, i proveravala da li ispravno radi na određenim test primerima.

- sa svim potrebnim alatima za jedinično testiranje
- Objekat je proizvod za sebe
- Test testira pojedinačni objekat
- JUnit je Framework
- Jar biblioteka

PREDNOSTI TESTIRANJA SOFTVERA

JUnit testovi, za razliku od drugih tipova testova, proveravaju sve metode koje se nalaze u okviru jedne klase

Savremeni projekti obično insistiraju na ovim jediničnim testovima. Jedinični testovi značajno povećavaju kvalitet koda, zato što je svaka pojedinačna klasa istestirana. Ako imamo testove za svaku pojedinačnu klasu lakše je čitanje koda, ako nešto ne razumemo u kodu, možemo da pogledamo testove i da vidimo šta taj deo koda radi. Ako imamo testove, onda možemo bez problema da radimo naknadne refactoring-e, izmene koda, a da se ne bojimo da ćemo narušiti neku funkciju na odnos koji postoji. Nakon izvršenog refactoring-a pustimo ponovo test i, ako je sve u redu, znači da nismo narušili prvobitnu funkciju na odnos programa.

Testovi se koriste i za pravljenje testova prihvatanja. Dakle, budućem korisniku pokazujemo da naša aplikacija zaista radi ono što on očekuje od nje. Dakle, testovi mogu da pokažu da određene metode izračunavaju tačno ono što je korisnik tražio. Jedinični testovi se odlično uklapaju u procedure izrade softwere-a, u odgovarajući ProjectPlan, u odgovarajuću kontrolu kvaliteta koda. Ako svaka klasa koju napišemo ima odgovarajuće **Junit** testove, to sve povećava pouzdanost celog sistema, možemo sa većom pouzdanošću reći da sistem radi ono što treba da radi. Samim tim je smanjena i količina bug-ova koji mogu da se pojave u kodu.

JUnit testovi, za razliku od drugih tipova testova, proveravaju sve metode koje se nalaze u okviru jedne klase koja nešto konkretno radi, bez obzira kolika je verovatnoća pokretanja te metode u realnom radu. Problem sa drugim oblicima testiranja je u tome što se neke situacije nikada ne dese prilikom testa, već tek kod klijenta. **JUnit** testovi proveravaju svaki deo koda, bez obzira na njegovu verovatnoću pojavljivanja u kodu.

Rezime:

- Veći kvalitet koda
- Lakše čitanje koda
- Moguće raditi naknadne refaktoringe
- Mogućnost pravljenja testova prihvaćanja
- Lako se uklapa u procedure izrade softvera
- Povećava pouzdanost celog sistema
- Smanjuje količinu grešaka u kodu

MANE TESTIRANJA SOFTVERA

Mane jediničnog testiranja

Pored velikih prednosti, jedinično testiranje ima i izvesnih mana. Ove mane su obično vezane za kod koji je već ranije napravljen po neobjektno orijentisanim principima, pa je nemoguće implementirati na njega jedinično testiranje koje je specijalno optimizovano i prilagođeno za objektno - orijentisano programiranje. Pored toga, dosta je teško primeniti jedinično testiranje na GUI klasi. GUI klasa je u interakciji sa korisnikom pa je teško napisati **JUnit** testove koji bi izvršili takva testiranja.

Testiraju se samo klase, a ne i okruženje u kome se radi. Npr. preko **JUnit** testiranja se testira kod Javin, ali ne i SQL naredbe baze. Takođe, neke situacije u mreži ne mogu da se istestiraju, jer se odnosi na konkretan kod u aplikaciji. Pomoću ove vrste testiranja ne možemo da istestiramo neku situaciju koja može da se desi u okruženju (nasilan nestanak struje ili sl.).

Kada imamo **JUnit** testiranje, susrešćemo se sa sporijim razvojnim ciklusom. Naime, prilikom pisanja testova, mi moramo u proseku da napišemo oko tri puta više koda u okviru testa, nego što pišemo u samom kodu koji treba da se izvršava. Ovo, naravno, zbog količine posla koji treba uraditi, dosta usporava razvojni ciklus projekta.

Rezime:

- Ne može da se primeni na GUI klase

- Testira se samo klase ne i okurženje u kom kod radi (baze podataka, mreža, ...
- Kod mora da bude rađen po strogim Objektno Orijentisanim principima
- Sporiji razvojni ciklus
- do 3x više koda

NAPOMENE PRILIKOM IZRADE TESTOVA

Korisni saveti i napomene prilikom izrade JUnit testova

Prilikom jediničnog testiranja najviše se obraća pažnja na metode koje nešto rade. *Metode koje nešto rade* obično implementiraju nekakav algoritam (metode koje određuju funkcionalnost naše aplikacije) tako da je to upravo mesto koje trebamo najviše da testiramo.

Pored metoda, možemo da testiramo i *konstruktore*, ako se u okviru konstruktora generiše nekakav funkcionalni kod. Obično se ne testiraju **get**-er i **set**-er metodi, zato što oni samo postavljaju, odnosno prosleđuju vrednost iz polja klase, tako da oni nisu zanimljivi za detaljno testiranje (oni skoro uvek rade ono što treba da rade). **Set**-er metodi često i automatski generišemo, tako da oni nisu interesantni za **JUnit** testiranje. Ne testiraju se metode koje služe samo za adaptiranje, odnosno prosleđivanje naredbe drugim metodama. Takve metode takođe nisu interesantne za testiranje.

Rezime:

- metode koje nešto rade
- mogu se testirati konstruktori
- ne testira se geteri i seteri
- ne testiraju se metode koje samo pozivaju druge metode

GREŠKE

Postupak pronalaženja grešaka u JUnit testovima

Pogrešno je misliti da, ako imamo **JUnit** testove, da je nemoguće da se pojavi bug. Naravno da mi testovima ne možemo da predvidimo sve situacije, i da se ponekad bugipak potkrade i pojavi u aplikaciji. U tom slučaju trebamo da poštujemo sledeći postupak za otkrivanje bug-a u aplikacijama koje imaju **JUnit** testove:

Prvo i osnovno što treba da znamo je da *ne smemo da diramo kod.* Prvo što radimo je pronalaženje testa koji pokazuje da ta greška postoji, i taj test onda pišemo. Takav test treba da padne, treba da pokaže da prošao (failed**)**, odnosno da projekat ne daje zadovoljavajući rezultat.

Kada smo definisali grešku, odnosno tačno je locirali, onda menjamo kod tako da ispuni zahtev testa, odnosno da prođe. Pokrene se ponovo test, i ustanovi se da greške više nema. Na taj način smo ispravili grešku u aplikaciji.

Prilikom pokretanja testova obično pokrećemo sve testove u aplikaciji. Ovo je jako dobro prilikom otklanjanja grešaka, jer smo sigurni da otklanjanje ove greške nije prouzrokovalo druge greške, tj.da ispravka programa nije nešto poremetila

U slučaju da nemamo **JUnit** testove, i da pokušavamo da ispravimo greške, to radimo obično tako što pretpostavimo da je na nekom mestu greška, pa probamo da ispravimo, onda pokrenemo aplikaciju, pa shvatimo da nije bio tu bila greška, pa onda izmenimo neko drugo mesto, pa neko treće mesto. Tim menjanjima smo možda u potpunosti uništili neke stvari koje su radile kako treba. Ništa od ovoga mi ne vidimo u tom trenutku, već dajemo takav kod klijentu, klijent to startuje i ustanovljava da sada ima više grešaka nego što je imao prvi put.

Ako koristimo **JUnit** testove, i postupak za pronalaženje grešaka koji je definisan na ovom primeru, takva situacija se nikada neće desiti.

Kada se pojavi BUG u aplikaciji postupak je sledeći.

1. kod se ne dira
2. napiše se test koji pokazuje da greška postoji (test padne)
3. promeni se kod da ispuni zahtev testa
4. pokrenu se testovi i ustanovi da greške nema.
5. prilikom pokretanja pokreću se svi testovi kako bi ustanovili da ispravka bug-a nije napravila grešku na drzgom delu koda.

ŠTA TEST TREBA DA PROVERI

Jediničan test treba da proveri da li je rezultat tačan za realne vrednosti koje će se dešavati u aplikaciji

Jediničan test treba da proveri da li je rezultat tačan za realne vrednosti koje će se dešavati u aplikaciji. Zatim, treba da proveri kako se ponaša kod za potencionalno problematične vrednosti (npr. ako mu pošaljemo nulu sa kojom nešto deli). Takođe treba da vidimo kako se naš kod ponaša na graničnim vrednostima. Interesantno je videti i kako se kod ponaša na vrednostima koje su sigurno pogrešne, šta se dešava kada je vrednost u pogrešnom formatu, šta se dešava kada fali podatak.

Test treba da proveri da li je vraćeni rezultat u pravilnom formatu, da li pripada opsegu kojem treba. Ovo je sve jako bitno prilikom pisanja testova jer, ako predvidimo sve varijante u okviru testa i znamo tačno kako se aplikacija ponaša za svaku od ovih varijanti, onda tačno znamo šta ta klasa može i kako će se ponašati, i nećemo doći u situaciju da joj prosleđujemo podatke ili da je stavljamo u situacije koje ona ne ume da razreši.

Šta se proverava? Rezime:

- Da li je rezultat tačan za realne vrednosti
- Potencionalno problematične vrednosti (np. deljenje sa nulom)
- Granične vrednosti
- Vrednosti koje su sigurno pogrešne
- Format dobijenog rezuktata
- Kad fali podatak

- Provera formata rezultata
- Opsege

PRIMER 1 - JEDNOSTAVAN PRIMER JUNIT TESTA

Kako se može sprovesti jednično testiranje?

Ovde ćemo na vrlo jednostavnom primeru pokazati kako se može sprovesti jednično testiranje. Dovoljno je da imate na vašem računaru instaliran JDK i tekst editor.

Informacije o Junit radnom okruženju možete dobiti na sajtu http://junit.org/index.html odekle (https://github.com/junit-team/junit/wiki/Download-and-Install) možete preuzeti dve JAR datoteke: ***junit.jar*** i ***hamcrest-core.jar*** i stavite ih u foleder junit-example.

Kreirajte klasu Calculator.java sa sledećim kodom:

```
public class Calculator {
   public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
   }
 }
```

Sada izvršite kompilaciiu ove klase sa komandne linije:

javac Calculator.java

I tako dobijate datoteku: **Calculator.class.**

Kreirajte test klasu za klasu Calculator.java, tj. Klasu **CalculatorTest-java:**

```
import static org.junit.Assert.assertEquals;
 import org.junit.Test;

 public class CalculatorTest {
   @Test
   public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
   }
 }
```

Sada izvršite kompilaciju ove klase sa:

javac -cp .:junit.jar CalculatorTest.java

I dobijate datoteku. **CalculatorTest.class**. Sada možete da izvršite test kucajući sledeću komandnu liniju:

java -cp .:junit-.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTes

Dobićete informaciju o uspešnosti testa.

ZADATAK 1

Samostalna provera znanja.

Pokušajte da date objašnjenje sledećeg listinga:

```
@Test
   public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
   }
```

VIDEO: PRIMENA JUNIT SA NETBEANS IDE

Working with JUNIT in NetBeans (15,25 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

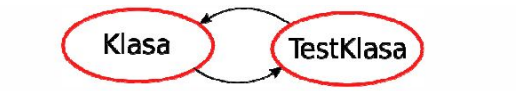
Poglavlje 2

JUnit test

DELOVI JUNIT TESTA

Odnos klase i Test klase, prikaz iz kojih delova se sastoji JUnit test kao i njegov životni ciklus (Life Cycle)

Rekli smo da su **JUnit** testovi testovi nad klasama, odnosno nad pojedinačnim klasama. To znači da uvek imamo par: Klasa koje nešto radi, i njena odgovarajuća TestKlasa koja testira ispravnost rada ove klase. Oni uvek idu u paru, i, kada govorimo o JUnit testovima, uvek govorimo o paru klase i klase koja je obrađuje, odnosno testira. Ne može da se desi da jedna ista TestKlasa testira više klasa konkretnih, ili da jedna konkretna klasa bude testirana od strane nekoliko TestKlasa. Uvek je to odnos jedan na jedan, klasa i odgovarajuća TestKlasa.



Slika 2.1 - Odnos klase i test klase

Osnovne klase koje se koriste u okviru **JUnit** testova su:

• **TestCase**, koji predstavlja pojedinačni slučaj testiranja (dakle, naša klasa nasleđuje klasu **TestCase**, i pravi pojedinačni slučaj testiranja)

• **TestSuite**, je skup klasa koje se testiraju (imamo klasu koja nasleđuje **TestSuite**, i ona u sebi može da sadrži više **TestCase**-ova),

• **TestRunner** (**BaseTestRunner**), je klasa koja obezbeđuje funkcionalnost, odnosno pokretanje testova, i

• **TestResults**, je klasa u koju se smeštaju rezultati testa.

S' obzirom da je **JUnit** testiranje manje – više automatizovano, nama će, u većini slučajeva, trebati samo **TestCase** i **TestSuite** klase, odnosno njih ćemo nasleđivati da bi napravili svoje **JUnit** testove.

Postoje tri metode u okviru **TestCase**-a, koje mi nasleđujemo i implementiramo u okviru svoje TestKlase. To su metode:

• ***setUp()***, koja nam služi da u nju postavimo osnovne postavke (da se konektujemo na bazu, ili da se povežemo na mrežu, ili da učitamo neki dokument sa diska),

• ***testXXX()***, metode koje testiraju odgovarajuće metode iz konkretne klase (njih može da bude više, i svaka **test** metoda se odnosi na neku metodu u konkretnoj klasi),

• ***tearDown()*** , je metoda koja se pokreće nakon završetka testova (u njoj možemo da zatvorimo vezu ka bazi, ili vezu ka drugom serveru, ili bilo šta što je potrebno uraditi nakon izvršenih testova).

JUNIT ASSERT

Assert metode služe da se proveri ispravnost neke vrednosti

U slučaju ***assertEquals()*** metode proveravamo da li neki broj odgovara očekivanom broju, sa određenom tolerancijom. Prvi parametar je ***message***, odnosno poruka koju možemo da pošaljemo u slučaju da taj test ne prođe. Metod ***assertTrue()*** je test pomoću koga proveravamo da li je vrednost tačna. Dakle, imamo nekakav ***expression***, izraz, i proveravamo da li taj izraz vraća ***true***. Izraz može da bude promenljiva, a može da bude poziv metode, opciono možemo da pošaljemo i poruku. Metod ***assertFalse()*** je suprotan od metoda ***AssertTrue(),*** dakle proveravamo da li nam vraća ***false***, da li nam vraća netačno.

• ***assertEquals([String message], expected, current, tolerance);***

• ***assertTrue([String message], boolean);***

• ***assertFalse([String message], boolean expression)***;

Pored toga što možemo da proveravamo vrednosti prostih tipova, moramo da imamo mehanizam i za proveru objekata. To su metode ***assertNull()*** i ***assertNotNull()***, da bi videli da li objekat postoji ili ne postoji. Prosleđujemo kao parametar, naravno, objekat za koji se pitamo da li postoji ili ne postoji, i opcionalno poruku koja će se pojaviti u slučaju da kod test padne. Onda imamo metode ***assertSame()*** i ***assertNotSame()***, odnosno da li su dva objekta isti objekat ili ta dva objekta nisu isti objekat. Takođe, pored ta dva objekta prosleđujemo i onaj opcionalni ***message***, odnosno tekst. Ovde treba obratiti pažnju na to da postoji i vrednost i kontravrednost, dakle ***assertNull(), i assertNotNull(), assertSame() i assertNotSame().***

Kada testiramo, mi se ponekad pitamo da li će naš program vratiti ***null object***zato što nije umeo da razreši neku situaciju, i to postavljamo kao odgovarajući test ***assertNull()***, dakle da li će da vrati ***null***. Iako ***null*** obično predstavlja grešku u programu, mi mu često dajemo pogrešne vrednosti prilikom testiranja, i od njega očekujemo takve rezultate.

1.***assertNull([String message], Object object);***

2. ***assertNotNull([String message], Object object);***

3. ***assertSame([String message], Object o1, Object o2***);

4. ***assertNotSame([String message], Object o1, Object o2)***;

PRIMER 2 - PRIMENA ASSERT METODA

Provera tvrdnji u programima se vrši metodima assertX(), gde je X nastavak imena koji zavisi vrste provere, što je prikazano u datom primeru.

JUnit obezbeđuje metode assertion za sve primitivne tipove podataka, za objekte i nizove. Parametri se navode tako da se prvo navede očekivana vrednost, a onda stvarna vrednost, koja se programski dobija (i na taj način proverava da li se dobija očekivana vrednost).

Prvi parametar može da bude i ***String*** poruka koja je obaveštenje o dobijenoj grešci tokom testiranja. Postoje male razlike kod assertion metoda. Metod ***assertThat()*** ima tri parametra:

- opciona poruka o grešci,
- stvarna vrednost
- Matcher objekat, tj očekivana vrednost (ovaj redosled nije kao kod ostalih assertion metoda)

Sledeći primer sadrži sve tipove assertion metoda .

Listing klase AssertTests:

```
import static org.hamcrest.CoreMatchers.allOf;
 import static org.hamcrest.CoreMatchers.anyOf;
 import static org.hamcrest.CoreMatchers.equalTo;
 import static org.hamcrest.CoreMatchers.not;
 import static org.hamcrest.CoreMatchers.sameInstance;
 import static org.hamcrest.CoreMatchers.startsWith;
 import static org.junit.Assert.assertThat;
 import static org.junit.matchers.JUnitMatchers.both;
 import static org.junit.matchers.JUnitMatchers.containsString;
 import static org.junit.matchers.JUnitMatchers.everyItem;
 import static org.junit.matchers.JUnitMatchers.hasItems;

 import java.util.Arrays;

 import org.hamcrest.core.CombinableMatcher;
 import org.junit.Test;

 public class AssertTests {
   @Test
   public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    org.junit.Assert.assertArrayEquals("failure - byte arrays not same", expected,
actual);
   }

   @Test
   public void testAssertEquals() {
```

```
    org.junit.Assert.assertEquals("failure - strings are not equal", "text",
"text");
   }

   @Test
   public void testAssertFalse() {
    org.junit.Assert.assertFalse("failure - should be false", false);
   }

   @Test
   public void testAssertNotNull() {
    org.junit.Assert.assertNotNull("should not be null", new Object());
   }

   @Test
   public void testAssertNotSame() {
    org.junit.Assert.assertNotSame("should not be same Object", new Object(), new
Object());
   }

   @Test
   public void testAssertNull() {
    org.junit.Assert.assertNull("should be null", null);
   }

   @Test
   public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    org.junit.Assert.assertSame("should be same", aNumber, aNumber);
   }

   // JUnit Matchers assertThat
   @Test
   public void testAssertThatBothContainsString() {
    org.junit.Assert.assertThat("albumen",
both(containsString("a")).and(containsString("b")));
   }

   @Test
   public void testAssertThathasItemsContainsString() {
    org.junit.Assert.assertThat(Arrays.asList("one", "two", "three"),
hasItems("one", "three"));
   }

   @Test
   public void testAssertThatEveryItemContainsString() {
    org.junit.Assert.assertThat(Arrays.asList(new String[] { "fun", "ban", "net"
}), everyItem(containsString("n")));
   }

   // Core Hamcrest Matchers with assertThat
   @Test
   public void testAssertThatHamcrestCoreMatchers() {
```

```
    assertThat("good", allOf(equalTo("good"), startsWith("good")));
    assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
    assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
    assertThat(7, not(CombinableMatcher.<Integer>
either(equalTo(3)).or(equalTo(4))));
    assertThat(new Object(), not(sameInstance(new Object())));
   }

   @Test
   public void testAssertTrue() {
    org.junit.Assert.assertTrue("failure - should be true", true);
   }
 }
```

PRIMER 3 - PRIMERI JUNIT TESTOVA

Primer testiranja stanja naloga kao i datuma

Ovde se vidi primer jedne test klase (koja se u ovom slučaju zove AccountTest) i odgovarajućeg metoda testGetBalanceOk, u kome se proverava stanje te metode u odgovarajućoj originalnoj klasi. Iz samog naziva test klase vidimo da se originalna klasa zove Account, a da se metoda koju testiramo zove GetBalanceOk.

```
import junit.framework.TestCase;
public class AccountTest extends TestCase{
public void testGetBalanceOk () {
long balance = 1000;
Account account = new Account(balance);
long result = account.getBalance();
assertEquals(balance, result);
}
}
```

Datumi su izuzetno nezgodni za testiranje, zbog toga što se oni pomeraju, dakle u trenutku kada smo pravili test bio je jedan datum, u trenutku kada pokrećemo test može da bude drugi datum, tako da ne smemo nigde u okviru testa da se referenciramo na današnji datum, uvek moramo da pravimo nekakve apsolutne datume i da pomoću njih proveravamo. Problem je još komplikovaniji ako se sam kod koji testiramo bazira na trenutnom datumu. Onda u okviru test klasa moramo da pravimo pomeranje u odnosu na trenutni datum, vodeći računa da ta pomeranja odgovaraju željenim testovima. U ovom primeru se vidi kako se prave testovi kada su datumi u pitanju. Datumi mogu da budu najkomplikovaniji delovi testa.

```
public class MyTaskTest extends TestCase
{
SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");
private MyTask ob1;
Calendar calOb2;
private MyTask ob2;
```

```
Calendar calOb3;
private MyTask ob3;
public static Test suite()
{
return new TestSuite(MyTaskTest.class);
}
public MyTaskTest(String name)
{
super(name);
}
protected void setUp()
{
ob1 = new MyTask();
ob2 = new MyTask();
ob3 = new MyTask();
calOb2 = Calendar.getInstance();
setCalendarWithoutTime(calOb2);
calOb2.roll(Calendar.DAY_OF_MONTH, 2);
calOb3 = Calendar.getInstance();
setCalendarWithoutTime (calOb3);
calOb3.roll(Calendar.DAY_OF_YEAR, -41);
ob2.setDone(new Boolean(true));
ob2.setMyTask ("Task Test");
ob2.setContact("331984");
ob2.setFinalDate(df.format(calOb2.getTime()));
ob2.setPriority("Height");
ob3.setField(0, new Boolean(false));
ob3.setField(1, "Other Task");
ob3.setField(2, "43904");
ob3.setField(3, df.format(calOb3.getTime()));
ob3.setField(4, "Low");
}
private void setCalendarWithoutTime(Calendar c)
{
c.set(Calendar.HOUR, 0);
c.set(Calendar.MINUTE, 0);
c.set(Calendar.SECOND, 0);
c.set(Calendar.MILLISECOND, 0);
}
public void testGetField()
{
assertEquals(new Boolean(false), ob1.getField(0));
assertEquals(new String(), ob1.getField(1));
assertEquals(new String(), ob1.getField(2));
assertEquals(new String(), ob1.getField(3));
assertEquals(new String(), ob1.getField(8));
assertEquals(new Boolean(true), ob2.getField(0));
assertEquals("Task Test", ob2.getField(1));
assertEquals("331984", ob2.getField(2));
assertEquals(df.format(calOb2.getTime()), ob2.getField(3));
assertEquals("Height", ob2.getField(8));
assertEquals(new Boolean(false), ob3.getField(0));
assertEquals("Other Task", ob3.getField(1));
```

```
assertEquals("43904", ob3.getField(2));
assertEquals(df.format(calOb3.getTime()), ob3.getField(3));
assertEquals("Low", ob3.getField(8));
}
public void testNoDayFinalDate()
{
long diffOb2 = calOb2.getTimeInMillis() - Calendar.getInstance().getTimeInMillis();
diffOb2 /= 86400000;
assertEquals(diffOb2, ob2. noDayFinalDate());
long diffOb3 = calOb3.getTimeInMillis() - Calendar.getInstance().getTimeInMillis();
diffOb3 /= 86400000;
assertEquals(diffOb3, ob3. noDayFinalDate());
}
}
```

PRIMER 3 - TESTIRANA KLASA

Prikazana je klasa MyTask koja je testirana prethodnim testovima

Konstruktor klase MyTask prosleđuje napunjene vrednosti u druge konstruktore iste ove klase. To znači da ova klasa nikada ne može da se pojavi bez elemenata – oni su ili podešeni na prazne **String**ove, ili se napune kroz drugi konstruktor.

```
public class MyTask extends Object implements Serializable
{
private Boolean done;
protected String obaveza;
protected String kontakt;
protected String rok;
protected String prioritet;
private Calendar danas;
protected Calendar uradjeno;
public MyTask()
{
this(new Boolean(false), new String(), new String(), new String(), new String());
}
private MyTask(Boolean done, String obaveza, String kontakt, String rok, String
prioritet) {
super();
this.done = done;
this.obaveza = obaveza;
this.kontakt = kontakt;
this.rok = rok;
this.prioritet = prioritet;
this.danas = Calendar.getInstance();
this.uradjeno = null; }
public String getPrioritet()
{
return prioritet;
}
```

```
public void setPrioritet(String prioritet)
{
this.prioritet = prioritet;
}
....
public Object getField(int i)
{
if (i == 0) return getDone();
else if (i == 1) return getObaveza();
else if (i == 2) return getKontakt();
else if (i == 3) return getRok();
else return getPrioritet();
}
public void setField(int i, Object o)
{
if (0 == i) setDone((Boolean) o);
else if (i == 1) setObaveza((String) o);
else if (i == 2) setKontakt((String) o);
else if (i == 3) setRok((String) o);
else setPrioritet((String) o);
}
public long noDayFinalDate() {
long res;
if (rok != null || rok != "") {
res = RokOdStringa().getTimeInMillis() - Calendar.getInstance().getTimeInMillis();
res /= 86400000;
return res;
}
return 0;
}
```

RAZVOJNO OKRUŽENJE & JUNIT

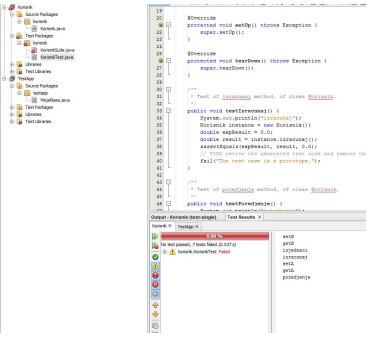
Upotreba NetBeans razvojnog alata za kreiranje Junit testova

Razvojna okruženja ponekad uvode razne olakšice, da možemo jednostavnije da napravimo **Junit** testove. U razvojno okruženje je obično ugrađen **JUnit**, tako da ne moramo posebno da ga dodajemo i da ga posebno pozivamo, već to radi razvojno okruženje umesto nas. **NetBeans**-a idu čak korak dalje, da nam ponude da umesto nas kreiraju metode, sve sa početnim nultim vrednostima. Takođe nam daju mogućnost da biramo da li hoćemo da kreirami i **setUp** i **tearDown** metodu, kao i opcionalno da li želimo da se generiše Java DOC komentari. Naš zadatak je samo da upišemo odgovarajući naziv test klase.

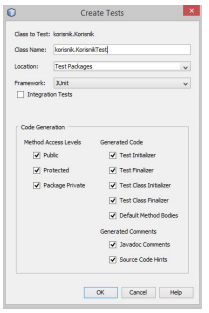
Kada prvi put kroz **NetBeans**-ov **Wizard** napravimo testove, nakon svakog testa postoji metoda **Failed** koja se izvršava uvek, odnosno svaki test izvršava kao da je pao. To je urađeno namerno, da se mi ne bismo oslonili na te osnovno automatske generisane testove, već da bismo ih izmenili i prilagodili konkretnim situacijama, da na bi došlo do slučajnog uspešnog testa. Kada pokrenemo testove, a da ništa nismo menjali, svi testovi će da se prikažu sa **Failed**, da su pali. Mi možemo to da izmenimo tako što pravimo konkretne, prave testove koje naša aplikacija treba da radi, i nakon toga možemo da pustimo izvršavanje testova. Može

da nam se desi da u okviru **testSuit**-a neki testovi prođu, a neki testovi padnu. To se vidi na ovom **ScreenShot**-u **NetBeans** a, gde se vidi da je ukupan test **Failed**, nije prošao, iako su neki od konkretnih testova prošli. Da bi celokupan test prošao, moraju svi pojedinačni testovi da prođu.

U trenutku kada uspešno propustimo test, odnosno kada su svi pojedinačni testovi koji postoje u okviru tog paketa uspešno prošli, onda je i celokupan test prošao. U donjem delu **NetBeans**-a se nalazi statistika za **JUnit** klasu **testResults**, i tu vidimo da su svi pojedinačni testovi prošli, pa je i ukupna ocena kompletnog testa **Passed**, prošao.



Slika 2.2 - Prikaz lošeg testa



Slika 2.3 - kreiranje JUnit

SPECIJALIZOVANI PROGRAMSKI ALATI ZA IZVRŠENJE JUNIT TESTOVA

Suite, Parametrized i Categories su tri specijalizovana programa za testiranje (runners) koja služe za određene specijalizovane slučajeve testiranja

Pored IDE platformi, kao i programa za testiranje (Runner programi):

1. **Suite:** Oni koji se pozivaju sa komandne linije (runnerprogrami), postoje i specijalizovani programi (suite programi) koji omogućavaju da se kreira test program za više klasa. Specifikacija je data na : http://junit.sourceforge.net/javadoc/org/junit/runners/Suite.html
2. **Parametrized**: Ovoj je standardni program za testiranje koji primenjuje tzv, parametrizovane testove. Kada se izvršava parametrizovani test neke klase, kreiraju se primerci elemanta ukrštanja test metoda i test podataka.

3. **Caregories:** Ovo je standardni program za testiranje koji omogućava da se pojedini podskupovi testova obeleže (tagovima) pojedinim kategorijama kojim se označavaju za testiranje ili za isključivanje od testiranja koje se vrši.

PRIMER 4 - SUITE PROGRAM ZA TESTOVE AGREGACIJA

Suite omogućava da jednim testom obuhvatite testiranje više klasa.

Upotreba Suite omogućava da kreirate test kojim testirate više klasa. Da bi ga koristili, potrebno je da klasu koju testirate označite sa:

- ***@RunWith(Suite.class)*** i sa
- ***@SuiteClasses(TestClass1.class, ...).***

Kada izvršavate ovu klasu, ona će izvršiti sve testove svih obuhvaćenih klasa.

Daćemo jedan primer. Prikazana klasa sadrži oznake ***@RunWith(Suite.class)*** i ***@SuiteClasses(TestClass1.class, ...)*** te na taj način označava da će se Junit 4 testiranje sprovesti sa ***org.junit.runners.Suite*** za određenu klasu koja se testira. Takođe se navode klase koje će na ovaj način biti testirane i sa kojim redosledom.

```
import org.junit.runner.RunWith;
 import org.junit.runners.Suite;

 @RunWith(Suite.class)
 @Suite.SuiteClasses({
   TestFeatureLogin.class,
   TestFeatureLogout.class,
   TestFeatureNavigate.class,
   TestFeatureUpdate.class
 })

 public class FeatureTestSuite {
   // the class remains empty,
   // used only as a holder for the above annotations
 }
```

REDOSLED IZVRŠENJA TESTOVA

Najšešće je redosled metoda koji se testiraju u test klasi slučajan, ali se može i urediti primenom @FixMethodOrder(MethodSorters.NAME_ASCENDING)

Od svog nastavka, JUnit ne specificira redosled izvršenja metoda koji se pozivaju radi tetsiranja. Metodi se pozivaju u redosledu koji daje API. Kako JDK 7 vraća metode po slučajnim redosledu, nije najbolje da se tim redosledom i vrši njihovo testiranje. Generalno gledano, poželjno je da se testiranje metoda vrši po redosledu u skladu sa predvidljivmim greškama koje se mogu javiti.

Od verzije 4.11 JUnit upotrebljava deterministički, ali nepredvidljiv redosled ***(MethodSorted.DEFAULT).*** Ako želite da definišete redosled tetsiranja vaše test klase, treba da upotrebite ***@FixMethodOrder*** i specificirajte raspoložive sortere sa ***MethodSorters***:

@FixMethodOrder(MethodSorters.JVM):

Ovaj primer ostavlja redosled metoda onakav kakav je proizvela JVM. Ovaj redosled se menje od izvršenja do izvršenja, jer je nepredvidiljiv.

@FixMethodOrder(MethodSorters.NAME_ASCENDING)

Ovaj iskaz sortira test metode prema imenu metoda, po alfabetskom redosledu.

PRIMER 5 - TESTIRANJE IZUZETAKA

Kod dužih testova, preporučljivo je da se upotrebi ExpectedException pravilo, koje proizvodi, pored izuzetka, i određenu poruku

Kako proveravate da je vaš program izbacio očekivani izuzetak? Na primer,

```
new ArrayList<Object>().get(0);
```

Ovaj kod bi trebalo da izbaci izuzetak ***IndexOutOfBoundsException.*** Oznaka ***@Test*** ima opcioni prametar "***expected***" koji kao vrednost ima potklase od ***Trowable***. Ako želimo da proverimo da li ***ArrayList*** izbacuje da tačan izuzetak, trebalo bi da napišemo

:

```
@Test(expected= IndexOutOfBoundsException.class)
 public void empty() {
     new ArrayList<Object>().get(0);
 }
```

Prametar ***expected*** treba koristiti sa oprezom. Gornji test će proći ako bilo koj kod u metodu izbacuje ***IndexOutOfBoundsException.*** Kod dužih testova,preoručljivo je da se upotrebi ***ExpectedException*** pravilo, koje proizvodi, pored izuzetka, i određenu poruku, na sledeći način:.

```
@Rule
    public ExpectedException thrown = ExpectedException.none();
    @Test
    public void shouldTestExceptionMessage() throws IndexOutOfBoundsException {
        List<Object> list = new ArrayList<Object>();
        thrown.expect(IndexOutOfBoundsException.class);
        thrown.expectMessage("Index: 0, Size: 0");
        list.get(0); // execution will never get past this line
    }
```

Testiranje metoda sa try-cach šemom se vrši na sledeći način:

```
@Test
   public void testExceptionMessage() {
    try {
      new ArrayList<Object>().get(0);
        fail("Expected an IndexOutOfBoundsException to be thrown");
    } catch (IndexOutOfBoundsException anIndexOutOfBoundsException) {
        assertThat(anIndexOutOfBoundsException.getMessage(), is("Index: 0, Size:
0"));
      }
   }
```

PRIMER 6 - PRIMENA METODA ASSERTTHAT

Metod assertThat() pojednostavljuje sintaksu za testiranje tvrdnji.

Sintaksa metoda ***assertThat()*** je sledeća:

```
assertThat([value], [matcher statement]);
```

Evo par primera primene ove sintakse:

```
assertThat(x, is(3));
 assertThat(x, is(not(4)));
 assertThat(responseString,
either(containsString("color")).or(containsString("colour")));
 assertThat(myList, hasItem("3"));
```

Prednosti primene metoda assetrThat() su sledeće:

1. Čitljivija je sintaksa
2. Omogućava negaciju date kombinacije **s**: (not(s)), ili kombinaciju either(s).or(t)) se preslikavaju u kolekciju (each(s)) ili (afterFiveSeconds(s))
3. Čitljive poruke o grešci. Uporedite:

```
assertTrue(responseString.contains("color") || responseString.contains("colour"));
 // ==> failure message:
 // java.lang.AssertionError:

 assertThat(responseString, anyOf(containsString("color"),
containsString("colour")));
 // ==> failure message:
 // java.lang.AssertionError:
 // Expected: (a string containing "color" or a string containing "colour")
 //   got: "Please choose a font"
```

PRIMER 7 - PARAMETRIZOVANI TESTOVI

Primerci sa ukrštenim elementima test metoda i test podataka.

Kada se izvršavaju parametrizovane test klase, kreiraju se primerci sa ukrštenim elementima test metoda i test podataka.

Na primer, testiranje Fibonacci funkcije:

```
@RunWith(Parameterized.class)
 public class FibonacciTest {
   @Parameters
   public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
      { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 },
      { 4, 3 }, { 5, 5 },{ 6, 8 }
    });
   }
   private int fInput;
   private int fExpected;
    public FibonacciTest(int input, int expected) {
      fInput= input;
      fExpected= expected;
    }
   @Test
   public void test() {
    assertEquals(fExpected,
       Fibonacci.compute(fInput));
   }
 }
```

Ovde je korišćen konstruktor sa dva argumenta i sa vrednostima podataka u metodu @Parameters.

.

Takođe je mogućno da se podaci direktno ubace u public polja bez korišćenja konstruktora upotrebom oznake ***@Parameter*** , kao na primer:

```
@RunWith(Parameterized.class)
 public class FibonacciTest {
   @Parameters
   public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
      { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 },
      { 4, 3 }, { 5, 5 },{ 6, 8 }
    });
   }
   @Parameter // prva vrednost podataka (0) je početna vrednost.
   public /* NOT private */ int fInput;
    @Parameter(value = 1)
    public /* NOT private */ int fExpected;
   @Test
   public void test() {
    assertEquals(fExpected,
      Fibonacci.compute(fInput));
   }
 }
```

PRIMER 8 - PRIMENA PRAVILA (@RULE)

Pravila dozvoljavaju fleksibilnu promenu definicije ponašanja svakog test metoda i test klasi

Pravila dozvoljavaju fleksibilnu promenu definicije ponašanja svakog test metoda I test klasi. Testeri mogu da koriste ranije definisana pravila ili da ih proširuju, ili da definišu nova pravila. Evo jednog primera u kome se koristi ***TemporaryFolder*** i ***ExpectedExeption***.

```
public class DigitalAssetManagerTest {
   @Rule
   public TemporaryFolder tempFolder =
     new TemporaryFolder();
   @Rule
   public ExpectedException exception =
     ExpectedException.none();
   @Test
   public void countsAssets() throws IOException {
    File icon = tempFolder.newFile("icon.png");
    File assets = tempFolder.newFolder("assets");
    createAssets(assets, 3);
    DigitalAssetManager dam =
       new DigitalAssetManager(icon, assets);
    assertEquals(3, dam.getAssetCount());
   }
   private void createAssets(File assets,
     int numberOfAssets) throws IOException {
```

```
    for (int index = 0; index < numberOfAssets; index++) {
      File asset = new File(assets,
           String.format("asset-%d.mpg", index));
      Assert.assertTrue("Asset couldn't be created.",
           asset.createNewFile());
    }
   }
   @Test
   public void throwsIllegalArgumentExceptionIfIconIsNull() {
    exception.expect(IllegalArgumentException.class);
    exception.expectMessage("Icon is null, not a file,
            or doesn't exist.");
    new DigitalAssetManager(null, null);
   }
 }
```

TemporaryFolder pravilo dozvoljava kreiranje foldera i datoteka koje se automastki brišu po završetku testiranja metoda.

:

```
public static class HasTempFolder {
   @Rule
   public TemporaryFolder folder = new TemporaryFolder();
   @Test
   public void testUsingTempFolder() throws IOException {
    File createdFile = folder.newFile("myfile.txt");
    File createdFolder = folder.newFolder("subfolder");
    // ...
   }
 }
```

PRIMER 9

Cilj ovog zadatka je provežbavanje pisanja JUnit testova.

Napraviti metodu reverse koja ima zadatak da okrene String naopačke.

Napisati 3 testa ove metode. Prvi test treba da proveri da li se prosleđena vrednost metodi razlikuje od rezultata i još dve metode koje upoređuju da li se reč okrenula sa realnom vrednošću, primer:

programiranje, ejnarimargorp

Objašnjenje:

Cilj ovog zadatka je ispitivanje ispravnosti napisane metode koja String pretvara u String u obrnutom redosledu. Za testiranje ispravnosti u sve tri metode koristimo osobine testova da definišemo vrednost koju očekujemo kao rezultat. Ukoliko poziv metode vrati vrednost koju smo i očekivali, test je uspešno završen. Metoda za proveru je:

assertEquals(expResult, result);

Važno je testirati i suprotan primer kada proveravamo ispravnost metode:

assertNotSame(expResult, result);

U ovom slučaju test je ispravan ukoliko vrednost koju očekujemo zaista nije ista kao vrednost koju metoda vraća, što potvrđuje ispravnost testa.

Klasa MainReverse:

```
public class MainReverse {

    public StringBuilder strBuild = new StringBuilder("");

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
    }

    public MainReverse() {
    }

    public String reverse(String str) {
        strBuild.setLength(0);
        for (int i = str.length() - 1; i >= 0; i--) {
            strBuild.append(str.charAt(i));
        }
        return strBuild.toString();
    }
}
```

Test klasa: MainReverseTest

```
import static junit.framework.Assert.assertEquals;
import static junit.framework.Assert.assertNotSame;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MainReverseTest {

    public MainReverseTest() {
    }
```

```
    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of reverse method, of class MainReverse.
     */
    @Test
    public void testReverse() {
        System.out.println("reverse");
        String str = "rados";
        MainReverse instance = new MainReverse();
        String expResult = "sodar";
        String result = instance.reverse(str);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testReverse1() {
        System.out.println("reverse");
        String str = "programiranje";
        MainReverse instance = new MainReverse();
        String expResult = "ejnarimargorp";
        String result = instance.reverse(str);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testReverse2() {
        System.out.println("reverse");
        String str = "naopako";
        MainReverse instance = new MainReverse();
        String expResult = "naopako";
        String result = instance.reverse(str);
        assertNotSame(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
```

```
    }

}
```

PRIMER 10

Cilj zadatka je provežbavanje pisanja JUnit testova.

Napraviti metodu sort koja String pretvara u niz karaktera, a nakon toga niz karaktera sortira i vraća kao String.

Napraviti sledeće testove.

1. Da li prosleđeni String: bbccaa vraća aabbcc

2. Da li prosleđeni String: 54321 vraća 12345

3. Da li prosleđeni String: bbccaarr ne vraća rraabbcc

Objašnjenje:

Koristeći se istim pristupom kao što je navedeno u prethodnom primeru, vršimo proveru ispravnosti napisane metode.

Klasa MainSort:

```
import java.util.Arrays;

public class MainSort {

    public StringBuilder strBuild = new StringBuilder("");

    public MainSort() {
    }

    public String sortChars(String str) {
        strBuild.setLength(0);
        char[] charVals = str.toCharArray();
        Arrays.sort(charVals);
        return strBuild.append(charVals).toString();
    }
}
```

Test klasa: MainSortTest:

```
import static junit.framework.Assert.assertEquals;
import static junit.framework.Assert.assertNotSame;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
```

```
public class MainSortTest {

    public MainSortTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
    * Test of sortChars method, of class MainSort.
    */
    @Test
    public void testSortChars() {
        System.out.println("sortChars");
        String str = "bbccaa";
        MainSort instance = new MainSort();
        String expResult = "aabbcc";
        String result = instance.sortChars(str);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testSortChars2() {
        System.out.println("sortChars");
        String str = "54321";
        MainSort instance = new MainSort();
        String expResult = "12345";
        String result = instance.sortChars(str);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testSortChars3() {
        System.out.println("sortChars");
        String str = "bbccaarr";
```

```
        MainSort instance = new MainSort();
        String expResult = "rraabbcc";
        String result = instance.sortChars(str);
        assertNotSame(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

}
```

PRIMER 11

Cilj primera 11 je provežbavanje pisanja JUnit testova.

Napraviti metodu za konkatenciju dva Stringa.

Napraviti sledeće testove:

1. Da li prosleđeni Stringovi foo i bar vraćaju vrednost foobar

2. Da li prosleđeni Stringovi KI103- i Objekti i apstrakcija podataka vraćaju KI103-Objekti i apstrakcija podataka

3. Da li prosleđeni Stringovi najjednostavniji i jednostavniji ne vraćaju jednostavnijinaj

Klasa MainConcatanate:

```
public class MainConcatanate {

    private StringBuilder strBuild = new StringBuilder("");

    public MainConcatanate() {
    }

    public String concatanate(String str1, String str2) {
        strBuild.setLength(0);
        return strBuild.append(str1).append(str2).toString();
    }
}
```

Test klasa MainConcatanate Test

```
import static junit.framework.Assert.assertEquals;
import static junit.framework.Assert.assertNotSame;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MainConcatanateTest {
```

```
public MainConcatanateTest() {
}

@BeforeClass
public static void setUpClass() {
}

@AfterClass
public static void tearDownClass() {
}

@Before
public void setUp() {
}

@After
public void tearDown() {
}

/**
 * Test of concatanate method, of class MainConcatanate.
 */
@Test
public void testConcatanate() {
    System.out.println("concatanate");
    String str1 = "foo";
    String str2 = "bar";
    MainConcatanate instance = new MainConcatanate();
    String expResult = "foobar";
    String result = instance.concatanate(str1, str2);
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    //fail("The test case is a prototype.");
}

@Test
public void testConcatanate2() {
    System.out.println("concatanate");
    String str1 = "KI103-";
    String str2 = "Objekti i apstrakcija podataka";
    MainConcatanate instance = new MainConcatanate();
    String expResult = "KI103-Objekti i apstrakcija podataka";
    String result = instance.concatanate(str1, str2);
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    //fail("The test case is a prototype.");
}

@Test
public void testConcatanate3() {
    System.out.println("concatanate");
    String str1 = "najjednostavniji";
```

```
        String str2 = "jednostavniji";
        MainConcatanate instance = new MainConcatanate();
        String expResult = "jednostavnijinaj";
        String result = instance.concatanate(str1, str2);
        assertNotSame(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

}
```

PRIMER 12

Cilj zadatka 12 je provežbavanje pisanja JUnit testova.

Napraviti metodu koja određuje faktorijel prosleđenog broja

Napraviti sledeće testove:

1. Da li metoda vraća broj 120 za prosleđeni broj 5

2. Da li metoda vraća broj 5040 za prosleđeni broj 7

3. Da li metoda ne vraća broj -120 za prosleđeni broj 5

Objašnjenje:

Kako metoda za određivanje faktorijela broja u slučaju da je broj manji od 0 izbacuje izuzetak AritmeticException, u okviru poslednjeg testa treba očekivati ArithmeticException:

@Test(expected = ArithmeticException.class)

Klasa MainFact:

```
public class MainFact {

    public MainFact() {
    }

    public int fact(int n) {
        int res = 1;
        if (n < 0) {
            throw new ArithmeticException();
        }
        for (int i = 1; i <= n; i++) {
            res *= i;
        }
        return res;
    }
}
```

Test klasa MainFactTest

```
import static junit.framework.Assert.assertEquals;
import static junit.framework.Assert.assertNotSame;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MainFactTest {

    public MainFactTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of fact method, of class MainFact.
     */
    @Test
    public void testFact() {
        System.out.println("fact");
        int n = 5;
        MainFact instance = new MainFact();
        int expResult = 120;
        int result = instance.fact(n);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testFact2() {
        System.out.println("fact");
        int n = 7;
        MainFact instance = new MainFact();
        int expResult = 5040;
        int result = instance.fact(n);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
```

```
        //fail("The test case is a prototype.");
    }

    @Test(expected = ArithmeticException.class)
    public void testFact3() {
        System.out.println("fact");
        int n = -5;
        MainFact instance = new MainFact();
        int expResult = -120;
        int result = instance.fact(n);
        assertNotSame(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }
}
```

PRIMER 13

Cilj 13. zadatka je provežbavanje pisanja JUnit testova

Napraviti metodu koja vraća n-ti član Fibonačijevog niza.

Napraviti sledeće testove:

1. Da li je šesti element fibonačijevog niza broj 8

2. Da li je osmi element fibonačijevog niza broj 21

3. Da li je treći element fibonačijevog niza jednak broju 2

Klasa MainFib:

```
public class MainFib {

    public MainFib() {
    }

    public int fibonachi(int n) {
        n = Math.abs(n);
        if (n <= 1) {
            return n;
        }
        int fibo = 1;
        int fiboPrev = 1;
        for (int i = 2; i < n; ++i) {
            int temp = fibo;
            fibo += fiboPrev;
            fiboPrev = temp;
        }
        return fibo;
    }
}
```

Test klasa MainFibTest:

```
import static junit.framework.Assert.assertEquals;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MainFibTest {

    public MainFibTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of fibonachi method, of class MainFib.
     */
    @Test
    public void testFibonachi() {
        System.out.println("fibonachi");
        int n = 6;
        MainFib instance = new MainFib();
        int expResult = 8;
        int result = instance.fibonachi(n);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testFibonachi2() {
        System.out.println("fibonachi");
        int n = 8;
        MainFib instance = new MainFib();
        int expResult = 21;
        int result = instance.fibonachi(n);
        assertEquals(expResult, result);
```

```
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testFibonachi3() {
        System.out.println("fibonachi");
        int n = -3;
        MainFib instance = new MainFib();
        int expResult = 2;
        int result = instance.fibonachi(n);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

}
```

PRIMER 14

Cilj 14. zadatka je provežbavanje pisanja JUnit testova.

Napraviti metodu koja racuna kvadratnu jednačinu (metoda prima a, b i c)

Napraviti sledeće testove:

1. Ukoliko je a = 1, b=-5 i c=6 da li će x1,2 biti jednako 3,2

2. Ukoliko je a =3, b=5 i c = 5 da li će x1,2 biti jednako 3,2

3. Ukoliko je a = 1, b= 0 a c=-36 da li će x1,2 biti jednako 6,-6

Objašnjenje:
U ovom zadatku za proveru vrednosti niza koristimo metodu assertArrayEquals. Pristup koji imamo kod testova je isti kao i za prethodne primere.

Klasa MainSqrEq:

```
public class MainSqrEq {

    public MainSqrEq() {
    }

    public double[] sqrEquation(int a, int b, int c) {
        double[] res = new double[2];
        int d = b * b - 4 * a * c;
        if (d < 0) {
            throw new ArithmeticException();
        }
        res[0] = (-b + Math.sqrt(d)) / 2 * a;
        res[1] = (-b - Math.sqrt(d)) / 2 * a;
```

```
        return res;
    }
}
```

Test klasa MainSqrEqTest

```
import org.junit.After;
import org.junit.AfterClass;
import static org.junit.Assert.assertArrayEquals;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class MainSqrEqTest {

    public MainSqrEqTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
    * Test of sqrEquation method, of class MainSqrEq.
    */
    @Test
    public void testSqrEquation() {
        System.out.println("sqrEquation");
        int a = 1;
        int b = -5;
        int c = 6;
        MainSqrEq instance = new MainSqrEq();
        double[] expResult =  new double[]{3,2};
        double[] result = instance.sqrEquation(a, b, c);
        assertArrayEquals(expResult, result, 0);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test(expected = ArithmeticException.class)
    public void testSqrEquation2() {
```

```
        System.out.println("sqrEquation");
        int a = 3;
        int b = 5;
        int c = 5;
        MainSqrEq instance = new MainSqrEq();
        double[] expResult = new double[]{3,2};
        double[] result = instance.sqrEquation(a, b, c);
        assertArrayEquals(expResult, result, 0);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }

    @Test
    public void testSqrEquation3() {
        System.out.println("sqrEquation");
        int a = 1;
        int b = 0;
        int c = -36;
        MainSqrEq instance = new MainSqrEq();
        double[] expResult = new double[]{6,-6};
        double[] result = instance.sqrEquation(a, b, c);
        assertArrayEquals(expResult, result, 0);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }
}
```

VIDEO: PRIMENA JUNIT TESTIRANJA

JUnit Testing Tutorial (20,31 minuta)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

ZADACI 2.1 - 2.3

Zadaci za samostalan rad - pisanje JUnit testova

Zadatak 1

Napraviti klasu koja predstavlja rezultat ispita. U okviru klase imamo niz od 6 celobrojnih cifara u intervalu od 0 do 20 koje predstavljaju ocene na pojedinačnim zadacima na ispitu. U klasi treba da imamo i dve metode brojIspitnihPoena() i brojPoena(). Metoda brojIspitnihPoena() treba da vrati zbir 5 najboljih zadataka dok metoda brojPoena() treba da vrati isti podatak pomnožen sa brojem 0.4. Za ovu klasu napraviti JUnit testove po pravilima datim na času. Neophodno je da se obuhvate sve varijante opisane na vežbama (nema vrednost, negativna vrednost, granična vrednost...)

Zadatak 2

Napraviti program kalkulator. Kalkulator treba da ima ulaz na konzolnoj liniji dva broja i operaciju koju treba da izvrši (sabiranje, oduzimanje, množenje, deljenje). Napraviti odgovarajuće Junit testove koji će demonstrirati ispravnost rada kalkulatora. **Napomena:** Kalkulator ne sme deliti sa nulom.

Zadatak 3

Napraviti funkciju koja vraća rezultat jednačine x=2n+2 za prosleđeno n

Napisati sledeće testove:

1. Ukoliko je n = 2 da li je rezultat metode 6

2. Ukoliko je n = -1 da li je rezultat metode 0

3. Ukoliko je n = 5 da li rezultat nije jednak 10

Poglavlje 3

Domaći zadaci

ZADACI 1 I 2

1. i 2. zadatak za samostalan rad studenata

Zadatak 1

Napraviti funkciju koja vraća rezultat jednačine x=2n+3-n

Napisati sledeće testove:

1. Ukoliko je n=2 da li je rezultat 5

2. Ukoliko je n=10 da li je rezultat 18

3. Ukoliko je n=4 da li je rezultat nije 8

Zadatak 2

Napraviti funkciju koja vraća rezultat jednačine x= (2x+2)/x

Napisati sledeće testove:

1. Ukoliko je x=0 da li metoda izbacuje izuzetak x ne može biti 0

2. Ukoliko je x=1 da li metoda vraća 4

3. Ukoliko je x=2 da li metoda vraća 3

ZADATAK 3

3. zadatak za samostalan rad studenata

Zadatak 3

Napraviti funkciju koja vraća rezultat jednačine x= (3x-2)/x

Napisati sledeće testove:

1. Ukoliko je x=0 da li metoda izbacuje izuzetak x ne može biti 0

2. Ukoliko je x=1 da li metoda ne vraća 4

3. Ukoliko je x=2 da li metoda vraća 2

ZADATAK 4

4. zadatak za samostalan rad studenata

Zadatak 4

Napraviti metodu koja vraća koliko puta postoji slovo b u prosleđenom Stringu

Napisati sledeće testove

1. Da li za String pera funkcija vraća broj 0

2. Da li za String kompjuter funkcija vraća 0

3. Da li za String smer funckija ne vraća 1

4. Da li za String automobil funkcija vraća 1

Zaključak

REZIME

Pouke

1. Jedinično testiranje je testiranje pojedinačnih objekata, tj. klasa i njenih metoda.
2. Jedinični testovi značajno povećavaju kvalitet koda, zato što je svaka pojedinačna klasa testirana. Ostale prednosti: lakše čitanje koda, moguće naknadno preuređivanje koda, mogućnost pravljenja testova prihvaćanja, lako se uklapa u procedure izrade softvera, i povećava pouzdanost celog sistema
3. JUnit testovi proveravaju svaki deo koda, bez obzira na njegovu verovatnoću pojavljivanja u kodu.
4. Nedostaci: usporava razvoja jer dodaje tri puta više koda, ne testira okruženje, ne obuhvate uslovemreže, SQL instrukcije i neprimenljiv kod interaktivnih GUI klasa.
5. Najviše se testiraju metodi za implementaciju funkcija, a ne seter i geter metodi.
6. Po javljanju greške, odgovarajući deo koda se ispravi, pa se ponavljaju svi testovi.
7. Testovima se proverava: tačnost rezultata za realne vrednost podataka, problematične vrednosti (np. deljenje sa nulom), granične vrednosti podataka i njihovi opsezi, rad sa pogrešnim podacima, kada nedostaje podataka i format dobijenog rezultata.
8. Uvek imamo par: Klasa koje nešto radi, i njena odgovarajuća TestKlasa koja testira ispravnost rada ove klase.
9. JUnit obezbeđuje metode za proveru tvrdnju ili pretpostavka (assertion) za sve primitivne tipove podataka, za objekte i nizove: Najpoznatije metode su: ***assertEquals(), assertTrue(), assertFalse(), assertNull(), assertNotNull(), assertSame(), assertNotSame() i assertThat().***
10. **Suite:** Omogućava da se kreira test program za više klasa. Da bi ga koristili, potrebno je da klasu koju testirate označite sa: ***@RunWith(Suite.class)*** i sa ***@SuiteClasses(TestClass1.class, ...).***
11. **Parametrized:** Ovoj je standardni program za testiranje koji primenjuje tzv, parametrizovane testove.
12. Pravila (***@Rule***) dozvoljavaju fleksibilnu promenu definicije ponašanja svakog test metoda i test klasi.