SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

JUnit - okvir za testiranje

Toni Martinčić

Voditelj: *dr. sc. Mirko Randić*

Zagreb, travanj, 2017.

Sadržaj

1. Uvod

Kod programiranja prilikom pisanja koda vrlo često dolazi do raznih pogrešaka. Može doći do pogrešaka u sintaksi, na primjer programer može zaboraviti staviti ";" na kraj retka. Ako dođe do ovakve pogreške prevodioc će ukazati na nju i program se neće prevesti. Ako se dogodi da programer slučajno neispravno napiše ime funkcije koju poziva, na primjer u programskom jeziku C umjesto „printf“ napiše „print“, doći će do pogreške povezivanja koda i kod se neće ispravno prevesti. Prevodioc će nam javiti grešku. Ove pogreške i pogreške slične njima ćemo lako uočiti i ispraviti ih.

Uz prethodno opisane, u kodu može doći i do logičkih pogrešaka. U slučaju takvih pogrešaka program će se prevesti i prevodioc će javiti da je sve u redu. Uspješno prevođenje nije dovoljan uvjet za ispravnost programa. Može se dogoditi da se napisani program preveo i može se uspješno pokretati, ali da pri tom za ulazne argumente ne daje očekivane rezultate. Zato nakon uspješnog prevođenja, program treba pokrenuti s različitim ulaznim argumentima te treba provjeriti da li program daje očekivane rezultate.

Dakle, kôd treba testirati. To nije uvijek jednostavno. Nakon što program prođe sve testove trebamo biti sigurni da program radi ispravno. Trebamo pokriti sve slučajeve ulaznih argumenata. Često nije moguće isprobati rad programa za sve moguće ulazne argumente jer njih može biti beskonačno mnogo. Iz informacije da program ispravno radi za neke ulazne argumente možemo zaključiti da će raditi ispravno i za neke druge. Trebamo provesti testove za sve rubne slučajeve koji su potencijalno opasni i koji bi mogli srušiti program.

Program nije dobro testirati i ispravljati na način da pokrećemo program za jedne ulazne argumente pa ako sve radi onda pokrećemo program za druge ulazne argumente, a ako nije da ispravljamo pogreške i tako dalje za različite podskupove ulaznih argumenata. To nije dobro iz razloga što, na primjer, program može ispravno raditi za prvi podskup ulaznih argumenata i nakon toga kada idemo probati da li program ispravno radi i za drugi podskup argumente može se dogoditi da ne radi. Mi sada možemo u kodu pronaći gdje je pogreška i koji je bio razlog neispravnog rada programa za drugi skup argumenata. Grešku ćemo ispraviti, a program će sada raditi ispravno za drugi skup argumenata i pretpostaviti ćemo da je sada sve dobro, ali se može dogoditi da je promjena prilikom ispravljanja greške sada uzrokovala neispravan rad programa za prvi skup ulaznih argumenata. To nam može lako promaknuti jer se testiranje za prve ulazne argumente obavilo prije promjene u kodu i tada je sve ispravno radilo. Zbog ovog razloga kôd se treba testirati na način da se napišu svi potrebni testovi koji će se kod provjere ispravnosti koda pokretati istovremeno i svi testovi trebaju biti zadovoljeni.

U nastavku će biti objašnjen JUnit, okvir za testiranje programa pisanih u programskom jeziku Java.

2. Junit – okvir za testiranje

Junit je programski okvir za testiranje programa pisanih u programskom jeziku Java. Inačica koja je aktualna u trenutku pisanja ovog rada je 4.12.

Programski jezik Java je objektno orijentirani jezik pa se tako pisanje programa temelji na definiranju klasa. Kada testiramo kod, svaku klasu testiramo zasebno. JUnit predviđa definiranje testova testnoj klasi koja je odvojena od testirajuće klase. Test case je testna klasa u kojoj se nalaze svi testovi. Testna klasa obično ima naziv kao i klasa koju testira samo sa dodanim sufiksom „`Test`".

2.1 Pisanje testnih klasa

Kod pisanja testne klase svaki test se piše u posebnoj metodi. Kod alata JUnit test je svaka metoda koja je anotirana sa `@Test`. Metoda koja je tako anotirana treba biti javna metoda i ne smije ništa vraćati, dakle treba imati signaturu `public void`. Metoda anotirana sa `@Test` također ne smije ni primati nikakve ulazne argumente. Iz naziva testne metode moralo bi biti jasno što taj test provjerava. Običaj je da se nazivi testnih metoda kao i svih drugih metoda pišu malim početnim slovom, a sve riječi su spojene na način da svaka riječ osim prve počinje pisati velikim početnim slovom. Takav način pisanja imena se naziva camel case. Naziv testne metode ne mora biti kratak, dapače on se može sastojati od više riječi ako je tako potrebno da bi se iz naziva moglo vidjeti što ta metoda testira.

Kada pokrenemo testnu klasu svaka metoda anotirana s `@Test` se izvodi zasebno. Svaka takva metoda dojavljuje jednu od ukupno tri stvari. Prva stvar koju metoda može dojaviti je ta da je test uspješno prošao. To znači da se dio funkcionalnosti koda koji ta metoda testira uspješno izvršio. Druga stvar koju metoda može dojaviti je ta da test nije prošao. To znači da taj dio koda koji testiramo ne radi na način na koji mi to očekujemo. Taj dio koda se ispravno izvršio bez pogrešaka, ali nije dao ispravne rezultate. Treća stvar koju metoda može dojaviti je ta da se izvođenje nije uopće uspješno izvršilo već je došlo do greške. Nakon pokretanja testova testne klase dobivamo ispis u kojemu možemo vidjeti koje su sve testne metode uspješno izvršene, u kojima test nije prošao i možemo vidjeti one metode u kojima je došlo do neke pogreške. U ispisu za metode u kojima je test prošao nemamo dodatnih informacija osim informacije da je test prošao jer nam one nisu potrebne. U ispisu za metode u kojima nije bilo pogrešaka, ali test nije prošao možemo vidjeti zašto nam test nije prošao. Možemo vidjeti u kojoj liniji testne metode je test pao. Imamo informaciju o vraćenom rezultatu i možemo vidjeti očekivani rezultat testa. U ispisu za metode u kojima je došlo do pogreške možemo vidjeti koja se točno pogreška dogodila i možemo vidjeti u kojoj se klasi i u kojoj liniji koda dogodila.

Osim testnih metoda koje su anotirane s `@Test` u testnim klasama mogu se nalaziti i neke druge metode. To mogu biti bilo koje pomoćne metode koje su nam potrebne prilikom testiranja. Na primjer, ako testiramo program koji radi s podacima koje čita iz tekstualne datoteke, u testnoj klasi možemo imati pomoćnu metodu koja čita potrebne podatke iz tekstualne datoteke i onda te podatke sprema u neku kolekciju.

Osim metoda s anotacijama `@Test` moguće su i metode s drugim anotacijama. To mogu biti metode s anotacijama `@Before` i metode s anotacijama `@After`. Metode anotirane s `@Before` se pozivaju točno jednom **prije** izvođenja svake testne metode koje su anotirane s `@Test`. Te metode služe za inicijalizaciju potrebnih podataka, to jest za stvaranje potrebnih početnih preduvjeta za izvođenje testnih metoda. Metode anotirane s `@After` se pozivaju točno jednom **nakon** izvođenja svake od testnih metoda. Ako imamo više metoda označenih bilješkama `@Before` ili `@After`, ne znamo kojim će redoslijedom one biti pozivane. Te metode označene anotacijama `@Before` i `@After` nisu obavezne. One su se u inačici JUnit 3 zvale `setUp()` i `tearDown()` pa se često tako nazivaju i danas. Možemo imati i metode označene anotacijama `@BeforeClass` i `@AfterClass`. Metoda označena anotacijom `@BeforeClass` se poziva točno jednom prije izvođenja testne klase. Metoda označena anotacijom `@AfterClass` se poziva točno jednom nakon izvođenja testne klase.

U testnim klasama možemo imati i bilo koje pomoćne varijable koje su nam potrebne i možemo ih koristiti u svim metodama.

2.2 Pisanje testova

U prethodnom poglavlju je već navedeno da svaka testna metoda mora biti anotirana s `@Test`. Navedeno je i da nazive testnih metoda treba pažljivo odabrati tako da se iz naziva može shvatiti što pojedina metoda testira. Svaka testna metoda trebala bi testirati samo jedan dio funkcionalnosti koda. Ne bi bilo dobro da jednoj testnoj metodi dajemo uloge testiranja više različitih funkcionalnosti jer ako taj test ne bi prošao morali bismo tražiti zbog čega nije prošao i koji dio programa ne radi. Ovako ako svakoj metodi zadamo samo jednu odgovornost onda odmah nakon testiranja znamo na temelju toga je li test prošao ili pao, koji dio koda je ispravan odnosno nije ispravan. Kod pisanja testnih metoda treba se pridržavati svih pravila kojih se treba pridržavati i kod pisanja ostalih metoda, ali i kod pisanja koda općenito. Kod mora biti što kraći te treba biti što je moguće više čitljiv. Kada netko čita kod naše testne metode, treba mu iz naziva metode biti jasno što ta testna metoda provjerava, a iz koda testne metode mora biti jasno na koji način to provjerava.

Poznato je da klasa sadržava metode. Kod testiranja neke klase za svaku metodu te klase treba napisati posebne testove. Svaku metodu ćemo testirati za različite slučajeve i za svaki slučaj ćemo provjeriti je li rezultat izvođenja očekivani.

JUnit sadržava razne metode za definiranje tvrdnji. Nakon izvođenja metode koju testiramo možemo rezultat te metode spremiti u neku varijablu. Nakon toga za tu

varijablu možemo tvrditi da je njezina vrijednost jednaka očekivanom rezultatu. Ako je tvrdnja istinita onda je test prošao, a ako nije onda je test pao. Jedna testna metoda može sadržavati više tvrdnji. Ta testna metoda je uspješno prošla samo u slučaju ako su sve tvrdnje istinite. Ako barem jedna tvrdnja nije istinita onda testna metoda nije prošla.

Alat JUnit sadrži mnogo metoda za definiranje tvrdnji. Primjer takve metode je `assertNull(Object x)`. Ta metoda tvrdi da je predani objekt jednak `null` (ne postoji). Ako predani objekt je jednak `null` onda je tvrdnja prošla, a ako nije jednak null onda tvrdnja nije prošla i taj test u kojemu je ova tvrdnja pozvana pada. Postoji i metoda `assertNotNull(Object x)` koja radi na suprotan način od metode `assertNull(Object x)`.

Postoje i metode koje za ulazni argument primaju boolean. To su metode `assertTrue(boolean x)` i `assertFalse(boolean x)`. Metoda `assertTrue(boolean x)` tvrdi da je predani boolean jednak true, a metoda `assertFalse(boolean x)` tvrdi da je predani boolean jednak false. Te metode ne moraju primiti neku boolean varijablu izravno već im se kao argument može predati neki složeniji logički izraz čiji će rezultat izvođenja biti boolean.

Postoji i puno metoda koje za dva primljena argumenta tvrde da su jednaki. Jedna od njih je `assertEquals(Object x, Object y)` koja za dva primljena objekta tvrdi da su jednaki i uspoređuje ih metodom `equals()`. Kod metoda koje tvrde da su dva argumenta jednaka trebamo paziti kada radimo s onom metodom koja prima dva argumenta koja su tipa double. Ne smijemo pozivati metodu `assertEquals(double expected, double actual)` jer nam ona nekada za vrijednosti za koje mi mislimo da su jednake može vratiti da nisu jednake. To se događa zbog toga što se realni brojevi tipa double zapisuju s određenom nepreciznošću u računalu. Zbog tog razloga umjesto metode `assertEquals(double expected, double actual)` trebamo koristiti metodu `assertEquals(double expected, double actual, double delta)` gdje je argument delta dopušteno odstupanje. Metode oblika `assertEquals` postoje s raznim tipovima ulaznih podataka. Kao što je ovdje navedeno da postoje metode koje primaju Object i double argumente, imamo još i metode koje primaju argumente ostalih primitivnih tipova float, long i int. Za sve navedene tipove ulaznih argumenata postoje i metode oblika `assertNotEquals`.

Također, postoji metoda koja za dva primljena polja tvrdi da su jednaka. To je metoda `assertArrayEquals([] x, [] y)`. Kao i kod ranije opisane metode, `assertEquals`, tipovi argumenata ove metode mogu biti različiti. Argumenti mogu biti tipa: boolean, byte, char, double, float, int, long, Object i short.

Uz metodu `assertEquals` postoji i metoda slična njoj koja se zove `assertSame`. Metoda `assertSame` prima reference na dva objekta tipa Object. Ta metoda se razlikuje od metode `assertEquals` po tome što primljene argumente uspoređuje na način da li se radio o referencama na isti objekt u memoriji dok metoda `assertEquals` primljene argumente uspoređuje na način da nad njima poziva metodu equals. Metoda `assertSame`, dakle, tvrdi da primljeni argumenti zapravo predstavljaju reference na jedan te isti primjerak klase u memoriji.

Sve navedene metode imaju i preopterećenu inačicu koja prima dodatni parametar tipa String koji predstavlja poruku koja će se ispisati ako tvrdnja ne bude istinita prilikom izvođenja testa.

JUnit sadrži još jednu korisnu metodu. To je metoda `assertThat`. Ta metoda kao i prošle „`assertXXX`" metode ima svoju preopterećenu inačicu koja prima dodatni parametar tipa String koji predstavlja poruku koja će se ispisati ako tvrdnja ne bude istinita prilikom izvođenja testa.
Metoda `assertThat` je oblika `assertThat(T actual, Matcher<? Super T> matcher)`. T je tip argumenta za kojega provjeravamo zadovoljava li određeni uvjet. Ulazni argument za kojega provjeravamo zadovoljava li određeni uvjet je argument actual. Ako actual zadovoljava uvjet test je prošao, a ako ne zadovoljava test je pao. Naravno, ako u testu imamo više metoda koje tvrde nešto, sve moraju biti zadovoljene da bi taj test prošao. Uvjet predajemo kao drugi argument metode koji je primjerak klase `Matcher`. Klasa `Matcher` sadrži razne metode koje za primljeni argument provjeravaju zadovoljava li određeni uvjet ili ne. Takvih metoda ima puno, a ovdje navodimo samo neke.

Jedna od metoda je metoda `equalTo(T operand)`. Ona za prvi argument metode `assertThat, T actual`, provjerava je li jednak kao `T operand` tako što nad njima poziva metodu `equals()` razreda `Object`. Primjer poziva te metode bi bio `assertThat(„NekiString", equalTo(„NekiString")).`
Metoda `instanceOf(java.lang.Class<?> type)` za argument T actual metode `assertThat` provjerava je li primjerak razreda tipa `type`.
Metoda `is(Matcher<T> matcher)` omotava neki drugi matcher i ne mijenja njegovu funkcionalnost, ali je test malo prirodnije napisan za čovjekovo shvaćanje. Na primjer umjesto da napišemo `assertThat(student1,equalTo(student2))` možemo pisati `assertThat(student1, is(equalTo(student2))).`
Umjesto `assertThat(student1, is(equalTo(student2)))` možemo i napisati kraće samo `assertThat(student1, is(student2)).`
Metoda `isA(java.lang.Class<?> type)` je skraćeni oblik poziva metode `instanceOf(java.lang.Class<?> type).` Dakle, umjesto `assertThat(student3, is(instanceOf(Student.class)))` možemo pozvati metodu `assertThat(student3, isA(Student.class)).`
Metoda `not(Matcher<T> matcher)` prima neki drugi matcher i samo invertira rezultat koji taj primljeni matcher vraća. Ako primljeni matcher javi da je uvjet zadovoljen onda metoda `assertThat` neće biti zadovoljena i obrnuto. Primjer uporabe ove metode bi bio:
`assertThat(student1, is(not(equalTo(student2)))).` U ovome primjeru vidimo jednu od velikih prednosti korištenja metode `assertThat`. Poziv metode je doslovno napisan poretkom riječi kojim bi čovjek tu tvrdnju napisao prirodnim jezikom tj. izvan koda. „Assert that student1 is not equal to student2", dakle pretpostavi da student1 nije jednak studentu 2. Korištenjem metode `assertThat` postigli smo odličnu čitljivost koda.

Ima još mnogo metoda razreda `Matcher` , a ovdje su opisane samo neke od njih. Također, opisan je osnovni način na koji se one koriste u testiranju koda.

Kod pisanja programa redovito je slučaj da se za ulazne argumente ne može predati bilo koja vrijednost. Dakle, postoje nedozvoljene vrijednosti koje se ne bi smjele predati kao ulazni argument. Korisniku programa se u nekim slučajevima ne može zabraniti da ih unese kao ulazne argumente makar je jasno da takvi ulazni argumenti mogu srušiti program. Zato se taj problem treba riješiti na drugačiji način. Program, kada primi nedozvoljeni ulazni argument, mora biti sposoban situaciju obraditi na ispravan način i ne smije se srušiti. Uobičajeni način u programskom jeziku Java za obradu takvih slučajeva su iznimke. Ako se kao argument za neku metodu preda ilegalna vrijednost doći će do pogreške i ta metoda će izbaciti odgovarajuću iznimku. Iznimka će se iz metode odbaciti pozivatelju metode i pozivatelj je može uhvatiti. Pozivatelj je neka druga metoda koja je pozvala ovu metodu u kojoj je došlo do iznimne situacije. Metoda koja uhvati iznimku treba je obraditi na odgovarajući način. Dakle, sada smo dobili situaciju takvu da umjesto da se program sruši, iznimka može biti uhvaćena i korisniku se može ispisati da se uneseni argumenti ne mogu prihvatiti i program će normalno nastaviti s radom. Trivijalni primjer ovakve situacije bio bi kada bismo imali neku metodu koja prima dva broja te prvi broj dijeli drugim, rezultat pohranjuje u pomoćnu varijablu i taj rezultat vraća nazad. Ilegalna kombinacija argumenata bi bila bilo koja čiji je drugi argument jednak 0 jer bi takva metoda pokušala dijeliti s nulom. U tom slučaju bi metoda bi trebala izbaciti odgovarajuću iznimku, metoda koja ju je pozvala tu bi iznimku uhvatila i obavijestila bi korisnika. U složenijim slučajevima ovakvih nedozvoljenih situacija može biti jako puno. Te nedozvoljene situacije se tijekom pisanja koda nikako ne smiju zanemariti i svaki program treba biti napisan tako da na odgovarajući način može odreagirati na nedozvoljenu kombinaciju ulaznih argumenata.

Tijekom pisanja testova treba uzeti u obzir prethodno napisane stvari. Dakle, kada testiramo neku klasu, dobro je za svaku njenu metodu napisati testove koji provjeravaju na koji se način ta metoda ponaša kada primi nedozvoljene argumente. S obzirom na to da za nedozvoljene argumente metode trebaju baciti iznimke, korištenjem samo onoga što je do sada napisano u ovom radu, svi testovi koji provjeravaju ponašanje metoda za nedozvoljene argumente bi se srušili. Došlo bi do grešaka u testiranju.

Zato JUnit nudi način na koji se ispravno takve stvari mogu testirati. To se radi tako da se iznad metode za koju se očekuje da tijekom izvođenja može doći do izbcivanja iznimke unutar anotacije `@Test` napiše: `@Test(expected="Naziv iznimke".class)`. Primjer bi bio za ovu metodu koja dijeli dva broja i koja kao drugi argument ne smije primiti broj 0 da iznad testa koji poziva tu metodu s drugim argumentom 0 napišemo `@Test(expected=IllegalArgumentException.class)`. Metoda koja dijeli dva broja bi u tom slučaju trebala biti napisana na način da na početku svog izvođenja prvo provjerava je li drugi argument jednak 0. Ako je onda treba baciti iznimku `IllegalArgumentException` s odgovarajućom porukom. Kada imamo napisan ovakav test, dakle test koji očekuje da će se dogoditi iznimka, taj test će proći samo u slučaju ako se dogodila iznimka i to samo onda ako je iznimka koja se dogodila jednaka iznimci koja je očekivana.

2.3 Primjer testa klase IntegerNumber

Pristup testiranju klasa pokazat ćemo na primjeru jednostavne klase `IntegerNumber`. Za klasu će biti napisani testovi uz detaljno objašnjenje. Klasa sadrži jednu člansku varijablu int value. te četiri nestatičke metode koje primaju referencu na drugi objekt tipa `IntegerNumber`. To su metode add, sub, mul i div. Dakle, metode će vrijednosti primjerka razreda IntegerNumber nad kojim su pozvane dodati, oduzeti ili će je pomnožiti ili podijeliti s vrijednosti primjerka razreda IntegerNumber predanog kao argument metode.

Kod klase `IntegerNumber` napisan je u nastavku.

```
package hr.fer.seminar.demonstracija;

public class IntegerNumber {

	private int value;

	public IntegerNumber(int value) {
		this.value = value;
	}

	public int getValue() {
		return value;
	}

	public void add(IntegerNumber integerNumber) {
		this.value += integerNumber.getValue();
	}

	public void sub(IntegerNumber integerNumber) {
		this.value -= integerNumber.getValue();
	}

	public void mul(IntegerNumber integerNumber) {
		this.value *= integerNumber.getValue();
	}

	public void div(IntegerNumber integerNumber) {
		if(integerNumber.getValue() == 0) {
			throw new IllegalArgumentException("Can't divide with zero.");
		}
		this.value /= integerNumber.getValue();
	}

}
```

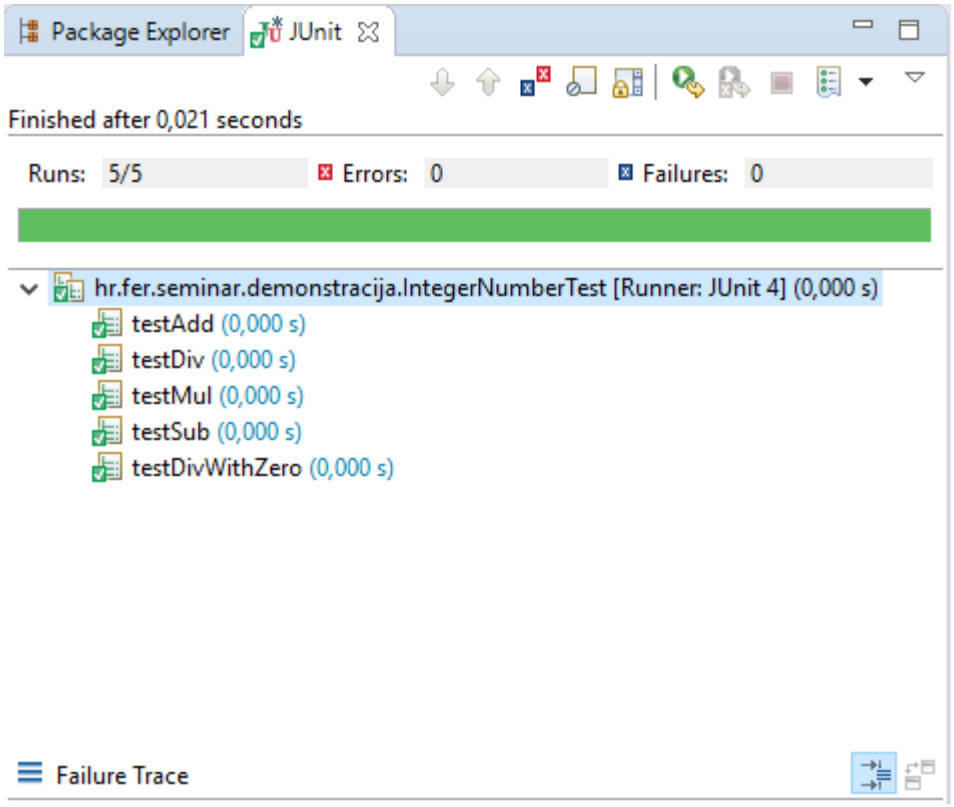
U nastavku je opisana testna klasa s odgovarajućim testovima za klasu `IntegerNumber`. Testna klasa će biti nazvana `IntegerNumberTest` i nalaziti će se u istom paketu kao i klasa `IntegerNumber` samo što će se nalaziti u drugom direktoriju unutar projekta. Klasa `IntegerNumber` se nalazi unutar direktorija src, a klasa `IntegerNumberTest` se nalazi unutar direktorija test.

```java
package hr.fer.seminar.demonstracija;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class IntegerNumberTest {

	private IntegerNumber integerNumber1;
	private IntegerNumber integerNumber2;
	private IntegerNumber integerNumber3;

	@Before
	public void setUp() {
		integerNumber1 = new IntegerNumber(10);
		integerNumber2 = new IntegerNumber(5);
		integerNumber3 = new IntegerNumber(0);
	}

	@Test
	public void testAdd() {
		integerNumber1.add(integerNumber2);
		assertEquals(15, integerNumber1.getValue());
	}

	@Test
	public void testSub() {
		integerNumber1.sub(integerNumber2);
		assertEquals(5, integerNumber1.getValue());
	}

	@Test
	public void testMul() {
		integerNumber1.mul(integerNumber2);
		assertEquals(50, integerNumber1.getValue());
	}

	@Test
	public void testDiv() {
		integerNumber1.div(integerNumber2);
		assertEquals(2, integerNumber1.getValue());
	}

	@Test(expected=IllegalArgumentException.class)
	public void testDivWithZero() {
		integerNumber1.div(integerNumber3);
	}
}
```

Za klasu `IntegerNumber` je jednostavno napisati testove zbog toga što je i sama klasa jednostavna. Ulazni argumenti su odabrani tako da se odmah bez većeg napora može znati očekivano rješenje. Nema smisla odabrati neke velike brojeve za koje ne možemo na brzinu "u glavi" izračunati zbroj, razliku, umnožak i količnik. Za

metodu koja dijeli dva broja je još napisan i test koji provjerava hoće li za dijeljenje s nulom doći do odgovarajuće iznimke.

Nakon pokretanja testova dobije se slijedeći ispis.



Na slici možemo vidjeti sve informacije o izvedenom testu. Vidimo da se svih pet testova uspješno izvelo i svih pet je prošlo. Niti jedan test nije pao i u niti jednom testu nije došlo do pogreške.

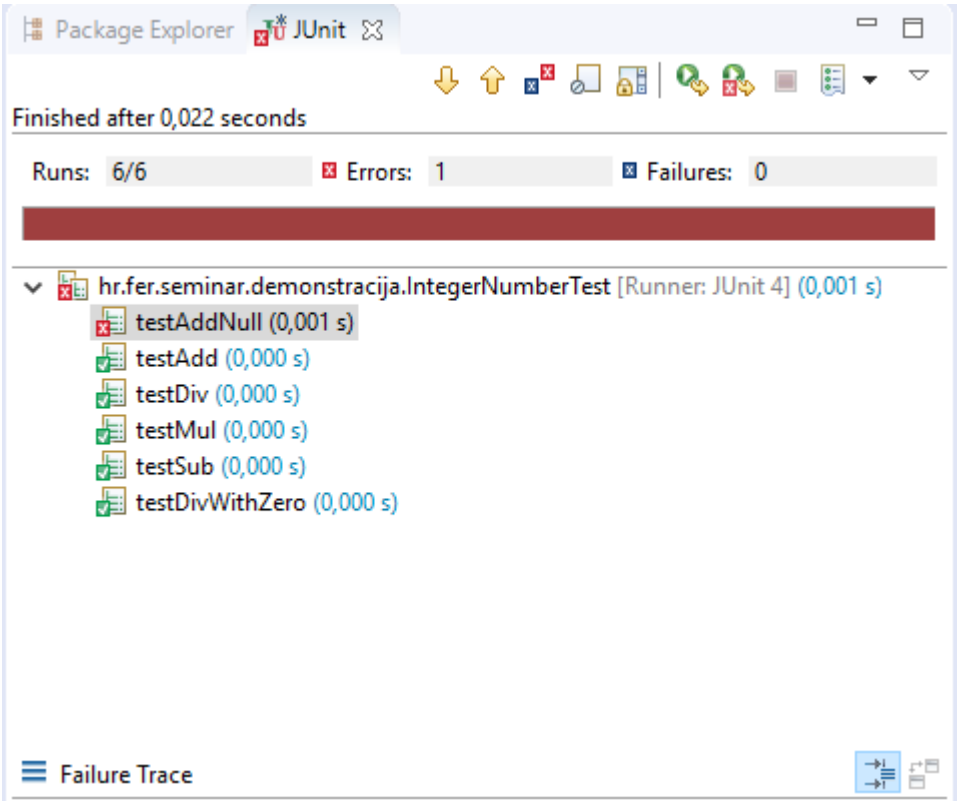
Sada kada smo kod testirali i vidjeli da ispravno radi za sve slučajeve, pa čak i za one nedozvoljene možemo reći da je naš kod ispravan i možemo ga dalje koristiti. Ili ipak možda ne možemo? Na prvi pogled ovdje se čini da je sve u redu i da sve ispravno radi. Ali osobu koja će koristiti naš kod nitko ne sprječava da kao argument bilo koje od metoda klase IntegerNumber preda null vrijednost.

Pogledajmo što će se tada dogoditi. Testnu klasu ćemo proširiti s još jednom testnom metodom koja će metodi add kao argument predati null vrijednost.

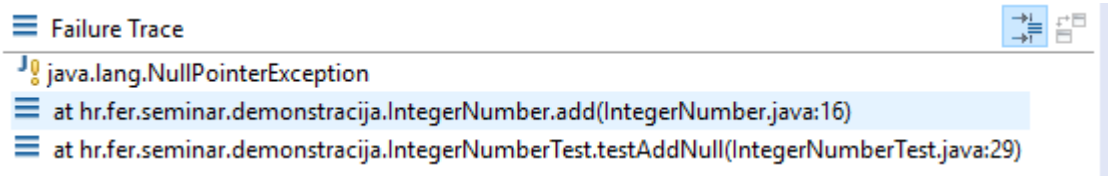
Dodana testna metoda je napisana u nastavku.

```
@Test
public void testAddNull() {
	integerNumber1.add(null);
	assertEquals(10, integerNumber1.getValue());
}
```

Nakon pokretanja testne klase ispis sada izgleda ovako.



Vidimo da su svi testovi, koji su bili i kod prethodnog pokretanja testova, prošli. Novo dodani test nije prošao. Kod njega je došlo do pogreške.



U donjem prozoru možemo vidjeti do koje je pogreške došlo i gdje je u kodu došlo do te pogreške. Pogreška je izbila unutar razreda IntegerNumber unutar metode add u ovoj liniji koda: `this.value += integerNumber.getValue();`. Pogreška koja se dogodila je `NullPointerException`. Ta pogreška se dogodila u navedenoj liniji koda jer varijabla `integerNumber` u ovom slučaju ne pokazuje na objekt tipa `IntegerNumber`, već pokazuje na `null`. Dakle u ovoj liniji koda se nad `null` vrijednosti pokušava pozvati metoda `getValue()` što naravno nije moguće i zato dolazi do iznimke. Mi u testu nismo predvidjeli mogućnost pojave `NullPointerException` iznimke i zato se test srušio. Da smo imali neki program

koji bi pozvao istu ovu metodu na ovakav način, taj program bi se srušio ako ne bi imao dio koda koji hvata ovu iznimku.

Klasa `IntegerNumber` nije u potpunosti dobra jer ne želimo da nam se program sruši ako korisnik kao argument unese null vrijednost. Zato ćemo kod sve četiri metode dodati odmah na početku dio koda koji će provjeravati je li predani argument `null` vrijednost. Ako je, baciti ćemo `IllegalArgumentException`. Na isti način smo postupali kod metode `div` za predani `IntegerNumber` s vrijednosti 0.

Klasa `IntegerNumber` će sada izgledati ovako.

```
package hr.fer.seminar.demonstracija;

public class IntegerNumber {

	private int value;

	public IntegerNumber(int value) {
		this.value = value;
	}

	public int getValue() {
		return value;
	}

	public void add(IntegerNumber integerNumber) {
		if(integerNumber == null) {
			throw new IllegalArgumentException("Null value can't be given
			as argument!");
		}
		this.value += integerNumber.getValue();
	}

	public void sub(IntegerNumber integerNumber) {
		if(integerNumber == null) {
			throw new IllegalArgumentException("Null value can't be given
			as argument!");
		}
		this.value -= integerNumber.getValue();
	}


	public void mul(IntegerNumber integerNumber) {
		if(integerNumber == null) {
			throw new IllegalArgumentException("Null value can't be given
			as argument!");
		}
		this.value *= integerNumber.getValue();
	}
```

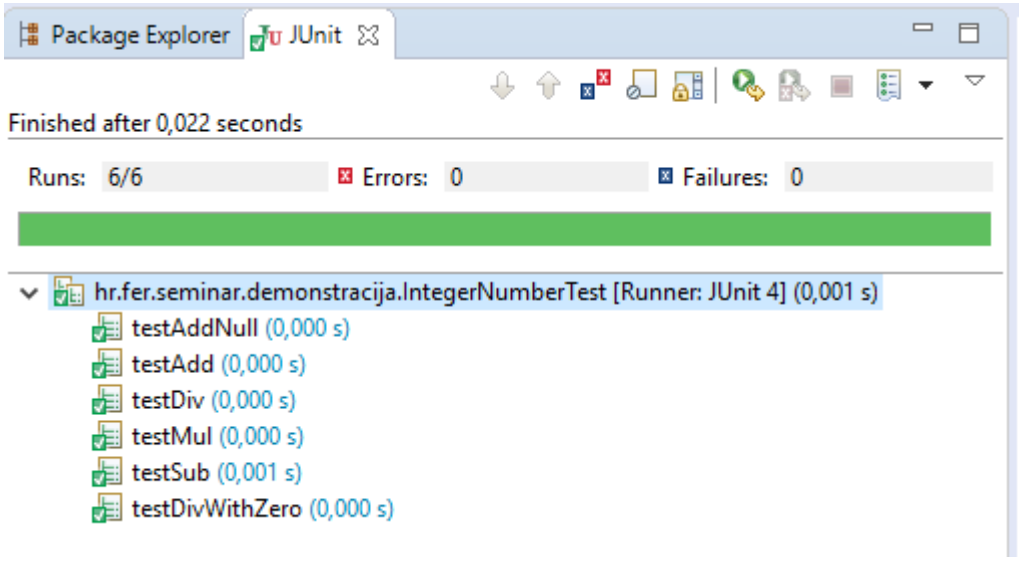
```
	public void div(IntegerNumber integerNumber) {
		if(integerNumber == null) {
			throw new IllegalArgumentException("Null value can't be given
		as argument!");
		}
		if(integerNumber.getValue() == 0) {
			throw new IllegalArgumentException("Can't divide with zero.");
		}
		this.value /= integerNumber.getValue();
	}

}
```

Sada zadnje dodani test u klasi `IntegerNumberTest` možemo promijeniti na način da u anotaciji označimo da očekujemo `IllegalArgumentException`.

Test će sada izgledati ovako.

```
@Test(expected=IllegalArgumentException.class)
	public void testAddNull() {
		integerNumber1.add(null);
}
```

Nakon pokretanja testne klase dobit ćemo slijedeći ispis.



Vidimo da su svi testovi prošli i sada za kod možemo reći da ispravno radi i da je otporan na pogreške. Svi programi koji koriste klasu `IntegerNumber` moraju biti svjesni da za nedozvoljene ulazne argumente dolazi do iznimke `IllegalArgumentException` i moraju na odgovarajući način tu iznimku obraditi. Za klasu `IntegerNumber` treba napisati javadoc i navesti situacije u kojima dolazi do kojih iznimki.

3. Mockito

Nije uvijek slučaj da se program koji pišemo sastoji samo od jedne klase. Programi često unutar neke klase stvaraju primjerke drugih klasa i tijekom svoga izvođenja se oslanjaju na te druge klase. Ispravan rad klase koja u sebi koristi druge klase ovisi o tome funkcioniraju li te druge klase ispravno. Ako u nekoj našoj klasi koristimo klase koje sadrže neke neispravnosti onda ni naša klasa neće raditi na ispravan način.

Zbog ovog razloga prilikom testiranja naše klase možemo naići na neke probleme. Problem je taj što kada testiramo našu klasu i u testovima pozivamo neke metode drugih klasa mi uzimamo pretpostavku da te druge klase rade savršeno. To naravno u praksi vrlo često nije istina. Može se dogoditi da nama neki test nije prošao i mi iz toga zaključujemo da metoda koju taj test testira ne radi ispravno. Ta metoda može raditi potpuno ispravno, ali se oslanja na neke druge metode iz nekih drugih klasa koje ne rade ispravno i zbog toga se naš test srušio.

Kod takvih situacija,a i kod nekih drugih koje se navode u nastavku tijekom testiranja koda u programskom jeziku Java možemo se služiti imitiranjem objekata. To znači da tijekom testiranja nećemo stvarati stvarne objekte koji će biti primjerci nekog razreda već ćemo te objekte imitirati. To se radi tako da za imitirajući objekt odredimo za sve njegove metode koje su nam potrebne što će vratiti ovisno o ulaznim argumentima ako ih imaju. Na primjer, ako imamo neki objekt koji sadrži metodu koja prima dva broja i vraća neki treći broj na temelju nekog računa mi nećemo stvarno pozivati tu metodu već ćemo kod imitirajućeg objekta odrediti što će ta metoda vratiti za rezultat za predane određene argumente. Takav način testiranja koda nam je koristan jer sada testiramo izolirane dijelove koda koji ne ovise o nekim drugim dijelovima koda. Minimizirali smo utjecaj drugih jedinica koda na rezultate testa.

Imitiranje objekata nam može biti korisno još u nekim slučajevima. Imitiranje nam je korisno onda kada su podaci koje koristimo nedeterministički, to jest kada ovise o vremenskom trenutku. Ako mi u našem testu koristimo neki objekt koji nam vraća podatke ovisno o vremenskom trenutku mi u testovima ne znamo koje ćemo točno vrijednosti podataka dobiti. Zato ne možemo napisati testove na način da stvorimo taj objekt i od njega tražimo podatke. Ovdje trebamo koristiti imitiranje objekata. Imitirati ćemo taj objekt i točno ćemo odrediti koje će nam on podatke vratiti. Sada kada znamo koje vrijednosti imamo možemo dalje testirati rad neke naše metode s tim vrijednostima.

Imitiranje objekata možemo koristiti i u slučaju kada je dohvaćanje nekih podataka presporo, na primjer ako podatke moramo dohvaćati s weba.

Za imitaciju objekata u javi postoji više različitih okvira. Svi okviri imaju slične funkcionalnosti, ali različite sintakse. Jedan od poznatijih okvira za imitaciju objekata je Mockito.

U nastavku će bit objašnjen način korištenja Mockita.

3.1 Korištenje Mockita

Kod Mockita prije korištenja imitiranog objekta najprije moramo stvoriti taj imitirani objekt. Svaki imitirani objekt mora biti anotiran s `@Mock`. Ako koristimo imitirane objekte onda testirajuća klasa koja koristi imitirane objekte mora biti anotirana s `@RunWith(MockitoJUnitRunner.class)`.

Nakon što smo stvorili primjerak imitiranog objekta moramo definirati ponašanje tog objekta. To se radi pozivom metode when nad tim objektom. Poziv metode when izgleda ovako :
`when(mock.someMethod(methodParameters)).thenReturn(mockedResult)`.

Imitirani objekt je mock. Ovdje u metodi when smo odredili da kada nad objektom mock bude pozvana metoda someMethod s argumentima methodParameters da vraćena vrijednost bude mockedResult. Metodom when možemo odrediti i još neke aspekte ponašanja osim toga koje će vrijednosti biti vraćene. Možemo definirati da ako pozovemo neku metodu iz objekta mock sa određenim parametrima da ta metoda baci iznimku. Metodom when to se određuje na ovaj način:
`when(mock.someMethod(methodParameters)).thenThrow(new SomeException())`.

Nakon što smo stvorili primjerak imitiranog objekta i nakon što smo definirali funkcionalnost imitiranog objekta možemo napisati testove. U testovima ćemo koristiti imitirani objekt i pozivati ćemo metode s ulaznim argumentima za koje smo definirali što vraćaju.

Unutar testnih metoda u kojima koristimo imitirane objekte možemo raditi neke provjere. Te provjere radimo da se uvjerimo da je imitirani objekt korišten na način na koji smo mi to očekivali. Provjeru radimo pozivom metode `verify`:
`verify(mock, times(expectedNo)).someMethod(methodParameters)`.
Pozivom metode `verify` provjeravamo je li nad objektom mock metoda `someMethod` s parametrima `methodParameters` pozvana `expectedNo` puta. Ako to nije istina onda test nije prošao. U metodi `verify` umjesto metode `times()` kojoj predajemo informaciju koliko puta očekujemo da se određena metoda izvodi, možemo koristiti i jednu od sljedećih metoda verifikacije: `never()`, `atLeastOne()`, `atLeast()`, `atMost()`. Ako koristimo metodu `never()` onda tvrdimo da se metoda `someMethod` s argumentima `methodParameters` nije izvela niti jedan put. Ako koristimo metodu `atLeastOne` onda tvrdimo da se metoda izvela barem jednom. Ako koristimo metodu `atLeast` onda tvrdimo da se metoda izvela barem onoliko puta kolika je vrijednost argumenta koji predajemo metodi `atLeast`. Korištenjem metode `atMost` provjeravamo da li se metoda izvela najviše onoliko puta kolika je vrijednost argumenta koji predajemo metodi `atMost`.

3.2 Primjer korištenja Mockita

Primjer korištenja Mockita prikazan je na testiranju Javine kolekcije `List`.

Testna klasa je napisana u nastavku.

```
package hr.fer.seminar.demonstracija;

import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import java.util.LinkedList;
import java.util.List;

import org.junit.Test;
import org.mockito.internal.verification.Times;

public class ListTest {

	LinkedList mockedList = mock(LinkedList.class);

	@Test
	public void testGet() {
		LinkedList mockedList = mock(LinkedList.class);

		when(mockedList.get(0)).thenReturn("first");

		assertEquals("first", mockedList.get(0));

		verify(mockedList, times(1)).get(0);
	}

	@Test(expected=IndexOutOfBoundsException.class)
	public void testGetIlegal() {
		List mockedList = mock(List.class);

		when(mockedList.get(0)).thenReturn("first");
		when(mockedList.get(1)).thenThrow(IndexOutOfBoundsException.class);

		assertEquals("", mockedList.get(1));
	}

	@Test
	public void testSize() {
		List mockedList = mock(List.class);

		when(mockedList.size()).thenReturn(0);

		assertEquals(0, mockedList.size());

		verify(mockedList, times(1)).size();
	}
}
```

Testove sam napisao na prethodno opisan način. Jedina razlika je ta što ovdje nije korištena anotacija @Mock nego se imitirani objekt stvara pozivom metode mock.

4. Zaključak

Svaki program napisani u bilo kojem programskom jeziku je potrebno testirati. Pri tome se treba pridržavati pravila koja nam govore kako ispravno testirati kod. Ako napisani kod ne testiramo ili ga ne testiramo na ispravan način u kodu mogu postojati pogreške koje će kasnije prouzrokovati neispravan rad našeg programa. Zato nakon testiranja programa i ispravljanja eventualnih pogrešaka trebamo biti sigurni da program sada dobro radi i da se neće srušiti.

Testiranje obavljamo uz pomoć odgovarajućih testove za program. Testove pišemo tako da svaki test provjerava jedan dio funkcionalnosti koda.

Kod testiranja koda možemo se služiti raznim programskim alatima. Na primjer, kod testiranja koda napisanoga u programskom jeziku Java možemo se služiti alatima JUnit I Mockito.

5. Literatura

Marko Čupić – Programiranje u Javi

https://en.wikipedia.org/wiki/JUnit

http://www.vogella.com/tutorials/JUnit/article.html

https://www.tutorialspoint.com/mockito/index.htm

http://junit.org/junit4/

http://junit.sourceforge.net/javadoc/

6. Sažetak

U programskom jeziku Java za testiranje koda na raspolaganju imamo alate JUnit i Mockito. U Javi imamo još druge alate za testiranje koda koji nisu prikazani u ovom seminarskom radu.

Kod testiranja koda alatom JUnit za svaku klasu koju testiramo pišemo posebnu testnu klasu. Testna klasa zove se jednako kao i testirajuća klasa samo s nadodanim sufiksom Test.

Za svaku metodu iz testirajuće klase koju želimo testirati pišemo zasebne testove. Testne metode anotiramo s `@Test`.

Kod korištenja alata JUnit na raspolaganju imamo mnoge korisne metode. To su metode kojima možemo specificirati tvrdnje.

Testove pišemo na način da napišemo sve testne metode i onda ih pokrenemo. Nakon što smo ih pokrenuli dobivamo obavijesti o tome kako su se testovi izvršili, to jest, da li su prošli ili pali. Ako su neki od testova pali onda popravljamo kod i ponovo pokrećemo testove i to sve dok nam svi testovi ne prođu.

Alat Mockito koristimo u slučaju ako ne želimo stvarati stvarne objekte nekih klasa veće objekte želimo imitirati. To nam može biti vrlo korisno u nekim slučajevima. Na primjer onda kada su podaci koje koristimo nedeterministički, to jest kada ovise o vremenskom trenutku. Imitiranje objekata možemo koristiti i u slučaju kada je dohvaćanje nekih podataka presporo, na primjer ako podatke moramo dohvaćati s Web-a.

Kod korištenja Mockita imitirane objekte moramo anotirati s `@Mock` ili ih stvoriti pozivom metode `mock()`. Kada ih stvorimo onda definiramo njihovo ponašanje za određene metode pozivom metode `when()`. Nakon toga koristimo naš imitirani objekt u testnim metodama i onda još na kraju testnih metoda metodom `verify` možemo provjeriti je li imitirani objekt bio korišten na način na koji smo mi to očekivali. Možemo za određene metode provjeriti jesu li bile pozvane s očekivanim argumentima i jesu li bile pozvane očekivani broj puta.