8. homework assignment; JAVA, Academic year 2015/2016; FER

First: read last page. I mean it! You are back? OK. This homework consists of 1 problem.

1. Izrada jednostavnog računalnog sustava

Sve razrede koje napišete u okviru rješenja ovog zadatka potrebno je smjestiti u odgovarajuće podpakete paketa `hr.fer.zemris.java.simplecomp.impl`.

1.1 Priprema

Na Ferku u repozitoriju postoji kategorija "Dodatci domaćim zadaćama"/"Uz domaću zadaću 08" i tamo ćete pronaći datoteke:

```
primjeri.zip
computer-models.jar
computer-models-src.jar
parser.jar
```

Napravite u Eclipse-u novi projekt. Preuzmite te datoteke. Datoteka `primjeri.zip` sadrži primjere asemblerskih programa za računalo koje gradite u ovom zadatku. Raspakirajte tu arhivu u Vašem projektu tako da dobijete `examples` poddirektorij u direktoriju projekta. Preostale tri JAR datoteke smjestite u poddirektorij `lib` Vašeg projekta. Biblioteke `computer-models.jar` i `parser.jar` dodajte ih u *Build Path* projekta. Potom napravite desni klik na `computer-models.jar` koji će se pojaviti u kategoriji Referenced libraries, odaberite properties i potom postavite datoteku `computer-models-src.jar` kao datoteku koja ćuva njezin izvorni kod. Time ćete u Eclipse-u automatski dobiti i dokumentaciju razreda i sučelja definiranih u biblioteci `computer-models.jar`. Nemojte biblioteku `computer-models-src.jar` raspakiravati i .java datoteke kopirati u Vaš projekt! Biblioteka `parser` sadrži implementaciju parsera za asemblerski kôd pisan prema formatu koji je opisan u nastavku ovog dokumenta. Biblioteka `computer-models` sadrži definicije potrebnih sučelja.

1.2 Zadatak

Cilj ovog zadatka je upoznati se još malo bolje sa sučeljima i njihovom uporabom. Stoga ćete tijekom izrade ovog zadatka napraviti jednostavnu implementaciju “mikroprocesora” koji može izvršavati jednostavne programe. Primjerice, neki jednostavan program mogao bi izgledati ovako:

```
#Ovaj program 3 puta ispisuje "Hello world!"

            load r0, @brojac        ; ucitaj 3 u registar r0
            load r1, @nula          ; ucitaj 0 u registar r1
            load r7, @poruka        ; ucitaj poruku u r7
@petlja:    testEquals r0, r1       ; je li r0 pao na nulu?
            jumpIfTrue @gotovo      ; ako je, gotovi smo
            decrement r0            ; umanji r0
            echo r7                 ; ispisi na konzolu poruku
            jump @petlja            ; skoci natrag u petlju
@gotovo:    halt                    ; zaustavi procesor

#podaci koje koristimo u programu

@poruka:    DEFSTR "Hello world!\n" ; poruka na jednoj mem. lokaciji
@brojac:    DEFINT 3                ; broj 3 na drugoj mem. lokaciji
@nula:      DEFINT 0                ; broj 0 na trecoj mem. lokaciji
```

U okviru ovog zadatka računalni sustav i njegove komponente modelirane su kroz niz sučelja. Parser za program napisan asemblerskim jezikom prikazanim u prethodnom primjeru nalazi se u biblioteci koju koristite i ne trebate ga sami pisati. Parser, međutim, ne zna koje sve instrukcije postoje i kako ih treba izvršavati; ono što parser razumije je sintaksa asemblerskog jezika: što su komentari, gdje se nalazi naziv instrukcije, gdje su parametri i slično. Primjerice: pogledajte prvi redak koji sadrži naredbu `load`. Analizom tog retka, parser će zaključiti da je potrebno stvoriti naredbu koja se zove `load` te koja ima dva operanda: prvi je registar r0 a drugi je adresa memorijske lokacije (tj. cijeli broj) na kojoj se nalazi zapisana "varijabla" brojač (potražite ga u kodu i uočite da je njegova početna vrijednost postavljena na 3). Jednom kada pročita redak, prevodioc će pokušati pronaći razred koji implementira sučelje `Instruction` i koji predstavlja implementaciju ove instrukcije. Potom će pozvati konstruktor tog razreda kako bi dobio primjerak koji će predstavljati tu konkretnu instrukciju, i njega će pohraniti u memoriju mikroračunala. Vaš će zadatak, između ostaloga, biti i pisanje implementacija svih potrebnih instrukcija.

O mikroračunalu

Naše se računalo sastoji od uobičajenih dijelova: registara i memorije. Od registara, na raspolaganju su nam registri opće namjene, programsko brojilo te jedna zastavica. Za razliku od konvencionalnih procesora, naši registri opće namjene (kao i sama memorija) umjesto jednostavnih brojeva mogu pamtiti proizvoljne Java objekte. Memorija, dakle, nije niz okteta, već niz objekata. Na svaku lokaciju možemo staviti referencu na proizvoljno veliki objekt (koji se čuva na *heapu* Javinog virtualnog stroja).

Zadatak 1.2.1.

U paketu `hr.fer.zemris.java.simplecomp.models` nalaze se sučelja `Computer`, `Memory` te `Registers`. Proučite ova sučelja (svako sučelje dodatno je dokumentirano) i za svako napišite jedan razred koji ih implementira. Ova sučelja dostupna su u biblioteci `computer-models`. Razrede nazovite kao i samo sučelje i nadodajte `Impl` na kraju (tako će Vaš razred `ComputerImpl` implementirati sučelje `Computer`). Ove implementacije smjestite u paket `hr.fer.zemris.java.simplecomp.impl`. Za čuvanje registara opće namjene te za memoriju koristite obično polje. Implementacija memorije trebala bi imati konstruktor koji prima broj memorijskih lokacija (tj. veličinu memorije):

```
public MemoryImpl(int size) { ... }
```

a implementacija registara trebala bi imati konstruktor koji prima broj registara koji će procesoru biti na raspolaganju:

```
public RegistersImpl(int regsLen) { ... }
```

Jednom kada su definirani razredi potrebni za rad našeg računala, možemo pogledati kako se to računalo može koristiti. U paketu `hr.fer.zemris.java.simplecomp` stvorite razred `Simulator` i u njegovu metodu `main` stavite sljedeći kôd.

```
// Stvori računalo s 256 memorijskih lokacija i 16 registara
Computer comp = new ComputerImpl(256, 16);

// Stvori objekt koji zna stvarati primjerke instrukcija
InstructionCreator creator = new InstructionCreatorImpl(
      "hr.fer.zemris.java.simplecomp.impl.instructions"
);

// Napuni memoriju računala programom iz datoteke; instrukcije stvaraj
// uporabom predanog objekta za stvaranje instrukcija
ProgramParser.parse(
   "examples/asmProgram1.txt",
```

```
   comp,
   creator
);

// Stvori izvršnu jedinicu
ExecutionUnit exec = new ExecutionUnitImpl();

// Izvedi program
exec.go(comp);
```

Prethodni primjer sastoji se od nekoliko cjelina:

1. stvaranje novog računala,
2. stvaranje objekta pomoću kojeg će parser stvarati primjerke instrukcija,
3. prevođenje programa zapisanog u tekstualnoj datoteci i njegovo punjenje u memoriju računala,
4. stvaranje izvršne jedinice (tj. sklopa koji će izvoditi program) te
5. pokretanja izvršne jedinice.

Ako ste riješili zadatak 1.2.1, prva cjelina će raditi. U drugoj liniji poziva se parser iz JAR datoteke čiji je zadatak obraditi datoteku s programom. Kako sam parser ne zna na koji način stvoriti odgovarajuću instrukciju, potrebna mu je pomoć. Tu u igru ulazi razred `InstructionCreatorImpl` čija Vam je implementacija također unaprijed dana i dostupna je u istom paketu u kojem se nalazi i sam parser. Zadaća ovog razreda je stvoriti primjerak razred koji implementira instrukciju koju je parser pronašao u kodu. Parser s ovim razredom može razgovarati jer sam razred implementira sučelje `InstructionCreator` (pogledajte izvorni kod tog sučelja). Svaki puta kada parser otkrije neku instrukciju, poziva metodu:

```
public Instruction getInstruction(
  String name, List<InstructionArgument> arguments
);
```

Prvi argument je naziv pronađene instrukcije a kao drugi argument se predaje lista argumenata instrukcije koje je parser pronašao u datoteci.

Npr. kod obrade retka u kojem piše:

```
mul r0, r1, r2
```

ime će biti "`mul`", a lista će sadržavati tri objekta (za svaki registar po jedan; za detalje pogledati izvorni kod sučelja `InstructionArgument`).

Pogledajte dokumentaciju sučelja `InstructionArgument`. Tamo je objašnjen pojam deskriptora registra te indirektnog adresiranja.

Napišite razred `hr.fer.zemris.java.simplecomp.RegisterUtil` i u njega stavite sljedeće tri metode:

```
public static int getRegisterIndex(int registerDescriptor) {
      ...
}

public static boolean isIndirect(int registerDescriptor) {
      ...
}

public static int getRegisterOffset(int registerDescriptor) {
      ...
}
```

Metoda `getRegisterIndex` iz predanog deskriptora vadi indeks registra. Metoda `isIndirect` provjerava radi li se o indirektnom adresiranju (vraća `true`) ili ne (vraća `false`) a metoda `getRegisterOffset` vadi pomak koji je potrebno koristiti. Kao pomoć za korektnu implementaciju ovih metoda nudim sljedeće unit testove:

```
@Test
public void testUnpacking1() {
	int registerDescriptor = 0x1000102;

	assertEquals(2, RegisterUtil.getRegisterIndex(registerDescriptor));
	assertEquals(1, RegisterUtil.getRegisterOffset(registerDescriptor));
	assertEquals(true, RegisterUtil.isIndirect(registerDescriptor));
}

@Test
public void testUnpacking2() {
	int registerDescriptor = 0x1FFFF02;

	assertEquals(2, RegisterUtil.getRegisterIndex(registerDescriptor));
	assertEquals(-1, RegisterUtil.getRegisterOffset(registerDescriptor));
	assertEquals(true, RegisterUtil.isIndirect(registerDescriptor));
}

@Test
public void testUnpacking3() {
	int registerDescriptor = 0x18000FE;

	assertEquals(254, RegisterUtil.getRegisterIndex(registerDescriptor));
	assertEquals(-32768, RegisterUtil.getRegisterOffset(registerDescriptor));
	assertEquals(true, RegisterUtil.isIndirect(registerDescriptor));
}

@Test
public void testUnpacking4() {
	int registerDescriptor = 0x17FFFFF;

	assertEquals(255, RegisterUtil.getRegisterIndex(registerDescriptor));
	assertEquals(32767, RegisterUtil.getRegisterOffset(registerDescriptor));
	assertEquals(true, RegisterUtil.isIndirect(registerDescriptor));
}
```

Kako bi Vam se olakšao rad, uporabom ponuđenog razreda `InstructionCreatorImpl` dovoljno (*i nužno*) je razred koji implementira svaku instrukciju smjestiti u paket koji se predaje u konstruktoru od korištenog razreda `InstructionCreatorImpl`. Ova implementacija očekuje da će razredi koji predstavljaju instrukcije biti nazvani `Instr`*`Ime`*; primjerice, razred koji predstavlja instrukciju `mul` imat će ime `InstrMul`, razred koji predstavlja instrukciju `add` imat će ime `InstrAdd`, itd. Pri tome razred `InstructionCreatorImpl` očekuje da će svaki razred koji predstavlja implementaciju neke instrukcije imati jedan javni konstruktor koji prima listu argumenata. Koristeći taj konstruktor razred `InstructionCreatorImpl` će stvarati primjerke instrukcije predajući mu kao jedini argument upravo onu listu koju je parser pripremio i predao metodi `getInstruction` kao drugi argument. Primjerice, instrukcija `mul r0, r1, r2` koja uzima sadržaje registara `r1` i `r2`, množi ih i rezultat pohranjuje u `r0` može se implementirati kako je prikazano u nastavku. Ova instrukcija ne podržava indirektno adresiranje te radi isključivo izravno s registrima.

```
package hr.fer.zemris.java.simplecomp.impl.instructions;

import java.util.List;

import hr.fer.zemris.java.simplecomp.RegisterUtil;
import hr.fer.zemris.java.simplecomp.models.Computer;
import hr.fer.zemris.java.simplecomp.models.Instruction;
import hr.fer.zemris.java.simplecomp.models.InstructionArgument;

public class InstrMul implements Instruction {
	private int indexRegistra1;
	private int indexRegistra2;
	private int indexRegistra3;

	public InstrMul(List<InstructionArgument> arguments) {
		if(arguments.size()!=3) {
			throw new IllegalArgumentException("Expected 2 arguments!");
		}
		for(int i = 0; i < 3; i++) {
			if(!arguments.get(i).isRegister() ||
					RegisterUtil.isIndirect((Integer)arguments.get(i).getValue())) {
				throw new IllegalArgumentException(
				  "Type mismatch for argument "+i+"!"
				);
			}
		}
		this.indexRegistra1 =
			RegisterUtil.getRegisterIndex((Integer)arguments.get(0).getValue());
		this.indexRegistra2 =
			RegisterUtil.getRegisterIndex((Integer)arguments.get(1).getValue());
		this.indexRegistra3 =
			RegisterUtil.getRegisterIndex((Integer)arguments.get(2).getValue());
	}

	public boolean execute(Computer computer) {
		Object value1 = computer.getRegisters().getRegisterValue(indexRegistra2);
		Object value2 = computer.getRegisters().getRegisterValue(indexRegistra3);
		computer.getRegisters().setRegisterValue(
				indexRegistra1,
				Integer.valueOf((Integer)value1*(Integer)value2)
		);
		return false;
	}

}
```

Zadatak 1.2.2.

Napišite implementacije instrukcija:

`load rX, memorijskaAdresa`

koja uzima sadržaj memorijske lokacije (dobit će to kao broj u drugom argumentu) i pohranjuje taj sadržaj u registar `rX` (index će dobiti kao broj u prvom argumentu); instrukcija ne dozvoljava indirektno adresiranje;

`echo rX`

koja uzima sadržaj registra `rX` i ispisuje ga na ekran (pozivom metode `System.out.print()`); ova instrukcija podržava indirektno adresiranje (legalan oblik instrukcije je i `echo [rX+offset]`), te

`halt`

koja zaustavlja rad procesora.

Zadatak 1.2.3

Napišite razred koji implementira sučelje `ExecutionUnit`. Za pseudo-kod pogledajte u izvorni kod tog sučelja. Modificirajte razred `Simulator` tako da kao jedini argument naredbenog retka prima stazu do datoteke s asemblerskim kodom programa koji je potrebno prevesti i pokrenuti. Ako taj argument nije prisutan, onda program treba korisnika pitati da utipka putanju do datoteke.

Prilagodite metodu `main` tako da implementira tu logiku (u trenutnom primjeru putanja je bila "hardkodirana").

Stvorite datoteku `examples/prim1.txt` sljedećeg sadržaja:

```
        load r7, @poruka    ; ucitaj poruku u r7
        echo r7             ; ispisi na konzolu poruku
        halt                ; zaustavi procesor

@poruka:  DEFSTR "Hello world!\n"
```

Prilikom prevođenja ovog programa, parser će u memoriju na lokaciju 0 pohraniti instrukciju `load` (odnosno primjerak vašeg razreda `InstrLoad`), na lokaciju 1 instrukciju `echo`, na lokaciju 2 instrukciju `halt` te na lokaciju 3 string `“Hello world!\n”` (za ovo posljednje je zaslužna direktiva `DEFSTR` – *define string*; integeri se u memoriju pohranjuju s `DEFINT`). Direktive `DEFSTR` i `DEFINT` te direktive `RESERVE` odnosno `RESERVE:n` koje ćete kasnije vidjeti ne pišete Vi – to parser sam zna obaviti. Također, kod instrukcija koje primaju memorijsku lokaciju, u argumentima Vašeg konstruktora parser Vam neće dati ime te lokacije (tipa `@poruka`) već ćete dobiti broj koji predstavlja stvarnu memorijsku lokaciju na koju je smješten podatak ili instrukcija (ovisno ispred čega ste napisali labelu).

Sada biste trebali moći pokrenuti program `Simulator`, i na ekranu dobiti ispis `“Hello world!”`. Ako nešto ne radi, sada je pravo vrijeme za otkriti u čemu je problem.

Konačni cilj

Cilj je napraviti procesor koji će moći “izvrtiti” kodove priložene u primjerima u ZIP datoteci; primjere (odnosno čitav direktorij `examples`, ako već niste, iskopirajte u Eclipseov projekt, tako da u direktoriju projekta uz `src` i `target` imate i `examples`). Prvi primjer nekoliko puta ispisuje istu poruku a drugi generira

dobro poznati slijed brojeva. Da bi to ostvarili, još Vam trebaju neke instrukcije koje morate ostvariti.

Zadatak 1.2.4

Napomena: instrukcije prikazane u tablici u nastavku ne podržavaju indirektno adresiranje, osim ako to nije eksplicitno prikazano u primjeru instrukcije.

Instrukcija	***Opis***
`add rx, ry, rz`	`rx` ← `ry` + `rz`
`decrement rx`	`rx` ← `rx` - 1
`increment rx`	`rx` ← `rx` + 1
`jump lokacija`	`pc` ← lokacija
`jumpIfTrue lokacija`	ako je `flag=1` tada `pc` ← lokacija

`move rx, ry`
`move rx, broj`
`move [rx+o1],[ry+o2]`
`...`
	`rx` ← `ry`
	`rx` ← `broj` ; npr: `move r7, 24`
	Prvi argument može biti registar ili indirektna adresa; drugi argument može biti registar, indirektna adresa ili broj.

`testEquals rx, ry`	postavlja zastavicu `flag` na `true` ako su sadržaji registara `rx` i `ry` isti, odnosno na `false` ako nisu.

Ostvarite ove instrukcije i provjerite rad prva tri priložena programa. Ispisi bi redom morali biti:

asmProgram1.txt	`Hello world!` `Hello world!` `Hello world!`

asmProgram2.txt	`Program za ispis fib. brojeva.`
	`0. broj je: 0`
	`1. broj je: 1`
	`2. broj je: 1`
	`3. broj je: 2`
	`4. broj je: 3`
	`5. broj je: 5`
	`6. broj je: 8`
	`7. broj je: 13`
	`8. broj je: 21`
	`9. broj je: 34`

asmProgram3.txt	`4242424242`

Napišite potom sljedeće instrukcije.

Instrukcija	***Opis***

`push rx`	Sadržaj registra rx sprema na memorijsku lokaciju koja je trenutni vrh stoga; potom adresu vrha stoga umanjuje za 1. U procesoru koji radite, adresu vrha stoga čuva registar r15 – u sučelju Registers postoji odgovarajuća konstanta koja to definira. Efekt izvođenja ove instrukcije je:
	
	`[r15]` ← `rx`
	`r15` ← `r15 - 1`

`pop rx`	S vrha stoga skida podatak koji pohranjuje u registar rx. Kazaljku vrha stoga

Instrukcija	*Opis*

	uvećava za 1. Efekt izvođenja ove instrukcije:
	
	`rx ← [r15+1]`
	`r15 ← r15 + 1`

`call adresa`	Poziv potprograma. Trenutni sadržaj registra PC (program counter) pohranjuje na stog; potom u taj registar upisuje predanu adresu čime definira sljedeću instrukciju koja će biti izvedena.
`ret`	Vraća se iz potprograma pozvanog instrukcijom call. S vrha stoga skida adresu i postavlja je kao vrijednost registra PC (program counter).

Ako ste ovo korektno napravili, sada biste morali moći izvesti 4 program. Očekivani ispis je:

asmProgram4.txt	`4`
	`Pozdravi!`
	`Pozdravi!`
	`Pozdravi!`

Uočite: kompajlerskim direktivama (prevoditeljskim?) RESERVE moguće je naložiti da se u memoriji rezervira određen slijed memorijskih lokacija. Ovdje to koristimo za definiranje prostora koji čini stog. Parser će prilikom punjenja memorije računala programom definiranim datotekom automatski u registar r15 upisati adresu memorijske lokacije koja je labelirana kao "@stackTop" (to ponašanje je hardkodirano u parser). Želite li promijeniti adresu početka stoga (ili ako je program ne definira), to možete učiniti i sami nakon učitavanja programa a prije njegova pokretanja izravnim unosom nove vrijednosti u registar r15.

Pokrenite potom i peti primjer: on ilustrira rekurzivnu metodu koja računa sumu prvih *n* zadanih prirodnih brojeva. Parametre prenosi putem stoga te registre koje "uništava" pri izvođenju također najprije pohranjuje na stog a prije povratka restaurira sa stoga. Bez stoga, rekurzivne metode ne bismo mogli pisati! Operacijski sustavi koji podržavaju višedretveno izvođenje, za svaku dretvu rezerviraju njezin vlastiti stog (spominjem ovo samo da Vas podsjetim na ono što ste već učili na Operacijskim sustavima; u okviru ove zadaće, za ovaj procesor nećemo razvijati višedretveni operacijski sustav, iako bi to bila simpatična vježbica). Očekivani ispis petog programa je:

asmProgram5.txt	`6`

Hoće li program raditi ako zatražite izračun sume prvih 5 brojeva? A prvih 6? Možete li objasniti ispis koji ćete dobiti u ovom posljednjem slučaju?

Napišite još i sljedeću instrukciju:

Instrukcija	*Opis*
`iinput lokacija`	Čita redak s tipkovnice. Sadržaj tumači kao `Integer` i njega zapisuje na zadanu memorijsku lokaciju. Dodatno postavlja zastavicu `flag` na `true` ako je sve u redu, odnosno na `false` ako konverzija nije uspjela ili je drugi problem s čitanjem. `[lokacija]` ← pročitani `Integer`

Sada u datoteku `examples/prim2.txt` napišite asemblerski kod programa čijim ćete pokretanjem dobiti ponašanje prikazano u sljedećem primjeru (korisnikovi unosi su prikazani crveno), ispisi programa

crno.

```
Unesite početni broj: perica
Unos nije moguće protumačiti kao cijeli broj.
Unesite početni broj:
Unos nije moguće protumačiti kao cijeli broj.
Unesite početni broj: 3.58
Unos nije moguće protumačiti kao cijeli broj.
Unesite početni broj: -23
Sljedećih 5 brojeva je:
-22
-21
-20
-19
-18
```

Konačno, u datoteku `examples/prim3.txt` napišite asemblerski kod programa koji sadrži potprogram koji računa n-ti fibonaccijev broj **rekurzivno**. Program po pokretanju treba korisnika pitati za cijeli broj *n*. Kako nemamo instrukciju za usporedbu brojeva (u smislu manje, veće, jednako), pretpostavite da će korisnik unijeti pozitivan broj iz intervala 0 do 6. Vaš program za te vrijednosti mora producirati korektan rezultat (podesite veličinu stoga na odgovarajuću). Ako korisnik unese nešto što je manje od 0 ili veće od 6, ponašanje programa nije bitno – ne se time zamarati. Ali, ponavljam još jednom, implementacija potprograma mora biti klasična rekurzivna – nikakvo tabeliranje ili hrpa if-ova tipa ako je 2, vrati ovo, ako je 3 vrati ono, ako je 4 vrati ono treće.

Ako još niste, modificirajte program `Simulator` tako da nema hardkodiranu stazu do datoteke s programom već da očekuje jedan argument: stazu do datoteke koju treba izvesti. U programu najprije provjerite postoji li takva datoteka; ako ne postoji, ispišite korisniku poruku i prekinite program. Tek ako datoteka postoji, stvorite parser i dalje nastavite s radom programa.

Ako tijekom izvođenja programa dođe do iznimke, uhvatite tu iznimku, napišite poruku da je tijekom izvođenja programa iz datoteke XY došlo do iznimke W; ispišite potom stack trace i prekinite program.

Testovi i ovaj zadatak

Ovaj zadatak lijep je primjer situacije u kojoj testiranje nije baš jednostavno jer za testiranje jedne instrukcije moramo izgraditi cjelokupno računalo. Stoga ćemo se ovdje poslužiti alterativnim pristupom: koristit ćemo mock-objekte.

Najprije u Eclipse uključite biblioteku JUnit verzije 4.

Potom otiđite na stranicu:

http://search.maven.org/#artifactdetails|org.mockito|mockito-core|1.10.19|

i skinite jar arhivu mockito-core-1.10.19.jar.

Sa adrese:

http://search.maven.org/#search|gav|1|g%3A%22org.hamcrest%22%20AND%20a%3A%22hamcrest-core%22

skinite hamcrest-core 1.3 jar.

Sa adrese:

http://search.maven.org/#search|gav|1|g:"org.objenesis" AND a:"objenesis"

skinite

objensis 2.2 jar.

U Eclipse projektu napravite direktorij lib, i sve ove arhive smjestite unutra i zatim ih dodajte u Build Path. Prilikom predajte zadaće na Ferka, obavezno iz ZIP arhive obrišite ove biblioteke prije no što napravite upload; arhive su preko megabajta, vas je preko 100 :-) Prilikom recenzije svatko će si ponovno prekopirati ove biblioteke.

Kratki tutorijal dostupan je izravno na adresi:

http://mockito.org/

Evo jednostavan primjer koji ćemo ukratko prokomentirati.

```
package hr.fer.zemris.demo;

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

import java.util.List;

public class DemoTest {

	@Test
	public void test1() {
		@SuppressWarnings("unchecked")
		List<Integer> list = (List<Integer>)mock(List.class);

		when(list.get(0)).thenReturn(5);
		when(list.get(1)).thenReturn(6);

		int rez = metoda(list);

		assertEquals(17, rez);

		verify(list,times(1)).get(0);
		verify(list,times(2)).get(1);
		verify(list, times(1)).set(3, 17);
	}

	/**
	 * Ova metoda bi trebala u listu na poziciju 3 upisati
	 * vrijednost koja je jednaka zbroju elementa na poziciji 0
	 * i dva puta elementa na poziciji 1.
	 * @param list ulaz i izlaz
	 * @return izračunatu sumu
	 */
	public int metoda(List<Integer> list) {
		int suma = list.get(0) + list.get(1) + list.get(1);
		list.set(3, suma);
		return suma;
	}
}
```

Želimo testirati metodu `metoda` (koju smo zbog kratkoće napisali kao člansku metodu samog testa – u realnom slučaju to je kod koji je napisan negdje drugdje). Naša metoda "po ugovoru prema korisniku" prima listu, čita broj s nulte lokacije, dva puta čita broj s prve lokacije, sve zajedno sumira, rezultat upisuje u treću lokaciju i istovremeno ga vraća preko povratne vrijednosti.

Želimo provjeriti ponaša li se naša implementacija doista tako. Kada bismo implementaciji poslali referencu na neku konkretnu listu (primjerice, new ArrayList() koji bismo prigodno napunili elementima), napisan test mogao bi eventualno provjeriti kakvo je konačno stanje u toj listi; međutim, ne bismo mogli provjeriti broj poziva pojedinih metoda i slično. Stoga je ideja da metodi, umjesto neke konkretne liste, pošaljemo referencu na zamjenski objekt ("mock" objekt) koji se predstavlja kao lista ali u pozadini može pratiti još mnoštvo stvari koje kasnije možemo analizirati. Prvi korak je stvoriti zamjenski objekt:

```
List<Integer> list = (List<Integer>)mock(List.class);
```

Metoda prima kao argument razred objekta koji želimo te stvara i vraća novi zamjenski objekt koji je tog razreda. Zamjenski objekt nije ništa posebno – to je Java objekt kao i svi ostali "normalni" objekti; metoda

vraća referencu na njega i mi je pamtimo u lokalnoj varijabli.

Međutim, vraćeni objekt nije stvarno lista – ništa ne znamo o tome kako on pamti elemente i to nas ne zanima. Prvi korak koji treba napraviti nakon stvaranja zamjenskog objekta jest reći mu što da vraća pozivatelju kad pozivatelj nad njim krene pozivati određene metode. To se radi metodom:

when(mock.poziv()).thenReturn(vrijednost);

U našem konkretnom slučaju, želimo mock objektu reći da kada se nad njim pozove metoda get(0), da treba vratiti vrijednost 5, odnosno kada se pozove get(1) da vrati vrijednost 6.

```
when(list.get(0)).thenReturn(5);
when(list.get(1)).thenReturn(6);
```

Jednom kad smo ga podesili, pozivamo testiranu metodu nad tim objektom:

```
int rez = metoda(list);
```

Sada možemo provjeriti je li rezultat u skladu s očekivanim:

```
assertEquals(17, rez);
```

ali možemo provjeriti i jesu li određene metode pozvane točno očekivani broj puta. Sintaksa koja se koristi je:

when(mock, ogranicenjeBrojaPoziva).metoda();

Konkretno:

```
verify(list,times(1)).get(0);
verify(list,times(2)).get(1);
verify(list, times(1)).set(3, 17);
```

gdje prvi redak provjerava je li nad mockom metoda get s argumentom 0 pozvana točno jednom, drugi provjerava je li nad mockom metoda get s argumentom 1 pozvana točno dva puta, itd.

Nekoliko napomena.

Zamjenski objekt nije prava implementacija konkretnog razreda (u smislu: ako je lista, onda ima neku podatkovnu strukturu koja pamti elemente i slično; mock to ne zna). Stoga je prije predaje zamjenskog objekta testiranoj metodi potrebno podesiti odgovore na sva “pitanja” (pozive metoda) koje pozivatelj može pozvati nad tim objektom. U navedenom slučaju, nismo očekivali da će testirana metoda pozvati primjerice size() pa nismo rekli što ta metoda treba vratiti. S obzirom da to nismo učinili, poziv te metode bi (s obzirom da je ista tipa int) vratio defaultnu vrijednost za int što je 0. Kako naša metode ne treba ovu informaciju, to nismo niti podešavali.

Drugo, postoji više ograničenja na broj poziva koja se mogu zadavati. Mi smo koristili samo times(n); pogledajte što se još nudi.

Treće, za uporabu ove biblioteke zajedno s JUnit bibliotekom ne trebate raditi apsolutno ništa dodatnoga ako ćete sami stvarati mock objekte. Ako u više testova trebate istu vrstu mock objekata, tada stvaranje možete izdvojiti u zasebnu metodu koju ćete pozvati iz svakog testa.

Alternativa koja je malo čitljivija jest potrebne varijable deklarirati kao članske varijable testa i označiti iz

anotacijom @Mock. U tom slučaju, morate modificirati način na koji se pokreću testovi jer sam Junit ništa ne zna o navedenoj anotaciji pa će sve reference ostati null umjesto da se prije pokretanja testova inicijaliziraju mock objektima. Želite li raditi na ovaj način, razred koji sadrži testove treba anotirati s `@RunWith(MockitoJUnitRunner.class)` i potom možemo deklarirati potrebne članske varijable i njih anotirati anotacijom `@Mock`. Svaka takva članska varijabla prije pokretanja testa bit će inicijalizirana mock objektom zadanog tipa. Naravno, sam test i dalje mora podesiti što će taj odgovarati na pojedina pitanja. Evo koda koji to ilustrira.

```
package hr.fer.zemris.demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

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

import java.util.List;

@RunWith(MockitoJUnitRunner.class)
public class DemoTest2 {

	@Mock
	private List<Integer> list;

	@Test
	public void test1() {
		when(list.get(0)).thenReturn(5);
		when(list.get(1)).thenReturn(6);

		int rez = metoda(list);

		verify(list,times(1)).get(0);
		verify(list,times(2)).get(1);
		verify(list, times(1)).set(3, 17);

		assertEquals(17, rez);
	}

	public int metoda(List<Integer> list) {
		int suma = list.get(0) + list.get(1) + list.get(1);
		list.set(3, suma);
		return suma;
	}
}
```

Vaš je zadatak sljedeći. Za instrukcije load, move, push, pop, call, ret napišite odgovarajuće JUnit testove koji će provjeriti ispravan rad ovih instrukcija (ovo je obavezni dio rješenja domaće zadaće). Provjerite za svaku instrukciju da su njezinim izvođenjem doista pozvane sve očekivane metode registara/memorije te da je konačni rezultat u skladu s očekivanjem.

Skrećem pažnju da ćete za pojedine testove možda trebati i više zamjenskih objekata (jedan koji glumi memoriju, jedan koji glumi registre i slično).

Sve testove spremite u novi direktorij s izvornim kodovima naziva `tests`.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries for this homework (whether it is yours old code or someones else). However, you are free to use classes which are part of Java Standard Edition platform and which were covered on lectures. Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. Create additional source folder tests. This is for junit tests of problem 1 which are required. Read a discussion on Mockito library at the end of problem 1. Please write tests for required instructions concurrently with the rest of your homework.

You must name your project's main directory (which is usually also the project name) `HW08-`*yourJMBAG*; for example, if your JMBAG is 0012345678, the project name and the directory name must be `HW08-0012345678`. Once you are done, export the project as a ZIP archive and upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is May 5th 2016. at 08:00 AM.