M. Bagić Babac, M. Kušek

Preddiplomski studij

Računarstvo

Telekomunikacije i informatika

Informacija, logika i jezici

Skripta: Testiranjem upravljano programiranje

Ak. g. 2009./2010.

Sadržaj

1. Testiranjem upravljano programiranje

Ekstremno programiranje (engl. *Extreme Programming - XP*) je paradigma razvoja softvera koja je potpuno podređena klijentu. Kôd je u središtu razvojnog procesa. Zbog ovako provokativno promijenjenog fokusa sa zahtjeva prema sustavu na njegovu implementaciju, XP odbacuje detaljne i sofisticirane metodologije analize i dizajna kôda te podilazi onim programerima kojima je smetao birokratizirani pristup razvoju softvera. U ovu paradigmu spada između ostalog i jedan drukčiji pogled na testiranje programskog kôda – testiranjem upravljano programiranje.

Testiranjem upravljano programiranje (engl. *test–driven development, test-first development, test-driven design*) jedan je od načina pristupa programskim sustavima koji teži jednostavnosti u povezanosti dizajna kôda i njegova testiranja.

Prema idejnom tvorcu testiranjem upravljanog programiranja K. Becku sljedeće točke glavne su karakteristike takvog programiranja:

- prije pisanja bilo kakvog kôda pišu se testovi koji „ne prolaze“, a zapravo imaju veze s onim što moramo isprogramirati,
- nakon testa koji „ne prolazi“ pišemo samo onoliko minmalno kôda koliko je potrebno da test „prolazi“,
- čim je kôd proradio, prestrukturiramo ga da bude čistiji odnosno uklonimo nepotrebna ponavljanja istih redaka kôda,
- razvijanje kôda napreduje u malim koracima u kojima alterniraju testiranje i kôdiranje. Ovakve iteracije ne bi smjele trajati dulje od desetak minuta, i
- kad integriramo kôd cijelog sustava, svi jedinični testovi moraju uspješno raditi.

Iako ovakav pristup programiranju i testiranju može isprva zvučati neobično, postoje jake činjenice koje ga opravdavaju;

- svaki komadić kôda je testiran pa je bilo kakva sitna pogreška otkrivena u startu,
- testovi su ujedno i dokumentacija kôda jer u idealnom slučaju pokazuju i normalnu uporabu i očekivano ponašanje u slučaju pogreške, i
- dizajn programa potpuno je podređen testovima što u većini slučajeva dovodi do jednostavnije dizajna kôda

1.1 Jedinično testiranje

Testirati zapravo znači provjeriti ispravnost nečega. Da bi test bi valjan (*valid*), potrebno je provjeriti da ako je početno stanje A, krajnje stanje će biti B. Tradicionalni integrirani pristup testiranju, nakon što je programski kôd gotov, našao je svoj „protupristup“ u jediničnom testiranju i programiranju upravljanom testiranjem. Jedinično testiranje polazi od malih testova koji testiraju svaku metodu programskog kôda.

Jedinični (*Unit*) test je test jedne izolirane komponente, sa ponavljanjem. Izolirani test znači da se komponenta koja se testira tretira u izolaciji, bez ostalih komponenti sustava (koje se ne

testiraju) što je razlika u odnosu na integracijska testiranja. Test treba ponavljati jer može doći do razlika u rezultatima pri jednom izvršavanju komponente i pri više ponavljanja. Dakle, ako očekujemo da će komponenta koja se testira iz stanja A, poslije izvršavanja dati rezultat B, onda to mora biti i pri ponavljanju izvođenja. Može postojati dinamika kretanja stanja komponente sve dok je ona točno unaprijed definirana.

Kod testiranjem upravljanog programiranja test se uglavnom sastoji od četiri faze:

- priprema: priprema okruženja za testiranje i definiranje očekivanih rezultata,
- izvršavanje: pokretanje testa,
- validacija: usporedba dobivenih rezultata sa očekivanim rezultatima, i
- čišćenje: postavljanje okruženja onako kako je bilo prije testiranja.

2. JUnit

JUnit je besplatni alat odnosno programski okvir (engl. *open source framework*) za jediničnо testiranje u Javi. Pripada obitelji XUnit alata gdje X odgovara jeziku (ili kratici jezika) u kojem se testira. Trenutna inačica JUnita koja se koristi u trenutku pisanja ovog teksta je JUnit 4. JUnit definira testove (*test cases*) u odvojenim klasama te zbog jednostavnosti razdvajamo testirajuću od testirane klase. Testirana klasa je klasa koju testiramo, a testirajuća je ona koja testira (testiranu) klasu. Test u širem smislu može označavati testirajuću klasu (engl. *test case*) ili skupni test (engl. *test suite* - skup testirajućih klasa), a u užem smislu pod testom obično podrazumijevamo jednu testirajuću metodu.

2.1. JUnitove bilješke (*annotations*) i metode

Kad napravimo testirajuću klasu, potrebno je imati jednu ili više testirajućih metoda. Testirajuća metoda je standardana Javina metoda s tim što mora biti označena bilješkom `@Test`. Mora biti javna (`public`) i ne smije vraćati ništa (`void`). Prethodna inačica, JUnit 3, zahtijevala je da ime testirajuće metode počinje sa „test", ali to više nije nužno mada se u praksi i dalje zadržava zbog veće preglednosti.

Pored testirajuće metode koja je obvezna postoje i opcionalne metode. Donedavno su se zvale `setUp()` i `tearDown()`, no sad više nije bitno njihovo ime već one odgovaraju onim metodama koje su označne bilješkama `@Before` i `@After`. Metoda označena s `@Before` poziva se točno jednom prije izvođenja svake testirajuće metode, a metoda označena s `@After` nakon poziva svake testirajuće metode. Ako imamo više metoda označenih bilješkama `@Before` ili `@After`, ne znamo kojim će redoslijedom one biti pozivane. Ako pak specificiramo metodu označenu bilješkom `@BeforeClass` odnosno `@AfterClass`, ona će se izvršavati samo jednom na početku izvođenja testirajuće klase (`@BeforeClass`) odnosno samo jednom nakon izvođenja testirajuće klase (`@AfterClass`).

JUnit koristi takozvane `assertion` metode za testiranje. Postoje razne metode (detaljan popis potražiti na adresi http://www.junit.org/apidocs/org/junit/Assert.html), a ovdje su objašnjene one koje se najčešće koriste:

- `assertNull(Object x)` - provjera je li objekt x null
- `assertNotNull(Object x)` - provjera nije li objekt x null
- `assertTrue(boolean x)` - provjera je li uvjet x true odnosno tačan
- `assertFalse(boolean x)` - provjera je li uvjet x false odnosno netačan
- `assertEqual(Object x, Object y)` - provjera jednakosti objekata pomoću metode `equals()`
- `assertSame(Object x, Object Y)` - provjera jednakosti objekata pomoću operatora '= ='
- `assertArrayEquals([] x, [] y)` - provjera jednakosti dvaju nizova

Zapravo sve ove metode `assertXXX` imaju svoju proširenu inačicu gdje je u potpisu dodan prvi parametar vrste `String` koji predstavlja poruku koju sami pišemo, a koja će se pojavljivati ako metoda „ne prolazi“.

Uzmimo za početak klasu `StringBuffer` iz Javine knjižnice (*Java library*). Prema principu testiranjem upravljanog programiranja najprije stvorimo praznu testirajuću klasu:

```
public class StringBufferTest {

}
```

Konvencionalno je da se ime testirajuće klase sastoji od imena testirane klase sa dodanim sufiksom (ili prefiksom) „Test“. Dakle, to nije inherentno svojstvo JUnita već dogovorna premisa. Prevede li se gornji kôd i pokrene izvođač testa (*test runner*), pojavit će se poruka o pogrešci `StringBufferTest without tests`. JUnit „zna“ da u ovoj testirajućoj klasi nema nijedna metoda označena bilješkom `@Test`.
Za testiranje ispravne inicijalizacije praznog `StringBuffer`a, dodajemo sljedeću metodu u klasu `StringBufferTest`:

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

@Test
public void testEmptyBuffer() {
   StringBuffer buffer = new StringBuffer();
   assertTrue(buffer.toString().equals(""));
   assertTrue(buffer.length() == 0);
}
```

Ovdje je prikazana tipična struktura testirajuće metode koja prvo kreira objekt koji testiramo, a zatim provjerava odgovarajuća svojstva, tj. jesu li ona onakva kakvima ih očekujemo da budu. Ove (testirajuće) metode u jednoj jedinoj liniji kôda nazivaju se još i tvrdnjama (*assertions*). Detaljnije gledano;

- `assertTrue()`
  Metoda uzima booleov izraz kao parametar, a rezultat se verificira za vrijeme izvođenja. Ako je istinit (`true`), program nastavlja izvođenje, a ako nije (`false`), onda ovaj test prekida izvođenje, registrira se ispad programa (*failure*) i počinje izvođenje drugog testa ako postoji.

Napišemo li novu testirajuću metodu možemo koristiti i ostale oblike tvrdnji `assertXXX`.

```
@Test
public void testAppendString() {
   StringBuffer buffer = new StringBuffer();
   buffer.append("A string");
   assertEquals("A string", buffer.toString());
   assertEquals(9, buffer.length());
}
```

U ovoj izvedbi testa pojavljuje se:

- `assertEquals(expected, actual)`
  Metoda uspoređuje dva objekta (ili osnovnih vrsta podatka) pri čemu se za usporedbu objekata koristi metoda `equals()`, a za usporedbu osnovnih vrsta podataka operator ==.

Izvođenje ovog testa također javi ispad (*failure*) o tvrdnji koja „ne prolazi". Pomoću `assertEquals()` utvrđujemo značenje poruke o pogrešci (`expected: <9> but was: <10>`). Popravljanje podatka u ovoj metodi tvrdnje omogućit će izvođenje testa bez pogreške.

```
@Test
public void testAppendString() {
   StringBuffer buffer = new StringBuffer();
   buffer.append("A string");
   assertEquals("A string", buffer.toString());
   assertEquals(10, buffer.length());
}
```

JUnit razlikuje ispade (*failure*) od pogrešaka (*error*). Pogreška (error) se dogodi kad iznimka (*exception*) nađe put do testirajuće metode dok se test izvodi. Ovu razliku valja uvijek imati na umu. Primjerice, sljedeći test uzrokuje `NullPointerException` i JUnit ga registrira kao pogrešku (error);

```
@Test
public void testProvokeError() {
   StringBuffer buffer = null;
   buffer.append("A string");
}
```

Pogledamo li detaljnije gore napisane metode, možemo uočiti neke duple retke kôda:

```
@Test
public void testEmptyBuffer() {
   StringBuffer buffer = new StringBuffer();
   assertTrue(buffer.toString().equals(""));
   assertTrue(buffer.length() == 0);
}

@Test
public void testAppendString() {
   StringBuffer buffer = new StringBuffer();
   buffer.append("A string");
   assertEquals("A string", buffer.toString());
   assertEquals(9, buffer.length());
}
```

Kôd je dvostruk zbog generiranja testirajućih objekata što je sasvim normalna pojava kad se skup testova odnosi na istu komponentu. Obično se napravi skup objekata za testove testirajuće klase, a oni zovu priprema testa (*fixture*). JUnit dakle dopušta izdvajanje kôda potrebnog za stvaranje pripreme testa pomoću metode označene bilješkom `@Before`. Ova metoda izvodi se prije kôda testa i, što je još važnije, jedanput za svaku testirajuću metodu,

čime se osigurava da promjene objekata iz pripreme testa uzrokovane jednim testom ne mogu utjecati na drugi test. Ovako izgleda integracija pripreme i testova za našu testirajuću klasu:

```
public class StringBufferTest {
	private StringBuffer buffer;

	@Before
	protected void setUp() {
		buffer = new StringBuffer();
	}

	@Test
	public void testEmptyBuffer() {
		assertEquals("", buffer.toString());
		assertEquals(0, buffer.length());
	}

	@Test
	public void testAppendString() {
		buffer.append("A string");
		assertEquals("A string", buffer.toString());
		assertEquals(10, buffer.length());
	}
}
```

Nasuprot metodi označenoj kao `@Before`, JUnit pruža i mogućnost oslobađanja zauzetih resursa pomoću metode označene bilješkom `@After` (resursi poput datoteka, poveznica na baze podataka i slično) nakon izvođenja testa bez obzira na njegov ishod.

2.2. Pisanje testova prije programa - korak po korak

Razradimo u cijelosti jedan jednostavan primjer testiranjem upravljanog programiranja. Recimo da moramo napraviti rječnik koji prevodi njemačke riječi na engleski. Neka rječnik ima formu klase `Dictionary` koja se incijalizira datotekom i omogućava nam postavljanje upita. Treba omogućiti i alternative kod prevođenja, dakle da jedna riječ ima više mogućih prijevoda.

Počnimo s prvom mikro-iteracijom i napravimo praznu testirajuću klasu. Podsjećamo, princip testiranjem upravljanog programiranja je uvijek prvo pisati test, a onda prema sugestiji testa programirati kôd.

```
	public class DictionaryTest{

	}
```

Ime klasi `Dictionary` je već zadano pa iz nje izvedemo ime testirajuće klase, `DictionaryTest`. Obično je dovoljno jednoj testiranoj klasi dodijeliti samo jednu testirajuću klasu. No, zasad nemamo ništa osim gole klase pa će nam izvođač testa (*test runner*) kod izvođenja javiti `No tests found in DictionaryTest`. Pa dodajmo naš prvi test.

```
@Test
public void testCreation(){
	Dictionary dict = new Dictionary();
}
```

Već sad nas test tjera na donošenje nekih odluka. Koji nam parametri trebaju za konstruktor? Kako nazvati test? Neka konstruktor na početku bude bez parametara. A tipično ime za prvi test je `testCreation()`. Pokušaj izvođenja ovog testa naravno neće uspjeti. Zapravo to ćemo znati već kod prevođenja (Prevođenje možemo slobodno smatrati prvim ciklusom u testiranju!). Nedostaje nam klasa `Dictionary` pa je moramo napraviti. Praznom, naravno, ali samo zasad na početku.

```
public class Dictionary {

}
```

Iz ovoga ne vidimo ništa korisnoga mada se test uspješno izvršava. Idemo provjeriti je li rječnik prazan. Trebao bi biti.

```
@Test
public void testCreation(){
	Dictionary dict = new Dictionary();
	assertTrue(dict.isEmpty());
}
```

Dodavanjem ove metode zapravo smo donijeli svoju prvu dizajnersku odluku, a to je da sučelje naše klase `Dictionary` mora sadržavati metodu `isEmpty()`. Ovakve odluke moraju se konstantno donositi u procesu programiranja upravljanog testiranjem. Suma ovakvih malih koračića treba u konačnici dovesti do cjelokupnog dizajna sustava (evolucijski dizajn). Našim testovima dokazujemo da dizajn odgovara implementaciji što nije osobina dizajna nacrtanog na papiru. Uostalom, pomoću testova možemo utjecati i na promjene u dizajnu i mijenjati naš kôd i to tako da se ulazno-izlazni parametri ne mijenjaju.

Možda je bilo čudno testirati je li rječnik u početku prazan, a ne odmah prijeći na prijevode. To je zbog toga što uvijek testiramo osnove osnova koje nam onda služe dalje kao stabilan oslonac na koji se uvijek možemo vratiti ako nešto tijekom programiranja pođe ukrivo. Dodajemo dotičnu metodu u rječnik, opet najjednostavniju moguću, a to je vraćanje konstantne vrijednosti.

```
public class Dictionary {

	public boolean isEmpty(){
		return false;
	}
}
```

Opet se držimo postulata testiranjem upravljanog programiranja; piši samo onoliko kôda koliko je potrebno da test prođe. U prvoj iteraciji idemo čak toliko daleko da test ne treba proći. Kad to popravimo, dobivamo sljedeću metodu:

```
public class Dictionary {

      public boolean isEmpty(){
            return true;
      }
  }
```

Iako znamo da ovo neće biti naša konačna verzija metode `isEmpty()`, puštamo je zasad na miru jer naš rječnik uostalom i jest prazan u početku. Prepuštamo testovima da nas dovedu do „poboljšane“ inačice dotične metode. Idemo korak dalje u funkcioniranju rječnika.

```
public void testAddTranslation() {
          Dictionary dict = new Dictionary();
          dict.addTranslation("Buch", "book");
          assertFalse(dict.isEmpty());
}
```

Ovaj test izgleda također malo oskudno jer ne radi ništa osim što provjerava je li rječnik i dalje prazan nakon što u njega dodamo prvi prijevod. Ovako sitni koraci nikad nas neće dovesti do neke iznenadne pojave, ali naravno da u praksi možemo malo i ubrzati postupak, ali samo kad dobro vladamo „teritorijem“ kojim programiramo. Prilagodimo metodu `isEmpty()` novonastaloj situaciji.

```
public class Dictionary {

      private boolean empty = true;
      public boolean isEmpty() {
          return empty;
      }
      public void addTranslation(String german, String translated) {
            empty = false;
        }
}
```

Implementacija je jednostavna jer još nema prijevoda nijedne riječi pa idemo dodati pokoju riječ u naš mali rječnik.

```
@Test
public void testAddTranslation() {
          Dictionary dict = new Dictionary();
          dict.addTranslation("Buch", "book");
          assertFalse(dict.isEmpty());
          String trans = dict.getTranslation("Buch");
          assertEquals("book", trans);
}
```

Sad već imamo obrise pravog testa. Test pronalazi riječ i pronalazi prijevod. Ali opet ne idemo u konačnu implementaciju. Da prođe naš test, dovoljno je tek sljedeće:

```
public class Dictionary {
      public String getTranslation(String german) {
            return "book";
      }
```

```
    }
```

Ako dodamo još jedan poziv metode `addTranslation()`, onda će nas ovo vraćanje konstante natjerati da promijenimo metodu `getTranslation()`.

```
    public void testAddTwoTranslations() {

            Dictionary dict = new Dictionary();
            dict.addTranslation("Buch", "book");
            dict.addTranslation("Auto", "car");
            assertFalse(dict.isEmpty());
            assertEquals("book", dict.getTranslation("Buch"));
            assertEquals("car", dict.getTranslation("Auto"));

    }
```

Sad već razmišljamo o složenijoj izvedbi metode koja grupira dva pojma.

```
    import java.util.Map;
    import java.util.HashMap;
    public class Dictionary {

       private Map translations = new HashMap();
       public void addTranslation(String german, String translated) {
          translations.put(german, translated);
       }
       public String getTranslation(String german) {
          return (String) translations.get(german);
       }
       public boolean isEmpty() {
          return translations.isEmpty();
       }
    }
```

Mapa nam omogućava elegantno slaganje prijevoda. Testovi nam pomažu da se prisjetimo i metode `isEmtpy()` pa njezina varijabla `empty` može slobodno ići u smeće jer nam mapa daje puno toga gotovog. Ali sad moramo srediti i testirajuću klasu odnosno napraviti potrebne izmjene u kôdu (*refactoring*). Npr. možemo promijeniti imena metoda `testAddTranslation()` i `testAddTwoTranslations()` u `testOneTranslation()` i `testTwoTranslations()`. Linija stvaranja rječnika može se preseliti u `@Before` odsječak za pripremu testa. Metodama `assertXXX()` možemo dodati deskriptivne komentare kao prvi parametar za buduću dijagnostiku. Naš obnovljeni `DictionaryTest` sad izgleda ovako;

```
    public class DictionaryTest {

            private Dictionary dict;

            @Before
            protected void setUp() {
               dict = new Dictionary();
            }

            @Test
            public void testCreation() {
```

```
            assertTrue(dict.isEmpty());
        }

        @Test
        public void testOneTranslation() {
            dict.addTranslation("Buch", "book");
            assertFalse("dict not empty", dict.isEmpty());
            String trans = dict.getTranslation("Buch");
            assertEquals("translation Buch", "book", trans);
        }

        @Test
        public void testTwoTranslations() {
            dict.addTranslation("Buch", "book");
            dict.addTranslation("Auto", "car");
            assertFalse("dict not empty", dict.isEmpty());
            assertEquals("translation Buch", "book",
                         dict.getTranslation("Buch"));
            assertEquals("translation Auto", "car",
                         dict.getTranslation("Auto"));
        }
    }
```

Zaboravili smo da smo obećali alternative kod prevođenja riječi. Pa evo odmah test iz toga. Najjednostavniji.

```
    public void testTranslationWithTwoEntries() {

        dict.addTranslation("Buch", "book");
        dict.addTranslation("Buch", "volume");
        String trans = dict.getTranslation("Buch");
        assertEquals("book, volume", trans);

    }
```

Nije baš najelegantniji izbor slagati riječi odvojene zarezom u jedan objekt, ali nema veze. Programiranje upravljano testiranjem dopušta loše trenutne odluke jer ih možemo promijeniti u bilo kojem trenutku kad izađe na vidjelo da su stvarno loše. Jednostavna rješenja su prihvatljiva tako dugo dok novi testovi ne dovedu do kompleksnijeg dizajna.

```
public class Dictionary {

    public void addTranslation(String german, String translated) {
            String before = this.getTranslation(german);
            String now;
            if (before == null) {
                now = translated;
            } else {
                now = before + ", " + translated;
            }
            translations.put(german, now);
        }
}
```

Osvrnemo li se na trenutak unazad vidimo da smo počeli s testom koji je bio motiviran specifikacijskim zahtjevima. Dok smo pisali testove morali smo donositi odluke o našem

testiranom objektu (OUT – *object under test*) odnosno njegovom javnom sučelju. Testovi su na neki način pisali dokumentaciju o testiranoj klasi.

Dosad smo ignorirali dio koji se odnosi na datoteku rječnika. Neka su u njoj riječi odnosno prijevodi posloženi prema uzorku: <German word>=<translation >. Ovo dakako nema nikakve veze s XML-om (barem ne u ovome trenutku). Neka su mogući višestruki ulazi. Prvi pokušaj pisanja testa neka bude ovakav;

```
@Test
public void testSimpleFile() {
        dict = new Dictionary("C:\\temp\\simple.dic");
        assertTrue(! dict.isEmpty());
}
```

No sada smo iznenada postali ovisni o vanjskoj datoteci. Da malo reduciramo tu ovisnost, ubacit ćemo `java.io.InputStream` umjesto datoteke, koji uostalom lako omotamo u datoteku.

```
@Test
public void testTwoTranslationsFromStream() {
   String dictText = "Buch=book\n" + "Auto=car";
   InputStream in = new StringBufferInputStream(dictText);
   dict = new Dictionary(in);
   assertFalse(dict.isEmpty());
}
```

Dodajmo u naš test još nekoliko tvrdnji. Zanemarit ćemo na trenutak to što je klasa `StringBufferInputStream` zastarjela (*deprecated*).

```
@Test
public void testTwoTranslationsFromStream() {
        String dictText = "Buch=book\n" + "Auto=car";
        InputStream in = new StringBufferInputStream(dictText);
        dict = new Dictionary(in);
        assertFalse("dict not empty", dict.isEmpty());
        assertEquals("translation Buch", "book",
           dict.getTranslation("Buch"));
        assertEquals("translation Auto", "car",
           dict.getTranslation("Auto"));
}
```

S obzirom na odabir dizajna naše datoteke možemo pomoću Javinih *properties* uvesti sljedeću implementaciju.

```
import java.util.*;
import java.io.*;

public class Dictionary {

   ...
   public Dictionary(InputStream in) throws IOException {
      this.readTranslations(in);
   }
```

```
        private void readTranslations(InputStream in) throws IOException{
            Properties props = new Properties();
            props.load(in);
            Iterator i = props.keySet().iterator();
            while (i.hasNext()) {
               String german = (String) i.next();
               String trans = props.getProperty(german);
               this.addTranslation(german, trans);
            }
        }
    }
```

Bacanje iznimke "throws IOException" osigurava da testirajuća metoda također mora baciti `IOException`. Time povjeravamo JUnitu hvatanje iznimki u obliku pogrešaka. U tome duhu pišemo sljedeći test.

```
    @Test
    public void testTranslationsWithTwoEntriesFromStream()
          throws IOException {
       String dictText = "Buch=book\n" + "Buch=volume";
       InputStream in = new StringBufferInputStream(dictText);
       dict = new Dictionary(in);
       String trans = dict.getTranslation("Buch");
       assertEquals("book, volume", trans);
    }
```

I sad, nakon svega, odjednom shvatimo da korištenje Javinih *properties* ne vodi nikuda jer `load(InputStream)` metoda pregazi *property* istog imena ako se pojavi dupli ulaz, a nama treba da budu zapisani svi prijevodi neke riječi. Zapravo uviđamo da parsiranje datoteke uopće nije tako jednostavna stvar i najbolje bi bilo to prepustiti zasebnoj klasi. Neka se ona zove `DictionaryParser`. Konačno smo došli do točke kad treba ozbiljnije rekonstruirani naš kôd. Mijenjamo `java.io.InputStream` u `java.io.Reader` jer je ova klasa bolja za čitanje teksta i brine o ispravnoj konverziji između bajtova i znakova.

```
    import java.io.*;

    public class DictionaryParserTest {

    public DictionaryParserTest(String name) {...}
    private DictionaryParser parser;
    private DictionaryParser createParser(String dictText)
       throws IOException {
       Reader reader = new StringReader(dictText);
       return new DictionaryParser(reader);
    }

    private void assertNextTranslation(String german, String trans)
       throws Exception {
       assertTrue(parser.hasNextTranslation());
       parser.nextTranslation();
       assertEquals(german, parser.currentGermanWord());
       assertEquals(trans, parser.currentTranslation());
    }
```

```
        @Test
        public void testEmptyReader() throws Exception {
            parser = this.createParser("");
            assertFalse(parser.hasNextTranslation());
        }

        @Test
        public void testOneLine() throws Exception {
            String dictText = "Buch=book";
            parser = this.createParser(dictText);
            this.assertNextTranslation("Buch", "book");
            assertFalse(parser.hasNextTranslation());
        }

        @Test
        public void testThreeLines() throws Exception {
            String dictText = "Buch=book\n" +
                "Auto=car\n" +
                "Buch=volume";
            parser = this.createParser(dictText);
            this.assertNextTranslation("Buch", "book");
            this.assertNextTranslation("Auto", "car");
            this.assertNextTranslation("Buch", "volume");
            assertFalse(parser.hasNextTranslation());
        }
    }
```

Uduplane metode su zamijenjene jednostrukima u privatnim metodama. Zastarjela klasa `StringBufferInputStream` zamijenjena je klasom `StringWriter`. Sve izjave `throws` zamijenjene su općom iznimkom `Exception` umjesto `IOException`. Time nije ništa izgubljeno, a održavanje testa je pojednostavljeno jer više ne treba mijenjati `throws` dio. I, na kraju, neka vrijedi sljedeća implementacija parsera.

```
        import java.io.*;

        public class DictionaryParser {

            private BufferedReader reader;
            private String nextLine;
            private String currentGermanWord;
            private String currentTranslation;

            public DictionaryParser(Reader unbufferedReader) throws
                IOException {
                reader = new BufferedReader(unbufferedReader);
                this.readNextLine();
            }

            public String currentTranslation() {
                return currentTranslation;
            }

            public String currentGermanWord() {
                return currentGermanWord;
            }

            public boolean hasNextTranslation() {
                return nextLine != null;
```

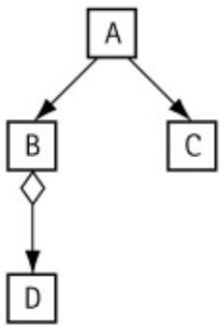
```
        }

        public void nextTranslation() throws IOException {
            int index = nextLine.indexOf('=');
            currentGermanWord = nextLine.substring(0, index);
            currentTranslation = nextLine.substring(index + 1);
            this.readNextLine();
        }

        private void readNextLine() throws IOException {
            nextLine = reader.readLine();
        }
    }
```

A ovako može izgledati klasa rječnika.

```
public class Dictionary {
        ...
        private void readTranslations(Reader reader)
            throws IOException {
            DictionaryParser parser = new DictionaryParser(reader);
            while (parser.hasNextTranslation()) {
                parser.nextTranslation();
                String german = parser.currentGermanWord();
                String trans = parser.currentTranslation();
                this.addTranslation(german, trans);
            }
        }
}
```

U ovom primjeru s rječnikom koristili smo pristup odozgo-dolje (*top-down*) jer smo krenuli s općenitim funkcionalnostima. No, u praksi je lakše razvijati i testirati sustav odozdo-gore (*bottom-up*) jer obrnut pristup traži korištenje, tzv. lažnih (*mock*) objekata.

Odozgo-dolje programiranje počinje s klasom A, a klase B i C su zamijenjene lažnima. Onda razvijamo klase B i C i za to nam treba i lažna klasa D. Time programiramo „izvana“ dakle, javno sučelje i napredujemo ka unutarnjoj privatnoj strukturi klasa. Zato ovaj pristup zovemo i izvana-ka-unutra.



Slika 1. Primjer za pristupe testiranju

Odozdo-gore programiranje počinje neovisnim klasama koje koristimo za gradnju složenijih i ovisnijih objekata. Dakle, s obzirom na sliku 1, redom D-B-C-A.

3. Lažni (mock) objekti

Osnovi postulat jediničnog testiranja je napraviti testirajuću klasu što je moguće više „lokalnom“. To znači da treba testirati samo objekt kojim se trenutno bavimo, a ne druge koji mu trebaju za njegov posao odnosno s kojima surađuje ili im proslijeđuje zadatke. Ovaj cilj maksimalne neovisnosti je donekle moguće postići korištenjem, tzv. umjetnih ili lažnih objekata (engl. *dummy, mock objects*). Dakle, zamijenimo „pomoćne“ objekte objektima koji glume ono što nama treba. Pogledajmo to na sljedećem primjeru.

3.1. Prvi primjer: EuroCalculator

Napravimo Euro-kalkulator koji pretvara i vraća zadanu valutu u eurima. Instanca klase `ExchangeRateProvider` koristit će se za vraćanje trenutnih omjera (engl. *exchange rates*). U duhu testiranjem upravljanog programiranja prvo pišemo test:

```
public class EuroCalculatorTest {

      @Test
      public void testEUR2EUR() {
         double result = new EuroCalculator().valueInEuro(1.0, "EUR");
         assertEquals(1.0, result, 0.00001)
      }
      @Test
      public void testUSD2EUR() {
         double result = new EuroCalculator().valueInEuro(1.0, "USD");
         assertEquals(1.1324, result, 0.00001);
   }
}
```

Klasu `ExchangeRateProvider` koristimo kao gotovu komponentu koja nam daje omjer za sve valute na mreži.

```
public class ExchangeRateProvider {

         public double getRateFromTo(String fromCurrency, String to) {
            double retrievedRate = ... // Pristup poslužitelju preko mreže.
            return retrievedRate;
         }
}
```

Iako je implementacija klase `EuroCalculator` relativno jednostavna, naši testovi nailaze na neke probleme. Prvo, testirajuća metoda `testUSD2EUR()` će vjerojatno raditi dobro samo neko vrijeme dok se omjer (engl. *exchange rate*) između eura i dolara ne promijeni. Drugo, pristup poslužitelju s podacima o omjerima tečaja je vrlo delikatna stvar jer mu pristupamo preko mreže. Poslužitelj stoga može imati veliko vrijeme odziva ili čak je nedostupan s obzirom na opterećenost mreže. Dakle, ovakva ovisnost o vanjskim uvjetima našeg objekta može značiti da test neće valjati iako nam neće javiti nikakvu pogrešku. Ovaj problem u postupku testiranja možemo riješiti pomoću oponašajućeg (lažnog) poslužitelja kojemu možemo odmah dati očekivani omjer.

```
public class DummyProvider extends ExchangeRateProvider {

        private double dummyRate;
        public DummyProvider(double dummyRate) {
           this.dummyRate = dummyRate;
        }
        public double getRateFromTo(String from, String to) {
           return dummyRate;
        }
}
```

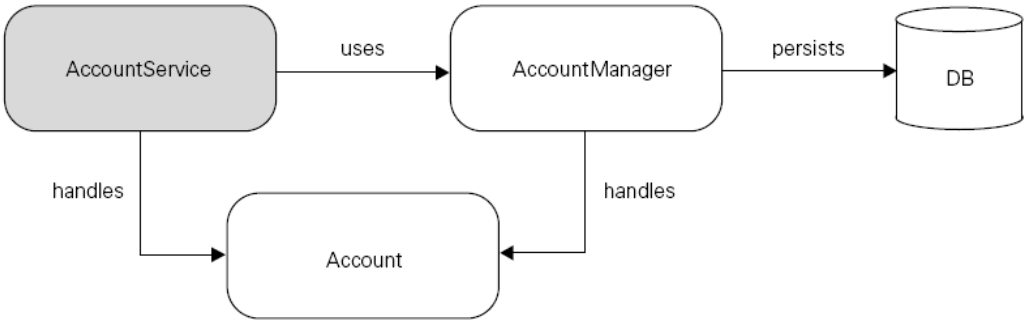
Sad još samo treba ovaj lažni objekt „prokrijumčariti“ u `EuroCalculator`. Jedan način je proširiti potpis metode `valueInEuro()` parametrom `ExchangeRateProvider type`. Time svi klijenti `EuroCalculator`a imaju dostupnu instancu `ExchangeRateProvider`a. Ovakav pristup možemo naknadno promijeniti ako ne bude zadovoljavao. Sad imamo mogućnost staviti očekivanu valutu u konstantu ACCURACY pa neka ovako izgleda promijenjeni test.

```
public class EuroCalculatorTest {

        private final static double ACCURACY = 0.00001;

        @Test
        public void testEUR2EUR() {
           ExchangeRateProvider provider = new DummyProvider(1.0);
           double result = new EuroCalculator().
           valueInEuro(2.0, "EUR", provider);
           assertEquals(2.0, result, ACCURACY);
        }

        @Test
        public void testUSD2EUR() {
           ExchangeRateProvider provider = new DummyProvider(1.1324);
           double result = new EuroCalculator().
           valueInEuro(1.5, "USD", provider);
           assertEquals(1.6986, result, ACCURACY);
        }
}
```

Dobili smo brze i stabilne testove koji ne ovise o mijenjajućim omjerima, ali moramo biti svjesni da ovime testiramo samo klasu `EuroCalculator`, a ne poslužitelja koji daje omjere. Pretpostavljamo da je poslužitelja već testirao davatelj usluge. Da mi moramo testirati i poslužitelja, bilo bi logično to napraviti u posebnom odvojenom skupnom testu (engl. *test suite*).

3.2. Drugi primjer: AccountService

Sljedeći primjer opisuje sustav `AccountService` kojim je ostvarena usluga prijenosa novca s jednog računa na drugi [3].



Slika 2. Usluga upravljanja računima

Klasa `AccountService` pruža uslugu koristeći instance klase `Account` te klase `AccountManager` za prijenos podataka iz baze podataka. Usluga prijenosa koja nas zanima je realizirana metodom `AccountService.transfer()`. Bez lažnih objekata njezino testiranje bi impliciralo postavljanje baze podataka i niz drugih zahtjevnih aktivnosti što je prekomplicirano. U realizaciji usluge koristimo klasu `Account` s atributima `accountID` (broj računa) i `balance` (stanje računa).

```
public class Account
{
   private String accountId;
   private long balance;

   public Account(String accountId, long initialBalance){
      this.accountId = accountId;
      this.balance = initialBalance;
   }
   public void debit(long amount){
      this.balance -= amount;
   }
   public void credit(long amount){
      this.balance += amount;
   }
   public long getBalance(){
      return this.balance;
   }
}
```

Sučelje `AccountManager` ima za cilj upravljati životnim vijekom objekata klase `Account` (pronalazi ih preko broja računa i ažurira ih).

```
public interface AccountManager{
      Account findAccountForUser(String userId);
      void updateAccount(Account account);
}
```

Metoda za prijenos novca između dva računa koristi sučelje `AccountManager` za pronalaženje računa preko korisničkog imena te njihovo ažuriranje.

```
public class AccountService{

   private AccountManager accountManager;
```

```
    public void setAccountManager(AccountManager manager){
        this.accountManager = manager;
    }

    public void transfer(String senderId, String beneficiaryId,long amount){
        Account sender = this.accountManager.findAccountForUser(senderId);
        Account beneficiary =
            this.accountManager.findAccountForUser(beneficiaryId);
        sender.debit(amount);
        beneficiary.credit(amount);
        this.accountManager.updateAccount(sender);
        this.accountManager.updateAccount(beneficiary);
    }
}
```

Želimo pomoću jediničnog testa ispitati ponašanje metode `AccountService.transfer`. U tu svrhu koristimo lažnu implementaciju sučelja `AccountManager` jer metoda prijenosa koristi ovo sučelje, a mi je moramo testirati u izolaciji. Ovdje ćemo stvoriti lažni objekt „ručno“ pa ovako može izgledati takva izvedba;

```
public class MockAccountManager implements AccountManager{
    private Hashtable accounts = new Hashtable();
    public void addAccount(String userId, Account account){
        this.accounts.put(userId, account);
    }
    public Account findAccountForUser(String userId){
        return (Account) this.accounts.get(userId);
    }
    public void updateAccount(Account account){
        // do nothing
    }
}
```

A ovako može izgledati testirajuća klasa.

```
public class AccountServiceTest {

        @Test
        public void testTransferOk(){

        MockAccountManager mockAccountManager = new MockAccountManager();
        Account senderAccount = new Account("1", 200);
        Account beneficiaryAccount = new Account("2", 100);
        mockAccountManager.addAccount("1", senderAccount);
        mockAccountManager.addAccount("2", beneficiaryAccount);
        AccountService accountService = new AccountService();
        accountService.setAccountManager(mockAccountManager);

        accountService.transfer("1", "2", 50);

        assertEquals(150, senderAccount.getBalance());
        assertEquals(150, beneficiaryAccount.getBalance());
        }
}
```

3.3. EasyMock

Jedinično (*unit*) testiranje je dakle izolirano testiranje dijela programa od ostatka. Međutim, većina programskih cjelina, komponenti ili dijelova koji se testiraju ne mogu funkcionirati neovisno o ostatku te se tako ni njihova funkcionalnost ne može testirati izolirano. Da bi ga ipak testirali izolirano, potrebno je simulirati komponente o kojima ovisi komponenta koju testiramo. Tako smo došli do lažnih (engl. *mock*) objekata. Za njihovo automatizirano stvaranje i korištenje razvijeno je niz alata i okruženja od kojih ćemo ovdje upoznati EasyMock.

EasyMock je programski okvir koji osigurava lažne objekte za sučelja u JUnit-u generirajući ih tijekom izvođenja koristeći `proxy` katalog, točnije objekt `java.lang.reflect.Proxy`. Zahvaljujući načinu na koji EasyMock pamti očekivana stanja većina mijenjanja kôda neće prouzrokovati promjene na lažnim objektima. To EasyMock čini iznimno korisnim u testiranjem upravljanom programiranju. Kad se kreira lažni objekt, ustvari se kreira posrednički (engl. *proxy*) objekt koji zauzima mjesto pravom objektu. Za ovo je sposoban jer mu se pri kreiranju predaje sučelje objekta za koji se želi kreirati lažni (mock) objekt. Postoje više vrsta lažnih objekata: engl. *regular mock, strict mock, nice mock*.

U svim slučajevima u testu se definira koje se metode lažnog objekta pozivaju i što se očekuje kao rezultat. Kod običnih (engl. *regular mock*) lažnih objekata test ne prolazi ako je pozvana metoda, a nije se očekivalo da se pozove, ili ako nije pozvana metoda, a očekivalo se da se pozove. Redoslijed pozivanja ovdje nije bitan. Kod finih (engl. *nice*) lažnih objekata test ne prolazi ako nije pozvana metoda za koju se očekivalo da bude pozvana, a ako je pozvana metoda za koju se nije očekivalo da bude, ta metoda će vratiti odgovarajuću podrazumijevanu vrijednost (null, 0 , false). Kod strogih (engl. *strickt mock*) lažnih objekata test ne prolazi ako je pozvana metoda, a nije se očekivalo, ili ako nije pozvana metoda, a očekivalo se. Redoslijed pozivanja je bitan, tj. test ne prolazi ako redoslijed pozivanja metoda nije isti kao očekivani.

Postoje dva načina stvaranja lažnih (mock) objekata; direktno i kontrolirano. Kod direktnog načina pojedinačni lažni objekti su neovisni dok kontroliranim načinom postaju povezani, što može biti korisno kod testiranja poziva metoda iz drugih lažnih objekata.

Nekad je potrebno izazvati poziv testirajuće metode dva ili više puta kako bi test bio vjerodostojan. EasyMock osigurava takvu simulaciju pomoću sljedećih metoda:

- `times(min, max)` – broj poziva mora biti u definiranom opsegu,
- `atLeastOnce()` – mora postojati barem jedan poziv,
- `anyTimes()` – metoda koja se testira mora biti pozvana nula ili više puta.

Životni ciklus EasyMockovog objekta može se podijeliti u četiri različita dijela kojima zapravo odgovaraju i istim imenom nazvane osnovne metode:

- CREATE - kreiranje lažnog objekta,
- EXPECT - zapis očekivanja,

- REPLAY - pokretanje izvršavanja sa prethodno zapisanim očekivanjima,
- VERIFY - verifikacija očekivanih poziva.

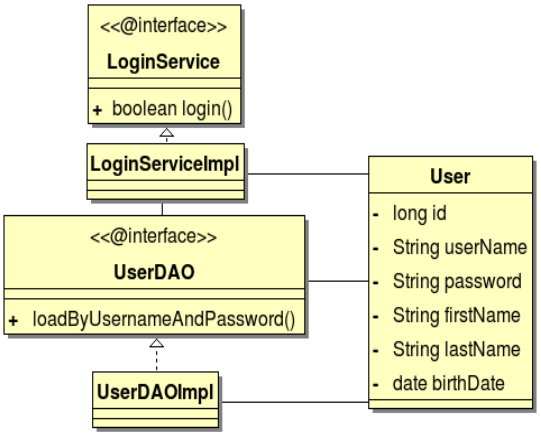
Korištenjem EasyMocka prethodni primjer usluge prijenosa novca s jednog računa na drugi može se implementirati tako da testirajuća klasa izgleda ovako;

```
import static org.junit.Assert.*;

import org.easymock.*;
import org.junit.*;

public class AccountServiceTest {

	private AccountManager mockAccountManager;

	@Before
	public void setUp(){
		mockAccountManager = EasyMock.createMock(AccountManager.class);
	}
	@Test
	public void testTransferOk(){
		Account senderAccount = new Account("1", 200);
		Account beneficiaryAccount = new Account("2", 100);

		mockAccountManager.updateAccount(senderAccount);
		mockAccountManager.updateAccount(beneficiaryAccount);


	EasyMock.expect(mockAccountManager.findAccountForUser("1")).andReturn
		(senderAccount);

	EasyMock.expect(mockAccountManager.findAccountForUser("2")).andReturn
		(beneficiaryAccount);

		EasyMock.replay(mockAccountManager);

		AccountService accountService = new AccountService();
		accountService.setAccountManager(mockAccountManager);
		accountService.transfer("1", "2", 50);

		assertEquals(150, senderAccount.getBalance());
		assertEquals(150, beneficiaryAccount.getBalance());
	}
}
```

Metodom `createMock()` stvara se lažni objekt, a kao parametar joj se proslijedi sučelje koje lažni objekt treba odglumiti prilikom izvođenja. Metodom `expect()` specificira se ponašanje lažnog objekta koje se prilikom testiranja izvodi. Navede se ime metode koju treba pozvati te njezini ulazni i izlazni parametri. Metoda `replay()` aktivira lažni objekt pa joj treba kao parametar proslijediti referencu lažnog objekta.

3.4. Treći primjer: LoginService

Sljedeći primjer koristi JUnit i EasyMock. Za scenarij testiranja koristi se pojednostavljeni primjer usluge za autentikaciju korisnika prikazan na slici.



Slika 3. Dijagram usluge za autentikaciju

Realizacija sučelja sa slike može izgledati ovako:

```
public interface LoginService {
    boolean login(String userName, String password);
}
```

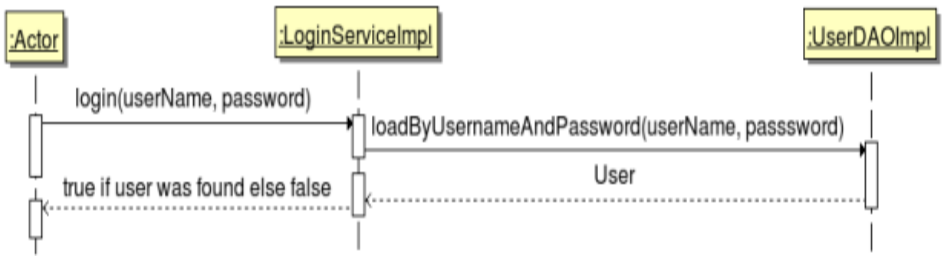
Metoda `login()` procesira zahtjev za prijavu na sustav. Ako korisnik sa proslijeđenom lozinkom i korisničkim imenom, postoji metoda vraća `true`, inače vraća `false`.
Evo i realizacije suečelja `UserDAO`:

```
//Osigurava pristup podacima za prijavu
public interface UserDAO {
     User loadByUsernameAndPassword(String userName, String password);
}
```

Metoda `loadByUsernameAndPassword()` vraća objekt `User` identificiran korisničkim imenom i lozinkom. Sada kad znamo sučelja osnovnih funkcija sustava i dijelove koje želimo testirati možemo napisati testirajuću metodu u JUnitu. Testiranje se sastoji se od tri cjeline: pripreme okruženja (priprema testa), samog testa i vraćanja u prethodno stanje.

```
public class LoginServiceTest {

      private LoginServiceImpl service;
      private UserDAO mockDao;

      @Before
      public void setUp() {
          service = new LoginServiceImpl();
          mockDao = createStrictMock(UserDAO.class);
          service.setUserDAO(mockDao);
      }
}
```

Metoda `setUp()` u kojoj se instanciraju potrebni objekti, sama usluga koja se testira i lažni objekt `userDAO`. Test bi trebao provjeriti sljedeći scenarij:



Slika 4. Slijedni dijagram usluge autentikacije

Prilikom realizacije svakog testa potrebno je predvidjeti što više mogućih scenarija. Jedan od njih predviđa da su korisničko ime i lozinka ispravni, pa može pripadati gore navedenoj klasi.

```
@Test
public void testRosyScenario() {
        User result = new User();
        String userName = "testUserName";
        String password = "testPassword";
        String passwordHash = "[illegible]";
        expect(mockDao.loadByUsernameAndPassword(eq(userName),
              eq(passwordHash))).andReturn(results);
        replay(mockDao);
        assertTrue(service.login(userName, password));
        verify(mockDao);
}
```

Prvo se definira očekivani rezultat, `result`. U ovom slučaju to je samo `User` objekt, tj. samo se provjerava je li rezultat određenog tipa. Zatim se deklariraju varijable koje se prosljeđuju servisu, `password` i `userName`. Zatim se poziva metoda `expect`, statička metoda klase EasyMock. Ovaj dio govori da će metoda `loadByUserNameAndPassword` biti pozvana, tj. test neće proći ako metoda ne bude pozvana. Također se definira da rezultat poziva ove metode bude objekt `User`. Poziv metode `reply()` određuje pokretanje samog testa. Metoda `assertTrue()` pokreće kôd koji se testira i provjerava je li rezultat vrijednost `true`. I konačno, metoda `verify()` validira je li svaka metoda, za koju se očekuje da se pozove, pozvana i da li je to bilo određenim redoslijedom.

Literatura

[1] K. Beck, *Test-Driven Developmet by Example*, Addison-Wesley, 2003.

[2] J. Link, P. Frohlich, *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufmann Publishers, 2003.

[3] V. Massol, T. Husted, *JUnit in Action,* Manning Publications, 2004.

[4] J. B. Rainsberger, *JUnit Recepies, Practical Methods for Programmer Testing,* Manning Publications, 2005.

[5] http://www.junit.org/ JUnitova kućna adresa, svibnja 2010.

[6] http://easymock.org/ EasyMockova kućna adresa, svibnja 2010.