SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 2535

Sustav grupiranja i praćenja komponenti pri usluzi m-zdravstva

Marko Martinović

Zagreb, lipanj 2012.

Sadržaj

Uvod

Pojam sustava grupiranja i praćenja komponenti odnosi se na testiranje programske potpore u obliku samostalnih testnih jedinica i integracijskog testa čime se vrši ispitivanje veze između dvaju ili više komponenata. Programski jezik Java nudi dva radna okvira za izvođenje testova pri čemu se u ovom radu koristi JUnit. Grupiranje i praćenje komponenti se vrši nad aplikacijom koja obavlja uslugu m-zdravstva. Aplikacija je izvedena u programskom jeziku Java te rađena u Spring MVC-u. U daljnjem tekstu objašnjen je rad Spring MVC-a kako bi se razumio način na koji aplikacija funkcionira te kako se provode testovi.

Praktični dio ovog rada je izrada testova u programskom jeziku Java za jedinične i integracijske testove. Opisan je način na koji se provode takvi testovi.

1 Testiranje

U ovom poglavlju opisan je pojam testiranja i koji su načini na koji se obavlja testiranje programskog kôda. Testiranje je pojam koji označava proces izvođenja programa sa svrhom pronalaženja pogrešaka. Aktivnosti koje se provode u procesu testiranja služe radi otkrivanja informacija o ispravnosti i kvaliteti, te poboljšanja pronalaženjem kvarova i problema testiranog produkta. Testiranje programske podrške zasniva se na dinamičkoj verifikaciji ponašanja programa na konačnom broju testnih scenarija, pogodno odabranih iz beskonačne domene izvođenja, obzirom na očekivano ponašanje [1]. Za testiranje programa pisanih u programskom jeziku Java moguća su dva radna okvira (eng. *frameworks*) za korištenje:

- JUnit
- Test NG

JUnit je jednostavan radni okvir otvorenog kôda (eng. *open source*) koji se koristi za pisanje i pokretanje automatiziranih testova koji se mogu ponovno koristiti. JUnit uključuje:

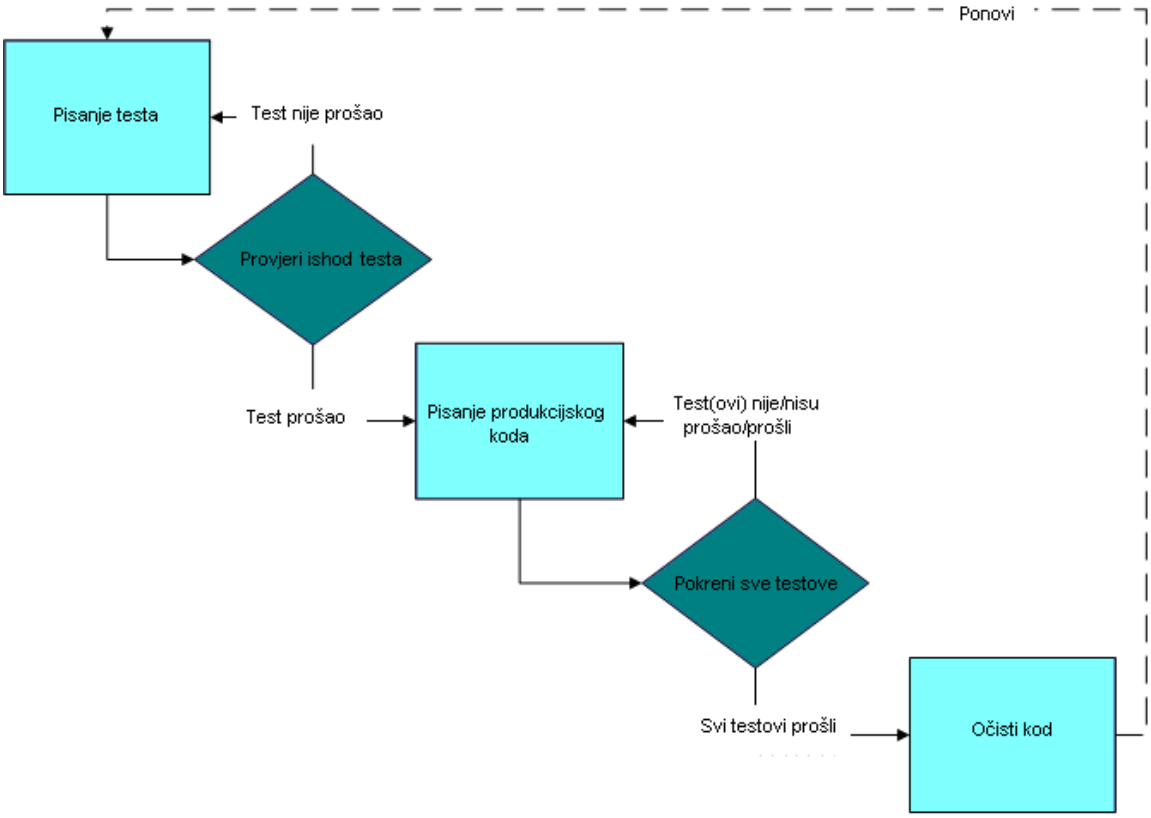
- korištenje tvrdnji za ispitivanje očekivanih rezultata
- pribor za ispitivanje podataka koji su zajednički testnim slučajevima
- skupine testnih slučajeva za jednostavno organiziranje i vođenje ispitivanja
- grafičke i tekstualne pokretače testova

Testni slučaj je razred koji sadrži određeni broj testnih metoda. JUnit dolazi u dvije inačice. Starija inačica JUnit3 koristi tehniku nasljeđivanja klase TestCase koja sadrži metode za postavljanje i brisanje objekata prije i poslije svakog testa. Također sadrži metode za uspoređivanje traženog i dobivenog rezultata ispitivajući metodu neke klase. JUnit4 inačica koristi bilješke (eng. *annotations*) kako bi se označilo koja metoda je test. Poboljšanje koje uvodi JUnit4 je ta da nema više klase koja se mora naslijedit kako se bi se obavilo testiranje.

TestNG je radni okvir za testiranje koji se koristi za jedinične testove (razred u izoliranoj okolini neovisno od ostalih) pa sve do integracijskog testiranja (testira se cijeli sustav izrađen od nekoliko razreda i paketa). Tri su koraka u pisanju testova:

- napiše se poslovna logika koja se treba provesti te se stavi TestNG bilješka
- doda se informacija o testu u xml dokument
- pokrene se testiranje

XML dokument sadrži podatke o tome koji će se testovi pokrenuti. Dijagram toka izvođenja testova prikazuje slika (Sl. 1.1) [2].



Sl. 1.1 Dijagram toka testiranja

Testiranje programske potpore, ovisno o metodi s kojom se testira, može se provesti u bilo koje vrijeme tijekom procesa razvoja. Međutim, sav trud uložen u testiranje se pojavljuje kada su svi zahtjevi definirani i kada je proces kôdiranja završen. Pokazuje se da je ispravak pogreške jeftiniji ako se pojavi u ranijoj fazi razvoja.

1.1 Opseg testiranja (eng. Scope)

Osnovna uloga testiranja je otkrivanje grešaka kako bi se otkrile mane i ispravile. Testiranje ne osigurava ispravan rad programske podrške u svim uvjetima, ali može osigurati neispravan rad u određenim uvjetima. Opseg testiranja često uključuje pregled tog kôda kao i njegovo izvršenje u različitim okruženjima i uvjetima. Trenutna kultura razvijanja programske potpore dijeli razvojni tim od tima za testiranje [9].

1.2 Funkcijsko i nefunkcijsko testiranje

Funkcijsko testiranje odnosi se na aktivnosti koje provjeravaju određenu akciju ili funkciju u kôdu. Funkcijsko testiranje pokušava odgovoriti na pitanja „da li koristnik može uraditi ovo“ ili „da li ova određena odlika radi“.

Nefunkijsko testiranje se odnosi na aspekt programske podrške koji ne mora biti povezan na određenu funkciju ili korisnikovu akciju kao što je skalabilnost ili neka druga performansa, ponašanje u određenim uvjetima ili sigurnost. Testiranje će odrediti ekstreme u skalabilnosti ili performansi što dovodi do nestabilnog izvršavanja. Nefunkcijski zahtjevi su oni koji reflektiraju kvalitetu proizvoda, posebice u kontekstu perspektive pogodnosti prema svojim korisnicima [9].

1.3 Nedostaci i kvarovi (eng. defects and failures)

Programski nedostaci nisu uvijek uzrokovani pogreškama u kodiranju. Izvor skupih nedostataka se nalazi u zahtjevima, primjerice u neotkrivenim zahtjevima koji rezultira u pogreškama programskog dizajnera.

Kvarovi nastaju tijekom sljedećih radnji: programer učini pogrešku, što rezultira nedostatkom (eng. *bug*) u izvornom kodu. Ako je taj nedostatak izvršen u pojedinim slučajevima će sustav proizvesti pogrešan rezultat što dovodi do kvara. Svaki nedostatak ne mora rezultirati kvarom. Primjerice, nedostatci u mrtvom kôdu (kôd koji se nikad ne izvrši) nikada neće rezultirati kvarom [9].

1.4 Rani pronalazak kvara

Tablica (Tablica 1.1) prikazuje trošak popravka kvara ovisno u kojoj fazi razvoja se kvar pronašao. Na primjer, kvar koji se pronašao nakon izdavanja programa je 10-100 puta skuplji za ispraviti od onoga koji se pronašao tijekom prikupljanja zahtjeva [9].

Tablica 1.1 Trošak popravka kvara ovisno o fazi razvoja

Trošak popravka kvara		Vrijeme otkrivanja				
		Zahtjevi	Arhitektura	Konstrukcija	Sustavski test	Nakon izdavanja
Vrijeme uvođenja	Zahtjevi	1x	3x	5-10x	10x	10-100x
	Arhitektura	-	1x	10x	15x	25-100x
	Konstrukcija	-	-	1x	10x	10-25

1.5 Razine testiranja

Testovi su često grupirani ovisno o tome gdje su dodani u razvojnom procesu ili ovisno o razini specifičnosti samog testa [9].

- Jedinični testovi – poznato kao testiranje komponenti, odnosi se na testove koji potvrđuju rad određenog dijela koda, uobičajeno na fukcijskoj razini. U objektno-orijentiranom okruženju to je uobičajeno na razini razreda, a minimalno što test treba provjeriti je konstruktor i destruktor. Neka funkcija može imati više testova kako bi se uhvatili rubni slučajevi ili grananja u kôdu. Jedinični test ne može potvrditi funkcionalnost dijela programa, nego se koristi kako bi se osiguralo da građevni blokovi koje program koristi rade neovisno jedni o drugima.
- Integracijsko testiranje – bilo koja vrsta testiranja programske podrške koji želi provjeriti sučelja između komponenti protiv programskog dizajna. Služi kako bi prikazao nedostatke u sučeljima i interakciji između ugrađenih komponenti. Progresivno veće skupine ispitivanih programskih komponenti koji odgovaraju elementima dizajna arhitekture su integrirani i testirani dok god program ne radi kao sustav.
- Testiranje sustava – testiranje cjelokupno integriranog sustava radi provjere podudarnosti sa zadanim zahtjevima.

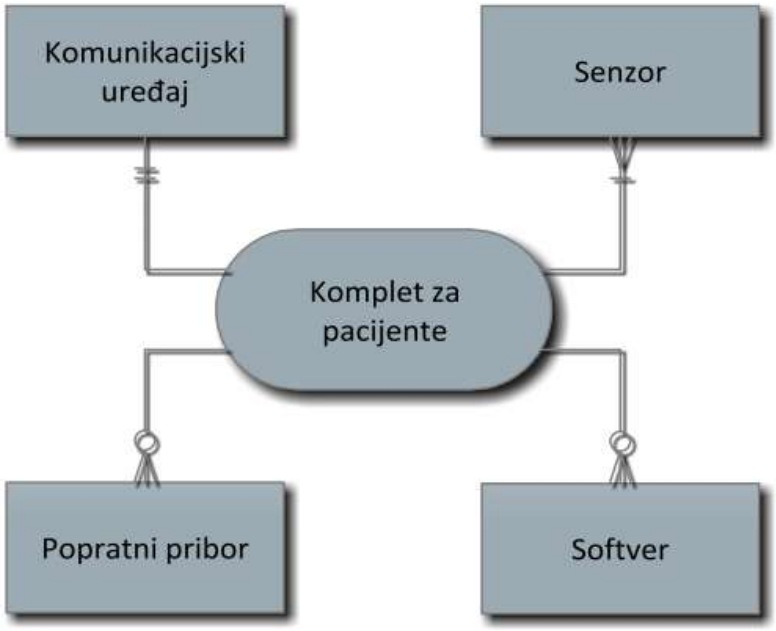
2 Usluga m-zdravstva

Usluga mobilnog zdravstva je sustav daljinskog praćenja zdravstvenog stanja pacijenta. Usluga je namijenjena za provedbu mjerenja unutar i izvan bolničkog prostora. Prvenstveno je namijenjen pacijentima s dijagnozama koje zahtijevaju dugoročno praćenje stanja s ciljem smanjenja potrebnih odlazaka na rutinske pretrage. Ciljane kategorije korisnika su odrasle osobe te djeca tjelesne težine veće ili jednake 10 kilograma. Kroz različite konfiguracije senzora, usluga omogućava pretrage poput: mjerenja krvnog tlaka, mjerenja otkucaja srca, mjerenja zasićenja kisikom, spirometrije i elektrokardiograma. Korištenjem omunikacijskog uređaja rezultati tih pretraga lako su dostupni medicinskom osoblju. Integracija usluge s postojećim informacijskim sustavima za praćenje podataka o pacijentima upotpunjuje zdravstvenu sliku pacijentovog stanja čak i kada se pacijent nalazi izvan bolnice. Područja primjene ovog sustava su [10]:

- bolesnici s kroničnim bolestima iz područja kardiologije i pulmologije
- bolesnici na kućnoj njezi
- bolesnici otpušteni iz bolnice ili nakon hitne medicinske intervencije
- preventivni pregledi
- programi za poboljšanje zdravstvenog stanja
- podrška kliničkim ispitivanjima u svim situacijama kad je potrebno prenijeti rezultate mjerenja medicinskih parametara.

Rješenje usluge m-zdravstva se sastoji se od više komponenti koje uključuju komplet za pacijenta, sustav poslužitelja i aplikacije. Komplet pacijenta (PU), kako je prikazano na slici (Sl. 2.1), sastoji se od jednog ili više medicinskih senzora, komunikacijskog uređaja, popratnog pribora (punjač, baterije i pribor senzora) i softvera.

Komunikacijski uređaj središnji je dio kompleta za pacijenta. On prikuplja mjerenja od senzora dostupnih preko Bluetooth sučelja i odašilje ih prema sustavu poslužitelja dostupnih putem mobilne mreže. Liječnik pristupa sustavu poslužitelja i pregledava podatke o pacijentu koristeći aplikaciju namijenjenu liječniku [10].



Sl. 2.1 Komponente kompleta za pacijente

Prednosti usluge za pacijente:

- sigurnost i bolja kvaliteta života
- odgovarajuće terapija i skrb putem mobilnog nadzora
- mobilnost omogućuje normalan život i uobičajene aktivnosti

Prednosti usluge za medicinsko osoblje:

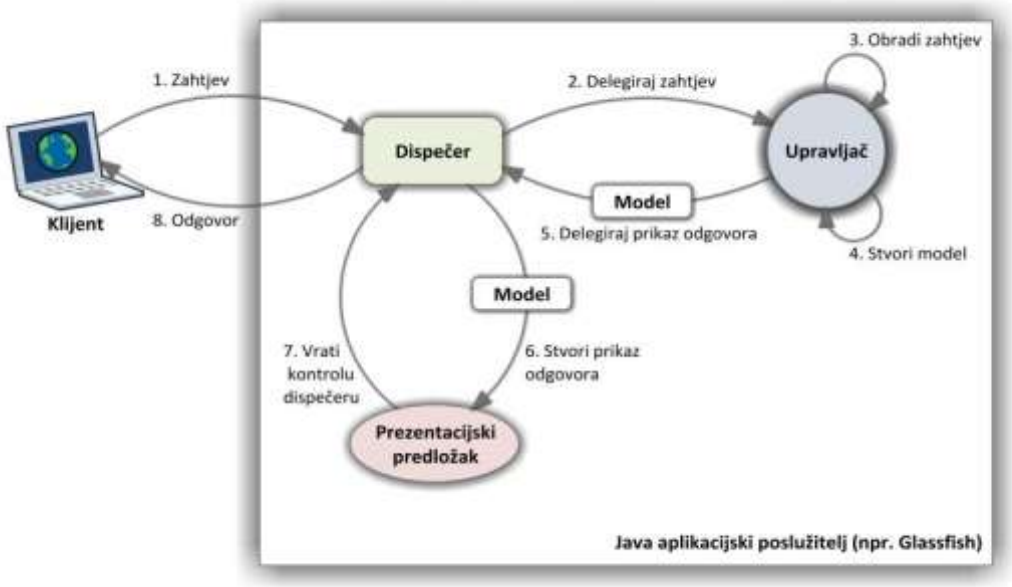
- učinkovito upravljanje resursima, nadzor i terapija
- kvalitetna zdravstvena skrb bez pritiska na medicinsko osoblje
- povećana efikasnost korištenja dijagnostike i liječenja na daljinu
- uštede u liječenju i kontrola troškova

Jedna od aplikacija koja vrši uslugu m-zdravstva je LITRA (skraćenica od *LIcence and TRAcebility tool*), alat za licenciranje i praćenje. Aplikacija služi za upravljanje medicinskom opremom koja prikuplja podatke o pacijentima. Korištenja tehnologija za razvoj te aplikacije je Spring MVC.

2.1 Spring MVC (eng. Model-View-Controller)

Spring MVC dio je Spring radnog okvira namijenjen izradi web stranica. Spring MVC koristi arhitekturu model – prikaz – upravljač (eng. *Model – View – Controller*). U toj arhitekturi se odvaja poslovna logika od logike za navigaciju i logike za prikaz. Svaki dio odrađuje jedan dio posla:

- upravljač – obrađuje zahtjeve i donosi odluku o prikazu korisniku
- model – sadrži podatke o prikazu, veza između upravljača i prikaza
- prikaz – iscrtava odgovor na zahtjev, izvlači podatke iz modela



Sl. 2.2 Prikaz obrade zahtjeva Spring MVC

Tok obrade zahtjeva u web aplikacijama koje koriste Spring MVC radni okvir prikazan je slikom (Sl. 2.2). Tok obrade je sljedeći:

1. Klijent šalje zahtjev putem web preglednika
2. Dispečer (eng. *servlet*) presreće sve zahtjeve koji zadovoljavaju određeni format URL-a te na temelju danog URL-a dispečer pronalazi upravljač kojem prosljeđuje zahtjev
3. Upravljač obrađuje zaprimljeni upit, ukoliko je potrebno u ovom koraku se dohvaćaju podaci iz baze podataka

4. Stvara se model s potrebnim podacima
5. Upravljač šalje model dispečeru zajedno s imenom predloška za prikaz
6. Dispečer na temelju imena pronalazi predložak za prikaz te šalje dobiveni model tom predlošku iz čega se generira konačan prikaz s predanim podacima iz modela
7. Konačan prikaz vraća se dispečeru
8. Dobiveni prikaz prosljeđuje se klijentu koji je podnio zahtjev

Iz navedenog toka vidljivo je da se pri svakom zahtjevu model nanovo puni podacima koji su traženi u zahtjevu te se na taj način ostvaruje dinamički prikaz web sadržaja. Dispečer je prednji upravljač koji raspoređuje zahtjeve individualnim upravljačima [3].

Spring MVC radni okvir projektiran je i dizajniran na način da je svaki komad logike i funkcionalnosti vrlo podesiv. Također ga je moguće integrirati sa drugim radnim okvirima za Web kao što su Struts, WebWork, Java Server Faces i Tapestry. To znači da je moguće zadati Springu instrukcije kako bi koristio neki od tih radnih okvira. Spring nije usko vezan uz *servlete* ili JSP (eng. *JavaServer Pages*) za prikaz stranice klijentu. Moguća je integracija s drugim tehnologijama za prikaz kao Velocity, Freemarker, Excel ili Pdf [4].

3 Testiranje sa JUnit 4

JUnit4 je radni okvir koji razvijatelju programske podrške omogućava pisanje testova i relativno brzo testiranje jediničnih testova. Jedinica je razred sa svojim metodama. Također je moguće u bilo kojem trenutku testirati skupinu testova (eng. *Test Suite*). Radi lakšeg baratanja koriste se bilješke kako bi se metode mogle razlikovati (Tablica 3.1). JUnit nudi različite metode kojima se ispituju različiti uvjeti kao što prikazuje tablica (Tablica 3.2). [5]. Kako bi se neki razred mogao testirati stvaraju se sljedeće metode:

- jedna metoda koja stvara sve potrebne objekte za izvođenje testa
- niz metoda koje sadrže dijelove kôda za testiranje
- jednu metodu koja „počisti“ nakon svakog testa

Tablica 3.1 Popis bilješki

Bilješka	Opis
`@Test`	Identificira metodu kao testni slučaj.
`@Before`	Izvršit će metodu prije svakog testa.
`@After`	Izvršit će metodu nakon svakog testa.
`@BeforeClass`	Izvršit će metodu jednom, prije svih testova.
`@AfterClass`	Izvršit će metodu jednom, nakon svih testova.
`@Ignore`	Ignorirat će metodu označenu sa @Test.
`@Test(expected = Exception.class)`	Test pada ako metoda ne baci imenovanu iznimku.

Tablica 3.2 Popis metoda

Naziv metode	Opis
`fail(String)`	Moguće koristiti za provjeru dijela kôda koji još nije testiran ili da test ne prođe prije no što se implementira kôd za test.
`assertTrue(true)` `assertTrue(false)`	Uvijek će biti true/false. Moguće koristiti kako bi se predefiniro testni rezultat ako test još nije implementiran.
`assertTrue([message], boolean condition)`	Provjerava da li je condition jednak true.
`assertsEquals([String message], expected, actual)`	Provjerava da li su dvije vrijednosti jednake. Opaska: za polja se provjeravaju reference, ne elementi u polju.
`assertNull([message], object)`	Provjerava da li je objekt jednak null.
`assertNotNull([message], object)`	Provjerava da li je objekt različit od null.
`assertSame([String], expected, actual)`	Provjerava da li obje varijable refenciraju isti objekt.
`assertNotSame([String], expected, actual)`	Provjerava da li obje varijable refenciraju različite objekte.

3.1 Primjer izvođenja testa

Razmotrimo sljedeću klasu:

```
public class Subscription {
        private int price ;

        private int length ;

        public Subscription(int p, int n) {
          price = p ;
          length = n ;
        }

        public double pricePerMonth() {
           double r = (double) price / (double) length ;
           return r ;
        }

        public void cancel() { length = 0 ; }
}
```

Kôd 3.1 Razred Subscription

Razred sadrži dva atributa:

- `int price` – sadrži cijenu
- `int length` – sadrži broj mjeseci na pretplatu

Razred sadrži dvije metode:

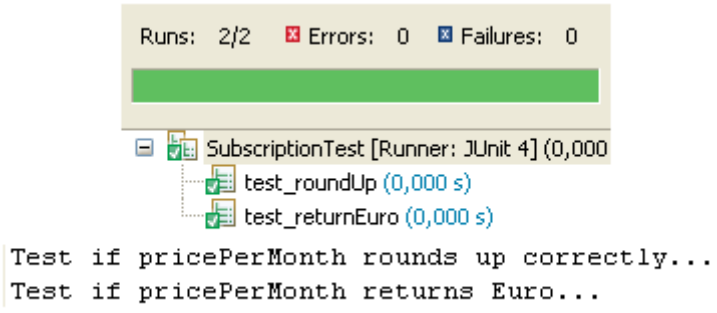
- `public double pricePerMonth()` – vraća vrijednost pretplate za jedan mjesec
- `public void cancel()` – postavlja vrijednost broja mjeseci pretplate na nulu

Testni razred za razred Subscription izgleda ovako:

```
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

import org.junit.Test;

public class SubscriptionTest {

	@Test
	public void test_returnEuro() {
	    System.out.println("Test if pricePerMonth returns
Euro...") ;
	    Subscription S = new Subscription(200,2) ;
	    assertFalse(S.pricePerMonth() == 1.0);
	    assertTrue(S.pricePerMonth() == 100.0);
	    }


	@Test
	public void test_roundUp() {
	      System.out.println("Test if pricePerMonth rounds up
	            correctly...") ;
	      Subscription S = new Subscription(200,4) ;
	      assertTrue(S.pricePerMonth() == 50) ;
	    }
}
```

Kôd 3.2 Testni razred SubscriptionTest

Razred SubscriptionTest koristi bilješku @Test koja označava metode `test_returnEuro()` i `test_roundUp()`. To znači da će se te dvije metode tretirati kao testovi te ovisno o ishodu reći da li su metode uspješno ili neuspješno testirane [6]. Testovi se ne moraju obaviti u onom redoslijedu kako su napisane. Ishod testova prikazan je na slici (Sl. 3.1).



Sl. 3.1 Ishod testova klase SubscriptionTest

4 Testiranje Spring MVC-a

4.1 Jedinični testovi

Tehnika ovisnog ubrizgavanja (eng. *Dependency Injection*) bi trebala napraviti kôd manje ovisnim o spremiku nego što je bilo sa tradicionalnim Java EE razvojem. Java objekti u aplikaciji bi trebalo biti moguće testirati s JUnit ili Test NG testovima, s objektima instancirani koristeći operator `new` nebitno o spremniku koji se koristi. Mogu se koristiti *mock* objekti za testiranje kôda u izolaciji.

Pravi jedinični testovi se pokreću prilično brzo kao da ne postoji infrastruktura koja obavlja pokretanje u stvarnom vremenu. Ističući prave jedinične testove kao dio razvojne metodologije povaćava se produktivnost. Spring radni okvir nudi sljedeće *mock* objekte i klase koje podržavaju testiranje [7]:

- JNDI – `org.springframework.mock.jndi` paket sadrži implementaciju JNDI SPI (eng. *Java Naming and Directory Interface Service Provider Interface*) koja se može koristiti za postavljanje jednostavnog JNDI okruženja za skupinu testova ili samostalne aplikacije
- ServerAPI – `org.springframework.mock.web` paket sadrži ServerAPI *mock* objekte koje služe za testiranje web sadržaja i upravljača

4.2 Integracijski test

Spring radni okvir pruža podršku za integracijski testovima. Skup razreda se nalazi u paketu `org.springframework.test`. Ovi testovi se ne oslanjaju na poslužitelja niti na neku drugu okolinu za razmještanje. Takvi tipovi testova su sporiji od testnih jedinica [7]. Ciljevi integracijskog testiranja:

- ovisno ubrizgavanje u testnom slučaju
- upravljanje transakcijama
- specifične klase koje su korisne za pisanje integracijskih testova

4.3 Bilješke (eng. Annotations)

Spring radni okvir pruža skup bilješki koje se koriste u testovima u suradnji s TestContext radnim okvirom. Postoje više vrsta tipova bilješki. Bilješke za konfiguraciju konteksta (Tablica 4.1) koristi Spring kako bi rukovao stvaranjem i ubrizgavanjem komponenata [8].

Tablica 4.1 Bilješke za konfiguraciju konteksta

Bilješka	Upotreba	Opis
@Autowired	Konstruktor, atribut, metoda	Deklarira konstruktor, atribut, *setter* metode ili konfiguracijske metode da budu povezane po tipu.
@Configurable	Tip	Koristi se za deklaraciju tipa koji treba ubrizgati, čak i ako nisu instancirane.
@Qualifier	Atribut, parametar, tip	Upravljenje ubrizgavanja po drugom smislu nego po tipu
@Required	*Setter* metode	Specificira da se određeno svojstvo mora ubrizgati

Bilješke koje se koriste pri testiranju su prikazane tablicom (Tablica 4.2). Koriste se pri jediničnim testovima u suradnji sa JUnit4 koja ovisi o Spring *beans*ima (komponente) i/ili imaju transakcijski kontekst.

Tablica 4.2 Bilješke za testiranje

Bilješka	Upotreba	Opis
@AfterTransaction	Metoda	Identificira metodu koja će biti izvršena nakon obavljene transakcije.
@BeforeTransaction	Metoda	Identificira metodu koja će se izvršiti prije obavljanja transakcije.
@ContextConfiguration	Tip	Konfigurira Spring aplikacijski kontekst.
@NotTransactional	Metoda	Metoda se ne smije izvršiti u transakcijskom kontekstu.
@Rollback	Metoda	Specifira da li metoda transakciju treba ukloniti ili ne.
@DirtiesContext	Metoda	Specificira da metoda „prlja“ Spring spremnik i mora se ponovno sagraditi nakon završetka testa.

4.4 Spring TestContext radni okvir

Spring TestContext radni okvir (`org.springframework.test.context` paket) pruža generičku podršku za izradu jediničnih testova i integracijskih testova upravljane bilješkama novisno o radnom okviru koji se koristi za testiranje (JUnit3, JUnit4 ili Test NG). Jezgra radnog okvira se sastoji od razreda `TestContext` i `TestContextManager` te od sučelju `TestExecutionListener`. `TestContextManager` upravlja `TestContext`om koji drži kontekst aktualnog testa. Također obnavlja stanje `TestContext`a kako napreduje test i usmjerava ka `TestExecutionListener`u koji upravlja trenutnim izvršavanjem testa pružajući ovisno ubrizgavanje, upravljanje transakcijama i slično [7]:

- `TestContext` – skriva kontekst u kojem je test izvršen, neovisno o radnom okviru za testiranje
- `TestContextManager` – glavna ulazna točka u TestContext radni okvir, koji upravlja jedan `TestContext` i signalizira događaje svim registriranim `TestExecutionListener`ima na dobro definiranim točkama ispitivanja: priprema testne instance, prije bilo koje @Before metode konkretnog radnog okvira za testiranje, i nakon bilo koje @After metode konkretnog radnog okvira za testiranje.
- `TestExecutionListener` – definira slušatelja (eng. *listener*) API koji reagira na događaje testa objavljene od strane `TestContextManager`a s kojima je slušatelj registriran

Spring nudi 3 implementacije `TestExecutionListener`a :

- `DependencyInjectionTestExecutionListener` – podržava ovisno ubrizgavanje u testove
- `DirtiesContextTestExecutionListener` – upravlja @DirtiesContext bilješkama
- `TransactionalTestExecutionListener` – transakcijski testovi sa postavljenom semantikom za uklanjanjem transakcije

Svaki `TestContext` pruža upravljanje i spremanje konteksta za svaku instancu za koju je odgovorna. Testovi nemaju izravan pristup konfiguriranom `ApplicationContext`u. Ako testni razred implementira sučelje `ApplicationContextAware` dodana je referenca na `ApplicationContext`. Kao alternativa za implementaciju sučelja, aplikacijski kontekst je moguće ubrizgati preko bilješke @Autowired na atribut ili *set* metodu.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class MyTest {

@Autowired
private ApplicationContext applicationContext;

// tijelo razreda...
}
```

Kako bi se dodao aplikacijski kontekst moguće je koristiti bilješku @ContextConfiguration na nivou razreda. Ako testni razred eksplicitno ne deklarira putanju odakle će čitati kontekst tada `ContextLoader` odlučuje odakle će čitati.

```
package com.example;

@RunWith(SpringJUnit4ClassRunner.class)
// Kontekst aplikacije biti će učitan s lokacije
//"classpath:/com/example/MyTest-context.xml"
@ContextConfiguration
public class MyTest {
    // tijelo razreda...
}
```

Kada konfiguriramo `DependencyInjectionTestExecutionListener`, koji je zadan prilikom @ TestExecutionListeners bilješke zavisnost testnih instanci su ubrizgani iz komponenti aplikacijskog konteksta koji je konfiguriran preko @ContextConfiguration bilješke za *set* metode, atribute ili oboje, zavisno od toga koju bilješku koristimo i uz šta ju se poveže.

Bilješka @Autowired nema utjecaja na testne klase. Ta bilješka provodi povezivanje po tipu. Ukoliko u aplikacijskom kontekstu imamo više komponenata istog tipa nemožemo se osloniti na taj pristup ubrizgavanja. Ukoliko želimo dobiti komponentu po nazivu tada koristimo bilješku @Resource [7].

U TestContext radnom okviru transakcije su upravljane preko `TransactionalTestExecutionListener`a koji je zadan preko @TestExecutionListener bilješke iako se ekcplicitno ne navede ta bilješka. Kako bi se omogućila podrška za transakcijama u aplikacijskom kontekstu se mora uvesti komponenta `PlatformTransactionManager`. Bilješka @Transaction se može navesti na nivou cijelog razreda ili pojedine metode. `TransactionalTestExecutionListener` podržava bilješke @BeforeTransaction i @AfterTransaction za slučaj ispitivanja stanja prije i poslije transakcije.

5 Testiranje aplikacije izrađene Spring MVC-om

U ovom poglavlju opisati će se način na koji se testiraju upravljači i klase koji koriste rad sa bazom podataka u aplikaciji LITRA. Primjer jednog jednostavnog upravljača prikazuje sljedeći programski isječak (Kôd 5.1).

```
import java.util.Date;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/welcome")
public class WelcomeController {

    @RequestMapping(method = RequestMethod.GET)
    public String welcome(Model model) {
        Date today = new Date();
        model.addAttribute("today", today);
        return "welcome";
    }
}
```

Kôd 5.1 Upravljač Welcome

Upravljač Welcome koristi bilješke @Controller i @RequestMapping. Prva bilješka označava da je taj razred upravljač te da će biti dodana u popis upravljača. Kada korisnik upiše URL s nastavkom /welcome dispečer će znati da taj zahtjev treba proslijediti ovom razredu. Bilješka povrh metode 'welcome' ima još dodatnu karakteristiku, a to je da se ta metoda izvršava samo kada je HTTP zahtjev tipa GET. Unutar metode se izvršava naredba koja u model dodaje trenutni datum i metoda vraća naziv JSP-a koji se naziva 'welcome' te će taj JSP izvući podatke iz modela. Primjer testa nad ovim upravljačem prikazuje sljedeći isječak koda (Kôd 5.2).

```
import diplomski.controller.WelcomeController;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import org.junit.*;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.ui.Model;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "file:build/web/WEB-
INF/applicationContext.xml")
public class WelcomeControllerJUnit{

    @Autowired
    private WelcomeController wc;

    @Autowired
    private Model model;

    @Test
    public void welcomeTest() {
        String view;
        view = wc.welcome(model);
        assertEquals(view, "welcome");
        assertNotNull(view);
        System.out.println(model.containsAttribute("today"));
    }
}
```

Kôd 5.2 Jedinični test upravljača Welcome

U svakom testu je obavezno imati bilješke @RunWith i @ContextConfiguration. S prvom bilješkom označavamo da ćemo podatke čitati iz aplikacijskog konteksta te da će biti pokrenute razredom `SpringJUnit4ClassRunner`. S drugom bilješkom određujemo lokaciju s koje treba čitati. Bilješkom @Autowired označavamo atribute koje će biti ovisno ubrizgane iz aplikacijskog konteksta. Zbog toga ih je potrebno prvo dodati u aplikacijski kontekst.

```
<bean id="welcome" class="diplomski.controller.WelcomeController"></bean>

<bean id="model" class="org.springframework.ui.ExtendedModelMap"></bean>
```

Kao što se može vidjeti, u komponenti „model“ stoji drugi razred, zbog toga što nije moguće ubrizgati sučelja, kao što je Model. Provedeni test donosi tri rezultata: prvi rezultat govori da li metoda vraća vrijednost „welcome“, drugi rezultat provjerava da li vrijednost varijable 'view' null vrijednost, dok treći rezultat provjera postoji li u modelu atribut s nazivom „today“.

Primjer integracijskog testa jednog upravljača prikazuje sljedeći primjer.

```
import diplomski.dao.StatustypesDAO;
import diplomski.model.Statustypes;
import diplomski.validator.StatustypesValidator;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.beans.propertyeditors.StringTrimmerEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.bind.support.SessionStatus;
import org.springframework.web.context.request.WebRequest;

@Controller
public class StatustypesController {

    @Autowired
    private StatustypesDAO statustypesDAO;

    @Autowired
    private StatustypesValidator validator;

    @InitBinder
    public void initBinder(WebDataBinder binder, WebRequest request)
    {
        binder.registerCustomEditor(String.class, new
        StringTrimmerEditor(false));
    }

...
```

Kôd 5.3 Dio kôda upravljača StatustypesController

Uočavamo sličnost sa prethodnim primjerom. Jedina razlika je što se koristi bilješka @InitBinder koja služi za prilagođavanje povezivanja podataka. Uloga ovog upravljača je prikaz statusa pojedinih medicinskih uređaja u sustavu. Za prikaz integracijskog testa koristit ćemo jednu metodu iz tog razreda.

```
    @RequestMapping("/statusTypes/searchStatustypes")
    public String searchStatustypes(@RequestParam(required = false,
      defaultValue = "") String title, Model model) {
        List<Statustypes> statustypes =
                  statustypesDAO.searchStatustypes(title.trim());
        model.addAttribute("SEARCH_STATUSTYPES_RESULTS_KEY",
                  statustypes);
        return "/statusTypes/showStatustypes";

    }
```

Metoda traži u bazi podataka onu opremu koja čiji naziv trenutnog stanja jednak argumentu 'title'. Kôd koji izvodi tu funkciju prikazuje sljedeći isječak koda.

```
public List searchStatustypes(String title) throws
HibernateException {

        List statustypes;

        Transaction tx = null;
        session = HibernateUtil.getSessionFactory().openSession();
        try {
            tx = session.beginTransaction();
            Criteria criteria =
                  session.createCriteria(Statustypes.class);
            criteria.add(Restrictions.ilike("title", "%" + title +
                  "%"));
            statustypes = criteria.list();
            tx.commit();
        } catch (HibernateException he) {
            if (tx != null) {
                tx.rollback();
            }
            throw he;
        } finally {
            session.close();
        }
        return statustypes;
}
```

Izvođenje testa nad ovom metodom prikazuje sljedeći programski odsječak.

```
import diplomski.controller.StatustypesController;
import static org.junit.Assert.assertEquals;
import org.junit.*;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.ui.Model;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "file:web/WEB-
INF/applicationContext.xml")
public class StatustypesControllerJUnit {

    public StatustypesControllerJUnit() {
    }

    private String result;

    @Autowired
    private StatustypesController status;

    @Autowired
    private Model model;

    @Test
    public void searchStatustypesTest(){
        result = status.searchStatustypes("", model);
        assertEquals(result, "/statusTypes/showStatustypes");
        result = status.searchStatustypes("INSPECTED", model);
        assertEquals(result, "/statusTypes/showStatustypes");
        try{
            result = status.searchStatustypes(null, model);
            assertEquals(result, "/statusTypes/showStatustypes");
        }catch(NullPointerException e){
            System.out.println("Vraćena vrijednost je null");
        }

    }

...
```

Prikaz razreda za izvođenja testa je vrlo sličan prethodnom primjeru. Jedina je razlika u aplikacijskom kontekstu, tj. u datoteci applicationContext.xml. Potrebno je dodati komponentu upravljača StatustypesController, no u samom razredu imamo dva atributa koja su vezana bilješkom @Autowired i zbog toga je potrebno i njih unijeti u aplikacijski kontekst kako bi se testovi mogli izvršiti.

```
<bean id="statustypescontroller" class="diplomski.controller.StatustypesController" ></bean>

<bean id="statustypesDAO" class="diplomski.dao.StatustypesDAO"></bean>

<bean id="statustypesValidator" class="diplomski.validator.StatustypesValidator"> </bean>
```

Rezultati testova su sljedeći: prvi test pokazuje ako kao argument naslova stanja stavimo prazan niz, dobivamo povratnu vrijednost metode 'searchStatustypes', drugi test govori ukoliko navedemo stvarni naslov stanja koji postoji u bazi također dobivamo dobar rezultat, dok treći govori ukoliko kao argument predamo null vrijednost, povratna vrijednost je također null vrijednost i taj dio testa moramo staviti u try-catch blok kako bi ishod testa bio ispravan.

Zaključak

U sklopu rada uspješno je istraženo način na koji se provode jedinični i integracijski testovi te su prema tom istraživanju i urađeni. Rješenje se sastoji od testnih programa pisanih u programskom jeziku Java. Testovi su izvršeni nad upravljačima i bazom podataka.

Za uspješno dovršenje u zadanom vremenskom roku veliku važnost ima istraživački dio rada zbog kojega sam naučio način na koji radi Spring MVC te kako se provode testovi nad njim. Tijekom izrade samoga rada došlo je do manjih tehničkih problema koji su uspješno riješeni.

Nakon izvođenja različitih testova došlo se do zaključka da aplikacija LITRA radi sa velikom točnošću. To znači da su prilikom izvođenja testova nađeni tek manji broj *bug*ova koji se mogu lako ispraviti.

Literatura

[1] *Oblikovanje programske potpore, predavanje Test*, travanj 2012.

[2] Gary Mak, Josh Long, Daniel Rubio, *Spring Recipes, A Problem-Solution Approach, 2nd Edition, 2010.*, travanj 2012.

[3] Dario Šafar, diplomski rad, *Aplikacija za upravljanje paketima opreme za uslugu m-zdravstva, veljača 2012,* travanj 2012.

[4] Viral Patel, *Spring 3 MVC – Introduction to Spring 3 MVC Framework, 21. lipanj 2010,* svibanj 2012.
http://viralpatel.net/blogs/2010/06/tutorial-spring-3-mvc-introduction-spring-mvc-framework.html

[5] Anthony Anjorin, *JUnit 4 Tutorial, 2006.*, svibanj 2012.
http://www.mm.informatik.tu-darmstadt.de/courses/helpdesk/junit4.html

[6] Wishnu Prasetya, *JUnit 4.x Quick Tutorial, 8. lipanj, 2009.*, svibanj 2012.
http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial

[7] *9. Testing, Part III. Core Technologies*, svibanj 2012.
http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/testing.html

[8] Craig Walls, *Spring Annotations*, svibanj 2012.
http://refcardz.dzone.com/refcardz/spring-annotations

[9] Wikipedia, *Software testing*, svibanj 2012.
http://en.wikipedia.org/wiki/Software_testing

[10] *Ericsson Mobile Health*, svibanj 2012.
http://www.ericsson.com/hr/ict_solutions/e-health/emh/index.shtml

Sažetak

Pojam mobilnog zdravstva odnosi se na upotrebu mobilnih komunikacijskih tehnologija u zdravstvu. Budući da se mobilno zdravstvo zasniva na komunikacijskoj i medicinskoj opremi javlja se potreba za efikasnim upravljanjem tom opremom. Na tržištu se nudi nekoliko aplikacija koje realiziraju tu uslugu. Takva jedna aplikacija je LITRA. Za njen ispravan rad potrebno je testirati njene komponente. U svrhu ispitivanja se koristi Javin radni okvir za testiranje, JUnit. Objašnjen je princip kako funkcionira JUnit te kako se ona koristi prilikom testiranja aplikacije izrađene Spring MVC-om. Prikazani su ispitni testovi tijekom jediničnog testa te integracijskog testa.

Ključne riječi: m-zdravstvo, JUnit, Spring, Spring MVC, bilješke, jedinični test, integracijski test

Summary

Mobile health is a term used for practice of medicine and public health, supported by mobile communication technologies. Mobile health is driven by communication and medical equipment therefore it needs an efficient system for managing aforementioned equipment. There are several applications on the market who realize this service. LITRA is one those applications. To see if it works well it needs to be tested. Java's framework JUnit is used for testing this application. It is explained how JUnit works and how do we use it on an application created in Spring MVC. Test cases are shown for unit testing and integration testing.

Key words: m-health, JUnit, Spring, Spring MVC, annotations, unit test, integration test

Skraćenice

XML	*Extensible Markup Language*
MVC	*Model-View-Controller*
URL	*Uniform Resource Locator*
API	*Application Programming Interface*
HTTP	*HyperText Transfer Protocol*
LITRA	*LIcence and TRAcebility*
JSP	*JavaServer Pages*