JavaScript testiranje

Pisanje programskog koda samo je jedan deo programerske profesije. Osiguravanje ispravnog funkcionisanja programa podjednako je važno kao i sam čin pisanja koda. Stoga je posao programiranja uvek praćen temeljnim testiranjem. U redovima koji slede biće izneti osnovni postulati softverskog testiranja kada je u pitanju pisanje programske logike korišćenjem JavaScript jezika.

Šta je softversko testiranje?

Softversko testiranje je vrlo širok pojam, koji podrazumeva nekoliko različitih operacija. Uprošćeno se može reći da softversko testiranje podrazumeva:

- proveru programa, kako bi se utvrdilo da program ne poseduje softverske greške,
- utvrđivanje zadovoljenja softverske specifikacije koja je nastala na osnovu klijentskih zahteva,
- proveru upotrebljivosti softverskog proizvoda iz ugla korisnika.

Ovo su samo osnovni i najznačajniji zadaci softverskog testiranja, a u zavisnosti od tipa softverskog proizvoda, testiranjem se mogu obuhvatiti i mnoge druge radnje. U ovom kursu nas prevashodno zanima testiranje web sajtova i web aplikacija i to iz ugla programskog jezika JavaScript.

Kada se govori o testiranju JavaScript funkcionalnosti, može se reći da se takvo testiranje definiše kao čin utvrđivanja podudarnosti dobijenih i očekivanih rezultata. U primerima iz prethodne lekcije je već prikazano kako jedan jednostavan JavaScript program može da proizvodi neočekivane rezultate. Testiranjem se osigurava adekvatan izlaz za bilo koju kombinaciju ulaznih parametara.

Zbog čega je testiranje bitno?

Softversko testiranje je veoma značajan proces razvoja softvera. Ukoliko se zna da je vremenski najzahtevnija, pa samim tim i najskuplja etapa u razvoju softvera održavanje, lako se može zaključiti da se temeljnim testiranjem može uštedeti dosta vremena i novca.

Softverskim testiranjem se nekada mogu sačuvati i ljudski životi. Tako je 1994. godine softverska greška na letelici Airbus A300 kineske avio-kompanije imala za posledicu gubitak 264 života.

Naravno, posledice softverskih grešaka prilikom kreiranja programske logike web sajtova ne mogu biti toliko ozbiljne, ali svakako mogu napraviti razliku između funkcionalnog i nefunkcionalnog sistema, te na kraju mogu prouzrokovati gubitak vremena, novca ili klijenata.

Tipovi softverskih testova

Softversko testiranje je pojam koji se može odnositi na veliki broj različitih radnji u cilju postizanja korektnog funkcionisanja nekog programa. Stoga postoji veliki broj različitih kriterijuma na osnovu kojih je moguće napraviti razliku između tipova softverskih testova. Osnovna podela u obzir uzima način na koji se testiranje obavlja, pa tako postoji:

- ručno testiranje i
- automatizovano testiranje.

Ručno testiranje podrazumeva testiranje tokom kojeg se ne koristi bilo koji alat za automatizaciju, odnosno program koji je specijalno namenjen testiranju. Ručnim testiranjem osoba zadužena za test može da preuzme ulogu krajnjeg korisnika, te da samostalno ispituje rad sistema. Nekada je ovakav tip testiranja i više nego prihvatljiv, posebno u situacijama u kojima se obavlja testiranje korisničkog okruženja i njegovih vizuelnih osobina. Ipak, u nekim situacijama ručno testiranje može biti suviše vremenski zahtevno, pa gotovo nemoguće, te se pribegava korišćenju alata za automatizovano testiranje.

Automatizovano testiranje podrazumeva sprovođenje testova korišćenjem nekog specijalnog alata, odnosno programa koji je napisan kako bi testirao druge programe. Ovakva vrsta testiranja nije ništa drugo do automatizovano ručno testiranje, koje osobu koja testira oslobađa od potrebe za pokretanjem svakog pojedinačnog testa, iznova i iznova. Automatizovano testiranje se obavlja korišćenjem programske logike koja je specijalno namenjena za te svrhe. Osoba koja se bavi testiranjem sama može da napiše logiku za automatizovano testiranje ili da iskoristi neki od gotovih alata namenjenih obavljanju takvog posla.

U nastavku lekcije biće ilustrovana oba upravo navedena načina za testiranje JavaScript koda.

Pored ove osnovne podele, u zavisnosti od načina na koji se testiranje obavlja, još jedna podela softverskih testova odnosi se i na način na koji se posmatra kompletan sistem koji se testira. Na osnovu takve podele postoji:

- Black Box testiranje i
- White Box testiranje.

Black Box testiranje je tehnika koja podrazumeva testiranje bez ikakvog znanja o unutrašnjoj strukturi sistema. Osoba koja se bavi ovakvim testiranjem ne mora znati kako i korišćenjem kojih tehnologija je program napisan. Štaviše, ona ne mora ni biti programer. Tako se Black Box testiranje obavlja poređenjem dobijenih i očekivanih rezultata.

Slika 19.1. Black Box pogled na sistem

Slika 19.1. ilustruje neki program u formi crne kutije. Unutrašnja realizacija programa nije poznata, a jedino što je bitno jeste ispravnost izlaza na osnovu različitih ulaznih podataka.

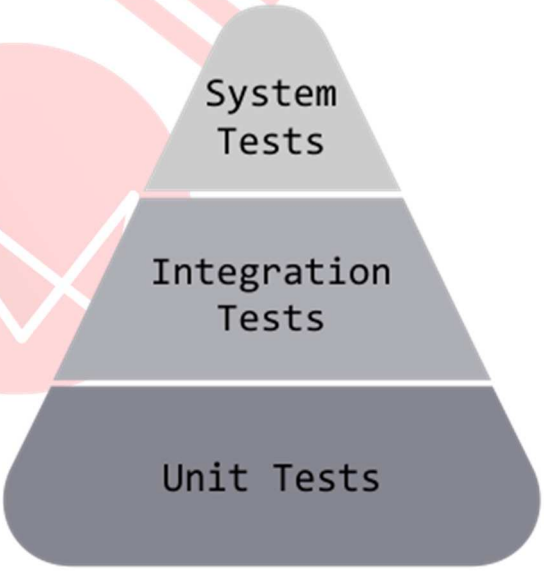
White Box testiranje, za razliku od upravo ilustrovanog pristupa, podrazumeva potpuno poznavanje unutrašnjeg funkcionisanja sistema. Osoba koja se bavi testiranjem u ovakvoj situaciji ima pristup izvornom kodu i zadužena je za to da utvrdi da li sve interne operacije sistema funkcionišu po unapred utvrđenoj specifikaciji (slika 19.2).

Slika 19.2. White Box pogled na sistem

Softverska testiranja se mogu razvrstati i na osnovu nivoa sistema koji se testira. Tako postoje sledeći testovi:

- jedinični testovi (engl. *unit tests*),
- integracioni testovi (engl. *integration tests*),
- sistemski testovi (engl. *system tests*).

Slika 19.3. ilustruje upravo navedene testove.



Slika 19.3. Testiranja na osnovu nivoa sistema koji se testira

Slika 19.3. ilustruje piramidu testiranja, koja definiše podelu na tri (3) osnovne grupe testiranja, u zavisnosti od nivoa sistema koji je obuhvaćen.

Na dnu piramide nalaze se jedinični testovi, odnosno **Unit Tests**. Kao što i samo ime kaže, ovakvi testovi su usmereni na pojedinačne funkcionalnosti, od kojih je sačinjen kompletan program. Unit testovima testira se najmanja moguća jedinica koda. Najčešće su to funkcije, odnosno metode objekata, pa se tako Unit testovima utvrđuje da li pojedinačni elementi od kojih je sačinjen kompletan sistem obavljaju svoj posao bez greške. Ovo je i najznačajnija vrsta testova iz ugla ove lekcije. Naime, najkorisnija vrsta testova za nas u ovom trenutku su upravo Unit testovi, te će stoga u nastavku ove lekcije posebna pažnja biti posvećena praktičnoj realizaciji testova ovog tipa.

Središnji nivo piramide testiranja zauzimaju integracioni testovi – **Integration Tests**. Oni su namenjeni testiranju interakcije između različitih manjih komponenata jednog softverskog sistema. Naime, ozbiljni softverski sistemi uglavnom se sastoje iz većeg broja manjih pojedinačnih komponenata, čijom saradnjom se gradi funkcionalni program. Tako se, nakon Unit testova svih pojedinačnih komponenata, testira njihova međusobna interakcija korišćenjem integracionih testova.

Na vrhu piramide nalaze se sistemski testovi – **System Tests**. Sistemski testovi usmereni su na testiranje kompletnog sistema kao celine. Naime, u idealnim uslovima svi unutrašnji bugovi detektuju se jediničnim i integralnim testovima, tako da na kraju preostaje da se verifikuje ponašanje kompletnog sistema i to uglavnom iz ugla samog korisnika. Stoga su sistemski testovi najčešće tipa Black Box.

Pored ove tri osnovne podele softverskih testiranja, postoji još dosta klasifikacija, te različitih softverskih testova. U nastavku će biti navedeni neki od najznačajnijih:

- **test prihvatljivosti** – test koji je usmeren na proveru usklađenosti sistema sa zahtevima naručioca, ukoliko sistem ispunjava zahteve naručioca, sistem je prihvatljiv za isporuku;
- **instalacioni test** – test koji se sprovodi nakon instalacije programa na produkcioni kompjuter, u slučaju web aplikacija to bi podrazumevalo testiranje nakon postavljanja aplikacije na web server;
- **alfa testiranje** – testiranje koje sprovodi mala grupa korisnika pre nego što se program objavi ili isporuči krajnjim korisnicima;
- **beta testiranje** – testiranje koje sprovodi mala grupa finalnih korisnika u realnim uslovima korišćenja;
- **regresiono testiranje** – testiranje koje se tipično obavlja nakon neke izmene na programu, kako bi se osiguralo da i nakon izmene sistem funkcioniše bez greške;
- **testiranje performansi** – testovi koji su usmereni ka ispunjenju traženih zahteva kada su u pitanju performanse sistema – vreme odziva, maksimalno vreme čekanja na dobijanje odgovora ili slično;
- **testiranje korisničkog interfejsa** – testiranje koje je usmereno na grafičko okruženje koje korisnici direktno koriste prilikom rada sa programom, testira se razumljivost, upotrebljivost, jednostavnost, funkcionisanje komandi...

Pitanje

Kako se nazivaju testovi najnižeg nivoa, koji su usmereni na testiranje pojedinačnih funkcionalnosti od kojih je sačinjen kompletan program?

- **Unit testovi**
- Beta testovi
- Sistemski testovi
- Regresioni testovi

Objašnjenje:

Ukoliko se testovi posmatraju u odnosu na nivo sistema koji se testira, na dnu lestvice se nalaze unit testovi, odnosno testovi pojedinačnih jedinica funkcionalnosti. Reč je o testovima najnižeg nivoa, koji su usmereni na najmanje jedinice koda koje se mogu samostalno testirati. Uglavnom su to funkcije ili kompletni objekti.

Referentni primer

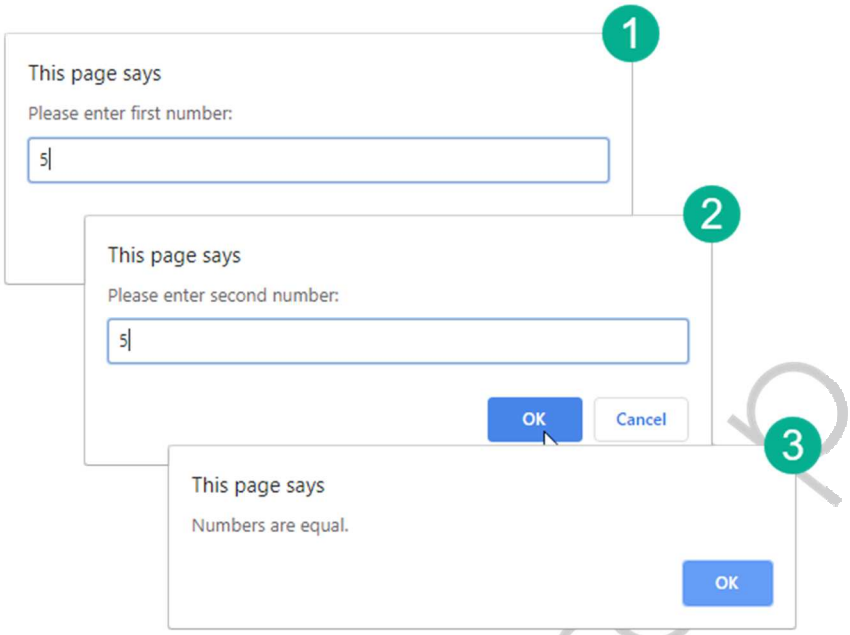
Primer koji će u ovoj lekciji poslužiti za demonstraciju testiranja biće isti onaj primer koji je korišćen i u prethodnoj lekciji, kada je bilo reči o procesu pronalaska grešaka, odnosno o debugovanju. Podsetimo se koda:

```
	var number_a = prompt("Please enter first number:");
	var number_b = prompt("Please enter second number:");

	number_a = parseInt(number_a);
	number_b = parseInt(number_b);

	if (number_a > number_b) {
		alert("Number " + number_a + " is greater then " +
number_b);
	else if (number_b > number_a) {
		alert("Number " + number_b + " is greater then " +
number_a);
	} else {
		alert("Numbers are equal.");
	}
```

Prikazani kôd omogućava da se od korisnika preuzmu dva broja i da se zatim utvrdi njihova jednakost (ili nejednakost).

Ručno testiranje referentnog primera

U prethodnim redovima je kao osnovni tip softverskog testiranja spomenuto ručno testiranje. Stoga će tako započeti i testiranje našeg referentnog primera. Osnovni princip ručnog testiranja podrazumevao bi korišćenje mehanizama za kontrolu programa koji su na raspolaganju i samim korisnicima. U našem primeru to su modalni prozori za unos vrednosti (slika 19.4).



Slika 19.4. Ručno, Black Box testiranje referentnog primera

Kao što i naziv slike 19.4. kaže, ovakva vrsta ručnog testiranja mogla bi se podvesti pod Black Box kategoriju testiranja, s obzirom na to da ne zahteva poznavanje unutrašnje logike sistema. Ipak, ovakav pristup prilično je bazičan, pa iako na ovako jednostavnom primeru može da izgleda zadovoljavajuće, u većini slučajeva nije prihvatljiv. Stoga će u nastavku biti prikazan nešto napredniji način ručnog testiranja referentnog primera. Kako bi takav pristup mogao biti realizovan, neophodno je izvršiti refaktoring referentnog primera.

Pisanje koda koji se lakše testira

Nije svaki kôd lak za testiranje, pogotovu ukoliko je reč o softverskim ili automatizovanim testovima. Drugim rečima, prilikom pisanja programskog koda potrebno je pridržavati se nekih pravila, kako bi kôd bio lakši za testiranje:

- organizacija koda, razdvajanjem prezentacije od unutrašnje logike, tako jedna celina treba da bude kôd prezentacije, druga celina kôd za stilizovanje, a treća programski kôd unutrašnje logike,
- jedna funkcionalnost treba da bude izolovana i smeštena unutar jedne funkcije,
- jedna funkcija treba da ima samo jednu odgovornost, odnosno nije dobro da jedna funkcija obavlja više različitih operacija,
- uvek je bolje imati više manjih, pojedinačnih funkcija, nego jednu veću unutar koje bi bilo objedinjeno više odgovornosti.

Ovo su samo neke od smernica kojih je dobro pridržavati se prilikom razvoja softvera, a koje su u ovom trenutku za nas interesantne. Ukoliko analiziramo ono što je navedeno, lako se može zaključiti da kôd našeg referentnog primera ne zadovoljava većinu navedenih smernica.

Za početak, primer ne poseduje nijednu funkciju. Stoga je dobro osnovnu logiku skripte smestiti unutar zasebne funkcije:

```
      function compare(a, b) {

            let number_a = parseInt(a);
            let number_b = parseInt(b);

            if (number_a > number_b) {

                  alert("Number " + number_a + " is greater then " +
number_b);

            }else if (number_b > number_a) {
                  alert("Number " + number_b + " is greater then " +
number_a);

            }else{

                  alert("Numbers are equal.");
            }

      }
```

Sada je osnovna logika naše skripte smeštena unutar zasebne funkcije. Ovakav pristup je svakako bolji, ali ne i idealan. U prethodnim redovima je rečeno da svaka funkcija treba da ima samo jednu odgovornost, odnosno da treba da obavlja jedan zadatak. Upravo kreirana funkcija `compare()`, pored poređenja, obavlja i emitovanje poruka za krajnjeg korisnika. Stoga bi mnogo bolje bilo nju transformisati na sledeći način:

```
      function compare(a, b) {

            let number_a = parseInt(a);
            let number_b = parseInt(b);

            if (number_a > number_b) {
                  return 1;

            } else if (number_b > number_a) {
                  return -1;

            }else {
                  return 0;
            }

      }
```

Sada je funkcija oslobođena logike za prikaz poruka korisniku. Umesto poruka, ona sada emituje jednu od tri vrednosti: -1, 1 i 0. Kada je prvi prosleđeni broj veći od drugog, funkcija emituje vrednost 1. Kada je drugi broj veći od prvog, funkcija vraća vrednost -1. Na kraju, kada su prosleđeni brojevi jednaki, funkcija emituje vrednost 0. Ovakva funkcija je mnogo lakša za testiranje, što će uskoro biti i demonstrirano.

Na kraju, kompletan transformisani primer sada izgleda ovako:

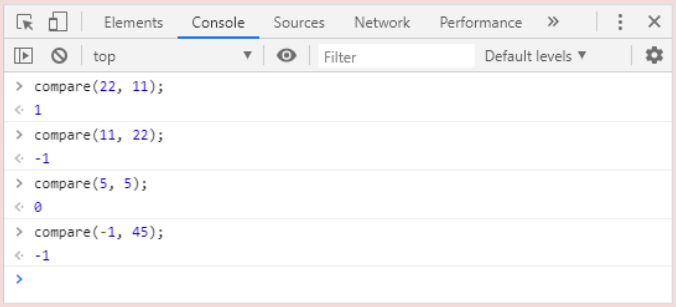
```
      var number_a = prompt("Please enter first number:");
      var number_b = prompt("Please enter second number:");

      let result = compare(number_a, number_b);

      switch (result){
            case 1:

                  alert("Number " + number_a + " is greater then " +
number_b);
                  break;
            case -1:
                  alert("Number " + number_b + " is greater then " +
number_a);
                  break;
            case 0:

                  alert("Numbers are equal.");
                  break;
      }

      function compare(a, b) {

            let number_a = parseInt(a);
            let number_b = parseInt(b);

      if (number_a > number_b) {
            return 1;
      } else if (number_b > number_a) {
            return -1;
      } else {
            return 0;
      }
}
```

Osnovna funkcionalnost upravo refaktorovanog primera sada bi mogla biti testirana pozivanjem funkcije `compare()` sa različitim ulaznim parametrima i upoređivanjem dobijenih i očekivanih rezultata:

```
      compare(22, 11);  //1
      compare(11, 22);  //-1
      compare(5, 5);  //0
      compare(-1, 45);  //-1
```

Zanimljivost

Ukoliko niste znali, JavaScript kôd je moguće pisati i unutar konzole. Takva osobina konzole može biti veoma korisna upravo u situacijama kada je potrebno obaviti testiranje. Konzola web pregledača omogućava i interakciju sa JavaScript kodom sa stranice, pa je moguće uraditi i nešto ovakvo (slika 19.5).



Slika 19.5. Testiranje funkcije compare() direktno unutar konzole web pregledača

Automatizovano testiranje

U prethodnim redovima ilustrovan je osnovni pristup za testiranje, koji je podrazumevao ručno pozivanje jedne iste funkcije više puta i poređenje dobijenih i očekivanih rezultata.

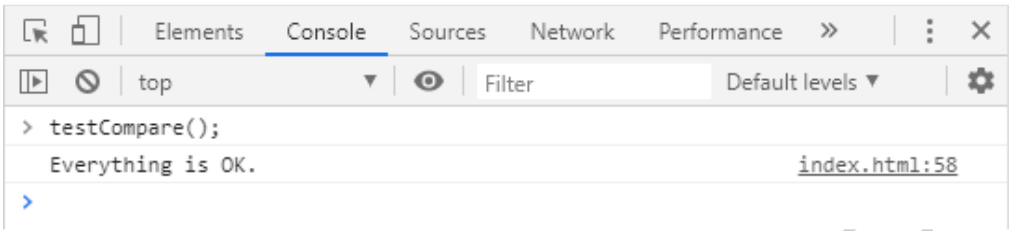
Pristup je, kao što ste mogli da vidite, veoma spor i zahteva ponavljanje jedne iste logike svaki put kada želimo da ponovimo testiranje. Jednostavno, veoma često se može javiti potreba za ponovnim testiranjem jedne iste jedinice koda, naročito nakon detekcije i uklanjanja grešaka. U takvim situacijama bilo bi neophodno uvek iznova pozivati funkciju `compare()` nekoliko puta i to na identičan način. Pored ponavljanja poziva funkcije, svaki put bi se ručno i upoređivali dobijeni i očekivani rezultati. Jasno je da bi najbolje bilo proces testiranja što više automatizovati:

```
function testCompare() {

      let fail = false;

      if (compare(22, 11) !== 1) { fail = true; }
      if (compare(11, 22) !== -1) { fail = true; }
      if (compare(5, 5) !== 0) { fail = true; }
      if (compare(-1, 45) !== -1) { fail = true; }


      if (fail) {
            console.log('Tests failed!');
      } else {
            console.log('Everything is OK.');
      }

}
```

Sada je kompletna logika za testiranje funkcije `compare()` smeštena unutar jedne zasebne funkcije `testCompare()`. Unutar funkcije nalaze se svi oni slučajevi korišćenja koji su nešto ranije već definisani. Ipak, ovoga puta unutar testova su definisani i očekivani rezultati.

Na taj način je postignuta potpuna autonomnost funkcije za testiranje. Drugim rečima, sve što preostaje jeste da se ovakva funkcija pozove, a ona će unutar konzole ispisati da li funkcija `compare()` svoj posao obavlja ispravno ili da ne radi dobro.

Osnovna prednost ovakvog pristupa jeste mogućnost pokretanja testova proizvoljan broj puta, kad god se za takvim nečim javi potreba. Na primer, kada se napravi neka izmena unutrašnje logike `compare()` funkcije, dovoljno je pozvati funkciju `testCompare()` i utvrditi da li funkcija `compare()` i dalje svoj posao obavlja ispravno (slika 19.6).



Slika 19.6. Testiranje funkcije compare()

Funkcije za testiranje najčešće se smeštaju unutar jednog zasebnog fajla, odvojene od programskog koda koji testiraju.

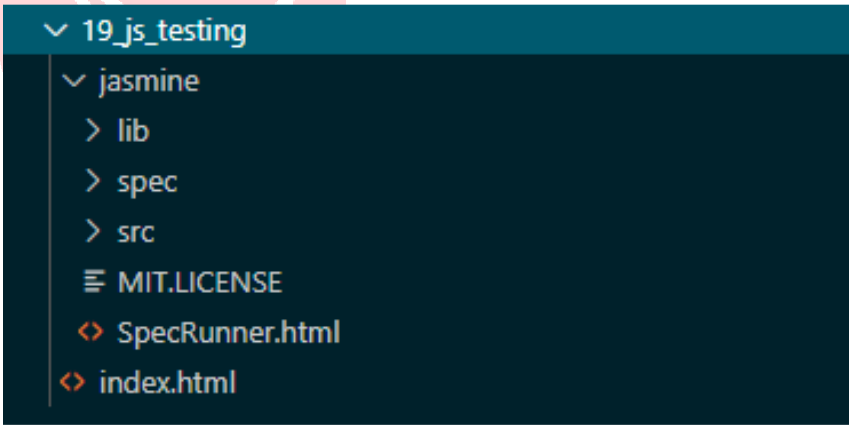
Korišćenje Jasmine biblioteke za pisanje jednostavnog Unit testa

Za kraj, biće demonstrirano testiranje referentnog primera korišćenjem jedne JavaScript biblioteke koja je specijalno namenjena za obavljanje takvog posla. Naime, postoji veliki broj različitih biblioteka namenjenih testiranju JavaScript koda. Jedna od najpopularnijih jeste biblioteka *Jasmine*.

Prvi korak jeste preuzimanje poslednje verzije ove biblioteke sa sledeće adrese:

https://github.com/jasmine/jasmine/releases

Potrebno je preuzeti samostalnu verziju *Jasmine* biblioteke (fajl čiji naziv započinje sa *jasmine-standalone*). Nakon preuzimanja, arhivu je potrebno raspakovati unutar projekta koji će biti testiran. Najbolje je napraviti poseban folder, sa nazivom `jasmine`, a zatim unutar njega smestiti sve fajlove i foldere iz preuzete arhive (slika 19.7).



Slika 19.7. Struktura projekta nakon dodavanja Jasmine biblioteke

Preuzimanjem Jasmine biblioteke dobijaju se sledeći folderi i fajlovi:

- `/lib` – folder unutar koga se nalaze fajlovi Jasmine biblioteke,
- `/spec` – folder unutar koga se smeštaju testovi,
- `/src` – folder unutar koga se smešta JavaScript kôd koji je potrebno testirati,
- `SpecRunner.html` – fajl kojim se pokreće testiranje.

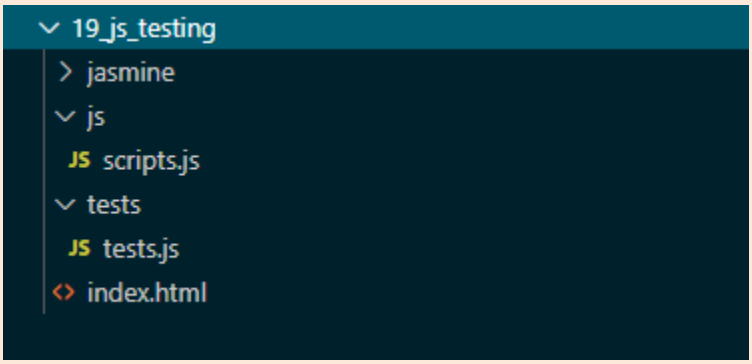
Unutar foldera `spec` i `src` automatski se nalaze fajlovi sa primerima testiranja. Unutar foldera `src` smešten je pokazni izvorni kôd za testiranje, dok se unutar foldera `spec` nalaze testovi. Ovakvi predefinisani fajlovi, koji dolaze sa *Jasmine* bibliotekom, mogu se koristiti za upoznavanje njenog osnovnog načina funkcionisanja.

Ipak, nije neophodno koristiti foldere `src` i `spec`, već je moguće referencirati kôd koji se nalazi bilo gde. Referenciranje se obavlja unutar fajla `SpecRunner.html`.

Primer

Dodatni refaktoring projekta

Pre nego što napišemo prvi Unit test korišćenjem Jasmine biblioteke, obavićemo dodatni refaktoring koda referentnog primera. JavaScript funkcionalnosti biće prebačene u zaseban fajl, u posebnom folderu. Takođe, biće kreiran još jedan dodatni folder sa nazivom `test` i u njemu će biti smešten jedan prazan `.js` fajl. Tako će kompletna struktura referentnog primera izgledati kao na slici 19.8.



Slika 19.8. Struktura projekta nakon refaktoringa

Unutar `scripts.js` fajla prebačena je `compare()` funkcija:

```
      function compare(a, b) {
            let number_a = parseInt(a);
            let number_b = parseInt(b);
            if (number_a > number_b) {
                  return 1;
            } else if (number_b > number_a) {
                  return -1;
            } else {
                  return 0;
            }
}
```

Zatim je `scripts.js` fajl referenciran unutar `index.html` fajla:

```
      <script src="js/scripts.js"></script>
      <script>
            var number_a = prompt("Please enter first number:");
            var number_b = prompt("Please enter second number:");
            let result = compare(number_a, number_b);
            switch (result) {
                  case 1:
                        alert("Number " + number_a + " is greater then " +
number_b);
                        break;
                  case -1:
                        alert("Number " + number_b + " is greater then " +
number_a);
                        break;
                  case 0:
                        alert("Numbers are equal.");
                        break;
            }
      </script>
```

Naravno, iz `index.html` fajla uklonjena je `compare()` funkcija, s obzirom na to da je ona prebačena u zasebni fajl. Na ovaj način referentni primer je prilagođen testiranju Jasmine biblioteke.

Nakon refaktoringa može se obaviti konfigurisanje *Jasmine* test okruženja. Unutar fajla `SpecRunner.html` biće referencirani upravo kreirani JavaScript fajlovi `scripts.js` i `tests.js`:

```
<!DOCTYPE html>
<html>

      <head>
            <meta charset="utf-8">
            <title>Jasmine Spec Runner v3.5.0</title>

            <link rel="shortcut icon" type="image/png" href="lib/jasmine-
            3.5.0/jasmine_favicon.png">
            <link rel="stylesheet" href="lib/jasmine-3.5.0/jasmine.css">

            <script src="lib/jasmine-3.5.0/jasmine.js"></script>
            <script src="lib/jasmine-3.5.0/jasmine-html.js"></script>
            <script src="lib/jasmine-3.5.0/boot.js"></script>

            <!-- include source files here... -->
            <script src="../js/scripts.js"></script>

            <!-- include spec files here... -->
            <script src="../tests/tests.js"></script>

      </head>

      <body>
      </body>
</html>
```

Bitno je primetiti dve izmene unutar fajla `SpecRunner.html`. Prva izmena je napravljena nakon komentara *include source files here*. Tu je navedena putanja do fajla sa JavaScript izvornim kodom koji je potrebno testirati. Druga izmena je napravljena nakon komentara *include spec files here*, gde je referenciran JavaScript fajl unutar koga ćemo napisati Jasmine Unit test:

```
describe("compare() function", function () {

      //Spec for comparing two numbers, when second number is greater
      it("should be able to compare two numbers, when second number is
greater", function () {
            expect(compare(5, 8)).toEqual(-1);
      });

});
```

Upravo je prikazan sadržaj `tests.js` fajla koji je kreiran nešto ranije. Kôd predstavlja *Jasmine* logiku za testiranje. *Jasmine* testovi se sastoje iz dva dela:

- suite i
- spec.

Suite predstavlja grupu testova. Na primer, prilikom testiranja jedne JavaScript funkcije najčešće se definiše više pojedinačnih testova. Svi takvi testovi se objedinjuju unutar jednog suitea. Jasmine suite se kreira pozivom funkcije `describe()`, koja prihvata dva parametra. Prvi parametar jeste naziv grupe testova. Drugi parametar jeste funkcija unutar koje se objedinjuje više testova.

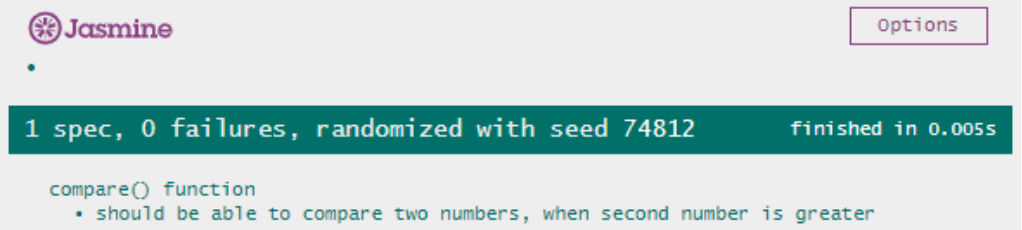
Jasmine suite iz primera poseduje samo jedan spec, odnosno jedan test. Jasmine spec započinje pozivom funkcije `it()`. Ova funkcija prihvata dva parametra. Prvi parametar jeste naziv pojedinačnog testa, dok je drugi parametar funkcija koja predstavlja jedan test.

Svaki pojedinačni Jasmine test može imati jednu ili više provera, odnosno očekivanja. Svaka provera se formira u obliku tvrdnje koja može biti tačna (engl. *true*) ili netačna (engl. *false*).

Takva tvrdnja se u primeru formira korišćenjem metode `expect()`. Kao parametar se navodi konkretna funkcija koja se testira. Dobijena vrednost se proverava korišćenjem još jedne *Jasmine* funkcije – `toEqual()`. Njom se proverava jednakost. Tako je u primeru, jednostavnim rečima, definisano: proveri funkciju `compare()` sa ulaznim parametrima 5 i 8.

Rezultat treba da bude `-1`. Ukoliko se od funkcije `compare()` dobije vrednost `-1`, test je uspešan.

Pokretanje definisanih Jasmine testova obavlja se otvaranjem stranice `SpecRunner.html` (slika 19.9).



Slika 19.9. Rezultat Jasmine testiranja

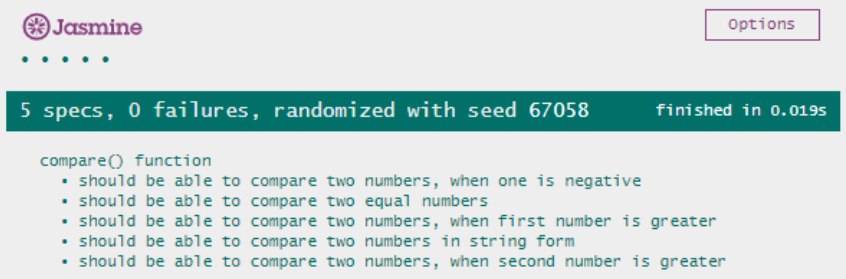
Sa slike 19.9. se može videti da testiranje prolazi uspešno. Ipak, u prikazanom primeru je definisan samo jedan test (*spec*), unutar grupe testova (*suite*). Za adekvatno testiranje funkcije `compare()` potrebno je definisati još testova:

```
describe("compare() function", function () {

      //Spec for comparing two numbers, when second number is greater
      it("should be able to compare two numbers, when second number is
greater", function () {
            expect(compare(5, 8)).toEqual(-1);
      });

      //Spec for comparing two numbers, when first is greater
      it("should be able to compare two numbers, when first number is
greater", function () {
            expect(compare(8, 5)).toEqual(1);
      });

      //Spec for comparing two numbers in string form
      it("should be able to compare two numbers in string form",
      function () {
            expect(compare('8', '5')).toEqual(1);
      });

      //Spec for comparing two numbers when one is negative
      it("should be able to compare two numbers, when one is negative",
function () {
            expect(compare(-8, 5)).toEqual(-1);
      });

      //Spec for comparing two equal numbers
      it("should be able to compare two equal numbers", function () {
            expect(compare(5, 5)).toEqual(0);
      });

});
```

Sada je definisan veći broj testova, zaduženih za proveru funkcije `compare()`. Proveravaju se sledeće situacije, po redu kojim su definisane:

- kada se funkciji proslede dva broja, pri čemu je prvi veći,
- kada se funkciji proslede dva broja, pri čemu je drugi veći,
- kada se funkciji proslede dva broja u `string` obliku,
- kada se funkciji proslede dva broja, pri čemu je jedan od njih negativan,
- kada se funkciji proslede dva identična broja.

Otvaranjem stranice `SpecRunner.html` dobijaju se rezultati testiranja (slika 19.10).



Slika 19.10. Rezultat Jasmine testiranja (2)

Savet za odabir ulaznih parametara i definisanje poslednjeg testa

U ovoj lekciji ilustrovano je testiranje jedne jednostavne JavaScript funkcije, sa dva ulazna parametra i jednim izlaznim parametrom. Prilikom testiranja funkcija sa ulaznim parametrima posebnu pažnju je potrebno obratiti na odabir ulaznih vrednosti. Za ulazne parametre je najbolje birati granične ili netipične vrednosti, jer će se greške najpre ispoljiti u takvim situacijama. U prethodnim redovima propušteno je testiranje jednog takvog scenarija – kada se funkciji proslede vrednosti koje se ne mogu konvertovati u brojeve.

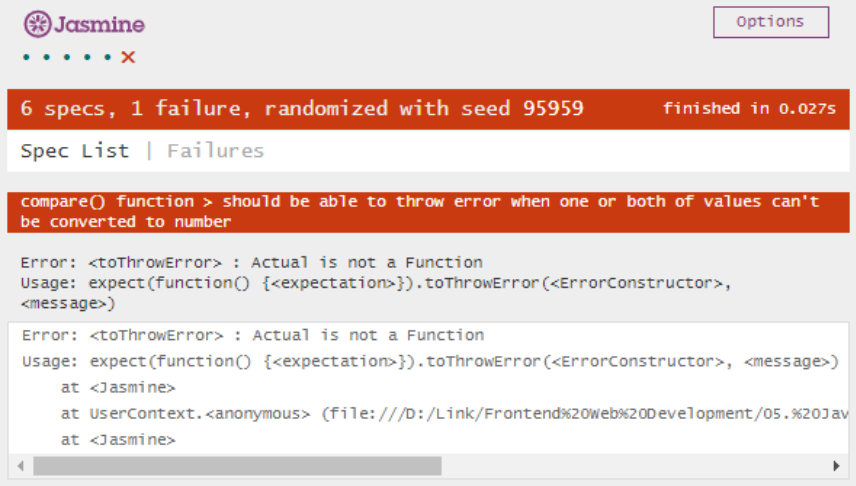
```
compare('fdagdgds', 'dssdgdgsd');  //0
```

Funkciji `compare()` sada su prosleđene dve `string` vrednosti koje se ne mogu konvertovati u podatke tipa `number`. Iz prikazane linije se može videti da funkcija `compare()` kao svoju povratnu vrednost emituje `0`, što znači da govori da su prosleđene vrednosti jednake. Očigledno da je reč o bugu. Ispravno ponašanje funkcije u ovakvim situacijama podrazumevalo bi emitovanje prikladnog izuzetka, s obzirom na to da su prosleđeni parametri neodgovarajućeg tipa. JavaScript poseduje i jedan ugrađeni tip izuzetka koji je prikladan za ovakve situacije – `TypeError`. Stoga je potrebno dodati još jedan test unutar grupe *Jasmine* testova:

```
it("should be able to throw error when one or both of values can't be
converted to number ", function () {
expect(function() {
    compare('sgrberh', 5)
}).toThrowError(TypeError);
});
```

Za formulisanje novog testa, iskorišćena je funkcija `toThrowError()`, kojom se proverava da li će testirana funkcija izbaciti izuzetak tipa `TypeError`.

Ponovnim pokretanjem testiranja dobija se rezultat kao na slici 19.11.



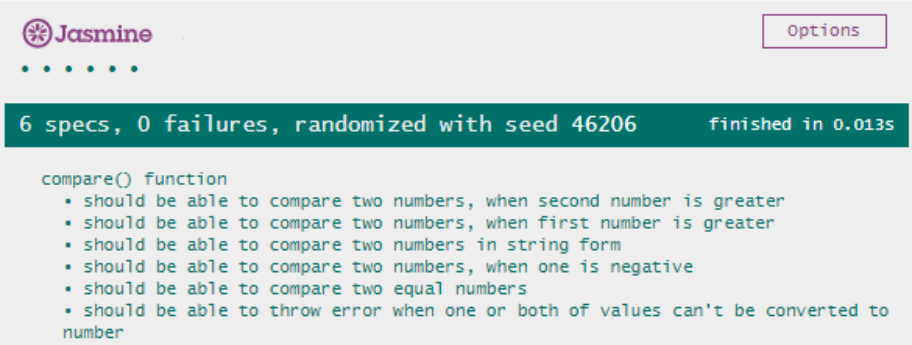
Slika 19.11. Rezultat Jasmine testiranja (3)

Sada se može videti da se tokom testiranja dobija jedna greška. Testiranje se pokazalo kao uspešno, jer je pomoglo u pronalasku greške. Korigovanje logike može se obaviti na sledeći način:

```
function compare(a, b) {

	let number_a = parseInt(a);
	let number_b = parseInt(b);

	if(isNaN(number_a) || isNaN(number_b))
		throw new TypeError("One of the parameters cannot be converted to
number.", "scripts.js");

	if (number_a > number_b) {
		return 1;
	} else if (number_b > number_a) {
		return -1;
	} else {
		return 0;
	}
}
```

Sada je, nakon parsiranja tekstualnih podataka, dodata provera toga da li neka od ulaznih vrednosti ima vrednost `NaN`. To bi značilo da neki od parametara nije mogao biti konvertovan u tip `number`. U takvoj situaciji se emituje izuzetak tipa `TypeError`.

Ponovnim pokretanjem testiranja možemo se uveriti da funkcija `compare()` sada obavlja svoj posao bez greške (slika 19.12).



Slika 19.12. Rezultat Jasmine testiranja (4)

Napomena

Cilj prethodnih redova u kojima je bilo reči o *Jasmine* biblioteci za testiranje nije bio detaljno upoznavanje takve biblioteke, već samo osnovni osvrt na specijalizovane sisteme za automatizovano testiranje. Više informacija o Jasmine biblioteci se može pronaći u zvaničnoj dokumentaciji:

https://jasmine.github.io/pages/docs_home.html

Rezime

- Softversko testiranje podrazumeva proveru programa, kako bi se utvrdilo da program ne poseduje softverske greške.
- Postoji veliki broj različitih tipova softverskih testova i kriterijuma na osnovu kojih je moguće napraviti razliku između njih.
- Ručno testiranje podrazumeva testiranje tokom kojeg se ne koristi bilo koji alat za automatizaciju, odnosno program koji je specijalno namenjen testiranju.
- Automatizovano testiranje podrazumeva sprovođenje testova korišćenjem nekog specijalnog alata, odnosno programa koji je napisan kako bi testirao druge programe.
- Black Box testiranje je tehnika koja podrazumeva testiranje bez ikakvog znanja o unutrašnjoj strukturi sistema.
- White Box testiranje podrazumeva potpuno poznavanje unutrašnjeg funkcionisanja sistema.
- Unit testovi su usmereni na pojedinačne funkcionalnosti od kojih je sačinjen kompletan program.
- Integracioni testovi su namenjeni testiranju interakcije između različitih manjih komponenata jednog softverskog sistema.
- Sistemski testovi usmereni su na testiranje kompletnog sistema kao celine.
- Prilikom pisanja programskog koda potrebno je pridržavati se nekih pravila, kako bi kôd bio lakši za testiranje.
- Postoji veliki broj različitih biblioteka namenjenih testiranju JavaScript koda; jedna od najpopularnijih jeste biblioteka Jasmine.