Primjena više programskih paradigmi u razvoju programske podrške

Ziterbart, Bruno

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

PRIMJENA VIŠE PROGRAMSKIH PARADIGMI U RAZVOJU PROGRAMSKE PODRŠKE

Bruno Ziterbart

Split, svibanj 2017.

Sadržaj

UVOD

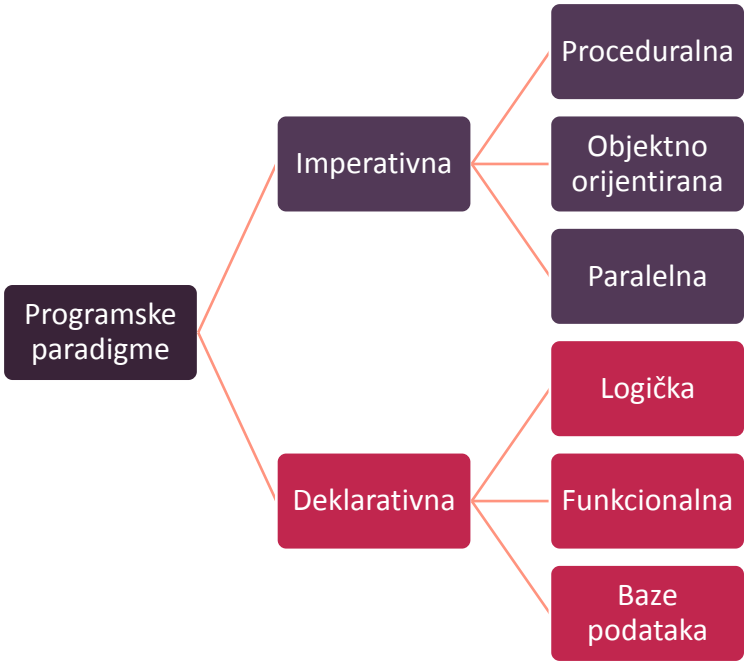
Razvoj programske podrške je svakodnevan posao bilo kojeg programera. Kako vrijeme prolazi tako se mijenjaju i tehnologije pomoću kojih se razvija programska podrška. Programerima je neophodno bit u koraku s tehnologijama kako bi ostali kompetentni na tržištu. Ipak, nove tehnologije nisu posve drugačije od njihovih prethodnika. Dobar primjer je promjena programskih jezika. Novi programski jezici imaju novu sintaksu koja je više ili manje slična sintaksi programskog jezika kojim se programer prethodno služio. S druge strane, logika razvoja programske podrške se puno sporije mijenja, dok stil pisanja programskog koda ostaje gotovo nepromijenjen. Zato nema potrebe za kontinuiranim savladavanjem svih mogućih programskih jezika koji izlaze na dnevnoj bazi. Puno je jednostavnije i korisnije savladati nekoliko programskih paradigmi koje su učestale na tržištu. Poznavanje programskih paradigmi ima veliki utjecaj u razumijevanju i savladavanju novih programskih jezika. U svakodnevnom životu postoji tvrdnja prema kojoj čovjek vrijedi onoliko koliko jezika poznaje. Ukoliko se povuče paralela s programerima, može se reći da programer vrijedi onoliko koliko poznaje programskih jezika. Uzme li se u obzir ono prethodno navedeno, točnije je tvrditi da programer vrijedi onoliko koliko poznaje programskih paradigmi. U nastavku će se definirati pojam programskih paradigmi te će se neke od njih detaljno opisati. Kako bi se jednostavnije objasnili određeni pojmovi koristit će se primjeri programskog koda napisani u programskom jeziku JavaScript. Riječ je o programskom jeziku koji je u nekoliko desetaka godina potvrdio svoju kvalitetu i prema dostupnim informacijama izgleda da neće skoro nestati s tržišta. Rezultat je to velike zajednice programera koji ga preferiraju te svakodnevnim izlaskom novih biblioteka koje pojednostavljuju i ubrzavaju razvoj programske podrške. Dodatno, JavaScript je programski jezik kojega i sam preferiram te služi kao dobar alat za demonstraciju korištenja različitih programskih paradigmi unutar jednog programskog jezika. Iz navedenog se može zaključiti da se u JavaScriptu može pisati programski kod pomoću više programskih paradigmi što je svakako velika prednost koju treba iskoristiti u razvoju programske podrške.

1. Programske paradigme

Programska paradigma (*eng. programming paradigm*) je stil ili način programiranja [1]. Općenito, paradigma je model prema kojem se nešto stvara, radi ili izvršava, pa se tako može razmatrati i programiranje. Temelj programskih paradigmi su matematičke teorije ili skup principa [2]. Neki programski jezici omogućavaju programerima da na jednostavan način pišu programe koristeći se određenom programskom paradigmom. Tako je na primjer dobra praksa koristiti funkcionalnu programsku paradigmu ukoliko se program piše u Scali, o funkcionalnoj programskoj paradigmi će se više govoriti u nastavku. S druge strane, nije ispravno koristiti frazu „paradigma programskog jezika" (*eng. „programming language paradigm"*) [1]. Točno je da se u nekim programskim jezicima određena paradigma jednostavno izražava, ali nije točno da je riječ o paradigmi programskog jezika. Na primjer, često se u literaturi može pronaći da je Haskell funkcionalni programski jezik, što nije ispravno. Istina je da se u Haskellu mogu pisati programi koristeći se funkcionalnom programskom paradigmom, ali nije istina da postoje funkcionalni programski jezici. Štoviše, moderni programski jezici omogućavaju pisanje programa koristeći se različitim programskim paradigmama. Danas se programi sve više pišu pomoću nekoliko programskih paradigmi. Tako nije rijetkost vidjeti da je jedan dio programa napisan koristeći objektno orijentiranu programsku paradigmu, dok je drugi dio programa napisan pomoću funkcionalne programske paradigme. Nije rijetkost da programe piše nekoliko programera, a neki dijelovi programa su obično biblioteke koje je netko tko nema veze s trenutnim programom napisao za široke potrebe. Tako se vrlo lako dođe do ispreplitanja različitih programskih paradigmi. Izbor programske paradigme bi se trebao temeljiti na problemu koji se želi riješiti, tj. na značajkama programa koji se piše. Određeni programski jezici sami po sebi nameću pisanje programa koristeći određenu programsku paradigmu. Kao primjer se mogu uzeti programi napisani programskim jezikom C#. Obično je riječ o programima koji su napisani u objektno orijentiranoj programskoj paradigmi, što nije nužnost nego navika programera koji su ih pisali. Bitna vještina programera je da zna kako se prebaciti s jednog programskog jezika na drugi [3]. Navedena vještina se ne postiže učenjem što više programskih jezika ispočetka. Programski jezici baš kao i prirodni jezici imaju sličnosti, ali postoje i karakteristike prema kojima se razlikuju. Poznavanje mehanizama pomoću kojih se implementiraju programi može pomoći. Štoviše, poznavanje različitih programskih paradigmi također može bit od pomoći prilikom prebacivanja s jednog programskog jezika na drugi. U nastavku će se predstavit nekoliko programskih paradigmi te njihove sličnosti i razlike.

1.1.Podjela programskih paradigmi

Napraviti podjelu programskih paradigmi nije jednostavan posao. Broj podjela programskih paradigmi je iznimno velik, skoro pa je jednak broju knjiga ili članaka koji se bave programskim paradigmama. Programske paradigme se obično prvo dijele na proceduralne i deklarativne programske paradigme. Proceduralna programska paradigma se temelji na korištenju procedura [4]. Procedure se često još nazivaju i potprogramima. Svaka procedura se može pozvati u bilo kojem trenutku izvršavanja programa, uključujući pozivanje unutar drugih procedura pa čak i unutar samih sebe. Proceduralna programska paradigma se često povezuje s imperativnom programskom paradigmom, nekada se čak smatraju sinonimima, iako to nije točno. Imperativna programska paradigma opisuje izvršavanje programa kao izraze koji mijenjaju stanje programa [5]. Programi napisani imperativnom programskom paradigmom se mogu promatrati kao niz naredbi koje računalo mora izvršiti u danom redoslijedu. Prema navedenim definicijama proceduralne i imperativne programske paradigme, smatram da je proceduralna programska paradigma podvrsta imperativne programske paradigme. Prema tome, smatram da se na najvišoj razini programske paradigme dijele na imperativne i deklarativne programske paradigme. Slika 1 prikazuje detaljniju podjelu programskih paradigmi.



Slika 1 Podjela programskih paradigmi

Na prethodnoj slici nisu prikazane sve programske paradigme koje postoje nego ih je prikazano samo nekoliko. Dalje u radu će se detaljnije obraditi nekoliko programskih paradigmi te njihove prednosti i mane. Kao primjeri će se koristiti programi napisani u programskom jeziku JavaScript. JavaScript je jedan od najpopularnijih programskih jezika u ovome trenutku. Jedini programski jezik koji mu trenutnu može donekle konkurirati u popularnosti je Java. JavaScript je sveprisutan u internet preglednicima, raznim mrežnim servisima, klijent-server i single-page aplikacijama. Velika prednost JavaScripta je velika zajednica programera koji ga koriste te široki spektar različitih biblioteka koje nastaju u velikom broju na dnevnoj bazi. Dodatno, osobno preferiram JavaScript jer ga aktivno koristim nekoliko godina i smatram da je idealan izbor za naredne primjere.

2. Imperativna programska paradigma

Prethodno je rečeno da imperativna programska paradigma opisuje izvršavanje programa kao izraze koji mijenjaju stanje programa [5]. Dodatno je rečeno da se programi napisani imperativnom programskom paradigmom mogu promatrati kao niz naredbi koje računalo mora izvršiti u danom redoslijedu. Radi jednostavnijeg razumijevanja, može se povući paralela s prirodnim jezikom kojim ljudi pričaju na dnevnoj bazi. Kako prirodni jezik sadrži naredbe koje se mogu uputiti nekome, tako i programski jezik sadrži naredbe koje se daju računalu kako bi računalo izvršilo program. Može se reći da se imperativna programska paradigma fokusira na izravno opisivanje računalu kako da izvrši određeni program, korak po korak. Imperativna programska paradigma je suprotnost od deklarativne programske paradigme koja se fokusira na to što računalo treba odraditi bez opisivanja kako računalo treba doći do željenog rezultata. Gotovo svako računalno sklopovlje je dizajnirano tako da izvršava strojni kod koji je prirodan računalu, a navedeni dizajn je implementiran najčešće koristeći imperativnu programsku paradigmu. Gledano s najniže razine, stanje programa je definirano sadržajem memorijskih lokacija, a naredbe su instrukcije napisane strojnim jezikom koji je razumljiv računalu. Viši programski jezici koriste varijable i kompleksnije naredbe, ali i dalje mogu pratiti imperativnu programsku paradigmu. U nastavku će se navesti primjer zbog jednostavnijeg shvaćanja imperativne programske paradigme. Neka je zadan niz koji sadrži brojeve: 1, 32, 13, 46, 15 , 62, 83, 4, 16 i 980. Potrebno je sve parne brojeve iz danog niza pohraniti u novi niz, novi niz je potrebno uzlazno sortirati i ispisati. Kod 1 prikazuje program napisan u imperativnoj programskoj paradigmi koji rješava navedeni zadatak.

```
let numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]; // niz brojeva
let even = []; // niz parnih brojeva
// pohranjivanje svih parnih brojeva iz niza numbers u niz even
for (let i = 0; i < numbers.length; i++){
  if(numbers[i] % 2 == 0)
      even.push(numbers[i]);
}
```

```
// sortiranje niza even
for(let i = 0; i < even.length - 1; i++){
  for( let j = i + 1; j < even.length; j++){
    if(even[i] > even[j]){
      let temp = even[i];
      even[i] = even[j];
      even[j] = temp;
    }
  }
}
// ispis niza even
console.log(even);
```

Kod 1 Imperativna programska paradigma

Navedeno rješenje je napisano u JavaScriptu i ono nije jedino točno rješenje, ali je dobar primjer za imperativnu programsku paradigmu. Analizom rješenja se vidi da je računalu izravno rečeno korak po korak što treba odraditi da bi se došlo do željenog rezultata, odnosno, vidi se da je kontrola toka eksplicitna.

Koraci izvršavanja programa:

1. U niz *numbers* pohrani brojeve: 1, 32, 13, 46, 15 , 62, 83, 4, 16 i 980.
2. Napravi prazan niz *even*.
3. Prođi kroz sve elemente niza *numbers*, ako je element paran broj, onda ga pohrani u niz *even*.
4. Prođi sve elemente niza *even* od prvog do predzadnjeg i indeks trenutnog elementa pohrani u varijablu *i* , prođi kroz sve elemente niza *even* od *i + 1* do zadnjeg i indeks trenutnog elementa pohrani u varijablu *j*, ako je vrijednost elementa na *i-toj* poziciji veća od vrijednosti elementa na *j-toj* poziciji, onda vrijednost elementa na *i-toj* poziciji pohrani u varijablu *temp*, vrijednost elementa na *j-toj* poziciji pohrani na *i-tu* poziciju niza *even*, a vrijednost varijable *temp* pohrani na *j-tu* poziciju niza *even.*
5. Ispiši niz *even.*

Analiza je svedena na samo pet koraka zbog jednostavnijeg praćenja i razumijevanja, inače bi se svaka linija programa mogla analizirati unutar jedne točke analize. Prethodni primjer prikazuje relativno kratak i jednostavan program. S druge strane, moguće je pisati iznimno kompleksne programe s složenim algoritmima. Neke algoritme je jednostavnije pisati u imperativnoj programskoj paradigmi nego u drugima. Dobar primjer je program za izračun umnoška dvije matrice. Naravno, takav program se može napisati i nekom drugom programskom paradigmom kao što je funkcionalna. Programerima takav program napisan funkcionalnom programskom paradigmom često nije prirodan i obično će se opredijeliti za imperativnu programsku paradigmu u tom slučaju. Programi napisani imperativnom programskom paradigmom su poprilično jednostavni za shvatit u usporedbi s onima napisanim u drugim paradigmama te se iz tog razloga imperativna programska paradigma učestalo koristi prilikom samih početaka učenja programiranja. Prethodno su nabrojane neke podvrste imperativne programske paradigme, u nastavku će se govoriti o proceduralnoj programskoj paradigmi.

2.1. Proceduralna programska paradigma

Podsjetimo se, proceduralna programska paradigma se temelji na korištenju procedura koje se često još nazivaju i potprogramima [4]. Svaka procedura se može pozvati u bilo kojem trenutku izvršavanja programa, uključujući pozivanje unutar drugih procedura pa čak i unutar samih sebe. Velika prednost proceduralne programske paradigme je modularnost koda, tj. svojstvo da se program napisan proceduralnom programskom paradigmom promatra kao skup modula ili potprograma. Svaki potprogram je zadužen za obavljanje određene zadaće, odnosno za izvršavanje određenog skupa instrukcija. Program se implementira pisanjem potprograma u obliku metoda i funkcija, a izvršava se izvršavanjem potprograma u ispravnom redoslijedu. Općenito, modularnost koda je poželjna, a posebno u složenim programima [6]. Ukoliko se modularnost koda realizira proceduralnom programskom paradigmom odnosno pisanjem potprograma u obliku metoda, tada metode dobivaju ulazne vrijednosti u obliku argumenata koje prima metoda, a izlaz metode je definiran vrijednostima koje metoda vraća. Potrebno je obratiti pozornost na *scope* (pojednostavljeno, dio koda unutar kojega možemo dohvatiti varijablu) varijabli. Metodama je potrebno onemogućiti dohvaćanje varijabli drugih metoda ili vlastitih prethodnih poziva. Ukoliko metoda može dohvaćati varijable drugih metoda, tada se narušava modularnost koda što nije dobra praksa. Prisjetimo se spomenutog zadatka: „Neka je zadan niz koji sadrži brojeve: 1, 32, 13, 46, 15 , 62, 83, 4, 16 i 980. Potrebno je sve parne brojeve iz danog niza pohraniti u novi niz, novi niz je potrebno uzlazno sortirati i ispisati.“. Zbog jednostavnije shvaćanja proceduralne programske paradigme, zadatak ćemo riješiti koristeći proceduralnu programsku paradigmu. Kod 2 prikazuje rješenje zadatka napisano koristeći proceduralnu programsku paradigmu.

```
function getNumbers(){ // niz brojeva
  return [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];
}
function getEvenNumbers(numbers){ // niz parnih brojeva
  let even = [];
  for (let i = 0; i < numbers.length; i++){
    if(numbers[i] % 2 == 0)
        even.push(numbers[i]);
  }
```

```
  return even;
}
function sortArray(numbers){ // sortiranje niza
  let sorted = numbers;
  for(let i = 0; i < sorted.length - 1; i++){
    for( let j = i + 1; j < sorted.length; j++){
      if(sorted[i] > sorted[j]){
        let temp = sorted[i];
        sorted[i] = sorted[j];
        sorted[j] = temp;
      }
    }
  }
  return sorted;
}
console.log(sortArray(getEvenNumbers(getNumbers()))); // ispis niza
```

Kod 2 Proceduralna programska paradigma

Program se sastoji od tri funkcije: *getNumbers(),getEvenNumbers(numbers)* i *sortArray(numbers)*. Svaka funkcija se može gledati kao zasebni potprogram koja obavlja određenu zadaću:

1. *getNumbers()* – ne prima ulazne vrijednosti, vraća niz zadanih brojeva.
2. *getEvenNumbers(numbers)* – kao ulaz prima niz cijelih brojeva, vraća niz koje sadrži samo parne brojeve prethodnog niza.
3. *sortArray(numbers)* – kao ulaz prima niz cijelih brojeva, a vraća uzlazno sortirani niz.

Zadatak je riješen pozivanjem navedenih funkcija u ispravnom poretku. Kod programa je sličan rješenju iz Koda 1. Grubo rečeno, neki dijelovi Koda 1 su samo napisani unutar funkcija što je dovoljno da se rješenje smatra napisanim proceduralnom programskom paradigmom koja je podvrsta imperativne programske paradigme. Ponekad je teško napraviti razliku između programskih paradigmi jer pisanje programa i algoritama unutar programskih paradigmi nije

jednoznačno određeno. Kod 3 prikazuje još jedno rješenje zadatka napisano proceduralnom programskom paradigmom.

```
function getEvenNumbers(numbers){ // niz parnih brojeva
  let even = [];
  for (let i = 0; i < numbers.length; i++){
    if(numbers[i] % 2 == 0)
        even.push(numbers[i]);
  }
  return even;
}

function sortArray(numbers){ // sortiranje niza
  let sorted = numbers;
  for(let i = 0; i < sorted.length - 1; i++){
    for( let j = i + 1; j < sorted.length; j++){
      if(sorted[i] > sorted[j]){
        let temp = sorted[i];
        sorted[i] = sorted[j];
        sorted[j] = temp;
      }
    }
  }
  return sorted;
}

const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]; // niz  brojeva

let even = getEvenNumbers(numbers); // niz parnih brojeva
even = sortArray(even); // sortiraj niz even
console.log(even); // ispis
```

Kod 3 Proceduralna programska paradigma 2

Važno je imati na umu da je programska paradigma samo stil pisanja programskog koda, a ne skup jednoznačno definiranih pravila kako bi svi programi koji rješavaju isti problem napisani istom paradigmom bili identično napisani. Tablica 1 prikazuje Kod 2 i Kod 3.

Kod 2 Proceduralna programska paradigma	Kod 3 Proceduralna programska paradigma 2

`function getNumbers(){ // niz brojeva`
`  return [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];`
`}`
`function getEvenNumbers(numbers){ // niz parnih brojeva`
`  let even = [];`
`  for (let i = 0; i < numbers.length; i++){`
`  if(numbers[i] % 2 == 0)`
`  even.push(numbers[i]);`
`  }`
`  return even;`
`}`
`function sortArray(numbers){ // sortiranje niza`
`  let sorted = numbers;`
`  for(let i = 0; i < sorted.length - 1; i++){`
`  for( let j = i + 1; j < sorted.length; j++){`
`  if(sorted[i] > sorted[j]){`
`  let temp = sorted[i];`
`  sorted[i] = sorted[j];`
`  sorted[j] = temp;`
`  }`
`  }`
`  }`
`  return sorted;`
`}`
 	
`// ispis niza`
`console.log(sortArray(getEvenNumbers(getNumbers())));`
	`function getEvenNumbers(numbers){ // niz parnih brojeva`
	`  let even = [];`
	`  for (let i = 0; i < numbers.length; i++){`
	`  if(numbers[i] % 2 == 0)`
	`  even.push(numbers[i]);`
	`  }`
	`  return even;`
	`}`
	
	`function sortArray(numbers){ // sortiranje niza`
	`  let sorted = numbers;`
	`  for(let i = 0; i < sorted.length - 1; i++){`
	`  for( let j = i + 1; j < sorted.length; j++){`
	`  if(sorted[i] > sorted[j]){`
	`  let temp = sorted[i];`
	`  sorted[i] = sorted[j];`
	`  sorted[j] = temp;`
	`  }`
	`  }`
	`  }`
	`  return sorted;`
	`}`
	
	`// niz  brojeva`
	`const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];`
	
	`let even = getEvenNumbers(numbers); // niz parnih brojeva`
	`even = sortArray(even); // sortiraj niz even`
	`console.log(even); // ispis`

Tablica 1 Proceduralna programska paradigma: usporedba programskog koda

Oba rješenja su napisana proceduralnom programskom paradigmom, ali se ipak razlikuju. Prva razlika je broj funkcija, Kod 2 se sastoji od tri funkcije dok Kod 3 ima dvije. Kada bi se svaka funkcija smatrala potprogramom, može se reći da se rješenja razlikuju po broju potprograma. Funkcija `getNumbers()` ne postoji u Kodu 3, tj. umjesto poziva funkcije koja vraća zadani niz brojeva, isti niz je pohranjen u varijablu `numbers`. Slijedeća bitna razlika između dva koda je ta što se unutar Koda 3, konačno sortirani niz parnih brojeva pohranjuje u varijablu `even` dok se u Kodu 2 taj niz samo ispisuje bez pohranjivanja u varijablu. Ostale razlike između dva koda trenutno nisu bitne. Niti jedno od dva navedena rješenja nije optimalno te služe isključivo za potrebe savladavanja proceduralne programske paradigme odnosno boljeg razumijevanja iste. Nakon što je dano objašnjenje proceduralne programske paradigme, u nastavku će se razmatrati njene karakteristike.

Neke od karakteristika proceduralne programske paradigme [7]:

1. Naglasak na izvršavanju algoritama
2. Složeni problemi podijeljeni u manje probleme odnosno potprograme, metode ili funkcije
3. Većina metoda koristi globalne podatke
4. Podaci se slobodno kreću sustavom iz jedne metode u drugu
5. Metode mijenjaju podatke iz jednog oblika u drugi
6. Potiče dizajniranje programske podrške odozgo prema dolje

Problem proceduralne programske paradigme je što veliki broj bitnih podataka obično bude globalno dostupan unutar cijelog programa, odnosno mogu mu pristupiti sve metode ili potprogrami. Drugim riječima, nije rijetkost da prilikom implementacije programa dođe do pogrešaka u obliku nenamjernog mijenjanja globalno dostupnih podataka unutar neke od metoda. U tom slučaju je potrebno dobro provjeriti strukturu globalno dostupnih podataka, ali i sve metode koje im imaju pristup. Možda najveći problem proceduralne programske paradigme je taj što je modeliranje problema iz stvarnoga svijeta iznimno komplicirano. Kompliciranje je zato što su metode akcijski orijentirane [7], izvršavaju se samo onda kada se pozovu, a nekada je nemoguće predvidjeti kada se sve određena metoda treba pozvati. Poznavanje proceduralne programske paradigme može uvelike pomoći prilikom savladavanja nešto kompleksnijih programskih paradigmi kao što su objektno orijentirana ili funkcionalna.

2.2. Objektno orijentirana programska paradigma

Objektno orijentirana programska paradigma se temelji na konceptu objekata [8]. Objekti sadržavaju podatke u obliku polja koja se još nazivaju atributima ili svojstvima. Dodatno, sadrže kod u obliku procedura koje se još nazivaju metodama. Svojstvo objekata je da pomoću procedura mogu pristupiti i mijenjati polja odnosno podatke s kojima su povezani. Programi napisani objektno orijentiranom paradigmom su dizajnirani tako da sadrže objekte koji su u interakciji. Postoje jezici koji su objektno orijentirani te između njih postoje značajne razlike. Najpopularniji objektno orijentirani jezici se temelje na klasama, što znači da su objekti instance klase koja objektima određuje tip. Primjer takvoga jezika je C#.

Kada se priča o objektno orijentiranoj paradigmi, najčešće se spominju [9]:

1. Objekt, metoda, svojstvo
2. Klasa
3. Enkapsulacija
4. Agregacija
5. Nasljeđivanje
6. Polimorfizam

Navedeni koncepti će se detaljnije opisati u nastavku.

2.2.1. Objekt

Kao što je i ranije spomenuto, koncept objekta je temelj objektno orijentirane paradigme. Objekt je reprezentacija nekoga ili nečega, ta reprezentacija je izražena pomoću programskog jezika. Objekt može biti bilo što, može bit iz stvarnoga svijeta ili nešto izmišljeno. Kao primjer objekta može se uzeti automobil, njegove karakteristike su: proizvođač, godina proizvodnje, zapremnina motora, broj brzina, dužina, širina, visina i mnoge druge. Automobil ima i određene akcije kao što su: kretanje, ubrzavanje, usporavanje, otvaranje prozora i slično. U terminima objektno orijentirane paradigme, navedene karakteristike su svojstva automobila, a navedene akcije su njegove metode.

```
let vehicle = new Car();
```

Kod 4 Objekt u JavaScriptu

2.2.2. Klasa

U objektno orijentiranom programiranju, klasa predstavlja proširivi predložak za stvaranje objekata [10]. Klasa je skup svojstava i metoda koje imaju njene instance odnosno objekti. Na temelju jedne klase se može napraviti više objekata jer je klasa samo predložak za stvaranje tih objekata. S druge strane, objekti su konkretne instance, temeljene na predlošku tj. klasi. Na primjer, orao i sokol su ptice, može se reći da oni pripadaju klasi *Ptica.* Kako je u ovome radu naglasak na programskom jeziku JavaScript, odmah naglašavam da unutar JavaScripta ne postoje klase u uobičajenom smislu nego se sve temelji na objektima i prototipiranju. O tome će se govoriti nešto kasnije.

```
class Bird{
  constructor(type){
    this.type = type;
  }
  toString(){
    console.log(this.type);
  }
}
let eagle = new Bird(`eagle`);
let hawk = new Bird(`hawk`);

eagle.toString(); // ispisuje 'eagle' na konzolu
hawk.toString(); // ispisuje 'hawk' na konzolu
```

Kod 5 Klasa u JavaScriptu

2.2.3. Enkapsulacija

Obično se pojam enkapsulacije se povezuje sa skrivanjem informacija. Pojam enkapsulacije unutar objektno orijentirane paradigme je najlakše objasniti primjerom. Zamislite mobitel, mobitela ima sučelje za korištenje, npr. tipke i ekran. Sučelje se koristi kako bi korisnik uputio mobitel da odradio nešto, npr. nazove nekoga. Napraviti poziv na mobitelu je iznimno jednostavno, ali obično ne znamo što se događa u pozadini tog procesa. Drugim riječima, implementacija sučelja je skrivena od nas. Slično se događa i u objektno orijentiranom programiranju. Programski kod koji smo napisali može pozivati metode nekog drugog objekta, a taj drugi objekt može bit dio biblioteke koju je napisao nama netko potpuno nepoznat. Nas kao programera ne zanima kako ta metoda radi dokle god nam daje ispravan rezultat. U programskim jezicima koji koriste komplajlere, taj kod biblioteke je nemoguće pročitati. S druge strane, kako JavaScript koristi interpreter, moguće je vidjeti izvorni kod, ali je nebitan i koncept ostaje isti. Drugi aspekt skrivanja informacija je vidljivost metoda i svojstava [9]. U nekim jezicima, kao što je C#, objekti imaju *public, private* i *protected* metode i svojstva. Navedena kategorizacija definira razinu dostupnosti. Tako npr. svojstvima i metodama koji su označeni s *private* se može pristupiti samo unutar implementacije samoga objekta, dok onima označenim s *public* mogu pristupiti svi. Ponovo će se povući paralela s JavaScriptom, gdje su sva svojstva i metode označene s *public* ,ali postoje načini za zaštitu podataka unutar objekata. Tako se varijable mogu zaštititi pomoću *Closurea* koji globalne varijable pretvaraju u privatne odnosno lokalne. U nastavku je prikazana funkcija `add` koja se koristi za brojanje. Funkcija `add` ima lokalnu varijablu `counter` postavljenu na 0, navedena varijabla se uvećava za 1 svakim pozivom funkcije `add`, ali ju nije moguće uređivati izvan funkcije.

```
let add = (() => {
  let counter = 0;
  return () => counter +=1;
})();

add(); // rezultat: 1
add(); // rezultat: 2
add(); // rezultat: 3
```

Kod 6 Enkapsulacija u JavaScriptu

2.2.4. Agregacija

Agregacija označava kombiniranje nekoliko objekata u jedan [9]. Povezuje se s pojmom kompozicije objekata. Neki programeri agregaciju i kompoziciju smatraju sinonimima, dok drugi smatraju da su to dva različita pojma. Recimo da imamo dva objekta, objekt *A* i objekt *B*. Programeri koji rade razliku između agregacije i kompozicije tvrde da je riječ o agregaciji ukoliko objekt *A* „koristi“ objekt *B*, tj. objekt *B* postoji na konceptualnoj razini neovisno o objektu *A*. Primjer takvog slučaja je odnos *playliste* i *pjesama*, nakon što obrišemo *playlistu*, *pjesme* i dalje postoje te se mogu koristiti neovisno o *playlisti*. S druge strane ako brisanjem objekta *A*, objekt *B* gubi smisao i svrhu, onda je riječ o kompoziciji. Na primjer, ako obrišemo sve *pjesme*, onda *playlista* sama po sebi nema značenje niti smisao pa prema tome ne treba niti postojati u programu. Agregacija je korisna za dijeljenje složenih problema na niz manjih problema. Na taj način je moguće razmišljati o problemu na nekoliko razina apstrakcije. Na primjer, osobno računalo je složen predmet. Iznimno je teško razmišljati o svemu što se događa prilikom pokretanja računala. Kako bi se problem pojednostavnio, računalo možemo podijeliti na nekoliko objekata kao što su tipkovnica, monitor, miš i drugi. Zatim se možemo pozabaviti događanjima unutar svakog pojedinog objekata što je dosta jednostavnije. Ono što smo napravili je podjela objekta na manje dijelove koje ćemo opet moći koristiti prema potrebi.

```
class Song{
  constructor(title, author){
    this.title = title;
    this.author = author;
  }
}

class PlayList{
  constructor(title){
    this.title = title
    this.songs = new Array();
  }
  addSong(title, author){
    this.songs.push(new Song(title,author));
```

```
  }
  toString(){
    console.log(this.songs);
  }
}

let list = new PlayList(`Favourite`);

list.addSong(`Another Brick In The Wall`, `Pink Floyd`);
list.addSong(`Far From Any Road`, `The Handsome Family`);
list.toString();
// rezultat:
// [ Song { title: 'Another Brick In The Wall', author: 'Pink Floyd' },
//  Song { title: 'Far From Any Road', author: 'The Handsome Family' } ]
```

Kod 7 Agregacija u JavaScriptu

2.2.5. Nasljeđivanje

Nasljeđivanje je vrlo elegantan način ponovnog korištenja već napisanog programskog koda. Osim što smanjuje vrijeme potrebno za razvoj programske podrške dodatno povećava i pouzdanost. Ukoliko se koristi programski kod koji je već testiran i korišten, razumno je smatrati ga pouzdanijim od programskog koda kojega tek treba napisati. Zamislite da imate generički objekt *vozilo* koji ima svojstva kao što su *proizvođač, godina proizvodnje* i *maksimalna brzina* te dodatno ima metode kao što su *vozi, koči, kreni* i *stani*. S vremenom se pojavi potreba za nastankom objekta *automobil*. Moguće je sva svojstva i metode iz objekta *vozilo* ponovno implementirati unutar vozila *automobil*, ali bi bilo pametnije da se sva svojstva i metode iz *vozila* naslijede i kao takve koriste unutar objekta *automobil*, a da se dodatna svojstva i metode koje ne postoje u objektu *vozilo* implementiraju, kao npr. metoda *otvoriVrata*. Obično u klasično objektno orijentiranom programiranju, klase nasljeđuju metode i svojstva iz drugih klasa. Kod JavaScripta je situacija nešto drugačija, objekti nasljeđuju od objekata jer klase ne postoje.

```
class Vehicle{
  constructor(brand, year, topSpeed){
    this.brand = brand;
    this.year = year;
    this.topSpeed = topSpeed;
  }
  drive(){
    ...
  }
  start(){
    ...
  }
  stop(){
    ...
  }
}

class Car extends Vehicle{
```

```
  constructor(brand, year, topSpeed){
    super(brand, year, topSpeed);
  }
  doorOpen(){
    ...
  }
}
```

Kod 8 Nasljeđivanje u JavaScriptu

2.2.6. Polimorfizam

Prisjetimo se ranije spomenutog primjera vezanog za nasljeđivanje svojstava i metoda kod *vozila* i *automobila*. Zamislite da se u programskom kodu nalazi varijabla *prijevoznoSredstvo*, a da pri tome ne znamo je li riječ o *vozilu, automobilu* ili nekom drugom prijevoznom sredstvu. Unatoč navedenom problemu, i dalje možemo pozvati metodu *vozi* te neće doći do pogreške prilikom izvršavanja programa. Upravo to svojstvo nazivamo polimorfizam. Polimorfizam je pružanje jedinstvenog sučelja entitetima različitih tipova [11]. Tako će u navedenom primjeru *prijevoznoSredstvo* izvršiti metodu *vozi* neovisno o tome je li ono *vozilo, automobil* ili neko drugo prijevozno sredstvo, ako je tipa *vozilo* onda će se izvršiti metoda *vozi* definirana u klasi *vozilo*, a ako je *automobil* onda će se izvršiti metoda *vozi* definirana u klasi *automobil*, pri čemu metoda *vozi* ne mora biti implementirana na isti način u obje klase.

```
let vehicles = [new Car(), new Airplane()];


for(let i = 0;i < vehicles.length; i++)
  vehicles[i].drive();
```

Kod 9 Polimorfizam u JavaScriptu

Sada kada je detaljno objašnjenja objektno orijentirana programska paradigma. Prisjetimo se zadatka koji je rješavan koristeći imperativnu i proceduralnu programsku paradigmu. Zadatak glasi: „Neka je zadan niz koji sadrži brojeve: 1, 32, 13, 46, 15 , 62, 83, 4, 16 i 980. Potrebno je sve parne brojeve iz danog niza pohraniti u novi niz, novi niz je potrebno uzlazno sortirati i ispisati.“. Riješimo zadatak koristeći objektno orijentiranu programsku paradigmu. Kod 10 prikazuje traženo rješenje.

```
class Numbers{
  constructor(values){
    this.values = values;
  }
  toString(){
    console.log(this.values);
  }
}
```

```
class Services{
  static getEvenNumbers(numbers){ // niz parnih brojeva
    let even = [];
    for (let i = 0; i < numbers.length; i++){
      if(numbers[i] % 2 == 0)
          even.push(numbers[i]);
    }
    return even;
  }
  static sortArray(numbers){ // sortiranje niza
    let sorted = numbers;
    for(let i = 0; i < sorted.length - 1; i++){
      for( let j = i + 1; j < sorted.length; j++){
        if(sorted[i] > sorted[j]){
          let temp = sorted[i];
          sorted[i] = sorted[j];
          sorted[j] = temp;
        }
      }
    }
    return sorted;
  }
}

const numbers = new Numbers([1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]);

let even = new Numbers();
even.values = Services.getEvenNumbers(numbers.values);
even.values = Services.sortArray(even.values);
even.toString();
```

Kod 10 Objektno orijentirana programska paradigma

Prije same analize rješenja, potrebno je još jedan puta napomenuti da JavaScript nije objektno orijentiran jezik koliko god to netko htio. Korištenje ključne riječi *class* u JavaScriptu je omogućemo izlaskom ECMAScript 6 [12]. Prije toga ne bi bilo moguće ovako strukturirati rješenje. Odabrao sam ovakvu strukturu rješenja jer je ona najsličnija objektno orijentiranom programiranju u drugim jezicima. Iako značenje klasa unutar JavaScripta nije jednako onome unutar C# ili Jave, rješenje se sastoji od dvije klase *Numbers* i *Services.* U jedan objekt tipa *Numbers* je spremljen početno zadani niz brojeva, a u drugi je spremljen traženi niz brojeva. Izdvajanje parnih brojeva iz nekog niza i sortiranje niza su postignuti statičnim metodama iz klase *Services.* Za samo ispisivanje traženog niza se koristila metoda *toString* iz klase *Numbers.* Objekti se u JavaScriptu obliku prikazuju u obliku JSON-a (*JavaScript Object Notation*) [13]. Kod 11 prikazuje primjer JSON-a.

```
let numbers = {
  value: [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980],
  toString: function(){
    console.log(this.value);
  }
}
// ispis: 1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]
numbers.toString();
```

Kod 11 JSON

2.3. Paralelna programska paradigma

Paralelna programska paradigma je stil pisanja programa u kojemu se više izračuna ili procesa izvršavaju simultano [14]. Kompleksni problem se može podijeliti na manje probleme koji se onda u isto vrijeme rješavaju. Paralelna programska paradigma se često povezuje s raspodijeljenom programskom paradigmom, ali između njih postoje razlike. Raspodijeljena programska paradigma se odnosi na stil pisanja programa u kojemu računala ili njihove komponente komuniciraju putem poruka koristeći mrežu kako bi se koordinirale akcije i riješio problem [15]. S druge strane, kod paralelne programske paradigme se svi procesi i izračuni izvršavaju na jednom računalu. JavaScript nije popularan jezik za paralelno programiranje, ali ipak postoje biblioteke koje to omogućavaju. Primjer jedne takve biblioteke je Parallel.js [16]. Koristeći JavaScript i Parallel.js će se riješiti prethodni zadatak koji glasi: „Neka je zadan niz koji sadrži brojeve: 1, 32, 13, 46, 15 , 62, 83, 4, 16 i 980. Potrebno je sve parne brojeve iz danog niza pohraniti u novi niz, novi niz je potrebno uzlazno sortirati i ispisati.“. Kod 12 prikazuje rješenje pisano paralelnom programskom paradigmom.

```
// biblioteka za parelelno programiranje
const Parallel = require('paralleljs');


// niz  brojeva
const numbers = new Parallel([1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]);


// na drugoj niti filtriraj niz tako da ostanu samo parni brojevi, sortiraj i
// ispisi rezultat na glavnoj niti
numbers.spawn(data => data.filter(number => number % 2 == 0)).then(data =>
data.sort((x, y) => x - y)).then(console.log);
```

Kod 12 Paralelna programska paradigma

Kao i u prethodnim primjerima, niz zadanih brojeva je pohranjen u varijablu `numbers`, ali se ovaj put za pohranu niza koristi objekt iz biblioteke Parallel.js što omogućava korištenje paralelne programske paradigme za manipulaciju elementima niza. Rješenje je implementirano tako da se koristeći funkciju `spawn` stvara novi proces na drugoj niti koji filtrira niz tako da u njemu ostanu samo parni brojevi. Zatim se dobiveni niz sortira te na kraju ispisuje na glavnoj niti. Rješenje ne ističe prednosti paralelnog programiranja, ali je strukturirano tako da se jasno vidi zašto je

paralelna programska paradigma podvrsta imperativne programske paradigme. Iako se ne izvršava sve na istoj niti i nije poznato vrijeme izvršavanja svakog procesa, ipak je redoslijed izvršavanja točno opisan korak po korak. Štoviše, ukoliko se razmatraju programske paradigme, nije dovoljno samo promatrati način izvršavanja programa, nego je potrebno posvetiti pažnju i stilu kojim se implementira odnosno piše programski kod. Kako JavScript trenutno nije popularan jezik za pisanje programa koristeći se paralelnom programskom paradigmom, ista se neće detaljno razmatrati.

2.4. Pregled imperativne, proceduralne i objektno orijentirane programske paradigme

Prisjetimo se programskog koda napisano u imperativnoj, proceduralnoj i objektno orijentiranoj programskoj paradigmi. Tablica 2 prikazuje programski kod napisan u navedenim paradigmama.

Imperativna programska paradigma

```
let numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]; // niz brojeva
let even = []; // niz parnih brojeva
// pohranjivanje svih parnih brojeva iz niza numbers u niz even
for (let i = 0; i < numbers.length; i++){
  if(numbers[i] % 2 == 0)
      even.push(numbers[i]);
}


// sortiranje niza even
for(let i = 0; i < even.length - 1; i++){
  for( let j = i + 1; j < even.length; j++){
    if(even[i] > even[j]){
      let temp = even[i];
      even[i] = even[j];
      even[j] = temp;
    }
  }
}
// ispis niza even
console.log(even);
```

Proceduralna programska paradigma

```
function getEvenNumbers(numbers){ // niz parnih brojeva
  let even = [];
  for (let i = 0; i < numbers.length; i++){
    if(numbers[i] % 2 == 0)
        even.push(numbers[i]);
  }
  return even;
}

function sortArray(numbers){ // sortiranje niza
  let sorted = numbers;
  for(let i = 0; i < sorted.length - 1; i++){
    for( let j = i + 1; j < sorted.length; j++){
      if(sorted[i] > sorted[j]){
        let temp = sorted[i];
        sorted[i] = sorted[j];
        sorted[j] = temp;
      }
    }
  }
  return sorted;
}

const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]; //
niz  brojeva

let even = getEvenNumbers(numbers); // niz parnih brojeva
even = sortArray(even); // sortiraj niz even
console.log(even); // ispis
```

Objektno orijentirana programska paradigma

```
class Numbers{
  constructor(values){
    this.values = values;
  }
  toString(){
    console.log(this.values);
  }
}

class Services{
  static getEvenNumbers(numbers){ // niz parnih brojeva
    let even = [];
    for (let i = 0; i < numbers.length; i++){
      if(numbers[i] % 2 == 0)
          even.push(numbers[i]);
    }
    return even;
  }
  static sortArray(numbers){ // sortiranje niza
    let sorted = numbers;
    for(let i = 0; i < sorted.length - 1; i++){
      for( let j = i + 1; j < sorted.length; j++){
        if(sorted[i] > sorted[j]){
          let temp = sorted[i];
          sorted[i] = sorted[j];
          sorted[j] = temp;
        }
      }
    }
    return sorted;
  }
}

const numbers = new Numbers([1, 32, 13, 46, 15 , 62, 83, 4,
16, 980]);

let even = new Numbers();
even.values = Services.getEvenNumbers(numbers.values);
even.values = Services.sortArray(even.values);
even.toString();
```

Tablica 2 Pregled imperativne, proceduralne i objektno orijentirane programske paradigme

Kako su proceduralna i objektno orijentirana programska paradigma podvrste imperativne programske paradigme, odmah na prvi pogled se vide sličnosti. Računalu je eksplicitno navedeno korak po korak što treba napraviti kako bi došli do željenog rješenja. U rješenju pisanom proceduralnom programskog paradigmom se vidi da je programski kod ovisno o zadaćama podijeljen na funkcije dok je kod objektno orijentirane paradigme programski kod prema zadaćama podijeljen na objekte. Početnicima je najjednostavnije shvatiti rješenje napisano imperativnom programskom paradigmom, a kako znanje vezano za razvoj programske podrške raste, obično će s vremenom preko proceduralne doći do preferiranja objektno orijentirane programske paradigme. Svaka programska paradigma ima prednosti i mane, odnosno karakteristike o kojima se prethodno pisalo. Kako je programski kod iz Tablice 2 zapravo rješenje vrlo jednostavnog problema, teško je njime ilustrirati apstrakciju koju omogućava proceduralna programska paradigma ili polimorfizam koji je velika prednost objektno orijentirane programske paradigme. Ono što se programskim kodom prethodno napisanim u trima paradigmama postiže je uvid u sličnosti i razlike programskih paradigmi. Kada bi se pogledali nešto složeniji programi i njihov izvorni kod, razlike bi bile sve veće a prednosti i mane pojedine paradigme sve izraženije. Izbor pojedine programske paradigme bi se trebao svoditi na problem koji je potrebno riješiti te ne postoji jedinstveno rješenje koja paradigma je idealan izbor. Ukoliko se pišu kratki i jednostavni programi, imperativna programska paradigma je dobar izbor, s druge strane je iznimno loš izbor ako je potrebno napisati složeni program za kontrolu letova u Europi. Također, nema potrebe koristiti objektno orijentiranu programsku paradigmu ako je cilj programa vrlo jednostavan poput zbrajanja dva broja. Osim što u izboru programske paradigme veliku ulogu igra problem koji se želi riješiti, bitno je i znanje programera te njegovo iskustvo u pisanju programskog koda.

3. Deklarativna programska paradigma

Kontrola toka kod programa napisanih deklarativnom programskom paradigmom je implicitna, programer navodi kako bi željeni rezultat trebao izgledati, ali ne i kako ga dobiti korak po korak [17]. Dobar primjer alata za deklarativno programiranje su relacijske baze podataka. Neki smatraju da su oni zapravo preteča deklarativnog programiranja. Prije relacijskih baza podataka, podacima iz baza podataka se manipuliralo koristeći se programskim kodom napisanim u imperativnoj programskoj paradigmi. Problem se javlja što takav kod ovisi o nizu sitnih i nepredvidivih detalja koji su vezani za podatke, npr. broj podataka, indeksi, fizička putanja do podataka i slično. Kako se navedeni elementi mogu mijenjati kroz vrijeme, problem se javljao u obliku prestanka rada već gotovog programa. Očito, takav kod je bilo teško pisati, čitati, otklanjati mu pogreške i održavati. Nešto bliži primjer današnjim programerima što se tiče deklarativnog programiranja je pisanje testova. Radi jednostavnijeg razumijevanja će se navesti primjer. Zamislite da pišete program za knjižnicu. Program mora imati mogućnost unosa nove knjige, ali uz određene restrikcije. Knjigu je moguće unijeti tek onda kada ima sve potrebne podatke, a potrebni podaci su: naslov, autor, godina izdavanja i broj stranica. Osim što treba napisati odgovarajući program, potrebno je i napisati testove za program. Kod 13 prikazuje jednostavan test za navedeni program napisan koristeći Chais.js [18].

```
describe('/POST book', () => {
    it('it should not POST a book without pages field', (done) => {
      let book = {
          title: "The Lord of the Rings",
          author: "J.R.R. Tolkien",
          year: 1954
      }
      chai.request(server)
          .post('/book')
          .send(book)
          .end((err, res) => {
              res.should.have.status(200);
              res.body.should.be.a('object');
              res.body.should.have.property('errors');
```

```
                res.body.errors.should.have.property('pages');
                res.body.errors.pages.should.have.property('kind').eql('required');
              done();
            });
      });
      it('it should POST a book ', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954,
            pages: 1170
        }
        chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('message').eql('Book successfully added!');
                res.body.book.should.have.property('title');
                res.body.book.should.have.property('author');
                res.body.book.should.have.property('pages');
                res.body.book.should.have.property('year');
              done();
            });
      });
  });
```

Kod 13 Deklarativno programiranje

Kod 13 prikazuje test za pohranu nove knjige. Test se može razdvojiti na dva dijela, prvi dio gdje se pokušava pohraniti knjigu bez svih potrebnih podataka i drugi dio u kojemu se želi na ispravan način sa svim podacima pohraniti knjigu u bazu podataka preko servera. U kodu je navedeno što se očekuje kao rezultate svake od dvije navedene akcije, ali nije navedeno kako će se doći do toga rezultata. Tako je za prvi dio testa npr. navedeno da se knjiga neće pohraniti, server će vratiti objekt koji sadrži svojstvo *errors*, a svojsvtvo *errors* treba imati svojstvo *pages*, dok svojstvo *pages* treba

imati svojstvo *kind* koje ima vrijednost *required*. Ukoliko se dobije navedeni rezultat, prvi dio testa je uspješno proveden, a inače postoji pogreška u programskom kodu. U drugom dijelu testa je navedeno da program treba ispravno pohraniti knjigu, rezultat koji server vrati treba imati *status kod* čija je vrijednost *200*, rezultat treba bit objekt koji ima svojstvo *message* čija je vrijednost *Book successfully added!*, dodatno je detaljnije opisan rezultat. Dakle, u testu je koristeći deklarativnu programsku paradigmu navedeno kakvo ponašanje programa se očekuje da bi se program smatrao ispravnim. Bitno je primijetiti da je samo navedeno što se očekuje, ali ne i kako provjeriti je li dobiven očekivani rezultat. U nastavku će se razmatrati podvrste deklarativne programske paradigme.

3.1. Logička programska paradigma

Logička programska paradigma se temelji na formalnoj logici [19]. Svaki program napisan logičkom programskom paradigmom je zapravo skup rečenica koje sadrže činjenice i pravila koja su vezana za neki problem. Za logičko programiranje su se uvelike koristili Prolog i Datalog. U navedenim jezicima, pravila su se pisala u obliku rečenica: G :- $T_1, \ldots, T_n$. Gdje se G naziva glavom pravila, a $T_1, \ldots, T_n$ tijelom pravila. Činjenice su pravila koja nemaju tijelo i jednostavno se pišu u obliku: G. Kako logička programska paradigma trenutno nije popularna, neće se ulaziti u teoriju nego će se više o njoj pokazati na primjerima. Primjeri će se pisati programskim jezikom JavaScript koristeći LogicJS biblioteku [20] i Node.js [21]. Kod 14 prikazuje primjer programa napisan logičkom programskom paradigmom.

```
let logic = require('logicjs');

let add = logic.add;
let and = logic.and;
let div = logic.div;
let eq = logic.eq;
let lvar = logic.lvar;
let mul = logic.mul;
let or = logic.or;
let run = logic.run;
let sub = logic.sub;

let x = lvar();

run(add(x,10,5), x); // rezultat: [ -5 ]
run(add(x,5,10), x); // rezultat: [ 5 ]
run(add(10,x,5), x); // rezultat: [ -5 ]
run(add(10,5,x), x); // rezultat: [ 15 ]
```

Kod 14 Logička programska paradigma

U Kodu 14 su najzanimljivije posljednje četiri linije koda, sve prethodno napisane linije se odnose na dohvaćanje LogicJS biblioteke i pridjeljivanje metoda iz biblioteke određenim varijablama. U komentaru se nalazi rezultat svake od posljednje četiri linije koda. Ukoliko pokušate pretpostaviti rezultat, postoji velika šansa da ćete pogriješiti. Razlog pogreške je vjerojatno taj što niste naviknuti na logičku programsku paradigmu. Navedene linije koda predstavljaju pravila koja se mogu napisati jednadžbama zbog jednostavnijeg shvaćanja. Analizirajmo te linije koda pomoću Tablice 3.

Programski kod	Jednadžba	x
`run(add(x,10,5), x);`	*x +10 = 5*	-5
`run(add(x,5,10), x);`	*x + 5 = 10*	5
`run(add(10,x,5), x);`	*10 + x = 5*	-5
`run(add(10,5,x), x);`	*10 + 5 = x*	15

Tablica 3 Analiza koda napisanog logičkom programskom paradigmom

Napisani programski kod se svodi na vrijednosti varijable *x* unutar četiri vrlo jednostavne jednadžbe. Metoda *add* označava operaciju zbrajanja,a prima *x,y i z* argumente. Odnos tih argumenata se može zapisati kao: *x + y = z.* Jednostavno se metodi *add* mogu proslijediti dva argumenta i pomoću nje pronaći vrijednost trećeg. Da programski kod ne mora biti tako jednostavan pokazuje Kod 15.

```
let logic = require('logicjs');


let   add = logic.add;
let   and = logic.and;
let div = logic.div;
let eq = logic.eq;
let lvar = logic.lvar;
let mul = logic.mul;
let or = logic.or;
let run = logic.run;
```

```
let sub = logic.sub;

let x = lvar();
let  y = lvar();

function mother(x,y) {
      return or(
            and(eq(x,'Ivana'), eq(y,'Ana')),    // Ivana je Anina majka
            and(eq(x,'Ana'), eq(y,'Petra')),    // Ana je Petrina majka
            and(eq(x,'Petra'), eq(y,'Marija')), // Petra je Marijina majka
            and(eq(x,'Marija'), eq(y,'Josipa')) // Marija je Josipina majka
      );
}

function grandmother(x,y) {
      let z = lvar();
      return and(mother(x,z), mother(z,y));
}

// Tko je Anina majka?
console.log(run(mother(x,'Ana'), x)); // rezultat: ['Ivana']

// Tko je kome baka?
console.log(run(grandmother(x,y), [x,y]));
// rezultat: [ ['Ivana', 'Petra'], ['Ana', 'Marija'], ['Petra', 'Josipa'] ]
```

Kod 15 Logička programska paradigma 2

Kod 15 prikazuje program unutar kojega je logičkom programskom paradigmom opisan odnos između ljudi, odnosno tko je kome majka i koje je pravilo da bi netko nekome bio baka. Funkcija *mother* definira tko je kome majka, a funkcija *grandmother* definira pravilo koje mora bit zadovoljeno da bi se osoba *x* smatrala bakom osobe *y*. Na kraju je pozvana metoda *grandmother*

kako bi se ustanovilo tko je sve kome baka. Na prvi pogled je teško za shvatiti programski kod, ali nakon nekog vremena isti kod postaje intuitivan i posve jasan.

3.2. Funkcionalna programska paradigma

Funkcionalna programska paradigma se temelji na jednostavnoj pretpostavci s dalekosežnim implikacijama: konstruiranje programske podrške korištenjem čistih funkcija (*eng. pure functions*), drugim riječima, funkcijama koje nemaju neželjene efekte (*eng. side effects*) [22]. Smatra se da funkcija ima neželjeni efekt ukoliko radi bilo što osim jednostavnog vračanja rezultata.

Neki primjeri neželjenih efekata [22]:

1. Modificiranje varijable
2. Postavljanje polja na objektu
3. Ispisivanje na konzolu ili čitanje s konzole
4. Čitanje iz datoteke ili pisanje u datoteku
5. Crtanje na ekran

Zamislite samo kakvi bi to programi bili kada unutar njih ne bi bilo dozvoljeno raditi nabrojane operacije. Teško je uopće zamisliti. Takvi programi ne bi smjeli sadržavati petlje, bilo kakve podatke koji se mijenjaju, hvatanje pogrešaka, čitanje iz datoteka i slično. Naravno, koristeći funkcionalnu programsku paradigmu je moguće vršiti navede operacije, ali poštujući određena pravila. Izračuni se tretiraju kao evaluacije matematičkih funkcija te se nastoji mijenjanje stanja programa svesti na minimum. Izlazna vrijednost neke funkcije ovisi samo o argumentima koje je funkcija primila [23]. Što znači, neovisno o tome koliko god da se puta pozove neka funkcija, ukoliko joj se proslijedi jedan te isti ulaz, uvijek će se dobiti isti izlaz. Funkcionalna programska paradigma ima podrijetlo u lambda računu, a postepenim razvijanjem je postala moderan i snažan alat za razvoj programske podrške. Prisjetimo se zadatka iz prethodnog poglavlja i riješimo ga koristeći se funkcionalnom programskom paradigmom: „Neka je zadan niz koji sadrži brojeve: 1, 32, 13, 46, 15 , 62, 83, 4, 16 i 980. Potrebno je sve parne brojeve iz danog niza pohraniti u novi niz, novi niz je potrebno uzlazno sortirati i ispisati.“. Kod 16 prikazuje rješenje zadatka napisano koristeći se funkcionalnom programskom paradigmom. Za razliku od prijašnjih rješenja, program koji prikazuje Kod 16 je znatno kraći, a samim time i jednostavniji za pratit.

```
const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]; // niz  brojeva

let even = numbers.filter(number => number % 2 == 0); // niz parnih brojeva

let sorted = even.sort((x, y) => x - y); // sortirani niz

console.log(sorted); // ispis
```

Kod 16 Funkcionalna programska paradigma

Program se sastoji od pet funkcija koje rješavaju zadatak:

1. `Array.prototype.filter()` – stvara novi niz koji sadrži sve elemente danog niza koji su prošli test koji funkcija prima kao argument, a taj test je opet u funkcija.
2. `number => number % 2 == 0` – funkcija koja predstavlja test kojim se provjerava je li broj paran.
3. `Array.prototype.sort()` – sortira niz i vraća ga, prima funkciju koja određuje kriterij za sortiranje.
4. `(x, y) => x - y` - kriterij za uzlazno sortiranje niza.
5. `console.log()` – ispisuje poruku na konzolu.

Zadatak je vrlo jednostavno riješen navedenim funkcijama. U nastavku će se više govoriti o prednostima funkcionalne programske paradigme.

3.2.1. Prednosti funkcionalne programske paradigme

Bitna prednost funkcionalne programske paradigme u odnosu na druge je čistoća programskog koda, što je prethodni primjer djelomično i dokazao. Da bi se dobila bolja slika o tome, Tablica 4 predstavlja usporedbu rješenja napisanog funkcionalnom programskom paradigmom i objektno orijentiranom programskom paradigmom.

Funkcionalna programska paradigma	Objektno orijentirana programska paradigma

`// niz brojeva`
`const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];`
 	
`// niz parnih brojeva`
`let even = numbers.filter(number => number % 2 == 0);`
 	
`let sorted = even.sort((x, y) => x - y); // sortirani niz`
 	
`console.log(sorted); // ispis`
	`class Numbers{`
	`  constructor(values){`
	`  this.values = values;`
	`  }`
	`  toString(){`
	`  console.log(this.values);`
	`  }`
	`}`
	
	`class Services{`
	`  static getEvenNumbers(numbers){ // niz parnih brojeva`
	`  let even = [];`
	`  for (let i = 0; i < numbers.length; i++){`
	`  if(numbers[i] % 2 == 0)`
	`  even.push(numbers[i]);`
	`  }`
	`  return even;`
	`  }`
	`  static sortArray(numbers){ // sortiranje niza`
	`  let sorted = numbers;`
	`  for(let i = 0; i < sorted.length - 1; i++){`
	`  for( let j = i + 1; j < sorted.length; j++){`
	`  if(sorted[i] > sorted[j]){`
	`  let temp = sorted[i];`
	`  sorted[i] = sorted[j];`
	`  sorted[j] = temp;`
	`  }`
	`  }`
	`  }`
	`  return sorted;`
	`  }`
	`}`
	
	`const numbers = new Numbers([1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]);`
	
	`let even = new Numbers();`
	`even.values = Services.getEvenNumbers(numbers.values);`
	`even.values = Services.sortArray(even.values);`
	`even.toString();`

Tablica 4 Usporedba rješenja napisanog funkcionalnom i objektno orijentiranom programskom paradigmom

Program napisan funkcionalnom programskom paradigmom je čišći, jednostavniji i kraći. Navedene prednosti rezultiraju jednostavnijim otklanjanjem pogrešaka, testiranjem i održavanjem programske podrške. Dodatno, funkcionalno programiranje na određeni način forsira dijeljenje kompleksnih problema na manje probleme koji se onda rješavaju [24]. Tako da je modularnost još jedna od prednosti funkcionalne programske paradigme. Kako obično svaki modul ima pomoćne funkcije, modularnost za sobom povlači ponovno upotrebljavanje koda. Na primjer, ako je jedna funkcija riješila problem sortiranja velikog broja ljudi prema njihovom imenu, ista funkcija se

kasnije može iskoristiti za sortiranje gradova prema njihovom nazivu. Dodatno, ovisnost modula o drugim modulima je svedena na minimum. U sustavima koji su pisani striktno objektno orijentiranom programskom paradigmom, interakcija između objekata uzrokuje promjene svojstava objekata što se odražava promjenom stanja programa [25]. S druge strane, sustavi pisani funkcionalnom programskom paradigmom nastoje promjene stanja programa svesti na minimum. Kada se sve navedeno o funkcionalnoj i objektno orijentiranoj programskoj paradigmi uzme u obzir, odnos između dvije paradigme postaje jasniji. Smatram da su navedene paradigme danas najzastupljenije u razvoju programske podrške i koliko god da se razlikuju, ipak nisu suprotnost. Preferiram razvoj programske podrške koristeći se programskim jezikom JavaScript. Kako sam ranije naveo, nije riječ o objektno orijentiranom jeziku, ali podržava pisanje koda objektno orijentiranom i funkcionalnom paradigmom. Smatram da se program najbolje implementira korištenjem kombinacije objektno orijentirane i funkcionalne paradigme. JavaScript se razvija već nekoliko desetljeća, nove biblioteke izlaze na dnevnoj bazi, a i dalje podržava objektno orijentiranu i funkcionalnu programsku paradigmu za što očito postoji razlog. Traženje balansa između objektno orijentirane i funkcionalne programske paradigme nije jednostavan posao. Kao i uvijek, najbitniji je problem koji se rješava, a onda znanje programera i njegove osobne preferencije. Poznavanje različitih programskih paradigmi, programera čini kompetentnijim, a kombinacija programskih paradigmi mu otvara nove mogućnosti u rješavanju problema.

3.2.2. Pisanje funkcija u JavaScriptu

Kako je u ovome radu naglasak na programskom jeziku JavaScript, karakteristike funkcionalne programske paradigme će se pokazati primjerima napisanim u JavaScriptu. Da bi ti primjeri bili razumljivi, potrebno je znati pisati funkcije u JavaScriptu. Bitno je imati na umu da je svaka funkcija u JavaScriptu zapravo *function* objekt [26]. *Function* objekt se stvara *function* konstruktorom. Postoje različiti načini za pisanje i pozivanje funkcija u JavaScriptu. Kod 17 prikazuje uobičajeni način pisanja i pozivanja funkcije.

```
function Hi(name){
  return `Hi ${name}!`;
}
Hi('Luka'); // rezultat: 'Hi Luka!'
```

Kod 17 Funkcija u JavaScriptu

U prethodnom primjeru postoji funkcija `Hi` koja prima ime kao parametar i vraća *string.* Nakon što je funkcija `Hi` definirana, pozvana je i dala je očekivani rezultat. Kod 18 prikazuje anonimnu funkciju koja poziva samu sebe (*eng. anonymous self-invoking function*).

```
(function(){return `Hi ${arguments[0]}!`;})('Petar');
// rezultat: 'Hi Petar!'
```

Kod 18 Anonimna funkcija u JavaScriptu koja poziva samu sebe

Pozivanje funkcije u Kodu 18 počinje automatski bez posebnog navođenja programu da je potrebno izvršiti funkciju. Funkcija je anonimna zato što nema ime, za razliku od funkcije u Kodu 11 čije je ime `Hi`. Cijelu funkciju je potrebno ograditi zagradama kako bi se izrazilo da je riječ o funkciji. Vratimo se na Kod 17, sada će se ista radnja napisati na drugačiji način.

```
let Hi = (name) => { return `Hi ${name}!`; };
Hi('Luka'); // rezultat: 'Hi Luka!'
```

Kod 19 *Arrow* funkcija u JavaScriptu

Pojavom funkcionalne programske paradigme i njene sve veće zastupljenosti u JavaScriptu, pojavila se potreba za jednostavnijim i kraćim zapisom funkcija, što je rezultiralo implementacijom *arrow* funkcija u JavaScriptu. Kod 19 predstavlja *arrow* funkciju koja obavlja isti posao kao i funkcija iz Koda 16. Kako bi zapis *arrow* funkcija bio što kraći, ukoliko funkcija prima samo jedan parametar, onda nije potrebno koristiti zagrade, a ukoliko sadrži samo jednu liniju koja odmah vraća vrijednost, nije potrebno pisati niti zagrade niti ključnu riječ *return*. Kod 20 prikazuje skraćeni zapis funkcije iz Koda 19.

```
let Hi = name => `Hi ${name}`;
Hi('Petar'); // rezultat: 'Hi Petar!'
```

Kod 20 Skraćena *arrow* funkcija

Kao što u kodu 18 postoji anonimna funkcija koja poziva samu sebe, tako Kod 21 prikazuje anonimnu *arrow* funkciju koja poziva samu sebe.

```
(name => `Hi ${name}`)('Luka'); // rezultat: 'Hi Luka!'
```

Kod 21 Anonimna *arrow* funkcija u JavaScriptu koja poziva samu sebe

3.2.3. Čiste funkcije

Čiste funkcije (*eng. pure functions*) vraćaju vrijednost koja je dobivena manipulacijom jedino ulaznih podataka. Drugim riječima, varijable izvan funkcije i globalna stanja programa se ne koriste u čistim funkcijama. Čiste funkcije se koriste jedino za dobivanje vrijednosti koju vraćaju [24]. Matematička interpretacija funkcija se može povezati s čistim funkcijama u programskom inženjerstvu, prati se odnos između ulaznih i izlaznih podataka. Koliko god puta pozvali čistu funkciju, ukoliko joj se uvijek proslijede isti ulazni podaci, uvijek će se dobiti isti izlazni podaci. Kako čiste funkcije ne ovise o globalnim varijablama i stanju programa, jednostavne su za implementaciju i ponovno korištenje. Tablica 5 prikazuje usporedbu funkcije i čiste funkcije.

Funkcija	**Čista funkcija**

`const name = \`Luka\`;`
 	
`let Hi = () => {`
`  console.log(\`Hi ${name}!\`);`
`}`
 	
`Hi();`
`// ispisuje na konzolu: \`Hi Luka!\``
	`let Hi = name => \`Hi ${name}!\`;`
	
	`Hi(\`Petar\`); // rezultat: \`Hi Petar!\``

Tablica 5 Usporedba funkcije i čiste funkcije

Funkcija u lijevom stupcu tablice nije čista funkcija iz dva razloga. Prvi razlog je taj što koristi varijablu `name` koja se nalazi izvan funkcije i tako izravno ovisi o vrijednosti varijable koju ne prima kao parametar. Drugi razlog je što koristi operaciju zapisivanja na konzolu, što se tumači kao mijenjanjem stanja programa. Naravno, u funkcionalnoj programskoj paradigmi je dozvoljeno zapisivati na konzolu, ali nije dobra praksa da se takva operacija izvodi na navedeni način. S druge strane, funkcija u desnom stupcu tablice je čista funkcija. Kao parametar prima `name` te vraća string `` `Hi ${name}!` ``. Nakon pozivanja funkcije, korisnik dobivenu vrijednost može iskoristiti prema potrebama programa.

3.2.4. Funkcije visoke razine

Funkcije visoke razine (*eng. high-order functions*) su funkcije koje ili primaju funkciju kao ulazni argument ili vračaju funkciju kao izlaznu vrijednost [27]. Prisjetimo se, funkcije koje pozivaju same sebe su jedan od primjera funkcija visokog reda [24]. Funkcije visoke razine nisu uobičajene za tradicionalno programiranje. Na primjer, programeri koji koriste imperativnu programsku paradigmu za uređivanje elemenata nekog niza koriste petlju unutar koje manipuliraju elementima niza. S druge strane, programeri koji koriste funkcionalnu programsku paradigmu imaju skroz drugačiji pristup, koriste se funkcijama visoke razine. Tablica 6 prikazuje primjer programa napisan navedenim programskim paradigmama.

Imperativna programska paradigma	**Funkcionalna programska paradigma**

`// niz brojeva`
`const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];`
 	
`let even = []; // niz parnih brojeva`
 	
`// pohranjivanje svih parnih brojeva iz niza numbers u niz even`
`for (let i = 0; i < numbers.length; i++){`
 	
`  if(numbers[i] % 2 == 0)`
`  even.push(numbers[i]);`
`}`
	`// niz brojeva`
	`const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];`
	
	`let even = numbers.filter(number => number % 2 == 0); // niz parnih brojeva`

Tablica 6 Usporedba imperativne i funkcionalne programske paradigme

Očito, cilj programa je sve parne brojeve iz niza `numbers` spremiti u niz `even`. Oba rješenja su dio zadatka koji je riješen i objašnjen u prethodnim poglavljima. Usredotočimo se na funkciju `filter` koja se nalazi u programu napisanim funkcionalnom programskom paradigmom. Riječ je o funkciji visoke razine. `filter` je funkcija visoke razine zato što kao argument prima funkciju `number => number % 2 == 0`. Dodatno, vrača niz parnih brojeva na način da je svaki element niza `numbers` testiran funkcijom `number => number % 2 == 0`, a samo parni brojevi su pohranjeni u niz koji funkcija vrača.

3.2.5. Anonimne funkcije

Kao što samo ime nameće, anonimne funkcije (*eng. anonymous functions*) su funkcije koje nemaju ime, ali one su puno više od toga. Najbolje će se objasniti na primjeru. Kod 22 prikazuje upotrebu anonimne funkcije.

```
function add(x){
  return function(y){
    return x + y;
  }
}
add(3)(4); // rezultat: 7
```

Kod 22 Anonimna funkcija

Kod 22 prikazuje dvije funkcije. Prva funkcija je funkcija `add`, ona vraća funkciju koja je anonimna. Nema potrebe da se imenuje jer se ta funkcija neće ponovno koristiti u programskom kodu. Njena jedina zadaća je da zbroji broj koji je proslijeđen funkciji `add` i broj koji je proslijeđen njoj, te da vrati rezultat. Anonimne funkcije su često dio funkcija visokih razina. Kako je funkcija u JavaScriptu objekt, vrijednost anonimne funkcije se može pridijeliti varijabli te se ista anonimna funkcija može više puta pozvati u kodu.

```
let Hi = name => `Hi ${name}!`;
Hi(`Luka`); // rezultat: `Hi Luka!`
Hi(`Petar`); // rezultat: `Hi Petar!`
```

Kod 23 Anonimna funkcija pohranjena u varijabli

Anonimne funkcije obično otežavaju otklanjanje pogrešaka kod programa, stoga ih treba pametno koristiti. Kao što je pokazano, anonimne funkcije su puno više od lijepog načina zapisivanja programskog koda. Utjelovio ih je Lambda račun [24] koji je nastao puno prije računala i programskih jezika. Zbog toga se anonimne funkcije ponekada nazivaju lambda izrazima [24].

3.2.6. Ulančavanje funkcija

Ulančavanje funkcija u JavaScriptu je uobičajena praksa i vrlo efikasna metoda za pisanje preglednog programskog koda. Riječ je o jednostavnoj metodi pojednostavljivanja programskog koda kada je potrebno nekoliko funkcija zaredom primijeniti nad objektom. Kod 24 prikazuje primjer ulančavanja funkcija.

```
const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];
const changed = numbers.reverse().concat([32, 17]).map(number => number * 2);
// [ 1960, 32, 8, 166, 124, 30, 92, 26, 64, 2, 64, 34 ]
```

Kod 24 Ulančavanje funkcija

Kako je u JavaScriptu sve objekt, navedeni način ulančavanja funkcija se može postići samo kada su funkcije metode objekta nad kojim se izvršavaju. Ako programer želi ulančavati svoje funkcije, onda ih mora deklarirati kao metode objekta nad kojim ih želi izvršiti.

```
Array.prototype.plusOne = Array.prototype.plusOne || function(){
  return this.map(element => isNaN(element) ? element : element + 1);
};
const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980];
const changed = numbers.reverse().concat([32, 17]).plusOne();
// [ 981, 17, 5, 84, 63, 16, 47, 14, 33, 2, 33, 18 ]
```

Kod 25 Pisanje vlastite funkcije za ulančavanje

Kod 25 prikazuje primjer kako se piše vlastita metoda za ulančavanje. Primjer prikazuje dodavanje funkcije `plusOne` nizovima. Dobra praksa je prvo provjeriti postoji li takva funkcija već ugrađena, tako da se ne premosti ako već postoji. Ukoliko takva funkcija ne postoji, izvršit će se korisnikova funkcija. Napisana funkcija je vrlo jednostavno, njena zadaća je da svaki element niza koji je tipa `Number` uveća za 1. Slično, programer može pisati funkcije kakve god želi, ali pri tome treba paziti kako će ih organizirati i primijeniti.

3.2.7. Currying

Currying je proces preoblikovanja funkcije koja prima više argumenata u funkciju koja prima jedan argument, a vraća drugu funkciju koja prima više argumenata prema potrebi [24]. Na primjer, funkcija koja prima *N* argumenata se može preoblikovati u *N* ulančanih funkcija od kojih svaka prima po jedan argument. Prednost curryinga je dobra preglednost i kontrola argumenata koji se prosljeđuju funkcijama. Kod 26 prikazuje primjer currya.

```
let curry = greeting => name => `${greeting} ${name}!`;
curry('Hi')('Petar'); // rezultat: `Hi Petar!`
```

Kod 26 Curry

Curry nije dobro primjenjivati ukoliko broj argumenata koje neka funkcija treba primiti nije poznat ili može varirati. U tom slučaju je teško implementirati curry te programski kod postaje nepregledan, a samim time i težak za otklanjanje pogrešaka, testiranje i održavanje. Curry je moguće koristiti s lijeva i s desna. Što se pod time misli, jednostavno će se demonstrirati primjerom.

```
let leftCurry = x => y => x/y;
let rightCurry = x => y => y/x;


leftCurry(5)(10); // rezultat: 0.5
rightCurry(5)(10); // rezultat: 2
```

Kod 27 Curry s lijeva i s desna

Primjer prikazuje dvije jednostavne funkcije napisane u obliku curry, a zadaća im je podijeliti dva broja. Funkcija `leftCurry` prikazuje pisanje currya s lijeva, dok funkcija `rightCurry` prikazuje pisanje currya s desna. Poanta je da argumenti nisu asocijativni i da treba paziti kako se pišu takve funkcije, kao što treba paziti i na redoslijed prosljeđivanja argumenata. Prethodno je obrađeno dosta metoda i načina za pisanje i korištenje funkcija u JavaScriptu što je dovoljno za početak učenja i savladavanja funkcionalne programske paradigme.

3.3. Programske paradigme u bazama podatka

Još jedna u nizu zanimljivih programskih paradigmi je ona koja se koristi za pisanje upita nad bazama podataka (*eng. database programming paradigm*). Riječ je o podvrsti deklarativne programske paradigme. Štoviše, smatra se da su relacijske baze podataka preteča deklarativne programske paradigme. Kontrola toka kod upita nad bazama podataka je implicitna, programer koristeći odgovarajući programski jezik navodi kako rezultat treba izgledati, iako ne navodi kako doći do takvog rezultata. Kako je u ovome radu naglasak na programskom jeziku JavaScript, koristit će se primjeri upita pisanih u JavaScriptu nad MongoDB bazom podataka [28]. MongoDB je baza podataka koja se temelji na dokumentima (*eng. document database*). Drugim riječima, svaki zapis u bazi podataka je dokument koji se sastoji polja i njihovih vrijednosti. Dokumenti su slični JSON objektima koji su standardni u JavaScriptu. Ono što je ovdje zanimljivo je kako zapravo izgledaju upiti nad takvom bazom podataka. Kod 28 pokazuje primjer upita nad MongoDB bazom podataka.

```
USER.findOne({'_id': '24cd0g2u864k62710f9b0f2'},function(error,user){
      ...
}
```

Kod 28 MongoDB upit

Iz upita je vidljivo da se upit vrši nad kolekcijom `USER`, te je s `findOne` naglašeno da se traži točno jedan zapis, čiji je `_id` jednak `'24cd0g2u864k62710f9b0f2'`. Kao što je i prethodno rečeno, naglašeno je kako željeni rezultat treba izgledati, ali ne i kako doći do tog rezultata. Upit sadrži funkciju koja specificira što će se dogoditi s dobivenim zapisom, ukoliko dođe do pogreške prilikom izvođenja upita, detalji o pogrešci će bit zapisani u objektu `error`, a ako se upit uspješno izvrši, onda će rezultat bit zapisan u objektu `user`. Ovisno o uspjehu izvršavanja upita, programer može odlučiti što će se dalje izvršavati u programu.

3.4. Pregled deklarativne, logičke, funkcionalne i database programske paradigme

Prisjetimo se programskog koda napisano u deklarativnoj, logičkoj, funkcionalnoj i database programskoj paradigme. Tablica 2 prikazuje programski kod napisan u navedenim paradigmama.

Deklarativna programska paradigma

```
describe('/POST book', () => {

 it('it should POST a book ', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954,
            pages: 1170
        }
        chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');

res.body.should.have.property('message').eql('Book
successfully added!');
                res.body.book.should.have.property('title');
                res.body.book.should.have.property('author');
                res.body.book.should.have.property('pages');
                res.body.book.should.have.property('year');
              done
            });
      });
  });
```

Logička programska paradigma

```
let logic = require('logicjs');

let add = logic.add;
let and = logic.and;
let div = logic.div;
let eq = logic.eq;
let lvar = logic.lvar;
let mul = logic.mul;
let or = logic.or;
let run = logic.run;
let sub = logic.sub;

let x = lvar();

run(add(x,10,5), x); // rezultat: [ -5 ]
run(add(x,5,10), x); // rezultat: [ 5 ]
run(add(10,x,5), x); // rezultat: [ -5 ]

run(add(10,5,x), x); // rezultat: [ 15 ]
```

Funkcionalna programska paradigma

```
const numbers = [1, 32, 13, 46, 15 , 62, 83, 4, 16, 980]; //
niz  brojeva

let even = numbers.filter(number => number % 2 == 0); // niz
parnih brojeva

let sorted = even.sort((x, y) => x - y); // sortirani niz

console.log(sorted); // ispis
```

Database programska paradigma

```
USER.findOne({'_id':
'24cd0g2u864k62710f9b0f2'},function(error,user){
            ...
}
```

Kod 29 Pregled deklarativne, logičke, funkcionalne i database programske paradigme

Na prvi pogled se vidi sličnost između navedenih programskih paradigmi. Kako je riječ o deklarativnom programiranju, unutar svakog programskog koda je navedeno kako rješenje treba izgleda bez da je korak po korak navedeno kako će se doći da željenog rješenja. Deklarativno programiranje definitivno ima svrhu u svakodnevnom životu programera. Kako vrijeme odmiče, funkcionalna programska paradigma je sve zastupljenija u implementaciji programske podrške. Dodatno, upiti nad bazama podataka su već dugi niz desetljeća sastavni dio programskog koda skoro svih kompleksnijih aplikacija. S druge strane, logička programska paradigma nije pretjerano zastupljena u razvijanju programske podrške, ali svakako ima svoje prednosti u rješavanju specifičnih problema. Kako je riječ o potpuno drugačijem stilu programiranja od imperativnog,

potrebno je uložiti malo više truda u učenje kako bi se programi mogli kvalitetno implementirati nekom vrstom deklarativne programske paradigme. Učenjem programskih paradigmi se postaje bolji programer te se otvaraju nove mogućnosti u rješavanju različitih problema.

4. Odabir programske paradigme

Prethodno su objašnjene imperativna, proceduralna, objektno orijentirana, deklarativna, logička, funkcionalna i database programska paradigma. Nabrojane programske paradigme su samo jedan dio programskih paradigmi koje postoje. Izbor programske paradigme je ponekad težak zadatak. Prije svega, potrebno je uzeti u obzir problem koji se treba riješiti. Logično je da različiti problemi zahtijevaju različite pristupe razmišljanja, a samim time i različit programski kod. Dobrim odabirom programske paradigme se može itekako utjecati na kvalitetu programskog koda što kasnije rezultira jednostavnijim otklanjanjem pogrešaka, testiranjem i održavanjem. Ponekad se neka programska paradigma čine idealnom za rješavanje određenog dijela problema. Odnosno, idealan izbor za pisanje jednog dijela programa. S druge strane, ta ista programska paradigma ne mora bit idealan izbor za drugi dio programa. Samim time nema potrebe forsirano pisati program od nekoliko desetaka tisuća linija jednom te istom programskom paradigmom. Dakako, puno više se može postići kombinacijom više programskih paradigmi. Neki takve programe nazivaju hibridnima, neki jednostavno kažu da je riječ o programu čiji kod sadrži više programskih paradigmi. Ukoliko se teorija ostavi sa strane, programski kod je samo oruđe za rješavanje problema te mu je tako potrebno i pristupiti. Mnogi jezici podržavaju pisanje koda koristeći se više programskih paradigmi. Kao što je to ovaj rad pokazao, očiti primjer takvog jezika je JavaScript. Riječ je o programskom jeziku koji ima mnoštvo biblioteka koje olakšavaju pisanje koda u funkcionalnoj i objektno orijentiranoj paradigmi. Nema razloga da se ne iskoriste obje mogućnosti. Veliki utjecaj na izbor paradigme ima i iskustvo programera. Neki se ugodnije osjećaju u poznatoj okolini te se ne žele izlagati promjenama. Savladavanje programskih paradigmi uvelike pojednostavljuju kasnije savladavanje bilo kojeg programskog jezika. Programiranje je dosta dinamična disciplina, dolazi do novih otkrića i novih tehnologija gotovo pa svaki dan. Što više programskih jezika programer poznaje, to je kompetentniji za rad. Nema potrebe učiti što više programskih jezika da bi bili bolji u svojoj struci. Bolje je uložiti vrijeme u savladavanje programskih paradigmi koje će kasnije pomoći prilikom učenja bilo kojeg jezika prema potrebi. Današnja situacija je takva da treba prihvatiti kombiniranje različitih programskih paradigmi te pomoću njih pokušati riješiti problem na što bolji način.

Zaključak

U prethodnim poglavljima je objašnjen pojam programskih paradigmi te su neke od njih detaljno opisane. Očito je da svaka spomenuta programska paradigma ima svoje prednosti i mane. Sukladno tome, izbor programske paradigme treba prilagoditi problemu a ne obrnuto. Za objašnjenja programskih paradigmi se koristio programski kod napisan u programskom jeziku JavaScript, ali se isti princip može primijeniti i u drugim programskim jezicima. Prethodna rečenica zapravo i pokazuje važnost programskih paradigmi, tj. govori o tome kako poznavanje određene programske paradigme ne ovisi direktno o programskom jeziku nego samo objašnjava stil ili način pisanja programskog koda. Isječci programskog koda iz prethodnih poglavlja mogu služiti kao dobar početak za savladavanje različitih programskih paradigmi pišući kod u programskom jeziku JavaScript. Nakon savladavanja pojedine programske paradigme, programer se jednostavno može koristiti istim principima u bilo kojem drugom programskom jeziku koji podržava pisanje programskog koda u toj paradigmi. Prilikom razvoja programske podrške potrebno je izabrati programsku paradigmu koja odgovara problemu kojega se želi riješiti. Također, kao što je i ranije spomenuto, veliku ulogu u izboru programske paradigme imaju i iskustvo i preferencije programera. Dobra praksa prilikom razvoja kompleksne programske podrške je koristiti više programskih paradigmi. Na taj način se može pojednostaviti razvoj, testiranje i održavanje. Savladavanje programskih paradigmi ima veliki utjecaj u kasnijem savladavanju pojedinog programskog jezika. Dodatno, poznavanje programskih paradigmi pomaže programeru da ostane kompetentan na tržištu.

Slike

Tablice

Kod

Literatura

[1] Loyola Marymount University, Los Angeles, »Programming Paradigms,« Loyola Marymount University, Los Angeles, 26 4 2017. [Mrežno]. Available: http://cs.lmu.edu/~ray/notes/paradigms/. [Pokušaj pristupa 26 4 2016].

[2] P. V. Roy, »Programming Paradigms for Dummies: What Every Programmer Should Know,« p. 39.

[3] S. M. Maurizio Gabbrielli, Programming Languages: Principles and Paradigms, London: Springer, 2010.

[4] Progopedia, »Paradigm: Procedural,« [Mrežno]. Available: http://progopedia.com/paradigm/procedural/. [Pokušaj pristupa 27 4 2017].

[5] Wikipedia, »Imperative programming,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Imperative_programming. [Pokušaj pristupa 27 4 2017].

[6] Wikipedia, »Procedural programming,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Procedural_programming. [Pokušaj pristupa 4 28 2017].

[7] E. Balagurusamy, Object Oriented Programming with C++, 2013.

[8] Wikipedia, »Object-oriented programming,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Object-oriented_programming. [Pokušaj pristupa 29 4 2017].

[9] S. Stefanov, Object-Oriented JavaScript, Birmingham: Packt Publishing, 2008.

[10] Wikipedia, »Class (computer programming),« [Mrežno]. Available: https://en.wikipedia.org/wiki/Class_(computer_programming). [Pokušaj pristupa 29 4 2017].

[11] Wikipedia, »Polymorphism (computer science),« [Mrežno]. Available: https://en.wikipedia.org/wiki/Polymorphism_(computer_science). [Pokušaj pristupa 29 4 2017].

[12] ECMAScript 6, »New Features:Overview & Comparison,« [Mrežno]. Available: http://es6-features.org. [Pokušaj pristupa 29 4 2017].

[13] JSON, »Introducing JSON,« [Mrežno]. Available: http://www.json.org/. [Pokušaj pristupa 29 4 2017].

[14] Wikipedia, »Parallel computing,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Parallel_computing. [Pokušaj pristupa 19 5 2017].

[15] Wikipedia, »Distributed computing,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Distributed_computing. [Pokušaj pristupa 19 5 2017].

[16] »Parallel.js Easy multi-core processing with javascript,« [Mrežno]. Available: https://parallel.js.org/. [Pokušaj pristupa 19 5 2017].

[17] Wikipedia, »Declarative programming,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Declarative_programming. [Pokušaj pristupa 29 4 2017].

[18] Chai Assertion Library, »JavaScript testing framework,« [Mrežno]. Available: http://chaijs.com/. [Pokušaj pristupa 29 4 2017].

[19] Wikipedia, »Logic programming,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Logic_programming. [Pokušaj pristupa 30 4 2017].

[20] GitHub, »LogicJS,« [Mrežno]. Available: https://github.com/mcsoto/LogicJS. [Pokušaj pristupa 30 4 2017].

[21] Node.js, »Node.js,« [Mrežno]. Available: https://nodejs.org/en/. [Pokušaj pristupa 30 4 2017].

[22] R. B. Paul Chiusano, Functional Programming in Scala, Shelter Island: Manning Publications Co., 2015.

[23] Wikipedia, »Functional programming,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Functional_programming. [Pokušaj pristupa 30 4 2017].

[24] D. Mantyla, Functional Programming in JavaScript, Birmingham - Mumbai: Packt Publishing, 2015.

[25] M. Fogus, Functional JavaScript, Sebastopol: O'Reilly Media, Inc., 2013.

[26] Mozilla Developer Network, »Function,« Mozilla, [Mrežno]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function. [Pokušaj pristupa 1 5 2017].

[27] Wikipedia, »High-order function,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Higher-order_function. [Pokušaj pristupa 2 5 2017].

[28] MongoDB, Inc., »MongoDB Documentation,« [Mrežno]. Available: https://docs.mongodb.com/. [Pokušaj pristupa 2 5 2017].

Sažetak

Programska paradigma je stil ili način programiranja. Općenito, paradigma je model prema kojem se nešto stvara, radi ili izvršava, pa se tako može razmatrati i programiranje. Na najvišoj razini, programske paradigme se dijele na imperativne i deklarativne. Poznavanje programskih paradigmi pojednostavljuje savladavanje novih programskih jezika. Izbor programske paradigme ovisi o problemu kojega se želi riješiti te o iskustvu i preferencijama programera.

Summary

Programming paradigm is a style of programming. In general, the paradigm is a model according to which something can be created or executed, so programming can also be considered. Programming paradigms are divided into imperative and declarative at the highest level. Knowledge of programming paradigms simplifies learning of new programming languages. Choice of the programming paradigm depends on the problem that is to be solved and the experience and preferences of a programmer.