OD MODELA DO KODA: PRAKTIČNA IMPLEMENTACIJA PRELASKA SA OBJEKTNO-ORIJENTISANOG NA FUNKCIONALNO PROGRAMIRANJE

Tanja Krunić [1]

Rezime: U okviru rada je predstavljen pristup predavanju funkcionalne programske paradigme studentima koji su prethodno savladali objektno-orijentisanu paradigmu, i već duže vremena je aktivno koriste. Ideja rada se oslanja na pristup predavanju funkcionalne programske paradigme predstavljen u prethodnom radu autora [1], gde se predlaže da se nakon izlaganja osnovnih principa funkcionalnog programiranja studentima prikažu UML modeli rešenja nekog problema iz realnog života primenom objektno-orijentisane i funkcionalne paradigme, kako bi uočili osnovne razlike u konceptu rešenja. Ovaj rad se bavi predstavljanjem razlike u kodiranju prilikom rešavanja pomenutog problema. Studentima se takođe predstavljaju rešenja dobijena primenom obe programske paradigme, kako bi uočili razlike u detaljima, i na taj način lakše usvojili novu programsku paradigmu.

Ključne reči: programske paradigme, objektno-orijentisano programiranje, funkcionalno programiranje, tehnike predavanja programskih paradigmi, JavaScript, Node.js

FROM MODEL TO CODE: PRACTICAL IMPLEMENTATION OF THE TRANSITION FROM OBJECT-ORIENTED TO FUNCTIONAL PROGRAMMING

Abstract: This paper presents an approach to teaching the functional programming paradigm to students who have previously mastered the object-oriented paradigm and have been actively using it for a longer period of time. The idea of the paper builds upon the approach to teaching functional programming presented in the authors' previous work [1], where it is proposed that, after introducing the basic principles of functional programming, students are shown UML models of a real-life problem solved using both object-oriented and functional paradigms, in order to highlight the fundamental conceptual differences between the solutions. This paper focuses on presenting the differences in coding when solving the aforementioned problem. Students are also presented with implementations using both programming paradigms to help them observe the differences in detail and thereby more easily adopt the new programming paradigm.

Key words: programming paradigms, object-oriented programming, functional programming, teaching techniques for programming paradigms, JavaScript, Node.js

1. UVOD

Na mnogim visokoškolskim ustanovama, kako u Republici Srbiji, tako i u inostranstvu, u okviru studijskih programa na kojima se izučavaju informacione tehnologije, studenti uglavnom na osnovnim studijama izučavaju objektno-orijentisanu programsku paradigmu, pa tek onda na višim godinama studija, ili u okviru master studijskih programa, se upoznaju sa funkcionalnom programskom paradigmom. U prethodnom istraživanju autora [1] se navode primeri ovakve prakse iz Visoke tehničke škole strukovnih studija u Novom Sadu (VTSNS), Fakulteta tehničkih nauka i Prirodno-matematičkog fakulteta u Novom Sadu, kao i sa dve insotrane visoke škole: ZHAW School of Engineering, Winterthur (Švajcarska) i Vorarlberg University of Applied Sciences (Austrija).

Kada studenti savladaju programiranje u jednoj programskoj paradigmi, koju potom više godina koriste, prelaz na drugu programsku paradigmu može biti izazov jer zahteva promenu načina na koji programer razmišlja prilikom rešavanja problema, [2]. Na primeru prelaska sa objektno orijentisanog-programiranja na funkcionalno programiranje, programerima je najteže da se naviknu na činjenicu da nema klasa, objekata, promenljivih koje menjaju vrednost, petlji i dr. Stoga u fazi privikavanja na novu programsku paradigmu rešavanje jednostavnih problema može trajati znatno duže nego inače.

[1]dr, Visoka tehnička škola strukovnih studija u Novom Sadu, Školska 1, e-mail: krunic@vtsns.edu.rs

Takođe, u procesu navikavanja na novu programsku paradigmu, programeri mogu osećati frustracije, jer im za pisanje koda treba mnogo više vremena nego što su navikli, [3]. U [4], radi lakšeg prelaska sa objektno-orijentisanog programiranja na funkcionalno, predlaže se programerima da se najpre spuste na nivo proceduralnog programiranja (koje je mnogo sličnije funkcionalnom programiranju nego objektno-orijentisano programiranje), dodajući mu postepeno osobine funkcionalnog programiranja kao što su čiste funckije, nepromenljivost i dr.

U nameri da se studentima master studija Informacionih tehnologija u VTSNS olakša prelazak sa objektno-orijentisanog programiranja na funkcionalno, u [1] je predstavljen pristup predavanju funkcionalnog programiranja koji omogućava lakši prelazak sa objektno-orijentisane na funkcionalnu paradigmu. Pomenuti pristup podrazumeva da se nakon izloženih pravila koje čine funckionalnu programsku paradigmu, studentima predoči problem iz realnog života, koji se potom rešava najpre primenom objektno-orijentisane paradigme kojom studenti dobro vladaju, a potom koristeći funkcionalnu paradigmu, kako bi studenti, kroz diskusiju uočili razlike. Da bi studenti bili oslobođeni pritiska da se odmah udube u kod, rešenje se najpre prezentuje primenom odgovarajućih UML dijagrama. Na taj način studenti lako mogu da uoče ključne razlike između ove dve programske paradigme. Detalje potom mogu uočiti kada pomenuta rešenja budu predstavljena putem programskog koda. Fokus ovog rada jeste u razlikama u programskom kodu objektno-orijentisanog i funckionalnog rešenja problema čiji modeli su prikazani primenom UML dijagrama u [1]. Radi lakšeg praćenja koda, za rešavanje problema je odabran programski jezik JavaScript, koji podržava obe programske paradigme, [5][6].

U drugom poglavlju su ukratko opisane glavne osobine objektno-orijentisane i funkcionalne programske paradigme. U trećem poglavju je ukratko objašnjen problem koji je studentima predočen, kao i deo programskog koda rešenja ovog problema primenom pomenute paradigme, sa osvrtom na osnovne razlike u pristupu rešavanju pomenutog problema primenom ovih programskih paradigmi.

2. PROGRAMSKE PARADIGME

Programske paradigme se mogu razumeti kao stil, model ili metodologija programiranja, koji ukazuju na najbolji način za rešavanje problema korišćenjem datog programskog jezika, [7][8]. Programska paradigma se određuje još pre samog procesa pisanja koda u programskom jeziku. Ona dolazi do izražaja tokom faze analize, kada se koristi za modelovanje problema i pronalaženje rešenja na specifičan način, [9]. Postoji mnogo programskih paradigmi, kao što su proceduralno, imperativno, objektno-orijentisano, funkcionalno i logičko programiranje (detaljan spisak se može videti u [7]). Međutim, u ovom radu fokus je na postepenom prelasku sa objektno-orijentisanog na funkcionalno programiranje, te su u nastavku navedene isključivo karakteristike ove dve programske paradigme.

2.1. Objektno-orijentisana programska paradigma

U objektno-orijentisanom programiranju, osnovni elementi su klase koje definišu strukturu i ponašanje, i objekti koji predstavljaju njihove instance. Drugim rečima, klase možemo zamisliti kao neku vrstu šablona za objekte. U njima se definišu atributi i metode koje će sadržati objekti, kao i tzv. konstruktori koji služe za kreiranje konkretnih objekata. U osnovne koncepte objektno orijentisanog programiranja spadaju [9][10]:

- Enkapsulacija – Predstvalja sakrivanje unutrašnjih detalja objekta i zaštita podataka tako da se pristupa samo preko definisanih metoda. Ovo se postiže kroz razne nivoe pristupa članovima klase (atributima, metodama). Nivoi pristupa mogu biti:
    - Javni (public) – Atribut ili metoda koji imaju ovu odrednicu su dostupni kako unutar, tako i izvan klase.
    - Zaštićeni (protected) – Članovi koji imaju ovu odrednicu su dostupni isključivo unutar klase ili njenih izvedenih klasa.
    - Privatni (private) – Privatni članovi su dostupni iskjlučivo unutar klase u kojoj su definisani.

- Apstrakcija – Omogućava fokusiranje na važne karakteristike objekta, dok se detalji implementacije skrivaju.
- Nasleđivanje – Omogućava se da iz jedne klase koju često nazivamo osnovnom, roditeljskom ili superklasom izvedemo klasu (izvedena klasa, klasa dete, podklasa) koja nasleđuje atribute i metode od osnovne klase koje se po potrebi mogu redefinisati (*engl. override*), ali može imati i neke dodatne, vlastite atribute i metode. Time se omogućava ponovna upotreba koda, što obezbeđuje kraći, jasniji i pregledniji kod.
- Polimorfizam – Omogućava da se nasleđene metode redefinišu u izvedenoj klasi. Samim tim se metoda sa istim nazivom može ponašati drugačije u zavisnosti od instance klase koja je poziva.

Klase mogu biti u međusobnim relacijama, kao što su [11][12][13]:

- Asocijacija predstavlja strukturnu vezu između dve klase, pri čemu jedna klasa sadrži referencu (atribut) koji ukazuje na instancu druge klase. Objekti tih klasa mogu postojati nezavisno jedni od drugih.
- Agregacija je relacija između dve klase koja se zasniva na tome da jedna klasa sadrži drugu. Često ovakvu relaciju označavamo sa 'has-a', tj. jedna klasa 'ima' drugu. Pri tome životni vek jedne klase nema uticaja na životni vek druge klase.
- Kompozicija je jača relacija od agregacije, u kojoj objekat jedne klase sadrži objekat druge klase, ali sadržani objekat ne može postojati nezavisno od objekta koji ga sadrži – zavisi od njegovog životnog veka.

2.2. Funkcionalna programska paradigma

U funkcionalnom programiranju, problemi se rešavaju pomoću funkcija. Jedan od osnovnih principa funkcionalnog programiranja jeste korišćenje čistih funkcija. Pod čistim funkcijama podrazumevaju se funkcije koje za iste vrednosti ulaznih parametara uvek vraćaju iste povratne vrednosti. To znači da zavise isključivo od svojih ulaznih parametara i nemaju sporedne efekte. Još jedan od osnovnih principa funkcionalnog programiranja jeste korišćenje nepromenljivih varijabli. Varijable se dodavanjem prefiksa *const* mogu pretvoriti u nepromenljive varijable. Ovo u stvari čini kod znatno jednostavnijim za praćenje. Naime, ukoliko se koriste promenljive varijable, nakon izmene u kodu mora se ispratiti svaka funkcija koja menja tu varijablu, jer u suprotnom može doći do greške.

U funkcionalnom programiranju, funkcije mogu biti ulazni parametri (argumenti) ili povratne vrednosti drugih funkcija. Funkcije koje primaju druge funkcije kao parametre ili ih vraćaju kao rezultat, zovu se funkcije višeg reda. Funkcija koja se prosleđuje kao argument drugoj funkciji, i namenjena je da bude pozvana kasnije — bilo odmah ili nakon što se neki događaj ili operacija završi se naziva callback funckija. Funkcije se takođe mogu dodeljivati varijablama [14] [15]. Funkcije višeg reda se mogu koristiti za rad sa nizovima, bez korišćenja petlji, što je takođe praksa u funkcionalnom programiranju [16]. U JavaScriptu se koriste funckije kao što su *foreach(), map(), filter(), reduce(), some(), any(), sort(),* i dr. Ove ili slične funkcije postoje i u drugim programskim jezicima koji podržavaju funkcionalnu paradigmu. Generalni način za izbegavanje primene petlji u kodu je korišenje rekurzija – funkcija koje pozivaju samu sebe radi rešavanja nekog problema [17].

3. REŠAVANJE PRAKTIČNOG PROBLEMA PRIMENOM DVE RAZLIČITE PROGRAMSKE PARADIGME I NJIHOVO MEĐUSOBNO POREĐENJE

Kao što je navedeno u uvodu, osnovna ideja izložena u [1] i ovom radu je da se studentima olakša prelazak sa objektno-orijentisane na funkcionalnu programsku paradigmu. U tu svrhu u [1] je prikazan pristup predavanjima funkcionalne paradigme na način da se nakon što se izlože osnovna pravila funkcionalnog programiranja, studentima predoči praktičan problem za koji se najpre prikazuje UML model rešenja za objektno-orijentisanu paradigmu koja je studentima dobro poznata, a potom primenom funkcionalne paradigme. Poređenjem ova dva modela, studentima je omogućeno da uoče osnovne razlike u konceptima rešenja, bez kodiranja. Nakon što su prihvatili konceptualne razlike ove dve paradigme, ključne detalje mogu savladati kodiranjem rešenja prema prikazanom modelu. U

nastavku sledi najpre opis problema koji se rešava, a potom i delovi koda karakteristični za svaku od dveju programskih paradigmi.

3.1. Opis problema i metodologija rešavanja

Praktični problem koji studenti rešavaju, a čiji su modeli prethodno dobijeni primenom objektno-orijentisane i funkcionalne paradigme su prikazani u [1], je zapravo pojednostavljeni scenario sa kojim se studenti susreću tokom studija. Studenti imaju ime i prezime, broj indeksa, studijski program na koji su upisani, kao i studijsku godinu. Prilikom upisa u školsku godinu, biraju predmete za slušanje, pri čemu svaki predmet ima ime, određen broj ESPB bodova, pripada određenom studijskom programu i studijskoj godini. Od izabranih predmeta za slušanje, formira se lista predmeta za slušanje. Studenti imaju pravo da se ispišu sa predmeta koji su izabrali za slušanje. Drugim rečima, tada se navedeni predmet briše sa spiska predmeta za slušanje. Predmete koje su studenti izabrali za slušanje, mogu i polagati. U tom smislu postoje rezultati predispitnih obaveza i ispita, kao i prosečna ocena svih položenih ispita. Jednostavnosti radi, uvedena je pretpostavka da na svim predmetima u sklopu predispitnih obaveza postoji samo po jedan kolokvijum. Da bi se kolokvijum smatrao položenim, neophodno je da student osvoji 30-50 poena, dok je za polaganje završnog ispita neophodno 21-50 poena. Stoga je za polaganje ispita sa najnižom ocenom 6 minimalno potrebno osvojiti ukupno 51 poen, pri čemu se ponei sa kolokvijuma i ispita sabiraju. Za svaku narednu ocenu, neohodno je osvojiti dodatnih 10 poena. Pozitivne ocene na ispitu se kreću u opsegu 6-10.

Rešenje gore prikazanog problema koje se u okviru predavanja studentima prikazuje primenom objektno-orijentisane i funkcionalne paradigme se izvodi u programskom jeziku JavaScript, u Node.js okruženju. Programski jezik JavaScript je izabran jer podržava obe programske paradigme, ali takođe iz razloga što je jedan od trenutno najtraženijih programskih jezika na tržištu rada [18], i kao takav je zastupljen u većem broju predmeta na master studijama Informacionih tehnologija u VTSNS [19], a sa osnovama ovog programskog jezika studenti se susreću još na osnovnim studijama, [20]. Zbog ograničenja obima rada, u nastavku su prikazani samo neki karakteristični fragmenti koda.

3.2. Rešavanje problema primenom objektno-orijentisane paradigme

Objektno-orijentisani kod se satoji iz klasa *Test, Colloquium, Exam, Subject, Subject* i *Student.* Klasa *Test* (Listing 1) u stvari predstavlja šablon za kreiranje klasa *Colloquium* (Listing 2) i *Exam* koje je nasleđuju. Iz koda koji su prikazani u Listingu 1 i 2 vidimo da je metoda *attemptPass*() redefinisana u izvedenoj klasi *Colloquium.* S obzirom na činjenicu da se na ispitu dobija konačna ocena, klasa *Exam* mora sadržati i sopstvenu metodu za računanje ocene, *calculateMark().* Implementacija klase *Exam* nije prikazana u radu, već je prepuštena čitaocu. Drugim rečima, u kodu možemo videti kreiranje klasa, konstruktore, nasleđivanje, kao i redefinisanje nasleđenih metoda.

```
// Baza za zajedničke karakteristike i metode za Kolokvijum i Ispit
class Test {
   constructor(subject, points) {
      this.subject = subject;
      this.points = points;
      this.passed = false; // Status da li je test položen}

   // Osnovna metoda za pokušaj polaganja testa
   attemptPass(minPointsForPass) {
      this.passed = this.points >= minPointsForPass;
      return this.passed;}

   // Prikaz statusa testa
   displayResult() {
      return this.passed ? "Passed" : "Failed";
      }
}
```

Listing 1 – Klasa Test

```
// Klasa Kolokvijum koja nasleđuje Test
class Colloquium extends Test {
   constructor(subject, points) {
      super(subject, points);  // Pozivanje roditeljske klase
   }
```

```
  // Kolokvijum se polaže ako je ostvareno između 30 i 50 poena
    attemptPass() {
      return super.attemptPass(30);  // Kolokvijum se polaže sa minimalno 30 poena
    }
}
```

Listing 2 – Klasa Colloquium

Listing 3 i Listing 4 prikazuju redom klase *Subject* i *Subjects*. Ove dve klase su povezane relacijom agregacije. Kao što vidimo, klasa *Subjects* sadrži objekte klase *Subject* u svojoj listi *this.subjects*. Međutim, objekti klase *Subject* mogu postojati potpuno nezavisno od instance klase *Subjects*. Na primer, može se kreirati novi objekat naredbom *new Subject("Funkcionalno programiranje", 10, 2, "Informacione tehnologije")*, i on će postojati bez dodavanja u listu klase *Subjects*. Takođe, ako se instanca klase *Subjects* obriše, to ne znači da će i svi objekti klase *Subject* koje je sadržavala biti automatski uništeni. Ti objekti klase *Subject* mogu i dalje postojati ako su referencirani negde drugde u kodu (npr. u *Student.selectedSubjects* listi koja je deo klase Student i čuva podatke o listi predmeta koje je studnet izabrao. Ova klasa nije prikazana u radu, zbog ograničenog obima rada).

```
class Subject {
    constructor(name, ESPB, year, studyProg) {
        this.name = name;
        this.ESPB = ESPB;
        this.year = year;
        this.studyProg = studyProg;
    }

    displaySubjectInfo() {
        return `${this.name} (${this.ESPB} ESPB) - Year: ${this.year}, Study Program: ${this.studyProg}`;
    }
}
```

Listing 3 – Klasa Subject

```
class Subjects {
    constructor() {
        this.subjects = [];
    }

    addSubject(subject) {
        this.subjects.push(subject);
    }

    findSubject(name) {
        return this.subjects.find(subject => subject.name === name);
    }
}
```

Listing 4 – Klasa Subjects

3.3. Rešavanja problema primenom funkcionalne paradigme

Kod koji rešava problem izložen u poglavlju 3.1. funkcionalnom paradigmom se sastoji od skupa funkcija (*createTest, createColloquium, createExam, createSubject, createSubjects, createStudent)*. Ove funkcije se uglavnom čuvaju unutar nepromenljivih varijabli, što možemo videti na primeru varijable *createTest* u Listingu 5. Deo koda prikazan u ovom listingu je karakterističan za funkcionalno programiranje i po tome što je funkcija *createTest* funkcija višeg reda, jer kreira i vraća objekat koji sadrži druge funkcije (*attemptPass* i *displayResult*). Pored toga, ova funckija je čista funkcija, jer zavisi isključivo od vrednosti ulaznih parametara.

Još jedna od karakteristika funkcionalnog programiranja koju ovde možemo uočiti je nepromenljivost podataka (engl. *immutability*), jer kao što vidimo, funkcija *attemptPass* ne menja postojeći objekat, već vraća novu instancu pomoću *createTest.* Pored toga, funckija *createTest* je zapravo tzv. fabrička funkcija (engl. *Factory function*), jer kreira novi objekat bez upotrebe ključnih reči *class* ili *new*.

```
const createTest = (subject, points, passed = false) => ({
  subject,
  points,
  passed,
  attemptPass: function (minPointsForPass) {  // Vraća novu instancu Testa sa ažuriranim 'passed' statusom
    return createTest(this.subject, this.points, this.points >= minPointsForPass);
  },
  displayResult: function () {
    return this.passed ? "Passed" : "Failed";
  }
});
```

Listing 5 – Funkcija createTest

U kodu koji prikazuje Listing 6, možemo videti funkciju *createColloquium*, koja ne nasleđuje funkciju *createTest*, već koristi njegov rezultat (objekat) kao osnovu, pa kopira i po potrebi menja njegove metode. U ovom primeru, spread operator (...test) se koristi za kopiranje svih svojstava objekta dobijenog iz *createTest* u novi objekat *createColloquium*, kako bi se zadržalo osnovno ponašanje testa i po potrebi zamenile ili dodale nove metode. To je kompozicija, ne nasledjivanje, što čini razliku u odnosu na objektno-orijentisani princip.

```
const createColloquium = (subject, points, passed = false) => {
  const test = createTest(subject, points, passed);
  return {
    ...test, // Kopira svojstva iz osnovnog testa
    attemptPass: function () {
      // Vraća novu instancu Kolokvijuma sa ažuriranim 'passed' statusom
      return createColloquium(this.subject, this.points, this.points >= 30);
    }
  };
};
```

Listing 6 – Funkcija createColloquium

3.4. Ključne razlike između rešenja dobijenog primenom objektno-orijentisane i funkcionalne paradigme

Razlika u primeni funkcionalnog i objektno-orijentisanog programiranja u rešavanju problema izloženog u poglavlju 3.1 ima više, ali među ključnim razlikama uočavamo sledeće: Kod funkcionalnog principa reševanja problema stanje objekata se uglavnom ne menja, jer se pomoću funkcija kreiraju i vraćaju novi objekti, dok je kod objektno-orijentisanog pristupa stanje promenljivo jer objekti menjaju stanje. Samim tim je lakše testirati funkcionalan kod, nego objektno-orijentisani kod, kod kojeg se menja unutrašnje stanje. S druge strane, objektno-orijentisani kod je efikasniji, jer imamo manje kreiranja novih objekata, dok se kod funkcionalne paradigme kreira veći broj objekata, pa je veći tzv. *Garbage collector* – mehanizam čiji je zadatak da pronađe i obriše objekte u RAM memoriji koji se više ne koriste kako bi oslobodili memoriju za nove objekte, [21]. Nove funkcionalnosti se kod funkcionalnog pristupa rešavanju problema dodavaju pomoću novih funkcija, dok se kod objektno-orijentisanog pristupa ovo rešava proširivanjem klase.

4. ZAKLJUČAK

Radi lakšeg zapošljavanja u struci, neophodno je da se studenti tokom studija upoznaju sa različitim programskim paradigmama. U praksi je najčešći scenario da se studenti najpre upoznaju sa proceduralnim, pa objektno-orijentisanim, a zatim i sa funkcionalnim programiranjem. U trenutku upoznavanja funkcionalnog programiranja, studenti obično već duže vremena vladaju objektno-orijentisanim programiranjem, pa prihvatanje druge programske paradigme predstavlja izazov. U ovom radu je predstavljen primer učenja funkcionalne paradigme na način da se studentima prikazuju rešenja praktičnih problema najpre objektno-orijentisanom, a zatim funkcionalnom paradigmom kako bi se mogle uočiti ključne razlike. Nakon što shvate ključne razlike u načinu rešavanja problema pomoću dve najpoznatije programske paradigme, pretpostavka je da će studentima biti olakšano usvajanje funkcionalne programske paradigme. U budućem radu, uspešnost ovog pristupa se može testirati anketiranjem studenata ili testiranjem nakon kojeg bi se utvrđeni nivo znanja uporedio sa prethodnim generacijama kojima je nova programska paradigma predstavljena na klasičan način.

Metodologija pristupa upoznavanja studenata sa novom programskom paradigmom je opšta, što znači da se ona može primeniti i prilikom prelaska sa bilo koje programske paradigme na drugu, kao što je npr. prelazak sa proceduralnog na objektno-orijentisanu paradigmu, što je u obrazovanju kadra iz oblasti informacionih tehnologija takođe veoma često.

5. LITERATURA

[1] Krunić T.: *Using UML modeling for project-based teaching of the shift from object-oriented to functional programming*, 14th International Scientific Conference Science and Higher Education in Function of Sustainable Development – SED 2025, 2025, https://sed.akademijazs.edu.rs/fajlovi/papers25/proceedings/1-24.pdf

[2] Brasil V.: *3 Things I Learned In 1 Year Working with Functional Programming*, 2020, https://dev.to/vinibrsl/3-things-i-learned-in-1-year-working-with-functional-programming-3gkj

[3] Avramenko O.: *Switching from OOP to Functional Programming*, 2019, https://medium.com/@olxc/switching-from-oop-to-functional-programming-4187698d4d3

[4] Speakman K.: *Why you are scared of Functional Programming*, 2018, https://dev.to/kspeakman/why-you-are-scared-of-functional-programming-hh4

[5] Morbel M.: *JavaScript: The Best of Both Worlds — OOP and Functional Programming*, 2024, https://medium.com/@mkare/javascript-the-best-of-both-worlds-oop-and-functional-programming-257e01821ad1

[6] Imeri E.: *The Yin and Yang of JavaScript: Exploring OOP and Functional Programming I*, 2023, https://medium.com/justeattakeaway-tech/the-yin-and-yang-of-javascript-exploring-oop-and-functional-programming-8fd7e37fec4b

[7] *Programming Paradigms*, 2025, https://cs.lmu.edu/~ray/notes/paradigms/

[8] *Programming Paradigms*, 2025, https://beecrowd.com/blog-posts/programming-paradigms/

[9] Veličković Z.: *Objektno orijentisano programiranje –OOP*, Visoka tehnička škola Niš, 2019

[10] *Šta je objektno-orijentisnao programiranje*, https://cubes.edu.rs/programiranje/sta-je-objektno-orijentisano-programiranje/, 2025

[11] Sözer B.: *The Relationships of Classes in OOP*, https://medium.com/@bsozer06/the-relationships-of-classes-in-oop-a072506325ad, 2023

[12] *Association, Composition and Aggregation in Java*, https://www.geeksforgeeks.org/association-composition-aggregation-java/, 2025

[13] Mohamed E.; *Association vs Aggregation vs Composition vs Inheritance*, https://medium.com/@emad-mohamed/association-vs-aggregation-vs-composition-vs-inheritance-3d63ac32ed92, 2024

[14] Neumann J.; *Fundamentals of functional programming*, https://medium.com/twodigits/fundamentals-of-functional-programming-e808918c8b47, 2024

[15] Aryan A.; *Introduction to Functional Programming: JavaScript Paradigms*, https://www.toptal.com/javascript/functional-programming-javascript, 2025

[16] Walsh D.; *Breaking the Loop: How to use higher order functions to process arrays in JavaScript*, https://medium.com/@RhinoDavid/processing-javascript-arrays-with-foreach-map-reduce-bf40d1e5eac4, 2016

[17] Tran M. Q.; *Why Functional Programming Languages Have No Loops?*, https://medium.com/the-art-of-software-development/why-functional-programming-languages-have-no-loops-67a611b6a244, 2022

[18] Krunić T.; *Analysis of required programming languages and skills in the information technologies labour market,* SYM-OP-IS 2023, ISBN 978-86-335-0836-0 , http://www.symopis2023.mod.gov.rs/

[19] *Knjiga predmeta, studijski program Informacione tehnologije, master strukovne studije*, 2023, https://vtsns.edu.rs/wp-content/uploads/2023/06/Knjiga-predmeta-MIT.pdf

[20] *Knjiga predmeta, studijski program Informacione tehnologije, osnovne strukovne studije,* 2023, https://vtsns.edu.rs/wp-content/uploads/2023/06/Knjiga-predmeta-IT-1.pdf

[21] *Garbage Collection in JavaScript*, 2025, https://www.geeksforgeeks.org/javascript/garbage-collection-in-javascript/