SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Tin Tomašić, Daniel Škrlac

ANALIZA PRAKTIČNE UČINKOVITOSTI FUNKCIJSKOG I OBJEKTNO-ORIJENTIRANOG PROGRAMIRANJA U KONTEKSTU OBRADE PODATAKA

Varaždin, 2023.

Ovaj rad izrađen je na Sveučilištu u Zagrebu, Fakultetu organizacije i informatike u Varaždinu, u Laboratoriju za Web arhitekture, tehnologije, servise i sučelja pod vodstvom **doc. dr. sc. Matija Novak** i predan je za natječaj za dodjelu Rektorove nagrade u akademskoj godini 2022/2023.

Kratice korištene u radu

FP funkcijsko programiranje

OOP objektno-orijentirano programiranje

ES specifikacija programskog jezika ECMAScript

JS programski jezik JavaScript

TS programski jezik TypeScript

NodeJS međuplatformsko poslužiteljsko okruženje za izvođenje JavaScript skripata izvan Web preglednika

tsc TypeScript kompajler (engl. *TypeScript Compiler, tsc*)

Bash Unix ljuska i naredbeni jezik za GNU projekt

JSON često korišten format za pohranu i prijenos podataka u Web okruženjima

RAM memorija s nasumičnim pristupom (engl. *Random Access Memory*); primarna radna memorija procesora

Sadržaj

1. Uvod

Godine 1989., Tim Berners-Lee u suradnji s European Particle Physics Laboratory (CERN) predlaže ideju koju naziva “World Wide Web” (WWW) s ciljem postignuća komunikacije kroz dijeljeno znanje (engl. *shared knowledge*). Tim Berners-Lee definira WWW kao skup svih informacija dostupnih korištenjem računala i umrežavanja gdje je svaka informacija predstavljena kao jedinstven URI identifikator. WWW definira tri temeljne specifikacije [1]:

- URI (engl. *Uniform Resource Identifier*) - predstavlja identifikator određenog resursa,
- HTTP (engl. *HyperText Transfer Protocol*) - definira format i slijed razmjene poruka između subjekata i
- HTML (engl. *HyperText Markup Language*) - određuje strukturu hipertekstualnog (sekvencijalnog) dokumenta.

U travnju 1991. godine objavljuje prvi klijentski program (preglednik) kojeg naziva “WorldWideWeb”, čije je izvršavanje bilo moguće samo na NeXT-baziranim računalima [1]. Razvojem novih, složenijih Web aplikacija i konceptualnog shvaćanja istih javila se potreba za validacijom i upravljanjem HTML elementima.

HTML (i CSS) su jezici oznaka te se koriste u svrhu definiranja strukture, odnosno izgleda Web stranice. Međutim, nisu dovoljni za stvaranje složenih funkcionalnosti kao što su provjera valjanosti obrazaca, obrada podataka na strani klijenta i dinamička korisnička sučelja. Zbog toga nastaje potreba za Web programskim jezikom koji će omogućiti dodavanje interaktivnosti i dinamičko ponašanje Web stranicama. U tu je svrhu, Brendan Eich u svibnju 1994. godine razvio programski jezik Mocha, današnji JavaScript.

Ukoliko promatramo najtraženije programske jezike, najviše rezultata pretraživanja putem Web tražilica izbaciti će JavaScript kao jedan od najpopularnijih programskih jezika. Prema istraživanju TIOBE Index[1] u travnju 2023. godine, JavaScript je, nakon programskog jezika Python, vodeći jezik skripata. Ukoliko promatramo GitHub platformu 2021. godine, prema istraživanju The 2021 state of the Octoverse[2], najveća se količina objavljenih programskih rješenja temelji na programskom jeziku JavaScript. Također, godine 2021. organizacija StackOverflow objavila je rezultate Stack Overflow Developer Survey 2021[3] ankete, gdje je programski jezik JavaScript završio na prvome mjestu u kategoriji skriptnih jezika i jezika oznaka. Postavlja se pitanje zašto je tome tako. Zašto je programski jezik

[1] TIOBE INDEX za travanj 2023. godine - https://www.tiobe.com/tiobe-index/
[2] The 2021 state of the Octoverse - https://octoverse.github.com/2021/
[3] Stack Overflow Developer Survey 2021 - https://insights.stackoverflow.com/survey/2021

JavaScript danas vodeći među skriptnim jezicima i jezicima oznaka? Koje su ključne značajke i mogućnosti koje ovaj jezik nudi, te koji su faktori doprinijeli njegovoj popularnosti?

Programske paradigme definiraju strukturu programskog koda te način organizacije datoteka i programskog koda. Danas, programski jezici omogućuju pisanje programskog koda korištenjem različitih programskih paradigmi. Takva se primjena naziva programiranje primjenom više programskih paradigmi (engl. *multiparadigm programming*) [2], a podržavaju je jezici kao što su C, C++, Kotlin i JavaScript. Naravno, ne postoji nešto što se naziva ispravna programska paradigma, već svaka programska paradigma ima svoje mjesto u primjeni, ali na nama je da odlučimo koju ćemo izabrati za koji slučaj korištenja. S druge strane, postoje programski jezici koji omogućuju pisanje programskog koda primjenom određene programske paradigme - npr. programski jezik Haskell koji se vodi isključivo funkcijskom paradigmom pisanja programskog koda.

Kako se danas programska rješenja susreću s vrlo različitim problemima, bitno je da sam programski jezik ima modularan pristup rješavanju istih. Primjer višenamjenskog i modularnog programskog jezika bio bi JavaScript. Jezik podržava primjenu različitih programskih paradigmi poput objektno-orijentirane, funkcijske, programiranje pogonjeno događajima (engl. *event based programming*), asinkrono programiranje (engl. *asynchronous programming*) i slično. Upravo je zbog tih karakteristika postigao dominantnost programiranja klijentske strane Weba, a kasnije se proširio i na poslužiteljsku stranu putem JavaScript izvršnih okruženja kao što su NodeJS, Rhino, Deno i ostali.

U drugom je poglavlju postavljeno glavno istraživačko pitanje, zajedno s povezanim, manjim općim istraživačkim pitanjima. Treće poglavlje bavi se pregledom dostupne relevantne literature te opisom glavnih koncepata potrebnih za bolje razumijevanje rada. Četvrto se poglavlje bavi detaljnim opisom korištene metodologije - način provedbe eksperimenta ponovljenog mjerenja. U petom je poglavlju dan pregled arhitekture testnih skripata te njihovo objašnjenje. U šestom su poglavlju dani rezultati eksperimentalne analize te je njihova diskusija provedena u narednom, sedmom poglavlju, gdje su također dani odgovori na postavljena istraživačka pitanja. U posljednjem, osmom poglavlju ovog rada, dostupan je kratak zaključak autora gdje su sumirani glavni doprinosi istraživačkog rada.

2. Istraživačka pitanja i ciljevi rada

Glavno istraživačko pitanje koje se postavlja jest: Koja je učinkovitost funkcijskog programiranja (FP) i objektno-orijentiranog programiranja (OOP) u kontekstu obrade stvarnih meteoroloških podataka putem programskog jezika JavaScript? U skladu s tim, postavljena su sljedeća istraživačka pitanja (IP):

1. **Koje su karakteristike funkcijske i objektno-orijentirane paradigme programiranja značajne u kontekstu obrade podataka?**
2. **Koji pristup programiranju - FP ili OOP - ima vremenski efikasnije izvođenje u kontekstu obrade podataka?**
3. **Koji pristup programiranju - FP ili OOP - ima efikasniju prostornu složenost (potrošnja radne memorije računala) u kontekstu obrade podataka?**

Kako bismo utvrdili mogući doprinos ovog rada u području funkcijskog i objektno-orijentiranog programiranja u obradi podataka, neophodno je istražiti dostupnu znanstvenu literaturu na internetu te tako dobiti odgovor na prvo IP. Odgovor je bitan za razumijevanje osnovnih razlika između funkcijske i objektno-orijentirane paradigme programiranja. Detaljno poznavanje karakteristika navedenih paradigmi omogućuje nam bolje razumijevanje u kojim situacijama iskoristiti koji pristup te kako možemo maksimizirati učinkovitost i minimizirati pojavu pogrešaka u kodu.

Drugo IP ima za cilj razumjeti koji pristup programiranju ima vremenski efikasnije izvođenje u kontekstu obrade podataka. Ako se može utvrditi da jedan pristup ima statistički značajno bolju izvedbu od drugog, to bi moglo dovesti do veće učinkovitosti u obradi podataka. Također, ako se može utvrditi da nema značajne razlike u vremenskoj izvedbi, to bi nam moglo pomoći u odabiru pristupa na temelju drugih kriterija, poput čitljivosti koda ili jednostavnosti održavanja. Stoga je važno provesti istraživanje koje bi nam omogućilo da donesemo informirane odluke u odabiru pristupa programiranju u kontekstu obrade podataka.

Relevantni podaci o učinkovitosti prostorne složenosti funkcijskog i objektno-orijentiranog pristupa programiranju u kontekstu obrade podataka su rijetki, što predstavlja izazov za adekvatan odgovor na treće IP. U ovom je istraživačkom radu cilj dobiti relevantne podatke o korištenju radne memorije računala kod JavaScript skripata napisanih u funkcijskoj i objektno-orijentiranoj varijanti. Temeljem rezultata provedenog istraživanja, nastoji se donijeti odgovarajuća odluka o odabiru pristupa programiranju u kontekstu obrade podataka, ovisno o specifičnim potrebama aplikacije i raspoloživim računalnim resursima.

Osim sistematske teorijske podloge funkcijske i objektno-orijentirane programske paradigme, rad uključuje empirijsko istraživanje i analizu praktične primjene navedenih paradigmi. Samo istraživanje temelji se na obradi meteoroloških podataka kroz programske jezike JavaScript i TypeScript s ciljem testiranja učinkovitosti (vremenske i prostorne) funkcijskog i objektno-orijentiranog pristupa programiranju.

Za provedbu istraživanja koristit će se baza prikupljenih meteoroloških podataka od gotovo pola milijuna zapisa, a čije je korištenje omogućeno od strane Fakulteta organizacije i informatike u Varaždinu - Laboratorij za Web arhitekture, tehnologije, servise i sučelja (WATSS). Meteorološki podaci bili su prikupljeni u vremenskom razdoblju između 17.02.2023. i 11.04.2023. kroz ukupno 5 različitih fizičkih lokacija upotrebom senzora temperature, vlage i tlaka zraka.

3. Pregled literature

Ovo poglavlje započinje analizom rezultata već provedenih istraživanja. Nastoji se dobiti odgovor na domenski slična istraživačka pitanja koja su postavljena u ovome radu. Radovi će se analizirati prema sljedećim kriterijima:

- na kojim elementima i aktivnostima se bazira provedba istraživanja,
- koje su tehnologije i alati korišteni u realizaciji,
- nad kojim skupovima podataka se istraživanje temelji te
- koje su metode i metrike korištene u svrhu donošenja zaključka istraživanja.

Nakon pregleda istraživačkih radova, dan je opis relevantnih teorijskih koncepata vezanih uz programske paradigme - osnovna podjela i karakteristike s naglaskom na OOP i FP. Uz teorijsku podlogu programskih paradigmi, dan je i pregled programskih jezika korištenih u radu - povijest razvoja, temeljne karakteristike i sama realizacija u kontekstu OOP-a i FP-a.

U svrhu pronalaska relevantnih radova, baziranih na sličnoj domeni rada u kojoj se nalazi i ovaj istraživački rad, korišteni su radovi iz sljedećih baza podataka: Web of Science, Scopus i Google Scholar. Baze podataka pretraživane su prema različitim ključnim riječima i frazama kao što su "object-oriented programming" i "functional programming", uz druge povezane termine kao što su "comparison", "metrics", "algorithms", "data structures", "languages", "technologies" i slično. Također, pretraživane su i različite publikacije - znanstveni časopisi, konferencijski zbornici, knjige i drugi srodni izvori. Cilj je bio pronaći radove koji se bave temama usporedbe između OOP-a i FP-a, primjenom navedenih paradigmi u različitim programskim jezicima, algoritmima, strukturama podataka i tehnologijama te mjerenjem učinkovitosti (prostorne i vremenske složenosti) u kontekstu obrade podataka. ili s razlogom definiranja jednostavnosti korištenja, modularnosti pristupa, razumljivosti, lakoće otklanjanja pogrešaka i ostalih. Odabir relevantnih radova izvršen je prema kriterijima poput kvalitete rada, autentičnosti i relevantnosti u kontekstu istraživanja.

Istraživanje [3] bavi se usporedbom OOP-a i FP-a u kontekstu automatskog generiranja mikroprocesorskih sklopova u VHDL (HSIC Hardware Description Language) izvornom obliku pomoću programskih jezika Ruby i Clojure. Rezultati istraživanja baziraju se na jednostavnosti upotrebe određene programske paradigme. Rješenje u funkcijskom obliku implementirano je u programskom jeziku Closure, dok je rješenje u objektno-orijentiranom obliku implementirano putem programskog jezika Ruby. Istraživanje navodi kako je realizacija programskog rješenja, i kod FP-a i kod OOP-a, zahtijevala približnu količinu programskog koda. No ipak malu prednost daje rješenju upotrebom FP-a, barem što se tiče

broja fizičkih linija koda. Istraživanje također navodi kako to ipak ne mora predstavljati prednost jer takav način pisanja programskog koda u budućnosti može rezultirati kontra produktivnošću, baš zbog semantičke složenosti FP-a.

Nadalje, autor navodi kako je nakon implementirane prve verzije uvidio mogućnost optimizacije oba rješenje te kako je rješenje pisano u funkcijskoj paradigmi bilo izazovnije za optimizirati u odnosu na rješenje putem OOP-a. Sama optimizacija rješenja temeljila se na promjeni arhitekture aplikacije te autor navodi kako je sama arhitektura implementirana putem FP-a, zbog svojih karakteristika, bila specifičnija te iz tog razloga spriječila fleksibilnost optimizacije. Na kraju autor zaključuje kako, u budućim aplikacijama, sama arhitektura funkcijskog rješenja može rezultirati napretkom u kontekstu jednostavnosti prilagodbe zahtjevima okruženja, no u trenutnoj je izvedbi predstavljala smetnju.

Istraživanje [4] također se bavi kombinacijom OOP-a i FP-a, ali s ciljem definiranja metrika koje bi potencijalno mogle ukazivati na pogreške uzrokovane takvim pristupom razvoja softvera. Za potrebe istraživanja, autor je odabrao programski jezik Scala koji kombinira te dvije paradigme programiranja. Istraživanje uključuje analizu metrika koje uključuju: broj linija koda, broj metoda, broj klasa, broj varijabli (Landkroonova metodologija), kompleksnost metoda i klasa, međusobnu koheziju između metoda i klasa (Birandova metodologija). Autor navodi da je u analiziranim projektima postojao veći postotak pogrešaka pri miješanju funkcijskog i objektno-orijentiranog stila pisanja programskog koda. Odnosno, utvrđeno je da se konstrukti poput korištenja vanjskih (engl. *outer*) varijabli, funkcija s nuspojavama (engl. *side effects*), koncepata ugnježđivanja metoda i podudaranja uzoraka (engl. *pattern matching*) često pojavljuju u objektima koji su kasnije rezultirali pogreškama u kodu.

Istraživanje [5] proučava integraciju FP-a u rješenja bazirana na OOP-u te definira koje su prednosti takvog pristupa. Istraživanje se zasniva na C# programskom jeziku gdje se granularnost programa pokušava smjestiti na razinu funkcija / metoda. Rad nastoji integrirati osnovne karakteristike FP-a u OOP, implementacijom uzorka dizajna zvanog iterator nad objektima, kompozicijama funkcija, prosljeđivanjem funkcija funkciji te ostalim karakteristikama FP-a. Nakon raznih pokušaja implementacija, autor navodi kako se granularnost OOP-a može spustiti na razinu funkcija pomoću koncepata enkapsulacije i refleksije. Odnosno precizinije, pomoću koncepta introspekcije koja predstavlja sposobnost procesa da analizira svoju strukturu.

Također, autor navodi kako nekontrolirano korištenje svojstva enkapsulacije može rezultirati složenom strukturom grafa nasljeđivanja (engl. *inheritance graph*) te kako učestala primjena svojstva introspekcije rezultira lošom sigurnošću tipova podataka. Shodno tome

predlaže rješenje navedenih problema na način poimanja funkcija u OOP-u kao građana prve klase. Takav bi pristup rezultirao integracijom fleksibilnosti koju pruža FP s osiguranjem sigurnosti tipova podataka i ostalih prednosti koje donosi OOP.

Istraživanje [6] posvećuje veliku važnost odabiru ispravne programske paradigme u procesu razvoja aplikacija. Nastoji dobiti odgovor na pitanje kako poznavanje prednosti i ograničenja FP-a i OOP-a može utjecati na sam proces implementacije. Svoje istraživanje zasnivaju na empirijskoj usporedbi između OOP-a i FP-a korištenjem analognih problemskih zadataka u programskim jezicima C#, F#, Haskell i Java. Usporedba se bazirala na tri različita algoritma: problem N kraljica (engl. *N queens problem*), n-ti lijevo-uklanjajući prosti broj (engl. *N-th left-truncatable prime*) i algoritam sortiranja spajanjem (engl. *merge sort*). Tri metrike koje su proučavane u istraživanju su broj linija programskog koda, vremenska složenost (vrijeme trajanja izvođenja pojedinog algoritma u određenom programskom jeziku) i prostorna složenost (veličina zauzete radne memorije prilikom izvođenja algoritma). Sami programski jezici podijeljeni su na način da je objektno-orijentirana varijanta korištena u programskim jezicima Java i C#, funkcijska varijanta u programskom jeziku Haskell te implementacija temeljena na OOP-u i FP-u u programskom jeziku F#.

Rezultati istraživanja pokazali su da Java rješenje ima najbrže vrijeme izvođenja u sva tri algoritma, dok su Haskell rješenja imala najmanje linija koda u svim analiziranim algoritmima. Istraživanje je također pokazalo da su .NET programski jezici značajno sporiji u rješavanju predstavljenih problema u usporedbi s Haskellom i Javom. Međutim, programsko rješenje u čisto funkcijskom obliku (Haskell) zahtijevalo je znatno manje memorije - u prosjeku dva puta manje u usporedbi s drugim programskim rješenjima. Haskell je imao u prosjeku tri puta manje fizičkih linija programskog koda u usporedbi s objektno orijentiranim jezicima. Programski jezik F#, koji kombinira funkcijski i objektno-orijentiran pristup, pokazao se značajno sporijim od svih ostalih što ukazuje na to da kombiniranjem dviju različitih programskih paradigmi ne mora nužno dovesti do boljih performansi programa.

Eksperimenti u radu još jednom su potvrdili da je teško donijeti opći zaključak koja je programska paradigma bolja. Ipak, autori kao zaključak navode da je funkcijska varijanta poželjan izbor prilikom odabira načina implementacije navedenih algoritama s obzirom na broj linija koda, upotrebu memorije i brzinu izvođenja.

Istraživanje [7] temelji se na usporedbi OOP-a i FP-a na temelju četiri različita algoritma - binarno stablo pretraživanja (engl. *binary search tree*), sortiranje ljuske (engl. *shellsort*), Hanojevi tornjevi (engl. *the tower of Hanoi*) i Dijkstrin algoritam (engl. *Dijkstra's algorithm*) - implementiranih u programskom jeziku JavaScript. Promatrane metrike uključuju: vrijeme potrebno za implementaciju algoritma, broj fizičkih linija koda, vrijeme

izvođenja i zauzeće radne memorije računala prilikom izvođenja algoritma. Svaki od algoritama implementiran je na dva načina - kroz FP i OOP.

Prije početka implementacije, autor definira određena pravila implementacije vezana uz OOP i FP. Kod FP-a, autor ne koristi klase i objekte već varijable i funkcije, te tretira funkcije kao varijable radi kompozicije funkcija. Također koristi čiste funkcije, ne mijenja vrijednost varijabli i koristi rekurzije umjesto iteracije. Kod OOP-a, koristi klase i objekte, koncept nasljeđivanja, primjenjuje SRP (engl. *Single Responsibility Principle*) i koristi iteracije prije rekurzija.

Obje su testirane osobe (dva programera) ukupno potrošile manje vremena na funkcijsku implementaciju. Uočeno je kako su oboje potrošili više vremena na objektno-orijentirana rješenja za rekurzivne algoritme, a više vremena na funkcijska rješenja za iterativne algoritme.

Istraživanje je pokazalo da nema jasne prednosti između FP i OOP s obzirom na vrijeme potrebno za razvoj pojedinih algoritama, budući da je vrijeme razvoja ovisilo o algoritmu i načinu implementacije, a ne o samoj programskoj paradigmi. Međutim, u radu je navedeno kako obojica autora imaju više iskustva u OOP-u te kako do prije ovog testiranja nisu koristili FP, što znači da bi rezultati mogli biti drugačiji ukoliko bi se testirali programeri koji su prethodno već koristili FP. Oba su autora primjetila kako funkcijske implementacije za sve algoritme (osim za algoritam sortiranja ljuske) imaju manje koda po pojedinačnoj implementaciji i ukupnog koda, u usporedbi s objektno orijentiranim implementacijama. Za algoritam sortiranja ljuske, obje objektno-orijentirane implementacije imale su manje koda. Međutim, uočeno je kako su obje objektno-orijentirane implementacije bile učinkovitije od funkcijskih u pogledu vremena izvršavanja, dok su funkcijske implementacije zahtijevale kraće vrijeme razvoja i manje fizičkih linija programskog koda. Autori kao zaključak navode da su različiti pristupi dobri za različite zadatke i nijedan (FP i OOP) nije očito bolji od drugog te da je najbolje koristiti kombinaciju programskih paradigmi ukoliko je to moguće.

Istraživanje [8] analizira razlike između funkcijske i objektno-orijentirane dekompozicije u smislu proširivosti i izražajnosti. Autori su zaključili da postoje važni kompromisi između funkcijske i objektno-orijentirane dekompozicije te da su programeri obično prisiljeni odabrati određeni stil dekompozicije u ranoj fazi programiranja. S obzirom da je, jednom već donešenu, pogrešnu dizajnersku odluku teško promijeniti, u radu se predlaže dvosmjerni transformacijski sustav (engl. *bidirectional transformation system*) između funkcijske i objektno-orijentirane dekompozicije nazvan FOOD, a koji je formaliziran u FOOD kalkulusu i implementiran u Scala programskom jeziku kao alat za prevođenje (Cook). Transformacijski sustav ima za cilj riješiti problem odabira određenog stila dekompozicije u ranoj fazi programiranja i visoku cijenu prebacivanja između njih. Rad

pokazuje primjenjivost i učinkovitost Cook-a kroz nekoliko studija slučaja i predlaže budući rad na istraživanju više značajki, poboljšanju upotrebljivosti i mehanizaciji ručnih provjera.

Istraživanje [9] analizira kombinaciju funkcijskih i objektno-orijentiranih pristupa u modeliranju softvera, posebno korištenjem dijagrama toka podataka (engl. *Data Flow Diagram, DFD*) i jezika za unificirano modeliranje (engl. *Unified Modeling Language, UML*) u fazi analize ugrađenih softverskih sustava. Autori predlažu tri načina kombiniranja DFD-a i UML-a: korištenje DFD-a za preciziranje modela slučaja upotrebe (engl. *use case*), određivanje operacija komponente sustava ili transformacija DFD-a u dijagrame klasa (engl. *class diagram*). Tvrde da, dok su slučajevi korištenja popularna tehnika u objektno-orijentiranom razvoju, DFD-ovi bi mogli biti prikladniji za neke autonomne ugrađene sustave pogonjene podacima (engl. *data-triggered*).

Autori zaključuju da kombinacija funkcijskog i objektno-orijentiranog pristupa može dati dobre rezultate samo ako se koristi s oprezom. Objektno-orijentirani programski jezici nude mnoge prednosti u programiranju, a DFD-ovi mogu biti korisni u modeliranju nekih vrsta ugrađenih sustava. Međutim, autori naglašavaju potrebu za budućim radom na procjeni korisnosti ovih tehnika u složenim primjerima, istraživanju implikacija i posljedica kombiniranja DFD-a i UML-a na razini semantičkog i meta-modela te analiziranju učinkovitih načina za proširenje 4SRS-a.

Istraživanje [10] uspoređuje 20 programa napisanih u C++ i Java programskim jezicima koristeći različite objektno-orijentirane metrike za mjerenje kompleksnosti. Metrike korištene u ovom istraživanju uključuju funkcije/metode, veličinu, klase, njihovu koheziju i ponovnu iskoristivost. Autori su sveukupno zaključili da rezultati pokazuju da je Java, s obzirom na promatrane metrike, bolji objektno-orijentirani programski jezik od C++-a. Korištenje objektno-orijentiranih metrika pokazalo se učinkovitim u poboljšanju kvalitete softvera.

Istraživanje [11] bavi se diskusijom razlika između OOP-a i FP-a, ističući njihove prednosti i upotrebu. U radu se navedene paradigme uspoređuju u kontekstu dizajna softvera. Obje paradigme imaju za cilj stvoriti sveobuhvatne kvalitetne aplikacije bez pogrešaka, ali pristupaju programiranju na različite načine. OOP temelji se na skupu primitiva koje jezik pruža, dok je FP deklarativnog stila i fokusira se na ono *što* treba učiniti, a ne na *kako* to učiniti.

Autori smatraju da odabir paradigme programiranja treba biti u rukama programera jer oni najbolje znaju, iz praktičnog aspekta, kakvu paradigmu odabrati za odgovarajuću vrstu zadatka. Kao zaključak usporedbe, navode da je FP prikladnije kada postoji potreba za

dodavanjem novih operacija fiksnom skupu entiteta, dok je OOP prikladnije kada je fiksan skup operacija nad entitetima zadovoljavajuć. Generalno, autori smatraju da obje paradigme imaju svoje mjesto u modernom razvoju softvera, pri čemu se FP preispituje i sve češće koristi u tehnologijama velikih podataka (engl. *Big Data Technologies*), dok je OOP i dalje popularno rješenje za predstavljanje entiteta iz stvarnog svijeta kao objekte u kodu.

Zbog sličnosti domene istraživanja s ovim istraživačkim radom, posebno su nam interesantni radovi [6], [7] i [11].

U odnosu na [6], [7] ovaj se rad razlikuje u primjeni algoritama i programskog jezika. U oba je rada cilj izmjeriti praktičnu učinkovitost različitih programskih paradigmi. Međutim, ovaj se rad fokusira na praktičnu primjenu funkcijskog i objektno-orijentiranog programiranja u kontekstu obrade podataka, dok se spomenuti rad fokusira na primjenu algoritama za rješavanje problema N kraljica, n-tog lijevo-uklanjajućeg prostog broja i algoritma sortiranja spajanjem. Ovaj je rad fokusiran na programski jezik JS i njegov tipiziran pandan TS u kontekstu obrade podataka, dok spomenuti rad koristi C#, F#, Haskell i Javu.

Naspram [7], ovaj će rad pružiti statističku analizu i potvrdu značajnosti rezultata mjerenja vremenske i prostorne složenosti algoritma obrade podataka u JS programskom jeziku. Dodatan je doprinos ovog rada u tome što se, umjesto predefiniranih algoritama poput sortiranja ljuske, koristi praktičan scenarij obrade podataka, a koji po svojoj prirodi i jedinstvenosti nema unaprijed definirane stroge korake implementacije. Također, ovaj rad nastoji usporediti vremensku i prostornu složenost JS skripata za veće ulaze u algoritam - gotovo pola milijuna podataka, za razliku od najvećih, deset tisuća nasumično generiranih testnih podataka u slučaju binarnog stabla pretraživanja.

Ovaj istraživački rad, u usporedbi s [11], ima sličnu ideju: analizirati i usporediti FP i OOP, ali u kontekstu obrade podataka. Iako su autori identificirali FP kao sve popularniju paradigmu u kontekstu obrade velikih podataka, ovaj rad nastoji statistički utvrditi postoji li, nakon dvije godine napretka obiju paradigmi, zaista statistički značajna razlika u njihovoj usporedbi, s fokusom na vremensku i prostornu složenost.

Prema navedenim izvorima možemo zaključiti kako je zadnje istraživanje, s domenom rada čiji je cilj definirati razlike na temelju prostorne i vremenske složenosti problemskih zadataka temeljenih na objektno-orijentiranom i funkcijskom pristupu uporabom JS tehnologija, provedeno 2018. godine [7]. Zbog izrazite popularnosti JS i TS tehnologija u proteklih 5 godina, ovaj će istraživački rad *osvježiti* pogled o njihovom poimanju objektno-orijentiranih i funkcijskih koncepata na konkretnom problemskom zadatku koji se bavi obradom meteoroloških podataka.

3.1 Programske paradigme

Područje informacijsko-komunikacijskih tehnologija neprestano se razvija te samim time i broj programskih paradigmi kontinuirano raste. Shodno tome, ne postoji točan podatak o njihovom broju danas. Podjela programskih paradigmi razlikuje se od autora do autora, no jedna je od osnovnih klasifikacija temeljena na imperativnom i deklarativnom načinu pisanja programskog koda. Točan autor ove podjele nije poznat, no klasifikacija na imperativnu i deklarativnu paradigmu korištena je još davne 1996. godine u knjizi "*Structure and Interpretation of Computer Programs*" [12], gdje autori već tada definiraju razliku između imperativnih i deklarativnih izjava.

3.2 Imperativna paradigma

Program vođen imperativnom paradigmom temelji se na nizu imperativnih naredbi koje modificiraju stanje programa. Naziv "*imperativan*" preuzet je iz prirodnog jezika. Imperativom izričemo zapovijed, odnosno definiramo naredbu - na primjer "dodijeli varijabli a vrijednost 3". Dakle, imperativna paradigma se fokusira na definiranje potrebnih instrukcija koje zadajemo računalu - ono što program treba napraviti. Imperativna paradigma se prema [13] dijeli još u dvije osnovne paradigme:

- **proceduralna paradigma** (engl. *procedural paradigm)*
- **objektno-orijentirana paradigma** (engl. *object-oriented paradigm*)

3.2.1 Proceduralna paradigma

Proceduralna paradigma se bazira na tradicionalnom strukturnom programiranju (npr. programski jezik C) i na konceptu **procedura** (engl. *procedures*). Procedura je definirana kao blok koda koji se sastoji od niza instrukcija koje čine neki kompleksniji zadatak (algoritam) [13]. Kako proceduralna paradigma nije glavni fokus ovog istraživačkog rada, neće biti daljnjeg razmatranja na ovu temu.

3.2.2 Objektno-orijentirana paradigma

Glavni konstrukt objektno-orijentirane paradigme je **objekt**. On se sastoji od **atributa** (svojstava) te operacija (procedura/funkcija) koje u kontekstu objektno-orijentirane paradigme zovemo **metodama**. Sami se objekti, u većini programskih jezika, kreiraju dinamički[4]. Dinamičko kreiranje predstavlja alociranje memorijskog prostora objekta za

[4] Primjer programskog jezika koji podržava i statičko i dinamičko kreiranje objekata bio bi C++.

vrijeme izvođenja programa, dok se statičko instanciranje objekta događa prilikom pokretanja programa.

Kako bismo smanjili dupliciranje koda u slučaju definiranja objekata, uvodimo koncept klase (engl. *class*). Klasa predstavlja model/predložak (engl. *blueprint*) za objekte sličnih karakteristika.

Uz koncepte klase i objekta, OOP zasniva se i na sljedećim karakteristikama [13]:

- **učahurivanje/enkapsulacija** (engl. *encapsulation*) - učahurivanje predstavlja grupiranje podataka i operacija sličnih svojstava, što nam omogućuje da izoliramo određeni dio programske logike,
- **skrivanje informacija** (engl. *information hiding*) - predstavlja način na koji je učahurivanje implementirano, a cilj mu je omogućiti pristup određenim svojstvima i metodama u kontroliranom stanju i pod određenim uvjetima,
- **apstrakcija** (engl. *abstraction*) - skriva kompleksne funkcionalnosti koda te otkriva samo određene funkcionalnosti kako bi se od korisnika sakrili *nepotrebni* detalji implementacije,
- **nasljeđivanje** (engl. *inheritance*) - omogućuje da određena podklasa (klasa dijete) naslijedi određena svojstva i metode od nadklase (klasa roditelj); koja svojstva i metode će naslijediti ovisi o primjeni modifikatora pristupa u roditeljskoj klasi, te
- **dinamički polimorfizam** (engl. *dynamic polymorphism*) - možemo definirati kao mogućnost da određeni objekt ima više različitih oblika, odnosno da se objekti različitih tipova mogu tretirati kao objekti istog tipa.

3.3 Deklarativna paradigma

Deklarativna paradigma omogućuje rješavanje određenog problema na način da u prvi plan stavlja rezultat rješavanja problema - *što* želimo postići, a ne *kako* je samo rješenje postignuto (kojim strukturama, algoritmima i slično). Deklarativno programiranje, prema [13], sadrži sljedeća svojstva:

- pretvara određeni ulaz u određeni izlaz,
- deklarativne operacije su determinističke,
- sve su deklarativne operacije nezavisne i
- deklarativne operacije su bez stanja.

Deklarativna paradigma se prema [13] dijeli još u dvije osnovne paradigme:

- **logička paradigma** (engl. *logic paradigm*)
- **funkcijska paradigma** (engl. *functional paradigm*)

3.3.1 Logička paradigma

Logički program je niz aksioma i pravila koji određuju pravila između objekata. Logički program radi na principu deduktivnih zaključaka. Osnovni konstrukti logičkog programiranja su termini (engl. *terms*) i izjave (engl. *statements*) [14]. Kako logička paradigma nije glavni fokus ovog istraživačkog rada, neće biti daljnjeg razmatranja na ovu temu.

3.3.2 Funkcijska paradigma

Funkcijsko programiranje zasniva se na matematičkom sustavu, koji je predstavljen davne 1930. godine od strane matematičara Alonza Churcha, koji se zove λ (lambda) kalkulus (engl. λ-calculusa). Lambda izrazi vrlo su popularni u današnjim programskim jezicima koji podržavaju paradigmu funkcijskog programiranja. Predstavljaju izraze koji se ponašaju kao funkcije. Sintaksa λ-kalkulusa temelji se na 3 koncepta:

- **identifikatori** (engl. *identificators*) ili varijable - služe za referenciranje vrijednosti,
- **apstrakcije** (engl. *abstractions*) - lambda izraz koji vraća određeni rezultat te
- **aplikacije** (engl. *applications*) - radnja dodjeljivanja određenog argumenta (ili više njih) lambda izrazu.

Uzmimo za primjer funkciju koja ima za zadatak izračunati umnožak dvaju identifikatora $(x,\ y)$. Kažemo [13]:

$$(x,\ y) \in \text{Identifikator},$$

$$xy \in \text{Apstrakcija},$$

$$e_1, e_2 \in \text{Aplikacija i}$$

$$e \in \text{Izraz},$$

$$\text{gdje je } e\ = xy \mid e_1 e_2 \mid \lambda_x \lambda_y xy$$

Ukoliko vrijednosti identifikatora iznose $e_1 = 1$ i $e_2 = 2$, zapisani lambda izraz $((\lambda_x \lambda_y xy)2\ 3)$ možemo definirati kao primjena funkcije koja prima dva argumenta x i y, a zatim se primjenjuje na argumente 2 i 3. Nakon primjene, funkcija vraća umnožak argumenata, što u ovom slučaju iznosi 6.

U funkcijskom programiranju stanje programa definirano je jednoznačno putem funkcijskih argumenata ili povratne vrijednosti, što dodatno potvrđuje svojstvo deklarativnog programiranja - deklarativne operacije su nezavisne, odnosno stanje funkcije definirano je samo ulaznim argumentima i izlaznim rezultatom, te ista ne ovisi o nekom vanjskom

kontekstu. Ovaj je zaključak u funkcijskom programiranju definiran pojmom referentne transparentnosti (engl. *referential transparency*) koja opisuje da će neki izraz ili funkcija dati isti izlaz za isti ulaz, bez obzira koliko je puta izraz ponovljen. Takve funkcije još nazivamo i čistim (engl. *pure*) funkcijama.

Dodatno, svojstvo nepromjenjivosti (engl. *immutability*) osigurava da vrijednosti složenih ili primitivnih tipova varijabli budu nepromjenjive, odnosno da promjena vrijednosti nad određenom varijablom rezultira kreiranjem nove varijable.

Još jedno vrlo važno svojstvo funkcijskog programiranja je to da tumači funkcije kao građane prve klase (engl. *first class citizen*). Tada se funkcija može dodijeliti nekoj varijabli, vratiti kao rezultat neke druge funkcije, proslijediti kao argument nekoj drugoj funkciji, može se spremiti u odgovarajuću strukturu podataka i slično. Ukoliko funkciju možemo proslijediti drugoj funkciji kao parametar, tada takav koncept nazivamo funkcije višega reda (engl. *higher-order functions*) [15].

Uz koncept funkcija višeg reda, FP bazira se i na kompozicijama funkcija, gdje komopozicija definira spajanje dvije ili više funkcija u jednu funkciju koja izvršava operacije svih funkcija u lancu. Nadalje, u FP-u također postoji koncept kariranja (engl. *currying*) koji nam omogućuje pretvaranje funkcije, koja prima više argumenata, u seriju funkcija kod koje svaka prima samo jedan argument [16].

3.4 Usporedba OOP-a i FP-a

U tablici 1 prikazane su glavne razlike između OOP-a i FP-a kroz nekoliko ključnih kategorija - specifičnosti, prednosti i nedostaci - na temelju prethodnih ulomaka o FP-u i OOP-u. Kroz nekoliko glavnih karakteristika, tablica na sažet način prikazuje njihovu usporedbu.

OOP smatra se imperativnom programskom paradigmom koja se temelji na organizaciji programa oko objekata i metoda. S druge strane, FP spada u deklarativnu programska paradigmu koja se temelji na funkcijama i njihovim argumentima. U tablici se također uspoređuju ključni elementi korištenja u OOP-u i FP-u, poput korištenja objekata i metoda u OOP-u te varijabli i funkcija u FP-u.

Osim toga, tablica prikazuje i ostale bitne razlike, poput krivulje učenja same programske paradigme, promjenjivosti podataka, redoslijeda izvođenja, operacija, stanja, nezavisnosti operacija, nuspojava u programu i slično.

Tablica 1: Usporedba između OOP-a i FP-a

Karakteristika	OOP	FP
Programska paradigma	Imperativna	Deklarativna
Glavni elementi korištenja	Objekti i metode	Varijable i funkcije
Krivulja učenja	Visoka	Srednja
Promjenjivost podataka	Promjena je moguća	Promjena nije moguća
Redoslijed izvođenja	Slijedno izvođenje	Redoslijed izvođenja nije predefiniran
Operacije	U većini slučajeva nisu determinističke	Determinističke
Stanje	Pamte stanje	Bez stanja
Nezavisnost operacija	U većini slučajeva ne vrijedi	Uvijek vrijedi
Nuspojave	Dolazi do nuspojava u kodu	Nuspojave nisu moguće
Ponavljanje izvođenja	Koristi koncept iteracija strukturnog programiranja	Koristi koncept rekurzije
Korištenje naredbi	Nasumično	Naredbe imaju slijed

4. Metodologija

U ovom je poglavlju pobliže objašnjena metodologija plana istraživanja, zajedno sa znanstvenim metodama koje će biti korištene u ovom istraživanju, a kako bi se dobili odgovori na istraživačka pitanja postavljena u ovom radu. Detaljno je opisan način izvedbe eksperimenta ponovljenog mjerenja, a što uključuje metrike i varijable samog eksperimenta.

4.1 Način provedbe

Za ispitivanje vremenske i prostorne učinkovitosti FP-a i OOP-a u kontekstu obrade podataka kreirat će se dvije JS skripte. Prva će JS skripta biti izrađena isključivo koristeći FP (funkcije i njihovu kompoziciju), dok će druga JS skripta implementirati identičan algoritam obrade meteoroloških podataka, s jedinom razlikom da će koristiti objektno-orijentirane koncepte (klase i njihove instance, objekte). Osim spomenutih JS skripata, implementirat će se i njihovi tipizirani pandani u TS inačicama - FP u TS-u i OOP u TS-u.

Potrebno je napomenuti da će navedene skripte mjeriti isključivo vrijeme trajanja obrade meteoroloških podatka, dakle bez vremena potrebnog za učitavanje istih u radnu memoriju računala.

Što se tiče prostorne složenosti, ona će se računati na način da će svaka skripta, nakon što učita potrebne podatke u radnu memoriju, zabilježiti trenutnu veličinu rezidentnog skupa (engl. *resident set size, RSS*) radne memorije, a koju će oduzeti nakon što završi s obradom meteoroloških podataka.

NodeJS izvršno okruženje radi na V8 stroju (engl. *V8 engine*[5]) koji, osim što kompajlira JavaScript kod u strojni jezik i izvršava ga, također upravlja dodjeljivanjem i oslobađanjem memorije programa tijekom izvođenja [17]. Prema [18], V8 dijeli memoriju NodeJS aplikacije u tri ključna segmenta:

1. **kodni segment** (engl. *code segmet*) - sadrži sav kod koji se treba izvršiti,
2. **segment stoga** (engl. *stack segment*) - sadrži sve lokalne varijable; odnosno, pokazivače na objekte referencirane na hrpi i
3. **segment hrpe** (engl. *heap segment*) - sadrži sve objekte koje referenciraju pokazivači na stogu.

Kombinacija svih troje memorijskih segmenta čini memorijski blok koji se naziva rezidentni skup (engl. *resident set*), a njegova veličina predstavlja ukupnu memoriju dodijeljenu jednoj NodeJS aplikaciji.

[5] Googleov stroj visokih performansi za izvršavanje JavaScripta i WebAssembly-a.

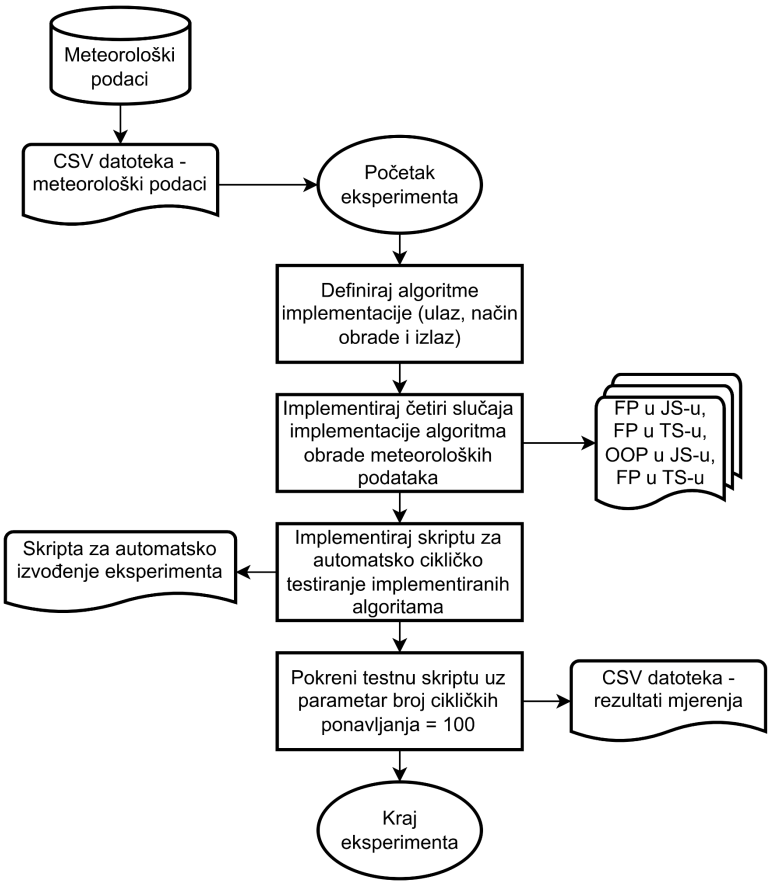
Dakle, svaka će JS skripta kao izlaz imati tri podatka u točno predefiniranom redoslijedu:

1. **vremensku složenost** - vrijeme trajanja izvođenja skripte u milisekundama,
2. **prostornu složenost** - veličina zauzete radne memorije u bajtovima (RSS) i
3. **obrađeni meteorološki podaci** - meteorološki podaci grupirani po lokacijama uređaja i danima unutar njih, u JSON (engl. *JavaScript Object Notation*) formatu.

Kako bismo dobili što objektivniju sliku stvarne vremenske i prostorne složenosti pojedinih programskih paradigmi u kontekstu obrade podataka, eksperiment će se ponoviti 100 puta i to na način da će se skripte izvoditi **ciklički**. Razlog tome je uklanjanje bilo kakve mogućnosti predmemoriranja - bilo na razini operacijskog sustava ili NodeJS okruženja. Takvim izvođenjem želimo osigurati podjednake scenarije izvođenja za sve testne skripte.

Također, ponavljanjem eksperimenta više puta možemo dobiti prosječne vrijednosti mjera algoritama obrade podataka te procijeniti varijabilnost i raspon tih mjera, a sve sa svrhom smanjenja utjecaja slučajnih varijacija poput pozadinski procesa koji mogu utjecati na jedno izvođenje eksperimenta.

Radi lakšeg razumijevanja procesa eksperimenta, dijagram toka u nastavku opisuje način odvijanja eksperimenta s visoke razine apstrakcije.

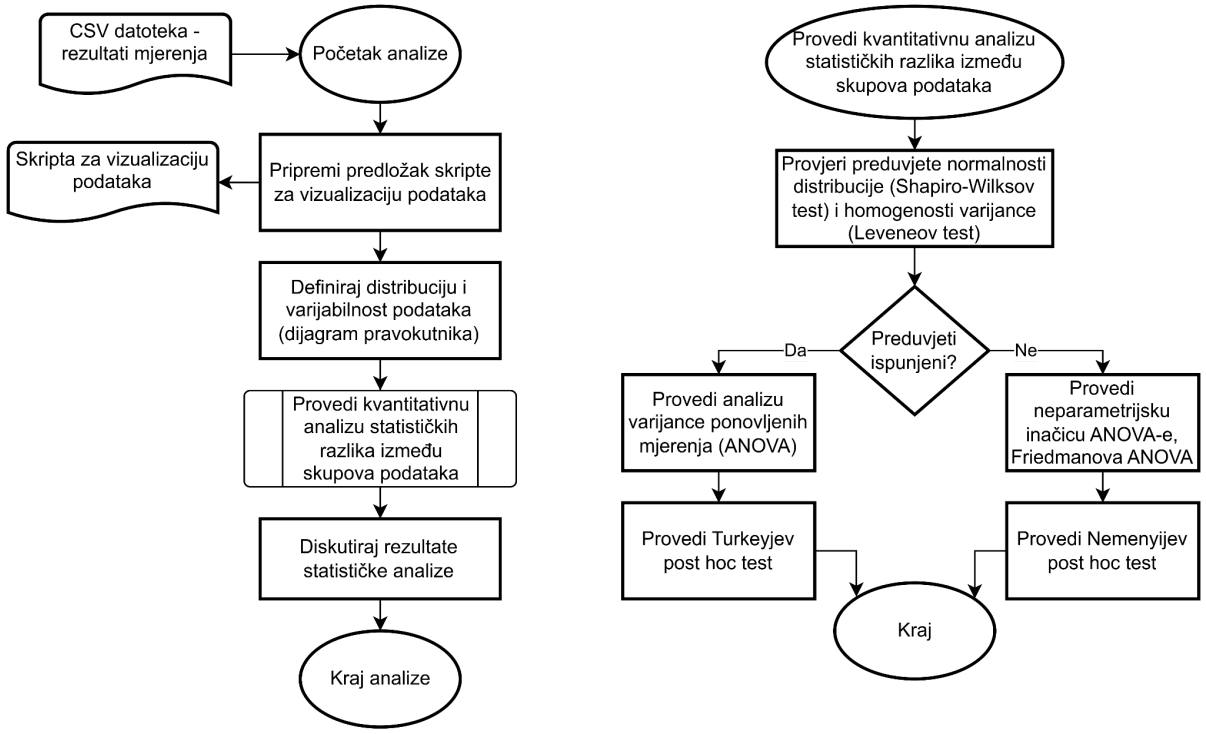


Slika 1: Dijagram toka - pregled odvijanja eksperimenta

Kod analize dobivenih rezultata, planira se koristiti analiza varijance ponovljenih mjerenja (engl. *Repeated Measures ANOVA*). Ukoliko preduvjeti normalnosti distribucije i homogenosti varijance ne budu zadovoljeni, uzet će se neparametrijska inačica ANOVA-e ponovljenog mjerenja sa zavisnim uzorcima - Friedman ANOVA test - kao alternativa.

Post-hoc testovi koji će se koristiti nakon izvršene glavne analize varijance, a kako bi se utvrdile specifične razlike između testiranih uzoraka, su Tukeyjev test i Nemenyijev test respektivno [19]. Za izračun navedenih statistika i vizualizaciju koristit će se programski jezik Python 3, odnosno njegovi moduli - *scipy* [20], *scikit-posthocs* [21] i *matplotlib* [22] moduli.

Sam proces analize dobivenih rezultata vizualiziran je dijagramom toka u nastavku.



Slika 2: Dijagram toka - pregled odvijanja analize rezultata eksperimenta

Nakon provedene odgovarajuće analize rezultata i usporedbe odabranih programskih paradigmi, ukoliko se utvrdi statistički značajna razlika između istih, primijenit ćemo strategije optimizacije kako bismo poboljšali rezultate lošije programske paradigme te vidjeli mogu li se izjednačiti ili barem približiti vremenski boljoj.

4.2 Eksperimentalne varijable

Zavisne varijable koje će se promatrati u ovom eksperimentu su vremenska (vrijeme potrebno za izvođenje skripte) i prostorna složenost (veličina zauzete radne memorije računala). Za izračun vremenske složenosti koristiti će se trenutno vrijeme (*Date*

klasa), dok će izračun ukupne potrošene fizičke radne memorije računala biti određen pomoću *rss()* metode iz *NodeJS.Process.memoryUsage* klase [23].

Eksperiment sadrži četiri **nezavisne varijable** (faktore) nad kojima će se testiranje vršiti. One su predstavljene u obliku JS i TS skripata, odnosno algoritama obrade meteoroloških podataka, implementiranih u FP i OOP varijanti. Pa tako dobivamo sljedeća četiri slučaja implementacije:

1. FP u JS-u,
2. OOP u JS-u,
3. FP u TS-u i
4. OOP u TS-u.

Nadalje, uočeno je i nekoliko **kontroliranih varijabli**, a koje su pobrojane u tablici ispod (Tablica 2).

Tablica 2: Identificirane kontrolirane varijable eksperimenta

Varijabla	Specifikacija
Računalo i operacijski sustav	Eksperiment će se provoditi na Acer Predator Helios 300 laptopu s instaliranim Linux Mint 21.1 operacijskim sustavom. Laptop ima Intel© 6-core™ i7-8750H procesor @ 2.20Ghz / 4.10GHz (9MB cache) Coffee Lake, s 16GB 2666 MHz DDR4 SDRAM-a.
Skup i količina podataka	Eksperiment će obrađivati isti skup podataka - 447193 meteoroloških zapisa o temperaturi, vlazi i tlaku zraka.
Testne skripte	Eksperiment će mjeriti tražene metrike tako da uvijek pokreće iste testne skripte.
Algoritam obrade podataka	Sve pripremljene testne skripte imaju jednak ulaz i izlaz, te jednaku vremensku složenost algoritma obrade meteoroloških podataka.
Verzije alata	Za potrebe eksperimenta, koristit će se jednake verzije testnih alata: NodeJS verzija 18.12.1 i tsc verzija 4.9.4.

5. Opis korištenih tehnologija i programskih jezika

Kao što je prethodno spomenuto, za potrebe istraživanja učinkovitosti FP i OOP u kontekstu obrade stvarnih meteoroloških podataka, implementirana su četiri *jednaka* slučaja algoritma obrade meteoroloških podataka. Algoritmi obrade podatka implementirani su u skriptnom programskom jeziku JS, odnosno njegovom tipiziranom pandanu TS. Shodno tome, nastale su sljedeće testne skripte, datoteke:

1. *FP_JS.js* - FP u JS-u,
2. *OOP_JS.js* - OOP u JS-u,
3. *FP_TS.ts* - FP u TS-u i
4. *OOP_TS.ts* - OOP u TS-u.

5.1 JavaScript

U ovom ćemo poglavlju detaljnije protumačiti povijest i razvoj programskog jezika JavaScript. Zatim ćemo se usredotočiti na objektno-orijentiranu paradigmu u JavaScriptu, objasniti njezine osnovne koncepte i pokazati kako se ona implementira u praksi. Opisat ćemo klase, objekte, nasljeđivanje, metode i druge osnovne elemente OOP-a u JavaScriptu.

Konačno, razmotrit ćemo funkcijsku paradigmu u JavaScriptu, koja je postala vrlo popularna u posljednjih nekoliko godina. Objasnit ćemo osnovne koncepte FP-a, uključujući funkcije prvog reda, visoko redne funkcije, lambda izraze i druge koncepte. Pokazat ćemo kako se funkcijska paradigma primjenjuje u JavaScriptu i kako se koristi za rješavanje raznih problema i izazova.

No prije svega, potrebno je spomenuti kako JavaScript spada u kategoriju jezika skripata. Skriptni jezici zasnivaju se na precima - tumačima naredbi (engl. *command interpreters*) kao što je Unix ljuska i alatima za procesiranje i generiranje podataka (npr. *awk*, *sed* i slično). Osnove karakteristike jezika skripata su [24]:

- **Terminologija** - Programski kod napisan korištenjem jezika skripata naziva se skripta.
- **Način procesiranja** - Postoje razne implementacije procesiranja podataka kod jezika skripata. Kod nekih skriptnih jezika poput Perla, postoji mogućnost kompajliranja cijele skripte prije reproduciranja nekog izlaza. Danas su jezici skripata većinom interpreterski jezici te se kao takvi interpretiraju liniju po liniju - zahtijeva korištenje programa interpretera. Primjeri takvih jezika skripata uključuju Python, JavaScript, Ruby i slično.

- **Ekonomija izraza** - Jezici skripata vode se tendencijom jednostavnije sintakse programskog jezika, čime smanjuju pisanje ponavljajućeg koda (engl. *boilerplate code*).
- **Niska razina tipizacije** - Jezici skripata imaju slabo povezivanje tipova podataka. Takve varijable nemaju stalni tip podataka, što znači da se prilikom izvođenja programa sam tip varijable može mijenjati.
- **Pristup naredbama operacijskog sustava** - Za razliku od sistemskih programskih jezika (npr. programski jezik C), skriptni jezici pružaju jednostavniji pristup naredbama operacijskog sustava.
- **Strukture podataka** - Skriptni jezici uvode strukture podataka u samu sintaksu programskog jezika.

5.1.1 Povijest i razvoj

Razvoja Web preglednika - Ervise i ViolaWWW bili su preteča razvoju Web preglednika Mosaic koji je osmišljen 1993. godine. Mosaic preglednik je bio prvi preglednik koji je omogućio prikaz slike i teksta u jednom prozoru. Nakon razvoja preglednika Mosaic, započela je znatna popularizacija World Wide Weba. Iste godine, Marc Andreessen osnovao je tvrtku Netscape te odmah nakon toga razvio novi Web preglednik 1994. pod nazivom Netscape Navigator. Preglednik Mosaic nakon toga nije nestao, već je njegov izvorni kod iskoristila tvrtka Microsoft kako bi izgradila Web preglednik pod nazivom Internet Explorer 1995. godine.

Marc Andreessen 1995. godine dolazi do ideje kako bi bilo korisno imati mogućnost dinamičkog prikazivanja podataka u Web pregledniku, manipulacijom objektnog modela preglednika (engl. *Browser Object Model*). Vođen tom idejom, zadaje zadatak Brendanu Eichu da osmisli jezik sličan, tada najpopularnijem jeziku, Javi te da ga integrira u Web preglednik. Nakon samo 10 dana, Brendan dolazi do rješenja i razvijeni programski jezik naziva Mocha. U rujnu 1995. godine dolazi do nove verzije Netscape preglednika - Netscape Navigator 2. Zajedno s tim dolazi i do izmjene naziva programskog jezika Mocha u LiveScript. No ni taj naziv nije dugo opstao. Kako bi LiveScript dostigao popularnost Jave, LiveScript naziv ponovno se mijenja u, danas već svima poznati, JavaScript. Vrlo brzo, tvrtka Microsoft kreirala je svoju verziju JavaScripta koju su nazvali JScript, a koja je bila integrirana u Web preglednik Internet Explorer [25].

Nakon toga, došlo je do podjele na JScript i JavaScript implementacije što se nije svidjelo većini razvojnih inženjera. Shodno tome, godine 1997., organizacija ECMA International započinje sa standardizacijom koju su nazvali ECMAScript. ES programski

jezik temelji se na programskom jeziku JavaScript i programskom jeziku JScript. Iako nije najispravnije nazivati današnji ECMAScript JavaScriptom, zbog svoje popularnosti i povijesti, termin JavaScript znatno se više primjenjuje u praksi te još uvijek znatno veći broj autora oslovljava današnji ES programski jezik kao JavaScript. ES se do danas razvijao kroz trinaest izdanja, a trenutno trinaesto izdanje naziva se ECMAScript 2022 [26].

5.1.2 OOP u JavaScriptu

JavaScript je objektno-orijentiran programski jezik; preciznije rečeno, prototipno baziran na OOP-u. Od verzije ECMAScript 2015 (ES6), radi sličnosti poimanja objektno-orijentiranih koncepata s ostalim programskim jezicima, ECMA International uvodi ključnu riječ *class* za definiranje klasa u JS-u. JS kao objektno-orijentirani jezik omogućava sljedeće koncepte OOP-a: objekti i klase/konstruktorske funkcije, enkapsulacija i skrivanje podataka, nasljeđivanje, apstrakcija i dinamički polimorfizam.

5.1.2.1 Objekti i klase

Objekt je spremnik vrijednosti kombiniranih u jednu podatkovnu strukturu koja ima određeni identitet. Obično se koristi za predstavljanje određene cjeline kao što je osoba, narudžba, student, evidencija, rezervacija i slično, kroz agregaciju podataka i funkcionalnosti. U programskom jeziku JS, podaci se zovu svojstva (engl. *properties*), dok se funkcionalnosti nazivaju metode (engl. *methods*). U JS-u postoje dva pristupa kreiranja objekata: doslovno utemljen pristup (engl. *literal-based approach*) i pristup na temelju konstruktora (engl. *constructor-based approach*) [27].

Primjer kreiranja objekta na temelju doslovno utemeljenog pristupa:

```
let prazanObjekt = {}
let doslovniObjekt = {
  'svojstvo': 'vrijednost',
  'ispis': function() {
     console.log(this.svojstvo)
  }
}
```

Kao što možemo vidjeti iz prethodnog primjera, korištenjem doslovno utemeljenog pristupa ne možemo ostvariti ponovnu iskoristivost svojstava i metoda, već moramo za svaki objekt definirati ista pravila. Kako bismo izbjegli takvu situaciju možemo primijeniti pristup na temelju konstruktora:

```
function KonstruktorskaFunkcija() {
```

```
  this.svojstvo = 'Vrijednost'

  this.ispis = function() {
      console.log(this.svojstvo)
  }
}

let objekt = new KonstruktorskaFunkcija()
```

Od specifikacije ECMAScript 2015, JS omogućuje korištenje koncepta klasa putem ključne riječi *class*. Treba napomenuti kako klasa u JS-u nije ni približno onome što klasa predstavlja u programskim jezicima kao što su Java i C#. Klasa u Javi i C#-u predstavlja apstraktan opis strukture objekta, dok klasa u JS-u nije ništa drugo, već samo sintaktička pojednostavljenost pristupa na temelju konstruktora.

```
class Klasa {
  constructor() {}
}
let objekt = new Klasa()
```

5.1.2.2 Učahurivanje i skrivanje informacija

Jezgru učahurivanja čine objekti. Učahurivanje predstavlja grupiranje svojstava i metoda u isti objekt, što nam omogućuje da izoliramo određeni dio programske logike. Na taj način možemo pružiti pojednostavljen i razumljiv način korištenja objekta, bez potrebe da se razumije složenost unutrašnjosti te omogućiti jednostavnije upravljanje promjenama. Ukoliko koristimo funkciju (npr. *sort()* u JS-u), ne trebamo poznavati njenu unutarnju implementaciju kako bismo je koristili. Također, promjena unutarnjeg stanja implementacije ne mijenja način na koji ćemo funkciju koristiti [27].

Koncept koji se smatra usko povezan s učahurivanjem je koncept skrivanja informacija. Skrivanje informacija predstavlja način na koji je učahurivanje implementirano [27]. Ideja skrivanja podataka jest da određena struktura otkriva minimalnu količinu informacija koja je nužna da bi se mogla koristiti, dok sve ostalo ostaje skriveno.

```
class Rad {
  #naslov
  #opis
  constructor(naslov, opis) {
      this.#naslov = naslov
      this.#opis = opis
  }

  get naslov() {
```

```
        return this.#naslov
    }

    get opis() {
        return this.#opis
    }

    set naslov(naslov) {
        if (naslov.length !== 0)
          this.#naslov = naslov
        else
          throw Error('Naslov ne može biti prazan.')
    }

    set opis(opis) {
        this.#opis = opis
    }
}
```

5.1.2.3 Apstrakcija

Apstrakcija je koncept u programiranju koji omogućuje pojednostavljivanje kompleksnih procesa, fokusirajući se samo na relevantne značajke za rješavanje određenog problema. To znači da se naglasak stavlja na opće i apstraktne koncepte, a ne na detalje i konkretne implementacije [27].

```
class IstrazivackiRad {
  constructor(naslov, opis) {
      this.naslov = naslov
      this.opis = opis
  }
}

let istrazivackiRad = new IstrazivackiRad()
```

Na primjer, objekt *istrazivackiRad* sadrži samo potrebne podatke, kao što su naslov i opis, dok nam ostale informacije trenutno nisu od značaja.

5.1.2.4 Nasljeđivanje

Koncept nasljeđivanja omogućuje novokreiranom objektu da naslijedi svojstva i metode već postojećih objekata. Kako je JS zapravo prototipno baziran objektno-orijentirani programski jezik, koncept nasljeđivanja implementiran je na drugačiji način. Naime, u JS-u svaki kreirani objekt (ukoliko ne kreiramo objekt na temelju null vrijednosti) nasljeđuje svojstva i metode svima zajedničke konstruktorske funkcije (klase) *Object* [27]. Također možemo kreirati i svoje konstruktorske funkcije/klase koje će biti nasljeđivanje od ostalih.

```
function Rad(naslov, opis) {
  this.naslov = naslov
  this.opis = opis
}

function IstrazivackiRad(naslov, opis, istrazivackaPitanja) {
  Rad.call(this, naslov, opis)
  this.istrazivackaPitanja = istrazivackaPitanja
}

IstrazivackiRad.prototype = new Rad()
```

Međutim, od verzije ECMAScript 2015, omogućeno je korištenje i ključne riječi *extends* koju možemo poistovjetiti s programskim jezikom Java [27].

```
class Rad {
  constructor(naslov, opis) {
      this.naslov = naslov
      this.opis = opis
  }
}

class IstrazivackiRad extends Rad {
  constructor(naslov, opis, istrazivackaPitanja) {
      super(naslov, opis)
      this.istrazivackaPitanja = istrazivackaPitanja
  }
}
```

5.1.2.5 Dinamički polimorfizam

Podloga dinamičkom polimorfizmu u JS-u je nasljeđivanje. Naime, kada JS okruženje naiđe na određeni poziv metode. započinje s traženjem te metode po prototipnom lancu. Ukoliko metodu ne pronađe nad objektom nad kojim je pozvana, započinje s pretragom metode nad roditeljom pozvanog objekta. Kada konačno pronađe traženu metodu, započinje s njezinim izvođenjem [27].

```
class Rad {
  ispisRada() {
      console.log('Rad:' + this.constructor.name)
  }
}

class IstrazivackiRad extends Rad {
  constructor() {
      super()
  }
```

```
  ispisRada() {
      console.log('Istrazivacki:' + this.constructor.name)
  }
}

class ZavrsniRad extends Rad {
  constructor() {
      super()
  }
}
```

5.1.3 FP u JavaScriptu

Kao što već znamo, FP fokusira se na ono *što* bi se trebalo napraviti, a ne *na koji način* bi se to trebalo ostvariti. U većini slučajeva korištenjem FP-a, naš bi se programski kod trebao pojednostaviti i skratiti te postati elegantniji i jednostavniji za testiranje i pronalaženje pogrešaka. Baš zbog svojih karakteristika (karakteristika skriptnih jezika) JS je skloniji pisanju funkcijskim načinom i kao takav podržava različite koncepte čisto-funkcijskih programskih jezika (npr. Haskell). Naravno, JS ne sadrži sve karakteristike programskih jezika vođenih samo funkcijskom paradigmom pisanja programskog koda, ali sadrži one koje nam omogućuju da se ponaša kao takav. Glavne karakteristike FP-a koje su podržane u JS-u jesu sljedeće: funkcije kao građani prve klase, rekurzije, lambda funkcije, zatvaranje i operator širenja (engl. *spread operator*) [16].

5.1.3.1 Anonimne funkcije

Anonimna funkcija je funkcija koja nema naziv i koristi se odmah na mjestu gdje je definirana. Anonimne funkcije mogu se koristiti gotovo svugdje gdje se može koristiti i klasična funkcija. ECMAScript International, od verzije ECMAScript 2015, uvodi koncept funkcije operatorom strelica (=>). Koncept anonimne funkcije/funkcije operatorom strelica u JS-u možemo povezati s konceptom lambda funkcija u FP [16].

```
let brojevi = [1, 2, 3, 4, 5]

brojevi.forEach(function(broj) {
  console.log(broj * 2)
})

brojevi.forEach((broj) => {
  console.log(broj * 2)
})
```

5.1.3.2 Funkcije kao građani prve klase

Funkcije kao građani prve klase znači da ih možemo tretirati kao bilo koji objekt te da ih možemo koristiti na način na koji su i objekti korišteni. To obuhvaća spremanje funkcije u varijablu, prosljeđivanje funkcije kao argument drugoj funkciji, vraćanje funkcije kao rezultat neke druge funkcije, itd [16].

Spremanje funkcije u varijablu:

```
function zbrajanje(x, y) {
  return x + y
}

let a = zbrajanje(2, 2)
let zbroj = zbrajanje
let b = zbroj(2, 2)
console.assert(a === b)
```

Prosljeđivanje funkcije kao argument drugoj funkciji:

```
function zbrajanje(x, y) {
  return x + y
}

function mnozenje(x, y) {
  return x * y
}

function izvrsiOperaciju(operacija, x, y) {
  return operacija(x, y)
}
```

Vraćanje funkcije kao rezultat neke druge funkcije:

```
function zbrajanje(x, y) {
  return x + y
}

function zbrojiDva(x, y) {
  return zbrajanje(x, y)
}
```

Postoji mnogo načina kako kreirati slijedove funkcija i kako ih povezati. No, kako to nije cilj ovoga rada, biti će prikazana samo dva najčešće korištena načina, a to su pomoću kompozicije ili pomoću kariranja.

Primjer kompozicije u programskom jeziku JS:

```
let pribrojiJedan = x => x + 1
```

```
let pomnoziSaDva = x => x * 2
let oduzmiTri = x => x - 3

let kompozicija = x => oduzmiTri(pomnoziSaDva(pribrojiJedan(x)))
```

Primjer kariranja u programskom jeziku JS:

```
function zbroji(x) {
  return function(y) {
     return x + y
  }
}
```

5.1.3.3 Rekurzije

Rekurzija je često korišten koncept prilikom razvoja algoritama. Odnosi se na proces u kojem funkcija poziva samu sebe kao dio svog izvršavanja [16]. U nastavku je prikazan primjer računanja n-tog faktorijela putem rekurzija u JS-u.

```
function fakt(n) {
  if (n === 0) {
     return 1
  } else {
     return n * fakt(n - 1)
  }
}
```

5.1.3.4 Zatvaranje

Korištenjem koncepta zatvaranja možemo na jednostavniji način kreirati čiste funkcije. Razlog tome je taj što zatvaranje omogućuje da se funkcija osloni samo na svoje ulazne parametre i vrijednosti koje drži u svojim okvirima, a ne na bilo kakvo vanjsko stanje programa koje bi moglo utjecati na njezin izlaz [16].

```
function kreirajBrojac() {
  let brojac = 0

  function inkrementiraj() {
     brojac++
  }

  return inkrementiraj
}
```

5.1.3.5 Operator širenja

Kako se FP zasniva ne nepromjenjivosti podataka, operator širenja nam omogućuje kreiranje novih objekata na temelju prijašnjih elemenata, ali bez promjene stanja prijašnjih

elemenata. Sličan rezultat možemo dobiti i s ključnom riječi *const* kod deklaracije varijabli, gdje onemogućujemo promjenjivost vrijednosti varijable [16]. No, nepromjenjivost korištenjem ključne riječi *const* nije uvijek ispravno rješenje iz razloga ukoliko je varijabla zapravo objekt, recimo polje, korištenjem ključne riječi *const* onemogućujemo promjenu vrijednosti određene varijable, ali ne i promjenu vrijednosti elemenata polja.

```
let brojevi1 = [1, 2, 3, 4, 5]
let brojevi2 = [...brojevi1, 6]


const brojevi1 = [1, 2, 3, 4, 5]
brojevi1 = [1, 2, 3] // Pogreska
brojevi1[2] = 6
```

5.2 TypeScript

Andres Hejlsberg (autor C# programskog jezika) je 2012. godine započeo s projektom otvorenog koda (engl. *open source*) koji je nazvan TypeScript. Programski jezik TS zapravo predstavlja samo jedan omotač oko programskog jezika JS, čime pospješuje njegovu sintaksu, uvodi sigurnost tipova podataka i još mnogo drugih funkcionalnosti. Ali što to zapravo znači? Naime, programski jezik TS zapravo se kompajlira u određenu verziju JS koda koju definiramo prilikom kompajliranja TS programskog koda.

Općenito, sam proces kompajliranja započinje tako što kompajler pretvori programski kod u apstraktno sintaksno stablo (engl. *Abstract Syntax Tree*, *AST*) - struktura podataka koja zanemaruje znakove poput razmaka, komentara i tabovima i slično. Kompajler zatim pretvara AST u nižu razinu predstavljanja nazvanu *bytecode*, koja predstavlja vrstu binarnog koda koja je neovisna o okruženju u kojem se nalazi. Znači, kada pokrenemo program, kažemo okruženju u kojem izvodimo programski kod, da evaluira *bytecode* generiran od strane kompajlera. Jedina je razlika kod kompajliranja TS koda ta što se TS ne kompajlira u *bytecode* već u JS kod. Još jedan dodatak TS-a je taj, da prije generiranja AST-a, TS napravi provjeru tipova podataka i zatim taj JS kod izvršavamo [28].

5.2.1 TypeScript kao nadogradnja JavaScripta

Programski jezik TS osmišljen je s razlogom da donese promjene u programski jezik JS, kako bi upotpunio njegove nedostatke. Prednosti koje TS donosi su sljedeće: kompajliranje, nasljeđivanje, jaka tipizacija, definicije tipova (engl. *definition types*), otvorena rekurzija (engl. *open recursion*), delegacije (engl .*delegations*) i generičko programiranje.

5.2.1.1 Kompajliranje

Kako smo već upoznati s načinom na koji JS interpreterski program funkcionira, postane znatno frustrirajuće kako, svaki puta kada želimo utvrditi pogrešku, JS interpreter mora doći u pokušaj njene interpretacije, a što može znatno utjecati na vrijeme pronalaženja pogrešaka u programskom kodu. TS kompajlator nudi nam mogućnost statičkog analizatora koda (engl. *static code analyzer*), što omogućuje detektiranje sintaktičkih pogrešaka za vrijeme pisanja programskog koda. Time možemo znatno ubrzati proces implementacije i testiranja samog programskog koda.

5.2.1.2 Jaka tipizacija

Kao što je već spomenuto, JS je dinamički (engl. *dynamic*) programski jezik - program se može mijenjati i prilagođavati zahtjevima korisnika prilikom izvršavanja. S druge strane, programski jezik TS je statički (engl. *static*) programski jezik te kao takav nije podložan dinamičkoj promjeni tipova podatak prilikom izvršavanja programa, a što ga čini jako tipiziranim (engl. *strong typed*) programskim jezikom.

Primjer slabe tipizacije u programskom jeziku JS:

```
let primjer = 'String'
primjer = 5
primjer = (x, y) => x + y
```

Primjer jake tipizacije u programskom jeziku TS:

```
let primjer: string = 'String'
primjer = 5 // Pogreska
primjer = (x, y) => x + y // Pogreska
```

Kako bi programski jezik TS ostao kompatibilan s JS-om, omogućio je korištenje tipa podataka *any* koji se ponaša na identičan način kao i dinamičko izvršavanje u JS-u. No, preporučuje se izbjegavanje korištenja ključne riječi *any* baš zbog veće sklonosti na pogreške u programskom jeziku JS [29]. U nastavku slijedi primjer slabe tipizacije i u programskom jeziku TS.

```
let primjer: any = 'String'
primjer = 5 // Pogreska
primjer = (x: any, y: any) => x + y // Pogreska
```

5.2.1.3 Definicije tipova

Sada, kada smo zaključili kako TS uvodi jaku tipizaciju postavlja se pitanje kako onda integrirati postojeće JS biblioteke u TS programski kod. Ovaj se problem rješava kreiranjem definicijske datoteke (engl. *definition file*). Definicijska datoteka ima ekstenziju *.d.ts* te je

možemo poistovjetiti s datotekom zaglavlja (engl. *header*) u programskom jeziku C++ [29]. Dakle, datoteke definicija sadrže informacije koje opisuju svaku dostupnu funkciju i/ili varijable, zajedno s njihovim povezanim tipovima.

5.2.1.4 Otvorena rekurzija

Otvorena rekurzija je tehnika koja omogućava funkciji ili objektu da referencira samog sebe putem svojih svojstava ili metoda [29]. Iako koncept otvorene rekurzije postoji i u programskom jeziku JS, zbog niske je tipizacije programskog jezika izbjegavana. No, kako TS omogućava sigurnost tipova podataka, pronašla je svoju primjenu u području definiranja složenih i sigurnih (engl. *type-safe*) struktura.

```
type BinarnoStablo = {
  vrijednost: number
  lijevo?: BinarnoStablo
  desno?: BinarnoStablo
}
```

5.2.1.5 Delegacije

Isto kao i kod koncepta otvorene rekurzije, JS također može implementirati uzorak dizajna delegacija (engl. *delegation design pattern*), no kako TS omogućava korištenje sučelja (engl. *interfaces*), zato što osigurava ispoštivanje ugovora između delegirajućeg i i delegiranog objekta, znatno je sigurniji za implementirati [29].

```
interface IBiljeznik {
  obavijesti(message: string): void
}

class ConsoleLogBiljeznik implements IBiljeznik {
  obavijesti(poruka: string) {
      console.log(poruka)
  }
}

class ConsoleInfoBiljeznik implements IBiljeznik {
  obavijesti(poruka: string) {
      console.info(poruka)
  }
}

class Biljeznik {
  private biljeznik: IBiljeznik

  constructor(logger: IBiljeznik) {
```

```
        this.biljeznik = logger
    }

    log(poruka: string) {
        this.biljeznik.obavijesti(poruka)
    }
}
```

5.2.1.6 Nasljeđivanje

Radi zbunjujućeg koncepata prototipa i njihovog nasljeđivanja u programskom jeziku JS, TS uvodi koncept apstraktnih klasa koje su danas neizostavne u klasičnom objektno-orijentiranom programiranju.

```
abstract class Biljeznik {
    abstract obavijesti(poruka: string): void

    protected zajednickaObavijest(): void {
        console.log('Zajednicka obavijest')
    }
}

class ConsoleLogBiljeznik extends Biljeznik {
    obavijesti(poruka: string) {
        console.log(poruka)
    }
}

class ConsoleInfoBiljeznik extends Biljeznik {
    obavijesti(poruka: string) {
        console.info(poruka)
    }
}
```

5.2.1.7 Generičko programiranje

Generičko programiranje omogućuje definiranje struktura na način da se tipovi podataka mogu specificirati kasnije. To omogućuje da se razni tipovi podataka obrađuju identično, bez izostavljanja sigurnosti tipova ili zahtijevanja zasebnih instanci algoritma za svaki tip. U TS-u je moguće kreirati generičke funkcije, sučelja i klase. Generičko programiranje baš zbog svoje modularnosti pronalazi područja primjene i kod OOP-a i kod FP-a [29].

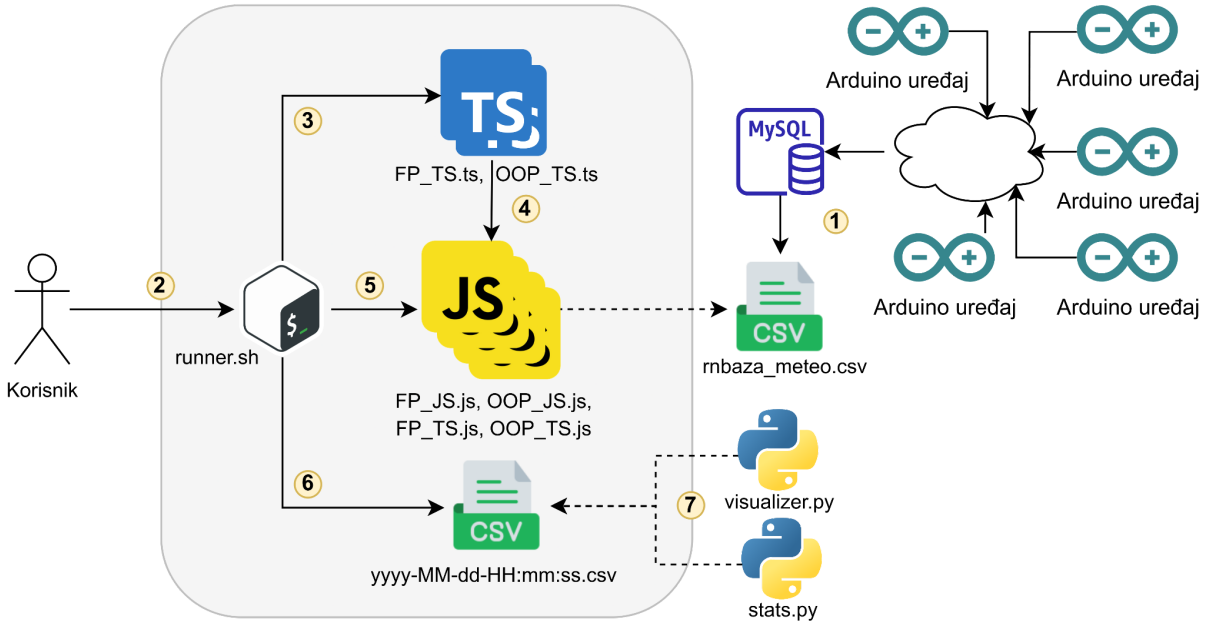
Primjer generičke funkcije u TS:

```
function obrni <T> (elementi: T[]): T[] {
```

```
    return elementi.reverse();
}
```

5.3 Pregled arhitekture

Pregled arhitekture eksperimenta na najapstraktnijoj razini prikazan je pomoću dijagrama u nastavku. Redni brojevima (1 - 7) označen je redoslijed radnji vezanih uz domenu ovog eksperimenta - način pripreme, pokretanja te generalni tok podataka.



Slika 3: Pregled arhitekture eksperimenta na najapstraknijoj razini

Meteorološki se podaci svakodnevno šalju s 5 različitih Arduino uređaja s različitih fizičkih lokacija. Podaci o temperaturi, vlazi i tlaku zraka pohranjeni su u obliku MySQL baze podataka, nad kojom je napravljen izvoz istih - *rnbaza_meteo.csv* datoteka. Testna CSV datoteka sadrži 447193 meteoroloških zapisa o temperaturi, vlazi i tlaku zraka.

Nakon što se eksperiment provede pomoću *runner.sh* Bash skripte, za vizualizaciju dobivenih mjerenja i izračun statističkih podataka koriste se dvije Python3 skripte - *visualizer.py* i *stats.py* respektivno.

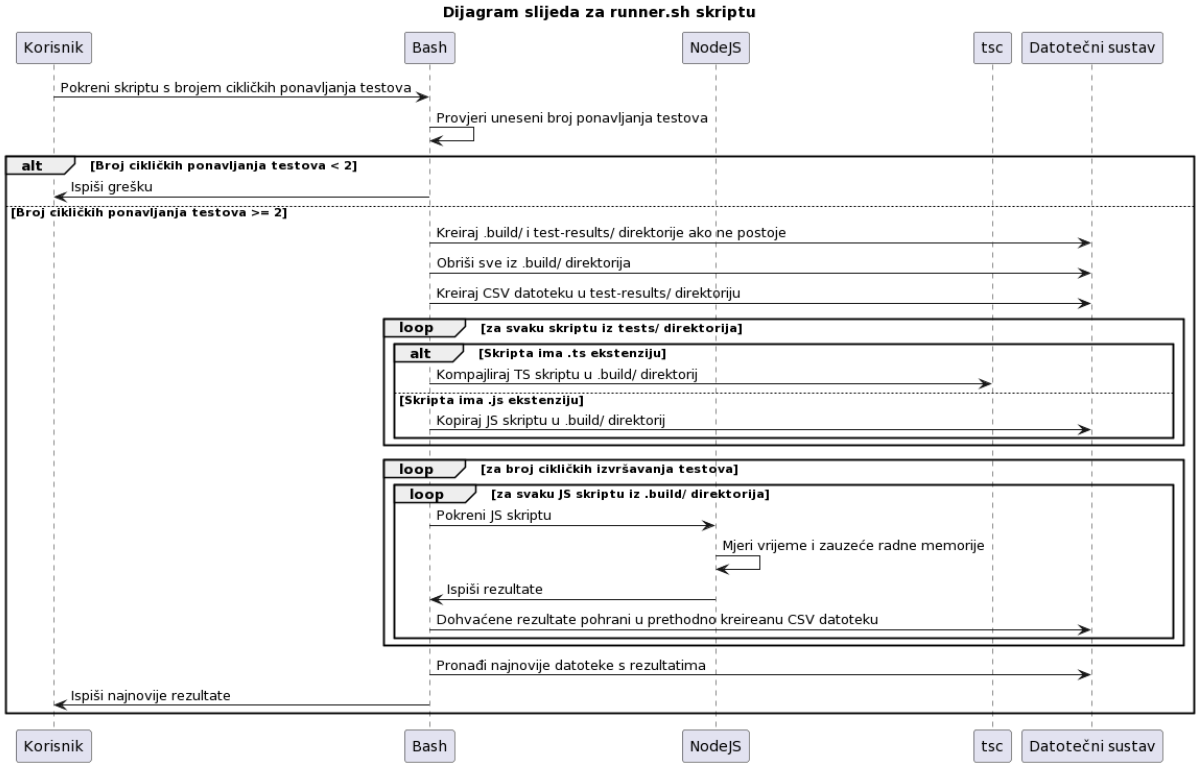
Struktura direktorija, zajedno sa svim potrebnim komponentama i njihovim uputama za korištenje, dostupna je putem GitHub repozitorija: https://github.com/ttomasicc/analysis-fp-oop. Glavni direktoriji i njihove predefinirane datoteke uključuju, ali nisu limitirani na:

```
.
├── .build
├── data
│   └── rnbaza_meteo.csv
├── runner.sh
├── stats.py
├── test-results
├── tests
│   ├── FP_JS.js
│   ├── FP_TS.ts
│   ├── OOP_JS.js
│   └── OOP_TS.ts
└── visualizer.py
```

Kako bi se automatiziralo testiranje, odnosno izvršavanje eksperimenta, kreirana je kratka Bash skripta - *runner.sh*. Kao obavezni parametar prima pozitivan broj cikličkih pokretanja testnih JS skripata, a njezina je uloga da:

- pripremi testno okruženje - kreira potrebne direktorije (*.build* i *test-results*) ukoliko već ne postoje i briše prethodni skup testnih skripata (*.build* direktorij),
- pripremi skup testnih skripata - kompajlirane TS skripte i JS skripte kopira u *.build* direktorij,
- ciklički pokreće sve testne JS skripte iz *.build* direktorija zadani broj puta te
- u *test-results* direktoriju, u CSV formatu zapisuje vremensku i prostornu složenost svakog izvođenja odgovarajuće skripte na način da pročita prva dva izlaza iz svake skripte.

U nastavku je prikazan dijagram slijeda koji pobliže prikazauje interakciju između *runner.sh* Bash skripte i različitih ostalih komponenti u procesu automatskog izvođenja eksperimenta. Glavni su učesnici korisnik, Bash skripta, NodeJS, TS kompajler (tsc) i datotečni sustav. Za kompajliranje TS skripata, koristi se zastavica `--target es2022` s kojom označavamo kompajleru da koristi zadnju, trenutno najnoviju verziju ES standarda [30].



Slika 4: Dijagram slijeda za runner.sh Bash skriptu

5.4 Ulaz i izlaz algoritma

Kao ulaz u algoritam očekuje se CSV datoteka s meteorološkim podacima u sljedećem formatu:

```
ident;vrijemeTekst;temp;vlaga;tlak
<lokacija>;<datumVrijeme>;<temperatura>;<vlaga>;<tlak>
```

, gdje

- *lokacija* označava tekstualni naziv lokacije uređaja (npr. ‘watss_3’),
- *datumVrijeme* označava tekstualni zapis datuma i vremena očitanja uređaja u formatu ‘d.m.yyyy. HH:mm:ss’ (npr. ‘17.2.2023. 20:32:30’),
- *temperatura* označava očitanu decimalnu vrijednost temperature u Celzijevim stupnjevima (npr. 10.2),
- *vlaga* označava očitanu decimalnu vrijednost, postotak vlage u zraku (npr. 74.2) i
- *tlak* označava očitanu decimalnu vrijednost tlaka zraka u hektopaskalima (npr. 1003.3)

Svaki uređaj ima mogućnost mjerenja trenutne temperature, no samo neki posjeduju mogućnost mjerenja i vlage i tlaka zraka. Ukoliko uređaju nedostaje senzor za mjerenje vlažnosti zraka i/ili senzor za mjerenje tlaka zraka, njegova je vrijednost, odnosno nedostatak očitane vrijednost, naznačena brojem -999.

Svaka testna skripta obrađuje meteorološke podatke na način da ih najprije grupira po lokacijama uređaja, a zatim po danima. Za svaki je dan zatim izračunata statistika koja uključuje minimalnu, maksimalnu i prosječnu vrijednost za svaku od karakteristika (temperatura, vlaga i tlak zraka). Ukoliko bilo koju vrijednost karakteristike nije moguće izračunati, ona će u JSON formatu imati *null* vrijednost. Dodatno primijenjeno ograničenje je da sve testne skripte imaju jednak izlaz - obrađeni meteorološki podaci su u sljedećem JSON formatu:

```
{
  "watss[1-5]": {
    "<dd.mm.yyyy>": {
      "minTemp": number,
      "maxTemp": number,
      "avgTemp": number,
      "minHumidity": number | null,
      "maxHumidity": number | null,
      "avgHumidity": number | null,
      "minPressure": number | null,
      "maxPressure": number | null,
      "avgPressure": number | null
    },
    ...
  },
  ...
}
```

5.5 Opis rada algoritma obrade podataka

U ovom su poglavlju opisani detalji implementacije algoritama obrade meteoroloških podataka kroz dvije promatrane paradigme - FP i OOP. Za primjer obje paradigme, uzet će se JS varijante zbog jednostavnosti objašnjenja i manje programskog koda.

Algoritmi obrade podataka u TS-u implementirani su koristeći identične, istoimene funkcije, klase i metode kao u JS varijanti, uz naravno, nadopunu tipova koju TS zahtijeva. Treba napomenuti kako se ipak, zbog toga što TS nameće tipove, može ponegdje dogoditi malo odstupanje u samoj implementaciji TS algoritma. Međutim, razina vremenske, a gotovo i prostorne složenosti, zadržana je na jednakoj razini kao i u JS varijanti.

5.5.1 Implementacija algoritma koristeći koncepte FP-a

U nastavku je dana definicija glavne *main* funkcije, čijim pozivom na samom kraju skripte, počinje izvršavanje *FP_JS.js* datoteke.

```
function main() {
  const meteoData = loadMeteoData(meteoDataFilePath)

  const startTime = new Date()
  const startMemory = process.memoryUsage.rss()

  const groupedMeteoData = groupMeteoData(meteoData)
  const processedMeteoData = processMeteoData(groupedMeteoData)

  const endTime = new Date()
  const endMemory = process.memoryUsage.rss()

  console.info(endTime - startTime)
  console.info(endMemory - startMemory)

  console.info(JSON.stringify(processedMeteoData))
}
```

Možemo vidjeti da se na početku izvođenja skripte najprije učitaju svi meteorološki podaci iz *rnbaza_meteo.csv* datoteke u RAM te je to vrijeme izuzeto iz mjerenja vremenske složenosti.

Za učitavanje podataka u RAM koristi se *loadMeteoData()* funkcija koja kao parametar prima putanju do *rnbaza_meteo.csv* datoteke - vrijednost sadržana u konstanti *meteoDataFilePath*, definiranoj na početku skripte. Za čitanje CSV datoteke koristi se blokirajuća metoda *readFileSync()* iz *fs* modula [31].

```
function loadMeteoData(filePath) {
  try {
    return fs.readFileSync(filePath, 'utf-8')
  } catch (err) {
    console.error(err)
    process.exit(1)
  }
}
```

Nakon što su meteorološki podaci učitani u RAM, zabilježi se trenutno vrijeme (konstanta *startTime*) i trenutno zauzeće RAM-a, odnosno RSS (konstanta *startMemory*). Zatim se pozivaju funkcije *groupMeteoData()* i *processMeteoData()* čije se vrijeme izvršavanja i potrošnja RAM-a promatra, te se na kraju, nakon obrade podataka, ponovno zapiše trenutno vrijeme (konstanta *endTime*) i trenutno zauzeće RAM-a (konstanta *endMemory*). Konačno, skripta kao izlaz ispiše vrijeme trajanja obrade podataka u milisekundama, veličinu zauzetog RAM-a u bajtovima (RSS) i obrađene meteorološke podatke, grupirane po lokacijama uređaja i danima unutar njih, u JSON formatu.

Funkcija *groupMeteoData()* koristi se u svrhu grupiranja meteoroloških podataka po lokaciji uređaja, a zatim po svakom zasebnom danu unutar promatrane lokacije.

```
function groupMeteoData(meteoData) {
  const groupedBySensorLocation = groupBy(
    formatCSV(meteoData).slice(1),
    (row) => row[0]
  )

  const groupedBySensorLocationAndDay = {}
  Object.keys(groupedBySensorLocation)
    .forEach((sensorLocation) => {
      groupedBySensorLocationAndDay[sensorLocation] = groupBy(
        groupedBySensorLocation[sensorLocation],
        (row) => row[1].split(' ')[0]
      )
    })

  return groupedBySensorLocationAndDay
}
```

Najprije se, pomoću povezivanja funkcija[6] *formatCSV()* i *groupBy()*, meteorološki podaci grupiraju po lokaciji uređaja (konstanta *groupedBySensorLocation*). Potom se, pomoću *Object.keys().forEach()* iteracije prolazi kroz svaku dobivenu lokaciju (ključevi objekta *groupedBySensorLocation*) i ponovno grupira, ali po atributu vremena - tako da se odvoje mjerenja za pojedinačne dane unutar lokacije uređaja.

Funkcija *formatCSV()* zapravo je anonimna funkcija, dodijeljena konstanti *formatCSV*, a koja kao argument prima čisti string (engl. *raw string*) cijele *rnbaza_meteo.csv* datoteke:

```
const formatCSV = (data) =>
  data.split('\n').filter(Boolean).map(line => line.split(';'))
```

Anonimna funkcija vraća niz koji sadrži nizove u kojima su redom odvojeni meteorološki podaci:

```
[
  [ 'ident', 'vrijemeTekst', 'temp', 'vlaga', 'tlak' ],
  [ 'watss_2', '17.2.2023. 16:24:31', '14.4', '50', '1003.8' ],
  [ 'watss_2', '17.2.2023. 16:24:31', '14.8', '-999', '-999' ],
  [ 'watss_1', '17.2.2023. 16:24:17', '23.5', '36.9', '-999' ],
  ...
]
```

Iz tog se rezultata zatim, pomoću *slice()* metode, miče zaglavlje CSV datoteke i proslijeđuje funkciji *groupBy()*.

Funkcija *groupBy()*, u kontekstu FP-a, predstavlja funkciju višeg reda (HOF) jer kao drugi argument prima funkciju koja služi za dohvaćanje ključa, odnosno atributa po kojem se

[6] Izlaz jedne funkcije prenosi se kao ulaz u sljedeću funkciju, bez eksplicitnog spremanja međurezultata u varijablu.

nizovi žele grupirati. Za grupiranje nizova koristi *Map* strukturu podataka, gdje kao ključ uzima element dohvaćen pomoću proslijeđene funkcije (*attrGetterFn*), a kao vrijednost koristi niz u kojem su ugnježđeni (cijeli) nizovi koji sadrže zadani ključ. Odnosno, ukoliko grupiramo podatke po lokaciji uređaja, ključ u *Map* strukturi biti će lokacija uređaja, dok će vrijednost na zadanom ključu biti niz koji sadrži nizove gdje se pojavljuje odgovarajuća lokacija uređaja. Na kraju se tako grupirani podaci prebacuju iz *Map* strukture u klasični JS objekt koristeći statičku metodu *fromEntries()* iz *Object* klase:

```
function groupBy(data, attrGetterFn) {
  const groups = new Map()

  data.forEach((meteoData) => {
    let id = attrGetterFn(meteoData)
    groups.has(id) ? groups.get(id).push(meteoData) :
      groups.set(id, [meteoData])
  })

  return Object.fromEntries(groups)
}
```

Nakon što su svi meteorološki podaci grupirani po odgovarajućim atributima, prelazi se u funkciju *processMeteoData()* čija je svrha računanje statistike meteoroloških podataka za sve grupirane dane unutar svih lokacija uređaja.

```
function processMeteoData(meteoData) {
  const processedMeteoData = {}

  Object.keys(meteoData).forEach((sensorLocation) => {
    processedMeteoData[sensorLocation] =
      meteoData[sensorLocation]
    Object.keys(meteoData[sensorLocation]).forEach((day) =>
      processedMeteoData[sensorLocation][day] =
        getDayStatistics(meteoData[sensorLocation][day]))
  })

  return processedMeteoData
}
```

Kao što možemo vidjeti, funkcija *processMeteoData()* također spada u čiste funkcije, a u implementaciji prolazi kroz sve lokacije uređaja i sve dane unutar njih, te za svaki dan poziva funkciju *getDayStatistics()*.

Funkcija *getDayStatistics()* na početku kreira klasični JS objekt koji sadrži sve potrebne atribute računanja “statistike” meteoroloških podataka po svakom danu. To uključuje minimalnu, maksimalnu i prosječnu vrijednost svih karakteristika uređaja - temperatura, vlaga i tlak zraka. Minimalne se vrijednosti na početku postavljaju na Infinity, a

maksimalne na -Infinity, kako bi se kasnije mogle koristiti statičke metode min() i max() iz Math klase.

```
const statistics = {
 minTemp: Infinity, maxTemp: -Infinity, avgTemp: NaN,
 minHumidity: Infinity, maxHumidity: -Infinity, avgHumidity: NaN,
 minPressure: Infinity, maxPressure: -Infinity, avgPressure: NaN
}
```

Funkcija *getDayStatistics()*, zbog uštede na vremenskoj i prostornoj složenosti, sve potrebne atribute računa u jednoj iteraciji - prolazi kroz niz meteoroloških podataka za zadani dan samo jednom:

```
day.forEach((reading) => {
  const temp = parseFloat(reading[2])
  tempTotal += temp
  tempTotalCount++
  statistics.minTemp = Math.min(statistics.minTemp, temp)
  statistics.maxTemp = Math.max(statistics.maxTemp, temp)

  if (reading[3] !== '-999') {
    const humidity = parseFloat(reading[3])
    humidityTotal += humidity
    humidityTotalCount++
    statistics.minHumidity =
      Math.min(statistics.minHumidity, humidity)
    statistics.maxHumidity =
      Math.max(statistics.maxHumidity, humidity)
  }

  if (reading[4] !== '-999') {
    const pressure = parseFloat(reading[4])
    pressureTotal += pressure
    pressureTotalCount++
    statistics.minPressure =
      Math.min(statistics.minPressure, pressure)
    statistics.maxPressure =
      Math.max(statistics.maxPressure, pressure)
  }
})
```

Prosječne se vrijednosti računaju na način da *forEach()* iteracija u svakom koraku također broji ukupne zbrojeve svih karakteristika i njihovog ponavljanja, *[temp | humidity | pressure]Total* i *[temp | humidity | pressure]TotalCount* varijable koje su inicijalno postavljene na 0. Na primjer:

```
if (tempTotalCount !== 0)
  statistics.avgTemp = tempTotal / tempTotalCount
```

Kao rezultat obrade meteoroloških podataka po danu, funkcija *getDayStatistics()* na kraju vraća modificiranu konstantu *statistics* - najprije se pretvara u string reprezentaciju pomoću statičke *JSON.stringfy()* metode, a zatim natrag u objektnu reprezentaciju pomoću statičke metode *JSON.parse()*. Time dobivamo konzistentne rezultate statistike za one vrijednosti koje ne postoje / nisu izračunate - *Infinity, -Infinity* i *NaN* postaju *null*.

U globalu, možemo vidjeti kako se u ovom algoritmu obrade meteoroloških podataka velika većina koncepata FP-a ispoštovala. Međutim, nužno je za napomenuti kako su se neki koncepti poput nepromjenjivosti podataka ipak narušili. Razlog je tome ušteda na prostornoj i vremenskoj složenosti - modifikacijom već postojećih objekata skraćujemo vrijeme potrebno za kreiranje / kopiranje objekata, a koje u konačnici i smanjuje prostornu složenost jer ne zauzimamo nove memorijske lokacije potrebne kod dinamičkog kreiranja novih objekata, nego iskorištavamo već postojeću zauzetu radnu memoriju računala.

5.5.2 Implementacija algoritma koristeći koncepte OOP-a

Kao što je već i prethodno spomenuto, OOP varijanta implementacije algoritma obrade statističkih podataka ne razliku se značajno od FP varijante. Stoga ćemo se ovdje fokusirati samo na značajnije razlike, a to je uvođenje klasa; objekata i metoda. Također, *forEach()* iteracije idiomatske za FP, zamijenjene su klasičnim *for-of* i *for-in* JS petljama

Za početak, svaki je red iz CSV datoteke *rnbaza_meteo.csv* predstavljen klasom *MeteoData*. Klasa sadržava 5 javnih atributa - *id* (string), *date* (Date), *temperature* (number), *humidity* (number) i *pressure* (number) - te jedan konstruktor gdje se kao parametar prima jedan red iz CSV datoteke. Dobiveni se red zatim pomoću destruktivnog raspakiravanja[7] (engl. *destructive assignment*) preko ";" separatora dodijeli prethodno spomenutim atributima.

```
class MeteoData {
  ...
  constructor(row) {
    [this.id, this.date, this.temperature, this.humidity,
      this.pressure] = row.split(';')
    this.#convertTypes()
  }
  ...
}
```

Nakon što smo inicijalizirali atribute klase, potrebno ih je pretvoriti u odgovarajuće tipove podataka radi lakše kasnije manipulacije. Pretvorba tipova podataka atributa klase *MeteoData* apstrahirana je i izdvojena u zasebnu privatnu metodu *convertTypes()* koja *date*

[7] JS izraz koji omogućuje raspakiravanje vrijednosti iz nizova ili atributa objekata u različite varijable.

atribut pretvara u objekt klase Date i za to koristi drugu privatnu metodu *convertDate()*, a vrijednosti temperature, vlage i tlaka zraka parsira u brojčani oblik (number). Ukoliko očitanje vlage ili tlaka ne postoji (predstavljeno vrijednošću “-999”), vrijednost odgovarajućeg atributa postaje *NaN*.

```
class MeteoData {
  ...
  #convertTypes() {
    this.date = this.#convertDate()
    this.temperature = parseFloat(this.temperature)
    this.humidity = this.humidity !== '-999' ?
      parseFloat(this.humidity) : NaN
    this.pressure = this.pressure !== '-999' ?
      parseFloat(this.pressure) : NaN
  }

  #convertDate() {
    const [datePart, timePart] = this.date.split(' ')
    const [date, month, year] = datePart.split('.')
    const [hours, minutes, seconds] = timePart.split(':')
    return new Date(
      parseInt(year), parseInt(month) - 1, parseInt(date),
      parseInt(hours), parseInt(minutes), parseInt(seconds)
    )
  }
}
```

Nadalje, za obradu meteoroloških podataka uvedena je klasa *MeteoProcessor* s jednim privatnim dinamičkim atributom *meteoData*. Klasa apstrahira već poznate funkcionalnosti:

- parsiranje meteoroloških podataka iz CSV datoteke - privatna metoda *parseCSV()*,
- grupiranje podataka - privatna metoda *getGroupedBySensorLocationAndDay()*, a koja grupiranje obavlja uz pomoć metode *groupBy()* i
- računanje statistike meteoroloških podataka po danu - javna metoda *getStatistics()*, uz pomoć privatne metode *getDailyStatistics()*.

```
class MeteoProcessor {
  constructor(rawMeteoData, hasHeaders) {
    this.#meteoData = this.#parseCSV(rawMeteoData, hasHeaders)
  }

  getStatistics() { ... }

  #getDailyStatistics(day) { ... }
```

```
  #getGroupedBySensorLocationAndDay() { ... }
  #groupBy(data, attrGetterFn) { ... }

  #parseCSV(data, hasHeaders) { ... }
}
```

Treća klasa, *MeteoStatistics*, uvedena je sa svrhom da zamijeni JSON konstantu *statistics* iz FP varijante implementacije algoritma:

```
class MeteoStatistics {
  minTemp = Infinity
  maxTemp = -Infinity
  avgTemp = NaN
  minHumidity = Infinity
  maxHumidity = -Infinity
  avgHumidity = NaN
  minPressure = Infinity
  maxPressure = -Infinity
  avgPressure = NaN
}
```

Shodno navedenim promjenama, *main* funkcija sada učitava meteorološke podatke putem iste *loadMeteoData()* funkcije, instancira objekt *MeteoProcessor* klase kojem prosljeđuje učitane podatke zajedno sa zastavicom da učitani set podataka sadrži zaglavlje te poziva *getStatistics()* metodu nad prethodno instanciranim objektom - konstanta *meteoProcessor*.

```
function main() {
  const meteoData = loadMeteoData(meteoDataFilePath)

  const startTime = new Date()
  const startMemory = process.memoryUsage.rss()

  const meteoProcessor = new MeteoProcessor(meteoData, true)
  const processedMeteoData = meteoProcessor.getStatistics()

  const endTime = new Date()
  const endMemory = process.memoryUsage.rss()

  console.info(endTime - startTime)
  console.info(endMemory - startMemory)

  console.info(JSON.stringify(processedMeteoData))
}
```

6. Analiza rezultata

U ovom su poglavlju dani statistički rezultati analize mjerenja vremenske i prostorne složenosti algoritama za obradu meteoroloških podataka u funkcijskoj i objektno-orijentiranoj paradigmi, koristeći programski jezik JS i njegov tipizirani pandan TS.

6.1 Inicijalni pristup mjerenju

Kako bismo dobili statistički odgovor na glavno istraživačko pitanje, postavljeno u drugom poglavlju ovog rada, nastojimo statistički testirati vremensku i prostornu složenost izvedbe algoritama u funkcijskoj i objektno-orijentiranoj varijanti, koristeći JS i TS programske jezike. Glavne postavljene hipoteze navedene su u nastavku.

Vremenska složenost

H_0: Nema statistički značajne razlike u prosječnom rezultatu mjerenja vremena izvođenja između FP-a u JS-u i OOP-a u JS-u.

H_A: Postoji statistički značajna razlika u prosječnom rezultatu mjerenja vremena izvođenja između FP-a u JS-u i OOP-a u JS-u.

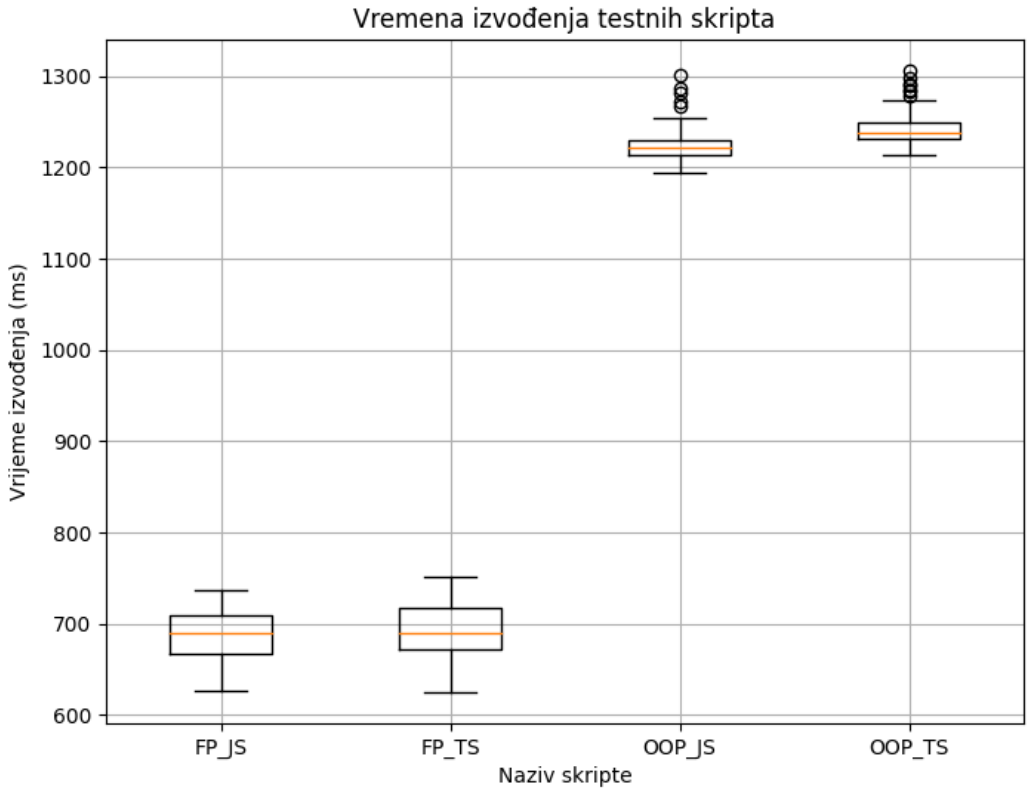
Prostorna složenost

H_0: Nema statistički značajne razlike u prosječnom rezultatu mjerenja zauzeća radne memorije računala između FP-a u JS-u i OOP-a u JS-u.

H_A: Postoji statistički značajna razlika u prosječnom rezultatu mjerenja zauzeća radne memorije računala između FP-a u JS-u i OOP-a u JS-u.

6.1.1 Vremenska složenost

Na dijagramu pravokutnika (slika 5) možemo vidjeti inicijalne rezultate mjerenja vremenske složenosti za četiri testirane skripte - implementacije algoritma obrade meteoroloških podataka. Na osi apscisa redom su pobrojan nazivi četvero testiranih skripata, dok je na osi ordinata označeno vrijeme izvođenja pojedine skripte u milisekundama.



Slika 5: Dijagram pravokutnika za vremensku složenost četiri slučaja implementacije algoritma obrade meteoroloških podataka

Možemo uočiti kako se kod funkcijske implementacije u oba slučaja interkvartilni raspon proteže od otprilike 670ms do 720ms. Razina donjeg brka u oba slučaja nalazi se na otprilike 630ms, dok se gornji brk kod funkcijske implementacije u JS-u proteže do otprilike 720ms, a u TS varijanti do otprilike 750ms. Medijan se u oba slučaja nalazi na razini od 680ms.

U oba slučaja imamo asimetričnu distribuciju podataka. U slučaju JS varijante, radi se o blagoj lijevoj iskrivljenosti (engl. *skewness*) dok se u TS varijanti može vidjeti jasna desna iskrivljenost distribucije podataka.

Kod funkcijske implementacije ne dolazi do pojave stršila. Međutim, u objektno-orijentiranoj implementaciji asimetričnost distribucije podataka znatno je naglašenija te shodno tome dolazi do pojave stršila - zaključujemo da se vrijeme izvođenja u nekoliko situacija *neočekivano* produžilo.

Kod objektno-orijentirane implementacije u JS-u, medijan se nalazi na otprilike 1220ms. Gornji brk nalazi se na 1250ms, dok se donji nalazi na otprilike 1190ms. U TS varijanti objektno-orijentirane implementacije, medijan je nešto veći - približno 1240ms. Gornji se brk nalazi na otprilike 1260ms, a donji na otprilike 1210ms.

Iako možemo naslutiti da podaci, u svim varijantama implementacije algoritma, nisu normalno distribuirani (nepravilan oblik pravokutnika, medijana i postojanje stršila), potrebno je napraviti formalni test normalnosti skupa uzorka kako bismo to zaista i statistički potvrdili. Jedan od takvih testova je Shapiro-Wilk test. U nastavku su navedene njegove hipoteze, zajedno s rezultatima provedbe istog koristeći *stats.py* skriptu.

H_0: Ne postoji značajno odstupanje od normalne distribucije.

H_A: Postoji značajno odstupanje od normalne distribucije.

Skripta	Shapiro-Wilk p vrijednost	Normalna distribucija
FP_JS	0.009311062283813953	Ne
FP_TS	0.016165832057595253	Ne
OOP_JS	6.05762053851322e-08	Ne
OOP_TS	3.7436652178257646e-07	Ne

Ovime smo potvrdili da su sve p vrijednosti manje od 0.05, odnosno da ni jedan skup uzoraka mjerenja brzine izvođenja skripata FP_JS, FP_TS, OOP_JS i OOP_TS nije normalno distribuiran - odbacujemo nul-hipoteze.

Shodno tome, za statističku analizu usporedbe rezultata, umjesto ANOVA-e s ponovljenim mjerenjem, koristit ćemo njezinu neparametrijsku alternativu - Friedmanova ANOVA test. Sve su pretpostavke Friedmanove ANOVA-e zadovoljene pa se može preći u postupak provedbe. Kao hipoteze uzet ćemo:

H_0: Ne postoji statistički značajna razlika u vremenima izvođenja bilo kojeg para testnih skripata.

H_A: Postoji statistički značajna razlika u vremenima izvođenja barem jedne testne skripte.

Nakon provedene Friedmanove ANOVA-e, dobivamo da je p vrijednost jednaka $4.6702060544967294 \times 10^{-55}$. Uz razinu značajnosti od 95% možemo odbaciti nultu hipotezu te donijeti zaključak da postoji statistički značajna razlika u vremenu izvođenja barem jedne testne skripte.

Kako bismo otkrili između kojih testnih skripata postoji statistički značajna razlika u vremenu izvođenja, poslužit ćemo se Nemenyi post hoc testom. Rezultati usporedbe testnih skripata temeljem vremena izvođenja dani su u nastavku.

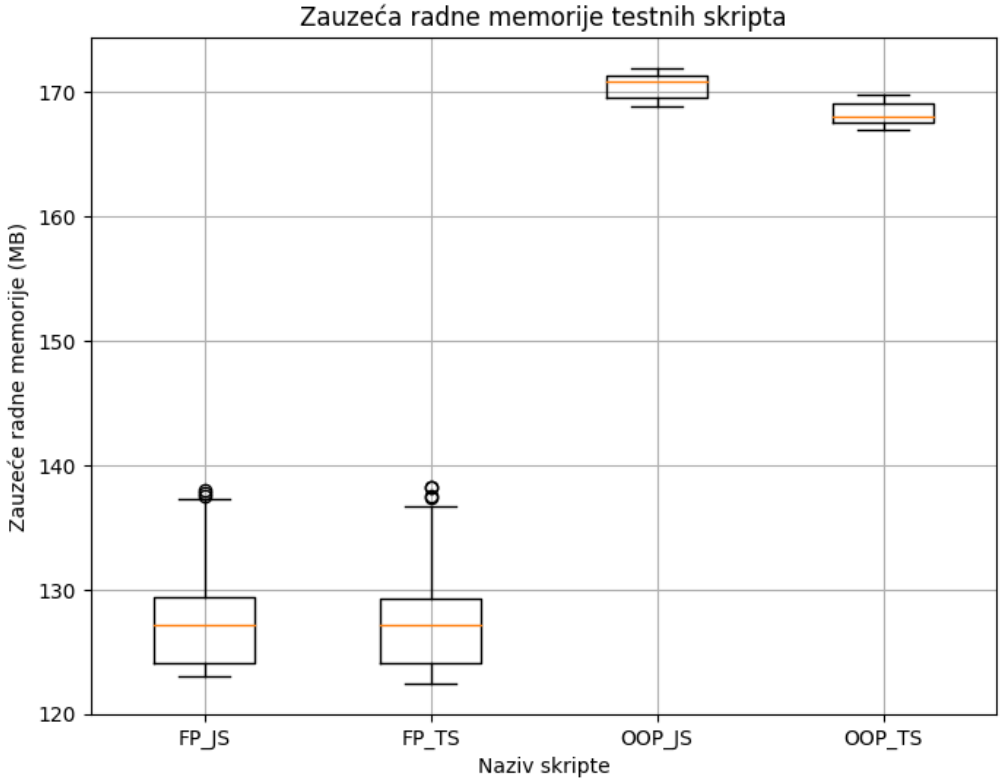
0 - FP_JS; 1 - FP_TS; 2 - OOP_JS; 3 - OOP_TS;

	0	1	2	3
0	1.000000	0.639529	**0.001000**	**0.001000**
1		1.000000	**0.001000**	**0.001000**
2			1.000000	**0.001387**
3				1.000000

Iz priloženog možemo zaključiti kako, uz razinu pouzdanosti od 95%, postoji statistički značajna razlika između grupa 0 i 2, 0 i 3, 1 i 2, 1 i 3 te 2 i 3. Odnosno, statistički značajna razlika postoji između FP-a i OOP-a, neovisno o programskom jeziku (JS ili TS). Dodatno se pokazalo kako zaista ne postoji razlika između JS i TS varijante funkcijske implementacije algoritma. Međutim, ovim je testom otkriveno kako ipak postoji statistički značajna razlika u vremenu izvođenja između objektno-orijentiranih varijanti implementacije algoritama u JS i TS programskim jezicima.

6.1.2 Prostorna složenost

Na dijagramu pravokutnika (slika 7) možemo vidjeti inicijalne rezultate mjerenja vremenske složenosti za četiri testirane skripte - implementacije algoritma. Na osi apscisa redom su pobrojan nazivi četvero testiranih skripata, dok je na osi ordinata označeno vrijeme izvođenja pojedine skripte u milisekundama.



Slika 6: Dijagram pravokutnika za prostornu složenost četiri slučaja implementacije algoritma obrade meteoroloških podataka

Možemo uočiti kako se kod funkcijske implementacije u oba slučaja interkvartilni raspon proteže od otprilike 125MB do 129MB. Razina donjeg brka u oba slučaja nalazi se na otprilike 124MB, dok se gornji brk kod funkcijske implementacije u JS-u proteže do otprilike

138MB, a u TS varijanti do otprilike 137MB. Medijan se u oba slučaja nalazi na razini od 127MB. U oba slučaja imamo asimetričnu distribuciju podataka, točnije lijevu iskrivljenost distribucije podataka.

Kod objektno-orijentirane implementacije ne dolazi do pojave stršila. Međutim, u funkcijskoj implementaciji asimetričnost distribucije podataka znatno je naglašenija te shodno tome dolazi do pojave stršila - zaključujemo da je zauzeće memorije u nekoliko situacija bilo veće od *očekivanih* rezultata.

Kod objektno-orijentirane implementacije u JS-u, medijan se nalazi na otprilike 172MB. Gornji brk nalazi se na 173MB, dok se donji nalazi na otprilike 168MB. U TS varijanti objektno-orijentirane implementacije, medijan je nešto manji - približno 167MB. Gornji se brk nalazi na otprilike 170MB, a donji na otprilike 166MB.

Iako možemo naslutiti da podaci, u svim varijantama implementacije algoritma, nisu normalno distribuirani (nepravilan oblik pravokutnika, medijana i postojanje stršila), potrebno je napraviti formalni test normalnosti skupa uzorka kako bismo to zaista i statistički potvrdili. Jedan od takvih testova je Shapiro-Wilk test. U nastavku su navedene njegove hipoteze, zajedno s rezultatima provedbe istog koristeći *stats.py* skriptu.

H_0: Ne postoji značajno odstupanje od normalne distribucije.

H_A: Postoji značajno odstupanje od normalne distribucije.

Skripta	Shapiro-Wilk p vrijednost	Normalna distribucija
FP_JS	3.9425382403202036e-10	Ne
FP_TS	1.4641381440583245e-09	Ne
OOP_JS	8.38607593323104e-06	Ne
OOP_TS	2.446230973873753e-05	Ne

Ovime smo potvrdili da su sve p vrijednosti manje od 0.05, odnosno da ni jedan skup uzoraka nije normalno distribuiran - odbacujemo nul-hipoteze. Shodno tome, za statističku analizu usporedbe rezultata, umjesto ANOVA-e s ponovljenim mjerenjem, koristit ćemo njezinu neparametrijsku alternativu - Friedmanova ANOVA test. Sve su pretpostavke Friedmanove ANOVA-e zadovoljene pa se može preći u postupak provedbe. Kao hipoteze uzet ćemo:

H_0: Ne postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije računala bilo kojeg para testnih skripata.

H_A: Postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije računala barem jedne testne skripte.

Nakon provedene Friedmanove ANOVA-e, dobivamo da je p vrijednost jednaka $2.9921722528899117 \times 10^{-57}$. Uz razinu pouzdanosti od 95% možemo odbaciti nultu hipotezu te donijeti zaključak da postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije računala barem jedne testne skripte.

Kako bismo otkrili između kojih testnih skripata postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije, poslužit ćemo se Nemenyi post hoc testom. Rezultati usporedbe testnih skripata temeljem vremena izvođenja dani su u nastavku.

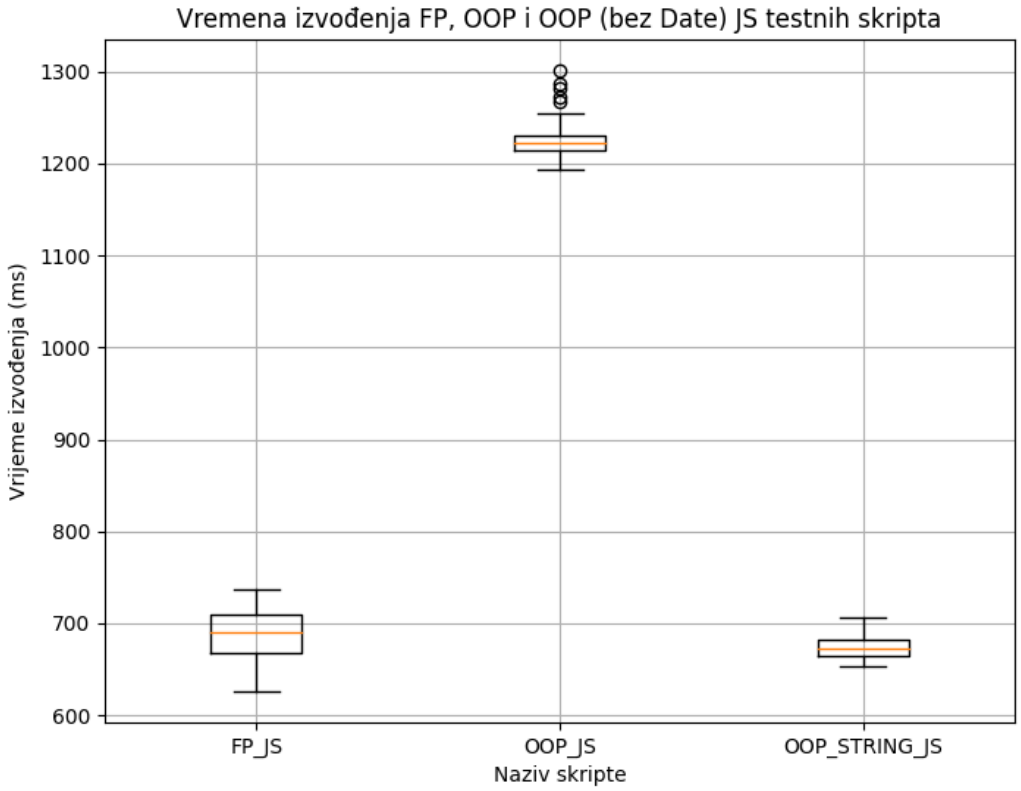
```
0 - FP_JS; 1 - FP_TS; 2 - OOP_JS; 3 - OOP_TS;
        0       1       2       3
0   1.000   0.900   0.001   0.001
1           1.000   0.001   0.001
2                   1.000   0.001
3                           1.000
```

Iz priloženog možemo zaključiti kako, uz razinu pouzdanosti od 95%, postoji statistički značajna razlika između grupa 0 i 2, 0 i 3, 1 i 2, 1 i 3 te 2 i 3. Odnosno, statistički značajna razlika postoji između FP-a i OOP-a, neovisno o programskom jeziku (JS ili TS). Dodatno se pokazalo kako zaista ne postoji razlika između JS i TS varijante funkcijske implementacije algoritma. Međutim, ovim je testom otkriveno kako ipak postoji statistički značajna razlika u zauzeću radne memorije računala između objektno-orijentiranih varijanti implementacije algoritama u JS i TS programskim jezicima.

6.2 Vremenska optimizacija

Zbog uočenih značajnih statističkih razlika u vremenima izvođenja funkcijske naspram objektno-orijentirane inačice JS skripte, odlučili smo detaljnije proučiti kod te saznati potencijalne probleme zbog kojih bi se to dešavalo. Ukoliko poboljšamo objektno-orijentiranu JS, isto bi se trebalo moći primijeniti i na TS pa će shodno tome, ovdje biti fokus samo na JS varijanti.

Nakon detaljnije analize i testiranja različitih izmjena, uočili smo konstrukciju i kreiranje *Date* objekata kao potencijalni vremenski *problem* pri izvršavanju objektno-orijentirane varijante. Naime, u našem specifičnom kontekstu obrade meteoroloških podataka, kreiranje *Date* objekata za apstrahiranje strukturnog tipa datuma i nije toliko nužno. Datum se koristi isključivo kao ključ u JSON objektu ili *Map* strukturi te ne zahtijeva neke posebne obrade. Shodno tome, odlučili smo izbaciti pretvorbu string datuma u *Date* objeke te tako dobili sljedeće rezultate (skripta pod nazivom *OOP_STRING_JS.js*):



Slika 7: Dijagram pravokutnika za vremensku složenost između FP_JS, OOP_JS i OOP_STRING_JS testnih skripata

Već se prema dijagramu pravokutnika na slici 5 može vidjeti iznimno značajno poboljšanje OOP_STRING_JS skripte, gdje se koristi string kao struktura za datum, naspram OOP_JS skripte gdje se datum u stringu tipu pretvara u *Date* objekte. Možemo uočiti kako se OOP_STING_JS varijanta spustila do razine brzine FP_JS varijante.

Interkvartilni raspon novotestirane OOP_STRING_JS skripte proteže se od otprilike 670ms do 690ms. Razina donjeg brka nalazi se na otprilike 660ms, dok se gornji brk proteže do otprilike 705ms. Medijan se nalazi gotovo u sredini interkvartilnog raspona, te iznosi približno 680ms.

Unatoč naizgled normalno distribuiranim podacima, p vrijednost u Shapiro-Wilk testu za OOP_STRING_JS iznosi 0.00325, a što uz razinu pouzdanosti od 95% znači da uzorci mjerenja brzine izvođenja skripte nisu normalno distribuirani. Zbog toga se opet prelazi u postupak Friedmanove ANOVA-e, gdje se kao hipoteze uzimaju:

H_0: Ne postoji statistički značajna razlika u vremenima izvođenja bilo kojeg para testnih skripata.

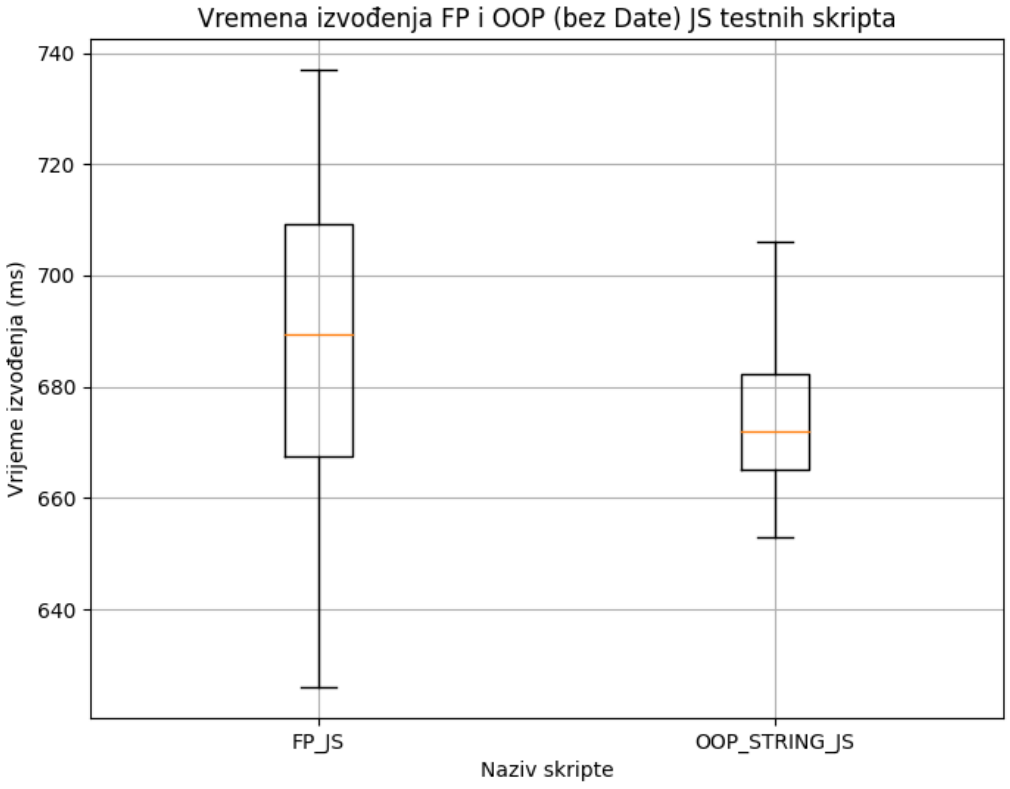
H_A: Postoji statistički značajna razlika u vremenima izvođenja barem jedne testne skripte.

Nakon provedene Friedmanove ANOVA-e, dobivamo da je p vrijednost jednaka $4.6702060544967294 \times 10^{-55}$. Uz razinu pouzdanosti od 95% možemo odbaciti nultu hipotezu te donijeti zaključak da postoji statistički značajna razlika u vremenu izvođenja barem jedne testne skripte.

Kako bismo otkrili između kojih testnih skripata postoji statistički značajna razlika u vremenu izvođenja, poslužit ćemo se Nemenyi post hoc testom. Rezultati usporedbe testnih skripata temeljem vremena izvođenja dani su u nastavku.

```
0 - FP_JS; 1 - OOP_JS; 2 - OOP_STRING_JS;
          0         1          2
0  1.000000  0.001   0.019738
1            1.000   0.001000
2                    1.000000
```

Iz priloženog možemo zaključiti kako, uz razinu pouzdanosti od 95%, postoji statistički značajna razlika između svih grupa. Možemo također primjetiti kako je razlika između grupe 0 i 2 manja nego razlika između grupa 1 i 2, a što ukazuje na to da je OOP_STRING_JS implementacija bliža vremenskoj kompleksnosti FP_JS implementacije, nego li OOP_JS implementaciji algoritma obrade podataka. Kako bismo bolje vidjeli tu razliku, u nastavku je prikazan isti dijagram pravokutnika, ali bez OOP_JS varijante.

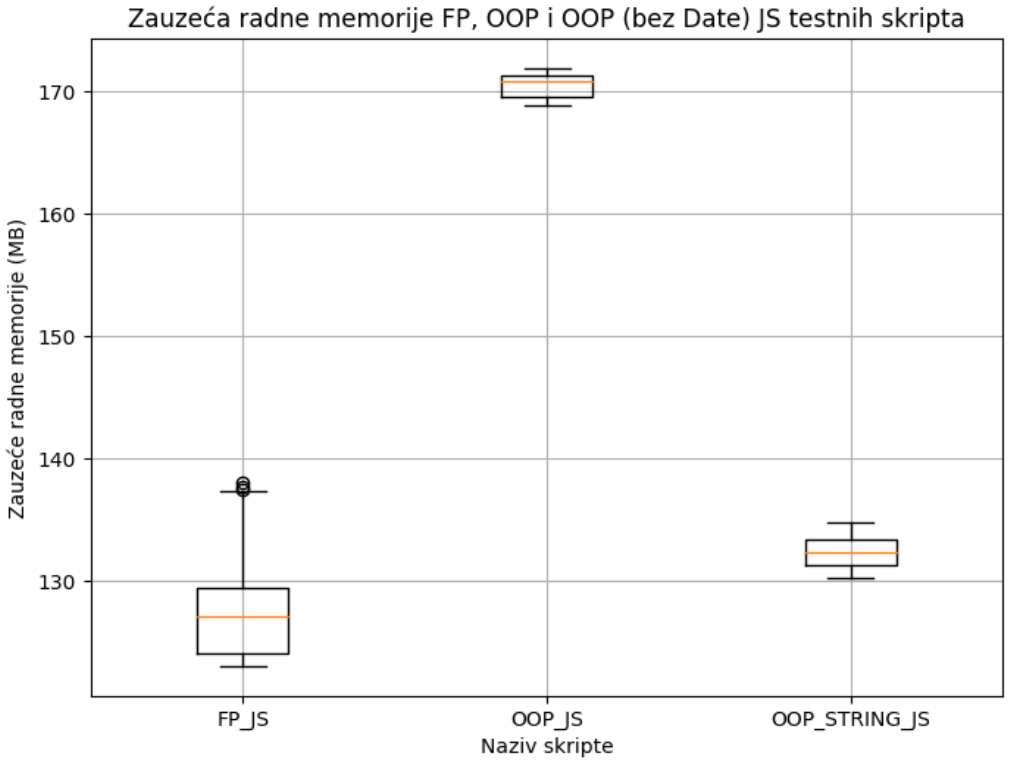


Slika 8: Dijagram pravokutnika za vremensku složenost FP_JS i OOP_STRING_JS testnih skripata

6.2.1 Utjecaj na prostornu složenost

Prethodno smo statistički potvrdili da se uklanjanjem pretvorbe string datuma u *Date* objekte, vrijeme izvođenje obrade meteoroloških podataka zaista poboljšalo. Međutim, korisno je napraviti i kratki osvrt na to kako je ta promjena utjecala na prostornu složenost.

U nastavku možemo vidjeti dijagram pravokutnika gdje su vizualizirani uzorci mjerenja zauzeća radne memorije računala kod FP_JS, OOP_JS i OOP_STRING_JS testnih skripata.



Slika 9: Dijagram pravokutnika za prostornu složenost između FP_JS, OOP_JS i OOP_STRING_JS testnih skripata

Na slici 7, jasno je vidljivo kako uklanjanje *Date* objekata iz OOP_JS varijante implementacije algoritma pozitivno djeluje na zauzeće radne memorije računala. Zauzeće RAM-a u OOP_STRING_JS implementaciji algoritma palo je za gotovo 40MB te je sada smješteno u višem rangu funkcijske varijante algoritma.

Interkvartilni raspon OOP_STRING_JS skripte proteže se od otprilike 131MB do 133MB. Razina donjeg brka nalazi se na otprilike 130MB, dok se gornji brk proteže do otprilike 135MB. Medijan se nalazi gotovo u sredini interkvartilnog raspona, te iznosi približno 132MB.

Unatoč naizgled normalno distribuiranim podacima, p vrijednost u Shapiro-Wilk testu za OOP_STRING_JS iznosi 0.00196, a što uz razinu pouzdanosti od 95% znači da uzorci

mjerenja zauzeća radne memorije računala, tijekom izvođenja skripte, nisu normalno distribuirani. Zbog toga se opet prelazi u postupak Friedmanove ANOVA-e, gdje se kao hipoteze uzimaju:

H_0: Ne postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije računala bilo kojeg para testnih skripata.

H_A: Postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije računala barem jedne testne skripte.

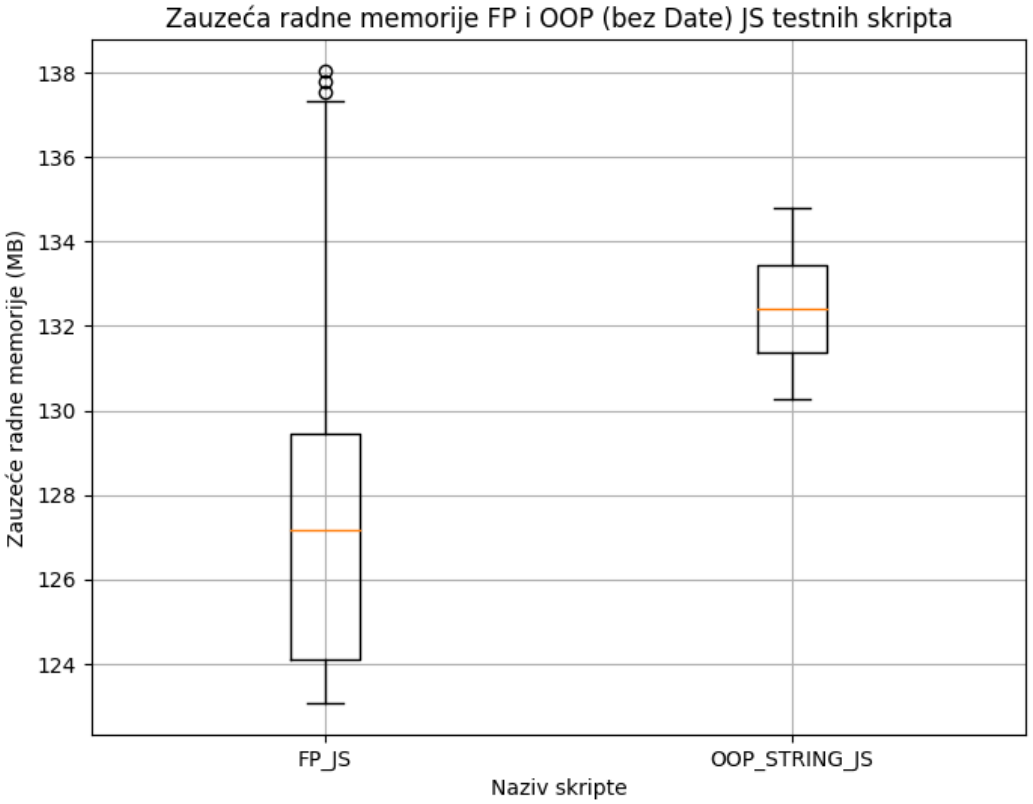
Nakon provedene Friedmanove ANOVA-e, dobivamo da je p vrijednost jednaka $1.0544964430181827 \times 10^{-36}$. Uz razinu pouzdanosti od 95% možemo odbaciti nultu hipotezu te donijeti zaključak da postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije računala barem jedne testne skripte.

Kako bismo otkrili između kojih testnih skripata postoji statistički značajna razlika u rezultatima mjerenja zauzeća radne memorije, poslužit ćemo se Nemenyi post hoc testom. Rezultati usporedbe testnih skripata temeljem vremena izvođenja dani su u nastavku.

```
0 - FP_JS; 1 - OOP_JS; 2 - OOP_STRING_JS;
       0      1      2
0  1.000  0.001  0.001
1         1.000  0.001
2                1.000
```

Iz priloženog možemo zaključiti kako, uz razinu pouzdanosti od 95%, postoji statistički značajna razlika između svih grupa.

S obzirom da se *unaprijeđena* varijanta objektno-orijentiranog algoritma smjestila u višem rangu funkcijske varijante, prikladno je detaljnije vidjeti kak taj odnos zaista izgleda. Prema, tome, u nastavku je prikazan isti dijagram pravokutnika s izuzećem OOP_JS varijante implementacije algoritma obrade meteoroloških podataka.



Slika 10: Dijagram pravokutnika za prostornu složenost FP_JS i OOP_STRING_JS testnih skripata

7. Diskusija

Prema petom poglavlju, navodimo kvalitativnu usporedbu funkcijskog i objektno-orijentiranog programiranja u kontekstu obrade velike količine podataka.

Funkcijska paradigma fokusira se na definiranje funkcija koje transformiraju ulazne podatke u izlazne, bez stanja i nuspojava. To znači da funkcija ovisi samo o svojim ulazima i ne utječe na druge dijelove programa ili vanjsko okruženje. Ova paradigma korisna je u kontekstu obrade podataka jer je često potrebno provesti niz transformacija nad velikim količinama podataka. Funkcijska paradigma omogućuje definiranje funkcija za različite transformacije podataka koje se mogu primijeniti na cijele setove podataka, jednostavno pozivajući funkciju s odgovarajućim argumentima.

S druge strane, objektno-orijentirana paradigma programiranja temelji se na konceptu objekata koji sadrže podatke i metode koje rade s tim podacima. Objekti se mogu naslijeđivati i proširivati, a objektno orijentirana paradigma omogućuje stvaranje složenijih struktura podataka i modeliranje realnih, praktičnih problema. Ova paradigma korisna je u kontekstu obrade podataka jer omogućuje stvaranje kompleksnih modela podataka koji se mogu lako manipulirati i analizirati.

Ukratko, funkcijska paradigma programiranja korisna je za obradu velikih količina podataka, dok je objektno-orijentirana paradigma programiranja korisna za modeliranje složenijih struktura podataka i rješavanje stvarnih problema. U praksi, obično se koriste obje paradigme zajedno, kako bi se postigla optimalna učinkovitost i skalabilnost. Time je dan odgovor na **IP1**.

Nakon provedenog mjerenja vremenske složenosti JS i TS implementacija algoritama, u funkcijskoj i objektno-orijentiranoj varijanti, na razini pouzdanosti od 95% možemo tvrditi kako postoji statistički značajna razlika između OOP-a i FP-a, neovisno o programskom jeziku. Nadalje, utvrđeno je da ne postoji statistički značajna razlika između funkcijskih implementacija algoritama, što je bilo i prema očekivanom jer TS zapravo preuzima sve mogućnosti FP-a iz JS-a. Međutim, ono što nije bilo za očekivat jest da će postojati statistički značajna razlika između objektno-orijentiranih varijanti implementacije algoritama obrade meteoroloških podataka. Pokazalo se da objektno-orijentirani TS ima vremenski lošije performanse naspram objektno-orijentiranog JS-a, a što je začuđujuće s obzirom na to da nam je TS *nametnuo* korištenje vremenski efikasnijih struktura podataka kao što su *Map* i *Set*. S obzirom na to da objektno-orijentirani pristup nalaže rad s tipovima podataka, ključna riječ *any* u programskom jeziku TS ne uklapa se u taj koncept. Shodno tome, određene rezultate obrade podataka u objektno-orijentiranoj TS implementaciji bili smo *prisiljeni* preslikati u određene tipove podataka, što dodatno usporava algoritam obrade.

U optimiziranoj varijanti objektno-orijentirane JS implementacije algoritma, uočilo se kako konstrukcija *Date* objekata značajno usporava vremensko izvođenje skripte. Nakon zadržavanja string reprezentacije datuma, postiglo se toliko vremensko poboljšanje da je objektno-orijentirana, bez *Date* objekata, implementacija putem JS-a postala bliža vremenskoj kompleksnosti funkcijske implementacije algoritma obrade meteoroloških podataka.

U optimiziranoj varijanti objektno-orijentirane TS implementacije algoritma (varijanta OOP_OPT_TS, prilog 3) možemo uočiti, da ukoliko uklonimo ograničenje korištenje tipova podataka, u situacijama gdje bi elegantnije rješenje bilo ostvareno korištenjem ključne riječi *any*, vremensko izvođenje se poboljšalo.

Prema navedenim rezultatima, možemo dati odgovor na **IP2**. U našem se kontekstu obrade meteoroloških podataka pokazalo da funkcijski pristup programiranju ima efikasniju vremensku složenost; vremenski efikasnije izvođenje skripte obrade podataka. Međutim, objektno-orijentirana JS varijanta može se poboljšati na način da se uklone kontekstualno-nepotrebne apstrakcije podataka (prikaz datuma pomoću *Date* objekta, umjesto string tipa), te tako postigne brže izvođenje, a koje je u rangu funkcijskih implementacija algoritama obrade podataka.

Nakon provedenog mjerenja prostorne složenosti JS i TS implementacija algoritama, u funkcijskoj i objektno-orijentiranoj varijanti, na razini pouzdanosti od 95% možemo tvrditi kako postoji statistički značajna razlika između između FP-a i OOP-a, neovisno o programskom jeziku (JS ili TS). Rezultati pokazuju da objektno-orijentirane varijante implementacije algoritama imaju konstantno lošije rezultate - zauzeće memorije je čak za 40MB više, naspram funkcijskih varijanti implementacija algoritama.

Dodatno se pokazalo kako zaista ne postoji razlika između JS i TS varijante funkcijske implementacije algoritma. Međutim, post hoc testom otkriveno je kako ipak postoji statistički značajna razlika u zauzeću radne memorije računala između objektno-orijentiranih varijanti implementacije algoritama u JS i TS programskim jezicima. Rezultati pokazuju da JS varijanta, u prosjeku, ima veće zauzeće memorije. Nakon detaljnije analize, točan razlog nije poznat. Naslućuje se da su u pitanju promjene vezane uz strukturne tipove podataka koje nam TS nameće - *Map* i *Set*. Odnosno, čini se da navedene strukture imaju bolju prostornu složenost od klasičnih nizova. Nakon detaljnije analize, točan razlog nije poznat. Što se tiče optimizacije objektno-orijentirane varijante JS-a (bez *Date* objekata), osim bolje vremenske efikasnosti, uočeno je i prostorno poboljšanje - zauzeće memorije smanjeno je za gotovo 40MB, te se nalazi u rangu funkcijskih implementacija algoritama. Time zaključujemo kako *Date* objekti stvaraju veći pritisak na memoriju, za razliku od *primitivnog*

tip string, koji se doima kao bolji izbor u našem kontekstu zauzeća memorije kod obrade podataka.

Prema navedenim rezultatima, možemo dati odgovor na **IP3**. U našem se kontekstu obrade meteoroloških podataka pokazalo da funkcijske implementacije imaju bolju prostornu složenost u odnosu na njihove objektno-orijentirane varijante. Ipak, potrebno je naglasiti da se objektno-orijentirana varijanta može poboljšati na način da se uklone određeni objektno-orijentirani koncepti, poput apstrahiranja datuma u *Date* objekt, te tako postigne prostorna složenost bliža funkcijskim implementacijama algoritama.

Kao potencijalni problem u metodici rada može se dovesti u pitanje broj ponovljenih rezultata mjerenja. Hoće li veći broj ponavljanja rezultata mjerenja obrade podataka skripata potencijalno rezultirati drugačijom interpretacijom rezultata. Shodno tome, provedeno je 500 cikličkih mjerenja sljedećih testnih skripata: FP_JS, FP_TS, OOP_JS, OOP_STRING_JS i OOP_TS (Prilog 1). Također, kroz više dana, testirano je po 100, 200 i 300 cikličkih izvođenja istih skripata. No, ono što možemo zaključiti je kako interpretacija samih rezultata ostaje sačuvana te ne dolazi do statistički značajne promjene ukoliko se broj rezultata mjerenja poveća. Međutim, povećanjem broja provedenih testova, preko granice od 100 mjerenja u našem slučaju, mogu se znatno opteretiti procesorski resursi. Shodno tome, dolazi do značajnog prigušivanja (engl. *throttle*) rada procesora, a što na kraju rezultira smanjenjem performansi procesora. Samim time možemo uočiti i značajan porast stršila na dijagramu pravokutnika.

Nadalje, u knjizi [29] autor bazira jedno poglavlje na samom mehanizmu automatskog sakupljanja smeća (engl. *automatic garbage collection, AGC*) u programskom jeziku JS. Autor navodi kako mehanizam automatskog sakupljanja smeća može znatno utjecati na performanse programa te kako bi se trebao uzeti u obzir prilikom razmatranja performansi programa. Shodno toj informaciji, postavlja se pitanje, hoće li doći do promjeni performansi pojedinih implementacija ukoliko se mehanizam automatskog sakupljanja smeća deaktivira. Pretpostavka bi bila da se nakon deaktiviranja mehanizma sakupljanja smeća, vrijeme izvođenja skripata smanji zbog toga što, u tom slučaju, mehanizam sakupljanja smeća ne mora konstantno u pozadini čistiti nekorištene objekte. Što se tiče prostorne složenosti, očekuje se da će zauzeće memorije porasti zbog toga što se prethodno alocirani memorijski prostor više ne koristi, ali ne biva očišćen iz memorije procesa.

Nakon kratkog neformalnog testiranja, pokazalo se da je vremenska složenost zapravo postala veća (Prilog 2). Shodno tome, možemo zaključiti kako se naša pretpostavka

ispostavila netočnom. Naime, nakon što smo *isključili*[8] mehanizam automatskog sakupljanja smeća, u kontekstu obrade velike količine podataka, nakon promatranja ponašanja radne memorije prilikom izvođenja programa, možemo dati zaključak, kako prilikom deaktivacije mehanizma, radna memorija je postala znatno opeterećenija što je rezultiralo lošijim performansama obe metrike promatranja.

Kako smo vidjeli u pregledu literature, različiti autori predlažu jednu određenu programsku paradigmu (OOP ili FP) kao bolje rješenje [3] [4] [11], dok s druge strane, veći broj autora navodi kako ne postoji uvijek ispravan odabir određene programske paradigme. Odnosno, želi se naglasi da svaki pristup ima svoje prednosti i nedostatke u različitim područjima rada. Navode kako je potrebno uzeti u obzir različite programske paradigme i njihove specifičnost te ukoliko je potrebno, pokušati napraviti određenu integraciju prednosti obje programske paradigme [5] [6] [7] [8] [11]. U ovom se istraživanju pokazalo da funkcijsko programiranje, prema promatranim metrikama, daje bolje rezultate u kontekstu obrade velike količine (gotovo pola milijuna) podataka, te kako je sama primjena čistog objektno-orijentiranog pristupa rezultirala lošijim performansama. Nakon optimizacije objektno-orijentiranih pristupa, primjenom osnovnih koncepata FP-a u kontekstu obrade podataka, utvrdilo se znatno poboljšanje promatranih performansi, što dodatno potvrđuje naš zaključak.

Na temelju provedenih 100 mjerenja, možemo uočiti kako se vrijeme izvođenja algoritma implementiranog objektno-orijentiranim pristupom razlikuje za otprilike 500ms, odnosno za pola sekunde. Ukoliko sagledamo širu sliku i stvarni slučaj u kojem će se navedene implementacije koristiti, postavlja se pitanje, predstavlja li funkcijski pristup toliko bolje rješenje. Isto tako moramo uzeti u obzir ono što gubimo primjenom čistog FP-a, a to je nedostatak intuitivnosti, jednostavnosti strukture i arhitekture programskog koda i slično.

Kao nastavak ovog istraživanja, otvoreno je nekoliko novih tema. Jedna moguća modifikacija uključuje promjenu ES specifikacije. Kada smo smanjili ES specifikaciju ispod ES2015, zauzeće memorije se povećalo. Prema tome, ostale daljne mogućnosti istraživanja na ovu temu uključuju testiranje većih broja ulaznih podataka, mijenjanje postavki TypeScript kompajlatora, izvođenje skripata putem drugog JS okruženja (npr. Dino) i slično.

[8] NodeJS okruženje nema potporu za potpuno isključivanje automatskog sakupljanja smeća, čak ni kada postavimo ručno upravljanje memorijom [32]. Taj se problem zaobišao tako da se maksimalno zauzeće procesa (NodeJS aplikacije) postavilo na 1000MB (zastavica `--max-old-space-size=1000`), a što u svakom slučaju mora biti dovoljno za učitavanje i obradu testirane količine podataka.

8. Zaključak

Ovaj je istraživački rad uspješno odgovorio na glavno istraživačko pitanje temeljeno na učinkovitosti funkcijskog programiranja (FP) i objektno-orijentiranog programiranja (OOP) u kontekstu obrade stvarnih prikupljenih meteoroloških podataka. Kao jezik implementacije odabrani su skriptni programski jezici JavaScript (JS) i njegov tipizirani pandan TypeScript (TS). Kako bi se dobio odgovor na tri postavljena istraživačka pitanja (IP), rad je obuhvatio sustavni pregled koncepata FP-a i OOP-a u JS i TS skriptnim jezicima te napravio praktičan eksperiment ponovljenog mjerenja sa svrhom analize, odnosno potvrde statističke značajnosti njihovih efikasnosti u kontekstu vremenske i prostorne složenosti.

Obje promatrane paradigme programiranja imaju svoje prednosti i nedostatke u kontekstu obrade podataka. Funkcijska paradigma programiranja korisna je za obradu velikih količina podataka, s obzirom na njezinu efikasniju vremensku i prostornu složenost, dok je objektno-orijentirana paradigma programiranja korisna za modeliranje složenijih struktura podataka. U ovom se istraživanju funkcijska implementacija algoritma obrade meteoroloških podataka pokazala vremenski i prostorno efikasnijom naspram njezine objektno-orijentirane varijante. Međutim, detaljnijom analizom, objektno-orijentirana se varijanta uspjela, uz nekoliko odstupanja koncepata OOP-a, optimizirati te tako približiti kompleksnosti funkcijske implementacije. U praksi se najčešće kombiniraju obje paradigme, kako bi se postigla optimalna učinkovitost i skalabilnost. U svakom slučaju, implementaciju određene paradigme treba prilagoditi specifičnim potrebama i zahtjevima konteksta obrade podataka kako bi se postigao optimalan rezultat.

Ovo istraživanje ima i određena ograničenja. Između ostalog, same implementacije algoritama obrade meotoroloških podataka mogle bi se uvijek dodatno optimizirati i bolje prilagoditi kontekstu korištenja. Također, kod kompajliranja TS-a, moguća je visoka razina konfiguriranja, a koja nije bila detaljno izučavana u ovom radu zbog nedostatka vremena te samog zadanog opsega istraživanja.

Tema istraživanja nije nova, ali je već dugo godina veoma aktualna. Ono što prema recentnim trendovima možemo vidjeti jest da funkcijsko programiranje definitivno raste na popularnosti, uglavnom zahvaljujući ogromnom napretku u znanosti o podacima i strojnom učenju.

Zahvale

Hvala **doc. dr. sc. Matiji Novak** na savjetima za provođenje eksperimenta te vremenu utrošenom za mentoriranje ovog istraživačkog rada. Također hvala **WATSS laboratoriju** na dozvoli za korištenje stvarnih meteoroloških podataka, a bez kojih ovo istraživanje ne bi bilo moguće.

Popis literature

[1] Tim Berners-Lee, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. Harper Business, 2000. [Online]. Available: https://docdrop.org/download_annotation_doc/Tim-Berners-Lee---Weaving-the-Web_-The-Original-Design-and-U-88myd.pdf

[2] P. Van Roy, *Programming Paradigms for Dummies: What Every Programmer Should Know.* in New Computational Paradigms for Music, G. Assayag and A. Gerzso, Eds. Paris: Delatour France, IRCAM, 2009, pp. 9–49.

[3] E. Allen, "Comparison of Object-oriented and Functional Programming for Code Generation".

[4] S. Konings and M. Thesis, "Source Code Metrics for Combined Functional and Object-Oriented Programming in Scala," Studeni 2020, [Online]. Available: http://essay.utwente.nl/85223/1/Konings_MA_EEMCS.pdf

[5] Ph. Narbel, "Functional Programming at Work in Object- Oriented Programming.," *J. Object Technol.*, vol. 8, no. 6, p. 181, 2009, doi: 10.5381/jot.2009.8.6.a5.

[6] D. Alic, S. Omanovic, and V. Giedrimas, "Comparative analysis of functional and object-oriented programming," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2016, pp. 667–672. doi: 10.1109/MIPRO.2016.7522224.

[7] K. Svensson Sand and T. Eliasson, "A comparison of functional and object-orientedprogramming paradigms in JavaScript," Jun. 2017. Accessed: Apr. 21, 2023. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-14717

[8] W. Zhang, C. David, and M. Wang, "Decomposition Without Regret." arXiv, Apr. 21, 2022. Accessed: Apr. 22, 2023. [Online]. Available: http://arxiv.org/abs/2204.10411

[9] J. M. Fernandes and J. Lilius, "Functional and object-oriented views in embedded software modeling," in *Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2004.*, Brno, Czech Republic: IEEE, 2004, pp. 378–387. doi: 10.1109/ECBS.2004.1316722.

[10] R. Panigrahi, S. Baboo, and N. Padhy, "The Statistical Measurement of an Object-Oriented Programme Using an Object Oriented Metrics," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (ficta) 2014, Vol 2*, S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. K. Mandal, Eds., Cham: Springer Int Publishing Ag, 2015, pp. 605–618. doi: 10.1007/978-3-319-12012-6_67.

[11] A. Koval, O. Yashyna, G. Radelchuk, and Y. Forkun, "Comparison of object-oriented and functional programming paradigms in software design," *Her. KHMELNYTSKYI Natl. Univ.*, vol. 297, no. 3, pp. 34–38, Jul. 2021, doi: 10.31891/2307-5732-2021-297-3-34-38.

[12] H. Abelson, G. J. Sussman, J. Sussman, and A. J. Perlis, *Structure and interpretation of computer programs*, 2. ed., 7. [pr.]. in Electrical engineering and computer science series. Cambridge, Mass.: MIT Press [u.a.], 2002.

[13] P. Van-Roy, *Concepts, techniques, and models of computer programming*. Cambridge, Mass: MIT Press, 2004.

[14] L. Sterling and E. Y. Shapiro, *The art of Prolog: advanced programming techniques*. in MIT Press series in logic programming. Cambridge, Mass: MIT Press, 1986.

[15] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, Sep. 1989, doi: 10.1145/72551.72554.

[16] F. Kereki, *Mastering JavaScript Functional Programming - Second Edition*, 2nd edition. Packt Publishing, 2020.

[17] "V8 JavaScript engine." https://v8.dev/ (accessed Apr. 18, 2023).

[18] J. Zhu, "A Scalability-oriented Benchmark Suite for Node. js in the Cloud," University of New Brunswick, 2018. Accessed: Apr. 18, 2023. [Online]. Available: https://unbscholar.lib.unb.ca/islandora/object/unbscholar:9524/datastream/PDF/view

[19] A. P. Field, J. Miles, and Z. Field, *Discovering statistics using R*. London ; Thousand Oaks, Calif: Sage, 2012.
[20] "SciPy documentation." https://scipy.org/ (accessed Apr. 18, 2023).
[21] "scikit-posthocs documentation." https://scikit-posthocs.readthedocs.io/en/latest/ (accessed Apr. 18, 2023).
[22] "Matplotlib documentation." https://matplotlib.org/stable/index.html (accessed Apr. 18, 2023).
[23] "Process," *Node.js v20.0.0 Documentation*. https://nodejs.org/api/process.html#processmemoryusagerss (accessed Apr. 18, 2023).
[24] M. L. Scott, *Programming language pragmatics*, Fourth edition. Waltham, MA: Morgan Kaufmann, an imprint of Elsevier, 2016.
[25] M. Ohuru, "JavaScript How we got here. An indepth history.," Sep. 2020.
[26] E. International, "ECMA-262, 13th edition, June 2022 ECMAScript® 2022 Language Specification." https://262.ecma-international.org/
[27] A. Chiarelli, *Mastering JavaScript object-oriented programming: unleash the true power of JavaScript by mastering object-oriented programming principles and patterns*. Birmingham: Packt Publishing, 2016.
[28] B. Cherny, *Programming TypeScript: making your JavaScript applications scale*, First edition. Sebastopol, CA: O'Reilly Media, Inc, 2019.
[29] S. Fenton, *Pro TypeScript: application-scale JavaScript development*. in The expert's voice in TypeScript. New York, N.Y: Apress, 2014.
[30] "TSConfig Reference - Docs on every TSConfig option." https://www.typescriptlang.org/tsconfig (accessed Apr. 24, 2023).
[31] "File system," *Node.js v20.0.0 Documentation*. https://nodejs.org/api/fs.html (accessed Apr. 18, 2023).
[32] "Node.js Under the Hood #9: Collecting the Garbage," *DEV Community*, Mar. 17, 2020. https://dev.to/_staticvoid/node-js-under-the-hood-9-collecting-the-garbage-772 (accessed Apr. 24, 2023).

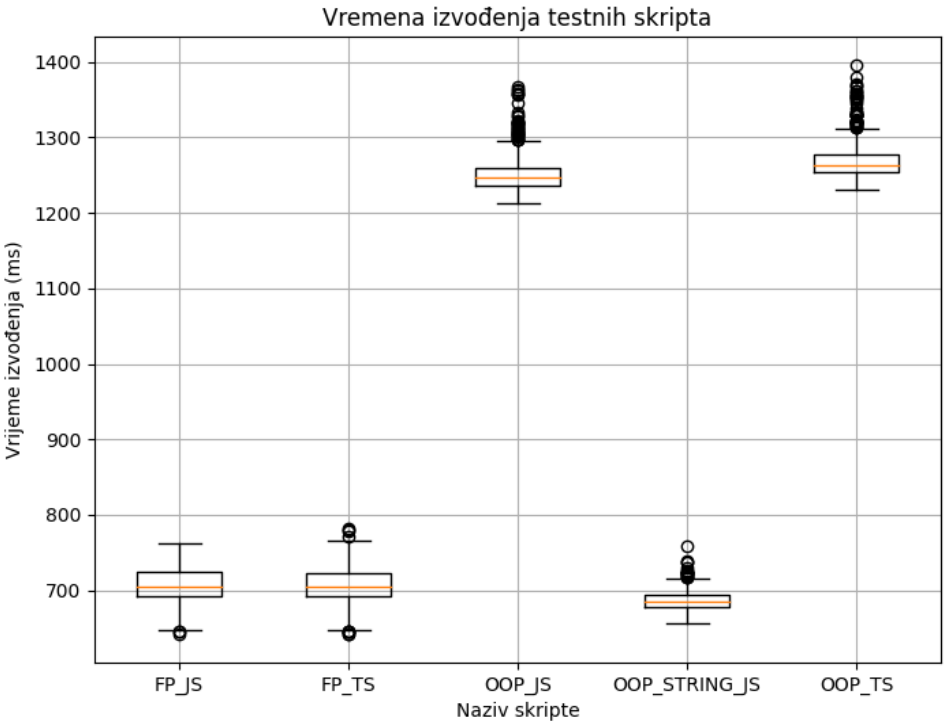
Popis slika

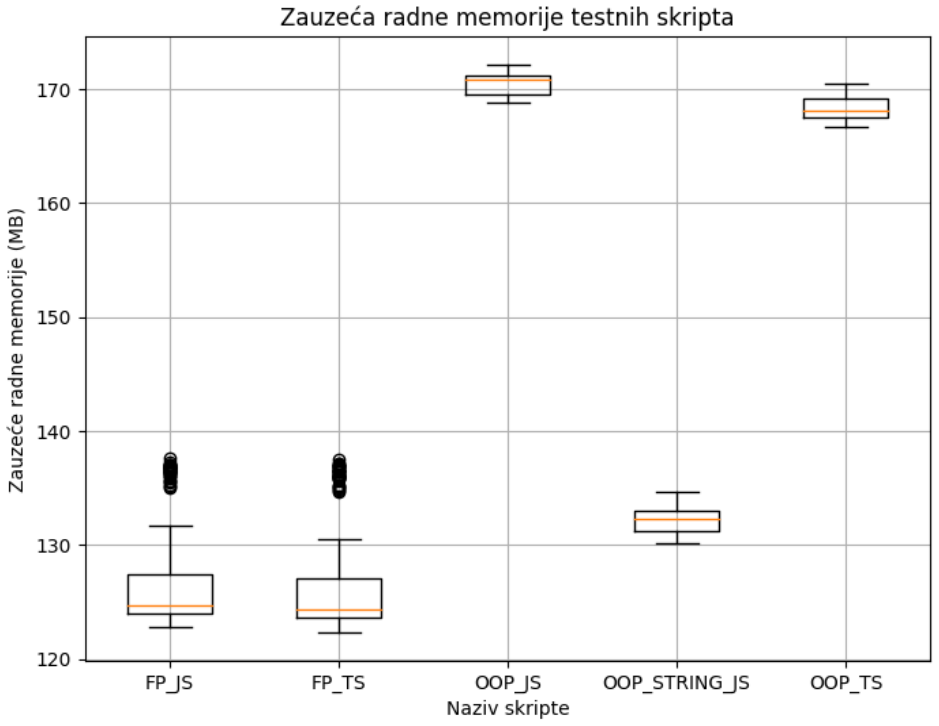
Popis tablica

Prilozi

Prilog 1 - rezultati mjerenja za 500 pokretanja

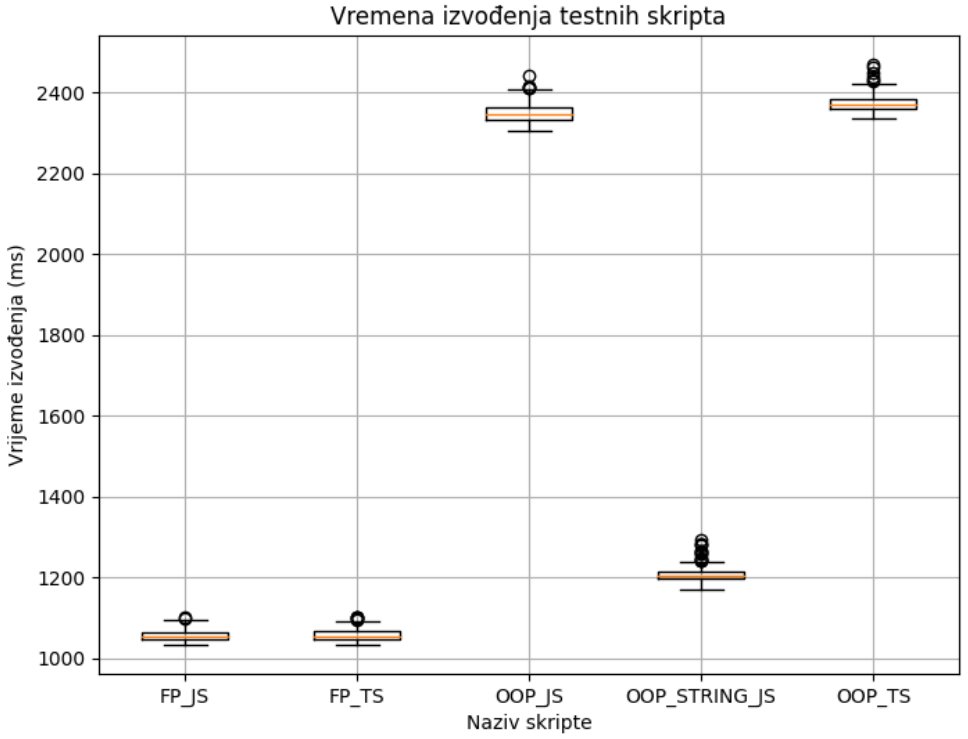
Dijagrami pravokutnika za rezultate mjerenja vremenske i prostorne složenosti algoritama obrade meteoroloških podataka, gdje su testne skripte pokrenute ciklički 500 puta.

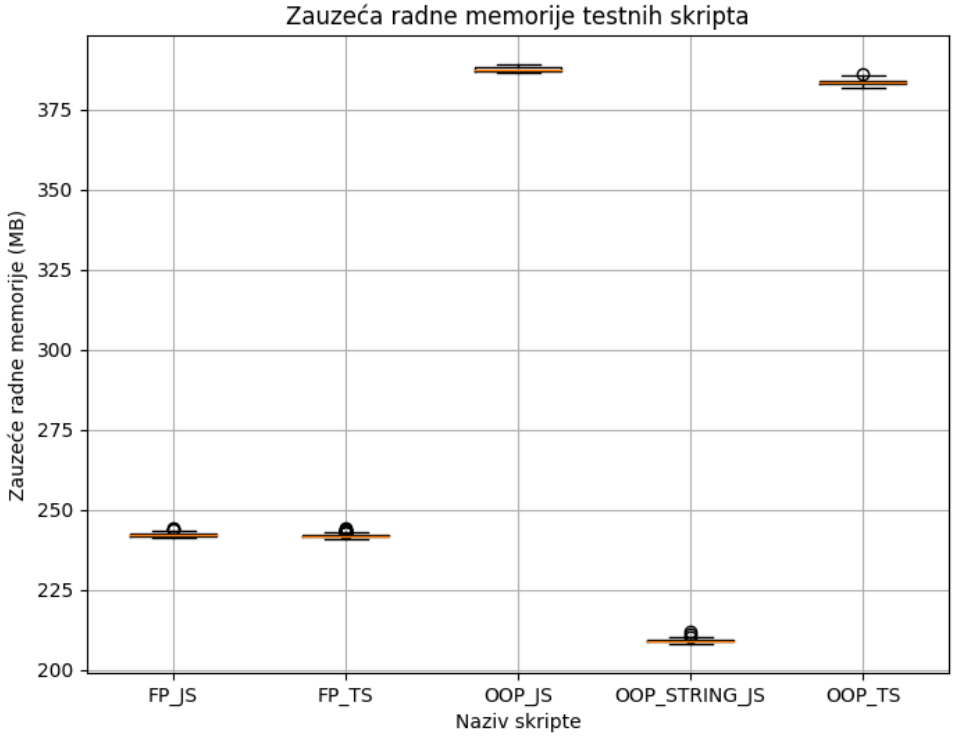




Prilog 2 - rezultati mjerenja bez uključenog automatskog sakupljanja smeća

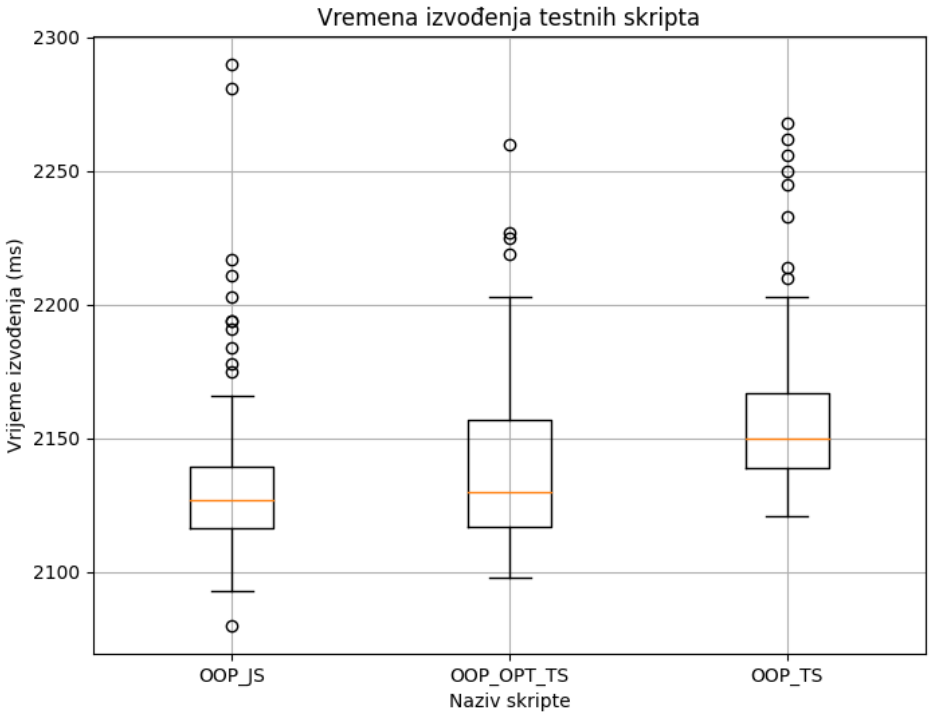
Dijagrami pravokutnika za rezultate mjerenja vremenske i prostorne složenosti algoritama obrade meteoroloških podataka, gdje je *isključeno* automatsko sakupljanje smeća. Testne skripte pokrenute su ciklički 150 puta.

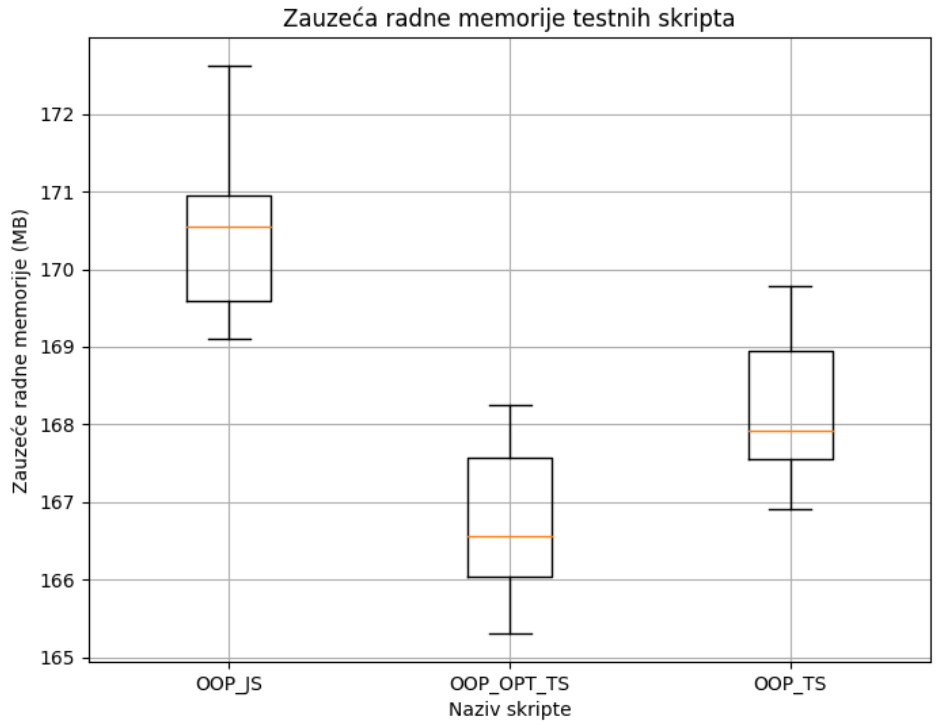




Prilog 3 - rezultati mjerenja optimizirane varijante objektno-orijentiranog TypeScripta

Dijagrami pravokutnika za rezultate mjerenja vremenske i prostorne složenosti algoritama obrade meteoroloških podataka sljedećih skripata: OOP_JS, OOP_TS i OOP_OPT_TS (optimizirana varijanta objektno-orijentiranog TS-a). Testne skripte pokrenute su ciklički 100 puta.





Sažetak

Imena autora: Tin Tomašić, Daniel Škrlac
Naslov rada: Analiza praktične učinkovitosti funkcijskog i objektno-orijentiranog programiranja u kontekstu obrade podataka

Istraživački rad analizira učinkovitost funkcijskog i objektno-orijentiranog programiranja u kontekstu obrade prikupljenih meteoroloških podataka na skriptnim jezicima JavaScript i TypeScript. Kroz sustavni pregled koncepata funkcijskog i objektno-orijentiranog programiranja te praktičan eksperiment ponovljenog mjerenja, funkcijska se implementacija pokazala prostorno i vremenski efikasnijom, ali se objektno-orijentirana varijanta optimizirala i približila složenosti funkcijske. Istraživanje pokazuje da obje paradigme programiranja imaju svoje prednosti i nedostatke, te da implementaciju određene paradigme treba prilagoditi specifičnim potrebama i zahtjevima konteksta obrade podataka, kako bi se postigao optimalan rezultat. Rad također ističe ograničenja ovog istraživanja i ukazuje na rastuću popularnost funkcijskog programiranja zbog napretka u znanosti o podacima i strojnom učenju.

Ključne riječi: objektno-orijentirano programiranje, funkcijsko programiranje, programske paradigme, eksperiment

Summary

Authors: Tin Tomašić, Daniel Škrlac
Title: Analysis of the practical effectiveness of functional and object-oriented programming in the context of data processing

The research paper analyzes the effectiveness of functional and object-oriented programming in the context of processing collected meteorological data in the scripting languages JavaScript and TypeScript. Through a systematic review of the concepts of functional and object-oriented programming and a practical experiment of repeated measurement, the functional implementation proved to be more efficient in terms of space and time. Still, the object-oriented variant was optimized and approached the complexity of the functional one. The research shows that both programming paradigms have their advantages and disadvantages and that the implementation of a particular paradigm should be adapted to the specific needs and requirements of the data processing context, in order to achieve an optimal result. The paper also highlights the limitations of this research and points to the growing popularity of functional programming due to advances in data science and machine learning.

Key words: object-oriented programming, functional programming, programing paradigms, experiment

Životopisi

Tin Tomašić

Zovem se Tin Tomašić, 22 su mi godine i dolazim iz grada Ludbrega. Trenutno sam treća godina revidiranog prijediplomskog studija Informacijski i poslovni sustavi na Fakultetu organizacije i informatike Varaždin (FOI). Programiranje me privuklo već u osnovnoj školi, a intenzivnijim se programiranjem počinjem baviti 2018. godine kada sam upisao međunarodni smjer (IBDP) na Prvoj gimnaziji Varaždin te uzeo informatiku i matematiku na najvišem nivou. Odličan se uspjeh nastavio i na FOI-u te sam već nakon prve godine postao demonstrator na kolegijima vezanim uz matematiku - Matematika 1, Matematika 2 i Matematičke metode za informatičare. Moje predznanje informatike (i matematike) pokazalo se već kao brucoš - nalazio sam se u 3% najboljih studenata generacije prema prosjeku. Međutim, uz još više uloženog rada i predanosti u studiranje, prošle sam godine uspio doći na sam vrh, a za što sam od FOI-a dobio Dekanovu nagradu za najbolji prosjek generacije. Trenutno područje interesa su mi Web tehnologije, servisi i sučelja.

Daniel Škrlac

Zovem se Daniel Škrlac, 21 mi je godina i dolazim iz grada Jastrebarsko. Trenutno sam treća godina revidiranog prijediplomskog studija Informacijski i poslovni sustavi na Fakultetu organizacije i informatike (FOI) u Varaždinu. Moji neki počeci programiranja započeli su u srednjoj školi za čije vrijeme sam odlučio kako želim usmjeriti svoje obrazovanje u smjeru informacijskih tehnologija. Na drugoj godini studija na FOI-u, nakon odslušanog predmeta OWT (Osnove Web tehnologija) dodatno su me zainteresirala područja Web tehnologija te sam shodno tome upisao ljetnu akademiju usmjerenu stjecanju dodatnog znanja iz područja razvoja Web aplikacija korištenjem Java programskog jezika i SpringBoot programskog okvira.