Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/03BE16Kapetina

ANALIZA SOLID PRINCIPA U PROGRAMSKOM JEZIKU JAVASCRIPT

SOLID PRINCIPLES ANALYSIS IN JAVASCRIPT

Milica Kapetina, *Fakultet tehničkih nauka, Novi Sad*

Oblast – Elektrotehničko i računarsko inženjerstvo

Kratak sadržaj – *Pružen je opis SOLID principa i dati su primeri njihove primene u JavaScript-u, uz osvrt i na primenu u TypeScript-u. Opisani su i dizajn šabloni koji se mogu koristiti pri implementaciji SOLID principa.*

Ključne reči: *SOLID principi, dizajn principi, dizajn šabloni, JavaScript, TypeScript, objektno orijentisani programski jezici*

Abstract – *A description of SOLID principles with examples on how to apply them in JavaScript, and TypeScript also. Design patterns that help apply SOLID principles are also described.*

Keywords: *SOLID principles, design principles, design patterns, JavaScript, TypeScript, Object oriented programming languages*

1. UVOD

JavaScript je slabo tipiziran jezik i nema koncepte interfejsa i klasa, kakvi postoje u objektno orijentisanim programskim jezicima. Zbog toga je primena SOLID [1] principa, koji su pisani za objektno orijentisane programske jezike, u JavaScriptu drugačija od njihove primene u objektno orijentisanim jezicima. Da bi se neki od principa primenili potrebno je imitirati interfejs, ali tako da suština i efekat principa ostanu isti. SOLID principi su principi koji govore o tome kako dizajnirati softver da bude što čitljiviji i lakši za održavanje. Primenom ovih principa se takođe smanjuju greške pri kodiranju i omogućava se mnogo lakše menjanje delova koda u budućnosti kada nastane potreba za izmenama. Ako se nikakvi principi programiranja ne primenjuju, tada je kod vrlo podložan greškama, delovi softvera su međusobno jako zavisni i svaka izmena nekog dela bi zahtevala i izmene u drugim delovima koji inače ne bi trebalo da se menjaju.

2. DIZAJN PRINCIPI I ŠABLONI PROGRAMIRANJA

Dizajn principi i šabloni programiranja [2] pomažu u pisanju kvalitetnijeg koda. To nisu zahtevi koji se moraju poštovati već smernice koje bi bilo dobro, ako mogu, da se primene u pisanju softvera. Ne treba stremiti tome da se primene svi principi koji postoje, već je vrlo važno dobro razumeti sve principe, njihove mane i prednosti, efekte koje izazivaju i naspram tog znanja i poznavanja problema koji se rešava, odlučiti se za one principe koji zaista najviše odgovaraju i koji će doneti dobre efekte svojom primenom. Često se dešava da, preteranim pokušavanjem da se iskoriste neki principi, se zapravo postigne suprotan efekat, to jest, dobije se prekompleksan kod koji nije čitljiv i vrlo ga je teško održavati jer je teško snalaziti se u njemu. SOLID principi su jedni od najčešće primenjivanih principa, ali osim njih postoje i drugi poznati kao što su KISS (eng. Keep it simple, stupid), YAGNI (eng. You aren't going to need it), DRY (eng. Don't repeat yourself) i tako dalje. Svi ti principi, na svoj način, doprinose čitljivijem kodu, sa manje grešaka i koji je lakši za testiranje.

Akcenat je na pisanju što modularnijih delova koda, koji su međusobno nezavisni, tako da svaki deo koda ima svoju jednu odgovornost. Ako bi se desila greška, mnogo bi se lakše otkrila u modularnom kodu nego u kodu gde su svi delovi međusobno isprepleteni i čvrsto povezani. Takođe, dizajn šabloni nude načine na koje se treba programirati tako da se dobija što modularniji i čitljiviji kod. Dizajn šabloni se dele u tri glavne grupe: konstrukcioni, strukturalni i bihevioristički.

Konstrukcioni šabloni daju razne načine za kreiranje objekata, neki od njih su Konstruktor, Prototip, Fabrika, Singleton…

Strukturalni šabloni govore o kompoziciji objekata i veza između njih, tako da kada se deo sistema promeni, ne mora da se menja ceo sistem. Neki od njih su Dekorator, Façade, Adapter…

Bihevioristički šabloni se koriste za bolje osposobljavanje što bolje i kvalitetnije komunikacije između objekata. To su na primer, Observer, Iterator, Mediator…

3. SOLID PRINCIPI

SOLID je akronim koji se sastoji iz 5 principa: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation i Dependency Inversion. Ovih 5 principa se najčešće koriste i imaju veoma velike pozitivne efekte na softver i kod u kome se primene.

3.1. Single Responsibility princip

Princip jednostruke odgovornosti govori o tome da svaka klasa treba da ima samo jednu odgovornost. To ne znači da treba da ima samo jednu funkciju. Ona može imati i više funkcija ako one zajedno izvršavaju jedan zadatak i zajedno čine samo jedan razlog za promenu te klase. Na primer, klasa koja ima funkcije za slanje i prijem poruka i funkcije za otvaranje i zatvaranje konekcije, nema ispoštovan ovaj princip. Princip jednostruke odgovornosti bi se ispoštovao ako bi se razdvojila klasa na dve klase,

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Aleksandar Kupusinac, vanr. prof.

jednu koja prima i šalje poruke, i drugu koja otvara i zatvara konekciju. Time bi svaka klasa imala po jednu odgovornost i jedan razlog za promenu.

3.2. Open-Closed princip

Otvoren-zatvoren princip govori o tome da klasa treba biti otvorena za proširenja, a zatvorena za izmene. To znači da ako je potrebno uneti neke nove funkcionalnosti, na primer, onda ne treba menjati potrebnu klasu, već je proširiti. U ovom kontekstu proširenje klase bi podrazumevalo nasleđivanje te klase i implementacija njenih funkcija onako kako nove funkcionalnosti zahtevaju. Ovaj princip se postiže upotrebom apstrakcija i polimorfizmom.

3.3. Liskov Substitution princip

Princip Liskove zamene kaže da sve podklase neke nadklase mogu da se zamene svojom nadklasom a da se pri tome ponašanje programa ne promeni. Na mestu gde se koristi objekat podklase može se staviti da je tip objekta zapravo tip nadklase i da se ponašanje programa ne promeni i ne pokvari. Na taj način sve podklase neke nadklase mogu da se koriste na potpuno isti način, samo što će svaka podklasa imati svoju implementaciju određenih funkcija i njih izvršavati na različite načine. Poštovanjem ovog principa dobija se i ispoštovan Otvoren-zatvoren princip.

3.4 Interface Segregation princip

Princip razdvajanja interfejsa potencira to da se ne kreiraju interfejsi sa puno funkcija, već taman onoliko koliko će ih, klasa koja nasledi taj interfejs, implementirati. U suprotnom klasa koja implementira taj interfejs će imati i funkcije koje nikada neće koristiti. Zbog toga ovaj princip savetuje prvo dobru analizu interfejsa i klasa koje će ga nasleđivati, pa tek onda kreiranje samog interfejsa. Time se dobija mnogo čitljiviji kod jer nema viška linija koda koje se nigde neće koristiti.

3.5. Dependency Inversion princip

Princip inverzije zavisnosti kaže da moduli višeg nivoa ne treba da zavise od modula nižeg nivoa, a takođe i apstrakcije ne treba da zavise od detalja, već detalji od apstrakcija. To zapravo znači da treba da se kreiraju apstrakcije ili interfejsi koji predstavljaju ugovor. Apstrakcija samo govori šta neka klasa treba da radi, ne i kako. Klase koje će taj interfejs implementirati treba da znaju šta sve taj interfejs ima od funkcija i na taj način zapravo detalji zavise od apstrakcija, a apstrakcije ne zavise od detalja.

4. OBJEKTI U JAVASCRIPT-U

JavaScript je ujedno i objektno orijentisan programski jezi i funkcionalan programski jezik. Postoji mnogo polemika oko toga kojoj tačno vrsti pripada, međutim on na određene načine, može se reći, pripada obema vrstama. JavaScript je objektno orijentisan jezik jer, iako nema koncept klasa, koristi objekte. Za kreiranje objekata u JavaScriptu mogu se koristiti određeni dizajn šabloni kao što su konstruktorski šabloni, singleton, šabloni prototipskog nasleđivanja, šabloni modula i tako dalje. Singleton šablon predstavlja kreiranje objektnog literala koji ima samo jednu instancu. Ako se želi više instanci, potrebno je kopirati isti deo koda više puta za svaku instancu. Na slici 1 prikazan je objektni literal u JavaScript-u.

```
var User = {
    name: "John",
    age: 20,
    printUser: function() {
        return "I am John";
    }
}
```

SLIKA 1. OBJEKTNI LITERAL

Konstruktorski šablon predstavlja korišćenje konstruktorske funkcije za kreiranje objekta. Konstruktorska funkcija je funkcija koja se ponaša kao konstruktor, to jest, vraća kreirani objekat kao povratnu vrednost. Šabloni modula su slični konstruktorskim šablonima, samo što mogu da imaju i privatna polja i funkcije kojima se ne može pristupiti izvan objekta. Takođe postoji i šablon otkrivajućih modula, koji pokazuje da se kreiraju objekti sa privatnim poljima i javnim funkcijama generičkog naziva koje će ta polja čitati ili menjati. Šablon prototipskog nasleđivanja pokazuje kako se mogu naslediti objekti u JavaScriptu nasleđivanjem prototipa objekta. Prototipskim nasleđivanjem kreira se *Protoype Chain*, to jest, lanac prototipa koji povezuje prototipove svih nasleđenih objekata. Vrhovni objekat je tipa *Object* i njega svaki objekat nasleđuje, to jest, njegov prototip. Nasleđivanjem prototipa objekat nasleđuje i svojstva drugog objekta, pa i njegov tip. Ovaj šablon je vrlo važan za shvatanje primene SOLID principa u JavaScriptu. Na slici 2 prikazan je primer prototipskog nasleđivanja gde „klasa“ Square nasleđuje „klasu“ *Rectangle.*

```
function Rectangle(length, width){
    this.length = length;
    this.width = width;
}
Rectangle.prototype.getArea = function() {
    return this.length * this.width;
}

function Square(size){
    this.length = size;
    this.width = size;
}
Square.prototype = new Rectangle();
Square.constructor = Square;

var rect = new Rectangle(5, 10);
var square = new Square(6);
```

SLIKA 2 PROTOTIPSKO NASLEĐIVANJE

ECMAScript2015 donosi velike novine u JavaScript, u obliku koncepta klase i nasleđivanja klase, koje i dalje u pozadini zapravo radi prototipsko nasleđivanje. Ovo donosi povećanje čitljivosti koda i razumljivosti jer sintaksno JavaScript od tada postaje mnogo bliža i sličnija objektno orijentisanim programskim jezicima. Šablon

fabrika predstavlja funkciju koja omogućava kreiranje objekata bez poznavanja kog tipa će oni biti.

Podklase pri instanciranju biraju koju klasu žele da instanciraju. Ovo je vrlo rasprostranjen šablon, koji se koristi u raznim vrstama softvera.

5. PRIMENA SOLID PRINCIPA U JAVASCRIPT-U

Primena SOLID principa u JavaScript-u je dosta zanemarena. S obzirom na to da je JavaScript slabo tipiziran jezik i da je veoma fleksibilan, ne forsira primenu nikakvih principa i vrlo lako može doći do neke greške u kodu. Na primer, česte greške se dešavaju kada objekat promeni svoj tip i više nema polja koja su se očekivala.

Takođe, pošto nema koncepte interfejsa i apstraktnih klasa, primena nekih SOLID principa je otežana, međutim interfejs se može na određene načine imitirati i tako uspeti u pokušaju da se primene SOLID principi. TypeScript je, na primer, programski jezik koji je nastao od JavaScripta i u njemu ovih problema nema. On podržava statičko tipiziranje objekata i takođe ima koncepte interfejsa. Samim tim dosta je lakše i intuitivnije primeniti SOLID principe u TypeScript-u.

5.1. Single Responsibility Principle

Princip jednostruke odgovornosti se može lako primeniti u JavaScriptu. Da bi se primenio, potrebno je dobro razumeti problem koji se rešava datim objektom. Potrebno je pokušati, na osnovu znanja o objektu, kreirati takav objekat da ima samo jednu odgovornost u sebi, to jest, samo jedan razlog da se promeni. Problem kod ovog principa nastaje kada se previše pokušavaju smanjiti objekti i time se dobija ogroman broj sitnih objekata čime se mnogo poveća kompleksnost koda.

Zbog toga je vrlo važno razumeti šta taj objekat treba da radi i dobro razmisliti da li je poštovanje ovog principa zaista potrebno u tom slučaju. Ako je objekat mali i njegova funkcionalnost zanemarljiva, pri čemu se zna da se neće u budućnosti menjati, onda uglavnom nema potrebe da se vodi računa o ovom principu.

5.2. Open-Closed Principle

Otvoren-zatvoren princip se postiže apstrakcijom klase koja treba da bude otvorena za proširenja i zatvorena za izmene. Ovo se dobija kreiranjem klase koja nema implementirane funkcije, a pri čemu će klase koje nasleđuju tu klasu da implementiraju na svoje načine te funkcije.

Kao što je već spomenuto, u JavaScriptu se nasleđivanje može izvršiti prototipskim nasleđivanjem tako što se prototip nadklase kopira u prototip podklase, ili korišćenjem ECMAScript2015 standarda gde postoji ključna reč „extends“ kojom se može naslediti neka klasa. Time bi se moglo, za svaki novi potreban tip kreirati klasa koja će naslediti nadklasu.

5.3. Liskov Substitution Principle

Kao što je već spomenuto, princip Liskove zamene se svodi na to da objekat podklase može da se zameni objektom nadklase a da se pri tom ponašanje programa ne promeni. U JavaScript-u se može postići prototipskim nasleđivanjem jer tada objekat podklase prima tip nadklase koju nasledjuje. Međutim, ovaj princip je više primenljiv u TypeScript-u gde postoje tipovi i provere tipova. Moglo bi se, na primer, u funkciji, koja može da primi bilo koji tip podklasa koje nasleđuju jednu nadklasu, staviti da prima objekat tipa nadklase kao ulazni parametar. Tada bi se funkciji mogao proslediti bilo koji objekat podklasa i ne bi došlo do greške.

Ovime se štedi mnogo linija koda, jer se dobija generalizacija funkcije, i umesto više funkcija ili umesto više uslova provere unutar jedne funkcije, može se iskoristiti samo jedna generička funkcija. Takođe, ovo je mnogo čitljivije, jer uglavnom nije toliko važno koje sve podklase postoje, koliko je važno koje su ponašanje nasledile od nadklase i koje funkcije postoje u nadklasi.

5.4 Interface Segregation Principle

Princip razdvajanja interfejsa se može postići u JavaScriptu tako što će se objekti koji imitiraju interfejse kreirati tako da sadrže što manje funkcija, to jest, tačno onoliko koliko će objekti koji implementiraju taj interfejs koristiti. Ovo je dosta korisno u TypeScript-u koji zaista ima interfejse i može dosta doprineti smanjenju viška koda koji se nikad ne izvršava i ne koristi.

Takođe, klase koje implementiraju taj interfejs, neće morati da implementiraju i metode koje im neće nikad trebati. Ovde opet treba u potpunosti razumeti ulogu i funkciju interfejsa, kao i zadatak koji treba da izvršava, da bi se moglo planirati unapred kakve će sve izmene biti moguće u budućnosti.

5.5 Dependency Inversion Principle

Princip inverzije zavisnosti može se u JavaScriptu postići prototipskim nasleđivanjem objekata koji imitiraju interfejs, ili pomoću funkcija višeg reda. Funkcije višeg reda su funkcije koje primaju druge funkcije kao ulazni parametar (eng. *callback function*) ili kreiraju novu funkciju koju vraćaju kao povratnu vrednost.

Na ovaj način dobija se apstraktna funkcija (funkcija višeg reda) koju definišu, druge funkcije, funkcije koje joj se proslede kao ulazni parametar. Time se postiže to da apstrakcija ne zavisi od detalja, već detalji od implementacije, što je i suština ovog principa. Samim tim povećava se iskoristivost ove funkcije u mnogo više slučajeva upotrebe.

6. ZAKLJUČAK

Iz priloženog može se zaključiti da su SOLID principi u JavaScriptu poprilično zanemareni, jer da bi se primenili, potrebno je malo dodatnog truda. Međutim, primena SOLID principa je od velikog značaja jer doprinosi kreiranju čitljivijeg koda i koda koji je lak za održavanje. Kada se u budućnosti budu unosile neke izmene mnogo ih je lakše uneti ako se unose na jednom mestu, a to bi bio slučaj ako su SOLID principi ispoštovani.

Ako nisu, onda je potrebno menjati veliki deo koda, jer su mnogi delovi čvrsto povezani i to može koštati mnogo i izazvati dosta grešaka u kodiranju.

Takođe, kada se izmene unose posle dužeg vremena, čak i autor koda zaboravi šta je sve i gde radio, pa je i zbog toga vrlo važno da je kod čitljiv i da je lako snalaziti se u njemu iako se ne poznaje u potpunosti ceo kod.

Osim povećane čitljivosti i razumljivosti koda, poštovanjem SOLID principa dobija se i veća otpornost na greške. Modularnost koda dovodi do toga da manje dolazi do grešaka jer su svi delovi koda modularni, to jest, izolovani.

Čak i ako se desi greška, mnogo ju je lakše pronaći kada je kod modularan, jer se tačno zna koji deo koda ima koju odgovornost. Ako se zna u kojoj odgovornosti se desila greška, znaće se i deo koda u kom je greška nastala. Korišćenjem interfjesa, nasleđivanjem i polimorfizmom dobija se dosta generalizacije i razdvajanja zavisnosti između delova koda, što doprinosi smanjenju grešaka, lakšem testiranju, ali i boljoj čitljivosti koda. SOLID principi nisu zahtevi već saveti kako bolje programirati, uz manje grešaka i lakše menjanje u budućnosti. Ne treba se uvek truditi da se po svaku cenu implementiraju svi SOLID principi, jer to često može dovesti do prekompleksnog koda.

Zapravo, treba prvo dobro shvatiti sve SOLID principe, njihove mane, prednosti, načine primene i efekte koje proizvode. Potom, treba dobro razumeti koji se problem rešava, kakve su njegove osobine i kako će se on menjati u budućnosti. Tek onda treba dobro razmisliti koji SOLID principi bi mogli najviše pomoći u rešavanju tog problema, bez suvišnog komplikovanja i nagomilavanja koda.

S obzirom na to da se JavaScript ubrzano i stalno razvija, i to ne samo JavaScript već i drugi jezici i radni okviri koji su nastali od JavaScript-a, vrlo je verovatno da će on sve više vremenom ličiti na objektno orijentisane programske jezike i da će se u potpunosti moći koristiti svi principi koje koriste i čisti objektno orijentisani jezici.

7. LITERATURA

[1]Robert C. Martin (2000), „Design Principles and Design Patterns“, (pristupljeno u septembru 2018. godine)

[2]Addy Osmani (2012), “Learning JavaScript Design Patterns”, (pristupljeno u septembru 2018. godine)

Kratka biografija:

Milica Kapetina rođena je 31.05.1994. godine u Sarajevu. Završila je osnovnu školu “Prva vojvođanska brigada” u Novom Sadu. 2009. godine. Srednju medicinsku školu “7. april” završila je 2013. godine i iste se upisala na Fakultet tehničih nauka u Novom Sadu, odsek Računarstvo i automatika. Osnovne studije na Fakultetu tehničkih nauka završila je 2017. godine i iste godine se upisala na master studije, takođe na Fakultetu tehničkih nauka.