Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/02BE43Momcilovic

OSNOVNI PRINCIPI, ŠABLONI I PRAVILA ZA PISANJE ČISTOG KODA

BASIC PRINCIPLES, PATTERNS AND RULES FOR WRITING CLEAN CODE

Saša Momčilović, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U okviru rada opisane su najbolje prakse koje dovode do čistog dizajna prilikom rešavanja određenog problema. Pored dizajna akcenat je stavljen i na implementaciju, tj. na konkretne preporuke prilikom pisanja koda kako bi bio na visokom stepenu održivosti.*

Ključne reči: *Objektno orijentisano programiranje, SOLID principi, dizajn šabloni, C#, čist kod.*

Abstract – *This document describes the best practices that lead to a clean design when solving a particular problem. In addition to design, emphasis is placed on the implementation, i.e. on concrete recommendations when writing code to be of a high degree of maintenance.*

Keywords: *Object oriented programming, SOLID principles, design patterns, C#, clean code.*

1. UVOD

U radu biće više reči o tome kako napisati kod sa što većim stepenom održivosti, vodeći računa prvenstveno na kvalitet dizajna, a nakon toga i na kvalitet samog koda. Takođe, biće reči i o tradicionalnom načinu programiranja, kao i o problemima koje je imao proceduralni pristup programiranju, a koji su bili motiv za uvođenje velikih promena u ovoj oblasti.

Kao osnovni korak za uprošćavanje kompleksnog koda koji je prouzrokovan proceduralnim pristupom programiranju, biće predstavljeni aspekti objektno orijentisanog programiranja koji utiču na poboljšanje čitkosti koda i rešavanje problema proceduralnog programiranja. Nakon toga, akcenat će biti na SOLID principima. Biće opisano kako pridržavanje ovih principa pomaže da se dizajn i kod učine što kvalitetnijim i čitljivijim. Nakon SOLID principa prelazi se na dizajn šablone koje strukturno dosta pomažu poboljšanju dizajna, a samim tim i koda. Nakon aspekata koji se u većoj meri odnose na dizajn, potrebno je opisati kako da dobar dizajn ne bude narušen pisanjem nečitkog koda.

2. MOTIVACIJA

Rešenja problema u oblasti informacionih tehnologija su podložna promenama zahteva od strane klijenata. S tim u vidu, potrebno je bilo ustanoviti određena pravila koja će prouzrokovati pisanje koda takvog da je uz najmanje napore moguće dodati implementacije novih zahteva, a da se pritom ne naruši dotadašnja implementacija.

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

U ranim fazama sveta informacionih tehnologija nailazilo se na probleme koji su znatno povećavali vreme potrebno da se neki posao obavi. Kod koji je bio pisan je bio neodrživ, nastao je veliki nedostatak programera.

3. OBJEKTNO ORIJENTISANO PROGRAMIRANJE

Kao što je već napominjano, programiranje je evoluiralo kroz više faza. Jedna od njih je bilo proceduralno programiranje za čije je probleme bilo potrebno naći adekvatno rešenje. Problemi proceduralnog programiranja u najvećoj meri rešeni su ustanovljavanjem objektno orijentisanog programiranja. Uvođenjem objektno orijentisanog progra-miranja, rešeni su problemi nepotrebnog čuvanja globalnih podataka, nemogućnost ponovne iskoristivosti koda, redudantnost koda, pravljanje složenih programa i tako dalje. Rešavanje pomenutih problema omogućeno je kroz četiri osnovna principa objektno orijentisanog programiranja, a to su: enkapsulacija, nasleđivanje, apstrakcija i polimorfizam.

Objektno orijentisano programiranje ujedinjuje funkcije i varijable u jednu celinu nazvanu klasom. Na taj način je omogućeno odvajanje logike u celine, tako da se svaka logika može dodeliti celini kojoj pripada.

3.1. Enkapsulacija

Enkapsulacija je jedan od četiri osnovna koncepta objektno orijentisanog programiranja. Omogućava zaštitu atributa i funkcija objedinjenih u celinu. Zaštita elemenata klase od klijentskih klasa, tj. klasa koji je koriste, omogućena je kroz modifikatore pristupa elementima klase. Klasi koja koristi enkapsuliranu klasu se omogućava da vidi samo neophodne njene elemente, dok su interni elementi klase sakriveni, na taj način se klasa štiti spoljnih promena koje se ne bi trebale desiti, uprošćava se interfejs klase što daje određenu fleksibilnost prilikom dalje implementacije i testiranja klase.

3.2. Nasleđivanje

Nasleđivanje je koncpet objektno orijentisanog programiranja koji omogućava izvođenje novih klasa iz postojeće, tako da nove klase dobijaju osobine bazne klase uz mogućnost dodavanja novih osobina. Osnovni oblici nasleđivanja su: jednostruko, višestruko, hijerarhijsko, višeslojno i kombinovano nasleđivanje. Sa stanovišta čistog koda nasleđivanje pruža: izbegavanje ponovnog pisanja koda, ponovna iskoristivost koda, jednostavnije proširivanje koda.

3.3. Apstrakcija

Apstrakcija je koncept koji omogućava definisanje ponašanja u baznoj klasi koje će biti implementirano u

klasi naslednici. Apstrakcija se postiže apstraktnim klasama koje mogu imati apstraktne članove koji se obeležavaju ključnom rečju ***abstract*** i koje nemaju telo. Prilikom kreiranja konkretnih klasa iz definisanih apstrakcija, apstraktnim članovima je potrebno implementirati ponašanje. Osim apstraktnih klasa apstrakcija se može postići i interfejsima.

3.4. Polimorfizam

Polimorfizam je koncept koji omogućava izvršavanje istih funkcija na različit način. Pomenute funkcije, tj. metode se nalaze u konkretnim klasama koje su podklase zajedničkoj baznoj klasi. Sa aspekta čitkog koda polimorfizam ima veliki udeo. Njime se može izbeći ne baš najčistije implementacije koje imaju veliki broj ***if else*** grananja. Polimorfizmom je omogućeno napraviti više objekata takvih da predstavljaju instance klasa koji su podklase iste bazne klase. Te klase će izvršavati istu metodu na različit način.

4. S.O.L.I.D principi

SOLID je kolekcija najboljih praksi objektno orijentisanih dizajn principa koji se mogu primeniti na novi dizajn, omogućavajući ostvarenje raznih ciljeva kao što su, veća održivost, intuitivna lokacija koda, slaba povezanost komponenti i tako dalje. [1]

SOLID je akronim koji predstavlja sledeće principe:

- SRP (***The Single Responsibility Principle***)
- OCP (***The Open Closed Principle***)
- LSP (***The Liskov Substitution Principle***)
- ISP (***The Interface Segregation Principle***)
- DIP (***The Dependency Inversion Principle***)

4.1. *The Single Responsibility* princip

SRP je princip koji nalaže da se svaka pojedinačna odgovornost nalazi u okviru tačno jedne klase. U situacijama kada se više odgovornosti nađe u okviru jedne klase može doći do problema prilikom dodavanja novog koda vezanog za jednu odgovornost. U tim situacijama može doći do narušavanja ostalih odgovornosti koji se nalaze u toj klasi. Moduli u koje je potrebno rasporediti odgovornosti trebaju imati što niži stepen međusobne povezanosti.

```
public class Student
{
    public void Save()...

    public double DoTheExam(Exam exam)...

    public Student FindById(long id)...

}
```

Slika 1. *Klasa student koja narušava SRP princip*

Na slici 1 nalazi se klasa ***Student*** koja ima metode ***Save***, ***DoTheExam*** i ***FindById.*** Svaka od ovih metoda jeste usko povezana sa studentom, međutim ne predstavlja svaka metoda i odgovornost klase student. Klasa definisana kao na slici ima odgovornost rada sa kolekcijom studenata, što ne bi trebala biti odgovornost pojedinačnog objekta klase ***Student***. Promene u implementaciji načina skladištenja studenata mogu uticati na logiku koja bi se zaista trebala nalaziti u klasi ***Student.*** Da bi se ovakav problem rešio potrebno je u jednoj klasi držati samo jednu odgovornost, tj. izdvojiti logiku vezanu za skladište podataka u posebnu klasu. Ta klasa bi trebala preuzeti metode za čuvanje i za pronalaženje studenata po jedinstvenom idenfitikatoru.

4.2. *The Open Closed* princip

OCP princip nalaže da svaki modul treba biti: otvoren za proširenje i zatvoren za modifikaciju. Implementacije u kojima su klijentska i serverska klasa konkretne, je dobar osnov za primenu OCP principa. Da bi se primenio OCP princip potrebno je da serverska klasa bude implementacija određene apstrakcije. Takođe potrebno je i da klijentska klasa ne barata sa konkretnom implementacijom, već sa pomenutom apstrakcijom.

Počnimo od primera u kojem je potrebno implementirati rad sa zaposlenima u određenom sistemu. Konkretno, u zavisnosti od tipa zaposlenog potrebno je na određeni način izračunati platu koja zavisi od procenta na platu i dodatnog novca. Izvešćemo klasu ***BonusCalculator*** u kojoj će se u metodi ***CalculateBonus*** računati bonus za prosleđenog zaposlenog. U metodi je potrebno ispitati kog je tipa zaposleni, što će se čuvati u klasi ***Employee***, i na osnovu tipa računati bonus. Prilikom dodavanja novog tipa potrebno je menjati metodu novim slučajem. Ovakav pristup krši OCP. Kao što je pomenuto, potrebno je uvesti apstrakciju nad klijentskom klasom, tj. u ovom slučaju nad klasom ***Employee.*** Pomenuta apstrakcija će imati onoliko implementacija koliko će biti vrsta zaposlenih, i u njima će biti podaci o procentu bonusa i dodatnom novcu za svakog zaposlenog. ***BonusCalculator*** klasa će raditi sa apstrakcijom i neće imati potrebu za menjanjem.

4.3. *The Liskov Substitution* princip

Formalna definicija LSP principa koju je iznela Barbara Liskov, po kojoj je ovaj princip i dobio naziv glasi: **„**Ako je S podtip tipa T, tada objekti tipa T mogu biti zamenjeni objektom tipa S, bez prekida rada programa**.“** [2]

Funkcije koje koriste reference na baznu klasu moraju biti sposobne da koriste objekte klasa naslednica bez znanja o tome. [3] Osim pomenutih definicija za LSP princip bitno je da klasa naslednica u svojoj implementaciji ne sme bacati nikakav novi izuzetak. Loš primer modelovanja koji nije u skladu sa ovim principom je u sledećem primeru. Ukoliko imamo klasu ***Employee*** koja ima referencu ka svom menadzeru. Svaki konkretni zaposleni može implementirati klasu ***Employee.*** Međutim, u hijerarhiji zaposlenih nemaju svi menadzera, te je potrebno tu logiku izmestiti u novu klasu koja predstavlja zapslene koji mogu imati menadzera, a koja nasleđuje klasu ***Employee***. Sve klase koje predstavljaju zaposlene koji nemaju menadzera će naslediti klasu ***Employee***. Dok će klase koje predstavljaju zaposlene koje imaju menadzere naslediti novokreiranu klasu.

4.4. *The Liskov Substitution* princip

LSP se bavi nedostatkom koji prouzrukuju kompleksni interfejsi koji po svojoj definiciji primoravaju sve klase koje ga implementiranju da definišu implementaciju svih metoda intefejsa. Ukoliko su ti interfejsi preveliki, tj. sa velikim brojem članova, potrebno je te članove pregrupisati u više manjih interfejsa. Recimo da postoji interfejs ***IFile*** koji ima ***property***-e ***Title, Author, Duration, PageCount*** i metode ***GetVideoSnapshot, Play, Pause, Stop, GetTextContent.*** Pomenuti članovi

interfejsa se mogu koristiti za implementaciju nekog fajla. Međutim ukoliko se radi o tekstualnom fajlu, metode kao što su ***Play*** nemaju svrhu. Potrebno je razdvojiti članove u više interfejsa koji će se moći, po potrebi i u kombinaciji, koristiti tako da ih klase koje predstavljaju bilo koji fajl mogu implementirati bez potrebe implementacije nepotrebnih metoda.

4.5. *The Dependency Inversion* princip

DIP princip se bazira na programiranju spram apstrakcije. tj. oslobađanja aplikacije zavisnosti od konkretnih implementacija određenih komponenti. To oslobađanje zavisnosti se postiže korišćenjem apstrakcija. Osnovna dva načela DIP principa su: 1. Moduli visokog nivoa ne bi trebali da zavise od modula niskog nivoa, oba trebaju zavisiti od apstrakcija. 2. Apstrakcija ne bi trebala zavisiti od detalja. Detalji trebaju zavisiti od apstrakcije. [4].

U situacijama kada se ne koristi apstrakcija iznad implementacija određenih komponenti onemogućeno je podmetanje druge implementacije prilikom promene zahteva od strane klijenata. Ukoliko dođe do novih zahteva, a ovaj princip nije ispoštovan, kod koji će se morati menjati može da prouzrukuje velike probleme i kršiće većinu SOLID principa. Pored pomenutog problema, veoma je otežano jedinično testiranje ukoliko se ovaj princip ne poštuje.

5. DIZAJN ŠABLONI

Dizajn šabloni predstavljaju koncepte rešenja za neke ponavljajuće programerske probleme. Ovakvi šabloni imaju veliki stepen ponovne iskoristivosti i veoma su lako prepoznatljivi u svetu programiranja, te će lako biti uočeni od programera koji za njih znaju i time im olakšati snalaženje u kodu koji je u skladu sa nekim šablonom. Postoje tri osnovne grupe dizajn šablona: kreacioni, strukturalni i šabloni ponašanja.

5.1. Singleton

Singleton je strukturalni dizajn šablon koji omogućava pristup objektu određene klase sa jedne, svima dostupne, pristupne tačke. Takođe klasa koja implementira ovaj šablon treba obezbediti da postoji samo jedna instanca ove klase. ***Singleton*** se može kombinovati sa velikim brojem dizajn šablona.

5.2. Abstract factory

Abstract factory je dizajn šablon koji obezbeđuje kreiranje srodnih ili zavisnih objekata bez navođenja njihovih konkretnih klasa. [5] Predstavlja proširenje ***Factory*** i ***Factory Method*** dizajn šablona. **Abstract factory** mogućava da se korišćenjem osnovnih koncepata objektno orijentisanog programiranja ispoštuju ključni SOLID principi prilikom rešavanja određenih programerskih problema.

5.3. Composite

Composite dizajn šablon predstavlja strukturu rešenja za probleme u kojima je potrebno organizovati objekte u stablo. Mogu postojati dve vrste objekata, oni koji mogu imati još podobjekata i oni koji predstavljaju listove u stablu. Može se koristiti u kombinaciji sa dosta drugih dizajn šablona. Omogućava proširenja u kodu koje je u skladu sa SOLID principima.

5.4. Chain of responsibility

Chain of responsibility je dizajn šablon koji omogućava raspoređivanje objekata u lanac. Lanac se okida tako što klijentska klasa šalje zahtev prvom objektu u nizu, koji ili obradi zahtev ili ga prosledi sledećem objektu i tako dalje do poslednjeg objekta u lancu. Ovaj omogućava dizajniranje rešenja tako da ne narušava OCP i SRP princip.

5.5. Command

Ovaj dizajn šablon jedan je od najčešće korišćenih šablona. Enkapsulira komande u objekte kojima se može manipulisati. ***Command*** šablon razdvaja objekat koji zna kako da izvrši operaciju od objekta koji zna kako da je izvede. Ima veliku primenu prilikom kreiranja WPF aplikacija primenom MVVM šablona. Takođe, velika prednost ovog šablona jeste mogućnost implementacije ***Undo***/***Redo*** mehanizma.

5.6. Null object

Prilikom rešavanja programerskih problema često se nailazi za problem u kome bi jedno od rešenja bilo vraćanje ***null*** reference. U tim situacijama dobro je koristiti ovaj šablon. Kreira se objekat koji ima logiku jednaku logici koju bi klijentska klasa izvršila ukoliko bi u ranijoj implementaciji bio vraćen ***null.*** Često to bude prazna metoda. Ovaj dizajn šablon oslobađa klijentsku klasu od provere ***null***-a***.*** Ovaj šablon se može kombinovati sa ***Singleton, Command, Strategy*** i drugim dizajn šablonima.

5.7. Repository

Repository je šablon koji omogućava implementaciju rada sa skladištenjem podataka tako da razdvaja sloj poslovne logike aplikacije od samog skladišta gde će poslovna logika znati samo za apstrakciju. Tako da, kada dođe do promene zahteva o skladštu, može se menjati implementacija bez uticaja na poslovnu logiku.

5.8. Strategy

Strategy šablon predstavlja jedan od najjednostavnijih dizajn šablona sa veoma širokom primenom. Ovaj dizajn šablon služi za promenu ponašanja određenog objekta tako da on zna samo za apstrakciju, te mu je moguće po potrebi podmetnuti ponašanje koje odgovara jednoj od konkretnih implementacija te apstrakcije.

5.9. Facade

Facade dizajn šablon obezbeđuje interfejs za set operacija koje se dešavaju u određenom delu aplikacije. ***Facade*** oblikuje te operacije u jedan podsistem koji zna kojim komponentama da prosleđuje zahteve koji su došli od drugih podsistema. Ovaj dizajn šablon ima veliki broj varijacija.

6. OSNOVNA NAČELA PISANJA ČISTOG KODA

Osim bitnih koncepata i šablona vezanih za dizajnisanje rešenja, potrebno je ispoštovati i smernice vezane za konkretnu implementaciju koje dovode do kompletno čitljivog i održivog rešenja.

6.1. Smisleno imenovanje

Prilikom pisanja koda veoma bitnu ulogu ima imenovanje promenljivih, klasa, metoda i parametara. Potrebno je davati imena koja otkrivaju nameru člana koji se kreira.

Ukoliko ime zahteva komentar onda je potrebno razmisliti o boljem imenu. Postoje situacije u kojima je kod, koji je potrebno napisati, krajnje jednostavan, međutim i pored toga može dovesti do velike zabune ukoliko dođe do loših imenovanja. Imena klasa trebaju biti imenice, tako da se izbegavaju reči poput ***Processor***, ***Data***, ***Manager*** ili ***Info.***

6.2. Metode

Prilikom pisanja metode treba obratiti pažnju na sve komponente metode: naziv, povratnu vrednost, broj parametara, tip parametara, naziv parametara, dužinu tela metode, strukturu tela metode i tako dalje.

Osnovna pravila vezana za pisanje metoda govore da telo metode treba biti što kraće i da metoda treba imati što manje parametara. Ukoliko je u metodi potrebno koristiti neke od naredba grananja, npr ***if***, treba nastojati da u ***if*** bloku bude samo jedna linija koda, pa makar to bila jedna metoda koja obuhvata više radnji, nazvana tako da iz njenog naziva bude jasno šta će se obavaiti u njoj. Kroz parametre metoda potrebno je prosleđivati samo podatke koji su potrebni metodi. Prilikom pisanja metoda treba izbegavati opcione parametre, pogotovo ako je u pitanju virtuelna metoda. Takođe, dobro je izbegavati ***out*** i ***flag*** parametre.

6.3. Komentari

U situacijama kada se kod pretvori u nečist i neodrživ, veliki deo programera se odluči da doda komentare kako bi olakšao čitanje tog koda. Međutim, najbolja reakcija na takav kod jeste refaktorisanje. Takođe, ukoliko postoji neki deo koda koji nije najčitljiviji ne bi se trebali ostavljati komentari iznad njega. Najbolja opcija u takvim situacijama je ubacivanje tog koda u metodu koja će svojim imenom reći šta taj kod radi. Ukoliko programer u određenim situacijama nije u mogućnosti da odradi neki deo posla odlučuje se da ostavi ***TODO*** komentar, koji služi kao podsetnik. Iako su takvi komentari korisni u današnjim razvojnim okruženjima, treba nastojati izbegavati potrebu za ostavljanje istih.

Dodatne neophodne podatke za određenu metodu ili klasu moguće je dopisati u ***summary*** blok koji se nalazi iznad klase ili njenih clanova.

6.4. Korišćenje uslova

Prilikom grananja koda na više mogućih tokova koriste se ***boolean*** promenljive. Ukoliko je ***boolean*** promenljiva nazvana na adekvatan način ona se može koristiti u ***if*** grananju, bez poređenja sa ***true*** ili ***false*** vrednostima, jer bi se time dodatno uprostio kod, a bio bi i čitljiviji.

```
bool passwordLenghtIsValid;

if (_password.Length > 6)
{
    passwordLenghtIsValid = true;
}
else
{
    passwordLenghtIsValid = false;
}
```

Slika 2. *Kompleksniji način dodele vrednosti* ***boolean*** *promenljivama*

Na slici 2 prikazan je komplikovaniji način dodele vrednosti ***boolean*** promenljivama koji se može znatno uprostiti načinom dodele sa slike 3.

```
var passwordLenghtIsValid = _password.Length > 6;
```

Slika 3. *Uprošćena dodela vrednosti* ***boolean*** *promenljivama*

6.5. Rukovanje greškama

U slučaju kada dođe do određene predviđene greške u kodu potrebno je na adekvatan način obavestiti klijentsku klasu. Rukovanje vraćene poruke o eventualnoj grešci može biti zaboravljeno na klijentskoj klasi, te je potrebno baciti izuzetak. Osnovno pravilo za rukovanje izuzecima jeste da ih treba bacati što ranije, a hvatati što kasnije. Za neko specifično ponašanje treba dodati posebnu klasu koja nadleđuje klasu ***Exception.***

6.6. Ostala pravila o čistom kodu

Postoji još mnogo pravila koja dovode do čitko napisanog koda, kao što su: dopisivanje donje crte ispred privatnog polja, izbegavanje korišćenja regiona, brisanje neiskorišćenih ***using***-a, formatiranje koda, korišćenje ***var*** ključne reči, korišćenje principa „**tell**, **dont ask“**, izbegavanje hardkodiranja, umereno korišćenje ***Linq*** biblioteke.

7. ZAKLJUČAK

U radu su predstavljena osnovna načela, principi i šabloni koji prouzrukuju pisanje boljeg dizajna i samog koda prilikom rešavanja određenih programerskih problema. Današnje aplikacije dostižu veoma visok nivo kompleksnosti, dolazi do velikog broja promene klijentskih zahteva, te je jako bitno voditi računa o pisanju koda koji ima visok stepen održivosti.

8. LITERATURA

[1] Pablo’s SOLID Software Development, LosTechies.com
[2] Adaptive Code via C#: Agile coding with design patterns and SOLID principles, Gary McLean Hall
[3] The Liskov Substitution Principle, Robert C. Martin
[4] Agile Principles, Patterns and Practices in C#, Robert C. Martin
[5] Design Patterns - Elements of Reusable Object Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides

Kratka biografija:

Saša Momčilović rođen je 09.02.1994. godine u Zrenjaninu. Srednju tehničku školu „Mileva Marić“ u Titelu, smer ekonomski tehničar, završio je 2013. godine. Iste godine upisao je Fakultet tehničkih nauka u Novom Sadu. Ispunio je sve obaveze i položio je sve ispite predviđene studiskim programom.