UDC: 004.4:004.8 Info M: str. 24-30

PREDNOSTI I ZNAČAJ KORIŠĆENJA IMPLEMENTACIONIH PATERNA U RAZVOJU SOFTVERA
BENEFITS AND IMPORTANCE OF IMPLEMENTATION PATTERNS IN SOFTWARE DEVELOPMENT

Nada Sretović, Saša Lazarević

REZIME: Implementacioni paterni (uzori) su paterni koji se nalaze na najnižem nivou u hijerarhiji paterna i vezani su za samu implementaciju softvera. Oni se bave problemom kako na najbolji mogući način implementirati konstrukcije koje se u programerskoj praksi svakodnevno javljaju. U ovom radu izložene su prednosti i značaj upotrebe implementacionih paterna prilikom razvoja softvera. Objašnjeno je kako upotrebom ovih paterna utičemo na cenu održavanja softvera, na strukturu i preglednost koda.

KLJUČNE REČI: implementacioni paterni, razvoj softvera, mikroarhitektura, .NET, C#

ABSTRACT: Implementation patterns are patterns that are at the lowest level in the hierarchy of patterns and are related to the implementation of the software. Its deal with the problem what is the best possible way to implement the constructions which occurring daily in developer's practices. This paper presents software development using implementation patterns, and benefits and significance of their usage. It is explained how the use of these patterns affect the cost of software maintenance, the structure and layout of the code.

KEY WORDS: implementation patterns, software development, microarchitecture, .NET, C#

UVOD

Proces razvoj softvera može da se podeli na dva dela: razvoj i održavanje. Slikovito, razvoj softvera može da se uporedi sa izgradnjom zgrade. Temelj i prizemlje zgrade bi predstavljali početnu fazu razvoja softvera, a nadgradnja ostalih spratova je održavanje softvera. Ukoliko temelj i prizemlje nisu dobro i kvalitetno urađeni spratovi bi se teško i sporo izgrđivali i stalno bi bilo vraćanja na popravke temelja i prizemlja. Isto tako ukoliko se početana faza razvoja softvera loše uradi, kasnije održavanje biće još teže. Da bi održavanje bilo lakše mora se voditi računa na prvom mestu da se softver dobro isprojektuje ali takođe i o samoj implementaciji. Kod treba da bude strukturiran, čitljiv, pregledan, sažet, dosledan i samoopisujući.

Kent Bek (Kent Back) je rekao: „*Životni vek od sedamdeset godina sadrži oko dve milijarde sekundi. To nije dovoljno sekundi da gubim na posao na koji nisam ponosan!*“ [1].

Upotreba implementacionih paterna nam štedi vreme i energiju i omogućava da kod učinimo čitljivijim i razumljivim za druge. U ovmo radu predstavićemo implementacione paterne, i prednosti i značaj njihove upotrebe u svakodnevnom korišćenju tokom razvoja softvera. Postoje dve grupe ovih paterna, to su tehnološki nezavisi i tehnološki zavisni implementacioni paterni. Ovde će biti izloženi i detaljnije alalizirani tehnološki nezavisnim implementacionih paterna, jer su to paterni koji mogu da se koriste nezavisno od tehnologije u kojoj razvijamo softver.

Rad je podeljen u šest celina. U prvom delu su opisani osnovni pojmovi vezani za paterne. U drugom delu su predstavljeni implementacioni paterni kao i specifičnosti njihove upotrebe. U trećem delu su opisani tehnološki nezavisni implementacioni paterni i neki od ovih paterna su teorijski analizirani. U narednim delu su praktnično primenjeni paterni iz prethodnog dela (primeri su implementirani u programskom jeziku C#). Poslednje dve celine su zaključak i spisak korišćene literature.

1. POJAM PATERNA

U životu svakodnevno upotrebljavamo paterne susrećući se sa različitim situacijama i problemima. Ti problemi mogu da se ponavljaju ili da budu međusobno slični. Kada smo jednom prevazišli i rešili uspešno neki problem, svaki sledeći koji je bio sličan ili isti, mnogo smo brže rešavali vođeni prethodnim iskustvom.

Pojam paterna prvi je definisao arhitekta Kristofer Aleksandar (Christopher Alexander), koji kaže: *„Svaki patern opisuje problem koji se stalno ponavlja u našem okruženju i zatim opisuje suštinu rešenja problema tako što se to rešenje može upotrebiti milion puta, a da se dva puta ne ponovi na isti način“* [5].

Dakle, paterni su poreklom iz oblasti arhitekture, ali kasnije su primenjeni u različitim oblastima kao što su softversko inženjerstvo, muzika, ekonomija, itd.

Možemo da zaključimo da patern opisuje rešenje za klasu problema. Predstavlja opštu formulu (šablon), a ne konkretno rešenje. Patern je uređena trojka (problem, kontekst, rešenje) [2].

2. IMPLEMENTACIONI PATERNI

Implementacione paterne možemo da posmatramo na dva načina. Prvi način je da na implementacione paterne gledamo kao na navike koje kod čine čitljivijim. A drugi da se na njih gleda kao na način razmišljanja o tome šta drugima želimo da kažemo o svom kodu. Implementacioni paterni pružaju skup rešenja za zajedničke probleme u programiranju.

2.1. *Svrha implementacionih paterna*

Za većinu programa važe sledeća pravila:

– Programi se češće čitaju nego što se pišu.
– Mnogo više se investira u modifikovanje programa nego na razvoj programa u početku.
– Onaj ko čita program treba da shvati i koncept i detalje programa.

Zato je potrebno pisati program na način da bude svima razumljiv, a ne samo onome ko ga piše. Radeći sa paternima vremenom može se osetiti ograničenost ali korišćenje paterna štedi vreme i energiju i omogućava drugima da lakše čitaju i shvate kod.

Takođe upotreba implementacionih paterna a i ostalih ima i svoje ekonomske razloge. Kada se tek počelo sa razvojom softvera, cena softvera je bila podeljena na početni trošak i trošak održavanja.

Ukupna cena = cena razvoja + cena održavanja[1]

Vremenom kako su se softveri razvijali i postajali veći, cena održavanja softvera je postala veća od cene početnog razvoja. Održavanje koda danas obuhvata prvo razumevanje šta postojeći kod radi, izmenu koda, a onda i testiranje novog koda i na kraju uvođenje.

Trošak održavanja = trošak razumevanja + trošak izmene + trošak testiranja + trošak uvođenja[1]

Zato je potrebno više ulagati na početku razvoja softvera, gde treba pokušati predvideti buduće promene i kod prilagoditi tome. Normalno je da ne možemo predvideti sve promene, ali neke možemo i na taj naći ćemo smanjiti troškove održavanja.

2.2. *Klasifikacija implementacionih paterna*

Kao što je spomenuto u uvodom delu implementacione paterne možemo podeliti na:

– tehnološki zavisne (idiomi ili programsko-jezički paterni) i
– tehnološki nezavisne.

Implementacioni, tehnološki zavisni paterni su paterni koji zavise od tehnologije u kojoj se softver implementira. Oni predstavljaju rešenje problema vezano za konkretan programski jezik, što će reći da se razlikuju od jezika do jezika.

3. IMPLEMENTACIONI PATERNI – TEHNOLOŠKI NEZAVISNI

Implementacioni, tehnološki nezavisni paterni su paterni koji ne zavise od tehnologije u kojoj se softver implementira. Oni su uopšteni paterni i važe za sve objektno-orijentisane jezike. Oni mogu da budu vezani za klase, za stanja objekata, za ponašanje sistema, za metode itd.

3.1. *Paterni vezani za klase*

U objektno-orijentisanom programiranju osnovnu gradivnu jedinicu programa čini klasa. Klasa predstavlja opšti opis čitavog niza sličnih tipova ili objekata. U okviru klase definisani su podaci i funkcije. Klase treba da budu hijerarhijski organizovane na taj način da se krene od najprostijih, apstraktnih elemenata kako bi se došlo do kompleksnijih, specijalizovanih elemenata.

Klase su relatinvno skupi gradivni elementi programa. One treba da rade nešto značajno. Dobro projektovana klasa trebala bi da definiše samo jednu logičku celinu. Smanjenje broja klasa u programu predstavlja poboljšanje, ali samo dok preostale klase ne postanu prenatrpane.

Neki od paterna vezanih za klase su:

– Proširenje interfejsa i
– Klasa kao biblioteka.

3.1.1. *Proširenje interfejsa*

Problem: Kako proširiti interfejs?

Rešenje: Šta da radimo kada nam je potrebno da promenimo interfejs, ali ne možemo? Obično se ovo dešava kada želimo da dodamo operacije. Pošto će dodavanje operacija da naruši sve postojeće implementatore, ne možemo to da uradimo. Međutim, možemo da deklarišemo novi interfejs koji implementira postojeći i dodaje nove operacije. Klase koje žele nove funkcionalnosti, implementiraće prošireni interfejs, dok postojeći implementatori ostaju nesvesni postojanja novog interfejsa.

3.1.2. *Klasa kao biblioteka*

Problem: Gde staviti funkcionalnosti koje se ne uklapaju u nijednu klasu?

Rešenje: Kada nam je potrebna neka metoda a ne znamo gde da je smestimo jer se ne uklapa u nijednu klasu, to možemo da rešimo tako što ćemo kreirati klasu koja će da bude kao biblioteka. Metode u toj klasi su statičke. Na taj način omogućavamo objektima drugih klasa da im pristupe, bez kreiranja objekta klase u kojoj se metode nalaze. Ova klasa predstavlja čuvara funkcionalnosti. Naravno, nije dobro ni sve funkcionalnosti izdvajati na ovaj način, jer se tako gubi prednost objektno-orijentisanog programiranja.

3.2. *Paterni vezani za stanja*

Da bismo objasnili šta je stanje objekta poslužićemo se primerom automobila. Ako želimo da vozimo automobil potrebno je prvo da sipamo gorivo a zatim da upalimo automobil. Dužina puta koju možemo da pređemo zavisi od količine goriva koju smo sipali. Ako pokušamo da upalimo autmobil a da prethodno nismo sipali gorivo, automobil neće upaliti i nećemo moći da se vozimo.

Iz ovog primera možemo da zaključimo da na ponašanje objekta utiče njgova istorija, odnosno da je bitan redosled po kojem objekat radi. Razlog za ovakovo ponašanje, koje zavisi od vremena i događaja je da se objekat nalazi u nekom stanju. U našem primeru imamo osobinu da automobil ide na gorivo što predstavlja statičku osobinu i da u automobil možemo da sipamo različite količine goriva i to je dinamička osobina. Na osnovu ovog primera možemo da kažemo da stanje nekog objekta obuhvata sve nepromenljive osobine, zajedno sa trenutnim vrednostima promenljivih osobina.

Neki od paterna vezanih za stanja su:

– Stanje objekta sa stanjima i

– Konstante.

3.2.1. *Stanje objekta sa stanjima*

Problem: Kako predstaviti stanje objekta koje ima svoja stanja?

Rešenje: Stanje objekta koje ima svoja stanja najbolje je predstaviti kao novi objekat. Naziv nove klase biće naziv stanja, a polja će biti njegova stanja. Prethodni objekta će da ima polje koje je tipa nove klase. Na ovaj način kod će biti jasniji i pozivi metoda kojima se prosleđuje stanje biće čitljiviji i preciznije definisani**.**

3.2.2. *Konstante*

Problem: Na koji način čuvati stanja koja se ne menjaju?

Rešenje: Ponekad nam je na više mesta u programu potrebna ista vrednost koja je uvek konstantna, tj. ne menja se. Da ne bismo stalno ponavljali tu vrednost kroz kod, najbolji način jeste da je čuvamo kao konstantu. Imena konstanti se uobičajeno pišu velikim slovima da bi se razlikovale od ostalih promenljivih. Takođe naziv konstante nam omogućava do kroz njega izrazimo ono što predstavlja vrednost konstante. Na taj način naš kod ćemo učiniti jasnijim i drugima**.**

3.3. *Paterni vezani za ponašanje*

Program predstavlja niz instrukcija koje se izvršavaju jedna po jedna. Ponašanje sistema iskazuje se kroz njegove funkcionalnosti. Stoga funkcionalnosti treba prikazati kao niz koraka. Ovaj niz koraka koji čini tok programa može da se iskaže kao jedan glavni tok sa izuzecima, kao više alternativnih tokova gde je svaki podjednako važan ili kao kombinacija.

Neki od paterna vezanih za ponašanje su:

– Glavni tok i

– Poruke dekomponovanja.

3.3.1. *Glavni tok*

Problem: Kako izraziti tok programa?

Rešenje: Tok programa treba izraziti kroz glavni tok. Treba znati gde obrada počinje, a gde se završava. Pored glavnog toka mogu da postoje i razne odluke i izuzeci, ali je potrebno da program ima put koji prati. Nije da su izuzeci i neki posebni uslovi nevažni, nego je fokusiranje na izvršavanje glavnog toka programa mnogo vrednije, jer kroz glavni tok jasnije sagledavamo programske funkcionalnosti.

3.3.2. *Poruke dekomponovanja*

Problem: Kako duge i nepregledne metode učiniti preglednim?

Rešenje: Kada imamo duge, nepregledne metode koje su sastavljene iz mnogo delova, možemo da grupišemo povezane korake i da ih izmestimo u novu metodu koju zatim samo pozovemo. Novu metodu treba opisno imenovati kako bi čitaoci koda iz naziva mogli da naslute šta metoda radi a da ne moraju da se udubljuju u implementacione detalje. Poziv metode predstavlja poruku koja prenosi šta se u tom delu koda zbiva.

4. STUDIJSKI PRIMER

Na osnovu prethodne teorijske analize paterna u nastavku sledi praktična primena istih. Svi primeri su implementirani u programskom jeziku C#.

4.1. *Proširenje interfejsa*

Zahtev: Imamo interfejs Obuca koji ima metode za vraćanje veličine obuće i za vraćanje materijala:

```
public interface IObuca
{
   public int vratiVelicinu();
   public string vratiMaterijal();
}
```

Ovaj interfejs implementiraju klase Cipele, Patike i Papuce. Potrebno je da klasama Cipele i Patike dodamo još jednu metodu za vraćanje vodootpornosti.

Antipatern:

```
public interface IObuca
{
   public int vratiVelicinu();
   public string vratiMaterijal();
   public string vratiVodootpornost();
}
```

Patern: Ovo rešnje je loše jer smo primorani da promenimo sve implementatore interfejsa IObuca. U našem primeru morali bismo da promenimo i klasu Papuce, a klasa Papuce ne treba da ima metodu za stepen vodootpornosti. Način da ovo rešimo jeste da kreiramo novi interfejs, IZatvorenaObuca, koji će da implementira interfejs IObuca i imaće novu metodu za vraćanje stepena vodootpornosti. Klase Cipele i Patike implementiraće umesto interfejsa IObuca novi interfejs IZatvorenaObuca i na taj način ćemo ove dve klase da proširimo bez menjanja klase Papuce.

Primena paterna:

```
public interface IObuca
{
   public int VratiVelicinu();
   public string VratiMaterijal();
}
//Kreiramo nivi interfejs koji implementira interfejs
IObuca i dodajemo mu novu metodu
public interface IZatvorenaObuca: IObuca
{
   public string VratiVodootpornost();
}
```

Posledica: Na ovaj način možemo lako da proširimo interfejs a da to ne poremeti postojeće implementatore. Ali ako nam je potrebno da mnogo puta proširujemo interfejs, onda treba da preispitamo sam dizajn (konstrukciju) aplikacije.

4.2. *Klasa kao biblioteka*

Zahtev: Potrebno je da kreiramo klase UpravljanjeProizvodima i UpravljanjeProdajom. Klasa UpravljanjeProizvodima treba da ima metodu za izračunavanje i vraćanje cene sa popustom kojoj se kao ulazni parametri proseleđuju trenutna cena i visina popusta. Klasa UpravljanjeProdajom treba da ima metodu za izračunavanje i vraćanje procenta od prodaje kojoj se kao ulazni parametri prosleđuju ukupna dobit i procenat.

Antipatern:

```
class UpravljanjeProizvodima
{
   public double VratiCenuSaPopustom(double cena,
double procenat)
   {
      double novaCena = cena * (100 – procenat) /
100;
      return novaCena;
   }
}

public class UpravljanjeProdajom
{
   public double VratiProcenatOdProdaje(double
ukupnaDobit, double procenat)
   {
      double dobit = ukupnaDobit * procenat / 100;
      return dobit;
   }
}
```

Patern: Ako malo bolje pogledmo prethodni kod, možemo da zaključimo da su metoda VratiCenuSaPopustom i VratiProcenatOdProdaje iste. Implementaciju ovih metoda možemo da izdvojimo u zasebnu klasu PomocneMetode kako bismo izbegli dupliciranje koda i kako bi metodu mogli ponovo da koristimo ako to bude neophodno. Metoda u novoj klasi može da bude statička i na taj način možemo direktno da pristupimo metodi bez kreiranja objekata.

Primena paterna:

```
public sealed class PomocneMetode
{
   public static double VratiProcenat(double celina,
double deo)
   {
      return celina * deo / 100;
   }
}

class UpravljanjeProizvodima
{
   public double VratiCenuSaPopustom(double cena, do-
uble procenat)
   {
      procenat = 100 – procenat;
      return PomocneMetode.VratiProcenat(cena, 100 -
procenat);
   }
}

public class UpravljanjeProdajom
{
   public double VratiProcenatOdProdaje(double uku-
pnaDobit, double procenat)
   {
      return PomocneMetode.VratiProcenat(ukupnaDobit,
procenat);
   }
}
```

Posledica: Korišćenjem klasa kao biblioteka, ostale klase rasterećujemo sa funkcionalnostima koje nisu usko vezane za njih, a opet su im potrebne. Takođe, na ovaj način samanjujemo i dupliciranje koda. Naravno, nije dobro ni sve funkcionalnosti izdvajati na ovaj način, jer se tako gubi prednost objektno-orijentisanog programiranja.

4.3. *Stanje objekta sa stanjima*

Zahtev: Potrebno je da kreiramo klase Krug i Trougao. Klasa Krug je opisana pomoću poluprečnika i centra koji je opisan pomoću x i y koordinata. Klasa Trougao je opisana pomoću tačaka A, B i C koje su takođe predstavljene pomoću x i y koordinata.

Antipatern:

```
class Krug
{
   private double poluprecnik;
   private double centarPoX;
   private double centarPoY;
}


class Trougao
{
   private double temeAPoX;
   private double temeAPoY;
   private double temeBPoX;
   private double temeBPoY;
   private double temeCPoX;
   private double temeCPoY;
}
```

Patern: U prethodnom primeru možemo da primetimo da obe klase imaju stanja koja imaju svoja stanja. Centar kruga i temena A, B i C trougla opisana su pomoću x i y koordinata. Da bi kod postao jasniji možemo da kreiramo novu klasu koja će da predstavlja tačku koja ima x i y koordinate. I zatim ćemo centar kruga i temena trougla predstaviti pomoću nove klase.

Primena paterna:

```
class Tacka
{
   private double x;
   private double y;
}

class Krug
{
   private double poluprecnik;
   private Tacka centAR;
}

class Trougao
{
   private Tacka temeA;
   private Tacka temeB;
   private Tacka temeC;
}
```

Posledica: Deklarisanjem stanja koja imaju svoja stanja povećava se čitljivost koda i pozivi metoda koji koriste stanje su jasnije i preciznije definisani.

4.4. *Konstante*

Zahtev: Potrebno je da napišemo metode za izračunavanje obima i površine kruga kojima se kao parametar prosleđuje poluprečnik kruga.

Antipatern:

```
class Program
{
 public double izracunajObimKruga(double poluprecnik)
   {
      double obim = 2 * poluprecnik * 3.14;
      return obim;
   }

   public double izracunajPovrsinuKruga(double
poluprecnik)
   {
      double povrsina = 3.14 * poluprecnik *
poluprecnik;
      return povrsina;
   }
}
```

Patern: Prethodni kod je u redu što se funkcionalnsti tiče, ali ako malo bolje pogledamo primetićemo da se vrednost broja pi ponavlja u kodu. Znamo da je ta vrednost uvek ista i da se ne menja, stoga možemo da je definišemo kao konstantu koju samo posle koristimo gde je to potrebno.

Primena paterna:

```
class Program
{
   private const double PI = 3.14;

   public double izracunajObimKruga(double
poluprecnik)
   {
      double obim = 2 * poluprecnik * PI;
      return obim;
   }

   public double izracunajPovrsinuKruga(double
poluprecnik)
   {
      double povrsina = PI * poluprecnik *
poluprecnik;
      return povrsina;
   }
}
```

Posledica: Upotreba konstanti čini kod čitljivijim. Takođe omogućava nam da ukoliko se javi potreba za promenom vrednosti to učinimo na jednom mestu, umesto da kroz kod tražimo sve upotrebe što povećava šansu da se pogreši.

4.5. *Glavni tok*

Zahtev: Potrebno je da opišemo glavni tok jednostavnog softvera koji služi za rezervisanje karata za pozorišne predstave.

Antipatern:

Opis glavnog toka:

– Korisnik unosi zahtev za prikaz predstava.
– Softver učitava podatke o predstavama iz baze.
– Ukoliko softver ne može da učita podatke iz baze iz bilo kog razloga obaveštava korisnika o tome u suprotnom prikazuje korsniku podatke o predstavama.
– Korisnik unosi neophodne podatke da bi se izvršila rezervacija.
– Softver proverava unete podatke i ukoliko podaci nisu u redu obaveštava korisnika o tome.
– Ako su podaci u redu, softver izvršava rezervaciju.
– Ukoliko dođe do greške prilikom izvršavanja rezervacije prikazuje se poruka korisnku.
– Ukoliko se rezervacija obavi uspešno, korisnik dobija poruku o tome.

Patern: Prethodni primer predstavlja loš opis glavnog toka programa, jer su opisani i svi izuzeci koji se mogu desiti tokom izvršavanja programa zbog čega se teže sagledava osnovna funkcionalnost programa.

Primena paterna:

Galvni tok programa iz našeg primera trebao bi da izgelda ovako:

– Korisnik unosi zahtev za prikaz predstava.
– Softver učitava podatke o predstavama iz baze i prikazuje ih korisnku.
– Korisnik unosi neophodne podatke da bi se izvršila rezervacija.
– Na osnovu unetih podataka, softver izvršava rezervaciju.
– Obaveštava korisnika da je rezervacija izvršena uspešno.

Ovo svakako ne znači da se zanameruju izuzeci, u našem primeru provera podataka koje korisnik unosi, provera da li ima još slobdnih mesta, greške koje se dešavaju ukoliko nešto nije u radu sa bazom itd., ali glavni tok mora da bude jasno izražen. Svi ovi izuzeci zapravo omogućavaju da se osnovna funkcionalnost, rezervisanje karta, izvrši.

Posledica: Jasnim izražavanjem glavnog toka programa omogućavamo de se jasnije i lakše sagledaju funkcionalnosti programa.

4.6. *Poruke dekomponovanja*

Zahtev: Potrebno je da kreiramo klasu Karta koja predstavlja bioskopsku kartu i klasu ObradaKarata. Klasa Karta od polja ima šifru, cenu i tip popusta koji može da bude studentski od 25%, za penzionere od 30% i za decu od 50%. U klasi ObradaKarata treba kreirati metodu koja izračunava ukupan iznos računa, gde se kao parametar prosleđuje lista karata.

Antipatern:

```
class Karta
{
   private int id;
   private double cena;
   private string tipPopusta;
}

class ObradaKarata
{
   private double IzracunajUkupnanIznos(List<Karta>
listaKarata)
   {
      double ukupanIznos = 0;
      foreach (Karta karta in listaKarata)
      {
         if (!String.IsNullOrEmpty(karta.TipPopusta))
         {
            ukupanIznos = ukupanIznos + karta.Cena;
         }
         else
         {
            switch (karta.TipPopusta)
            {
               case”studentska”:
                  ukupanIznos = ukupanIznos + karta.
Cena * 75 / 100;
                  break;
               case”zaPenzionere”:
                  ukupanIznos = ukupanIznos + karta.
Cena * 70 / 100;
                  break;
               case”deca”:
                  ukupanIznos = ukupanIznos + karta.
Cena * 50 / 100;
                  break;
             }
         }
      }
      return ukupanIznos;
   }
}
```

Patern: Prethodni primer ispunjava zahteve ali kod možemo da učinimo jasnijim i preglednijim ukoliko deo koda za izračunavanje popusta izdvojimo u novu metodu. Zatim samo pozovemo novu metodu i prosledimo kartu za koju računamo popust. Takođe u okviru nove metode možemo da izdvojimo kod koji direktno računa popust u zasebnu metodu kao formulu za računanje popusta. I naravno posle tu metodu pozivmo i prosleđujemo odgovrajuće parametre u zavisnosti od tipa popusta koji računamo.

Primena paterna:

```
class Karta
{
   private int id;
   private double cena;
   private string tipPopusta;
}

class ObradaKarata
{
   private double IzracunajUkupnanIznos(List<Karta>
listaKarata)
   {
      double ukupanIznos = 0;
      foreach (Karta karta in listaKarata)
      {
         if (!String.IsNullOrEmpty(karta.TipPopusta))
         {
            ukupanIznos = ukupanIznos + karta.Cena;
         }
         else
         {
            ukupanIznos = ukupanIznos +
IzracunajPopustKarte(karta);
         }
      }
      return ukupanIznos;
   }

   private double IzracunajPopustKarte(Karta karta)
   {
      double cenaSaPopustom = 0;
      switch (karta.TipPopusta)
      {
         case”studentska”:
            cenaSaPopustom = IzracunajPopust(karta.
Cena, 25);
            break;
         case”zaPenzionere”:
            cenaSaPopustom = IzracunajPopust(karta.
Cena, 30);
            break;
         case”deca”:
            cenaSaPopustom = IzracunajPopust(karta.
Cena, 50);
            break;
      }
      return cenaSaPopustom;
   }
```

```
      private double IzracunajPopust(double cena, int
popust)
      {
         double cenaSaPopustom = cena * (100 - popust)
* 100;
         return cenaSaPopustom;
      }
   }
```

Posledica: Korišćenje metoda kao poruka dekomponovanja povećava se preglednost koda i rasterećuju se metode sastavljene iz mnogo delova.

5. ZAKLJUČAK

U ovom radu izučavane su prednosti i značaj upotrebe implementacionih paterna prilikom razvoja softvera. Na osnovu teorijske analize i praktične primene implementacionih paterna u ovom radu može se izvesti sledeći zaključak.

Programiranje bi bilo efektivnije kada bi programeri manje vremena trošili na probleme sa kojim se svakodnevno susreću i kada bi im ostajalo više vremena za rešavanje jedinstvenih problema za koje je potrebna kreativnost i velika energija. Takođe, biti dobar programer ne znači napisati komplikovan kod koji je nerazumljiv ili teško razumljiv drugima. Upotreba implementacionih paterna omogućava da se kod učini čitljivim, preglednim, strukturiranim i smoopisujućim, zatim štedi vreme za rešavanje novih i nepoznatih problema na koje se nailazi tokom razvoja softvera i smanjuje dupliciranje koda.Takođe, implementacioni paterni u velikoj meri olakšavaju kasnije održavanje softvera.

6. LITERATURA

[1] Kent Beck: *Implementation Patterns*, Addison Wesley, 2008.

[2] Siniša Vlajić: *Softverski paterni* (radni materijal sa predavanja), FON, Beograd, 2011.

[3] Saša D. Lazarević: *Konstrukcija softvera: Priručnik za programere - okruženje, dizajn, arhitektura*, FON, Beograd, 2008.

[4] Imed Hammouda & Jakub Rudzki - *Pattern Calssification*, Software Systems Institute 2004. (http://www.cs.tut.fi/~kk/webstuff/PatternClassificationKalvot.pdf)

[5] Slobodan Mirković, Saša D. Lazarević: *Test driven development uzori i refaktorisanje testnog koda*, časopis *info M 45/2013*, Beograd, 2013.

[6] *Pattern language,* (http://www.patternlanguage.com/leveltwo/caframe.htm?/leveltwo/../bios/designpatterns.htm), decembra 2015. god. pristupano

[7] Appleton Brad: Patterns and Software: Essential Concepts and Terminology, (http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html), decembra 2015. god. pristupano.

Nada Sretović
Kontakt: nadasretovic@yahoo.com
Oblasti interesovanja: internet tehnologije i mobilno računarstvo

Dr Saša D. Lazarević, Univerzitet u Beogradu – Fakultet organizacionih nauka.
Kontakt: sasa.lazarevic@fon.rs
Oblasti interesovanja: softversko inženjerstvo, informacioni sistemi, baze podataka, sistemi za upravljanje dokumentacijom, .NET platform