6 - ...Obrasci za projektovanje

Saša Malkov

Odlomak iz knjige u pripremi...

6.1 Ekvivalentnost problema

Programeri i projektanti softvera se u svakodnevnom radu stalno iznova susreću sa različitim vidovima *ponavljanja problema*. Ako se neki problem ponovi u nekom projektu na različitim mestima u doslovno istom obliku, uobičajeno je da se rešenje problema izdvoji na odgovarajući način i zatim tako izdvojeno upotrebljava gde god je to potrebno i moguće. Najjednostavniji način izdvajanja koda koji rešava neki problem je pravljenje potprograma. U složenijim slučajevima nisu dovoljni pojedinačni potprogrami, već je neophodno da se prave moduli, klase ili biblioteke. Dobro izdvojen programski kod može da se upotrebljava za rešavanje ponovljenih problema ne samo u okviru jednog projekta, već i mnogo šire. Uobičajeno je da se iste biblioteke upotrebljavaju u različitim projektima u nekom razvojnom okruženju. Mnoge dobre biblioteke su prevazišle granice okruženja u kojima su nastale i koriste se širom sveta u mnogim projektima.

Ako se problem ne ponavlja u potpuno istom obliku, već postoje određene varijacije, onda nije uvek moguća primena nepromenljivih biblioteka. Zbog toga dobre biblioteke moraju da omoguće proširivanje u skladu sa potrebama korisnika. Da bi na prihvatljiv način mogle da se obuhvate što šire potencijalne varijacije, takve biblioteke se obično zasnivaju na primeni *polimorfizma*. U savremenom razvoju softvera se primenjuju svi vidovi polimorfizma – hijerarhijski polimorfizam je

najzastupljeniji, ali mnogi programski jezici podržavaju i parametarski, a sve veći broj jezika i implicitni polimorfizam[9]. Primena polimorfizma mora dobro da se odmeri. Ako se pretera u apstraktnosti (tj. širini) rešenja, lako može da dođe do suvišnog *naduvavanja* biblioteke, a time i povećane složenosti i smanjene efikasnosti pri njenom korišćenju. Sa druge strane, ako biblioteka nije dovoljno apstraktna, onda se potencijalno sužava prostor za njenu primenu.

Pri posmatranju i analiziranju nekog problema neophodno je da se prepoznaju značajni elementi koji su karakteristični za taj problem. Elemente nekog problema čine različiti entiteti koji figurišu u problemu, kao i različite vrste odnosa koji postoje među tim entitetima. Elementi problema obuhvataju sve zahteve i preduslove koji moraju da se uvaže da bi problem mogao da se rešava.

Naravno, jedan isti problem može da se posmatra na više različitih načina, pa će zbog toga i uočeni elementi problema biti različiti u različitim okolnostima. Mnogi aspekti posmatranja utiču na uočavanje elemenata. Ako se pri prosmatranju problema najviše pažnje posvećuje nekim njegovim opštim karakteristikama, onda kažemo da se radi o uopštenom posmatranju, a ako se posmatraju detalji konkretnog slučaja, onda se radi o specifičnom posmatranju.

Pri posmatranju problema, aspekti problema se modeliraju različitim sredstvima, da bi se poboljšalo njihovo razumevanje. Model problema može da bude veoma blizak samom problemu, kada su pojmovi koji se koriste u modelu veoma bliski pojmovima koji se koriste u samom opisu problemu. Suprotno od toga, ako se pojmovi modela ne mogu bijektivno preslikati u pojmove problema, onda model nije sasvim blizak posmatranom problemu i predstavlja nekakvu njegovu apstrakciju.

Od svih karakteristika posmatranja problema ovde nas najviše zanima upravo nivo apstraktnosti posmatranja. Apstraktnost posmatranja je proporcionalna stepenu uopštavanja i apstraktnosti modela. Što je stepen uopštavanja viši, to je viši i nivo apstraktnosti posmatranja. Takođe, ako je viši nivo apstraktnosti posmatranja, onda je viši i nivo apstraktnosti modela.

Za probleme kažemo da su *međusobno slični u odnosu na neke njihove modele* ako možemo da napravimo preslikavanje modela jednog problema u model drugog problema. Ako može da se uspostavi bijektivno preslikavanje između modela dvaju problema, onda svaki od tih modela jednako dobro modelira oba problema. U takvom slučaju, kada za dva problema praktično imamo jedan zajednički model, kažemo da su ti problemi *međusobno ekvivalentni u odnosu na taj model*.

Primetimo da za bilo koja dva problema možemo da pronađemo model u odnosu na koji su ekvivalentni. Potrebno je samo da dovoljno podignemo nivo apstrakcije

[9] Polimorfizmu i posebno parametarskom polimorfizmu ćemo posvetiti više pažnje u poglavlju *9 - ...Parametarski polimorfizam*, na strani 175.

pri posmatranju problema. Na primer, skoro svaki problem možemo da „rešimo“ sledećim algoritmom:

- prikupimo potrebne resurse;
- izvršimo izračunavanje i potrebne poslove;
- napravimo izveštaje.

Korist od takvog uopštenog „rešavanja“ problema je prilično ograničena. Da bismo od ekvivalentnosti problema u odnosu na neki model mogli da imamo neke praktične koristi, neophodno je da nivo apstrakcije modela bude u nekim razumnim granicama.

Ako su neka dva problema međusobno ekvivalentna na sasvim niskom nivou apstrakcije, onda se radi o *ponovljenim problemima*, kod kojih eventualno postoje određene manje varijacije. Takvi problemi se rešavaju na isti ili vrlo sličan način, uz moguću primenu postojećih ili pravljenje novih biblioteka. Za ovakve probleme ćemo reći da su *implementaciono ekvivalentni*. Odgovarajući nivo apstrakcije ne donosi novi kvalitet u odnosu na posmatranje samih problema, pa kažemo da je *suviše nizak*.

Drugi ekstremni slučaj predstavljaju problemi koji su međusobno ekvivalentni tek na nekom veoma visokom nivou apstrakcije. Za probleme čiji su modeli ekvivalentni tek na nekom ekstremno visokom nivou apstrakcije možemo da kažemo da *praktično nisu ekvivalentni* ili da su samo *trivijalno ekvivalentni*. Takva ekvivalentnost nam nije od posebne koristi pri rešavanju problema, pa nećemo detaljnije da je razmatramo.

U ovom poglavlju ćemo najviše pažnje da posvetimo trećoj vrsti ekvivalentnih problema – onima koji su ekvivalentni pri nekom *umereno* apstraktnom posmatranju. Probleme koji su netrivijalno ekvivalentni, a nisu implementaciono ekvivalentni nazivamo *konceptualno ekvivalentnim problemima*.

6.2 Pojam obrasca za projektovanje

Ako imamo konceptualno ekvivalentne probleme, tada oni nisu dovoljno slični da bismo mogli da ih implementiramo istim programskim kodom, ali među njima ipak ima dovoljno sličnosti da možemo da uočimo apstraktan model u odnosu na koji su ekvivalentni. Ključno pitanje za razvijaoce softvera je: da li možemo da uočene sličnosti na neki način prevedemo u uopšteno zajedničko rešenje, čak i kada su sličnosti ustanovljene na relativno visokom nivou apstraktnosti? Odgovor je, naravno, negde između – možemo da napravimo nešto poput *zajedničkog rešenja*, ali to ne može da bude zajednički programski kod. Kao što je i ekvivalentnost problema konceptualna, tako i zajedničko rešenje najčešće može da bude najviše konceptualno, a ne i implementaciono.

Ipak, to ne predstavlja veliko ograničenje, zato što i konceptualno rešenje može da nam bude od koristi. Ako neki problem rešimo konceptualno, onda takvo rešenje možemo ponovo primeniti pri rešavanju ekvivalentnih problema. Naravno, moraćemo ponovo da pišemo programski kod (celog rešenja ili dela rešenja), moraćemo da ponovo testiramo napisani kod i tražimo eventualne greške u kodiranju, ali ćemo uštedeti vreme tako što nećemo morati da ponovo analiziramo sve detalje problema ili da pokušavamo da primenimo neka potencijalno neodgovarajuća rešenja. U takvim slučajevima najčešće nećemo morati ni da od početka smišljamo algoritam i načine njegove implementacije.

Da bi jedno konceptualno rešenje bilo višestruko *konceptualno* primenjivo[10], ono mora da zadovolji neke uslove:

- mora da bude jasno prepoznato koji apstraktan problem se rešava;
- moraju da se uoče i istaknu svi preduslovi za uspešnu primenu rešenja;
- rešenje mora da bude dovoljno apstraktno da može da se primeni;
- rešenje mora da bude dovoljno konkretno da može da se razume i implementira;
- poželjno je da budu uočeni i naglašeni važni aspekti implementacije rešenja i
- moraju da budu poznate i dokumentovane posledice primene rešenja.

Ako jedno konceptualno rešenje, osim što ima sve navedene karakteristike, odgovara nekoj značajnoj klasi problema, onda se takvo rešenje, zajedno sa pripadajućom dokumentacijom, naziva *obrazac za projektovanje*[11].

Svaki obrazac za projektovanje ima četiri osnovna elementa:

- naziv obrasca;
- problem koji se obrascem rešava;
- rešenje problema i
- posledice rešenja.

Naziv obrasca za projektovanje se bira tako da, u svega nekoliko reči, što bolje opiše problem, njegova rešenja i posledice. Davanjem naziva obrascima se povećava rečnik projektovanja i omogućava raspoznavanje obrasca u komunikaciji. Olakšava se razmena mišljenja o obrascima i diskutovanje o konceptualnom rešenju, kao i

[10] Tj. primenjivo u vidu „ponovljene primene istog koncepta“.

[11] U srpskom jeziku su u upotrebi i termini „uzorak“ i „šablon“. Termin na engleskom jeziku je *design pattern*.

pisanje i čitanje projektne dokumentacije. Sam postupak projektovanja se podiže na viši nivo apstrakcije. Bez imenovanja obrazaca ne bi bilo moguće napraviti kataloge ili rečnike obrazaca.

Svakom obrascu odgovara neka vrsta problema koji njegovom primenom može da se reši. Detaljan ali istovremeno i dovoljno apstraktan opis problema predstavlja osnovno sredstvo za prepoznavanje slučajeva u kojima neki obrazac može da se primeni. Opis problema može da obuhvata i konkretne primere, kao i opise elemenata problema (pojmova, odnosa,...) i odgovarajuće dijagrame. Može da obuhvata i spisak uslova koje je potrebno ispuniti da bi obrazac mogao da se uspešno primeni.

Rešenje problema predstavlja detaljan apstraktan opis elemenata koji čine projekat (dizajn) rešenja, njihove odgovornosti i međusobne odnose, kao i opis i tok saradnje među elementima. Rešenje ne sme da bude ograničeno na opisivanje određenog konkretnog projekta ili implementacije, zato što obrazac treba da predstavlja uputstvo koje može da se primeni u mnogim različitim slučajevima.

Posledice rešenja obuhvataju različite rezultate i ocene primene obrasca. One mogu biti veoma značajne pri razmatranju pogodnosti primene nekog rešenja i odabiranju jednog od više mogućih načina rešavanja nekog problema. Sagledavanjem posledica omogućava se vaganje između prednosti i nedostataka nekog rešenja.

Obrasci za projektovanje su najpre prepoznati u urbanizmu i arhitekturi [Alex1997]. Kasnije se o obrascima počelo razmišljati i u drugim domenima. Značaj obrazaca u razvoju softvera je uočen početkom 1990-ih godina. Šira stručna javnost je sa obrascima u razvoju softvera upoznata 1995. godine, kada je objavljena knjiga „Obrasci za projektovanje – Elementi više puta upotrebljivog objektno-orijentisanog softvera“ (engl. *Design Patterns – Elements of Reusable Object-Oriented Software*) [Gamma1995]. „Obrasci“ se odlikuju izuzetno visokim kvalitetom sadržaja i izlaganja, zbog čega su postali (i ostali) deo obavezne *programerske lektire*.

6.3 Primer – Obrazac *Unikat*

Obrazac *Unikat* (engl. *Singleton*) spada u najjednostavnije obrasce, ali se relativno često primenjuje. I pored jednostavnosti on dobro ilustruje koncept obrazaca za projektovanje, zbog čega je pogodan da nam posluži kao prvi primer. Namena obrasca Unikat je obezbeđivanje da neka klasa može da ima najviše jedan primerak.

U nekim slučajevima je potrebno da se u programu napravi i upotrebljava tačno jedan objekat neke klase. Postoji mnogo primera, a među najčešće spadaju različiti vidovi upravljačkih ili kontrolnih objekata i različite vrste registara ili kataloga objekata. Na primer, ako bismo pravili program za crtanje koji omogućava dinamičko dodavanje umetaka (engl. *plug-in*), za upravljanje umecima bismo mogli

da napravimo klasu `UpravljačUmecima`, koja bi smela da ima tačno jednu instancu. Slično, ako bismo želeli da u program ugradimo interpretator nekakvih skriptova, imalo bi smisla da pamtimo prethodno parsirane skriptove radi podizanja efikasnosti, pri čemu bi implementacija takvog kataloga prevoda takođe morala da ima tačno jednu instancu.

U najjednostavnijim slučajevima je dovoljno da se definiše jedan globalni objekat i da se koristi u programu kao jedinstven primerak klase. Međutim, takav pristup ne rešava mnoge potencijalne probleme. Jedan od najvažnijih je da programer i dalje može da napravi više primeraka klase, što može da dovede u pitanje funkcionalanost ili performanse sistema. Drugi ozbiljan problem je što ne postoji mogućnost jednostavnog upravljanja redosledom pravljenja takvih „jedinstvenih" primeraka različitih, potencijalno međuzavisnih klasa. Promena redosleda prevođenja ili povezivanja delova programa može da ima uticaja na redosled pravljenja takvih objekata, što u slučaju složenih međuzavisnosti može da bude veoma problematično.

Osnovni zahtevi koji se postavljaju pred rešenje su:

- pravljenje jedinstvenog objekta;
- odloženo pravljenje objekta do prve upotrebe (tzv. lenjo pravljenje);
- uništavanje objekta pri završetku rada programa i
- inicijalizacija bezbedna po niti.

Osnovna ideja rešenja obrasca Unikat je da se staranje o jedinstvenom primerku klase prepusti samoj klasi. U programskom jeziku C++ to može da se ostvari pisanjem statičkog metoda `Primerak()`, koji vraća referencu na jedinstveni primerak klase – *unikat*, i istovremeno sakrivanje svih konstruktora tako da ne postoji mogućnost da se napravi drugi objekat te klase:

```
class PrimerUnikata
{
public:
    static PrimerUnikata& Primerak()
    {
        static PrimerUnikata primerak {};
        return primerak;
    }

    // Metodi koji čine interfejs klase
    ...

private:
    PrimerUnikata()
    {...}

    PrimerUnikata( const PrimerUnikata& ) = delete;
    PrimerUnikata& operator=( const PrimerUnikata& ) = delete;
```

```
};
```

U programskom jeziku C++ se statički objekti po pitanju pravljenja ne razlikuju značajno od globalnih automatskih promenljivih, pa ni za njih nije jednostavno predvideti redosled pravljenja. Lokalne statičke promenljive (kao promenljiva `primerak` u metodu `PrimerUnikata::Primerak`) se prave, tj. inicijalizuju, tek pri prvom izvršavanju bloka u kome su definisane, pa navedeno rešenje omogućava lenjo pravljenje unikata. Uz to, ono je bezbedno po niti.

Svi konstruktori unikatne klase moraju da budu privatni, da bi pravljenje objekata klase bilo moguće samo u okviru njene implementacije, konkretno u okviru implementacije metoda `Primerak`. U primeru je naveden konstruktor bez argumenata, ali u slučaju potrebe umesto njega može da se napravi neki drugi konstruktor sa argumentima.

Konstruktor kopije mora eksplicitno da se zabrani (tj. obriše), da bi se sprečilo da neko namerno ili slučajno napravi drugi objekat kopiranjem dobijenog. Radi kompletnosti je uobičajeno da se isto uradi i sa operatorom dodeljivanja, iako do njegove primene ne bi trebalo da može da dođe ako imamo najviše jedan objekat[12].

Na svakom mestu u programu gde je potrebno da se koristi unikatni objekat, njemu mora da se pristupa putem metoda `Primerak`, na primer:

```
... PrimerUnikata::Primerak().metod(...) ...
```

Potencijalan problem sa navednim rešenjem može da nastupi ako implementacije više različitih unikata koriste jedna drugu. Redosled pravljenja će biti ispravan, zato što će se objekti unikata praviti onda kada se po prvi put koriste. Međutim, problem je u tome što programer ne može da utiče na redosled deinicijalizovanja i brisanja statičkih objekata, pa može da se dogodi da primenjeni redosled[13] proizvede određene probleme, poput deinicijalizovanja objekata koji se još koriste. U takvom slučaju može da se primeni dinamička alokacija[14]:

```
static PrimerUnikata& Primerak()
{
```

[12] Navedena sintaksa je u upotrebi od verzije C++11. U starijim verzijama su se ovi metodi deklarisali kao privatni a nisu se obezbeđivale njihove implementacije, pa bi eventualni pokušaj upotrebe dovodio do greške u fazi prevođenja ili povezivanja.

[13] Redosled deinicijalizovanja zavisi od mnogo faktora u fazi prevođenja i izgradnje koda. Programer ima vrlo ograničen uticaj na ovaj redosled, a i to zavisno od konkretne implementacije alata za prevođenje.

[14] Primetimo da ovde nema mnogo smisla upotreblavati pametne pokazivače, zato što bi uz njihovu primenu rezultat bio po svemu isti kao u prvom rešenju, bez dinamičke alokacije.

```
        static PrimerUnikata* primerak = new PrimerUnikata {};
        return *primerak;
    }
```

Slabost ovog rešenja je da ono ne omogućava automatsko brisanje unikata. Dinamički alociran unikat mora da se obriše eksplicitno ili nikada neće biti obrisan, pa će doći do curenja memorije. Međutim, eksplicitno brisanje unikata takođe može da dovede do neispravnosti na sličan način kao i prethodno rešenje sa automatskim brisanjem. U takvim slučajevima se unikati ponekad namerno ne brišu. Iako zbog toga dolazi do *curenja* memorije, a potencijalno i drugih resursa, u ovom kontekstu to nekada može da bude prihvatljivo. Unikati se prave samo po jedanput i postoje tokom čitavog toka izvršavanja programa. Nakon završetka rada programa operativni sistem će osloboditi svu lokalnu memoriju procesa, pa i onu koja je bila zauzeta pravljenjem dinamičkih objekata koji nisu obrisani. Ipak, ako unikat alocira i koristi globalnu memoriju ili neke druge globalne resurse operativnog sistema (otvorene datoteke, mrežne resurse, veze sa bazama podataka i drugim servisima i sl.), koji se ne oslobađaju automatski pri završetku programa, onda je njihovo eksplicitno brisanje neophodno.

6.4 Primer – Obrazac *Sastav*

Pre razmatranja obrasca *Sastav*, najpre ćemo da pretpostavimo da imamo napisanu jednostavnu hijerarhiju ravnih likova, čiju osnovu predstavlja klasa `Lik`:

- svaka klasa hijerarhije odgovara nekom liku u ravni;
- svaki lik ima položaj, implementiran u klasi `Lik`;
- ne obraćamo posebnu pažnju na orijentaciju likova – pretpostavljamo da su stranice kvadrata i pravougaonika paralelne koordinatnim osama i
- jedini važan metod izračunava površinu: `double Povrsina() const`.

Metod `Povrsina()` izračunava površinu lika. Implementiran je na način koji je uobičajen za hijerarhije klasa, sa potpuno virtualnom (tj. apstraktnom) deklaracijom u baznoj klasi `Lik` i sa odgovarajućim implementacijama u konkretnim klasama hijerarhije.

```
#include <iostream>
#include <vector>
#include <math.h>

using namespace std;

//-----------------------------------------------------------------
class Tacka
{
```

```
public:
    Tacka( double x, double y )
        : _X(x), _Y(y)
        {}

private:
    double _X;
    double _Y;

    friend ostream& operator<<( ostream&, const Tacka& );
    friend istream& operator>>( istream&, Tacka& );
};

ostream& operator<<( ostream& ostr, const Tacka& t )
{
    ostr << "(" << t._X << "," << t._Y << ")";
    return ostr;
}

istream& operator>>( istream& istr, Tacka& a )
{
    char c1,c2,c3;
    istr >> c1 >> a._X >> c2 >> a._Y >> c3;
    if( c1!='(' || c2!=',' || c3!=')' )
        istr.setstate( ios::failbit );
    return istr;
}

//-------------------------------------------------------------
class Lik
{
public:
    Lik( double x, double y )
        : _Polozaj(x,y)
        {}

    virtual ~Lik()
        {}

    const Tacka& Polozaj() const
        { return _Polozaj; }

    Tacka& Polozaj()
        { return _Polozaj; }

    virtual double Povrsina() const = 0;

private:
    Tacka _Polozaj;
};

ostream& operator<<( ostream& ostr, const Lik& l )
{
    ostr << l.Polozaj() << " P=" << l.Povrsina();
```

```
    return ostr;
}

//-----------------------------------------------------------------
class Pravougaonik : public Lik
{
public:
    Pravougaonik( double x, double y, double s, double v )
        : Lik(x,y), _Sirina(s), _Visina(v)
        {}

    double Sirina() const
        { return _Sirina; }

    double Visina() const
        { return _Visina; }

    double Povrsina() const
        { return Sirina() * Visina(); }

private:
    double _Sirina;
    double _Visina;
};

//-----------------------------------------------------------------
class Kvadrat : public Pravougaonik
{
public:
    Kvadrat( int x, int y, int a )
        : Pravougaonik( x, y, a, a )
        {}
};

//-----------------------------------------------------------------
class Krug : public Lik
{
public:
    Krug( double x, double y, double r )
        : Lik(x,y), _R(r)
        {}

    double R() const
        { return _R; }

    double Povrsina() const
        { return R() * R() * M_PI; }

private:
    double _R;
};

//-----------------------------------------------------------------
int main()
{
```

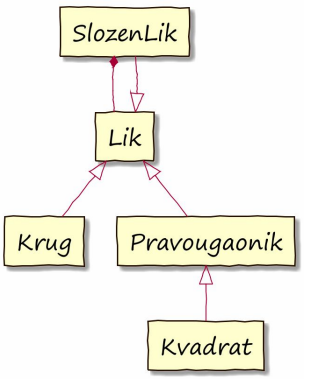
```
    vector<Lik*> likovi;
    likovi.push_back( new Pravougaonik(1,2,3,4) );
    likovi.push_back( new Kvadrat(5,6,7) );
    likovi.push_back( new Krug(8,9,10) );
    for( Lik* lik: likovi ){
        cout << *lik << endl;
        delete lik;
    }

    return 0;
}
```

Pretpostavimo sada da je za našu aplikaciju neophodno da omogućimo da se sa skupom likova radi kao da se radi o *jednom složenom liku*. Takav složen lik ima dvojaku prirodu – sa jedne strane on bi trebalo da bude u hijerarhiji likova, zato što složen lik *jeste* lik, a sa druge strane bi trebalo da predstavlja kolekciju jednostavnijih likova. Sličan problem se ponavlja veoma često u svakodnevnoj praksi. Na primer, ako modeliramo računske izraze, *suma* nekog niza podizraza je složen izraz i odnosi se prema hijerarhiji operacija i izraza na sličan način kao što se složen lik odnosi prema likovima.

Rešenje ovog tipa problema predstavlja obrazac *Sastav* (engl. *Composite*). Ovaj obrazac opisuje kako se prave elementi hijerarhije koji predstavljaju kolekcije drugih elemenata iste hijerarhije. Spada u strukturne obrasce. Ukratko ćemo predstaviti ideju, na uopšten način, nezavisno od programskog jezika:

- *sastav* se pravi kao klasa hijerarhije, tj. kao naslednik bazne klase hijerarhije;
- ima privatnu kolekciju objekata iste hijerarhije i javne metode za osnovne operacije sa tom kolekcijom, kao što je dodavanje novih ili brisanje postojećih elemenata;
- implementiraju se svi potrebni metodi hijerarhije.



Slika 19 – Obrazac *Sastav*, dijagram klasa

U našem slučaju, napravićemo klasu `SlozenLik`, kolekciju objekata `_Likovi` ćemo da predstavimo kao `vector<SlozenLik*>`, a za dodavanje novih elemenata ćemo da napišemo metod `void Dodaj( Lik* l )`:

```
class SlozenLik : public Lik
{
public:
    SlozenLik( double x, double y )
        : Lik(x,y)
        {}

    ~SlozenLik()
    {
        for( auto lik: _Likovi )
            delete lik;
    }

    void Dodaj( Lik* l )
        { _Likovi.push_back(l); }

    double Povrsina() const
    {
        double p = 0;
        for( auto lik: _Likovi )
            p += lik->Povrsina();
        return p;
    }

private:
    vector<Lik*> _Likovi;
};
```

Implementacija metoda `Povrsina` je napisana tako da se ne proverava da li se neki od likova možda preklapaju. U ovom kontekstu je prihvatljivo da

pretpostavimo da nema preklapanja, zato što nam je cilj da predstavimo obrasce, a ne da se bavimo algoritmima za izračunavanje preseka likova.

Primetimo da ova klasa nije dovršena, zato što nismo napisali operator dodeljivanja i konstruktor kopije, a oni su *neophodni* zbog načina na koji se čuvaju i brišu elementi kolekcije likova. Pisanje ovih metoda nije sasvim jednostavno. Problem je u tome što iz ugla složenog lika mi ne možemo da znamo kojim klasama pripadaju pojedinačni elementi, pa stoga ne možemo ni da ih iskopiramo. Ideja rešavanja ovog problema počiva na tome da *sami elementi* složenog lika *znaju* kojim klasama pripadaju. To *znanje* se ispoljava kroz ispravan odabir metoda koji se pozivaju pri dinamičkom vezivanju metoda. Znači, umesto da se složen lik brine o tome kojoj klasi pripada koji lik, on može da poruči svakom pojedinačnom liku da napravi svoju kopiju i da očekuje da će primenom dinamičkog vezivanja metoda biti upotrebljeni ispravni metodi i napravljene ispravne kopije elemenata kolekcije.

Prethodnom uopštenom opisu koncepta rešavanja dodajemo još neke stavke koje se odnose na slučaj upotrebe programskog jezika C++:

- u baznoj klasi hijerarhije se deklariše apstraktan metod koji pravi kopiju objekta:
  `virtual Lik* Kopija() const = 0;`
- u svim konkretnim klasama hijerarhije ovaj metod se implementira na odgovarajući način, ili pozivanjem konstruktora kopije ili pozivanjem nekog drugog konstruktora;
- u klasi koja predstavlja *sastav* se konstruktor kopije i operator dodeljivanja implementiraju tako da se elementi kolekcije kopiraju korišćenjem metoda `Kopija()`.

U našem slučaju to se implementira ovako:

```
class Lik { ...
public:
    virtual Lik* Kopija() const = 0;
};

class Pravougaonik : public Lik { ...
public:
    Lik* Kopija() const
        { return new Pravougaonik(*this); }
};

class Kvadrat : public Pravougaonik { ...
public:
    Lik* Kopija() const
        { return new Kvadrat(*this); }
};
```

```
class Krug : public Lik { ...
public:
    Lik* Kopija() const
        { return new Krug(*this); }
};

class SlozenLik : public Lik { ...
public:
    SlozenLik( const SlozenLik& sl )
        : Lik(sl)
        { init( sl ); }

    ~SlozenLik()
        { deinit(); }

    SlozenLik& operator=( const SlozenLik& sl )
    {
        if( &sl != this ){
            deinit();
            Polozaj() = sl.Polozaj();
            init(sl);
        }
        return *this;
    }

    Lik* Kopija() const
        { return new SlozenLik(*this); }

private:
    void deinit()
    {
        for( auto lik: _Likovi )
            delete lik;
        _Likovi.clear();
    }

    void init( const SlozenLik& sl )
    {
        for( auto lik: sl._Likovi )
            Dodaj( lik->Kopija() );
    }
};
```

Glavni program, koji ilustruje upotrebu složenih likova, može da izgleda ovako:

```
int main()
{
    SlozenLik sl(0,0);
    sl.Dodaj( new Pravougaonik(1,2,3,4) );
    sl.Dodaj( new Kvadrat(5,6,7) );
    sl.Dodaj( new Krug(8,9,10) );
    SlozenLik* sl2 = new SlozenLik(1,1);
    sl2->Dodaj( new Krug(2,3,4));
```

```
    sl2->Dodaj( new Kvadrat(2,4,5));
    sl.Dodaj(sl2);
    cout << sl << endl;
    return 0;
}
```

Detaljan opis obrasca *Sastav*, uz razmatranje preduslova za upotrebu i različitih složenih okolnosti koje mogu da utiču na rešenje, može se pročitati u knjizi *Obrasci za projektovanje* [Gamma1995].

6.5 Primer – Obrazac *Posetilac*

Kao treći primer obrazaca razmotrićemo obrazac *Posetilac* (engl. *Visitor*). To je obrazac ponašanja, koji opisuje kako možemo da na relativno jednostavan način apstrahujemo operacije koje se odvijaju uz obilaženje složenih struktura objekata, kao što su grafovi i stabla. Nastavićemo na mestu na kome smo stali u opisivanju obrasca *Sastav*, sa jednostavnom hijerarhijom likova. Za dalje razmatranje ovog primera najvažnije nam je da obratimo pažnju na sledeće:

- hijerarhija sadrži sastav `SlozenLik` i
- sve klase implementiraju metod `Povrsina()`.

Prethodni kod je relativno jednostavan i intuitivan. Svi metodi su tamo gde im je mesto i rade jasno razdvojene poslove. Ali šta bi bilo kada bismo osim metoda `Povrsina()` imali još nekoliko metoda, kao na primer, `Obim()`, `BrojLikova()` i druge? Sve nove metode bismo implementirali kroz celu hijerarhiju likova: deklarisali bismo virtualne metode u klasi `Lik`, a zatim napisali odgovarajuće implementacije u klasama hijerarhije. I dalje bi takvo rešenje bilo relativno jednostavno i intuitivno, ali možemo da primetimo i dve nezanemarljive slabosti:

- u svim novim metodima u klasi `SlozenLik` imali bismo ponavljanje koda koji implementira obilazak kolekcije;
- sa porastom broja klasa i metoda, sagledavanje ukupne implementacije jedne vrste izračunavanja bi postalo relativno teško – da bismo videli kako se računa površina morali bismo da pronađemo sve odgovarajuće metode u različitim klasama hijerarhije; takođe, i eventualno menjanje načina izračunavanja bi zahtevalo unošenje izmena u veliki broj klasa.

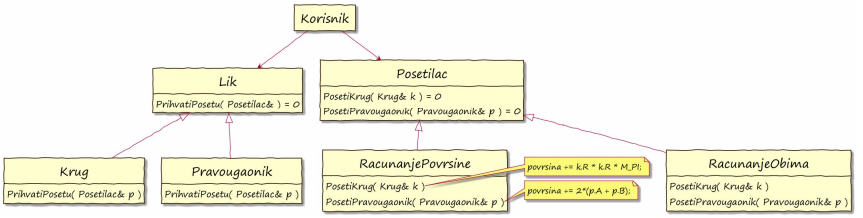
Tu stupa na scenu obrazac *Posetilac*. On uvodi važna unapređenja dizajna i prevazilazi obe navedene slabosti. Eliminiše ponavljanje delova koda koji se odnose na obilazak složene strukture i locira sve aspekte izvršavanja jedne operacije u jednoj klasi.

Koncept rešenja je relativno jednostavan. Logika koja odgovara poslu zbog koga se struktura obilazi se implementira u novoj klasi `Posetilac`. Klasa `Posetilac` ima po jedan metod za svaku klasu hijerarhije koja se obilazi. Logika obilaska strukture i prepoznavanja konkretnih klasa hijerarhije se implementira u okviru hijerarhije koja se obilazi, pomoću dodatnog metoda `PrihvatiPosetu(Posetilac&)`. Ideja je slična ideji implementacije metoda `Kopija` opisanoj u okviru opisa obrasca *Sastav*: posetilac ne može da zna kojoj klasi pripada objekat koji se posećuje ni da li on sadrži neke podobjekte, ali objekat koji se posećuje to *zna*.

Metod `PrihvatiPosetu` se implementira u svakoj od klasa hijerarhije na odgovarajući način. U slučaju klasa *listova*, koje ne sadrže druge objekte, implementacija se po pravilu svodi na pozivanje metoda posetioca koji odgovara klasi posećenog objekta. Na primer, na klasi `Pravougaonik` može da se implementira ovako:

```
void PrihvatiPosetu( PosetilacLikova& p )
    { p.PosetiPravougaonik( *this ); }
```

Na taj način implementacije metoda `PrihvatiPosetu` zapravo „javljaju“ posetiocu kojoj klasi pripadaju i koji konkretan metod posećivanja je potrebno da se primeni.



Slika 20 – Obrazac *Posetilac*, dijagram klasa

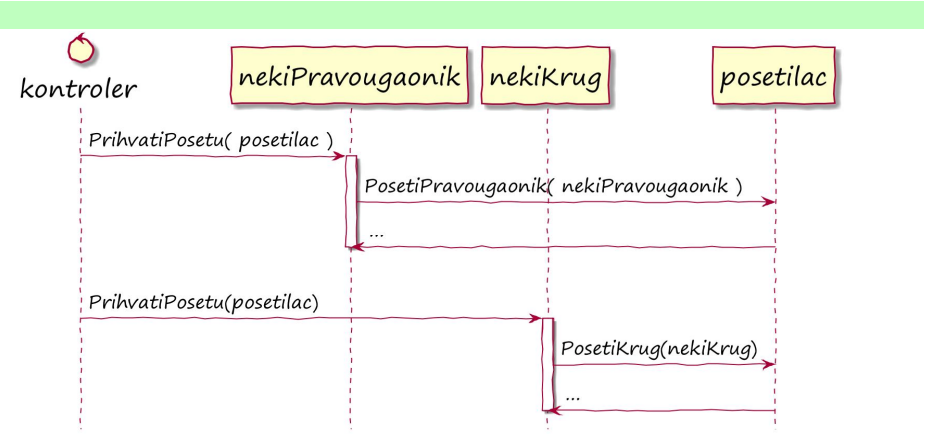
U slučaju klasa koje mogu da sadrže druge objekte, pored pozivanja metoda posetioca koji odgovara konkretnoj klasi, metod `PrihvatiPosetu` mora da omogući posetiocu da obiđe i sve sadržane objekte. To se radi pozivanjem metoda `PrihvatiPosetu` za svaki od sadržanih objekata. Na primer, u klasi `SlozenLik`, implementacija prihvatanja posete može da se napiše ovako:

```
void PrihvatiPosetu( PosetilacLikova& p )
{
    p.PosetiSlozeniLik( *this );
    for( auto lik: _Likovi )
        lik->PrihvatiPosetu(p);
}
```

U zavisnosti od prirode hijerarhije i poslova radi kojih se vrši obilazak, poseta sadržanih objekata može da ide pre ili posle poziva metoda `PosetiSlozeniLik`.

Štaviše, u nekim slučajevima ima smisla da posetilac implementira po dva metoda za posećivanje složenih objekata, od kojih se jedan poziva pre a drugi posle posećivanja sadržanih objekata.

Obilaženje neke strukture objekata započinje tako što se na osnovnom objektu strukture pozove metod `PrihvatiPosetu(posetilac)`. U zavisnosti od konkretne klase i njene unutrašnje strukture, metod `PrihvatiPosetu` poziva jedan ili više odgovarajućih metoda posetioca. Postupak obilaska ilustruje sledeći dijagam sekvence:



Slika 21 – Obrazac *Posetilac*, dijagram sekvence

Klasa `Posetilac` u potpunosti obuhvata i implementira sve što se odnosi na obavljanje konkretnog posla. Na primer, ako posetilac izračunava površinu likova, onda bi metod `PosetiPravougaonik` izračunavao površinu pravougaonika. Pri tome posetilac komunicira sa objektom koji posećuje iz samo dva razloga – posećeni objekat mora da „javi“ kojoj klasi pripada i da omogući obilaženje podobjekata. Posetilac sam obezbeđuje sve dodatne potrebne podatke i algoritme, kao što je na primer izračunavanje površine i čuvanje rezultata u našem primeru:

```
void PosetiPravougaonik( Pravougaonik& p )
    { povrsina += p.Sirina() * p.Visina(); }
```

Primetimo da predstavljena implementacija koristi rekurziju. To može da predstavlja problem u slučaju veoma složenih struktura podataka, zbog potencijalno velike dubine rekurzije. U tom slučaju posetilac može da se implementira tako da sadrži listu potrebnih poseta, a metod `SlozenLik::PrihvatiPosetu` bi umesto rekurzivnog pozivanja metoda `PrihvatiPosetu` samo dodavao sve sadržane objekte u tu listu.

Obrazac *Posetilac* se obično implementira samo ako je potrebno podržati više različitih posetilaca, tj. ako imamo više različitih razloga ili načina posećivanja (tj.

obilaženja) elemenata jedne hijerarhije. Svi potrebni posetioci se organizuju u posebnu hijerarhiju. Interfejs hijerarhije posetilaca mora da odgovara hijerarhiji klasa koja će se obilaziti. Bazna klasa samo deklariše interfejs (apstraktne metode) a konkretni posetioci ih implementiraju. U našem slučaju bazna klasa hijerarhije posetilaca može da bude:

```
class PosetilacLikova
{
public:
    virtual void PosetiPravougaonik( Pravougaonik& ) = 0;
    virtual void PosetiKvadrat( Kvadrat& ) = 0;
    virtual void PosetiKrug( Krug& ) = 0;
    virtual void PosetiSlozeniLik( SlozenLik& ) = 0;
};
```

Konkretan posetilac bi mogao da bude:

```
class RacunanjePovrsine : public PosetilacLikova
{
public:
    RacunanjePovrsine()
        : _Povrsina(0)
        {}

    double Povrsina() const
        { return _Povrsina; }

    void PosetiKrug( Krug& k )
        { _Povrsina += k.R() * k.R() * M_PI; }

    void PosetiKvadrat( Kvadrat& k )
        { _Povrsina += k.Sirina() * k.Sirina(); }

    void PosetiPravougaonik( Pravougaonik& p )
        { _Povrsina += p.Sirina() * p.Visina(); }

    void PosetiSlozeniLik( SlozenLik& )
        {}

private:
    double _Povrsina;
};
```

U narednom primeru predstavljamo primer koda, uz implementaciju dva posetioca, za računanje površine i obima lika. Radi uštede prostora navodimo samo delove koda koji su izmenjeni u odnosu na prethodni primer obrasca *Sastav*:

```
class Pravougaonik;
class Kvadrat;
class Krug;
class SlozenLik;
```

```
class PosetilacLikova
{
public:
    virtual ~PosetilacLikova() {}
    virtual void PosetiPravougaonik( Pravougaonik& ) = 0;
    virtual void PosetiKvadrat( Kvadrat& ) = 0;
    virtual void PosetiKrug( Krug& ) = 0;
    virtual void PosetiSlozeniLik( SlozenLik& ) = 0;
};

class Lik
{...
    virtual void PrihvatiPosetu( PosetilacLikova& p ) = 0;
};

class Pravougaonik : public Lik
{...
    void PrihvatiPosetu( PosetilacLikova& p )
        { p.PosetiPravougaonik( *this ); }
};

class Kvadrat : public Pravougaonik
{...
    void PrihvatiPosetu( PosetilacLikova& p )
        { p.PosetiKvadrat( *this ); }
};

class Krug : public Lik
{...
    void PrihvatiPosetu( PosetilacLikova& p )
        { p.PosetiKrug( *this ); }
};

class SlozenLik : public Lik
{...
    void PrihvatiPosetu( PosetilacLikova& p )
    {
        p.PosetiSlozeniLik( *this );
        for( auto lik: _Likovi )
            lik->PrihvatiPosetu(p);
    }
};

// posetilac koji računa površinu
class RacunanjePovrsine : public PosetilacLikova
{
public:
    RacunanjePovrsine()
        : _Povrsina(0)
        {}

    double Povrsina() const
        { return _Povrsina; }
```

```
    void PosetiPravougaonik( Pravougaonik& p )
        { _Povrsina += p.Sirina() * p.Visina(); }

    void PosetiKvadrat( Kvadrat& k )
        { _Povrsina += k.Sirina() * k.Sirina(); }

    void PosetiKrug( Krug& k )
        { _Povrsina += k.R() * k.R() * M_PI; }

    // složen lik nema dodatnu površinu u odnosu na komponente
    // a one će vec biti posebno posećene
    void PosetiSlozeniLik( SlozenLik& )
        {}

private:
    double _Povrsina;
};

// posetilac koji računa obim
class RacunanjeObima : public PosetilacLikova
{
public:
    RacunanjeObima()
        : _Obim(0)
        {}

    double Obim() const
        { return _Obim; }

    void PosetiPravougaonik( Pravougaonik& p )
        { _Obim += 2* ( p.Sirina() + p.Visina()); }

    void PosetiKvadrat( Kvadrat& k )
        { _Obim += 4 * k.Sirina(); }

    void PosetiKrug( Krug& k )
        { _Obim += 2 * k.R() * M_PI; }

    // složen lik nema dodatni obim u odnosu na komponente
    // a one će vec biti posebno posećene
    void PosetiSlozeniLik( SlozenLik& )
        {}

private:
    double _Obim;
};

int main()
{...
    RacunanjePovrsine rp;
    sl.PrihvatiPosetu(rp);
    cout << "POVRSINA: " << rp.Povrsina() << endl;

    RacunanjeObima ro;
```

```
    sl.PrihvatiPosetu(ro);
    cout << "OBIM: " << ro.Obim() << endl;

    return 0;
}
```

Već smo istakli kvalitete obrasca *Posetilac*. Međutim, moramo da istaknemo i neke slabosti. Prva slabost je da primena obrasca *Posetilac* može da oteža dodavanje novih klasa hijererhiji klasa koje se obilaze – svaki put kada se dodaje nova klasa u hijerarhiju mora da se doda i odgovarajući nov metod u sve postojeće klase posetilaca. Druga slabost je da za neka izračunavanja može da bude neophodno da posetilac naruši enkapsulaciju nekih klasa hijerarhije.

Implementacija posetilaca u programskom jeziku C++ pruža i neke dodatne pogodnosti. Na primer, ako u okviru klasa posetilaca za računanje površine i obima dodamo konstruktorima kao argument objekat od koga se započinje obilaženje, pa još implementiramo i konvertor posetioca u realan broj, onda klase posetilaca možemo da koristimo i kao funkcije:

```
class RacunanjePovrsine : public PosetilacLikova
{
public:
    RacunanjePovrsine( Lik& l)
        : _Povrsina(0)
        { l.PrihvatiPosetu(*this); }

    operator double()
        { return Povrsina(); }
...
};

class RacunanjeObima : public PosetilacLikova
{
public:
    RacunanjeObima( Lik& l )
        : _Obim(0)
        { l.PrihvatiPosetu(*this); }

    operator double()
        { return Obim(); }
...
};

int main()
{...
    cout << "POVRSINA: " << RacunanjePovrsine(sl) << endl;
    cout << "OBIM: " << RacunanjeObima(sl) << endl;

    return 0;
}
```

6.6 Katalog obrazaca

Svaki put kada u razvoju softvera naiđemo na neki problem, koji je sličan nekom poznatom obrascu, možemo da razmotrimo primenu tog prepoznatog obrasca na način koji je prilagođen konkretnom slučaju. Da bi pronalaženje odgovarajućeg obrasca bilo lakše, prave se *katalozi obrazaca*. Da bi obrasci u katalozima mogli da se lako pronađu i prepoznaju, daju im se prepoznatljiva imena i klasifikuju se.

Ne postoje formalno ustanovljeni kriterijumi za odlučivanje o tome da li neko konceptualno rešenje zaslužuje da bude dokumentovano i predstavljeno u katalogu obrazaca. Kriterijumi u velikoj meri zavise od vrste kataloga, a prvenstveno od njegove namene i ciljane korisničke grupe. Katalog obrazaca može biti opšte namene, ali i vrlo specifične namene (na primer, katalog obrazaca za razvoj korisničkih interfejsa). U skladu sa tim razlikuju se i nivoi apstrakcije obrazaca, kao i njihov značaj. Neki obrazac može da bude značajan u nekom katalogu specifične namene, a da istovreeno nema dovoljno značajnu ulogu da bi se pojavio u nekom katalogu šireg domena, ali i obratno.

U skladu sa ranije navedenim važnim elementima obrazaca, svaki obrazac u katalogu se detaljno dokumentuje. Dokumentacija obrazaca može da obuhvata njihove različite karakteristike, zavisno od vrste i namene kataloga. U opisima obrazaca u katalogu se obično navode sledeće stavke:

- **ime obrasca**, koje sažeto izlaže suštinu obrasca. Veoma je važno da ime bude dobro odabrano, zato što ono ulazi u rečnik projektovanja;
- **alternativna imena** su druga imena pod kojima je poznat isti obrazac;
- **klasifikacija** omogućava lakše snalaženje među obrascima u katalogu. Obično se izvodi prema kontekstu primene, prema tehnici koja je dominantna u obrascu ili oboje;
- **namena** ukratko opisuje čemu obrazac služi, koji problem rešava, šta pojednostavljuje, kada se koristi i slično;
- **motivacija** za primenu opisuje doprinose obrasca; može da sažeto ukazuje na osnovne odnose elemenata obrazaca i da predstavlja uvod u detaljnije opise;
- **primenjivost** opisuje gde i kada se obrazac može da se upotrebi i koje probleme može da reši;
- **preduslovi za primenu** moraju da se navedu u okviru opisa obrasca, da bi se predupredile potencijalne greške u vidu neodgovarajuće primene obrasca;
- **logička struktura rešenja** se obično predstavlja nekom od dijagramskih tehnika i pratećim tekstom; uobičajeno je da se koriste dijagrami *UML*-a;

- **entiteti rešenja** su klase ili objekti koji imaju važnu ulogu u obrascu za projektovanje; među entitetima su i specifični pomoćni entiteti koje uvodi obrazac, ali i oni koji postoje nezavisno od primene obrasca, a na koje se obrazac odnosi;
- **saradnja** između učesnika u rešenju predstavlja opis tokova poruka između objekata obrasca; uobičajeno je da se opisuje dijagramom sekvenci *UML*-a;
- **posledice primene rešenja** su neophodne kao i preduslovi, zato što mogu da utiču na odlučivanje o primeni obrasca ali i da ukažu na potencijalne probleme ili dodatne koristi od primene obrasca;
- **implementacija** predstavlja načelno uputstvo za primenu obrasca, pri čemu u složenijim slučajevima može biti da se navede i više uputstava za različite okolnosti ili za različite programske jezike;
- **poznati primeri korišćenja** predstavljaju najneposredniji način da se potencijalnom korisniku obrasca predstave njegovo ponašanje i struktura;
- **primeri koda** predstavljaju konkretne primere implementacije obrasca;
- **pregled srodnih obrazaca** ukazuje na srodne obrasce u istom ili drugom katalogu obrazaca.

Ukratko ćemo da predstavimo katalog obrazaca izložen u knjizi „Obrasci za projektovanje“. Ovaj katalog je opšte namene i obuhvata najvažnije obrasce za projektovanje koji su autorima bili poznati u vreme pisanja knjige. Obrasci su prema nameni klasifikovani u tri grupe:

- gradivni obrasci (engl. *Creational Patterns*);
- strukturni obrasci (engl. *Structural Patterns*) i
- obrasci ponašanja (*Behavioral Patterns*).

U svakoj od ovih grupa obrasci su dalje podeljeni prema domenu primene, na one koji rešavaju problem uspostavljanjem ili iskorišćavanjem odnosa između klasa (tzv. *obrasci sa domenom klase*) i one koji u rešenju počivaju na odnosima između objekata (tzv. *obrasci sa domenom objekata*).

Gradivni obrasci predstavljaju rešenja za različite načine pravljenja novih objekata. Oni apstrahuju postupak pravljenja novih objekata i omogućavaju da se na uopšten način pristupi kako samom pravljenju tako i međusobnom povezivanju napravljenih objekata. Iako u nekim uslovima mogu da se primene različiti gradivni obrasci, oni se međusobno veoma razlikuju – ako im se u nekom slučaju ipak poklope uslovi za primenu, onda im se značajno razlikuju posledice. Zajedničko za sve ove obrasce je da oni pokušavaju da od korisnika sakriju veći deo složenosti postupaka pravljenja i

povezivanja napravljenih objekata. Od korisnika se sakrivaju međusobni odnosi objekata i klasa, često i sami tipovi napravljenih objekata. Umesto toga, korisniku se stavljaju na raspolaganje samo odgovarajući (najčešće apstraktni) interfejsi za pravljenje, povezivanje i korišćenje objekata.

Ovi obrasci omogućavaju da se različite klase i objekti sa složenim ponašanjem grade na relativno jednostavan način sastavljanjem manjeg skupa osnovnih ponašanja. Oni tako enkapsuliraju složene aspekte pravljenja objekata.

Gradivni obrasci mogu da imaju domen klase ili domen objekata. Gradivni obrasci sa domenom klase uglavnom koriste nasleđivanje kao osnovni odnos među objektima koji se prave. Gradivni obrasci sa domenom objekta prepuštaju (delegiraju) pravljenje nekom drugom objektu.

U gradivne obrasce spadaju:

- Proizvodni metod (engl. *Factory Method*);
- Apstraktna fabrika (engl. *Abstract Factory*);
- Graditelj (engl. *Builder*);
- Prototip (engl. *Prototype*) i
- Unikat (engl. *Singleton*).

Strukturni obrasci rešavaju probleme organizovanja objekata i klasa u veće funkcionalne celine. Strukturni obrasci sa domenom klasa koriste nasleđivanje klasa i hijerarhije za povezivanje objekata i klasa. Strukturni obrasci sa domenom objekata neposredno povezuju objekte u veće celine. Zbog toga što su odnosi među klasama statički, a odnosi među objektima su dinamički i mogu se menjati tokom rada programa, strukturni obrasci sa domenom objekata se upotrebljavaju za pravljenje fleksibilnijih složenih struktura.

U strukturne obrasce spadaju:

- Adapter (engl. *Adapter*);
- Most (engl. *Bridge*);
- Sastav (engl. *Composite*);
- Dekorater (engl. *Decorator*);
- Fasada (engl. *Facade*);
- Muva (engl. *Flyweight*) i

- Proksi[15] (engl. *Proxy*).

Obrasci ponašanja se odnose na implementiranje algoritama ili raspodelu odgovornosti između objekata. Oni rešavaju različite probleme pomoću uspostavljanja odgovarajućih odnosa među klasama i objektima i kroz odgovarajuće načine razmenjivanja informacija (poruka) između objekata. Jedna od osnovnih namena im je da enkapsuliraju složenu prirodu kontrole i prebace složenost upotrebe na povezivanje objekata. Klasni obrasci ponašanja kao osnovno sredstvo za distribuiranje ponašanja među klasama koriste nasleđivanje. Sa druge strane, u objektnim obrascima ponašanja značajniju ulogu igra povezivanje pojedinačnih objekata.

U obrasce ponašanja spadaju:

- Lanac odgovornosti (engl. *Chain of Responsibility*);
- Komanda (engl. *Command*);
- Interpretator (engl. *Interpreter*);
- Iterator (engl. *Iterator*);
- Posrednik (engl. *Mediator*) ;
- Podsetnik (engl. *Memento*);
- Posmatrač (engl. *Observer*);
- Stanje (engl. *State*);
- Strategija (engl. *Strategy*);
- Šablonski metod (engl. *Template Method*) i
- Posetilac (engl. *Visitor*).

6.7 Umesto zaključka

Imajući u vidu značaj obrazaca za projektovanje, čitaocima se preporučuje da pročitaju knjigu *Obrasci za projektovanje* [Gamma1995] i prouče obuhvaćene obrasce. Pored toga što nosi mnogo informacija o obrascima za projektovanje, čitaoci imaju priliku da se iz nje upoznaju i sa mnogo dobrih primera objektno orijentisanog projektovanja i programiranja.

Napisano je još mnogo knjiga o obrascima za projektovanje, čiji su autori pokušali da ovu temu predstave na još bolji ili razumljiviji način, ili da prepoznaju i ponude

[15] Uobičajen prevod za termin *proxy* je *posrednik*, ali ovde to nije prihvatljivo, zato što postoji uzorak ponašanja koji se zove *Posrednik* (engl. *Mediator*). Da bi se izbegli nesporazumi, za ovaj uzorak je uobičajeno da se upotrebljava termin *Proksi*.

čitaocima nove obrasce. Na primer, Martin Fauler je napisao odličnu knjigu o obrascima za projektovanje u domenu arhitektura poslovnih aplikacija [Fowler2002].

Na internetu postoji mnogo izvora informacija o obrascima za projektovanje. Među njima može da se izdvoji veb lokacija Vinsa Hjustona [HustonDP].

Pregled referenci

[Alex2001]
Andrei Alexandrescu , **Modern C++ Design: Generic Programming and Design Patterns Applied**, *Addison-Wesley Professional*, 2001.

[Fowler2002]
Martin Fowler, **Patterns of Enterprise Application Architecture**, *Addison-Wesley*, 2002.

[Gamma1995]
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**, *Addison-Wesley*, 1995.
Izdanje na srpskom jeziku: **Gotova rešenja: Elementi objektno orijentisanog softvera**, *CET*.

[HustonDP]
Vince Huston, `http://www.vincehuston.org/dp/`