Objektno orijentisano programiranje u C++-u
Projektni uzorci

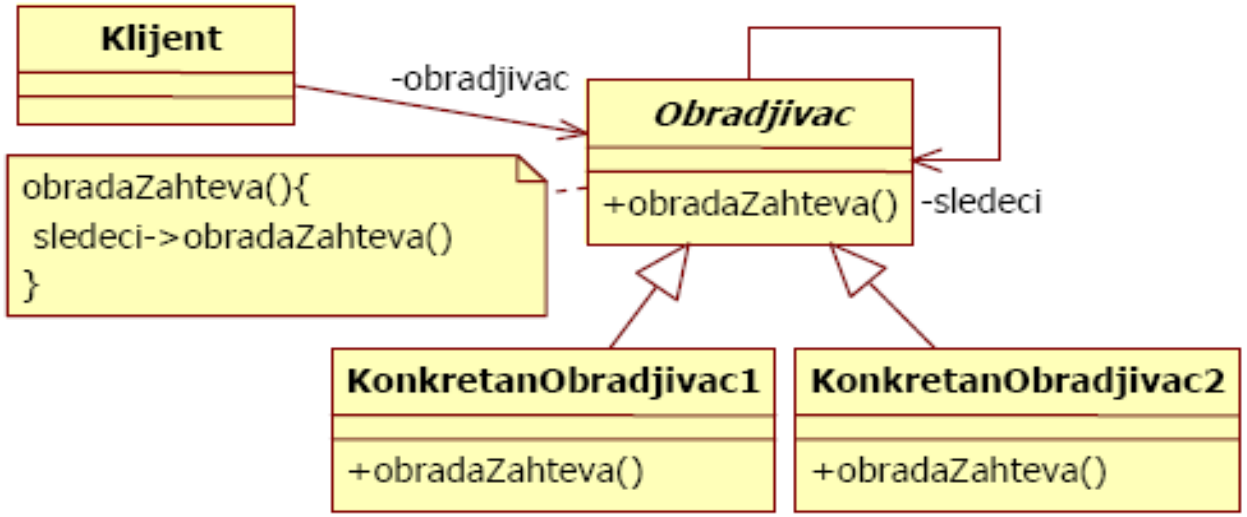
PROJEKTNI UZORCI PONAŠANJA OBJEKATA

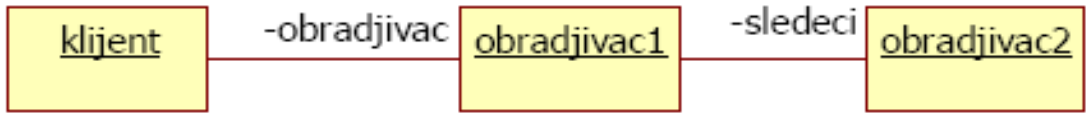
LANAC ODOGOVORNOSTI
KOMANDA
POSMATRAČ
POSREDNIK

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
1. Uvod

- **Ime i klasifikacija**
  - **Lanac odgovornosti (engl. Chain of responsibility)**
  - **Projektni uzorak ponašanja objekata**
- **Namena**
  - **Povezuje objekte kojima se upućuje zahtev u lanac i prosleđuje zahtev niz lanac, sve dok ga neki od objekata ne obradi**
  - **Izbegava neposredno vezivanje pošiljaoca zahteva sa primaocem, čime se omogućava većem broju objekata da obradi zahtev**
- **Primenljivost**
  - **Više objekata može da obradi zahtev, ali se ne zna unapred koji će ga obraditi**
  - **Dinamička “switch” naredba**

Struktura


Klijent
-obradjivac
Obradjivac
obradaZahteva(){
sledeci->obradaZahteva()
}
+obradaZahteva()
-sledeci
KonkretanObradjivac1
+obradaZahteva()
KonkretanObradjivac2
+obradaZahteva()


klijent
-obradjivac
obradjivac1
-sledeci
obradjivac2

- **Učesnici**
  - **Procesor (Obrađivač)**
    - Definiše interfejs obrade zahteva klijenta
    - Implementira vezu prema sledećem objektu u lancu
  - **KonkretanProcesor (KonkretanObrađivač)**
    - Obrađuje one zahteve za koje je napravljen.
    - Ukoliko ne može da obradi zahtev prosleđuje ga sledećem u lancu
- **Saradnja**
  - **Kada klijent izda zahtev, on putuje po lancu odgovornosti dok konkretni procesir ne preuzme odgovornost za obradu**
- **Povezani uzorci**
  - **Često se primenjuje sa uzorkom Kompozitnog. Roditelj komponente može da bude sledeći u lancu**

- **Posledice**
  - **Razdvajanje pošiljaoca i primaoca**
    - Pošiljalac i primalac ne treba da znaju ništa jedan o drugome
    - Objekat u lancu ne treba da poznaje strukturu lanca
    - Smanjuje se broj veza izmedju objekata. Objekat pristupa samo sledbeniku u lancu
  - **Dodatna fleksibilnost u pridruživanju odgovornosti objektima**
    - Odgovornosti za obradu zahteva se mogu dodavati i menjati u toku izvršavanja
  - **Prijem i obrada zahteva nisu garantovani**
    - Zahtev može da stigne do kraja lanca i da ne bude obrađen

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 1.


```
class Upit; //Deklaracija unapred
/* Interfejs objekta iz lanca odlgovornosti*/
class IOsoba{
private:
    string m_ime;
    IOsoba* m_sef;
public:
        IOsoba(string ime):m_ime(ime){};
    virtual void ObradiUpit( const Upit &upit) = 0;
    virtual string Ime() const { return m_ime; }
    virtual IOsoba* PtrToSef() const { return m_sef; }
    virtual void PostaviSefa(IOsoba* sef) { m_sef = sef; }
        /* Objekat nije odogovoran za unistavanje sledeceg clana u
lancu odgovornosti */
        virtual ~IOsoba(){}
};

enum ENivoOdgovornosti{NIZAK = 1, SREDNJI, VISOK};
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 1.


```
/* Konkretne klase objekata u lancu odgovornosti bice instancirane iz
ovog sablona */
template<enum ENivoOdgovornosti nivo>
class Radnik: public IOsoba{
public:
   Radnik(string ime): IOsoba(ime){}
   virtual void ObradiUpit(const Upit &upit){
      if(upit.NivoUpita() == nivo){
      /* Trenutni objekat u lancu JESTE sposoban da da odgovor */
         cout<<"Odgovor na pitanje: "<<upit.StaJeUpit().c_str()
                                    <<" " je dao "<<Ime().c_str()<<endl;
         return;
      }
      /* Trenutni objekat u lancu NIJE sposoban da da odgovor */
      cout<<"Osoba "<<Ime().c_str()
      <<" ne moze da obradi trenutni upit. Upit se salje na dalju
obradu: "
      << (PtrToSef()->Ime()).c_str()<<endl;
      /* Zahtev se salje na dalju obradu sledecem objektu u lancu
odgovornosti */
      PtrToSef()->ObradiUpit(upit);
   }
};
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 1.


```
/* Zahtev koji se salje objektima u lancu odgovornosti */
class Upit{
private:
    ENivoOdgovornosti m_nivo;
    string m_upit;
public:
    Upit(string upit, ENivoOdgovornosti nivo): m_upit(upit),
m_nivo(nivo){}
    ENivoOdgovornosti NivoUpita() const{ return m_nivo; }
   string StaJeUpit() const { return m_upit; }
};
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 1.


```
/* KLIJENT */

int main()
{
   Radnik<NIZAK> rad("Petar Djuric");
   Radnik<SREDNJI> sup("Vojislav Mitrovic");
   Radnik<VISOK> sef("Miroslav Stevanovic");

   rad.PostaviSefa(&sup);
   sup.PostaviSefa(&sef);
   sef.PostaviSefa(NULL);

   Upit q1("Mozes li da zavrsis ovaj zadatak?", NIZAK);
   Upit q2("Da li tim moze da zavrsi zadataka na vreme?", SREDNJI);
   Upit q3("Da li odeljenje proizvodnje moze da zavrsi zadatak?",
VISOK);

   rad.ObradiUpit(q1);
   rad.ObradiUpit(q2);
   rad.ObradiUpit(q3);
}
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 1.


```
/* KLIJENT */

int main()
{
    Radnik<NIZAK> rad("Petar Djuric");
    Radnik<SREDNJI> sup("Vojislav Mitrovic");
    Radnik<V
    rad.Post
    sup.Post
    sef.Post

    Upit q1(
    Upit q2(                                                    DNJI);
    Upit q3(                                                    k?",
VISOK);

    rad.Obra
    rad.ObradiUpit(q2);
    rad.ObradiUpit(q3);
}
```

```
Odgovor na pitanje: "Mozes li da zavrsis ovaj zadatak?" je dao Petar Djuric
Osoba Petar Djuric ne moze da obradi trenutni upit. Upit se salje na dalju obrad
u: Vojislav Mitrovic
Odgovor na pitanje: "Da li tim moze da zavrsi zadataka na vreme?" je dao Vojisla
v Mitrovic
Osoba Petar Djuric ne moze da obradi trenutni upit. Upit se salje na dalju obrad
u: Vojislav Mitrovic
Osoba Vojislav Mitrovic ne moze da obradi trenutni upit. Upit se salje na dalju
obradu: Miroslav Stevanovic
Odgovor na pitanje: "Da li odeljenje proizvodnje moze da zavrsi zadatak?" je dao
 Miroslav Stevanovic
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 2.


```
/* Apstraktna klasa objekata u lancu odgovornosti */
class AProcesor {
public:
   AProcesor() : m_Next(NULL) {}
   SmartPtr<AProcesor> setNext(AProcesor *ptr){
      return m_Next = ptr;
   }
   /* Ovom funkcijom se omogucava samo prenosenje zahteva sledecem
objektu u lancu odgovornosti */
   virtual void obradi( const string& adresa, const string& poruka){
      if (m_Next)
         m_Next->obradi(adresa, poruka);
   }
   virtual ~AProcesor(){}
private:
   /* Pametni pokazivac na sledeceg clana u lancu odgovornosti.
Upotrebom pametnog pokazivaca, koji broji reference na objekat i
unistava ga samo ako ne postiji ni jedna referenca na njega,
IZBEGAVAMO CURENJE MEMORIJE */
   SmartPtr<AProcesor> m_Next;
};
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 2.


```
/* KONRETNE KLASE - PROCESORI ZAHTEVA */
class MejlProcesor : public AProcesor {
public:
   void obradi(const std::string& adresa, const std::string& poruka){
      if (adresa.find_first_of('@') != string::npos)
      /* Objekat je sposoban da obradi zahtev*/
         cout << "Mail poslat " << adresa << endl;
      else
      /* Zahtev se delegira sledecem u lancanoj listi odgovornosti */
         AProcesor::obradi(adresa, poruka);
   }
   virtual ~MejlProcesor(){ }
};
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 2.


```
/* KONRETNE KLASE - PROCESORI ZAHTEVA */
class SMSHandler : public AProcesor
{
public:
   void obradi(const std::string& adresa, const std::string& poruka)
   {
      if (adresa.substr(0, 2) == "06")
      /* Objekat je sposoban da obradi zahtev*/
         cout << "SMS poslat " << adresa << endl;
      else
      /* Zahtev se delegira sledecem u lancanoj listi odgovornosti */
         AProcesor::obradi(adresa, poruka);
   }
   virtual ~SMSHandler(){}
};
```

LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 2.


```
/* KONRETNE KLASE - PROCESORI ZAHTEVA */
class InstantMessageHandler : public AProcesor{
public:
   void obradi(const std::string& adresa, const std::string& poruka)
   {
      if (adresa.substr(0, 3) == "IM:")
      /* Objekat je sposoban da obradi zahtev*/
         cout << "InstantMessage poslat "
                              << adresa << endl;
      else
      /* Zahtev se delegira sledecem u lancanoj listi odgovornosti */
         AProcesor::obradi(adresa, poruka);
   }
   virtual ~InstantMessageHandler(){}
};
```

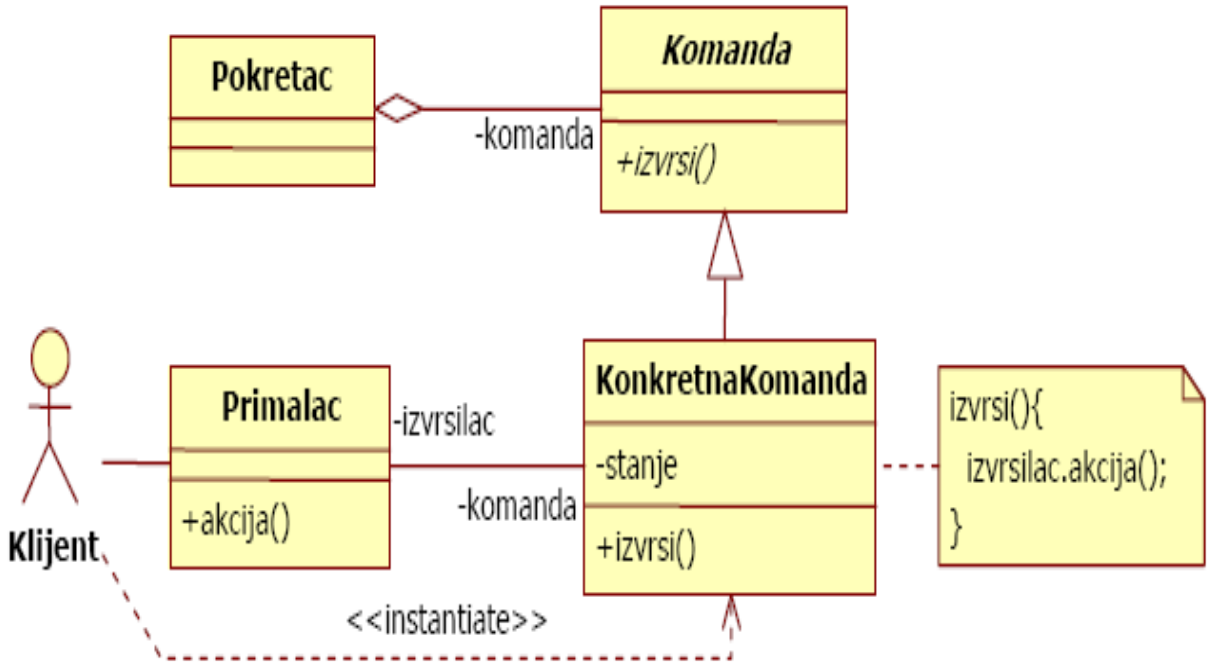
LANAC ODGOVORNOSTI (engl. CHAIN OF RESPONSIBILITY)
Primer 2.


```
/* KLIJENT */

int main(int argc, char *argv[]){
   /* Cuva se "head" lanca odgovornosti. Polazna tacka za slanje
poruka. SetNext u lancu odgovornosti uvek vraca SmartPointer na
sledecg u lancu. SmartPointer t pokazuje na kraj lanca. */
   SmartPtr<AProcesor> h = new InstantMessageHandler(), t = h;
   t=t->setNext(new MejlProcesor());
   t=t->setNext(new SMSHandler());
   /* Prethodni pristup registrovanja procesora u lancu odgovornosti
bi sigurno doveo do "curenja memorije" da nisu korisceni pametni
"reference counting" pokazivaci na procesore */

/*Lancu odgovornosti se pristupa preko njegovog prvo clana - glave */
   h->obradi("062 123 45 67", "SMS poruka");
   h->obradi("petar.pterovic@gmail.com", "Email poruka");
   h->obradi("I
poruka ");

   return 0;
}
```

```
SMS poslat 062 123 45 67
Mail poslat petar.petrovic@gmail.com
InstantMessage poslat IM:mojprijatelj@192.168.10.121
```

- **Ime i klasifikacija**
  - **Komanda (engl. Command)**
  - **Projektni uzorak ponašanja objekata**
- **Drugo ime**
  - **Akcija, Transakcija (engl. Action, Transaction)**
- **Namena**
  - **Inkapsulira zahtev u objekat, omogućavajući:**
    - da se zahtevi isporučuju kroz red čekanja,
    - da se pravi log(dnevnik) zahteva i
    - da se efekti izvršenog zahteva ponište.

Struktura


Pokretac
Komanda
-komanda
+izvrsi()
Klijent
Primalac
-izvrsilac
+akcija()
KonkretnaKomanda
-stanje
-komanda
+izvrsi()
izvrsi(){
izvrsilac.akcija();
}
<<instantiate>>

- **Primenljivost**
  - **Kada akciju treba proslediti kao parametar objektu (slanjem pokazivaca na funkciju na primer u C-u)**
  - **Kada zahtevi treba da se stavljaju u red čekanja i naknadno izvršavaju**
    - Objekat komande može da ima različit životni vek od objekta koji izdaje komandu
    - Objekat komande se može prepustiti drugom adresnom prostoru
  - **Kada treba da se podrži otkazivanje izvršene komande**
    - U tom slučaju operacija izvrsi() (execute()) mora da se sačuva prethodno stanje objekta
    - Interfejs mora da implementira i operaciju otkazi_izvrseno() (undo())i ponovi_izvrseno() (redo())
    - Neograničeni nivo undo i redu se postiže smeštanjem objekata izvršenih komandi u listu, odnosno prolaskom kroz listu unazad i unapred

- **Učesnici**
  - **Komanda**
    - Deklariše interfejs za izvršenje neke operacije
  - **KonretnaKomanda**
    - Definiše vezu između objekta primaoca i akcije
  - **Klijent**
    - Kreira objekat klase KonkretnaKomanda i postavlja objekat klase Primalac nad kojim treba da se izvrši akcija
  - **Pokretac**
    - Traži od komande da izvrši zahtev
  - **Primalac**
    - Zna kako da izvrši operacije od kojih se zahtev sastoji

- **Posledice**
  - **Razdvaja objekat koji pokreće zahtev od onog koji zna kako da ga izvrši**
  - **Komande su objekti i njima se može manipulisati kao i sa svakim drugim objektom**
  - **Komande se mogu prikupiti u kompozitne (Makro) komande**
  - **Dodavanje novih komandi je jednostavno**
- **Povezani uzorci**
  - **Kompozicija se koristi za kreiranje makrokomandi (tzv skriptova)**

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/*      Interfejs komande          */
class IKomanda {
public:
 virtual void Izvrsi() = 0; /* Execute */
 virtual void OtkaziIzvrseno() = 0; /* Undo */
 virtual void PonovoIzvrsi() = 0; /*Redo */
};

/*      MODEL kojim se upravlja  - RECEIVER    */
class TV {
   bool m_ukljucen;
   int m_kanal;
public:
   TV(){}
   void Ukljuci() { m_ukljucen = true;}
   void Iskljuci() { m_ukljucen = false;}
   void PromeniKanal(int kanal) { m_kanal = kanal; }
   bool JeUkljucen() const { return m_ukljucen;}
   int Kanal() const { return m_kanal; }
};
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/*      KONKRETNE KOMANDE          */
class KomandaUkljuciTV : public IKomanda {
   TV *m_ptrTV; /* KonretnaKomanda definiše vezu između objekta
primaoca i akcije. */
public:
   KomandaUkljuciTV(TV &tv):m_ptrTV(&tv) {}
   void Izvrsi(){ m_ptrTV->Ukljuci(); }
   void OtkaziIzvrseno(){ m_ptrTV->Iskljuci();}
   void PonovoIzvrsi(){ m_ptrTV->Ukljuci(); }
};

class KomandaIskljuciTV : public IKomanda {
/* Agregira objekat klase KomandaUkljuciTV i radi suprotno  */
   KomandaUkljuciTV m_KomandaUkljuciTV;
public:
   KomandaIskljuciTV(TV &tv):m_KomandaUkljuciTV(tv) {}
   void Izvrsi(){ m_KomandaUkljuciTV.OtkaziIzvrseno(); }
   void OtkaziIzvrseno(){ m_KomandaUkljuciTV.Izvrsi(); }
   void PonovoIzvrsi(){ m_KomandaUkljuciTV.OtkaziIzvrseno(); }
};
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/*      KONKRETNE KOMANDE          */
class KomandaPromeniKanal : public IKomanda {
  TV *m_ptrTV; /* KonretnaKomanda definiše vezu između objekta
primaoca i akcije. */
  int m_prethodniKanal, m_noviKanal; /* KonretnaKomanda omogucava
cuvanje starog stanja. */

public:
   KomandaPromeniKanal(TV *ptv, int
kanal):m_ptrTV(ptv),m_noviKanal(kanal) {}

   void Izvrsi() {
      m_prethodniKanal = m_ptrTV->Kanal();
      m_ptrTV->PromeniKanal(m_noviKanal);
   }

   void OtkaziIzvrseno() { m_ptrTV->PromeniKanal(m_prethodniKanal); }

   void PonovoIzvrsi() { m_ptrTV->PromeniKanal(m_noviKanal); }
};
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* Menadzer komandi je odgovoran za odrzavanje steka otkaza i
ponovnog izvrsavanja */
class MenadzerKomandi { /* INVOKER */
   stack< SmartPtr<IKomanda> > m_StekOtkaza;
   stack< SmartPtr<IKomanda> > m_StekPonavljanja;
   bool m_bOtkaziPokrenut;

   void IsprazniStekove() {
      m_StekOtkaza = stack<SmartPtr<IKomanda> >();
      m_StekPonavljanja = stack<SmartPtr<IKomanda> >();
   }
public:
   MenadzerKomandi():m_bOtkaziPokrenut(false){}

   void IzvrsiKomandu(SmartPtr<IKomanda> komanda) {
      if (m_bOtkaziPokrenut) {
         /* Isprazni stekove otkaza i ponavljanja */
         IsprazniStekove(); m_bOtkaziPokrenut = false;
      }
    komanda->Izvrsi();
    m_StekOtkaza.push(komanda);
   }
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* Menadzer komandi je odgovoran za odrzavanje steka otkaza i ponovnog
izvrsavanja */
class MenadzerKomandi {
. . .
public:
. . .
   void OtkaziIzvrseno() { /* Undo */
      if (!m_StekOtkaza.empty()){

         /* Otkazi poslednju komandu */
         m_StekOtkaza.top()->OtkaziIzvrseno();

         /* Dodaj otkazanu komandu u stek otkaza */
         m_StekPonavljanja.push(m_StekOtkaza.top());

         /* Ukloni element sa vrha steka otkaza */
         m_StekOtkaza.pop();
      }
  }
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* Menadzer komandi je odgovoran za odrzavanje steka otkaza i ponovnog
izvrsavanja */
class MenadzerKomandi {
. . .
public:
. . .
   void PonovoIzvrsi() { /* Redo */
      if (!m_StekPonavljanja.empty()){

         /* Ponovo izvrsi poslednju komandu */
         m_StekPonavljanja.top()->PonovoIzvrsi();

         /* Dodaj ponovljenu komandu u stek otkaza */
         m_StekOtkaza.push(m_StekPonavljanja.top());

         /* Ukloni element sa vrha steka ponavljanja */
         m_StekPonavljanja.pop();
      }
   }
};
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* KLIJENT */
int _tmain(int argc, _TCHAR* argv[]){
   TV tv;
   MenadzerKomandi commandManager;

   commandManager.IzvrsiKomandu(new KomandaPromeniKanal(&tv, 43));
   cout << "Prebacen na kanal " << tv.Kanal() << endl;

   commandManager.IzvrsiKomandu(new KomandaPromeniKanal(&tv, 39));
   cout << "Prebacen na kanal " << tv.Kanal() << endl;

   commandManager.IzvrsiKomandu(new KomandaPromeniKanal(&tv, 53));
   cout << "Prebacen na kanal " << tv.Kanal() << endl;


. . .
   return 0;
}
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* KLIJENT */
int _tmain(int argc, _TCHAR* argv[]){
   TV tv;
   MenadzerKomandi commandManager;
   . . .

   cout << "vracam na prethodni kanal..." << endl;
   commandManager.OtkaziIzvrseno();
   cout << "Trenutni kanal: " << tv.Kanal() << endl;

   cout << "vracam na prethodni kanal..." << endl;
   commandManager.OtkaziIzvrseno();
   cout << "Trenutni kanal: " << tv.Kanal() << endl;

. . .
}
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* KLIJENT */
int _tmain(int argc, _TCHAR* argv[]){
   TV tv;
   MenadzerKomandi commandManager;
   . . .


   cout << "Ponovo prebacujem na kanal..." << endl;
   commandManager.PonovoIzvrsi();
   cout << "Trenutni kanal: " << tv.Kanal() << endl;

   cout << "Ponovo prebacujem na kanal..." << endl;
   commandManager.PonovoIzvrsi();
   cout << "Trenutni kanal: " << tv.Kanal() << endl;


   return 0;
}
```

KOMANDA (engl. COMMAND)
Primer 1: Otkazivanje i ponovljeno izvršavanje komande


```
/* KLIJENT */
Prebacen na kanal 43
Prebacen na kanal 39
Prebacen na kanal 53
vracam na prethodni kanal...
Trenutni kanal: 39
vracam na prethodni kanal...
Trenutni kanal: 43
Ponovo prebacujem na kanal...
Trenutni kanal: 39
Ponovo prebacujem na kanal...
Trenutni kanal: 53
```

KOMANDA (engl. COMMAND)
Primer 2: Makro komanda


```
/* Implementacija reda pomocu lancane liste */
template <typename T>
class queue{
private:
   list<T> m_list;
public:
   /* Ovaj red je u stvari jednostruko spregnuta lancana lista. Novi
element se upisuje na "rep" liste. Element se uvek uzima sa "glave"
liste */
   /* Iteratori liste su I iteratori reda */
   typedef c_nc_iterator<T> iterator;
   typedef typename iterator::value_type value_type;
   typedef typename iterator::reference reference;
   typedef typename iterator::pointer pointer;

   typedef typename c_nc_iterator<T, true> const_iterator;
   typedef typename const_iterator::reference const_reference;
   typedef typename const_iterator::pointer const_pointer;


. . .
};
```

KOMANDA (engl. COMMAND)
Primer 2: Makro komanda


```
/* Implementacija reda pomocu lancane liste */
template <typename T>
class queue{
private:
   list<T> m_list;
public:
   /*Konstruktori */
   queue() :m_list(){}
   queue(const queue<value_type>& other):m_list(other.m_list){}
   queue<value_type>& operator=(const queue<value_type>& other){
      m_list = other.m_list;
      return *this;
   }
   /* Destruktor */
   ~queue(){}
   /* Funkcije koje omogucavaju pozicioniranje iteratora I njegov prolaz
po listi koja implementira red */
   iterator begin(){ return m_list.begin(); }
   iterator end() { return m_list.end(); }
   const_iterator begin() const { return m_list.begin(); }
   const_iterator end() const { return m_list.begin(); }
```

```
/* Implementacija reda pomocu lancane liste */
template <typename T>
class queue{
private:
   list<T> m_list;
public:
/* Funkcije koje sluze za pregledavanje prvog i poslednje elementa. Bez
izbacivanja iz reda */
   reference front() { return m_list.front(); }
   const_reference front() const { return m_list.front(); }
   reference back() { return m_list.back(); }
   const_reference back() const { return m_list.back(); }
/* FIFO First In First Out : Prvi element koji je upisan u red, je prvi
element koji ce biti izbacen iz reda. Element se u ovoj implementaciji
dodaje na kraj liste a uzima sa pocetka liste */
   void push(const_reference elem){ m_list.push_back(elem); }
   void pop(){ m_list.pop_front(); }

   size_t size() const { return m_list.size(); }
   bool empty() const { return m_list.empty(); }
};
```

KOMANDA (engl. COMMAND)
Primer 2: Makro komanda


```
class KlasaA {
public:
   KlasaA(){ static int sledeci = 0; m_id = sledeci++;  }
   void A_Fun_1() {
      cout << " A_Fun_1  izvrsena nad objektom KlaseA „
                  << m_id << endl;
   }
   void A_Fun_2() {
      cout << " A_Fun_2  izvrsena nad objektom KlaseA"
                  << m_id << endl;
   }
private:
   int m_id;
};
```

KOMANDA (engl. COMMAND)
Primer 2: Makro komanda


```
class KlasaB {
public:
        KlasaB(){ static int sledeci = 0; m_id = sledeci++;  }
        void B_Fun_1() {
                cout << " B_Fun_1  izvrsena nad objektom KlaseB"
                                                << m_id << endl;
        }
        void B_Fun_2() {
                cout << " B_Fun_2  izvrsena nad objektom KlaseB"
                                                << m_id << endl;
        }
private:
        int m_id;
};
```

KOMANDA (engl. COMMAND)
Primer 2: Makro komanda

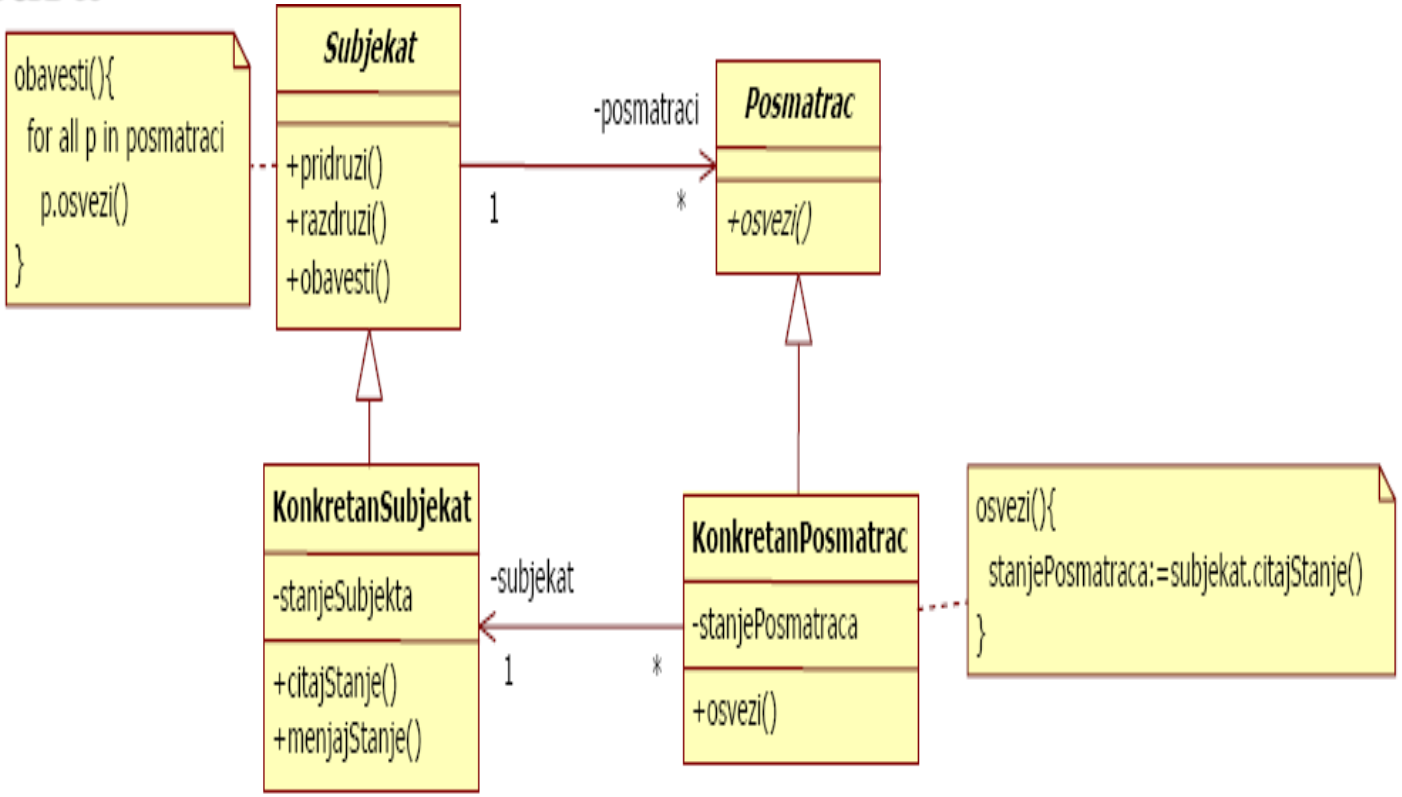

```
/* Interfejs komande */
class IKomanda{
public:
   virtual void execute()=0;
};

/* Sablon koji ce posluziti za instanciranje konkretnih klasa
jednostavnih komandi */
template<typename T>
class Komanda:public IKomanda{
public:
   typedef void(T::*Akcija)();
   Komanda(T *pPrim, Akcija pAk):m_pPrim(pPrim), m_pAkcija(pAk){}
   virtual void execute() { (*m_pPrim.*m_pAkcija)(); }
private:
   SmartPtr<T> m_pPrim;
   Akcija m_pAkcija;
};
```

```
/* Konkretna klasa MakroKomande sastavljene od nekoliko jednostavnih
komandi razlicitog tipa smestenih u red cekanja */
class MakroKomanda:public IKomanda{
   queue< SmartPtr<IKomanda> > qe;
public:
   void Add(IKomanda *p) { qe.push(p);}
   virtual void execute(){
      while(!qe.empty()){ qe.front()->execute(); qe.pop();}
   }
};
/* Execute funkcija nema parametre jer se oni implicitno prenose u vidu
stanja objekta nad kojim komanda treba da se izvrsi */

int main(){
   MakroKomanda m;
   m.Add(new Komanda<KlasaA>(new KlasaA(), &KlasaA::A_Fun_1));
   m.Add(new Komanda<KlasaA>(new KlasaA(), &KlasaA::A_Fun_2));
   m.Add(new Komanda<KlasaB>(new KlasaB(), &KlasaB::B_Fun_1));
   m.execute();
}
```

```
A_Fun_1  izvrsena nad objektom KlaseA 0
A_Fun_2  izvrsena nad objektom KlaseA 1
B_Fun_1  izvrsena nad objektom KlaseB 0
```

- **Ime i klasifikacija**
  - **Posmatrač (engl. Observer)**
  - **Projektni uzorak ponašanja objekata**
- **Drugo ime**
  - **Zavisni objekat, Publikovanje-Pretplata (engl. Dependents, Publish-Subscribe)**
- **Namena**
  - **Definiše 1:N zavisnost između objekata; kada jedan objekat promeni stanje, svi objekti koji su zavisni se obaveste i modifikuju automatski**

Struktura


obavesti(){
for all p in posmatraci
p.osvezi()
}
Subjekat
+pridruzi()
+razdruzi()
+obavesti()
-posmatraci
1
*
Posmatrac
+osvezi()
KonkretanSubjekat
-stanjeSubjekta
+citajStanje()
+menjajStanje()
-subjekat
1
*
KonkretanPosmatrac
-stanjePosmatraca
+osvezi()
osvezi(){
stanjePosmatraca:=subjekat.citajStanje()
}

- **Primenljivost**
  - **Kada jedna apstrakcija ima bar dva medjusobno zavisna dela, tako da promena jednog utiče na promenu drugog**
  - **Kada izmena jednog objekta zahteva izmenu nepoznatog broja objekata**
  - **Kada jedan objekat treba da signalizira promenu drugim objektima ne znajući prorodu (konretan tip) tih objekata**

- **Učesnici**
  - **Subjekat**
    - Zna svoje posmatrače; proizvoljan broj posmatrača može da nadgleda subjekat
    - Obezbeđuje interfejs za pridruživanje i razdruživanje posmatrača
  - **Posmatrač**
    - Definiše interfejs za signaliziranje promena subjekta

- **Učesnici**
  - **KonkretanSubjekat**
    - Čuva stanje od interesa za konkretne posmatrače.
    - Omogućava čitanje stanja
    - Inicira slanje signala posmatračima kada se promeni stanje
  - **KonkretanPosmatrač**
    - Poseduje referencu na konretan subjekat
    - Čuva stanje koje treba da bude konzistentno sa stanjem konkretnog subjekta
    - Implementira operaciju preko koje mu subjekat signalizira promenu
    - Čita stanje konkretnog subjekta da bi ažurirao sopstveno stanje

- **Saradnja**
  - **Konkretan subjekat signalizira svojim posmatračima svaku promenu svog stanja**
  - **Nakon poziva osvezi() konkretan posmatrac traži od subjekta informaciju o stanju**
  - **Posmatrač koristi informaciju o stanju subjekta da ažurira svoje stanje**

- **Posledice**
  - **Prednosti**
    - **Subjekat i posmatrač mogu da budu u različitim slojevima aplikacije**
  - **Nedostaci**
    - **Nepoznata cena promene**
- **Povezani uzorci**
  - **Za inkapsuliranje kompleksne semantike ažuriranja Posrednih može da posreduje između subjekata i posmatača**

```
class Posmatrani;
/* Interfejs posmatraca */
class IPosmatrac{
public:
        virtual ~IPosmatrac(){}
        virtual void Obavesti(Posmatrani* ptrPosmatrani) = 0;
protected:
        IPosmatrac(){}
private:
        IPosmatrac(const IPosmatrac&);
        IPosmatrac& operator=(const IPosmatrac&);
};
```

```
class Posmatrani{
public:
        virtual ~Posmatrani(){}
        bool DodajPosmatraca(IPosmatrac* ptrPosmatrac);
        bool UkloniPosmatraca(IPosmatrac* ptrPosmatrac);
        bool ObavestiPosmatrace();
        virtual string GetStatus() const = 0;
protected:
        //Ova klasa sluzi samo kao koren hijerarhije
        Posmatrani():m_listaPosmatraca(){}
private:
        list<IPosmatrac*> m_listaPosmatraca;
        Posmatrani(const Posmatrani&);

        Posmatrani& operator=(const Posmatrani&);
};
```

```
//Dodaje ptrPosmatrac u listu posmatraca
bool Posmatrani::DodajPosmatraca( IPosmatrac* ptrPosmatrac )
{
	list<IPosmatrac*>::iterator temp =
find(m_listaPosmatraca.begin(), m_listaPosmatraca.end(), ptrPosmatrac);
	if ( temp != m_listaPosmatraca.end() )
		return false;
	m_listaPosmatraca.push_back(ptrPosmatrac);
	return true;
}

//Uklanja ptrPosmatrac iz liste posmatraca
bool Posmatrani::UkloniPosmatraca( IPosmatrac* ptrPosmatrac )
{
	list<IPosmatrac*>::iterator temp =
find(m_listaPosmatraca.begin(), m_listaPosmatraca.end(), ptrPosmatrac);
	if ( temp == m_listaPosmatraca.end() )
		return false;
	else
		m_listaPosmatraca.erase(temp);
	return true;
```

```
/* Ova funkcija implementira kljucno ponasanje uzorka */
bool Posmatrani::ObavestiPosmatrace(){
        for (list<IPosmatrac*>::iterator it = m_listaPosmatraca.begin();
                                        it != m_listaPosmatraca.end();++it)
                (*it)->Obavesti(this);
        return (m_listaPosmatraca.size() > 0);
}
/* KONRETNA KLASA POSMATRANOG OBJETKA */
class Nakit : public Posmatrani{
public:
        Nakit():m_status("OK"){}
        virtual ~Nakit(){}
        void Kradja(){ m_status = "KRADJA";}
        string GetStatus() const { return m_status; }
private:
        string m_status;
        Nakit(const Nakit& yRef);
        Nakit& operator=(const Nakit& yRef);
};
```

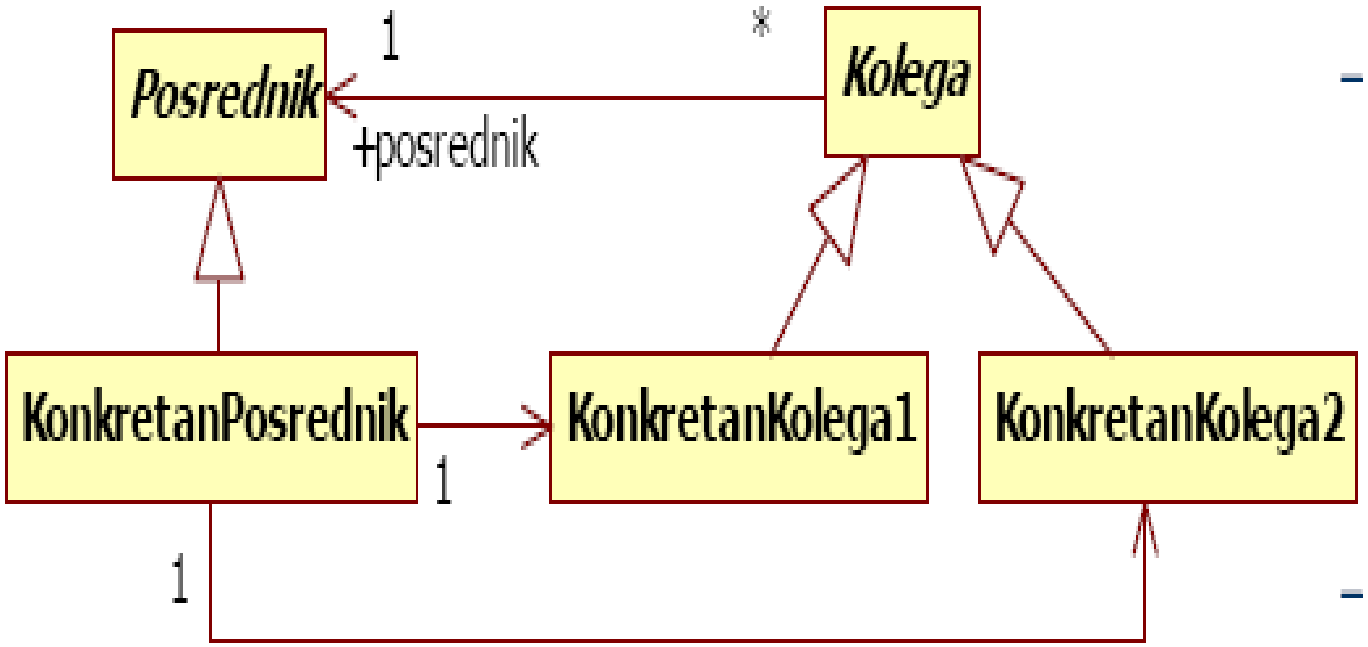
```
class SigurnosnaVrata : public IPosmatrac{
public:
        SigurnosnaVrata(): m_status("OTVORENA"){}
        virtual ~SigurnosnaVrata(){}
        void Print() const { cout << "Vrata: "
                                  << m_status.c_str() <<endl; }
        virtual void Obavesti(Posmatrani* ptrPosmatrani) {
/* Nakon poziva Obavesti() konkretan posmatrac traži od subjekta
informaciju o stanju. Posmatrač koristi informaciju o stanju subjekta da
ažurira svoje stanje. */
                if (ptrPosmatrani->GetStatus() == "OK")
                        if (m_status = "ZATVORENA"){/* otvori vrata */}
                        m_status = "OTVORENA";
                else    m_status = "ZATVORENA";
                Print();
        }
private:
        string m_status;
        SigurnosnaVrata(const SigurnosnaVrata&);
        SigurnosnaVrata& operator=(const SigurnosnaVrata&);
};
```

```
class Cuvar : public IPosmatrac {
public:
        Cuvar(string ime) :m_ime(ime), m_status("CUVA OBJEKAT"){}
        virtual ~Cuvar(){}

        void Print() const { cout << "Cuvar: "<< m_ime.c_str() <<" "
                                  << m_status.c_str() << endl; }

        virtual void Obavesti(Posmatrani* ptrPosmatrani){
                if (ptrPosmatrani->GetStatus() == "OK")
                        m_status = "CUVA OBJEKAT";
                else
                        m_status = "JURI ULJEZA";
                Print();
        }
private:
        string m_ime, m_status;
        Cuvar(const Cuvar&);
        Cuvar& operator=(const Cuvar&);
};
```

```
/* KLIJENT */
int _tmain(int argc, _TCHAR* argv[]){

        Cuvar* ptrCuvar1 = new Cuvar("Petar Mrkonjic");
        Cuvar* ptrCuvar2 = new Cuvar("Filip Jaksic");
        SigurnosnaVrata* ptrVrata = new SigurnosnaVrata();
        Nakit* ptrNakit = new Nakit();

        ptrNakit->DodajPosmatraca(ptrCuvar1);
        ptrNakit->DodajPosmatraca(ptrCuvar2);
        ptrNakit->DodajPosmatraca(ptrVrata);

        ptrNakit->ObavestiPosmatrace();

        ptrNakit->Kradja();

        ptrNakit->ObavestiPosmatrace();


        return 0;
}
```

- **Ime i klasifikacija**
  - **Posrednik (engl. Mediator)**
  - **Projektni uzorak ponašanja objekata**
- **Namena**
  - **Definiše objekat koji inkapsulira interakciju skupa objekata**
  - **Omogućava slabo sprezanje skupa objekata i omogućava da se njihova interakcija menja nezavisno**

Struktura


Posrednik
1
*
Kolega
+posrednik
KonkretanPosrednik
1
KonkretanKolega1
KonkretanKolega2
1

- **Primenljivost**
  - **Kada skup objekata komunicira na dobro definisan ali složen način – medjuzavisnosti nisu struktuirane**
  - **Kada objekti referenciraju i komuniciraju sa mnogim drugim objektima**
  - **Kada treba omogućiti prilagođavanje ponašanja distribuiranog na više klasa, a da se izbegne mnogo potklasa**

- **Učesnici**
  - **Posrednik**
    - Definiše interfejs za komunikaciju sa objektima klase Saradnik
  - **KonkretanPosrednik**
    - Referencira objekte klase Saradnik
    - Koordiniše komunikaciju izmedju objekata klase Saradnik
  - **Saradnik**
    - Interfejs objekata koji međusobno sarađuju
    - Sadrži referencu posrednika
  - **KonkretanSaradnik**
    - Implementira interfejs saradnika

- **Saradnja**
  - **Saradnici**
    - Šalju i primaju zahteve od objekata posrednika
  - **Posrednik**
    - Implementira kooperaciju, rutiranjem zahteva između odgovarajućih kolega

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
/* Interfejs posrednika */
class IPosrednik
{
public:
        virtual void PosaljiPoruku(unsigned IDSaradnika, const string&
message) = 0;
        virtual ~IPosrednik(){}
};
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
/* Klasa objekata koji ce komunicirati preko posrednika */
class Saradnik{
        IPosrednik* m_ptrPosrednik;
        unsigned m_ID;
        string m_ime;
public:
        Saradnik(unsigned ID, string ime) : m_ptrPosrednik(NULL),
                                            m_ID(ID), m_ime(ime){}
        unsigned getID() const { return m_ID; }
/* Funkcija kojom se objekat vezuje za posrednika */
        void RegistrujPosrednika(IPosrednik &p) { m_ptrPosrednik = &p; }

/* Akcije koje mogu da se izvrsavaju nad objektima klase Saradnik */
        void PrimiPoruku(const string &poruka){
                cout << "Poruku primio "<< m_ID<<": "
                                        <<poruka.c_str()<< endl;
        }
        void PosaljiPoruku(unsigned ID, const string &poruka){
                cout << m_ID << " Posalji poruku objektu " << ID
                                    << ": " << poruka.c_str() << endl;
                m_ptrPosrednik->PosaljiPoruku(ID, poruka);
        }
};
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
/* Konkretna klasa posrednika */
class Posrednik : public IPosrednik{
        Saradnik **ptrSaradnik;
public:
        Posrednik(unsigned num) :ptrSaradnik(new Saradnik*[num]){
                for (unsigned Idx = 0; Idx < num; ++Idx)
                        ptrSaradnik[Idx] = NULL;
        }
/* Funkcija kojom se posrednik vezuje za saradnika */
        void RegistrujSaradnika(Saradnik& s){
                if (!ptrSaradnik[s.getID()]){
                        s.RegistrujPosrednika(*this);
                        ptrSaradnik[s.getID()] = &s;
                }
        }
/* Funkcija kojom se implementira komunikacija sa objektima izmedju
kojih posredje */
        void PosaljiPoruku(unsigned IDSaradnika, const string& poruka){
                if (ptrSaradnik[IDSaradnika])
                        ptrSaradnik[IDSaradnika]->PrimiPoruku(poruka);
        }
        virtual ~Posrednik(){ delete []ptrSaradnik; }
};
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
/* KLIJENT KOJI KORISTI POSREDNIKA */
/* "Komunikacija" izmedju objekata je ostvarena preko posrednika */
int main()
{
        Saradnik mitar(0,"Mitar");
        Saradnik joksim(1,"Joksim");
        Saradnik svetlana(2,"Svetlana");

        Posrednik posrednik(3);
        posrednik.RegistrujSaradnika(mitar);
        posrednik.RegistrujSaradnika(joksim);
        posrednik.RegistrujSaradnika(svetlana);

        mitar.PosaljiPoruku(1, "PORUKA_1");
        joksim.PosaljiPoruku(2, "PORUKA_2");
        svetlana.PosaljiPoruku(0, "PORUKA_3");
        joksim.PosaljiPoruku(0, "PORUKA_4");

        return 0;
}
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
class Ucesnik;

class ApstraktnaPricaonica{
public:
    virtual void RegistrujUcesnika(Ucesnik* ptrUcesnik) = 0;
    virtual void Posalji(const std::string& od, const std::string& ka,
const std::string& poruka) = 0;
};

class Pricaonica : public ApstraktnaPricaonica{
private:
    std::unordered_map<std::string,Ucesnik*> m_Ucesnici;

public:
    void RegistrujUcesnika(Ucesnik* ptrUcesnik);
    void Posalji(const std::string& od, const std::string& ka, const
std::string& poruka);
};
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
class Ucesnik{
private:
    Pricaonica* m_pPricaonica;
    std::string m_Ime;

public:
    Ucesnik(const std::string& name): m_Ime(name) {}

    const std::string& GetIme()const { return m_Ime; }
    const Pricaonica* GetPricaonica()const { return m_pPricaonica; }
    void SetPricaonica(Pricaonica* ptrPricaonica) { m_pPricaonica =
ptrPricaonica; }

    virtual void Posalji(const std::string& ka, const std::string&
poruka) {
        m_pPricaonica->Posalji(m_Ime, ka, poruka);
    }

    virtual void Primi( const std::string& od, const std::string&
poruka) {
        std::cout << od << " ka " << GetIme() << ": " << poruka <<
std::endl;
    }
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
void Pricaonica::RegistrujUcesnika(Ucesnik* ptrUcesnik){
    std::unordered_map<std::string,Ucesnik*>::iterator it =
            m_Ucesnici.find(ptrUcesnik->GetIme());
    if( it == m_Ucesnici.end() )
        m_Ucesnici[ptrUcesnik->GetIme()] = ptrUcesnik;

    ptrUcesnik->SetPricaonica(this);
}

void Pricaonica::Posalji(const std::string& od, const std::string& ka,
const std::string& poruka){
    Ucesnik* ptrUcesnik = m_Ucesnici[ka];
    if (ptrUcesnik != NULL) ptrUcesnik->Primi(od, poruka);
}
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
class RealniUcesnik : public Ucesnik{
public:
    RealniUcesnik(const std::string& name) : Ucesnik(name) {}
    void Primi(const std::string& od, const std::string& poruka)
    {
        std::cout << "Poruku primio realni ucesnik";
        Ucesnik::Primi(od, poruka);
    }
};

class Bot : public Ucesnik{
public:
    Bot(const std::string& name) : Ucesnik(name) {}

    void Primi(const std::string& od, const std::string& poruka)
    {
        std::cout << "Poruku primio ";
        Ucesnik::Primi(od, poruka);
    }
};
```

POSREDNIK (engl. MEDIATOR)
Primer 1:


```
int main()
{
    Pricaonica* ptrPricaonica = new Pricaonica();

    Ucesnik* pPetar = new RealniUcesnik("Petar");
    Ucesnik* pMika = new RealniUcesnik("Mika");
    Ucesnik* pJovana = new RealniUcesnik("Jovana");
    Ucesnik* pPanta = new Bot("Pantelija");

    ptrPricaonica->RegistrujUcesnika(pPetar);
    ptrPricaonica->RegistrujUcesnika(pMika);
    ptrPricaonica->RegistrujUcesnika(pJovana);
    ptrPricaonica->RegistrujUcesnika(pPanta);

    pPetar->Posalji("Mika", "Za Miku");
    pPetar->Posalji("Pantelija", "Za Panteliju");
    pJovana->Posalji("Petar", "Za Petra");
    pMika->Posalji("Mika", "Za Miku");
    pPanta->Posalji("Jovana", "Za Jovanu");

        return 0;
}
```