

Objektno orijentisano programiranje u C++-u

Projektni uzorci

1

STRUKTURNI PROJEKTNI UZORCI

FASADA
ZASTUPNIK
MUVA

Ime i klasifikacija

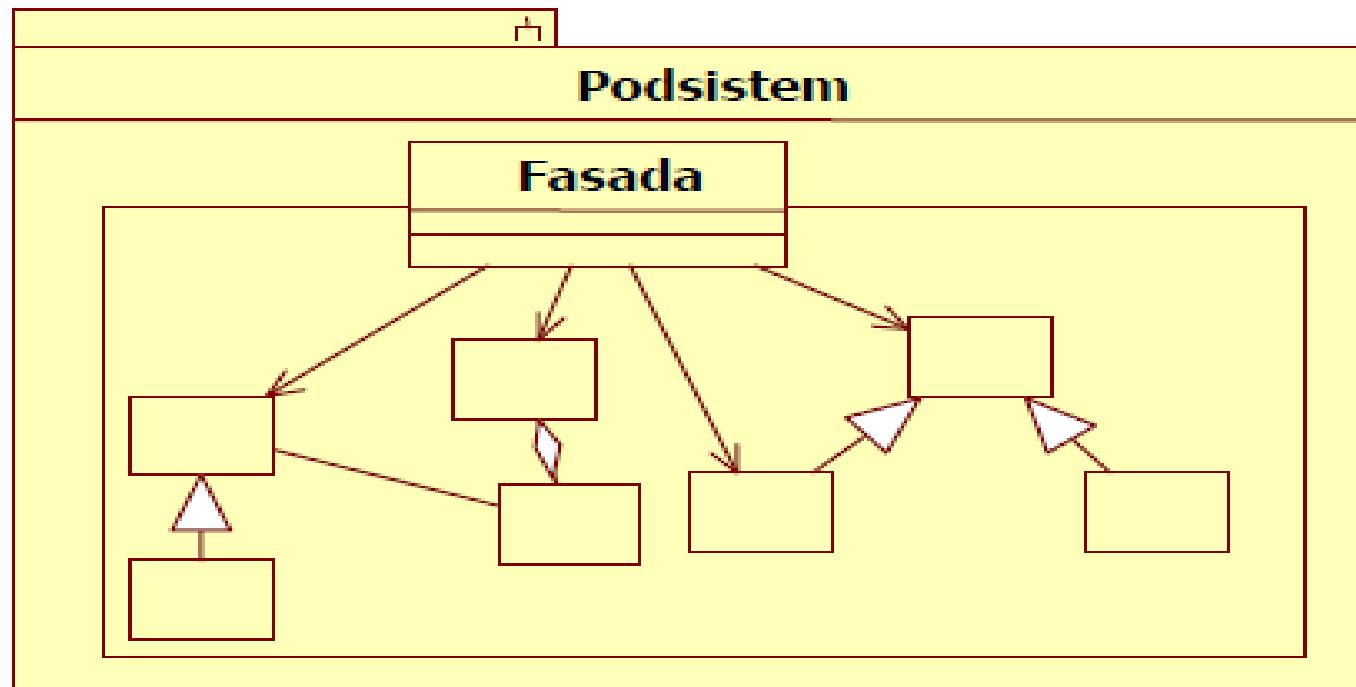
- Fasada (engl. Facade)**
- Strukturni projektni uzorak**

Drugo ime

Namena

- Obezbeđivanje jedinstvenog interfejsa podsistema**
- Definisanje jedinstvenog interfejsa višeg nivoa da bi se podsistemi lakše koristili**

❑ Struktura



Učesnici

Fasada

- Zna koje klase podistema su odgovorne za koji zahtev
- Delegira zahteve klijenata odgovarajućim objektima podistema

Klase podistema

- Implementiraju funkcionalnosti
- Izvršavaju zahteve fasade
- Ne znaju ništa o fasadi

❑ Saradnja

- ❑ **Klijenti šalju zahteve fasadi**
- ❑ **Fasada prosleđuje zahteve objektima podsistema**

❑ Posledice

- ❑ **Smanjivanje broja objekata od kojih klijenti zavise**
 - ❑ Olakšava korišćenje podsistema
- ❑ **Slabo vezivanje između podsistema i klijenata**
 - ❑ Olakšava održavanje podsistema. Promena klase podsistema ne utiče na klijenta.
Promena u podсистему не захтева ни поново преводјење клијента.

❑ Povezani uzorci

- ❑ **Apstraktna fabrika predstavlja Fasadu za kreiranje objekata podsistema**
- ❑ **Ako je potreban samo jedan objekat Fasade ona se realizuje kao Unikat.**

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

6

```
// tacka.h
#define _USE_MATH_DEFINES
#include <math.h>

class PolarnaTacka; // Deklarisanje unapred

class DeKartovaTacka{ //PODSISTEM 1
    double m_x,m_y;
public:
    DeKartovaTacka(double x, double y):m_x(x),m_y(y){}
    DeKartovaTacka(const DeKartovaTacka& other) :
        m_x(other.m_x), m_y(other.m_y){}
    DeKartovaTacka& pomeri(double dx, double dy)
        { m_x += dx; m_y += dy; return *this;}
/*
 * Koristi klasu PolarnaTacka i zato mora
 * da bude naknadno definisana */
    DeKartovaTacka& rotiraj(double theta);

    . . .
};
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

7

```
/* tacka.h */
#define _USE_MATH_DEFINES
#include <math.h>

class PolarnaTacka; /* Deklarisanje unapred */

class DeKartovaTacka{ /* PODSISTEM 1 */
    double m_x,m_y;
public:
    . . .

    /* operator konverzije */
    operator PolarnaTacka() const;
    double getX() const { return m_x; }
    double getY() const { return m_y; }
};
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

8

```
/* tacka.h */
class PolarnaTacka{ /* PODSISTEM 2 */
    double m_r, m_alpha;
public:
    /* double fmod (double deljenik, double delilac);
       fmod = deljenik - K * delilac;
       gde je K celobrojni rezultat deljenja deljenik/delilac
       zaokruzen na manji ceo broj. K = floor(deljenik/dellilac) */
    PolarnaTacka(double r, double alpha) : m_r(r),
                  m_alpha( fmod(alpha,360) ){}
    PolarnaTacka(const PolarnaTacka& other) : m_r(other.m_r),
                                                m_alpha(other.m_alpha){}
    ...
};
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

9

```
/* tacka.h */
class PolarnaTacka{ /* PODSISTEM 2 */
    double m_r, m_alpha;
public:
    /* theta je ugao u stepenima */
    PolarnaTacka& rotiraj(double theta) {
        m_alpha = fmod(m_alpha + theta, 360);
        return *this;
    }

    /* Pre pomeranja se konvertuje u tacku
    u Dekartovom koordinatnom sistemu */
    PolarnaTacka& pomeri(int dx, int dy);
    operator DeKartovaTacka() const;
};

/* tacka.cpp */
/* atan2(y,x) = arctg(y/x) - rezultat je u radijanima*/
DeKartovaTacka::operator PolarnaTacka() const {
    return PolarnaTacka(sqrt(m_x*m_x+m_y*m_y),
                        atan2(m_y,m_x)*180/M_PI);
}
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

10

/* tacka.cpp */

```
PolarnaTacka::operator DeKartovaTacka() const {
    return DeKartovaTacka(m_r*cos( m_alpha*M_PI/180) ,
                           m_r*sin(m_alpha*M_PI/180));
}
```

/* Zasto je u sledecoj funkciji moguce konstruisanje DeKartove tacke kopiranjem polarne tacke?

Zasto je moguce DeKartovu tacku dodeliti Polarnoj tacki? */

```
PolarnaTacka& PolarnaTacka::pomeri(int dx, int dy) {
    return *this = DeKartovaTacka(*this).pomeri(dx,dy);
}
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

11

```
/* tacka.cpp */
/* atan2(y,x) = arctg(y/x) - rezultat je u radijanima*/
DeKartovaTacka::operator PolarnaTacka() const {
    return PolarnaTacka(sqrt(m_x*m_x+m_y*m_y),
                         atan2(m_y,m_x)*180/M_PI);
}

/* Zasto je u sledecoj funkciji moguce konstruisanje polarne tacke
kopiranjem DeKartove tacke?
Zasto je moguce polarnu tacku dodeliti DeKartovoj tacki? */

DeKartovaTacka& DeKartovaTacka::rotiraj(double theta) {
    return *this = PolarnaTacka(*this).rotiraj(theta);
}
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

12

```
/* tacka.h */
class Tacka{ /* FASADA 1 */
/* Pojednostavljen interfejs u odnosu na komponente od kojih je
sacinjena. Vidljiva komponeta je DeKartovaTacka a nevidljiva
komponenta koja ucestvuje u izracunavanju je Polarna tacka */
DeKartovaTacka a;
public:
    Tacka(double x, double y):a(x,y){}
    Tacka(const Tacka& t):a(t.a){}
/* Polarnu tacku koristimo da inicijализујемо Dekartovu tacku. Zasto
je to moguce? */
    Tacka(const PolarnaTacka& p):a(p){}
    void pomeri(double dx, double dy) {a.pomeri(dx,dy);}
    void rotiraj(double theta){ a.rotiraj(theta);}

    void rotiraj(double theta, const Tacka& centar){
        a.rotiraj(theta, centar.a);
    }

    double getX() const { return a.getX(); }
    double getY() const { return a.getY(); }
};

Projektni uzorci
27.11.2018.
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

13

```
/* duz.h */
class Duz { /* FASADA 2 */
/* Jos jedan jednostavan interfejs */
private:
    Tacka poc, kraj;
public:
    Duz(Tacka p, Tacka k) :poc(p), kraj(k) {}

    void pomeri(double dx, double dy) {
        poc.pomeri(dx, dy);
        kraj.pomeri(dx, dy);
    }
/* Rotacija oko koordinatnog pocetka */
    void rotiraj(double theta) {
        poc.rotiraj(theta);
        kraj.rotiraj(theta);
    }
    Tacka getPoc() const { return poc; }
    Tacka getKraj() const { return kraj; }
};
```

FASADA (engl. FACADE)

Primer 1: Implementacija duzi u Dekartovom koordinatnom sistemu

14

```
/* duz.h */
/* KLIJENT koji koristi interfejs FASADE 1 I FASADE 2 */
int _tmain(int argc, _TCHAR* argv[])
{
    Tacka a(10,10);
    a.pomeri(5,5);
    a.rotiraj(-90);

    Duz d(Tacka(5, 5), Tacka(10, 10));
    d.rotiraj(45);

    cout << d.getPoc().getX() << " " << d.getPoc().getY() << endl;
    cout << d.getKraj().getX() << " " << d.getKraj().getY() << endl;

    return 0;
}
```

❑ Ime i klasifikacija

- ❑ Zastupnik (engl. Proxy)
- ❑ Strukturni projektni uzorak

❑ Namena

- ❑ Predstavlja surogat drugog objekta, koji:
 - ❑ kontroliše pristup orginalnom objektu,
 - ❑ odlaže kreiranje i inicijalizaciju objekta do trenutka kada je ovaj zaista potreban

❑ Drugo ime

- ❑ Predstavnik, Zamena, Surogat (engl. Surrogate)

❑ Vrste zastupnika prema primeni

❑ **Zastupnik udaljenog objekta (Remote Proxy)**

- ❑ Predstavlja lokalnog predstavnika objekta koji se nalazi u drugom adresnom prostoru. Ovakav Zastupnik se naziva Ambasador

❑ **Virtuelni Zastupnik (Virtual Proxy)**

- ❑ Kreira objekat na zahtev(onda kada je objekat zaista potreban)

❑ **Zaštitni Zastupnik (Protection Proxy)**

- ❑ Kontroliše pravo pristupa originalnom objektu

❑ **Pametni pokazivac (Smart Reference)**

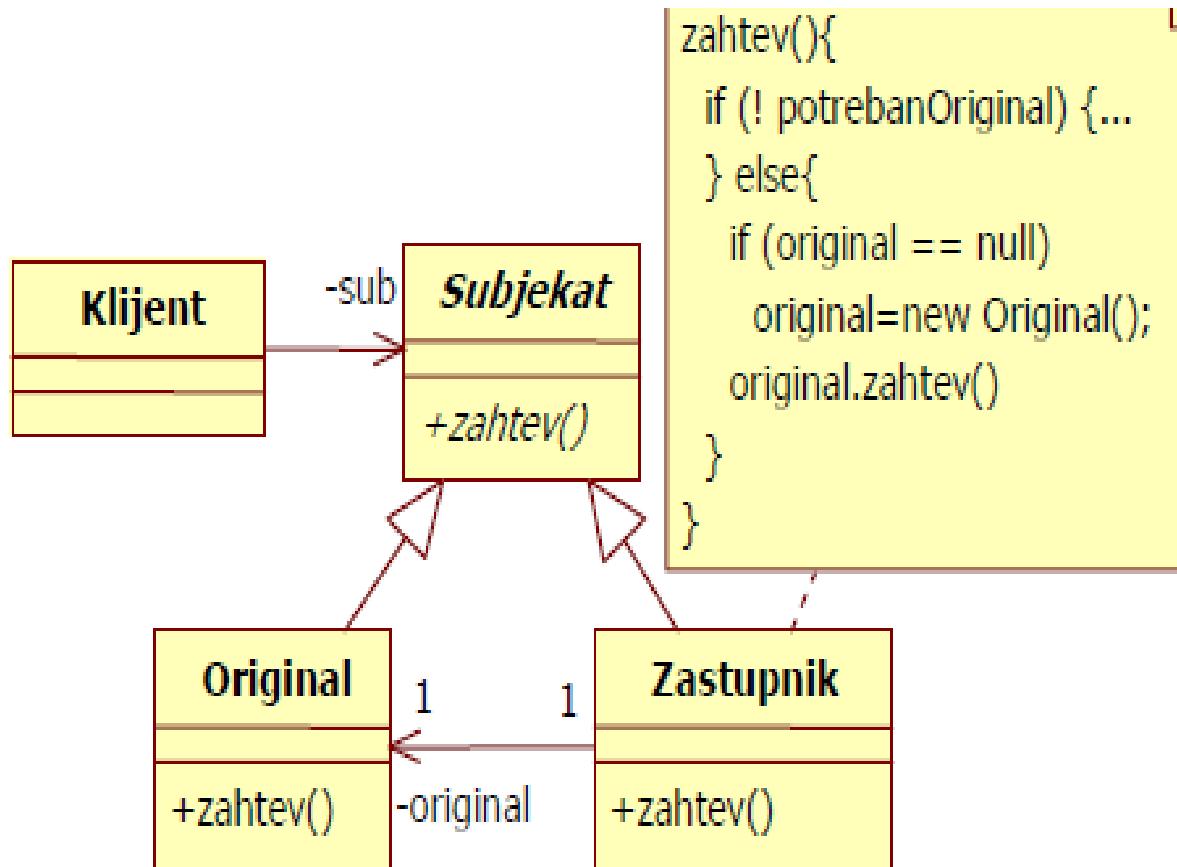
- ❑ Zamena za obični pokazivač, obavlja dodatne akcije prilikom pristupa

ZASTUPNIK (engl. PROXY)

1. Uvod

17

□ Struktura



❑ Učesnici

❑ Subjekt

- ❑ Zajednički interfejs za Original i Zastupnika, omogućava da se Zastupnik koristi kao Original

❑ Zastupnik

- ❑ Čuva referencu preko koje pristupa orginalu
- ❑ Implementira interfejs Subjekta, tako da može da predstavlja zamenu Originala
- ❑ Kontroliše pristup originalu, može da bude odgovoran za kreiranje uništavanje originala

❑ Original

- ❑ Realni subjekt predstavljen Zastupnikom

❑ Saradnja

- ❑ **Zastupnik prosleđuje zahteve objektu Originala (u zavisnosti od vrste zastupnika)**

❑ Posledice

- ❑ **Uvodi dodatni nivo indirekciju u pristupu objektu**
- ❑ **Optimizacija koju Zastupnik može da sakrije od klijenta je copy-on-write**
 - ❑ Kada klijent zahteva operaciju koja modificuje original, zastupnik stvarno kopira original i dekrementira broj referencija koje ukazuju na original.
 - ❑ Ako broj referenci padne na 0, original se uništava.

ZASTUPNIK (engl. PROXY)

Primer 1. Odlozeno izracunavanje

20

```
/* Sablon zastupnika */
template<typename LeviTip,
          typename DesniTip>
class PlusOp {
    const LeviTip& levi;
    const DesniTip& desni;
public:
    PlusOp(const LeviTip& le, const DesniTip& de) :
        levi(le), desni(de) {}

    const LeviTip& getLevi() const { return levi; }
    const DesniTip& getDesni() const { return desni; }
    int getX() const { return levi.getX() + desni.getX(); }
    int getY() const { return levi.getY() + desni.getY(); }
};
```

ZASTUPNIK (engl. PROXY)

Primer 1. Odlozeno izracunavanje

21

```
class Tacka {
    int x, y;
public:
    Tacka(int xi = 0, int yi = 0):x(xi), y(yi) { }
/* Pored standardnog konstruktora kopije i operatora dodele, dodajemo
sablone konstruktor kopije i operatora dodele vrednosti koji prihvataju
proxy, tj. operaciju kao parametar. */
template<typename LeviTip, typename DesniTip>
Tacka(const PlusOp<LeviTip, DesniTip>& op) {
    x = op.getX();
    y = op.getY();
}
template<typename LeviTip, typename DesniTip>
Tacka& operator=(const PlusOp<LeviTip, DesniTip>& op) {
    x = op.getX();
    y = op.getY();
    return *this;
}
int getX() const { return x; }
int getY() const { return y; }
};
```

ZASTUPNIK (engl. PROXY)

Primer 1. Odlozeno izracunavanje

22

/* Preklapanje operatora sabiranja sablona operacije sa tackom na desnoj strani operatora. Rezultat je sablon operacije */

```
template<typename LeviTip, typename DesniTip>
PlusOp<PlusOp<LeviTip, DesniTip>, Tacka>
operator+(const PlusOp<LeviTip, DesniTip>& levi, const Tacka& p) {
    return PlusOp< PlusOp<LeviTip, DesniTip>, Tacka >(levi, p);
}
```

/* Preklapanje operatora sabiranja sablona operacije sa tackom na levoj strani operatora. Rezultat je sablon operacije */

```
template<typename LeviTip, typename DesniTip>
PlusOp< Tacka, PlusOp<LeviTip, DesniTip> >
operator+(const Tacka& p, const PlusOp<LeviTip, DesniTip>& desni) {
    return PlusOp< Tacka, PlusOp<LeviTip, DesniTip> >(p, desni);
}
```

ZASTUPNIK (engl. PROXY)

Primer 1. Odlozeno izracunavanje

23

```
/* Preklapanje operatora sabiranja dva sablona operacije. Rezultat je  
sablon operacije */
```

```
template < typename LLeviTip, typename LDesniTip,  
          typename DLeviTip, typename DDesniTip>
```

```
PlusOp< PlusOp<LLeviTip, LDesniTip>, PlusOp<DLeviTip, DDesniTip> >
```

```
operator+( const PlusOp<LLeviTip, LDesniTip>& levi,  
            const PlusOp<DLeviTip, DDesniTip>& desni) {
```

```
    return PlusOp< PlusOp<LLeviTip, LDesniTip>,  
                  PlusOp<DLeviTip, DDesniTip> >(levi, desni);
```

```
}
```

```
/* Preklapanje operatora sabiranja dve tacke. Rezultat je sablon  
operacije */
```

```
PlusOp< Tacka, Tacka >
```

```
operator+(const Tacka& levi, const Tacka& desni) {  
    return PlusOp<Tacka, Tacka>(levi, desni);
```

```
}
```

ZASTUPNIK (engl. PROXY)

Primer 1. Odlozeno izracunavanje

24

```
int main() {  
  
    Tacka t1(1,2), t2(3,4), t3(5, 6);  
    Tacka t = (t1 + t2) + (t3 + t1);  
    cout<<t.getX()<<t.getY();  
    return 0;  
}
```

ZASTUPNIK (engl. PROXY)

Primer 2. Odlozeno kreiranje objekta

25

```
template <typename TipServisa>
class VirtuelniZastupnik{
public:
    VirtuelniZastupnik() : ptrToObj(NULL), kreiran(false) {}
    TipServisa &operator*() { kreiraj(); return *ptrToObj; }
    TipServisa *operator->() { kreiraj(); return ptrToObj; }
    ~VirtuelniZastupnik() { delete ptrToObj; }

private:
/* funkcija kreiraj moze da ima parametre kao i konstruktor */
    void kreiraj(){
        if (!kreiran){
            ptrToObj = new TipServisa;
            kreiran = true;
        }
    }
};
```

ZASTUPNIK (engl. PROXY)

Primer 2. Odlozeno kreiranje objekta

26

```
template <typename TipServisa>
class VirtuelniZastupnik{
public:
    VirtuelniZastupnik() : ptrToObj(NULL), kreiran(false) {}
    TipServisa &operator*() { kreiraj(); return *ptrToObj; }
    TipServisa *operator->() { kreiraj(); return ptrToObj; }
    ~VirtuelniZastupnik() { delete ptrToObj; }

    /* Zabranjeno je kopiranje Virtuelnog Zastupnika: Deleted funkcije
    od verzije standarda C++ 11 */
    VirtuelniZastupnik(const VirtuelniZastupnik&) = delete;
    VirtuelniZastupnik& operator=(const VirtuelniZastupnik&) = delete;

private:
    TipServisa* ptrToObj; /* Pokazivac zastupljenog objekta */
    bool kreiran; /* Indikator kreiranog objekta */
};
```

ZASTUPNIK (engl. PROXY)

Primer 2. Odlozeno kreiranje objekta

27

```
class ZastupljeniServis{
public:
    ZastupljeniServis(int pod = 0) : podS(pod){
        std::cout << "Konstruktor: ZastupljeniServis(" << podS << ")" << std::endl;
    }

    ZastupljeniServis(const ZastupljeniServis &drugi) : podS(drugi.podS) {
        std::cout << "Konstruktor Kopije:
                    ZastupljeniServis(const ZastupljeniServis &)" << std::endl;
    }

    ~ZastupljeniServis() {
        std::cout << "Destruktor:~ZastupljeniServis(" << podS << ")" << std::endl;
    }

    void Servisiraj(){
        std::cout << "Servisiraj(" << podS << ")" << std::endl;
    }

private:
    int podS;
};
```

ZASTUPNIK (engl. PROXY)

Primer 2. Odlozeno kreiranje objekta

28

```
int main()
{
    VirtuelniZastupnik<ZastupljeniServis> n;

    std::cout << "Pokrecem servis" << std::endl;

    n->Servisiraj();
    (*n).Servisiraj();
}
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

29

```
template <class T>
class SmartPtr{
    T* ptrToData; /* Pokazivac na podatak */
    unsigned* ptrToCount; /* Pokazivac na brojac reference */

    void Detach() {
        /* Dekrementiramo brojac referenci.
           Ako je njegova vrednost jednaka 0 brisemo
           objekat i brojac referenci */
        if (!--*ptrToCount) {
            delete ptrToCount;
            delete ptrToData;
        }
    }

public:
    T *GetPtrToData() const { return ptrToData; }
    unsigned *GetPtrToCount() const { return ptrToCount; }
    ...
};
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

30

```
template <class T>
class SmartPtr{
public:
    /* Ovaj konstruktor ne bi smeо da bude javni.
       Javni je samo zato sto unutar funkcije operatora konverzije
       klase SmartPtr<T> pozivam konstruktor klase SmartPtr<U> */
    explicit SmartPtr(T* pD, unsigned *pC) :
        ptrToData(pD), ptrToCount(pC) {}

    /* Ovaj konstruktor postoji kako bi smo omogucili
       kreiranje nizova pametnih pokazivaca */
    explicit SmartPtr() : ptrToData(0), ptrToCount(new unsigned(0)) {}

    /* Ovaj konstruktor nije eksplicitan samo da bi smo dozvolili
       sledeca kreiranja objekata klase SmartPtr<T>:
       SmartPtr<T> p = new T(); */
    SmartPtr(T* pD) : ptrToData(pD), ptrToCount(new unsigned(1)) {}

    ...
};
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

31

```
template <typename T>
class SmartPtr{
public:
    /* Konstruktor kopije */
    template<typename U>
    SmartPtr(const SmartPtr<U>& other) :
        ptrToData(other.GetPtrToData()), ptrToCount(other.GetPtrToCount()) {
        /* Inkrementiramo brojac "referenci".
           Jos jedan pokazivac ukazuje na objekat. */
        ++*ptrToCount;
    }
    /* Operator dodele vrednosti */
    template<typename U>
    SmartPtr<T>& operator = (const SmartPtr<U>& other) {
        if (this != &other) {
            Detach(); /* this vise ne ukazuje na stari objekat */
            ptrToData = other.GetPtrToData();
            ++*(ptrToCount = other.GetPtrToCount());
        }
        return *this;
    }
}
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

32

```
template <class T>
class SmartPtr{
public:
    /* Operator implicitne konverzije predstavlja alternativu sablonima
    konstruktora kopije i operatora dodele. Posto je implicitan, da bi se
    izbegle nezeljene konverzije, obicno se izbegava u implementaciji */
    template<typename U>
    operator SmartPtr<U>() const {
        ++*ptrToCount;
        return SmartPtr<U>(ptrToDate, ptrToCount);
    }

    ~SmartPtr() { Detach(); }

    T* operator->() { return ptrToDate; }
    T& operator*() { return *ptrToDate; }

    const T* operator->() const { return ptrToDate; }
    const T& operator*() const { return *ptrToDate; }
};
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

33

```
class A{
protected:
    unsigned ID;
public:
    A(unsigned id) :ID(id) {
        cout << "A Konstruktor objekta:" << ID << endl;
    }
    virtual ~A() {
        cout << "A Destruktor objekta:" << ID << endl;
    }
    virtual void Fun() const { cout << "Fun objekta:" << ID << endl; }
};

class B:public A{
public:
    B(unsigned id) :A(id) {
        cout << "B Konstruktor objekta:" << ID << endl;
    }
    virtual ~B() { cout << "B Destruktor objekta:" << ID << endl; }
};
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

34

/* Klijent */

```
int main(int argc, char* argv[]) {
    SmartPtr<A> p = new A(1);

    {
        p = new A(2);
    }

    /* Izlaz ?

    */
    return 0;
}
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

35

```
/* Klijent */

int main(int argc, char* argv[]){

    SmartPtr<A> p = new A(1);

    {

        p = new A(2);
    }

    /* Izlaz
    A Konstruktor objekta: 1
    A Konstruktor objekta: 2
    A Destruktor objekta:1  ->      Zasto se prvo unistava objekat 1 a
zatim objekat 2?
    A Destruktor objekta:2
    */

    return 0;
}
```

ZASTUPNIK (engl. PROXY)

Primer 3. Pametni pokazivac

36

```
/* KLIJENT */

int _tmain(int argc, _TCHAR* argv[])
{
    SmartPtr<B> pi = new B(3);
    SmartPtr<A> po = pi; /* Linija koda koja se kompajlira sablonu
konstruktora kopije */
    po->Fun(); /* Dinamicko povezivanje virtuelne funkcije klase Izvedena

    /* Izlaz
    A Konstruktor objekta:3
    B Konstruktor objekta:3
    Fun objekta:3
    B Destruktor objekta:3
    A Destruktor objekta:3
    */
    return 0;
}
```

static_cast

- Konverzija koja koristi “static type information”, statičke informacije o tipu, poznate u toku kompajliranja
- Obuhvata:
 - Konverzije numeričkih tipova bez poruke o gubitku informacija;
 - Konverzije u hijerarhiji klasa: upcast(pointer ili referenca izvedena u pointer ili referencu osnovne) i downcast(pointere ili referenca osnovne u pointer ili referencu izvedene klase). **Bezbednost konverzije garantuje programer.**
 - Konverzije pomoću konstruktora koji može da prihvati jedan argument
 - Konverzije pomoću operadora konverzije
 - Konverzije void* u odgovarajući tip i obrnuto.

static_cast

- ❑ **Static_cast konverzija nije bezbedna ko dynamic_cast konverzija, zato što ne vrši provere u toku izvršavanja (run time type check)**
- ❑ **Dynamic_cast pokazivača, u slučaju neuspeha, vraća 0 kao rezultat, dok static_cast vraća pokazivač koji ukazuje na „narušenu“ memorijsku strukturu i potencijalno kasnije izazove run time error.**
- ❑ **Dynamic_cast je bezbedan ali funkcioniše samo nad pokazivačima i referencama a neophodan je run time type check.**

STATIC_CAST operator

39

```
/*PRIMER 1*/  
class Osnovna {};  
class Izvedena : public Osnovna {};  
  
void f(Osnovna* pOsn) {  
/* BEZBEDNO SAMO AKO pOsn zaista ukazuje na objekat tipa Izvedena */  
    Izvedena* pIzv = static_cast<Izvedena*>(pOsn);  
}
```

STATIC_CAST operator

40

```
/*PRIMER 1 (polimorfni tipovi) */
class Osnovna {
public:
    virtual void Test() const { std::cout << "Pozdrav iz Osnovne\n"; }
};

class Izvedena : public Osnovna {
public:
    void Test() const { std::cout << "Pozdrav iz Izvedene\n"; }
};

void f(Osnovna* pOsn) {
/* BEZBEDNO SAMO AKO pOsn zaista ukazuje na objekat tipa Izvedena */
    Izvedena* pIzv1 = static_cast<Izvedena*>(pOsn);
    Izvedena* pIzv2 = dynamic_cast<Izvedena*>(pOsn);
    if (pIzv1) pIzv1->Test();
    if (pIzv2) pIzv2->Test();
}
```

STATIC_CAST operator

41

/*PRIMER 1 (polimorfni tipovi) */

```
int main()
{
    printf("%f\n", fmod(-270, 180));
    printf("%f\n", atan2(8, -5)*180/M_PI);
    Osnovna o;
    Izvedena i;
    std::cout << "Prvi poziv\n";
    f(&o);
    std::cout << "Drugi poziv\n";
    f(&i);
    return 0;
}
```

Prvi poziv
Pozdrav iz Osnovne

Drugi poziv
Pozdrav iz Izvedene
Pozdrav iz Izvedene

STATIC_CAST operator

42

```
/*PRIMER 2*/
class Osnovna {
    int m_i;
public:
    Osnovna(int i) :m_i(i){}
    virtual void Test() const{ std::cout << m_i << "\n"; }
};

class Izvedena : public Osnovna {
    int m_j;
public:
    Izvedena(int i, int j) :Osnovna(i), m_j(j){}
    void Test() const{
        Osnovna::Test();
        std::cout << m_j << "\n";
    }
};
```

STATIC_CAST operator

43

```
/*PRIMER 2 nastavak*/
void f(Osnovna* pOsn) {
    /* Ako pOsn zaista ukazuje na objekat tipa Izvedena obe konverzije
daju isti rezultat. Isti rezultat imaju i kada je pOsn = 0 */
    Izvedena* pIzv1 = dynamic_cast<Izvedena*>(pOsn);
    Izvedena* pIzv2 = static_cast<Izvedena*>(pOsn);
    /* Ako pOsn ukazuje na objekat tipa Osnovna dynamic_cast vraca 0, dok
static_cast vraca pokazivac tipa Izvedena na nekonstruisani objekat
tipa Izvedena */
    pIzv1->Test();
    pIzv2->Test();
}

int main() {
    Osnovna a(3);
    Izvedena b(3, 4);
    f(&b); /* Funkcija Test pozvana preko oba pokazivaca daje
isti rezultat */
    f(&a); /* RUN TIME ERROR */
}
```

STATIC_CAST operator

44

```
int main() {  
  
    /* Konverzije numerickih tipova */  
  
    int n = static_cast<int>(3.14); /* NEMA UPOZORENJA O GUBITKU  
INFORMACIJE */  
  
    n = 5.12; /* POSTOJI UPOZORENJE O GUBITKU INFORMACIJE */  
    std::cout << "n = " << n << '\n';  
  
}
```

STATIC_CAST operator

45

```
template<typename T>
struct A {
    T m_i, m_j;
    explicit A(T i, T j = 10) : m_i(i), m_j(j) {}

};

int main() {

    /* Konverzije prilikom inicijalizacije pomocu konstruktora */
    A<int> a1 = 10; /* NIJE DOZVOLJENO ZBOG EKSPLICITNOG KONSTRUKTORA
    PORUKA KOMPAJLERA: ne postoji odgovarajuci konstruktor*/

    A<int> a1 = static_cast<A<int>>(10); /* EKSPLICITNA KONVERZIJA int
    u A<int> objekat : Operator konverzije koristi konstruktor koga moze
    da aktivira jedan argument */

}
```

STATIC_CAST operator

46

```
template<typename T>
struct A {
    T m_i, m_j;
    explicit A(T i, T j = 10) : m_i(i), m_j(j) {}
    operator int() const { return m_i; }

};

int main(){

/* NAREDNE LINIJE KODA DOZVOLJENE SU OD STRANE KOMPJALERA SAMO AKO
POSTOJI OPERATOR KONVERZIJE*/
    int ia1 = a1;
    int ia2 = static_cast<int>(a1); /*static_cast koristi korisnicki
definisani operator konverzije*/
}

}
```

STATIC_CAST operator

47

```
struct Osnovna {};
        struct Izvedena : Osnovna {};

int main(){
    /* STATIC konverzija u pokazivac ili referencu izvedene klase
    (downcast) */
    Izvedena oI;
    Osnovna& rO = oI; /* Implicitna konverzija (upcast) */
    Izvedena& rI = static_cast<Izvedena&>(rO); /* Neophodna eksplicitna
    konverzija (downcast) */

    /* „Invertovanje“ implicitne konverzije pokazivaca bilo kog tipa u
    pokazivac tipa void */
    void* nv = &n; /* Implicitna konverzija */
    int* ni = static_cast<int*>(nv); /* Invertovanje konverzije */
    std::cout << "*ni = " << *ni << '\n';

}
```

STATIC_CAST operator

48

```
template<typename T>
struct A {
    T m_i, m_j;
    explicit A(T i, T j = 10) : m_i(i), m_j(j) {}
    operator int() const { return m_i; }
    enum E{eA=1, eB};
};

int main() {

/* void* u bilo koji tip pokazivaca */
A<int>::E e = A<int>::E::eA;
void* voidp = &e;
A<int>* p = static_cast<A<int>*>(voidp);
std::cout << "A<int> *p->m_i = " << p->m_i << "\n";
std::cout << "A<int> *p->m_j = " << p->m_j << "\n";
}
```

❑ **dynamic_cast (primenjuje se samo nad “polimorfnim” tipovima)**

- ❑ Konverzija pokazivača ili reference polimorfног tipa u pokazivač ili referencu drugog polimorfног tipa.
- ❑ Provera korektnosti konverzije se vrši u toku izvršavanja
- ❑ Pokušaj konverzije u pokazivač tipa koji nije tip objekta na koji konvertovani pokazivač ukazuje daje NULL kao rezultat.
- ❑ Pokušaj konverzije u referencu tipa koji nije tip objekta na koji konvertovana referencia ukazuje prosleđuje std::bad_cast izuzetak

DYNAMIC_CAST operator

50

```
struct A {  
    virtual void f() { }  
};  
struct B : public A { };  
struct C { };  
  
void f () {  
    A a;  
    B b;  
  
    A* ap = &b;  
    B* b1 = dynamic_cast<B*> (&a); /* b1 = NULL, zato sto je a tipa A,  
a ne tipa B */  
    B* b2 = dynamic_cast<B*> (ap); /* b2 = &b */  
    C* c = dynamic_cast<C*> (ap); /* c = NULL, zato sto ap ukazuje na  
objekat tipa B, a ne tipa C */  
  
    A& ar = dynamic_cast<A&> (*ap); /* Korektna konverzija */  
    B& br = dynamic_cast<B&> (*ap); /* Korektna konverzija */  
    C& cr = dynamic_cast<C&> (*ap); /* Konverzija prosledjuje  
std::bad_cast */  
} Projektni uzorci
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

51

```
#include <stdio.h>
#include <stdlib.h>

/* Osnovna klasa svih izraza */

template<typename SubType>
struct Exp {
    /* Operator konverzije */
    operator const SubType&() const {
        return *static_cast<const SubType*>(this);
    }
    const int Length() const {
        return static_cast<const SubType*>(this)->Length();
    }
};
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

52

/* Binarni izraz sabiranja */

```
template<typename TLhs, typename TRhs>
struct BinaryAddExp : public Exp<BinaryAddExp<TLhs, TRhs> > {
    const TLhs &lhs;
    const TRhs &rhs;
    BinaryAddExp(const TLhs& lhs, const TRhs& rhs) : lhs(lhs), rhs(rhs) {
        if (lhs.Length() != rhs.Length()) {
            printf("Pokusaj sabiranja vektora razlicitih duzina");
            exit(1);
        }
    }
    const int Length() const { return lhs.Length(); }
    const float operator[](int i) const {
        return lhs[i] + rhs[i];
    }
};
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

53

/* Binarni izraz oduzimanja */

```
template<typename TLhs, typename TRhs>
struct BinarySubExp : public Exp<BinarySubExp<TLhs, TRhs> > {
    const TLhs &lhs;
    const TRhs &rhs;
    BinarySubExp(const TLhs& lhs, const TRhs& rhs) : lhs(lhs), rhs(rhs) {
        if (lhs.Length() != rhs.Length()) {
            printf("Pokusaj oduzimanja vektora razlicitih duzina");
            exit(1);
        }
    }
    const int Length() const { return lhs.Length(); }
    const float operator[](int i) const {
        return lhs[i] - rhs[i];
    }
};
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

54

```
struct Vec : public Exp<Vec> {
private:
    int length;
    float *ptrToVec;
public:
    Vec(int length) :length(length), ptrToVec(new float[length]) {}
    Vec(float *ptrToVec, int length)
        :length(length), ptrToVec(new float[length]) {
        for (int i = 0; i < length; ++i)
            this->ptrToVec[i] = ptrToVec[i];
    }
    ~Vec() { delete[] ptrToVec; }

/* Preklopjeni operatori indeksiranja */
    const float operator[] (int i) const { return *(ptrToVec + i); }
    float& operator[] (int i) {      return *(ptrToVec + i);  }
    . . .
};
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

55

```
struct Vec : public Exp<Vec> {
/* Funkcija u kojoj se stvarna evaluacija izraza desava kroz rekurzivne
pozive operatora[] */

template<typename EType>
Vec& operator= (const Exp<EType>& src) {
    const EType& src_sub = src; /* Dozvoljeno bez eksplicitne
konverzije, samo zato sto u strukturi Exp postoji preklopljeni operator
konverzije */
    delete [] ptrToVec;
    length = src_sub.Length();
    ptrToVec = new float[length];

    for (int i = 0; i < length; ++i)
        ptrToVec[i] = src_sub[i];
    return *this;

}

};
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

56

```
struct Vec : public Exp<Vec> {
/* Funkcija u kojoj se stvarna evaluacija izraza desava kroz rekurzivne
pozive operatora[] */

template<typename EType>
Vec(const Exp<EType>& src):length(static_cast<const
EType&>(src).Length()), ptrToVec(new float[length]) {

    const EType& src_sub = src; /* Dozvoljeno bez eksplisitne
konverzije, samo zato sto u strukturi Exp postoji prekopljen operator
konverzije */
    for (int i = 0; i < length; ++i)
        ptrToVec[i] = src_sub[i];
}

const int Length() const { return length; }

};
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

57

```
/* Sablon operatora sabiranja */
template<typename TLhs, typename TRhs>
inline BinaryAddExp<TLhs, TRhs>
operator+(const Exp<TLhs> &lhs, const Exp<TRhs> &rhs) {
    return BinaryAddExp<TLhs, TRhs>(lhs, rhs);
}
```

```
/* Sablon operatora oduzimanja */
template<typename TLhs, typename TRhs>
inline BinarySubExp<TLhs, TRhs>
operator-(const Exp<TLhs> &lhs, const Exp<TRhs> &rhs) {
    return BinarySubExp<TLhs, TRhs>(lhs, rhs);
}
```

ZASTUPNIK (engl. PROXY)

Primer 4. Aritmeticke operacije nad vektorima – odloženo izračunavanje

58

```
const int n = 3;

int main(void) {

    float sb[n] = { 2, 3, 4 };
    float sc[n] = { 3, 4, 5 };
    Vec A(n), B(sb, n), C(sc, n);

    A = (B + C) + (C - B);
    for (int i = 0; i < n; ++i) {
        printf("%d: %f == (%f + %f) + (%f - %f)\n", i,
               A[i], B[i], C[i], C[i], B[i]);
    }

    return 0;
}
```

❑ Ime i klasifikacija

- ❑ Muva (engl. Flyweight)
- ❑ Strukturni projektni uzorak

❑ Namena

- ❑ Deljenje „lakih“ objekata sa ciljem da se izbegne hiperprodukcija ovakvih objekata
- ❑ Pod lakim objektom se podrazumeva
 - ❑ Objekat bez stanja
 - ❑ Objekat čije unutrašnje stanje ne zavisi od konteksta u kome se javlja

❑ Drugo ime

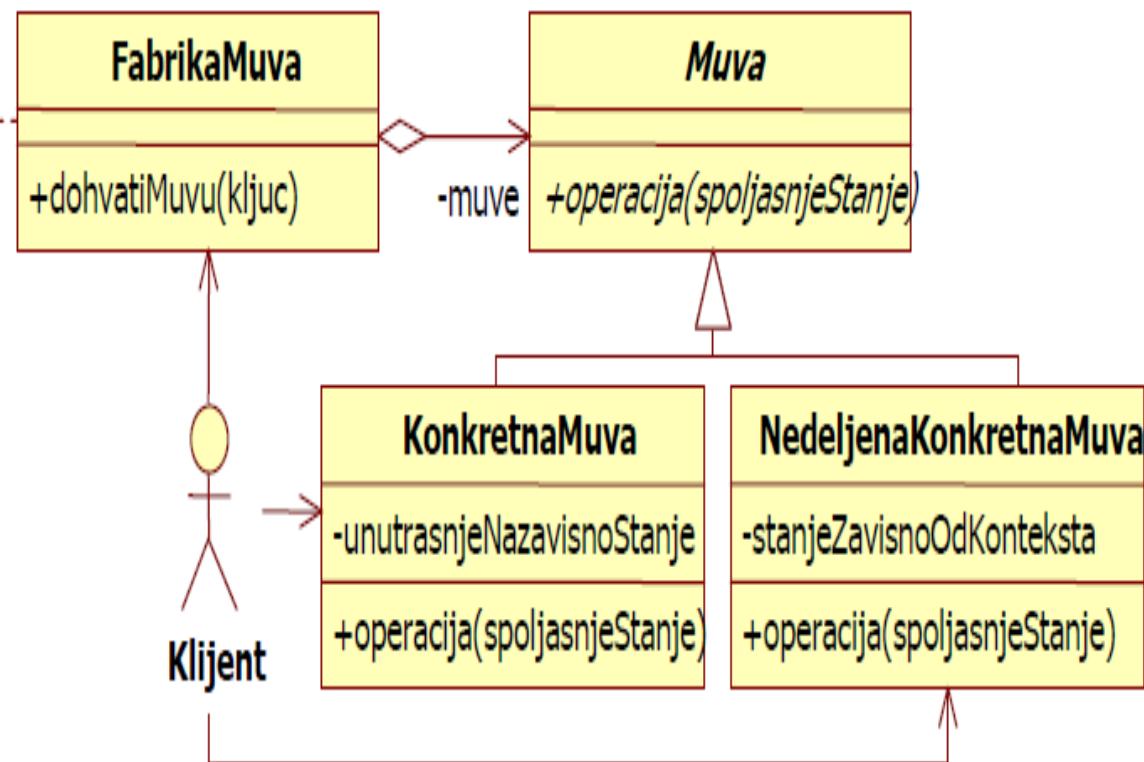
- ❑ Predstavnik, Zamena, Surogat (engl. Surrogate)

❑ Primenljivost

- ❑ **Aplikacija koristi veliki broj objekata istog tipa**
 - ❑ Objekti troše značajan memorijski prostor
- ❑ **Veći deo stanja objekta može da se prebaci u tzv „eksterno stanje“**
 - ❑ Eksterno stanje objekta definisano je kontekstom u kome se objekat javlja
- ❑ **Kada se ukloni eksterno stanje – „laki“ objekti postaju deljivi**
 - ❑ Objekti mogu da se zamene referencama ka deljenim objektima
 - ❑ Deljeni objekti su bez unutrašnje stanja ili sa unutrašnjim stanjem koje ne zavisi od konteksta u kome se objekat javlja
- ❑ **Identitet objekta nije bitan**

□ Struktura

```
dohvatiMuvu(kljuc){
    if (muva=muve[kljuc]){
        return muva;
    } else {
        muva=new KonkretnaMuva();
        muve.dodaj(muva);
        return muva;
    }
}
```



❑ Učesnici

❑ Muva (ApstraktnaMuva)

- ❑ Deklariše interfejs kroz koji muva prima eksterno stanje

❑ KonkretnaMuva

- ❑ Implementira interfejs muve i dodaje atribute internog stanja
- ❑ Objekti ove klase ne zavise od konteksta u kome se javljaju i moraju da budu interno deljivi

❑ NedeljenaKonkretnaMuva

- ❑ U hijerarhiji objekata muva neki objekti ne mogu da se dele.
- ❑ Ovakvi objekti čuvaju i atribute eksternog stanja koje zavisi od konteksta.

Učesnici (nastavak)

Fabrika Muva

- Kreira i upravlja muvama, obezbedjuje odgovarajuće deljenje objekata.
Kada klijent zahteva muvu, fabrika prosledi postojeću ili kreira novu.

Klijent

- Čuva reference dodeljenih muva.
- Čuva ili izračunava eksterno stanje objekata muva.

□ Saradnja

- **Klijent dobija od fabrike muva „lake objekte“**
- **Prilikom poziva operacija, klijent prenosi lakom objektu eksterno stanje.**

□ Posledice

- **Postoje potencijalni troškovi nalaženja muve i izračunavanja stanja**
- **Ušteda u memorijskom prostoru zavisi od**
 - Broja objekata
 - Odnosa internog i eksternog stanja po objektu i
 - Da li se eksterno stanje čuva ili izračunava

MUVA (engl. FLYWEIGHT)

Primer 1.

65

```
/* Apstraktna klasa koja definise interfejs */

class ApstraktnaMuva{
public:
/* Operacija nad eksternim stanjem */
    virtual void Operacija(const string& eksternoStanje)=0;

    string GetInternoStanje() const { return m_internoStanje; }

    virtual ~ApstraktnaMuva() {}

    ApstraktnaMuva(string internoStanje):
        m_internoStanje(internoStanje) {}

private:
/* Interno stanje lakog objekta */
    string m_internoStanje;
};
```

MUVA (engl. FLYWEIGHT)

Primer 1.

66

```
/* Konkretna klasa lakih objekata */

class Muva:public ApstraktnaMuva{
public:

/* Implementacija interfejsa */
    virtual void Operacija(const string& eksternoStanje) {
        cout << "Interno stanje: " <<
            (this->GetInternoStanje()).c_str() << endl;
        cout << "Eksterno stanje: " << eksternoStanje.c_str() << endl;
    }
    Muva(string internoStanje):ApstraktnaMuva(internoStanje) {}
    virtual ~Muva() {}
};
```

MUVA (engl. FLYWEIGHT)

Primer 1.

67

```
/* Klasa normalnih objekata */
class NedeljenaMuva:public ApstraktnaMuva{
/*Objekat ove klase je normalan objekat i
nije predvidjen da bude deljen */

    string m_dodatniAtributi;

public:
    virtual void Operacija(const string& eksternoStanje){
        cout << "Privatni atribut: " << m_dodatniAtributi.c_str()
            << "; EksternoStanje: " << eksternoStanje.c_str()<< endl;
    }

    NedeljenaMuva(string internoStanje,
                  string dodatniAtributi):
        ApstraktnaMuva(internoStanje),
        m_dodatniAtributi(dodatniAtributi) {}
    virtual ~NedeljenaMuva() {}}
};
```

MUVA (engl. FLYWEIGHT)

Primer 1.

68

```
/* Klasa koja kreira lake objekte I zaduzena je za njihovo cuvanje */
class FabrikaMuva{
public:
    FabrikaMuva() {}
    ~FabrikaMuva() {
        list< ApstraktnaMuva* >::const_iterator it = m_ListaMuva.begin();
        for (; it != m_ListaMuva.end(); ++it) delete (*it);
    }
    ...
/* Broj muva objekata */
    unsigned GetBrojMuva() const { return m_ListaMuva.size(); }
private:
    /* Cuvanje lakih objekata u listi */
    list<ApstraktnaMuva*> m_ListaMuva;
};
```

MUVA (engl. FLYWEIGHT)

Primer 1.

69

```
/* Klasa koja kreira laki objekte I zaduzena je za njihovo cuvanje */
class FabrikaMuva{
public:
/* Funkcija fabrike koja vraca laki objekat na osnovu kljuca */
ApstraktnaMuva* GetMuva(string kljuc){
    /* Ukoliko laki objekat postoji, vrati pokazivac na njega */
    list< ApstraktnaMuva* >::const_iterator it = m_ListaMuva.begin();
    for(; it!= m_ListaMuva.end(); ++it){
        cout<<"Vracam muvu sa kljucem " <<kljuc.c_str()<<endl;
        if( ((*it)->GetInternostanje()).compare(kljuc) == 0)
            return (*it);
    }
    /* Ukoliko laki objekat ne postoji kreiraj ga, smesti u listu i
vrati pokazivac na njega */
    cout<<"Kreiram muvu sa kljucem " <<kljuc.c_str()<<endl;
    ApstraktnaMuva* muva = new Muva(kljuc);
    m_ListaMuva.push_back(muva);
    return muva;
}
. . .
};
```

MUVA (engl. FLYWEIGHT)

Primer 1.

70

```
/* Klijent koji koristi lake objekte preko pokazivaca,  
odnosno referenci */
```

```
int main()  
{  
    string eksternoStanje = "EKSTERNO";  
    FabrikaMuva* ptrFab = new FabrikaMuva();  
    ApstraktnaMuva* ptrM1 = ptrFab->GetMuva("kljuc A");  
    ApstraktnaMuva* ptrM2 = ptrFab->GetMuva("kljuc A");  
    ptrM1->Operacija(eksternoStanje);  
    cout<<"Broj muva" << ptrFab->GetBrojMuva();  
    delete ptrFab;  
    return 0;  
}
```

MUVA (engl. FLYWEIGHT)

Primer 2

71

```
#include <string>
#include <iostream>
using namespace std;

/* Flyweight apstraktna klasa */
class Model{
private:
    string m_name;
    int m_capacity;
    int m_speed;

public:
    virtual void TechSpec()  {
        cout << "Name : " << m_name << ", Capacity : " << m_capacity << ",
        Speed: " << m_speed << " km/h " << endl;
    }
    Model(string name, int capacity, int speed) : m_name(name),
    m_capacity(capacity), m_speed(speed) {}
};
```

MUVA (engl. FLYWEIGHT)

Primer 2

72

```
/* Flyweight konkretna klasa */
class Airbus380 : public Model{
public: Airbus380() :Model("Airbus380", 200, 800) {}
};

/* Flyweight konkretna klasa */
class Boeing787 : public Model{
public: Boeing787() :Model("Boeing787", 600, 1000) {}
};

/* Flyweight konkretna klasa */
class Boeing797 : public Model{
public: Boeing797() :Model("Boeing797", 1200, 1500) {}
};
```

MUVA (engl. FLYWEIGHT)

Primer 2

73

```
/* Sablon unikata odredjenog tipa proizvoda */
template<typename T>
class AeroplaneFactory {
public:
    static T& Instance() {
        static T statInstance;
        return statInstance;
    }
};
```

MUVA (engl. FLYWEIGHT)

Primer 2

74

/* Klasa proizvoda */

```
class Aeroplane{
private: Model &m_model;
    string m_Date;
    int m_id;
public:
    Aeroplane(Model &model, string date, int id) : m_model(model),
m_Date(date), m_id(id) {}

    virtual void TechSpec() {
        m_model.TechSpec();
        cout << "Production Date : " << m_Date << ", Serial No: " << m_id
<< endl;
    }
};
```

MUVA (engl. FLYWEIGHT)

Primer 2

75

```
int main() {
    Aeroplane prvi =
        Aeroplane(AeroplaneFactory<Airbus380>::Instance(), "10th Feb 2018",
100213);

    Aeroplane drugi =
        Aeroplane(AeroplaneFactory<Boeing787>::Instance(), "1th Feb 2018",
100214);

    Aeroplane treci =
        Aeroplane(AeroplaneFactory<Airbus380>::Instance(), "10th Nov 2018",
100215);

    Aeroplane cetvrti =
        Aeroplane(AeroplaneFactory<Boeing787>::Instance(), "1th Dec 2017",
100216);

    prvi.TechSpec();
    drugi.TechSpec();
    treci.TechSpec();
    cetvrti.TechSpec();

    return 0;
}
```

MUVA (engl. FLYWEIGHT)

Primer 2

76

```
int main() {
    Aeroplane prvi =
        Aeroplane(AeroplaneFactory<Airbus380>::Instance(), "10th Feb 2018",
100213);

    Aeroplane drugi =
        Aeroplane(AeroplaneFactory<Boeing787>::Instance(), "1th Feb 2018",
100214);

    Aeroplane treci =
        Aeroplane(AeroplaneFactory<Airbus380>::Instance(), "10th Nov 2018",
100215);

    Aeroplane cetvrti =
        Aeroplane(AeroplaneFactory<Boeing787>::Instance(), "1th Dec 2017",
100216);

    prvi.TechSpec();
    drugi.TechSpec();
    treci.TechSpec();
    cetvrti.TechSpec();

    return 0;
}
```

**Name : Airbus380, Capacity : 200, Speed: 800 km/h
Production Date : 10th Feb 2018, Serial No:100213**

**Name : Boeing787, Capacity : 600, Speed: 1000 km/h
Production Date : 1th Feb 2018, Serial No: 100214**

**Name : Airbus380, Capacity : 200, Speed: 800 km/h
Production Date : 10th Nov 2018, Serial No: 100215**

**Name : Boeing787, Capacity : 600, Speed: 1000 km/h
Production Date : 1th Dec 2017, Serial No: 100216**