Objektno orijentisano programiranje u C++-u Projektni uzorci

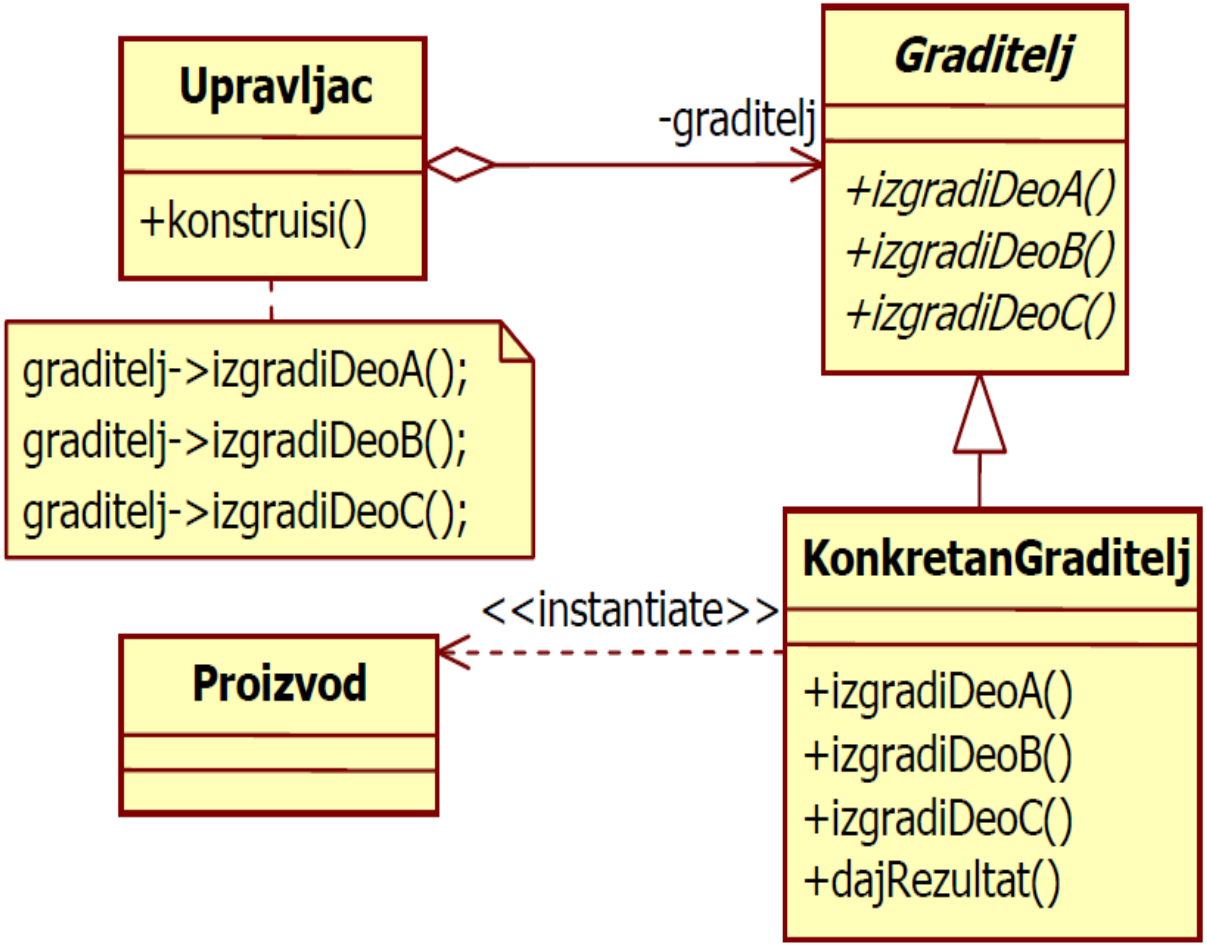
STVARALAČKI PROJEKTNI UZORCI

GRADITELJ
FABRIČKI METOD
PROTOTIP

- **Ime i klasifikacija**
  - **Graditelj (engl. Builder)**
  - **Stvaralački projektni uzorak**
- **Namena**
  - **Koristi se da bi kontrolisao instanciranje neke klase Proizvoda.**
  - **Primenjuje se u kreiranju složenih objekata, koji se sastoje iz delova koji moraju da budu kreirani istim redosledom, ili je redosled definisan nekim algoritmom**
  - **Algoritam konstrukcije implementiran je u klasi Direktor.**

- **Primenljivost**
  - **Graditelja treba koristiti kada je algoritam za kreiranje složenih objekata nezavisan od**
    - **delova koji čine objekat**
    - **od načina na koji se objekti spajaju u celinu.**
  - **Proces konstrukcije mora da dozvoli različite reprezentacije objekata koji se konstruišu.**

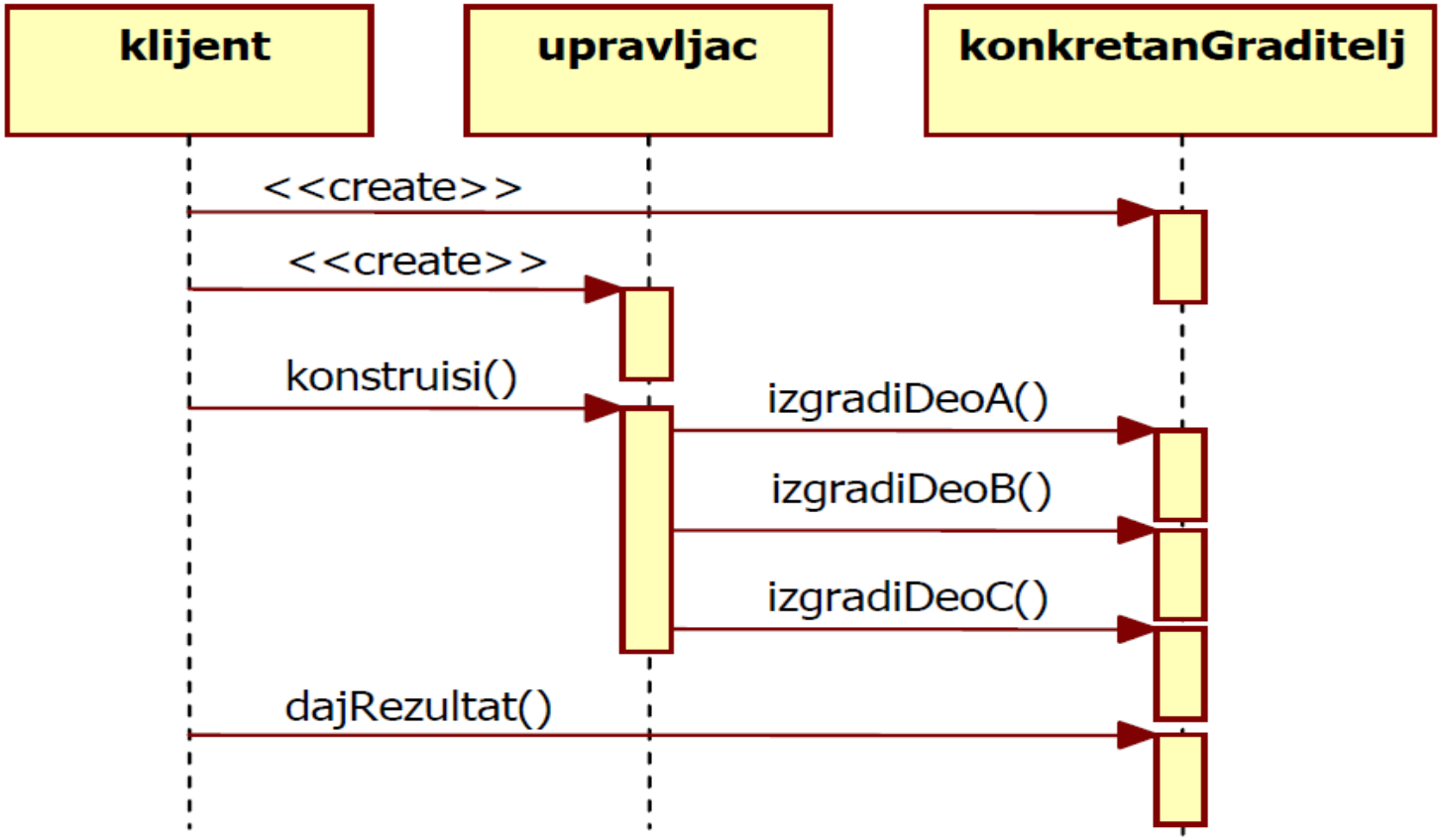
Struktura


Upravljac
+konstruisi()
-graditelj
Graditelj
+izgradiDeoA()
+izgradiDeoB()
+izgradiDeoC()
graditelj->izgradiDeoA();
graditelj->izgradiDeoB();
graditelj->izgradiDeoC();
KonkretanGraditelj
+izgradiDeoA()
+izgradiDeoB()
+izgradiDeoC()
+dajRezultat()
<<instantiate>>
Proizvod

- **Učesnici**
  - **Graditelj**
    - Definiše apstraktni interfejs za kreiranje delova objekata **Proizvod**
    - Apstraktna osnovna klasa koja deklariše sve korake od kojih se proces izgradnje složenog objekta sastoji. Funkcija **vratiProizvod** se koristi da bi vratila konačni proizvod.
  - **KonkretanGraditelj**
    - Konstruiše i sastavlja delove proizvoda implementiranjem interfesja **Graditelj**
    - Svaki korak je implementacija virtuelne funkcije interfejsa **Graditelj**

- **Učesnici**
  - **Direktor (Upravljac)**
    - Ova klasa implementira algoritam kreiranja složenog objekta, pri čemu za kreiranje pojedinačnih delova objekta poziva **Graditelj**.
    - Kada se instancira objekat klase **Direktor**, poziva se njegova funkcija **konstruiši** sa odgovarajućim parametrom kojim se definiše koji će od mogućih graditelja da bude pozvan da konstruiše objekat.
    - Funkcija **konstruiši** implementira algoritam izgradnje složenog objekta tako što se nad **Graditeljem** pozivaju odgovarajuće funkcije koje implementiraju korake izgradnje delova objekta.
    - Po završetku procesa izgradnje funkcija **vratiProizvod** se koristi da bi vratila konačan kreiran proizvod.
  - **Proizvod**
    - Predstavlja složeni objekat koji se konstruiše
    - Uključuje klase koje definišu sastavne delove

Saradnja


klijent
upravljac
konkretanGraditelj
<<create>>
<<create>>
konstruisi()
izgradiDeoA()
izgradiDeoB()
izgradiDeoC()
dajRezultat()

Posledice

- Omogućava izmene interne reprezentacije i načina sklapanja složenih objekata
  - **Graditelj** daje **Direktoru** apstraktan interfes za konstrukciju objekta
  - Reprezentacija i interna struktura, kao i način sklapanja su sakriveni
  - Za promenu interne reprezentacije potrebna je samo nova vrsta **Graditelja**.
- Klijent ne zna ništa o klasama koje definišu unutrašnju strukturu proizvoda, jer se one ne javljaju u interfejsu **Graditelja**.
- Preciznija kontrola nad procesom konstrukcije od drugih stvaraoca
  - Drugi stvaraoci kreiraju objekte u jednom potezu
  - Graditelj konstruiše objekat korak po korak, pod kontrolom direktora

- **Povezani uzorci**
  - **Graditelj** se fokusira na konstruisanje složenog objekta korak po korak.
  - **Apstraktna Fabrika** omogućava konstrukciju familije objekata (jednostavnih ili složenih)
  - **Graditelj** vraća proizvod kao konačni korak algoritma izgradnje
  - **Apstraktna Fabrika** vraća objekte u jednom koraku

GRADITELJ (engl. BUILDER)
Primer 1: Osnovna implementacija uzorka GRADITELJ


```
/* Klase koje implementiraju delove automobila */
class Tocak{int vel; public:Tocak(int v):vel(v){}};
class Motor{int snaga; public:Motor(int s):snaga(s){}};
class Menjac{std::string tip; public:Menjac(std::string t):tip(t){}};

/* Finalni proizvod - auto */
class Direktor;  //Deklaracija unapred
class Auto {
  friend Direktor;
  Tocak*  tockovi[4];
  Motor*  motor;
  Menjac* menjac;
public:
  virtual ~Auto() {
    for(int i = 0; i<4;++i) delete tockovi[i];
    delete motor; delete menjac;
  }
  void spec(){
    std::cout << "Tip menjaca:" << menjac->tip << std::endl;
    std::cout << "Snaga motora:" << motor->snaga << std::endl;
    std::cout << "Velicina guma:" << tocak[0]->vel << std::endl;
  }
};
```

GRADITELJ (engl. BUILDER)
Primer 1: Osnovna implementacija uzorka GRADITELJ


```
/* Graditelj gradi delove slozenog objekta */
class Graditelj{
public:
    virtual Tocak* napraviTocak() = 0;
    virtual Motor* napraviMotor() = 0;
    virtual Menjac* napraviMenjac() = 0;
};
/* Direktor trazi od graditelja da izgradi auto*/
class Direktor {
    Graditelj* ptrGrdt;
public:
    void setGraditelj(Graditelj* ptrNoviGrdt){ptrGrdt = ptrNoviGrdt;}
    Auto* napraviAuto(){
        Auto* ptrAuto = new Auto(); /* za sada prazna skoljka */
        ptrToAuto->menjac = ptrGrdt->napraviMenjac();
        ptrToAuto->engine = ptrGrdt->napraviMotor();
        for(int i = 0; i<4; ++i)
            ptrToAuto->tocak[i] = ptrGrdt->napraviTocak();
        return ptrToAuto;
    }
};
```

GRADITELJ (engl. BUILDER)
Primer 1: Osnovna implementacija uzorka GRADITELJ


```
/* Konkretni graditelj za Jeep SUV auto */
class PorscheGraditelj : public Graditelj{
public:
    Tocak* napraviTocak() { return new Tocak(22); }
    Motor* napraviMotor() { return new Motor(400); }
    Menjac* napraviMenjac() {return new Menjac("AUTOMATIC");}
};

/* Konkretni graditelj za Fiat 500 auto */
class FiatGraditelj : public Graditelj{
public:
    Tocak* napraviTocak() { return new Tocak(15); }
    Motor* napraviMotor() { return new Motor(95); }
    Menjac* napraviMenjac() {return new Menjac("MANUALNI");}
};
```

GRADITELJ (engl. BUILDER)
Primer 1: Osnovna implementacija uzorka GRADITELJ


```
/* Klijent*/
void Klijent(){
    // Konacni proizvod
    Auto *ptrAuto;

    // Direktor
    Direktor direktor;
    //Konkretni graditelji
    PorscheGraditelj porscheGraditelj;
    // Napravi Jeep
    std::cout << "Jeep" << std::endl;
    direktor.setGraditelj(&porscheGraditelj);
    ptrAuto = direktor.napraviAuto();
    ptrAuto->spec();
    delete ptrAuto;
}
```

GRADITELJ (engl. BUILDER)
Primer 1: Osnovna implementacija uzorka GRADITELJ


```
/* Klijent*/
void Klijent(){
    // Konacni proizvod
    Auto *ptrAuto;
    // Direktor
    Direktor direktor;
    //Konkretni graditelji
    FiatGraditelj fiatGraditelj;
    // Napravi Fiat
    std::cout << "Fiat" << std::endl;
    direktor.setGraditelj(&fiatGraditelj);
    ptrAuto = direktor.napraviAuto();
    ptrAuto->spec();
    delete ptrAuto;
}
/* Objekat tipa direktor ne moze da bude odgovoran za brisanje
objekata tipa Graditelj i Proizvod(u nasem slucaju Auto). Da bi smo
izbegli probleme sa curenjem memorije, koristicemo pametne
pokazivace, koji prebrojavaju reference na objekat I po potrebi,
ukoliko nema vise ni jedne reference koja ukazuje na neki bjekat,
taj objekat brisu iz memorije. */
```

GRADITELJ (engl. BUILDER)
Pametni “reference counting” pokazivac


```
template<typename T>
class RCPointer{
    T* _pToObj;
    size_t *_pToCnt;

    void clear() {
        /* Dereferenciranje NULL pointera prouzrokuje nedefinisano
ponasanje */
        if (_pToCnt && !--*_pToCnt) {
            delete _pToObj; delete _pToCnt;
            _pToObj = NULL; _pToCnt = NULL; }
    }
public:
…
};
```

GRADITELJ (engl. BUILDER)
Pametni "reference counting" pokazivac


```
template<typename T>
class RCPointer{
public:

/* Podrazumevani konstruktor. RCPointer potpuno preuzima kontrolu nad
brisanjem objekta koji je prenet preko pokazivaca p. */
    RCPointer(T* p = NULL)
    try :  _pToObj(p), _pToCnt(p ? new size_t(1) : NULL) {}
    catch (...) { delete p; }

/* Sablon konstruktora */
    template<typename U>
    RCPointer(U* p = NULL)
    try :  _pToObj(p), _pToCnt(p ? new size_t(1) : NULL){}
    catch (...) { delete p; }
/* try catch blokovi se koriste zbog toga sto new operator moze da
prosledi izuzetak (exception). Try se primenju nad listom za
inicijalizaciju i telom konstruktora. Ukoliko dodje do prosledjivanja
izuzetka iz new operatora, brise se objekat prenet preko p, cime se
sprecava curenje memorije. */
...
}; /* end RCPointer */
```

GRADITELJ (engl. BUILDER)
Pametni "reference counting" pokazivac


```
template<typename T>
class RCPointer{
public:
/*  Podrazumevani konstruktor kopije */
    RCPointer(const RCPointer& other) :
    _pToObj(other.PToObj()), _pToCnt(other.PToCnt()) {
/* Dereferenciranje NULL pointera izaziva nedefinisano ponasanje. Zato
ispitujemo da li je _pToCnt razlicit od 0 */
        if(_pToCnt) ++*_pToCnt;
    }

/* Sablon konstruktora kopije */
    template<typename U>
    RCPointer(const RCPointer<U>& other) : _pToObj(other.PToObj()),
        _pToCnt(other.PToCnt()){
        if(_pToCnt) ++*_pToCnt;
    }
/* Destruktor */
    ~RCPointer(){ this->clear(); }
. . .
}; /* end RCPointer */
```

```
template<typename T>
class RCPointer{
public:
/* Podrazumevani operator dodele vrednosti */
    RCPointer& operator=(const RCPointer& other) {
        if (this != &other) {
            this->clear();
            _pToObj = other.PToObj();
            _pToCnt = other.PToCnt();
            if(_pToCnt) ++*_pToCnt;
        }
        return *this;
    }
. . .
}; /* end RCPointer */
```

```
template<typename T>
class RCPointer{
public:
/* Sablon operatora dodele vrednosti, kada je vrednosti RCPointer
drugog tipa. */
    template<typename U>
    RCPointer& operator=(const RCPointer<U>& other){
        if (this != &other){
            this->clear();
            _pToObj = other.PToObj();
            _pToCnt = other.PToCnt();
            if(_pToCnt) ++*_pToCnt;
        }
        return *this;
    }
. . .
}; /* end RCPointer */
```

```
template<typename T>
class RCPointer{
public:
/* Sablon operatora dodele vrednosti, kada je vrednost jednostavni
pokazivac  */
    template<typename U>
    RCPointer& operator=(U* p){
        if (_pToObj != p){
            this->clear();
            _pToObj = p;
            try  _pToCnt = p ? new size_t(1) : NULL;
            catch (...)  delete p;
        }
        return *this;
    }
. . .
}; /* end RCPointer */
```

GRADITELJ (engl. BUILDER)
Pametni “reference counting” pokazivac


```
template<typename T>
class RCPointer{
public:
    T& operator*(){ return *_pToObj; }
    const T& operator*() const{ return *_pToObj; }
    T* operator->(){ return _pToObj; }
    const T* operator->() const{ return _pToObj; }
    size_t getRefCount(){ return *_pToCnt; }
    T* PToObj() const { return _pToObj; }
    size_t* PToCnt() const { return _pToCnt; }
}; /* end RCPointer */
```

```
/* Sabloni operatora poredjenja po jednakosti */
/* Sabloni operatora poredjenja sa null pointerom */
template<typename T>
bool operator==(const RCPointer<T> &rc, std::nullptr_t) {
    return rc.PToObj() == nullptr;
}
template<typename T>
bool operator==(std::nullptr_t, const RCPointer<T> &rc) {
    return rc.PToObj() == nullptr;
}

template<typename T>
bool operator!=(const RCPointer<T> &rc, std::nullptr_t) {
    return !(rc == nullptr);
}
template<typename T>
bool operator!=(std::nullptr_t, const RCPointer<T> &rc) {
    return !(rc == nullptr);
}
```

```
/* Sabloni operatora poredjenja po jednakosti */
/* Sabloni operatora poredjenja sa drugim RCPointerom */
template<typename T, typename U >
bool operator==(const RCPointer<T> &left, const RCPointer<U> &right) {
    return left.PToObj() == right.PToObj();
}
template<typename T, typename U>
bool operator!=(const RCPointer<T> &left, const RCPointer<U> &right) {
    return !(left == right);
}
```

```
/* Sabloni operatora poredjenja sa drugim jednostavnim pointerom */

template<typename T, typename U>
bool operator==(const RCPointer<T> &left, U *right) {
    return left.PToObj() == right;
}

template<typename U, typename T>
bool operator==(const U *left, const RCPointer<T> &right) {
    return right == left;
}

template<typename T, typename U>
bool operator!=(const RCPointer<T> &left, U *right) {
    return !(left == right);
}

template<typename U, typename T>
bool operator!=(U *left, const RCPointer<T> &right) {
    return !(right == left);
}
```

GRADITELJ (engl. BUILDER)
Primer 2: Proizvod


```
// Proizvod
class Racunar {
public:
    Racunar():r_komponente(){}
    ~Racunar() {}
    void setKomponenta(string komponenta) {
        r_komponente.add(komponenta);
    }
    void start() {
        for(int i = 1; i<r_komponente.size(); ++i)
            cout << "Start" << i << ":" << r_komponente[i] << endl ;
        }
    }
private:
    Array<string> r_komponente;
};
```

GRADITELJ (engl. BUILDER)
Primer 2: Apstraktni Graditelj


```
// Apstraktni Graditelj
class GraditeljRacunara {
public:
    GraditeljRacunara() {}
    ~GraditeljRacunara() {}

    virtual void napraviProcesor() = 0;
    virtual void napraviMaticnuPlocu() = 0;
    virtual void napraviMonitor() = 0;
};
```

GRADITELJ (engl. BUILDER)
Primer 2: Direktor


```
// Direktor
class Direktor {
public:
    Direktor():ptrGraditeljRacunara() {}
    ~Direktor() {}

    void setGraditeljRacunara( const RCPointer<GraditeljRacunara>
rcPtrGR){
        ptrGraditeljRacunara = rcPtrGR;
    }
    RCPointer<Racunar> napraviRacunar(){
        RCPointer<Racunar> p(new Racunar());
        ptrGraditeljRacunara->napraviProcesor();
        ptrGraditeljRacunara->napraviMaticnuPlocu();
        ptrGraditeljRacunara->napraviMonitor();
        return p;
    }
private:
    RCPointer<GraditeljRacunara> rcPtrToGradRacunara;
};
```

GRADITELJ (engl. BUILDER)
Primer 2: Konkretne klase graditlelji


```
// Konkretni graditelj racunara
class DeskTopGraditelj : public GraditeljRacunara {
public:
    DeskTopGraditelj() {}
    ~DeskTopGraditelj() {}

    void napraviProcesor() {r_komponente.setKomponenta("Procesor1");}

    void napraviMaticnuPlocu() {
        r_komponente.setKomponenta("MaticnaPloca1");
    }

    void napraviMonitor() { r_komponente.setKomponenta("Monitor1");}
};
```

GRADITELJ (engl. BUILDER)
Primer 2: Konkretne klase graditelji


```
// Konkretni graditelj racunara

class LapTopGraditelj : public GraditeljRacunara {
public:
    LapTopGraditelj() {}
    ~LapTopGraditelj() {}

    void napraviProcesor() { r_komponente.setKomponenta("Procesor2");}

    void napraviMaticnuPlocu() {
        r_komponente.setKomponenta("MaticnaPloca2");
    }

    void napraviMonitor() { r_komponente.setKomponenta("Monitor2");}

};
```

GRADITELJ (engl. BUILDER)
Primer 2: Klijent


```
int _tmain(int argc, _TCHAR* argv[]){
  Direktor direktor;

  cout << "Konstruisi LapTop" << endl;
  direktor.setGraditeljRacunara(new LapTopGraditelj());
  direktor.konstruisiRacunar();
  RCPointer<Racunar> ptrToRacunar = direktor.getRacunar();
  ptrToRacunar->start();
  ....

  return 0;
}
```

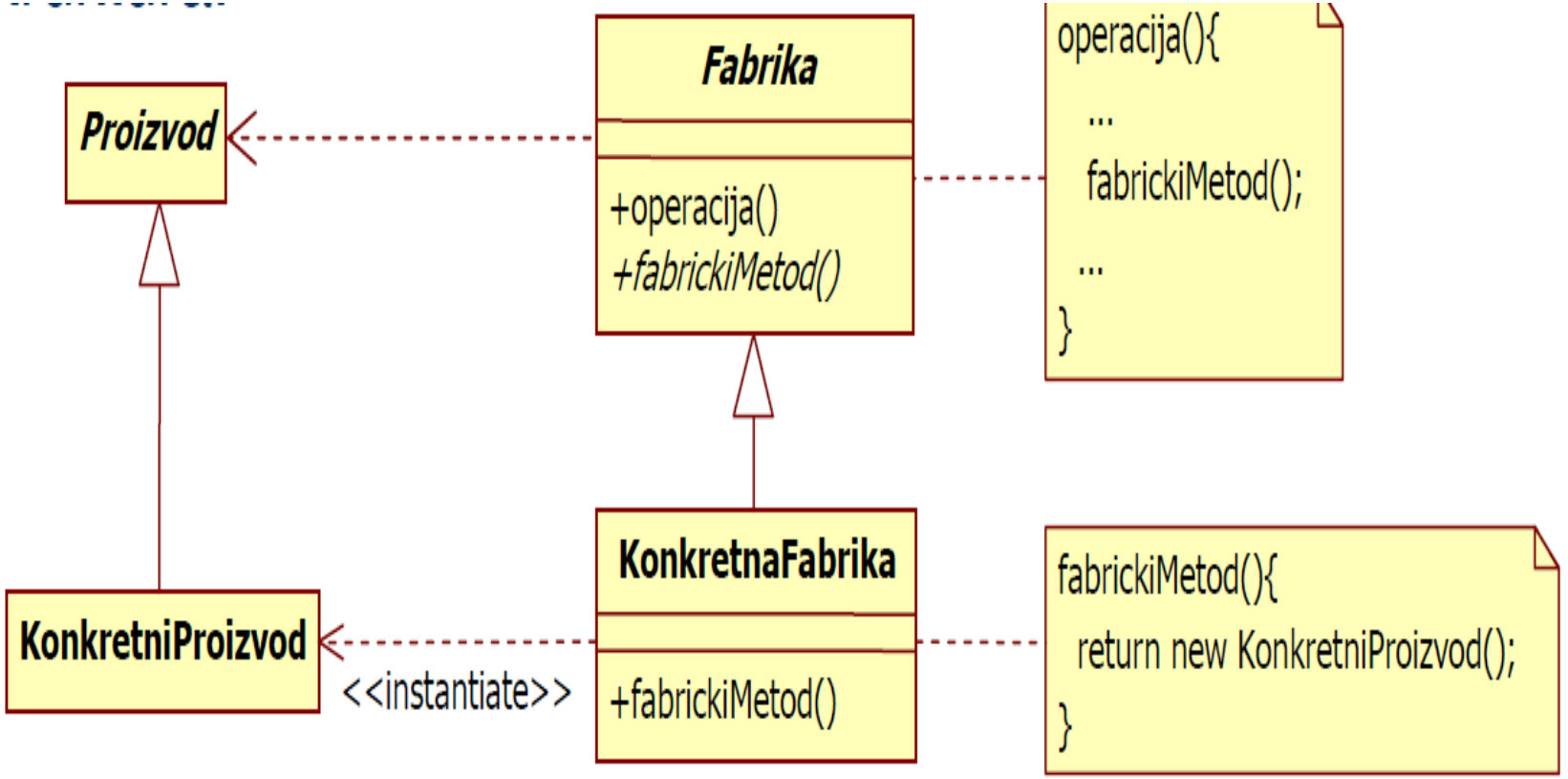
GRADITELJ (engl. BUILDER)
Primer 2: Klijent


```
int _tmain(int argc, _TCHAR* argv[]){
  Direktor direktor;

  cout << "Konstruisi DeskTop" << endl;
  direktor.setGraditeljRacunara(new DeskTopGraditelj());
  direktor.konstruisiRacunar();
  RCRacunar<Racunar> ptrToRacunar = direktor.getRacunar();
  ptrToRacunar->start();
  ....

  return 0;
}
```

FABRIČKI METOD (engl. FACTORY METHOD)

- **Ime i klasifikacija**
  - **Fabrički (proizvodni) metod (engl. Factory Method)**
  - **Stvaralački projektni uzorak**
- **Drugo ime**
  - ***Virtuelni konstruktor (engl. Virtual Constructor)***
- **Namena**
  - **Omogućava klasi da delegira stvaranje objekata potklasama**
  - **Definiše interfejs za pravljenje objekata, ali dozvoljava klasi da prepusti pravljenje objekata izvedenim klasama.**
- **Primenljivost**
  - **Klasa ne može da zna klasu objekta koji treba da pravi.**
  - **Klasa namerno hoće da njene podklase definisu objekte koje prave.**

Struktura


Proizvod
Fabrika
+operacija()
+fabrickiMetod()
operacija(){
...
fabrickiMetod();
...
}
KonkretniProizvod
<<instantiate>>
KonkretnaFabrika
+fabrickiMetod()
fabrickiMetod(){
return new KonkretniProizvod();
}

- **Učesnici**
  - **Proizvod**
    - Definiše interfejs objekata koje **fabrickiMetod()** kreira
  - **KonkretanProizvod**
    - Implementira interfejs **Proizvod**
  - **Fabrika**
    - Deklariše **fabrickiMetod()** koji vraća objekat tipa Proizvod
    - Može da definiše podrazumevanu implementaciju za **fabrickiMetod()**, koja mora da vrati podrazumevani objekat tipa **KonkretanProizvod**
    - Iz neke operacija poziva **fabrickiMetod()** da kreira objekat tipa **Proizvod**
  - **KonkretnaFabrika**
    - Redefiniše(ili preklapa) **fabrickiMetod()** tako da kreira objekat tipa **KonkretanProizvod**

Saradnja

- **Fabrika** delegira svojim potklasama da definišu **fabrickiMetod()**, tako da on vraća odgovarajući objekat tipa **KonkretanProizvod**

Posledice

- Ovaj projektni uzorak omogućava da se klijentski kod ne vezuje za specifične klase proizvoda
- **Klijent** radi sa interfejsom klase **Proizvod** pa može da radi sa bilo kakvom klasom **KonkretanProizvod**
- **Fabrički metod** omogućava da izvedene klase kreiraju proširenu verziju nekog objekta a da se pri tome klijentski kod promeni samo toliko što se novi **Fabrički metod** registruje.

Povezani uzorci

- **Apstraktna fabrika** se često implementira pomoću **Fabričkog metoda**

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 1: Fabrički metod implementiran virtuelnom funkcijom


```
using namespace std;
class INalivPero{
public:
        virtual string Pisi()=0;
};
class NalivPeroZlatno: public INalivPero{
public:
        string Pisi(){return "Zlatnim vrhom";}
};
class NalivPeroCelicno: public INalivPero{
public:
        string Pisi(){return "Celicnim vrhom";}
};
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 1: Fabrički metod implementiran virtuelnom funkcijom


```
class IFabrikaNalivPera{
public:
/* Kroz hijerarhiju klasa se redefinise cisto virtuelna funkcija */
    virtual INalivPero* NapraviNalivPero()=0;
};

class FabrikaZlatnihNalivPera: public IFactoryNalivPera{
public:
     INalivPero * NapraviNalivPero(){
        return new NalivPeroZlatno();
    }
};

class FabrikaCelicnihNalivPera: public IFactoryNalivPera{
public:
     INalivPero * NapraviNalivPero(){
        return new NalivPeroCelicno();
    }
};
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 1: Fabrički metod implementiran virtuelnom funkcijom


```
int main(){

    IFabrikaNalivPera* f = new FabrikaZlatnihNalivPera();
    INalivPero *p = f->NapraviNalivPerio();
    cout<<"\n Pisi: "<<p->Pisi()<<"\n";
    delete p;
    delete f;

    f = new FabrikaCelicnihNalivPera();
    p = f->NapraviNalivPero();
    cout<<"\n : "<<p->Pisi()<<"\n";
    delete p;
    delete f;
    return 0;
}
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 2: Fabrički metod implementiran šablonom globalne funkcije


```
struct Zlatno {};
struct Celicno {};

template < typename TipMaterijala > class NalivPero {};

template <>
class NalivPero<Zlatno> {/* Specijalizacija klase NalivPero */
public: void Pisi() const;
};
void NalivPero<Zlatno>::Pisi() const {
   std::cout << "Zlatnim vrhom" << std::endl;
}

template <>
class NalivPero<Celicno> { /* Specijalizacija klase NalivPero */
public: void Pisi() const;
};
void NalivPero<Celicno>::Pisi() const {
   std::cout << "Celicnim vrhom" << std::endl;
}
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 2: Fabrički metod implementiran šablonom globalne funkcije


```
/* Fabricki metod */
template < typename TipMaterijala >
NalivPero< TipMaterijala > *NapraviNalivPero(){
     return new NalivPero< TipMaterijala >();
}

//Klijent
int main()
{
/* Eksplicitno instanciranje funkcije iz sablona */
  NalivPero<Zlatno> *ptr = NapraviNalivPero<Zlatno>();
  ptr->Pisi();
  delete ptr;

  return 0;
}
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 3: Fabrički metod implementiran šablonom statičke funkcije šablona klase


```
class INalivPero{
public: virtual string Pisi()=0;
};

class NalivPeroZlatno: public INalivPero{
public: string Pisi(){return "Zlatnim vrhom";}
};

class NalivPeroCelicno: public INalivPero{
public: string Pisi(){return "Celicnim vrhom";}
};
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 3: Fabrički metod implementiran šablonom statičke funkcije šablona klase


```
template <typename T>
class Fabrika {
public:
/* Funkcija koja sluzi za registrovanje fabrickih metoda za razlicite
   tipove TIzveden koji moraju da nasledjuju tip T */
   template <typename TIzveden>
   void registrujTip(int IDTipa) {
/* Staticka provera (u toku kompajliranja da li je Tizveden zaista
   izveden iz T */
        static_assert(is_base_of<T, TIzveden>::value,
               "TIzveden nije izveden iz tipa T");
        fabrickiMetod[IDTipa] = &FabrickiMetod<TIzvedena>;
   }
....
private:
....
};
```

```
/* Primer implementacije structure is_base_of */

template<typename O, typename I>
struct is_base_of{
private:
   static I* Test(O*);
   static char Test(...);
public:
   static const bool value = sizeof(Test((I*)0))== sizeof(I*) ;
};
/* (I*)0 - konvertuje 0 u pokazivac tipa I
Ukoliko je I tip izveden iz O bice pozvana funkcija Test(O*),
Ukoliko nije izveden iz O, bice pozvana funkcija Test(…)
sizeof(Test(static_cast<I*>(0))) - vraca velicinu (u broju bajtova)
rezultata funkcije, a to moze da bude broj bajtova adrese I* (4 ili 8)
ili broj bajtova char tipa (1).
Ukoliko je I tip koji nije izveden iz O bice vracen rezultat 1 koji je
razlicit od 4 ili 8, pa value ima vrednost false, u suprotvnom ako je
I tip izveden iz O, value dobija vrednost true. */
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 3: Fabrički metod implementiran šablonom statičke funkcije šablona klase


```
template <typename T>
class Fabrika {
public:
....
/*Javna funkcija koja je zaduzena za pravljenje objekta na osnovu
  identifikatora njegovog tipa IDTipa. Preko ove funkcije Klijent
  trazi od fabrike da pozove odgovarajucu fabricki metod za tip koji
  je on identifikovao kljucem, tj. Identifikatorom IDTipa */

  T* napravi(int IDTipa) {
       if (nizFabMetoda[IDTipa] != nullptr)
/* Poziv fabricke metode preko pointera na fabricku metodu
  nizFabMetoda[IDTipa]  */
              return nizFabMetoda[IDTipa]();
       return nullptr;
  }
private:
....
};
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 3: Fabrički metod implementiran šablonom statičke funkcije šablona klase


```
template <typename T>
class Fabrika {
public:
....
private:
/* Staticki sablon Fabrickog Metoda, koji nije deo interfejsa vec
   predstavlja detalj implementacije sablona Fabrike. Instancira se
   prilikom registrovanja za razlicite tipove objekata, koji moraju da
   budu izvedeni iz T */
   template <typename TIzveden>
   static T* FabrickiMetod() {
         return new TIzveden();
   }
   /* Definisanje novog imena tipa pokazivaca na funkciju koja vraca T*
   i nema parametre */
   typedef T* (*PtrTipFabrickiMetod)();
   Array<PtrTipFabrickiMetod> nizFabMetoda; /* Niz u kome se cuvaju
   pokazivaci na instanciarane fabricke metode */
};
```

FABRIČKI METOD (engl. FACTORY METHOD)
Primer 3: Fabrički metod implementiran šablonom statičke funkcije šablona klase

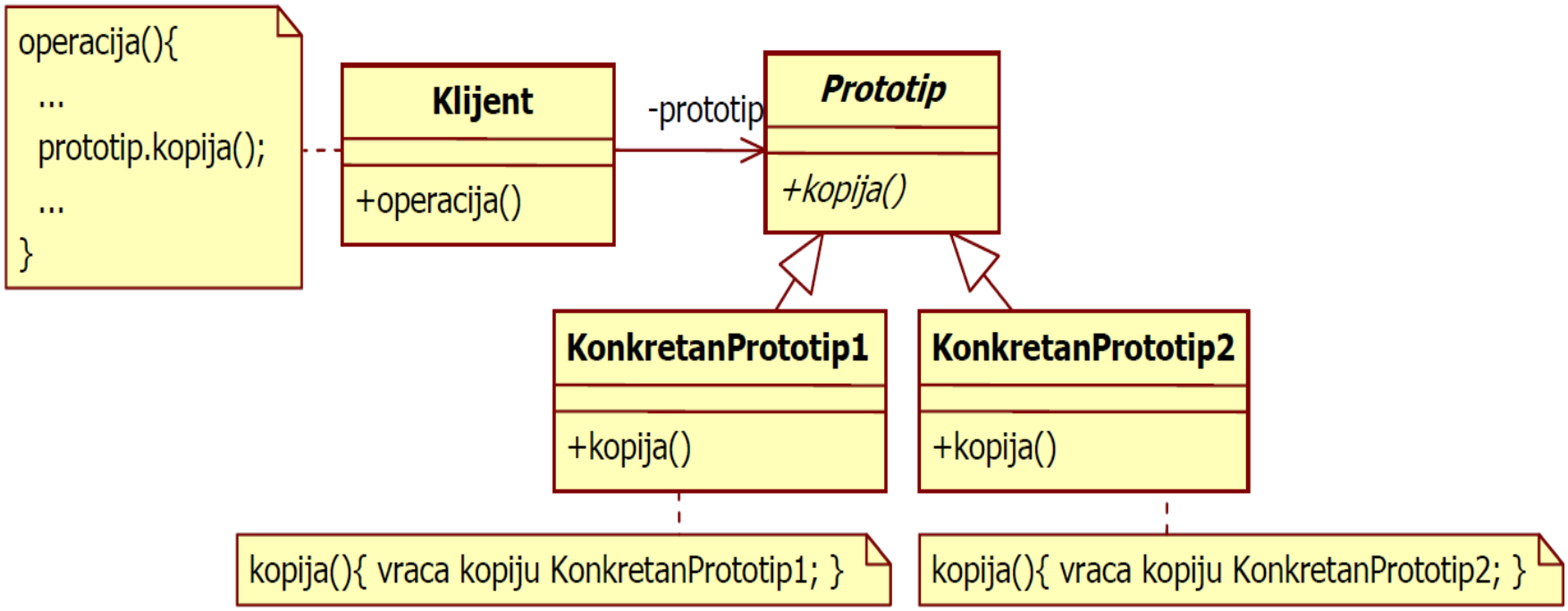

```
/* Klijent instancira fabriku. Registruje razlicite tipove, sto ima za
posledicu instanciranje odgovarajucih fabrickih metoda i njihovo
smestanje u niz na odgovarajucu poziciju */
int _tmain(int argc, _TCHAR* argv[]) {
  /* Instanciranje fabrike koja koristi fabricke metode */
  Factory<INalivPero> fabrika;

  fabrika.registrujTip<NalivPeroZlatno>(0);
  /* 0 je identifikator tipa I istovremeno pozicija u nizu gde ce biti
  smesten pokazivac na staticku fabricku metodu instanciranu za klasu
  objekata NalivPeroZlatno */

  fabrika.registrujTip<NalivPeroCelicno>(1);
  /* 1 je identifikator tipa I istovremeno pozicija u nizu gde ce biti
  smesten pokazivac na staticku fabricku metodu instanciranu za klasu
  objekata NalivPeroCelicno */
```

```
/* Klijent instancira fabriku. Registruje razlicite tipove, sto ima za
posledicu instanciranje odgovarajucih fabrickih metoda i njihovo
smestanje u niz na odgovarajucu poziciju */

int _tmain(int argc, _TCHAR* argv[]) {
  Factory<INalivPero> fabrika;
  fabrika.registrujTip<NalivPeroZlatno>(0);
  fabrika.registrujTip<NalivPeroCelicno>(1);

  INalivPero *b1 = fabrika.napravi(0);
  INalivPero *b1 = fabrika.napravi(1);

  return 0
}
```

PROTOTIP (engl. PROTOTYPE)

- **Ime i klasifikacija**
  - **Prototip (polimorfna kopija) (engl. Prototype)**
  - **Stvaralački projektni uzorak**
- **Drugo ime**
  - ***Virtuelni konstruktor kopije (engl. Virtual Copy Constructor)***
- **Namena**
  - **Specificira vrste objekata koji će biti kreirani kloniranjem prototipova**
  - **Kreira nove objekte kloniranjem prototipova**
- **Primenljivost**
  - **Inicijalno kreiranje objekat je skupa operacija i zahteva slanje upita bazi, složena numerička izračunavanja, inteligentno izdvajanje informacija iznanja iz podataka, signala i dokumenata.**

Struktura


operacija(){
...
prototip.kopija();
...
}
Klijent
+operacija()
-prototip
Prototip
+kopija()
KonkretanPrototip1
+kopija()
KonkretanPrototip2
+kopija()
kopija(){ vraca kopiju KonkretanPrototip1; }
kopija(){ vraca kopiju KonkretanPrototip2; }

- **Učesnici**
  - **Prototip**
    - Deklariše interfejs za sopstveno kloniranje
  - **KonkretanPrototip**
    - Implementira operaciju sopstvenog kloniranja deklarisanu u interfejsu **Prototip**
  - **Klijent**
    - Kreira novi objekat slanjem zahteva prototipu da se klonira
- **Saradnja**
  - Klijent zahteva od prototipa da se klonira

- **Posledice**
  - **Prednosti**
    - Dodavanje i uklanjanje prototipova u vreme izvršavanja
  - **Nedostaci**
    - Svaka podklasa mora da implementira **clone()**
- **Saradnja**
  - ApstraktnaFabrika je alternativni uzorak, ali može da bude dopunjena prototipom. Apstraktna Fabrika može da sadrži skup prototipova koje klonira i tako kreira nove proizvode

PROTOTIP (engl. PROTOTYPE)
Primer 1:


```
// Pen.h
class Pen{
   string type;
   static Pen* protoArray[];
public:
   Pen(const string &ty):type(ty){}
   Pen(const Pen &other):type(other.type){}
   virtual ~Pen() {}
   virtual Pen* clone() const = 0;
   virtual void Write() const { cout << "\nVrh pera " << type << endl;
}
   static Pen* create(int idx);
   static Pen* addPrototype(int idx, Pen* p);
   static void destroyPrototypes();
};
// Pen.cpp
Pen* Pen::protoArray[10];
```

PROTOTIP (engl. PROTOTYPE)
Primer 1:


```
// Pen.cpp
Pen* Pen::create(int idx){
   Pen* proto;
   if (proto = protoArray[idx])
      return proto->clone();
   return nullptr;
}

Pen* Pen::addPrototype(int idx, Pen* p){
   protoArray[idx] = p;
   return p;
}

void Pen::destroyPrototypes(){
   for (int i = 0; i < 10; ++i)
      delete protoArray[i];
}
```

PROTOTIP (engl. PROTOTYPE)
Primer 1:


```
//FountainPen.h
class FountainPen : public Pen{
public:
   FountainPen(const string &ty) :Pen(ty){}
   FountainPen(const FountainPen& other) : Pen(other){}
   virtual ~FountainPen() {}
   virtual Pen* clone() const;
   virtual void Write() const {
      cout << "\nNaliv pero " << endl;
      Pen::Write();
   }
};


//FountainPen.cpp
Pen* FountainPen::clone() const{
   return new FountainPen(*this);
}
```

PROTOTIP (engl. PROTOTYPE)
Primer 1:


```
// RollerBallPen.h
class RollerBallPen : public Pen{
public:
   RollerBallPen(const string &ty) :Pen(ty){}
   RollerBallPen(const RollerBallPen& other) : Pen(other){}
   virtual ~RollerBallPen() {}
   virtual Pen* clone() const;
   virtual void Write() const {
      cout << "\nRoller " << endl;
      Pen::Write();
   }
};


// RollerBallPen.cpp
Pen* RollerBallPen::clone() const{
   return new RollerBallPen(*this);
}
```

PROTOTIP (engl. PROTOTYPE)
Primer 1:


```
int _tmain(int argc, _TCHAR* argv[]){
   Pen::addPrototype(0, new FountainPen("Golden 14K"));
   Pen::addPrototype(1, new FountainPen("Golden 21K"));
   Pen::addPrototype(2, new RollerBallPen("0.7"));

   Pen* ptrPen = Pen::create(1);
   ptrPen->Write();
   delete ptrPen;

   ptrPen = Pen::create(2);
   ptrPen->Write();
   delete ptrPen;

   Pen::destroyPrototypes();
   return 0;
}
```

PROTOTIP (engl. PROTOTYPE)
Primer 2: Izbegavanje ponovljenog redefinisanja virtuelne metode clone (primenom šablona)


```
template <class TDerivedFromClonablePen>
class ClonablePen : public Pen {
public:
   ClonablePen(const string &ty) :Pen(ty){}

   ClonablePen(const ClonablePen<TDerivedFromClonablePen> &other)
      :Pen(other){}


   virtual Pen* clone() const {
   return new
         TDerivedFromClonablePen(*(TDerivedFromClonablePen*)this);

   /* Konverzija u prethodnoj liniji je moguca
      jer ce this zaista u trenutku konverzije ukazivati
      na objekta klaste TDerivedFromPen */
   }
};
```

PROTOTIP (engl. PROTOTYPE)
Primer 2: Izbegavanje ponovljenog redefinisanja virtuelne metode clone (primenom šablona)


```
class BallPointPen : public ClonablePen<BallPointPen>{
public:
   BallPointPen(const string &ty) : ClonablePen<BallPointPen>(ty){}

   BallPointPen(const BallPointPen& other) :
                        ClonablePen<BallPointPen>(other){}

   virtual ~BallPointPen() {}

   virtual void Write() const {
      cout << "\nOlovka " << endl;
      Pen::Write();
   }

   /*    Ova klasa implementira metodu clone
         tako sto je nasledjuje iz instance sablona
         ClonablePen<BallPointPen>                    */

};
```

PROTOTIP (engl. PROTOTYPE)
Primer 3: Izbegavanje ponovljenog redefinisanja virtuelne metode clone (primenom MAKROA)


```
// Pen.h    Dodajemo ovaj makro u Pen.h fajl
#define IMPLEMENT_CLONE(TYPE) \
   Pen* clone() const { return new TYPE(*this); }

// MechanicalPencil.h
// Definisemo novu klasu na sledeci nacin
class MechanicalPencil : public Pen{
public:
   MechanicalPencil(const string &ty) :Pen(ty){}
   MechanicalPencil(const MechanicalPencil& other) : Pen(other){}
   virtual ~MechanicalPencil() {}
   virtual void Write() const {
      cout << "\nTehnicka olovka " << endl;
      Pen::Write();
   }
   IMPLEMENT_CLONE(MechanicalPencil)
   /* Preprocesor direktno kopira telo metode clone iz makroa
IMPLEMENT_CLONE, pri cemu TYPE menja konkretnim tipom MechanicalPencil
*/
};
```

```
template<typename T>
class Buffer {
  int dim, end;
  T *ptr;
public:
  Buffer(int d) :dim(d), end(0), ptr(new T[dim]) {}
  Buffer(const Buffer &other) :dim(other.dim), end(other.end),
ptr(new T[dim]) {
    for (int idx = 0; idx < end; ++idx) ptr[idx] = other.ptr[idx];
  }
  ~Buffer() { delete [] ptr; }
  void push_back(T elem) { ptr[end++] = elem; }
  T& operator[] (int idx) { return ptr[idx]; }
  const T& operator[] (int idx) const { return ptr[idx]; }
  int End() const { return end; }
  int Dim() const { return dim; }
};
```

```
class IPerson{
public:
  virtual IPerson* Clone() const = 0;
  IPerson(const string& sName, int id):m_sName(sName), m_ID(id) {}
  IPerson(const IPerson& person) {
    this->m_sName = person.m_sName;
    this->m_ID = person.m_ID;
  }
  void SetName(const string& sName) { m_sName = sName; }
  void SetID(int ID) { m_ID = ID; }
  virtual ~IPerson() { cout << "~IPerson" << endl; }
private:
  string m_sName;
  int m_ID;
};
```

```
class Student : public IPerson{
public:
  Student(const string& sName, int id) : IPerson(sName, id){}
  Student(const Student& student) : IPerson(student){}
  IPerson* Clone() const { return new Student(*this);}
  virtual ~Student() { cout << "~Student" << endl; }
};
class Teacher : public IPerson{
public:
  Teacher(const string& sName, int id) :IPerson(sName, id){}
  Teacher(const Teacher& teacher) :IPerson(teacher){}
  IPerson* Clone() const { return new Teacher(*this); }
  virtual ~Teacher() { cout << "~Teacher" << endl; }
};
```

```
class University{
public:
  University(const string& sName) : m_sName(sName), buff(20){}
  University(const University& univ) : m_sName(univ.m_sName),
                                       buff(univ.buff.Dim()) {
    for (int idx = 0; idx < buff.End(); ++idx)
       buff.push_back(univ.buff[idx]->Clone());
  }
  void AddMember(IPerson* ptr) { buff.push_back(ptr); }
private:
  Buffer< RCPointer<IPerson> > buff;
  string m_sName;
};
```

```
int main(){

  University* pUniversity = new University("Nis");

  pUniversity->AddMember(new Student("Marko Markovic", 1));
  pUniversity->AddMember(new Student("Stevan Stevanovic", 1));
  pUniversity->AddMember(new Teacher("Petar Petrovic", 1));

  University* pUniversity2 = new University(*pUniversity);

  return 0;
}
```