

Objektno orijentisano programiranje u C++-u

Projektni uzorci

1

PROJEKTNI UZORCI PONAŠANJA OBJEKATA

STANJE
STRATEGIJA
SABLON

❑ Ime i klasifikacija

- ❑ Stanje (engl. State)
- ❑ Projektni uzorak ponašanja objekata

❑ Namena

- ❑ Omogućava objektu da menja svoje ponašanje kada se promeni njegovo unutrašnje stanje
- ❑ Izgleda kao da objekat menja svoju klasu

❑ Drugo ime

- ❑ Objekat za stanja (engl. Objects for States)

❑ Primenljivost

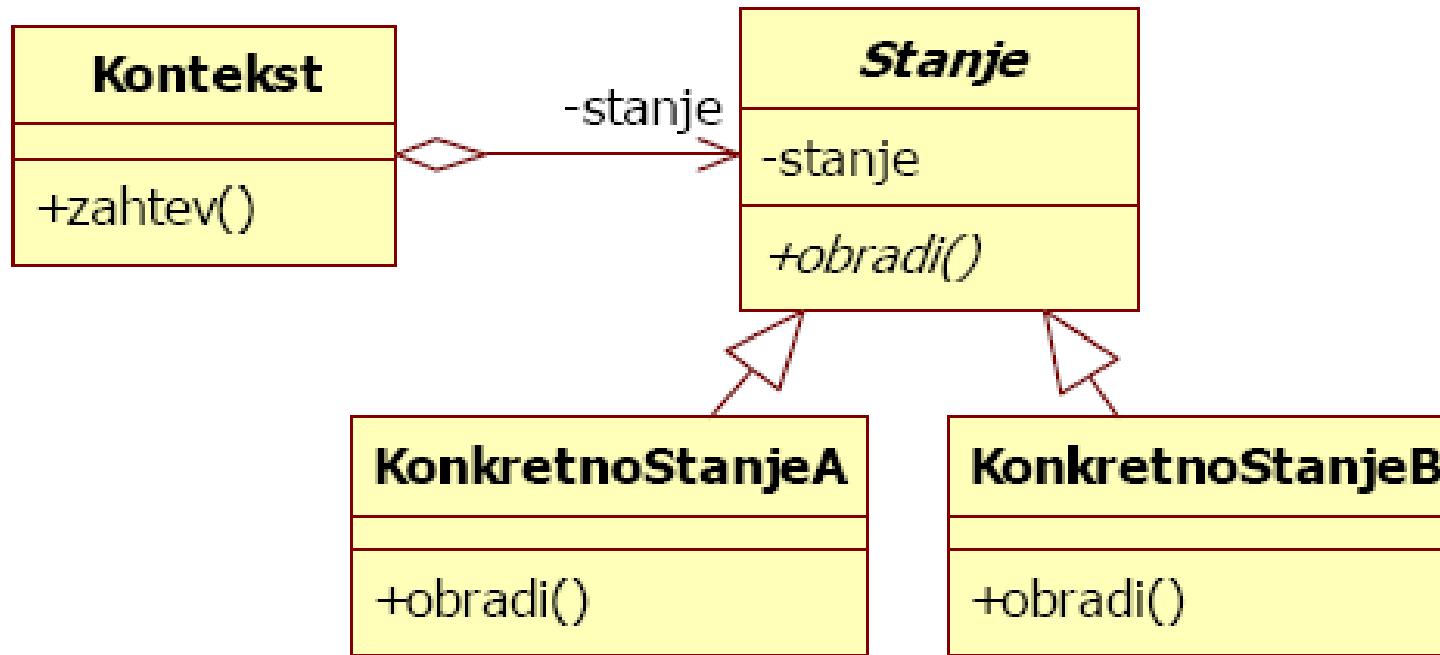
- ❑ Kada ponašanje objekata zavisi od stanja i mora da se menja u vreme izvršavanja
- ❑ Kada operacije imaju velike uslovne naredbe sa više grana, čije izvršenje zavisi od stanja objekata
 - ❑ Često više operacija sadrži istu uslovnu naredbu
 - ❑ Uzorak Stanje pravi od svake grane posebnu klasu koja određuje ponašanje operacije u datom stanju

STANJE(engl. STATE)

1. Uvod

4

□ Struktura



❑ Učesnici

❑ Kontekst

- ❑ Definiše interfejs od interesa za klijenta
- ❑ Sadrži referencu na objekat klase KonkretnoStanje

❑ Stanje

- ❑ Definiše interfejs za pojedina stanja
- ❑ Inkapsulira ponašanje pridruženog stanju objekta klase Kontekst

❑ KonkretnoStanje

- ❑ Implementira interfejs stanja

❑ Saradnja

- ❑ Kontekst prosleđuje zahteve koji su zavisni od stanja, tekućem objektu konkretnog stanja
- ❑ Kontekst može da prosledi konkretnom objektu stanja referencu na sebe, što omogućava objektu stanja da pristupi kontekstu (na primer da zahteva promenu stanja)
- ❑ Kontekst je primarni interfejs za klijente. Klijenti mogu da konfigurišu kontekst objektima stanja i posle toga ne moraju da interaguju direktno sa objektima stanja.

❑ Povezani uzorci

- ❑ Objekti stanja se često realizuju kao Unikati

□ Posledice

- **Dobra inkasulacija ponašanja specifičnog za stanje**
- **Jasno razdvajanje ponašanja u raličitim stanjima**
- **Jednostavno dodavanje novih stanja definisanjem novih potklasa stanja**
 - Nefleksibilna alternativa su uslovni iskazi rasuti svuda po kodu konteksta
- **Robusno rešenje – prelazi između stanja su eksplisitni i jednostavni**
 - Kada kontekst samostalno definiše svoje stanje vrednostima atributa
 - Prelazi izmedju stanja nemaju eksplisitnu reprezentaciju
 - Predstavljaju se dodelama vrednosti atributa
 - Objekti stanja štite objekat konteksta od nekonzistentnih stanja
 - Prelazi su jednostavni iz perspektive konteksta
 - Prelaz se svodi na prevezivanje jednog pokazivača
- **Objekti stanja mogu da budu deljeni**
 - Zahtev može da stigne do kraja lanca i da ne bude obrađen

□ Posledice

□ **Objekti stanja mogu da budu deljeni**

- Ako objekti stanja nemaju attribute, konteksti mogu da dele objekat stanja
- Tada su objekti stanja “MUVEĆ, tj. imaju samo ponašanje

STANJE(engl. STATE)

Primer 1.

9

```
class IStanjeBankomata{
public:
    virtual void UbaciKarticu() = 0;
    virtual void IzbaciciKarticu() = 0;
    virtual void UnesiPin_PodigniNovac() = 0;
    void PosaljiPoruku(string poruka) const {cout<< poruka << endl;}
    virtual ~IStanjeBankomata() {}
};

class UbacenaKartica: public IStanjeBankomata{
public:
    void UbaciKarticu(){
        PosaljiPoruku("Kartica ubacena. Ne mozete da ubacite drugu
... ");
    }
    void IzbaciciKarticu(){
        PosaljiPoruku("Kartica izbacena... ");
    }
    void UnesiPin_PodigniNovac(){
        PosaljiPoruku("Pin unet korektno. Preuzmite novac... ");
    }
};
```

STANJE(engl. STATE)

Primer 1.

10

```
class NemaKartice: public IStanjeBankomata{
public:
    void UbaciKarticu() {
        PosaljiPoruku("Kartica ubacena. Kucajte pin...") ;
    }
    void IzbaciciKarticu() {
        PosaljiPoruku("Nema kartice u automatu. Ne mozete da izbacite
karticu.");
    }
    void UnesiPin_PodigniNovac() {
        PosaljiPoruku("Nema kartice u automatu. Ne mozete da kucate
pin. ");
    }
};
```

STANJE(engl. STATE)

Primer 1.

11

```
/* KONTEKST */
#include <typeinfo>
class Bankomat: public IStanjeBankomata{
private:
    SmartPtr<IStanjeBankomata> stanje;
public:
    explicit Bankomat():stanje(new NemaKartice()) {}
    virtual ~Bankomat() {}
    SmartPtr<IStanjeBankomata> TrenutnoStanje() const {
        return stanje;
    }
    void PrebaciNa(const SmartPtr<IStanjeBankomata> &novoStanje) {
        stanje = novoStanje;
    }

    void UbaciKarticu(){
        stanje->UbaciKarticu();
        if(typeid(*stanje) == typeid(NemaKartice))
            PrebaciNa(new UbacenaKartica());
    }
    . . .
}
```

STANJE(engl. STATE)

Primer 1.

12

```
/* KONTEKST */
class Bankomat: public IStanjeBankomata{
private:
    SmartPtr<IStanjeBankomata> stanje;
public:
    explicit Bankomat():stanje(new NemaKartice()) {}
    virtual ~Bankomat() {}
    . . .
    void IzbaciKarticu(){
        stanje->IzbaciKarticu();
        if(typeid(*stanje) == typeid(UbacenaKartica))
            PrebacNa(new NemaKartice());
    }
    void UnesiPin_PodigniNovac(){
        stanje->UnesiPin_PodigniNovac();
    }
};
```

STANJE(engl. STATE)

Primer 1.

13

```
class IStanjeBankomata{
public:
    virtual void UbaciKarticu() = 0;
    virtual void IzbaciKarticu() = 0;
    virtual void UnesiPin_PodigniNovac() = 0;
    void PosaljiPoruku(string poruka) const {cout<< poruka << endl;}
    virtual ~IStanjeBankomata() {}
};

class Bankomat;

class UbacenaKartica: public IStanjeBankomata{
public:
    void UbaciKarticu(Bankomat &b) {
        PosaljiPoruku("Kartica ubacena. Ne mozete da ubacite drugu
...");
    }
    void IzbaciKarticu(Bankomat &b) {
        PosaljiPoruku("Kartica izbacena...");
        b.PrebaciNa(new IzbacenaKartica());
    }
    void UnesiPin_PodigniNovac(Bankomat &b) {
        PosaljiPoruku("Pin unet korektno. Preuzmite novac...");
    }
};
```

STANJE(engl. STATE)

Primer 1.

14

```
class NemaKartice: public IStanjeBankomata{
public:
    void UbaciKarticu(Bankomat &b) {
        PosaljiPoruku("Kartica ubacena. Kucajte pin...") ;
        b.PrebaciNa(new KarticaUbacena());
    }
    void IzbaciciKarticu(Bankomat &b) {
        PosaljiPoruku("Nema kartice u automatu. Ne mozete da izbacite
karticu.");
    }
    void UnesiPin_PodigniNovac(Bankomat &b) {
        PosaljiPoruku("Nema kartice u automatu. Ne mozete da kucate
pin. ");
    }
};
```

STANJE(engl. STATE)

Primer 1.

15

```
/* KONTEKST */
class Bankomat: public IStanjeBankomata{
private:
    SmartPtr<IStanjeBankomata> stanje;
public:
    explicit Bankomat():stanje(new NemaKartice()) {}
    virtual ~Bankomat() {}
    SmartPtr<IStanjeBankomata> TrenutnoStanje() const {
        return stanje;
    }
    void Prebacina(const SmartPtr<IStanjeBankomata> &novoStanje) {
        stanje = novoStanje;
    }

    void UbaciKarticu(){
        stanje->UbaciKarticu(*this);
    }
    . . .
};
```

STANJE(engl. STATE)

Primer 1.

16

```
/* KONTEKST */
class Bankomat: public IStanjeBankomata{
private:
    SmartPtr<IStanjeBankomata> stanje;
public:
    explicit Bankomat():stanje(new NemaKartice()) {}
    virtual ~Bankomat() {}
    . . .
    void IzbaciKarticu(){
        stanje->IzbaciKarticu(*this);
    }

    void UnesiPin_PodigniNovac(){
        stanje->UnesiPin_PodigniNovac(*this);
    }
};
```

STANJE(engl. STATE)

Primer 1.

17

```
int _tmain(int argc, _TCHAR* argv[]){
    SmartPtr<Bankomat> bankomat = new Bankomat();

    bankomat->UnesiPin_PodigniNovac();
    bankomat->IzbaciKarticu();

    ankomat->UbaciKarticu();
    bankomat->UnesiPin_PodigniNovac();
    bankomat->UbaciKarticu();
    bankomat->IzbaciKarticu();

}
```

```
Nema kartice u automatu. Ne mozete da kucate pin.
Nema kartice u automatu. Ne mozete da izbacite karticu.
Kartica ubacena. Kucajte pin...
Pin unet korektno. Preuzmite novac...
Kartica ubacena. Ne mozete da ubacite drugu ...
```

❑ Ime i klasifikacija

- ❑ Strategija (engl. Strategy)
- ❑ Projektni uzorak ponašanja objekata

❑ Namena

- ❑ Definiše familiju algoritama, inkapsulirajući svaki
- ❑ Omogućava dinamičku (u toku izvršavanja koda) promenu algoritama

❑ Drugo ime

- ❑ Politika (engl. Policy)

❑ Primenljivost

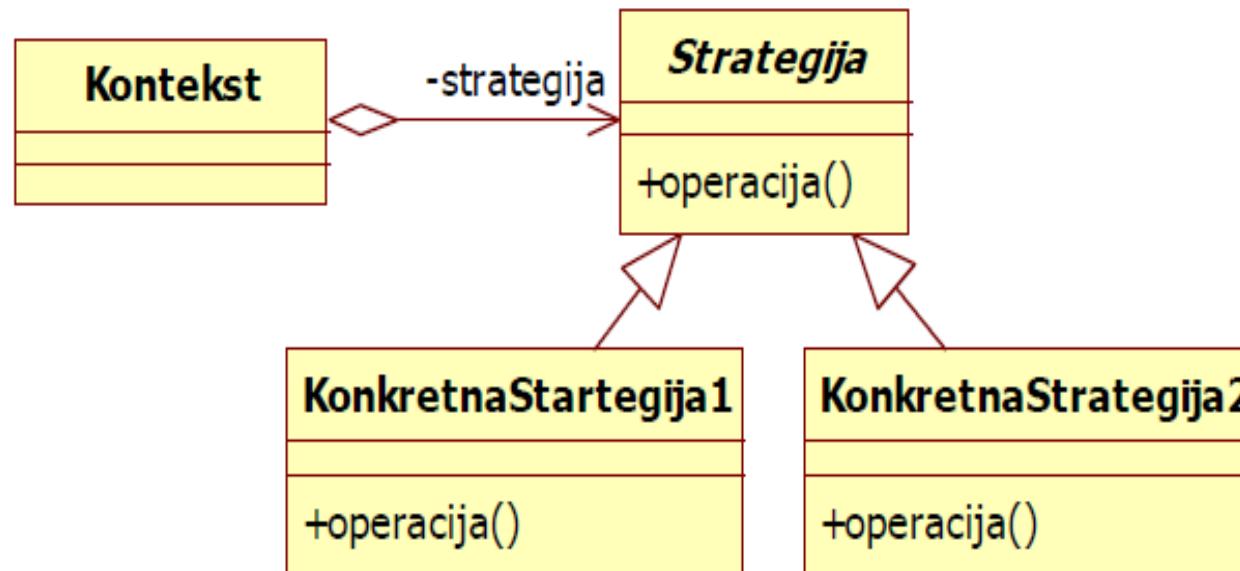
- ❑ Kada se više srodnih klasa razlikuje samo po ponašanju
 - ❑ Uzorak omogućava konfiguriranje jedne klase jednim od više ponašanja
- ❑ Kada su potrebne različite varijante nekog algoritma
- ❑ Kada algoritam koristi podatke o kojima klijent ne treba ništa da zna
 - ❑ Potrebna je inkapsulacija algoritama i struktura podatka
 - ❑ Često više operacija sadrži istu uslovnu naredbu
- ❑ Menja grane uslovne naredbe kada one predstavljaju različite operacije
 - ❑ Operacije iz grana uslovne naredbe se inkapsuliraju u strategije
 - ❑ Jedna strategija implementira jednu granu

STRATEGIJA(engl. STRATEGY)

1. Uvod

20

□ Struktura



❑ Učesnici

❑ Kontekst

- ❑ Definiše interfejs od interesa za klijenta
- ❑ Sadrži referencu na objekat klase KonkretnaStrategija

❑ Strategija

- ❑ Deklariše interfejs za strategije

❑ KonkretnaStrategija

- ❑ Implementira konkretni algoritam tako da odgovara interfejsu klase Strategija

❑ Saradnja

❑ Kontekst prosleđuje

- ❑ sve podatke koje zahteva algoritam pri pozivu objekta Strategija
- ❑ referencu na sebe i tako omogućava povratni poziv (callback)

❑ Kontekst prosleđuje zahteve svojih klijenata svom objektu strategije

- ❑ Klijenti obično kreiraju i prosleđuju objekte klase KonkretnaStrategija objektu klase Kontekst
- ❑ Kasnije klijenti interaguju samo sa objektom klase Kontekst

❑ Povezani uzorci

❑ Objekti Strategije često predstavljaju objekte Muve

□ Posledice

- **Familije srodnih algoritama definisane su kao hijerarhija klasa Strategije**
- **Strategije eliminisu potrebu za uslovnim naredbama u klijentskom kodu**
- **Mogućnost izbora implementacije**
 - Klijent bira implementaciju i postiže odgovarajuće performanse u vremenu i memorijskom prostoru
- **Nedostatak je što klijent mora da bude svestan strategija kojima može da parametrizuje objekat klase Kontekst**
- **Kontekst može da kreira parametre koji su nepotrebni za neke od KonkretnihStrategija**
- **Povećava se broj objekata u aplikaciji**

STRATEGIJA(engl. STRATEGY)

Primer 1.

24

```
class ILetenja{
public:
    virtual string leti() = 0;
    virtual ~ILetenja() {}
};

class Leti: public ILetenja{
    virtual string leti() { return "Letim visoko i brzo"; }
};

class NeLeti: public ILetenja{
    virtual string leti() { return "Ne mogu da letim"; }
};
```

STRATEGIJA(engl. STRATEGY)

Primer 1.

25

```
class Kicmenjaci{  
protected:  
    ILetenja *ptrAlgoLetenja;  
    string ime;  
public:  
    Kicmenjaci() : ptrAlgoLetenja(NULL) {}  
    void setIme(const string &novoIme) { ime = novoIme; }  
    const string getIme() { return ime; }  
    void setAlgoLetenja(ILetenja *pAlgoLetenja) {  
        clean();  
        ptrAlgoLetenja = pAlgoLetenja;  
    }  
    string leti(){ return ptrAlgoLetenja->leti(); }  
    virtual ~Kicmenjaci(){clean();}  
private:  
    void clean(){  
        if(ptrAlgoLetenja){  
            delete ptrAlgoLetenja;  
            ptrAlgoLetenja = NULL;  
        }  
    }  
}
```

STRATEGIJA(engl. STRATEGY)

Primer 1.

26

```
class Pas: public Kicmenjaci{  
public: Pas(){ setAlgoLetenja( new NeLeti() ) ; }  
};  
  
class Ptice: public Kicmenjaci{  
public: Ptice(){ setAlgoLetenja( new Leti() ) ; }  
};  
  
int main(int argc, char const *argv[]){  
    Kicmenjaci* ptrPas = new Pas();  
    Kicmenjaci* ptrSoko = new Ptice();  
    ptrPas->setIme("Pas");  
    ptrSoko->setIme("Soko");  
    cout << ptrPas->getIme().c_str() << ":" <<  
        ptrPas->leti().c_str() << endl;  
    cout << ptrSoko->getIme().c_str() << ":" <<  
        ptrSoko->leti().c_str() << endl;  
    delete ptrPas;  
    delete ptrSoko;  
    return 0;  
}
```

Pas: Ne mogu da letim
Soko: Letim visoko i brzo

STRATEGIJA(engl. STRATEGY)

Primer 2.

27

```
class PoreskiObveznik {  
public:  
    enum TipPO{ KOMPANIJA = 0, RADNIK = 1, KONZORCIJUM = 2};  
    static const double STOPA_KOMP;  
    static const double STOPA_RAD;  
    static const double STOPA_KONZ;  
private:  
    double prihod;  
    enum TipPO tip;  
public:  
    PoreskiObveznik(double p, TipPO t): prihod(p), tip(t) {}  
    double getPrihod() const { return prihod; }  
    double platiPorez() {  
        switch (tip) {  
            case KOMPANIJA: return prihod * STOPA_KOMP;  
            case RADNIK: return prihod * STOPA_RAD;  
            case KONZORCIJUM: return prihod * STOPA_KONZ;  
        }  
    }  
};
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

28

```
const double PoreskiObveznik::STOPA_KOMP = 0.30;
const double PoreskiObveznik::STOPA_RAD = 0.45;
const double PoreskiObveznik::STOPA_KONZ = 0.35;

int main(){
    PoreskiObveznik p(1000000, PoreskiObveznik::KOMPANIJA);
    cout<<p.platiPorez()<<endl;
    return 0;
}
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

29

```
/* KONVERZIJA SWITCH NAREDBE U STRATEGIJU */
class IPoreskaStrategija{
/*Inkapsuliranje algoritma i podataka
neophodnih za izvršavanje algoritma */
protected:
    double prihod;
public:
    IPoreskaStrategija(double p) :prihod(p) {}
    virtual double platiPorez() = 0;
    virtual double getPrihod() const {return prihod;}
};

class PoreskaStrategKomp: public IPoreskaStrategija{
    static const double STOPA;
public:
    PoreskaStrategKomp(double p) :IPoreskaStrategija(p) {}
    double platiPorez(){ double tmp = prihod * STOPA; prihod-=tmp;
return tmp;}
};
const double PoreskaStrategKomp::STOPA = 0.3;
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

30

```
class PoreskaStrategRad: public IPoreskaStrategija{
    static const double STOPA;
public:
    PoreskaStrategRad(double p) :IPoreskaStrategija(p) {}
    double platiPorez(){ double tmp = prihod * STOPA; prihod-=tmp;
return tmp;}
;
const double PoreskaStrategRad::STOPA = 0.45;

class PoreskaStrategKonz: public IPoreskaStrategija{
    static const double STOPA;
public:
    PoreskaStrategKonz(double p) :IPoreskaStrategija(p) {}
    double platiPorez(){ double tmp = prihod * STOPA; prihod-=tmp;
return tmp;}
;
const double PoreskaStrategKonz::STOPA = 0.35;
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

31

```
int main() {  
    PoreskiObveznik p(new PoreskaStrategijaKomp(1000000));  
    cout<<p.platiPorez()<<endl;  
    return 0;  
}
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

32

```
struct STOPA_KOMP{  
    static const double value;  
};  
const double STOPA_KOMP::value = 0.3;  
  
struct STOPA_RAD{  
    static const double value;  
};  
const double STOPA_RAD::value = 0.45;  
  
struct STOPA_KONZ{  
    static const double value;  
};  
const double STOPA_KONZ::value = 0.35;
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

33

```
template<typename STOPA>
class PoreskaStrategija: public IPoreskaStrategija{
public:
    PoreskaStrategija(double p) :IPoreskaStrategija(p) {}
    double platiPorez() { double tmp = prihod * STOPA::value; prihod-
    =tmp; return tmp; }
};

class PoreskiObveznik{
    SmartPtr<IPoreskaStrategija> m_p;
public:
    PoreskiObveznik(IPoreskaStrategija *p) :m_p(p) {}
    void PromeniStrategiju(IPoreskaStrategija *p) { m_p = p; }
    double getPrihod() const { return m_p->getPrihod(); }
    double platiPorez() { return m_p->platiPorez(); }
};
```

STRATEGIJA(engl. STRATEGY)

Primer 2.

34

```
int main() {  
    PoreskiObveznik p(new PoreskaStrategija<STOPA_KOMP>(1000000));  
    cout<<p.platiPorez()<<endl;  
    return 0;  
}
```

Ime i klasifikacija

- Šablonski metod (engl. Template Method)
- Projektni uzorak ponašanja klase

Namena

- Definiše skelet nekog algoritma delegirajući pojedine korake potklasama
- Omogućava potklasama da redefinišu neke korake algoritma bez izmene njegove strukture

❑ Primenljivost

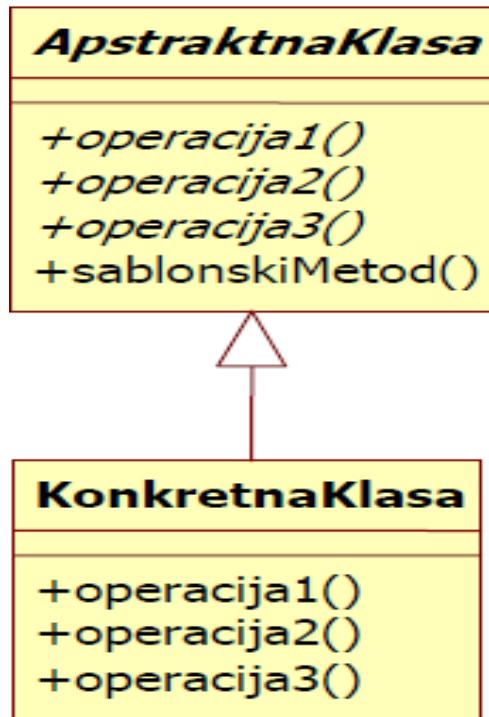
- ❑ Kada algoritam može da se implementira tako da se nepromenljivi delovi implementiraju jednom a delovi koji se menjaju se ostavljaju potklasama
- ❑ Kada zajedničko ponašanje grupe klasa treba izdvojiti, čime se izbegavaju dupliranja i greške u kodu

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

1. Uvod

37

□ Struktura



```
sablonskiMetod(){
    ...
    operacija1();
    ...
    operacija2();
    ...
    operacija3();
    ...
}
```



❑ Učesnici

❑ ApstraktnaKlase

- ❑ Implementira šablon (skelet) algoritma
 - ❑ Poziva primitivne operacije
- ❑ Deklariše apstraktne primitivne operacije za korake algoritma i delegira potklasama implementaciju tih primitivnih operacija

❑ KonkretnaKlasa

- ❑ Implementira primitivne operacije
 - ❑ Primitivne operacije predstavljaju delove algoritma
 - ❑ Delovi su specifični za potklase

❑ Saradnja

❑ KonkretnaKlasa implementira varijante koraka algoritma za klasu ApstraktnaKlase

❑ Posledice

- ❑ **Roditeljska klasa zove operacije dece (Neophodna konverzija pokazivača this)**
- ❑ **Primitivne operacije mogu da budu implementirane u osnovnoj klasi (tako se obezbeđuje podrazumevano ponašanje)**

❑ Povezani uzorci

- ❑ **Fabrički metod se često poziva iz Šablonskog Metoda**
- ❑ **ŠablonskiMetod menja deo algoritma prilikom kreiranja objekta, Strategija dinamički, u toku izvršavanja, menja ceo algoritam nekog objekta. Objekat u različitim fazama svog životnog veka može da koristi različite strategije.**

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

40

```
/* POGRESNA IMPLEMENTACIJA:  
Jabuke i Kruske implementiraju IComparable ali nisu poredive */  
class IComparable {  
private:  
    /* Privatne prijateljske funkcije */  
    friend bool operator <(const IComparable& levi, const IComparable& desni){  
        return levi.manje_od(desni);  
    }  
    friend bool operator >(const IComparable& levi, const IComparable& desni){  
        return desni.manje_od(levi);  
    }  
    friend bool operator <=(const IComparable& levi, const IComparable& desni){  
        return !desni.manje_od(levi);  
    }  
    friend bool operator >=(const IComparable& levi, const IComparable& desni){  
        return !levi.manje_od(desni);  
    }  
private:  
    /* Privatna striktno virtuelna funkcija */  
    virtual bool manje_od(const IComparable& drugi) const = 0;  
};
```

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

41

```
/* Konverzija pomocu static_cast operatora */
class Klasa : public IComparable {
public: Klasa(int ini) : vred(ini) {}
private:
    bool manje_od(const IComparable& drugi) const{
        const Klasa &tmp = static_cast<const Klasa&>(drugi);
        return vred < tmp.vred;
    }
    int vred;
};

class Osoba : public IComparable{
public: Osoba(string ime, unsigned godine) : m_ime(ime),
m_godine(godine){}
private:
    bool manje_od(const IComparable& drugi) const{
        const Osoba &tmp = static_cast<const Osoba&>(drugi);
        return m_godine < tmp.m_godine;
    }
    string m_ime;
    unsigned m_godine;
}, Projektni uzorci
```

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

42

```
/* Konverzija pomocu static_cast operatora - poredimo „KRUSKE I  
JABUKE“ */  
int main() {  
    Osoba ("Mika", 3) < Klasa (3);  
  
    Klasa (4) < Osoba ("Pera", 5);  
  
    return 0;  
}
```

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

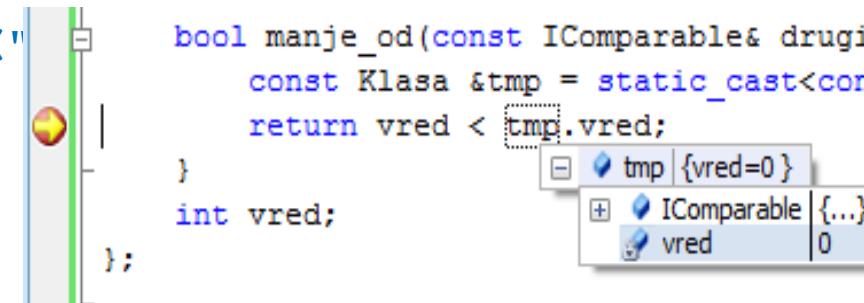
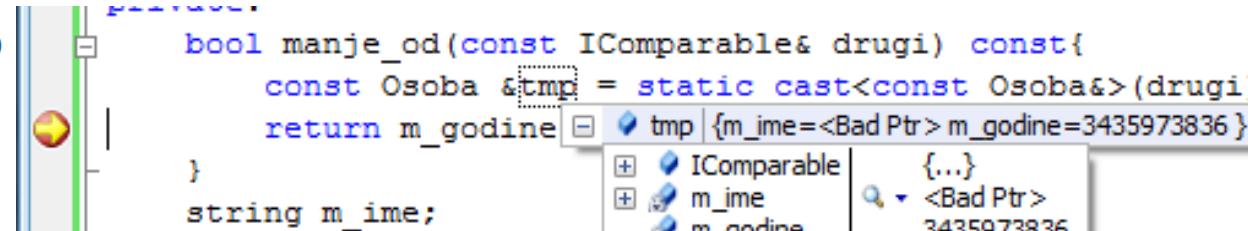
43

```
/* Konverzija pomocu static_cast operatora - poredimo „KRUSKE I JABUKE“ */
int main() {
    Osoba ("Mika", 3) // Kruska
        .manje_od(Klasa("Kruška", 4));
    Klasa("Kruška") // Jabuka
        .manje_od(Osoba("Mika", 3));
}

class Osoba : public IComparable {
public:
    bool manje_od(const IComparable& drugi) const {
        const Osoba &tmp = static_cast<const Osoba&>(drugi);
        return m_ime < tmp.m_ime;
    }
private:
    string m_ime;
    unsigned m_godine;
};

class Klasa : public IComparable {
public:
    bool manje_od(const IComparable& drugi) const {
        const Klasa &tmp = static_cast<const Klasa&>(drugi);
        return vred < tmp.vred;
    }
private:
    int vred;
};

return 0;
}
```



ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

44

```
/* Konverzija pomocu dynamic_cast operatora prosledjuje std::bad_cast */
class Klasa : public IComparable {
public:
    Klasa(int ini):vred(ini) {}
private:
    bool manje_od(const IComparable& drugi) const{
        const Klasa &tmp = dynamic_cast<const Klasa&>(drugi);
        return vred < dynamic_cast<const Klasa&>(drugi).vred;
    }
    int vred;
};

class Osoba : public IComparable{
public:
    Osoba(string ime, unsigned godine) : m_ime(ime), m_godine(godine) {}
private:
    bool manje_od(const IComparable& drugi) const{
        const Osoba &tmp = dynamic_cast<const Osoba&>(drugi);
        return m_godine < tmp.m_godine;
    }
    string m_ime;
    unsigned m_godine;
}
```

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

45

```
/* Konverzija pomocu dynamic_cast operatora prosledjuje std::bad_cast I  
on mora da se prihvati */  
int main(){  
    Osoba("Mika",3) < Klasa(3);  
  
    return 0;  
}
```

ŠABLONSKI METOD(engl. TEMPLATE METHOD)

Primer 2.

46

/* Konverzija pomocu dynamic_cast operatora prosledjuje std::bad_cast I on mora da se prihvati */

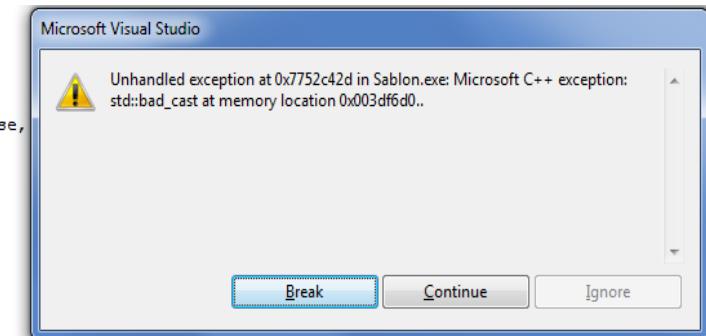
```
int main()
{
    Osoba ("Mik")
    {
        const char * szLineName,
        int nLine
    }

    int errno_tmp = 0;
    void * pvBlk = _nh_malloc_dbg_impl(nSize, nhFlag, nBlockUse,

    if ( pvBlk == NULL && errno_tmp != 0 && _errno())
    {
        errno = errno_tmp; // recall, #define errno *_errno()
    }
    return pvBlk;
}

/***/
```

```
return 0;
}
```



SIMULIRANO DINAMIČKO VEZIVANJE CURIOSLY RECURRING TEMPLATE PATTERN

47

```
template<typename Izvedena>
class Osnovna{
public:
    void Fun() { static_cast<Izvedena*>(this)->FunImpl(); }
    static void FunStatic() { Izvedena::FunStatic(); }
private:
    void FunImpl() {}
};

class KlasaA : public Osnovna<KlasaA> {
public:
    void FunImpl() { cout << "KlasaA::FunImpl" << endl; }
    static void FunStatic(){ cout << "KlasaA::FunStatic" << endl; }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE CURIOSLY RECURRING TEMPLATE PATTERN

48

```
template<typename Izvedena>
void CRTP_FunStatic() {
    Osnovna<Izvedena>::FunStatic();
}

template<typename Izvedena>
void CRTP_Fun(Osnovna<Izvedena>& r) {
    r.Fun();
}

int main ()
{
    KlasaA a;
    CRTP_FunStatic<KlasaA>();
    CRTP_Fun(a);
    a.Fun();
    return 0;
}
```

KlasaA::FunStatic
KlasaA::FunImpl
KlasaA::FunImpl

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 1: Brojac zivih objekata

49

```
/* BROJAC OBJEKATA SA GLOBALNOM STATICOM PROMENLJIVOM */
template <typename PrebrojavaniTip>
class BrojacObjekata {
private:
    static size_t broj; /* Broj zivih objekata */
protected:
    /* Podrazumevani konstruktor */
    BrojacObjekata() { ++BrojacObjekata<PrebrojavaniTip>::broj; }
    /* Konstruktor kopije */
    BrojacObjekata (const BrojacObjekata<PrebrojavaniTip>&) {
        ++BrojacObjekata<PrebrojavaniTip>::broj; }
    /* Destruktor */
    ~BrojacObjekata() { --BrojacObjekata<PrebrojavaniTip>::broj; }
public:
    /* Staticka funkcija koja vraca broj zivih objekata */
    static size_t BrojZivih() { return
        BrojacObjekata<PrebrojavaniTip>::broj; }
    ;
    /* Inicijalizacija statickog brojaca */
    template <typename PrebrojavaniTip>
    size_t BrojacObjekata<PrebrojavaniTip>::broj = 0;
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 1: Brojac zivih objekata



```
/* BROJAC OBJEKATA SA LOKALNOM STATICOM PROMENLJIVOM */
template <typename PrebrojavaniTip>
class BrojacObjekata {
private:
    /* broj zivih objekata */
    static size_t& broj() { static size_t broj; return broj; }
protected:
    /* podrazumevani konstruktor */
    BrojacObjekata() { ++BrojacObjekata<PrebrojavaniTip>::broj(); }
    /* konstruktor kopije */
    BrojacObjekata (const BrojacObjekata<PrebrojavaniTip>&) {
        ++BrojacObjekata<PrebrojavaniTip>::broj();
    }
    /* destruktor */
    ~BrojacObjekata() { --BrojacObjekata<PrebrojavaniTip>::broj(); }
public:
    /* staticka funkcija koja vraca broj zivih objekata */
    static size_t BrojZivih() { return
        BrojacObjekata<PrebrojavaniTip>::broj(); }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 1: Brojac zivih objekata

51

/* BROJAC OBJEKATA SA LOKALNOM STATICOM PROMENLJIVOM */

```
template< typename T >
class TKlasaA: public BrojacObjekata< TKlasaA <T> >
{};

int main()
{
    TKlasaA<int> a;
    {
        TKlasaA<int> b = a;
        cout<< a.BrojZivih() << endl;
    }
    cout<< a.BrojZivih() << endl;
    return 0;
}
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 2: Polimorfno kloniranje (Prva problematicna verzija)

52

```
class Vozilo{  
public:  
    virtual ~Vozilo() {}  
    virtual Vozilo *clone() const = 0;  
    virtual void TehSpec() const = 0;  
};  
  
/* Zadatak ovog sablona je da obezbedi algoritam za duboko kloniranje  
objekata klase koje su izvedene iz njegove instance. Sablon se  
instancira sa argumentom koji je upravo Izvedena klasa */  
  
template <typename Izvedena>  
class AutoKlonabilan : public Vozilo{  
public:  
    virtual Vozilo *clone() const {  
        return new Izvedena(static_cast<Izvedena const &>(*this));  
    }  
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 2: Polimorfno kloniranje (Prva problematicna verzija)

53

```
enum TipMenjaca{ eRUCNI = 0, eAUTOMATIK };

class Auto : public AutoKlonabilan<Auto>{
    TipMenjaca m_Menjac;
public:
    Auto(TipMenjaca m = eRUCNI) :m_Menjac(m) {}
    virtual void TehSpec() const{ std::cout << "Auto" << std::endl; }
};

/* Trazimo TrkackiAuto koji mora da ima osobine Auta a i da bude
klonabilan*/
class TrkackiAuto : public AutoKlonabilan<TrkackiAuto>{
/* Ocigledno da ovakav pristup ne vodi ka resenju */
public:
    virtual void TehSpec() const { std::cout << "Trkacki auto" <<
std::endl; }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 2: Polimorfno kloniranje (Druža, takodje problematicna verzija)

54

```
/* CRTP sa dva argumenta */
template <typename Osnovna, typename Izvedena>
class AutoKlonabilan : public Osnovna{
public:
    virtual Osnovna *clone() const {
        return new Izvedena(static_cast<const Izvedena &>(*this));
    }
};

enum TipMenjaca{ eRUCNI = 0, eAUTOMATIK };

/*Korektna implementacija klase Auto */
class Auto : public AutoKlonabilan<Vozilo, Auto>{
    TipMenjaca m_Menjac;
public:
    Auto(TipMenjaca m = eRUCNI) :m_Menjac(m) {}
    virtual void TehSpec() const{ std::cout << "Auto" << std::endl; }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 2: Polimorfno kloniranje (Pruga, takodje problematicna verzija)



```
/* POGRESNA IMPLEMENTACIJA KLASE TRKACKI AUTO */
/* C++03 ne omogucava nasledjivanje konstruktora */

class TrkackiAuto : public AutoKlonabilan<Auto, TrkackiAuto>{
public:
    /* Ne mozemo da zovemo konstruktor klase Auto, TkackiAuto je ne
nasledjuje direktno */
    /* AutoKlonabilan je sablon I ne moze da ima sve moguce
konstruktore */
    TrkackiAuto(TipMenjaca m) :AutoKlonabilan(m) {}
    virtual void TehSpec() const { std::cout << "Trkacki auto" <<
std::endl; }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 2: Polimorfno kloniranje (Treća verzija, korektna na C++11 kompjajlerima)

56

```
template <typename Osnovna, typename Izvedena>
class AutoKlonabilan : public Osnovna{
public:
    /* NASLEDJIVANJE KONSTRUKTORA OSNOVNE KLASE */

    using Osnovna::Osnovna;

    /* Za svaki konstruktor klase Osnovna:
       Osnovna(tip_1 param_1, ...)
       kompjajler napravi odgovarajucu verziju u AutoKlonabilan:
       AutoKlonabilan(tip_1 param_1,...) :Osnovna(param_1,...) {} */

    virtual Osnovna *clone() const {
        return new Izvedena(static_cast<const Izvedena &>(*this));
    }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 2: Polimorfno kloniranje (Treća verzija, korektna na C++11 kompjajlerima)

57

```
enum TipMenjaca{ eRUCNI = 0, eAUTOMATIK };

class Auto : public AutoKlonabilan<Vozilo, Auto>{
    TipMenjaca m_Menjac;
public:
    Auto(TipMenjaca m = eRUCNI) :m_Menjac(m) {}
    virtual void TehSpec() const{ std::cout << "Auto" << std::endl; }
};

class TrkackiAuto : public AutoKlonabilan<Auto, TrkackiAuto>{
public:
    TrkackiAuto(TipMenjaca m = eAUTOMATIK) :
        AutoKlonabilan<Auto, TrkackiAuto>(m) {}

    virtual void TehSpec() const { std::cout << "Trkacki auto" <<
std::endl; }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 3: Generički unikat

58

```
template<typename Izvedena>
class Unikat{
public:
    static Izvedena& GetInstance() {
        static Izvedena primerak;
        return primerak;
    }
protected:
    Unikat() {}
    Unikat(const Unikat&);
    Unikat& operator=(const Unikat&);
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 3: Generički unikat

59

```
class KlasaUnikat : public Unikat<KlasaUnikat>{
    friend Unikat<KlasaUnikat>;
    /* INACE NE BI MOGAO SABLON UNIKAT DA PRISTUPI PRIVATNOM (ili
     * ZASTICENOM) KONSTRUKTORU klase KlasaUnikat */
protected:
    KlasaUnikat() {}
    KlasaUnikat(const KlasaUnikat&);
    KlasaUnikat& operator=(const KlasaUnikat&);
};

int main (){
    KlasaUnikat::GetInstance();
}
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 4: Genericki unikat (C++ 11)



```
template <class Izvedena>
class Unikat {
public:
    template<typename... Args>
    static Izvedena& GetInstance(Args... args) {
        static Izvedena inst(std::forward<Args>(args)...);
        return inst;
    }
protected:
    Unikat() {}
    ~Unikat() {}
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 4: Genericki unikat (C++ 11)

61

```
class Par: public Unikat<Par>{
friend class Unikat<Par>;
/* INACE NE BI MOGAO SABLON UNIKAT DA PRISTUPI PRIVATNOM (ili
ZASTICENOM) KONSTRUKTORU */
private:
    Par(int levi, int desni) : m_levi{ levi }, m_desni{ desni } {}
public:
    int m_levi, m_desni;
};

int main(){
    Par &p = Par::GetInstance(4, 5);
    std::cout << p.m_desni << std::endl;
    return 0;
}
```

FUNKCIJE PRIJATELJI SABLONA

62

/*1. RESENJE KOJE DOZVOLJAVA da operator+<double> bude prijatelj klasa Skalar<int> */

```
template<class T>
class Skalar{
public:
    Skalar(T s = 0) : m_s(s) {}
    template<class U>
    friend Skalar<U> operator+(const Skalar<U>& levi,
                                const Skalar<U>& desni);
```

```
private:
    T m_s;
};
```

```
template<class T>
Skalar<T> operator+(const Skalar<T>& levi, const Skalar<T>& desni) {
    return levi.m_s + desni.m_s;
}
```

FUNKCIJE PRIJATELJI SABLONA

63

/*2. RESENJE KOJE DOZVOLJAVA da operator+<int> bude prijatelj klase Skalar<int> i SPRECAVA da operator+<double> bude prijatelj klase Skalar<int>*/

```
template<typename T>
class Skalar;
```

/* Deklaracija prijateljskih funkcija*/

```
template<typename T>
Skalar<T> operator+(const Skalar<T>&, const Skalar<T>&);
```

```
template<typename T>
Skalar<T> Fun(const Skalar<T>&);
```

FUNKCIJE PRIJATELJI SABLONA

64

```
/*2. RESENJE */
template<typename T>
class Skalar{
public:
    Skalar(T a = 0) : m_s(a) {}
    /* Konkretna instanca sablona funkcije za tip T se proglašava
     prijateljem sablona klase Skalar */
    friend Skalar operator+(const Skalar& levi, const Skalar& desni);
    friend Skalar Fun<T>(const Skalar& s);
private:
    T m_s;
};

/*Definicija sablona prijateljskih funkcija, koje će poslužiti za
instanciranje */
template<typename T>
Skalar<T> operator+(const Skalar<T>& levi, const Skalar<T>& desni) {
    return levi.m_s + desni.m_s;
}
template<typename T>
Skalar<T> Fun(const Skalar<T>& s) { return s.m_s + s.m_s; }
```

FUNKCIJE PRIJATELJI SABLONA

65

```
/* BINARNI OPERATOR PRIJATELJ SABLONA */
/*2. RESENJE */
int main(){
    Skalar<int> a(3), b(4);
    int i1 = 3, i2 = 4;

    Skalar<double> d = i1 + i2; /*Funkcionise bez problema */
```

FUNKCIJE PRIJATELJI SABLONA

66

```
/*2. RESENJE */
```

```
int main()
```

```
Skalar<int> a(3), b(4);  
int i1 = 3, i2 = 4;
```

```
Skalar<double> d = i1 + i2; /*Funkcionise bez problema kao obicno  
sabiranje integer-a i prosledjivanje celobrojne vrednosti konstruktoru  
Skalar<double> */
```

```
Skalar<int> c = i1 + a; /* error C2784: 'Skalar<T> operator +(const  
Skalar<T> &,const Skalar<T> &)' : could not deduce template argument for  
'const Skalar<T> &' from 'int' */
```

```
Fun(i1); /* error C2784: 'Skalar<T> Fun(const Skalar<T> &)' : could  
not deduce template argument for 'const Skalar<T> &' from 'int' */
```

FUNKCIJE PRIJATELJI SABLONA

67

```
/*3. RESENJE */
template<typename T>
class Skalar{
public:
    Skalar(T a = 0) : m_s(a) {}
    /* Operator i funkcija koji su prijatelji ali nisu sabloni i
    MORAJU da budu definisani unutar klase*/
    friend Skalar operator+(const Skalar& levi, const Skalar& desni){
        return levi.m_s + desni.m_s;
    }
    friend Skalar Fun(const Skalar& s){
        return s.m_s + s.m_s;
    }
private:
    T m_s;
};
```

FUNKCIJE PRIJATELJI SABLONA

68

```
/*3. RESENJE */
/* Operator + i funkcija Fun su AUTOMATSKI KREIRANI za svako
instanciranje sablona SKALAR onda kada se sablon instancira.
MORAJU da budu definisane unutar klase jer ne postoji, dok se ne
instancira konkretna klasa iz sablona Skalar u toku kompajliranja */

friend Skalar operator+(const Skalar& levi, const Skalar& desni){
    return levi.m_s + desni.m_s;
}

friend Skalar Fun(const Skalar& s) {
    return s.m_s + s.m_s;
}

private:
    T m_s;
};

/* ALI OVO ELEGANTO RESENJE IMA SVOJU CENU . . . */
```

FUNKCIJE PRIJATELJI SABLONA

69

```
/*3. RESENJE */
int main(){
    Skalar<int> a(3), b(4);
    int i1 = 3, i2 = 4;

    Skalar<int> c = i1 + a; /* FUNKCIONISE BEZ PROBLEMA */
    Skalar<int> d = b + i2; /* FUNKCIONISE BEZ PROBLEMA */

    Fun(i1); /* error C3861: 'Fun': identifier not found */

    Fun(Skalar<int>(i1)); /* FUNKCIONISE BEZ PROBLEMA */

}
```

C++ Argument Dependant Lookup (ADL) pretrazuje kroz funkcije koje nisu u trenutno oblasti ali postoje u klasama I prostorima imena na osnovu tipa njihovih argumenata, tj. vrednosti parametara prilikom poziva funkcija.

`c = i1 + a;` je uspesno prevedeno zato sto komajler “vidi” da je `a` tipa `Skalar<int>` i zato pokusava da implicitno konvertuje `i1` to `Skalar<int>`. Posto postoji konstruktor koji takvu konverziju podržava, poziv funkcije je uspešno prepoznat.

FUNKCIJE PRIJATELJI SABLONA

70

```
/*3. RESENJE */
int main(){
    Skalar<int> a(3), b(4);
    int i1 = 3, i2 = 4;

    Skalar<int> c = i1 + a; /* FUNKCIONISE BEZ PROBLEMA */
    Skalar<int> d = b + i2; /* FUNKCIONISE BEZ PROBLEMA */

    Fun(i1); /* error C3861: 'Fun': identifier not found */

    Fun(Skalar<int>(i1)); /* FUNKCIONISE BEZ PROBLEMA */

}
```

Medjutim `Fun(i1)` predstavlja poziv funkcije sa celebrojnim argumentom a kompjleru tip argumenta ne pomaže u pokušaju da nađe odgovarajuću implementaciju i odgovarajuće konverzije. Treće rešenje definitivno ne funkcioniše ukoliko ADL ne moze implicitno da zaključi kojoj oblasti funkcija pripada.

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 4: CRTP Comparable (Prelazno resenje)

71

```
template <typename Izvedena>
class Comparable {
public:
    friend bool operator >( const Comparable<Izvedena>& levi,
                             const Comparable<Izvedena>& desni)
    {
        const Izvedena& llevi = static_cast<const Izvedena&>(levi);
        const Izvedena& ddesni = static_cast<const Izvedena&>(desni);

        return ddesni < llevi;
    }
    friend bool operator <=( const Comparable<Izvedena>& levi,
                           const Comparable<Izvedena>& desni)
    {
        const Izvedena& llevi = static_cast<const Izvedena&>(levi);
        const Izvedena& ddesni = static_cast<const Izvedena&>(desni);

        return !(ddesni < llevi);
    }
    . . .
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 4: CRTP Comparable (Prelazno resenje)

72

```
template <typename Izvedena>
class Comparable {
public:

    friend bool operator >=( const Comparable<Izvedena>& levi,
                           const Comparable<Izvedena>& desni)
    {
        const Izvedena& llevi = static_cast<const Izvedena&>(levi);
        const Izvedena& ddesni = static_cast<const Izvedena&>(desni);

        return !(llevi > ddesni);
    }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 4: CRTP Comparable (Prelazno resenje)

73

```
template<typename T>
class Klasa : public Comparable< Klasa<T> > {
public:
    Klasa(T ini) : vred(ini) {}
private:
    friend bool operator <(const Klasa<T>& levi, const Klasa<T>& desni) {
        return levi.vred < desni.vred;
    }
    T vred;
};

class Osoba : public Comparable<Osoba>{
public:
    Osoba(string ime, unsigned godine) : m_ime(ime), m_godine(godine) {}
private:
    friend bool operator <(const Osoba& levi, const Osoba& desni) {
        return levi.m_godine < desni.m_godine;
    }
    string m_ime;
    unsigned m_godine;
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 4: CRTP Comparable (Prelazno resenje)

74

```
int main() {  
    Klasa<int>(3) < Klasa<int>(4);
```

```
    Klasa<int>(4) >= Osoba("Pera", 5);
```

```
/* error C2678: binary '>=' : no operator found which takes a  
left-hand operand of type 'Klasa<T>' (or there is no acceptable  
conversion) */
```

```
}
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 5: CRTP Comparable (Konačno resenje)

75

```
template <typename Izvedena>
class Comparable {
private:
    friend bool operator >(const Izvedena& levi, const Izvedena& desni) {
        return desni < levi;
    }

    friend bool operator <=(const Izvedena& levi, const Izvedena& desni) {
        return !(desni < levi);
    }

    friend bool operator >=(const Izvedena& levi, const Izvedena& desni) {
        return !(levi < desni);
    }
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 5: CRTP Comparable (Konačno resenje)

76

```
template<typename T>
class Klasa : public Comparable< Klasa<T> > {
public:
    Klasa(T ini) : vred(ini) {}
private:
    friend bool operator <(const Klasa<T>& levi, const Klasa<T>& desni) {
        return levi.vred < desni.vred;
    }
    T vred;
};

class Osoba : public Comparable<Osoba>{
public:
    Osoba(string ime, unsigned godine) : m_ime(ime), m_godine(godine) {}
private:
    friend bool operator <(const Osoba& levi, const Osoba& desni) {
        return levi.m_godine < desni.m_godine;
    }
    string m_ime;
    unsigned m_godine;
};
```

SIMULIRANO DINAMIČKO VEZIVANJE

CURIOSLY RECURRING TEMPLATE PATTERN

Primer 5: CRTP Comparable (Konacno resenje)

77

```
int main() {
    Klasa<int>(3) <= Klasa<int>(4); /* BEZ PROBLEMA */

    Klasa<int>(3) >= Klasa<int>(4); /* BEZ PROBLEMA */

    Klasa<int>(4) >= Osoba("Pera", 5); /* error C2678: binary '>=' : no
operator found which takes a left-hand operand of type 'Klasa<T>' (or
there is no acceptable conversion) */

}
```