Objektno orijentisano programiranje u C++-u
Projektni uzorci

STVARALAČKI PROJEKTNI UZORCI

UNIKAT

APSTRAKTNA FABRIKA

PROTOTIP

Uvod

Projektni uzorci, obrasci ili šabloni predstavljaju opšta, isprobana i ponovo iskoristiva programerska rešenja koja mogu da se primene na uobičajene probleme u (objektno orijentisanom) razvoju softvera.

Osnovni elementi projektnog uzorka su:

- naziv uzorka,
- postavka problema,
- opis rešenja i
- diskusija posledica

Klasifikacija uzoraka

Klasifikacija uzoraka doprinosi boljem i bržem snalaženju prilikom traženja odgovarajućeg uzorka, a istovremeno i usmerava napore ka otkrivanju novih uzoraka.

Za klasifikaciju uzoraka koriste se dva kriterijuma:

- kriterijum namene, koji razvrstava uzorke u zavisnosti od toga čime se bave
  - kreiranjem objekata (*creational patterns)*
  - strukturom, tj. kompozicijom klasa i objekata (*structural patterns)*
  - ponašanjem, tj načinom interakcije objekata i klasa (*behavioral patterns)*
- kriterijum domena, koji razvrstava uzorke zavisno od toga da li se primarno primenjuju na klase ili objekte
  - klasni uzorci koji se fokusiraju na relacije između klasa i podklasa (*classpatterns)*
  - objektni uzorci koji se fokusiraju na relacije između objekata (*object patterns)*

Naziv uzorka

Naziv uzorka se koristi da opiše projektni problem, njegova rešenja i konsekvence u par reči.

Uvođenje naziva uzorka:

- omogućava projektovanje na višem nivou apstrakcije
- pojednostavljuje komunikaciju u timu
- olakšava dokumentovanje projekta

Ponekad se u literaturi sreće više naziva za isti uzorak.

Postavka problema i opis rešenja

Postavka problema:

- objašnjava problem i njegov kontekst
- opisuje u kojim situacijama se razmatrani uzorakprimenjuje
- daje listu uslova koji se moraju ispuniti da bi se mogao primeniti dati uzorak
- opisuje strukture klasa i objekata

Opis rešenja:

- opisuje elemente koji predstavljaju dizajn,njihove relacije, odgovornosti i saradnje
- ne opisuje konkretan dizajn ili implementaciju, jer uzorak treba da posluži kao šablon koji se može primeniti u konkretnim slučajevima

Posledice

Posledice su rezultati primene projektnog uzorka. One su kritične za ispitivanje projektnih alternativa i razumevanje cene i dobiti zbog primene uzorka.

- Posledice se obično odnose na:
  - prostor
  - vreme
  - jezik
  - implementacione detalje
- Posledice utiču na:
  - fleksibilnost sistema
  - proširivost sistema
  - portabilnost sistema

Katalog projektnih uzoraka (I)

Za svaki uzorak, **katalog sadrži sledeće stavke:**

- Ime uzorka i klasifikacija
- Namena – odgovori na pitanja "šta radi?“, “čemu služi? “ i koji problem rešava? “
- Drugi nazivi – isti uzorak može imati više naziva
- Motivacija – scenario koji ilustruje projektni problem
- Primenljivost – situacije u kojima se uzorak može primeniti
- Struktura –grafička reprezentacija klasnih i objektnih dijagrama koji opisuju uzorak
- Učesnici – klase i objekti koji učestvuju u uzorku i njihove odgovornosti
- Kolaboracije – kako učesnici sarađuju da bi ispunili svoje odgovornosti

Katalog projektnih uzoraka (I)

Uzorci kreiranja	
Unikat	Obezbeđuje da klasa ima samo jednu instancu i daje globalni pristup toj instanci.
Prototip	Specificira vrste objekata koji se kreiraju korišćenjem prototipske instance i kreira nove objekte kopiranjem prototipa.
Fabrički metod	Definiše interfejs za kreiranje objekata, ali ostavlja potklasama da odluče čije objekte kreiraju. Uzorak dopušta klasi da delegira stvaranje objekta potklasi.
Apstraktna fabrika	Obezbeđuje interfejs za kreiranje familija povezanih ili zavisnih objekata bez specificiranja konkretnih klasa familije objekata.
Graditelj	Razdvaja konstrukciju kompleksnog objekta od njegove reprezentacije tako da isti proces može da kreira različite reporezentacije.
Lenja inicijalizacija	Taktički odlaže stvaranje objekta, izračunavanje neke vrednosti ili drugi skup proces dok on prvi put ne bude potreban.

Katalog projektnih uzoraka (I)

Uzorci strukture	
Kompozicija	Komponuje objekte u strukturu stabla (hijerarhija celina-deo). Kompozicija omogućava klijentima da uniformno tretiraju i individualne objekte i njihove kompozicije.
Dekorater	Dinamički dodaje mogućnosti nekom objektu. Dekorater predstavlja fleksibilnu alternativu izvođenju za proširivanje funkcionalnosti.
Muva	Deljenje malih objekata (objekata bez stanja) da bi se izbegla hiperprodukcija objekata.
Adapter	Konvertuje interfejs klase u drugi interfejs koji klijenti očekuju. Adapter omogućava rad zajedno klasa koje inače to ne bi mogle zbog različitog interfejsa.
Fasada	Pruža jedinstven interfejs skupu različitih interfejsa nekog podsistema. Fasada definiše interfejs višeg nivoa da bi se podsistem lakše koristio.
Proksi	Realizuje zamenu (surogat) drugog objekta koji kontroliše pristup originalnom objektu.
Most	Razdvaja apstrakciju od njene implementacije da bi se mogle nezavisno menjati.

Katalog projektnih uzoraka (I)

Uzorci ponašanja	
Posmatrač	Definiše zavisnost 1:N između objekata, takvu da kada jedan objekat promeni stanje svi zavisni objekti budu obavešteni i automatski se ažuriraju.
Iterator	Obezbeđuje sekvencijalni pristup elementima nekog agregatnog objekta bez eksponiranja unutrašnje strukture tog agregata.
Strategija	Definiše familiju algoritama, enkapsulirajući svaki i čini ih međusobno zamenjivim. Strategija omogućava jednostavnu promena algoritma u vreme izvršenja.
Šablonski metod	Definiše kostur nekog operacionog algoritma, delegirajući pojedine korake potklasama. Šablonski metod omogućava potklasama da redefinišu određene korake algoritma bez izmene njegove strukture.
Stanje	Omogućava objektu da menja svoje ponašanje kada se menja njegovo unutrašnje stanje. Izgleda kada da objekat menja svoju klasu.
Podsetnik	Bez narušavanja enkapsulacije snima i eksternalizuje stanje nekog objekat, tako da omogući da se objekat kasnije može vratiti u dato stanje.
Posrednik	Definiše objekat koji enkapsulira kako skup objekata interaguje. Posrednik omogućava slabo sprezanje objekata što postiže čuvanjem objekata koji se međusobno referišu, a to dozvoljava da im se interakcija menja nezavisno.
Komanda	Enkapsulira zahtev u jedan objekat, omogućavajući da se klijenti parametrizuju različitim zahtevima, da se zahtevi isporučuju kroz red čekanja, da se pravi dnevnik zahteva i da se efekti izvršenog zahteva poništе.
Lanac odgovornosti	Izbegava neposredno vezivanje pošiljaoca zahteva sa primaocem zahteva, dajući šansu većem broju objekata da obrade zahtev. Lanac odgovornosti povezuje objekte primaoce zahteva u lanac i prosleđuje zahtev niz lanac dok ga neki objekat ne obradi.

Katalog projektnih uzoraka (II)

- Posledice – diskusija dobrih i loših strana primene uzorka
- Implementacija – preporuke i tehnike kojih treba biti svestan pri implementaciji
- Primer koda – deo koda koji pokazuje kako se uzorak može implementirati
- Poznate primene – primeri primene uzorka u realnim sistemima
- Korelisani uzorci – uzorci bliski sa datim, razlike među njima
- UML notacija – grafički simbol za saradnju koja realizuje uzorak

Unikat (engl. Singleton)

Ime uzorka i klasifikacija

- *Singleton – stvaralački objektni projektni uzorak*

Namena

- Obezbeđuje da klasa ima samo jednu instancu i daje globalni pristup toj instanci

Motivacija

- Za neke klase je važno obezbediti da imaju samo po jednu instancu (u sistemu može da postoji više štampača, ali treba da postoji samo jedan dispečer zadataka štampanja).
- Da li je rešenje globalna promenljiva? Ne! Globalna promenljiva obezbeđuje globalni pristup objektu, ali ne sprečava kreiranje više objekata. Bolje rešenje je da klasa bude sama odgovorna za jedinstvenost svoje instance (objekta)

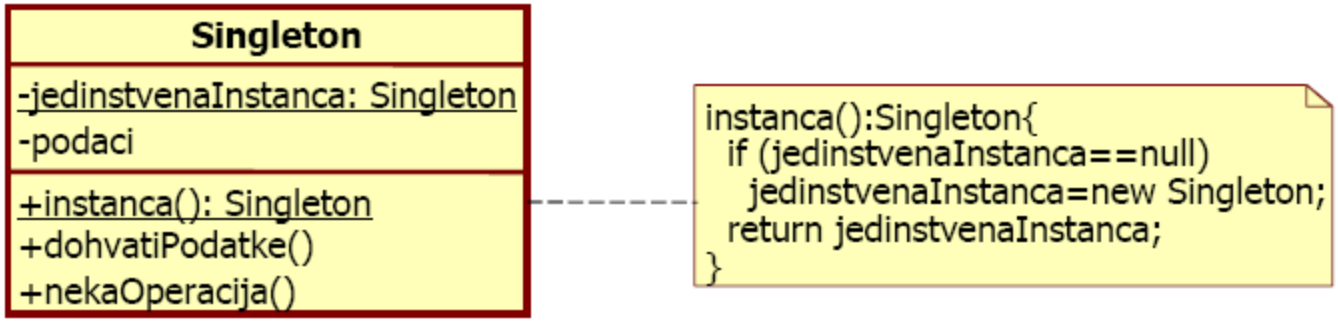
Unikat (engl. Singleton)

Primenljivost

- *Singleton uzorak se koristi kada:*
  - Mora postojati tačno jedna instanca klase i ona mora biti pristupačna klijentima preko svima poznate tačke pristupa
  - Ta jedina instanca može da se proširuje potklasama i treba da važi da klijenti mogu da koriste instancu izvedene klase bez potrebe da modifikuju svoj kod

Unikat (engl. Singleton)

Struktura



Učesnici

- Singleton klasa
  - Definiše operaciju instanca()koja omogućava klijentima pristup do jedinstvene instance.
  - Može biti odgovorna za kreiranje vlastite instance

Saradnja

- **Klijenti pristupaju objektu *Singleton klase isključivo kroz klasnu operaciju* instanca() (engl. Instance())**

Unikat (engl. Singleton)

Posledice

- Kontrolisani pristup do jedine instance
  - pošto klasa Singleton (sa slike) enkapsulira jedinu instancu, ona može imati striktnu kontrolu nad tim kako i kada klijenti pristupaju instanci
- Redukovan prostor imena
  - *Singleton uzorak je bolji koncept od globalne promenljive; on izbegava* opterećivanje prostora imena globalnom promenljivom koja čuva jedinu instancu

UNIKAT (engl. SINGLETON)
Primer 1:Implementacija pomoću **globalnog statičkog objekta**


```
/* Deklaracija Impl1.h fajl */
class Unikat {
public:
    static Unikat* instanca(){
    return &s_Objekat;
  }
  virtual void Operacija();
protected:
/*Zasticeni zbog
nasledjivanja*/
  Unikat(){}
  ~Unikat(){}
private:
/* Privatni jer ne smeju da se
koriste */
  Unikat(const Unikat&);
  Unikat& operator=(const
Unikat&);
    static Unikat s_Objekat;
};
```

```
/* Implementacija Impl1.cpp fajl
*/

Unikat Unikat::s_Objekat;

int Unikat::Operacija(){
  cout<<"Operacija nad
unikatom"<<endl;
  return 0;
}
```

```
/* Moj fajl.cpp  */
int global=
Unikat::instanca()->Operacija();
```

UNIKAT (engl. SINGLETON)
Primer 1:Implementacija pomoću **globalnog statičkog objekta**

PROBLEMI

- s_Objekat je statička promenljiva ali je korisnički definisanog tipa (ima konstruktor) i inicijalizovana u toku izvršavanja, pozivom konstruktora.
- C++ standard ne definiše redosled inicijalizacije dinamički inicijalizovanih statičkih objekata koji se nalaze u različitim translacionim jedinicama (fajlovima) i to predstavlja ozbiljan izvor problema.
- Primer:

```
/* Moj fajl.cpp  */
int global = Unikat::instanca()->Operacija();
```

- Ne postoji garancija da će s_Objekat da bude inicijalizovan pre nego što ga upotrebimo za inicijalizaciju vrednosti statičke promenljive global.

UNIKAT (engl. SINGLETON)

Primer 2:Implementacija pomoću globalnog **statičkog pokazivača na objekat**


```
/* Deklaracija Impl2.h fajl */

class Unikat {
public:
  static Unikat* instanca();
  virtual void Operacija();
protected:
/*Zasticene zbog nasledjivanja*/
  Unikat(){}
  ~Unikat(){}
private:
/* Privatne, jer ne smeju da se
koriste */
  Unikat(const Unikat&);
  Unikat& operator=(const
                        Unikat&);
  static Unikat* s_pObjekat;
};
```

```
/* Implementacija Impl2.cpp fajl
*/

Unikat* Unikat::s_pObjekat = 0;

Unikat* Unikat::instanca() {
  if (!s_pObjekat)
    s_pObjekat = new Unikat;
  return s_pObjekat;
}

void Unikat::Operacija(){
  cout<<"Operacija nad
unikatom"<<endl;
}
```

PREDNOSTI

- `s_pObjekat` je statička primitivna promenljiva bez konstruktora, inicijalizovana compile-time konstantom. Kompajler izvodu statičku inicijalizaciju pre nego što se izvrši prva asemblerska naredba programa. (Obično se statički inicijalizatori nalaze u fajlu u kome se nalazi izvršni kod programa, tako da samo učitavanje u memoriju pokreće inicijalizaciju).

- Konstruktor kopije i operator dodele vrednosti koje kompajler dodaje ukoliko ih korisnik ne deklariše. Da bi sprečili da ih kompajler doda (i implementira) OBAVEZNO moramo da ih deklarišemo kao privatne, da bi smo garantovali unikatnost objekta klase.

UNIKAT (engl. SINGLETON)
Primer 2:Implementacija pomoću **globalnog statičkog pokazivača na objekat**

- Da bi sprečili da korisnik Unikat-a slučajno izbriše unikatni objekat, destruktor proglašavamo zaštićenim.

- Ako se unikatni objekat nikad ne koristi (nikada nije pozvana funkcija instanca), unikatni objekat se nikada ne kreira. Prednost u slučajevima kada je njegovo kreiranje skupo a retko ili nikad se ne koristi.

PROBLEMI

- Iako ne prouzrokuje curenje memorije, Implementacija 2, prouzrokuje curenje resursa.

- Implementacija 2 kreira singleton na zahtev ali ga nigde ne uništava. Zašto ovo nije curenje memorije?
- Nema akumulacije podataka i gubljenja referenci na njih. Referenca na signleton se čuva do kraja izvršavanja aplikacije. Moderni operativni sistemi izvode kompletnu dealokaciju memorije procesa posle njegovog završavanja.

- Postoji curenje resursa. Konstruktor Unikata može da zahteva neograničen skup mrežnih resursa i konekcija na primer.
- Jedini pravilan način da se izbegne curenje resursa je da se izbriše Unikatni objekat u toku gašenja aplikacije. Problem je u tome što mora da se odabere pravilno trenutak brisanja, kako bi izbegli slučaj da neko pokušava da koristi izbrisani unikat.

UNIKAT (engl. SINGLETON)
Primer 3:Implementacija **pomoću lokalnog statičkog objekta**


```
// Deklaracija Impl3.h fajl
class Singleton{
public:
    static Singleton& get_instance();
    virtual void Operation ();
protected:
    Singleton(){}
    ~Singleton(){}
private:
     Singleton(const Singleton&);
     Singleton& operator=(const Singleton&);
};

// Implementacija Impl3.cpp fajl
Singleton& Singleton:: get_instance() {
//lokalna staticka promenljiva
    static Singleton inst;
    return inst;
}
void Singleton::Operacija() { cout<<"Operacija nad unikatom"<<endl;}
```

PREDNOSTI

- Jednostavna elegantna implementacija, oslanja se na karakteristike kompajlera. Statički objekat funkcije je inicijalizovan kada tok kontrole izvršavanja prvi put predje preko njegove definicije.
- U slučaju da se radi o primitivnim statički promenljivama, koje su inicijalizovane compile-time konstantama, one se inicijalizuju pre nego što se bilo koja linija koda izvrši, (zavisi od implementacije kompajlera) najčešće još u toku učitavanja programa u memoriju.

```
int Fun(){
        static int x = 100;
        return x;
}
```

- Kada inicijalizator nije compile-time konstanta, ili je statička promenljiva objekat sa konstruktorom, promenljiva se inicijalizuje u toku izvršavanja kada kontrola toka prvi put predje preko njene definicije.

PREDNOSTI

- Kada inicijalizator nije compile-time konstanta, ili je statička promenljiva objekat sa konstruktorom, promenljiva se inicijalizuje u toku izvršavanja kada kontrola toka prvi put predje preko njene definicije.
- Osim toga, kompajler generiše kod koji omogućava da se posle inicijalizacije, promenljiva regustruje u listu za uništavanje.

UNIKAT (engl. SINGLETON)
Primer 3:Implementacija **pomoću lokalnog statičkog objekta**


```
// Pseudo C++ kod Instance funkcije
Singleton& Singleton::Instance(){
  // Funkcije koje generiše kompajler
  extern void __ConstructSingleton(void* memory);
  extern void __DestroySingleton();
  // Promenljive koje generiše kompajler
  static bool __initialized = false;
  // Bafer koji čuva unikat
  static char __buffer[sizeof(Singleton)];
  if (!__initialized){
       // Prvi poziv, konstruiše objekat pozivajući konstruktor
  Singleton::Singleton
       __ConstructSingleton(__buffer);
       // Registrovanje uništavanja
       atexit(__DestroySingleton);
       __initialized = true;
  }
return *reinterpret_cast<Singleton *>(__buffer);
}
```

PROBLEMI(C++03 standard) Implementacija 3 nije thread safe

Pretpostavimo da je dat sledeći kod:

```
int RacunajNesto(){
  static int statRez = RacunajNestoZahtevno();
  return statRez;
}
```

- Prethodni kod koji sadrži inicijalizaciju statičke promenljive sa lokalnom oblašću važenja, kompajler konvertuje u sledeći (napomena sledi generički pseudo kod)

```
int RacunajNesto(){
  static bool statRez_computed = false; //Generise je kompajler
  static int statRez;
  if (! statRez_computed ) {
        statRez_computed = true;
        statRez = RacunajNestoZahtevno();
  }
  return statRez;
}
```

PROBLEMI(C++03 standard) Implementacija 3 nije thread safe

- Pretpostavimo da dve niti (engl. threads pozivaju funkciju RacunajNestoZahtevno po prvi put) . Prva nit dolazi do inicijalizacije promenljive **statRez_computed = true**, i biva privremeno prekinut od strane nekog zadatka većeg prioreiteta (engl. pre-empted) .
- Druga niti vidi da je **statRez_computed = true** i preskače if granu i vraća kao rezultat praznu statičku promenljivu.
- Ovakvo ponašenje je definisano standardom do verzije **C++ 11(ne uključujući i ovu verziju)**.

U verziji C++11 standarda implementacija 3 je thread safe što se inicijalizacije statčike lokalne prom tiče

- Deo C++11 standarda "If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization."
- „If multiple threads attempt to initialize the same static local variable concurrently, the initialization occurs exactly once (similar behavior can be obtained for arbitrary functions with std::call_once)."
- Ukoliko kompajler implementira C++11 standard kompletno prothodni problem sa inicijalizacijom statičke promenljive ne postoji.

UNIKAT (engl. SINGLETON)
Primer 4: Generički unikat


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

UNIKAT (engl. SINGLETON)
Primer 4: Generički unikat


```
/* ANALIZA KODA: Zasto kompajler prijavljuje gresku? */
class KlasaUnikat : public Unikat<KlasaUnikat>{



protected:
    KlasaUnikat() {}
private:
    KlasaUnikat(const KlasaUnikat&);
    KlasaUnikat& operator=(const KlasaUnikat&);
};


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

UNIKAT (engl. SINGLETON)
Primer 4: Generički unikat

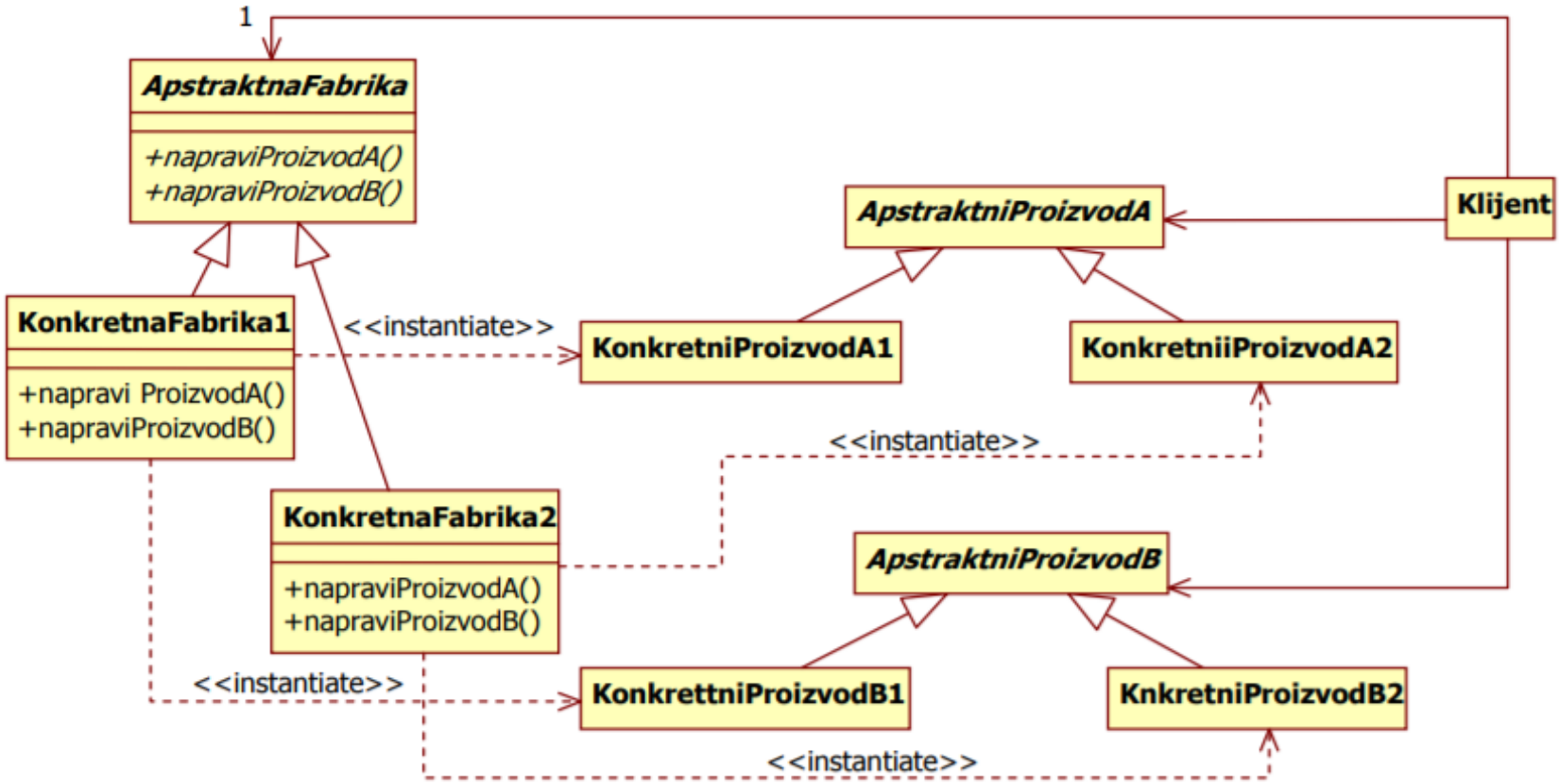

```
class KlasaUnikat : public Unikat<KlasaUnikat>{
  friend Unikat<KlasaUnikat>;
  /* INACE NE BI MOGAO SABLON Unikat DA PRISTUPI PRIVATNOM (ili
ZASTICENOM) KONSTRUKTORU KLASE KlasaUnikat */
protected:
    KlasaUnikat() {}
private:
    KlasaUnikat(const KlasaUnikat&);
    KlasaUnikat& operator=(const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


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

APSTRAKTNA FABRIKA (engl. Abstract Factory)

- Ime i klasifikacija
  - Apstraktna fabrika (engl. Abstract Factory)
  - Projektni objektni uzorak kreiranja
- Namena
  - Obezbedjuje interfejs za kreiranje različitih familija povezanih objekata (tzv. proizvoda)
  - Klijent “ne vidi” konkretne klase te familije proizvoda
- Drugo ime
  - Kit

APSTRAKTNA FABRIKA (engl. Abstract Factory)

- Struktura:



APSTRAKTNA FABRIKA (engl. Abstract Factory)

Učesnici

- ApstraktnaFabrika
  - Deklariše interfejs sa operacije koje kreiraju objekte apstraktnih proizvoda
- KonkretnaFabrika
  - Implementira operacije koje kreiraju objekte konkretnih proizvoda
- ApstraktniProizvod
  - Deklariše interfejs za odredjen tip proizvoda
- KonretanProizvod
  - Implementira interfejs apstraktno proizvoda
  - Definiše proizvod koji će biti kreiran pomoću odgovarajuće fabrike
- Klijent
  - Koristi fabrike i objekte preko interfejsa deklarisanog u apstraktnoj fabrici i apstraktnom proizvodu

APSTRAKTNA FABRIKA (engl. Abstract Factory)

Saradnja

- ApstraktnaFabrika odlaže kreriranje proizvoda do njenih potklasa – kokretnih fabrika
- Uobičajeno je da konkretne fabrike imaju samo po jednu instancu, tj. da su unikati (engl. Singleton)

APSTRAKTNA FABRIKA (engl. Abstract Factory)

Posledice

- Izolovanje konkretnih proizvoda
  - Klijent manipuliše proizvodima kroz njihove apstratkne interfejse
  - Klijent ne koristi imena konkretnih proizvoda(čak ni za njihovo stvaranje)
- Olakšava izmenu familije proizvoda
  - Klasa konkretne fabrike se pojavljuje samo jednom u aplikacijie – na mestu gde se kreira njen objekat
  - Aplikacija može da koristi različite konfiguracije proizvoda menjanjem konkretne fabrike
  - Pošto apstraktna fabrika kreira kompletnu familiju, cela familija se menja odjednom
- Unapredjuje konistentnost proizvoda
  - Aplikacija koristi objekte proizvoda samo iz jedne familije u jedom trenutku
- Problem: podrška novoj vrsti proizvoda nije jednostavna
  - Apstraktna fabrika fiksira skup proizvoda koji se mogu kreirati
  - Podrška novog proizvoda zahteva proširenje apstraktne fabrike i svih potklasa

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 1. Implementacija bez registrovanja funkcija za kreiranje


```
// Apstraktna klasa proizvoda
class Mobile {
public:
    virtual string Camera() = 0;
    virtual string KeyBoard() = 0;
    virtual void PrintSpecs() { cout << Camera() << endl; cout <<
KeyBoard() << endl; }
};
// Konkretna klasa proizoda
class NOKIA_LowEndMobile : public Mobile {
public:
    string Camera() { return "2 MegaPixel"; }
    string KeyBoard() { return "ITU-T"; }
    void PrintSpecs() { cout << NOKIA <<endl; Mobile::PrintSpecs }
};
// Konkretna klasa proizoda
class NOKIA_HighEndMobile : public Mobile {
public:
    string Camera() { return "5 MegaPixel"; }
    string KeyBoard() { return "Qwerty"; }
    void PrintSpecs() { cout << NOKIA <<endl; Mobile::PrintSpecs }
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 1. Implementacija bez registrovanja funkcija za kreiranje


```
// Konkretna klasa proizoda
class SAMSUNG_LowEndMobile : public Mobile {
public:
    string Camera() { return "2 MegaPixel"; }
    string KeyBoard() { return "ITU-T"; }
    void PrintSpecs() { cout << NOKIA <<endl; Mobile::PrintSpecs }
};

// Konkretna klasa proizoda
class SAMSUNG_HighEndMobile : public Mobile {
public:
    string Camera() { return "5 MegaPixel"; }
    string KeyBoard() { return "Qwerty"; }
    void PrintSpecs() { cout << NOKIA <<endl; Mobile::PrintSpecs }

};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 1. Implementacija bez registrovanja funkcija za kreiranje


```
// MobileFactory.h : Apstraktna fabrika koja prozvodi mobiline
class MobileFactory {
public:
   virtual Mobile* GetMobile(string type)=0;
};

// NOKIA.h : Konkretna fabrika koja proizvodi mobilne je unikat
(singleton)
class NOKIA: public MobileFactory {
private:
   NOKIA(const NOKIA&);
   NOKIA& operator=(const NOKIA&);
protected:
   NOKIA(){}
   virtual ~NOKIA(){}
public:
   static NOKIA& Factory();
   Mobile* GetMobile(string type);
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 1. Implementacija bez registrovanja funkcija za kreiranje


```
// NOKIA.cpp : Konkretna fabrika koja proizvodi mobilne je unikat
(singleton)
NOKIA& NOKIA::Factory() {
    static NOKIA factory;
    return factory;
}

Mobile* NOKIA::GetMobile(string type){
    if ( type == "Low-End" ) return new NOKIA_LowEndMobile();
    if ( type == "High-End" ) return new NOKIA_HighEndMobile();
    return NULL;
}
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 1. Implementacija bez registrovanja funkcija za kreiranje


```
// SAMSUNG.h : Konkretna fabrika koja proizvodi mobilne
class SAMSUNG: public MobileFactory {
private:
    SAMSUNG(const SAMSUNG&);
    SAMSUNG& operator=(const SAMSUNG&);
protected:
    SAMSUNG(){}
    virtual ~SAMSUNG(){}
public:
    static SAMSUNG& Factory();
    Mobile* GetMobile(string type);
};
// SAMSUNG.cpp : Konkretna fabrika koja proizvodi mobilne je unikat
SAMSUNG &SAMSUNG::Factory() {
    static SAMSUNG factory;
    return factory;
}
Mobile* SAMSUNG::GetMobile(string type){
    if ( type == "Low-End" ) return new SAMSUNG_LowEndMobile();
    if ( type == "High-End" ) return new SAMSUNG_HighEndMobile();
    return NULL;
}
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 1. Implementacija bez registrovanja funkcija za kreiranje


```
void main() {
        Mobile* myMobile1 =
        NOKIA::Factory().GetMobile("Low-End");
        myMobile1->PrintSpecs();

        Mobile* myMobile2 =
        NOKIA::Factory().GetMobile("High-End");
        myMobile2->PrintSpecs();

        Mobile* myMobile3 =
        SAMSUNG::Factory().GetMobile("Low-End");
        myMobile3->PrintSpecs();

        Mobile* myMobile4 =
        SAMSUNG::Factory().GetMobile("High-End");
        myMobile4->PrintSpecs();
}
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 2. Implementacija bez registrovanja funkcija za kreiranje


```
// Apstraktna klasa objekata
class ApstrGeoFigura {
public:
    ApstrGeoFigura() {
      id = brojac++;
    }
    virtual void Crtaj() const = 0;
protected:
    unsigned id;
    static unsigned brojac;
};
unsigned ApstrGeoFigura::brojac = 0;

//Konkretne klase objekata

class Krug : public ApstrGeoFigura {
  public:
    void Crtaj() const {
      cout << "Krug " << id << endl;
    }
};
```

```
class Kvadrat : public
ApstrGeoFigura {
  public:
    void Crtaj() const {
      cout << "Kvadrad " << id <<
endl;
    }
};
class Elipsa : public ApstrGeoFigura
{
  public:
    void Crtaj() const {
      cout << "Elipsa " << id <<
endl;
    }
};
class Pravougaonik : public
ApstrGeoFigura {
  public:
    void Crtaj() const{
      cout << "Pravougaonik " << id
<< endl;
    }
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 2. Implementacija bez registrovanja funkcija za kreiranje


```
// Apstraktna klasa fabrike
class ApstrFabrika {
  public:
    virtual ApstrGeoFigura* kreirajOblu() const = 0;
    virtual ApstrGeoFigura* kreirajPravougaonu() const = 0;
};
// Konkretne klase fabrika
class FabrikaOsnovnihFigura : public ApstrFabrika {
private:
   FabrikaOsnovnihFigura(){}
   FabrikaOsnovnihFigura(const FabrikaOsnovnihFigura&);
   FabrikaOsnovnihFigura& operator=(const FabrikaOsnovnihFigura&);
public:
   static FabrikaOsnovnihFigura& objekat()  {
         static FabrikaOsnovnihFigura OBJEKAT;
         return OBJEKAT;
   }
   ApstrGeoFigura* kreirajOblu() const {    return new Krug;}
   ApstrGeoFigura* kreirajPravougaonu() const { return new Kvadrat; }
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 2. Implementacija bez registrovanja funkcija za kreiranje


```
// Konkretne klase fabrika
class FabrikaIzvedenihFigura : public ApstrFabrika {
private:
   FabrikaIzvedenihFigura(){}
   FabrikaIzvedenihFigura(const FabrikaIzvedenihFigura&);
   FabrikaIzvedenihFigura& operator=(const FabrikaIzvedenihFigura&);
public:
   static FabrikaIzvedenihFigura& objekat() {
         static FabrikaIzvedenihFigura OBJEKAT;
         return OBJEKAT;
   }
public:
    ApstrGeoFigura* kreirajOblu() const   { return new Elipsa; }
   ApstrGeoFigura* kreirajPravougaonu() const { return new Pravougaonik; }
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 2. Implementacija bez registrovanja funkcija za kreiranje


```
#ifndef _MY_PAIR_H
#define _MY_PAIR_H
template<typename T1, typename T2>
void MySwap(T1 &first, T2 &second){ T1 tmp = first; first = second; second =
first;}

template<typename T1, typename T2>
struct MyPair
{
typedef T1 first_type;
typedef T2 second_type;
T1 first;
T2 second;
MyPair(): first(), second() { }
template<typename U1, typename U2>
MyPair(const U1 &a, const U2 &b): first(a), second(b) { }

template<typename U1, typename U2>
MyPair(const MyPair<U1,U2>& other):first(other.first) second(other.second){ }
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 2. Implementacija bez registrovanja funkcija za kreiranje


```
template<typename U1, typename U2>
MyPair& operator=(const MyPair<U1,U2>& other){
    first = other.first;
    second = other.second;
    return *this;
}

template<typename U1, typename U2>
void MySwap(const MyPair<U1, U2> &other){ MySwap(first, other.first);
MySwap(second, other.second);}

};

template<typename T1, typename T2, typename U1, typename U2>
inline bool operator==(const MyPair<T1, T2>& x, const MyPair<U1, U2>& y)
{ return x.first == y.first && x.second == y.second; }

template<typename T1, typename T2, typename U1, typename U2>
inline bool operator<(const MyPair<T1, T2>& x, const MyPair<U1, U2>& y)
{ return x.first < y.first || (!(y.first < x.first) && x.second < y.second); }
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 2. Implementacija bez registrovanja funkcija za kreiranje


```
template<typename T1, typename T2, typename U1, typename U2>
inline bool operator>(const MyPair<T1, T2>& x, const MyPair<U1, U2>& y)
{ return y < x; }

template<typename T1, typename T2, typename U1, typename U2>
inline bool operator<=(const MyPair<T1, T2>& x, const MyPair<U1, U2>& y)
{ return !(y < x); }

template<typename T1, typename T2, typename U1, typename U2>
inline bool operator>=(const MyPair<T1, T2>& x, const MyPair<U1, U2>& y)
{ return !(x < y); }


template<typename T1, typename T2, typename U1, typename U2>
inline void MySwap(MyPair<T1, T2>& x, MyPair<U1, U2>& y)
{ x.swap(y); }


template<typename T1, typename T2>
inline MyPair<T1, T2> Make_MyPair(T1 x, T2 y) { return MyPair<T1, T2>(x, y); }

#endif /* _MY_PAIR_H */
```

```
MyPair<ApstrGeoFigura*, ApstrGeoFigura*> nizParovaFigura[3];

nizParovaFigura[0] =
Make_MyPair(FabrikaOsnovnihFigura::objekat().kreirajOblu(),
FabrikaIzvedenihFigura::objekat().kreirajPravougaonu());

nizParovaFigura[1] =
Make_MyPair(FabrikaIzvedenihFigura::objekat().kreirajOblu(),
FabrikaIzvedenihFigura::objekat().kreirajPravougaonu());

nizParovaFigura[2] =
Make_MyPair(FabrikaIzvedenihFigura::objekat().kreirajOblu(),
FabrikaOsnovnihFigura::objekat().kreirajPravougaonu());

for (int i=0; i < 3; i++) {
        nizParovaFigura[i].first->Crtaj();
        nizParovaFigura[i].second->Crtaj();
}
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 3. Implementacija apstraktne fabrike


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

class ISedan{
public:
  virtual void Start() = 0;
  virtual void Stop() = 0;
  virtual void Accelarate() = 0;
  virtual ~ISedan() {};
};

class IHatchBack{
public:
  virtual void Start() = 0;
  virtual void Stop() = 0;
  virtual void Accelarate() = 0;
  virtual ~IHatchBack() {};
};
```

```
class BMWSedan : public ISedan{
public:
  BMWSedan(const std::string& sModelName):m_sModelName(sModelName){};
  void Start() { cout << "\n BMW Sedan Car Start..."; }
  void Stop() { cout << "\n BMW Sedan Car Stop..."; }
  void Accelarate() { cout << "\n BMW Sedan Car Accelarate..."; }
private:
  std::string m_sModelName;
};

class BMWHatchBack : public IHatchBack{
public:
  BMWHatchBack(const std::string& sModelName):m_sModelName(sModelName)
{};
  void Start() { cout << "\n BMW HatchBack Car Start..."; }
  void Stop() { cout << "\n BMW HatchBack CarStop..."; }
  void Accelarate() { cout << "\n BMW HatchBack Car Accelarate..."; }
private:
  std::string m_sModelName;
};
```

```
class AudiSedanCar : public ISedan{
public:
  AudiSedanCar(const std::string& sModelName) :
m_sModelName(sModelName) {};
  void Start() { cout << "\n Audi Sedan Car Start..."; }
  void Stop() { cout << "\n Audi Sedan Car Stop..."; }
  void Accelarate() { cout << "\n Audi Sedan Car Accelarate..."; }
private:
  std::string m_sModelName;
};

class AudiHatchBackCar : public IHatchBack{
public:
  AudiHatchBackCar(const std::string& sModelName) :
m_sModelName(sModelName) {};
  void Start() { cout << "\n Audi HatchBack Car Start..."; }
  void Stop() { cout << "\n Audi HatchBack CarStop..."; }
  void Accelarate() { cout << "\n Audi HatchBack Car Accelarate..."; }
private:
  std::string m_sModelName;
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 3. Implementacija apstraktne fabrike


```
class ICarFactory{
public:
  virtual ISedan* CreateSedan() = 0;
  virtual IHatchBack* CreateHatchBack() = 0;
  virtual ~ICarFactory() {};
};

class BMWCarFactory : public ICarFactory{
public:
  ISedan* CreateSedan() { return new BMWSedan("Sedan BMW"); };
  IHatchBack* CreateHatchBack() { return new BMWHatchBack("HatchBack
BMW"); };
};

class AudiCarFactory : public ICarFactory{
public:
  ISedan* CreateSedan() { return new AudiSedanCar("Sedan Audi"); };
  IHatchBack* CreateHatchBack() { return new
AudiHatchBackCar("HatchBack Audi"); }
};
```

APSTRAKTNA FABRIKA (engl. Abstract Factory)
Primer 3. Implementacija apstraktne fabrike


```
void main(){
  ICarFactory* pCarFactory = new BMWCarFactory();
  ISedan* pSedanCar = pCarFactory->CreateSedan();

  pSedanCar->Start();
  pSedanCar->Accelarate();
  pSedanCar->Stop();
  delete pCarFactory;
  delete pSedanCar;

  pCarFactory = new AudiCarFactory();
  IHatchBack* pHatchBackCar = pCarFactory->CreateHatchBack();

  pHatchBackCar->Start();
  pHatchBackCar->Accelarate();
  pHatchBackCar->Stop();
  delete pCarFactory;
  delete pHatchBackCar;

}
```

```
#ifndef SIMPLE_DYNAMIC_ARRAY_H_
#define SIMPLE_DYNAMIC_ARRAY_H_
#include <stdlib.h>
#include <stddef.h>

template <typename ValueType> class Array {
public:
  typedef ML_Size_t size_type;
  typedef ValueType Value_t;
  typedef Value_t Element_t;
  typedef Value_t* pointer_t;
  typedef const Value_t* const_pointer_t;
  typedef Value_t& reference_t;
  typedef const Value_t& const_reference_t;
  typedef pointer_t Iterator_t;
  typedef const_pointer_t ConstIterator_t;
  static const ML_Size_t csm_INIT_CAPACITY = 2;
public:
  Array() : m_Size(0), m_Capacity(0), m_pArray(0){}
  Array(ML_Size_t InitCapacity) : m_Size(0), m_Capacity(InitCapacity),
m_pArray = new Value_t[m_Capacity] {}
. . .
};
#endif
```

```
template <typename ValueType> class Array {
public:
/* Sablon konstruktora kopije */
  template<typename OtherType>
  Array(const Array<OtherType> &Other) :m_Size(Other.m_Size),
m_Capacity(Other.m_Capacity), m_pArray = new Value_t[m_Capacity]{
    std::copy(Other.m_pArray, Other.m_pArray + m_Capacity, m_pArray);
  }
  ~Array() { delete[] m_pArray; }
/* Sablon operatora dodele vrednosti */
  template<typename OtherType>
  Array<Value_t>& operator=(const Array<OtherType>& Other){
    if (this != &Other){
      delete[] m_pArray;
      m_Capacity = Other.m_Capacity;
      m_Size = Other.m_Size;
      m_pArray = new Value_t[m_Capacity];
      std::copy(Other.m_pArray, Other.m_pArray + m_Capacity, m_pArray);
    }
    return *this;
  }
. . .
};
#endif
```

```
template <typename ValueType> class Array {
public:
  pointer_t Append(const_reference_t AnElement){
    if (m_Size == m_Capacity) {
      Resize( m_Capacity +
                (m_Capacity > 1 ? m_Capacity >> 1 : csm_INIT_CAPACITY) );
    }
    Value_t *Ptr = m_pArray + m_Size;
    *Ptr = AnElement;
    ++m_Size;
    return Ptr;
  }
  reference_t operator[](ML_Size_t Idx) { return m_pArray[Idx];  }
  const_reference_t operator[](ML_Size_t Idx) const { return m_pArray[Idx]; }
  void ClearWithoutResizing() { m_Size = 0; }
. . .
protected:
  size_type m_Size;
  size_type m_Capacity;
  Value_t *m_pArray;
};
template< class Value_t>
const ML_Size_t Array<Value_t>::csm_INIT_CAPACITY;
#endif
```

```
template <typename ValueType> class Array {
public:
  /* Funkcije koje omogucavaju vezivanje iteratora za objekat tipa
Array<ValueType> */
  Iterator_t Begin() { return m_pArray; }
  Iterator_t End() { return m_pArray + m_Size; }

  /* Funkcije koje omogucavaju vezivanje iteratora za konstantni objekat tipa
Array<ValueType> */
  ConstIterator_t Begin() const { return m_pArray; }
  ConstIterator_t End() const { return m_pArray + m_Size; }

protected:
  size_type m_Size;
  size_type m_Capacity;
  Value_t *m_pArray;
};


template< class Value_t>
const ML_Size_t Array<Value_t>::csm_INIT_CAPACITY;

#endif
```

```
// Apstraktni proizvod
class AbsProduct {
public:
virtual ~ApsProduct() {}
virtual bool Get() = 0;
};

// Konkretan proizvod
class Product_A : public AbsProduct{
public:
  bool Get() { return true; }
};

//Konkretan proizvod
class Product_B : public AbsProduct{
public:
  bool Get() { return false; }
};
```

```
// Sablon apstraktnog stvaraoca : T je tip proizvoda koji stvara
template <class T>
class Creator {
public:
  virtual ~Creator(){}
  virtual T* Create() = 0;
};

// Sablon konretnog stvaraoca
template <class DerivedType, class BaseType>
class DerivedCreator : public Creator<BaseType> {
public:
  BaseType* Create() { return new DerivedType; }
};
```

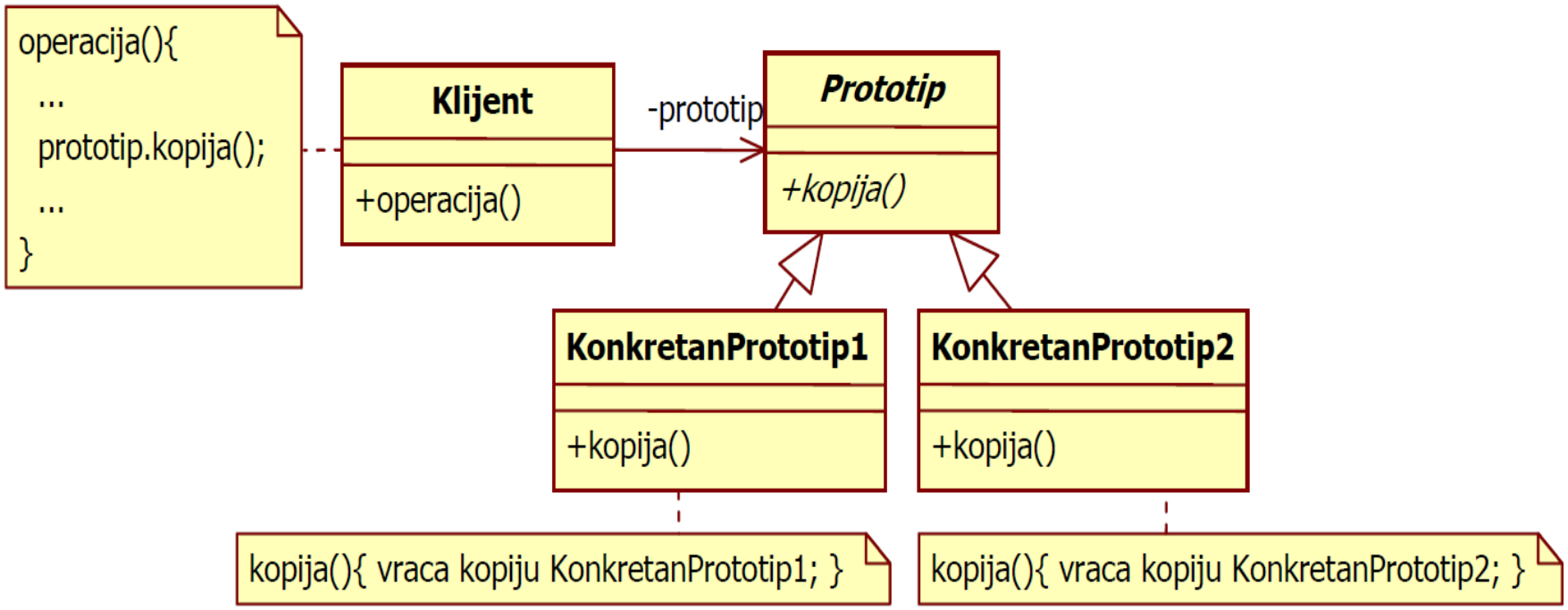
```
template <class T>
class Factory{
public:
  void Register(int idx, Creator<T>* ptrToCreator) {
    creatorRegister[idx] = ptrToCreator;
  }

  T* Create(int idx) { return cratorRegister[idx]->Create(); }

  ~Factory(){
    for(int idx = 0; idx < creatorRegister.Size(); ++idx)
      delete creatorRegister[idx];
  }
private:
  Array<Creator<T>*> creatorRegister;
}
```

```
Factory<AbsProduct> temp;

temp.Register(0, new DerivedCreator<Product_A, AbsProduct>);
temp.Register(1, new DerivedCreator<Product_B, AbsProduct>);

//Pokazivac na apstraktni proizvod
AbsProduct* ptrToProduct = 0;

//Kreiraj proizvod u koristi ga
ptrToProduct = temp.Create(0);
printf("Product_A %u\n", ptrToProduct->Get());
delete ptrToProduct;

//Kreiraj proizvod i koristi ga
pBase = temp.Create(1);
printf(" Product_A %u\n", pBase->Get());
delete ptrToProduct;
```

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
*/
};
```