

# KLASE

Milica Kojičić, 1066/2015.



# Apstraktni tipovi podataka (ADT)

- Apstraktni tip podataka predstavlja kolekciju podataka i operacija koje rade nad tim podacima
- Operacije opisuju podatke ostatku programa i dozvoljavaju ostatku programa da menjaju te podatke
- Razumevanje koncepta ADT-a je ključno za razumevanje objektno orijentisanog programiranja; programeri mogu da prave klase koje su inicijalno lakše za implementaciju, ali i lakše za održavanje tokom vremena



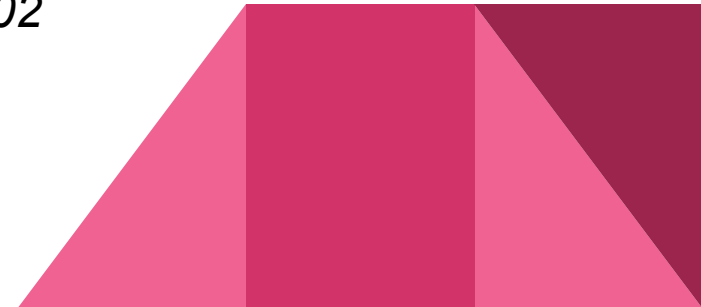
# ADT (2)

- Korišćenje apstraktnih tipova podataka nam omogućava da rukujemo entitetima koji postoje u stvarnom svetu, pre nego entitetima na niskom nivou implementacije
- Na primer, umesto da dodamo čvor u povezanu listu, možemo da dodamo ćeliju u tabelu
- Dakle, omogućava nam da radimo u domenu problema, a ne u domenu implementacije



# Primer ADT-a

- Zamislimo da pišemo program za obradu nekog teksta. Deo tog programa će se baviti fontovima. Ako bismo ovde koristili ADT, imali bismo skup metoda koje se bave fontovima zajedno sa podacima nad kojima te metode operišu
- Ako bismo morali da promenimo font na *bold*, to bismo uradili nekim od sledećih *ad hoc* načina:
  - *currentFont.attribute = currentFont.attribute or BOLD*
  - *currentFont.attribute = True*
  - *currentFont.attribute = currentFont.attribute or 0x02*



# Primer ADT-a (2)

- Ali *ad hoc* pristup može biti lako zamenjen boljom programerskom praksom koja ima sledeće prednosti:
  - sakrivanje implementacionih detalja
  - promene su lokalne, tj. ne utiču na čitav program
  - interfejs može biti informativniji
  - lakše je poboljšati performanse
  - korektnost programa je očiglednija
  - program sam po sebi može predstavljati dokumentaciju
  - *currentFont.SetBoldOn()* je čitljivije od  
*currentFont.attribute = currentFont.attribute or BOLD*



# Primer ADT-a (3)

- U ovom slučaju možemo da definišemo sledeće funkcije za manipulisanje fontom:
  - *currentFont.SetSizeInPoints( sizeInPoints)*
  - *currentFont.SetSizeInPixels( sizeInPixels )*
  - *currentFont.SetBoldOn()*
  - *currentFont.SetBoldOff()*
  - *currentFont.SetItalicOn()*
  - *currentFont.SetItalicOff()*
  - *currentFont.SetTypeFace( faceName )*
- Kod unutar ovih rutina je sigurno sličan kodu koji smo videli u prethodnom *ad hoc* pristupu, samo što smo ovde izolovali operacije nad fontovima u *skup operacija* i omogućen je veći nivo apstrakcije ostatku programa pri korišćenju fontova

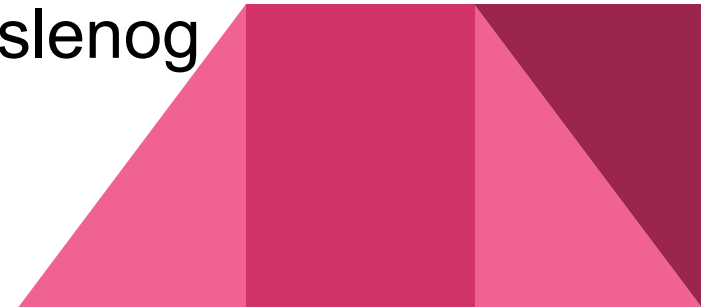
# ADT i klase

- Apstraktni tipovi podataka predstavljaju temelj u konceptu klasa
- U jezicima koji podržavaju klase možemo bilo koji apstraktni tip podataka implementirati kao posebnu klasu
- Klase obično uključuju dodatne koncepte nasleđivanja i polimorfizma. Možemo reći da je klasa apstraktni tip podatka uz nasleđivanje i polimorfizam



# Interfejsi

- Prva i verovatno najvažnija stvar pri kreiranju visoko kvalitetne klase jeste pravljenje dobrog interfejsa
- Ovo se odnosi na pravljenje dobre apstrakcije koja će da osigura da detalji implementacije ostanu skriveni
- Interfejs treba da ponudi grupu metoda koje zajedno imaju neki očigledni smisao
- Možemo, na primer, imati klasu koja implementira zaposlenog u nekom preduzeću. Ona bi sadržala podatke o njegovom imenu, prezimenu, adresi, itd. Takođe bi nudila usluge za inicijalizaciju i korišćenje zaposlenog

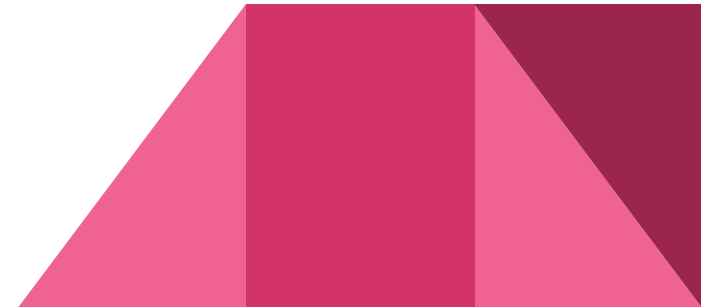




# Interfejsi (2)

- C++ primer interfejsa koji predstavlja dobru apstrakciju:

```
class Employee {  
    public:  
    // public constructors and destructors  
    Employee();  
    Employee(  
        FullName name,  
        String address,  
        String workPhone,  
        String homePhone,  
        TaxId taxIdNumber,  
        JobClassification jobClass  
    );  
    virtual ~Employee();
```



# Interfejsi (3)

*// public routines*

*FullName GetName() const;*

*String GetAddress() const;*

*String GetWorkPhone() const;*

*String GetHomePhone() const;*

*TaxId GetTaxIdNumber() const;*

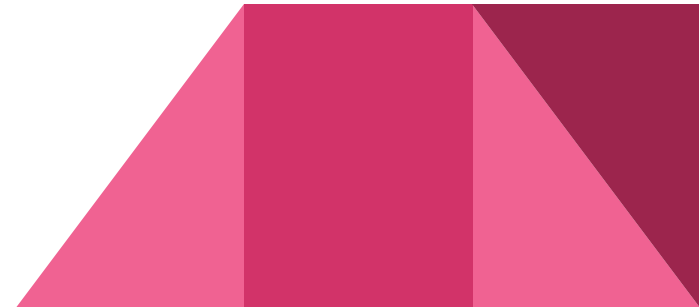
*JobClassification GetJobClassification() const;*

*...*

*private:*

*...*

*};*



# Interfejsi (4)

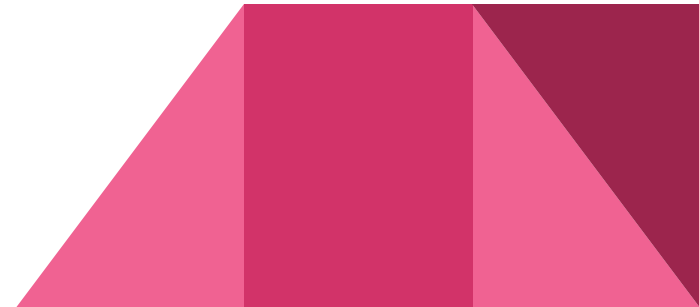
- Interno, ova klasa može imati dodatne rutine i podatke koje podržavaju ove usluge, ali korisnici klase ne moraju da znaju ništa o njima
- Klasa koja predstavlja lošu apstrakciju je ona koja u sebi sadrži više raznovrsnih funkcija



# Interfejsi (5)

- Primer interfejsa koji ne predstavlja dobru apstrakciju:

```
class Program {  
    public:  
    // public routines  
  
    void InitializeCommandStack();  
    void PushCommand( Command command );  
    Command PopCommand();  
    void ShutdownCommandStack();  
    void InitializeReportFormatting();  
    void FormatReport( Report report );  
    void PrintReport( Report report );  
    void InitializeGlobalData();  
    void ShutdownGlobalData();  
};
```



# Pravila pri pravljenju interfejsa

- U prethodnom primeru je teško videti bilo kakvu povezanost između rutina koje rade sa stekom i rutina o izveštajima. Ovaj interfejs ne predstavlja konzistentnu apstrakciju, tako da klasa nije kohezivna
- Ovo su neke od smernica za pravljenje dobrih i apstraktnih klasnih interfejsa:
  - treba da postoji konzistentan nivo apstrakcije u interfejsu
  - treba da budemo sigurni da znamo koju to apstrakciju klasa implementira



# Pravila pri pravljenju interfejsa (2)

- za svaku uslugu treba da obezbedimo njoj suprotnu (ako imamo funkciju za paljenje svetla, moramo imati i funkciju za gašenje istog)
- nepovezane informacije treba razdvojiti u zasebne klase
- treba se čuvati opadanja kvaliteta interfejsa tokom modifikacije klase
- ne treba dodavati javne (public) članove koji su nekonzistentni sa apstrakcijom koja se implementira
- posmatrati koheziju i apstrakciju kao jedno
- minimizovati pristup klasama i poljima klase
- ne treba javno otkrivati polja klase



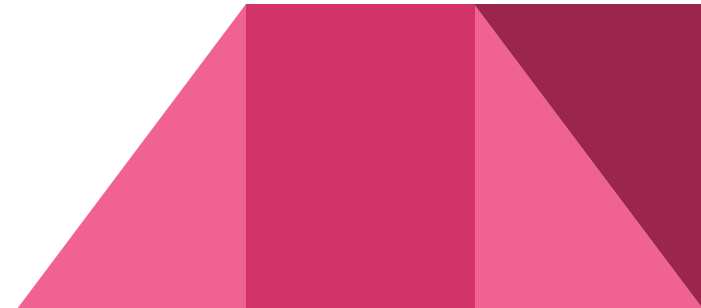
# Pravila pri pravljenju interfejsa (3)

- izbegavati otkrivanje implementacionih detalja u interfejsu
- ne praviti pretpostavke o korisnicima klase
- izbegavati “prijateljske” klase
- treba biti veoma svestan semantičkog narušavanja enkapsulacije
- čuvati se čvrste spregnutosti među klasama



# Pravila pri pravljenju interfejsa (4)

- Definisanje dobrih interfejsa znači pisanje visoko kvalitetnog programa
- Interni dizajn klase, kao i implementacija, su takođe važni





# Sadržavanje

- Sadržavanje znači da neka klasa sadrži neke primitivne tipove ili objekte u sebi
- Na primer, radnik ima ime, broj telefona, jmbg itd..
- Ovo možemo implementirati tako sto će polja klase *Radnik* biti upravo *ime*, *broj telefona*, *jmbg*...
- Takođe, mozemo ga implementirati preko privatnog nasleđivanja
- Optimalan broj atributa u klasi je 7



# Nasleđivanje (*is a* veza)

- Nasleđivanje znači da neka klasa može predstavljati specijalizaciju neke druge klase
- Svrha nasleđivanja je da imamo jednostavniji kod definisanjem bazne klase koja ima elemente koji su sadržani u svim izvedenim klasama
- Zajednički elementi mogu biti metode, atributi...
- Nasleđivanje nam omogućava da izbegnemo ponavljanje koda njegovim centralizovanjem u baznu klasu



# Nasleđivanje (2)

- Važno je da znamo da li će neka funkcija biti vidljiva u izvedenoj klasi, da li će imati neku podrazumevajuću implementaciju i da li ta implementacija može da bude redefinisana
- Takođe, bitno nam je da znamo da li će neko polje klase (bio to atribut, konstanta ili enumeracija) biti vidljivo u izvedenoj klasi
- Implementiranje *is a* veze može se ostvariti javnim nasleđivanjem



# Nasleđivanje (3)

- Ako ne dizajniramo kod i ne pišemo dokumentaciju u skladu sa nasleđivanjem, onda bi bilo bolje zabraniti nasleđivanje
- Ne koristiti nasleđivanje, osim ako izvedena klasa stvarno jeste (*is a*) specijalni slučaj bazne klase
- Treba da budemo sigurni da smo nasledili samo ono što smo želeli da nasledimo
- Ne prevazilaziti metode koje nisu namenjene za to (*non - overridable*)
- Na najvišem nivou drveta nasleđivanja treba da budu zajednička ponasanja, atributi, interfejsi



# Nasleđivanje (4)

- Biti oprezan sa klasama koje imaju samo jedan primerak
- Biti oprezan sa baznom klasom iz koje je izvedena samo jedna klasa
- Biti oprezan sa klasama koje prevazilaze neku metodu, a ta prevaziđena metoda ne radi ništa
- Izbegavati duboka stabla nasleđivanja
- Pribegavati polimorfizmu pre nego stalnoj proveru tipova
- Sve attribute definisati kao privatne

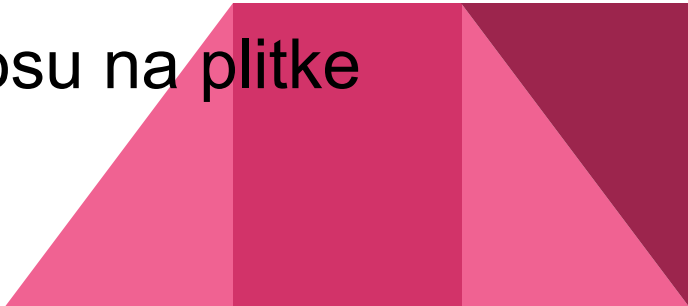


# Višestruko nasleđivanje

- Višestruko nasleđivanje podrazumeva da jedna klasa može naslediti više klasa
- Što se višestrukog nasleđivanja tiče, tu treba biti vrlo oprezan
- Kaže se da ono otvara Pandorinu kutiju kompleksnosti koje prosto ne postoje kod običnog nasleđivanja
- Java npr, dozvoljava samo višestruko nasleđivanje interfejsa, dok je u C++ dozvoljeno višestruko nasleđivanje i interfejsa i klasa



# Polja i metode klase

- Što je manji broj metoda u klasi, to bolje
  - Onemogućiti implicitno generisanje metoda i operatora koji nam ne trebaju
  - Minimizovati broj različitih funkcija koje se pozivaju u klasi
  - Minimizovati obim interakcije klase sa drugim klasama
  - Inicijalizovati sva polja klase u konstruktoru, ako je moguće
  - Primijeniti *singleton* obrazac korišćenjem privatnog konstruktora
  - Preferirati duboke kopije objekata u odnosu na plitke
- 

# Razlozi za pravljenje klase

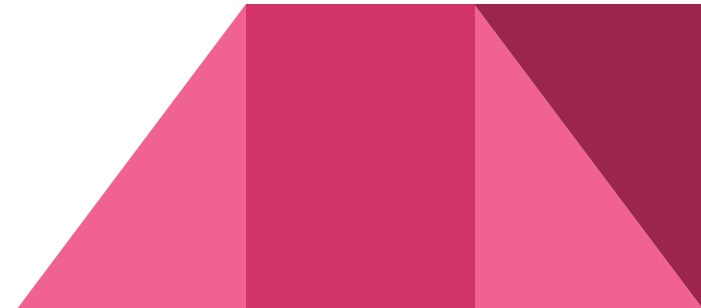
- Nije tačno da se klase prave samo da bi modelovale objekte iz stvarnog sveta
- U praksi se klase kreiraju iz mnogo više razloga
- Sledi lista dobrih razloga zasto kreirati klasu:
  - Modelovanje objekata iz stvarnog sveta
  - Modelovanje apstraktnih objekata
  - Smanjivanje kompleksnosti
  - Izolovanje kompleksnosti
  - Sakrivanje implementacionih detalja





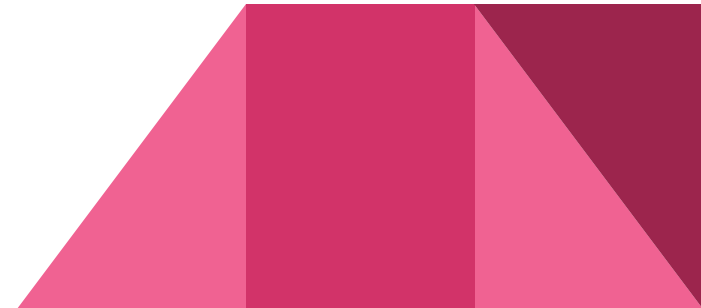
# Razlozi za pravljenje klase (2)

- Ograničavanje efekata promene u kodu
- Sakrivanje globalnih podataka
- Centralizovana kontrola
- Ponovna upotrebljivost koda
- Pakovanje povezanih operacija u pakete
- Refaktorisan kod



# Klase koje treba izbegavati

- Ne treba praviti takve klase koje će samo da kupe podatke od ostalih klasa *get* i *set* metodama
- Ako se klasa sastoji samo od atributa, ali ne i od ponašanja, razmisliti o njenom eliminisanju
- Isto tako, klasa koja ima samo ponašanje (bez atributa) je takođe kandidat za uklanjanje



# Klase i različiti programski jezici

- Pristupi u pravljenju klasa u različitim programskim jezicima variraju na interesantne načine
- U *Javi*, sve funkcije podrazumevano mogu da se prevaziđu i moraju biti deklarisanе kao *final* kako bi se sprečilo njihovo prevazilazenje u nasleđenim klasama
- U C++-u funkcije ne mogu podrazumevano da se prevaziđu. Moraju biti deklarisanе kao *virtual* u baznoj klasi kako bi to bilo omogućeno.
- U *Visual Basic*-u funkcija u baznoj klasi u deklaraciji mora da ima ključnu rec *overridable*, a u izvedenoj klasi ključnu rec *overrides*

# Zaključak

- Interfejsi treba da obezbede konzistentnu apstrakciju. Mnogi problemi dolaze od narušavanja samo ovog principa
- Interfejsi treba da sakriju nešto, npr. sistemski interfejs, dizajn, implementacione detalje
- Sadržavanje se obično više preferira od interfejsa, osim ako se ne modeluje *is a* odnos
- Nasleđivanje je korisna alatka, ali dosta doprinosi kompleksnosti
- Klase su primarne alatke za upravljanje kompleksnošću. Oko njihovog dizajna se treba truditi, da bi se ispunio taj osnovni cilj u projektovanju softvera

HVALA NA PAŽNJI!

