Programske paradigme

— Objektno-orijentisano programiranje —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

1 Potrebe za OOP

1.1 Funcionalna dekompozicija problema i izmene

Funcionalna dekompozicija problema — „Od opšteg ka posebnom"

- Postupak: podeli veliki problem u manje korake koji su potrebni da bi se problem rešio. Za baš velike probleme, podeli ih u manje potprobleme, pa onda dekomponuj manje potprobleme u odgovarajuće funkcionalne korake.

- Cilj je da se problem deli dok ne dođe do onog nivoa koji je jednostavan i može da se reši u nekoliko koraka, tada se ti koraci poređaju u odgovarajućem redosledu koji rešava identifikovane potprobleme i na taj način je i veliki problem rešen.
- Ovo je prirodan i jednostavan način razmišljanja...

Nedostaci funkcionalne dekompozicije problema

- Postoje dva osnovna problema ovog pristupa
  1. Na ovaj način se kreira program koji je dizajniran na osnovu osobina glavnog programa — ovaj program kontroliše i zna sve detalje o tome šta će biti izvršeno i koje će se strukture podataka za to koristiti.
  2. Ovakav dizajn ne odgovara dobro na izmene zahteva — ovi programi nisu dobro podeljeni na celine tako da svaki zahtev za promenom obično zahteva promenu glavnog programa, pri čemu mala promena u strukturama podataka, na primer, može da ima uticaja kroz ceo program.

Uticaj izmena

- Rešavanje problema orijentisanjem na procese koji se dešavaju da bi se problem rešio ne vodi do programskih struktura koje mogu da lako reaguju na izmene: izmene u razvoju softvera obično uključuju varijacije na postojeće teme.
- Na primer, razmatrali smo funkcionalnu dekompoziciju problema izračunavanja prosečnog rastojanja između tačaka u ravni:
  - Razmatrati prosečno rastojanje tačaka u prostoru.
  - Razmatrati prosečno rastojanje između nekih drugih objekata.
  - Razmatrati i druge odnose objekata, ne samo rastojanje.
- U glavnom programu, ovi tipovi izmena povećavaju kompleksnost i zahtevaju puno dodatnih fajlova da se ponovo kompiliraju.

Zašto je uticaj izmena važan?

- Menjaju se potrebe, menjaju se ideje, menjaju se zahtevi klijenata...
- Potrebe za izmenama i dopunama su stalne
- Mnoge greške nastaju prilikom izmena koda.
- Treba nam dizajn koda koji može da se prilagodi promenama na pravi način.

1.2 Kohezija, kopčanje i efekat talasanja

Kohezija i kopčanje

- Kohezija i kopčanje su uvedeni sedamdesetih godina dvadesetog veka kao metrike koje omogućavaju kvantitativno praćenje kvaliteta koda, sa ciljem da se smanje troškovi održavanja i modifikacije koda (uveo ih je *Larry Constantine*).
- **Kohezija** (engl.*cohesion*) je pojam koji se odnosi na to koliko blisko su povezane operacije koje se vrše u jednoj celini, tj rutini/klasi/modulu.
- Pojednostavljeno, ono što želimo je da svaka rutina radi samo jednu stvar, ili da se svaki modul odnosi na samo jedan zadatak.
- Na primer, nije dobro da funkcija radi računanje minimuma, maksimuma i prosečne vrednosti niza istovremeno, bolje je da to rade tri razdvojene funkcije.
- Kohezija je koncept koji opisuje odnose u **jednoj** rutini/klasi/modulu

Kohezija, primer (C#)

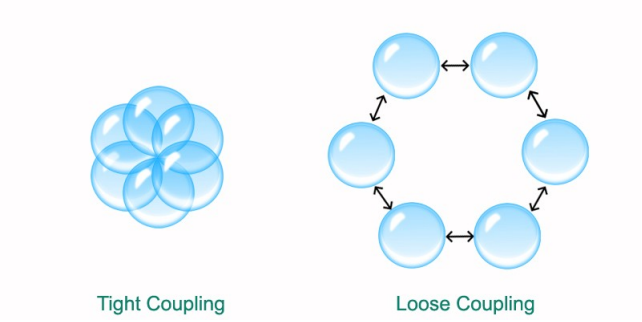
```
public class NumberManipulator
{
    private int _number;
    public int NumberValue => _number;
    public void AddOne() => _number++;
    public void SubtractOne() => _number--;
}
```

```
public class NonCohesiveNumberManipulator
{
    private int _firstNumber;
    private int _secondNumber;
    private int _thirdNumber;
    public void IncrementFirst() => _firstNumber++;
    public void IncrementSecond() => _secondNumber++;
    public void IncrementThird() => _thirdNumber++;
}
```

Kopčanje

- **Kopčanje** se odnosi na jačinu povezanosti dve celine, tj rutine/klase/modula.
- Na primer, nije dobro da jedan modul modifikuje i oslanja se na internu strukturu i rad drugog modula. To je jako kopčanje.
- Sa jakim kopčanjem, jedna promena u nekoj funkciji ili strukturi podataka uzrokuje potrebu za dodatnim promenama u svim drugim delovima sistema.
- Kopčanje je koncept koji se odnosi na odnose između **različitih** rutina/klasa/modula

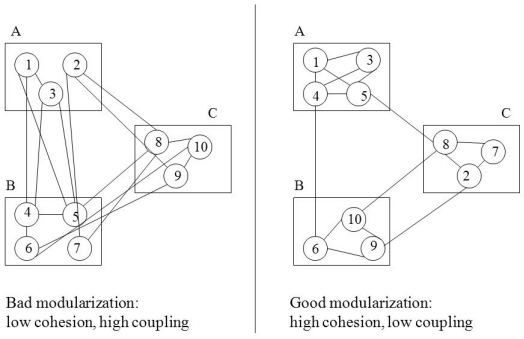
Kopčanje



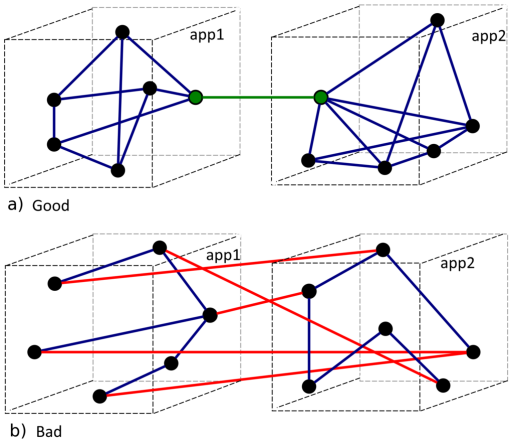
Osobine koda nastalog funkcionalnom dekompozicijom

- Kopčanje je pojam komplementaran koheziji, slaba kohezija povlači jako kopčanje i jaka kohezija povlači slabo kopčanje.
- Kôd koji se razvija primenom funkcionalne dekompozicije ima **slabu koheziju** i **jako kopčanje**, odnosno radi previše stvari i ima previše povezanosti.
- Želimo jaku kohezivnost i slabo kopčanje.

Kohezija i kopčanje



Kohezija i kopčanje



Efekat talasanja (engl. *ripple effect*)

- Potrebe za izmenama u okviru celog koda otežavaju debagovanje i razumevanje zadataka koje sistem obavlja.
- Napravimo izmenu, a neočekivano nešto drugo u sistemu ne funkcioniše — to je neželjeni propratni **efekat talasanja**
- Ukoliko imamo kôd sa jakim kopčanjem, otkrivamo da mnogi delovi sistema zavise od koda koji je izmenjen — treba vremena da se otkriju i razumeju ti odnosi.

1.3 Ciljevi OOP

Problemi

- Razvoj softvera krajem šezdesetih godina se suočavao sa velikim brojem problema.
- Simptomi softverske krize: kasne isporuke, probijanje rokova i budžeta, loš kvalitet, nezadovoljavanje potreba, slaba pouzdanost.
- Nepostojanje metodologije u razvoju softvera.
- Cena softvera se značajno povećala, održavanje i razvoj softvera je nadmašio troškove hardvera.
- OOP je nastala kao jedna od posledica softverske krize.

Ciljevi

- Kompleksnost softvera zahtevala je promene u stilu programiranja. Cilj je bio da se:
  - porizvodi pouzdan softver
  - smanji cena proizvodnje softvera
  - razvijaju ponovo upotrebljivi moduli
  - smanje troškovi održavanja
  - smanji vreme razvoja softvera
- OOP uvodi novi način razmišljanja za pronalaženje rešenja problema.

1.4 Novi način razmišljanja — primer

Novi način razmišljanja

Primer

- Studenti nakon predavanja idu na naredno predavanje.
- Funkcionalna dekompozicija problema:
  - Nastavnik je glavni program koji rešava problem:
    - Za svakog studenta nastavnik pronalazi kojoj grupi pripada.
    - U rasporedu časova nastavnik pronalazi koji čas data grupa treba da ima.
    - Nastavnik svakom studentu kaže gde treba da ide i šta treba da sluša.
- Da li se tako rešava problem u stvarnom životu?

Primer

- OO pristup
  - Nastavnik podrazumeva da svaki student zna raspored časova i kojoj grupi pripada, tako da sam zna gde treba da ide.
  - U najgorem slučaju, nastavnik može svima da saopšti gde mogu da nađu raspored časova i podelu po grupama i da kaže: koristite ove informacije da biste odredili lokaciju sledećeg predavanja.

Primer
Razlike između pristupa:

- Prvi pristup:
  - U prvom slučaju, nastavnik zna sve i za sve je odgovoran, ukoliko dođe do nekih promena, na njemu je odgovornost da te promene izvede
  - Nastavnik mora da daje detaljna uputstva svakom studentu (entitetu u sistemu)
- Drugi pristup:
  - Nastavnik očekuje da su studenti (entiteti u sistemu) sposobni da sami reše problem (da su samodovoljni).
  - Nastavnik daje samo opšte instrukcije i očekuje od studenata da znaju da ih primene u svojim specifičnim situacijama.

Primer

- Osnovna prednost drugog pristupa je u **podeli odgovornosti** — svaki deo sistema ima svoju odgovornost i odgovornost se prebacuje sa glavnog programa na entitete u sistemu.
- Pretpostavimo da u okviru grupe postoje i studenti volonteri koji između predavanja treba da urade nešto specijalno.

  - U prvom slučaju: nastavnik bi morao da zna da oni postoje i šta je to što oni treba da urade.
  - U drugom slučaju: nastavnik samo završava čas i kaže svima da idu na naredno predavanje, a svaki student radi šta treba, studenti volonteri sami znaju koje su njihove obaveze.

- Dodavanje novih vrsta obaveza i entiteta u sistemu ne remeti glavni program.

Primer
Prethodni primer ilustruje osnovne benefite OOP:

- Samodovljni entiteti — objekti
- Davanje opštih instrukcija — kodiranje interfejsa
- Očekivanje da entiteti umeju da primene opšte instrukcije na njihove specifične situacije — polimorfizam i podklase
- Dodavanje novih entiteta u sistem bez uticaja na vođu sesije — kodiranje interfejsa, polimorfizma, podkalse
- Prebacivanje odgovornosti — funkcionalnost je podeljena kroz mrežu objekata u sistemu

2 Razvoj i odnos sa imperativnom paradigmom

2.1 Nastanak i predstavnici

OOP

- Prvi OO jezik je nastao 1960. godine: Simula
- Simula (1960), Smalltalk (1970), Ada (1970), Objective-C (1980), C++ (1980), Eiffel (1992), Java (1996), C# (2000)
- Procvat OO programiranja dolazi sa jezikom C++
- Većina modernih jezika uključuje bar neke OO koncepte i zapravo su multiparadigmatski ili skript jezici: Python, Lua, Ruby, JavaScript, TypeScript, Scala...

OO jezici

Table 1.7 OO Programming languages
(a) OO programming languages and their inventors

Language	Inventor, Year	Organization
Simula	Kristen Nygaard and Ole-Johan Dahl, 1960	Norwegian Defense Research Establishment, Norway
Ada	Jean Ichbiah, 1970	Honeywell-CII-Bull, France
Smalltalk	Alan Kay, 1970	Xerox PARC, USA
C++	Bjarne Stroustrup, 1980	AT&T Bell Labs, USA
Objective C	Brad Cox, 1980	Stepstone, USA
Object Pascal	Larry Tesler, 1985	Apple Computer, USA
Eiffel	Bertrand Meyer, 1992	Eiffel Software, USA
Java	James Gosling, 1996	Sun Microsystems, USA
C#	Anders Hejlsberg, 2000	Microsoft, USA

2.2 Poređenje imperativne i OO paradigme

Objektno orijentisano programiranje

- OOP se razlikuje od imperativnog programiranja najviše po načinu pristupa problemu
- OOP pomera fokus sa algoritama na podatke.
- Dok u imperativnoj paradigmi imamo *top-down* proces, tj funkcionalnu dekompoziciju problema, u OOP su faze analize i dizajna softvera značajno kompleksnije i zahtevnije
- OOP se ostvaruje kôd koji je ponovo upotrebljiv i koji se lakše po potrebi modifikuje i nadograđuje

Poređenje strukturne i OO paradigme

Table 1.6 Difference between Structured and OO Programming

Structured Programming	Object-Oriented Programming
Top-down approach is followed.	Bottom-up approach is followed.
Focus is on algorithm and control flow.	Focus is on object model.
Program is divided into a number of submodules, or functions, or procedures.	Program is organized by having a number of classes and objects.
Functions are independent of each other.	Each class is related in a hierarchical manner.
No designated receiver in the function call.	There is a designated receiver for each message passing.
Views data and functions as two separate entities.	Views data and function as a single entity.
Maintenance is costly.	Maintenance is relatively cheaper.
Software reuse is not possible.	Helps in software reuse.
Function call is used.	Message passing is used.
Function abstraction is used.	Data abstraction is used.
Algorithm is given importance.	Data is given importance.
Solution is solution-domain specific.	Solution is problem-domain specific.
No encapsulation. Data and functions are separate.	Encapsulation packages code and data altogether. Data and functionalities are put together in a single entity.
Relationship between programmer and program is emphasized.	Relationship between programmer and user is emphasized.
Data-driven technique is used.	Driven by delegation of responsibilities.

Osnovne prednosti OOP u odnosu na strukturnu paradigmu

- Lakše održavanje

- Lakša ponovna upotrebljivost koda
- Veća skalabilnost (lakše i brže se grade kompleksne aplikacije)

3 Osnovni pojmovi OOP

3.1 Apstrakcija, interfejs, implementacija, enkapsulacija

Apstrakcija

- Apstrakcija je skup osnovnih koncepata koje neki entitet obezbeđuje sa ciljem omogućavanja rešavanja nekog problema.
  - Apstrakcija uključuje **atribute** koji oslikavaju osobine entiteta kao i **operacije** koje oslikavaju ponašanje entiteta.
  - Apstrakcije daju neophodne i dovoljne opise entiteta, a ne njihove implementacione detalje.
- Apstrakcija rezultuje u odvajanju interfejsa i implementacije.
- Šta je interfejs, a šta implementacija?

Interfejs i implementacija

- Veoma je važno znati razliku između interfejsa i implementacije.
- Interfejs je korisnički pogled na to šta neki entitet može da uradi.
- Implementacija vodi računa o internim operacijama interfejsa koji ne moraju da budu poznati korisniku.
- Interfejs govori ŠTA entitet može da uradi, dok implementacija govori KAKO entitet interno radi.

Interfejs i implementacija

Table 1.2 Comparison of interface and implementation

Interface	Implementation
It is user's viewpoint. (**What** part)	It is supplier's viewpoint. (**How** part)
It is used to interact with the outside world.	It describes how the delegated responsibility is carried out.
User is permitted to access the interfaces only.	Functions or methods are permitted to access the data. Thus, supplier is capable of accessing data and interfaces.
It encapsulates the knowledge about the object.	It provides the restriction of access to data by the user.

Enkapsulacija

- Enkapsulacija (učaurivanje) je skup mehanizama koje obezbeđuje jezik (ili skup tehnika za dizajn) za skrivanje implementacionih detalja (klase, modula ili podsistema od ostalih klasa, modula i podsistema). Enkapsulacijom su podaci zaštićeni od neželjenih spoljnih uticaja.

  - Na primer, u većini OO programskih jezika, označavanje privatne promenljive u okviru klase obezbeđuje da druge klase ne mogu da pristupe toj vrednosti direktno, već isključivo putem metoda za pristup i izmenu, ukoliko ih klasa obezbedi. Ovo se naziva skrivanje podataka.
  - Enkapsulacija je širi pojam od skrivanja podataka.

Enkapsulacija

- Sa korisničkog stanovišta, entitet nudi izvestan broj usluga preko interfejsa i skriva implementacione detalje — termin enkapsulacija se koristi za skrivanje implementacionih detalja.
  - Primer: klasa kompleksan broj. Interno, kompleksan broj može da se implementira koristeći njegov realni i imaginarni deo, ili koristeći trigonometrijski zapis kompleksnog broja. U oba slučaja potrebno je da interfejs ponudi uslugu osnovnih računskih operacija nad kompleksnim brojevima, ali će sama implementacija ovih operacija zavisiti od izbora strukture podataka
- Prednosti enkapsulacije su skrivanje informacija i implementaciona nezavisnost (promena implementacije može da se uradi bez promene interfejsa).
  - Izmena interne strukture kompleksnog broja ne utiče na ostatak sistema

3.2 Objekti i klase

Objekat

- **Filozofski**: Entitet koji se može prepoznati.
- **Konceptualno**: Skup odgovornosti.
- U terminima **objektne tehnologije**: Apstrakcija entiteta iz stvarnog sveta.
- **Specifikacijski**: Skup metoda.
- **Implementaciono**: Podaci sa odgovarajućim funkcijama.

Objekat

- Objekti imaju svoje **ponašanje** (operacije).
- Objekti imaju svoje **osobine** (atribute).
- Kako se određuje koje ponašanje i koje osobine su relevantne za neki objekat?
  - Pre svega, kako se određuju potrebni objekti? Zadatak OO analize i dizajna: u okviru analize treba da se sagleda šta je problem, koje su dogovornosti koje sistem treba da ostvari i u tome se pronalaze kandidati za potrebne objekte (objekti imaju svoje odgovornosti).

- Domen problema takođe sugeriše koji su objekti u sistemu potrebni.
  - Na primer, u okviru zdravstvenog sistema sigurno su potrebni lekari, medicinske sestre i pacijenti, dok u okviru fakultetskog sistema potrebni su nastavnici, asistenti i studenti...

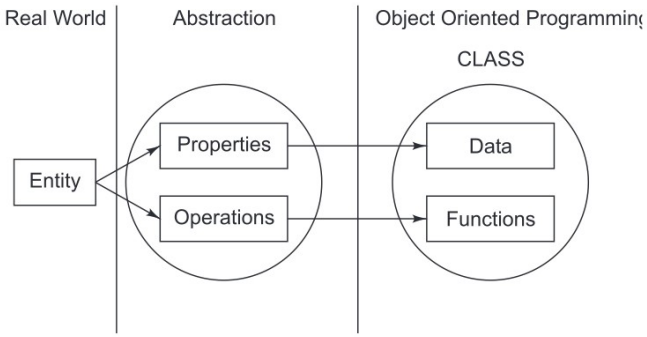
Objekti i poruke

- Objekti međusobno komuniciraju **slanjem poruka**.
- Poruke su komande koje se šalju objektu sa ciljem da izvrši neku akciju.
- Poruke se sastoje od objekta koji prima poruku, metoda koji treba da se izvrši i argumenata (opciono).
- Na primer: `Panel1.Add(Button1)` — primalac je `Panel1`, metod je `Add` a argument je `Button1`

Klase

- U svetu postoji previše objekata i oni se klasifikuju u kategorije, ili u klase.
- **Klase** su šabloni za grupe objekata, tj klase daju specifikaciju za sve podatke i ponašanja objekata.
- Za objekat kažemo da je instanca klase.
- Postoje različiti **odnosi između klasa**, osnovni su **nasleđivanje** i **agregacija**. Prema tome, i objekti mogu da budu specijalni slučajevi drugih objekata (nasleđivanje) ili da sadrže druge objekte (agregacija).

Klasa



Odnos klase i objekta

Table 1.3 Comparison of Class and Object

Class	Object
Class is a data type.	Object is an instance of class data type.
It generates object.	It gives life to a class.
It is the prototype or model.	It is a container for storing its features.
Does not occupy memory location.	It occupies memory location.
It cannot be manipulated because it is not available in the memory.	It can be manipulated.

Konstruktori i destruktori

- Klase definišu kreiranje i uništavanje objekata: konstruktori i destruktori
- **Konstruktori** obezbeđuju da se objekti ispravno inicijalizuju.
- **Destruktori** obezbeđuju da objekat oslobodi sve resurse koje je zauzimao dok je bio aktivan.

Objektno-zasnovni jezici i objektno-orijentisani jezici

- Apstrakcija vođena podacima je osnovni koncept objektnih pristupa.
- U zavisnosti od prisutnosti ostalih koncepata, jezik može da bude obejektno-zasnovan ili objektno-orijentisan.
- **Objektno zasnovan** jezik podržava enkapsulaciju i identitet objekta (jedinstvene osobine koje ga razdvajaju od ostalih objekata) i nema podršku za polimorfizam, nasleđivanje, komunikaciju porukama iako to može da se emuliraju.
- **Objektno orijentisani** jezici imaju sve osobine objektno zasnovanih jezika, zajedno sa nasleđivanjem i polimorfizmom.
- Primetimo da je OO stil programiranja delom nezavisan od jezika i da se može koristiti i u imperativnim jezicima (kroz strukture podataka, funkcije, pokazivače na funkcije), ali da je to onda veoma zahtevno i da je zapravo potrebno imati podršku ovim konceptima u samom jeziku

3.3 Nasleđivanje

Nasleđivanje

- **Nasleđivanje** je kreiranje novih klasa (izvedenih klasa) od postojećih klasa (osnovnih ili baznih klasa).
- Kreiranje novih klasa omogućava postojanje hijerarhije klasa koje simuliraju koncept klasa i potklasa iz stvarnog sveta.
- Primer: životinja, sisar, mačka, pas, riba, štuka, som ...
- Primer: vozilo, kopneno, vazdušno, vodeno, automobil, kamion, brod, čamac, avion, jedrilica
- Nasleđivanje omogućava proširivanje i ponovno korišćenje postojećeg koda, bez ponavljanja i ponovnog pisanja koda. Bazne klase se mogu koristiti puno puta.

Nasleđivanje

- Nasleđivanje je korisno za proširivanje i specijalizaciju.
  - **Proširivanje** koristi nasleđivanje da se razviju klase od postojećih dodavanjem novih osobina.
    - Na primer, u okviru stambene zgrade može postojati deo za poslovni prostor, pa je klasu `StambenaZgrada` potrebno proširiti tako da može da prati i poslovni prostor.
  - **Specijalizacija** koristi nasleđivanje da se preciznije definiše ponašanje opšte (apstraktne) klase. Potklase obezbeđuju specijalizovano ponašanje na osnovu zajedničkih elemenata koje obezbeđuje bazna klasa.
    - Na primer, bazna klasa može da bude student, a potklase koje obezbeđuju specijalizovano ponašanje su klase `StudentOsnovnihStudija`, `StudentMasterStudija` i `StudentDoktorskihStudija`.

Nasleđivanje

- Prilikom nasleđivanja, potklasa može da doda nova ponašanja i nove podatke koji su za nju specifične, ali takođe može i da izmeni ponašanje koje je nasledila od bazne klase kako bi njene specifičnosti bile uzete u obzir (polimorfizam).
- Poželjno je da svaka osobina koja važi za baznu klasu važi i za njenu podklasu, ali obrnuto ne mora da važi (npr, svaki pas ima sve osobine životinje, ali svaka životinja ne mora da ima sve osobine psa).
- Potklase se mogu tretirati kao bazne klase, jer sadrže sve atribute i metode kao i bazne klase tako da kôd koji je razvijen za rad sa baznim klasama može da se primeni i na potklase.

Primer

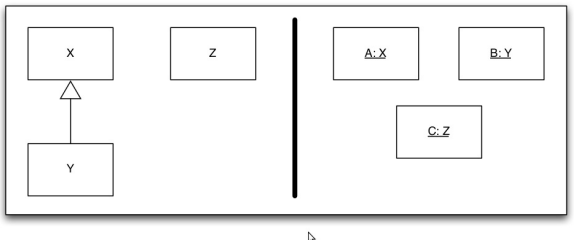
- Problem: imamo različite klase studenata (osnovne, master i doktorske studije) i treba da napravimo leksikografski sortirani spisak studenata.
- Rešenje: ukoliko su sve tri klase studenata podklase bazne klase `Student`, onda se svi studenti mogu staviti u istu kolekciju (npr u niz ili listu) i mogu se tretirati na isti način, bez obzira na njihove specifičnosti.
- Ne samo što možemo sve studente da stavimo u istu kolekciju, već ih možemo tretirati na isti način, pri čemu će se svaki student ponašati u skladu sa svojim specifičnostima (polimorfizam).

Klase definišu vidljivost

- Klase definišu vidljivost osobina svojim objektima: atribut ili metod može da bude javan, zaštićen ili privatan.
- Javna vidljivost omogućava svima da pristupe atributu ili metodu.
- Zaštićena vidljivost omogućava pristup samo potklasama date klase.
- Privatna vidljivost omogućava pristup samo u okviru same klase.

Vidljivost — primer

- Klasa Y nasleđuje klasu X
- Objekat A je instanca klase X
- Objekat B je instanca klase Y koja je potklasa klase X
- Objekat C je instanca klase Z koja je nezavisna od klasa X i Y



- Javna vidljivost karakteristika klase X znači da A, B i C mogu da im pristupe
- Zaštićena vidljivost karakteristika klase X znači da A i B mogu da im pristupe, ali da C ne može
- Privatna vidljivost karakteristika klase X znači da samo A može da im prisupi
- Ovo su, naravno, opšte definicije, različiti jezici implementiraju vidljivost na različite načine

Višestruko nasleđivanje

- Klasa može da se izvede kroz nasleđivanje iz više od jedne osnovne klase — **višestruko nasleđivanje**.
- Instance klase koja koristi višestruko nasleđivanje ima osobine koje nasleđuje od svake bazne klase.
- Primer mačka (životinja, ljubimac, član porodice), kuče (životinja, prijatelj, ljubimac), pegaz (konj, ptica)...
- Višestruko nasleđivanje ima očigledne prednosti, ali ima i mane (kompleksnost, uvođenje virtuelnog nasleđivanja, teže debagovanje)
- Podržano je u raznim programskim jezicima: C++, Eiffel, Python, OCaml...

Mix-in nasleđivanje

- Višestruko nasleđivanje uvodi kompleksnost u semantiku programskih jezika, kao i u implementaciju izvršavanja
- Java ne podržava višestruko nasleđivanje, ali podržava implemetiranje interfejsa, što do neke mere liči na višestruko nasleđivanje, a naziva se *mix-in* nasleđivanje
- **Mix-in** nasleđivanje: Java, C#, Scala, Objective-C, Swift, Go, Ada 2005, Ruby

Apstraktne klase

- **Apstraktne klase** definišu ponašanja koja su relevantna za sve podklase.
- Apstraktne klase definišu potpise metoda koje potklase treba da implementiraju, definišu metode za ponašanja koja su zajednička i definišu podatke koji su zajednički i korisni za sve potklase.

- Apstraktne klase ne mogu da se instanciraju — instanciraju se konkretne klase, dok se pristupa preko interfejsa definisanog od strane apstraktne klase.
- Klase koje se nalaze na vrhu hijerarhije najčešće su apstraktne klase, dok su klase koje su na dnu hijerarhije obično konkretne klase.

4 Polimorfizam

Polimorfizam

- **Polimorfizam** — više formi
- Polimorfizam je ključni faktor ponovne iskoristivosti koda
- Polimorfizam — u OO terminologiji, najčešće se odnosi na predefinisanje/nadjačavanje metoda, odnosno na **hijerarhijski** polimorfizam koji počiva na konceptu nasleđivanja klasa
- Polimorfizam može da se odnosi i na mogućnost zadavanja istog imena funkcijama koje se razlikuju po tipu i/ili broju argumenata (**ad-hoc** polimorfizam, vezan za preopterećivanje funkcija. Osim u OO jezicima, može da bude prisutan i u drugim paradigmama, npr u imperativnim, funkcionalnim, skript jezicima)
- **Parametarski** polimorfizam ne mora da bude vezan za OO jezike, ali je često u njima prisutan. On odgovara generičkom programiranju
- **Implicitni** polimorfizam je uopštenje parametarskog i prisutan je npr u funkcionalnim jezicima

4.1 Hijerarhijski polimorfizam

Hijerarhijski polimorfizam

- **Hijerarhijski polimorfizam** — kôd napisan da radi sa objektima jedne klase može da radi i sa objektima svih njenih potklasa
- Dakle, možemo da tretiramo objekte kao da su objekti bazne klase ali od njih da dobijamo specifično ponašanje koje odgovara konkretnoj potklasi
- **Predefinisanje/nadjačavanje** (engl. *overriding*) — mogućnost objekata da odgovore na iste poruke na različiti način. Ukoliko u baznoj i izvedenoj klasi imamo metod koji ima isti potpis, onda kažemo da je izvedena klasa zapravo predefinisala metod iz bazne klase.
- U ovom slučaju, polimorfizam se odnosi na kasno (odnosno dinamičko) vezivanje poziva za jednu od više različitih implementacija metoda u hijerarhiji nasleđivanja.

Hijerarhijski polimorfizam

- Dinamičko vezivanje je "odlučivanje" kojaće metoda biti pozvana u vreme izvršavanja programa.
- Primer: Pretpostavimo da imamo baznu klasu `Zivotinja` i iz nje izvedene klase `Macka`, `Pas` i `Konj`. Možemo formirati niz pokazivača na klasu `Zivotinja` kojima u zavisnosti od situacije možemo dodeliti da pokazuju na različite objekte klasa `Macka`, `Pas` ili `Konj`. Ako su izvedene klase predefinisale neku metodu `f` klase `Zivotinja`, želimo da pozivom te metode uz pomoć pokazivača na klasu `Zivotinja` bude pozvana odgovarajuća predefinisana metoda i to iz klase `Macka` ukoliko pokazivač u fazi izvršavanja pokazuje na mačku, iz klase `Pas` ukoliko pokazivač u fazi izvršavanja pokazuje na psa ili iz klase `Konj`, ukoliko pokazivač u fazi izvršavanja pokazuje na konja.

Polimorfizam — dinamičko vezivanje

- Da bi se koristilo dinamičko vezivanje, potrebno je koristiti odgovarajuće mehanizme jezika (virtuelne metode).
- Cena dinamičkog vezivanja je kreiranje tabela virtulenih funkcija.
- Za svaki objekat kompajler implicitno dodaje pokazivač na tabelu virtuelnih funkcija.
- Na osnovu ove tabele određuje se u fazi izvršavanja koja će tačno metoda biti pozvana.

4.2 Ad-hoc polimorfizam

Ad-hoc polimorfizam

- Ad-hoc polimorfizam je primer statičkog polimorfizma, koji nije nužno vezan za OO jezike
- Statičko vezivanje je "odlučivanje" koja će metoda biti pozvana u vreme prevođenja.
- **Preopterećivanje** (engl. *overloading*) — isto ime ili operator može da bude pridružen različitim operacijama, u zavisnosti od tipa podataka koji mu se proslede. Na primer `int f() {...}` i `int f(int x) {...}`

Ad-hoc polimorfizam

- Preopterećivanje se odlučuje statički.
- Pretpostavimo da postoje dve metode sa istim imenom u istoj klasi koje se razlikuju po broji i/ili tipu argumenata — odluka o tome koja će od ove dve metode biti pozvana može se doneti u fazi prevođenja i to tako što se izvrši poređenje tipova argumenata, a ukoliko postoji dvosmislenost onda prevodilac javlja grešku.
- Ako ne postoji metod sa datim imenom u izvedenoj klasi onda se on traži u baznoj klasi.

4.3 Parametarski polimorfizam

Parametarski polimorfizam — generičko programiranje

- Stroga tipiziranost ima kao posledicu da se i jednostavne funkcije moraju definisati više puta da bi se mogle upotrebljavati na raznim tipovima.
- Generičko programiranje omogućava prevazilaženje ograničenja strogo tipiziranih jezika (*templates* C++, *generics* Java).
- Na primer, generičko programiranje se korsiti za definisanje generičkih
  - funkcija, npr funkcija koja računa minimum dva prirodna broja, dva realna broja, dva razlomka, sortiranje raznih tipova...
  - klasa, npr klasa niz, lista, skup, ...

4.4 Implicitni polimorfizam

Implicitni polimorfizam

- Implicitni polimorfizam — uopštenje parametarskog
- Implicitni polimorfizam podrazumeva da se pri pisanju koda uopšte ne navode tipovi vrednosti i izraza.
- Pretpostavlja se da će prevodilac analizirati svaki konkretan segment programskog koda i sam zaključiti za koje tipove može da se prevede.
- Javlja se i u statički i u dinamički tipiziranim jezcima
- Primer: tipovi u Haskell-u
- Specifični primer su i makroi u programskom jeziku C

4.5 OO jezici i njihove osobine

OO jezici i njihove osobine

(b) OO programming languages and comparison of their features

Feature	Java	C++	Smalltalk	Objective C	Simula	Ada	Eiffel	C#
Encapsulation	√	√	Poor	√	√	√	√	√
Single inheritance	√	√	√	√	√	X	√	√
Multiple inheritance	X	√	X	√	X	X	√	X
Polymorphism	√	√	√	√	√	√	√	√
Binding (early or late)	Late	Both	Late	Both	Both	Early	Early	Late
Concurrency	√	Poor	Poor	Poor	√	Difficult	√	√
Garbage collection	√	X	√	√	√	X	√	√
Persistent objects	X	X	promised	X	X	Like 3GL	Limited	X
Genericity	√	√	X	X	X	√	√	√
Class libraries	√	√	√	√	√	Limited	√	√

5 Pitanja i literatura

5.1 Pitanja

Pitanja

- Koji su principi funkcionalne dekompozicije i koji su osnovni problemi ovogo pristupa?
- Šta je osnovni uzrok problema kod rešavanja funkcionalnom dekompozicijom?
- Zašto je uticaj izmena zahteva važan?
- Šta je kohezija, a šta kopčanje i kako su povezani?
- Šta je efekat talasanja i da li je on poželjan?
- Koji su bili simptomi prve softverske krize?

Pitanja

- Šta je apstrakcija?
- Šta je interfejs?
- Šta implementacija?
- Objasniti odnos interfejsa i implementacije.
- Šta je enkapsulacija?
- Koji je odnos apstrakcije i enkapsulacije?

Pitanja

- Šta je objekat? (filozofski? konceptulano? u objektnoj terminologiji? specifikacijski? implementaciono?)
- Kako komuniciraju objekti?
- Šta je klasa?
- Koji je odnos klase i objekta?
- Koji je prvi objektni jezik i kada je nastao?
- Šta su objektno zasnovani, a šta objektno orijenitsani jezici?
- Koji su najpopularniji objektno orijentisani jezici?

Pitanja

- Šta je nasleđivanje?
- Na koje načine se koristi nasleđivanje? Šta je proširivanje, a šta specijalizacija?
- Šta omogućava nasleđivanje?
- Šta je višestruko nasleđivanje?
- Koji jezici omogućavaju višestruko nasleđivanje, a koji ne?
- Koje su osnovne vidljivosti koje klase definišu?

Pitanja

- Šta je polimorfizam?
- Koja je razlika između preopterećivanja i predefinisanja?
- Kada se koristi statičko a kada dinamičko vezivanje?
- Šta definišu apstraktne klase?
- Koje su mogućnosti generičkog programiranja?
- Obrazložiti sličnosti i razlike strukturnog i OO programiranja?
- Koje su osnovne prednosti OO programiranja u odnosu na strukturno programiranje?

5.2 Literatura

Literatura

- The Object-Oriented Paradigm, Kenneth M. Anderson, 2012
- C++ Izvornik — Lippmna Lajoie
- Object Oriented Programming with Java: Essentials and Applications — Rajkumar Buyya, S. Thamarai Selvi, Xingchen Chu http://www.buyya.com/java/Chapter1.pdf