# Oblikovanje programske potpore

2012./2013. grupa P01

## Modularizacija i objektno usmjerena arhitektura

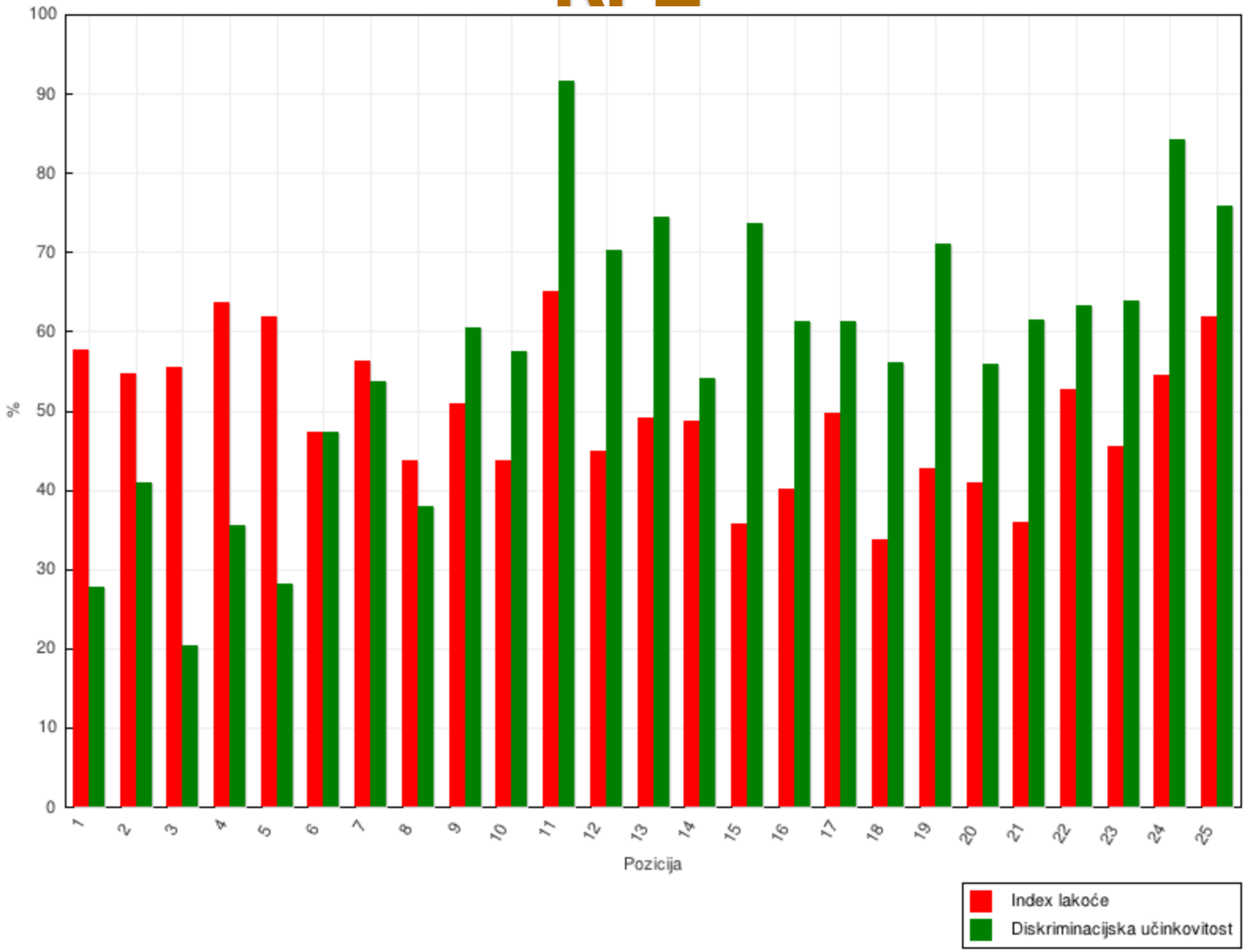
Prof.dr.sc. Vlado Sruk

***Sveučilište u Zagrebu***
***Fakultet elektrotehnike i računarstva***
*Zavod za elektroniku, mikroel., računalne i inteligentne sustave*

# Tema

- Podsjetnik
  - metode programskog inženjerstva
- Programske paradigme
- Objektno usmjerena paradigma
- Koncepti objektnog usmjerenja
  - objekt, Razred, Nasljeđivanje, Polimorfizam

# KPZ


100
90
80
70
60
50
40
30
20
10
0
%
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
Pozicija
Index lakoće
Diskriminacijska učinkovitost

# Literatura

- Timothy C. Lethbridge, Robert Laganière: ***Object-Oriented Software Engineering: Practical Software Development using UML and Java***, McGraw Hill, 2001.
  - http://www.lloseng.com
- Sommerville, I., ***Software engineering***, 8th ed, Addison Wesley, 2007.
- O'Docherty, Mike: ***Object-oriented analysis and design : understanding system development with UML 2.0 / Mike O'Docherty***, John Wiley & Sons Ltd, 2005
- Šribar, J.; Motik, B.: ***Demistificirani C++,*** Element, 2001 ("Dobro upoznajte protivnika da biste njime ovladali")

# Programsko inženjerstvo

- ***Sustavan i organiziran*** pristup procesu izrade programske potpore;
- Upotrebljavati ***prikladne alate i tehnike*** ovisno o ***problemu*** koji treba riješiti, ***ograničenjima*** u procesu izrade i postojećim ***resursima***.

# Metode programskog inženjerstva

- Strukturni pristup razvoju i oblikovanju programske potpore
  - modele sustava;
  - notaciju (označavanje);
  - pravila;
  - preporuke i napuci.
- Opisi modela
  - najčešće grafički
- Pravila
  - ograničenja primijenjena na modele sustava
- Preporuke
  - “dobra inženjerska praksa”
- Naputke o procesu
  - slijed aktivnosti

# Inženjerstvo zahtjeva

- To je postupak ***pronalaženja, analiziranja, dokumentiranja i provjere*** zahtijevanih usluga sustava, te ograničenja u uporabi.
- Zahtjevi sami za sebe su opisi usluga sustava i ograničenja koja se generiraju tijekom procesa inženjerstva zahtjeva.
- Obzirom na razinu detalja razlikujemo:
  - ***specifikacija visoke razine apstrakcije***
    - obično u okviru ponude za izradu programskog produkta = ***korisnički zahtjevi***. Pišu se u prirodnom jeziku i grafičkim dijagramima. Moraju biti razumljivi netehničkom osoblju.
  - ***vrlo detaljna specifikacija***
    - uobičajeno nakon prihvaćanja ponude, a prije sklapanja ugovora = ***zahtjev sustava***.
    - Pišu se strukturiranim prirodnim jezikom, posebnim jezicima za oblikovanje sustava, dijagramima i matematičkom notacijom.
  - ***specifikacija programske potpore***
    - najdetaljniji opis i objedinjuje korisničke i zahtjeve sustava.

# Procesi inženjerstva zahtjeva

- Procesi koji su u upotrebi u ***inženjerstvu zahtjeva*** razlikuju se ovisno o domeni primjene, ljudskim resursima i organizaciji koja oblikuje zahtjeve.
- Postoje neke generičke aktivnosti zajedničke svim procesima:
  - izlučivanje zahtjeva (*engl. requirements elicitation*)
  - analiza zahtjeva
  - validacija zahtjeva
  - upravljanje zahtjevima

# Oblikovanje programske potpore

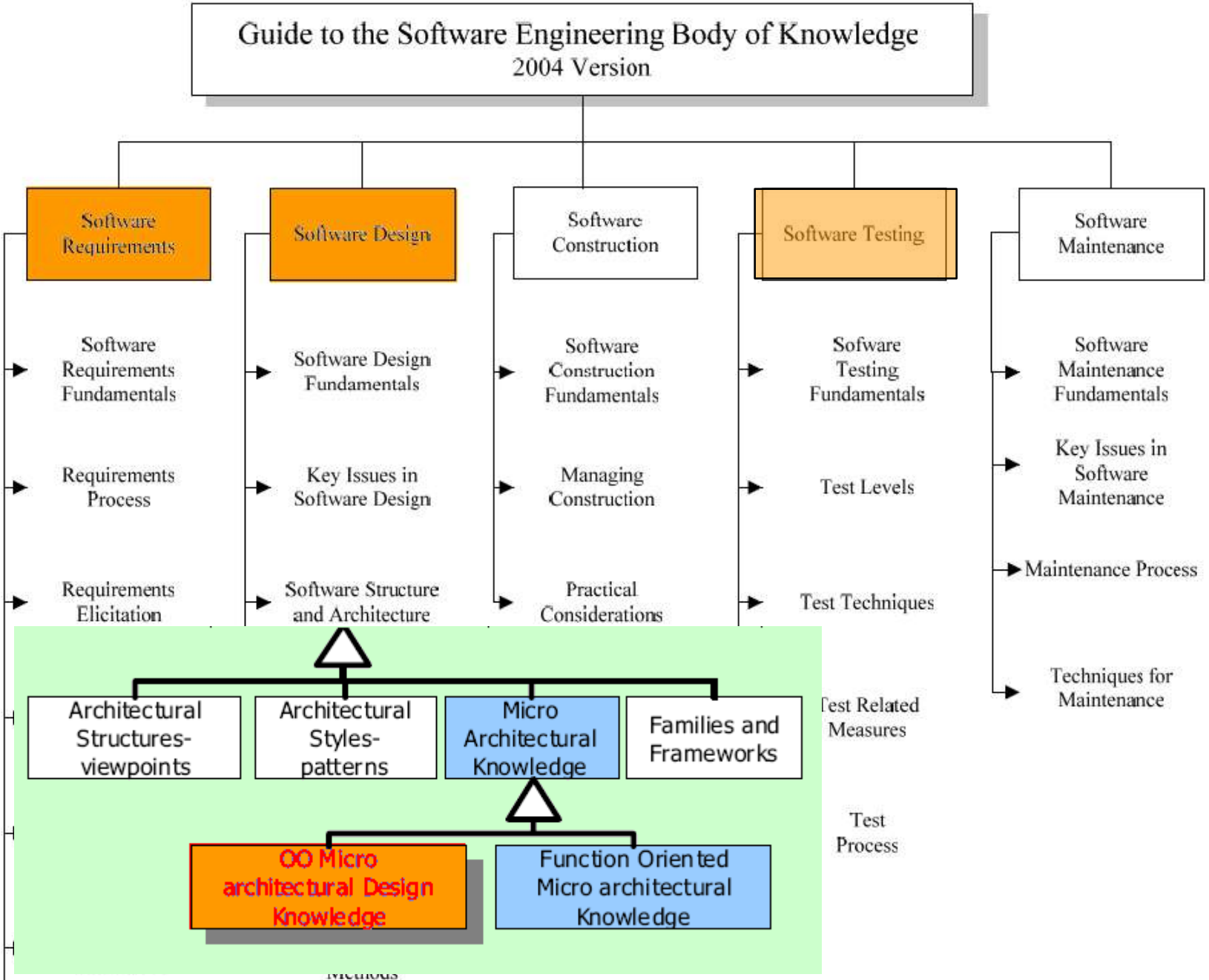
- Inženjerstvo zahtjeva
  - Što graditi?
- Oblikovanje
  - Kako graditi?
- Podjela u dvije faze
  - oblikovanje arhitekture
    - *engl. High-level design*
    - rezultat je arhitektura/struktura programa
  - implementacija
    - *engl. Detailed/Low-level design*
    - rezultira podatkovnim strukturama i algoritmima pojedinih modula
    - detaljno oblikovanje moguće je izravno implementirati uporabom programskog jezika

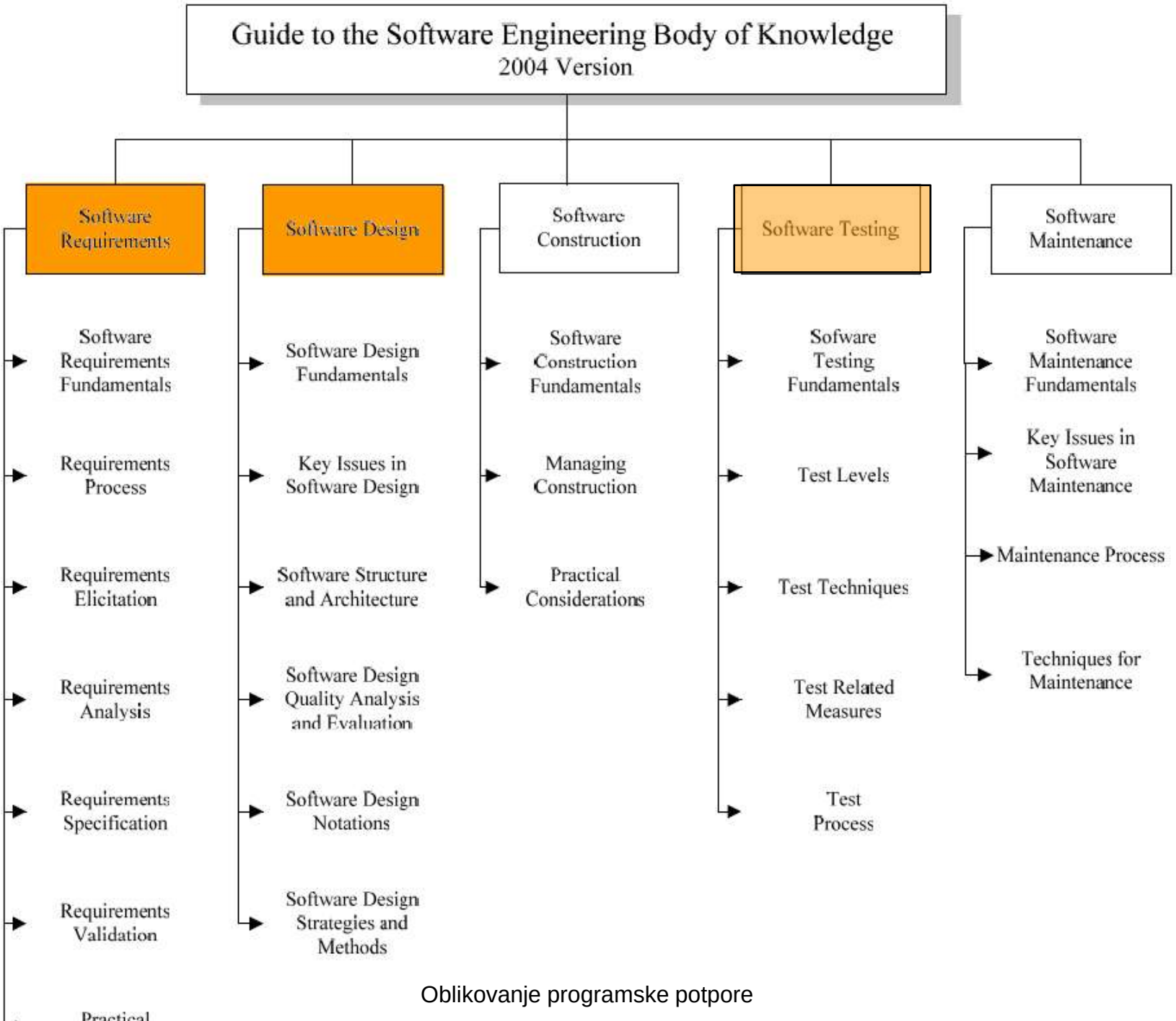
# Principi oblikovanja

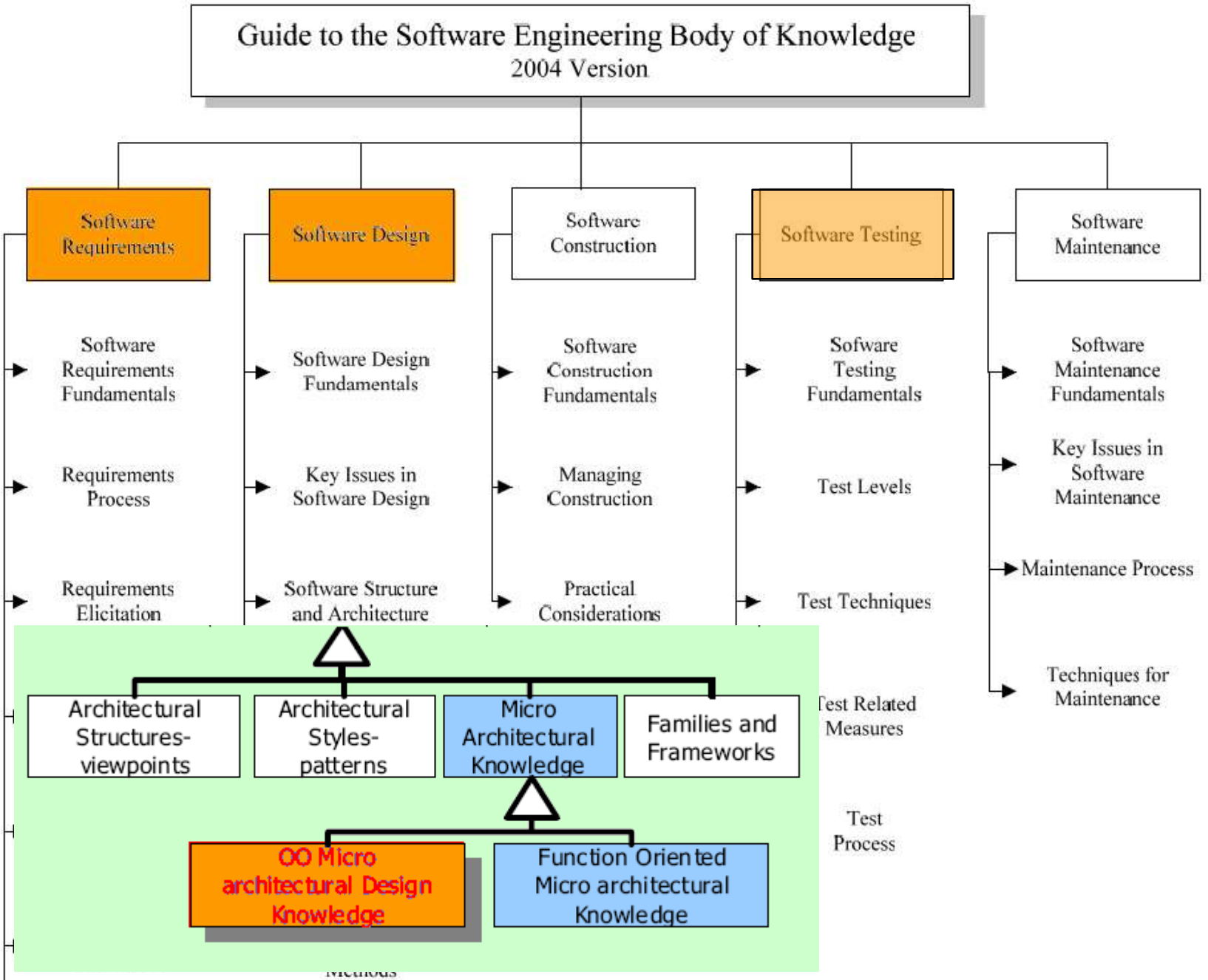
- Dekompozicija i modularizacija
- Apstrakcija, međuovisnost , koheziju
- Sakrivanje informacija
  - *engl. encapsulation/information hiding*
- Odvajanje sučelja i implementacije
- Samodostatnost i kompletnost
  - komponenta obuhvaća samo važna svojstva
- Održavanje

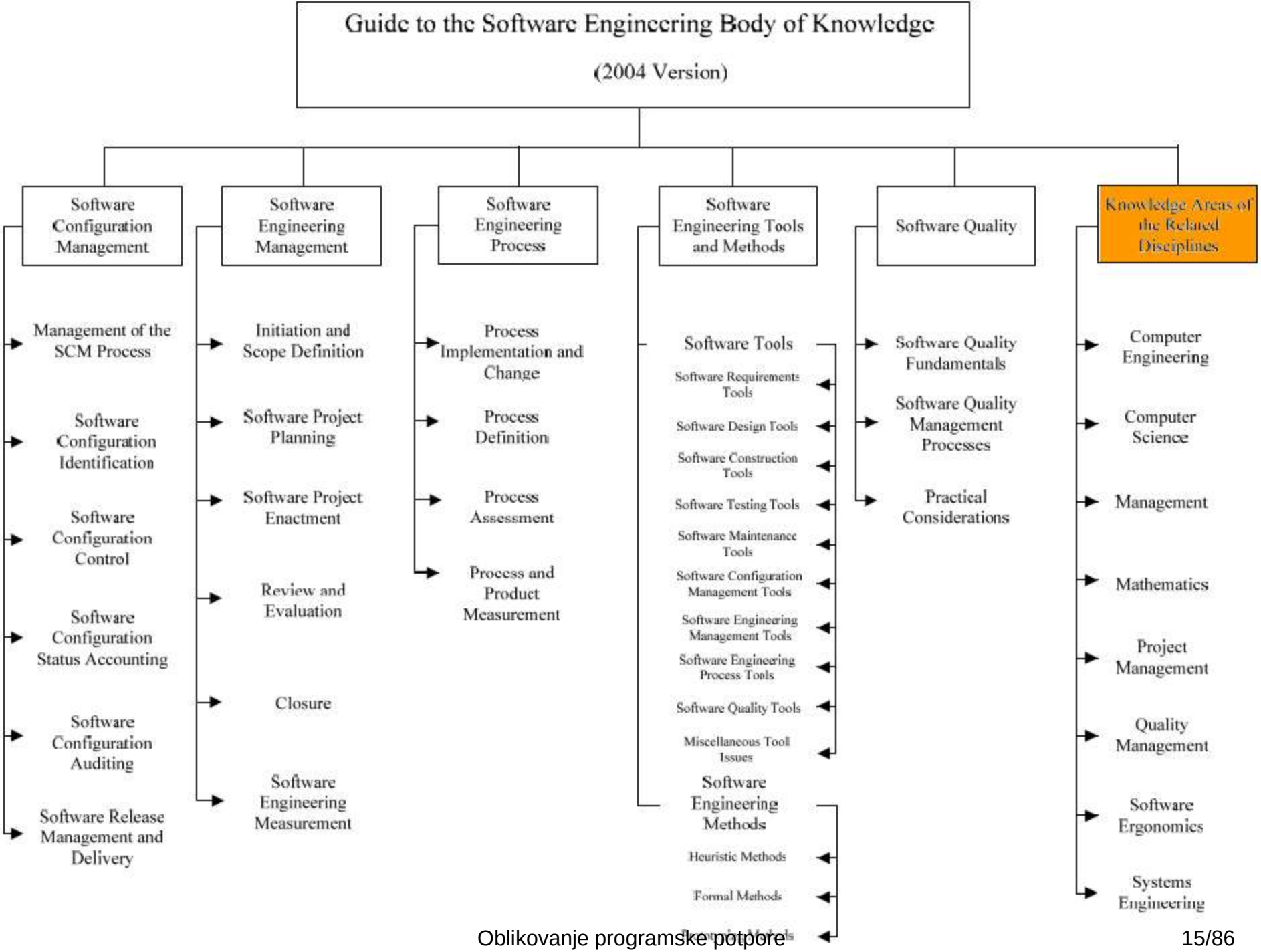
# Područje programskog inženjerstva

- Guide to the Software Engineering Body of Knowledge
- Http://www.computer.org/portal/web/swebok
  - 2004 Version SWEBOK®
  - a project of the IEEE Computer Society Professional Practices Committee
- U izradi nova inačica SWEBOK V3 Review

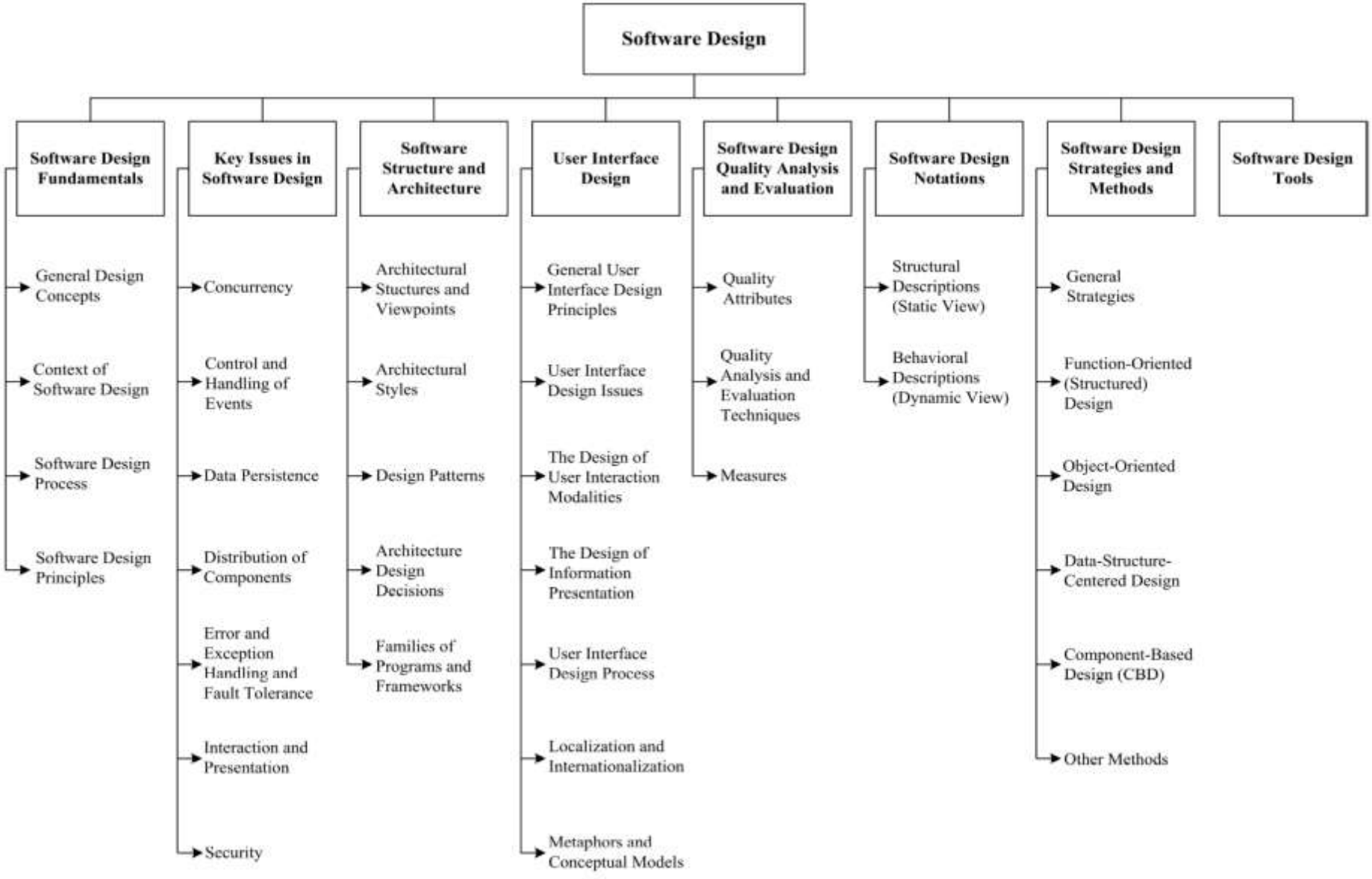

Guide to the Software Engineering Body of Knowledge
2004 Version
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Requirements Fundamentals
Requirements Process
Requirements Elicitation
Software Design Fundamentals
Key Issues in Software Design
Software Structure and Architecture
Software Construction Fundamentals
Managing Construction
Practical Considerations
Sofware Testing Fundamentals
Test Levels
Test Techniques
Software Maintenance Fundamentals
Key Issues in Software Maintenance
Maintenance Process
Techniques for Maintenance
Test Related Measures
Test Process
Architectural Structures- viewpoints
Architectural Styles- patterns
Micro Architectural Knowledge
Families and Frameworks
OO Micro architectural Design Knowledge
Function Oriented Micro architectural Knowledge


Guide to the Software Engineering Body of Knowledge
2004 Version
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Requirements Fundamentals
Requirements Process
Requirements Elicitation
Requirements Analysis
Requirements Specification
Requirements Validation
Software Design Fundamentals
Key Issues in Software Design
Software Structure and Architecture
Software Design Quality Analysis and Evaluation
Software Design Notations
Software Design Strategies and Methods
Software Construction Fundamentals
Managing Construction
Practical Considerations
Sofware Testing Fundamentals
Test Levels
Test Techniques
Test Related Measures
Test Process
Software Maintenance Fundamentals
Key Issues in Software Maintenance
Maintenance Process
Techniques for Maintenance


Guide to the Software Engineering Body of Knowledge
2004 Version
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Requirements Fundamentals
Requirements Process
Requirements Elicitation
Software Design Fundamentals
Key Issues in Software Design
Software Structure and Architecture
Software Construction Fundamentals
Managing Construction
Practical Considerations
Sofware Testing Fundamentals
Test Levels
Test Techniques
Software Maintenance Fundamentals
Key Issues in Software Maintenance
Maintenance Process
Techniques for Maintenance
Test Related Measures
Test Process
Architectural Structures- viewpoints
Architectural Styles- patterns
Micro Architectural Knowledge
Families and Frameworks
OO Micro architectural Design Knowledge
Function Oriented Micro architectural Knowledge


Guide to the Software Engineering Body of Knowledge
(2004 Version)
Software Configuration Management
Management of the SCM Process
Software Configuration Identification
Software Configuration Control
Software Configuration Status Accounting
Software Configuration Auditing
Software Release Management and Delivery
Software Engineering Management
Initiation and Scope Definition
Software Project Planning
Software Project Enactment
Review and Evaluation
Closure
Software Engineering Measurement
Software Engineering Process
Process Implementation and Change
Process Definition
Process Assessment
Process and Product Measurement
Software Engineering Tools and Methods
Software Tools
Software Requirements Tools
Software Design Tools
Software Construction Tools
Software Testing Tools
Software Maintenance Tools
Software Configuration Management Tools
Software Engineering Management Tools
Software Engineering Process Tools
Software Quality Tools
Miscellaneous Tool Issues
Software Engineering Methods
Heuristic Methods
Formal Methods
Software Quality
Software Quality Fundamentals
Software Quality Management Processes
Practical Considerations
Knowledge Areas of the Related Disciplines
Computer Engineering
Computer Science
Management
Mathematics
Project Management
Quality Management
Software Ergonomics
Systems Engineering

# Oblikovanje programske potpore

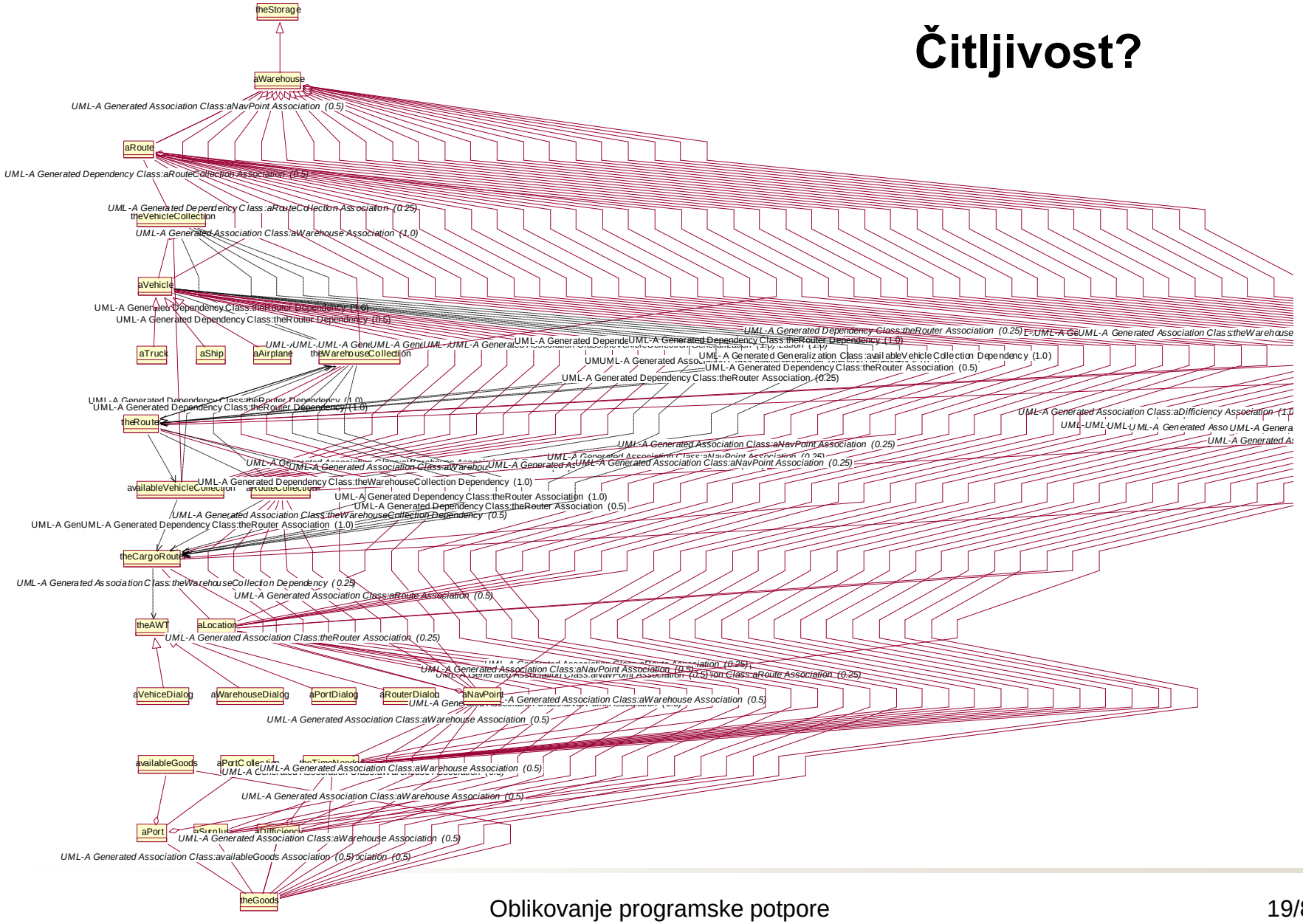


# Problemi u oblikovanju programske potpore

- Ranjivost na globalne - široko dijeljene varijable
  - klasični programski jezici kreiraju dijeljenje (blokovske strukture, globalne varijable).
- Nenamjerno otkrivanje interne strukture
  - vidljiva reprezentacija može se manipulirati na neželjen način.
- Prodiranje odluka o oblikovanju
  - jedna promjena utječe na mnoge module.
- Disperzija koda koji se odnosi na jednu odluku
  - vrlo je teško utvrditi što je sve pogođeno promjenom.
- Povezane odluke o oblikovanju
  - povezane definicije raspršuju odluke
  - trebale bi biti lokalizirane na jednom mjestu

# Primjer programa

## Čitljivost?



# Modularizacija

- Moguće rješenje problema:
- Modularizacija
  - jednostavnije upravljanje sustavom
    - princip podijeli i vladaj
  - evoluciju sustava
    - promjene jednog dijela ne utječu na druge dijelove
  - razumijevanje
    - sustav se sastoji od razumno složenih dijelova

- Koje kriterije koristiti za modularizaciju?
- Što je to modul?
  - dio koda
  - jedinica kompilacije, koja uključuje deklaracije i sučelje.
  - D.Parnas, Comm. ACM, 1972. : “Jedinica posla.”

# Povijest modularizacije

- Glavni program i potprogrami
  - dekompozicija u procesne korake s jednom niti izvođenja.
- Funkcijski moduli
  - agregacija (skupljanje) procesnih koraka u module.
- Apstraktni tipovi podataka
  - *engl. Abstract Data Types - ADT*
  - zatvaranje podataka i operacija, skrivanje predstavljanja.
- Objekti i objektno usmjerena arhitektura
  - procedure (metode) se povezuju dinamički, polimorfizam, nasljeđivanje.
- Komponente i oblikovanje zasnovano na komponentama (*engl. Component based design CBD*)
  - višestruka sučelja, posrednici, binarna kompatibilnost.

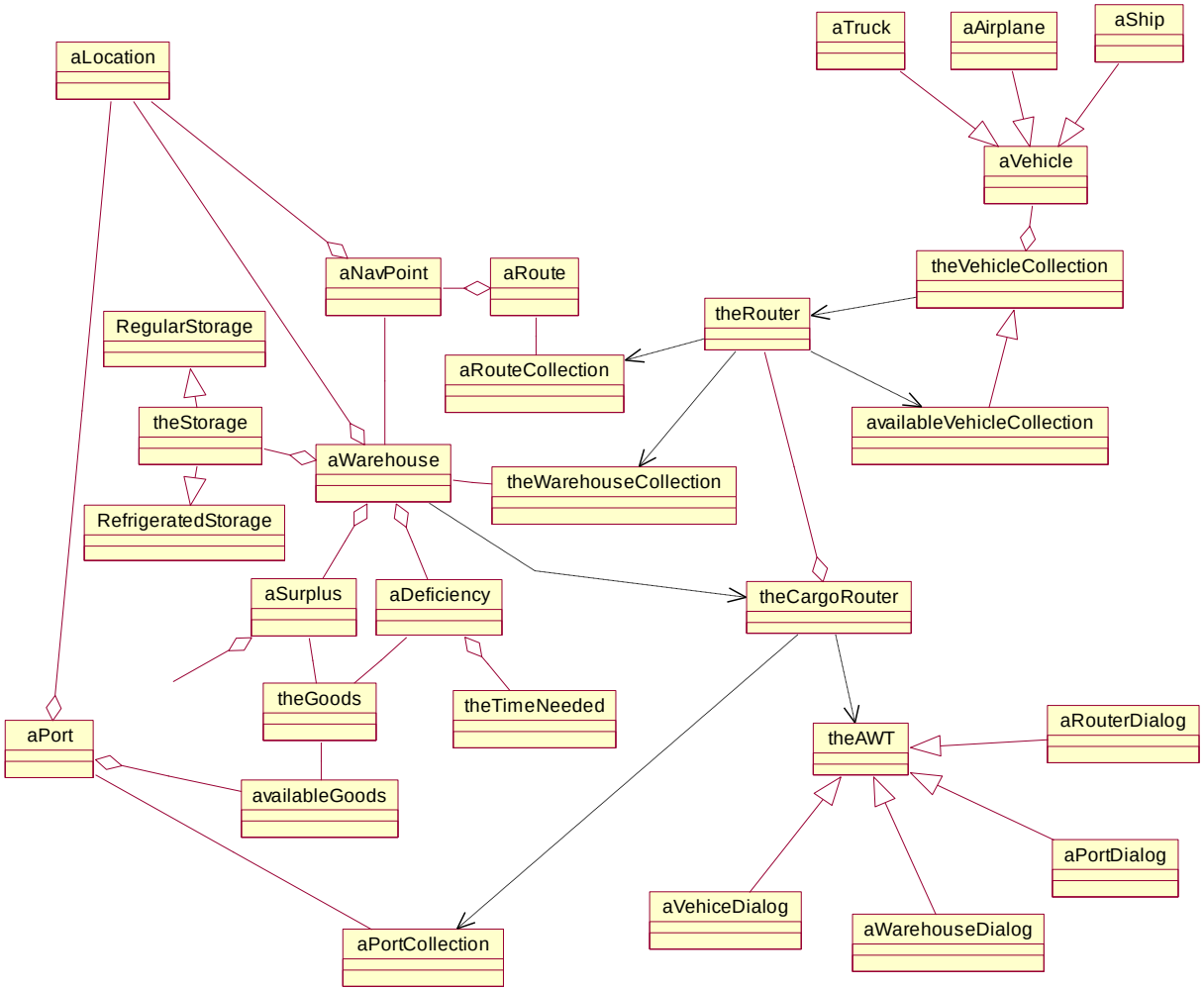
# Glavni program i potprogrami

- Obilježja:
- Hijerarhijska dekompozicija
  - temeljena na odnosu definicija – uporaba. Pozivi procedura su interakcijski mehanizam.
- Jedna nit izvođenja
  - potpomognuto izravno programskim jezikom.
- Hijerarhijsko rasuđivanje
  - ispravno izvođenje programa ovisi o ispravnom izvođenju potprograma koja se poziva.
- Implicitna struktura podsustava
  - potprogrami/subrutine su tipično skupljene u (funkcijske) module.

# Apstraktni tipovi podataka

- ’60 - “Ako dobro definirate strukture podataka ostatak programa je mnogo jednostavniji.”
- ’70 - intuicija o skrivanju informacija i apstraktnim tipovima podataka (***ADT – abstract data types***):
- Definicija:
- Apstraktni tip podataka - ADT je skup dobro definiranih ***elemenata*** i skup pridruženih ***operacija*** na tim elementima definiranih s matematičkom preciznošću ***neovisno o implementaciji***.
  - skup elemenata je jedino dohvatljiv preko skupa operacija.
  - skup operacija se naziva sučelje (*engl. interface*).
  - programski jezici mogu po volji i različito implementirati ADT.
- Taj je pristup potpuno usvojen u objektno usmjerenoj arhitekturi programske potpore.

# Primjer programa: Apstrakcija




**Bolje?**

# Primjer: Apstraktni tipovi podataka

neka ***k*** označuje cijeli broj (*integer*).

***ADT integer***

***podatak –*** *engl. Data*

**tip** elemenata: cijeli brojevi s opcijskim prefiksom plus ili minus.

takav jedan cijeli broj s predznakom neka je ***N***.

**operacije** – *engl. Operations*

***constructor*** – kreira novi cijeli broj.

***add(k)*** – kreira novi cijeli broj koji je suma ***N*** i ***k***. Posljedica ove operacije je ***sum=N+k***. To nije naredba pridruživanja, već matematička operacija koja je **istinita** za svaku vrijednost ***sum***, ***N***, ***k*** nakon operacije ***add***.

***sub(k)*** – slično kao gore kreira novi cijeli broj. Posljedica je ***sum=N-k***.

***set(k)*** – Posljedica je ***N=k***.

…

- Gornji opis naziva se ***specifikacija*** za ***ADT integer***. Imena ***add***, ***sub***, … , predstavljaju ***sintaksu***, dok je ***semantika*** određena preduvjetima i posljedicama (post-uvjetima) operacija.

# Primjer: Apstraktni tip podatka

- Lista kao ADT: je sekvencija 0 ili više objekata danog tipa.
- Operacije: `insert(x,p,L);//ubaci element x, na poziciju p, u listu L`
  `first(L); locate(x,L); retrieve(p, L); delete(p, L);`
  `next(p,L);`

Primjer programa koji eliminira duplikate u listi L:

```
p = first(L);
  while (p != end_of_List) {
    q = next(p,L);
    while(q != end_of_List) {
      if (same(retrieve(p,L),retrieve(q,L))
        delete(q,L);
      else
        q = next(q,L)
    }
    p = next(p, L);
  }
```

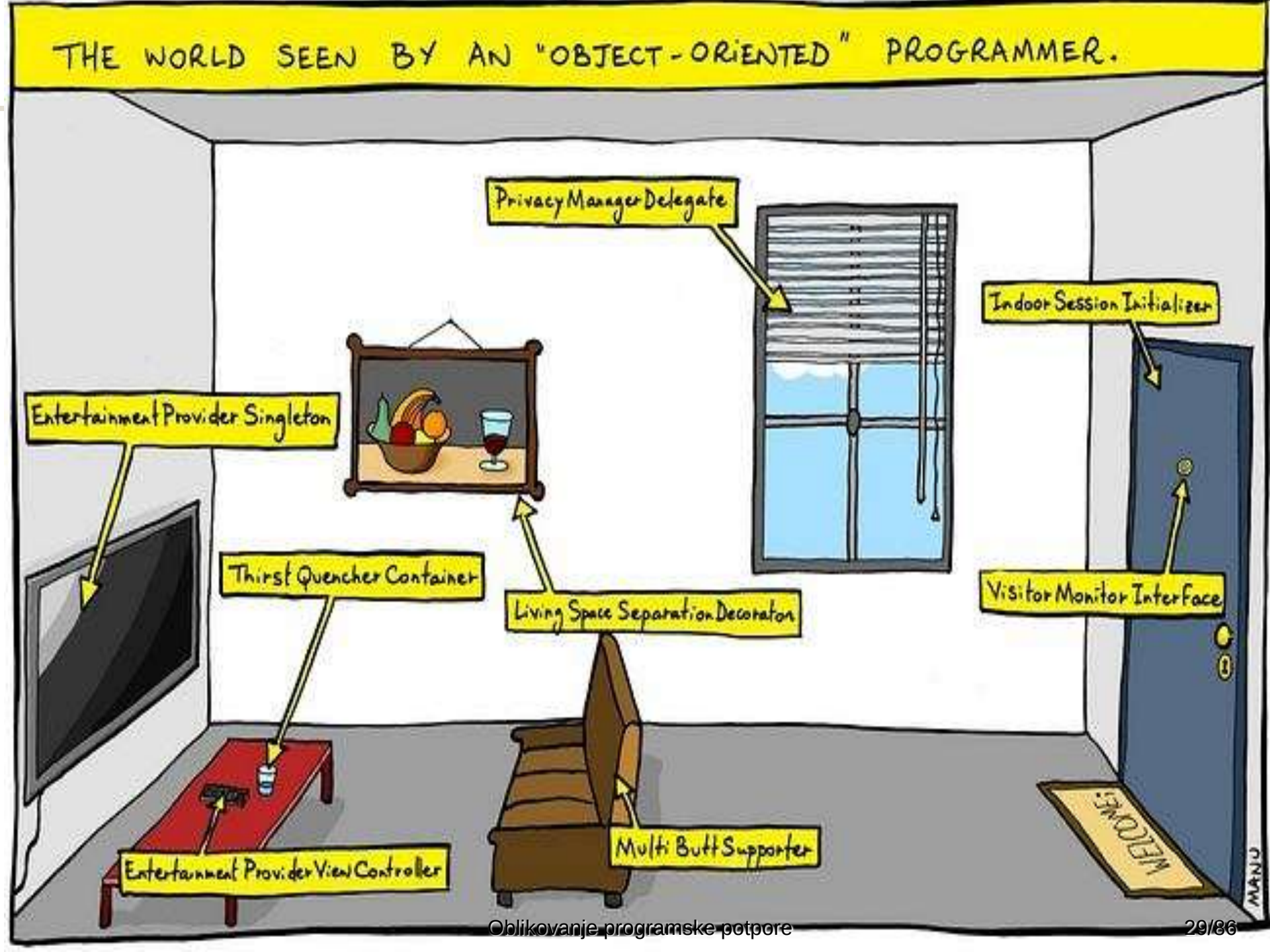
- Prednost
  - nezavisnost o strukturi podataka i moguća uporaba u bilo kojoj implementaciji liste (npr. od niza do povezane liste). Promjena implementacije liste ne utječe na ovaj kod.

# Proceduralna paradigma

- Programska paradigma definira osnovni stil programiranja
  - razlike su u načinu predstavljanja elemenata programa i definiciji koraka za rješavanje problema
- Program je organiziran oko pojma ***procedura*** *(funkcija, rutina)*.
  - ***proceduralna apstrakcija***
    - zadovoljavajuće rješenje za jednostavne podatke
  - *dodaje se* ***podatkovna apstrakcija***
    - Grupiranje dijelova podataka koji opisuju neki entitet i manipulacija s njima kao cjeline.
    - Pomaže u smanjenu složenosti sustava.
      - Npr. ***records*** i ***structures*** *(ali različiti zapisi traže različite procedure).*
  - sustav se promatra kao niz operacija nad procedurama
  - oblikovanje započinje najvišom funkcijom i razlaže na više detaljnih funkcija
  - stanje sustava je centralizirano i dijeljeno između različitih funkcija

# Objektno usmjerena paradigma

- *engl. Object oriented*
- Tehnika modeliranja koja promatra svijet kroz objekte
  - imitacija načina razmišljanja u kojoj se rješenje traži uporabom objekata koji su reprezentacija stvarnih objekata.
  - ne pišemo programe za obradu podataka!!
  - izražavamo ponašanje programskih objekata.
- Objektno usmjerena (orijentirana) paradigma:
  - ***organiziranje proceduralnih apstrakcija u kontekstu podatkovnih apstrakcija.***
- Pristup rješenju problema u kojem se sva izračunavanja (*engl. computations*) obavljaju u kontekstu objekata.
  - sustav se promatra kao skup objekata
  - stanje sustava je decentralizirano
  - svaki objekt ima svoje interne podatke koji opisuju njegovo stanje

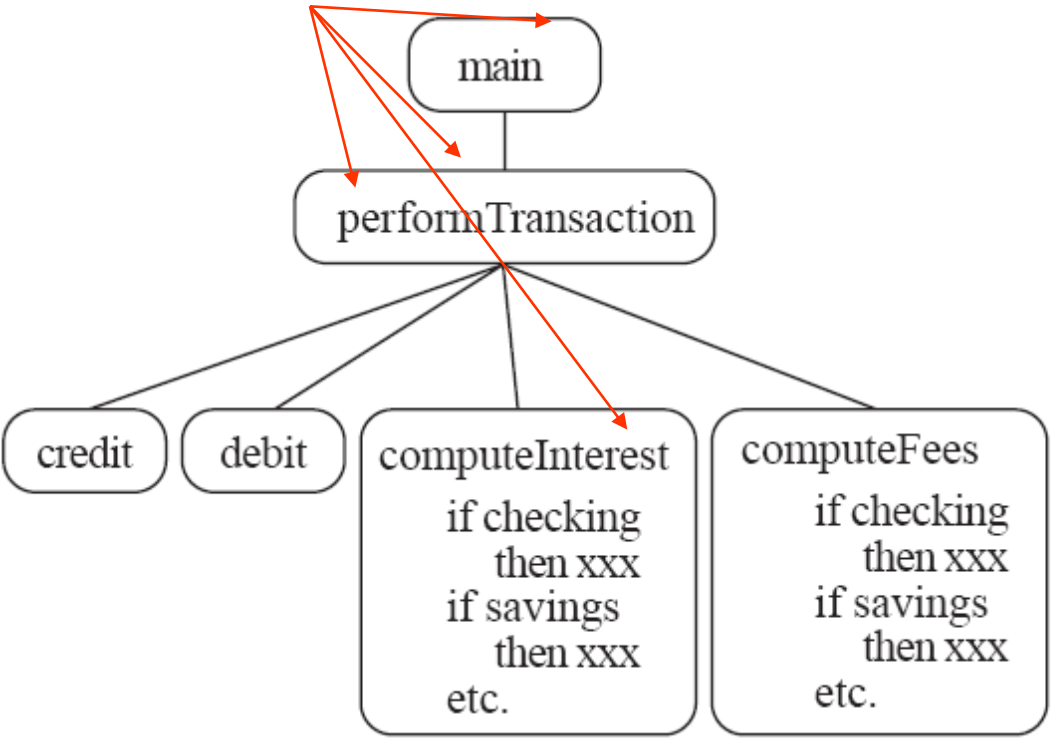

THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Privacy Manager Delegate
Indoor Session Initializer
Entertainment Provider Singleton
Thirst Quencher Container
Living Space Separation Decorator
Visitor Monitor Interface
Entertainment Provider View Controller
Multi Butt Supporter
WELCOME
MANU

# Objektno usmjerena paradigma

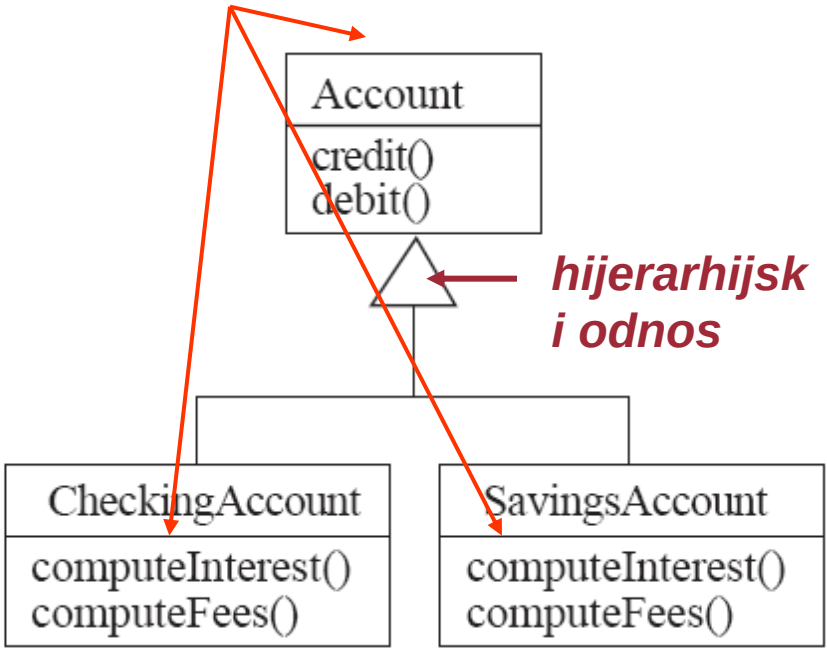
- ***Objekti*** su instance programskih konstrukcija koje nazivamo razredima (klasama).
  - razredi:
    - Podatkovne apstrakcije
    - Sadrže proceduralne apstrakcije koje izvode operacija na objektima.
- Program u radu se može sagledati kao ***skup objekata*** koji u međusobnoj kolaboraciji obavljaju dani zadatak.

# Primjer paradigmi

***Procedure manipuliraju različitim tipovima podataka***




***Procedure su svezane (zatvorene) s podacima***



# Koncepti objektnog usmjerenja

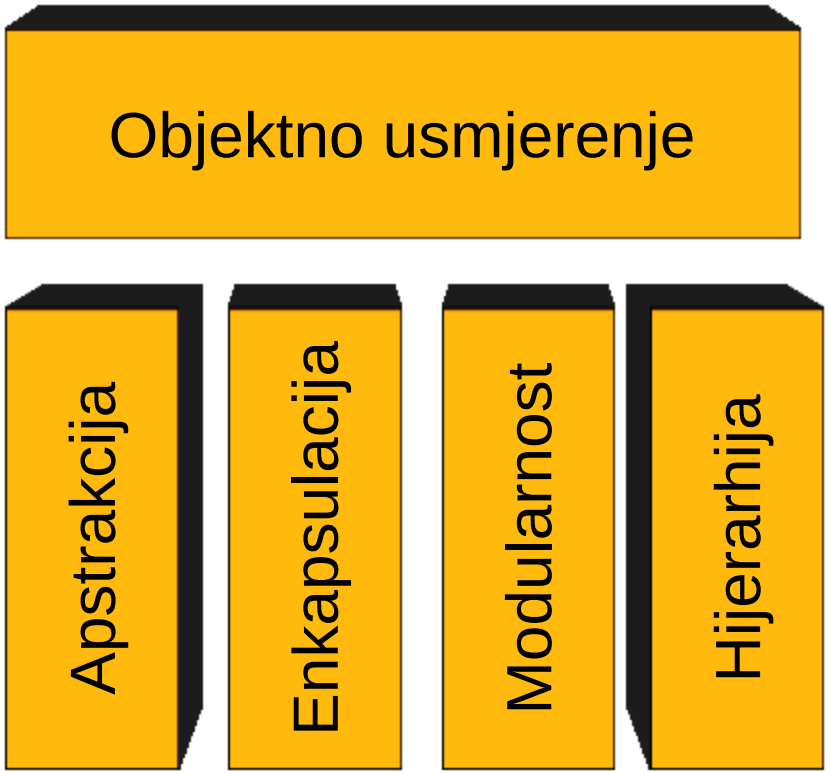
- Nužna obilježja koja definiraju sustav ili programski jezik da bi ga smatrali objektno usmjerenim.
- Neophodno ih je razumjeti za primjenu objektnog usmjerenja
- ***Identitet, Objekt*** (Apstrakcija)
  - svaki objekt je jedinstven i može se referencirati (adresom).
  - dva objekta mogu imati identične podatke ali su jedinstveni
- ***Razredi*** (Apstraktni tip podataka)
  - programski kod je organiziran uporabom koncepta razreda, koji svaki za sebe opisuju skup objekata.
- ***Nasljeđivanje*** (Ponovna uporaba)
  - to je mehanizam u kojem se značajke podrazreda implicitno nasljeđuju od superrazreda.
- ***Polimorfizam*** (Dinamičko povezivanje)
  - mehanizam u kojem postoji više metoda istog naziva koje različito (ovisno o razredu objekta) implementiraju istu apstraktnu operaciju.

# Terminologija objektnog oblikovanja

- Objekt – *engl. Object*
- Razred – *engl. Class*
- Atribut – *engl. Attribute*
- Operacija – *engl. Operation*
- Sučelje – *engl. Interface* (*Polymorphism*)
- Komponenta – *engl. Component*
- Paket – *engl. Package*
- Podsustav – *engl. Subsystem*
- Pridruživanja – *engl. Relationships*

# Osnovni principi objektnog usmjerenja

- Apstrakcija
  - olakšava savladavanje složenih problema
  - objekt -> nešto u realnom svijetu
  - razred -> Objekti (instancije)
  - superrazred -> Podrazred
  - operacija -> Metode
  - atributi i pridruživanje (asocijacije) -> Varijable instanci
- Enkapsulacija
  - detalji mogu biti skriveni u razredima.
  - potiče skrivanje informacija (*engl. information hiding*):
    - Programeri ne moraju znati sve detalje razreda.
- Modularnost
  - program se može oblikovati samo iz razreda (bez globalnih varijabli?).
- Hijerarhija
  - elementi istog hijerarhijskog nivoa moraju biti na istom nivou apstrakcije

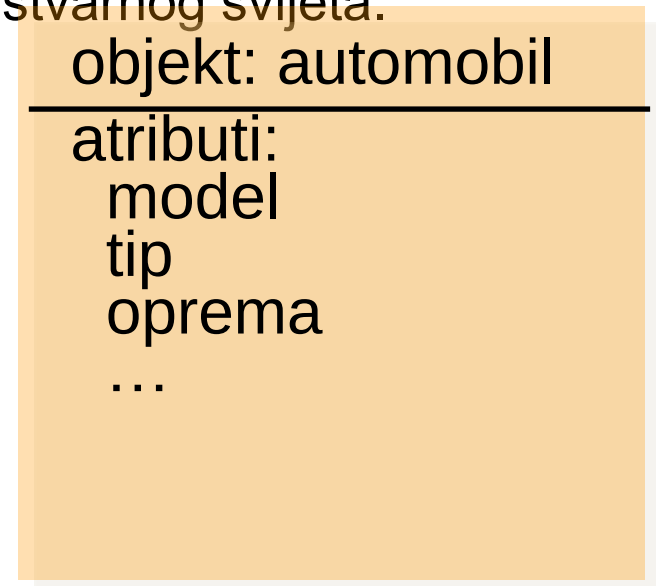


# Objekt

- Objekti su rezultat instanciranja razreda.
  - instanciranje je proces uzimanja predloška (nacrta) i definiranja svih pridruženih atributa i ponašanja
  - pri tome se rezervira memorijski prostor za smještaj atributa i pridruženih metoda
  - stvaranje objekta: operator new + konstruktor razreda:
    - `new Ball();`
  - pri instanciranju vraća se referenca na objekt
    - `Ball b = new Ball();`
- Dio strukturiranih podataka programa u radu.

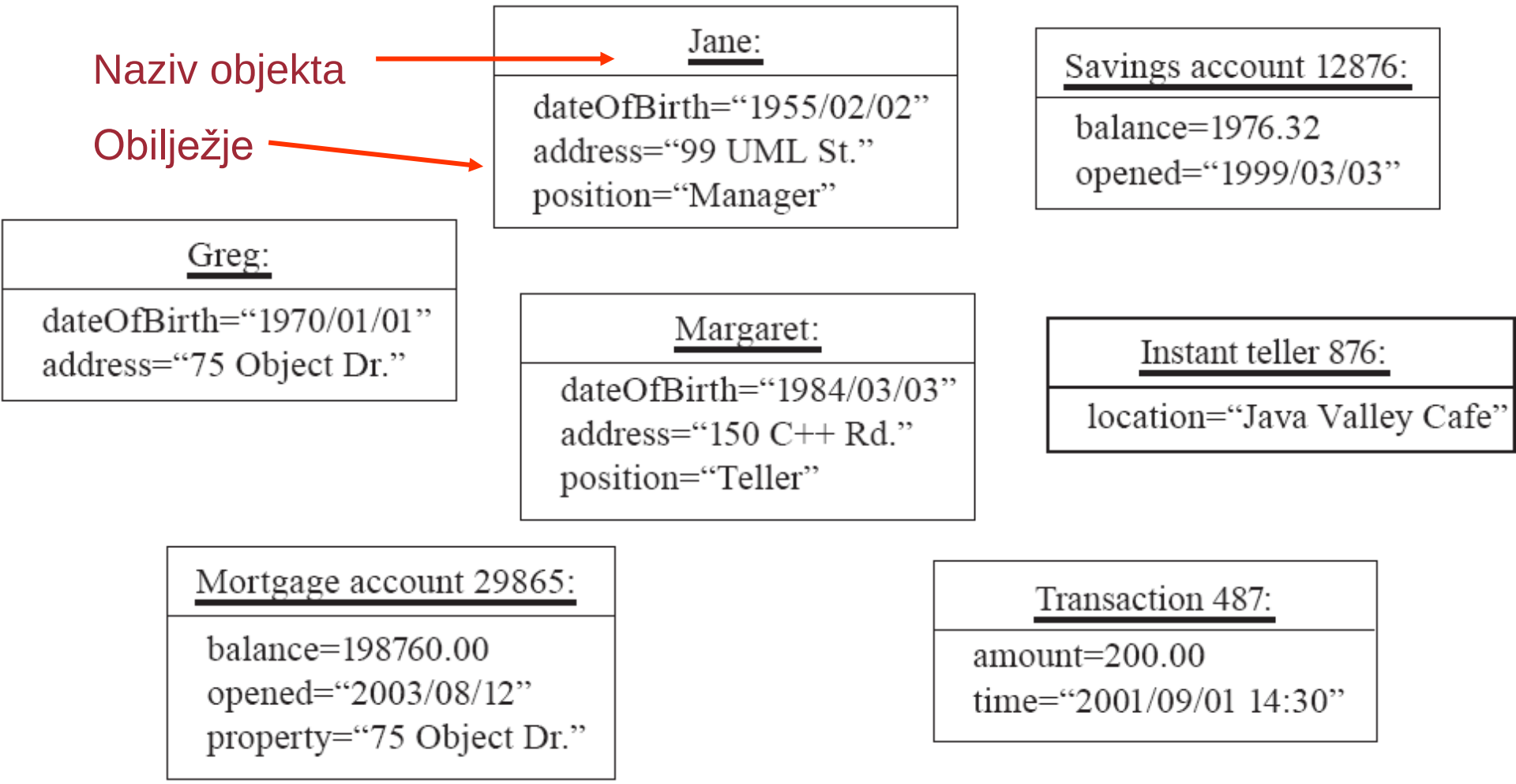
# Svojstva objekta

- Može predstavljati bilo što iz stvarnog svijeta čemu se mogu pridružiti:
  - ***obilježja*** (*engl. properties*) koja karakteriziraju objekt.
    - opisuju trenutno stanje objekta.
  - ***ponašanje*** (*engl. behavior*)
    - Kako objekt reagira (što može rezultirati u promjeni stanja).
    - Može simulirati ponašanje objekata iz stvarnog svijeta.
- Svojstva objekta:
  - stanje – *engl. state*
    - obilježja, atributi
  - ponašanje – *engl. behavior*
    - implementiraju metode
  - jedinstvena identifikacija



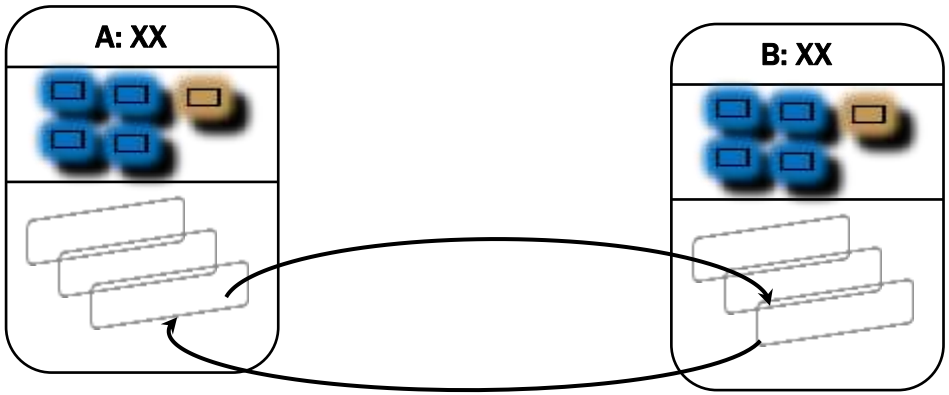
# Primjer: Objekti u bankovnom sustavu

- Analiza sustava temeljenog na objektno usmjerenoj paradigmi neovisna o programskom kodu i problemima smještaja u memoriji ili disku.



# Interakcija objekata

- Razmjenom poruka (*engl. Message passing*)
- Npr.:
  - objekt A želi da objekt B obavi neku operaciju traži od objekta B može to učiniti jedino slanjem poruke

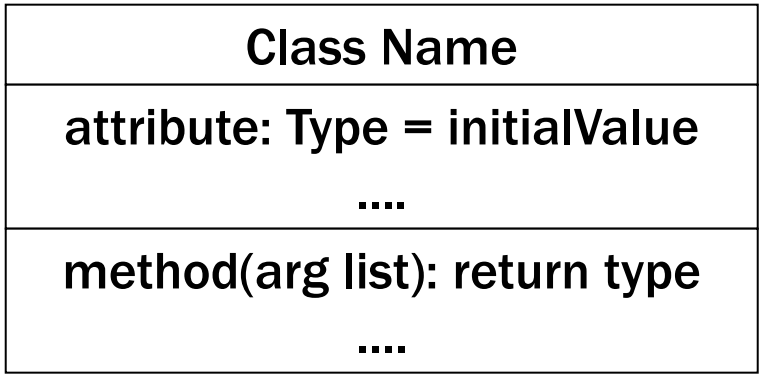


# Razred

- Razred/Klasa – *engl. Class*
- Objektno usmjereno razmišljanje započinje definiranjem razreda (opći opis, predložak, nacrt)
- Jedinica apstrakcije u objektno usmjerenoj paradigmi.
  - to je jedan definiran **tip podataka**.
- Razredi predstavljaju slične objekte.
  - objekti su ***instance*** razreda.
- Vrsta programskog modula koji sadrži:
  - opis strukture instanca, tj. ***obilježja*** (*engl. properties*)
    - podatci koji implementiraju obilježja
  - sadrže ***metode*** (procedure) koje ***implementiraju ponašanje*** objekata.
    - npr. procedure, funkcije za promjenu obilježja

# Razred

- Predložak za instanciranje objekata
- Nivo detalja u specificiranju razreda ovisi o stanju razvojnog procesa
- Razred specificira naziv, atribute stanja i pridružene metode



# Odnos razreda i instance

- Nešto je razred ako može imati instance.
- Nešto je instanca ako je jasno da je to jedan član skupa definiranog kao razred.
  - *Film*
    - ***Razred,*** instance su individualni filmovi.
  - *Distribucijski medij na kojem je film*
    - *npr. Digital Cinema Package):*
    - ***Razred***; instance su fizički mediji.
  - *Medij sa serijskim brojem W19876*
    - ***Instanca*** razreda
  - *Science Fiction*
    - ***Instanca*** razreda
  - *Science Fiction Film*
    - ***Razred,*** instanca je npr.: 'Star Wars'
  - *Prikazivanje filma 'Star Wars' u Cinestaru u 19:00***:**
    - ***Instanca*** razreda **PrikazivanjeFilma**

# Imenovanje razreda

- Koristi velika slova
  - bankAccount a ne bankAccount
- Koristi imenice u ***singularu***.
- Koristi ispravnu razinu generalizacije
  - npr. Ne **Student_FERa** nego općenitiji pojam, npr. **Student**
- Budi siguran da ime ima samo jedno značenje.
  - npr. ‘Bus’ ima više značenja.

# Primjer razreda

- Primjer razreda **`Circle`** u Javi:

```
Public class Circle{
//podaci koji implementiraju obiljezje
Public double x,y;  // koordinate središta
Public double r;    // radius
//metode (procedure) koje implementiraju ponasanje
Public double opseg(){return 2 * 3.14 *r};
Public double povrsina(){return 3.14 * r *r}
}
```

# Razlika instanca - objekt

- Nema razlike, odnosi se na istu jedinku (entitet)
- Razlika je nastala u korištenju prirodnog jezika.
- Npr. kćer – djevojka

  “Imala je sedam kćeri” (ne djevojaka).

  “Vidio sam prekrasnu djevojku” (ne kćer).
- Implementacijske razlike:
- Objekt:
  - memorija koja sadrži informacije o objektu
- Instanca:
  - referenca na objekt (pokazuje na početnu adresu na kojoj je objekt pohranjen)
  - dvije instance mogu pokazivati na isti objekt
  - životni vijek instance i objekta nije povezan. kada su sve instance koje pokazuju na objekt obrisane briše se i objekt.

# Varijable instanci

- *engl. Instance variables*
- Definirane unutar razreda
- Varijabla je mjesto (u instanciji) gdje se smještaju podaci (*engl. place-holder, slot, field*).
- Varijable definirane unutar razreda odgovaraju podacima (različitim vrijednosti) koji se nalaze u svakoj instanciji toga razreda.
- Primjer varijable instanci u razredu `Circle : public double r;`
- Postoje dvije skupine varijabli instanci:
  - atributi (obilježja objekta)
    - to su jednostavni podaci kao npr.:
      - name, dateOfBirth
  - asocijacije (pridruživanje) između instanci različitih razreda.
    - to su odnosi (*engl. relationships*) prema drugim instancama drugih razreda. Npr.:
      - `supervisor`, (odnos prema instancijama razreda `Manager`.
      - `coursesTaken` (odnos prema instancijama razreda `Course`).

# Varijable instanci

- Ako neki razred ima definiranu varijablu instanci `var`, tada sve instance toga razreda imaju mjesto (*engl. field, slot*) s nazivom `var`.
- Stvarni podaci smješteni u varijablu `var` razlikuju se od objekta do objekta.
- Npr.:
  - **razred:** `Employee`
  - **varijabla:** `supervisor`
  - postoje različiti supervizori u svakoj instanci razreda `Employee`

# Variable i objekti

- Varijable i objekti su *zasebni i različiti koncepti*
  - uobičajena zabuna
- Tip varijable:
- Određuje koje razrede objekata može sadržavati.
  - u Javi postoje dva tipa:
    - *Primitive* (sadrži jednu vrijednost, nije objekt, evaluira se u vrijednost koju sadrži.
    - *Reference* (ili *object*) tip (evaluira se u adresu objekta). Slično pokazivaču ali u širem kontekstu.
    - U različitim trenucima može se odnositi na različite objekte.
- Jedan objekt može u isto vrijeme referencirati više različitih varijabli.
  - referenciranje je općenitiji pojam od “sadrži”. U varijabli tipa “***reference***” se smješta adresa objekta na koji se referencira, a ne sam objekt.
  - *objekt je dostupan preko varijable koja ga referencira!!*

# Primjer: Variable i Objekti

Neka postoji razred `Ball`.

Deklariramo varijablu `b1` koja referencira objekt iz razreda `Ball`:

```
Ball b1;     // varijabla b1 je "reference" tipa
      // Ball. U nju se mogu "smjestiti" samo
      // objekti razreda Ball. Za sada još ništa
      // nije u njoj "smješteno" (referencirano).
```

Sada želimo kreirati objekt iz razreda `Ball` i želimo da varijabla `b1` referencira baš taj objekt:

```
Ball b1 = new Ball();
// rutina Ball() konstruira objekt iz razreda
// Ball. Zove se "konstruktor"
```

Taj isti objekt može biti referenciran i od druge varijable:

```
Ball b2 = b1;
```

Obje varijable referenciraju isti objekt iz razreda `Ball`.

Primjer pokazuje da pojam "smještanje podataka u varijablu" treba shvatiti općenitije i apstraktnije. Ovdje "smještanje" = "referenciranje na".

# Varijable razreda

- *engl. class variables*
- Varijable razreda se identificiraju ključnom riječi (modifikatorom) ***static***.
- Varijabla razreda može sadržavati vrijednost.
- Tu vrijednost *dijele* sve instance toga razreda
  - ne postoji više kopija te varijable kao za varijable instanci, već samo jedna pridružena razredu.
  - (npr. u C++ “static data member”).
  - ako jedna instanca upiše vrijednost u varijablu razreda, sve instance toga razreda vide izmijenjenu vrijednost.
  - uvijek postoji sam o jedna vrijednost te varijable
  - varijable razreda su korisne za:
    - zadane početne (*engl. default*) *ili konstantne vrijednosti (npr. PI*).
    - Lookup tablice i slične strukture.

# Primjer: Varijable razreda

- Primjer varijable razreda:

```
Public class Circle{
Public static int num_circles; // varijabla razreda
Public double x,y;              // varijable
Public double r;                // instanc
Public double opseg(){return 2 * 3.14 *r};
Public double povrsina(){return 3.14 * r *r} }
```

- Kako se pristupa varijabli razreda ?
- - preko naziva razreda (a ne preko objekta – t.j. varijable koja referencira taj objekt):

```
System.out.ptintln("No of circles:" +
                        Circle.num_circles);
```

# Metode razreda

- *engl. class methods, static methods*
- Metode definirane unutar nekog razreda i deklarirane ključnom riječi `static`
  - analogno varijablama razreda

- Primjer poziva takve metode:

```
double distance = Math.sqrt(dx*dx + dy*dy);
```

  - metoda sqrt() definirana je unutar razreda Math.
  - metoda se poziva preko naziva razreda a ne preko objekta (t.j. varijable koja referencira objekt).

# Lokalne varijable

- Slično kao što objekti pohranjuju svoja stanja u slotove, procedure (metode) često privremeno pohranjuju stanje u lokalne varijable.
- Sintaksa deklaracije lokalne varijable je slična
  - npr. `int count = 0;`
- Ne postoji poseban ključna riječ za određivanje lokalnih varijabli.
- Doseg varijable je unutar zagrada procedure (metode).
  - kreiraju se pri izvođenju metode i nestaju s njenim završetkom.
- Lokalna varijabla je vidljiva samo u metodi gdje je deklarirana.
  - nije joj moguće pristupiti iz ostatka razreda.

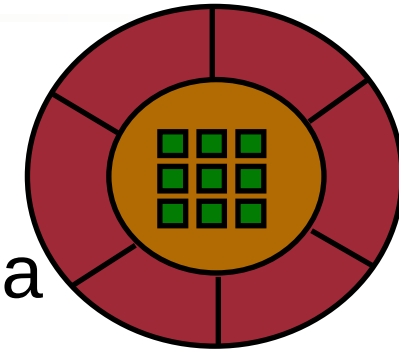
# Parametri

- Varijable kao parametri metode
- Npr. neka `main` metoda ima oblik:

```
Public static void main(String[] args)
```

  - `args` varijabla je parametar procedure (metode) `main`.
- Parametri se uvijek klasificiraju kao "varijable".
- To se odnosi i na druge konstrukcije koje prihvaćaju parametre (npr. *engl. constructors i exception handlers*).
- Broj parametara metoda u objektno usmjerenoj paradigmi treba biti što manji.

# Metoda



- Metoda (način izvođenja neke operacije)
  = procedura, funkcija, rutina
  - proceduralna apstrakcija koja se koristi za implementaciju ponašanja razreda.
- Oblikovana za rad na jednom ili više atributa razreda
  - jedna operacija može biti implementirana s više metoda.
- Metoda se poziva razmjenom poruka (*engl. message passing*)
- Nekoliko različitih razreda može imati metodu istog naziva.
  - sve te metode implementiraju istu apstraktnu operaciju na način kako odgovara pojedinom razredu.
    - Npr.: Operacija Izračun_površine u pravokutniku je različito implementirana nego za krug (iako je ime metode isto).

# Metode - primjer

Neka su u razredu Ball deklarirane i definirane dvije metode:

```
SetSpeed();  // postavlja brzinu na int vrijednost
GetSpeed();  // očitava brzinu u int
```

Svi stvoreni objekti iz Ball imaju te dvije metode.

Stvaranje objekta referenciranog varijablama b1 i b2:

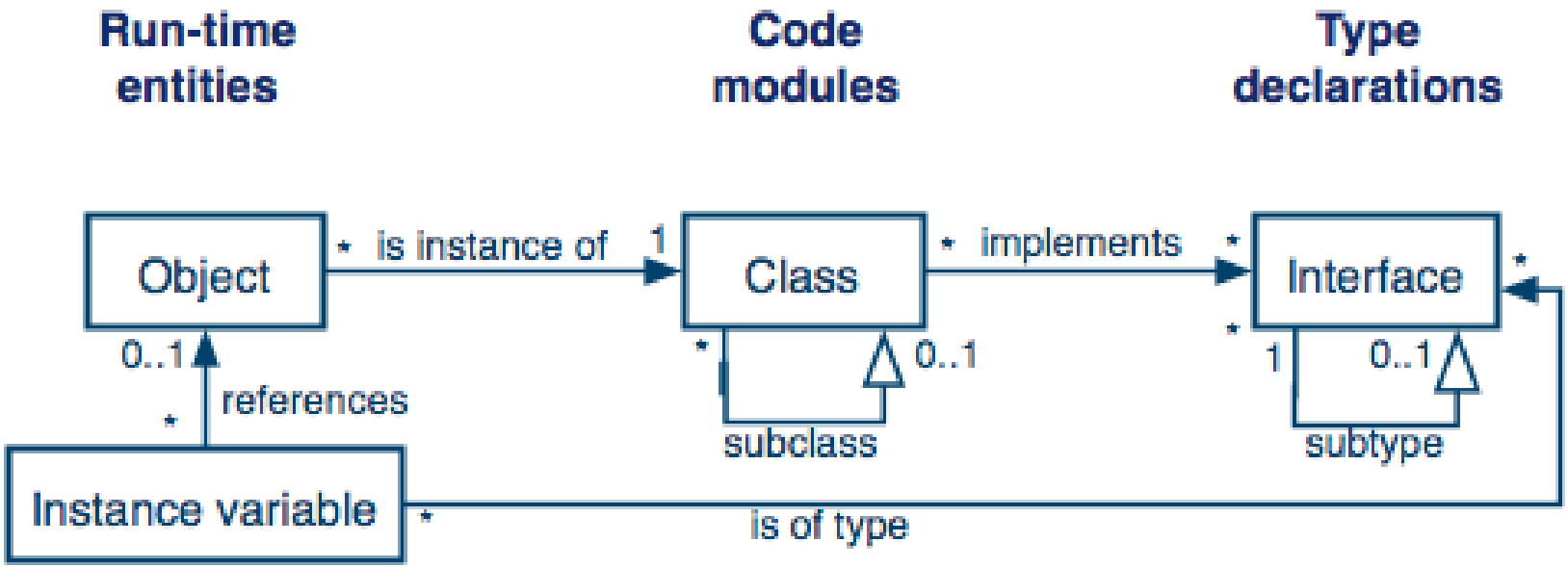
```
Ball b1 = new Ball();
Ball b2 = b1;

B1.Setspeed(50)     // u objektu kojega referencira
                    // b1 postavlja brzinu na 50
B2.setSpeed(100)    // u objektu kojega referencira
                    // b2 postavlja brzinu na 100
```

Jasno da se radi o istom objektu i vrijedi zadnje postavljanje brzine, te:

```
Int current_speed = b1.getSpeed();
```

Vraća vrijednost 100 i stavlja u neku varijablu “primitivnog” tipa int.

# Pregled odnosa



# Operacija

- Proceduralna apstrakcija više razine nego metoda.
- Operacija specificira tip ponašanja.
- Jedna operacija može biti implementirana s više metoda.
- Neovisna je o kodu koji implementira njeno ponašanje.
  - npr. Izračun_površine (sasvim općenito).

# Vidljivost operacija

- U modeliranju objektno usmjerenog sustava (npr. u UML dijagramima) potrebno je jasno označiti vidljivost operacija koje se implementiraju metodama.
  - public, protected, private, ..
- Operacija obilježena ključnom riječi public dostupna je svima (javno).
- Operacija obilježena ključnom riječi protected dostupna je unutar hijerarhije razreda u kojem je deklarirana.
- Operacija obilježena ključnom riječi private dostupna je unutar razreda u kojem je deklarirana. U nekim programskim jezicima to se podrazumijeva (*engl. default*).
- Preporuka:
  - broj javnih metoda u nekom razredu trebao bi biti što manji.
  - mnogo javnih metoda sugerira da bi neke trebale biti privatne.

# Više metoda istog naziva

- Višestrukost metoda *engl. Overloading*
- U objektno usmjerenom programiranju dopušta se postojanje više metoda istog naziva, ali različitog broja, tipova i mjesta parametara.
- Pri pozivu, prevoditelj (*engl. compiler*) odabere onu metodu koja ima isti naziv, broj i tip parametara, te su parametri na istom mjestu kao i u pozivu metode.
- To se ne smije se miješati s konceptima nadjačavanja ili polimorfizma!!

# Primjer

- Neka postoji razred ***DataArtist*** koji može kaligrafski crtati različite tipove podataka (string, int, …). Umjesto različitih naziva metoda: **drawString, drawInteger, drawFloat, …** možemo uporabiti isti naziv **draw**, ali uz različite parametre:

```
public class DataArtist {
...
public void draw(String s) { ... }
public void draw(int i) { ... }
public void draw(double f) { ... }
public void draw(int i, double f) { ... } }
```

- Taj pristup treba koristiti rijetko jer programski kod čini manje čitkim.

# Nasljeđivanje

- *engl. Inheritance*
- Način za podržavanje principa ponovne uporabe objekata
- Razredi mogu nasljeđivati značajke drugih podrazreda
- Organizacija razreda

# Organizacija razreda u hijerarhije

- Superrazredi (*engl. Superclasses*) (u C++ “base class”)
  - sadrže značajke zajedničke jednom skupu razreda.
- **Hijerarhije nasljeđivanja** (*engl. Inheritance hierarchies*)
  - pokazuju odnos između superrazreda i podrazreda (*engl. superclasses i subclasses*).
  - oznaka trokuta pokazuje generalizaciju.

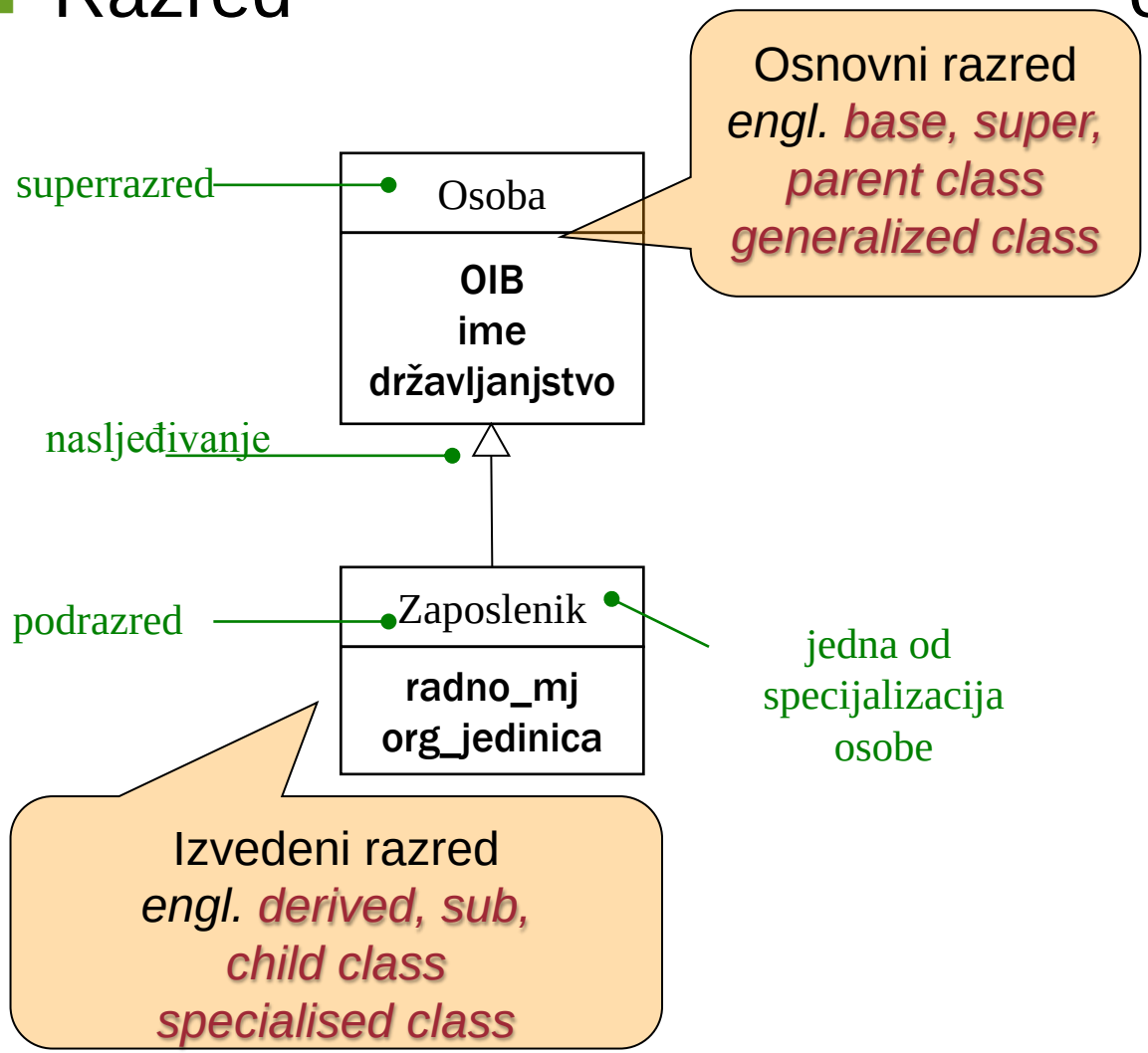
UML notacija

- Nasljeđivanje (*engl. Inheritance*)
  - svi podrazredi *implicitno* posjeduju značajki koje su definirane u superrazredu.

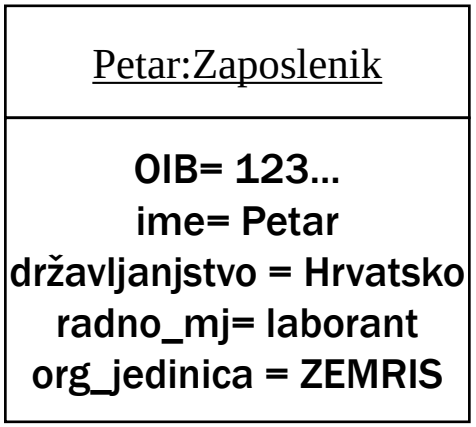
```
public class MortgageAccount extends Account
{
    // naredbe u tijelu dodane tijelu Account
}
```

# Primjer: hijerarhija nasljeđivanja

- Razred




objekt




objek tipa zaposlenik ima sve atribute osobe

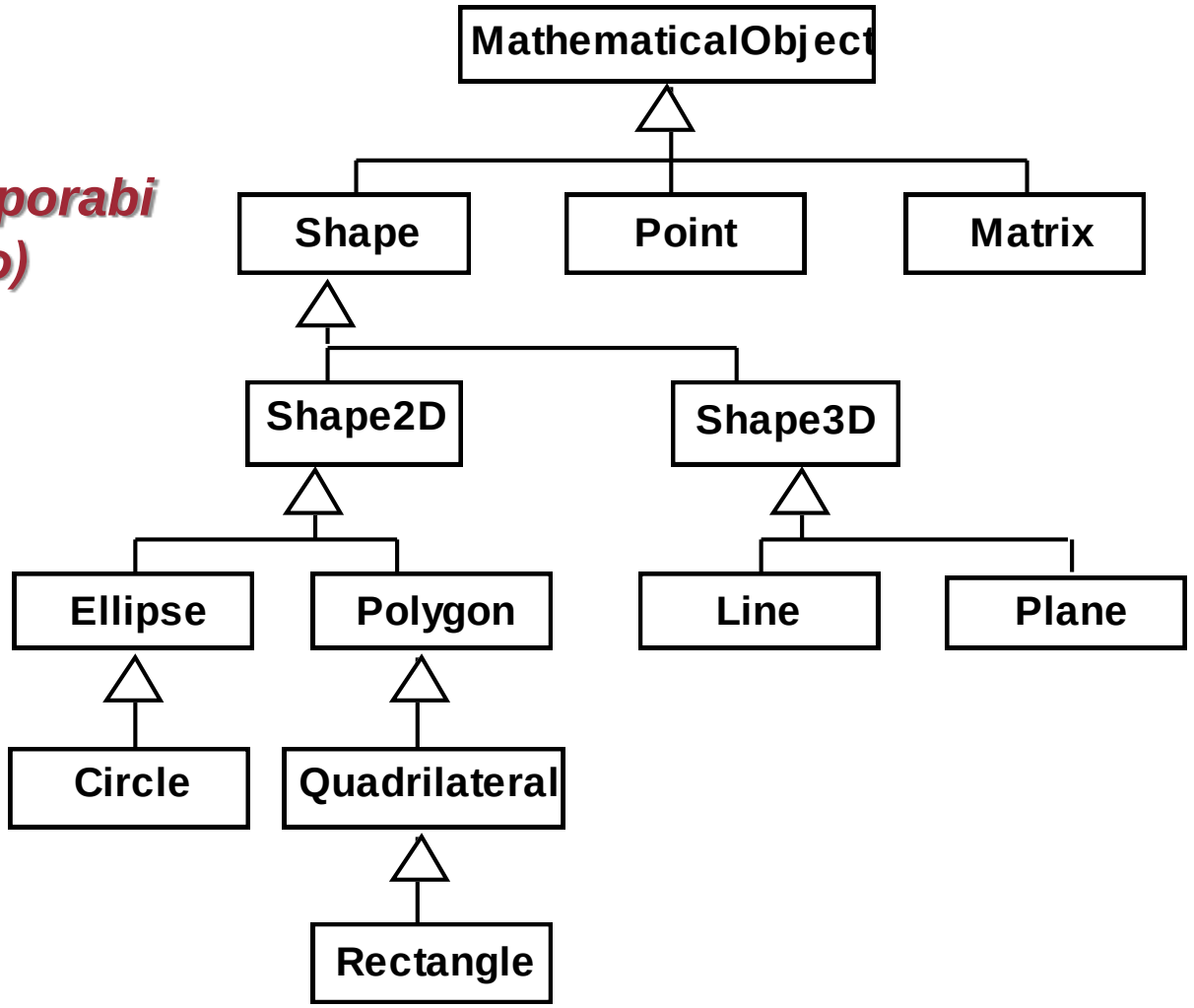
# Pravila nasljeđivanja: “Is-a” pravilo

- *engl. `is a’ Hierarchy - ISA*
- Uvijek provjeri da li generalizacija zadovoljava “is a” pravilo (pravilo “je”, odnos podskup → skup).
  - “checking account is an account”
  - “village is a municipality”
- Da li bi “Županija” bila podrazredom “Država” ?
  - ne, jer ne zadovoljava “is-a” pravilo:
  - “Županija je država” ne vrijedi !
- Pri nasljeđivanju je potrebno provjeriti da li sve naslijeđene značajke imaju smisla.

# Primjer

- Moguća hijerarhija nasljeđivanja matematičkih objekata

***(U provjeri uporabi "is a" pravilo)***

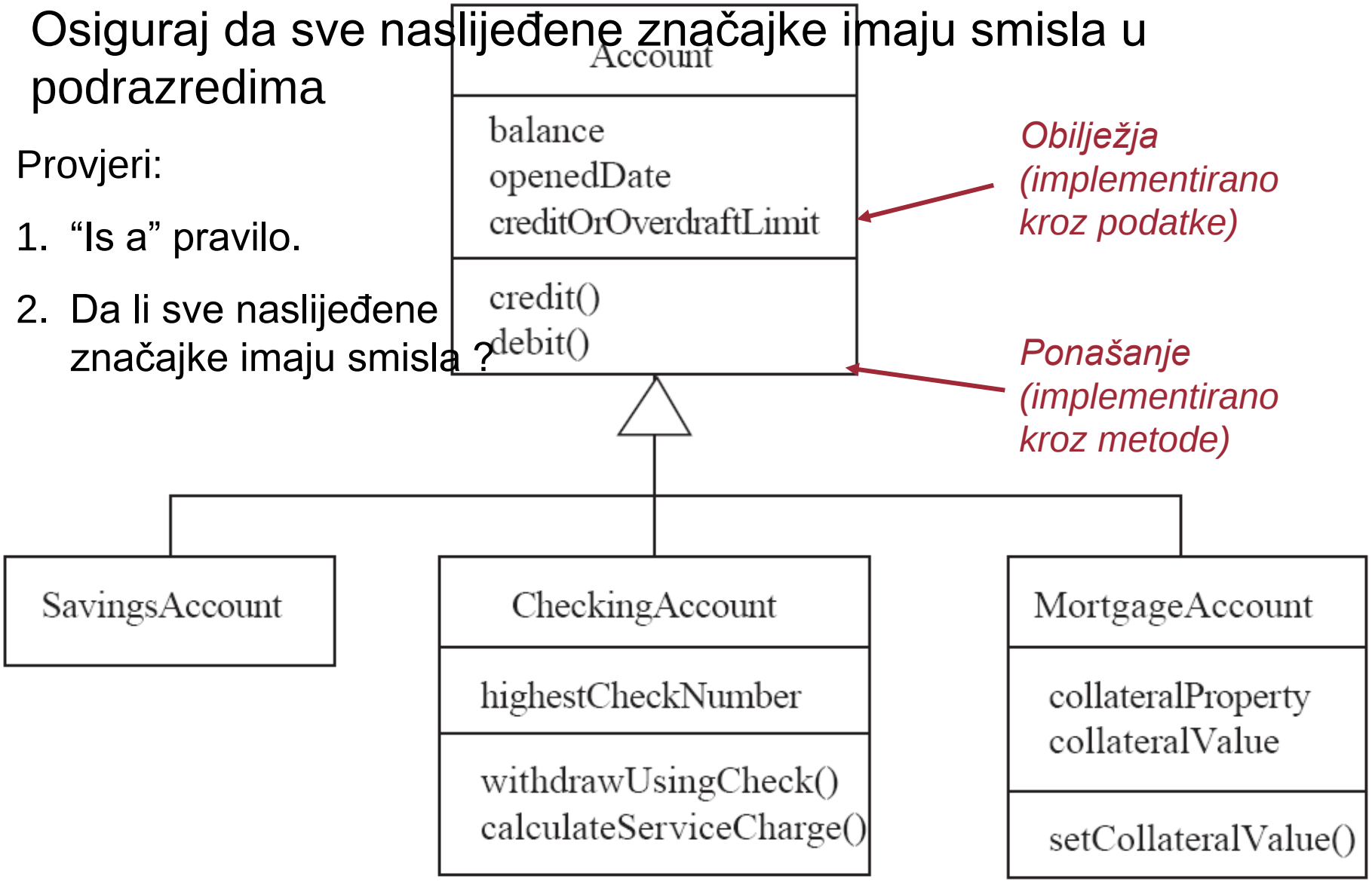


# Primjer:

- Osiguraj da sve naslijeđene značajke imaju smisla u podrazredima

Provjeri:

1. "Is a" pravilo.
2. Da li sve naslijeđene značajke imaju smisla ?



# Provjera ispravnosti nasljeđivanja

- Liskov princip zamjene (supstitucije)
  - *engl. Liskov substitution principle - LSP*
- Ako postoji varijabla čiji tip je superrazred, program se mora korektno izvoditi ako se u varijablu pohrani instancija tog superrazreda ili instancija bilo kojeg podrazreda.
  - podrazredi nasljeđuju sve od superrazreda
- Barbara Liskov, prof. na Massachusetts Institute of Technology
  - 2008 ACM Turingova nagrada “for her work in the design of programming languages and software methodology that led to the development of object-oriented programming”.

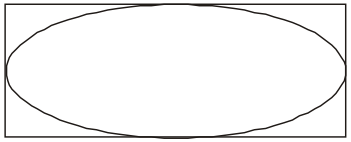
# Svojstva nasljeđivanja

- Prednosti
  - mehanizam apstrakcije pogodan za organizaciju
  - mehanizam ponovne uporabe u oblikovanju i implementaciji
  - organizacija znanja o domeni i sustavu
- Nedostaci
  - razredi nisu samodostatni i ne mogu se razumjeti bez poznavanja superrazreda
  - nasljeđivanja uočena u fazi analize mogu dati neefikasna rješenja
    - potrebno zasebno promatrati

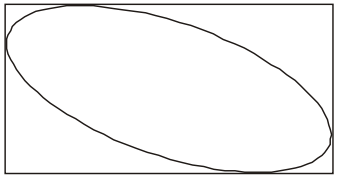
# Primjer: operacije na grafičkim objektima

- operacije ništa ne govore o implementaciji

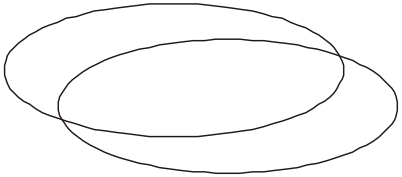
Original objects
(showing bounding rectangle)



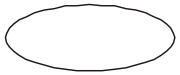
Rotated objects
(showing bounding rectangle)



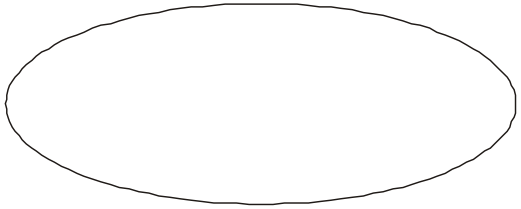
Translated objects
(showing original)



Scaled objects
(50%)



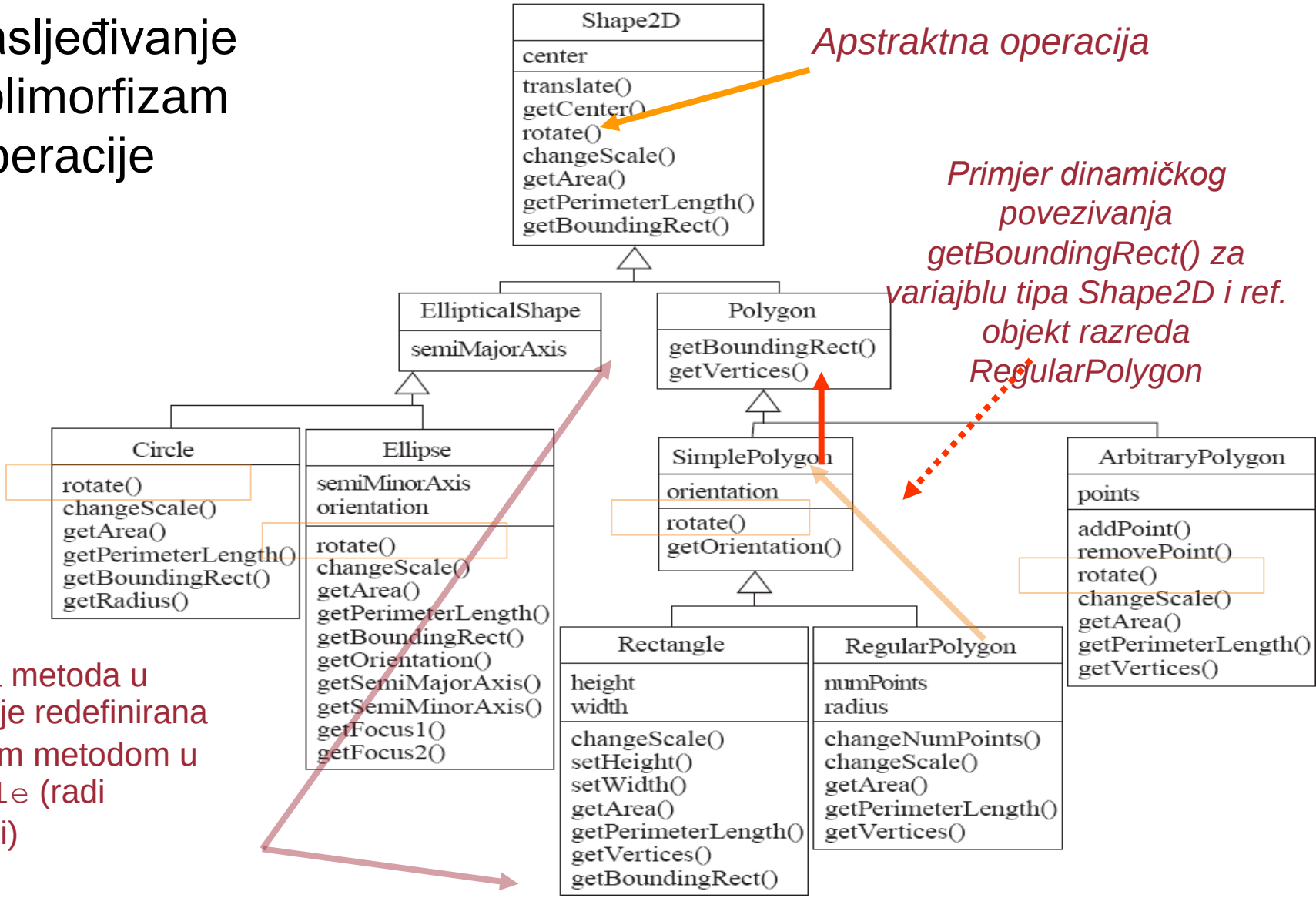
Scaled objects
(150%)



Izvor: T.C. Lethbridge, R. Laganière: Object-Oriented Software Engineering: Practical Software Development using UML and Java

# Primjer ...

- Nasljeđivanje
- Polimorfizam
- Operacije




Konkretna metoda u `Polygon` je redefinirana konkretnom metodom u `Rectangle` (radi efikasnosti)

# Apstraktni razredi i metode

- Pojedina operacija treba biti *deklarirana* da postoji u najvišem hijerarhijskom razredu gdje ima smisla.
- Na toj razini operacija može biti *apstraktna* (bez implementacije)
  - npr. `rotate` u `Shape2D` (vidi sliku hijerarhije matematičkih objekata, gdje će podrazredi će imati svoje specifične `rotate`).
- Ako neka operacija nema implementacije, cijeli razred je "***apstraktan***"
  - ***apstraktni razred ne može se kreirati instance.***
  - suprotno, ako postoje sve implementacije (naslijeđene ili definirane) , razred je "***konkretan***".
- Ključne riječi za apstraktnu operaciju su `abstract` (Java) ili `virtual` (C++).
- Ako superrazred ima *apstraktnu* operaciju, tada na nekoj nižoj razini hijerarhije ***mora postojati konkretna metoda*** za tu operaciju.
  - krajnji razredi u hijerarhiji ("lišće") moraju **implementirati ili naslijediti** konkretne metode **za sve operacije**.
  - ti razredi moraju biti konkretni.

# Polimorfizam

- Moć poprimanja više oblika, *engl. Polymorphism*
- Svojstvo objektno usmjerenog programa da se jedna apstraktna operacija može izvesti na različite načine u različitim razredima.
- Polimorfizam zahtjeva da postoji više metoda istog naziva.
  - izbor koja metoda će se izvesti ovisi o razredu objekta koji se nalazi u varijabli.
  - polimorfizam smanjuje potrebu da programeri kodiraju velik broj `if-else` ili `switch` naredbe

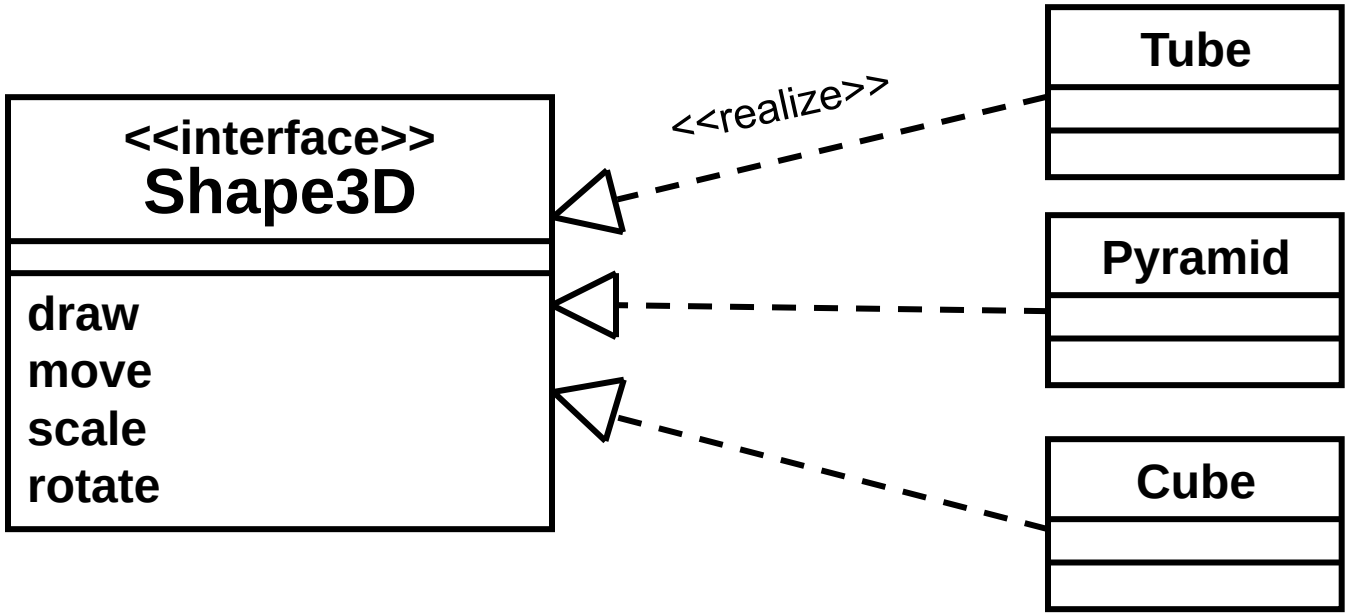
# Polimorfizam - primjer

```
class Animal {
  abstract string makeNoise (); } // apstraktna operacija

class Cat extends Animal {                    // podrazred Cat
  string makeNoise() {return "Meow";} } // konkretna metoda

class Dog extends Animal {          // podrazred Dog
   string makeNoise() {return "Bark";} }// konkretna metoda

main () {
  Animal animal = zoo.getAnimal(); // animal je tipa Animal
  Console.WriteLine (animal.makeNoise ()); }
```

- Varijabla animal je tipa Animal i može referencirati objekte iz razreda Cat i iz razreda Dog (hijerarhija naslijeđenih razreda).
- Metoda zoo.getAnimal() "stavlja" Cat ili Dog objekt (jedan) u varijablu animal.
- Metoda animal.makeNoise() pozvana preko varijable animal generira ispis ovisno o objektu koji je referenciran varijablom animal.
- Jedan apstraktna operacija izvodi se na razne načine (ovisno o razredu objekta kojega referencira varijabla animal).

# Sučelje

- *engl. Interface*
- formalizira polimorfizam
- podržava “plug-and-play” koncept
- ne predstavlja apstraktni razred
  - ne pruža opis ponašanja metoda



# Nadjačavanje

- Redefiniranje, ne obaziranje, *engl. Overriding*
  - uvijek je vezano uz hijerarhiju i nasljeđivanje razreda.
- Metoda iako je definirana u superrazredu i može se naslijediti, redefinira se u podrazredu
  - podrazred sadrži inačicu metode
- To se koristi za:
  - restrikciju
    - Npr. `scale(x,y)` ne bi radila u `Circle` (`Circle` bi postao `Elipse`)
  - proširenje (ekstenziju)
    - Npr. `SavingsAccount` razred bi mogao zaračunavati neku dodatnu pristojbu.
  - optimizaciju
    - Npr. `getPerimeterLength` metoda u `Circle` je jednostavnija od one u `Ellipse`.
- Npr. u primjeru `Shape2D` hijerarhije, konkretna metoda u `Polygon` je nadjačana/redefinirana konkretnom metodom u `Rectangle` (radi efikasnosti).

# Odabir metode za izvođenje

- Poziva se neka metoda iz objekta (varijable koja “sadrži” objekt **b** iz nekog razreda; tip varijable je taj razred).

```
ResultVariable = b.methodName(argument);
```

- Koja će se metoda izvoditi donosi se temeljem slijedećeg algoritma:
  1. ako postoji **konkretna** metoda za operaciju u trenutnom razredu, ta se metoda izvodi.
  2. inače, metoda je **naslijeđena** i pogledaj da li postoji implementacija u neposrednom superrazredu.
     - Ako postoji izvedi ju.
  3. ponovi korak pod 2., provjeravajući sukcesivno u višim superrazredima dok ne nađeš konkretnu metodu, te je izvedi.
  4. ako metoda nije pronađena, postoji pogreška u programu.
     - Java i C++ program neće se moći kompilirati.

# Dinamičko povezivanje

- *engl. Dynamic binding*
- Pojavljuje se u slučaju kada se odluka o izboru konkretne metode donosi za vrijeme izvođenja programa (*engl. at run-time*).
- To je potrebno kada:
  - varijabla je deklarirana da je tipa superrazreda (tj. postoji hijerarhija podrazreda toga tipa varijable).
  - postoji više polimorfnih metoda koje se mogu izvesti u sklopu hijerarhije razreda određene superrazred tipom varijable.
  - npr.:

    neka postoji varijabla **`aShape`**, a njen tip je **`Shape2D`**.

    to znači da **`aShape`** može sadržavati objekt (instancu) iz bilo kojeg konkretnog razreda u hijerarhiji razreda **`Shape2D`**.

    pretraživanje i odabir konkretne metode započinje pregledom razreda čiji objekt se stvarno nalazi u varijabli **`aShape`**.

    ako metoda nije pronađena u tom razredu, pretražuje se sukcesivno hijerarhija razreda idući prema gore sve do superrazreda **`Shape2D`** koji je naveden kao tip varijable.

    ako metoda nije pronađena deklarira se pogreška.

# Primjer dinamičkog povezivanja

- `Shape2D` hijerarhija
  - neka postoji objekt iz razreda `RegularPolygon` u varijabli `aShape` koja je tipa `Shape2D`, i želi se izvesti metoda `getBoundingRect`.
    - Program prvo traži tu konkretnu metodu u razredu `RegularPolygon`, zatim u razredu `SimplePolygon`, te u razredu `Polygon` (gore po hijerarhiji) gdje ju i nalazi).
    - To je dinamičko povezivanje.
  - ako `aShape` (koja je tipa `Shape2D`) sadrži objekt iz razreda `Rectangle`, program odmah nalazi konkretnu metodu (`getBoundingRect` je konkretna metoda u tom razredu).
    - To je također dinamičko povezivanje, jer se unaprijed ne zna koji je razred objekta u varijabli.
  - ako pak neka varijabla `myRect` je tipa `Rectangle` (koji nema podrazreda), ta varijabla sadrži objekt iz razreda `Rectangle`, prevoditelj statički određuje metodu za izvođenje.
    - Dinamičko povezivanje odigrava se samo u slučaju kada je tip varijable superrazred (t.j. kada postoje podrazredi toga tipa varijable).

# Konstruktori i destruktori

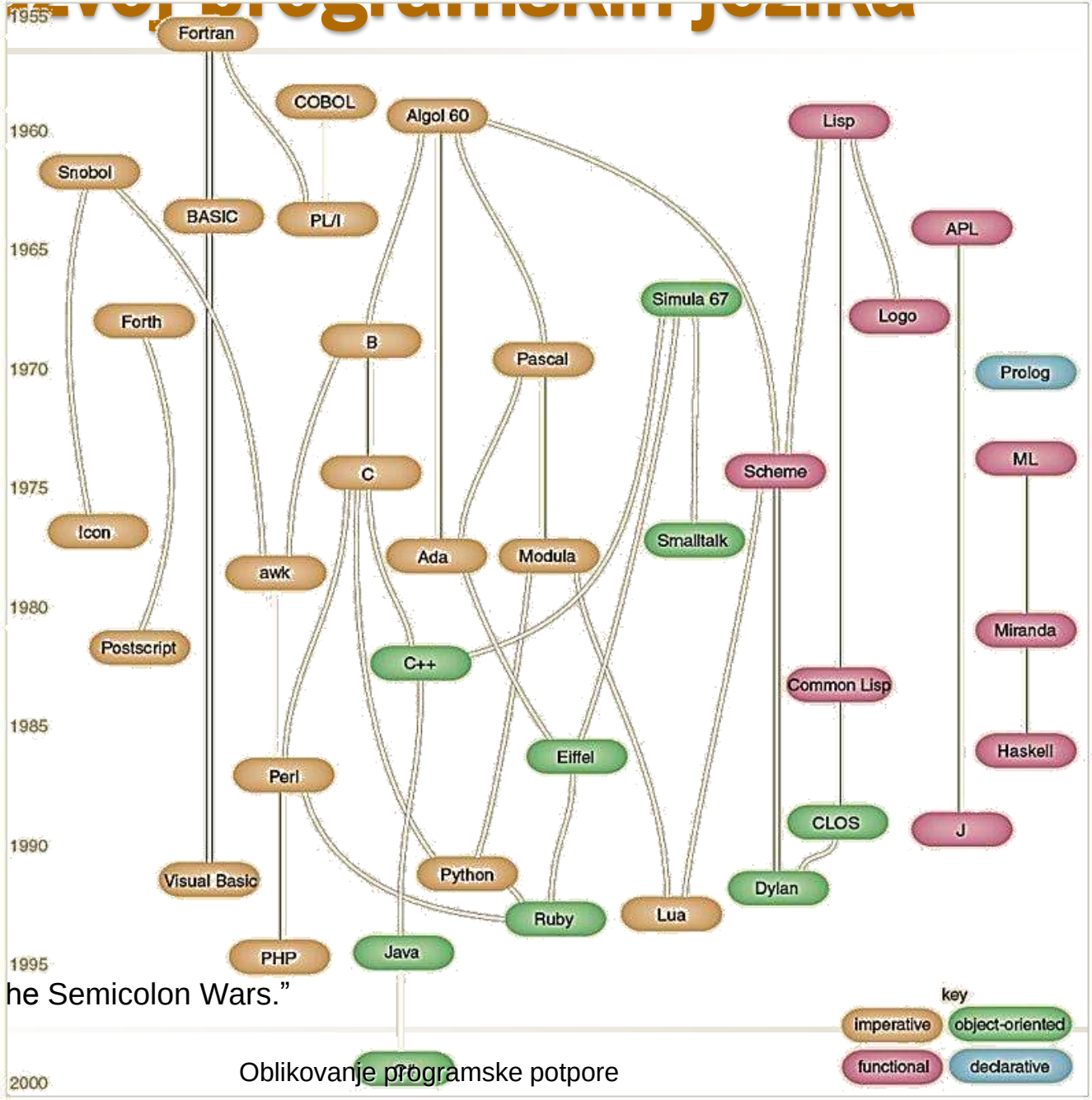
- Predstavljaju programske implementacijski detalje
  - nisu povezani s modeliranjem
- To su posebne metode koje se pozivaju kada se kreira ili definira objekt.
- Konstruktor inicijalizira objekt i njegove varijable.
- Naziv konstruktorske metode je isti kao i naziv njenog razreda.
- Ako konstruktor nije deklariran to će učiniti prevoditelj (*engl. compiler*), ali bez inicijalizacija varijabli.

```
class rectangle { // jednostavan razred
      int height;
      int width;
public:
      rectangle(void); // konstruktor
      ~rectangle(void); // destruktor
};
// inicijalizacija objekta
rectangle::rectangle(void) // konstruktor
{ height = 6; width = 6; }
```

# Dobra praksa objektnog usmjerenja

- Komentari
  - komentiraj sve što nije očigledno.
  - nemoj komentirati očigledno.
  - komentari čine 25-50% koda.
- Konzistentna organizacija elemente razreda
  - redom: varijable, konstruktore, javne metode, privatne metode.
- Izbjegavaj dupliciranje koda
  - ne “kloniraj” ako je to moguće (kloniranje može imati pogrešku u obje kopije, ispravljanje u jednoj ne ispravlja drugu).
- Pridržavaj se principa objektnog usmjerenja
  - npr.: ‘is a’ pravilo.
- Preferiraj nedostupnost informacija
  - npr. deklariranjem private.
- Ne miješaj kod korisničkog sučelja s ostalim kodom u programu
  - interakciju s korisnicima stavi u posebne razrede.
  - time je ostatak koda ponovo uporabljiv.

# Razvoj programskih jezika




- 2006, Brian Hayes, “The Semicolon Wars.”

# Objektno usmjereni programski jezici

- Prvi objektno usmjeren programski jezik bio je Simula-67.
  - oblikovan kako bi programeri pisali simulacijske programe.
- U ranim 1980-ima razvijen je Smalltalk u Xerox PARC.
  - nova sintaksa, velike knjižnice otvorenog koda spremnog za višestruku uporabu (*engl. reuse*), "bytecode", nezavisnost o platformi, skupljanje smeća (*engl. garbage collection*).
- Kasne 1980-te, razvijen je C++ (B. Stroustrup),
  - prepoznate su prednosti objektnog usmjerenja, ali također i činjenica da postoji ogromna skupina C programera.
- 1991, Sun Microsystems je započeo projekt koji je predložio jezik za programiranje potrošačkih (*engl. consumer*) pametnih naprava.
  - 1995., novi jezik je nazvan Java, i formalno predstavljen na konferenciji SunWorld '95.
- 2000., Microsoft predstavlja C# kao kompeticiju Javi.
  - prva specifikacija C# jezika dana je 2001.

# Svojstva OO

| Proceduralna | OO |
|---|---|
| Dekompozicija problema u funkcije | Dekompozicija u skupove objekata |
| Odvojeno modeliranje podataka i funkcija | Podaci i povezane opreracije na jednom mjestu |
| Velika međuovisnost komponenti – teškoće održavanja | Neovisnost komponenti |
| Komponente slabo odgovaraju stvarnom problemu – teško u slučaju rješavanja složenih problema | Blisko ljudskom rješavanju složenih problema |
| Često neprilagodljiv i linearan proces razvoja | Pogodno za iterativan i inkrementalan razvoj |

# Sažetak

- Uvođenje principa modularizacije, apstraktnih tipovi podataka smanjuje složenost razvoja programske podrške
- Osnovni koncepti objektno usmjerene arhitekture:
  - objekt
  - razred
  - nasljeđivanje
  - polimorfizam

# Diskusija

# Primjer:

```
Public abstract class Shape
{
  //...Class implementation

  public abstract void Draw(int x, int y)
  {
    // Na ovom mjestu ne smijete implementirati metodu
    // Kakav je to tip greške?
  }
}


Public abstract class Shape2D : Shape
{
  // Class implementation
  // Moramo li implementirati metodu Draw(int x, int y)?
}

Public class Cricle : Shape2D
{
  // obavezna implementacija Draw(int x, int y)
  public override void Draw(int x, int y)
  {
    //Kod ...
  }

}
```

# Primjer

```
Shape moj_Oblik;
   // deklaracija reference koja se odnosi na instance
     konkretne razrede

Circle moja_Kruznica = new Circle();
   // instanciranje novog oblika

Moj_oblik = moja_Kruznica ;
   // Da li je ovo ispravno?
```