Objektno orjentirano programiranje

Predavanje 2

Uvod u objektno orijentirani koncept

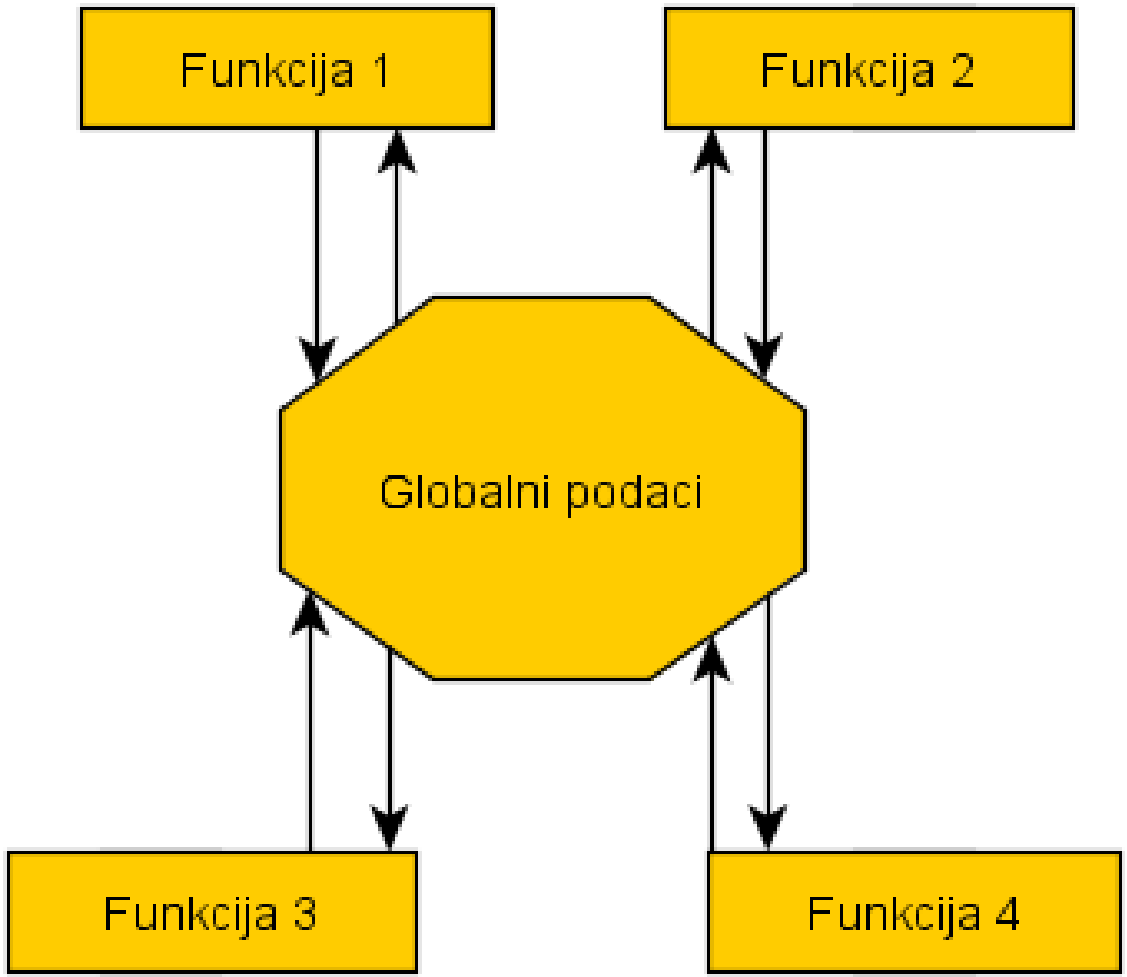
Uvod

- Objektno orijentirani razvoj softvera je prisutan od 1960ih
- Danas se koristi u industriji razvoja softvera
- Razvoj interneta i programiranja za Internet je pridonio još bržoj primjeni OO pristupa
- Kod koji nije OO može se prilagoditi korištenjem Object wrappera
- Objekti se koriste u Informacijskim sustavima (IS)
- Uspjeh Jave i .NET tehnologija još više promovira OO

Proceduralno programiranje i OO programiranje

- Što je objekt? (npr. Osoba)
- Ljudi već razmišljaju u okvirima objekata
- Objekt je definiran sa dvjema značajkama: atributi i ponašanje (attributes and behavior)
- Osoba – boja očiju, visina, dob …
    - hodanje, pričanje disanje
- Objekt sadrži **i** atribute **i** ponašanje (osnovna razlika između OO i ostalih pristupa)

- Prelazak sa ne-OO sustava je bio polagan jer su već postojali sustavi koji su funkcionirali
- Proceduralno programiranje



Objekti

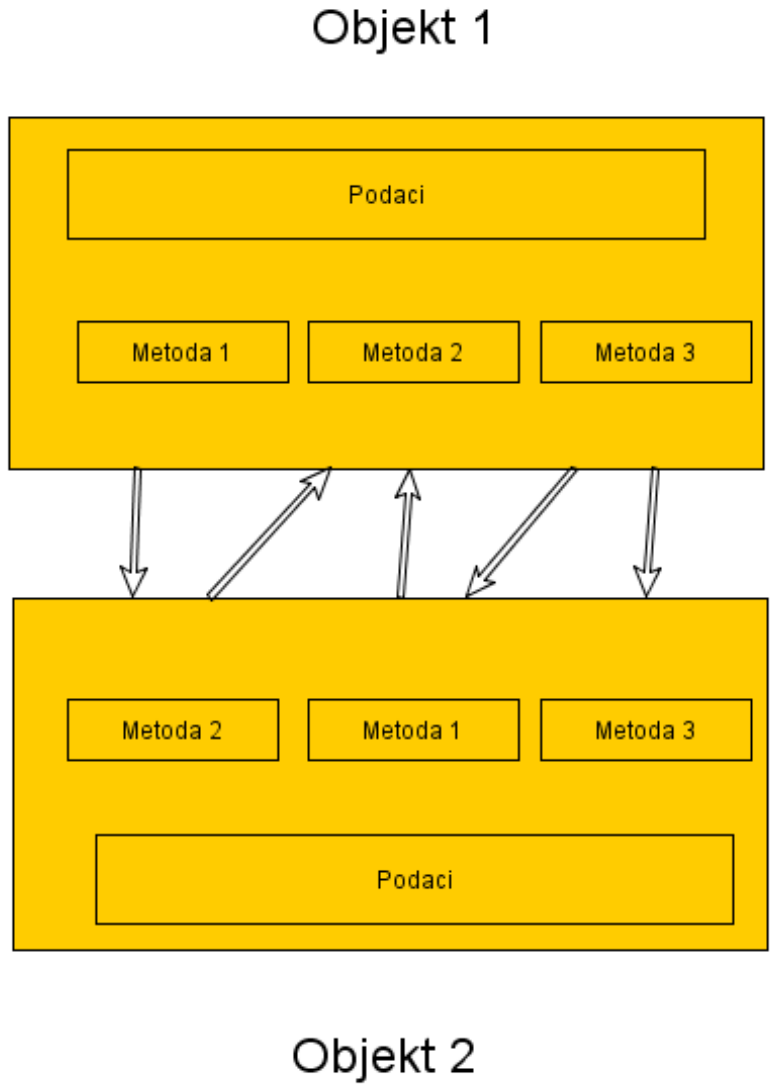
- Kod ispravnog dizajna možemo reći da nema pravih „globalnih” podataka
- Na ovaj način se čuva integritet podataka
- Objekti su više od struktura i primitivnih tipova podataka – teže zatvaranju ili skrivanju implementacijskih detalja
- U OO terminologiji podaci se nazivaju atributima, a ponašanja metodama
- Ograničavanje pristupa pojedinim atributima ili metodama zove se skrivanje podataka (data hiding)

Objekti

- Uklapanjem atributa i metoda u isti entitet možemo kontrolirati vanjski utjecaj na unutrašnje stanje objekta (enkapsulacija)
- Npr. dozvoljamo samo određeni raspon vrijednosti određenih atributa i pristup njima se ograničava pristupnicima
- Napomena – loš dizajn OO klasa može omogućiti pristup osjetljivim podacima
- Pratiti dobre prakse programiranja u OO pristupu

Komunikacija među objektima

- Objekti međusobno šalju poruke (međusobno pozivaju metode)



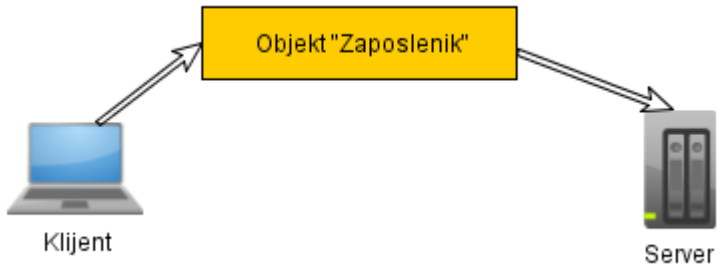
Pomak od proceduralnog prema objektno orijentiranom programiranju

- Proceduralno programiranje razdvaja podatke i operacije koje manipuliraju podacima
- Npr. slanje podataka kroz mrežu – šalju se samo bitni podaci uz uvjet da program na drugoj strani mreže zna što treba očekivati

- Prije samog slanja podataka potrebno je imati tzv. hadshake za inicijalizaciju

Pomak od proceduralnog prema objektno orijentiranom programiranju

- Osnovna razlika kod OO pristupa je da su podaci i operacije koje manipuliraju podacima (kod) enkapsulirani u objektu
- Kada se objekt šalje kroz mrežu, onda se šalje u cjelini –i podaci i ponašanje



- Dobar primjer ovog koncepta je web objekt, npr. Java applet. Browser ne zna što će objekt raditi jer unaprijed nema njegov kod. Prilikom učitavanja objekt dobivaju se i podaci i ponašanje objekta

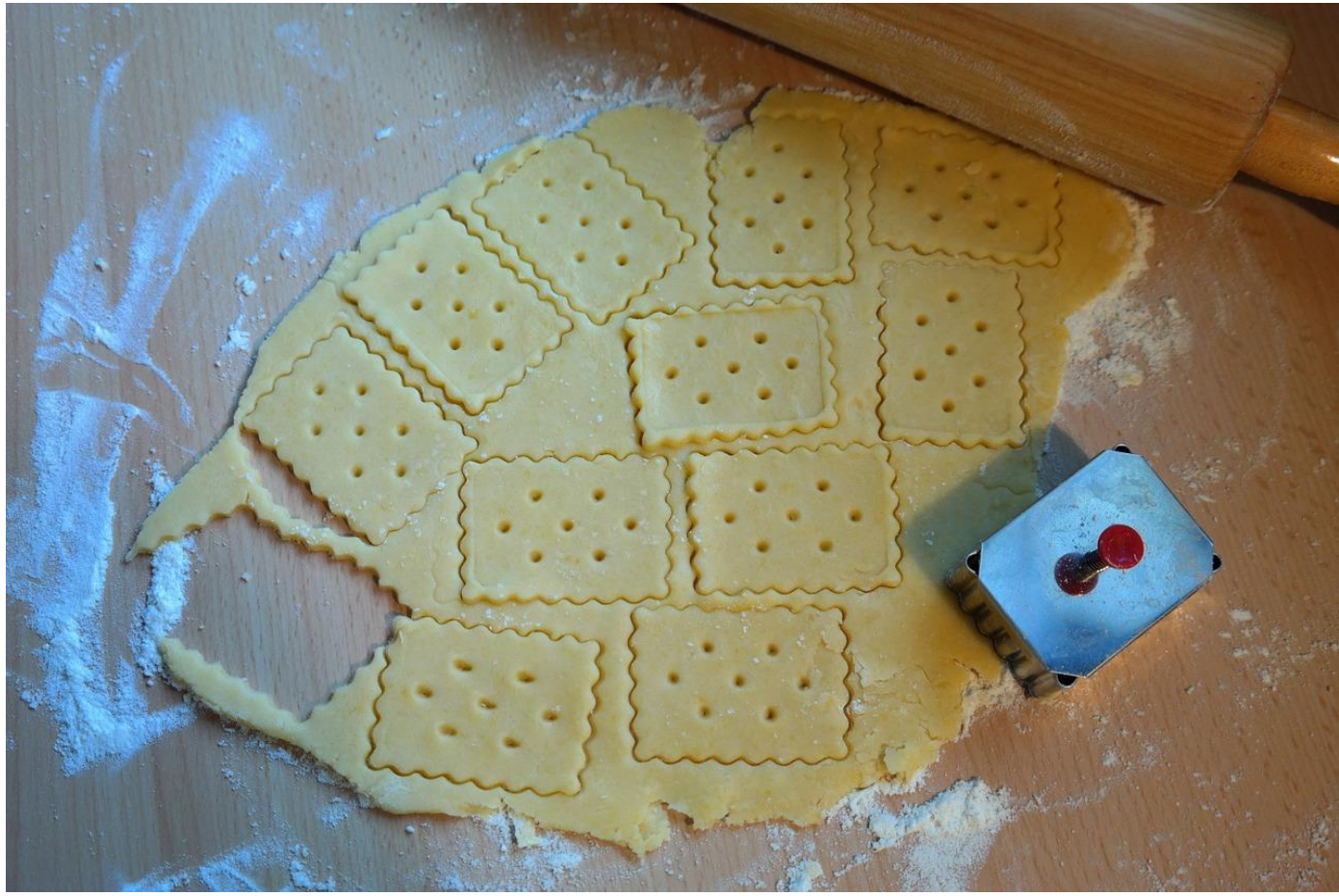
Što su objekti?

- Objekti su građevni blokovi OO programa
- Program je skup – kolekcija objekata
- Podaci objekta definiranju njegovo unutarnje stanje
- Pristupnici i mutatori (getters and setters)
- Primjer – klasa Osoba
- atributi ime i prezime
- getName()
- setName (String n) {name = n)
- Svrha – kontrolirani pristup atributu

- Koncept pristupnika i mutatora koristi se za skrivanje podataka. Neki atributi se ne bi smjeli direktno manipulirati izvana (putem drugih objekata)
- Poželjno je samo prikazivati sučelje (interface) prema metodama, a ne i implementaciju
- Ovo je sve što korisnik treba znati da bi učinkovito koristio metodu
  - ime metode
  - argumenti koji se šalju metodi
  - povratni tip metode
  - svrhu metode (obično ukratko opisana u imenu)
- Napomene – nije nužno da postoji stvarna (fizička) kopija metode u svakom objektu (misli se na programski kod). Svaki objekt samo pokazuje na istu implementaciju. Ovo je stvar kompajlera/platforme
- S konceptualnog nivoa može se smatrati da svaki objekt ima svoju implementaciju metoda

Klase

- Klasa je nacrt po kojem se izrađuje objekt
- Kada se objekt instancira koristi se klasa kao osnova
- Objekt se ne može instancirati bez klase, prvo definiramo klasu, a onda kroz nje definiramo objekte



Klasa se može smatrati tipom podatka višeg reda

- Kao što definiramo varijable za osnovne tipove podataka npr:
- int x;
- float y;
- možemo definirati i svoj tip
- mojaKlasa mojObjekt;
- Klasa definira atribute i ponašanja koja će svi objekti nastali iz te klase imati

Klasa Osoba

```
public class Osoba
    {
        //Atributi
        private String strIme;
        private String strAdresa;
        //Metode
        public String DohvatiIme() { //Getter ili pristupnik
                return strIme;
        }
        public void PostaviIme(String i){  //Setter ili mutator
            strIme = i;
        }
        public String DohvatiAdresu() { //Getter ili pristupnik
                    return strAdresa;
        }
        public void PostaviAdresu(String a) { //Setter ili mutator
                    strAdresa = a;
        }
    }
```

- Podaci u klasi su definirani kroz atribute
- Klase sadrže atribute koji definiraju unutrašnje stanje objekta instanciranog iz te klase
- Određivanje pristupa:
- **public** - kada se podatak ili metoda definiraju kao public onda im ostali objekti mogu direktno pristupati
- **private** – kada se podatak ili metoda definiraju kao private onda im samo taj sam objekt može pristupiti
- **protected** - kada se podatak ili metoda definiraju kao protected onda im samo srodni objekti mogu pristupiti (posebno predavanje)

- Metode – implemetiraju ponašanje klase
- Svaki instancirani objekt ima metode definirane u klasi
- Metode mogu služiti da bi implementirale ponašanje potrebno za interakciju sa okolinom (drugim objektima) ili ponašanje potrebno za radu unutrašnjih aspekata klase
- Unutrašnje ponašanje se implementira isključivo privatnim metodama i na taj način nije dostupno okolini

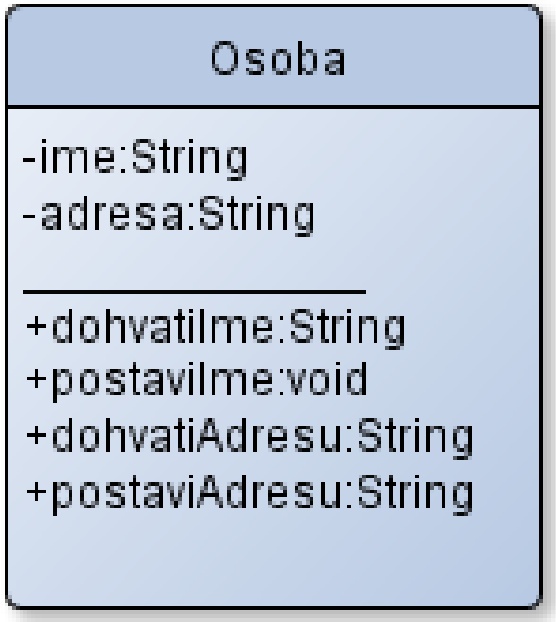
Poruke (Messages)

- Poruke su komunikacijski mehanizmi među objektima
- Kada objekt A poziva metodu objekta B, onda objekt A šalje poruku objektu B. Odgovor objekta B je definiran povratnom vrijednošću.

```
public class PlatniRacun
{
    string ime;
    Osoba o = new Osoba();
    o.postaviIme("Jure");
    ....
    ....
    string i=o.DohvatiIme();
}
```

UML dijagrami za predstavljanje klasa

- UML – Unified Modeling Language



Učahurivanje (enkapsulacija) i skrivanje podataka

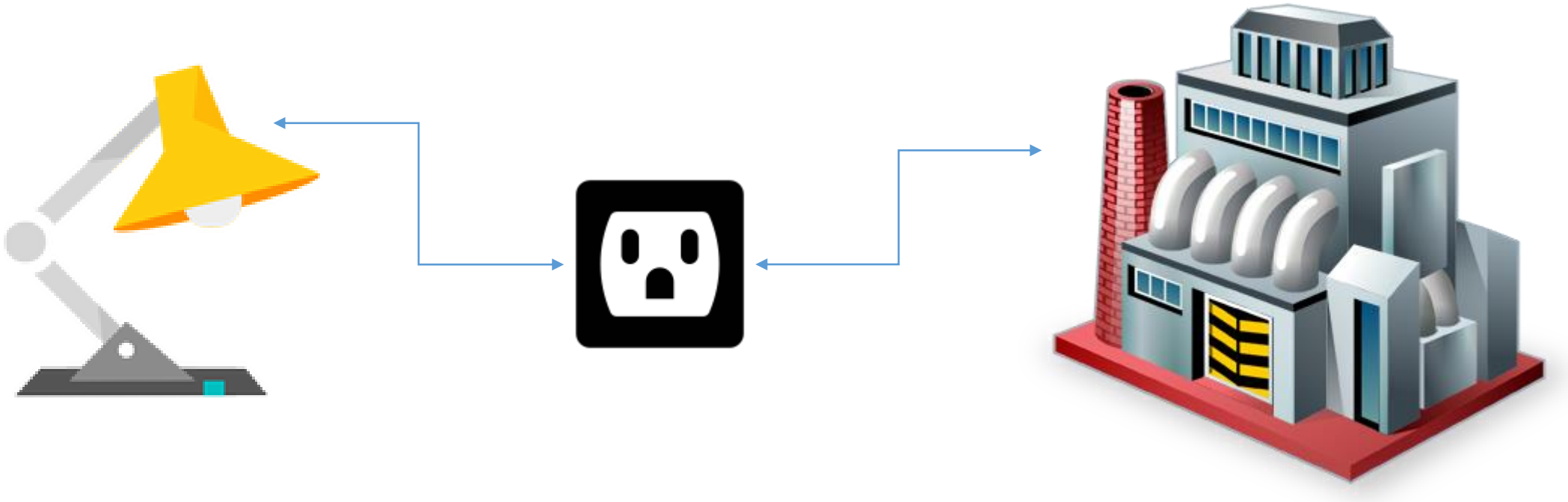
- Jedna od osnovnih prednosti objekata je da se ne trebaju otkriti svi atributi i ponašanja
- Detalji koji nisu potrebni za upotrebu/korištenje objekta trebaju se sakriti
- Prednosti ovog pristupa su vidljive na većim projektima
- Interface definira osnovne komunikacije među objektima
- Svaka klasa specificira sučelje za instanciranje i operacije objekta
- Bilo kakvo ponašanje se inicira slanjem poruke korištenjem sučelja
- U većini OO jezika metode koje su dio sučelja se definiraju kao *public*

- Da bi skrivanje podataka funkcioniralo svi atributi (ili barem osjetljivi atributi) se definiraju kao privatni i nikada nisu dio sučelja. Samo public metode predstavljaju sučelje
- Deklariranje atributa kao public poništava koncept skrivanja podataka
- U pravilu se atributima pristupa isključivo preko pristupnika i vrijednosti atributa se mijenjaju preko mutatora. Na takav način se osiguravamo da se atributima pristupa na kontroliran i siguran način te štitimo unutrašnje stanje objekta

Implementacija metoda

- Sa aspekta korisnika nekog objekta nije bitno na koji način je metoda implementirana, samo je bitno da vraća pravi rezultat
- Skrivanje implementacije olakšava programerima rad u timovima
- Za ovakav pristup je vrlo bitno testiranje koda
- Metode trebaju biti detaljno testirane prije nego li se počnu primjenjivati u projektima
- Npr. implementacije onda može naknadno mijenjati da bude brža/učinkovitija bez da sam korisnik tog objekta to uzima u obzir

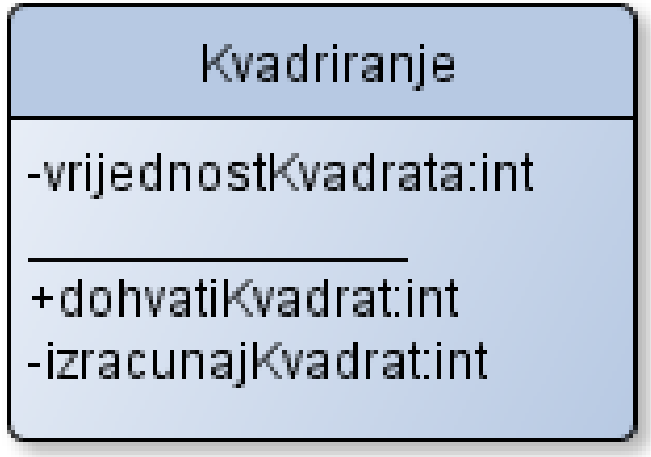
Primjer interface-implementacija



Icons by PixelKit ,WPZOOM and Iconshock

Primjer interface-implementacija

- Razdvajanje interface-a i implementacije

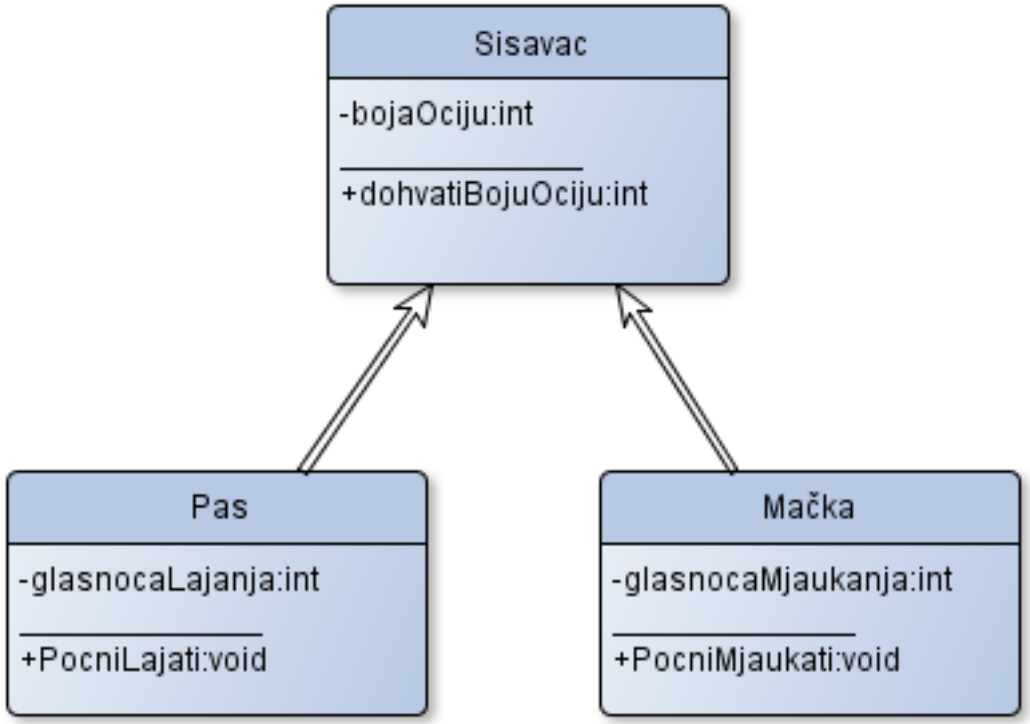



```
public class Kvadriranje
{
    //privatni atributi
    private int vrijednostKvadrata;

    //javno sučelje
    public int dohvatiKvadrat(int vrijednost)
    {
        vrijednostKvadrata = izracunajKvadrat(vrijednost);
        return vrijednostKvadrata;
    }
    //privatna implementacija
    private int izracunajKvadrat(int vrijednost)
    {
        return vrijednost * vrijednost;
    }
}
```

Nasljeđivanje

- Jedna od značajki programiranja je ponovno korištenja koda (code reuse) – procedure…
- OO ide korak dalje korištenjem nasljeđivanja – organiziranje klasa sa međusobnim odnosima, code reuse, bolji dizajn
- Klase prilikom nasljeđivanja dobiju atribute i metode druge klase
- Jedna od osnovnih pristupa prilikom stvaranja klase izdvajamo značajke koje su zajedničke nekom apstraktnom objektu



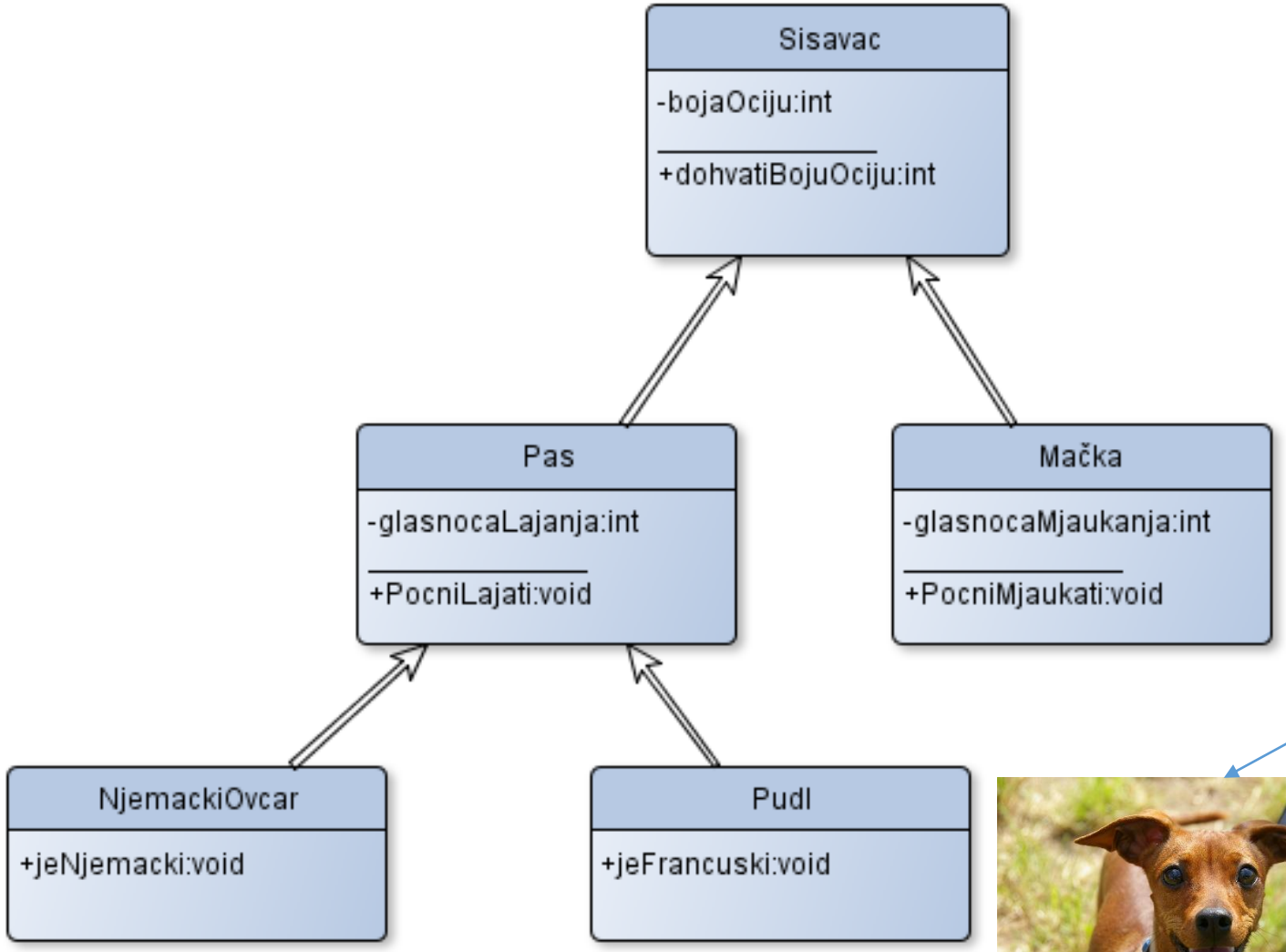
- Pas i mačka nasljeđuju klasu Sisavac i imaju atribute koje ima ta klasa
- Za klasu koja nasljeđuje možemo definirati i zasebne/specifične atribute

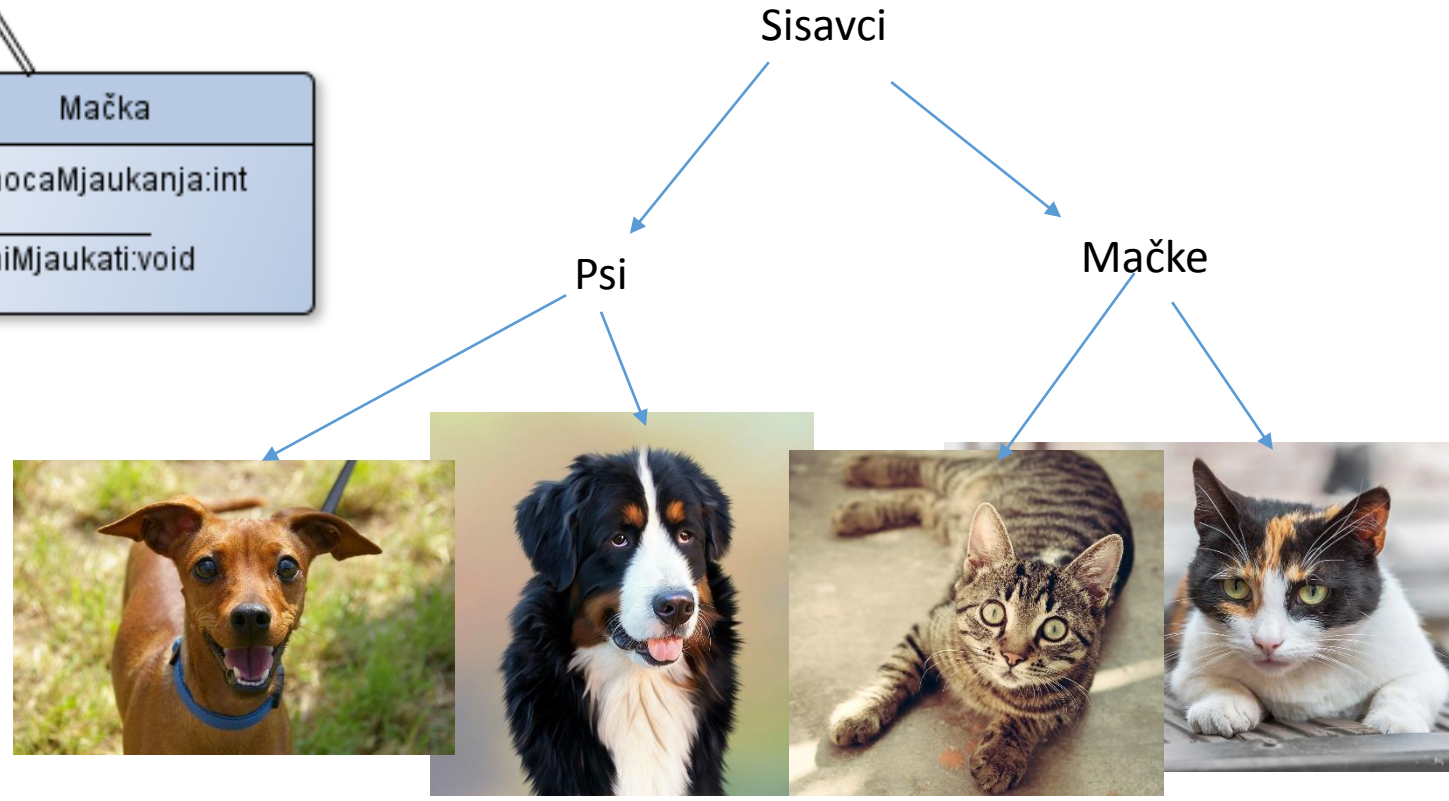
Nadklase i podklase

- Nadklasa - roditelj (eng. superclass) grupira/sadrži sve atribute koji su zajednički toj apstaktnoj cjelini i klasama koje ju nasljeđuju
- Točna apstrakcija je potrebna na bi se izbjegao dupli kod u nasljeđenim klasama i bolje modelirao problem
- Podklasa – dijete (eng. subclass) implementira samo one značajke koje su specifične za taj podskup
- Podklasa može biti nadklasa nekoj drugoj klasi

Apstrakcija

- Glavna prednost nasljeđivanja je u mogućnosti apstrakcije i organizacije
- Klasa može imati više podklasa
- U novijim OO programskim jezicima (Java i .NET) klasa može imati sam jednog roditelja i više djece
- Neki jezici poput C++ dozvoljavaju da klasa ima više roditelja
- Single inheritance ili multiple inheritance

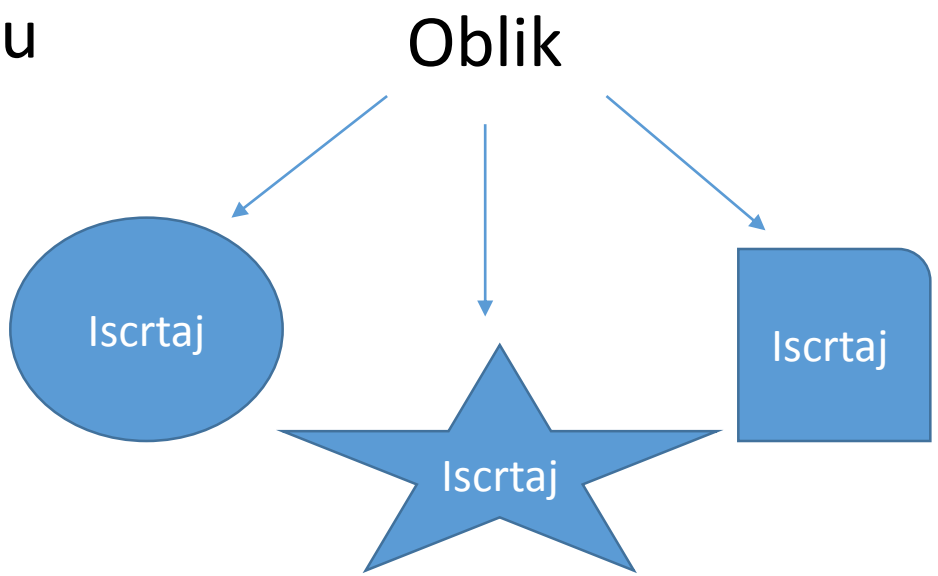

Sisavac
-bojaOciju:int
+dohvatiBojuOciju:int
Pas
-glasnocaLajanja:int
+PocniLajati:void
Mačka
-glasnocaMjaukanja:int
+PocniMjaukati:void
NjemackiOvcar
+jeNjemacki:void
Pudl
+jeFrancuski:void


Sisavci
Psi
Mačke

Is-a odnos

- Svaka podklasa implementira svoju metodu Draw ali se sve isto zovu
- Standardiziranje korištenja objekata
- Osnovni koncept polimorfizma – „odgovornost” leži na svakom pojedinom objektu da se nacrta
- Ovo je standardan koncept u razvoju

softvera za crtanje ili obradu teksta



Polimorfizam

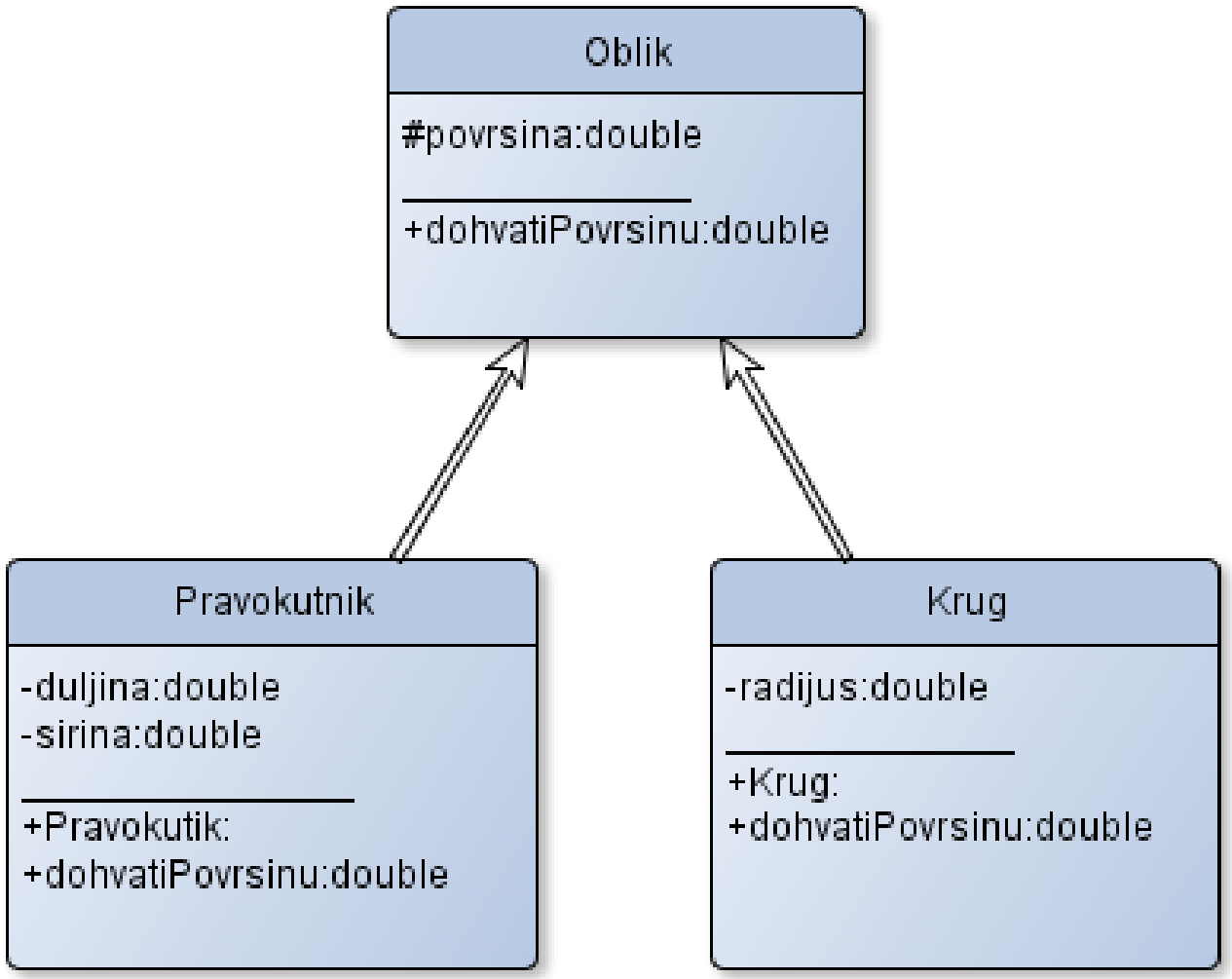
- Grčka riječ koja znači - mnogo oblika
- Iako je povezan sa nasljeđivanje često se navodi sam za sebe kao jedna od najvećih prednosti OOP
- Kada se poruka šalje objektu on mora odgovoriti na tu poruku (pozivanje metode)
- Pri nasljeđivanju sve podklase sadrže isti interface kao i roditeljska klasa
- Stvaraju se situacije da svaka podklasa treba imati svoju implementaciju iste metode (odgovora na poruku)

Primjer – polimorfizam i apstraktne metode

```
public abstract class Oblik
{
    private double povrsina;
    public abstract double dohvatiPovrsinu();
}
```

```
public class Krug extends Oblik
{
    double radijus;
    public Krug(double r)
    {
        radijus =r;
    }
    public double dohvatiPovrsinu()
    {
        povrsina=3.14*radijus*radijus;
        return povrsina;
    }
}
```

```
public class Pravokutnik extends Oblik{
    double sirina;
    double visina;
    public Pravokutnik(double s, double v)
    { sirina=s;
        visina=v;
    }
    public double dohvati Povrsinu()
    {
        povrsina=sirina*visina;
        return povrsina;
    }
}
```


Oblik
#povrsina:double
+dohvatiPovrsinu:double
Pravokutnik
-duljina:double
-sirina:double
+Pravokutik:
+dohvatiPovrsinu:double
Krug
-radijus:double
+Krug:
+dohvatiPovrsinu:double

Instanciranje objekata iz klase Oblik

```
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4,5);
```

- Dodavanje objekata na stog

```
stack.push(circle);
stack.push(rectangle);
```

- Implementacija

```
while (  !stack.empty()) {
    Shape shape = (Shape) stack.pop();
    System.out.println ("Area = " + shape.getArea());
}
```

Kompozicija/skup objekata

- Jedan objekt može biti sačinjen od više različitih objekata
- Računalo se sastoji od različitih komponenti – grafička kartica, disk, memorija, procesor…
- Na sličan način se programski objekti mogu ugnježđivati

„Is-a” i „has-a” odnosi

- Postoje dva načina da se klase konstruiraju iz drugih klasa: nasljeđivanje i kompozicija
- Nasljeđivanjem se apstrahiraju svojstva i ponašanja zajedničke klase
- „pas” je u „is-a” odnosu prema klasi „sisavac”
- „is-a” se koristi kada se opisuje odnos u nasljeđivanju
- „has-a” se koristi kada se opisuje odnos u kompoziciji
- „auto” je u „has-a” odnosu sa „volanom”

Kompleksnost

- „The more complex the system, the more open it is to total breakdown”
- Naknadni zahtjevi – npr. građevinski radovi dodavanja podruma ispod zgrade od 100 katova
- Sličnost sa korisnicima i zahtjevima prilikom izrade softvera (npr. to je samo stvar programiranja)
- Problemi kod savladavanja kompleksnosti softvera dovode do:
  - kašnjenja projekata
  - povećana cijena
  - loš softver

Kompleksnost

- Loše organizacija kompleksnih projekata vodi:
  - neučinkovitom korištenju ljudskih resursa
  - gubitak prilika na tržištu
- Nema dovoljno dobrih developera da bi se loše vodilo projekt
- Određeni broj je posvećen održavanju i nadogradnjama
- Najbitnije je u startu početi sa ispravnim konceptom i dizajnom arhitekture našeg softvera
- Naknadne promjene vode velikim troškovima i softverom teškim za održavanje

Struktura osobnog računala

- Osobno računalo je uređaj određene složenosti
- Sastoji se od nekoliko glavnih elemenata:
  - CPU, monitor, tipkovnica, DVD, HDD
- Svaki od ovih elemenata može dalje rastaviti na manje cjeline, npr: CPU se sastoji od: primarne memorije, aritmetičko logičke jedinice (ALU), sabirnice na koju su spojeni periferni uređaji
- ALU se može onda detaljnije raščlaniti na registre i upravljačku logiku, koji se onda mogu raščlaniti na primitivnije elemente kao: NAND gates, inverteri
- Vidimo da postoji hijerarhija kompleksnog sustava

Struktura osobnog računala

- Možemo razmišljati o tome kako računalo radi na način da ga raščlanimo na dijelove koje onda zasebno možemo analizirati
- Različiti nivoi hijerarhije predstavljaju različite razine apstrakcije
- Na svakoj razini apstrakcije imamo različite kolekcije „uređaja” koji zajedno surađuju da bi pružili uslugu višim slojevima
- Prilikom analize odabiremo nivo apstrakcije koji nam je u tom trenutku potreban
- Npr. ako želimo pronaći grešku u tajmingu primarne memorije možemo promatrati gate-level arhitekturu računala, ali ovaj nivo apstrakcije nam neće koristiti ako tražimo pogrešku u aplikaciji za generiranje izvještaja

Primjer – nivoi apstrakcije biljke

- Biljka se sastoji od osnovnih elemenata:
  - korijen
  - stabljika
  - listovi
- Svaki od ovih elemenata ima svoju specifičnu strukturu
- Korijen se sastoji od grana korijena, korijenski kapilari, korijensko tkivo
- Na različitim nivoima apstrakcije elementi različito međudjeluju
  - Korijen je zadužen za apsorpciju vode i minerala iz zemlje
  - Stabljika ih transportira u lišće
  - Lišće koristi vodu i minerale, te obavlja fotosintezu
- Ono što su NAND elementi kod računala kod biljke su stanice

Definiranje kompleksnosti softvera

- Postoje određene kategorije softvera koji nisu kompleksni
- Obično su to aplikacije koje je napravila, održava i koristi ista osoba
- Takav softver ima ograničenu svrhu i životni vijek
- Možemo priuštiti da odbacimo takav softver i iznova napravimo novi ako nam zatreba drukčija funkcionalnost
- S druge strane postoji tzv. industrijski softver
- Sadrži vrlo bogatu lepezu ponašanja, npr. reagira na događaje iz stvarnog svijeta, sa značajnim vremenskim uvjetima i ograničenjima
- Npr. sustav za kontrolu leta
- Takav tip softvera ima dug životni vijek

Definiranje kompleksnosti softvera – nastavak

- Veliki broj ljudi ovisi o ispravnom funkcioniranju softvera.
- Kod industrijskog softvera postoji i framework za izradu komponenti za određenu domenu
- Framework olakšava izradu softvera korištenjem već dizajniranih funkcionalnosti
- Kod industrijskog softvera velikih razmjera nemoguće je za individualnog developera da razumije cjelokupni sustav u potpunosti
- Kompleksnost cjelokupnog takvog sustava nadilazi kapacitet pojedinog čovjeka
- Možemo savladati ovakvu kompleksnost apstrakcijom ali ona i dalje postoji

Zašto je softver kompleksan?

- “The complexity of software is an essential property, not an accidental one” Fred Brooks
- Accidental complexity – kompleksnost koju dizajnom uvodi developer
- Essential complexity – kompleksnost koja je svojstvena domeni

Kompleksnost problemske domene

- Problemi koje rješavamo prilikom dizajna softvera često uključuju neizbježne elemente kompleksnosti, gdje nailazimo na veliku količinu zahtjeva koji su čestu neusklađeni, a ponekad i kontradiktorni
- Uzmimo probleme autonomnih robota, ili centrale za mobitele – osnovna funkcionalnost je u startu kompleksna – sada treba dodati i nefunkcionalne zahtjeve kao što su usability, performanse, cijena pouzdanost…
- Ovakvi zahtjevi se često sami po sebi podrazumijevaju
- Korisnici često ne znaju ni jasno artikulirati svoje zahtjeve „komunikacijski jaz” da bi ih developeri razumjeli

Kompleksnost problemske domene

- Često korisnici imaju samo nejasnu ideju što žele softverskom sustavu
- Nemogućnost potpune komunikacije nije grešaka ni korisnika ni developera
- Oboje često nemaju znanja o domeni onog drugog i imaju drukčiju perspektivu o problemu i potencijalom rješenju
- Dodatnu kompleksnost uvodi i činjenica da se zahtjevi mijenjaju tijekom razvoja
- Rani prototipovi često korisnika natjeraju da preispita promjeni ili nadopuni svoje zahtjeve, te bolje artikulira svoje potrebe

Poteškoće upravljanja razvojnim procesom

- Osnovni cilj razvoja softvera je da se stvori iluzija jednostavnosti – zaštititi korisnika od inherentne kompleksnosti
- Veličina softvera – manje je više
- Koristimo različite mehanizme (fremework i sl.)da bi reducirali broj linija koda i povećali jednostavnost
- Danas nije neuobičajeno da se broj linija koda mjeri u stotinama tisuća ili milijunima (i to u high-order programskim jezicima)
- Pojedina osoba ne može toliki sustav u cijelosti poznavati
- Timski rad – po mogućnosti manji timovi
- Veliki timovi: otežana komunikacija, koordinacija, geografska ograničenja

Fleksibilnost softvera

- Građevinarska firma
  - nema svoju šumu od koje radi materijal
  - ne lijeva čelične grede na lokaciji gradnje
- U softverskom svijetu ove prakse su česte, developeri često proizvode i najmanje građevne jedinice programa
- Ovakva fleksibilnost softvera dozvoljava developeru da izrazi bilo kakvu vrstu apstrakcije
- Građevna industrija ima standarde kvalitete za jednostavne građevne elemente, to nije toliko izraženo softverskoj industriji

Problemi karakteristični za diskretne sustave

- Kod velikih aplikacija može sadržavati stotine i tisuće varijabli i kontrolnih mehanizama
- Kolekcija ovih varijabli, njihova stanja, adresni prostor i pozivni stog (calling stack) predstavljaju trenutno stanje programa
- Digitalna računala predstavljaju diskretni sustav
- Diskretni sustavi po njihovoj prirodi imaju konačan broj stanja (iako taj broj može biti jako velik)
- Sustavi se dizajniraju na način da promjena u jednom dijelu sustava minimalno ili nikako utječe na operativnost ostalih dijelova sustava
- Ako modeliramo ili obavljamo interakciju sa kontinuiranim sustavom svaki vanjski događaj može dovesti naš sustav u novo stanje (možda i nepredviđeno)

Problemi karakteristični za diskretne sustave

- Postoje brojni problemi i zabilježeni su razni propusti vezani za softver koji upravlja npr. podzemnim željeznicama, automobilima, satelitima…
- Ovo je motivacija za iscrpno testiranje softvera
- Ali za veliku većinu, osim za trivijalne sustave potpuno testiranje je nemoguće

Atributi kompleksnog sustava

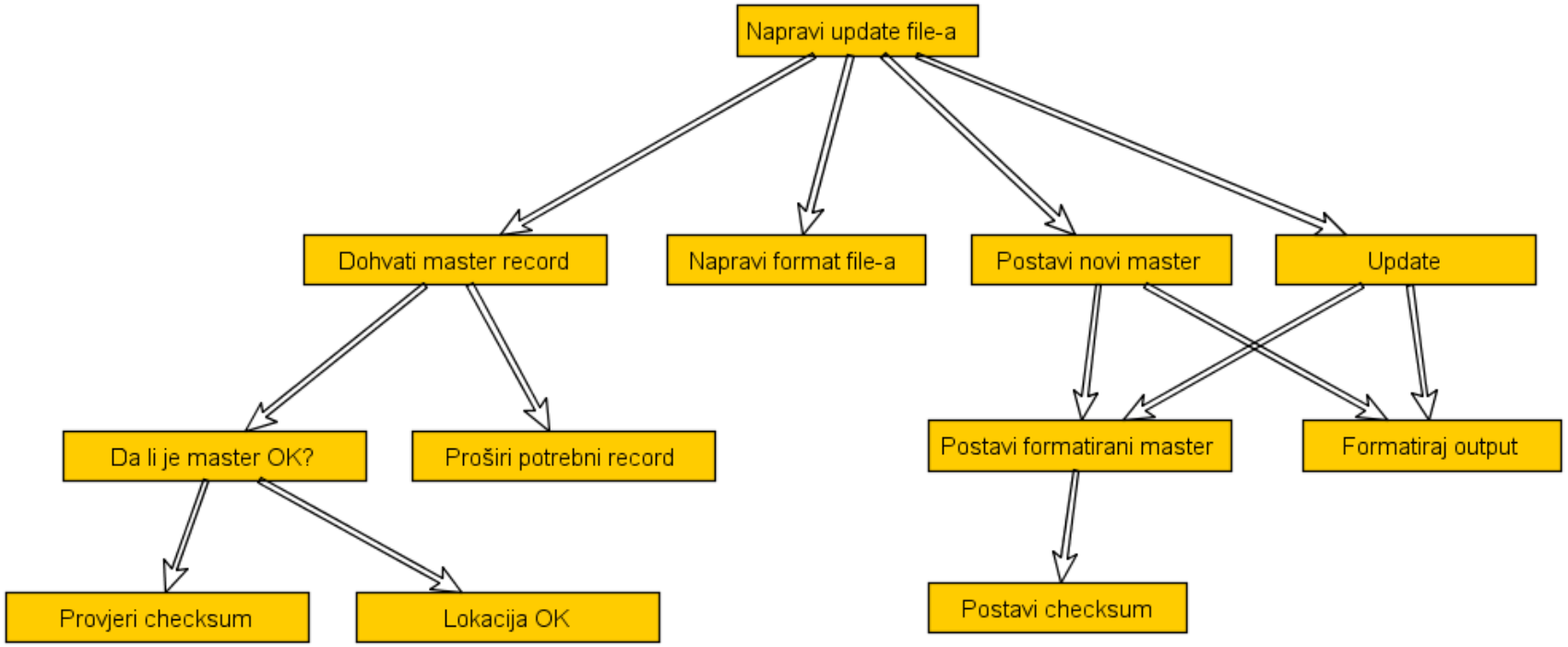
- Hijerarhijska struktura
- Relativni osnovni djelovi – najmanja komponenta sustava se arbitrarno određuje i ovisi o perspektivi sagledavanja sustava
- Zajednički uzorci (eng. patterns) – hijerarhijski sustavi su uglavnom sastavljeni manjih komponenti koji se nalaze u većini podsustava (npr. stanice u biljkama i životinjama)
- Stabilne među-faze sustava – Kompleksni sustavi se uglavnom razvijaju iterativno od manje kompleksnosti prema višoj. Svaka među-faza tog procesa treba biti stabilna

Nošenje s kompleksnošću

- Razvoj kompleksnih sustava je iznimno zahtjevan
- System architects – ljudi koji definiraju arhitekturu novog sustava
- Uloga procesa **dekompozicije** – „podijeli pa vladaj”
- Svaki dio dekompozicije se može dalje raščlaniti u manje dijelove do najmanje potrebne razine apstrakcije
- Da bi razumjeli neki nivo sustava potrebno je poznavati određeni dio dijelova, ali ne i sve

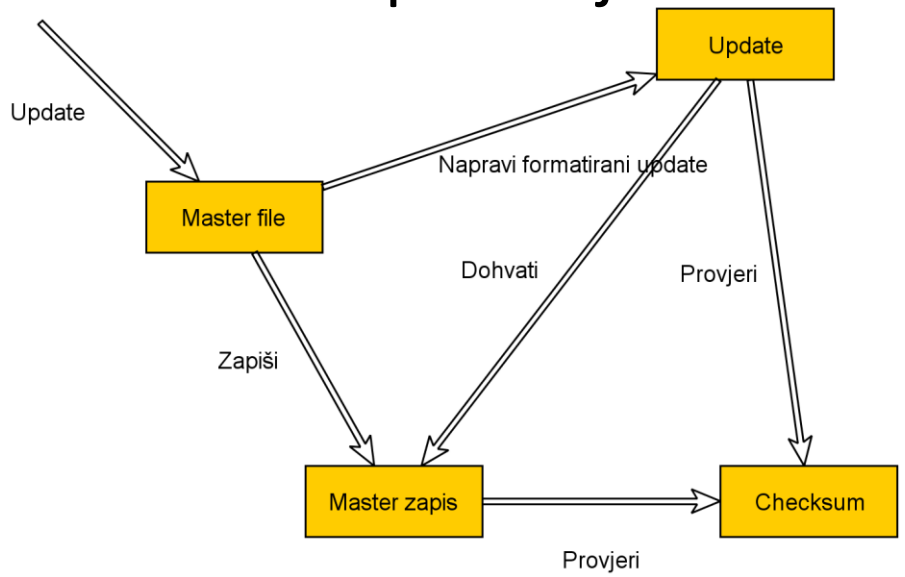
Algoritamska dekompozicija

- Većina ljudi za dekompoziciju koristi pristup algoritamske dekompozicije gdje se važniji procesi definiraju kao veće cjeline u sustavu



Objektno-orijentirana dekompozicija

- Umjesto dekompozicije u korake/procese radi se dekompozicija u objekte
- Oba pristupa rješavaju problem, ali na različite načine
- U OO pristupu gledamo na svijet kao na skup autonomnih agenata koji surađuju da bi odradili kompleksnije zadatke



Algoritamska ili OO dekompozicija?

- Oba pristupa su važna
- Algoritamska dekompozicija naglašava redoslijed događaja u nekom procesu
- OO dekompozicija naglašava koji su agenti (objekti) inicijatori neke aktivnosti, a koji su oni na kojima odrađuju radnje
- Ne možemo dizajnirati sustav simultano koristeći oba procesa
- ZA kompleksne sustave uobičajeno je da se koristi OO dekompozicija budući da je na takav načni lakše organizirati kompleksne sustave
- OO dekompozicija rezultira manjim sustavima ponovnim korištenjem koda (eng. code reuse)
- OO sustavi su otporniji na buduće promjene i bolje evoluiraju kroz vrijeme zbog stabilnih među-faza

Uloga apstrakcije

- Pojedina osoba može kratkoročno zapamtiti 7 ± 2 pojmova/informacija
- Ovaj broj je neovisan o sadržaju informacije
- Organizacijom unosa informacija u skupine srodnih dijelova i podjelom u više različitih osnova/dimenzija možemo olakšati ovaj proces razumijevanja kompleksnosti
- U slučaju da kompleksni objekt ne možemo pojmiti u cijelosti koristimo **apstrakciju**, tj. odbacimo neesencijalne detalje i fokusiramo se samo na generalizirani idealni model

Uloga apstrakcije

- Prilikom proučavanja fotosinteze – fokusiramo se samo na kemijske procese u listovima, a zanemarujemo ostale dijelove biljke poput korijena i stabljike
- Prilikom apstrakcije i dalje smo ograničeni količinom informacija koje možemo zadržati, ali u obzir uzimamo samo one dijelove informacija s većim semantičkim sadržajem ili značajem
- Ovo je posebno važno kada modeliramo probleme iz stvarnog svijeta koji sadrže veliku količinu informacija

Elementi metodologije dizajna softvera

- Ne postoji jedinstven način dizajna koji vodi od zahtjeva do konačne implementacije kompleksnog softverskog sustava
- Radi se o inkrementalnom i iterativnom procesu
- Ali ipak, postoje određeni elementi koje se koriste u razvojnom procesu:
  - **Notacija** – jezik za izražavanje modela
  - **Proces** – Aktivnosti koje vode konstrukciju (prikupljanje zahtjeva, implementacija, testrianje)
  - **Alati** – služe za izgradnju modela, definiranje pravila o modelima …