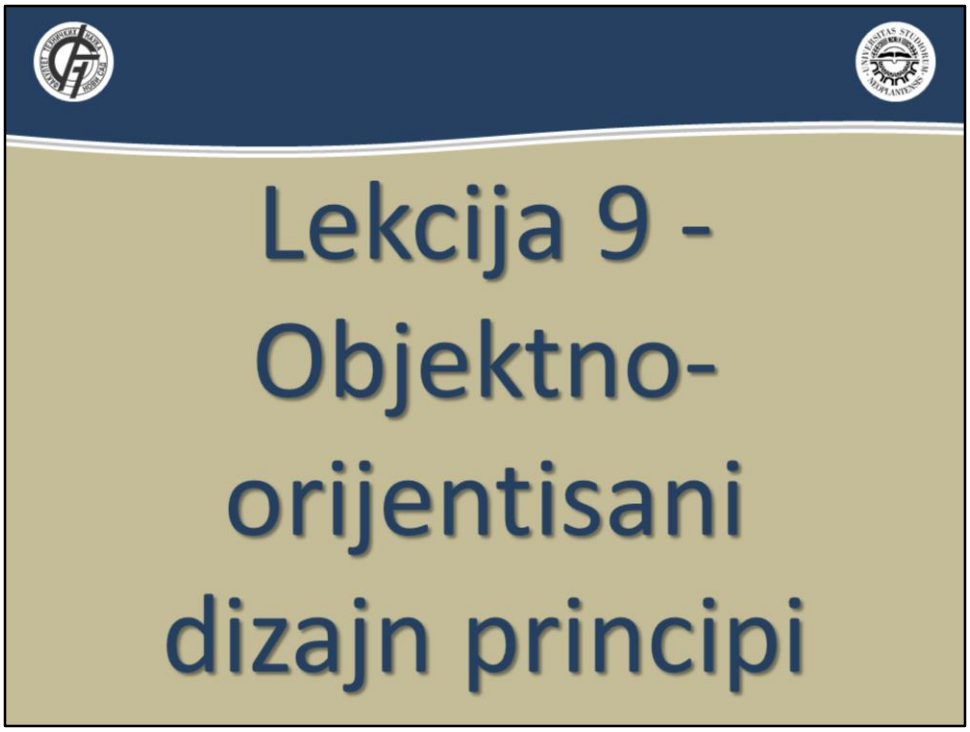
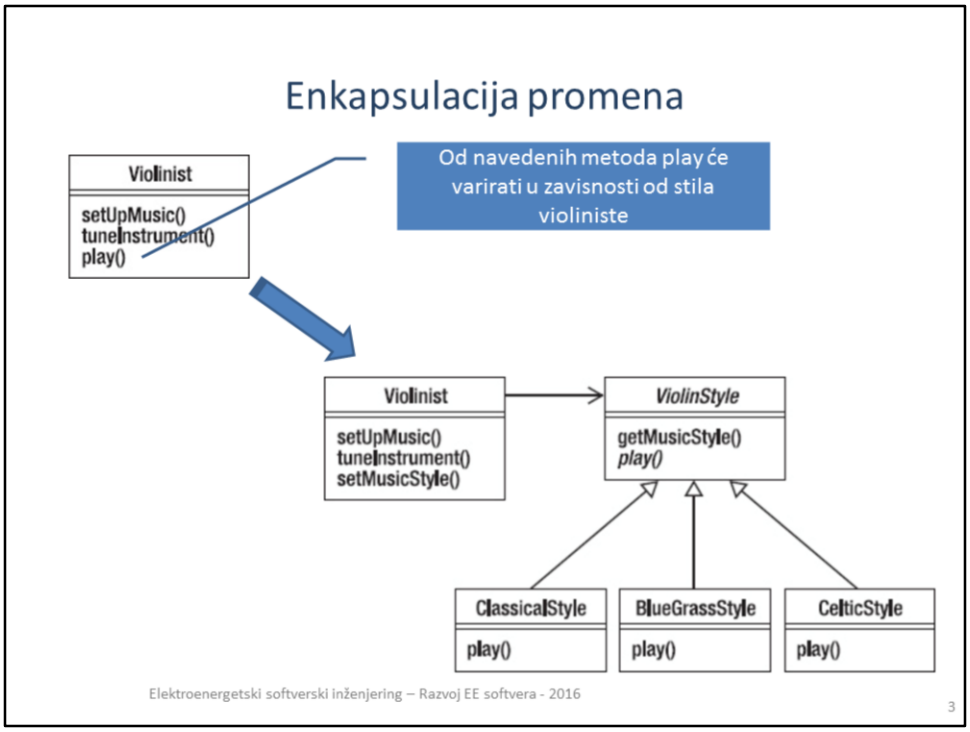

Lekcija 9 -
Objektno-
orijentisani
dizajn principi

Lista fundamentalnih OO principa

- Razdvojte stvari koje se menjaju od onih koji su statični
- Programirajte korišćenjem interfejsa umesto konkretnih klasa
- *Open-Closed* Princip (OCP)
- *Don't Repeat Yourself* Princip (DRY) – veoma je bliska sa *One Right Place* principom, jer se izbegava dupliranje funkcionalnosti na više mesta
- *Single Responsibility* Princip (SRP) – veoma usko povezana sa principom visoke kohezije klasa
- Liskovljev princip substitucije
- *Dependency Inversion* Princip (DIP) – usko vezana za drugi princip u listi u vezi korišćenja interfejsa umesto implementacionih konstrukcija
- *Interface Segregation* Princip (ISP)
- Princip *Least Knowledge* (PLK) – takođe poznat kao Demeterov zakon (ne govori sa "strancima")
- Princip slabe sprege

U ovom spisku se mnogi principi navode na engleskom, pošto su toliko poznati da bi njihov prevod uneo više zabune nego koristi.

Dizajn paterni su bazirani na ovim fundamentalnim principima i daju smernice kako ih primeniti u konkretnim situacijama.



Ovde vidimo da je cela hijerarhija klasa kreirana za aspekt sistema, gde se očekuju najviše promena. Apstraktna klasa ViolinStyle daje okvir (interfejs) ostalim konkretnim izvedenim klasama.

NAPOMENA:

Da li su metode setMusicStyle, getMusicStyle usklađene sa nazivom ViolinStyle? Kako poslati poruku viloniste da svira, tj. kako mu reći play? Koji je smisao metode getMusicStyle kao deo klase ViolinStyle (ona je tu navedena da je konkretna, pa stoga implementaciju nasleđuju sve izvedene klase)? Da li je realno očekivati da će neki violinista biti voljan i sposoban da podrži sve moguće stilove?

Ovde takođe moramo biti sigurni da različiti stilovi violina ne zahtevaju drugačije štimovanje (metoda tuneInstrument).

Kodiranje ka interfejsu

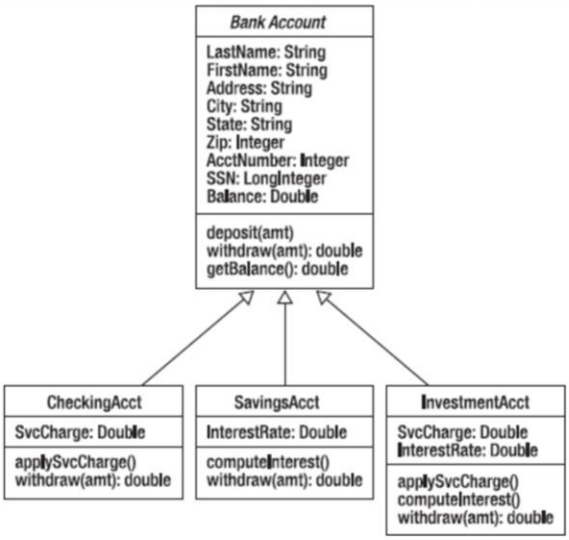
- Umesto da klijent referencira konkretne klase treba da koristi interfejse.
- Interfejs skriva implementacione detalje od klijenta (konkretna klasa je jedan vid takvog detalja).
- Interfejs služi kao medijator između klijenta i konkretnih klasa. Ono definiše međusobni ugovor između ovih subjekata.
- Primer: videti hijerarhiju klasa *Shapes* iz knjige.

NAPOMENA:

Unutar klase ShapesTest vidimo da su neke linije iskomentarisane unutar metode main. Koliko je zgodno kontrolisati tok izvršenja programa komentarisanjem linija kôda?

Open-Closed princip (OCP)

- Klase treba da su otvorene za proširivanje, a zatvorene za modifikaciju.



Don't Repeat Yourself princip (DRY)

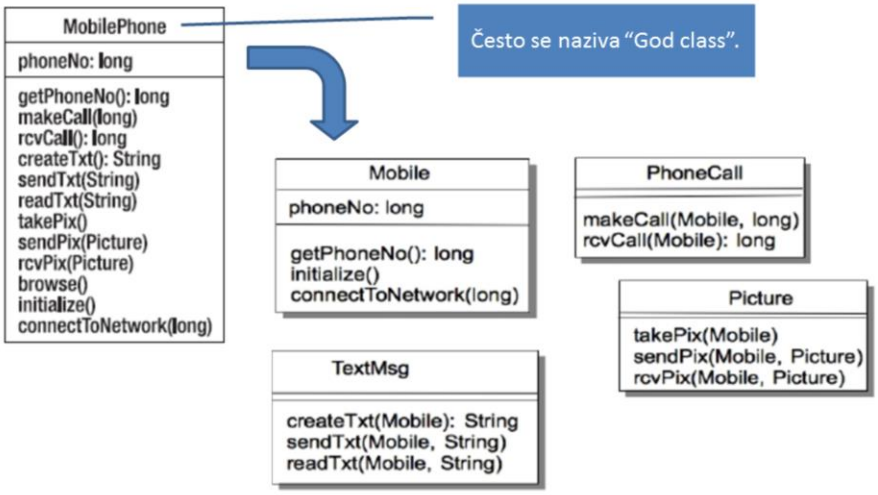
- Izbegavati dupliranje (osobina, zahteva, kôda), jer se samo tako može ostvariti ideal jedan zahtev -> jedno mesto implementacije.
- Dupliranje kôda znači dupliranje ponašanja sistema.
- *Copy-paste* je uvek dupliranje i nikako mehanizam za ponovno korišćenje (*reuse*).
- Primer: analizirati proširenu B^4++ aplikaciju iz knjige. Ovde je dodata mogućnost otvaranja/zatvaranja vrata na osnovu sistema za prepoznavanje govora ptica (na primer, vrata se automatski zatvaraju ako je detektovan zvuk nepoznate ptice). Međutim, FeedingDoor klasa je i dalje ostala nepromenjena sa jednom jedinom funkcijom – kontrola vrata.

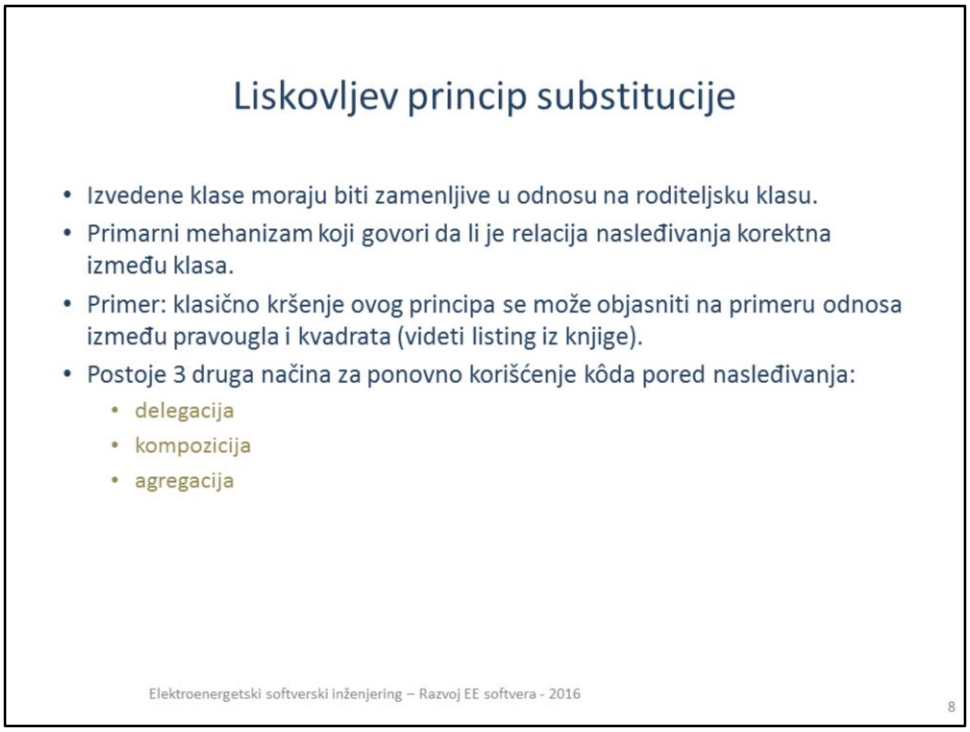
NAPOMENA:

Klasa BirdFeeder je zadužena za skladištenje zvukova ptica pevačica, dok klasa SongIdentifier šalje upit za listom zvukova ka BirdFeeder-u. Da li bi bilo bolje pesme ptica čuvati u SongIdentifier-u? Obrazložite odgovor (da ili ne) u oba slučaja. Koliko ova odluka zavisi i od vrste sistema (desktop, lokalno distribuiran sistem, itd.)? BirdFeeder ima metodu getDoor, i tu metodu eksplicitno poziva SongIdentifier. Međutim, RemoteControl je ni dalje ne koristi.

Single Responsibility princip (SRP)

- Klasa treba da ima jednu jedinu funkciju, i naravno razlog da se menja.





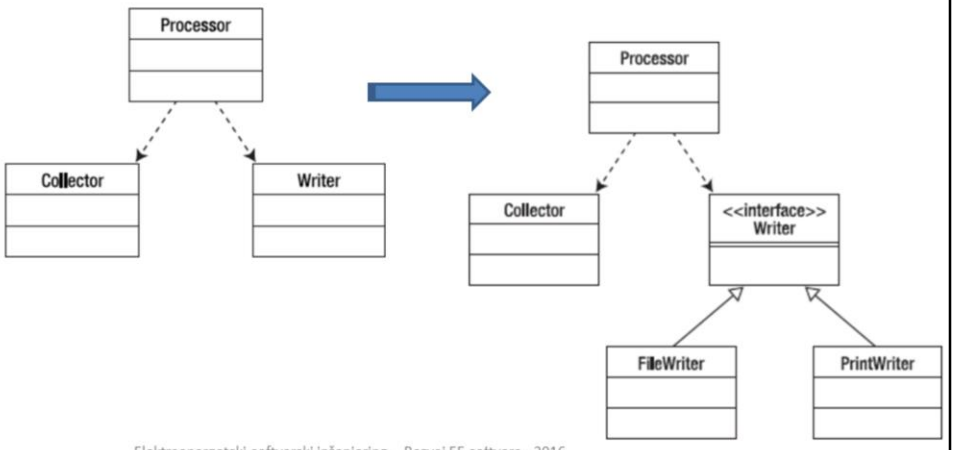
Odličan primer za kršenje ovog principa je klasa Properties iz Javine biblioteke. Ona je izvedena iz klase Hashtable. Šta sve može da se dogodi zbog ovoga? Koja druga forma (delegacija, kompozicija ili agregacija) bi bila bolja u ovom slučaju od nasleđivanja? Analizirajte i drugi standardni primer iz Jave, a to je klasa Stack. Ovde je čak i apsurdno govoriti da je stek IS-A vektor. U prvom slučaju je još nekako i moglo da se kaže da je Properties IS-A Hashtable, jer liči na mapu.

NAPOMENA:

Metode za postavljanje “stranica” kvadrata su iskomentarisane unutar Square.java. Da bi se dobio izuzetak u klasi SquareTest ove prethodno pomenute metode treba omogućiti (ne zaboravite da na kraju izvršite: javac Square.java). Program pozivate sa: java –cp . –ea SquareTest

Dependency inversion princip (dip)

- Veoma usko vezana za princip kodiranja korišćenjem interfejsa umesto konkretnih klasa. Naime, svaka zavisnost od višeg ka nižem nivou mora ići preko odgovarajuće apstrakcije.



NAPOMENA:

Na slici to nije prikazano, ali se sličan pristup treba primeniti i u slučaju zavisnosti ka klasi Collector (pogotovo ako u budućnosti postoje planovi za različitom vrstom akvizicije podataka).

Interface Segregation Princip (ISP)

- Praktično ovo je *Single Responsibility* princip primenjen na interfejse.
- Štiti klijente od zavisnosti koje ne koriste, dok izvedene klase od nepotrebne implementacije "višak" metoda.

Demeterov zakon

- Klase treba indirektno da sarađuju sa što manjim brojem drugih klasa.
- Treba izbegavati sledeću konstrukciju u programu (recimo, da se sledeća linija nalazi u nekoj metodi objekta `Obj`):

```
vr = obj1.getObj2().getObj3().getObj4().vrednost()
```

Ovde postoji zavisnost ne samo ka klasi `Obj1` (preko reference `obj1`), već indirektno i na `Obj2`, `Obj3` i `Obj4`. Bolje je:

```
vr = obj1.vrednost()
```

Ovde sada `Obj1` treba da bude *Facade* ka ostalim klasama.

Uvek se treba truditi zavisnosti eksplicitno izraziti. U primeru, klasa Obj bi samo rekla import Obj1, ali realno importuje i sve ostale. Skrivanje veza umnogome otežava održavanje programa, jer promena u nekoj od referenciranih klasa može dovesti do čitave lančane reakcije.