PROJEKTOVANjE
SOFTVERA

Projektovanje softvera

Skripta za kurs „Projektovanje softvera"
na Elektrotehničkom fakultetu u Beogradu

Sadržaj

1. Predgovor **3**

2. Uvod u projektovanje softvera **4**

3. UML **9**

3.1. Jezik UML 10
3.2. Primer modela za program 17
3.3. Dijagrami klasa 19
3.4. Dijagrami paketa 27
3.5. Dijagrami objekata 30
3.6. Dijagrami interakcije 32
3.7. Dijagrami slučajeva korišćenja 38
3.8. Dijagrami stanja 42
3.9. Dijagrami aktivnosti 47
3.10. Dijagrami složene strukture 54
3.11. Dijagrami komponenata 56
3.12. Dijagrami raspoređivanja 59
3.13. Dijagrami klasa - napredni pojmovi 61
3.14. Dijagrami interakcije - napredni pojmovi 66

3.15. Arhitektura metamodeliranja 69

4. Projektni uzorci 71

4.1. Uvod u projektne uzorke 72
4.2. Unikat 77
4.3. Šablonski metod 79
4.4. Prototip 81
4.5. Sastav 83
4.6. Dekorater 85
4.7. Posmatrač 88
4.8. Iterator 90
4.9. Strategija 94
4.10. Stanje 96
4.11. Podsetnik 98
4.12. Muva 100
4.13. Adapter 102
4.14. Fasada 105
4.15. Fabrički metod 107
4.16. Apstraktna fabrika 109
4.17. Most 112
4.18. Komanda 115
4.19. Zastupnik 118
4.20. Posrednik 121
4.21. Lanac odgovornosti 123
4.22. Graditelj 125
4.23. Posetilac 128
4.24. Interpreter 131

Predgovor

Ova skripta je napravljena u svrhu organizacije svih prezentacija koje se koriste na predavanjima iz Projektovanja softvera u jedan dokument radi preglednosti i jednostavnosti štampanja materijala.

Skripta sadrži tekst i slike koje se nalaze na predmetnom sajtu za Projektovanje softvera, uz minimalne izmene. Kako je celokupan sadržaj preuzet sa prezentacija, sva prava pripadaju autoru istih, prof. Igoru Tartalji.

Prezentacije korišćene u ovoj skripti se nalaze na sledećem linku: https://rti.etf.bg.ac.rs/rti/ir4ps/

Iako je uloženo maksimalno truda da ne bude grešaka u formatiranju teksta, kao i gramatičkih grešaka, moguće je da su neke ipak promakle, te da su prisutne.

Prva verzija skripte je napravljena u septembru 2022. godine. Naknadno, u trenutnoj verziji, su ispravljene greške koje su uočene tokom semestra.

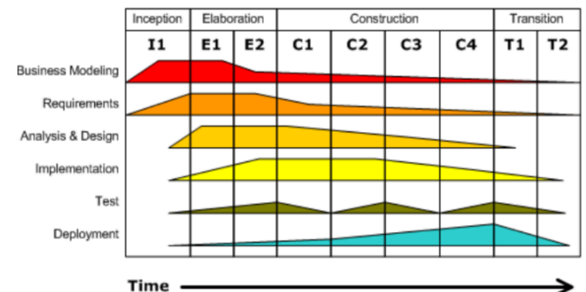
Nadam se da će ova skripta da vam pomogne da lakše naučite i položite ovaj ispit!

Februar 2023. Nikola Stanković (2020/15)

Uvod u projektovanje softvera

Proces razvoja softvera

- Faze razvoja softvera (SW) po RUP-u: započinjanje, razrada, konstrukcija, tranzicija
- U svakoj fazi se iterativno i inkrementalno prolazi kroz aktivnosti:
  - » modeliranje poslovanja,
  - » specifikacija zahteva,
  - » analiza i **projektovanje**,
  - » implementacija,
  - » testiranje,
  - » isporuka
- Različit je procenat aktivnosti po fazama



Pojmovi

- Objektno-orijentisana metodologija razvoja je dominantna u proizvodnji softvera danas
- Pojmovi:
  - » objektno-orijentisana analiza – OOA
  - » objektno-orijentisano projektovanje – OOD
  - » objektno-orijentisano programiranje – OOP
  - » objektno-orijentisani jezik – OOL

Objektno-orijentisana analiza

- Tradicionalne tehnike strukturirane analize - fokus na toku podataka u sistemu
- **Booch** (1994): Objektno-orijentisana analiza je metod analize koji ispituje zahteve iz perspektive klasa i objekata pronađenih u rečniku iz domena problema
- Proizvod OOA – konceptualni model - ulaz u fazu OOD

Objektno-orijentisano projektovanje

- Tradicionalno strukturirano projektovanje – fokus na algoritamskim apstrakcijama
- Booch (1994): Objektno-orijentisano projektovanje je metod projektovanja koji obuhvata
  - » proces OO dekompozicije
  - » notaciju za predstavljanje
    - — logičkih i fizičkih
    - — statičkih i dinamičkih

    aspekata modela sistema koji se projektuje
- Proizvod OOD – model projektovane aplikacije ili sistema – ulaz u fazu OOP

Objektno-orijentisano programiranje

- Tradicionalno strukturirano programiranje – fokus na implementaciji algoritama
- Booch (1994): Objektno-orijentisano programiranje je metod implementacije po kojem su:
  - » programi organizovani kao kolekcije objekata koji sarađuju
  - » svaki objekat predstavlja primerak neke klase i
  - » sve klase su članovi neke hijerarhije klasa u kojoj su klase povezane relacijama nasleđivanja
- Proizvod OOP je izvršna aplikacija ili sistem

Objektno-orijentisani jezik

- **Cardelli** & **Wegner** (1985): Jezik je **objektno-orijentisan** ako i samo ako ispunjava:
  - » da podržava objekte koji su apstrakcije podataka
    - — sa javnim interfejsom preko imenovanih operacija i
    - — skrivenim lokalnim stanjem
  - » da objekti imaju pridružen tip (klasu)
  - » da tipovi (klase) mogu nasleđivati atribute nadtipa (natklase)
- Ako jezik ne podržava samo nasleđivanje naziva se **objektno-baziranim** jezikom
- Objektno-orijentisani jezici su: Simula, Smalltalk, Object Pascal, Eiffel, Python, Ada95, C++, Java, C#, Visual Basic.NET, ...
- Objektno-bazirani jezici su: Ada83, VisualBasic v6,...

Principi objektno-orijentisanog modela

- **Booch OOA&D (1994):**
- Osnovni (obavezni):
  - » apstrakcija
  - » kapsulacija
  - » modularnost
  - » hijerarhija
- Dodatni (neobavezni):
  - » tipizacija
  - » konkurentnost
  - » perzistencija

- **Modifikacija:**
- Osnovni (obavezni):
  - » apstrakcija
  - » kapsulacija
  - » modularnost
  - » hijerarhija
  - » polimorfizam
- Dodatni (neobavezni):
  - » konkurentnost
  - » perzistencija

Apstrakcija i kapsulacija

- Shaw (1984): **Apstrakcija** je uprošćeni opis ili specifikacija sistema koja naglašava neke od detalja ili osobina, dok potiskuje druge
- Booch (1994): Apstrakcija ističe esencijalne karakteristrike objekta koje ga razlikuju od drugih vrsta objekata i tako definiše jasne konceptualne granice iz perspektive posmatrača
- **Kapsulacija** je proces sakrivanja onih elemenata apstrakcije koji definišu strukturu i ponašanje
- Kapsulacija služi da razdvoji konceptualni interfejs (ugovor) od implementacije apstrakcije

Modularnost i hijerarhija

- **Modularnost** je osobina sistema da se razlaže na skup kohezivnih i slabo spregnutih modula
- Moduli su fizičke jedinice (nezavisno prevođenje)
  - » predstavljaju komponente sistema
  - » mogu se održavati nezavisno
- **Hijerarhija** je rangiranje ili uređivanje apstrakcija
- Nasleđivanje - “is a” hijerarhija
  - » jednostruko/višestruko
  - » potpuno (javno)/strukturno (privatno)
- Sadržanje - “part of” hijerarhija
  - » po vrednosti/po referenci (relevantno u C++, ali ne u Javi)
  - » agregacija/kompozicija

Tipizacija i polimorfizam

- **Tipizacija** je osobina da se objekti različitih klasa ne mogu uopšte ili se mogu zamenjivati na ograničene načine
  - » stroga i slaba tipizacija
  - » statička i dinamička tipizacija (vezivanje)
- Dinamička tipizacija i dinamičko vezivanje
  - » tehnički preduslov za ispoljavanje polimorfizma
- **Polimorfizam** je osobina da se objekat kojem se pristupa kao objektu osnovne klase ponaša različito:
  - » kao objekat osnovne klase ili kao objekat izvedene klase
  - » ponašanje zavisi od dinamičkog tipa objekta, ne statičkog tipa reference
- Polimorfizam objekta se zasniva na virtuelnim metodima, odnosno dinamičkom vezivanju

Konkurentnost i perzistencija

- Principi koji se dobro uklapaju u OO paradigmu
- Nisu suštinski principi koji određuju da li je softver OO
  - » OO softver ih ne mora posedovati
  - » softver koji nije OO ih može posedovati
- **Konkurentnost** je osobina koja razlikuje aktivne objekte od pasivnih
  - » proces - ima vlastiti adresni prostor (tipično njime upravlja OS)
  - » nit - deli isti adresni prostor sa drugim nitima
- **Perzistencija** je osobina po kojoj se postojanje objekta proteže
  - » kroz vreme (obj. nastavlja da živi nakon nestanka njegovog stvaraoca)
  - » kroz prostor (obj. se premešta iz adresnog prostora u kojem je stvoren)

Model i modeliranje

- Model je pojednostavljenje realnosti
- Model nekog sistema je apstrakcija tog realnog sistema iz određenog ugla posmatranja
- Osnovna namena modela je da se sistem koji se razvija bolje razume
- Modeliranje je važnije što je sistem kompleksniji, kompleksnost je odlika današnjih SW sistema
- Savremena metodologija razvoja softvera – Model Driven Development (MDD)

Ciljevi modeliranja

- Model pomaže da se sistem vizuelizuje
- Model omogućava da se specificira struktura sistema i ponašanje sistema
- Model daje šablon koji usmerava konstrukciju sistema
- Model dokumentuje projektne odluke koje se donose
- Model smanjuje cenu razvoja – omogućava ispitivanje projektnih odluka po nižoj ceni

OO model i pogledi na model

- Model OO analize i projektovanja obuhvata više pogleda na sistem koji se razvija
- Dve dimenzije pogleda na sistem:
  - » logički/fizički aspekti
  - » statički/dinamički aspekti
- AiP OO sistema se najčešće obavlja u terminima klasa, objekata, njihovih relacija i interakcija
- Tokom AiP koriste se različiti uglovi gledanja na model sistema u datom 2D prostoru

Dijagrami

- Za svaki pogled na model sistema može se definisati adekvatan dijagram
- Svaki dijagram predstavlja jednu projekciju modela
- Primer - aplikacija sa 100 klasa: potrebno je više klasnih dijagrama (svaki prikazuje jedan pogled na model)
- Jedno ime na svakom dijagramu označava isti entitet (sa izuzetkom operacija zbog preklapanja imena)

Logički i fizički aspekti modela

- **Logički model** sistema opisuje ključne apstrakcije (klase) i definiše:
    - strukturu klasa (atribute i operacije)
    - relacije između klasa ili komponenata sistema
    - interakcije između uloga (prototipskih objekata i aktera)
- **Fizički model** sistema:
    - opisuje konkretnu softversku i hardversku kompoziciju
    - definiše arhitekturu modula i arhitekturu procesa

Statički i dinamički aspekti modela

- **Statički aspekti** modela se fokusiraju na strukturu sistema
- **Dinamički aspekti** modela se fokusiraju na ponašanje sistema
- Realni sistemi uvek imaju dinamičko ponašanje:
    - objekti se kreiraju i uništavaju
    - objekti šalju poruke drugim objektima nekim redosledom
    - spoljašnji događaji izazivaju reakcije izvesnih objekata

Notacija za opis modela

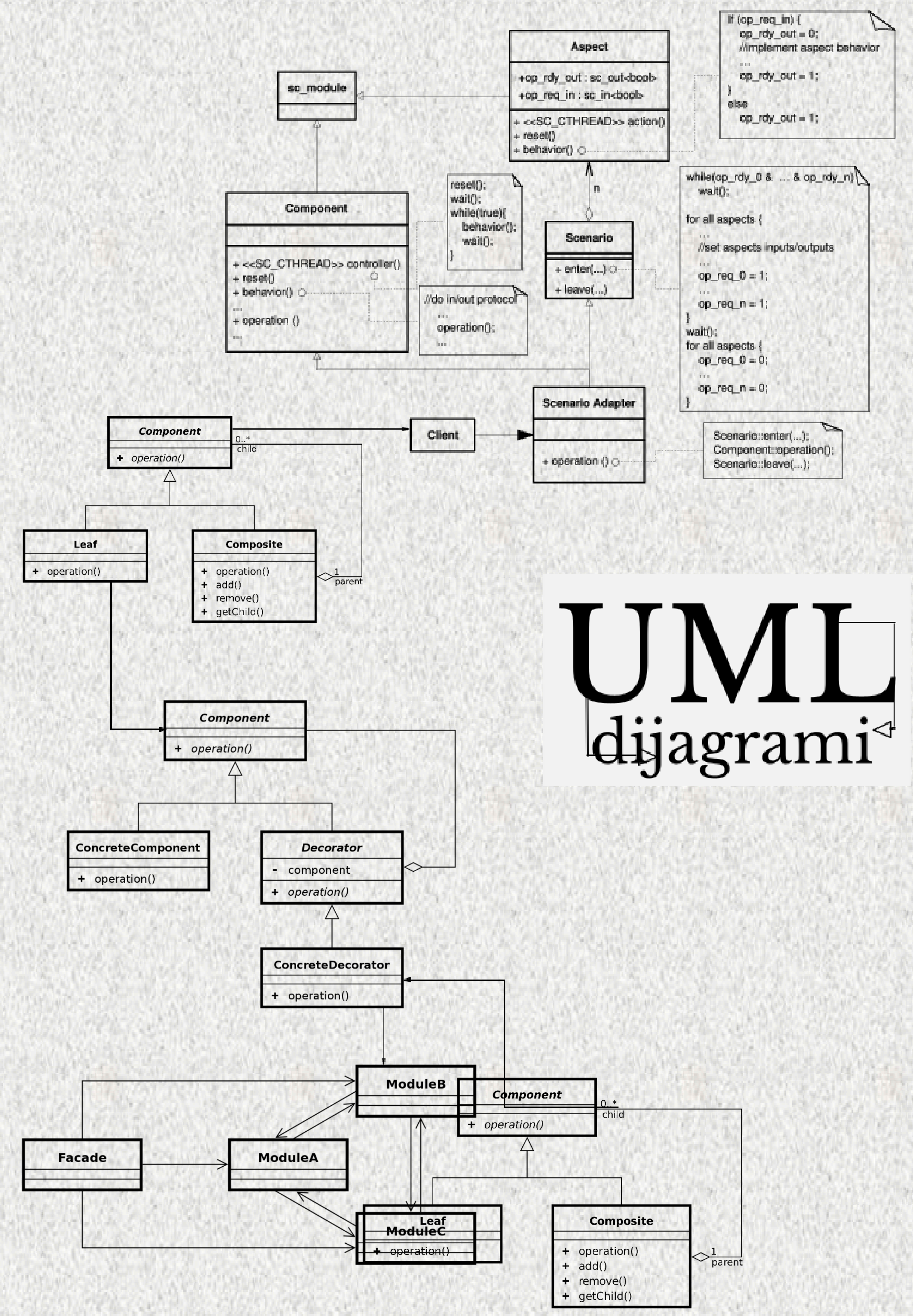
- Nekoliko notacija zaslužuju posebnu pažnju:
    - Booch i OMT notacija (iz istorijskih razloga)
    - UML notacija (standard)
- Pogodnosti standardne formalne grafičke notacije:
    - olakšava se komunikacija između:
        - korisnika i sistem-analitičara
        - članova razvojnog tima
    - projektant se rasterećuje od nebitnih detalja i koncentriše se na bitne
    - omogućava se razvoj i upotreba automatizovanih alata za :
        - proveru konzistencije i korektnosti modela
        - izvršavanje modela

Upotreba notacije

- Nije neophodno koristiti celu notaciju
- Primeri: Booch Lite, UML Basic (UML User Guide)
- Notacija treba da omogućava različit stepen detaljnosti
    - ponekad su potrebne samo grube skice modela
    - skice je potrebno nekad crtati i ručno, u toku razgovora
    - na kraju procesa projektovanja potreban je detaljan model
- Notacija treba da bude nezavisna od programskog jezika
    - neki elementi notacije nemaju podršku u konkretnom jeziku
    - neki elementi nekih jezka nemaju podršku u notaciji

Alati za modeliranje

- IBM Rational: Software Architect (Rose, Rose XDE Developer, Software Modeler)
- Borland: Together
- Gentleware: Poseidon for UML (?)
- Open Source: StarUML
- Altova: Umodel
- Omondo: EclipseUML
- Sparx Systems: Enterprise Architect
- Visual Paradigm: Visual Paradigm for UML



UML
dijagrami
sc_module
Aspect
+op_rdy_out : sc_out<bool>
+op_req_in : sc_in<bool>
+ <<SC_CTHREAD>> action()
+ reset()
+ behavior()
if (op_req_in) {
op_rdy_out = 0;
//implement aspect behavior
...
op_rdy_out = 1;
}
else
op_rdy_out = 1;
Component
+ <<SC_CTHREAD>> controller()
+ reset()
+ behavior()
...
+ operation ()
...
reset();
wait();
while(true){
behavior();
wait();
}
//do in/out protocol
...
operation();
...
n
Scenario
+ enter(...)
+ leave(...)
while(op_rdy_0 & ... & op_rdy_n)
wait();
for all aspects {
...
//set aspects inputs/outputs
...
op_req_0 = 1;
...
op_req_n = 1;
}
wait();
for all aspects {
op_req_0 = 0;
...
op_req_n = 0;
}
Scenario Adapter
+ operation ()
Scenario::enter(...);
Component::operation();
Scenario::leave(...);
Client
Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent
Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()
ModuleB
Component
+ operation()
0..*
child
Facade
ModuleA
Leaf
ModuleC
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

#

Standardni jezik za modeliranje UML

- UML (Unified Modeling Language) je grafički jezik za vizuelizaciju, specifikaciju, konstruisanje i dokumentovanje softverski-intenzivnih sistema
- UML omogućava konstruisanje šema koje modeliraju sistem opisujući:
  - » konceptualne stvari
    - — npr. proces poslovanja i funkcije sistema
  - » konkretne stvari
    - — npr. klasne tipove, šeme baza podataka, softverske komponente

Korisnici UML-a

- Sledeće kategorije korisnika UML-a se uočavaju:
  - » **sistem-analitičari** i **krajnji korisnici**: specificiraju zahtevanu strukturu i ponašanje sistema
  - » **arhitekti**: projektuju sistem koji zadovoljava zahteve
  - » **razvojni inženjeri** (developers): transformišu arhitekturu u izvršni kod
  - » **kontrolori kvaliteta** (quality assurance personel): proveravaju strukturu i ponašanje sistema
  - » **bibliotekari** (librarians): kreiraju i katalogiziraju komponente
  - » **rukovodioci projekata** (managers): vode i usmeravaju kadrove i upravljaju resursima

Praistorija UML-a

- Jezici za OO modeliranje se pojavljuju još od sredine 70ih
  - » uzrok njihove pojave je pojava nove generacije OO jezika i povećana kompleksnost softverskih sistema
- U periodu 1989-1994: broj OO metoda je porastao sa manje od 10 na više od 50
- Metode koje su ostvarile najveći uticaj na oblast OO modeliranja su:
  - » Booch metoda
  - » OMT (Object Modeling Technique, Rumbaugh)
  - » OOSE (Object Oriented Software Engineering, Jacobson)
  - » Fusion
  - » Shlaer-Mellor
  - » Coad-Yourdon

Istorija UML-a

- 1994: početak rada na UML-u - Rumbaugh se pridružio Booch-u u firmi Rational
- Oktobar 1995: pojavila se verzija 0.8 drafta UM-a (Unified Method)
- Jesen 1995: Jacobson se pridružio Rational-u – rad na objedinjenju UM sa OOSE
- Jun 1996: pojavila se verzija 0.9 UML-a
- Važniji partneri (učestvovali u definisanju verzije 1.0): DEC, HP, IBM, Microsoft, Oracle, Rational, TI, ...
- U januaru 1997: OMG-u (Object Management Group) podnet predlog std. UML 1.0
- Grupa partnera je proširena drugim podnosiocima predloga npr. ObjecTime, Ericsson,...
- Jul 1997: podnet predlog UML 1.1 koji je prihvaćen od OMG 14.11.1997.
- Jun 1998: verzija UML 1.2,   2000: verzija UML 1.3
- 2001: v1.4 (v. 1.4.2 => ISO std. 19501:2005),   2003: v1.5 (akciona semantika)
- 2005: v 2.0, 2007: v 2.1, 2009: v 2.2, 2010: v 2.3
- 2011.: v2.4 (v. 2.4.1 => ISO std. 19505:2012), 2013.: v2.5
- 2017.: 2.5.1

Konceptualni model UML-a

- Elementi UML-a su:
  - » Osnovni gradivni blokovi
  - » Pravila za povezivanje gradivnih blokova
  - » Opšti mehanizmi koji se primenjuju u UML-u
- Gradivni blokovi UML-a
  - » **Stvari** (things) - apstrakcije koje su "građani prvog reda" u modelu
  - » **Relacije** (relationships) - povezuju stvari
  - » **Dijagrami** (diagrams) - grupišu interesantne skupove povezanih stvari

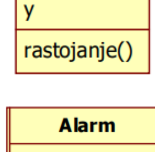
Stvari

- **Stvari strukture** - statički delovi modela, reprezentuju logičke ili fizičke elemente (imenice)
- **Stvari ponašanja** - dinamički delovi modela, reprezentuju ponašanje kroz prostor i vreme (glagoli)
- **Stvari grupisanja** - organizacioni delovi modela, kutije u koje model može biti dekomponovan
- **Stvari anotacije** - objašnjavajući delovi modela, komentari koji se primenjuju na bilo koji element

Stvari strukture – klasa

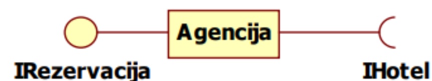
- **Klasa** je opis skupa objekata koji dele zajedničke karakteristike (atribute i operacije), ograničenja i semantiku

- **Aktivna klasa** je klasa čiji objekti imaju vlastitu nit kontrole i tako mogu da započnu neku upravljačku aktivnost
  - » ponašanje objekta aktivne klase je konkurentno sa drugim aktivnim objektima



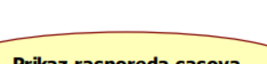
Stvari strukture – interfejs

- **Interfejs** je skup operacija koje specificiraju uslugu klase ili komponente
  - » opisuje ponašanje elementa koje je spolja vidljivo (ugovor)
  - » interfejs definiše skup deklaracija (prototipova) operacija, ali ne i njihove implementacije
  - » klasa i komponenta mogu da implementiraju više interfejsa
  - » razlikuju se implementirani i zahtevani interfejs



Stvari strukture – slučaj korišćenja

- **Slučaj korišćenja** (use-case) je opis skupa sekvenci akcija koje obavlja sistem da bi proizveo vidljiv rezultat vredan za pojedinog aktera
- Sekvenca akcija
  - » primerak slučaja korišćenja (scenario)
  - » opisuje ponašanje



- Slučaj korišćenja
  - » reprezentuje funkcionalnost sistema
  - » koirsti se da bi se strukturirale stvari ponašanja u modelu
  - » realizuje se kroz saradnju (kolaboraciju)

Stvari strukture – saradnja

- **Saradnja** (collaboration) opisuje strukturu skupa uloga koje imaju specifične funkcije da bi zajedno ostvarile ciljnu funkcionalnost
  - » ima strukturalnu, kao i dimenziju ponašanja - ponašanje se opisuje posebnim dijagramima
  - » klasa ili objekat može da učestvuje u više saradnji
- Projektni uzorci (design patterns)
  - » predstavljaju se kao saradnje uloga

Posmatrac

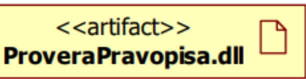
Stvari strukture – komponenta

- **Komponenta** je modularni deo sistema koji kapsulira neki sadržaj
  - » ostvaruje realizaciju skupa interfejsa i sakriva implementaciju
  - » ima ponuđene (realizovane) i zahtevane interfejse
  - » implementira operacije ponuđenog interfejsa
  - » nema atribute
  - » može se zameniti drugom koja realizuje iste interfejse
  - » ista komponenta se može koristiti u raznim sistemima
  - » u UML-u 1 – fizička stvar, ali u UML-u 2 – logička

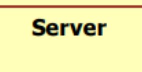
<<component>>
ProveraPravopisa

Stvari strukture – artefakt i čvor

- **Artefakt** je fizički i zamenljivi deo sistema koji sadrži informacije
  - » predstavlja fizičku manifestaciju elementa modela (npr. manifestaciju komponente)
  - » može biti fajl sa izvornim ili izvršnim kodom, dokument i sl.



- **Čvor** (node) je fizička stvar koja postoji u vreme izvršenja i reprezentuje resurs obrade
  - » svakako poseduje neku memoriju
  - » često poseduje i mogućnost procesiranja



- Skup artefakata može biti u čvoru, a može i migrirati sa čvora na čvor
- Artefakt i čvor reprezentuju fizičke stvari

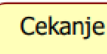
Stvari ponašanja – interakcija

- **Interakcija** je ponašanje koje specificira sekvence poruka koje se razmenjuju između skupa uloga unutar posebnog konteksta da se ostvari specifična svrha
- Interakcija uključuje određen broj elemenata:
  - » poruke (priložen grafički simbol)
  - » sekvence akcija (ponašanje izazvano porukom)
  - » konektori (komunikacione putanje između objekata)

prikazi()

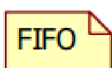
Stvari ponašanja – automat stanja

- **Automat stanja** je ponašanje koje specificira sekvence stanja kroz koje prolazi jedan objekat ili jedna interakcija za vreme svog životnog veka, sa prelazima kao posledicama događaja, zajedno sa odgovorima na te događaje
- Automat stanja uključuje određen broj elemenata:
  - » stanja (priložen grafički simbol)
  - » tranzicije (prelaze između stanja)
  - » događaje (stvari koje izazivaju tranziciju)
  - » akcije (odgovore na tranzicije)



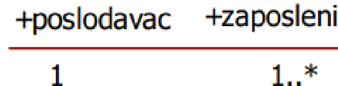
Stvari organizacije i anotacije

- **Paket** je opštenamenski mehanizam za organizovanje elemenata u grupe
- Stvari strukture, ponašanja i druge stvari grupisanja mogu biti smeštene u paket
- Za razliku od komponente, paket postoji samo u vreme razvoja
- Pored paketa postoje i sledeće stvari grupisanja: radni okviri (frameworks), modeli
- **Napomena** (note) je simbol za prikazivanje komentara pridruženih jednom elementu ili kolekciji elemenata



Relacije

- **Zavisnost** je semantička relacija između dve stvari u kojoj izmena jedne (nezavisne) stvari može uticati na semantiku druge (zavisne) stvari
- **Asocijacija** je strukturna relacija koja opisuje skup veza između objekata
  - » sadržanje je specijalna vrsta asocijacije koja reprezentuje strukturnu relaciju između celine i njenih delova
  - » često grafički simbol sadrži ukrase kao što su multiplikativnost i imena uloga



- **Generalizacija** je relacija u kojoj su objekti specijalizovanog elementa (deca) zamene za objekte generalizovanog elementa (roditelja) – dete deli strukturu i ponašanje roditelja

- **Realizacija** je semantička relacija između stvari u kojoj jedna stvar propisuje ugovor a druga stvar ostvaruje taj ugovor

- Realizacija se sreće:
  - » između interfejsa i klase koja ga realizuje
  - » između interfejsa i komponente koja ga realizuje
  - » između slučajeva korišćenja i saradnji koje ih realizuju

Dijagrami

- **Dijagram** je grafička reprezentacija skupa povezanih elemenata
  - » najčešće se pojavljuje u obliku grafa temena (stvari) povezanih granama (relacijama)
- Dijagrami se crtaju da bi se sistem vizualizovao iz različitih perspektiva
- Vrste dijagrama u UML-u:
  - » dijagrami za prikaz strukturnih aspekata sistema
    - — strukturni aspekti odgovaraju statičkim aspektima
  - » dijagrami za prikaz aspekata ponašanja sistema
    - — aspekti ponašanja (uglavnom) odgovaraju dinamičkim aspektima

Dijagrami strukture

- **Dijagram klasa** (class diagram) prikazuje logičku strukturu apstrakcija: skup klasa, interfejsa, saradnji i njihovih relacija
- **Dijagram paketa** (package diagram) [UML 2] prikazuje statičku strukturu grupisanja elemenata modela u pakete
- **Dijagram objekata** (object diagram) prikazuje logičku strukturu primeraka: skup objekata (primeraka klasa) i njihovih veza
- **Dijagram složene strukture** (composite structure diagram) [UML 2] prikazuje hijerarhijsko razlaganje primerka klase, komponente ili saradnje na delove
- **Dijagram komponenata** (component diagram) prikazuje komponente, njihovu unutrašnju strukturu i zavisnosti između skupa komponenata
- **Dijagram raspoređivanja** (deployment diagram) prikazuje konfiguraciju čvorova obrade i artefakata koji se raspoređuju na njih

Dijagrami ponašanja

- **Dijagram slučajeva korišćenja** (use case diagram) prikazuje skup slučajeva korišćenja, aktera (specijalne vrste klasa) i njihovih relacija
- **Dijagram interakcije** (interaction diagram) prikazuje interakciju koju čine skup uloga i njihovih veza sa porukama koje razmenjuju
    - **Dijagram sekvence (**sequence diagram) je dijagram interakcije koji naglašava vremenski redosled poruka
    - **Dijagram komunikacije** (communication diagram) je dijagram interakcije koji naglašava strukturnu organizaciju povezanih uloga koje razmenjuju poruke;
    - **Dijagram pregleda interakcije** (interaction overview diagram) [UML 2] je dijagram interakcije koji definiše interakcije kroz vrstu dijagrama aktivnosti
    - **Vremenski dijagram** (timing diagram) [UML 2] je dijagram interakcije koji prikazuje promenu stanja uloge u vremenu
- **Dijagram stanja** (statechart diagram) prikazuje konačni automat koji obuhvata stanja, tranzicije, događaje i aktivnosti
- **Dijagram aktivnosti** (activity diagram) prikazuje tok između aktivnosti u sistemu (nije specijalna vrsta dijagrama stanja u UML 2)

Pravila UML-a

- UML ima pravila koja specificiraju kako izgleda dobro formiran model
- Dobro formiran model je semantički konzistentan i u harmoniji sa korelisanim modelima
- UML ima semantička pravila za:
    - imena - kako se nazivaju stvari, relacije i dijagrami
    - doseg - koji kontekst daje specifično značenje imenu
    - vidljivost - gde se imena mogu videti i koristiti od strane drugih
    - integritet - kako se stvari propisno i konzistentno korelišu prema drugim stvarima
    - izvršenje - šta nešto znači za izvršenje ili simulaciju dinamičkog modela
- Tokom razvoja se ne prave samo modeli koji su dobro formirani, već mogu biti i:
    - skraćeni (elided) - izvesni elementi su sakriveni da se pojednostavi izgled
    - nekompletni - izvesni elementi nedostaju
    - nekonzistentni - integritet modela nije garantovan
- Pravila UML-a vode kroz vreme ovakve modele prema dobro formiranim

Opšti mehanizmi UML-a

- Gradnja je jednostavnija i harmoničnija ako se poštuju opšti obrasci
- Postoje četiri opšta mehanizma koja se primenjuju konzistentno kroz jezik:
  - » specifikacije
  - » ukrasi
  - » opšte podele
  - » mehanizmi proširivosti

Specifikacije

- Iza elementa grafičke notacije UML-a leži specifikacija – tekstualni iskaz sintakse i semantike gradivnog bloka
- Iza grafičkog simbola (sličice, ikone) klase stoji specifikacija koja navodi:
  - » potpun skup atributa
  - » potpun skup operacija (uključujući kompletne deklaracije)
  - » odgovornosti klase
- Grafički simbol može prikazivati samo jedan deo potpune specifikacije
  - » alati podržavaju suspendovanje pojedinih detalja
- Može postojati i drugi izgled iste klase
  - » prikazuje drugi skup delova iste klase
  - » konzistentan je sa specifikacijom klase
- UML grafička notacija se koristi za vizuelizaciju
- UML specifikacija se koristi da se saopšte detalji
- Modeli se mogu graditi:
  - » najpre pomoću crtanja dijagrama, a zatim dodavanjem semantike u specifikaciju
    - — tipično za **direktni inženjering** pri kreiranju novog sistema
  - » direktnim kreiranjem specifikacije, pa naknadnim kreiranjem dijagrama koji su njene projekcije
    - — tipično za **reverzni inženjering** postojećeg sistema

Ukrasi

- Detalji specifikacije se prikazuju kao stilski, simbološki, grafički ili tekstualni ukras osnovnog grafičkog elementa
- Na primer:
  - » za klasu se može naglasiti da je apstraktna tako što se ime piše italic slovima
  - » vidljivost (pravo pristupa) atributa i operacija se može naglasiti pomoću simbola: + (javni), # (zaštićeni), – (privatni) i ~(paketni)
  - » agregacija se predstavlja dodatnim simbolom na simbolu asocijacije

Opšte podele

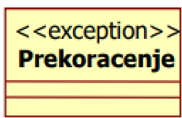
- Dve osnovne podele:
  - » apstrakcije i primerci (instance)
  - » interfejsi i implementacije (interfejs=ugovor, implementacija=realizacija ugovora)
- Primeri prve podele:
  - » klase/objekti (klasa je apstrakcija, a objekat primerak te apstrakcije)
  - » slučajevi korišćenja/primerci slučajeva korišćenja (scenarija)
  - » čvorovi/primerci čvorova
- Primeri druge podele:
  - » interfejsi/komponente, slučajevi korišćenja/saradnje, operacije/metodi
- U UML-u se razlika između apstrakcije i primerka pravi tako što se imena primeraka podvlače
- Relacijom realizacije se uspostavlja veza između ugovora i implementacije

Mehanizmi proširivosti

- UML je otvoren za proširenja jezika na kontrolisani način
- Mehanizmi proširivosti uključuju:
    - » stereotipe
    - » obeležene vrednosti
    - » ograničenja

Stereotipi

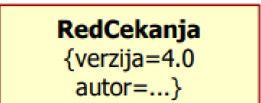
- Stereotip proširuje rečnik UML-a dopuštajući kreiranje novih vrsta gradivnih blokova specifičnih za problem
- Novi gradivni blokovi su izvedeni iz postojećih
- Stereotip se prikazuje kao ime uokvireno znacima << i >> smešteno iznad imena odgovarajućeg elementa
- Na primer, izuzeci su klase čiji se objekti mogu bacati i hvatati



- Može se definisati i grafički simbol za određeni stereotip

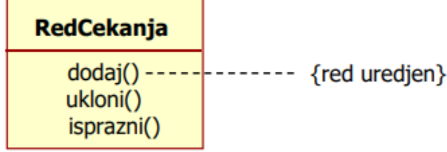
Obeležene vrednosti

- Obeležene vrednosti proširuju osobine UML gradivnog bloka dopuštajući dodavanje nove informacije
- Obeležene vrednosti se prikazuju kao niska okružena zagradama { i } ispod imena odgovarajućeg elementa
- Niska sadrži ime (tag), separator (simbol =) i vrednost
- Na primer, verzija i autor klase nisu primitivni koncepti u UML-u, a mogu se dodati bilo kom gradivnom bloku kao što je klasa



Ograničenja

- Ograničenja proširuju semantiku UML gradivnog bloka dopuštajući da se dodaju nova pravila ili promene postojeća
- Ograničenja se mogu pisati:
    - » kao slobodan tekst
    - » na jeziku OCL (Object Constraint Language)



2 Primer modela za program

Uvod

- Najbolji način učenja UML-a je kroz kreiranje modela na UML-u
- Većina programera kada uči novi jezik prvo napiše program koji ispiše "Pozdrav svima!"
- Početak učenja modeliranja na jeziku UML je model za program koji ispiše "Pozdrav svima!" uz mehanizme Jave koji omogućavaju izvršenje

Primer na Javi

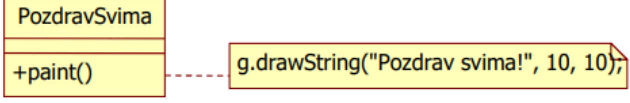
- Trivijalni aplet koji ispisuje "Pozdrav svima!":

```
import java.awt.Graphics;
class PozdravSvima extends
java.applet.Applet{
   public void paint(Graphics g){
     g.drawString("Pozdrav svima!", 10, 10);
   }
}
```

- Iako je primer programa trivijalan, infrastruktura potrebna da bi aplet radio nije trivijalna

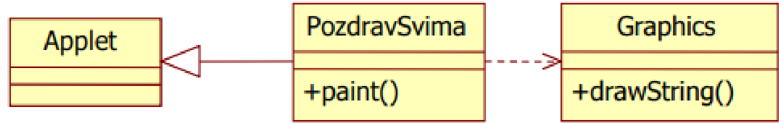
Ključne apstrakcije

- Osnovni klasni dijagram:



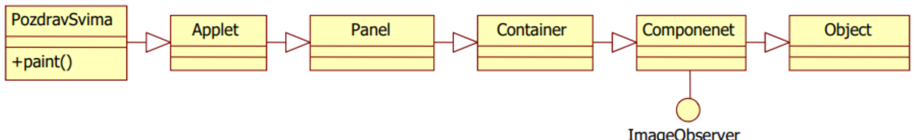
- Klasa PozdravSvima ima jednan metod (operaciju) paint()
  - » redefinisani metod klase Component koji "iscrtava" datu komponentu na željeni način
  - » metod se poziva iz okruženja (ne poziva ga programer) i to inicijalno, pri pomeranju, otkrivanju, ili promeni veličine komponente
- Komentar (note) kaže šta radi metod paint() – alat ne dozvoljava, ali bi isprekidana linija trebalo da ide do metoda

- Osnovne relacije:



- Klasa PozdravSvima se izvodi iz klase Applet (specijalizacija), a koristi klasu Graphics (zavisnost)
  - » klasa Graphics se pojavljuje kao tip parametra (formalnog arg.) (redefinisanog) metoda paint() u klasi PozdravSvima
  - » metodi klase Graphics omogućavaju crtanje i pisanje na komponentama (grafički kontekst)
  - » crtanje i pisanje se obavlja korišćenjem tekućih atributa datog Graphics objekta

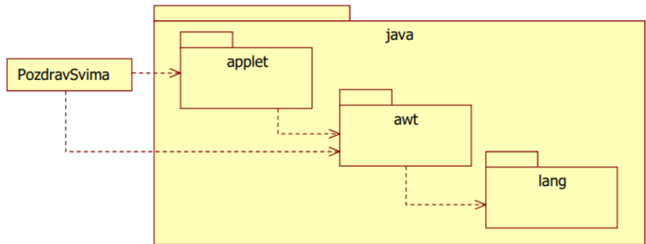
- Hijerarhija nasleđivanja:



- PozdravSvima je izveden iz klase Applet, ova iz klase Panel, Panel iz Container, ova iz Component, a ova iz Object
- ImageObserver je interfejs preko kojeg se primaju obaveštenja o konstrukciji slike
    - » interfejs sadrži (callback) metod imageUpdate() preko kojeg se javlja progres/status konstrukcije slike

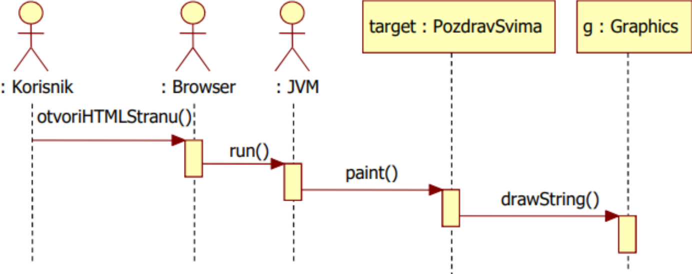
Grupisanje

- Dijagram paketa – odnosi (zavisnosti) između paketa
- Ovde paketi grupišu bibliotečke klase
- PozdravSvima zavisi direktno od applet i awt paketa



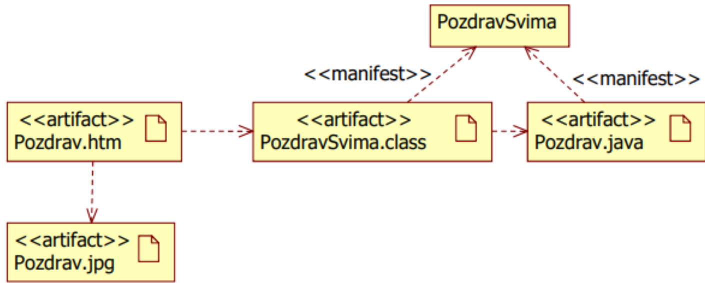
Ponašanje

- Dijagram sekvence (originalni primer je nešto kompleksniji):



Artefakti

- Dijagram artefakata (standard ga ne navodi):



- Artefakti izvornog i izvršnog koda su manifestacije klase PozdravSvima

3 Dijagrami klasa

Uvod

- Dijagram klasa prikazuje skup klasa, interfejsa, saradnji i drugih stvari strukture, povezanih relacijama
- Dijagram klasa je graf obrazovan od temena (stvari) povezanih granama (relacijama)
- Specificira logičke i statičke aspekte modela
- Elementi dijagrama klasa
  - » stvari: klase, interfejsi, tipovi, izuzeci, šabloni, saradnje, paketi, …
  - » relacije: zavisnosti, generalizacije, asocijacije, realizacije
- Dijagrami klasa su najčešći u objektnom modeliranju
- Većina alata podržava generisanje skeleta koda iz dijagrama klasa

Klasifikator

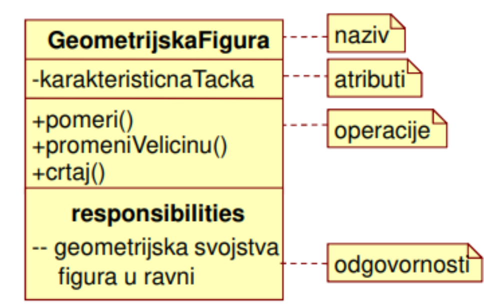
- Klasifikator je
  - » karakterizacija primeraka (instances) sa zajedničkim karakteristikama
  - » mehanizam koji opisuje strukturu i ponašanje (ima atribute i operacije)
  - » apstraktna metaklasa
  - » prostor imena
  - » tip koji je moguća generalizacija ili specijalizacija drugog
  - » potencijalno ugnežđen u drugi klasifikator
- Ne pojavljuju se svi klasifikatori na dijagramima klasa
- Konkretni klasifikatori su: klase, interfejsi, tipovi podataka, komponente, čvorovi, slučajevi korišćenja, ...
- Klasifikator ima odeljke, koji se ne moraju svi prikazati
  - » odeljci mogu imati svoje nazive, da bi se izbegla dvoznačnost kada neki odeljak nedostaje

Klasa

- UML 2 definicija: klasa je opis skupa objekata koji dele istu specifikaciju karakteristika (features), ograničenja (constraints) i semantike – karakteristike: atributi i operacije
- Klasa implementira jedan ili više interfejsa
- Klase opisuju apstrakcije iz domena problema i rešenja
- Koriste se da reprezentuju
  - » konkretne softverske stvari
  - » konceptualne stvari

Simbol klase

- Simbol klase je pravougaonik podeljen horizontalnim linijama u odeljke – odeljak može imati i naziv odeljka (npr. responibilities)
- Naziv klase
  - » jednostavan: <naziv>
  - » sa putanjom: <naziv paketa>:: <jednostavan naziv> npr: java::awt::Rectangle
- Odgovornosti klase
  - » neformalna semantika – dokumentacija
  - » u zasebnom odeljku (responibilities)
  - » slobodan tekst
  - » svaka počinje sa --
- Svaka dobro strukturirana klasa bi trebalo da ima barem jednu i ne više od nekoliko odgovornosti



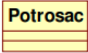
Atributi

- **Atributi** su imenovana svojstva klase koja opisuju opsege vrednosti koje pojave tog svojstva mogu sadržati
- Drugi nazivi: članovi podaci (C++), polja (Java)
- Atributi su strukturne karakteristike klase
- Notacija: ime, opciono sa tipom i podrazumevanom vrednošću
- Primer: `izbor:Boolean=false`

Operacije

- **Operacije** su servisi klase koji se mogu zahtevati od nekog objekta klase
- Operacije su karakteristike ponašanja klase
- Operacije su deklaracije metoda klase
- Notacija: potpis koji sadrži listu parametara sa eventualnim tipovima i podrazumevanim vrednostima, kao i tipom rezultata
- Primer: `pomeri(novaPozicija:Pozicija=koordinatniPocetak):Pozicija`

Opcije odeljaka



- Simbol klase može sadržati prazne odeljke
  - prazan odeljak atributa/operacija ne znači da ih klasa nema, već da nisu relevantni za dati pogled (dijagram)
- Simbol klase može biti i bez odeljaka
- Ako odeljak ima članova, mogu se koristiti i tri tačke (...) da naznače postojanje dodatnih atributa/operacija
- Atributi/operacije se mogu grupisati, a svakoj grupi može da prethodi opisni prefiks
  - naziv kategorije grupisanih članova
  - piše se kao stereotip, npr. <<constructor>> ili <<query>>

Potrosac

Ukrasi klase i članova klase

- Apstraktna klasa i apstraktna operacija – naziv se piše *italikom* ili uz ograničenje {abstract}
- Zajednički članovi (atributi i operacije) - podvučeno
- Pravo pristupa članu (vidljivost, visibility) znak se piše ispred atributa/operacije:
  - javni član: + (podrazumevano)
  - zaštićeni član: #
  - paketski član: ~
  - privatni član: -

Tipovi podataka

- Tipovi podataka su tipovi čije vrednosti (podaci) nemaju identitet
- Vrste tipova podataka:
  - apstraktni tipovi podataka (<<dataType>>)
    - primerci imaju samo vrednost, bez identiteta (npr. Datum)
  - tipovi nabrajanja (<<enumeration>>)
    - uzimaju vrednost iz definisanog skupa simboličkih imena (npr. boolean)
  - primitivni tipovi (<<primitive>>)
    - postojeći prosti tipovi podataka u jeziku implementacije (npr. int)

<<dataType>>
Datum

<<enumeration>>
boolean
true
false

<<primitive>>
int
{vrednosti u opsegu: od –2**31-1 do 2**31}

Osobine klasa

- Multiplikativnost – osobina koja ograničava broj primeraka klase
- Multiplikativnost se navodi u gornjem desnom uglu
- Specifične vrednosti multiplikativnosti:
  - 0 – uslužna klasa, sadrži samo zajedničke (statičke) atribute i operacije
  - 1 – unikat (singleton) – klasa koja ograničava broj primeraka na 1
- Podrazumevani slučaj je proizvoljan broj primeraka
- Koren hijerarhije klasa (root) je klasa koja nema roditelje
- List hijerarhije klasa (leaf) je klasa koja nema potomke, tj. ona iz koje se ne može izvoditi
- Osobine {root} i {leaf} se pišu u odeljku naziva klase

Sintaksa atributa

- Sintaksa atrubuta:
  ```
  [pravo_pristupa] [/] ime [:tip] [multiplikativnost]
  [=inicijalna_vrednost][{osobine}]
  ```
- Simbol / označava da je atribut "izveden" (može se "izračunati" na osnovu drugih)
- Multiplikativnost atributa (podrazumevano 1), navodi se u zagradama [ ]
  - specificira se kao interval celih brojeva, gornja granica može biti neograničena (*)
  - primer: consolePort:Port[2..*]
- Osobine atributa se navode razdvojene zarezima; neke od njih:
  - readOnly – vrednost se ne može menjati, dodavati ni uklanjati nakon inicijalizovanja
  - ordered – vrednosti elemenata su uređene (podrazumevano unordered)
  - unique – vrednosti elemenata (u jednom objektu zbirke) su jedinstvene (skup) (podrazumevano unique), dakle {unordered , unique} – skup
  - bag – isto kao unordered nonunique (nije skup, nije uređena zbirka)
  - seq ili sequence – isto kao ordered nonunique (uređen niz elemenata sa ponavljanjem)
  - composite – atribut složene strukture
  - redefines ime – redefiniše nasleđen atribut ime
  - subsets ime – podskup skupa vrednosti atributa ime

Sintaksa operacije

- Sintaksa operacije:
  ```
  [pravo_pristupa] ime ([lista_parametara])
  [: tip_rezultata] [{osobina}]
  ```
- Sintaksa parametra u listi:
  ```
  [smer] ime : tip [multiplikativnost]
  [ = podrazumevana_vrednost] [{osobina}]
  ```
- Smer može biti:
  - in – ulazni parametar, ne sme biti modifikovan (podrazumevano)
  - out – izlazni parametar, mora se inicijalizovati u metodu
  - inout – ulazno-izlazni parametar, inicijalizovan pre poziva, može se modifikovati
  - return – vrednost parametra se vraća kao rezultat operacije pozivaocu
- Osobine operacije:
  - query – izvršenje ne menja stanje objekta, operacija je čista funkcija bez bočnih efekata
  - exception lista – operacija može da baca izuzetke iz liste
  - leaf – operacija nije polimorfna i ne može se redefinisati u izvedenoj klasi
  - concurrency = vrednost – šta garantuje operacija pri izvršenju u konkurentnoj sredini
    - sequential – pozivaoci moraju obezbediti da je samo jedna nit u jednom trenutku poziva
    - guarded – operacija garantuje sekvencijalizaciju svih poziva svih štićenih operacija
    - concurrent – operacija se izvršava kao atomska

Aktivne klase

- Aktivni objekat je onaj koji poseduje vlastiti tok kontrole koji se odvija simultano sa drugim tokovima kontrole
- Simultano – konkurento (isti resurs) ili paralelno (posebni resursi)
- Aktivna klasa je klasa čiji su primerci aktivni objekti
- Aktivni objekti su “koreni” pojedinih tokova kontrole
  - » oni pozivaju svoje ili metode drugih (aktivnih ili pasivnih) objekata
- Tok kontrole se pokreće/zaustavlja:
  - » kada se aktivni objekat stvara/uništava, ili
  - » posebnim operacijama za pokretanje i zaustavljanje
- Mnogi jezici podržavaju aktivne objekte (Ada, Java, Smalltalk)
- Aktivni objekti i klase - mogu se pojaviti u dijagramima gde i pasivni

Procesi i niti

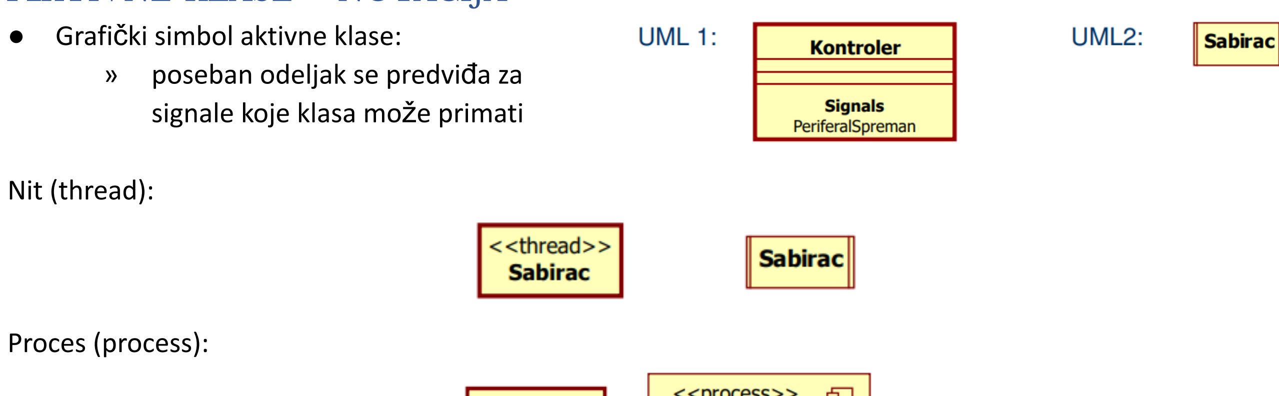
- Proces (process, task) je "teški" (heavyweight) tok kontrole – ima vlastiti adresni prostor
- Nit (thread) je "laki" (lightweight) tok kontrole – deli zajednički adresni prostor sa drugim nitima
- Jedan proces može sadržati više niti
- Proces je jedinica konkurentnosti u operativnim sistemima: procesi konkurišu za resurse
- Primeren mehanizam za komunikaciju:
  - » procesi – razmena poruka (message passing)
  - » niti – deljenje zajedničkih podataka (data sharing)

Aktivne klase - notacija

- Grafički simbol aktivne klase:
  - » poseban odeljak se predviđa za signale koje klasa može primati

Nit (thread):

Proces (process):



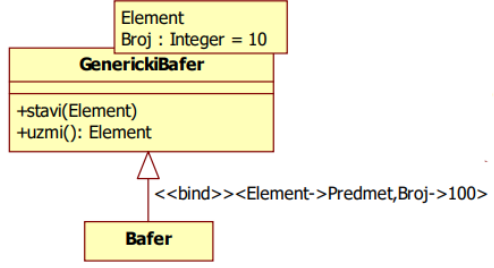
Šabloni

- Šablon (template) je parametrizovani element – C++, Ada, Java podržavaju šablone (generike)
- Primer na C++:

```
template <class Element, int Broj> class GenerickiBafer{
public:
   virtual void stavi(const Element&);
   virtual Element& uzmi();
   ...
};
typedef GenerickiBafer<Predmet,100> Bafer;       // eksplicitno generisanje
GenerickiBafer<Predmet,100> bafer;               // implicitno generisanje
```

Šabloni - notacija

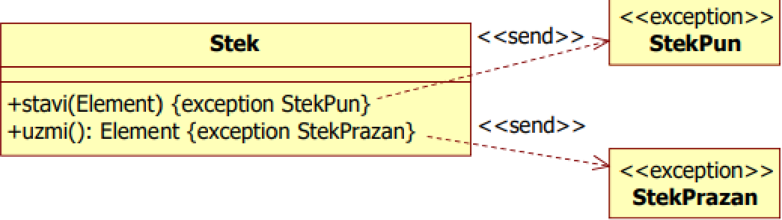
- Primer generičkog bafera:
  - » definicija šablona i eksplicitno generisanje iz njega:



  - » implicitno generisanje: **GenerickiBafer**<Element->Predmet, Broj->100>
  - » alternativno eksplicitno generisanje: **Bafer: GenerickiBafer**<Element->Predmet, Broj->100>

Izuzeci

- Izuzeci su (prema UML 1) vrste signala
- Mogu da se modeliraju nestandardnim stereotipom klase <<exception>>
- Signal šalje operacija koja izaziva izuzetak
  - » modelira se stereotipom <<send>> relacije zavisnosti
  - » samo vizuelna informacija, jer operacija već navodi kao osobinu exception
- Primer:



- UML 2 tretira izuzetke na način kako ih tretiraju savremeni OO jezici

Relacije

- Na klasnim dijagramima se pojavljuju sve četiri (prve tri su češće) vrste relacija:
  - » zavisnost (dependency)
  - » asocijacija (association)
  - » generalizacija (generalization)
  - » realizacija (realization)

Zavisnost

- Povezuje stvari kod kojih izmena nezavisne stvari utiče na ponašanje zavisne stvari
- Zavisna stvar koristi nezavisnu stvar
- Grafička notacija: klasa A zavisi od klase B

- Često se koristi kad je jedna klasa (B) tip parametra operacije druge klase (A)
- Klasa B može biti i tip rezultata operacije klase A, a može biti i tip lokalnog objekta metoda klase A

Generalizacija

- Povezuje opštije sa specijalizovanim stvarima
    - » opštija stvar – superklasa, natklasa, osnovna klasa ili roditelj
    - » specijalizovana stvar – subklasa, potklasa, izvedena klasa ili dete
- Grafička notacija: klasa A je dete, B je roditelj
- Drugi nazivi relacije:
    - » vrsta (is-a-kind-of),
    - » izvođenje (A se izvodi iz B),
    - » nasleđivanje,
    - » proširivanje
- Generalizacija znači: objekti dece se mogu pojaviti gde god se očekuje objekat roditelja
- Deca nasleđuju osobine svojih roditelja, atribute (strukturu) i operacije (ugovor)
- Operacija deteta koja ima isti potpis kao operacija roditelja redefiniše operaciju roditelja
- Redefinicija operacije omogućava njeno polimorfno ponašanje
- Klasa koja ima samo jednog roditelja koristi jednostruko nasleđivanje
- Klasa koja ima više roditelja koristi višestruko nasleđivanje

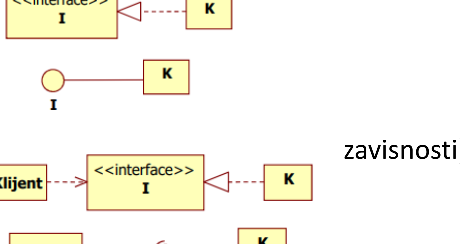
Asocijacija

- Specificira da li su primerci jedne stvari povezani sa primercima druge stvari

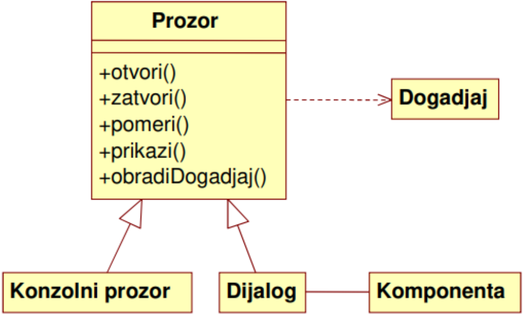
- Asocijacija je apstrakcija veza između objekata klasa u asocijaciji
- Asocijacija je strukturna relacija
- Asocijacija može biti unidirekcionalna ili bidirekcionalna
    - » jednosmerna, odnosno dvosmerna, navigabilnost
- Dozvoljena je i asocijacija sa samim sobom – postoje veze između objekata iste klase
- Uobičajeno je da asocijacija povezuje dve klase (binarna asoc.)
- Moguće je da asocijacija povezuje i više klasa (n-arna asocijacija)
- Grafička notacija: klasa A je u asocijaciji sa klasom B

Realizacija

- Semantička relacija između klasifikatora od kojih jedan specificira ugovor a drugi garantuje njegovu implementaciju
- Grafička notacija:
    - » kanonička forma:
    - » skraćena forma:
- Korišćenje interfejsa (relacija zavisnosti)
    - » kanonička forma:
    - » skraćena forma (ball & socket):

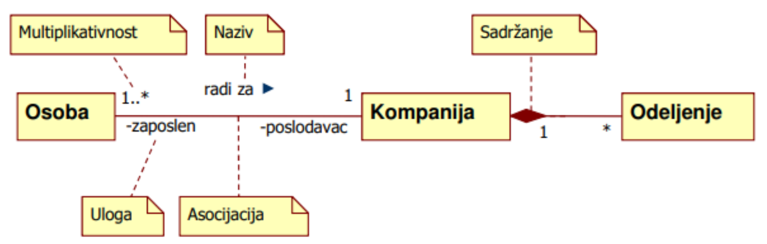


Primer osnovnih relacija



Ukrasi asocijacije

- Na asocijaciji se mogu pojaviti sledeći ukrasi:
  - » naziv
  - » smer čitanja
  - » uloge
  - » navigabilnost
  - » multiplikativnost
  - » sadržanje (agregacija ili kompozicija)
  - » pravo pristupa (vidljivost) kraju preko asocijacije
  - » vlasništvo kraja asocijacije



Multiplikativnost

- Na jednoj strani asocijacije označava se broj objekata sa te strane koji su u vezi sa tačno jednim objektom sa druge strane relacije
- Može biti:
  - » tačno n, gde n može biti proizvoljan pozitivan broj (npr. 1)
  - » proizvoljno mnogo, uključujući nijedan (*),
  - » opseg (npr. 0..1 ili 2..*)
- UML 2.0 više ne dozvoljava izraze sa diskontinuitetom – npr. 0..1,3..4,6..* – proizvoljan broj osim 2 i 5
- Ograničenja uz multiplikativnost: – {oredered}, {unique} – kao kod atributa

Agregacija

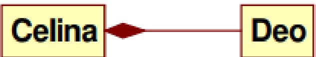
- Vrsta asocijacije kod koje jedna strana igra ulogu celine a druga ulogu dela (whole/part)
- Celina sadrži delove ("has-a" relacija)
- Agregacija najčešće označava grupisanje delova
- Agregacija ne govori ništa o uzajamnom odnosu životnih vekova celine i dela
- Deo u agregaciji može biti zajednički deo više celina
- Grafička notacija:

Kompozicija

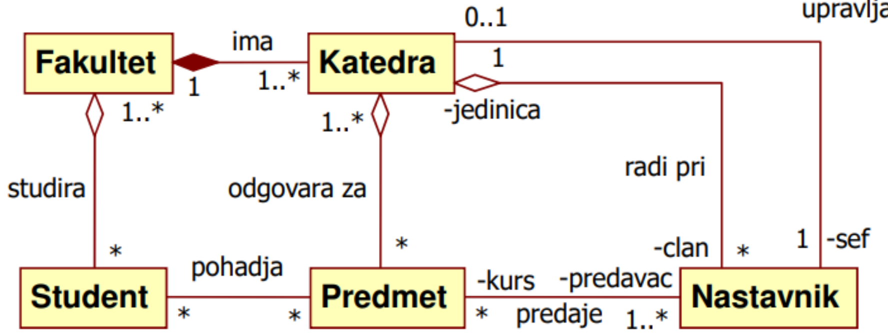
- Asocijacija kod koje postoji odnos celina/deo, ali je celina odgovorna za životni vek dela
- Sadržanje sa strogim vlasništvom
- Koincidentni životni vek dela u odnosu na celinu

» deo može nastati u toku života celine i može biti uništen pre uništenja celine

- Deo u kompoziciji može biti deo samo jedne celine
- Grafički notacija:

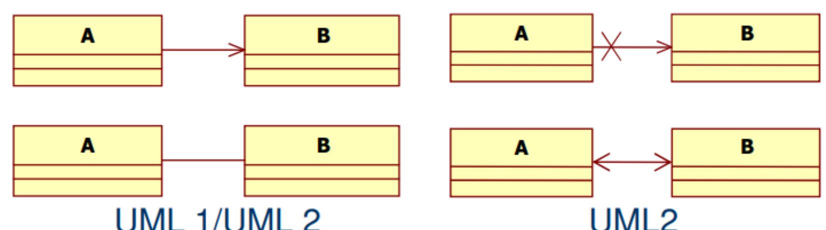


Primer asocijacija



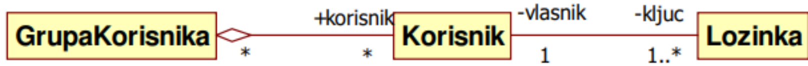
Navigabilnost

- Strelica označava navigabilnost u naznačenom pravcu
- Krstić označava da nema navigabilnosti prema označenoj strani
- Za asocijaciju bez ukrasa navigabilnosti se smatra da je navigabilnost neodređena
- Grafička notacija:



Pravo pristupa preko asocijacije

- Ograničava vidljivost (visibility) objekata klasa u asocijaciji za spoljašnji svet
- Označava se sa +, #, -, ~ ispred imena uloge odgovarajuće strane relacije
- + znači da objektima sa te strane mogu pristupati svi objekti preko objekta sa druge strane
  - znači da objektima klase sa te strane mogu pristupati samo objekti klase sa druge strane
- # znači da i objekti klasa izvedenih iz klase sa drugog kraja asocijacije imaju pristup
- ~ znači da i objekti klasa iz istog paketa kao klasa sa drugog kraja asocijacije imaju pristup
- Primer:



- Podrazumevan je javni vizibilitet uloge u asocijaciji

Ugnežđivanje

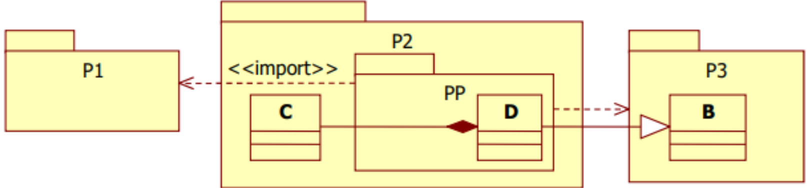
- Označava pojam kada je klasa B deklarisana u prostoru imena klase A
- Primer:

- Ugnežđenje ograničava vidljivost ugnežđene stvari na prostor imena čiji je član
- Relacija ne implicira relaciju između objekata odgovarajućih klasa

4 Dijagrami paketa

Uvod

- Dijagrami paketa se koriste da prikažu dekompoziciju modela u organizacione jedinice i njihove zavisnosti
- Dijagrami paketa pomažu:
  - » da se uoče zavisnosti između logičkih celina i
  - » da se te zavisnosti drže pod kontrolom
- Primer:



Paket

- Paket je organizaciona stvar koja se koristi za grupisanje elemenata
- Paket predstavlja prostor imena i element koji se može pakovati, tako da se može sadržati u drugim paketima
- Definicija uz UML RM:
  - » paket je opšti mehanizam za organizovanje elemenata u grupe, koji uspostavlja vlasništvo nad elementima i obezbeđuje jedinstvenost imena elemenata
- Paketi se koriste
  - » uobičajeno za grupisanje logičkih apstrakcija modela (klasa)
  - » mogu se koristiti i za grupisanje fizičkih stvari (artefakata ili čak čvorova)
- Paket može da obuhvata druge pakete i proste elemente
  - » obično postoji jedan paket u korenu hijerarhije paketa koji predstavlja ceo model
- Koncept paketa donekle odgovara
  - » istoimenom konceptu u jeziku Java
  - » konceptu prostora imena u jezicima C++ i C#

Imenovanje paketa i elemenata

- Jedno ime elementa mora biti jedinstveno u okviru paketa
  - » ali se može koristiti za označavanje drugih elemenata iz drugih paketa
- Jednoznačnost imena se odnosi na puna (kvalifikovana) imena

   » puna imena sadrže redom imena svih paketa u hijerarhiji od korenog paketa do datog elementa - lista u stablu, razdvojena simbolom ::

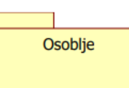
- Primer:
   » klasa Panel koju sadrži potpaket awt paketa java nosi puno ime java::awt::Panel
- Elementi kojima se unutar paketa može obraćati jednostavnim (nekvalifikovanim) imenima su:
   » sopstveni elementi paketa,
   » uvezeni elementi i
   » elementi iz obuhvatajućih (spoljašnjih) prostora imena (paketa)

Vlasništvo i pravo pristupa

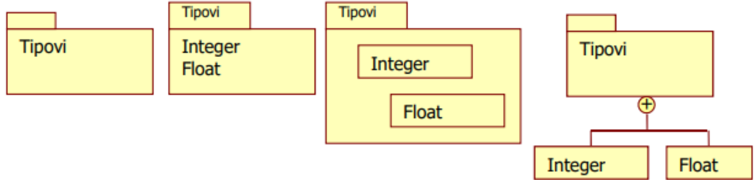
- Vlasništvo nad sopstvenim elementom implicira
   » ako se paket ukloni iz modela, uklanjaju se i sopstveni elementi paketa
   » svaki element modela mora imati kao vlasnika
      — neki paket ili
      — drugi element modela
- Sopstveni i uvezeni elementi mogu imati pravo pristupa
- Pravo pristupa određuje da li su elementi na raspolaganju izvan paketa
- Pravo pristupa elementa u paketu može biti:
   » javno (+) ili
   » privatno (-)
- Javni sadržaj paketa je uvek pristupačan izvan paketa preko punih imena
- Paket "izvozi" svoj javni sadržaj
- Za "uvoz" imena iz drugih paketa koriste se posebne relacije zavisnosti

Notacija

- Za paket se koristi simbol pravougaonika sa jezičkom – simbol sugeriše folder koji sadrži:
   » fajlove (jednostavne elemente) i
   » druge foldere (potpakete)



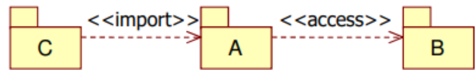
- Sadržani elementi paketa se mogu predstaviti na više načina:
   » samo tekstualno nabrojani unutar pravougaonika simbola paketa (tada se ime paketa piše unutar jezička)
   » nacrtani unutar pravougaonika simbola paketa (i tada se ime paketa piše unutar jezička)
   » povezani punom linijom sa simbolom + unutar kružića na strani paketa



Relacije

- Zavisnosti i posebne zavisnosti: <<import>>, <<access>>, <<merge>>
- Zavisnost:
   » označava da barem jedan element u zavisnom paketu na neki način zavisi od nekog elementa iz nezavisnog paketa
   » primer: ako je klasa X u paketu P1 izvedena iz klase Y iz paketa P2, paket P1 zavisi od paketa P2
- Javno uvoženje (<<import>>):

- » omogućava u paketu u koji se uvozi (na strani repa strelice) korišćenje javnih imena iz uvezenog paketa (na strani glave strelice) bez kvalifikacije
  - » uvezeni elementi se ponašaju kao javni u paketu u koji su uvezeni
- Privatno uvoženje (<<access>>):
  - » omogućava u paketu A u koji se uvozi (na strani repa strelice) korišćenje javnih imena iz uvezenog paketa B (na strani glave strelice) bez kvalifikacije, ali se uvezena imena paketa B ne mogu koristiti bez kvalifikacije u paketu C koji (javno) uvozi imena iz paketa A
  - » uvezeni elementi paketa B imaju status privatnih elemenata u paketu A u koji su uveženi



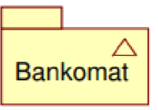
- Primeri u jezicima: Java i C# → <<access>>, C++ → <<import>>
- Alternativna notacija za uvoz imena iz drugog paketa je navođenje unutar paketa u koji se uvozi:
  - » `{import <puno ime paketa>}` ili
  - » `{access <puno ime paketa>}`
- Može da se uveze i pojedino ime iz drugog paketa uz (opcioni) alijas:
  - » `{element import <puno ime elementa> as <alijas>}` ili
  - » `{element access <puno ime elementa> as <alijas>}`
- Uvoz imena sadržanog paketa se ne podrazumeva, a može se naznačiti na sledeći način:
- Mešanje (<<merge>>):
  - » komplikovana relacija čije korišćenje nije potrebno u modeliranju
  - » koristi se u metamodeliranju
- Elementi paketa unutar paketa ili između paketa mogu biti povezani svim vrstama relacija
  - » na primer, unutar paketa se može predstaviti klasni dijagram, koji može da sadrži i druge pakete

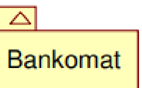
Principi modeliranja

- Pri projektovanju sistema se nastoji da se broj zavisnosti između paketa minimizira
  - » u paket se smeštaju apstrakcije koje uzajamno imaju veći broj relacija i relativno mali broj relacija prema apstrakcijama izvan paketa
- Nije dobro da postoje cirkularne zavisnosti između paketa
  - » dobro je da se apstrakcije grupišu slojevito tako da jedan paket predstavlja jedan sloj (nivo) apstrakcija, pa se relacije zavisnosti između paketa orijentišu samo u jednom smeru
- Načelo zajedničkog zatvaranja (Common Closure Principle [R.C. Martin]):
  - » preporučuje da u istom paketu budu klase koje treba menjati iz sličnih razloga
- Načelo zajedničkog ponovnog korišćenja (Comon Reuse Principle [R.C. Martin])
  - » preporučuje da u istom paketu budu klase koje će se zajedno ponovo koristiti
- Tehnika za sužavanje javnog interfejsa paketa
  - » svodi se na izvoženje samo malog broja operacija javnih klasa paketa
  - » klase paketa se učine privatnim, a uvede se posebna klasa sa javnim operacijama koje pozivaju odgovarajuće javne operacije privatnih klasa paketa (uzorak Fasada)

Stereotipovi paketa

- Iznad imena paketa može stajati naznaka << … >>
- Standardni stereotipovi paketa:
  - » koriste se određene ključne reči ili stereotipi da označe vrstu paketa
  - » model: <<model>>
    - — semantički potpun opis sistema





        - umesto ključne reči može da se koristi mali trougao
    - radni okvir: <<Framework>>
        - generička arhitektura kao proširiv obrazac za aplikacije u nekom domenu
        - tipično elementi predstavljaju osnovu za specijalizaciju
- Podsistem <Subsystem>> nije stereotip paketa, već komponente
    - u UML-u 1 je smatran stereotipom paketa

Dijagrami objekata

Uvod

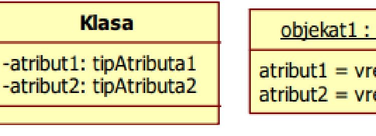
- Dijagrami objekata prikazuju primerke (objekte) apstrakcija (klasa) i njihove veze preko kojih objekti mogu da komuniciraju
- Oni predstavljaju “snimak” pogleda na sistem u jednom trenutku
  - » prikazuju se objekti sa trenutnim stanjem i trenutne veze
- Dijagram objekata opisuje fizički i statički aspekt modela
  - » fizički, jer je objekat fizička stvar, nalazi se u memoriji
  - » statički, jer se prikazuju samo veze, a ne i interakcija preko njih
- Elementi dijagrama objekata su:
  - » stvari: objekti i paketi
  - » relacije: veze
- Dijagram ima strukturu grafa: objekti su čvorovi, a veze grane

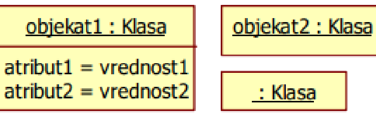
Namena

- Dve osnovne namene:
  - » prikaz složene strukture koju čini više objekata
  - » prikaz ponašanja kroz vreme preko niza “snimaka” povezanih objekata
- Dijagram objekata predstavlja samo primer, a ne specifikaciju (definiciju) modela
- Ima dokumentacionu svrhu, može da pomogne razumevanju modela

Objekti

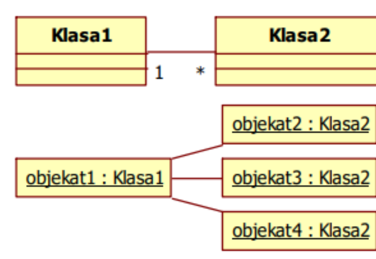
- Objekat je nešto (u memoriji) što ima stanje, ponašanje i identitet
- Na dijagramu objekata – objekat u konkretnom stanju
- Na dijagramu interakcije – objekat koji igra neku ulogu, menja stanje
- Objekat je primerak apstrakcije (tipa, klase)
- UML notacija objekta u konkretnom stanju:
  - » pravougaonik sa odeljkom naziva i odeljkom vrednosti atributa
  - » naziv ime:tip
  - » naziv podvučen
  - » može samo :tip – anonimni
  - » može i samo ime, bez tipa
  - » može i ime[n]:tip, npr: zbika[10]:KlasaElementa





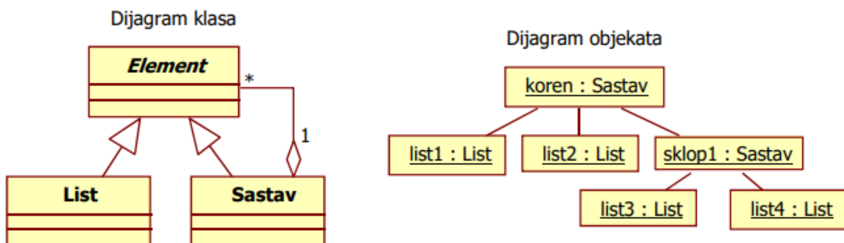
Veze

- Veze su komunikacione putanje između objekata
- Veze su primerci (instance) asocijacija
  - » jedna asocijacija između dve klase predstavlja skup veza između objekata tih klasa
- Na vezama se mogu naći:
  - » svi ukrasi asocijacije osim multiplikativnosti
  - » multiplikativnost je uvek 1



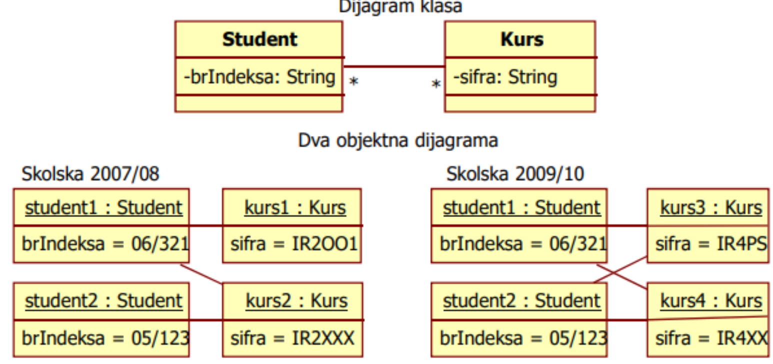
Primer – uzorak Kompozicija

- Opis hijerarhije objekata uzorka Kompozicija
  - » Sastav je Element koji sadrži druge elemente
  - » rekurzivna struktura sadržanja
  - » hijerarhija objekata tipa stabla



Primer – relacija "pohađa"

- Opis relacije "pohađa" između studenata i kurseva
  - » asocijacija označava veze koje se menjaju u vremenu
  - » veze su različite u svakoj školskoj godini



Uvod

- Interakcija je ponašanje koje obuhvata skup poruka koje se razmenjuju između skupa objekata u nekom kontekstu sa nekom namenom
- UML 2.0: interakcija je specifikacija slanja stimulusa između objekata sa ciljem obavljanja nekog zadatka
  - » definiše se u kontekstu neke saradnje (kolaboracije)
- Interakcija se koristi za modeliranje dinamičkih aspekata modela
- Objektni dijagram je reprezentacija statičkih aspekata komunikacije
  - » prezentiraju se samo objekti i veze između objekata u jednom trenutku
- Interakcija uvodi dinamički aspekat komunikacije
  - » prezentira se i sekvenca poruka koje razmenjuju uloge

Kontekst interakcije

- Kontekst – **sistem ili podsistem** kao celina
  - » interakcije su u saradnji objekata koji postoje u sistemu ili podsistemu
  - » primer – sistem za e-trgovinu: sarađuju objekti na strani klijenta sa objektima na strani servera
- Kontekst – **operacija**
  - » interakcije su među objektima koji implementiraju operaciju
  - » tekući objekat, parametri operacije, lokalni i globalni objekti mogu interagovati da izvrše algoritam operacije
- Kontekst – **klasa**
  - » atributi klase mogu sarađivati međusobno kao i sa drugim objektima (globalnim, lokalnim i parametrima operacija)
  - » interakcija se koristi da opiše semantiku klase
- Kontekst – **slučaj korišćenja** - interakcija reprezentuje scenario za slučaj korišćenja

Poruka

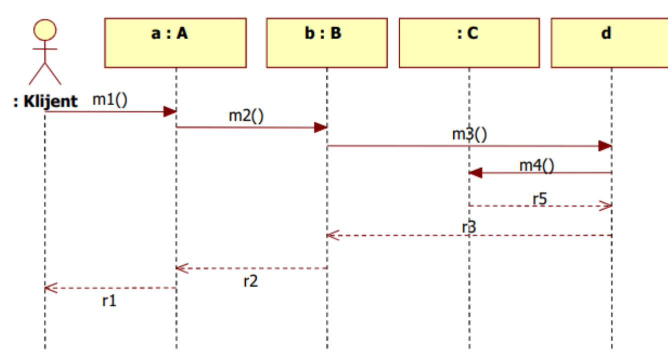
- Poruka je specifikacija komunikacije između objekata koja prenosi informaciju, iza koje se očekuje da sledi aktivnost
- Slanje poruke ulozi označava samo stimulus za aktivnost
- Poruka može biti: asinhrona (slanje signala) ili sinhrona (poziv operacije)
- Poruka se prikazuje kao strelica na dijagramu interakcije
  - » razne vrste strelica odgovaraju raznim vrstama poruka
  - » na dijagramu sekvence – poruke su linije između linija života
  - » na dijagramu komunikacije – poruke su strelice pored konektora

Vrste dijagrama interakcije

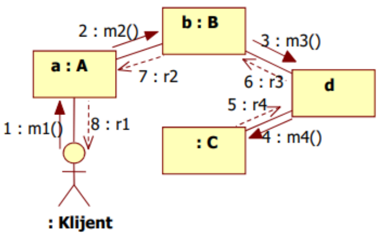
- UML 2 ima 4 vrste dijagrama interakcije
- **Dijagrami sekvence** naglašavaju vremensko uređenje interakcije
- **Dijagrami komunikacije** naglašavaju strukturu veza između učesnika u interakciji
  - » u UML-u 1 nazivali su se dijagramima saradnje (kolaboracije)
  - » vizuelizuju na različit način praktično iste informacije kao d. sekvence
- **Dijagrami pregleda** interakcije (UML 2) kombinuju dijagram aktivnosti sa dijagramima sekvence
  - » u okviru toka kontrole, blokovi (čvorovi) se opisuju interakcijama
- **Vremenski dijagrami** prikazuju promenu stanja objekta (ili uloge) u vremenu
  - » promena stanja se dešava kao posledica prijema poruka (stimulusa) i dešavanja događaja

Dijagrami sekvence i komunikcije

Dijagram sekvence

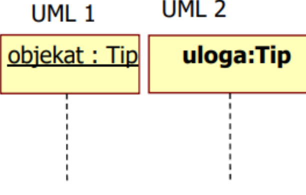


Dijagram komunikacije



Uloge i njihove linije života

- Linija života (lifeline)
  - » reprezent jednog učesnika (entiteta, objekta) u interakciji
  - » notacija (UML 2): naziv uloge (na liniji života) se ne podvlači
- Učesnici u jednoj interakciji mogu biti:
  - » UML 1: objekti - konkretni primerci u konkretnom stanju
    - — označavaju realnu stvar nekog tipa u tekućem stanju
    - — na primer, konkretan objekat osobe klase Osoba
  - » UML 2: uloge - prototipske stvari predstavnici konkretnih stvari
    - — označavaju proizvoljnu stvar nekog tipa
    - — na primer, referenca na objekat klase Osoba: Osoba o;



- U interakciji se mogu pojaviti "primerci" apstraktnih klasa i interfejsa
  - » takvi primerci ne označavaju konkretne stvari (nemogući su primerci apstraktnih klasa i interfejsa)
  - » predstavljaju prototipske stvari – uloge (to su primerci potklasa)

Konektori

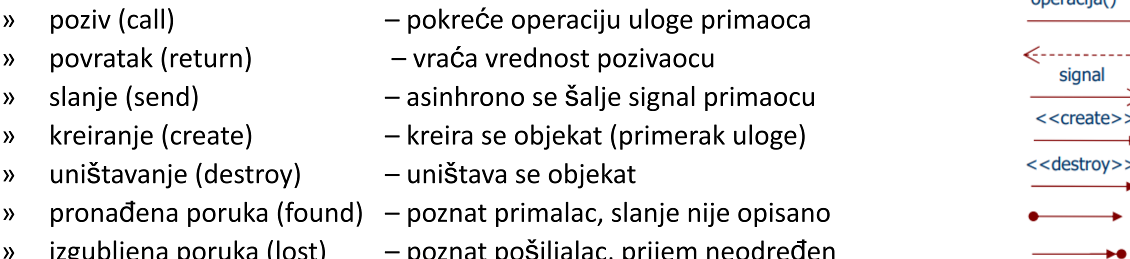
- Na dijagramima objekata se prikazuju primerci (objekti), a na dijagramima interakcije – uloge
- Na dijagramima objekata se prikazuju veze, a na dijagramima komunikacije – konektori
- Veza – strukturna sprega između objekata – primerak asocijacije
- Konektor – komunikaciona putanja između uloga
  - » ne mora da bude primerak asocijacije, može da se ostvaruje kao zavisnost
  - » može biti privremena

uloga1 : Klasa1 — uloga2 : Klasa2

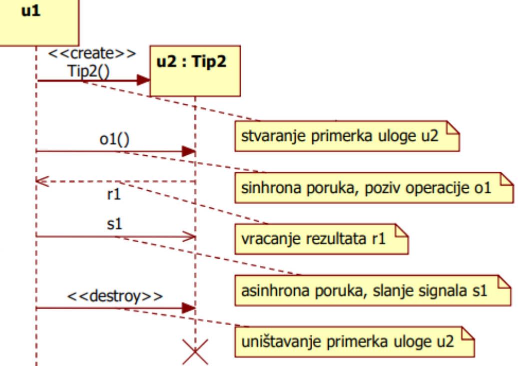
- Ukrasi načina pristupa drugoj strani konektora
  - » specificiraju način na koji uloga koja šalje poruku vidi drugu stranu
  - » tekstualni ukrasi koji se pišu na udaljenom kraju konektora (kod primaoca)
  - » konkretni ukrasi:
    - — {association} – objektu uloge se pristupa preko primerka asocijacije - veze
    - — {self} – objekat uloge sam sebi može da šalje poruku
    - — {global} – uloga je u nekom okružujućem dosegu imena
    - — {local} – uloga je u lokalnom dosegu imena
    - — {parameter} – uloga je argument operacije

Slanje i prijem poruke

- Prijem jedne poruke se može smatrati pojavom jednog događaja
- Kada se pošalje i primi poruka, sledi aktivnost na strani prijema
  - » izvršenje naredbe koja predstavlja apstrakciju metoda kod sinhrone operacije
- UML predviđa sledeće vrste poruka:
  - » poziv (call) – pokreće operaciju uloge primaoca
  - » povratak (return) – vraća vrednost pozivaocu
  - » slanje (send) – asinhrono se šalje signal primaocu
  - » kreiranje (create) – kreira se objekat (primerak uloge)
  - » uništavanje (destroy) – uništava se objekat
  - » pronađena poruka (found) – poznat primalac, slanje nije opisano
  - » izgubljena poruka (lost) – poznat pošiljalac, prijem neodređen

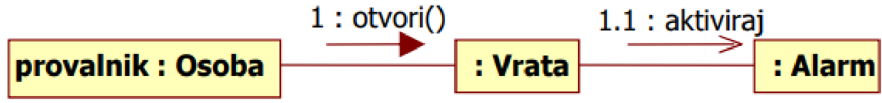


- Kod poruke vrste poziva (call) podrazumeva se "sinhronost":
  - » pozivalac ne napreduje dok pozvani objekat ne završi obradu poziva
  - » cela sekvenca ugnežđenih poziva se završava pre nego što se spoljašnji nivo izvršenja nastavi
  - » koristi se za:
    - — proceduralne pozive u jednoj niti
    - — pozive za randevu (npr. Ada) u višeprocesnom okruženju
- Primer raznih vrsta poruka na dijagramu sekvence:
- Poruke su horizontalne
  - » podrazumeva se atomičnost stimulusa
  - » ako se ne može smatrati atomičnim, poruka može biti crtana i ukoso naniže



Sekvenciranje poruka

- Unutar toka kontrole neke niti – poruke su uređene u vremensku sekvencu
- Na dijagramima sekvence sekvenca se modelira implicitno ređanjem poruka odozgo-naniže
- Na dijagramima komunikacije sekvenca se modelira rednim brojem ispred imena
- Grafička notacija:
  - » proceduralni (ugnežđeni) tok kontrole se prikazuje strelicama sa popunjenom glavom
    - — redni brojevi poruka imaju hijerarhijsku strukturu (nivoi hijerarhije se razdvajaju tačkom)
    - — primer: 2.1.3:op()
  - » ravni (flat) tok kontrole se prikazuje običnim strelicama (asinhrone poruke - signali)
    - — redni brojevi poruka nemaju hijerarhijsku strukturu
    - — primer: 5: s
- Primer kombinovane sekvence



Sekvenciranje poruka u nitima

- Identifikacija niti se piše iza rednog broja poruke kojom se pokreće aktivnost u kojoj se vrši konkurentno grananje:
    - » primer: 1a.5:uradi()
        - — aktivnost pokrenuta 1. porukom ima konkurentno grananje, pa se posmatra 5. poruka u niti a
    - » primer: 3b.5.2:dohvati()
        - — aktivnost pokrenuta 3. porukom se konkurentno grana, reč je o 2. poruci u okviru aktivnosti pokrenute 5. porukom u niti b

Sintaksa poruke

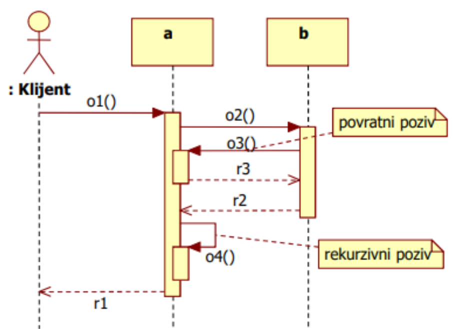
- Na poruci se osim imena mogu prikazati i
    - » njeni (stvarni) argumenti
    - » vraćena vrednost
    - » pridruživanje vraćene vrednosti promenljivoj
        - — vrednost se može koristiti kao argument neke naredne poruke
        - — promenljiva je po pravilu (ne mora biti) atribut klase pošiljaoca
    - » primer: `1.2:prosekGod=uspeh(godStud):srednjaOcena`
- Argumenti se mogu pisati i sa imenom parametra
    - » primer: `5:trazenaOsoba=trazi(ime="Petar Petrović")`

Životni vek objekata i veza

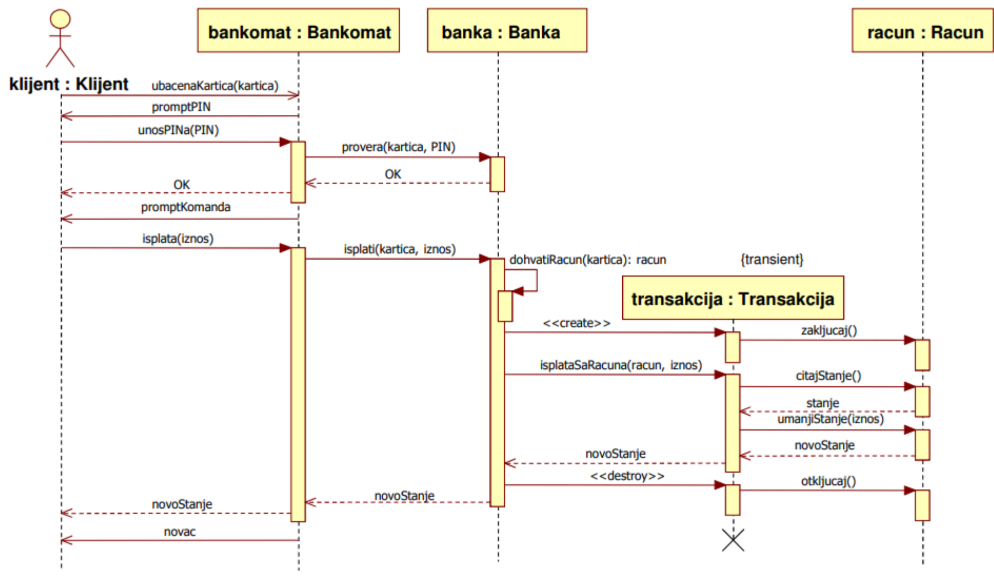
- Po nekad se životni vek objekta ili veze ne poklapa sa trajanjem interakcije
- Objekti i veze mogu nastajati i nestajati u toku interakcije
- Sledeća ograničenja se mogu pripisati objektu i/ili vezi (UML 1)
    - » {new} – objekat/veza se kreira za vreme izvršenja interakcije
    - » {destroyed} – objekat/veza se uništava pre završetka interakcije
    - » {transient} – objekat/veza se kreira i uništava za vreme interakcije
- Promena stanja ili uloge objekta na dijagramu interakcije se naznači njegovom replikacijom (UML 1)
    - » na dijagramu sekvence sve varijante jednog objekta se smeštaju na istu vertikalnu liniju
    - » na dijagramu komunikacije varijante se povezuju porukom <<become>>

Događanje izvršenja (fokus kontrole)

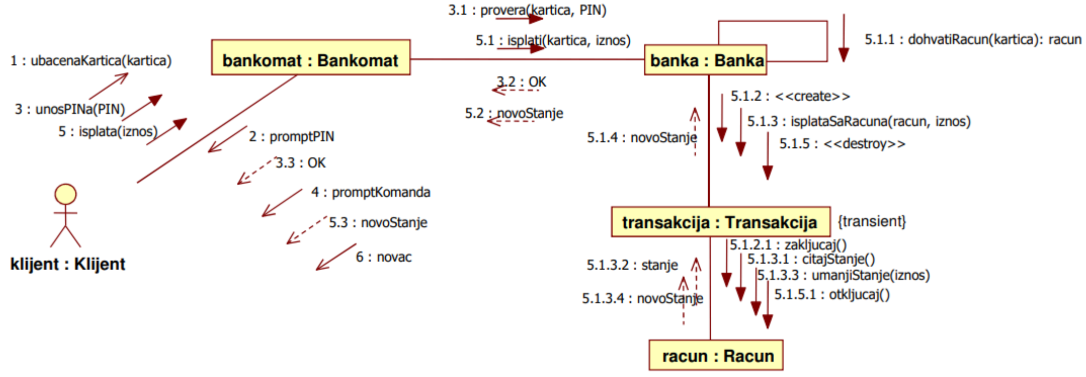
- Može naznačiti samo na dijagramima sekvence
    - » uski pravougaonik na liniji života
    - » definiše period u kojem uloga obavlja aktivnost izazvanu porukom
- Moguće je i ugnežđivanje događanja izvršenja iz sledećih razloga:
    - » (A) zbog rekurzije ili poziva sopstvene (druge) operacije
    - » (B) zbog povratnog poziva (call back) od pozvanog objekta
- Grafička notacija:



Primer Bankomat – dijagram sekvence



Primer Bankomat – dijagram komunikacije

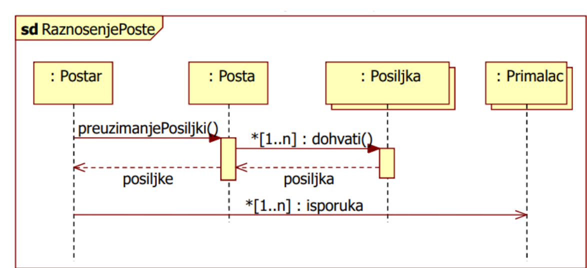


Iteracije i grananje

- Izrazi za iteracije i grananje se pišu iza broja za sekvenciranje poruke, na bilo kom nivou ugnežđenja
- Izraz za sekvencijalne iteracije:
  - brojač ponavljanja *[i:=1..n] ili uslov *[a>b] ili samo *
  - poruka sa r.br. ispred izraza se ponavlja u skladu sa izrazom
- Oznaka za paralelne iteracije: *||
- Izraz za grananje: [x>0]

» dve ili više poruka u sekvenci mogu imati isti redni broj, ali onda moraju imati disjunktne uslove

Fragment (okvir) interakcije



- Fragment interakcije je najopštija jedinica interakcije
- Opisuje deo interakcije i konceptualno je isti kao i sama interakcija
- Višestruki primerci (multiobjekti) – UML1, zastareli u UML2

Operatori kombinovanih fragmenata

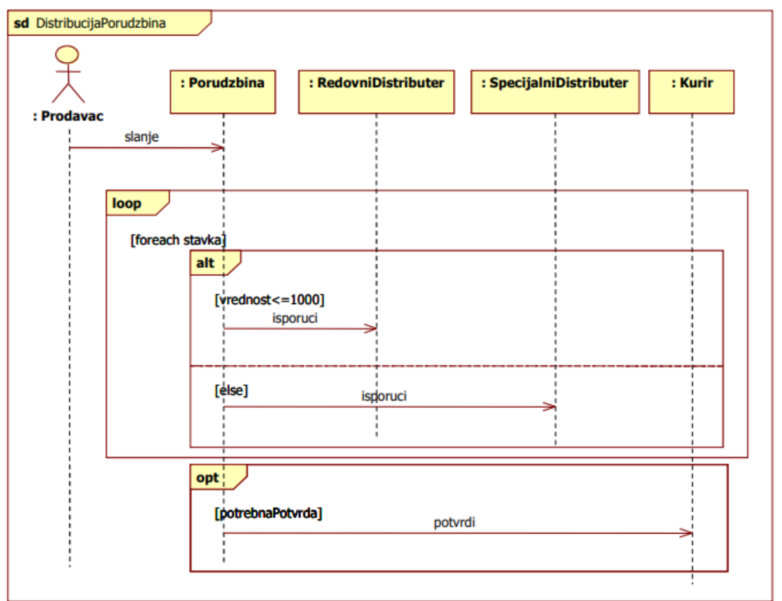
- Opšti:
  - sd - dijagram sekvence ili komunikacije (uokviruje ceo dijagram)
  - neg - negativno – fragment prikazuje pogrešnu interakciju
  - ref - referenca – interakcija je definisana na drugom dijagramu
- Kontrola sekvencijalnog toka:
  - opt - opcioni fragment – izvršava se samo ako je ispunjen uslov
  - alt - alternativni izbor između više fragmenata
  - loop - petlja – fragment se izvršava više puta
  - break - scenario se izvršava umesto ostatka okružujućeg fragmenta
- Kontrola paralelnih tokova:
  - par - paralelno se izvršavaju fragmenti
  - region - kritični region – u fragmentu se ne može istovremeno izvršavati više niti

Primer operatora

- Distribucija porudžbina
  - neka uloga tipa :Porudzbina ima operaciju slanje()

```
procedure slanje()
   foreach (stavka)
     if (vrednost<=1000) redovniDistributer.isporuci()
     else specijalniDistributer.isporuci()
     endif
   endfor
   if (potrebnaPotvrda) kurir.potvrdi()
   endif
end procedure
```

  - neka akter :Prodavac pokreće slanje porudžbina signalom
  - neka je cela komunikacija asinhrona


sd DistribucijaPorudzbina
: Prodavac
: Porudzbina
: RedovniDistributer
: SpecijalniDistributer
: Kurir
slanje
loop
[foreach stavka]
alt
[vrednost<=1000]
isporuci
[else]
isporuci
opt
[potrebnaPotvrda]
potvrdi

7 Dijagrami slučajeva korišćenja

Uvod

- Dijagram slučajeva korišćenja (eng. use-case diagram) prikazuje skup slučajeva korišćenja i aktera
- Tipično predstavlja neki skup funkcionalnosti nekog subjekta i aktere koji ih koriste
- Dijagram vizuelizuje deklarativne aspekte ponašanja sistema, podsistema, komponente ili čak klase i interfejsa
- Služi korisniku da razume šta sistem radi, a verifikatoru da proveri funkcionisanje
- Elementi dijagrama su:
  - » slučajevi korišćenja
  - » akteri
  - » relacije: asocijacija (komunikacija), stereotipi zavisnosti (uključivanje i proširivanje) i generalizacija
  - » paketi

Slučajevi korišćenja

- Slučaj korišćenja je opis skupa sekvenci akcija, uključujući varijante, koje subjekat (sistem) obavlja da bi proizveo vidljiv rezultat od vrednosti za pojedinog aktera
- Sekvenca akcija reprezentuje interakciju aktera sa subjektom, odnosno ulogama ključnih apstrakcija subjekta
- Jedna sekvenca akcija predstavlja jedan mogući scenario slučaja korišćenja
  - » jedan scenario je jedna pojava (događanje) slučaja korišćenja
- Slučaj korišćenja specificira šta subjekat radi, a ne kako radi
- Grafički simbol i alternativne notacije:

Opis slučaja korišćenja

- Ponašanje slučaja korišćenja se opisuje tokom događaja:
  - » kada slučaj korišćenja počinje i kada završava
  - » kada slučaj korišćenja interaguje sa akterima
    - — kada se razmenjuju poruke i podaci (objekti)
- Postoje primarni (osnovni) i alternativni tokovi događaja
- Tok događaja se može opisati na sledeće načine:
  - » neformalan tekst na govornom jeziku
  - » strukturirani tekst (sa pred- i post-uslovima)
  - » pseudokod
  - » dijagrami interakcije - jedan za primarni i dodatni za alternativne tokove
  - » dijagram stanja subjekta
  - » dijagram aktivnosti
- Saradnja (učestvuju i akteri) – strukturni aspekt slučaja korišćenja

Primer opisa ponašanja

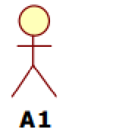
- Primer – provera identiteta korisnika pri transakciji na bankomatu
- Preduslov za sve tokove događaja: platna kartica u bankomatu
- Glavni tok događaja
  - » 1. slučaj korišćenja počinje kada sistem ispiše prompt za PIN broj
  - » 2. korisnik unosi PIN broj preko numeričke tastature
  - » 3. korisnik potvrđuje unos pritiskom na taster Enter
  - » 4. sistem proverava da li PIN broj odgovara kartici
  - » 5. provera uspela, završava se slučaj korišćenja
- Postuslov: omogućena promena na računu korisnika

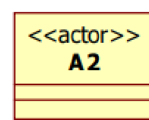
- Prvi alternativan tok događaja (poništavanje transakcije):
  - » 3. korisnik poništava transakciju pritiskajući taster Cancel
  - » 4. slučaj korišćenja se ponavlja
- Postuslov: nije omogućena promena na računu korisnika

- Drugi alternativan tok događaja (pogrešan PIN):
  - » 5. bankomat informiše korisnika o pogrešnom PIN-u
  - » 6. slučaj korišćenja se ponavlja
  - » 7. ako se ovo ponovi tri puta za redom sistem poništava celu transakciju i sprečava korisnika da ponovo pokuša 60s
- Postuslov (posle svakog neuspeha): nije omogućena promena na računu korisnika

Akteri

- Akter predstavlja neki koherentan skup uloga
- Akter može biti čovek (korisnik) ili neki sistem sa kojim modelirani subjekat interaguje
- Subjekat interaguje sa jednim ili više aktera
- Akter je standardni stereotip klase sa posebnim grafičkim simbolom



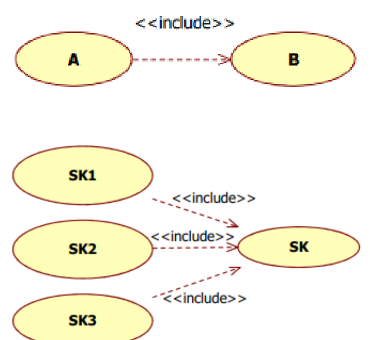


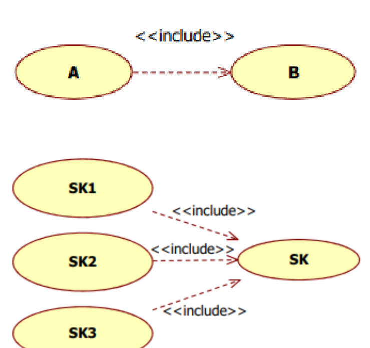
Relacija komunikacije

- Prikazuje se punom linijom (asocijacija)
- Komunikaciju može inicirati akter ili slučaj korišćenja (bidirekcionalna veza)
- Relacija dozvoljena između:
  - » aktera i slučaja korišćenja
  - » dva slučaja korišćenja koja se ne odnose na isti subjekat
- Multiplikativnost >1 na strani aktera
  - » za događanje slučaja korišćenja potrebno je više aktera (konkurentno ili sekvencijalno)

Relacija uključivanja

- Prikazuje se isprekidanom linijom sa strelicom i natpisom <<include>>
  - » relacija je stereotip relacije zavisnosti
- Relacija uključivanja od slučaja korišćenja A prema slučaju korišćenja B ukazuje da će slučaj korišćenja A uključiti i ponašanje slučaja korišćenja B
- Ponašanje opisano u B je obavezno za A
- Koristi se da opiše zajedničko ponašanje između više slučajeva korišćenja

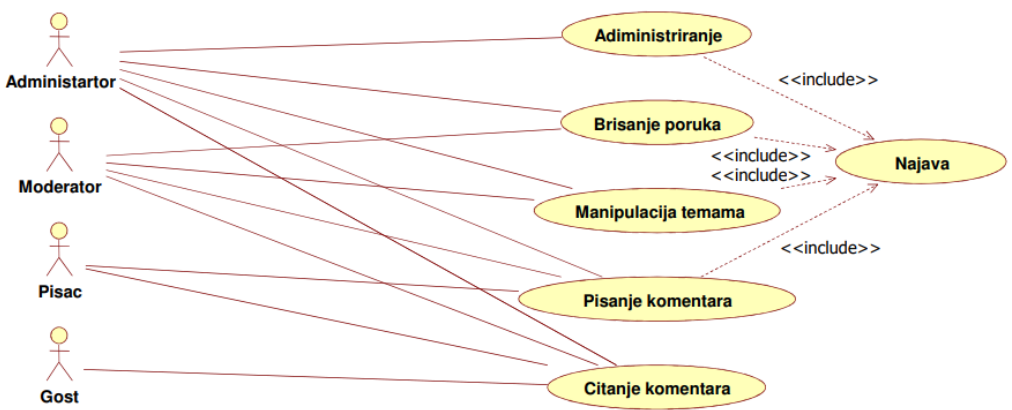




» na primer: slučajevi korišćenja SK1, SK2 i SK3 uključuju ponašanje SK

PRIMER – KORISNICI FORUMA

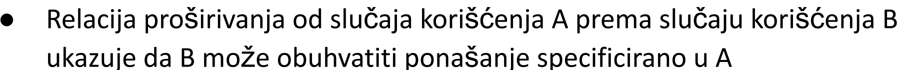
- Dijagram opisuje najavu korisnika na forum



RELACIJA PROŠIRIVANJA

- Prikazuje se isprekidanom linijom sa strelicom i natpisom <<extend>>
  - » relacija je stereotip relacije zavisnosti

<<extend>> A B

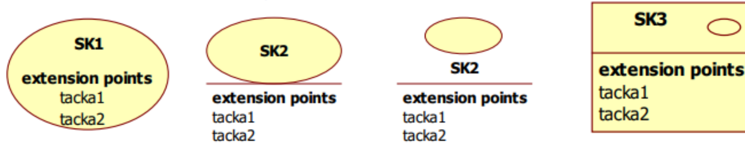


- Relacija proširivanja od slučaja korišćenja A prema slučaju korišćenja B ukazuje da B može obuhvatiti ponašanje specificirano u A
  - » praktično B može da se proširi i ispolji celokupno ponašanje opisano u A

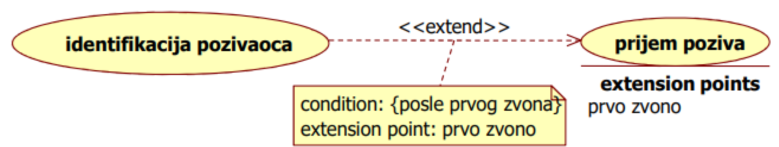
- Koristi se da se izrazi opciono ponašanje osnovnog slučaja korišćenja
  - » ponašanje opisano u A je opciono, a ono u B osnovno
- Problem sa terminom stereotipa <<extend>>
  - » sličnost sa ključnom rečju extends jezika Java
  - » sasvim različito značenje

TAČKE PROŠIRENJA SLUČAJA KORIŠĆENJA

- Osnovni slučaj korišćenja se proširuje u određenim tačkama ponašanja
  - » tačka se naziva tačkom proširenja (ekstenzije)
- Alternativne notacije:

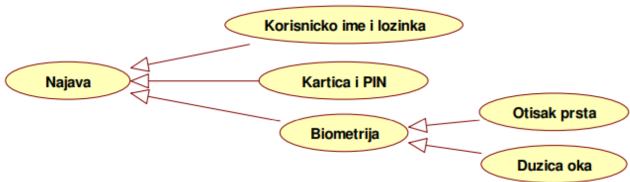


- Tačka se navodi po sintaksi: ime [: objašnjenje]
- Primer: identifikacija pozivaoca je opciona funkcija telefona



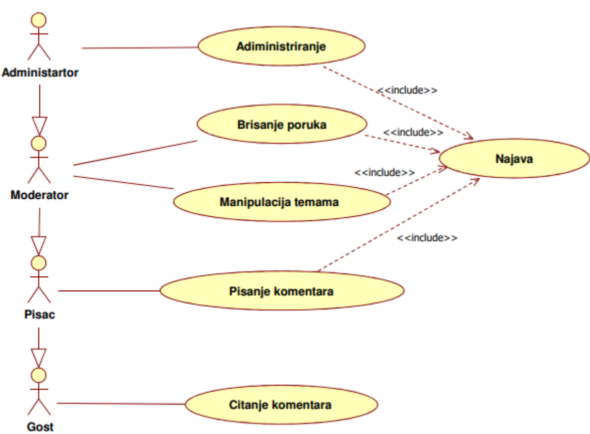
Relacija generalizacije

- Prikazuje se punom linijom sa trougaonom strelicom
- Relacija je osnovna relacija generalizacija/specijalizacija
- Relacija generalizacije od slučaja korišćenja A prema slučaju korišćenja B ukazuje da je slučaj korišćenja A specifičan slučaj opštijeg slučaja B
- Primer:



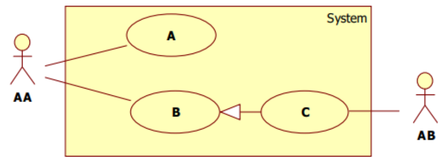
Generalizacija aktera

- Relacija generalizacije može postojati i između aktera



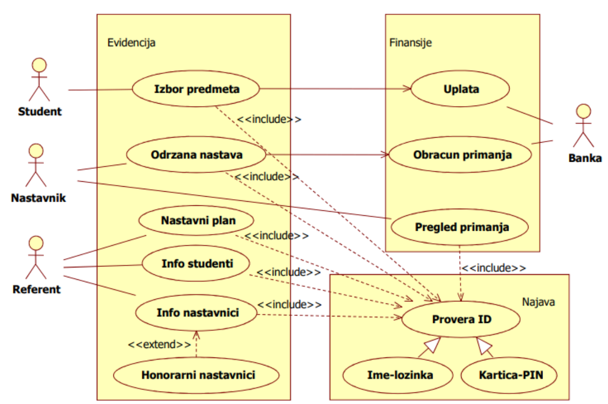
Okvir subjekta

- Slučajevi korišćenja su “unutar”, a akteri “izvan” subjekta modeliranja
- Vizuelizacija – okvir subjekta:



- Subjekat nije vlasnik slučajeva korišćenja koji predstavljaju njegove funkcionalnosti
- Vlasnik može biti klasa, paket ili model

Primer – Info sistem fakulteta



8 Dijagrami stanja

Uvod

- Automat stanja (state machine)
  - » ponašanje koje specificira sekvence stanja kroz koja prolazi neki objekat ili interakcija, kao odgovor na događaje, proizvodeći akcije
  - » modelira **ponašanje** nekog entiteta ili **protokol** interakcije
- Entitet reaguje na događaje promenom stanja
  - » promena stanja izaziva nove događaje i akcije
- Dijagram stanja je graf koji prikazuje automat stanja
  - » čvorovi su stanja
  - » grane su prelazi
- Dijagrami stanja se fokusiraju na događajima vođeno ponašanje
- Dijagrami aktivnosti se fokusiraju na tok aktivnosti
- Dijagrami stanja se kreiraju za entitete koji pokazuju bitno dinamičko ponašanje
- Dijagram stanja - formalna specifikacija ponašanja
- U osnovi se koriste Harelovi dijagrami sa modifikacijama da budu OO

Kontekst primene automata stanja

- Kontekst primene može biti:
  - » objekat klase
  - » slučaj korišćenja
  - » akter
  - » podsistem/sistem
  - » metod/operacija
- Automat stanja se primenjuje da specificira ponašanje
  - » elemenata modela koji moraju da odgovaraju na asinhrone događaje
  - » elemenata modela čije tekuće ponašanje zavisi od istorije
- Automat stanja se uspešno koristi za modeliranje ponašanja reaktivnih sistema
  - » reaktivni sistem - odgovara na signale koje daju akteri - akteri su uloge iz spoljašnjeg sveta

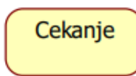
Elementi dijagrama stanja

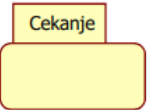
- Dijagram stanja prikazuje:
  - » stanja i pseudostanja (čvorovi grafa),
  - » prelaze (tranzicije) između stanja (grane grafa)

» događaje koji prouzrokuju promenu stanja i
» akcije koje rezultuju iz promene stanja

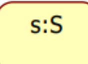
Stanja i podautomati stanja

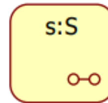
- Stanje je situacija u toku životnog veka entiteta u kojoj entitet može da postoji
- U jednom stanju entitet
  - » zadovoljava neki uslov,
  - » obavlja neku aktivnost ili
  - » čeka događaj
- Primeri:
  - » uslov – student je u stanju upisan ili stanju mirovanja
  - » aktivnost – student pohađa nastavu ili student polaže ispite u ispitnom roku
  - » čekanje – student je završio pripremu ispita i čeka na početak ispitnog roka
- Grafička notacija:





- Podautomat stanja – stanje koje reprezentuje automat stanja koji se može pojaviti na više mesta unutar dijagrama stanja
  - » referenca na drugi automat stanja unutar nekog automata stanja

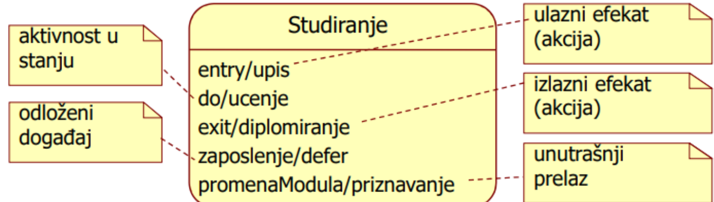




Elementi stanja

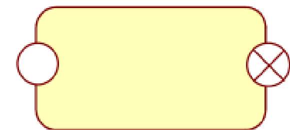
- Ime – tekst koji razlikuje jedno od drugih stanja
  - » upisuje se u odeljak imena (unutar simbola stanja) ili
  - » u poseban pravougaoni jezičak iznad gornjeg levog ugla simbola stanja
  - » stanje može biti i anonimno (bez imena)
- Ulazni efekat (akcija) – radnja koja se obavi pri ulasku u stanje
- Izlazni efekat (akcija) – radnja koja se obavi pri izlasku iz stanja
- Aktivnost u stanju – radnja koja se izvršava dok je objekat u datom stanju
- Odloženi događaji – lista događaja koji se ne obrađuju u datom stanju već se smeštaju u red čekanja
- Unutrašnje tranzicije
  - » tranzicije koje obrađuju događaj i zadržavaju objekat u istom stanju
  - » različite su od samo-tranzicije: ne izazivaju izlaznu pa ulaznu akciju
- Podstanja – stanja koja postoje unutar datog stanja, sekvencijalno ili konkurentno aktivna

Primer stanja



Pseudostanja i specijalna stanja

- Početno (initial)
- Završno (final)
- Ulazno (entry)
- Izlazno (exit)
- Sinhro (fork/join)
- Izbor (choice)
- Spoj (junction)
- Plitka istorija (shallow h.)
- Duboka istorija (deep h.)

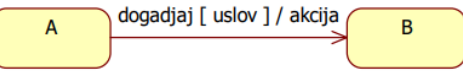


- Ulazno/izlazno pseudostanje:
  - » ulazna/izlazna tačka podautomata stanja
  - » identifikuje odgovarajuće stanje datog podautomata
- Pseudostanja izbor/spoj se koriste za grane (prelaze)

Prelazi

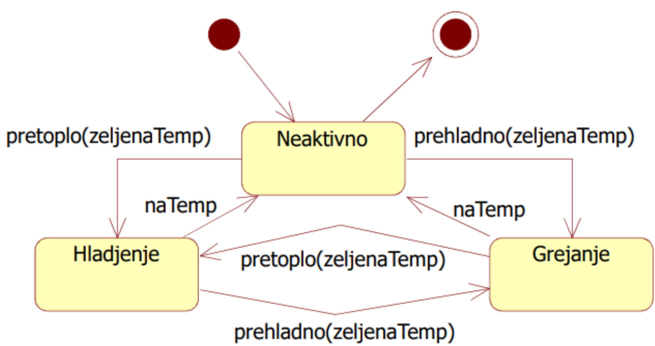
- Prelaz je relacija između dva stanja
- Prelaz ukazuje da objekat napušta jedno stanje obavlja akciju i ulazi u drugo stanje kada se dogodi specificirani događaj i kada je ispunjen specificirani uslov
- Grafička notacija – strelica:

- Elementi prelaza
  - » događaj je zbivanje koje nema trajanje i može prouzrokovati prelaz
  - » zaštitni uslov je Bulov izraz koji čini prelaz mogućim kada je uslov ispunjen
  - » akcija je radnja koja je pridružena prelazu i može biti
    - — poziv operacije objekta vlasnika automata stanja ili drugog objekta koji je vidljiv datom objektu
    - — slanje signala nekom objektu
    - — kreiranje ili uništavanje drugog objekta



Primer dijagrama stanja

- Sistem klima-uređaja

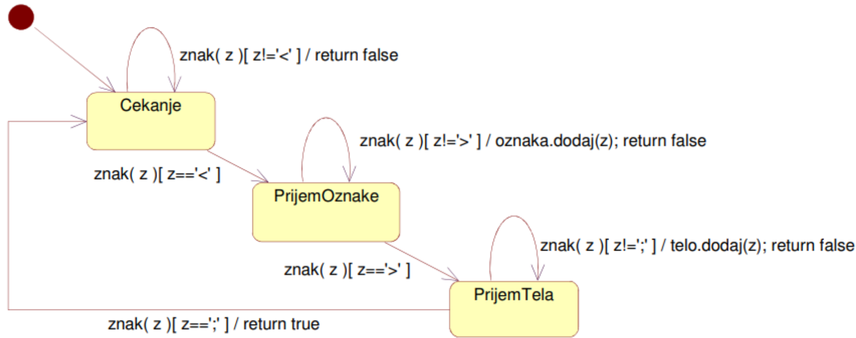


Mealy i Moor automati

- Kada se modelira ponašanje objekta akcije se mogu vezivati za: stanja ili prelaze
- Automat može biti:
  - » Moor-ovog tipa - sve akcije vezane za stanja
  - » Mealy-jevog tipa - sve akcije vezane za prelaze
- U praksi dijagrami stanja kombinuju Moor i Mealy automate

Primer automata Mealy-jevog tipa

- Automat stanja za parsiranje poruka - niza znakova koji odgovaraju sintaksi: '<' string '>' string ';'
- Prvi string predstavlja oznaku (tag), a drugi telo poruke (body)
- Automat radi beskonačno – nema završnog stanja

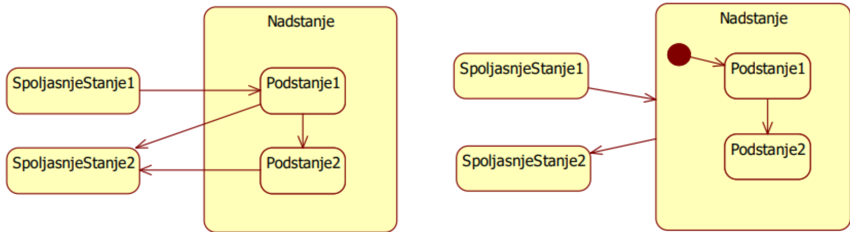


Složena stanja

- Jednostavno stanje – stanje koje nema unutrašnju strukturu automata stanja
- Složeno (kompozitno) stanje – stanje koje ima unutrašnja stanja, tj. predstavlja automat stanja
- Složena stanja se koriste da se smanji grafička kompleksnost
- Nadstanje – stanje koje obuhvata više unutrašnjih (ugnežđenih) stanja
- Podstanje je unutrašnje (ugnežđeno) stanje
- Kada se objekat nalazi u podstanju – istovremeno se nalazi i u nadstanju
- Podstanja mogu biti:
  - » sekvencijalna
  - » konkurentna

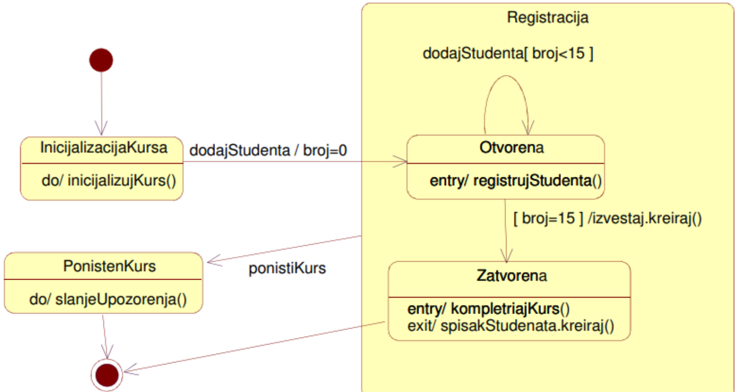
Sekvencijalna podstanja

- Prelazi se mogu događati:
  - » između podstanja
  - » između podstanja ili nadstanja i stanja izvan nadstanja (spoljašnjeg stanja)
- Ako je nadstanje cilj tranzicije iz spoljašnjeg stanja, nadstanje sa sekvencijalnim podstanjima mora sadržati početno stanje
- Ako je nadstanje izvor tranzicije, najpre se napušta podstanje pa nadstanje
- Pri tranziciji u/iz nadstanja izvršavaju se ulazne/izlazne akcije i nadstanja i podstanja

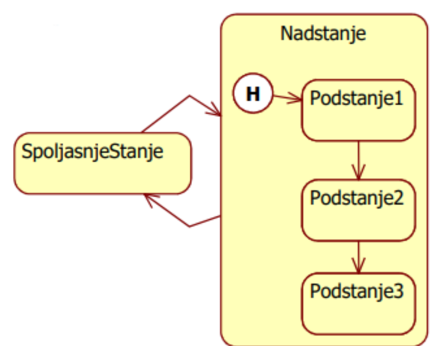


Primer sekvencijalnih podstanja

- Registracija studenata za kurs



Stanje sa istorijom

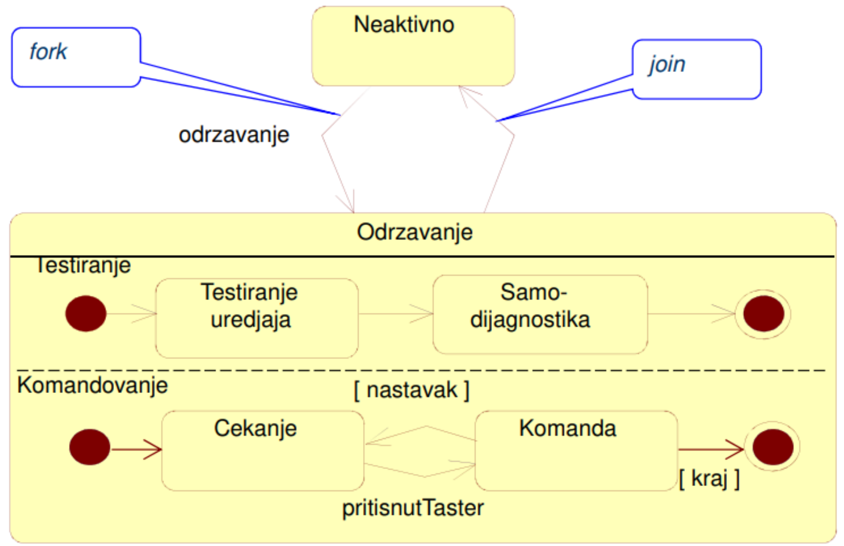


- Kada se uđe u nadstanje obično se kreće od inicijalnog podstanja
- Ponekad postoji potreba da se krene od podstanja iz kojeg je napušteno nadstanje
- Simbol H u kružiću ukazuje da nadstanje pamti istoriju
- Simbol H u kružiću označava "plitku" istoriju – pamti se istorija samo neposredno ugnežđenog automata stanja
- Simbol H* u kružiću označava "duboku" istoriju – pamti se istorija do najugnežđenijeg automata stanja proizvoljne dubine

Konkurentna podstanja

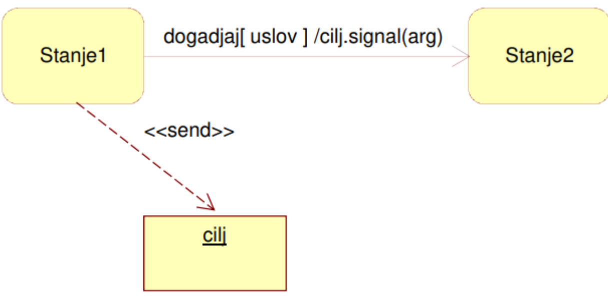
- Konkurentna podstanja
  - » dva ili više automata stanja koja se izvršavaju u paraleli
  - » izvršavaju se u kontekstu odgovarajućeg objekta, kao i sekvencijalna
- Drugi način da se modelira konkurentnost je pomoću aktivnih objekata
  - » umesto deljenja jednog automata stanja objekta na dva konkurentna podstanja definišu se dva aktivna objekta od kojih je svaki odgovoran za ponašanje jednog podstanja
- Od više sekvencijalnih podstanja na jednom nivou – objekat može biti samo u jednom
- Od više konkurentnih podstanja na jednom nivou – objekat je u svakom od njih
- Prelaz u stanje sa konkurentnim podstanjima predstavlja fork prelaz
- Prelaz iz stanja sa konkurentnim stanjima predstavlja join prelaz
  - » sva konkurentna podstanja moraju biti završena da bi se izvršio prelaz join iz nadstanja
  - » ako jedno konkurentno podstanje stigne do završnog stanja pre drugog – čeka na drugo
- Sekvencijalna podstanja koja obrazuju svako od konkurentnih podstanja
  - » mogu imati početno, završno ili pseudostanja

Primer konkurentnih podstanja

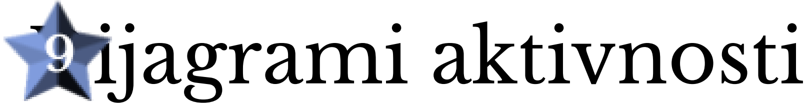


Slanje signala

- Kada se iz akcije šalje signal nekom objektu, taj objekt se može prikazati na dijagramu



- Objekat je vezan sa stanjem (izvornim) stereotipom send relacije zavisnosti



Uvod

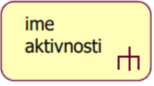
- Dijagrami aktivnosti su namenjeni modeliranju dinamičkih aspekata (ponašanja) sistema
- Slični konvencionalnim dijagramima
  - » kontrole toka (flow-chart)
  - » toka podataka (data-flow)
- Prikazuju sekvencijalne i konkurentne korake u procesu obrade
- Mogu se koristiti za opis:
  - » toka poslovnog procesa
  - » toka neke operacije
- Semantika akcija – uvedena u UML1.5
  - » uz definisanje sintakse akcija – formalna specifikacija ponašanja
  - » podrška za izvršive modele

Odnos prema drugim dijagramima

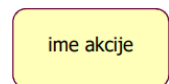
- Dijagram **interakcije** prikazuje – tok poruka koje se razmenjuju između objekata
- Dijagram **stanja** prikazuje – tok promene stanja objekta
- Dijagram **aktivnosti** prikazuje – tok aktivnosti koju izvršavaju objekti – eventualno i tok objekata između koraka aktivnosti
- Dijagram aktivnosti prikazuje ponašanje koristeći modele toka kontrole i toka podataka

Aktivnosti i akcije

- **Aktivnost** je specifikacija parametrizovanog ponašanja koje se izražava kroz tok izvršenja preko sekvenciranja i konkurisanja podaktivnosti
  - » aktivnost reprezentuje neatomsku obradu koja se dekomponuje na jedinice
  - » elementarne jedinice podaktivnosti – pojedine akcije
- Simbol aktivnosti:

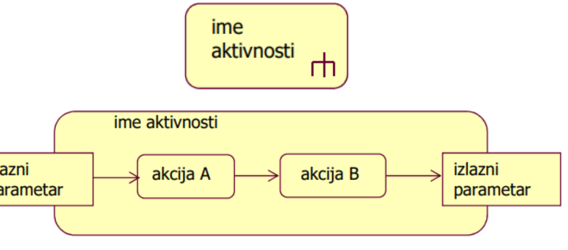


- **Akcija** je osnovna jedinica specifikacije ponašanja koja reprezentuje neku transformaciju ili obradu u modeliranom sistemu
  - » akcija je osnovni izvršni element aktivnosti - osnovna jedinica izvršne funkcionalnosti
  - » akcija predstavlja jedan korak u aktivnosti koji se obično dalje ne dekomponuje
  - » aktivnost predstavlja kontekst akcije
- Simbol akcije:



Aktivnosti

- Svaka aktivnost može da se predstavi posebnim dijagramom
- Aktivnost definiše parametrizovano ponašanje
- Mogu da se prikažu ulazni i izlani parametri aktivnosti
- Može da se prikaže unutrašnja struktura aktivnosti
- Simbol aktivnosti bez unutrašnje strukture - referenca na datu aktivnost
- Simbol aktivnosti može da se ponavlja na više mesta na dijagramu



Akcije

- Akcije mogu biti:
  - » pokretanje aktivnosti
    - — pozivi operacija
    - — slanje signala
  - » čitanje (vraćanje vrednosti) ili upis (promena stanja) podataka
    - — kreiranje ili uništavanje objekata (vrsta upisa)
  - » izračunavanje
    - — izvršenje primitivnih (npr. aritmetičkih) operacija i funkcija
- Izvršenje akcije koja pokreće neku aktivnost obuhvata izvršenje te aktivnosti (njenih akcija) - takva akcija nije atomska
- Akcija se može posmatrati i kao diskretan element i kao složeno ponašanje:
  - » kao deo strukture u modelu aktivnosti - akcija je diskretan element (aktivnosti)
  - » kao specifikacija ponašanja - akcija može pokrenuti ponašanje (aktivnost) proizvoljne složenosti

Izvršenje akcije/aktivnosti

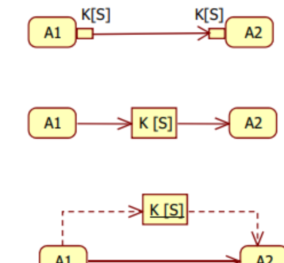
- Akcija/aktivnost (A) može da ima skupove ulaznih i izlaznih grana
  - » one specificiraju tok kontrole ili tok objekata od i prema drugim čvorovima dijagrama aktivnosti
  - » a neće početi izvršenje dok:
    - — nisu završene sve A koje prethode po ulaznim granama
    - — nisu svi objekti na ulaznim granama raspoloživi
    - — nisu poslati svi signali prethodnih A na ulaznim granama
    - — nisu svi ulazni uslovi ispunjeni
    - — nisu se desili svi čekani događaji i vremenski događaji na ulaznim granama
  - » završetak izvršenja jedne omogućava izvršenje skupa sledećih A

- Model žetona (tokena)
  - » da bi A počela, potrebno je da su raspoloživi žetoni na svim ulaznim granama
  - » kada A počne, konzumiraju se žetoni – ni jedna druga A ne može da ih koristi
  - » kada A završi, na svim izlaznim granama se pojavljuju žetoni

Elementi dijagrama aktivnosti

- Dijagrami aktivnosti su grafovi koji sadrže:
  - » čvorove:
    - — akcije i aktivnosti
    - — objekti
    - — slanja signala (send signal)
    - — prihvatanja događaja (accept event)
    - — prihvatanja vremenskog događaja (accept time event)
    - — kontrolni čvorovi:
      - ▪ sekvencijalna grananja i spajanja u toku kontrole (decision i merge)
      - ▪ konkurentna grananja i spajanja u toku kontrole (fork i join)
    - — pseudočvorovi: početni, završni i kraj toka
    - — konektori
  - » grane:
    - — prelazi (tranzicije) između akcija
    - — tok objekata

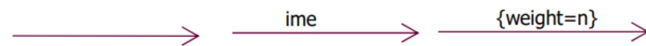
Tok objekata



- Tok objekta može da se naznači tako što se (prototipski) objekat povezuje simbolima prelaza (strelicama) sa susednim akcijama
- Akcije mogu da stvaraju, čitaju, modifikuju ili uništavaju objekat
- Grafička notacija:
  - » nožice (pinovi) koje predstavljaju objekat
    - — iznad nožice: klasa [stanje]
  - » alternativna sintaksa:
    - — u pravougaoniku: klasa [stanje]
    - — FIFO baferisanje više objekata
  - » tok objekata u UML 1:
    - — isprekidanim strelicama
    - — poseban tok u odnosu na tok kontrole: paralelno sa granom prelaza između akcija
    - — naziv objekta podvučen

Pseudočvorovi, tranzicije, konektori

- Grafička notacija pseudočvorova:
  - » za početni čvor:
  - » za završni čvor (kraj svih tokova):
  - » za kraj jednog (konkurentnog) toka:
- Prelaz (tranzicija) je legalna putanja od jednog do drugog čvora
- Grafička notacija:

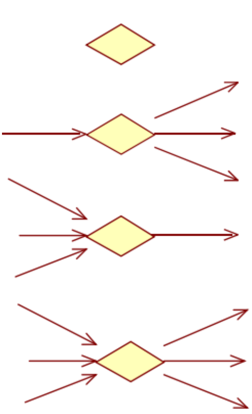


  - » poslednji oblik – izvorište je objekat (ne konkretni, već prototipski - uloga), a n je minimalan broj objekata izvorišta koji se koriste u čvoru akcije/aktivnosti
  - » podrazumevano: {weight=1}
  - » {weight=*} - proizvoljno mnogo
  - » primer: formiranje fudbalskog tima

Igrač
{weight=10}
Golman
{weight=1}
Formiranje fudbalskog tima

- Konektori sa istim imenom predstavljaju jednu tranziciju
- Grafička notacija:

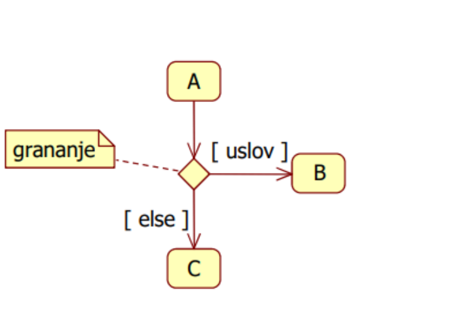
Sekvencijalna grananja

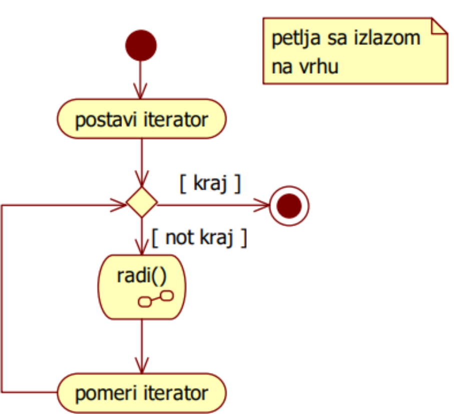


- Grananje specificira alternativne putanje kojima će se ići u zavisnosti od uslova
- Isti simbol se koristi za grananje i spajanje sekvencijalnog toka kontrole:
  - » više grana može izlaziti iz simbola sekvencijalnog grananja (decision)
    - — uslov se piše u uglastim zagradama na grani
    - — [else] grana – ako nije ispunjen ni jedan uslov
  - » više grana može ulaziti u simbol sekvencijalnog spajanja (merge)
- Dozvoljeno je kombinovanje grananja i spajanja u jednom kontrolnom čvoru
- Žeton se pojavljuje na bilo kojoj ulaznoj grani prosleđuje se na samo jednu izaznu granu

PRIMERI GRANANJA I ITERACIJE

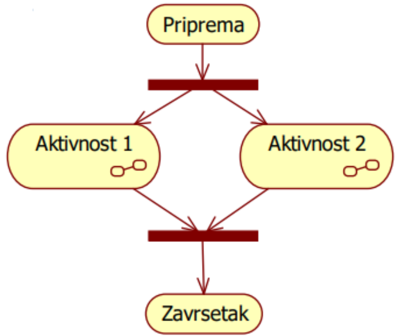
- Iteracija se formuliše pomoću grananja
- Primeri: grananje: iteracija:



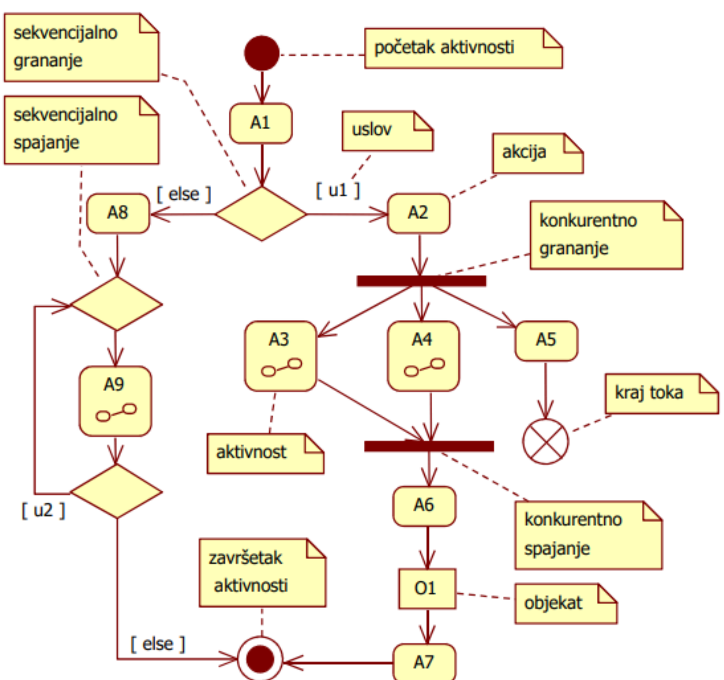


KONKURENTNA GRANANJA

- Nit kontrole se može u nekoj tački granati na više konkurentnih niti
- Račvanja (fork) i udruživanja (join) niti se obavljaju u sinhronizacionim tačkama
- Grafička notacija:

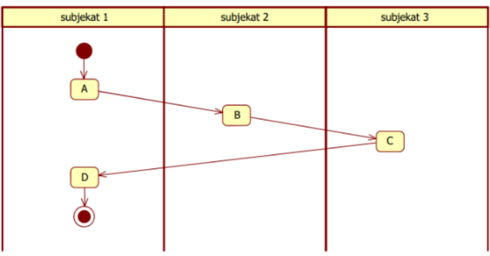


- Žeton treba da se pojavi na svum ulaznim granama i svim izlaznim
- Primer:



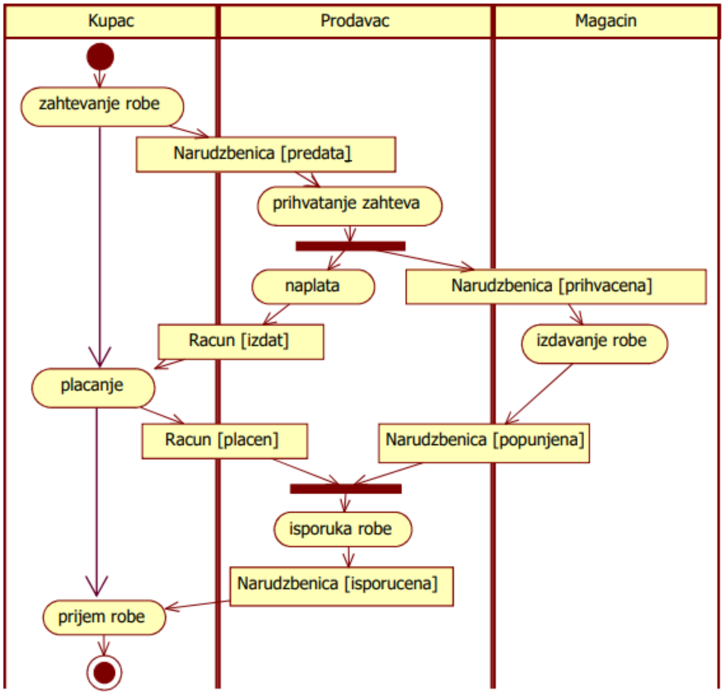
Plivačke staze

- Plivačke staze (swimlanes) specificiraju odgovornosti za delove celokupne aktivnosti
- Nemaju neku duboku semantiku
- Staza reprezentuje neki subjekat odgovoran za sprovođenje akcije
  - » prototipski objekat aplikacije ili entitet realnog sveta
- Notacija plivačkih staza:



- Akcije pripadaju stazama
- Tranzicije mogu prelaziti iz jedne staze u drugu plivačku stazu

Primer dijagrama aktivnosti



Hijerarhijske staze i particije

Hijerarjihska plivačka staza:

Particija 1

Particija 1.1 | Particija 1.2

Višedimenzione particije:

dimenzija 1

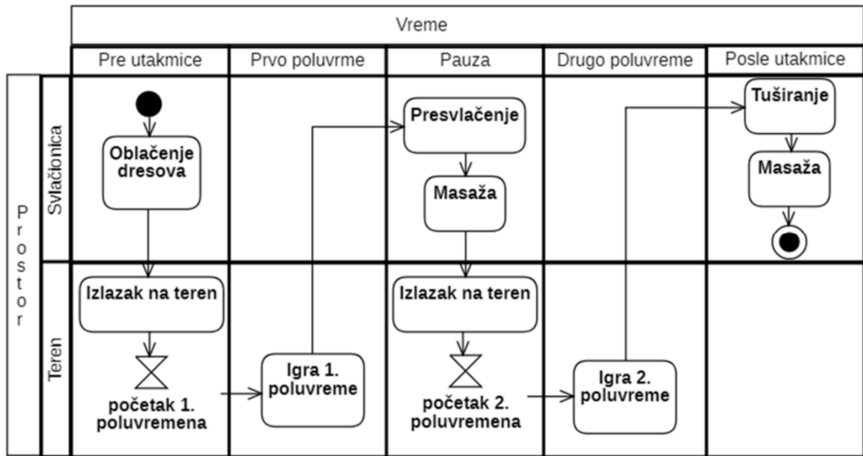
Particija 1.1 | Particija 1.2

dimenzija 2

Particija 2.1

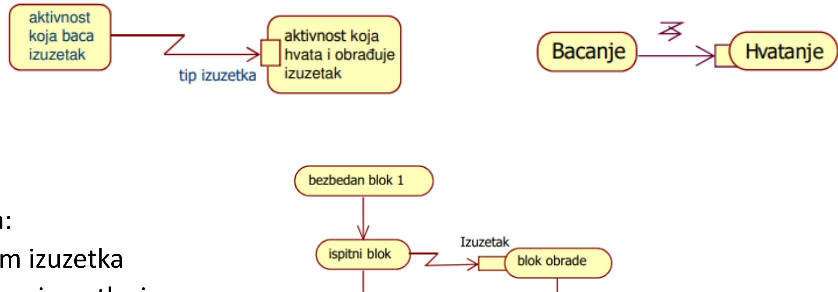
Particija 2.2

Primer – utakmica



Izuzeci

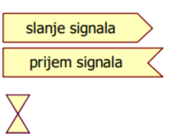
- Bacanje i obrada izuzetaka:



- Tok aktivnosti sa:
  - » bacanjem izuzetka
  - » hvatanjem izuzetka i
  - » nastavkom aktivnosti

Signali i događaji

- Grafička notacija za signale i događaje:
  - » slanje signala
  - » prihvatanje događaja
  - » prihvatanje vremenskog događaja npr. čekanje zadato vreme

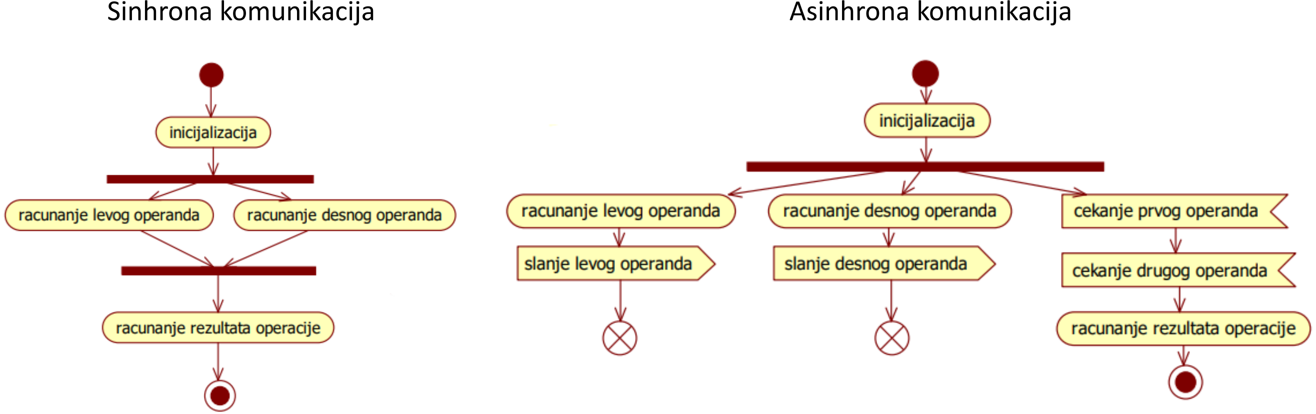


Primer slanja i prihvatanja signala

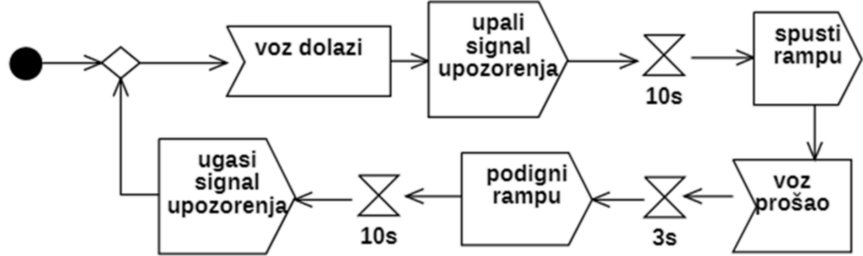
- Konkurentno izračunavanje binarne komutativne operacije

Sinhrona komunikacija

Asinhrona komunikacija

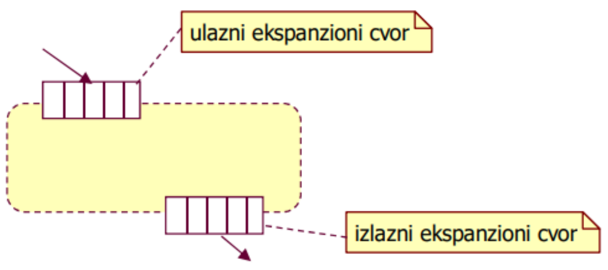


Primer – rampa na šinskom prelazu



Oblast ekspanzije

- Oblast strukturirane aktivnosti koja se izvršava više puta, u skladu sa elementima ulazne kolekcije (ekspanzionog čvora)
  - » izvršava se jednom za svaki element u ulaznoj kolekciji
- Grafička notacija:

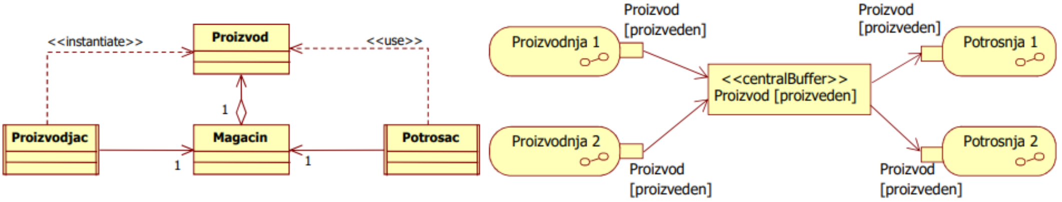


- Specijalne oblasti: ključne reči <<iterative>>, <<parallel>>

Centralni bafer

- Vrsta čvora objekta
- Namena: upravljanje tokovima objekata iz više izvora prema više odredišta
- Primer:

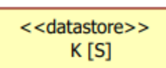
- Simbol:

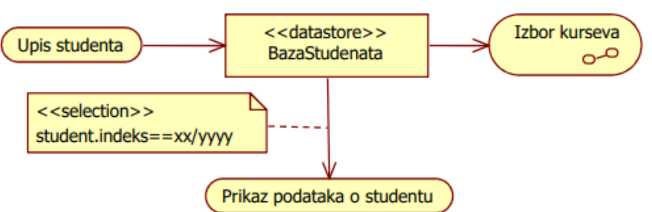


Skladište podataka

- Vrsta čvora objekata
- Namena: opis podataka koji su permanentno na raspolaganju
- Razlika u odnosu na centralni bafer – podaci centralnog bafera su prolazni, a podaci skladišta podataka su stalni
- Primer:

- Simbol:





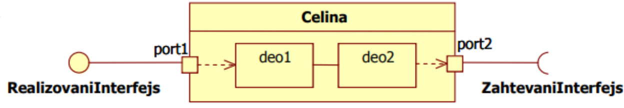
10 Dijagrami složene strukture

Uvod

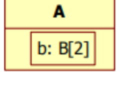
- Dijagrami složene strukture omogućavaju
  - » hijerarhijsku dekompoziciju klasifikatora na delove njegove unutrašnje strukture
  - » korišćenje događanja saradnji (collab. occurence) u saradnji (collaboration)
- Struktura – kompozicija povezanih elemenata
  - » elementi predstavljaju pojave koje sarađuju preko komunikacionih veza da postignu zajedničke ciljeve
- Unutrašnja struktura – struktura unutar pojave klasifikatora ili saradnje
- Port – tačka interakcije klasifikatora sa okruženjem
- Strukturirana klasa – klasa koja ima portove i unutrašnju strukturu
- Unutrašnju strukturu imaju
  - » klase – sadrže delove i portove povezane konektorima
  - » komponente – sadrže delove i portove povezane konektorima
  - » saradnje – sadrže uloge povezane konektorima i događanja saradnji

Notacija: delovi i portovi

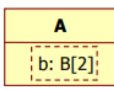
- Delovi klasifikatora se koriste da označe (povezane) osobine (atribute)
- Portovi se povezuju sa
  - » zahtevanim interfejsima (postolja, soketi)
  - » realizovanim interfejsima (loptice)
  - » unutrašnjim delovima



- Delovi mogu imati naznačene:
  - » ime
  - » tip
  - » multiplikativnost

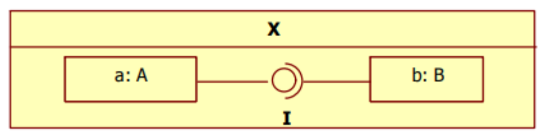


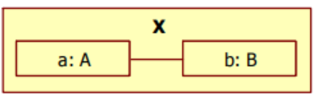
- Sadržanje dela:
  - » kompozicija – pravougaonik dela se crta punom linijom
  - » agregacija – pravougaonik dela se crta isprekidanom linijom



Konektori

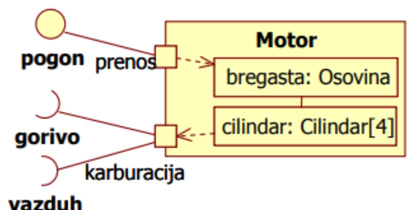
- Konektori povezuju
  - » delove međusobno
  - » delove sa portovima (delegirajući konektor)
- Konektor koji povezuje delove
  - » direktna veza (označava komunikaciju između delova - asocijacija)
  - » veza preko "lilihipa" – zahtevanog i realizovanog interfejsa



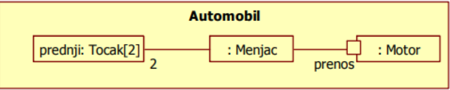


- Delegirajući konektor - povezuje delove sa portovima (zavisnost)
  - » od porta prema delu za realizovani interfejs
  - » od dela prema portu za zahtevani interfejs

Primer

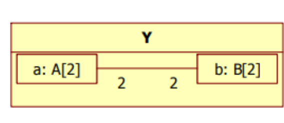


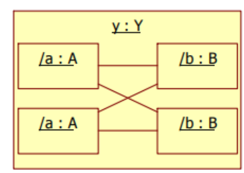
- Definisanje klase Automobil sa unutrašnjom strukturom
  - » klasa Motor
    - — realizovani interfejs (pogon)
    - — zahtevani interfejsi (gorivo, vazduh)
    - — portovi (prenos i karburacija)
    - — sadrži 4 cilindra i bregastu osovinu
  - » klasa Automobil sa unutrašnjom strukturom



Multiplikativnost

- Multiplikativnost se može naznačiti na svakom delu – u zagradama [] ili u gornjem desnom uglu
- Na krajevima konektora se može naznačiti multiplikativnost veze – tumači se kao i multiplikativnost na stranama asocijacije:
  - » broj objekata sa date strane konektora koji je u vezi sa tačno jednim objektom na drugoj strani veze (unutar jedne pojave okružujućeg klasifikatora)
- Primer:

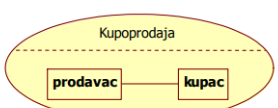


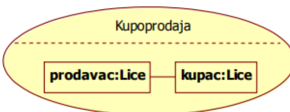


  - » pojave delova: a i b su imena uloga (oznaka /a, odnosno /b)

Saradnja

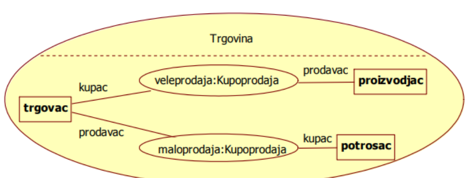
- Saradnja opisuje strukturu elemenata koji sarađuju (uloga) od kojih svaki obavlja specijalizovanu funkciju da bi zajedno postigli neku funkcionalnost
- Između uloga postoje komunikacione putanje - konektori
- Notacija saradnje:
  - » u gornjem delu elipse – ime saradnje
  - » ispod linije – uloge u saradnji povezane konektorima
  - » mogu biti naznačeni i tipovi uloga



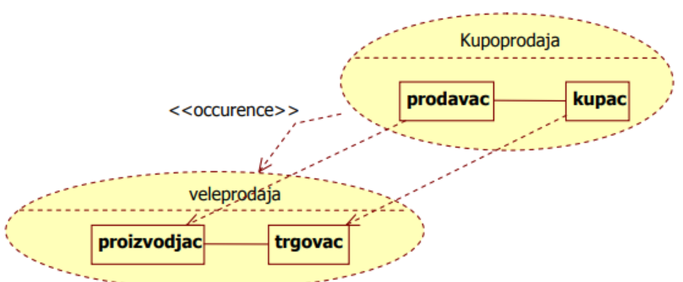


Korišćenje saradnje

- Notacija korišćenja saradnje (collaboration occurrence):
  - » u isprekidanoj elipsi
    ime_korišćenja_saradnje:ime_saradnje
- Primer:
  - » Trgovina je saradnja u kojoj se pojavljuju dva događanja saradnje Kupoprodaja: veleprodaja i maloprodaja
  - » uloge u trgovini imaju proizvodjac, trgovac i potrosac



- Drugi način prezentacije događanja saradnje – stereotip zavisnosti <<occurrence>>



11 Dijagrami komponenata

Uvod

- **Komponenta** je zamenjivi deo sistema koji realizuje skup interfejsa
- **Dijagram komponenata** prikazuje organizaciju i zavisnosti između komponenata
- Namenjen je prikazu organizacije softverskog sistema
- Predstavlja statički logički pogled na implementaciju sistema
- Dijagrami sadrže:
  - » stvari: komponente, artefakte, portove, interfejse, klase, pakete
  - » relacije: zavisnosti, generalizacije, asocijacije, realizacije

Najčešće primene

- Modeliranje:
  - » izvornog koda
  - » prevedenog koda
  - » izdanja za isporuku
  - » izvršnih izdanja i okruženja
  - » fizičkih baza podataka

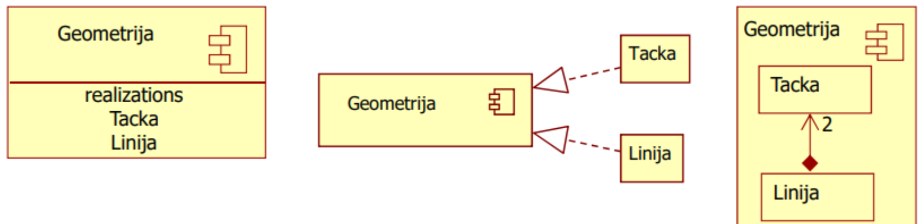
Komponenta

- Komponenta je modularni deo sistema
  - » manifestacija komponente je zamenjiva u okruženju
  - » kapsulira neki sadržaj (koji nije vidljiv osim kroz interfejse)
  - » definiše svoje ponašanje kroz ponuđene i zahtevane interfejse
- Komponenta predstavlja apstrakciju (logička stvar prema UML 2)
  - » obuhvata statičku i dinamičku semantiku
- Primeri komponenata – Java Bean, EJB, CORBA, COM+, .NET assembly
- Razvoj zasnovan na komponentama
  - » razvijani sistem se gradi i strukturira od postojećih komponenata
  - » bitna osobina – reupotreba ranije razvijenih komponenata

Grafička notacija

UML 1: UML 2:

- Komponenta može da sadrži i odeljak klasa koje realizuje



Artefakt

- **Artefakt** je fizička informacija koju koristi ili proizvodi
  - » razvojni proces ili
  - » izvršenje
- Primeri:
  - » modeli, izvorni fajlovi, skriptovi, binarni izvršni fajlovi, arhive, tabele baze podataka, dokumenti
- Može da predstavlja manifestaciju komponente
- Notacija:

Vrste i stereotipi artefakata

- Mogu se uočiti 3 vrste artefakata:
  - » iz razvojnog procesa - modeli, izvorni kod, projektni fajlovi, skriptovi, resursi, biblioteke
  - » za isporuku - exe, dll, jar, dokumenti, tabele
  - » izvršni - kreirani kao posledica izvršenja, npr. COM objekat kreiran iz DLL-a
- UML definiše sledeće standardne stereotipe za artefakte:
  - » `Executable` - komponenta koja se može izvršavati na čvoru
  - » `Library` - statička ili dinamička objektna biblioteka
  - » `File` - datoteka (proizvoljan sadržaj)
  - » `Document` - dokument
  - » `Script` - skript
  - » `Source` - datoteka sa izvornim kodom

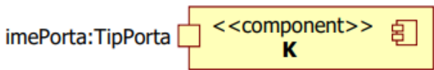
Komponente i klase/interfejsi

- Komponente i klase
  - » komponenta predstavlja "pakovanje" logičkih apstrakcija (klasa) u implementaciji
  - » klase implementiraju (realizuju) komponentu
  - » servisi klase mogu biti pristupačni direktno, a servisi komponente samo kroz interfejse
- Komponente i interfejsi
  - » interfejs je skup operacija koji specificira servis klase ili komponente
  - » komponenta realizuje jedan ili više interfejsa
  - » standardi COM, CORBA, EJB (Enterprise Java Beans) koriste interfejse

» relacija realizacije interfejsa (kanonička i skraćena forma):

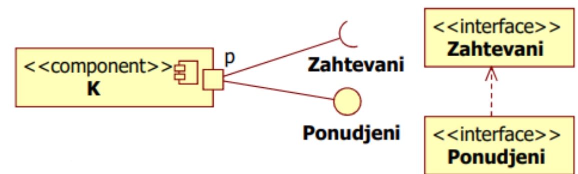
Port

- **Port** reprezentuje tačku interakcije između klasifikatora i njegovog okruženja
- Interfejsi pridruženi portu opisuju prirodu interakcija koje se dešavaju preko porta:
  - » zahtevani interfejsi opisuju zahteve koje može da pravi klasifikator svom okruženju
  - » ponuđeni interfejsi opisuju zahteve koje okruženje može da pravi klasifikatoru
- Komponenta potencijalno ispoljava interfejse preko portova
- Notacija:

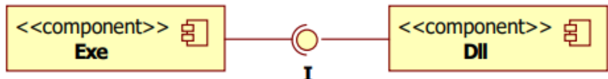


Port i interfejsi

- Port sa interfejsima (postolje/socket - zahtevani, loptica/ball - ponuđeni):
  - » Interfejs koji realizuje komponenta - izvozni (export) ili ponuđeni interfejs
  - » Interfejs koji komponenta koristi - uvozni (import) ili zahtevani interfejs



- Primer interakcije preko interfejsa:



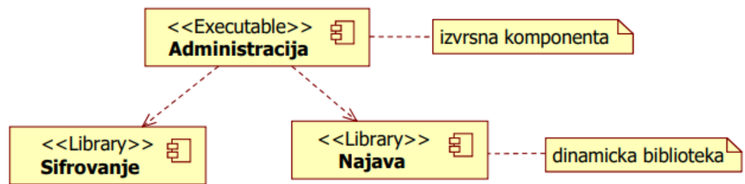
- Konstrukcija postolja i loptice se naziva veznik sklopa (assembly connector)
- Ako je port na ivici klasifikatora – vidljiv je spolja (public), inače je zaštićen (protected)
- Port može imati i multiplikativnost – piše se iza imena u zagradama [ ]

Paketi, podsistemi i relacije zavisnosti

- Paketi na dijagramima komponenata:
  - » sadrže druge pakete i logički povezane komponente
  - » koriste se i da predstave (fizičko) grupisanje artefakata
  - » kada sadrže artefakte, tipično reprezentuju kataloge (foldere) u sistemu datoteka
- Paketi iz dijagrama klasa se često preslikavaju u pakete na dijagramima komponenata
- U UML1 podsistemi su bili stereotip <<subsystem>> paketa
- U UML 2 podsistemi su stereotip komponente
- Paketi ili komponente se povezuju relacijom zavisnosti koja reprezentuje:
  - » zavisnost u vreme prevođenja kada se radi sa fajlovima izvornog koda
  - » zavisnost u vreme povezivanja kada se radi sa bibliotečkim i objektnim fajlovima
  - » zavisnost u vreme izvršenja kada se radi sa izvršnim fajlovima

Primer dijagrama komponenata

- Softver za administraciju nekog sistema
  - » izvršna komponenta Administracija
  - » za sifrovanje podataka koji se razmenjuju sa administriranim sistemom
    - — koristi se dinamička biblioteka (dll) Sifrovanje
  - » za najavu administratora na sistem
    - — koristi se dinamička biblioteka (dll) Najava



12 Dijagrami raspoređivanja

Uvod

- Dijagram raspoređivanja prikazuje
  - » statičke fizičke aspekte sistema
  - » hardversku i softversku izvršnu arhitekturu sistema
  - » konfiguraciju čvorova i softverskih artefakata koje žive na njima
- Dijagram raspoređivanja sadrži:
  - » stvari:
    - — čvorove (node)
    - — artefakte
    - — podsisteme
    - — pakete
  - » relacije:
    - — zavisnosti (između čvorova i artefakata)
    - — asocijacije (veze između čvorova)
    - — generalizacije

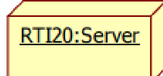
Najčešće primene

- Modeliranje:
  - » ugrađenih (embedded) sistema
  - » klijent-server sistema
  - » troslojnih sistema
  - » potpuno distribuiranih sistema

Čvorovi

- Čvor je apstrakcija fizičkog objekta koji reprezentuje resurs obrade
- Čvor u opštem slučaju ima barem memoriju, a često ima i mogućnost izvršavanja programa
- Čvorovi mogu biti procesori, uređaji, čak i izvršna okruženja (VM)
  - » procesor je čvor koji ima mogućnost izvršavanja programa
- Za višeprocesorske sisteme – multiplikativnost (u gornjem desnom uglu)
- Mogu postojati odeljci:
  - » atributa (npr. speed i memory)
  - » operacija (npr. ukljuci(), iskljuci(), samodijagnostika())
  - » artefakata koji se raspoređuju na čvor
- Čvor je apstrakcija (tip), pojave čvorova - podvučena imena

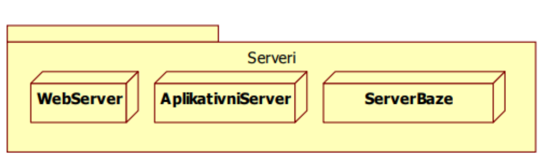
- Grafička notacija:



Organizovanje čvorova

- Čvorovi se mogu organizovati u pakete
- Grupisanje u paket se koristi za logički srodne čvorove
- Na primer:
  - » paket serveri sadrži razne tipove servera koji se koriste u sistemu
  - » paket klijenti sadrži razne tipove klijenata koji se koriste u sistemu
- Grupisanje može biti i fizičko - podsistem
  - » u UML 1 se koristio stereotip paketa <<subsystem>>
  - » u UML 2 <<subsystem>> je stereotip komponente

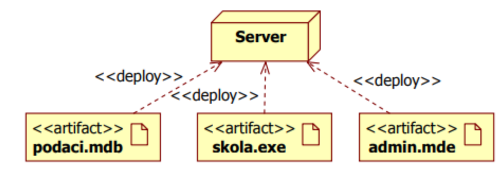
Klijenti
Desktop
Laptop



Čvorovi, komponente i artefakti

- Odnosi:
  - » komponente reprezentuju pakovanja logičkih elemenata (klasa)
  - » artefakti mogu biti manifestacije softverskih komponenata
  - » čvorovi reprezentuju fizičko odredište pri raspoređivanju artefakata
  - » procesori su čvorovi koje izvršavaju izvršne artefakte

Server
artifacts
podaci.mdb
skola.exe
admin.mde



- Notacija dozvoljava i da se artefakti koji se raspoređuju na neki čvor crtaju u njemu

Uređaji i izvršna okruženja

- Čvor može biti:
  - » fizički uređaj (npr. računar), stereotip <<device>> (podrazumevan)
  - » izvršno okruženje (npr. EJB kontejner), stereotip <<executionEnvironment>>
- Čvorovi mogu biti ugnežđeni
  - » uređaj može sadržati izvršno okruženje
  - » grafička notacija za ugnežđene čvorove:

<<device>>
AplikativniServer
<<executionEnvironment>>
J2EEserver

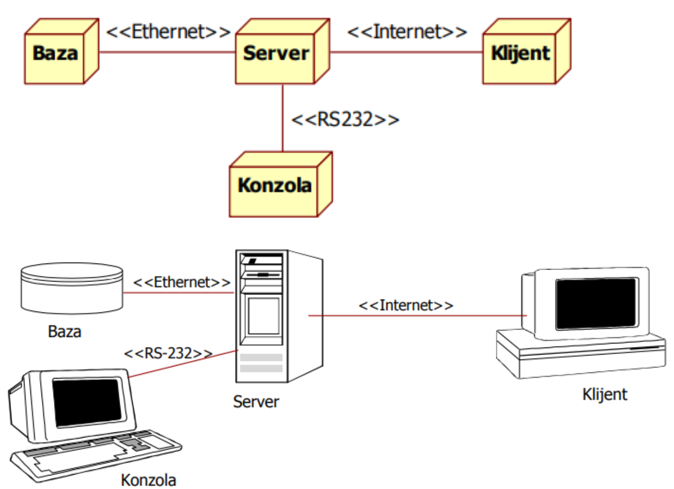
Veze

- Veza između čvorova ukazuje na komunikacioni put između njih
- Koristi se relacija asocijacije
- Veza može biti:
  - » direktna, kao što je RS232 serijska veza
  - » indirektna, kao što je komunikacija preko satelita
- Veze su obično bidirekcionalne
- Na asocijaciji se može ukazati na prirodu komunikacionog puta
  - » stereotip – Booch,..., “UML User Guide”
  - » naziv – Fowler, “UML Distilled”

<<RS232>>

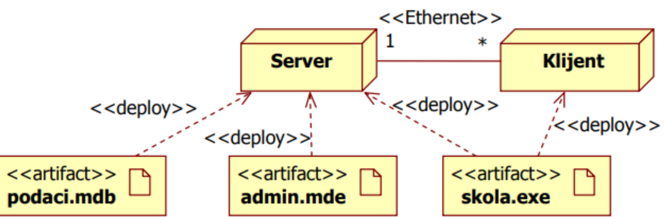
http/Internet

Primer – hardverska konfiguracija



Primer – čvorovi i artefakti

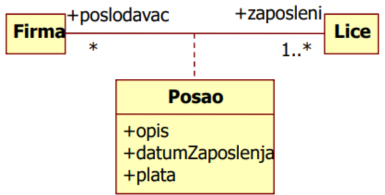
- Raspoređivanje artefakata na čvorove u lokalnoj mreži



13 Dijagrami klasa - napredni pojmovi

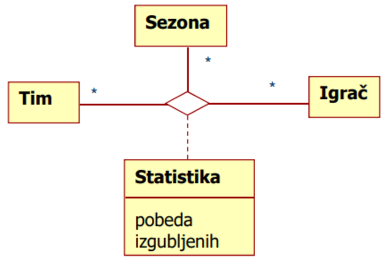
Klasa asocijacije

- Sama asocijacija može imati svoje atribute
- Ti atributi pripadaju klasi koja opisuje asocijaciju (slično veznoj tabeli kod relacionih baza)
- Primer:



N-arna asocijacija

- Asocijacija može povezivati više od 2 klase – takva asocijacija se naziva n-arnom
- Svaka pojava n-arne asocijacije je n-torka pojava odgovarajućih klasa
- Primer ternarne asocijacije (sa klasom asocijacije):



Kvalifikacija

- Koristi se da označi ključ (kvalifikator) koji se koristi za selekciju objekta iz neke strukture
- Objekat strukture za datu vrednost ključa selektuje jedan ili grupu elemenata
- Učesnici u relaciji su Struktura i njen Element
    - » element(i) strukture se izdvaja(ju) uz pomoć ključa
- Kvalifikator može imati više atributa, oni su atributi asocijacije
- Atributi kvalifikatora imaju istu sintaksu kao i atributi klasifikatora, sem inicijalne vrednosti
- Grafička notacija:
    - » ukras je deo asocijacije, ne klase

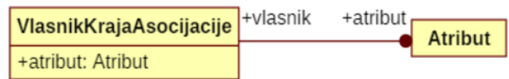
- Primeri struktura kojima se pristupa pomoću ključa su: heš tabela, B-stablo, …
- Multiplikativnost na kraju asocijacije kod klase Element:

» moguće kardinalnosti skupa selektovanih objekata uparivanjem sa objektom strukture uz zadatu vrednost ključa:
  — 0..1 – može da bude selektovan jedinstven objekat, ali postoje i vrednosti ključa za koje se ne selektuje ni jedan objekat
  — 1 – svaka moguća vrednost kvalifikatora selektuje jedinstven objekat elementa
  — * – vrednost kvalifikatora deli skup elemenata u podskupove

- Primeri:

Vlasništvo kraja asocijacije

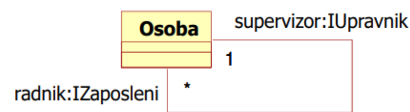
- Kraj asocijacije podrazumevano pripada asocijaciji
- Kraj asocijacije može pripadati klasi (sa druge strane asocijacije)
  » to je slučaj kada kraj asocijacije predstavlja atribut klase sa druge strane
- Označava se tačkicom („dot") na suprotnom kraju od vlasnika



- Ovde je izuzetno dozvoljena „redundansa":
  » predstavljeno vlasništvo kraja asocijacije tačkicom uz klasu
  » predstavljen i atribut u klasi vlasnika
- Standard ne obavezuje na eksplicitno korišćenje ove notacije

Specifikator interfejsa (UML 1)

- Uloge mogu implementirati samo neke od interfejsa koje realizuju klase u asocijaciji
- Ime interfejsa koji zadovoljava uloga - iza dvotačke koja sledi naziv uloge
- Primer 1:

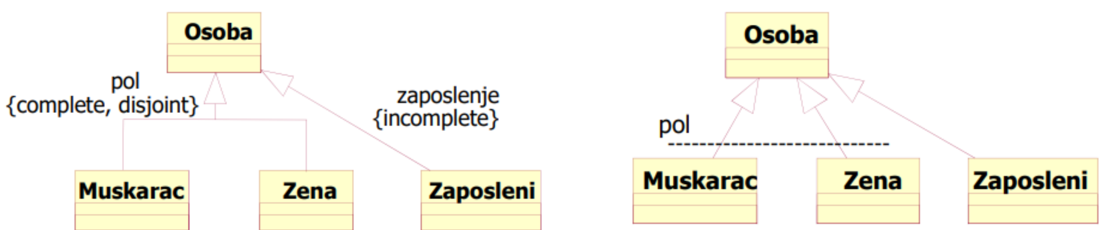


- Primer 2:

Generalizacioni skupovi

- Relacija generalizacije/specijalizacije uspostavlja odnos između posebnog i opšteg
- Ponekad se specijalizacija vrši po nekom klasifikacionom kriterijumu
  » kriterijum klasifikacije određuje podtipove
  » često objekti podtipova ne mogu biti i jedne i druge klase (ograničenje disjoint)
- Generalizacioni skup definiše poseban skup relacija generalizacije koji opisuje na koji način se natklasa specijalizuje
- Ograničenja generalizacionog skupa (pišu se u zagradama {}):
  » disjoint/overlapping
    — da li objekti podtipova mogu biti isključivo jednog od podtipova generalizacionog skupa
  » complete/incomplete
    — da li podtipovi predstavljaju potpuni skup mogućih podtipova

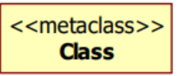
- Primer generalizacionog skupa pol:



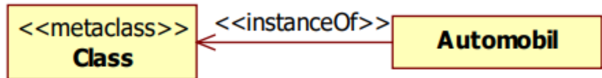
  » potklase klase Osoba mogu biti Muskarac, Zena i Zaposleni
  » generalizacije prema potklasama Muskarac i Zena pripadaju generalizacionom skupu pol

Metaklasa

- Metaklasa je klasifikator čiji su primerci klase
- Notacija: stereotip klase <<metaclass>>
- Primer:
  » metaklasa Class u jeziku UML opisuje apstrakciju klase



  » primerci ove metaklase su korisničke klase u konkretnom modelu



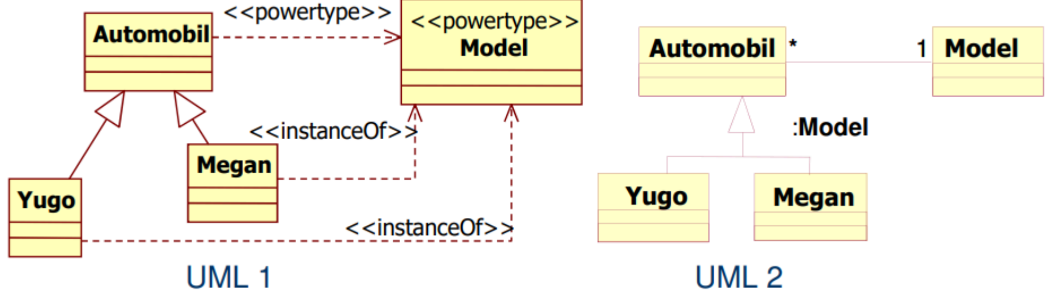
- Metamodel je model kojim se specificira jezik za modeliranje
  » UML je metamodel za konkretan model sistema koji se projektuje

Powertype

- Pojam relevantan za metamodeliranje
- Definicija powertype: klasifikator čiji su svi primerci deca (potklase) nekog roditelja
- Formalno:
  » ako je A tip, tada je Power(A) tip čiji su svi primerci podtipovi tipa A
  » ako je tip B primerak tipa Power(A), tada je B podtip A
- Powertype je metatip (tip u metamodelu), ali korisnički
  » primerci tog metatipa su tipovi (klase) koji su podtipovi nekog drugog tipa u modelu

Powertype - primer

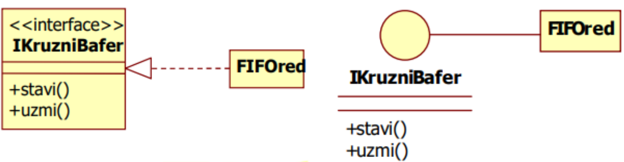
- Primer: instance Model (Yugo, Megan,...) su deca klase Automobil



- U gronjem primeru Model je metatip za klase Yugo, Megan
- UML 1:
  - » odredište stereotipa relacije zavisnosti <<powertype>> je powertype klasifikator
  - » stereotip zavisnosti <<instanceOf>> se koristri za veze klasa-->powertype (metatip)

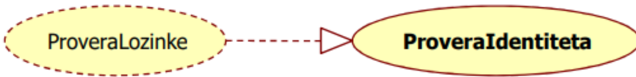
Konteksti relacije realizacije

- Realizacija se koristi u dva konteksta:
  - » kontekst interfejsa (realizuje ga klasa ili komponenta)
  - » kontekst slučajeva korišćenja (realizuje ga kolaboracija)
- Klasni dijagram:



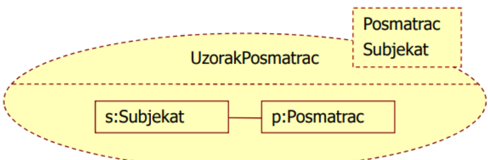
- Dijagram komponenata:

- Dijagram slučajeva korišćenja:

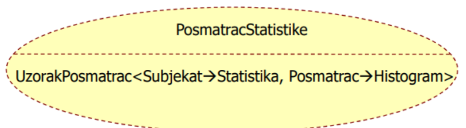


Parametrizovana saradnja

- Koristi se za opis projektnih uzoraka
- Definicija uzoraka "posmatrač" – parametrizovana saradnja:

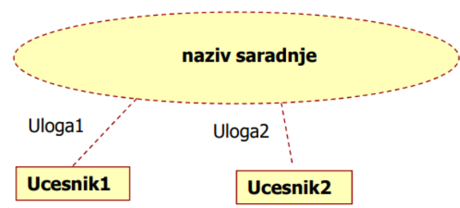


- Konkretna saradnja koja realizuje uzorak "posmatrač":

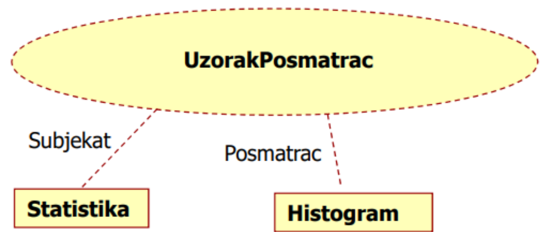


Projektni uzorci

- Drugi način za opis konkretne saradnje:



- Na prethodnom primeru:



Standardni stereotipovi klasifikatora

- **Utility** – klasa čiji su atributi i operacije zajednički za sve instance klase
- **Focus** – klasa koja implementira glavnu (poslovnu) logiku
- **Auxiliary** – klasa koja pomaže focus klasi u implementaciji logike
- <u>Stereotype</u> – klasifikator je stereotip koji se može primeniti na druge elemente
- **Metaclass** – klasifikator čije su instance klase
- *powertype* – klasifikator čije su instance deca datog roditelja (UML1)
- *thread* – klasa čiji su objekti aktivni sa deljenim adresnim prostorom (UML1)
- **Process** – **komponenta** čiji primerci imaju vlastite procese (u UML1 `process` je bio stereotip klase)

- Legenda:
  - » **naglašeno** – stereotip specificiran kao standardni i u UML 2 (profili L2 i L3)
  - » `obično` – nije naveden kao zastareo, ali nije naveden ni kao std. stereotip UML 2
  - » *kurziv* – stereotip eksplicitno naveden kao zastareo
  - » <u>podvučeno</u> – ključna reč u UML2

Standardni stereotipovi relacije zavisnosti

- Između klasa i/ili objekata na klasnom dijagramu:
  - » **Instantiate** – izvor stvara primerke odredišta
  - » `instanceOf` – izvor je objekat koji je primerak odredišnog klasifikatora
  - » **Send** – izvor (operacija) šalje signal koji je odredište relacije (odredište relacije nije primalac signala)
  - » **Derive** – izvor se može izračunati na osnovu odredišta; relacija između dva atributa ili dve asocijacije: jedan je konkretan, a drugi konceptualan (npr: DatumRodj i Starost)

- » **Refine** – izvor je finiji stepen apstrakcije od odredišta; na primer: klasa u projektu je na finijem stepenu apstrakcije od odgovarajuće klase u analizi
- » `permit` – izvoru se daju posebna prava pristupa odredištu
- » bind – izvor je generisan iz parametrizovanog šablona koji predstavlja odredište
- » *friend* – izvoru se daju posebna prava pristupa odredištu (samo UML 1)
- » *powertype* – odredište je powertype izvora (samo UML 1)

- Između paketa:
  - » access – izvorni paket ima pravo pristupa elementima odredišnog (privatni uvoz)
  - » import – javni sadržaj odredišnog paketa ulazi u prostor imena izvornog paketa (kao da su imena odredišnog paketa deklarisana u izvornom paketu)

Standardni stereotipovi generalizacije

- Standardni stereotip relacije generalizacije
  - » `implementation` – dete nasleđuje implementaciju roditelja, ali je čini privatnom i ne podržava interfejs roditelja, pa ne može zamenjivati roditelja

14. Dijagrami interakcije - napredni pojmovi

Dijagrami pregleda interakcije

- Dijagrami pregleda interakcije (Interaction Overview Diagrams)
  - definišu interakcije kroz jednu varijantu dijagrama aktivnosti
  - prikazuje se kontrola toka i interakcija u čvorovima aktivnosti
  - specijalizacija dijagrama aktivnosti da prikazuje interakcije u aktivnostima
- Čvorovi mogu biti
  - interakcije (dijagram sekvence ili komunikacije) ili
  - događanja interakcije – okvir sa ključnom rečju ref i imenom
- Interakcije i događanja interakcija
  - predstavljaju specijalne oblike izvršenja aktivnosti
- Na dijagramima se interakcije (sa linijama života i porukama) ređe pojavljuju
  - u najčistijem obliku sve aktivnosti su događanja interakcije i tada na dijagramu uopšte nema poruka i linija života

Umesto kombinovanih fragmenata

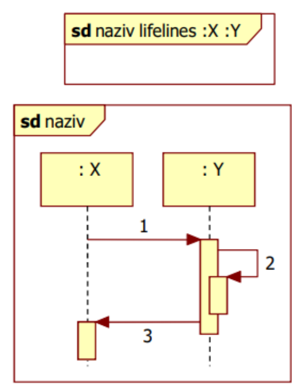
- Na dijagramima interakcije se koriste kombinovani fragmenti
- Na dijagramima pregleda interakcije nema potrebe za njima
  - zamenjuju se kontrolom toka uobičajenom za dijagrame aktivnosti
- Kombinovani fragmenti opcionih (opt) i alternativnih (alt) tokova interakcije
  - prikazuju se čvorom odluke (decision) i odgovarajućim čvorom spajanja (merge)
- Kombinovani fragmenti petlji (loop) u interakciji
  - prikazuju se pomoću čvorova odluke i spajanja, kao i povratnih grana u toku kontrole
- Kombinovani fragmenti paralelnih (par) tokova interakcije
  - prikazuju se čvorom konkurentnog grananja (fork) i spajanja tokova (join)
- Dijagrami pregleda interakcije se uokviruju istim vrstama okvira koji zatvaraju druge forme dijagrama interakcije;
  - tekst zaglavlja može da uključuje listu sadržanih linija života
  - linije života ne moraju da se pojavljuju grafički na dijagramu pregleda interakcije
  - linije života mogu da figurišu u interakcijama na koje upućuju događanja interakcije

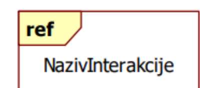
Grafička notacija

- Osim simbola sa dijagrama aktivnosti koriste se sledeći:
  - okvir interakcije (oko celog dijagrama) sa eventualnim imenom i nazivima linija života:
  - interakcija kao čvor (pokretanje aktivnosti) može biti imenovana, ili anonimna
  - objekti se ne pojavljuju na dijagramima pregleda interakcije

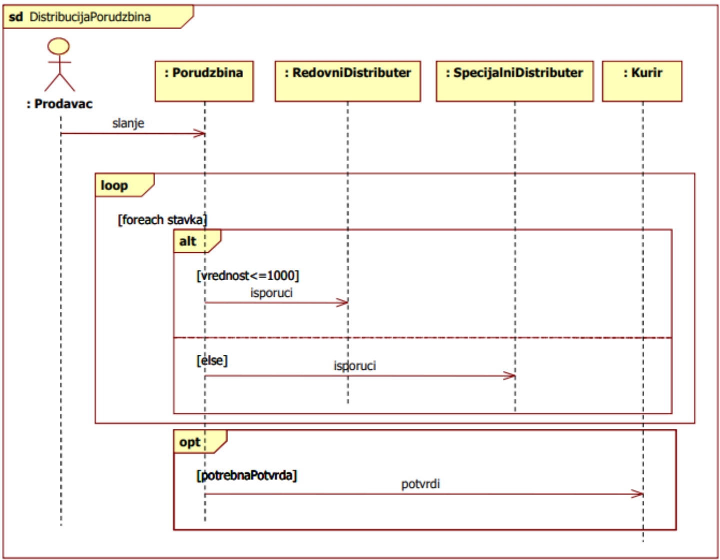
  - događanje interakcije - alatima se prepušta način prikaza:
    - referenca
    - ekspandovana i ugrađena interakcija

sd naziv lifelines :X :Y

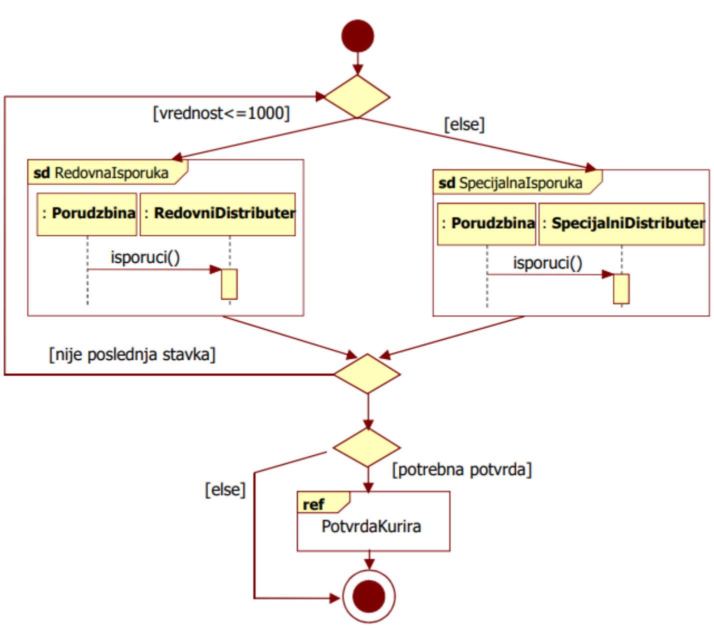




PRIMER – DIJAGRAM INTERAKCIJE



PRIMER – DIJAGRAM PREGLEDA INTERAKCIJE



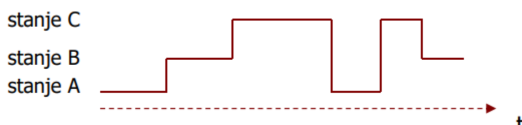
VREMENSKI DIJAGRAMI

- Prikazuju interakciju sa fokusom na vremensku dimenziju
- Vremenski dijagrami prikazuju promenu uslova/stanja unutar i između linija života duž linearne vremenske ose
- Zamišljena vremenska osa je horizontalna
- Vremenski dijagrami opisuju ponašanje
  - » individualnih klasifikatora
  - » interakcije klasifikatora

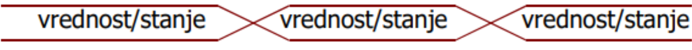
- Pažnja se fokusira na vremena zbivanja događaja koji uzrokuju promene stanja/uslova posmatranih linija života

Grafička notacija

- Okvir interakcije (oko dijagrama) – ključna reč sd
- Vremenska linija stanja/uslova



- Vrednosti/stanja linije života



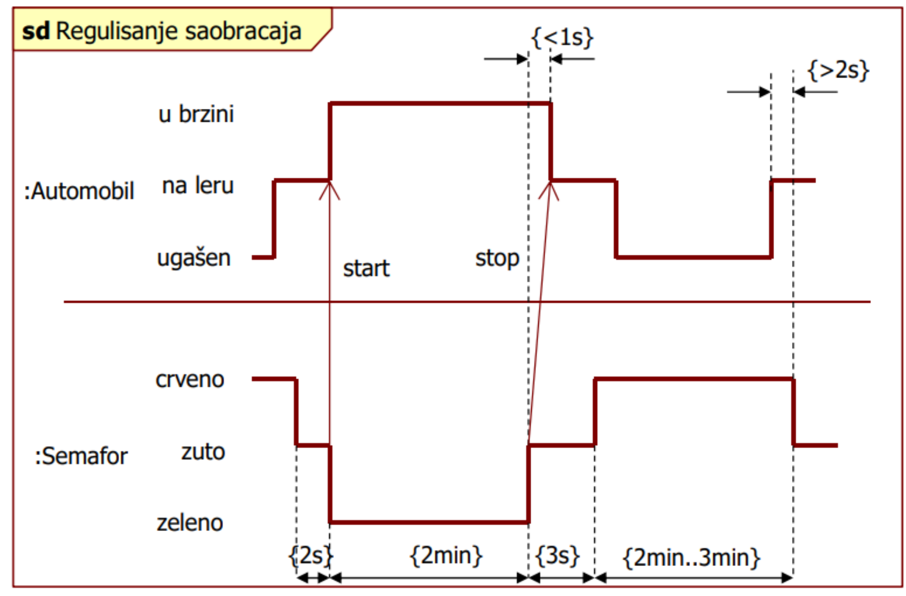
- Linije života

uloga1

uloga2

- Poruke (kao na dijagramima sekvence, ali su linije vertikalno orijentisane)
- Vremenska ograničenja intervala i trenutaka (u { }), na primer:
  - » {<5*d}, ili {d..3*d} gde je d interval vremena
  - » {t..t+3}, gde je t vremenski trenutak nekog događaja

Primer vremenskog dijagrama



Arhitektura metamodeliranja

Uvod

- UML je jedan od nivoa 4-nivoske arhitekture metamodeliranja
- 4-nivoska arhitektura je dokazana infrastruktura za definisanje precizne semantike koju zahtevaju kompleksni modeli
- Prednosti pristupa 4-nivoske arhitekture:
  - rafinira semantičke konstrukcije njihovom rekurzivnom primenom na sukcesivne metanivoe
  - obezbeđuje arhitektonsku osnovu za:
    - definisanje budućih proširenja UML metamodela
    - nivelaciju UML metamodela sa drugim standardima zasnovanim na 4-nivoskoj arhitekturi metamodeliranja, posebno OMG Meta-Object Facility (MOF)

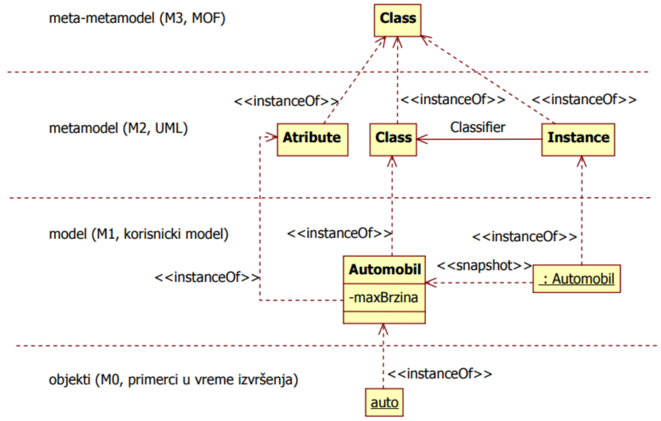
Četvoronivoska arhitektura

- Odnos između 2 susedna nivoa – niži nivo je instanca višeg nivoa
- Broj meta-nivoa je teorijski neograničen, ali su za praktične primene dovoljna 4 nivoa
- Opšte prihvaćeni radni okvir za metamodeliranje je zasnovan na arhitekturi sa sledeća četiri nivoa (sloja):
  - meta-metamodel
  - metamodel
  - model
  - korisnički objekti

Opis nivoa arhitekture

Nivo	Opis	Primer
meta-metamodel	Infrastruktura za arhitekturu metamodeliranja. Definiše jezik za specificiranje metamodela.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	Jedna instanca meta-metamodela. Definiše jezik za specificiranje modela.	*Class, Attribute, Operation, Component*
model	Jedna instanca metamodela. Definiše jezik za opis nekog informacionog domena.	*Dokument, velicina, otvaranje()*
korisnički objekti	Jedna instanca modela. Definiše neki specifičan informacioni domen.	*Dokument dokument = new Dokument("Text.rtf"); int i=100;*

Primer



Meta-metamodel

- Nivo meta-metamodeliranja daje temelj za arhitekturu metamodeliranja
- Primarna odgovornost nivoa meta-metamodela: da definiše jezik za specificiranje metamodela
- Meta-metamodel definše model na višem stepenu apstrakcije od metamodela i tipično je kompaktniji od metamodela koji opisuje
- Jedan meta-metamodel može definisati više metamodela, i može biti više meta-metamodela pridruženih svakom metamodelu
- Generalno je poželjno da povezani metamodeli i meta-metamodeli dele zajedničke projektne filozofije i konstrukte – ali ovo nije striktno pravilo
- Svaki nivo treba da održava svoj sopstveni projektni integritet
- Primeri meta-metaobjekata u sloju meta-metamodeliranja su: MetaClass, MetaAttribute i MetaOperation

Metamodel

- Metamodel je jedan primerak meta-metamodela
- Primarna odgovornost nivoa metamodela: da definiše jezik za specificiranje modela
- Metamodeli su tipično detaljniji od meta-metamodela koji ih opisuju, specijalno kada definišu dinamičku semantiku
- Primeri metaobjekata u sloju metamodeliranja su: Class, Attribute, Operation i Component

Model

- Model je jedan primerak metamodela
- Primarna odgovornost nivoa modela: da definiše jezik koji opisuje neki informacioni domen
- Primeri objekata u sloju modeliranja su: Dokument, velicina, otvaranje()

Korisnički objekti

- Korisnički objekti su jedan primerak modela
- Primarna odgovornost nivoa korisničkih objekata je da opišu specifičan informacioni domen
- Primeri objekata u sloju korisničkih objekata su:
  - » `Dokument dokument = new Dokument("Text.rtf") , int i=100`

Analogija

- Nije samo arhitektura OO meta-modeliranja 4-nivoska
  - » i arhitektura proceduralnog projektovanja i realizacije je 4-nivoska:

- Metajezik (meta-metamodel):
  - » stanje (podaci): primitivni tip, niz, struktura
  - » ponašanje (kontrola toka): sekvenca, selekcija, iteracija
- Jezik (metamodel):
  - » podaci: `int`, `[]`, `struct`
  - » kontrola toka: `{nar1; nar2;…}`, `if-else`, `for`, `while`
- Program (model):
  - » podaci: `typedef unsigned int Uint; struct S{int c; float n[2];}`
  - » kontrola toka: `if (a) {b=c; while(d<100){ }} else { }`

PROJEKTNI UZORCI

Uvod u projektne uzorke

Uvod

- Christopher Alexander govori o uzorcima u građevinskoj arhitekturi: “Svaki uzorak opisuje neki problem koji se ponavlja u našem okruženju i tada opisuje jezgro rešenja tog problema na takav način da se to rešenje može koristiti milion puta, a da se ne uradi ni dva puta na isti način”
- Definicija je primenljiva i na uzorke u objektno-orijentisanim sistemima
- Projektni uzorak predstavlja zabeleženo široko primenljivo iskustvo u projektovanju sistema
- Termini: projektni uzorak, uzor, obrazac, šablon (engl. Design Pattern)

O objektno-orijentisanim (OO) projektnim uzorcima

- OO projektni uzorci su opisi komunicirajućih objekata i njihovih klasa koji su prilagođeni da reše neki opšti projektni problem u posebnom kontekstu
- Karakteristike OO projektnog uzorka:
    - » identifikuje učestvujuće klase i objekte, njihove relacije, njihove uloge u saradnji i raspodelu odgovornosti
    - » sistematično imenuje, objašnjava i ocenjuje važno projektno rešenje ponavljajućeg problema u OO sistemima
    - » koristan je za kreiranje ponovo upotrebljivog OO projektnog rešenja
- Svaki projektni uzorak ima 4 bitna elementa:
    - » naziv uzorka
    - » postavku problema
    - » opis rešenja
    - » diskusiju posledica

Naziv uzorka

- Koristi se da opiše projektni problem, njegovo rešenje i posledice primene u par reči
- Imenovanje uzorka proširuje projektni rečnik:
    - » omogućava projektovanje na višem nivou apstrakcije
    - » pojednostavljuje komunikaciju u timu
    - » olakšava dokumentovanje projekta
- Ponekad se u literaturi sreće i više imena za isti uzorak

Postavka problema

- Objašnjava opšti problem
- Navodi motivaciju za primenu uzorka – na primeru u nekom posebnom kontekstu
- Opisuje:
    - » uopšten projektni problem, npr. kako reprezentovati algoritme kao objekte
    - » strukture klasa ili objekata simptomatične za nefleksibilni dizajn
    - » uslove koji se moraju ispuniti za primenu uzorka

Opis rešenja

- Opisuje elemente koji predstavljaju jezgro rešenja – njihove uloge, relacije, odgovornosti i saradnje
- Rešenje ne opisuje konkretan dizajn ili implementaciju
- Rešenje treba da posluži kao šablon – da se može primeniti u konkretnim slučajevima
- Rešenje predstavlja apstraktni opis načina aranžiranja elemenata (klasa i/ili objekata)

Posledice

- Posledice su rezultati primene uzorka
- Bitne su za razmatranje projektnih alternativa i razumevanje cene i dobiti primenom uzorka
- Posledice se često odnose na vreme i prostor
- Ponekad se odnose i na jezik i implementacione stvari
- Posledice najčešće uključuju uticaj na:
    - » prilagodljivost (fleksibilnost) sistema
    - » proširivost (ekstenzibilnost) sistema
    - » prenosivost (portabilnost) sistema

Primer primene uzoraka: MVC

- MVC je arhitekturni okvir za razvoj aplikacija – skraćeni naziv za trijadu klasa **M**odel, **V**iew i **C**ontroller
- Poreklo – realizacija korisničkih interfejsa u Smalltalk jeziku
- Model – konkretan objekat domenske (poslovne) logike
- View – ekranska prezentacija objekta
- Controller – opis reakcije korisničkog interfejsa na ulaz korisnika
- Interakcija (ponašanje): *Model-View*, *View-Controller*
- Strukturiranje: View
- Primeri za 3 projektna uzorka

Model – View komunikacija

- Prikaz mora da obezbedi da reflektuje aktuelno stanje modela
- Kad se podaci modela promene:
    - » model signalizira promenu prikazima koji zavise od njega
    - » svaki prikaz dobija priliku da se ažurira
- Pristup omogućava više uzajamno nezavisnih pogleda na model – svaki pogled je prezentacija posebnog aspekta modela
- Mogu se jednostavno dodavati novi prikazi – ne menja se model i ne menjaju se drugi prikazi
- Primer: model sadrži neke statističke podatke – postoje tri prikaza: tabelarni, histogram, pita
- Protokol između View i Model-a je tzv. subscribe-notify:
    - » prikazi se pretplaćuju kod modela na obaveštavanje
    - » model obaveštava pretplaćene prikaze kada se dogodi promena
    - » kada dobiju obaveštenje (signal) prikazi čitaju novo stanje modela
    - » prikazi prilagođavaju svoj izgled novom stanju modela
- Generalizacija problema obaveštavanja "pretplatnika" – promene jednog objekta treba da utiču na proizvoljan broj drugih – nema potrebe da menjani objekat zna za detalje drugih
- Rešenje generalnog problema
    - » opisano uzorkom Posmatrač (Observer)
    - » uloge: subjekat (MVC Model) i posmatrač (MVC View)

Komponovanje prikaza

- Jedna od mogućnosti MVC arhitekture – prikazi (Views) mogu biti ugnežđeni
- Primer: kontrolni panel je prikaz koji sadrži ugnežđene prikaze dugmadi
- Klasa CompositeView izvedena iz View
    - predstavlja vrstu prikaza koji sadrži druge prikaze
    - može se pojaviti gde god se očekuje objekat osnovne klase View
    - dobija se objektna hijerarhija (stablo) prikaza
- Generalizacija problema strukturno složenih objekata (sklopova) – grupisanje objekata gde je grupa istog (nad)tipa kao i pojedini objekat
- Rešenje generalnog problema – opisano uzorkom Kompozicija (Sastav, Sklop, engl. Composite)
- Sličan je odnos klasa Container i Component u Javi

View – Controller komunikacija

- MVC omogućava promenu načina reakcije View na ulaz korisnika
- Primer: moguće je redefinisati odgovore na događaje sa tastature/miša i vršiti istu obradu pritiska na ekransko dugme, taster i stavku menija
- MVC kapsulira mehanizam odgovora u objekat tipa Controller
- Postoji hijerarhija klasa kontrolera koja olakšava kreiranje novog – novi kontroler se kreira kao varijacija nekog postojećeg
- View koristi objekat potklase Controller – objekat kontrolera specificira strategiju odgovora
- Za promenu strategije odgovora – samo se zamenjuje objekat kontrolera drugom vrstom kontrolera
- Ekransko dugme i stavka menija mogu da koriste isti kontroler
- Izmenu strategije je moguće vršiti i u vreme izvršenja
- Primer: objektu klase View može da se onemogući interakcija (disabled stanje) tako što mu se dodeli kontroler koji ignoriše ulaze
- Generalizacija problema promenljivog algoritma
    - statička ili dinamička izmena algoritma
    - algoritam definiše ponašanje nekog objekta (konteksta)
- Rešenje generalnog problema
    - opisano uzorkom Strategija (Strategy)
    - objekti strategije apstrahuju i kapsuliraju algoritam ponašanja
    - lako se zamenjuju u nekom kontekstu (promena pokazivača)

Klasifikacija projektnih uzoraka

- Namena i nivo apstrakcije varira kod različitih uzoraka
- Kao i svaka druga klasifikacija, ova klasifikacija doprinosi boljem i bržem snalaženju pri traženju odgovarajućeg uzorka i usmerava napore ka otkrivanju novih uzoraka
- Klasifikacija koristi dva kriterijuma za razvrstavanje uzoraka
    - kriterijum namene - šta opisuje uzorak: stvaranje, strukturu ili ponašanje
    - kriterijum domena, odnosno nivoa apstrakcije -koji nivo apstrakcije opisuje uzorak: klasni/objektni
- Klasifikacija nije strogo formalna, pa tako ni sasvim precizna – zasniva se u dobroj meri na intuitivnoj proceni – ipak, pomaže razumevanju prirode projektnog uzorka

Kriterijum namene

- Kriterijum deli uzorke prema nameni – odražava čime se uzorak bavi (šta opisuje)
- Namena uzorka može biti da opiše:
    - kreiranje (creational patterns) - uzorci tretiraju stvaranje (proizvodnju) objekata
    - strukturu (structural patterns) - uzorci tretiraju kompozicije objekata ili klasa
    - ponašanje (behavioral patterns) - uzorci tretiraju načine interakcije objekata ili klasa

Kriterijum domena

- Kriterijum deli uzorke prema nivou apstrakcije – da li se uzorak fokusira na apstrakcije (klasni domen) ili na primerke apstrakcije (objektni domen)
- Klasni uzorci (class patterns)
  - » fokusiraju se na relacije između klasa i potklasa
  - » ove relacije su generalizacije/specijalizacije
  - » one su statičke – fiksirane u vreme prevođenja
- Objektni uzorci (object patterns)
  - » fokusiraju se na relacije između objekata
  - » ove relacije su uglavnom primerci asocijacija (veze)
  - » one su dinamičke – mogu menjati u vreme izvršenja
- Većina uzoraka je u objektnom domenu

Karakteristike vrsta uzoraka

- Klasni uzorci kreiranja – klase delegiraju stvaranje objekata (ili njihovih delova) potklasama
- Objektni uzorci kreiranja – neki objekti delegiraju stvaranje objekata (ili njihovih delova) drugim objektima
- Klasni uzorci strukturiranja – komponuju klase kroz speicjalizaciju i koriste nasleđivanje implementacije
- Objektni uzorci strukturiranja – opisuju načine asembliranja (sklapanja celina od nekih) objekata
- Klasni uzorci ponašanja – komponuju klase kroz specijalizaciju da kompletiraju algoritme/tok kontrole
- Objektni uzorci ponašanja – opisuju kako grupa objekata sarađuje da obavi neki zadatak

Prostor projektnih uzoraka

- Tabela prikazuje klasifikaciju uzoraka po dva kriterijuma: kolone sadrže vrste uzoraka po nameni, a vrste sadrže vrste uzoraka po domenu

		Namena		
		uzorci kreiranja	**uzorci strukture**	**uzorci ponašanja**
Domen	**klasni uzorci**	Fabrički metod	Adapter (klasni)	Interpreter Šablonski metod
	objektni uzorci	Apstraktna fabrika Graditelj Prototip Unikat	Adapter (objektni) Most Sastav Dekorater Fasada Muva Zastupnik	Lanac odgovornosti Komanda Iterator Posrednik Podsetnik Posmatrač Stanje Strategija Posetilac

Odnosi između uzoraka

- Postoje i drugi načini za organizovanje uzoraka:
  - » neki uzorci se često koriste zajedno npr. Kompozicija (Composite) i Iterator (Iterator) ili Posetilac (Visitor)
  - » neki uzorci su alternative jedni drugima npr. Prototip (Prototype) i Fabrički metod (Factory Method)
  - » neki uzorci različitih namena imaju sličnu klasnu strukturu npr. Kompozicija (Composite) i Dekorater (Decorator)

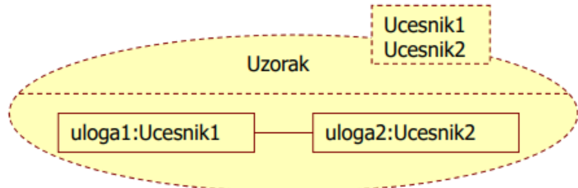
    - neki uzorci različitih namena imaju sličnu objektnu strukturu npr. objektni Adapter i Dekorater (Decorator)

Katalog projektnih uzoraka

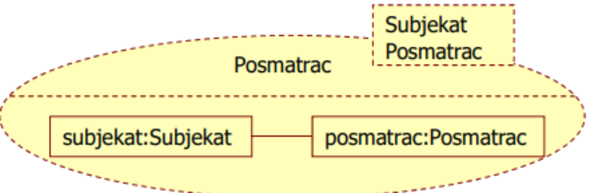
- Katalog sadrži za svaki uzorak sledeće stavke:
    - ime uzorka i klasifikacija
    - namena – odgovori na pitanja "šta radi? čemu služi? koji problem rešava?"
    - drugi nazivi – isti uzorak može imati više naziva
    - motivacija – scenario koji ilustruje projektni problem
    - primenljivost – situacije u kojima se uzorak može primeniti (da se izbegne loš dizajn)
    - struktura – klasni (eventualno i objektni) dijagram koji opisuje uzorak
    - učesnici – klase i objekti koji učestvuju u uzorku i njihove odgovornosti
    - saradnje – kako učesnici sarađuju da ostvare svoje odgovornosti (eventualno d. interakcije)
    - posledice – diskusija dobrih i loših strana primene uzorka
    - implementacija – zamke, preporuke i tehnike kojih treba biti svesan pri implementaciji
    - primer koda – fragment koda koji ilustruje kako se uzorak može implementirati
    - poznate primene – primeri primene uzorka u realnim sistemima (barem po dva)
    - povezani uzorci – koji uzorci su bliski sa datim, koje su razlike, sa kojima se često koristi
    - *UML notacija – grafički simbol za saradnju koja realizuje uzorak*

Definicija uzoraka u UML notaciji

- Definicija uzorka (strukturirana parametrizovana saradnja):

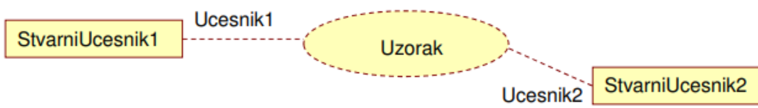


- Formalni parametri parametrizovane saradnje: – tipovi uloga učesnika u definiciji projektnog uzorka
- Primer:

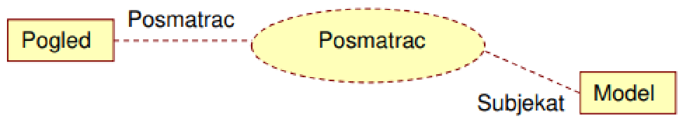


Primena uzoraka u UML notaciji

- Primena uzorka:



- Argumenti parametrizovane saradnje:
    - tipovi stvarnih učesnika
    - konkretne klase u modelu
- Relacija između saradnje i stvarnih učesnika je vezivanje (binding) – isprekidana linija na kojoj se navodi formalni tip učesnika iz definicije
- Primer:


Pogled
Posmatrac
Posmatrac
Subjekat
Model

Unikat

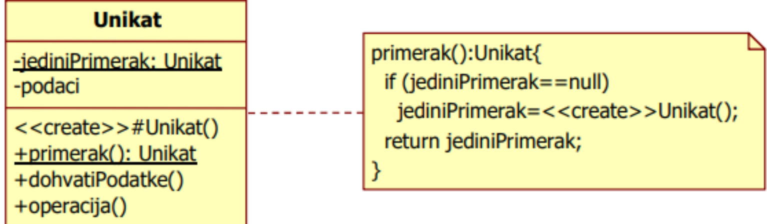
- *Ime i klasifikacija:* Unikat (Singleton) – objektni uzorak stvaranja

- *Namena:* – obezbeđuje da klasa ima samo jedan objekat i daje globalni pristup tom objektu

- *Motivacija:*
  - » za neke klase je važno obezbediti da imaju samo po jedan objekat
  - » na primer, u sistemu treba da postoji samo jedan dispečer štampe (spooler)
  - » globalna promenljiva obezbeđuje globalan pristup, ali ne brani više objekata
  - » rešenje je da klasa sama bude odgovorna za jedinstvenost njenog objekta

- *Primenljivost:* uzorak treba koristiti kada
  - » mora postojati tačno jedan objekat klase
  - » on mora biti pristupačan klijentima preko svima poznate tačke pristupa
  - » klasa treba da bude proširiva izvođenjem

- *Struktura:*



- *Učesnici:*
  - » Unikat
    - — definiše operaciju primerak() kao operaciju klase (statički metod)
      - ▪ operacija daje klijentima pristup do jedinstvenog objekta
      - ▪ operacija je odgovorna je za kreiranje jedinog objekta
    - — zaštićeni konstruktor sprečava da klijenti kreiraju objekte klase

- *Saradnje:*
  - » klijenti dohvataju objekat klase Unikat isključivo kroz operaciju primerak()

- *Posledice:* dobre strane
  - » kontrolisani pristup do jedinog objekta - pošto klasa kapsulira jedini objekat, ona može imati striktnu kontrolu nad tim kako i kada klijenti pristupaju objektu
  - » rasterećenje prostora imena - uzorak Unikat je bolji koncept od globalne promenljive jer izbegava opterećivanje prostora imena globalnom promenljivom koja čuva jedini objekat
  - » dozvoljeno doterivanje operacija - iz klase Unikat se može izvoditi - lako je objekat izvedene klase zameniti, čak i u vreme izvršenja
  - » moguće kontrolisano povećanje broja objekata - projektant može da po maloj ceni poveća broj dozvoljenih objekata
  - » fleksibilnost u odnosu na uslužnu (utility) klasu
    - — sa uslužnom klasom je praktično nemoguće promeniti broj objekata
    - — statičke funkcije nisu virtuelne, potklase ne mogu da ih polimorfno redefinišu

- Implementacija C++:

```
class Unikat {
public:
	static Unikat* primerak();
	virtual void operacija();
protected:
	Unikat();
private:
	static Unikat* jediniPrimerak;
}

// implementacija:
Unikat* Unikat::jediniPrimerak=nullptr;
Unikat* Unikat::primerak(){
	if (jediniPrimerak == nullptr) jediniPrimerak = new Unikat;
	return jediniPrimerak;
}

// koriscenje:
Unikat::primerak()->operacija();
```

- Implementacija Java:

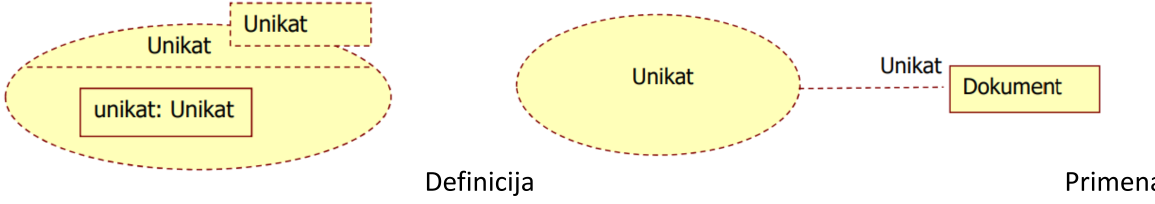
```
class Unikat{
	public static Unikat primerak(){
		if (jediniPrimerak == null) jediniPrimerak = new Unikat();
		return jediniPrimerak;
	}
	public void operacija(){/*...*/};
	protected Unikat(){}
	private static Unikat jediniPrimerak=null;
};

// koriscenje:
Unikat.primerak().operacija();
```

- *Poznate primene:*
  - » u aplikacijama sa jednim dokumentom (single-document) klasa Dokument je unikat
  - » u Smalltalk jeziku svaka metaklasa (klasa čiji su primerci klase) ima samo jedan objekat

- *Povezani uzorci* – mnogi uzorci koriste Unikat (Apstraktna fabrika, Graditelj, Fasada)

- *UML notacija:*



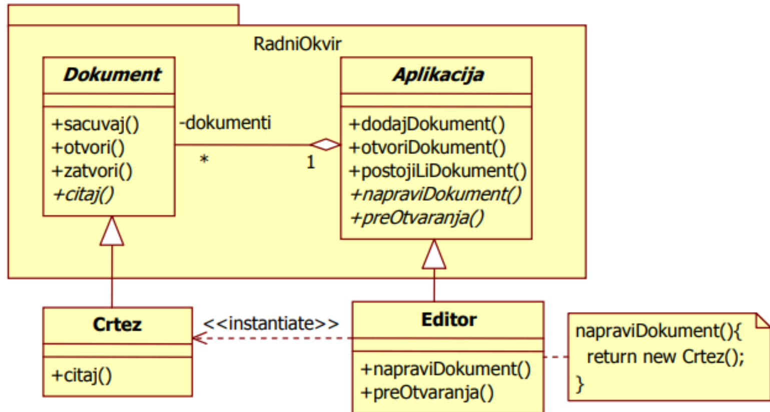
Definicija Primena

Šablonski metod

- *Ime i klasifikacija:* Šablonski metod (engl. Template Method) - klasni uzorak ponašanja

- *Namena:*
  - » definiše skelet nekog algoritma operacije, delegirajući pojedine korake potklasama
  - » omogućava potklasama da redefinišu neke korake algoritma bez izmene njegove strukture

- *Motivacija:*
  - » razmatra se radni okvir za razvoj aplikacije koji obezbeđuje klase Aplikacija i Dokument



  - » klasa Aplikacija je odgovorna za otvaranje postojećih dokumenata
  - » dokumenti se čuvaju u nekom eksternom formatu (fajlu)
  - » objekt klase Dokument reprezentuje informaciju u dokumentu nakon što je učitana iz fajla
  - » aplikacije koje se grade iz radnog okvira mogu da naslede klase Aplikacija i Dokument
    - — npr. aplikacija za crtanje definiše potklase Editor i Crtež

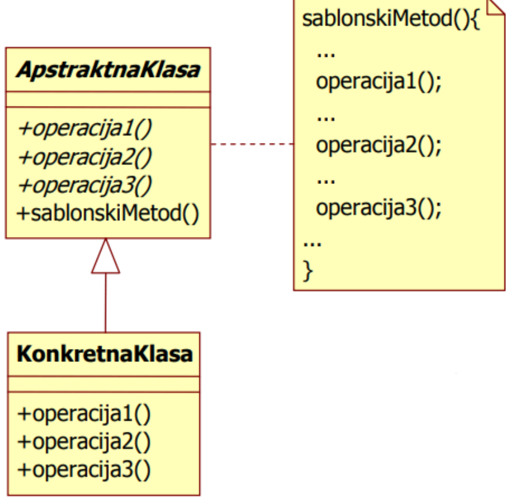
  - » klasa Aplikacija definiše algoritam za otvaranje dokumenta

```
void Aplikacija::otvoriDokument(const char* ime){
        if (!postojiLiDokument(ime))return;
        Dokument * dokument=napraviDokument();
        if (dokument!=nullptr){
                dodajDokument(dokument);
                preOtvaranja(dokument);
                dokument->otvori();
                dokument->citaj();
        }
}
```

  - » otvoriDokument() definiše korake za otvaranje dokumenta
  - » metod otvoriDokument() se naziva Šablonski metod
  - » šablonski metod definiše algoritam sastavljen od apstraktnih operacija
  - » potklase definišu operacije da obezbede konkretno ponašanje
  - » definisanjem koraka algoritma šablonski metod fiksira njihov redosled
  - » potklase prilagođavaju korake algoritma svojim potrebama

- *Primenljivost:* uzorak treba koristiti
  - » da se implementiraju invarijantni delovi algoritma jednom, a da se ostave potklasama za implementaciju delovi koji mogu varirati
  - » kada zajedničko ponašanje među potklasama treba lokalizovati da se izbegne dupliranje

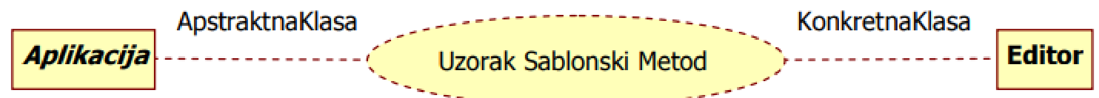
- *Struktura:*



- *Učesnici:*
  - » ApstraktnaKlasa (klasa Aplikacija)
    - — implementira šablon (skelet) algoritma - poziva primitivne operacije
    - — deklariše apstraktne primitivne operacije – za korake algoritma koje će potklase definisati
  - » KonkretnaKlasa (klasa Editor)
    - — implementira primitivne operacije
      - ▪ obavljaju delove algoritma
      - ▪ delovi su specifični za potklase

- *Saradnja:*
  - » KonkretnaKlasa implementira varijantne korake algoritma za klasu ApstraktnaKlasa

- *UML notacija:*



- *Posledice:*
  - » fundamentalna tehnika za reupotrebu koda (zajedničko ponašanje u klasama radnog okvira)
  - » vodi invertovanoj kontrolnoj strukturi (holivudski princip: "Ne zovite nas, mi ćemo zvati Vas" - roditeljska klasa zove operacije dece)
  - » izbegava se rizik da izvedena klasa iz redefinisane operacije ne pozove potrebnu operaciju roditeljske klase
  - » mogu se neke primitivne operacije realizovati u apstraktnoj klasi kao rudimentarne (hook) operacije koje obezbeđuju podrazumevano ponašanje

- *Povezani uzorci:*
  - » Fabrički Metod (npr. napraviDokument) se često poziva iz Šablonskog Metoda

» Šablonski Metod statički varira deo algoritma, a Strategija dinamički varira ceo algoritam

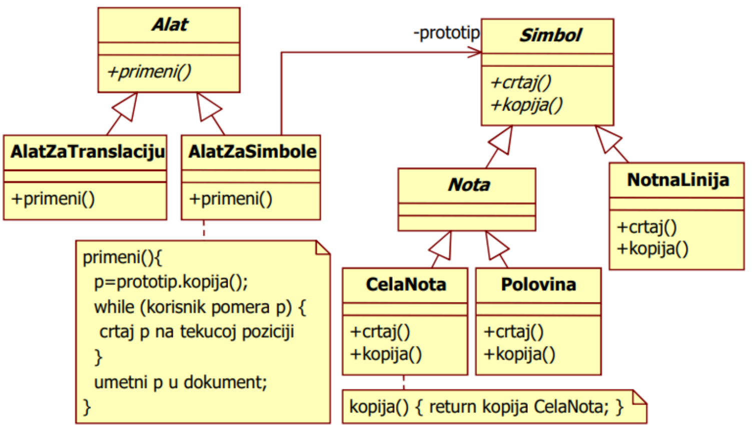
4 Prototip

- *Ime i klasifikacija:* Prototip (Polimorfna kopija, engl. Prototype) – objektni uzorak kreiranja

- *Namena:* specificira specifične vrste objekata koje se stvaraju i stvara nove objekte kopiranjem (kloniranjem) prototipskog primerka

- *Motivacija:*
  - » editor za muzičke partiture bi se mogao realizovati:
    - — prilagođenjem opšteg radnog okvira za grafičke editore
    - — dodavanjem novih objekata koji predstavljaju note, pauze i druge muzičke simbole
  - » radni okvir editora može imati paletu alata za dodavanje simbola dokumentu
    - — radni okvir ništa ne zna o konkretnim tipovima simbola koje treba dodavati
    - — paleta bi uključila i alate za selektovanje, pomeranje i druge radnje nad simbolima
  - » editor za muzičke partiture treba da omogući rad sa muzičkim simbolima
    - — korisnik bi kliknuo na alat "četvrtinka" i dodao ovu u partituru
    - — korisnik bi koristio alat za pomeranje da premesti notu na drugu notnu liniju

  - » apstraktna klasa Simbol za proizvoljne grafičke komponente
  - » apstraktna klasa Alat za alate u paleti
  - » konkretna klasa AlatZaSimbole za stvaranje i dodavanje objekata u dokument

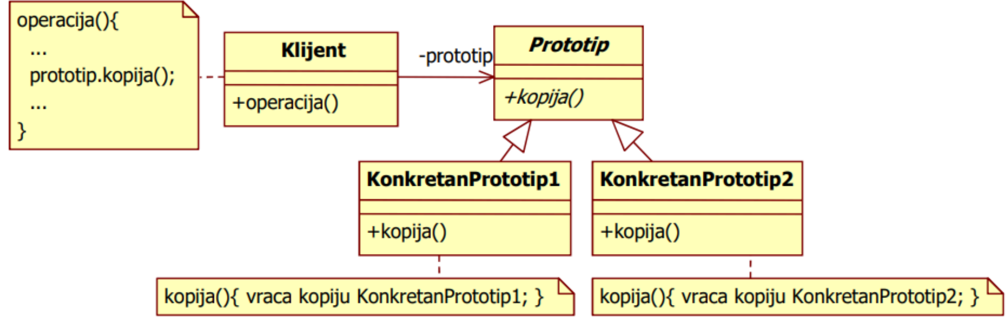


  - » problem za projektanta radnog okvira je klasa AlatZaSimbole
    - — klase za note i notne linije su specifične za aplikaciju, nepoznate radnom okviru
    - — mogla bi se praviti potklasa AlatZaSimbole za svaki muzički objekat, ali ima ih mnogo
    - — potklase bi se razlikovale samo po vrsti muzičkog objekta koje stvaraju
  - » kompozicija objekata je česta fleksibilna alternativa izvođenju i nasleđivanju
  - » radni okvir parametrizuje primerke AlatZaSimbole objektima klase Simbol
  - » rešenje:
    - — AlatZaSimbole kreira novi Simbol kopiranjem objekta potklase Simbol
    - — dotični objekat (primerak potklase Simbol) se naziva prototipom
    - — objekat AlatZaSimble je parametrizovan prototipom koji klonira i dodaje u dokument
    - — potrebno da sve potklase Simbol implementiraju operaciju kopija()
  - » u muzičkom editoru:
    - — alati za kreiranje muzičkih objekata su objekti klase AlatZaSimbole

— svaki objekat AlatZaSimbole se inicijalizuje različitim prototipom muzičkog objekta
— objekat AlatZaSimbole proizvodi muzički objekat kloniranjem njegovog prototipa

- *Primenljivost:* uzorak treba koristiti
  - » kada sistem treba da bude nezavisan od toga kako se njegovi proizvodi stvaraju i predstavljaju
  - » kada treba izbeći izgradnju hijerarhije fabrika paralelno sa hijerarhijom proizvoda
  - » kada se klase specificiraju u vreme izvršenja, npr. dinamičkim učitavanjem

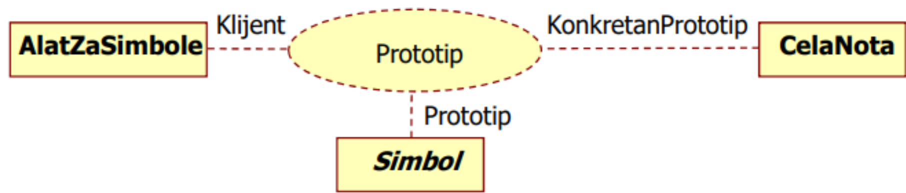
- *Struktura:*



- *Učesnici:*
  - » Prototip (klasa Simbol)
    - — deklariše interfejs za sopstveno kloniranje
  - » KonkretanProtorip (klase CelaNota, Polovina, NotnaLinija)
    - — implementira operaciju za sopstveno kloniranje
  - » Klijent (klasa AlatZaSimbole)
    - — stvara novi objekat zahtevom prototipu da se klonira

- *Saradnja:* – klijent zahteva od prototipa da se klonira

- *UML notacija:*



- *Posledice:*
  - » dobre strane:
    - — smanjivanje potrebe izvođenja
    - — dodavanje i uklanjanje proizvoda u vreme izvršenja (registracija prototipskog primerka)
    - — dinamičko konfigurisanje aplikacije klasama
    - — smanjivanje broja imena koje klijent treba da poznaje
  - » loša strana:
    - — svaka potklasa mora implementirati operaciju kopija() što je komplikovano – ako klasa već postoji ili ako objekat klase sadrži (pod)objekte koji ne podržavaju kloniranje

- *Povezani uzorci:*
  - » često zajedno sa uzorcima Sastav (Sklop, Kompozicija) i Dekorater (Dopuna)
  - » Fabrički metod takođe služi za stvaranje objekta čiji tip ne poznaje klijent

— stvaranje objekta se delegira potklasi, a ne drugom objektu

» Apstraktna Fabrika se implementira Fabričkim metodom ali može i Prototipom: fabrika može sadržati skup prototipa na osnovu kojih klonira i vraća proizvode

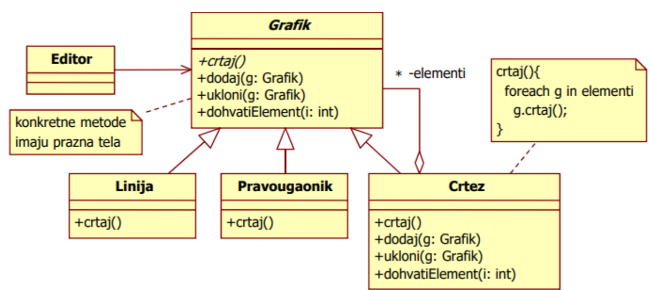
5 Sastav

- *Ime i klasifikacija:* Sastav (Sklop, Kompizicija, engl. Composite) – objektni uzorak strukture

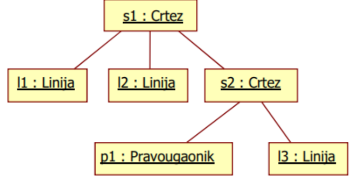
- *Namena:*
  - » komponuje objekte u strukturu stabla (hijerarhija celina-deo)
  - » omogućava klijentima da uniformno tretiraju
    - — individualne objekte
    - — njihove kompozicije

- *Motivacija:*
  - » grafičke aplikacije kao što su editori šema:
    - — omogućavaju crtanje kompleksnih šema sastavljenih od jednostavnih grafičkih elemenata (listova)
    - — elementi se mogu grupisati da se formiraju složeni elementi (sastavi)
    - — složeni element je potrebno tretirati i kao vrstu grafičkog elementa
  - » ključno: apstraktna klasa koja predstavlja i proste i složene elemente

klasni dijagram:

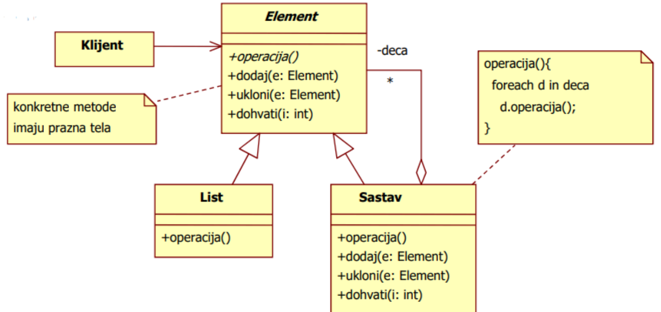


objektni dijagram – struktura rekurzivno komponovanih grafičkih objekata:



- *Primenljivost:* uzorak treba koristiti kada se želi da
  - » postoje hijerarhije objekata celina-deo takve da su celina i deo iste vrste
  - » klijenti mogu da ignorišu razlike između kompozicija i pojedinih objekata

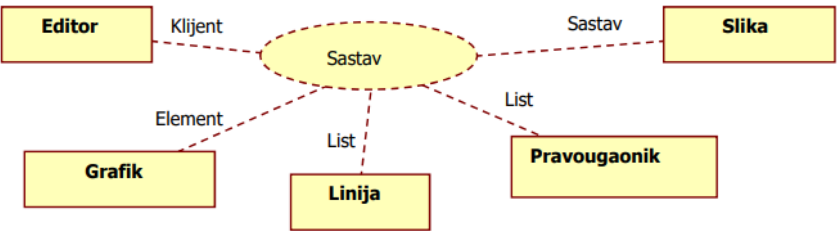
- *Struktura:*



- *Učesnici:*
  - » Element (klasa Grafik)
    - — deklariše zajednički interfejs za sve objekte u sastavu
    - — implementira podrazumevano ponašanje zajedničko za sve klase
    - — deklariše interfejs za pristupanje i upravljanje decom
    - — implementira prazne metode za pristup i upravljanje decom (zbog listova)
    - — opciono deklariše i implementira interfejs za pristup roditelju
  - » List (klase Linija, Pravougaonik)
    - — reprezentuje individualne objekte – listove u stablu
    - — definiše ponašanje za jednostavne objekte
  - » Sastav (klasa Crtez)
    - — definiše ponašanje za objekte koji imaju decu
    - — sadrži komponente decu
    - — implementira operacije za pristup i upravljanje decom
  - » Klijent (klasa Editor)
    - — manipuliše objektima u kompoziciji kroz interfejs klase Element

- *Saradnja:*
  - » klijenti koriste interfejs apstraktne klase Element da interaguju sa objektima složene strukture
    - — ako je primalac zahteva List, zahtev se neposredno izvršava
    - — ako je primalac zahteva Sastav, obično se zahtev prosleđuje deci

- *UML notacija*:
  - » na primeru grafičkog editora



- *Posledice:*
  - » uzorak čini jedostavnim klijente
    - — oni tretiraju na jedinstven način sve objekte u hijerarhiji
  - » uzorak čini jednostavnim dodavanje nove vrste elemenata
  - » nije jednostavno ograničiti vrste elemenata koje neki sastavi sadrže

- *Povezani uzorci:*
  - » često se veza element-roditelj koristi za uzorak Lanac odgovornosti
    - — ako element ne može da odgovori na zahtev – prosleđuje ga roditeljskom objektu

- Dekorater (Dopuna) ima sličnu strukturu klasa i često se koristi sa Sastavom
    - kada se koriste zajedno obično imaju zajedničku natklasu
- Muva dozvoljava strukture sa nedeljenim sastavima i deljenjim listovima
- Iterator se koristi često za obilazak strukture Sastava
- Posetilac se koristi sa Sastavom da lokalizuje operacije i ponašanje koje bi inače bilo distribuirano između klasa Sastav i List

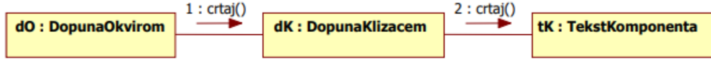
6 Dekorater

- *Ime i klasifikacija:* Dekorater (engl. Decorator) – objektni uzorak strukture

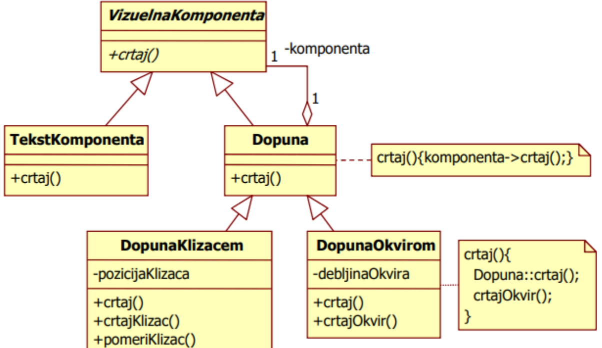
- *Namena:*
  - » dinamički dodaje odgovornosti (mogućnosti) nekom objektu
  - » fleksibilna alternativa izvođenju za proširivanje funkcionalnosti

- *Druga imena:* Dopuna, Omotač (engl. Wrapper)

- *Motivacija:*
  - » alati za GUI podržavaju dodavanje “ukrasa” na razne komponente: klizači, okviri sa raznim efektima…
  - » nefleksibilan način za dodavanje ukrasa – nasleđivanje sa proširivanjem
    - — nasleđivanje je nefleksibilno, jer se ukrasi definišu statički
    - — ukrasi se ne mogu dinamički kombinovati na osnovnoj komponenti
    - — opasnost od eksplozije broja izvedenih klasa kada se kombinuju ukrasi
  - » fleksibilniji pristup je da se komponenta obuhvati drugom komponentom
    - — komponenta koja obuhvata osnovnu dodaje ukras (okvir, klizače)
    - — ukrasi se dodaju pojedinom objektu, a ne celoj klasi
    - — ukrasi se mogu menjati dinamički
    - — generalno – proširenja mogu biti specifična stanja ili ponašanja
  - » komponenta koja obuhvata osnovnu se naziva dekoraterom, omotačem ili dopunom
  - » dekorater prosleđuje zahteve klijenta obuhvaćenoj komponenti i može da obavi dodatne akcije kao što je crtanje okvira ili dodavanje klizača za pomeranje
  - » dekorater nasleđuje interfejs komponente koju ukrašava
  - » prisustvo dekoratera je transparentno za klijente komponente
    - — transparentnost omogućava neograničen broj dodataka uvedenih posebnim dekoraterima
  - » primer:
    - — TekstKomponenta prikazuje tekst u prozoru i nema klizače za H/V pomeranje sadržaja, ni okvir
    - — ako su potrebni klizači dodaje se objekat klase DopunaKlizacem koji ih dodaje
    - — ako se želi i okvir dodaje se i objekat klase DopunaOkvirom koji crta okvir
    - — sledeći dijagram prikazuje objektnu kompoziciju i interakciju:

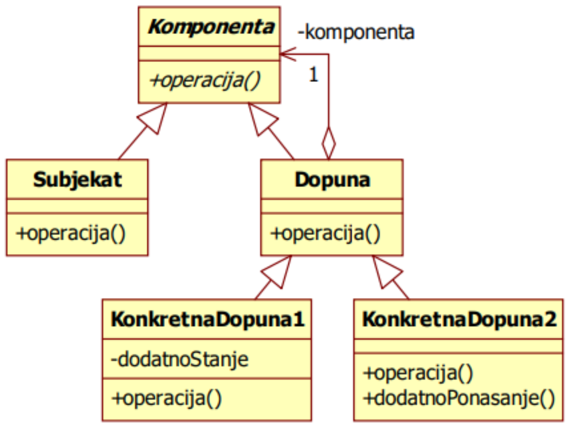


  - » sledeći dijagram prikazuje opisanu klasnu strukturu:



- *Primenljivost:*
  - » kada je potrebno dinamički dodavati odgovornosti objektima na transparentan način
  - » kada proširenje izvođenjem nije praktično (kada je moguć veliki broj nezavisnih proširenja, pri čemu se ona mogu kombinovati, što vodi eksploziji broja klasa)

- *Struktura:*



- *Učesnici:*
  - » Komponenta (klasa VizuelnaKomponenta)
    - — definiše interfejs za objekte kojima se mogu dinamički dodavati odgovornosti
  - » Subjekat (klasa TekstKomponenta)
    - — definiše objekat kojem se dodaju odgovornosti
  - » Dopuna (klasa Dopuna)
    - — održava referencu na objekat klase Komponenta i nasleđuje interfejs klase Komponenta
  - » KonkretnaDopunaX (klase DopunaKlizacem, DopunaOkvirom)
    - — dodaje odgovornost objektu tipa Komponenta

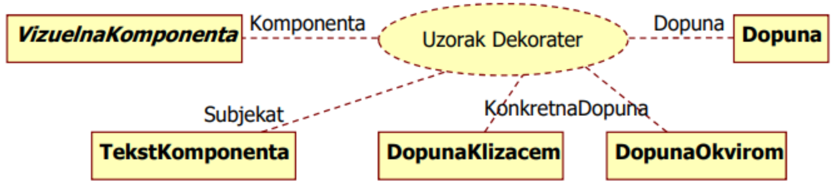
- *Saradnja:*
  - » objekat KonkretnaDopunaX
    - — prosleđuje zahteve obuhvaćenom objektu (tipa Komponenta)
    - — izvršava dodatne operacije pre i/ili posle prosleđivanja zahteva

- *Posledice:*
  - » prednosti
    - — veća fleksibilnost od statičkog nasleđivanja – dinamičko dodavanje odgovornosti

— izbegavanje eksplozije potklasa koje kombinuju dekoracije
— izbegavanje bogato parametrizovanih klasa visoko u hijerarhiji

» mane
— identitet dekoratera i dekorisanog objekta su različiti
— objekti se ne razlikuju po klasi već po načinu povezivanja
  ▪ potencijalan problem kod razumevanja i održavanja sistema
— nasleđeni atribut iz klase Komponenta – duplikat u svakoj dopuni
  ▪ dobro je da Komponenta nema atribute

- *UML notacija:*



- *Povezani uzorci:*

» Dekorater zadržava interfejs, a Adapter menja interfejs
» Kompozicija ima sličnu klasnu strukturu
— Dekorater je po objektnoj strukturi degenerisan Sastav (Kompozicija, Sklop)
  ▪ objekat Dopuna sadrži samo jednu komponentu, a objekat Kompozicija više
  ▪ Sastav formira objektno stablo, a Dekorater objektnu listu
— po nameni je Dekorater sasvim različit, jer dodaje odgovornosti i nije namenjen grupisanju

» Dekorater i Strategija su alternative za promenu ponašanja objekta
— Dekorater menja spoljašnjost objekta, a Strategija unutrašnjost

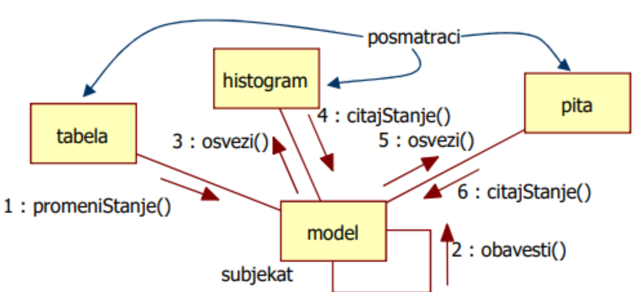
Posmatrač

- *Ime i klasifikacija:* Posmatrač (engl. Observer) – objektni uzorak ponašanja

- *Namena:*
  - » definiše 1:n zavisnost između objekata takvu da kada jedan objekat promeni stanje svi zavisni se obaveste i modifikuju automatski

- *Druga imena:*
  - » Zavisni objekti, Objavljivanje-Pretplata (Dependents, Publish-Subscribe, Subscribe-Notify)

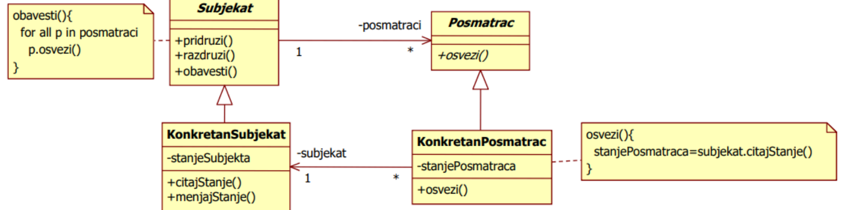
- *Motivacija:*
  - » nije dobro da su klase sistema čvrsto-spregnute (smanjuje se reupotrebljivost)
    - — na primer, alati za GUI razdvajaju prezentacione aspekte od aplikativnih podataka
    - — klase aplikativnih podataka i one za prezentaciju se mogu reupotrebiti nezavisno
  - » objekat za tabelarni prikaz i objekat za histogram mogu prikazati iste podatke
    - — objekat za tabelarni prikaz i objekat za histogram ne znaju jedan za drugog
    - — zbog toga dopuštaju nezavisnu reupotrebu, a ponašaju se kao da su spregnuti
    - — kada se podaci promene kroz tabelu – promeni se i histogram



  - » tabela i histogram su zavisni od modela: treba ih obavestiti kad model promeni stanje
  - » nema ograničenja u broju zavisnih objekata koje treba obaveštavati
  - » ključni objekti u uzorku su subjekat (subject) i posmatrač (observer)
  - » uzorak se naziva i Dependents, jer posmatrač zavisi od subjekta
    - — subjekat obaveštava posmatrača kad promeni stanje
    - — posmatrač zatim šalje upit subjektu o njegovom stanju
    - — na osnovu primljenog stanja subjekta, posmatrač ažurira svoje stanje
  - » uzorak se naziva i Publish-Subscribe
    - — posmatrači se "pretplaćuju" kod subjekta za obaveštenja (notifikaciju)
    - — subjekat šalje poruku obaveštenja o promeni posmatraču (notifikacija)
    - — poruke se šalju svim posmatračima i ne znajući ko su sve posmatrači

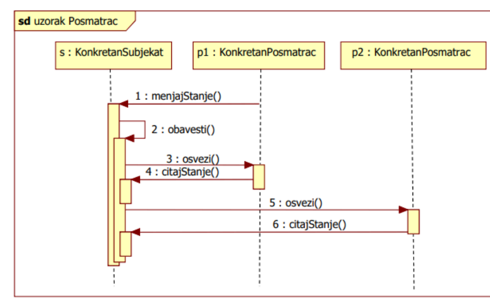
- *Primenljivost:* uzorak treba koristiti u sledećim situacijama:
  - » kada jedna apstrakcija ima barem dva međusobno zavisna apsekta takva da promena bilo kog utiče na promenu drugih
  - » kada izmena jednog objekta zahteva izmenu nepoznatog broja drugih objekata
  - » kada jedan objekat treba da signalizira promenu drugim objektima ne znajući prirodu (konkretni tip) tih objekata

- *Struktura:*

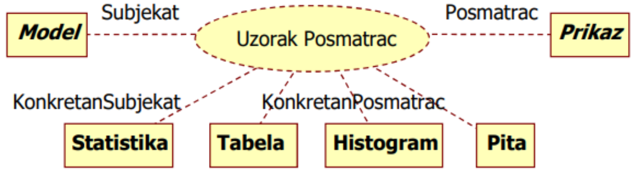


- *Učesnici:*
  - » Subjekat (klasa Model)
    - — zna svoje posmatrače (proizvoljan broj), ali ne i njihovu prirodu
    - — obezbeđuje interfejs za pridruživanje i razdruživanje posmatrača
  - » Posmatrac (klasa Prikaz)
    - — definiše interfejs za signaliziranje promena subjekta
  - » KonkretanSubjekat (klasa Statistika)
    - — čuva stanje od interesa za konkretne posmatrače
    - — inicira slanje signala posmatračima kada se promeni stanje
    - — omogućava čitanje stanja
  - » KonkretanPosmatrac (klase Tabela, Histogram, Pita)
    - — poseduje referencu na konkretan subjekat (subjekat može da bude i parametar)
    - — čuva stanje koje treba da bude u konzistenciji sa stanjem konkretnog subjekta
    - — implementira operaciju preko koje mu subjekat signalizira promenu
    - — čita stanje konkretnog subjekta da bi ažurirao sopstveno stanje

- *Saradnja:*
  - » konkretan subjekat signalizira svojim posmatračima svaku promenu svog stanja
  - » nakon poziva osvezi, konkretan posmatrač traži od subjekta informaciju o stanju
  - » posmatrač koristi informaciju o stanju subjekta da ažurira svoje stanje



- *UML notacija:*



- *Posledice:*
  - » dobre strane
    - — apstraktno vezivanje između subjekata i posmatrača; tako se omogućava da subjekat i posmatrač budu u različitim slojevima aplikacije
    - — podrška sa multicast komunikaciju; subjekat šalje signal svim registrovanim posmatračima
  - » nedostatak - nepoznata cena promene - pošto posmatrači ne znaju ko će sve dobiti signal, oni nisu svesni cene promene subjekta

- *Povezani uzorci:*

- za kapsuliranje kompleksne semantike ažuriranja Posrednik može da posreduje između subjekata i posmatrača

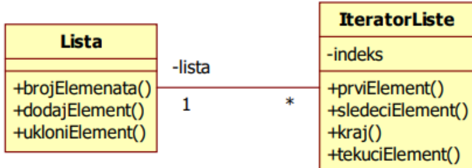
Iterator

- *Ime i klasifikacija:* Iterator (engl. Iterator) – objektni uzorak ponašanja

- *Namena:*
  - » obezbeđuje pristup elementima zbirke (agregata) redom, bez eksponiranja interne strukture te zbirke

- *Drugo ime:* Kurzor (engl. Cursor), za specifičnu vrstu iteratora

- *Motivacija:*
  - » lista (agregat) treba da omogući pristup njenim elementima
  - » interna struktura liste ne treba da bude eksponirana klijentima
  - » može postojati više načina za obilazak liste
    - — klasu liste ne treba opteretiti operacijama za razne načine obilazaka
    - — ni klijenta ne treba opteretiti specifičnostima načina obilaska liste
  - » potrebno je da više klijenata simultano obilazi listu
  - » ključna ideja uzorka Iterator:
    - — prenošenje odgovornosti za obilazak liste na poseban objekat - iterator
  - » klasa iteratora definiše interfejs za pristup elementima liste
  - » objekat iteratora zna način obilaska liste i to:
    - — od kog elementa početi obilazak
    - — koji su elementi liste već posećeni, odnosno koji je naredni na redu
    - — kada je obilazak završen (nema neobiđenih elemenata)
  - » objekat iteratora čuva informaciju o tekućem elementu obilaska
  - » relacija između klasa Lista i IteratorListe:

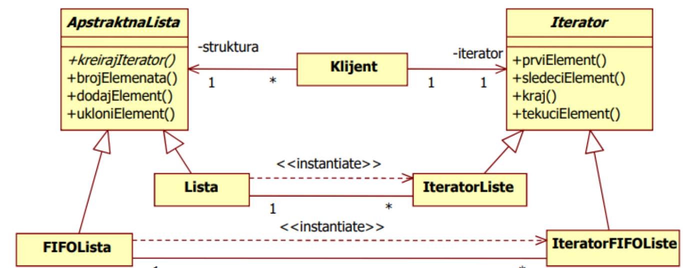


  - » prviElement() - inicijalizuje pokazivač tekućeg elementa postavljanjem na prvi element
  - » sledeciElement()- proglašava naredni element tekućim
  - » kraj() - testira da li je obilazak završen
  - » tekuciElement()- vraća tekući element liste

  - » razdvajanje mehanizma obilaska od objekta liste omogućava definisanje posebnih iteratora za različite politike obilaska
    - — na primer: IteratorFiltriraneListe može pružiti pristup samo onim elementima koji zadovoljavaju uslove filtra
    - — interfejs klase Lista nije opterećen raznim politikama obilaska
  - » nedostatak:
    - — klijent mora biti svesan da se obilazi baš konkretna vrsta liste i da se za nju koristi baš konkretna vrsta iteratora
    - — za konkretnu vrstu liste klijent mora da napravi objekat iteratora odgovarjuće klase za tu listu
    - — ukoliko bi se menjala vrsta liste koja se obilazi
      - ▪ morao bi se menjati kod klijenta za stvaranje iteratora

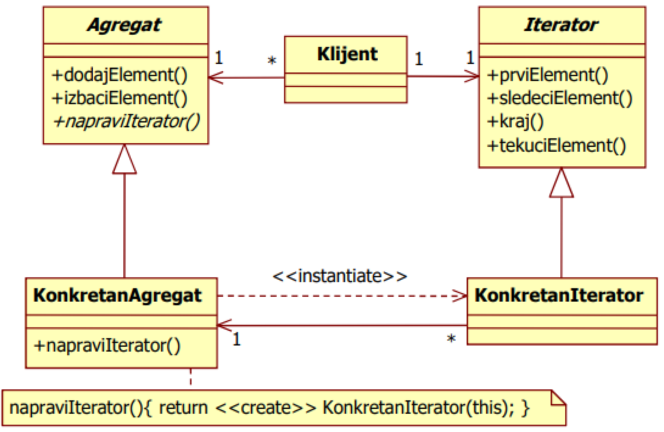
» generalizacija koncepta iteratora: polimorfni iterator
» primer: klijent treba da radi sa klasom Lista ali i sa klasom FIFOLista
» definiše se interfejs ApstraktnaLista za manipulisanje proizvoljnom listom
» definiše se interfejs Iterator za objekte iteratora svih vrsta lista
» potklase koje implementiraju interfejs Iterator biće iteratori za odgovarajuće liste
» klasa konkretne liste je odgovorna za stvaranje odgovarajućeg iteratora
» mehanizam je postao nezavisan od konkretnih klasa lista
  — klijent ne mora da vodi računa o vrsti liste da bi za nju napravio odgovarajući objekat iteratora

» da bi se klijent učinio nezavisnim od konkretne liste/iteratora:
  — definiše se operacija liste kreirajIterator() kojom ova treba da kreira svoj iterator
  — klijenti koriste apstraktnu operaciju i tako su nezavisni od konkretnih klasa liste
  — operacija kreirajIterator() je primer Fabričkog metoda



- *Primenljivost:* uzorak treba koristiti
  » da se podrže višestruki simultani obilasci agregata
  » da se obezbedi uniformni interfejs za obilazak različitih agregata
  » da se ni klijent ni agregat ne opterete politikom obilaska

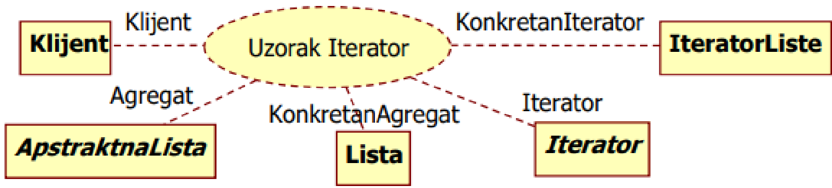
- *Struktura:*



- *Učesnici:*
  » Agregat (klasa ApstraktnaLista)
    — definiše interfejs za kreiranje objekta iteratora
  » Iterator (klasa Iterator)
    — definiše interfejs za obilazak elemenata agregata i pristup elementu
  » KonkretanAgregat (klase Lista, FIFOLista)
    — implementira interfejs za kreiranje iteratora tako što vraća odgovarajući konkretan iterator
  » KonkretanIterator (klase IteratorListe, IteratorFIFOListe)
    — implementira interfejs Iterator
    — čuva informaciju o tekućoj poziciji pri obilasku agregata
- *Saradnja:*

» klijent zahteva od agregata da napravi odgovarajući iterator
» klijent se obraća konkretnom iteratoru
    — za pozicioniranje na prvi i svaki sledeći element agregata i proveru kraja obilaska
    — za dohvatanje tekućeg elementa agregata

- *UML notacija:*



- *Posledice:*
    » Iterator ima sledeće dobre strane:
        — pojednostavljuje interfejs agregata: interfejs za obilazak je u klasi iteratora
        — više od jednog obilaska se može simultano sprovoditi nad agregatom
        — podržava varijacije u obilasku agregata:
            ▪ lako se kreira nova klasa konkretnog iteratora za novi način obilaska

- *Implementacija:*
    » ko upravlja iterativnim procesom?
        — Spoljašnji (external) iterator – klijent kontroliše iteraciju
            ▪ na klijentu je odgovornost za progres obilaska
            ▪ eksplicitno zahteva od iteratora sledeći element
            ▪ iterator o kojem je ovde bilo reči je spoljašnji iterator
        — Unutrašnji (internal) iterator – iterator kontroliše iteraciju
            ▪ klijent samo zahteva od iteratora da izvrši neku operaciju
            ▪ sam iterator primenjuje operaciju na svaki element agregata
        — spoljašnji iterator je fleksibilniji od unutrašnjeg

    » ko definiše način (algoritam, politiku) obilaska?
        — po pravilu – iterator, ali ne i obavezno
        — agregat može definisati algoritam obilaska
            ▪ agregat ima operacije za obilazak i dohvatanje: prviElement(),...
        — iterator se tada koristi samo da čuva stanje iteracije
            ▪ samo ukazuje na tekuću poziciju u agregatu
            ▪ ovakav iterator se naziva Kurzor (Cursor)
        — klijent poziva operaciju sledeciElement() sa kurzorom kao argumentom
            ▪ operacija sledeciElement() promeni stanje kurzor-objekta
        — ako se algoritam obilasaka definiše u klasi agregata gube se povoljnosti:
            ▪ lakog variranja iterativnih algoritama nad istim agregatom
            ▪ reupotrebe algoritma za obilazak sličnih agregata
        — zadržava se povoljnost višestrukih simultanih obilazaka agregata

    » koliko je robustan iterator?

    - opasno je modifikovati agregat dok se on obilazi
        - moguće je ukloniti tekući element (iterator postaje "viseći")
        - moguće je umetnuti ili ukloniti naredni element (problem ako je iterator već zapamtio adresu narednog)
    - robusno implementiran uzorak iteratora obezbeđuje da umetanje/uklanjanje elemenata agregata ne interferira sa obilaskom
    - sprečavanje interferencije
        - agregat treba da registruje iteratore koje je kreirao
        - pri umetanju i uklanjanju elementa agregat prilagođava stanje registrovanih iteratora

- » Dodatne operacije iteratora za uređene/indeksirane agregate
    - prethodniElement() pozicionira iterator na prethodni element
    - skociNa() pozicionira iterator na objekat koji odgovara specifičnim kriterijumima
- » Polimorfni iteratori u C++
    - polimorfni itereratori imaju svoju cenu – objekat iteratora se alocira dinamički
    - nedostatak je što je klijent odgovoran za dealokaciju objekta iteratora
- » Iteratori mogu imati privilegovani pristup
    - iteratori se mogu posmatrati kao ekstenzije agregata sa kojima su čvrsto spregnuti
    - na C++ se čvrsta sprega može iskazati tako što su iteratori prijatelji agregata
    - ako su iteratori prijatelji, agregat ne mora da implementira operacije za efikasni pristup
    - loša strana rešenja je što pri definisanju novih iteratora mora da se menja agregat (da bi se proglasio novi prijatelj)
    - klasa Iterator može da uključi neke zaštićene operacije za direktan pristup agregatu
        - date operacije koriste samo izvedene klase iz klase Iterator

- » Iteratori za objekte Sastava
    - objekti sastava (u hijerarhiji stabla) se često obilaze na razne načine -prefiksnim/infiksnim/postfiksnim redosledom, po širini/dubini stabla
    - nije jednostavno spolja pamtiti poziciju – ona treba da uključi putanju od korena
    - zato su spoljašnji iteratori komplikovaniji za agregate tipa sastava
    - unutrašnji iteratori implicitno čuvaju putanju na steku - rekurzivno se izvršava zadata operacija
    - spoljašnji iterator - klijent može da od tekućeg čvora traži iterator za obilazak njegove dece
- » NullIterator
    - degenerisani iterator pogodan za obradu graničnih uslova
    - po definiciji, njegova operacija kraj() uvek vraća true
    - pogodan je za obilazak agregata tipa stabla (kao što je Sastav)
        - svi čvorovi osim listova vraćaju iterator za obilazak dece, a list vraća NullIterator
        - ovim se dobija uniformnost u obilasku strukture stabla

- *Povezani uzorci:*
    - » Iterator se često primenjuje na rekurzivne strukture kao što je Sastav
    - » Polimorfni iteratori se zasnivaju na Fabričkom metodu
        - apstraktna operacija agregata (fabrike) za kreiranje iteratora
        - konkretizuje se u potklasi konkretnog agregata
        - objekat konkretnog agregata stvara objekat konkretne potklase iteratora
    - » Iterator može da interno koristi Podsetnik za čuvanje stanja iteracije
    - » Posmatrač može da se koristi za obaveštavanje o promeni stanja agregata

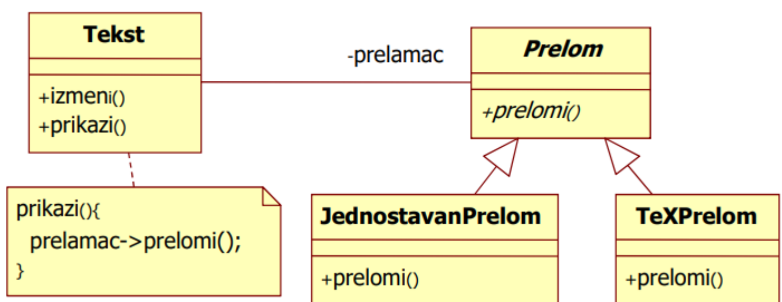
Strategija

- *Ime i klasifikacija:* Strategija (engl. Strategy) – objektni uzorak ponašanja

- *Namena:* definiše familiju algoritama, kapsulirajući svaki i čini ih međusobno zamenjivim u vreme izvršenja

- *Drugo ime:* Politika (Policy)

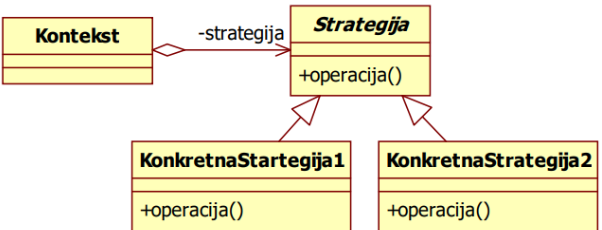
- *Motivacija:*
  - » postoji više algoritama za prelom teksta u linije
  - » ugrađivanje tih algoritama u klasu koja predstavlja apstrakciju teksta nije dobro iz više razloga:
    - — klasa postaje kompleksnija i teža za održavanje
    - — različiti algoritmi su odgovarajući u različitim situacijama a klijent je odgovoran za pozivanje odgovarajućih metoda
    - — teško je dodati novi algoritam i varirati postojeći
  - » ni izvođenje iz klase teksta nije dobro rešenje, jer je statičko
    - — ne može se promeniti algoritam preloma, a da se ne promeni objekat teksta
  - » alternativa - kapsulirani algoritam u posebnu klasu koja se naziva strategija



  - » klasa Tekst je odgovorna za prikaz i izmenu teksta
  - » strategije preloma nisu implementirane u klasi Tekst - implementirane su u potklasama Prelom
  - » klasa Tekst sadrži referencu na objekat tipa Prelom
  - » kada se Tekst izmeni, da bi se prikazao, treba da se reformatira
    - — objekat klase Tekst prosleđuje tu odgovornost objektu klase Prelom
  - » klijent klase Tekst specificira objekat Prelom instalirajući ga u Tekst

- *Primenljivost:*
  - » kada bi se više srodnih klasa razlikovalo samo po nekom ponašanju
    - — uzorak omogućava konfigurisanje jedne klase jednim od više ponašanja
  - » kada su potrebne različite varijante nekog algoritma
  - » kada algoritam koristi podatke o kojima ni kontekst ne treba ništa da zna
    - — izbegava se eksponiranje kompleksnih struktura podataka koje su specifične za algoritam
    - — potrebno je kapsuliranje metoda i strukture podataka za algoritam
  - » kada klasa konteksta definiše više ponašanja koja se pojavljuju kao grane uslovne naredbe u raznim operacijama
    - — grane uslovne naredbe treba kapsulirati u odgovarajuće strategije
    - — jedna strategija u svojim operacijama izvršava samo odgovarajuću granu

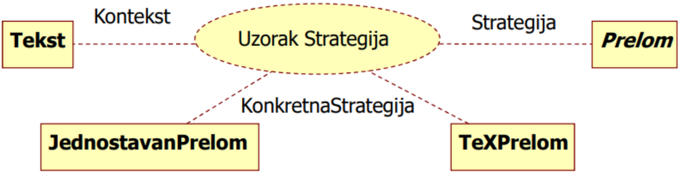
- *Struktura:*



- *Učesnici:*
  - » Strategija (klasa Prelom)
    - — deklariše zajednički interfejs za sve podržane algoritme
  - » KonkretnaStrategijaX (klase JednostavanPrelom, TeXPrelom)
    - — implementira konkretan algoritam tako da odgovara interfejsu klase Strategija
  - » Kontekst (klasa Tekst)
    - — ima referencu na objekat tipa Strategija
    - — konfigurisan je objektom KonkretnaStrategijaX
    - — može da pruži interfejs koji omogućava objektu strategije da pristupi njegovim podacima

- *Saradnja:*
  - » interaguju Kontekst i KonkretnaStrategijaX
  - » Kontekst može da pošalje
    - — ili sve podatke koje zahteva algoritam pri pozivu objekta Strategija
    - — ili referencu na sebe i tako omogući povratni poziv (callback)
  - » Kontekst prosleđuje zahteve svojih klijenata svom objektu Strategija
    - — klijenti obično kreiraju i prosleđuju objekte KonkretnaStrategijaX objektu Kontekst
    - — kasnije klijenti interaguju samo sa objektom Kontekst

- *UML notacija:*



- *Posledice:*
  - » familije srodnih algoritama definisanih kao hijerarhija klasa Strategija
  - » fleksibilna alternativa izvođenju iz klase Kontekst
    - — kad bi Kontekst implementirao algoritam
  - » strategije eliminišu potrebu za uslovnim naredbama u klijentskom kodu
  - » izbor implementacija
    - — klijent bira implementaciju da postigne performanse u vremenu/prostoru
  - » nedostatak – klijenti moraju biti svesni strategije kojom parametrizuju kontekst
  - » Kontekst kreira nepotrebne parametre za neke KonkretnaStrategijaX
  - » povećava se broj objekata u aplikaciji zbog objekata KonkretnaStrategijaX

- *Povezani uzorci:*
  - » Strategija menja ponašanje konteksta “iznutra”, dok Dekorater menja ponašanje dekorisanog subjekta “spolja”

- » Strategija varira ceo algoritam, a Šablonski metod varira korake algoritma
- » Objekti Strategije često predstavljaju dobre objekte Muve

10 Stanje

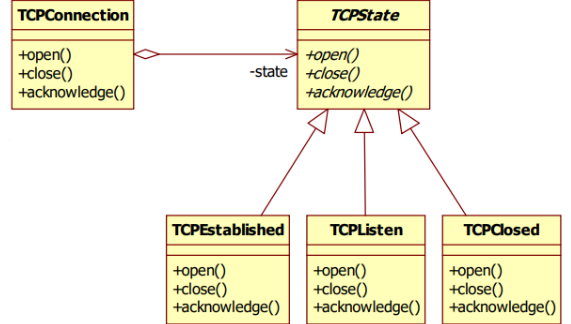
- *Ime i klasifikacija:* Stanje (engl. State) – objektni uzorak ponašanja

- *Namena:*
  - » omogućava objektu da pouzdano menja svoje ponašanje kada se menja njegovo unutrašnje stanje
  - » izgleda kao da objekat menja svoju klasu

- *Drugo ime:* Objekti za stanja (engl. Objects for States)

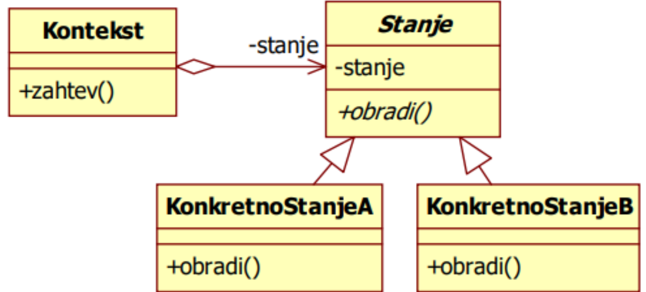
- *Motivacija*: primer vezan za mrežni TCP protokol
  - » vezu opisuje TCPConnection klasa
  - » moguća stanja veze: Established, Listening, Closed
  - » TCPConnection objekat odgovara na zahteve u zavisnosti od stanja
    - — npr. efekat open() zahteva se razlikuje u stanju Closed i Established
  - » uzorak Stanje
    - — opisuje kako ponašanje TCPConnection objekta zavisi od stanja
  - » ključna ideja: uvođenje apstraktne klase TCPState
    - — klasa predstavlja stanja veze
  - » klasa TCPState deklariše zajednički interfejs za klase stanja
  - » potklase TCPState realizuju specifično ponašanje pojedinih stanja

  - » objekat TCPConnection
    - — sadrži objekat stanja (objekat potklase TCPState) koji reprezentuje tekuće stanje veze
    - — prosleđuje objektu stanja zahteve koji su zavisni od tekućeg stanja
    - — kada konekcija menja stanje zamenjuje objekat stanja koji koristi



- *Primenljivost:*
  - » kada ponašanje objekta zavisi od stanja i mora se menjati u vreme izvršavanja
  - » kada operacije imaju velike uslovne naredbe sa više grana, čije izvršenje zavisi od stanja objekta
    - — često više operacija sadrži istu uslovnu naredbu
    - — uzorak Stanje pravi od svake grane posebnu klasu koja određuje ponašanje operacije u datom stanju

- *Struktura:*

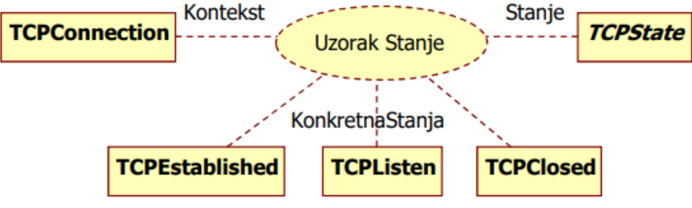


- *Učesnici:*
  - Kontekst (klasa TCPConnection)
    - definiše interfejs od interesa za klijente
    - održava primerak KonkretnoStanjeX
  - Stanje (klasa TCPState)
    - definiše interfejs za pojedina stanja
    - kapsulacija ponašanja pridruženog stanju objekta klase Kontekst
  - KonkretnoStanjeX (klasa TCPEstablished, TCPListen)
    - implementira interfejs stanja

- *Saradnja:*
  - kontekst prosleđuje zahteve koji su zavisni od stanja tekućem objektu konkretnog stanja
  - kontekst može da prosledi konkretnom objektu stanja referencu na sebe, što omogućava objektu stanja da pristupi kontekstu (npr. da zahteva promenu stanja)
  - kontekst je primarni interfejs za klijente, klijenti ga mogu konfigurisati objektima stanja; nakon toga klijenti ne moraju da interaguju direktno sa objektima stanja

- *Posledice:*
  - dobra kapsulacija ponašanja specifičnog za stanje
  - jasno razdvajanje ponašanja u različitim stanjima
  - jednostavno dodavanje novih stanja definisanjem novih potklasa stanja
    - nefleksibilna alternativa su uslovni iskazi rasuti svuda po kodu konteksta
  - robusno rešenje - prelazi između stanja su eksplicitni i atomični
    - kada kontekst samostalno definiše svoje stanje vrednostima atributa
      - prelazi između stanja nemaju eksplicitnu reprezentaciju
      - predstavljaju se dodelama vrednosti atributima
    - objekti stanja štite objekat konteksta od nekonzistentnih stanja
      - prelazi su atomični iz perspektive konteksta
      - prelaz se svodi na prevezivanje jednog pokazivača
  - objekti stanja mogu biti deljeni
    - ako objekti stanja nemaju atribute, konteksti mogu da dele objekat stanja
      - stanje koje objekti stanja reprezentuju je kodirano tipom objekta
    - tada su objekti stanja "muve" bez unutrašnjeg stanja, imaju samo ponašanje

- *UML notacija:*



- *Povezani uzorci:*
    - » Strategija – praktično ista klasna struktura
    - » Objekti stanja se često realizuju kao Unikati
    - » Muva – objašnjava kada se objekti stanja mogu deliti

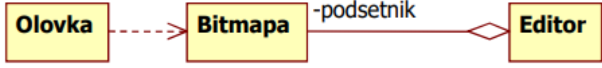
11 Podsetnik

- *Ime i klasifikacija:* Podsetnik (engl. Memento) – objektni uzorak ponašanja

- *Namena:*
  - » snima i spolja čuva unutrašnje stanje nekog objekta bez narušavanja njegove kapsulacije
  - » omogućava da se objekat kasnije može vratiti u dato stanje

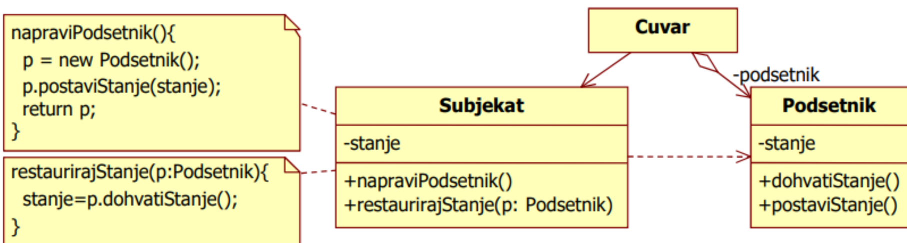
- *Drugo ime:* Oznaka (engl. Token)

- *Motivacija:*
  - » podrška za poništavanje operacija (undo)
  - » potrebno je sačuvati unutrašnje stanje objekta, kako bi ga kasnije vratili u to stanje
  - » problem: objekat kapsulira svoje stanje i nije dobro da ga eksponira
    - — zato menjani objekat treba sam da snimi svoje stanje
    - — snimak stanja predstavlja "podsetnik"
    - — objekat podsetnika ne mora da čuva objekat koji ga je snimio
  - » primer: crtanje olovkom u editoru ikona
    - — editor obezbeđuje da se nakon iscrtavanja traga isti može restaurirati
    - — snimanje i čuvanje svakog piksela na tragu bi usporilo crtanje
  - » klasa Editor
    - — pokreće operacije klase Olovka i poseduje operaciju ponisti()
  - » klasa Olovka poseduje operacije:
    - — spusti() – otpočinje crtanje - pamti se prethodna bitmapa ikone
    - — pomeri() – pomera olovku
      - ▪ ako je olovka spuštena, pikseli na tragu se menjaju
      - ▪ pamte se min/max X i Y koordinata
    - — podigni() – završava crtanje
      - ▪ odseca se deo bitmape izvan min/max X i Y koordinata
      - ▪ preostali deo bitmape (podsetnik) se daje editoru na čuvanje
  - » operacija ponisti() - olovci se dostavlja podsetnik da restaurira sliku



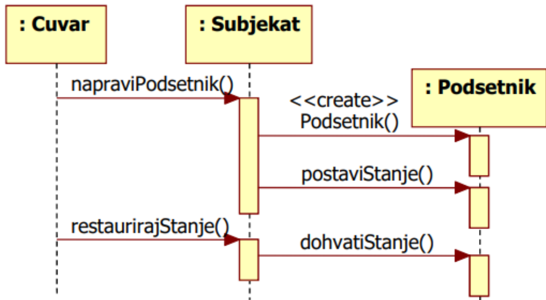
- *Primenljivost:* uzorak treba koristiti kada
  - » treba napraviti snimak stanja nekog objekta da bi se stanje kasnije restauriralo
  - » direktan interfejs za dobijanje stanja subjekta bi eksponirao njegove implementacione detalje

- *Struktura:*

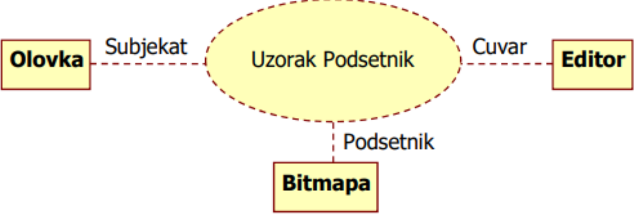


- *Učesnici:*
  - » Podsetnik (klasa Bitmapa)
    - — čuva unutrašnje stanje objekta Subjekat
    - — ne dozvoljava pristup stanju drugim objektima osim objektu Subjekat
    - — ima dva interfejsa: Cuvar vidi uski, Subjekat vidi široki
  - » Subjekat (klasa Olovka)
    - — kreira objekat Podsetnik koji sadrži snimak njegovog tekućeg stanja
    - — ima diskreciono pravo da odluči koji deo stanja se čuva
    - — koristi objekat Podsetnik da restaurira stanje
  - » Cuvar (klasa Editor)
    - — odgovoran za bezbedno čuvanje objekta Podsetnik
    - — ne ispituje i ne koristi stanje objekta Podsetnik
    - — prosleđuje podsetnika subjektu za restauraciju stanja

- *Saradnja:*
  - » objekti podsetnika su pasivni



- *UML notacija:*



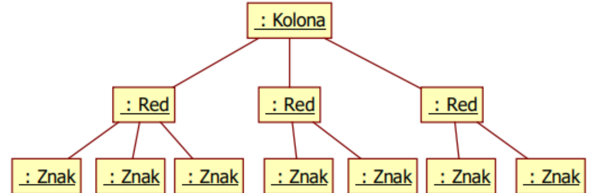
- *Povezani uzorci:*
  - » u uzorku Iterator - Podsetnik se može koristiti za čuvanje podatka o tekućem elementu
  - » u uzorku Komanda - Podsetnik se može koristiti za čuvanje stanja poništivih operacija

Muva

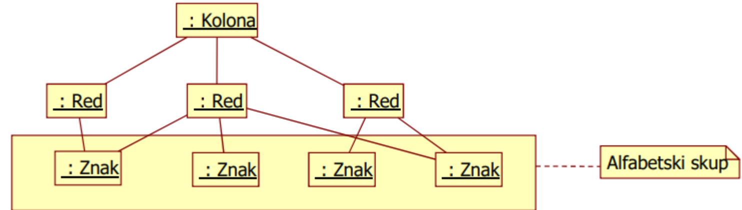
- *Ime i klasifikacija:* Muva (eng. Flyweight) – objektni uzorak strukture

- *Namena:*
  - » deljenje “lakih” objekata sa ciljem da se izbegne hiperprodukcija objekata
  - » laki objekti:
    - — objekti bez stanja
    - — objekti čije unutrašnje stanje ne zavisi od pojave u kontekstu
    - — kontekst definiše “spoljašnje” stanje lakog objekta
    - — spoljašnje stanje zavisi od pojave u kontekstu

- *Motivacija:*
  - » editor teksta organizuje dokument kao hijerarhiju
    - — kolona
    - — redova
    - — znakova
  - » problem: hiperprodukcija objekata znakova

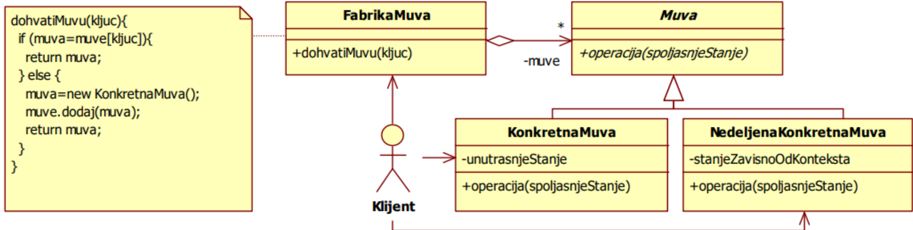


  - » rešenje: samo po jedan objekat koji reprezentuje pojedini znak
    - — red u kojem se znak nalazi - kontekst pojave znaka
    - — red određuje poziciju pojave znaka u redu (spoljašnje stanje)
    - — znak pamti samo specifični kod i geometriju (unutašnje stanje)
      - ▪ unutrašnje stanje ne zavisi od pozicije pojave znaka u kontekstu



- *Primenljivost:* ako su ispunjeni svi uslovi
  - » aplikacija koristi veliki broj objekata istog tipa
    - — objekti troše značajan memorijski prostor
  - » deo stanja objekta može da se prebaci u "spoljašnje" stanje
    - — spoljašnje stanje objekta određuje kontekst
  - » kada se ukloni spoljašnje stanje – objekti postaju deljivi
    - — objekti mogu da se zamene referencama ka deljenim objektima
    - — deljeni ("laki") objekti su
      - ▪ bez unutrašnjeg stanja ili
      - ▪ sa unutrašnjim stanjem koje ne zavisi od pojave objekta u kontekstu
  - » identitet objekta nije bitan
    - — testovi identiteta vraćaju true pri poređenju referenci u kontekstu

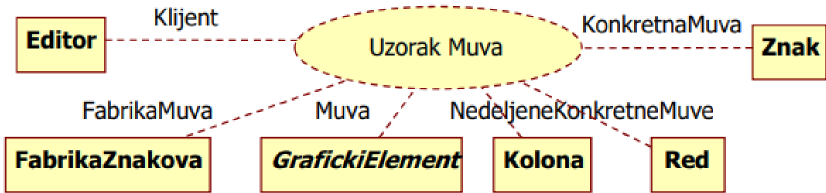
- *Struktura:*



- *Saradnja:*
  - » klijenti ne stvaraju objekte muve direktno, već preko fabrike muva
  - » pri pozivu operacija muve klijenti prenose spoljašnje stanje muvi

- *Učesnici:*
  - » Muva (klasa GrafičkiElement)
    - — deklariše interfejs kroz koji muva prima spoljašnje stanje
  - » KonkretnaMuva (klasa Znak)
    - — implementira interfejs muve i dodaje atribute unutrašnjeg stanja
    - — objekti moraju biti deljivi, ne smeju da zavise od spoljašnjeg stanja
  - » NedeljenaKonkretnaMuva (klase Kolona, Red)
    - — u hijerarhiji objekata muva neki objekti mogu biti nedeljivi
      - ▪ oni čuvaju i atribute spoljašnjeg stanja (zavisnog od konteksta)
      - ▪ oni nisu listovi u objektnoj hijerarhiji (deljive muve su listovi)
  - » FabrikaMuva (klasa FabrikaZnakova)
    - — stvara muve i čuva ih, obezbeđujući propisno deljenje i dohvatanje
      - ▪ kada klijent zahteva muvu, fabrika dohvata postojeću ili kreira novu
  - » Klijent (klasa Editor)
    - — održava reference na dodeljene muve
    - — čuva ili izračunava spoljašnje stanje objekata muva

- *UML notacija:*



- *Posledice:*
  - » dobra strana: ušteda u prostoru – zavisi od:
    - — smanjenja broja objekata
    - — odnosa unutrašnjeg i spoljašnjeg stanja po objektu
    - — da li se spoljašnje stanje čuva ili izračunava
  - » mana: potencijalni troškovi pronalaženja i izračunavanja spoljašnjeg stanja

- *Povezani uzorci:*
  - » često se kombinuje sa uzorkom Kompozicija (Sastav, Sklop)
    - — formiranje stabla sa nedeljivim muvama u čvorovima i deljenim muvama u listovima
    - — kao u primeru sa tekstom u kolonama i redovima
  - » objekti uzoraka Stanje i Strategija često nemaju unutrašnje stanje, mogu biti muve
  - » objekti deljenih muva mogu biti varijacije Unikata sa ograničenim brojem objekata.

Adapter

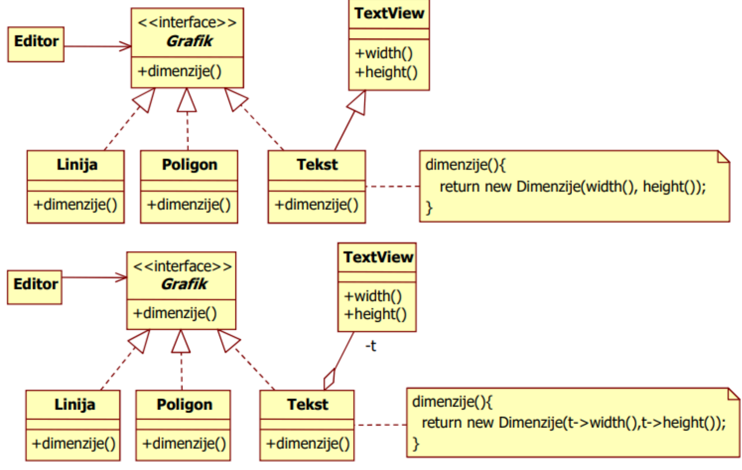
- *Ime i klasifikacija:* Adapter – klasni i objektni uzorak strukture (različite realizacije)

- *Namena:*
  - » konvertuje interfejs klase u drugi interfejs koji klijenti očekuju
  - » omogućava da rade zajedno klase koje inače to ne bi mogle, zbog različitog zahtevanog i realizovanog interfejsa

- *Drugo ime:* Omotač (engl. Wrapper) – dvoznačno, koristi se i za Dekorater

- *Motivacija:*
  - » ponekad neka korisna klasa iz biblioteke nije upotrebljiva
    - — razlog: njen interfejs ne odgovara domenski specifičnom interfejsu kakav očekuje aplikacija
  - » razmatra se primer grafičkog editora
    - — editor omogućava crtanje i aranžiranje grafičkih elemenata
    - — grafički elementi su linije, poligoni, tekst - oni se nalaze na crtežima
    - — ključna apstrakcija grafičkog editora je grafički objekat
    - — grafički objekat ima editabilan oblik i može sam da se nacrta
    - — interfejs za grafičke objekte je definisan apstraktnom klasom (Grafik)
    - — editor definiše potklase klase Grafik za svaki tip grafičkog objekta: Linija, Poligon, Tekst, ...

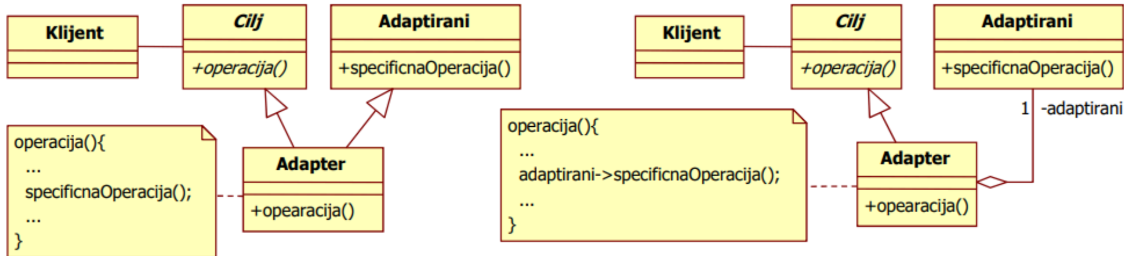
  - » klasu Tekst je značajno kompleksnije implementirati od drugih klasa
  - » postojeća klasna biblioteka nudi klasu TextView - za prikaz i editovanje teksta
  - » problem:
    - — klasna biblioteka nije imala u vidu klasu Grafik kada je definisan TextView
    - — ne mogu se koristiti naslednici Grafik i TextView polimorfno (različiti interfejsi)
  - » rešenje:
    - — definisati Tekst na način da adaptira interfejs TextView prema Grafik
    - — implementirati Tekst pomoću TextView
  - » ovo se može uraditi na jedan od dva načina:
    - — implementacijom interfejsa Grafik i nasleđivanjem TextView ili
    - — agregacijom TextView u Tekst
  - » ova dva pristupa odgovaraju klasnoj i objektnoj verziji uzorka Adapter



- *Primenljivost:*
  - » generalno, uzorak treba koristiti kada:
    - — želimo da koristimo neku raspoloživu klasu koja nema interfejs kakav nam odgovara
    - — želimo da kreiramo reupotrebljivu klasu koja sarađuje sa nepovezanim i nepredviđenim klasama, tj. klasama čiji interfejsi nisu neophodno kompatibilni
  - » objektni adapter treba koristiti kada:
    - — treba koristiti nekoliko postojećih klasa, ali je nepraktično adaptirati njihove interfejse višestrukim izvođenjem iz svake od tih klasa

- *Struktura:*

klasni adapter: objektni adapter:

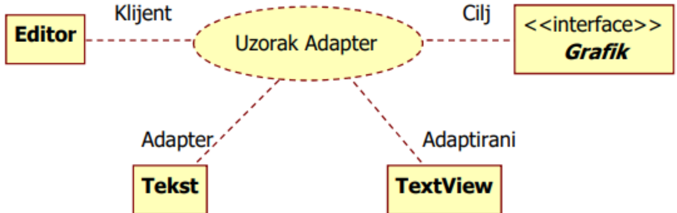


- *Učesnici:*
  - » Cilj (interfejs Grafik)
    - — definiše domenski-specifičan interfejs koji koristi klijent
  - » Klijent (klasa Editor)
    - — sarađuje sa objektima koji poštuju interfejs Cilj
  - » Adaptirani (klasa TextView)
    - — definiše neki postojeći realizovani interfejs koji zahteva adaptiranje
  - » Adapter (klasa Tekst)
    - — adaptira interfejs Adaptirani na interfejs Cilj

- *Saradnja:*
  - » klijenti pozivaju operacije objekta adaptera, a adapter poziva operacije adaptiranog podobjekta (nasleđivanjem ili agragacijom) da izvrše zahtev

- *Posledice:*
  - » klasni Adapter ima sledeće karakteristike:
    - — ne odgovara kada se želi istovremeno adaptiranje više potklasa neke klase
    - — dopušta adapteru kao potklasi da redefiniše metode adaptirane natklase,
    - — uvodi samo jedan objekat i nema potrebe za dodatnom indirekcijom da se stigne do adaptiranog objekta
  - » objektni Adapter ima sledeće karakteristike:
    - — dopušta jednom adapteru da radi sa hijerarhijom adaptiranih klasa (adapter može da doda funkcionalnost za sve adaptirane klase odjednom)
    - — identitet ciljnog objekta i adaptiranog objekta se razlikuje
  - » dvosmerni Adapter – zadržava i interfejs Adaptirani
    - — objekat se može koristiti i kao Cilj i kao Adaptirani
    - — postiže se transparentnost za različite klijente

- *UML notacija:*



- *Implementacija:*
  - C++: klasni uzorak može da se reallizuje
    - javnim izvođenjem iz Cilj i privatnim iz Adaptirani
    - ukoliko se ne želi dvosmerni adapter

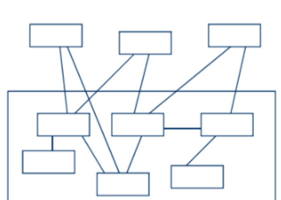
- *Povezani uzorci:*
  - i Dekorater i objektni Adapter prave omotače nekog objekta
  - Dekorater dopunjuje drugi objekat ne menjajući mu interfejs, a Adapter upravo menja interfejs adaptiranog objekta
    - posledica: Dekorater podržava rekurzivne kompozicije, a Adapter ne
  - Most ima sličnu strukturu objektnom adapteru
    - namena je drugačija

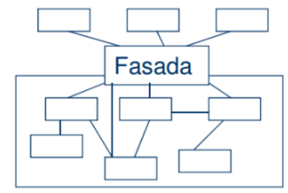
Fasada

- *Ime i klasifikacija:* Fasada (engl. Facade) – objektni uzorak strukture

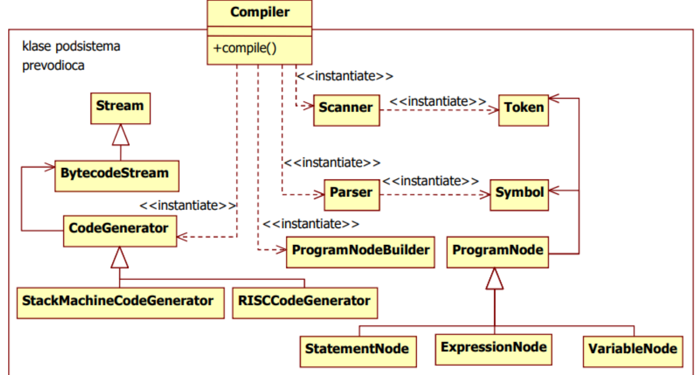
- *Namena:*
  - » pružanje jednistvenog interfejsa podsistema
  - » definisanje interfejsa višeg nivoa da bi se podsistem lakše koristio

- *Motivacija:*
  - » strukturiranje sistema u podsisteme pojednostavljuje proces projektovanja (i održavanja)
  - » cilj je da se međuzavisnosti podsistema svedu na minimum
  - » uvođenje objekta “fasada” ide u pravcu ovog cilja





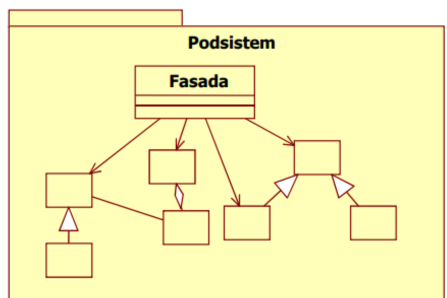
    - — fasada nudi uprošćeni interfejs za opšte upotrebe podsistema
  - » fasada ne brani direktan pristup klasama unutar podsistema
  - » primer: podsistem prevodioca:
    - — klase: Scanner, Parser, ProgramNode, BytecodeStream, …
    - — klasa Compiler ostvaruje opštenamenski interfejs podsistemu
      - ▪ ona predstavlja fasadu podsistema prevodioca
      - ▪ ona većini klijenata pruža dovoljan servis (kompletno prevođenje)
    - (a) olakšava se posao većini programera
      - ▪ ona ne skriva ostale klase koje pružaju servise nižeg nivoa
    - (a) ne sprečava se pristup pojedinim klasama unutar podsistema
    - (b) pristup drugim klasama potreban je samo specijalnim aplikacijama



- *Primenljivost:* uzorak treba koristiti kada
  - » potrebno je dati jednostavan interfejs složenom podsistemu
    - — složenost podsistema raste tokom razvoja
    - — primena uzoraka pospešuje porast broja manjih klasa
    - — podsistem postaje teži za upotrebu klijentima
    - — većini klijenata je dovoljan interfejs podsistema preko fasade
  - » postoje brojne zavisnosti između klijenata i klasa podsistema
    - — fasada razdvaja podsistem od klijenata i drugih podsistema
  - » postoji potreba da se rasloje podsistemi

— fasada se može koristiti za ulaznu tačku svakog nivoa podsistema

- *Struktura:*

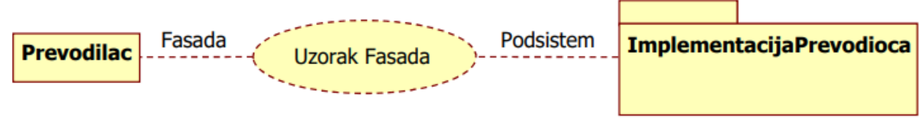


- *Učesnici:*
  - » Fasada (klasa Compiler)
    - — zna koje klase podsistema su odgovorne za koji zahtev
    - — delegira zahteve klijenata odgovarajućim objektima podsistema
  - » klase podsistema
    - — implementiraju funkcionalnosti
    - — izvršavaju zahteve fasade
    - — ne znaju ništa o fasadi

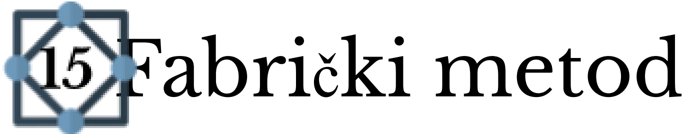
- *Saradnja:*
  - » klijenti šalju zahteve fasadi
  - » fasada prosleđuje zahteve objektima podistema

- *Posledice:*
  - » smanjivanje broja objekata od kojih klijenti zavise
    - — olakšava korišćenje podsistema
  - » slabo vezivanje između podistema i klijenata
    - — olakšava održavanje podsistema
      - ▪ promena klasa podsistema ne utiče na klijenta
      - ▪ promena u podsistemu ne zahteva ni ponovno prevođenje klijenta
    - — povećava portabilnost (pod)sistema
      - ▪ izmene u podsistemu zbog prenosa ne utiču na druge podsisteme
  - » uzorak ne sprečava korišćenje ostalih klasa podsistema

- *UML notacija:*



- *Povezani uzorci:*
  - » Apstraktna fabrika predstavlja Fasadu za kreiranje objekata podsistema
  - » potreban je samo jedan objekat Fasade, pa se realizuje kao Singleton
  - » Posrednik je sličan Fasadi samo što on apstrahuje proizvoljne komunikacije između objekata (podsistema), a Fasada apstrahuje interfejs podsistema
    - — objekat Posrednik može da dodaje funkcionalnost
    - — objekti podsistema znaju za objekat Posrednik
    - — objekti podsistema ne znaju za objekat Fasada



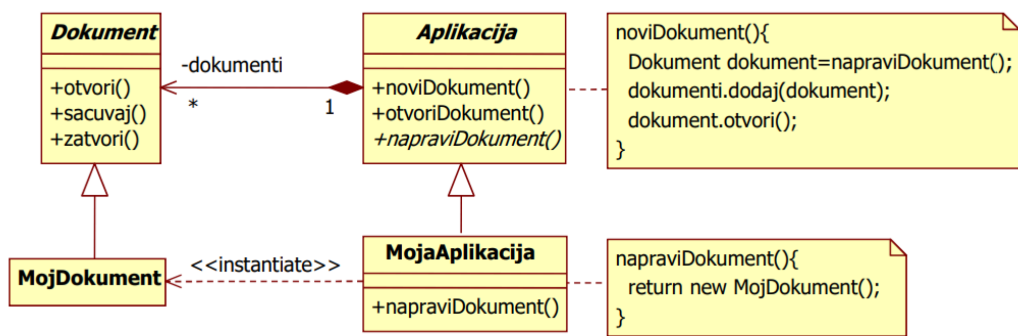
15 Fabrički metod

- *Ime i klasifikacija:* Fabrički (proizvodni) metod (Factory Method) – klasni uzorak kreiranja

- *Namena:*
  - » uzorak dopušta klasi da delegira stvaranje objekata potklasi
  - » definiše interfejs za stvaranje objekata, ali ostavlja potklasama da odluče čije objekte stvaraju

- *Drugo ime:* Virtuelni konstruktor (Virtual Constructor)

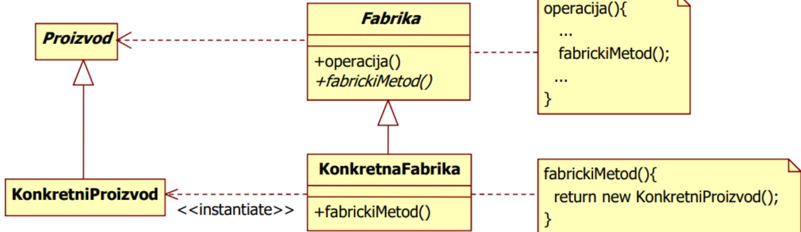
- *Motivacija:*
  - » razmatra se radni okvir (framework) za aplikacije koje mogu da rade sa više dokumenata
  - » ključne apstrakcije su apstraktne klase Aplikacija i Dokument
  - » programeri konkretnih aplikacija treba da naprave potklase u kojima će da ostvare specifične realizacije



  - » klasa aplikacije je odgovorna za upravljanje dokumentima
    - — ona stvara dokument kada korisnik zahteva Otvori ili Novi iz menija
    - — ona zna samo kada se stvara dokument
    - — ona ne zna konkretan tip dokumenta čiji objekat treba stvoriti
  - » radni okvir (framework)
    - — skup opštih klijentskih klasa
    - — korisnik projektuje serverske (klijentske pozivaju serverske)
    - — radni okvir mora da stvara objekat klase dokumenta
    - — on poznaje samo apstraktnu klasu dokumenta
  - » konkretna potklasa aplikacije
    - — nadjačava apstraktni metod klase aplikacije iz radnog okvira za stvaranje specifičnih dokumenata za datu aplikaciju
    - — metod za stvaranje dokumenata je proizvodni (fabrički) metod (odgovoran je za proizvodnju objekata konkretnog dokumenata)

- *Primenljivost:* uzorak treba koristiti kada:
  - » klasa ne može da predvidi klasu objekata koje mora da stvara
  - » klasa želi da njene potklase odrede tipove objekata čije će stvaranje ona zahtevati (ili će njoj biti zahtevano)

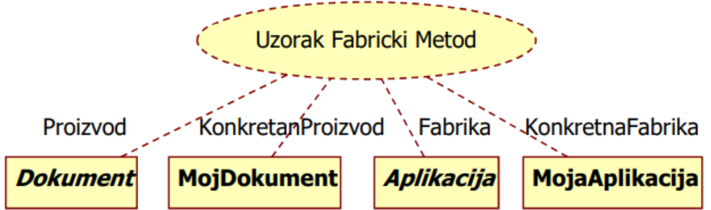
- *Struktura:*



- *Učesnici:*
  - » Proizvod (klasa Dokument)
    - — definiše interfejs objekata koje fabrickiMetod() stvara
  - » KonkretanProizvod (klasa MojDokument)
    - — implemetira interfejs Proizvod
  - » Fabrika (klasa Aplikacija)
    - — deklariše fabrickiMetod() koji vraća objekat tipa Proizvod
    - — može da definiše podrazumevanu implementaciju za fabrickiMetod(), koji vraća podrazumevani objekat tipa KonkretniProizvod
    - — iz neke operacije poziva fabrickiMetod() da kreira objekat Proizvod
  - » KonkretnaFabrika klasa (klasa MojaAplikacija)
    - — redefiniše fabrickiMetod() tako da vrati objekat KonkretanProizvod

- *Saradnja:*
  - » Fabrika delegira svojim potklasama da definišu fabrickiMetod() tako da on vraća odgovarajući objekat KonkretanProizvod

- *UML notacija:*



- *Posledice:*
  - » uzorak eliminiše potrebu da se klijentski kod vezuje za aplikativno-specifične klase
    - — klijent radi preko interfejsa Proizvod, pa može da radi sa bilo kakvim klasama KonkretanProizvod
    - — za klijenta je i konkretan tip prozvoda pri stvaranju transparentan
  - » Fabrički metod daje potklasama mogućnost da obezbede proširenu verziju nekog objekta

- *Povezani uzorci:*
  - » Fabrički metod se obično poziva iz Šablonskog metoda
  - » Prototip ne zahteva izvođenje potklase iz klase Fabrika
    - — on zahteva klon() operaciju u klasi Proizvod (Prototip), koju poziva Fabrika
    - — Fabrički metod ne zahteva takvu operaciju
  - » Apstraktna fabrika se često implementira pomoću Fabričkog metoda

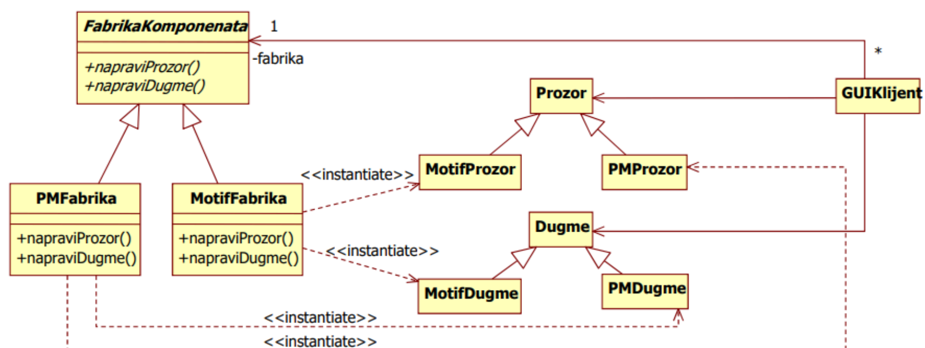
16 Apstraktna fabrika

- *Ime i klasifikacija:* Apstraktna fabrika (eng. Abstract Factory) – objektni uzorak kreiranja

- *Namena:*
  - » obezbeđuje interfejs za proizvodnju različitih varijanti familije povezanih proizvoda
  - » klijent nije svesan konkretnih klasa familije proizvoda

- *Drugo ime:* Kit

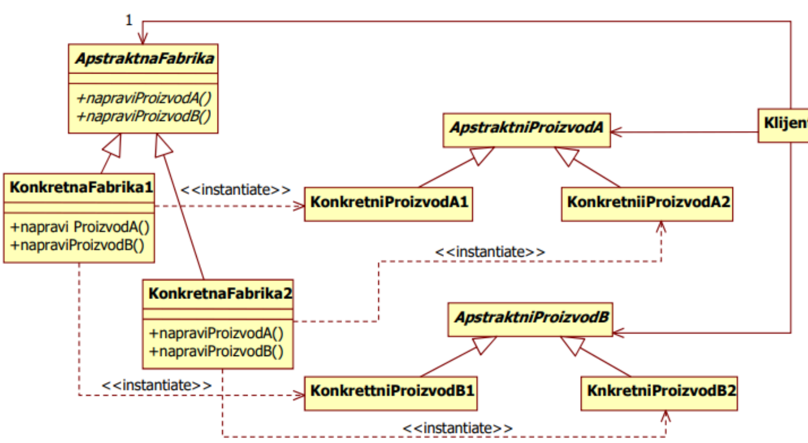
- *Motivacija:*
  - » razmatra se klasna biblioteka za realizaciju korisničkog interfejsa
    - — biblioteka podržava više standarda izgleda-i-osećaja (look-and-feel)
    - — primeri su Motif i Presentation Manager (PM)
  - » različit izgled i osećaj definiše različite pojave komponenata (widget)
    - — primeri su prozori (windows), klizači (scroll bars) i dugmad (buttons)
  - » aplikacija treba da bude portabilna između standarda izgleda i osećaja
    - — ne smeju se fiksno kodirati stvari izgleda i osećaja
    - — ne treba kreirati objekte specifičnog izgleda svuda po aplikaciji
  - » rešenje problema - definiše se apstraktna klasa FabrikaKomponenata
    - — klasa predstavlja apstrakciju fabrike familije komponenata
    - — klasa definiše interfejs za kreiranje svih vrsta komponenata od interesa
    - — za svaku vrstu komponente postoji apstraktan metod za kreiranje



  - » potrebna je po jedna izvedena klasa stvarne fabrike za svaki standard izgleda i osećaja
    - — one implementiraju operacije kreiranja koje je propisala apstraktna fabrika
    - — operacije kreiranja poštuju specifičan standard izgleda i osećaja
  - » potrebna je po jedna apstraktna klasa za svaku vrstu komponenata
    - — konkretne potklase implementiraju komponentu za jedan standard izgleda
  - » klijenti metodima apstraktne fabrike proizvode objekte komponenata
    - — klijenti nisu svesni konkretne fabrike koju koriste za proizvodnju komponenata
      - ▪ ostaju nezavisni od izgleda i osećaja
    - — klijenti moraju da poštuju interfejs koji definiše apstraktna fabrika
    - — samo pri kreiranju objekta konkretne fabrike klijent je svesan njegovog tipa
  - » osim apstraktne fabrike klijenti vide i apstraktne komponente (neopterećene specifičnostima izgleda i osećaja)

- *Primenljivost:* uzorak treba koristiti kada
  - » sistem treba da bude konfigurisan jednom od više varijanti familije proizvoda
  - » sistem treba da bude nezavisan od načina
    - — kreiranja proizvoda
    - — predstavljanja proizvoda
  - » potrebno je forsirati da proizvodi iz jedne varijante familije budu korišćeni zajedno
  - » potrebno je ponuditi klasnu biblioteku proizvoda otkrivajući samo interfejse, ne implementacije

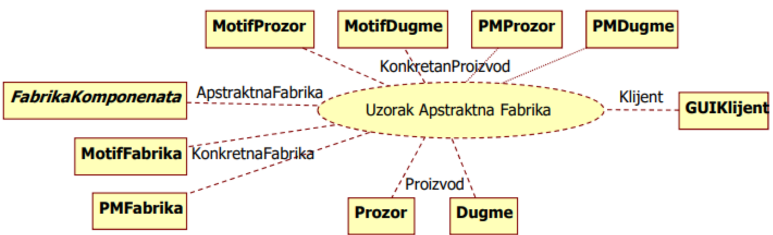
- *Struktura:*



- *Učesnici:*
  - » ApstraktnaFabrika (klasa FabrikaKomponenata)
    - — deklariše interfejs za operacije koje kreiraju objekte apstraktnih proizvoda
  - » KonkretnaFabrika (klase PMFabrika i MotifFabrika)
    - — implementira operacije koje kreiraju objekte konkretnih proizvoda
  - » ApstraktniProizvodX (klase Prozor i Dugme)
    - — deklariše interfejs za određeni tip proizvoda
  - » KonkretanProizvod (klase PMProzor, PMDugme, MotifProzor, MotifDugme)
    - — implementira interfejs apstraktnog proizvoda
    - — definiše proizvod koji će biti kreiran pomoću odgovarajuće konkretne fabrike
  - » Klijent (klasa GUIKlijent)
    - — koristi samo interfejse deklarisane pomoću apstraktne fabrike i apstraktnog proizvoda

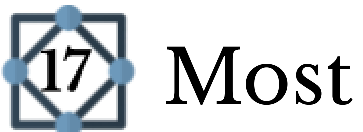
- *Saradnja:*
  - » apstraktna fabrika odlaže kreiranje proizvoda do njenih potklasa – konkretnih fabrika
  - » uobičajeno treba kreirati samo po jedan primerak konkretne fabrike

- *UML notacija:*



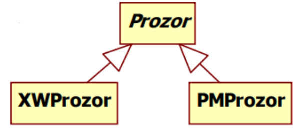
- *Posledice:*
  - » Apstrahovanje konkretnih klasa proizvoda
    - — klijent manipuliše proizvodima kroz njihove apstraktne interfejse
    - — klijent ne koristi imena konkretnih klasa proizvoda (čak ni za njihovo stvaranje)
  - » Olakšava izmenu familije proizvoda
    - — klasa konkretne fabrike se pojavljuje samo jednom u aplikaciji – tamo gde se pravi njen objekat
    - — aplikacija može da koristi različite konfiguracije proizvoda menjanjem konkretne fabrike
    - — pošto apstraktna fabrika kreira kompletnu familiju, cela familija se menja odjednom
  - » Unapređuje konzistenciju među proizvodima
    - — aplikacija koristi objekte proizvoda samo iz jedne familije u jednom trenutku
  - » Problem: podrška novoj vrsti proizvoda nije jednostavna
    - — razlog je taj što apstraktna fabrika fiksira skup proizvoda koji se mogu kreirati
    - — podrška novog proizvoda zahteva proširenje apstraktne fabrike i svih potklasa

- *Povezani uzorci:*
  - » Apstraktna fabrika se često implementira sa Fabričkim metodom ili koristeći Prototip
    - — konkretne fabrike mogu realizovati fabrilke metode koji konstruišu prozvode (uobičajeno)
    - — „apstraktna“ fabrika može biti i konkretna klasa parametrizovana prototipskim proizvodima čije klonove stvara
  - » konkretna fabrika je često Unikat
  - » konkretna fabrika predstavlja konkretnu Strategiju kreiranja familije proizvoda

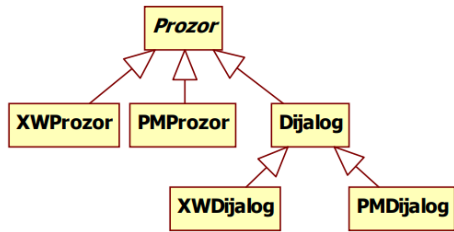


Most

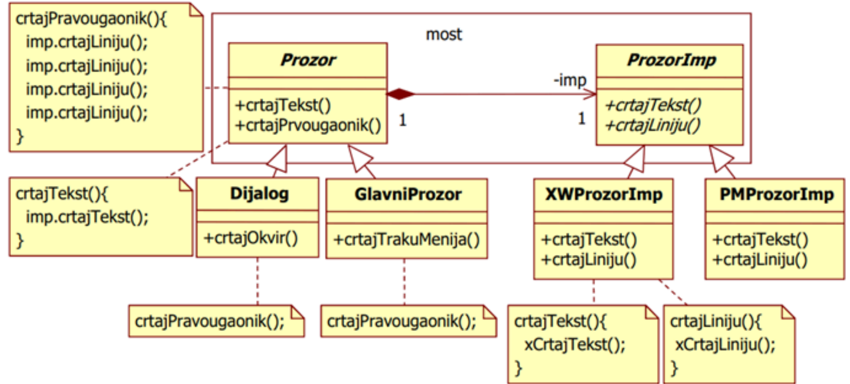
- *Ime i klasifikacija:* Most (eng. Bridge) – objektni uzorak strukture
- *Namena:* razdvaja apstrakciju od njene implementacije da bi se mogle nezavisno menjati
- *Drugo ime:* Ručka/Telo (Handle/Body)
- *Motivacija:*
  - » problem: postoji više implementacija za jednu apstrakciju
  - » tradicionalno OO rešenje: apstraktna klasa i izvedene klase
  - » primer: apstrakcija Prozor u alatima za GUI
    - — potrebno je razvijati aplikacije za razne prozorske sisteme
      - ▪ npr. X Window System i IBM Presentation Manager
    - — tradicionalno rešenje:



  - » rešenje nije dovoljno fleksibilno
    - — nasleđivanje čvrsto vezuje implementaciju za apstrakciju
    - — teško se nezavisno menjaju, proširuju i ponovo koriste
  - » 1. problem: uvodi se nova apstrakcija - prozor za dijalog Dijalog
    - — da bi se apstrakcija implementirala na obe platforme dobija se:



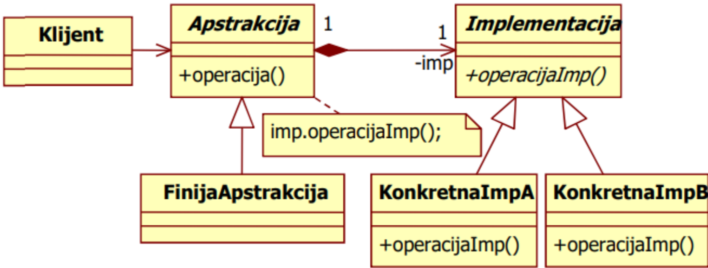
    - — za svaku novu vrstu prozora – moraju se definisati po dve klase
    - — podrška za novu platformu – po jedna klasa za svaku vrstu prozora
  - » 2. problem: klijentski kod postaje zavisan od platforme
    - — kad god klijent pravi prozor – pravi primerak konkretne klase sa specifičnom implementacijom za datu platformu
    - — otežano prenošenje klijentskog koda na druge platforme
    - — klijenti ne bi trebalo da se vezuju za konkretne implementacije
      - ▪ oni treba da vode računa samo o različitim apstrakcijama prozora
      - ▪ samo bi implementacija prozora smela da zavisi od platforme
  - » rešenje problema: uzorak Most
    - — apstrakcija prozora i njegova implementacija su dva odvojena korena odgovarajućih hijerarhija klasa
    - — uspostavlja se most između apstrakcije i implementacije
    - — apstrakcije i implementacije se mogu nezavisno menjati



» sve operacije klase Prozor se implementiraju primenom apstraktnih operacija klase ProzorImp

- *Primenljivost:* uzorak treba koristiti kada
  » treba izbeći trajno vezivanje apstrakcije i njene implementacije
    — npr. ako je potrebno implementaciju menjati u vreme izvršenja
  » i apstrakciji i implementaciji je potrebno proširivanje kroz potklase
    — most omogućava kombinovanje različitih apstrakcija i implementacija
  » promena u implementaciji apstrakcije ne sme da utiče na klijente
  » u jeziku C++: potpuno sakrivanje implementacije klase od klijenta
    — definicije klasa su u h fajlovima – delimično se otkriva implementacija
  » postoji opasnost od prevelikog broja klasa
    — npr. u primeru se vidi kvadratni rast (broj sistema x broj tipova prozora)
  » kada se želi da istu implementaciju deli više objekata a da to bude sakriveno od klijenta (eventualno uz brojanje referenci)

- *Struktura:*

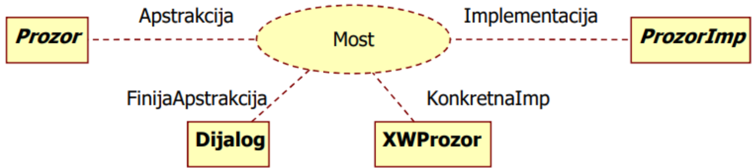


- *Učesnici:*
  » Apstrakcija (klasa Prozor)
    — definiše interfejs apstrakcije
    — održava referencu na objekat implementacije
  » FinijaApstrakcija (klasa Dijalog)
    — proširuje interfejs apstrakcije
  » Implementacija (klasa ProzorImp)
    — definiše interfejs klasa implementacija – interfejs ne mora da liči na interfejs apstrakcije
  » KonkretnaImpX (klase XWProzorImp, PMProzorImp)
    — implementira interfejs implementacije
    — definiše konkretnu implementaciju

- *Saradnja:*
  » Apstrakcija prosleđuje klijentske zahteve objektu Implementacija

- *Posledice:*
  - » razdvajanje ugovora od implementacije
    - — implementacija nije trajno vezana za apstrakciju, može da se konfiguriše dinamički
    - — eliminisane su zavisnosti implementacije i apstrakcije u vreme prevođenja
      - ▪ izmena klasa apstrakcije ne zahteva prevođenje klasa implementacije
      - ▪ izmena konkretnih implementacija ne izaziva prevođenje klasa apstrakcije
  - » bolje mogućnosti proširivanja
    - — hijerarhije apstrakcija i implementacija se mogu nezavisno proširivati
  - » skrivanje detalja implementacije od klijenta
    - — klijent ne vidi ništa od implementacije, vidi samo apstrakciju
    - — bitno kada se mora obezbediti kompatibilnost verzija biblioteke klasa
      - ▪ može se promeniti kompletna implementacija
      - ▪ samo ako se menja njen interfejs, menja se apstrakcija
      - ▪ klijent se ne menja do god se ne promeni interfejs apstrakcije

- *UML notacija:*



- *Povezani uzorci:*
  - » Apstraktna fabrika može da kreira i konfiguriše Most
  - » Adapter i Most prilagođavaju klijentu interfejs neke implementacije
    - — Adapter se obično projektuje retroaktivno
    - — Most se obično projektuje unapred

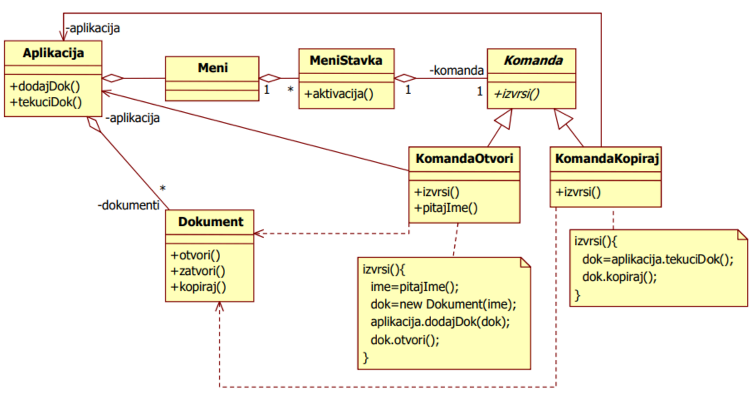
18 Komanda

- *Ime i klasifikacija:* Komanda (engl. Command) – objektni uzorak ponašanja

- *Namena:*
  - » kapsulira zahtev u jedan objekat, omogućavajući:
    - — da se klijenti parametrizuju različitim zahtevima,
    - — da se zahtevi isporučuju kroz red čekanja,
    - — da se pravi dnevnik (log) zahteva i
    - — da se efekti izvršenog zahteva ponište (undo)

- *Drugo ime:* Akcija, Transakcija (engl. Action, Transaction)

- *Motivacija:*
  - » nekad je potrebno izdati zahtev da se izvrši neka operacija bez znanja o samoj zahtevanoj operaciji i izvršiocu (primaocu zahteva)
  - » npr. GUI biblioteke sadrže objekte kao što su dugmad ili meniji
    - — ovi objekti izvršavaju zahteve koji su posledica akcije korisnika
    - — biblioteka klasa ne može da realizuje adekvatne operacije u ovim objektima
    - — samo ciljna aplikacija zna šta je specifična operacija za neko dugme
    - — projektant biblioteke ne zna ništa o operaciji ni o primaocu zahteva
  - » uzorak Komanda:
    - — dopušta objektima biblioteke da zahtevaju da nepoznate operacije budu izvršene od nepoznatih objekata aplikacije
    - — to postiže smeštajući zahteve za operacijama u posebne objekte
  - » objekat sa zahtevom se može zapamtiti ili proslediti drugom objektu

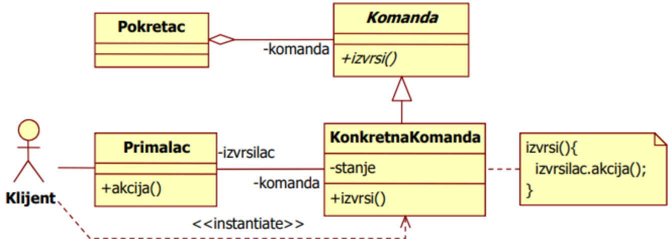


  - » ključna apstrakcija uzorka je klasa Komanda
    - — deklariše interfejs za izvršenje operacija
    - — u najjednostavnijoj formi, interfejs se sastoji od apstraktne operacije izvrsi()
  - » potklase klase Komanda specificiraju par (primalac komande, operacija)
    - — primalac zna kako da izvrši akcije koje su potrebne za ispunjenje zahteva
  - » meniji se lako mogu implementirati koristeći objekte potklasa klase Komanda
  - » uzorak raspreže objekat pozivaoca operacije od onog ko zna kako da je izvrši
    - — objekat koji izdaje komandu treba da zna samo kako se ona izdaje
    - — on ne treba da zna ništa o tome kako se izvršava i ko je izvršava
  - » komande se mogu menjati dinamički
    - — u konkretnom primeru menija, ovo omogućava kontekstno zavisne menije

- » jednostavna je implementacija skriptova (složenih komandi - sekvenci operacija)

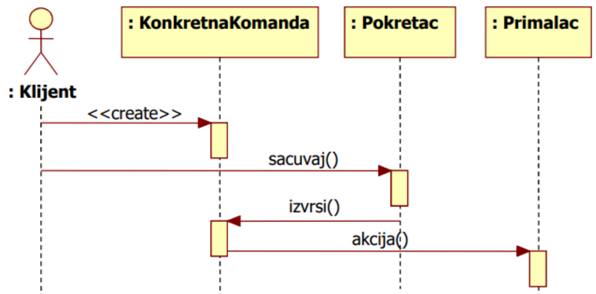
- *Primenljivost:*
  - » kada treba parametrizovati objekte akcijom koju treba da obave
    - — zamena za funkciju povratnog poziva (callback) u tradicionalnim jezicima
  - » kada treba specificirati i stavljati u red čekanja zahteve a kasnije ih izvršavati
    - — objekat komande može imati različit životni vek od onog ko izdaje zahtev
    - — objekat komande se može prepustiti drugom procesu (promena adresnog prostora), ako se primalac može adresirati univerzalno
  - » kada treba podržati undo
    - — izvrsi() operacija može sačuvati u samom objektu stanje za rastauraciju
    - — interfejs treba da sadrži i ponisti() operaciju koja restaurira stanje
    - — neograničen nivo undo i redo se postiže smeštanjem objekata izvršenih komandi u listu, odnosno prolaskom kroz listu unazad i unapred
  - » kada treba podržati oporavak u slučaju kraha (recovery)
    - — potrebno je snimati promene da bi se one mogle ponovo uraditi
    - — u interfejs klase Komanda se dodaju operacije za perzistenciju dnevnika promena
    - — oporavak se postiže učitavanjem dnevnika sa diska i ponovnim izvršavanjem izvrsi()
  - » kada treba podržati transakcije
    - — transakcije su složene operacije sastavljene od primitivnih
    - — transakcije kapsuliraju skup promena podataka

- *Struktura:*

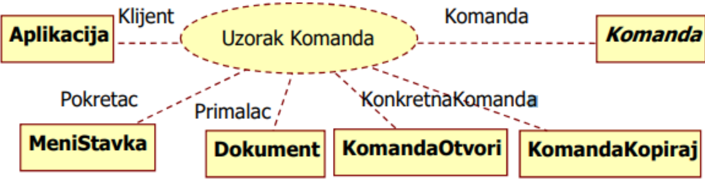


- *Učesnici:*
  - » Komanda (klasa Komanda)
    - — deklariše interfejs za izvršenje neke operacije
  - » KonkretnaKomanda (klase KomandaOtvori, KomandaKopiraj)
    - — definiše vezu između jednog objekta Primalac i akcije
  - » Kijent (klasa Aplikacija)
    - — kreira objekat KonkretnaKomanda i postavlja njen objekat Primalac
  - » Pokretac (klasa MeniStavka)
    - — traži od komande da izvrši zahtev
  - » Primalac (klasa Dokument)
    - — zna kako da izvrši operacije pridružene ispunjavanju zahteva

- *Saradnja:*



- *UML notacija:*



- *Posledice:*
  - » raspreže objekat koji pokreće operaciju od onog koji zna kako da je izvrši
  - » komande su objekti kao i svi drugi i njima se može manipulisati
  - » komande se mogu asemblirati u kompozitne komande (makrokomande)
  - » jednostavno je dodavanje novih komandi, ne treba menjati postojeće klase

- *Povezani uzorci:*
  - » Kompozicija se koristi za kreiranje makrokomandi (skriptova)
  - » Podsetnik može da čuva stanje pre izvršenja komande potrebno za Undo
  - » komanda čija se kopija stavlja u dnevnik se ponaša kao Prototip

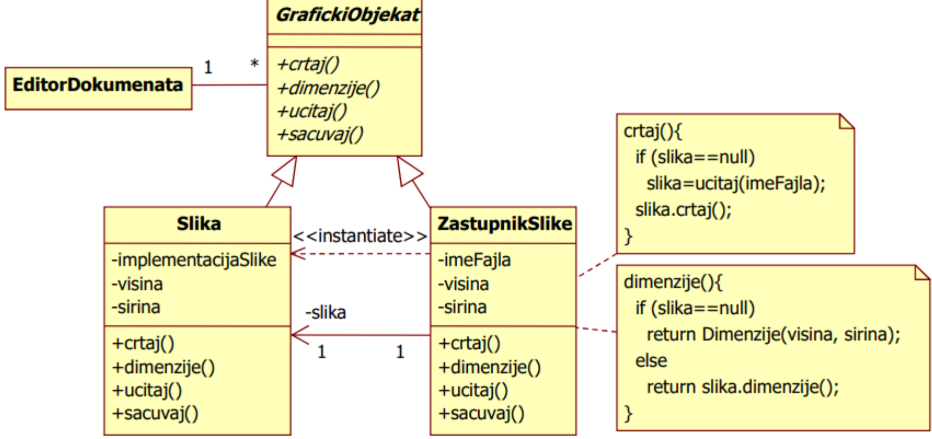
19 Zastupnik

- *Ime i klasifikacija:* Zastupnik (engl. Proxy) - objektni uzorak strukture

- *Namena:*
  - » realizuje zamenu (surogat) drugog objekta koji:
    - — kontroliše pristup originalnom objektu
    - — odlaže punu cene kreiranja i inicijalizacije originala do trenutka kada je ovaj zaista potreban

- *Drugo ime:* Predstavnik, Zamena, Surogat (engl. Surogate)

- *Motivacija:*
  - » problem: editor dokumenata koji mogu da sadrže grafičke objekte u sebi
    - — neki grafički objekti kakve su velike rasterske slike su skupi za kreiranje
    - — sa druge strane, otvaranje dokumenta treba da bude brzo
    - — pri otvaranju dokumenta treba izbeći kreiranje svih slika koje dokument sadrži
    - — sliku treba otvarati tek kad postaje vidljiva (stvarno potrebna), na zahtev
    - — činjenicu da se slika kreira tek na zahtev sakrivamo, da se ne bi komplikovao editor
    - — ova optimizacija ne treba da utiče npr. na kod za prikazivanje ili formatiranje dokumenta
  - » rešenje:
    - — u dokument stavljamo umesto realne slike njenog zastupnika, koji je zamenjuje
    - — zastupnik se ponaša kao slika i stvara objekat slike kad je baš on neophodan
    - — u slučaju da su slike u fajlovima, zastupnik čuva ime fajla kao referencu na sliku
    - — zastupnik slike takođe čuva dimenzije slike
    - — zastupnik može da da dimenzije slike formateru i bez učitavanja slike
    - — zastupnik stvara objekat slike tek kada editor od njega zahteva da se slika prikaže
    - — naredne zahteve zastupnik prosleđuje slici, jer čuva referencu na objekat slike



  - » EditorDokumenata pristupa ugrađenim slikama kroz interfejs apstraktne klase GrafickiObjekat
  - » ZastupnikSlike je klasa za slike koje se kreiraju na zahtev
    - — ona održava ime fajla kao referencu na sliku koja je na disku
    - — ime fajla se prosleđuje kao argument konstruktora
    - — ona čuva i dimenzije slike i referencu na objekat stvarne slike
    - — referenca na objekat slike je null dok se objekat slike ne kreira
    - — operacija crtaj() proverava da li je objekat slike kreiran pre nego što mu prosledi zahtev
    - — operacija dimenzije() prosleđuje zahtev objektu slike, ako ovaj postoji

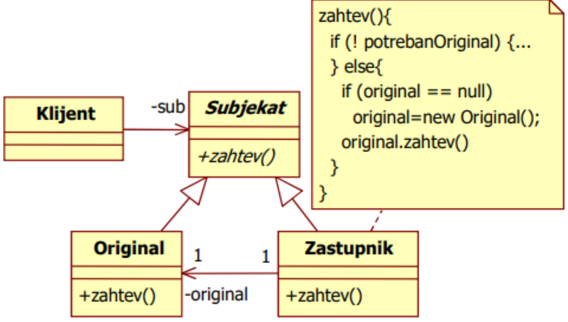
— ako slika nije kreirana dimenzije() vraća sačuvane dimenzije slike

- *Primenljivost:*
  - » uzorak je primenljiv kada postoji potreba za sofističnom referencom na objekat

- *Vrste zastupnika prema primeni:*
  - » udaljeni zastupnik (remote proxy)
    - — obezbeđuje lokalnog predstavnika objekta koji se nalazi u drugom adresnom prostoru
    - — Colpien ovu vrstu zastupnika naziva "ambasadorom"
  - » virtuelni zastupnik (virtual proxy)
    - — kreira relativno skup objekat na zahtev (ZastupnikSlike je primer)
  - » zaštitni zastupnik (protection proxy)
    - — kontroliše pravo pristupa originalnom objektu

- *Implementacija "pametni pokazivač" (smart reference):*
  - » zamena za obični pokazivač, obavlja dodatne akcije prilikom pristupa
  - » tipične primene:
    - — brojanje referenci na objekat, da bi se ovaj dealocirao kad je broj 0
    - — punjenje perzistentnog objekta u memoriju pri prvom referisanju
    - — provera da li je objekat zaključan, t.j. da ga neko drugi tada ne menja

- *Struktura:*

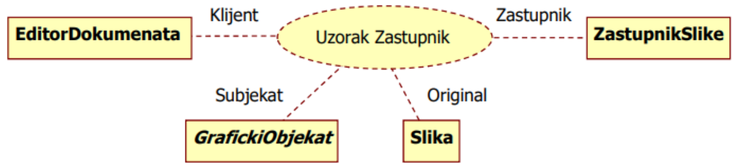


- *Učesnici:*
  - » Subjekat (klasa GrafickiObjekat)
    - — zajednički interfejs za Original i Zastupnik, da se Zastupnik može koristiti kao Original
  - » Zastupnik (klasa ZastupnikSlike)
    - — čuva referencu za pristup originalu
    - — realizuje interfejs subjekta tako da može predstavljati zamenu za original
    - — kontroliše pristup originalu, može biti odgovoran za njegovo kreiranje/uništavanje
    - — u zavisnosti od tipa zastupnika:
      - ▪ Udaljeni je odgovoran za slanje zahteva i argumenata u drugi adresni prostor
      - ▪ Virtuelni može keširati dodatne informacije o originalu da odloži pristup
      - ▪ Zaštitni proverava da li pozivalac ima pristupnu dozvolu za dati zahtev
  - » Original (klasa Slika)
    - — realni subjekat koji je reprezentovan zastupnikom

- *Saradnje:*

   - » Zastupnik prosleđuje zahteve objektu Original kada treba, u zavisnosti od vrste zastupnika
- *Posledice:*
   - » uzorak Zastupnik uvodi jedan nivo indirekcije u pristupu objektu
   - » optimizacija koju zastupnik može da sakrije od klijenta je copy-on-write
      - — kopiranje velikih i komplikovanih objekata može biti skupa operacija
      - — ako se kopija nikad ne modifikuje, nije neophodno platiti cenu
      - — korišćenjem zastupnika da odloži kopiranje, cena se plaća samo ako se objekat modifikuje
      - — zahtev za kopiranje rezultuje samo u inkrementiranju broja referenci na subjekat
      - — kada klijent zahteva operaciju koja modifikuje original, tada zastupnik
         - ▪ stvarno kopira original
         - ▪ dekrementira broj referenci na original
      - — kada broj referenci padne na 0 – original se uništava

- *UML notacija:*



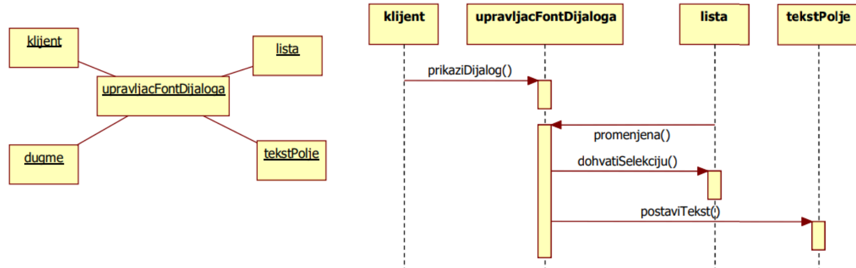
- *Povezani uzorci:*
   - » Adapter obezbeđuje različit interfejs objektu koji adaptira, dok Zastupnik obezbeđuje isti interfejs, kao i Dekorater
   - » Dekorater – može imati sličnu implementaciju kao Zastupnik, ali ima različitu namenu:
      - — Dekorater dodaje jednu ili više odgovornosti objektu, a Zastupnik kontroliše pristup objektu
      - — Zaštitni zastupnik može biti implementiran baš kao Dekorater

Posrednik

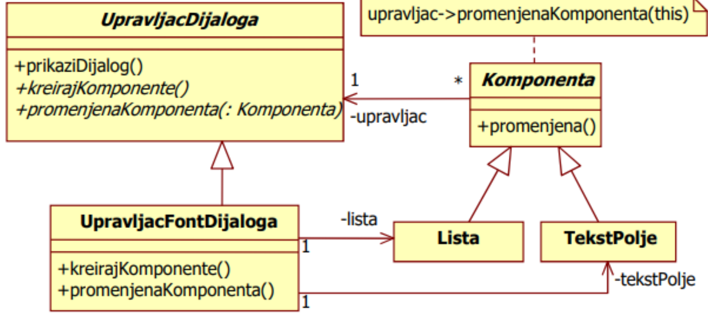
- *Ime i klasifikacija:* Posrednik (engl. Mediator) – objektni uzorak ponašanja

- *Namena:*
  - » definiše objekat koji kapsulira interakciju skupa objekata
  - » omogućava slabo sprezanje skupa objekata
    - — uklanja potrebu uzajamnog referisanja
    - — omogućava da im se interakcija menja nezavisno

- *Motivacija:*
  - » OO projektovanje podstiče distribuciju odgovornosti među objektima
    - — posledica je da raste broj objekata koji se lakše održavaju
    - — međutim, raste i broj veza među objektima – što otežava održavanje
    - — ako postoji mnogo zavisnosti među objektima – sistem deluje kao monolitan
      - ▪ da bi se promenilo ponašanje sistema mora da se menja više klasa
  - » primer: implementacija dijaloga u GUI (dugmad, polja za unos, liste,...)
    - — međuzavisnost komponenti (npr. onemogućeno dugme dok je polje prazno)
  - » rešenje: posrednik (medijator)
    - — kapsulira interakciju (ponašanje) skupa objekata
    - — objekti se ne poznaju uzajamno – poznaju samo objekat posrednika
    - — smanjuje se broj veza
    - — kada se selektuje stavka u listi potrebno je upisati je u tekst-polje
    - — brigu preuzima sam dijalog na kojem su lista i tekst-polje



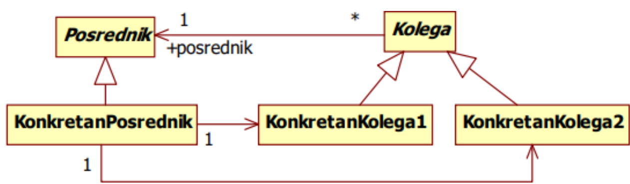
klasni dijagram:



- *Primenljivost:*
  - » kada skup objekata komunicira na dobro definisan ali složen način;
    - — međuzavisnosti su brojne, nisu strukturirane i teško se shvataju
  - » kada je reupotreba objekata teška zato što komuniciraju sa mnogim drugim objektima
  - » kada treba omogućiti prilagođavanje ponašanja distribuiranog na više klasa, a da se izbegne mnogo potklasa

— prilagođava se samo potklasa posrednika

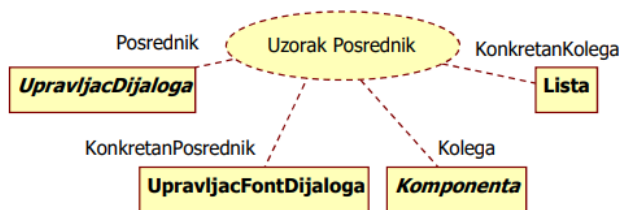
- *Struktura:*



- *Učesnici:*
  - » Posrednik (klasa UpravljacDijaloga)
    - — definiše interfejs za komunikacijusa objektima kolega
  - » KonkretanPosrednik (klasa UpravljacFontDijalog)
    - — implementira spregnuto ponašanje koordiniranjem objekata kolega
    - — poznaje i održava kolege
  - » Kolega (klasa Komponenta)
    - — interfejs za konkretne kolege
    - — poznaje samo klasu Posrednik
  - » KonkretanKolegaX (klase Lista i TekstPolje)
    - — nasleđuje poznavanje klase Posrednik

- *Saradnja:*
  - » kolege šalju i primaju zahteve od objekta posrednika
  - » posrednik rutira zahteve između odgovarajućih kolega

- *UML notacija:*



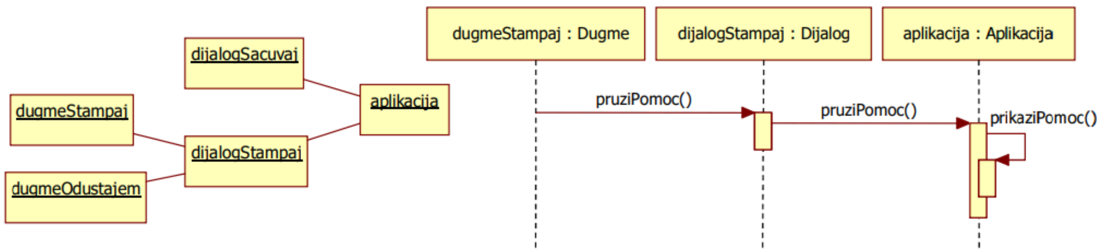
- *Povezani uzorci:*
  - » Fasada ima sličnosti sa Posrednikom (jedna klasa prema podsistemu) ali se razlikuje od uzorka Posrednik, jer ona:
    - — apstrahuje podsistem objekata da obezbedi pogodniji interfejs
    - — pri tome koristi jednosmeran protokol
      - ▪ samo se objekat fasade obraća objektima podsistema
    - — Posrednik koristi dvosmerni protokol u komunikaciji sa kolegama
  - » Slična objektna struktura sa Posmatračem
    - — Posrednik je centralni objekat sa kojim su povezane kolege

21 Lanac odgovornosti

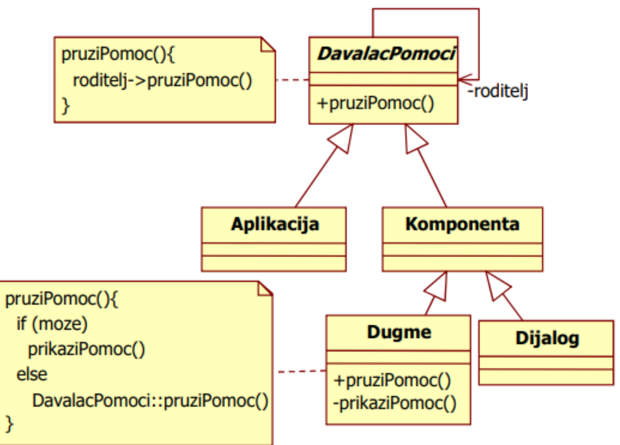
- *Ime i klasifikacija:* Lanac odgovornosti (engl. Chain of Responsibility) - objektni uzorak ponašanja

- *Namena:*
  - » povezuje objekte primaoce zahteva u lanac i prosleđuje zahtev niz lanac, dok ga neki objekat ne obradi
  - » izbegava neposredno vezivanje pošiljaoca zahteva sa primaocem, dajući šansu većem broju objekata da obrade zahtev

- *Motivacija:*
  - » sistem pomoći zavisne od konteksta u grafičkom korisničkom interfejsu
    - — treba omogućiti da se od svake komponente može dobiti pomoć
    - — ako za konkretnu komponentu ne postoji pomoć, treba da se prikaže pomoć za dijalog koji sadrži komponentu
    - — ako ni za dijalog ne postoji pomoć – treba da se prikaže pomoć za program
  - » informacije pomoći se organizuju prema opštosti
    - — od specifičnijih (u komponentama) prema opštijim (dijalog, aplikacija)
  - » zahtev za pomoć treba da obrađuje nekoliko objekata
    - — od specifičnijih ka opštijim, dok neko ne obradi u celini
  - » problem:
    - — objekat koji pruža pomoć nije unapred poznat onome ko zahteva pomoć
    - — potrebno je razdvojiti onoga ko inicira zahtev za pomoć (dugme) od objekta koji će na kraju da prikaže pomoć
  - » rešenje:
    - — lanac odgovornosti: zahtev se predaje duž lanca dok ga neko ne obradi

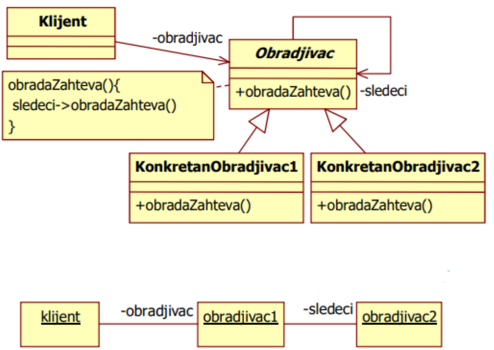


  - » da bi se obezbedilo prosleđivanje kroz lanac a da primaoci ostanu implicitni, svi objekti u lancu moraju imati isti interfejs



- *Primenljivost:*
  - kada više objekata može da obradi zahtev, ali se ne zna unapred koji od njih će ga obraditi
  - kad se hoće izdati zahtev jednom od nekoliko objekata, a da se ne odredi eksplicitno primalac
  - kada skup objekata koji obrađuju zahtev treba da se odredi dinamički (vrši se konfigurisanje lanca u vreme izvršenja)

- *Struktura:*



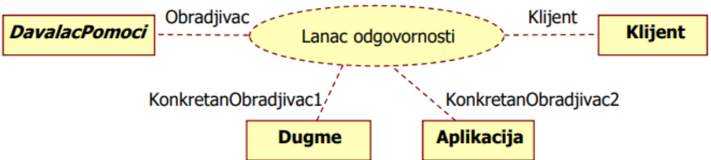
- *Učesnici:*
  - Obradjivac (klasa DavalacPomoci)
    - definiše interfejs obrade zahteva
    - implementira vezu prema sledećem u lancu - opciono, svi osim poslednjeg
  - KonkretanObradjivacX (klase Dugme, Dijalog)
    - obrađuje zahteve koje ume
    - može da pristupi naslednicima u lancu
    - ako može da obradi zahtev – to i čini, u suprotnom ga prosleđuje nasledniku

- *Saradnja*: kada klijent izda zahtev on putuje po lancu odgovornosti dok konkretni obrađivač ne preuzme odgovornost za obradu

- *Posledice:*
  - rasprezanje pošiljaoca i primaoca
    - pošiljalac i primalac ne treba da znaju ništa jedan o drugom
    - objekat u lancu ne treba da poznaje strukturu lanca
    - smanjuje se broj veza među objektima
      - objekat ne mora pristupati svima, dovoljno je sledbeniku
  - dodatna fleksibilnost u pridruživanju odgovornosti objektima
    - odgovornosti za obradu zahteva se mogu dodavati i menjati u vreme izvršenja
  - prijem nije garantovan

— moguća je situacija da zahtev stigne do kraja lanca, a da nije obrađen

- *UML notacija:*



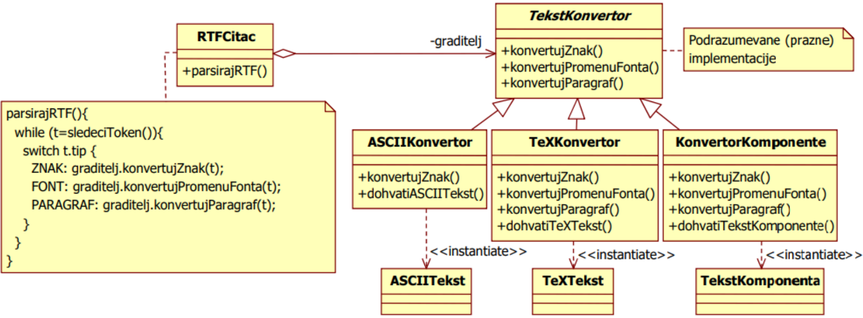
- *Povezani uzorci:*
  - » često se primenjuje sa uzorkom Kompozicija; roditelj komponente može da igra ulogu "sledećeg" u lancu
  - » Dekorater ima sličnu strukturu, ali kod njega svi objekti u lancu nužno učestvuju u obradi zahteva, a i zahtev se svakako na kraju obrađuje od strane dekorisanog subjekta

Graditelj

- *Ime i klasifikacija*: Graditelj (engl. Builder) – objektni uzorak kreiranja

- *Namena:*
  - » razdvaja proces upravljanja konstrukcijom složenog objekta od njegovog sklapanja u celinu i reprezentacije proizvoda
  - » posledica je da isti proces konstrukcije može da kreira različite reprezentacije finalnog proizvoda

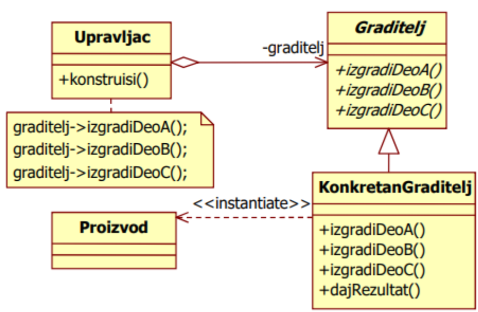
- *Motivacija:*
  - » posmatra se aplikacja za čitanje RTF dokumenata (čitač)
  - » potrebno je da učitane dokumente može da konvertuje u:
    - — čisti ASCII tekst
    - — TeX dokument
    - — tekstualnu komponentu koja može da se interaktivno edituje
    - — ... (broj mogućih konverzija je neograničen)
  - » potrebno je da se lako dodaje nova konverzija, bez izmene čitača
  - » rešenje je primena uzorka Graditelj:
    - — konfigurisanje RTFCitac klase objektom TekstKonvertor
    - — RTFCitac čita RTF tekst i parsira ga
    - — TekstKonvertor konvertuje RTF u drugu tekstualnu reprezentaciju
    - — kad RTFCitac prepozna RTF element – on izdaje zahtev TekstKonvertor objektu
    - — TekstKonvertor konvertuje i reprezentuje element u ciljnom formatu
    - — potklase TekstKonvertor specijalizuju korake u konverziji



  - » svaka vrsta konvertora, iza apstraktnog interfejsa, ima mehanizam za kreiranje i sastavljanje složenog objekta
  - » konvertor je razdvojen od čitača
    - — čitač je odgovoran samo za parsiranje ulaznog RTF dokumenta
  - » svaka klasa konvertora u uzorku se naziva graditeljem (builder)
  - » klasa čitača se naziva upravljačem (director)
  - » u ovom primeru uzorak graditelja je razdvojio
    - — algoritam interpretacije tekstualnog formata (parser za RTF dokumente)
    - — način kako se konvertovani format stvara i reprezentuje
  - » posledica je reupotreba algoritma parsiranja pri kreiranju različitih tekstualnih reprezentacija od RTF dokumenata
    - — RTFCitac se samo konfiguriše objektom neke od potklasa TekstKonvertor

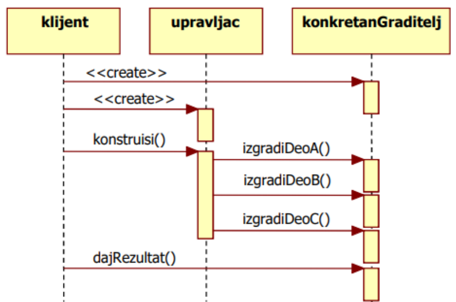
- *Primenljivost:* uzorak treba koristiti kada
  - » algoritam za kreiranje složenog objekta treba da bude nezavisan
    - — od delova koji čine objekat
    - — od načina na koji se delovi sklapaju u celinu
  - » proces konstrukcije mora da dopusti različite reprezentacije za objekat koji se konstruiše

- *Struktura:*



- *Učesnici:*
  - » Graditelj (klasa TekstKonvertor)
    - — specificira apstraktan interfejs za kreiranje delova objekta Proizvod
  - » KonkretanGraditelj (klase ASCIIKonvertor, TeXKonvertor)
    - — konstruiše i sastavlja delove proizvoda implementiranjem interfejsa Graditelj
    - — definiše i čuva proizvod koji kreira
    - — obezbeđuje interfejs za dohvatanje proizvoda
  - » Upravljac (klasa RTFCitac)
    - — konstruiše objekat koristeći interfejs Graditelj
  - » Proizvod (klase ASCIITekst, TeXTekst)
    - — predstavlja složeni objekat koji se konstruiše
    - — uključuje klase koje definišu sastavne delove
      - ▪ uključujući interfejse za sastavljanje delova u finalan rezultat

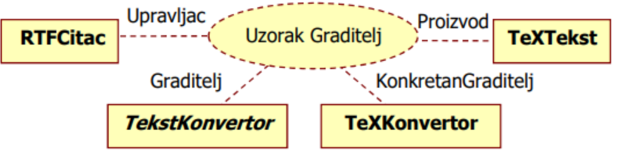
- *Saradnja:*



- *Posledice:*
  - » dopušta izmene interne reprezentacije i načina sklapanja složenog proizvoda
    - — graditelj pruža upravljaču apstraktan interfejs za konstrukciju proizvoda
    - — reprezentacija i interna struktura, kao i način sklapanja su sakriveni
    - — za promenu interne reprezentacije potrebna je samo nova vrsta graditelja
  - » izolovanje koda za konstrukciju i reprezentaciju
    - — bolja modularnost kroz kapsuliranje načina konstrukcije složenog objekta
    - — klijenti ne treba da znaju ništa o klasama koje definišu unutrašnju strukturu proizvoda
      - ▪ te klase se ne pojavljuju u interfejsu graditelja
  - » daje finiju kontrolu nad procesom konstrukcije od drugih fabrika
    - — drugi uzorci kreiranja konstruišu proizvod u jednom potezu

— graditelj konstruiše proizvod korak-po-korak, pod kontrolom upravljača

- *UML notacija:*



- *Povezani uzorci:*
  » Druge fabrike (npr. Apstraktna fabrika) takođe mogu da konstruišu kompleksne objekte
    — Graditelj vrši konstrukciju kompleksnog objekta korak-po-korak, druge fabrike u jednom koraku
    — Graditelj vraća proizvod kao finalni korak, druge fabrike vraćaju proizvod odmah
  » Graditelj često gradi objekat Kompozicije
  » Upravljac uzorka Graditelja se parametrizuje objektom klase Graditelj, kao što se Kontekst parametrizuje objektom Strategija u odgovarajućem uzorku
    — osim u nameni, razlika je što se Strategija najčešće implementira u jednoj metodi dok su kod Graditelja implementirani pojedini koraci gradnje kao posebne metode
  » Šablonska metoda poziva apstraktne metode sopstvene klase, dok kod uzorka Graditelj, Upravljac sadrži konkretnu metodu koja poziva apstraktne metode klase Graditelj

Posetilac

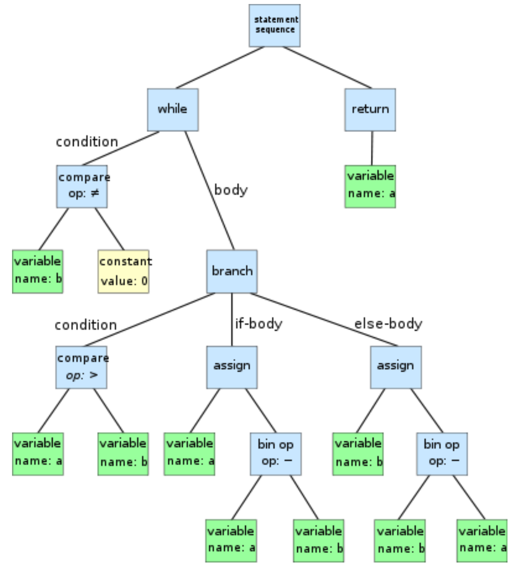
- *Ime i klasifikacija:* Posetilac (engl. Visitor) – objektni uzorak ponašanja

- *Namena:*
  - » postoji skup raznorodnih operacija koje treba primenjivati na raznorodne elemente jedne objektne strukture
  - » uzorak omogućava definisanje nove operacije bez izmene klasa elemenata nad kojima se izvršava

- *Motivacija:*
  - » razmatra se prevodilac koji reprezentuje strukturu programa kao stablo apstraktne sintakse (SAS)
  - » potrebno je obezbediti operacije nad SAS za:
    - — statičku proveru tipova
    - — generisanje koda
    - — ...
  - » većina operacija će na različit način tretirati čvorove SAS za:
    - — naredbu dodele vrednosti
    - — obraćanje promenljivoj
    - — ...
  - » hijerarhija čvorova SAS zavisi od jezika, ali je za dati jezik stabilna

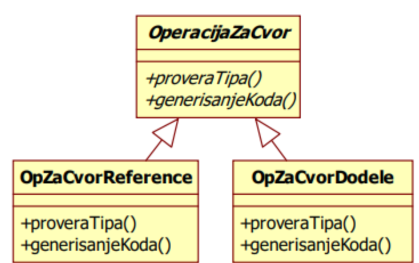
  - » Primer SAS za program:

```
while b ≠ 0
      if (a > b)
          a := a - b
      else
          b := b - a
end_while
return a
```



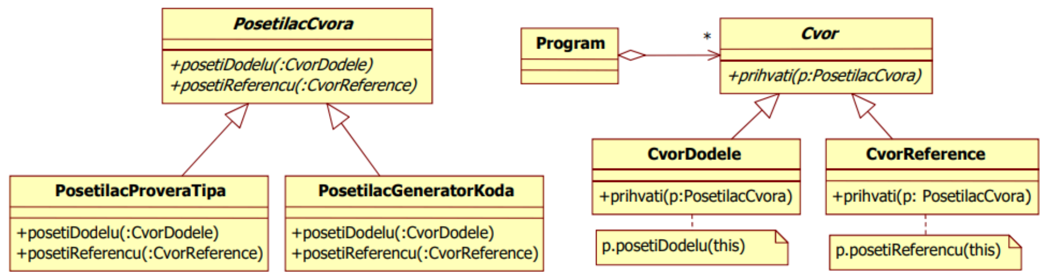
Stablo Apstraktne Sintakse

  - » nefleksibilan dizajn:



  - » problem: prisustvo svih operacija u različitim klasama čvorova vodi do sistema koji je teško razumeti, održavati i menjati
    - — mešanje koda za proveru tipova i koda za generisanje koda kvari razumljivost
    - — dodavanje nove operacije zahteva izmenu svih klasa čvorova
  - » cilj: razdvajanje hijerarhije čvorova od operacija koje se vrše nad njima
  - » rešenje: pakovanje operacija za različite čvorove u objekte posetilaca koji se prosleđuju čvorovima SAS kada se struktura obilazi

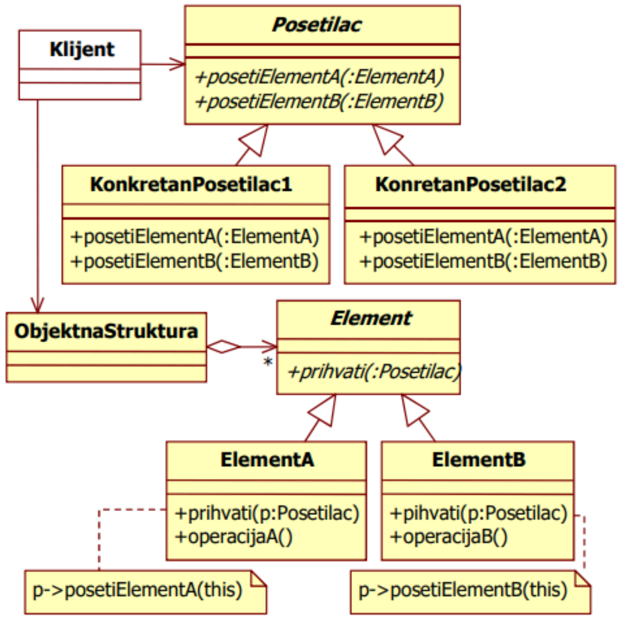
- » posetilac kapsulira jednu operaciju za različite čvorove SAS
- » kada čvor primi posetioca – on poziva operaciju posetioca
  - — operacija odgovara klasi čvora
  - — čvor dostavlja referencu na sebe kao parametar operacije
- » posetilac tada izvršava operaciju za dotični čvor



- » u uzorku Posetilac se definišu dve hijerarhije klasa
  - — za elemente nad kojima se vrše operacije (h. čvorova)
  - — za posetioce koji definišu operacije nad elementima (h. posetilaca)
- » nova operacija se dodaje kao nova potklasa u h. posetilaca
  - — ako bi aplikacija trebalo da računa metriku programa samo bi se definisala nova potklasa klase PosetilacCvora, a ne bi se dodavao aplikativno zavisan kod u klase čvorova SAS

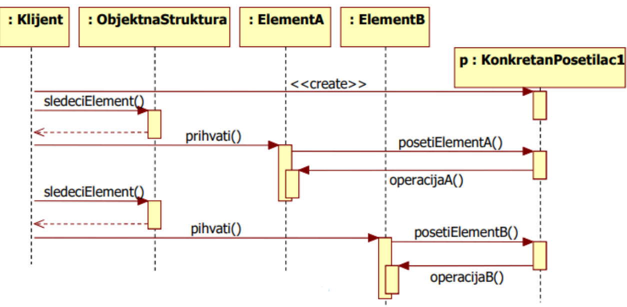
- *Primenljivost:* uzorak treba koristiti kada
  - » jedna objektna struktura sadrži objekte raznih klasa, a potrebno je nad tim objektima izvršiti raznorodne operacije koje zavise od konkretnih klasa objektne strukture
  - » više raznorodnih operacija se izvršava nad objektima strukture, pri čemu se njihove klase ne "zagađuju" tim operacijama
    - — posetilac omogućava grupisanje povezanih operacija (koje odgovaraju jednoj primeni) u jednoj klasi
  - » klase koje definišu strukturu objekata se retko menjaju, a često se definišu nove operacije nad elementima strukture
    - — promena klasa objektne strukture zahteva promenu svih posetilaca
    - — može mnogo da košta, pa ako je ovo često, bolje je definisati operacije u tim klasama

- *Struktura:*



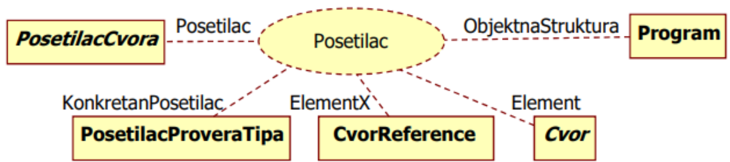
- *Učesnici:*
  - » Posetilac (klasa PosetilacCvora)
    - — po jedna operacija posetiX() za svaku potklasu Element
    - — potpis operacije identifikuje odgovarajući potklasu Element
  - » KonkretanPosetilacX (klase PosetilacProveraTipa,...)
    - — implementira sve operacije koje deklariše Posetilac
    - — predstavlja kontekst algoritma i čuva njegovo stanje koje često akumulira rezultate obilaska strukture objekata
    - — svaka operacija implementira deo algoritma za odgovarajuću klasu objekata u strukturi
  - » Element (klasa Cvor)
    - — deklariše operaciju prihvati() - Posetilac je argument
  - » ElementX (klase CvorDodele, CvorReference)
    - — implementira operaciju prihvati() tako što poziva posetilac.posetiElementX(this)
  - » ObjektnaStruktura (klasa Program)
    - — pruža interfejs visokog nivoa koji omogućava klijentu da obiđe elemente dostavljajući im posetioca
    - — može da bude Sastav (stablo) ili neka zbirka (npr. lista)

- *Saradnja:*
  - » Klijent:
    - — stvara objekat konkretnog posetioca
    - — obilazi objektnu strukturu posećujući svaki njen element
    - — svakom elementu prosledi posetioca
  - » Element:
    - — poziva operaciju posetioca kojom mu traži da ga poseti



- *Posledice:*
  - » grupiše srodne i razdvaja različite operacije
  - » olakšava dodavanje novih operacija (klasa posetilaca)
  - » dodavanje novih klasa konkretnih elemenata nije lako
  - » mogućnost akumuliranja stanja za vreme posećivanja elemenata
    - — bez posetioca, stanje bi se predavalo operacijama kao dodatni argument
  - » moguć nedostatak: probijanje kapsulacije elemenata
    - — često su potrebne javne operacije za pristup unutrašnjem stanju elemenata strukture
    - — proglašavanje posetioca prijateljem u elementu nije dobro rešenje
      - ▪ dodavanje nove vrste operacija (posetioca) bi zahtevalo izmenu svih elemenata

- *UML notacija:*



- *Povezani uzorci:*
  - » Sastav – posetilac može da se koristi da bi se neka operacija primenila na strukturu objekata definisanu kao Sastav (Kompozicija)
  - » Iterator se često koristi zajedno sa Posetiocem - služi za sistematičan obilazak objekata strukture kojima se šalju posetioci

- » Interpreter – posetilac se može koristiti da obavi interpretaciju

- *Ime i klasifikacija:* Interpreter – klasni uzorak ponašanja

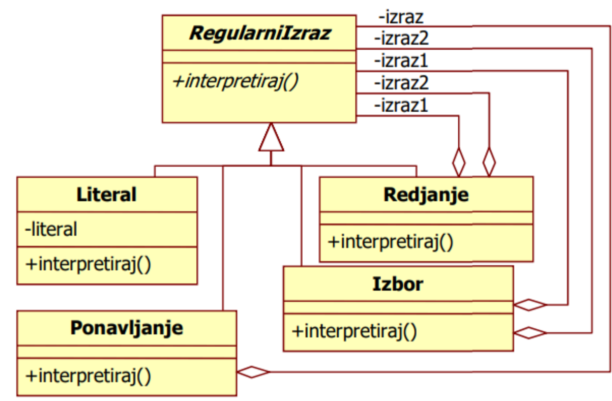
- *Namena:*
  - » za dati jezik, definiše:
    - — reprezentaciju njegove gramatike
    - — interpreter koji koristi tu reprezentaciju da interpretira iskaze jezika

- *Motivacija:*
  - » u nekim aplikacijama:
    - — pojedini zahtevi se mogu predstaviti iskazima jednostavnog jezika
    - — može se napraviti interpreter koji rešava zahteve interpretirajući iskaze
  - » primer: pretraga teksta na osnovu zadatog iskaza (uzorka za pretragu)
    - — iskaz predstavlja opis skupa mogućih niski koje se traže u tekstu
      - ▪ potrebno je u tekstu pronaći bilo koju pojavu niske koja odgovara opisu
    - — regularni izrazi su standardan jezik za iskaze koji opisuju skupove niski
    - — algoritam pretrage može da pokušava uparivanje dela teksta koji se pretražuje interpretirajući regularan izraz koji opisuje skup niski za uparivanje
  - » uzorak Interpreter opisuje:
    - — kako definisati gramatiku (gramatička pravila) za jednostavne jezike
    - — kako predstaviti iskaze u jeziku i
    - — kako interpretirati te iskaze
  - » sledeća gramatika definiše regularne izraze:

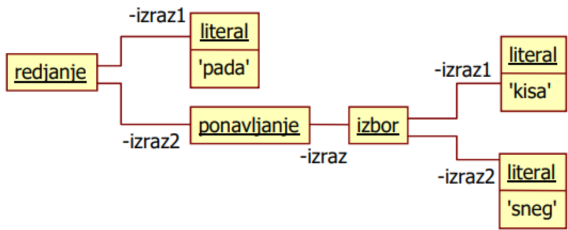
```
izraz = redjanje | izbor | ponavljanje | grupisanje | literal;
redjanje = izraz, "&", izraz;
izbor = izraz, "|", izraz;
ponavljanje = izraz, "*";
grupisanje = "(", izraz, ")";
literal = znak, {znak};
znak = "a" | "b" | "c" | ... ;
```

  - » na primer - zadavanje uzorka niski za pretragu:
    - — `projektni & uzorci & (stvaranja | strukture | ponašanja)`

  - » uzorak Interpreter koristi po klasu da reprezentuje gramatičko pravilo
  - » simboli na levoj strani pravila određuju klase interpretera
  - » data gramatika se opisuje sa 5 klasa
  - » neterminalni simboli na desnoj strani pravila odgovaraju objektima klasa koje se koriste u opisu pravila
  - » napomena: na dijagramu nisu predstavljene sve moguće klase interpretera (Izraz, Grupisanje, Znak)



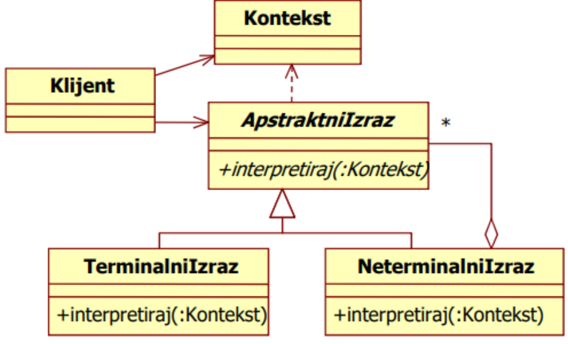
» desna strana pravila određuje sastavljanje objektne strukture
» svaki regularni izraz definisan na datoj gramatici može da se predstavi jednim stablom apstraktne sintakse (SAS) sastavljenim od objekata datih klasa
» primer:
  — regularni izraz: pada & (kisa | sneg)*
  — odgovarajuće SAS:



» može se napraviti interpreter za ovakve regularne izraze
» definiše se interpretiraj()operacija za svaku potklasu RegularniIzraz
  — interpretiraj() dobija kao argument kontekst u kojem interpretira izraz
  — kontekst sadrži ulazni tekst i informaciju o tome koji njegov deo je već obrađen
  — svaka potklasa RegularniIzraz implementira interpretiraj() da upari sledeći deo ulaznog teksta u datom kontekstu
  — na primer:
    ▪ Literal proverava da li ulaz odgovara literalu koji definiše
    ▪ Redjanje proverava da li ulaz odgovara sledu njegovih izraza
    ▪ Izbor proverava da li ulaz odgovara jednom od njegovih izraza
    ▪ Ponavljanje proverava da li ulaz ima ponavljanja njegovog izraza

- *Primenljivost:*
  » uzorak treba primeniti kada
    — postoji jednostavan jezik koji treba interpretirati
    — iskazi jezika se mogu predstaviti kao stabla apstraktne sintakse (SAS)
    — čvorovi u SAS su objekti klasa koje predstavljaju gramatička pravila jezika
  » najbolji rezultati se dobijaju pod sledećim uslovima
    — gramatika je jednostavna
      ▪ za kompleksne gramatike broj klasa u hijerarhiji postaje preveliki za održavanje
    — efikasnost nije kritična
      ▪ efikasni interpreteri se ne implementiraju interpretiranjem stabala parsiranja direktno, već se najpre ona transformišu u neku drugu formu
      ▪ na primer – regularni izrazi se često transformišu u automate stanja, ali i tada se translator može implementirati pomoću uzorka Interpreter

- *Struktura:*



- *Učesnici:*
  - » ApstraktniIzraz (klasa RegularniIzraz)
    - — deklariše apstraktnu operaciju interpretiraj() koja je zajednička za sve čvorove u SAS
  - » TerminalniIzraz (klasa Literal)
    - — implementira interpretiraj() operaciju za terminalne simbole u gramatici
    - — po primerak se zahteva za svaki terminalni simbol u iskazu
  - » NeterminalniIzraz (klase Redjanje, Izbor, Ponavljanje)
    - — zahteva se po jedna klasa za svako gramatičko pravilo R=R1 R2 ... Rn
    - — sadrži objekte tipa ApstraktniIzraz za svaki od simbola R1...Rn
    - — implementira op. interpretiraj() za neterminalne simbole tako što se ova poziva rekurzivno za objekte simbola R1...Rn
  - » Kontekst (klasa Tekst)
    - — sadrži informaciju koja je globalna za interpreter
  - » Klijent
    - — gradi (ili je već dato) SAS koje predstavlja pojedini iskaz na jeziku koji definiše gramatika
      - ▪ SAS je sastavljeno od objekata klasa NeterminalniIzraz i TerminalniIzraz
    - — poziva interpretiraj() operaciju korena SAS

- *Saradnja:*
  - » klijent gradi (ili je već dat) iskaz kao SAS od objekata terminalnih i neterminalnih izraza
  - » zatim klijent inicijalizuje kontekst i pokreće operaciju interpretiraj()korena SAS
  - » svaki čvor neterminalnog izraza definiše operaciju interpretiraj() tako što poziva operaciju interpretiraj() svojih podizraza
  - » operacija interpretiraj() svakog terminalnog izraza definiše krajnju tačku u rekurziji
  - » operacija interpretiraj() u svakom čvoru koristi kontekst da smesti stanje i pristupi stanju interpretacije

- *Posledice:*
  - » lako je menjati i proširivati gramatiku
    - — klase reprezentuju pravila
    - — ova se mogu menjati i dodavati nova kroz izvođenje
  - » lako je implementirati gramatiku
    - — klase koje definišu tipove čvorova SAS imaju slične implementacije
    - — generisanje ovih klasa može čak da bude automatizovano
  - » kompleksne gramatike je teško održavati
    - — za svako gramatičko pravilo se defiiše barem jedna klasa
    - — gramatike sa mnogo pravila rezultuju u velikom broju klasa
    - — treba primenjivati druge tehnike (generatori parsera/prevodioca)
  - » dodavanje novih načina za interpretaciju izraza
    - — dodavanje novog načina za interpretaciju izraza (npr. proveraTipa()) zahteva dodavanje nove operacije u sve klase izraza
    - — ako se ovo radi više puta, treba razmotriti uzorak Posetilac

- *Povezani uzorci:*
  - » SAS je primerak projektnog uzorka Kompozicija
  - » Muva omogućava da se efikasno dele terminalni simboli u SAS
  - » Interpreter može da koristi Iterator za obilazak SAS
  - » ako se operacija interpretiraj() realizuje u posebnoj klasi, onda je moguće lako dodavati druge načine interpretacije - Posetilac

♦ UML notacija: