Objektno-orijentisana analiza informacionih sistema

- Koraci analize informacionih sistema
- Poređenje funkcionalne, OO i SO analize
- Poređenje konceptualnog, logičkog i fizičkog modela
- Poređenje disciplina sistema poslovanja
- Šta je naučeno do sada i šta je ostalo u okviru analize?
- Modelovanje korisničkih zahteva
- Objektno-orijentisana analiza sistema
- Arhitektura softvera
- Use case analiza

Koraci analize informacionih sistema

- Sveobuhvatni koraci analize sistema su:
  1. **Modelovanje poslovanja** (*Business Modeling*) – dva glavna fokusa:
     - » Modelovanje poslovnih **procesa**
     - » Modelovanje **podataka**
  2. **Modelovanje korisničkih zahteva** (*Requirements*)
  3. Detaljna **analiza sistema**, na osnovu prethodnih modela, koja rezultira u kreiranju:
     - » **Konceptualnog** modela sistema
     - » **Logičkog** modela sistema

Poređenje funkcionalne, OO i SO analize

- **Funkcionalna** analiza sistema:
  - Analiza trenutnog stanja sa fokusom na reinženjering postojećeg sistema
  - Nema mogućnost mapiranja korisničkih zahteva!
  - Ne posmatra integrisano podatke i procese!
  - Ne postoji mogućnost modeliranja distribuiranog sistema koji koristi servise drugih provajdera/partnera!
  - Ne prikazuje sve aspekte sistema (komponente sistema, uvođenje - *deployment*…)!
  - Ne postoji mogućnost generisanja koda upotrebom objektno-orijentisanih ili servisno-orijentisanih tehnologija!
  - Logički modeli podataka: ERD, IDEF1X …
  - Logički modeli procesa: DFD, IDEF0 …

Poređenje funkcionalne, OO i SO analize (nastavak)

- **Objektno-orijentisana** (OO) analiza sistema:
  - Prikazuje integrisano podatke i procese
  - Fokus na korisničke zahteve
  - Postoji mogućnost generisanja koda upotrebom objektno-orijentisanih tehnologija
  - Nije prilagođen za analizu distribuiranog sistema (koji koristi spoljne servise drugih provajdera) i uopšte za servisno-orijentisane sisteme koji se zasnivaju na SOA arhitekturi!
  - Logički modeli procesa: Use case, Dijagram aktivnosti, Dijagrami interakcije (sekvenci i komunikacije)
  - Logički model podataka: Konceptualni dijagram klasa

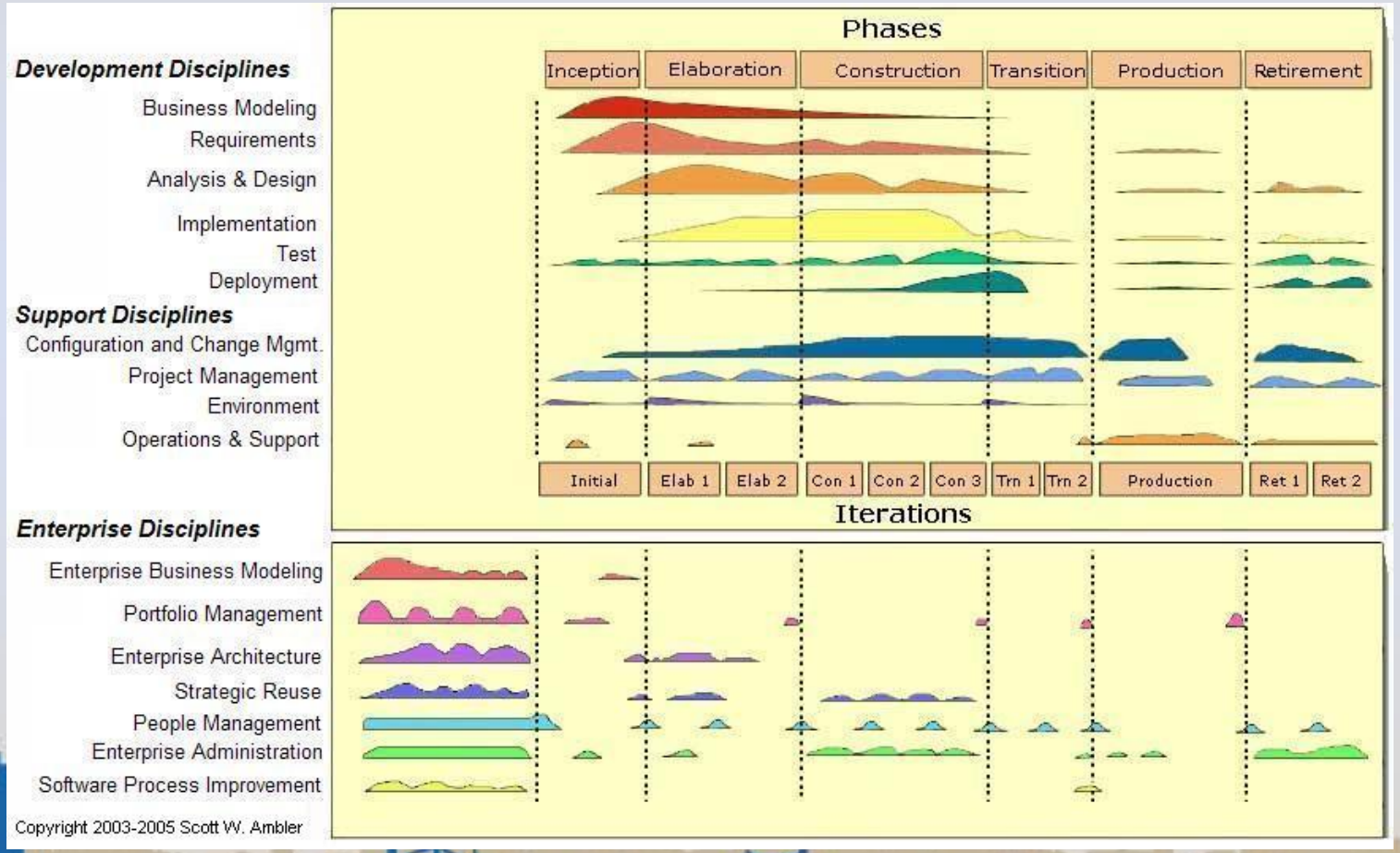
Poređenje funkcionalne, OO i SO analize (nastavak)

- **Servisno-orijentisana** (SO) analiza sistema:
  - Prikazuje integrisano i podatke i procese i kolaboraciju sa spoljnim servisima
  - Automatsko prevođenje u izvršni kod koristeći servisno-orijentisane tehnologije
  - Još uvek nisu usvojeni svi standardi!
  - Logički modeli procesa: BPMN dijagrami: orkestracije, koreografije i kolaboracije
  - Logički modeli podataka: Dijagram konverzacije

Poređenje konceptualnog, logičkog i fizičkog modela

- **Konceptualni/konceptni model**
  - Model visokog nivoa
  - Npr. klijent i arhitekta zgrade listaju zatheve klijenata, kao što su okruženje, dizajn sobe, neophodni uređaji, infrastruktura i sl.
- **Logički model**
  - Model detaljnog nivoa, ali bez aspekta implementacije
  - Identifikovanje svega onoga što sistem treba da radi
  - Npr. arhitekta na osnovu konceptnog modela kreira kompletno okruženje, sobe, infrastrukturu … odakle se vide konkretni zahtevi za električnim kapacitetima, nivoima svetlosti, uređajima za vodovod i sl.
- **Fizički model**
  - Uključen aspekt implementacije sistema, ali ne podrazumeva kodiranje rešenja, već služi kao osnova za fazu razvoja sistema!
  - Npr: biraju se uređaji koji će se koristiti i podržati prethodno definisane zahteve za strujom, kablovima …

Poređenje disciplina razvoja sistema, podrške i discipline poslovanja preduzeća



Šta smo naučili do sada?

- Modelovanje poslovanja (*Business Modeling*):
  - Modelovanje poslovnih procesa
    - Funkcionalno: DFD, IDEF0
    - Objektno-orijentisano: Use case i dijagram aktivnosti
    - Servisno-orijentisano: BPMN dijagrami
  - Modelovanje podataka
    - Funkcionalno: u okviru predmeta Baze podataka
    - Objektno-orijentisano: biće objašnjeno upotrebom konceptualnog i logičkog dijagrama klasa
    - Servisno-orijentisano: Dijagram konverzacije

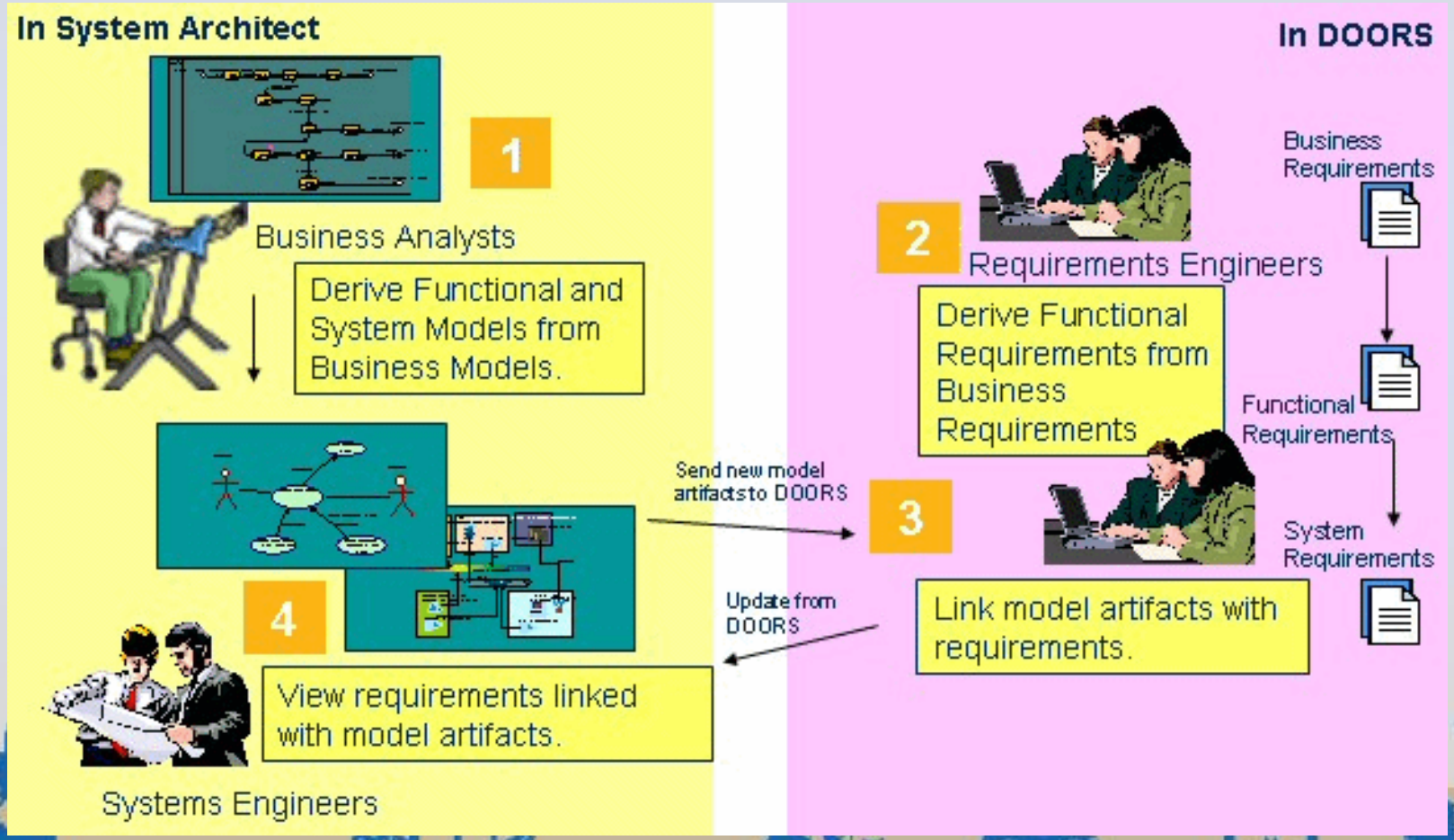
Šta je ostalo da se nauči u okviru analize sistema?

- Modelovanje korisničkih zahteva
  - Koja je razlika između modelovanja poslovanja i modelovanja zahteva?
  - Zašto se modeluju zahtevi?
  - Šta je izloženo čestim promenama, zahtevi ili poslovanje?
- Detaljna analiza sistema
  - Logički dizajn sistema
  - Dijagrami interakcije i konceptualni dijagram klasa

Modelovanje korisničkih zahteva

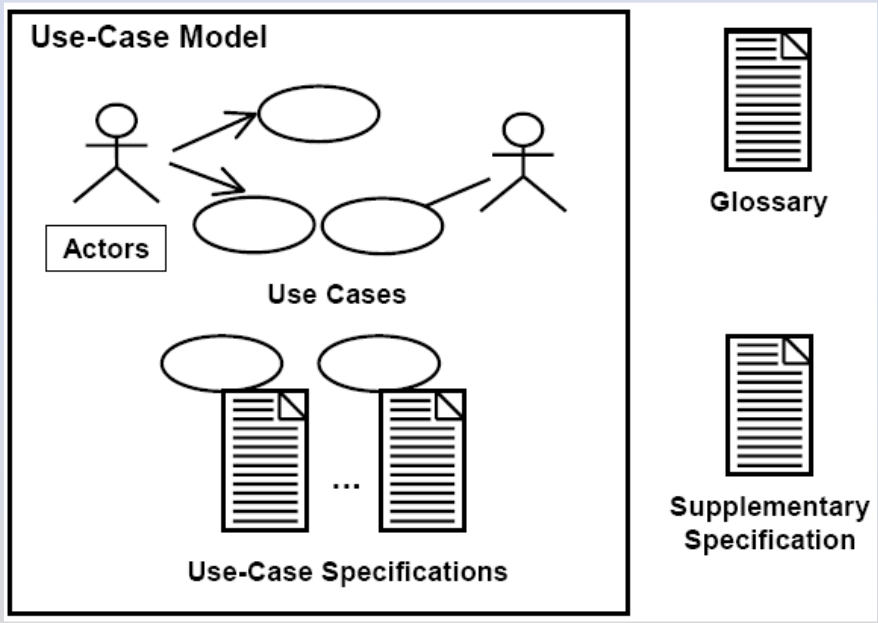
Tip zahteva	Govori nam o …
Poslovni zahtev	cilju/strategijama koje treba da se postignu
Korisnički zahtev	zadatku koji korisnik mora da bude u stanju da uradi kako bi postigao cilj poslovanja
Funkcionalni zahtevi	funkcionalnosti sistema koje sistem mora da obavlja kako bi pomogao korisniku da uradi zadatak koji je neophodan za postizanje poslovnog cilja
Nefunkcionalni zahtevi	osobinama kvaliteta koje sistem treba da ispunjava
Poslovna pravila	politikama/propisima/standardima koje sistem treba da sledi
Zahtevi za podacima	Informacijama koje su potrebne sistemu da bi obavljao svoje funkcije
Ograničenja	Ograničenjima koje su nametnute sistemu

Modelovanje zahteva

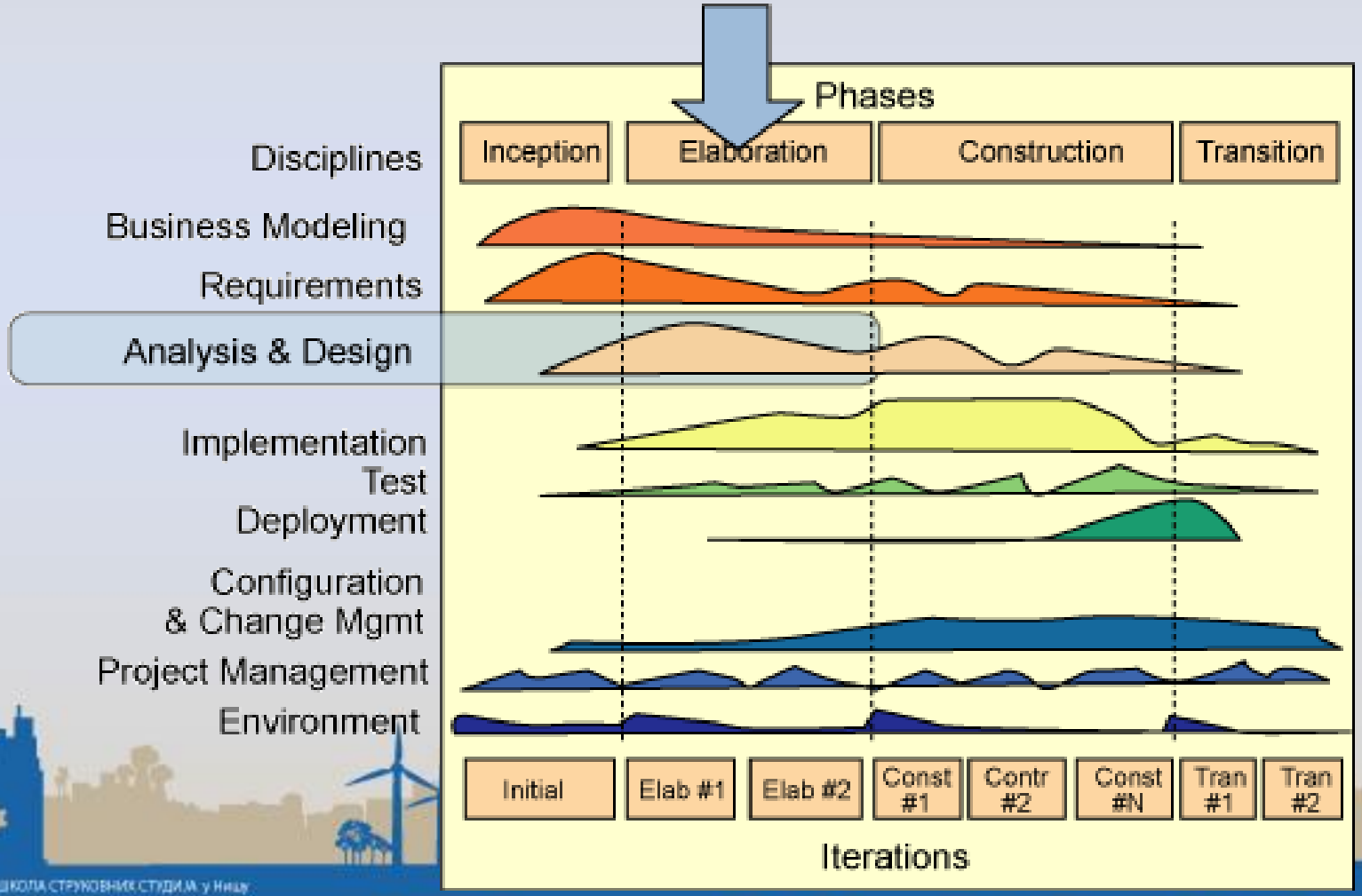


Relevantni artifakti zahteva

- Model slučajeva korišćenja (*use-case*):
  - Use case dijagram
  - Use case specifikacija
- Rečnik (*glossary*)
  - definiše opštu terminologiju
  - Olakšava komunikaciju između eksperata i developera kao i ostalih članova projekta
- Dodatna specifikacija (*supplementary specification*)
  - sadrži dodatne zahteve koji nisu opisani slučajevima korišćenja
  - Npr. nefunkcionalni zahtevi, zahtevi za obučavanjem korisnika …



Objektno-orijentisana analiza sistema



Analiza *vs* Projektovanje sistema

Analiza

- Fokus na razumevanje problema
- Idealizovanje dizajna
- Ponašanje
- Struktura sistema
- Funkcionalni zahtevi
- Mali model

- Cilj je razumevanje problema i početak razvoja vizuelnog modela o tome šta treba da se izgradi, nezavisno od implementacije i tehnologije

- Fokus je na prevođenju funkcionalnih zahteva u softverske koncepte

- Logički dizajn sistema

Projektovanje (*Design*)

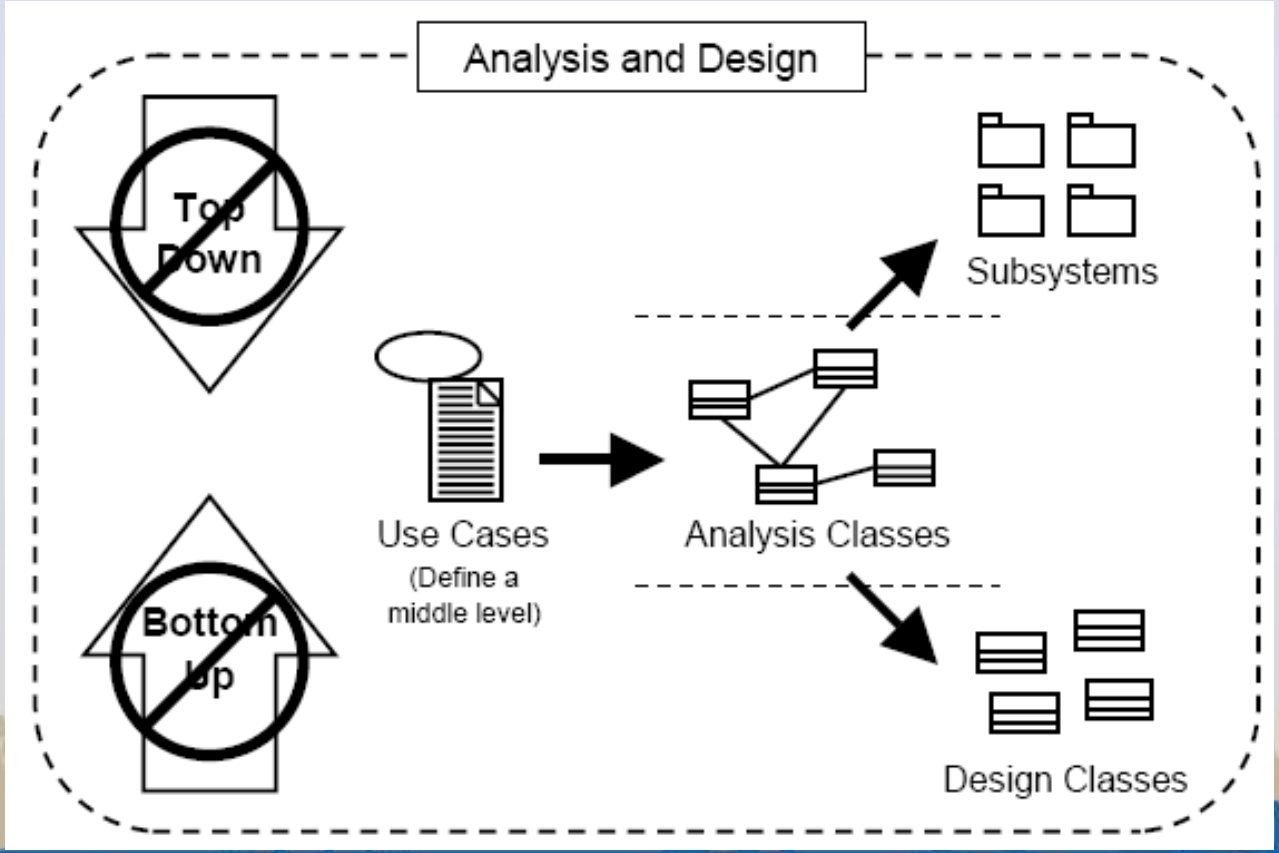
- Fokus na razumevanje rešenja
- Operacije i atributi
- Performanse
- Približavanje realnom kodu
- Životni ciklus objekta
- Nefunkcionalni zahtevi
- Veliki model

- Cilj je prerađivanje modela radi razvoja modela dizajna koji će omogućiti prelaz u fazu kodiranja

- U projektovanju se podešavaju okruženja implementacije i uvođenja

- Fizički dizajn sistema

Objektno-orijentisana analiza nije *top-down* ili *bottom-up* pristup kao kod funkcionalnog modelovanja



Koraci detaljne analize sistema

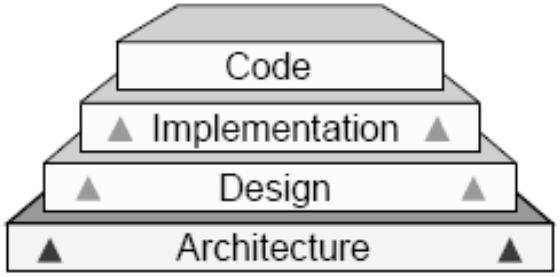
- Arhitektura softvera
  - » Obično se koristi neki od obrazaca (paterna) arhitekture, npr. slojevita, MVC i sl.
  - » Logički prikaz arhitekture sistema sa podsistemima i njihovim relacijama
- Analiza slučajeva korišćenja
  - » Dijagrami sekvenci
  - » Dijagrami komunikacije
  - » Konceptualni dijagram klasa

Šta je arhitektura softvera?

- Arhitektura softvera podrazumeva skup strateških odluka o organizaciji ili strukturi softverskog sistema, koji je prikazan kao kolekcija komponenata koji ispunjavaju željene funkcionalnosti sistema, optimizujući pritom kvalitet, performanse, bezbednost i upravljivost celokupnog sistema
  - Drugim rečima, arhitektura se fokusira na organizovanje komponenata kako bi se podržale specifične funkcionalnosti sistema
  - *"Arhitektura softvera obuhvata niz značajnih odluka o organizaciji softverskog sistema koji se tiču funkcionalnosti, upotrebljivosti, otpornosti, performansi, ponovne upotrebljivosti, razumljivosti, ekonomskih i tehnoloških ograničenja i kompromise."* (Philippe Kruchten, Grady Booch, Kurt Bittner i Rich Reitman)
  - *"Arhitektura softvera je struktura ili struktuiranje sistema koji obuhvata softverske elemente, njihove javno vidljive osobine i veze između njih"* (Bass, Clements i Kazman)

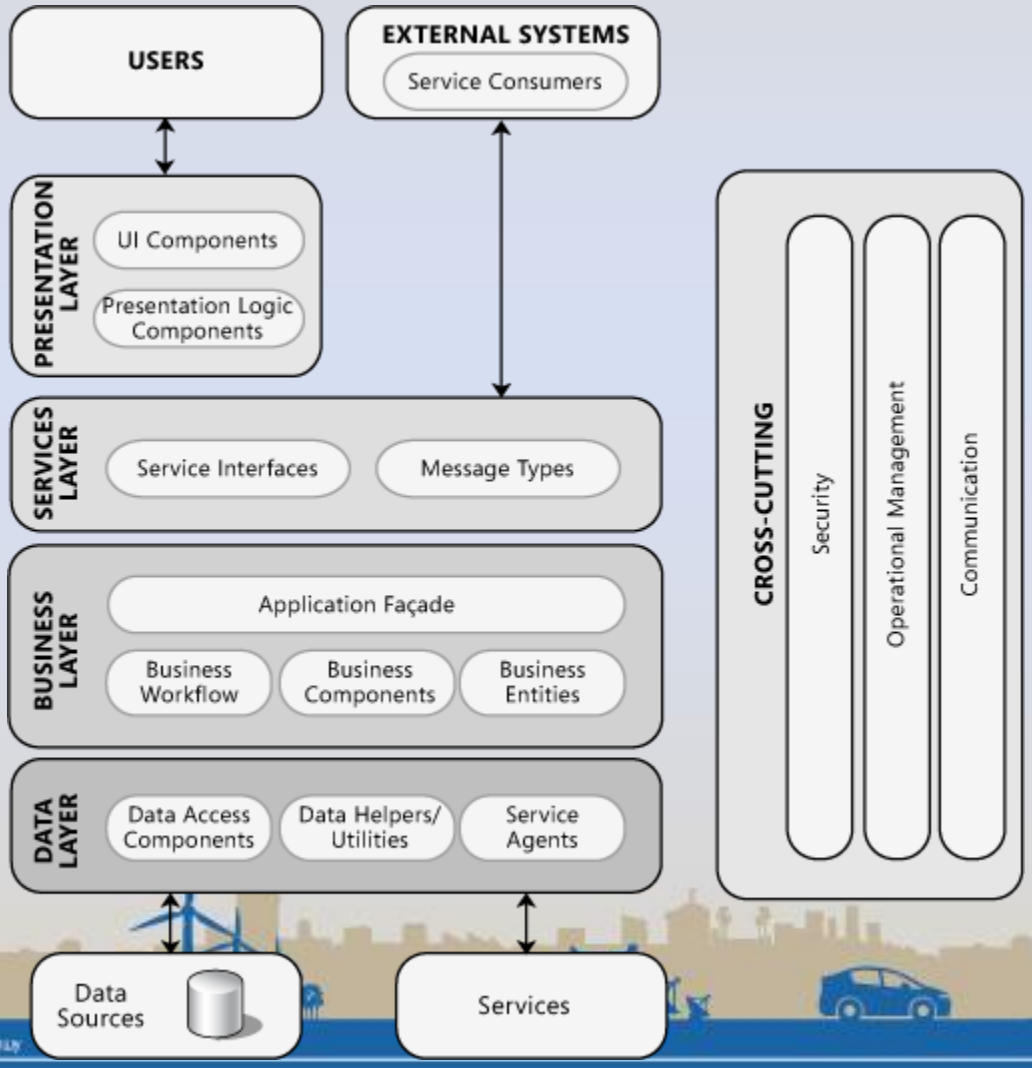
Zašto je arhitektura važna?

- Softver mora da bude izgrađen na čvrstim temeljima
- Savremeni alati i platforme pomažu da se pojednostavi zadatak razvoja aplikacija, međutim oni ne mogu zameniti potrebu da se pažljivo projektuje aplikacija na osnovu specifičnih scenarija i zahteva
- **Rizici** loše arhitekture su nestabilnost softvera, nemogućnost da podrži postojeće ili buduće poslovne zahteve ili ga je teško uvesti ili upravljati u proizvodnom okruženju



Architecture decisions are the most fundamental decisions, and changing them will have significant effects.

Primer opšte arhitekture softvera



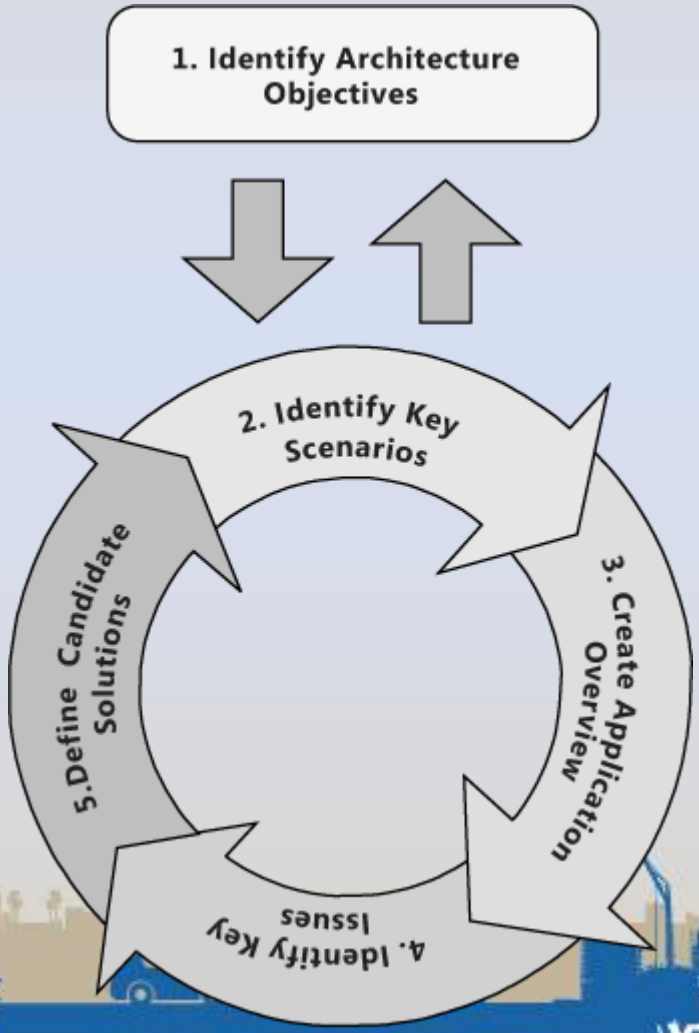
Obrasci arhitekture

- Obrasci (paterni) ili stilovi arhitekture softvera predstavljaju skup principa koji nude apstrktni okvir sistema koji olakšavaju podelu i ponovnu upotrebu pružajući pritom rešenja za učestale probleme
- Svaki od stilova rešava određene probleme i odnosi se na određene karakteristike sistema, performansi, procese, distribuciju …

Category	Architecture styles
Communication	Service-Oriented Architecture (SOA), Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Domain	Domain Driven Design
Structure	Component-Based, Object-Oriented, Layered Architecture

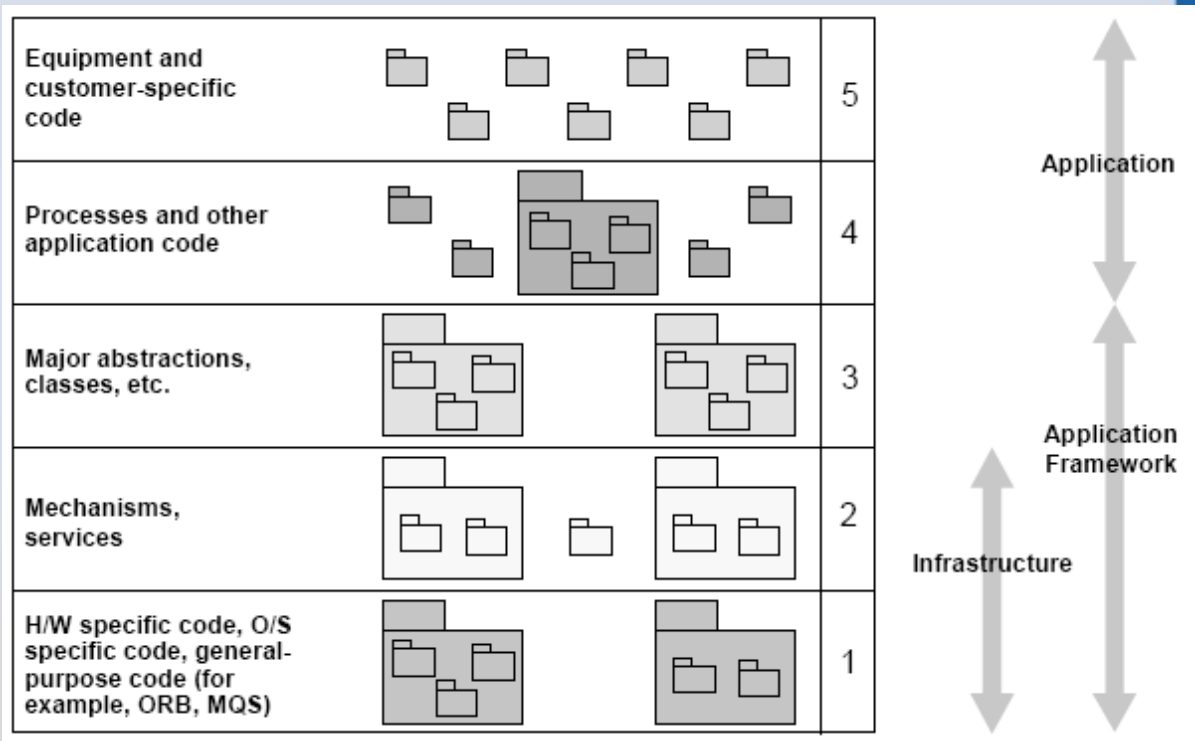
Koraci definisanja arhitekture softvera

- Ključne aktivnosti projektovanja arhitekture softvera su:
  - **Identifikovati cilj arhitekture** – jasni ciljevi pomažu u fokusiranju na arhitekturu i na rešavanje pravih problema
  - **Identifikovati ključna scenarija** – pomažu u fokusiranju dizajna na ono što je najvažnije
  - **Pregled aplikacije** – identifikovati tip aplikacije, arhitekturu uvođenja, stilove arhitekture i tehnologije kako bi se sagledali realni uslovi u kojima će aplikacija raditi
  - **Identifikovati ključna pitanja** – pitanja o kvalitetu softvera i drugim problemima
  - **Definisati kandidate rešenja** – napraviti prototip koji poboljšava i procenjuje rešenje u odnosu na ključna scenarija, probleme i ograničenja pre nego što se krene na narednu iteraciju projektovanja arhitekture

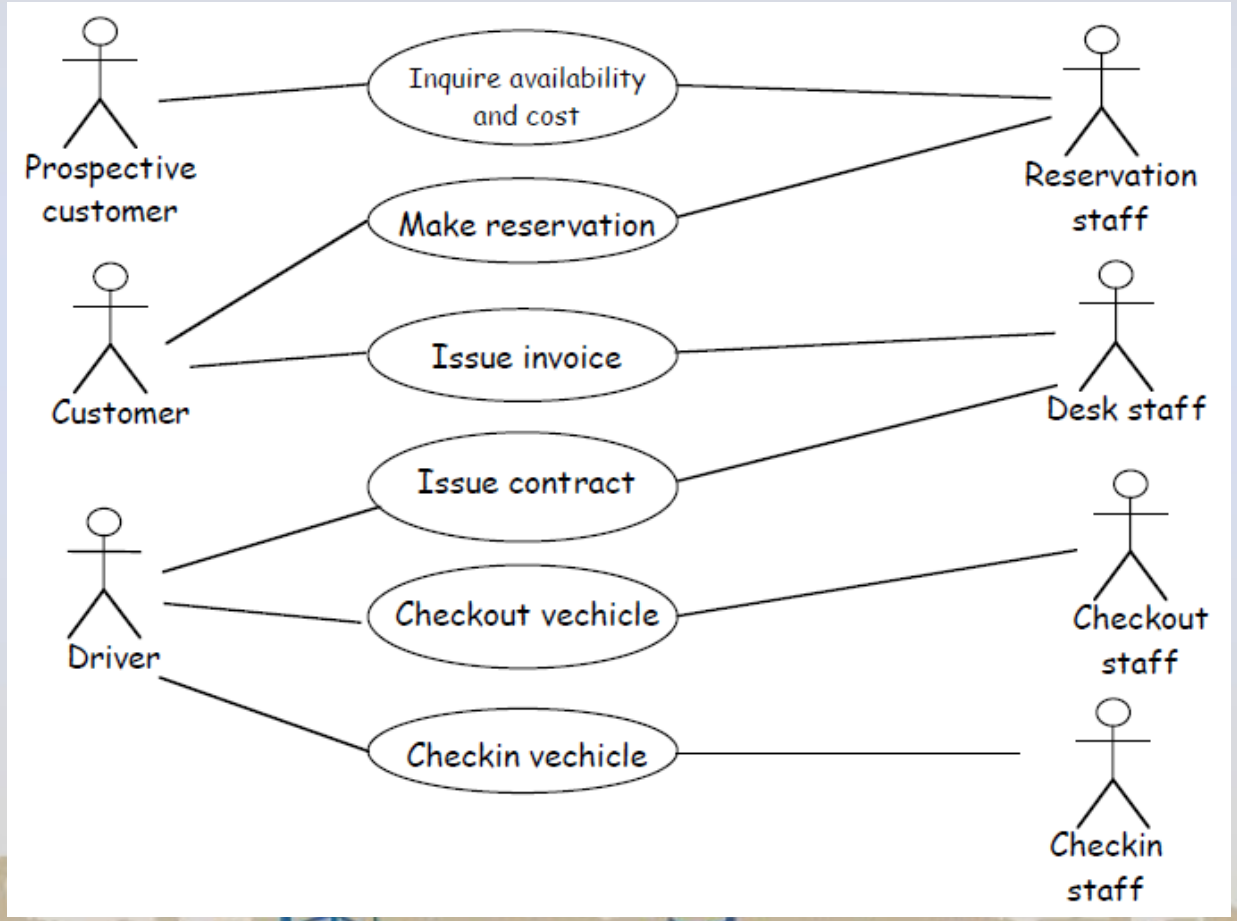


Slojevita arhitektura

- Veliki sistemi se moraju dekomponovati
- Sistem mora da vodi računa i o hardveru i o opštim servisima i domenu problema itd.
  - Nije poželjno pisati vertikalne komponente koje rade na svim nivoima!
  - Delovi sistema treba da budu zamenjivi
  - Promene u komponentama ne bi trebalo da se osete
  - Slične odgovornosti bi trebalo grupisati zajedno
  - Veličina komponenti – kompleksne komponente bi trebalo dekomponovati
  - Struktuirati sistem u grupe komponenata koji formiraju slojeve

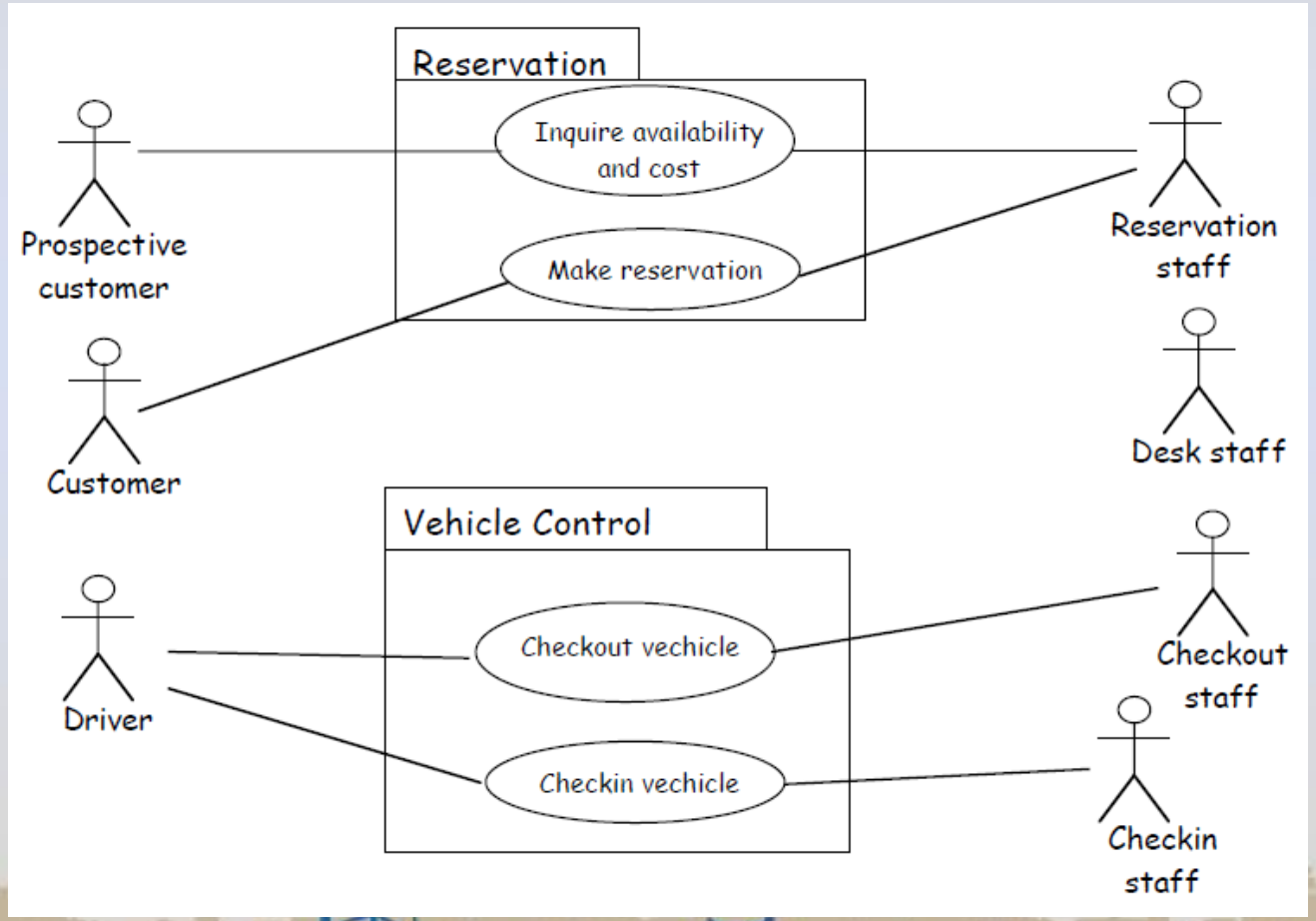


Primer grupisanja tj. pakovanja slučajeva korišćenja

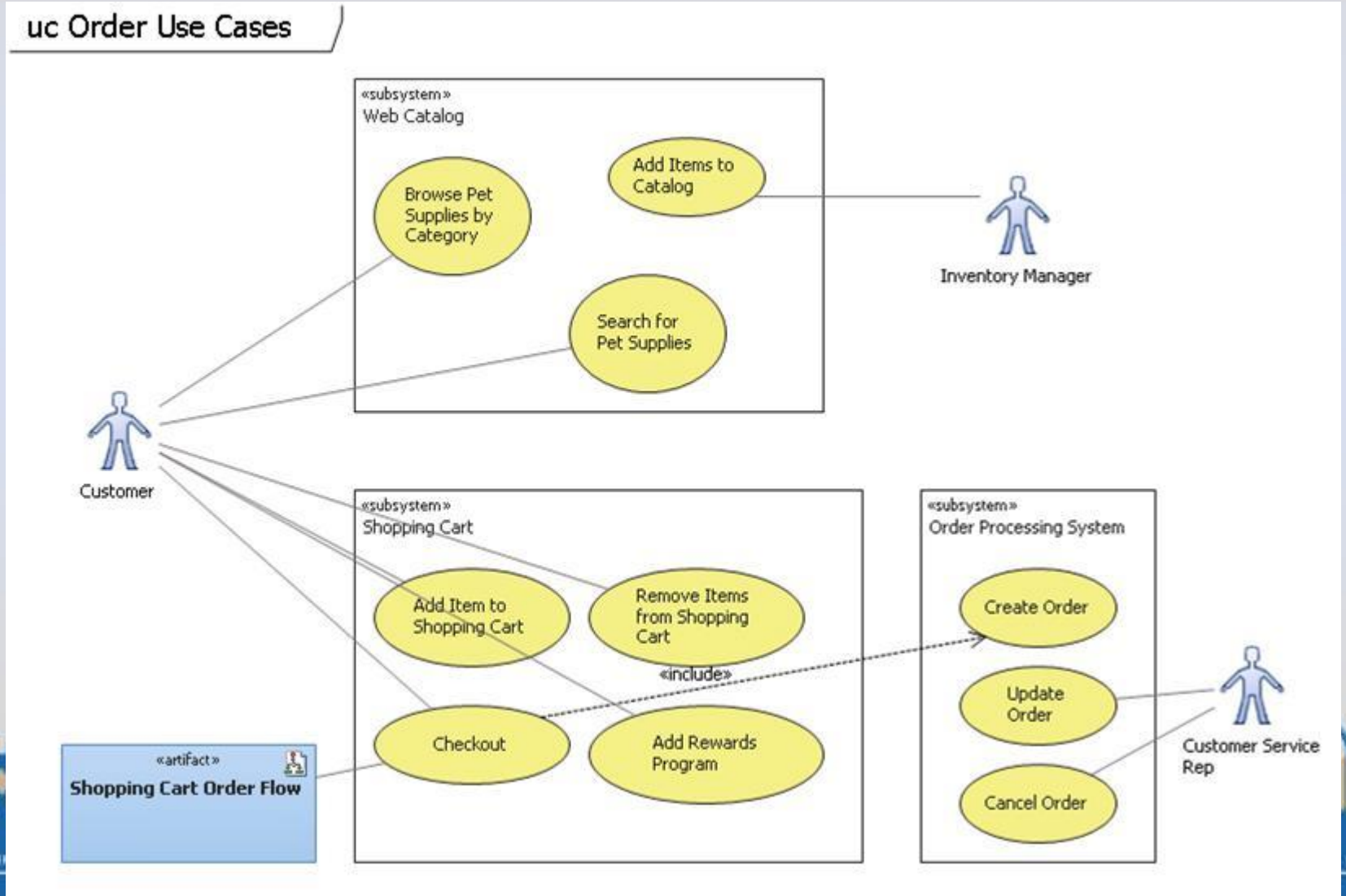


Use case model "Iznajmljivanje kola"

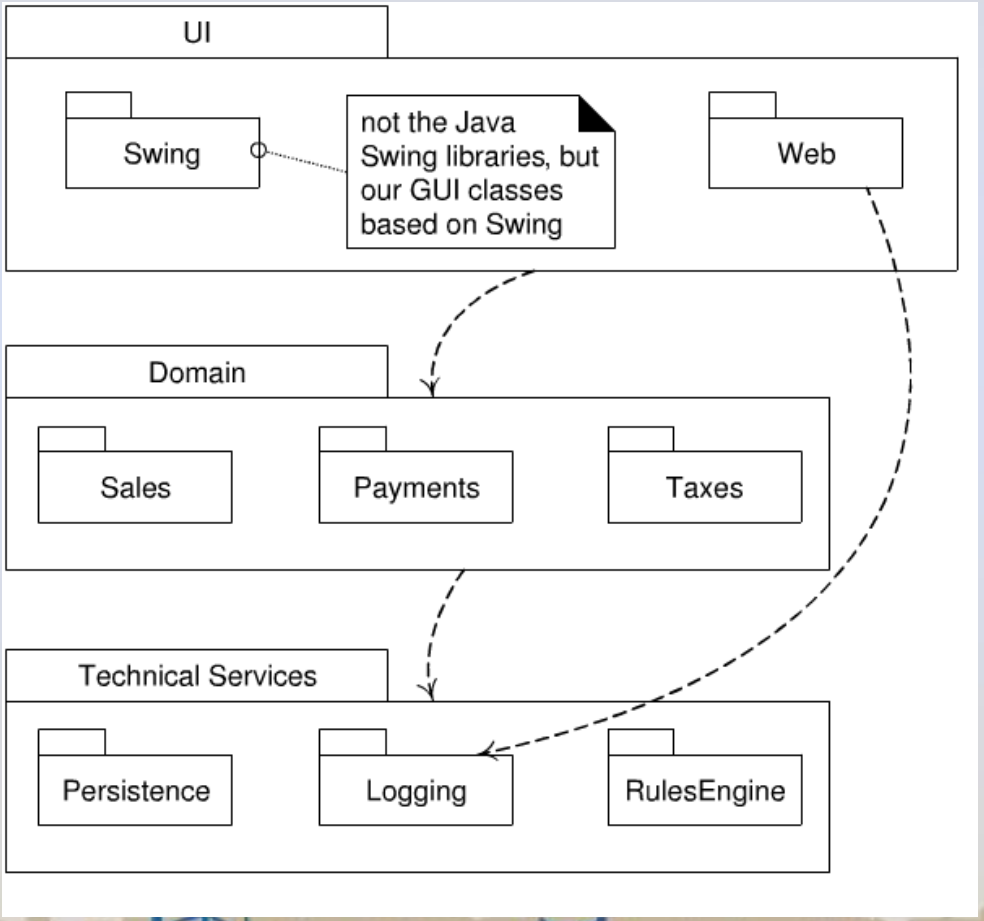
Primer grupisanja komponenata sistema “iznajmljivanja kola” u dva paketa



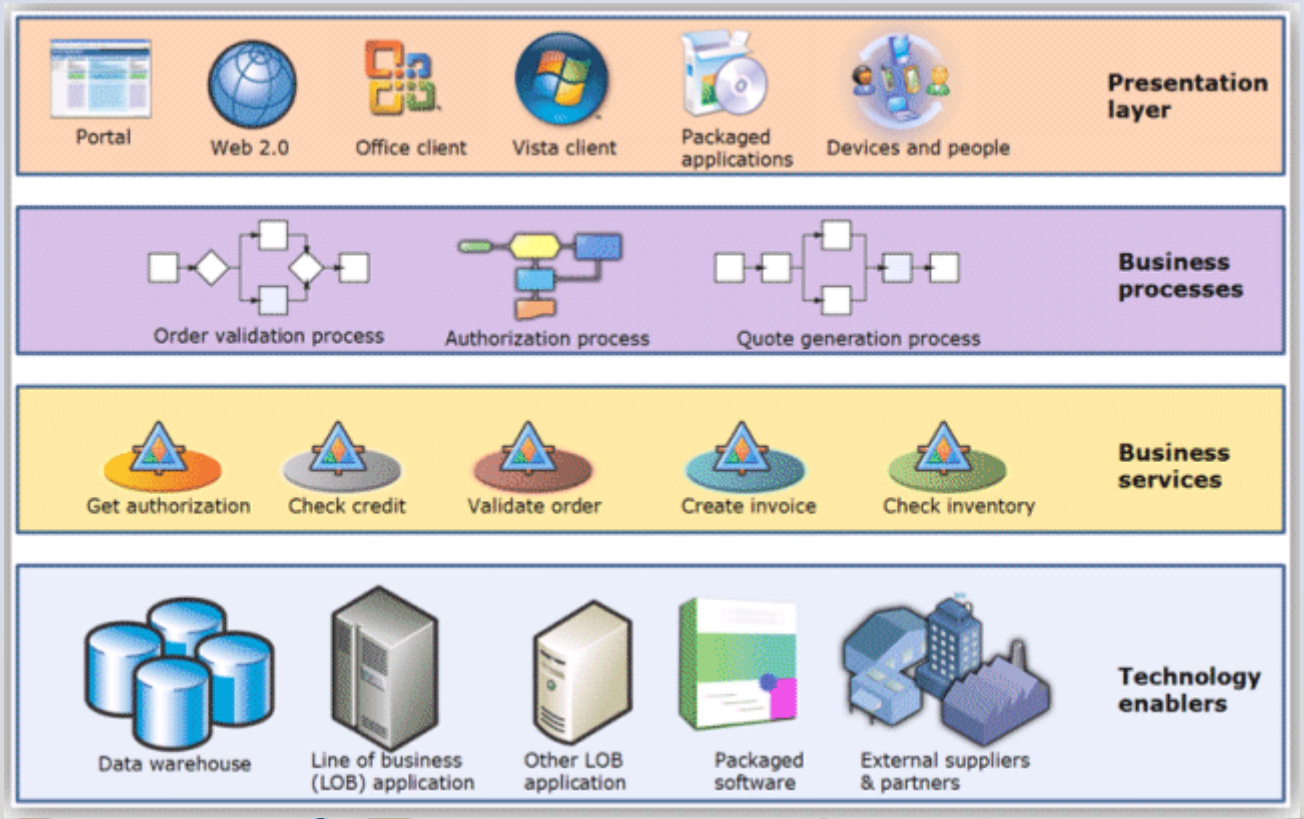
Primer grupisanja use case-ova



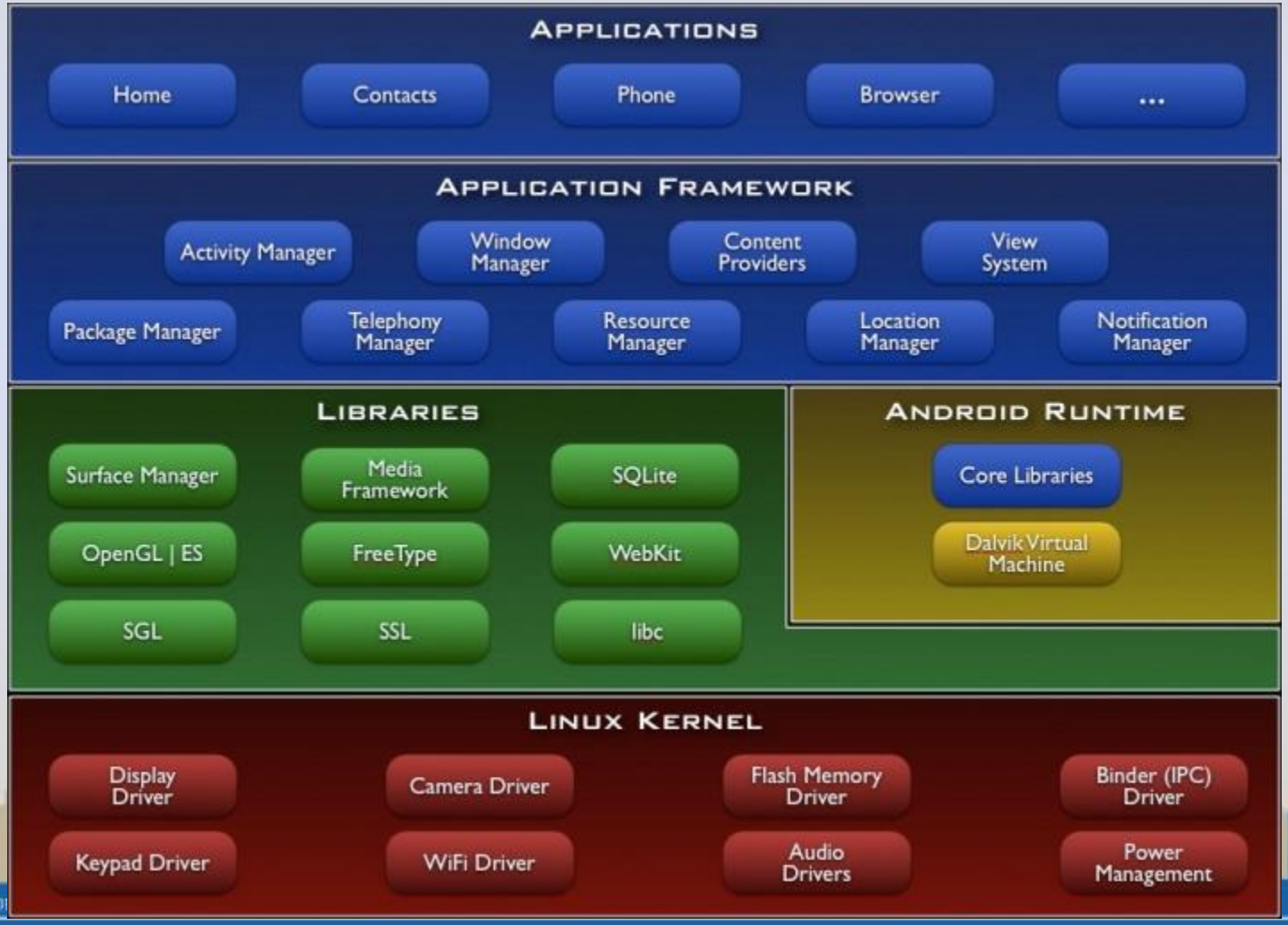
Primer raspoređivanja paketa prema slojevima arhitekture



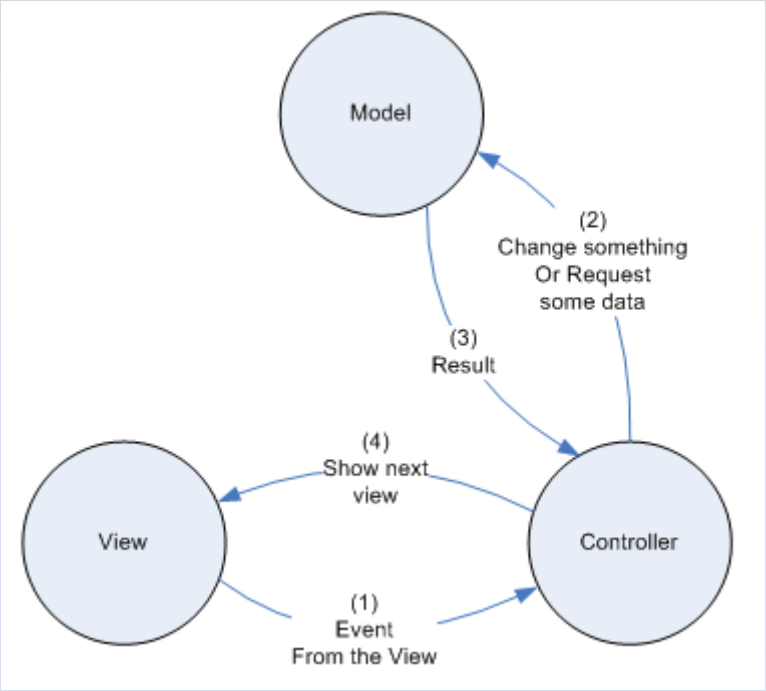
Slikoviti primer slojevite arhitekture



Primer Android arhitekture



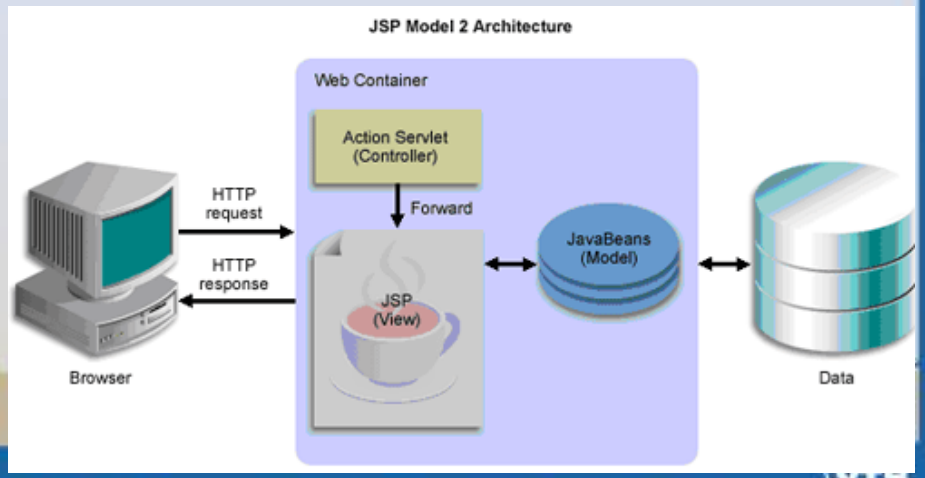
Model-View-Controller, MVC obrazac arhitekture



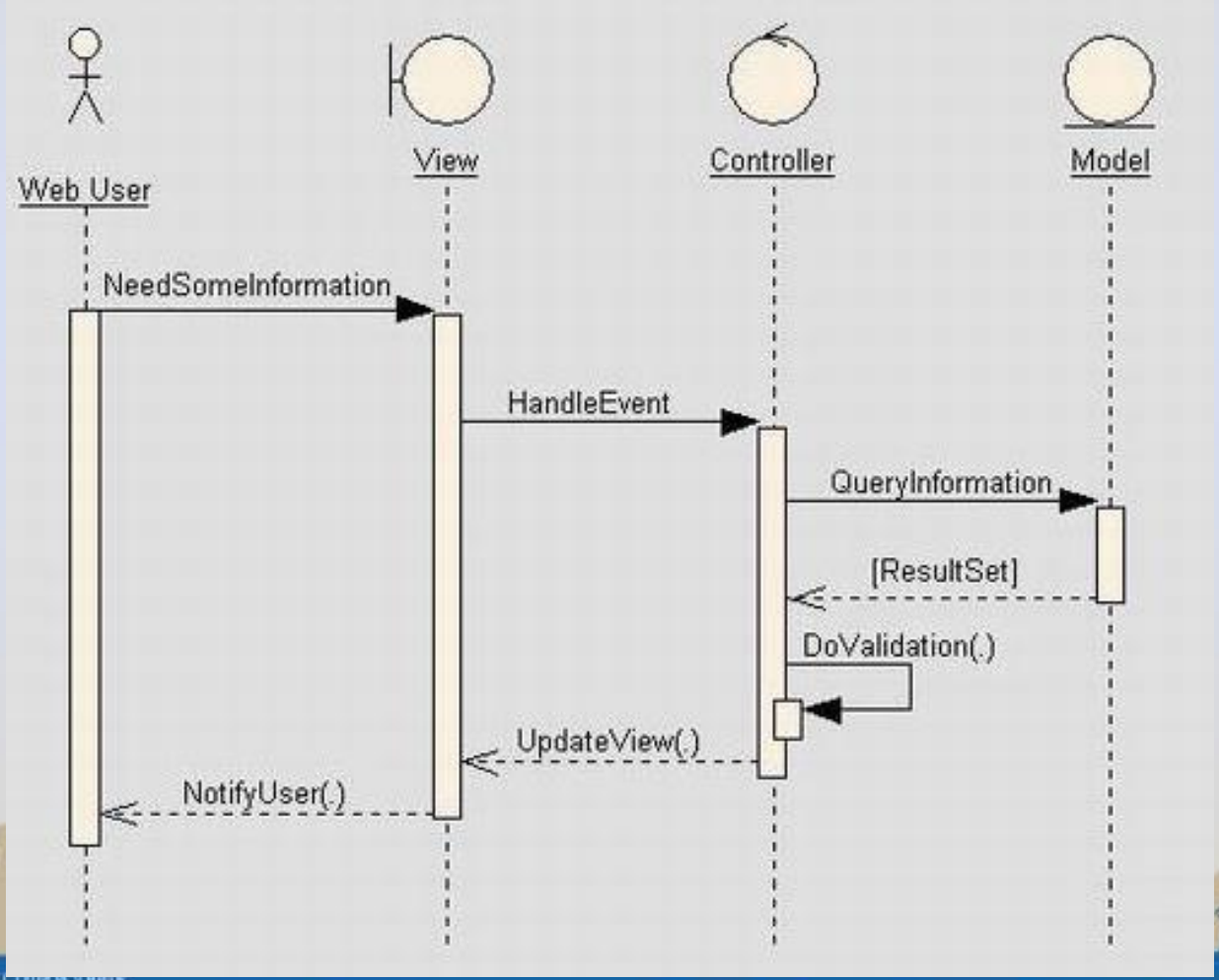
- Upotrebljivost softvera uglavnom zavisi od korisničkih interfejsa
- MVC se sastoji od tri ključne komponente:

1. **Model** – komponenta koja sadrži strukturu poslovnog sistema tj. podatke i njene operacije
2. **View** – obezbeđuje korisnički interfejs preko koga korisnik komunicira sa sistemom, a takođe šalje korisniku izveštaje dobijenih iz modela
3. **Controller** – upravlja izvršavanjem sistemskih operacija. Prihvata zahtev od klijenta i izvršava zadatke:
   - koordinira zahtevima klijenata
   - ažurira Model i View na osnovu korisničkog inputa
   - nadgleda i planira šta treba da se prikaže i izmeni
   - poziva izabrani Model i izvršava logiku
   - Npr., dok ažurira podatke zaposlenih, Controller odlučuje koju veb stranu treba prikazati

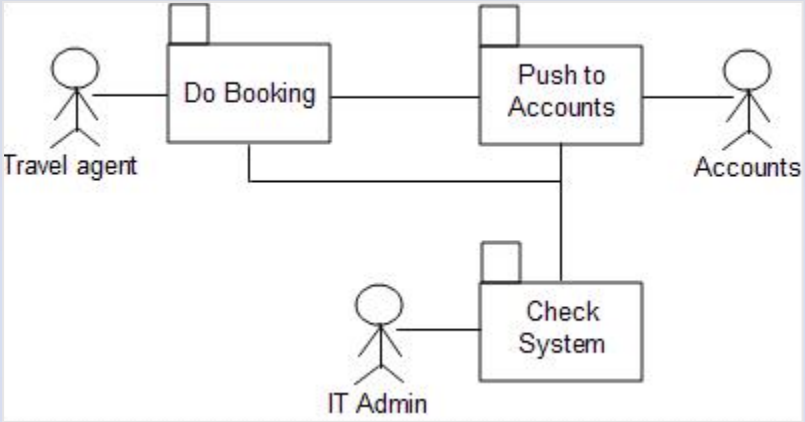
- U domenu razvoja Web aplikacija, JSP (*Java Server Pages)* specifikacija predviđa dva osnovna modela arhitekture Web aplikacije poznatih pod imenom Model 1 i Model 2 (slika dole).

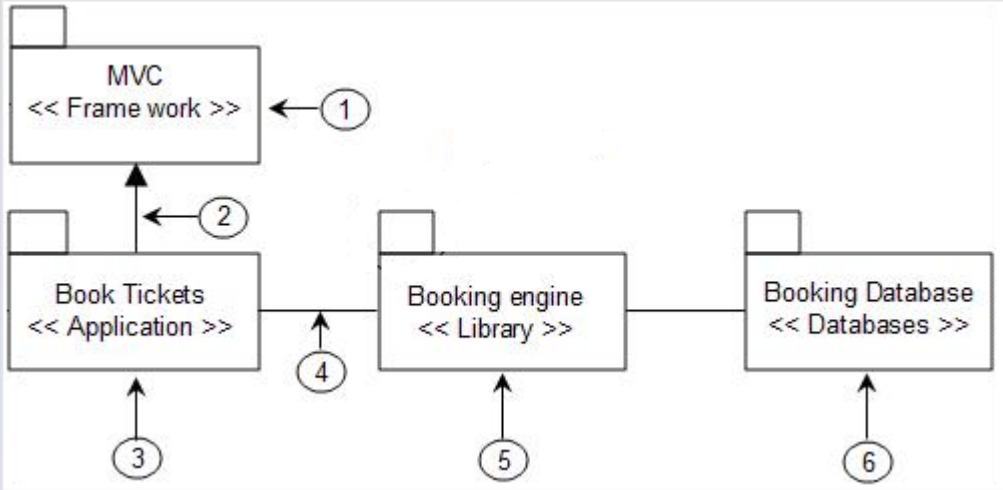


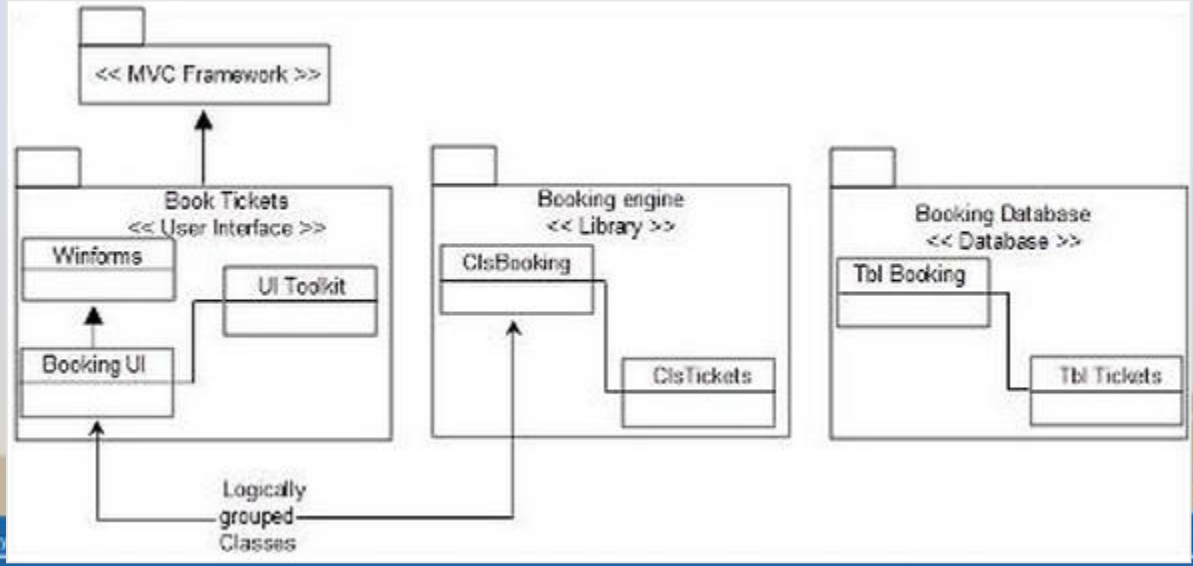
Opis MVC kroz dijagram sekvenci



Primer MVC

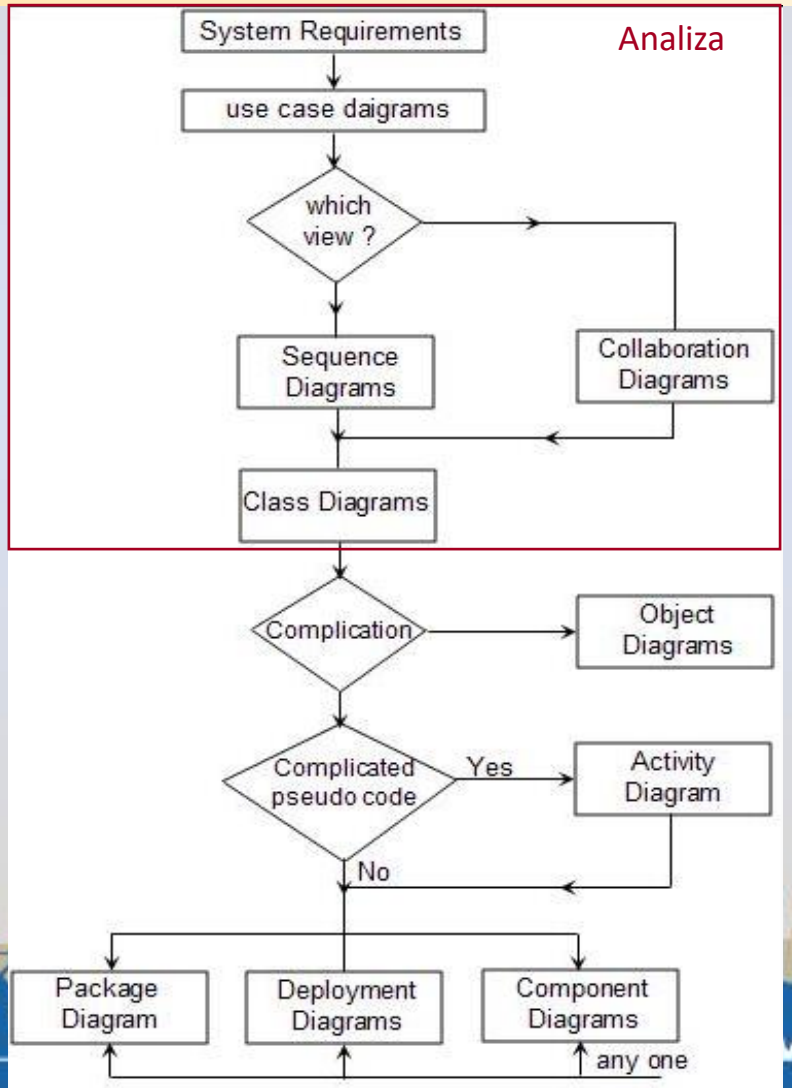

Travel agent
Do Booking
Push to Accounts
Accounts
Check System
IT Admin


MVC
<< Frame work >>
1
2
Book Tickets
<< Application >>
Booking engine
<< Library >>
Booking Database
<< Databases >>
3
4
5
6


<< MVC Framework >>
Book Tickets
<< User Interface >>
Winforms
UI Toolkit
Booking UI
Booking engine
<< Library >>
ClsBooking
ClsTickets
Booking Database
<< Database >>
Tbl Booking
Tbl Tickets
Logically
grouped
Classes

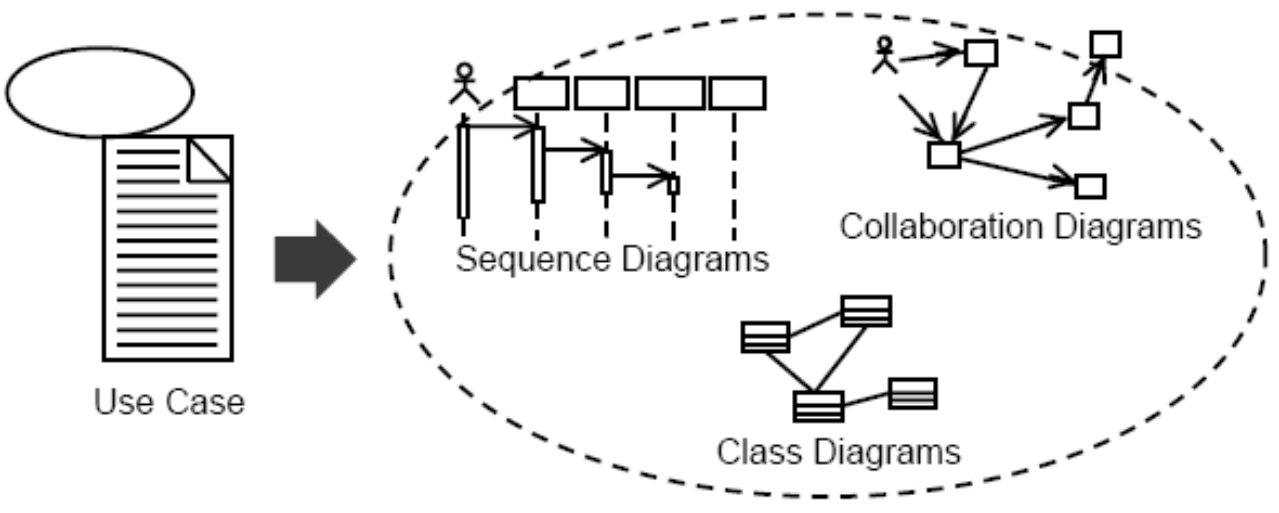
Analiza slučajeva korišćenja

- Za svaki tok događaja *use case*-a treba:
  - **Identifikovati klase** koje izvršavaju tokove događaja slučajeva korišćenja (upotrebe)
  - **Raspodeliti ponašanja use case-a na klase** - koristi se use case realizacija
  - **Modelovati interakcije između klasa** na dijagramima interakcije
  - Uočiti upotrebu **mehanizama arhitekture**



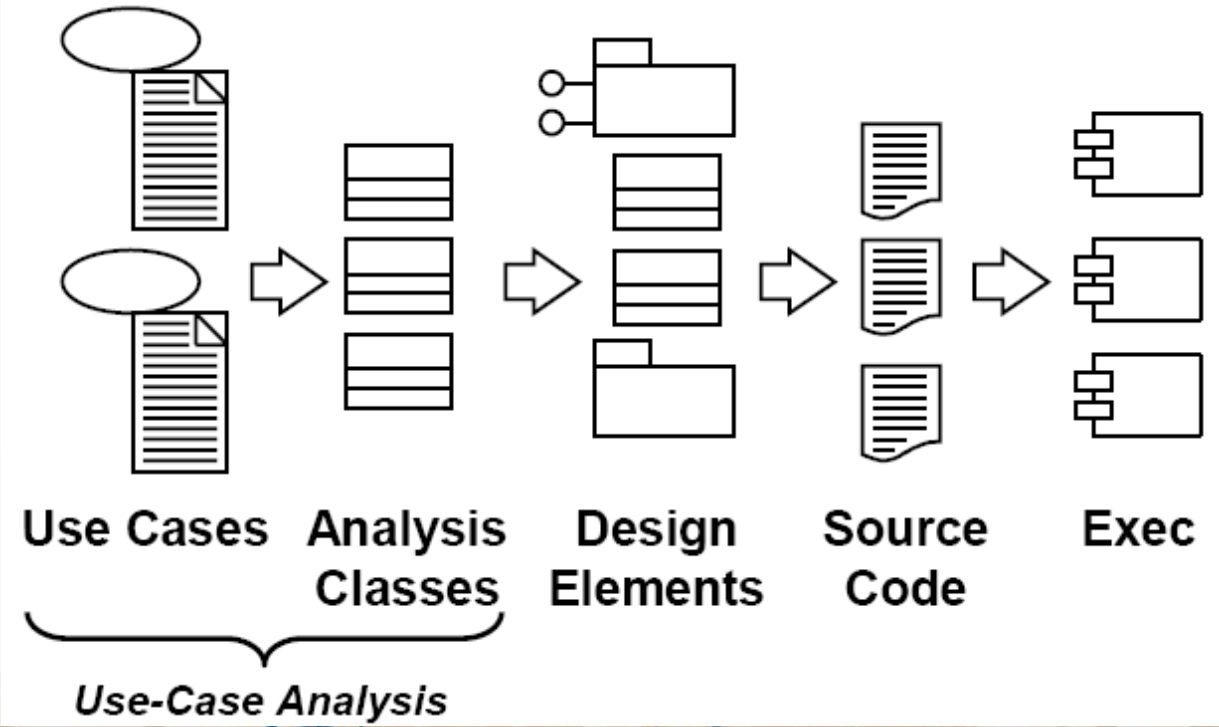
Use case realizacija

- *Use case* realizacija se može predstaviti skupom dijagrama koji:
  - **Modeluju sadržaj kolaboracije** - klase/objekti koji implementiraju *use case* i njihove relacije → **dijagrami klasa**
  - **Modeluju interakcije kolaboracije** - u kakvoj su interakciji ove klase/objekti da bi izvršile *use case* → **dijagrami komunikacije i sekvenci**



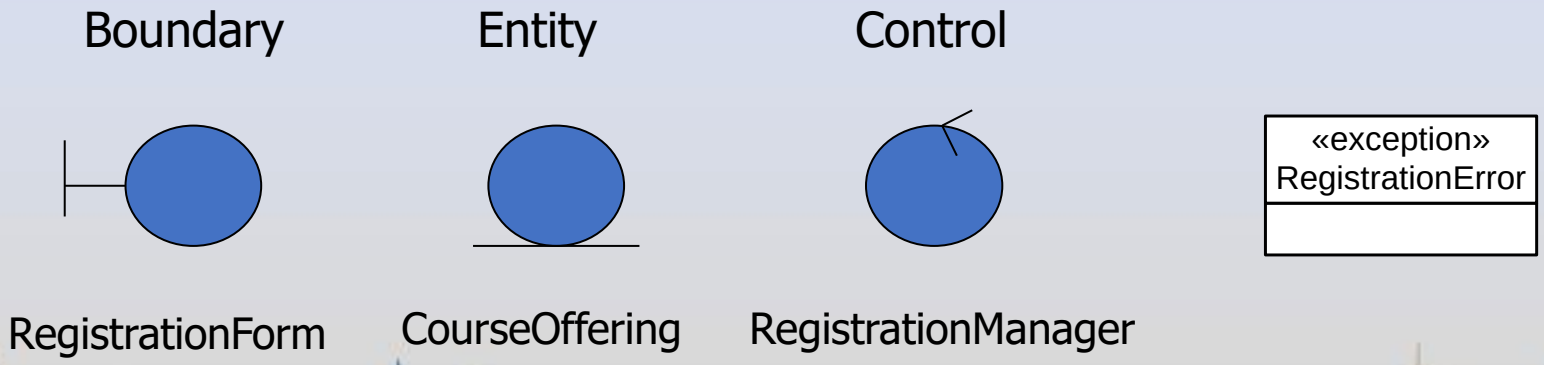
Analiza klasa

- Kompletno ponašanje *use case*-a mora da se raspodeli na klase



Analiza klasa

- Tehnika za pronalaženje klasa koristi tri različite perspektive sistema:
  - Granica između sistema i aktera (*Boundary*)
  - Informacije koje sistem koristi (*Entity*)
  - Logika kontrole sistema (*Control*)



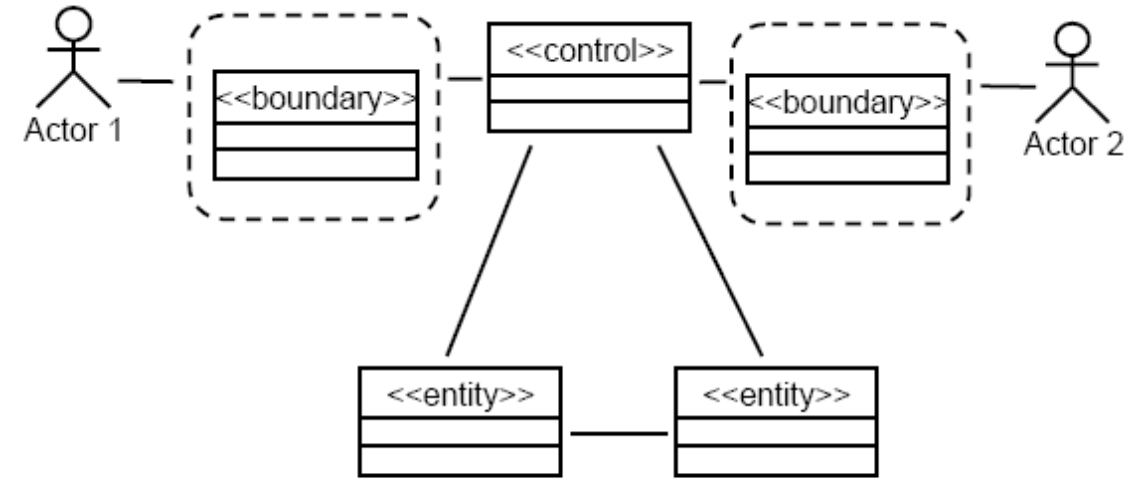
Šta su granične klase (*boundary class*)?

- Sistem može da ima nekoliko tipova graničnih klasa:
  - **Klase korisničkog interfejsa** (*user interface classes*) – klase koje posreduju u komunikaciji sa ljudima, korisnicima sistema
  - **Klase interfejsa sistema** (*system interface classes*) – klase koje posreduju u komunikaciji sa drugim sistemima
  - **Klase interfejsa uređaja** (*device interface classes*) – klase koje obezbeđuju interfejs ka uređajima koji detektuju spoljne događaje

- Za inicijalnu identifikaciju graničnih klasa preporučuje se jedna granična klasa po paru akter/*use-case*
- Granične klase se koriste za modelovanje sistemskih interfejsa

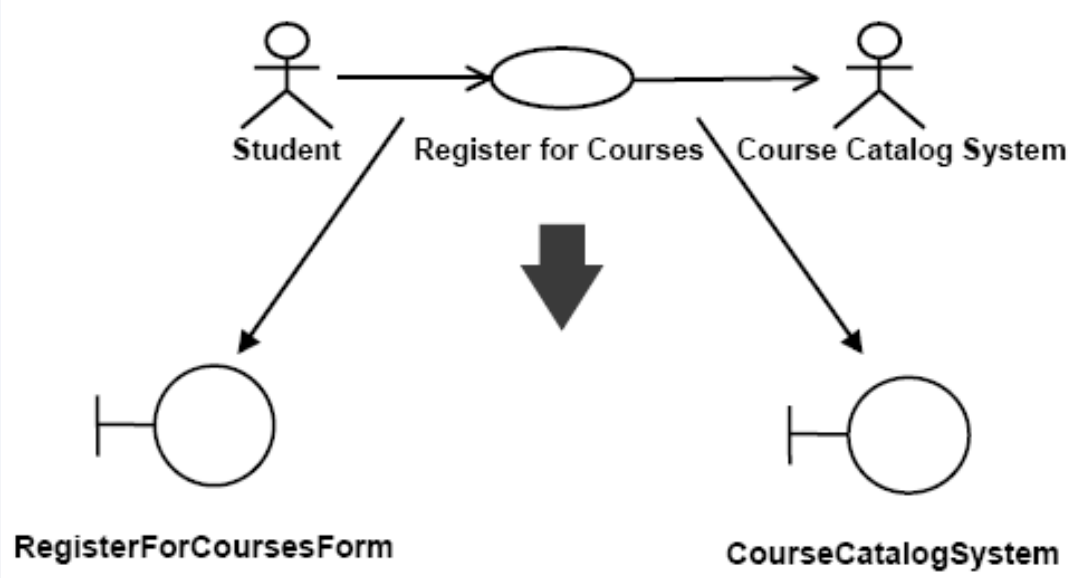
Uloga granične klase

- Granična klasa se koristi za modelovanje interakcija između okruženja sistema i njegovog unutrašnjeg rada



Model interaction between the system and its environment

Primer graničnih klasa

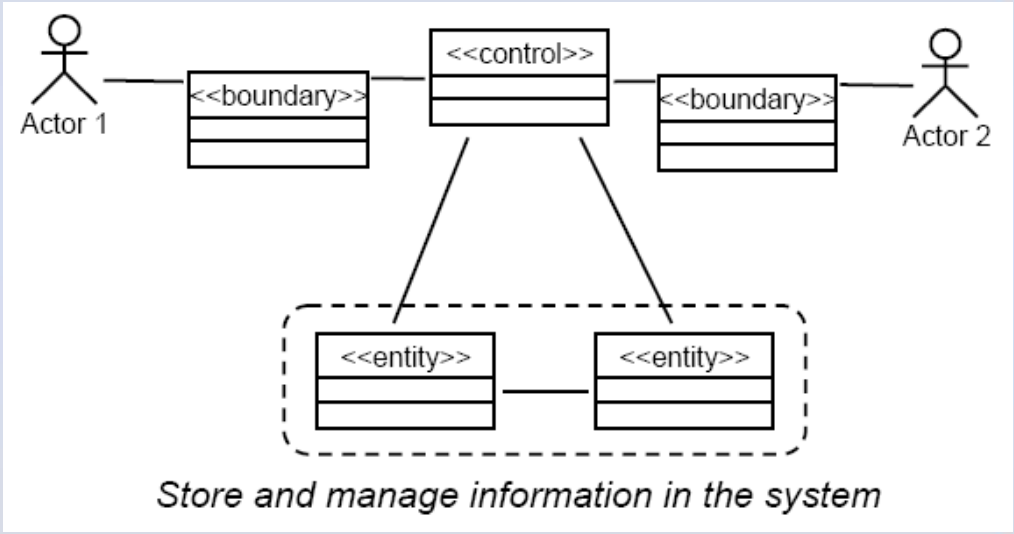


- *RegisterForCoursesForm* - prikazuje listu kurseva za tekući semestar od kojih student može da bira kurseve koje želi da doda na svoj raspored

- *CourseCatalogSystem* je interfejs između legacy sistema koji obezbeđuje katalog svih kurseva na univerzitetu

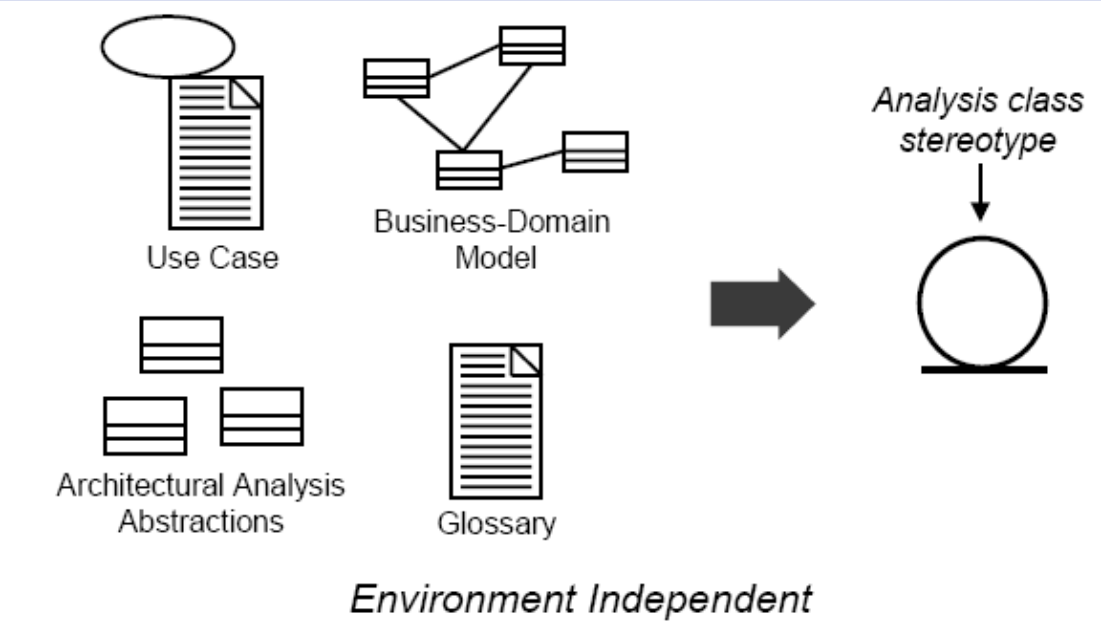
Šta su klase entiteta?

- Klase entiteta predstavljaju skladišta podataka u sistemu
  - Obično su perzistentni, sadrže atribute i ponašanja u toku životnog veka sistema
- Tipični primeri klasa entiteta:
  - u sistemu bankarskog poslovanja su Račun i Klijent
  - kod sistema umrežavanja su čvor i link
- Glavne odgovornosti klasa entiteta su skladištenje i upravljanje podacima u sistemu

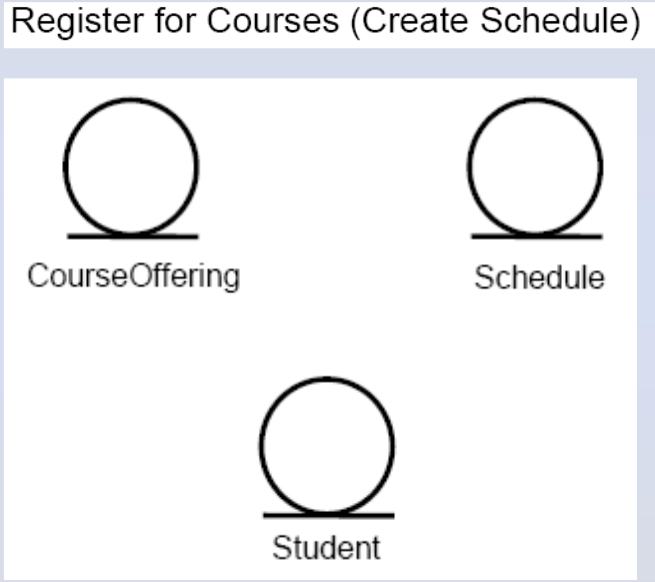


Klasa entiteta

- Izvori za kreiranje klasa entiteta su:
  - Rečnik (razvijen tokom zahteva)
  - Model domena poslovanja (razvijen tokom modelovanja poslovanja)
  - Use case tok događaja (razvijen tokom zahteva)
  - Ključne apstrakcije (identifikovane u analizi arhitekture)



Primeri kandidata klase entiteta

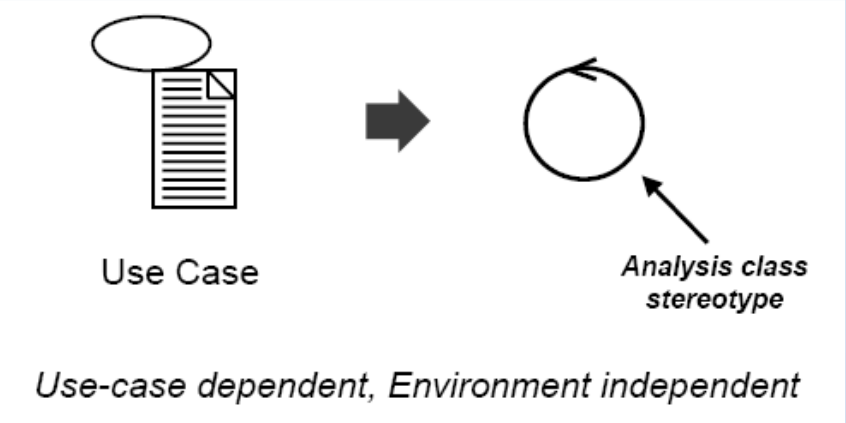


- Sistem registrovanja na kurseve održava informacije o studentima nezavisno od toga što je student i akter u sistemu

Šta je kontrolna klasa?

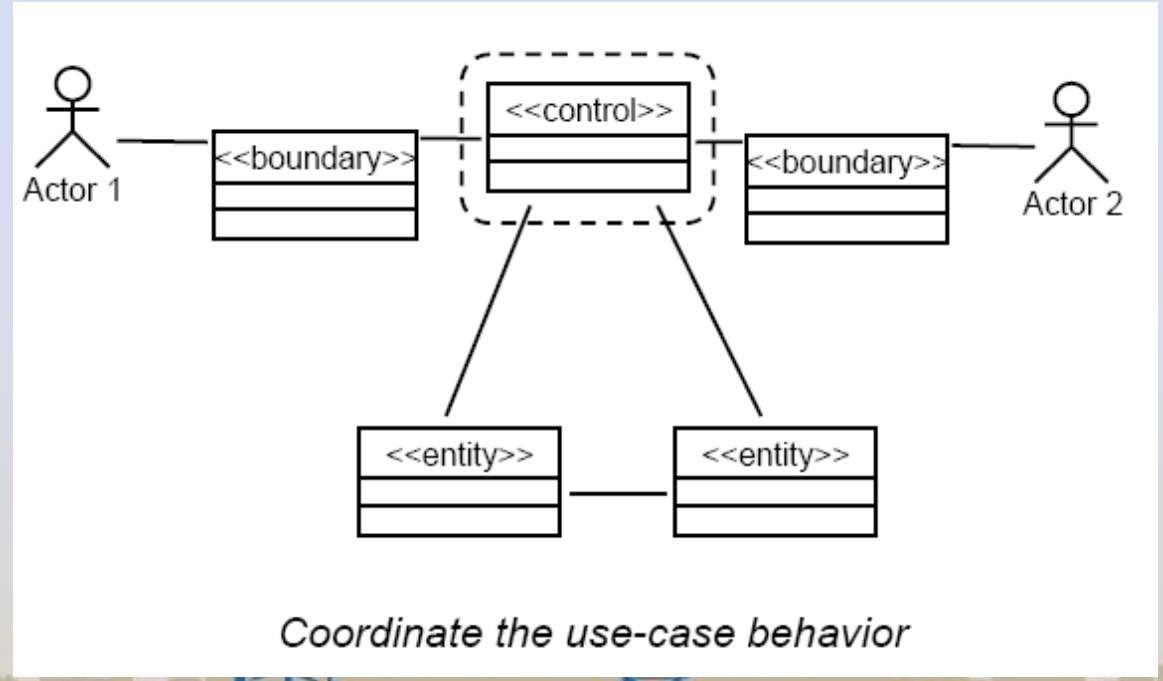
- Kontrolna klasa (*Control class*) je klasa koja se koristi za modelovanje ponašanja koje je specifično za jedan ili više slučajeva korišćenja
  - Kontrolna klasa enkapsulira ponašanje određenog use case-a
  - Ne zahtevaju svi use case-ovi kontrolne klase

Preporuka kod inicijalne identifikacije kontrolnih klasa da bude jedna kontrolna klasa po use case-u



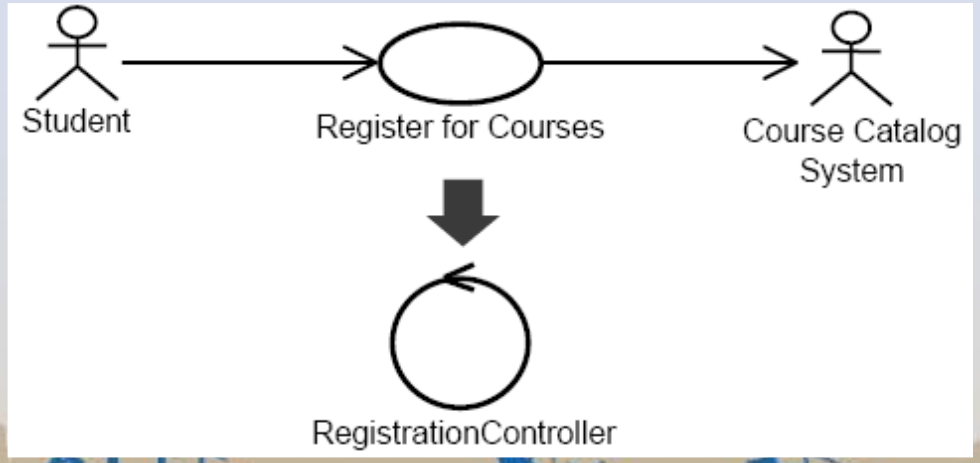
Uloga kontrolnih klasa

- Kontrolne klase obezbeđuju ponašanje koje definiše logiku kontrole (redosled između događaja) i transakcija slučajeva korišćenja

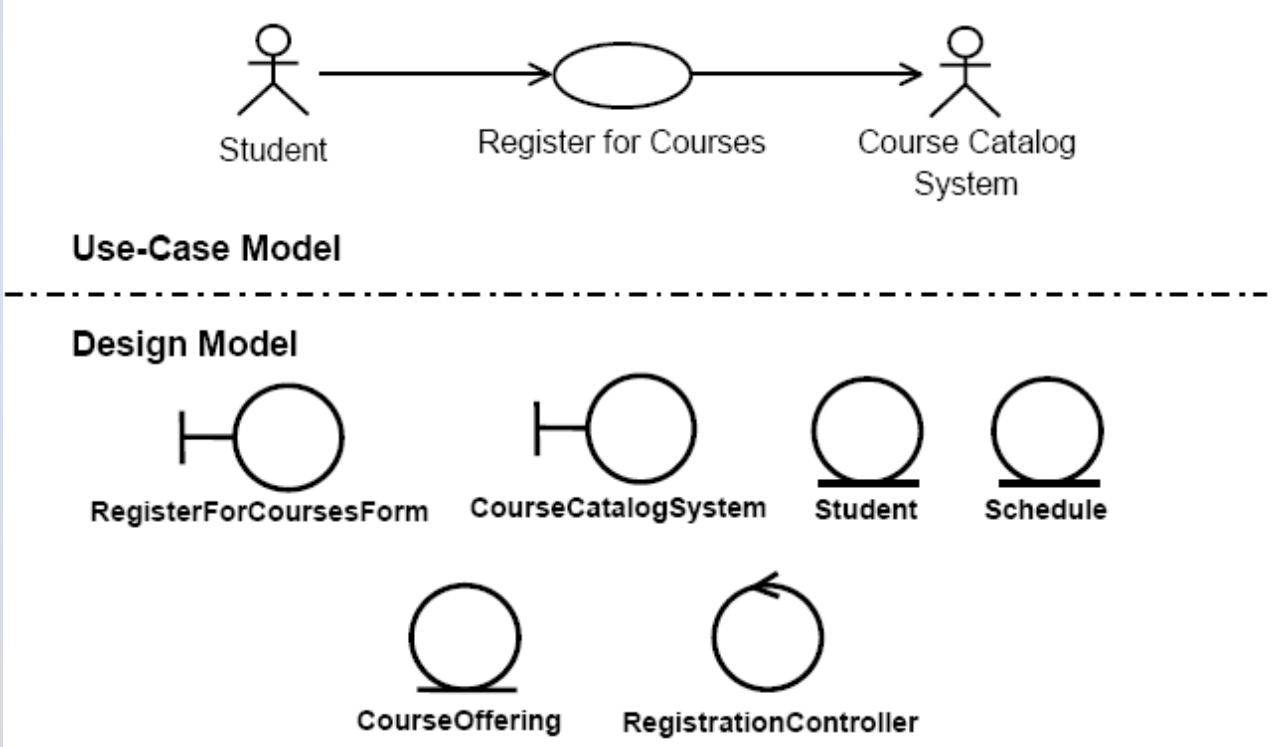


Primer pronalaženja kontrolnih klasa

- Identifikovati jednu kontrolnu klasu po *use case*-u
  - Složeniji *use case*-ovi mogu da zahtevaju i više kontrolnih klasa
  - Svaka kontrolna klasa je odgovorna za kontrolisanje procesa koji implementira funkcionalnost *use case*-a
- Na primeru, kontrolna klasa <<control>> *RegistrationController* je definisana da vodi proces *Register for courses* unutar sistema

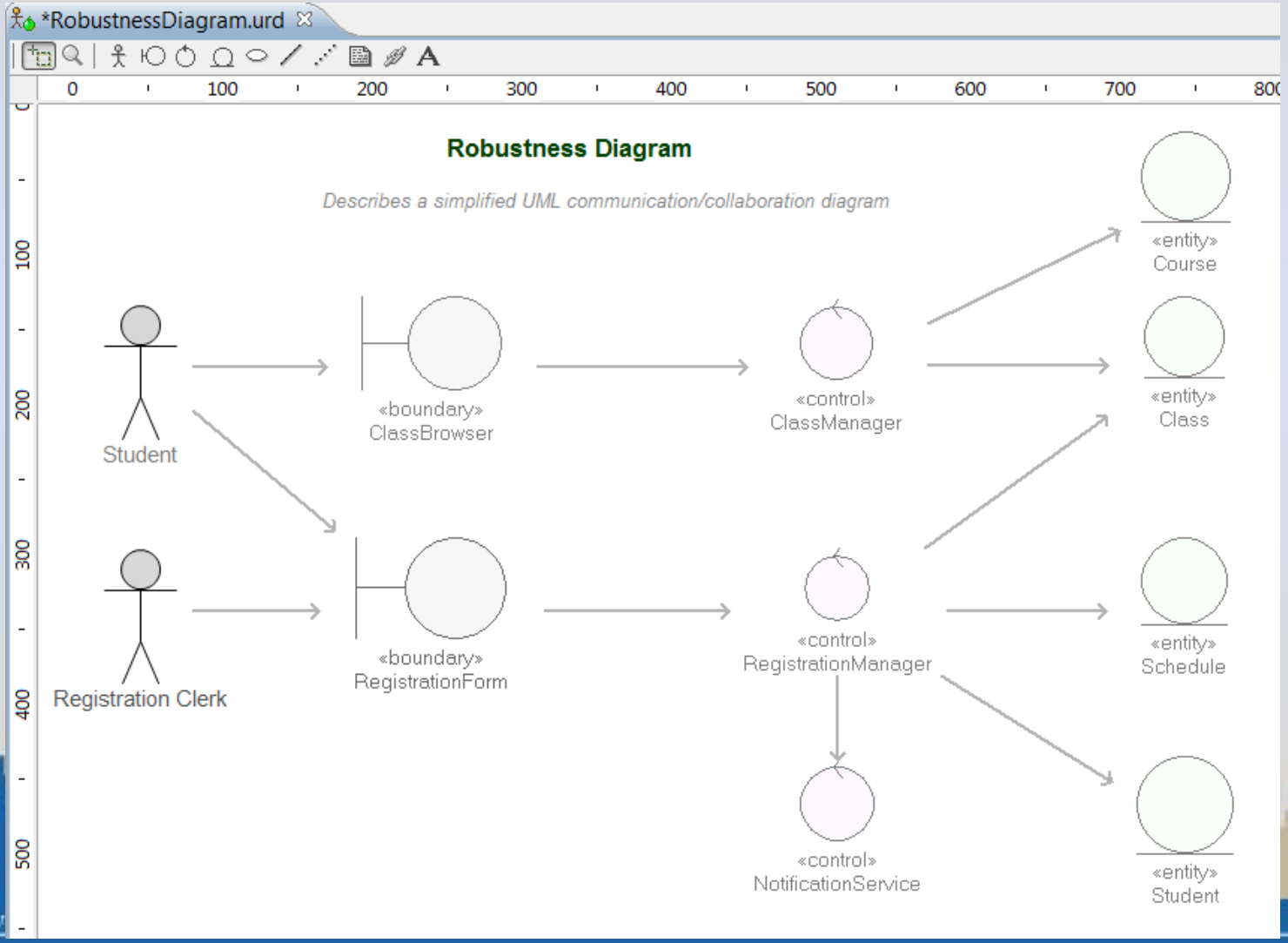


Primer analize klasa

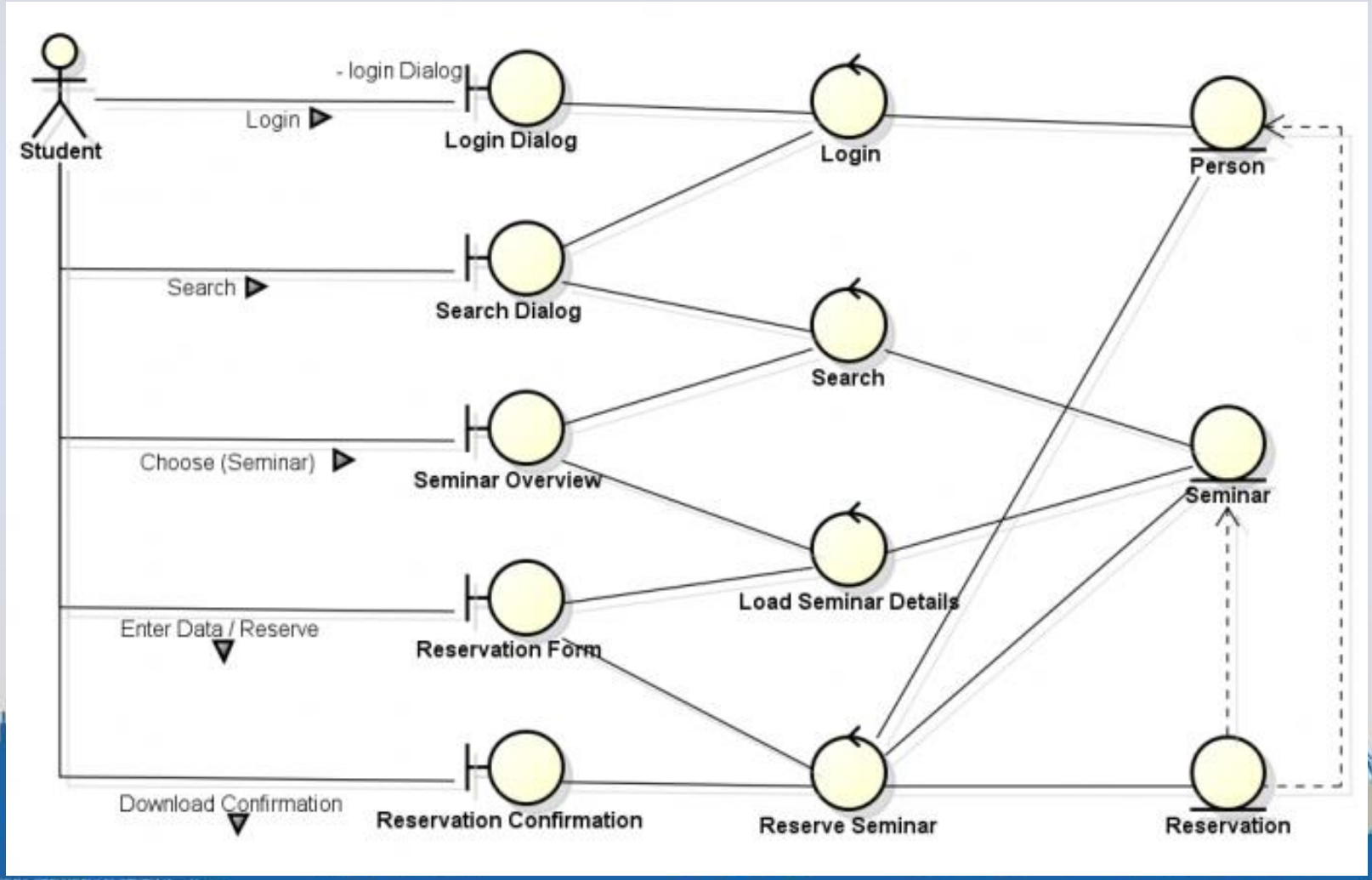


- Za svaku realizaciju *use case*-a, postoji jedan ili više dijagrama klasa
- Dijagrami klasa osiguravaju da postoji konzistentnost u use case implementaciji

Primer

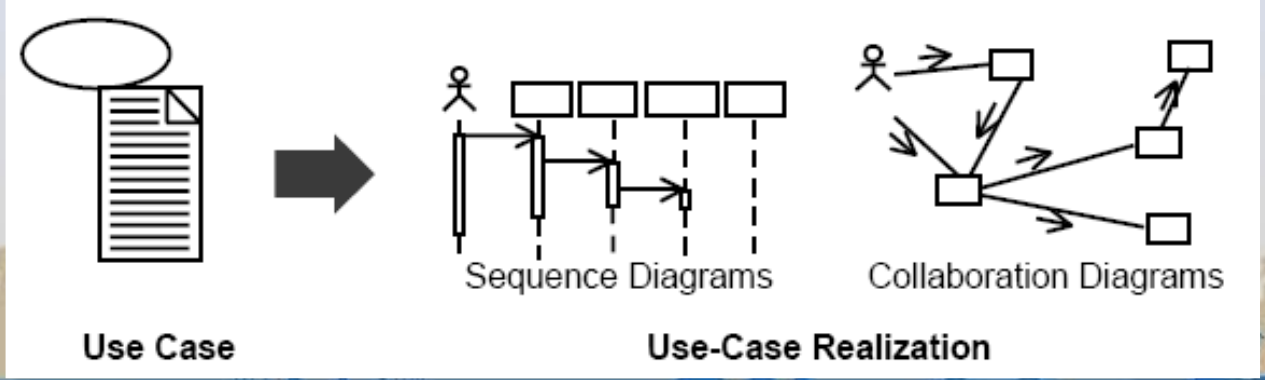

*RobustnessDiagram.urd
Robustness Diagram
Describes a simplified UML communication/collaboration diagram
Student
Registration Clerk
«boundary»
ClassBrowser
«boundary»
RegistrationForm
«control»
ClassManager
«control»
RegistrationManager
«control»
NotificationService
«entity»
Course
«entity»
Class
«entity»
Schedule
«entity»
Student

Primer


Student
- login Dialog
Login
Login Dialog
Login
Person
Search
Search Dialog
Search
Choose (Seminar)
Seminar Overview
Seminar
Load Seminar Details
Enter Data / Reserve
Reservation Form
Download Confirmation
Reservation Confirmation
Reserve Seminar
Reservation

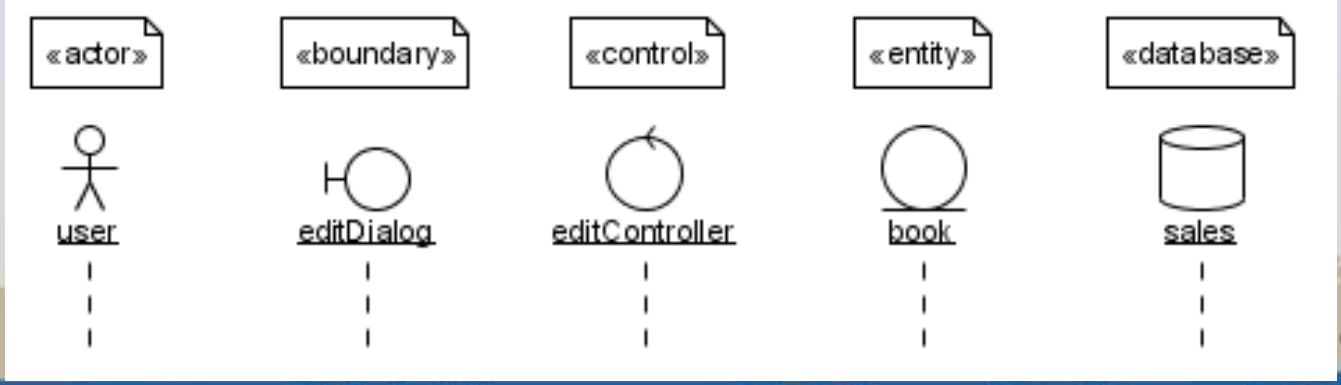
Modelovanje interakcija između klasa

- Do sada smo identifikovali kandidate klasa, a sada treba da **raspodelimo use case ponašanja na klase!**
- Možete kreirati dijagrame interakcije za svaki scenario use case-a
  - Dijagrami interakcije pokazuju interakcije sistema sa akterima
  - **Dijagrami interakcije** su dijagram **sekvenci** i dijagram **komunikacije**
  - Interakcije treba da počnu sa akterom, jer akter uvek poziva use case!
  - Ukoliko imate nekoliko instanci aktera na istom dijagramu, trudite se da ih stavite duž ivica dijagrama



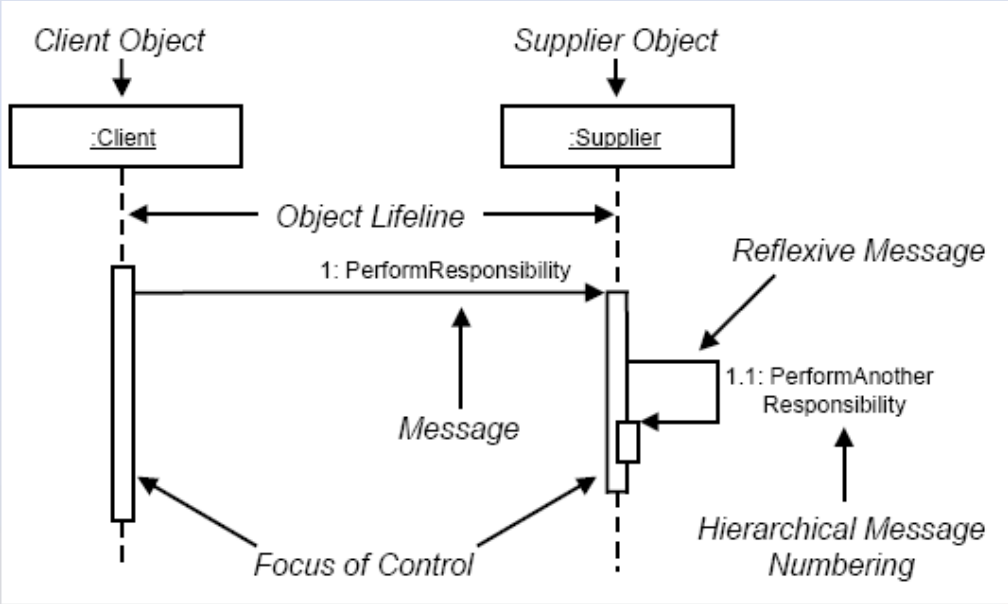
Dijagram sekvenci

- Dijagram sekvenci (*Sequence diagram*) opisuje **interakcije između objekata uređenih hronološkim redom**
  - Pokazuje objekte koji učestvuju u interakciji i poruke koje šalju
  - vizuelizuje **jedan scenario** ili određeno ponašanje poslovnih procesa u vremenu
- Objekat (*Object*) je prikazan kao vertikalna tačkasta linija koja se zove "*lifeline*"
  - ***Lifeline* ili životna linija prikazuje postojanost objekata u određenom vremenu**
  - U simbolu objekta se navodi naziv objekta i njegove klase odvojenih dvotačkom i podvučenih



Dijagram sekvenci

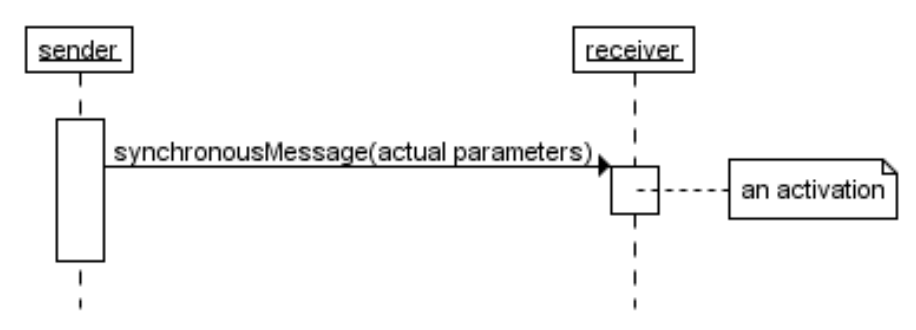
- Poruka (*Message*) je komunikacija između objekata koji prenosi informaciju
  - Poruka se prikazuje kao horizontalna linija od životne linije jednog objekta do životne linije drugog objekta
  - Strelica je označena nazivom poruke i njenim parametrima, a može biti označena i sa rednim brojem
- Fokus kontrole (*Focus of control*) se obeležava kao uzak pravougaonik i predstavlja **vreme kada je kontrola fokusirana na objekat**
- Hijerarhijsko numerisanje (*Hierarchical numbering*) - npr., poruka 1.1 zavisi od poruke 1.
- Tekstovi (*scripts*) – tekstualno opisuju tok događaja

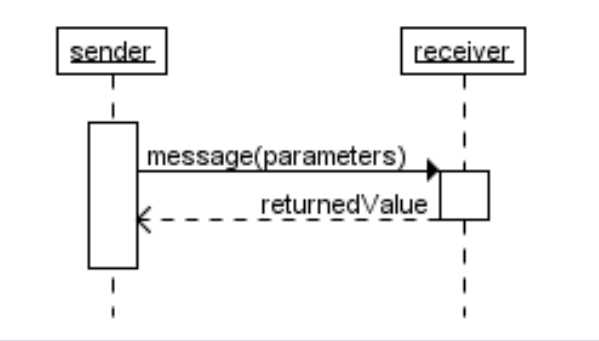


Poruke (*Messages*)

Sinhrone poruke (*Synchronous message*)

- Sinhrone poruke se koriste onda kada **pošiljalac čeka dok primaoc ne završi obradu poruke**, tek nakon toga pozivaoc može da nastavi
- Pravougaonik na životnoj liniji klase primaoca označava da se on odazvao na poruku
- Većina metoda poziva u objektno-orijentisanim programskim jezicima su sinhrone
- Zatvoren i **popunjen vrh strelice** označava da je poruka sinhronizovana



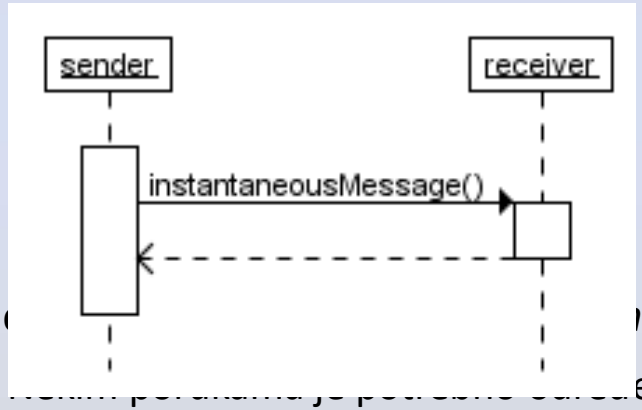


- Ukoliko hoćete da prikažete da je primaoc završio obradu poruke i vratio kontrolu pošiljaocu, nacrtajte isprekidanu liniju od primaoca do pošiljaoca
- Da bi dijagram bio lak za čitanje, poželjno je prikazati povratnu strelicu jedino ukoliko vraća vrednost pošiljaocu, u drugim slučajevima je ne bi trebalo prikazivati

Poruke

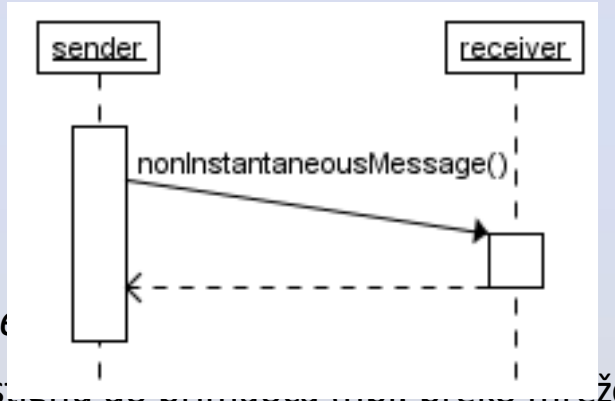
Trenutne poruke (*Instantaneous message*)

- Poruke se smatraju trenutnim ukoliko je vreme koje je potrebno da stignu do primaoca zanemarljivo (Slika 1)



N *neous message*

- Kod ovih poruka je potrebno određeno vreme da stigne do primaoca (npr. preko mreže) – takve poruke se crtaju kosom linijom (Slika 2)

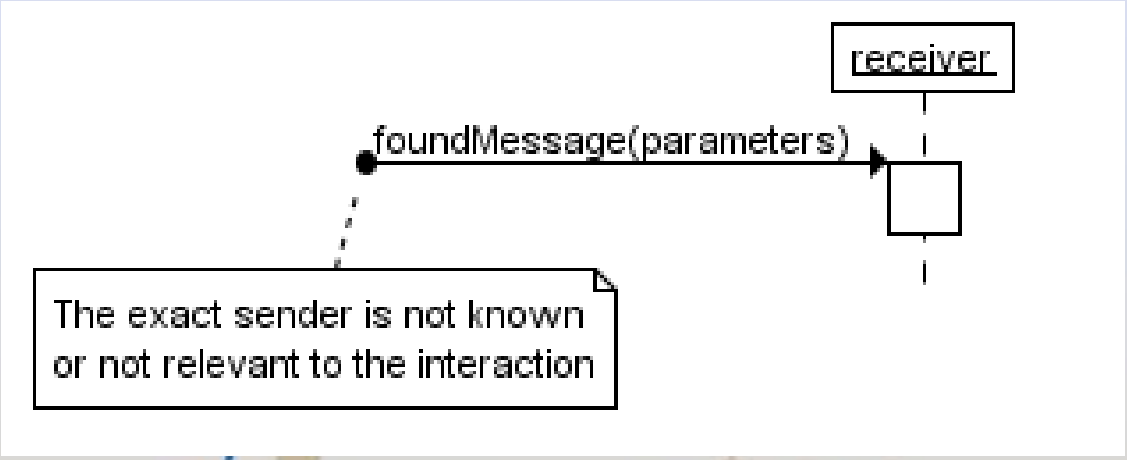


- Ovakve poruke bi trebalo crtati jedino ukoliko se želi naglasiti da poruka putuje sporim komunikacionim kanalom (i eventualno se želi dati iskaz o mogućem kašnjenju poruke)

Poruke

Pronađena poruka (*Found message*)

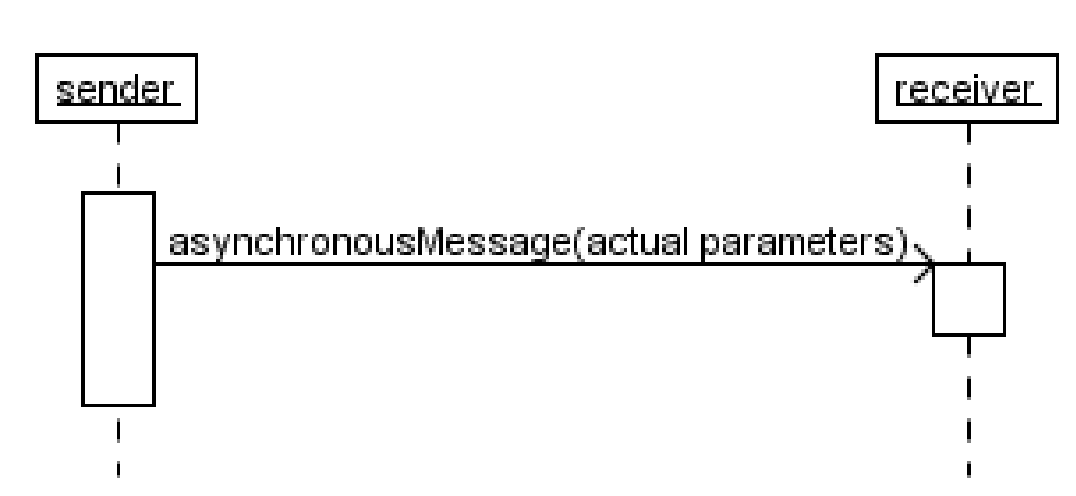
- Pronađena poruka je poruka kod koje pozivaoc nije prikazan (ili pošiljaoc nije poznat ili nije relevantno za interakciju ko je pošiljaoc)
- Prikazuje se kao pun krug na vrhu strelice



Poruke

Asinhrone poruke (*Asynchronous messages*)

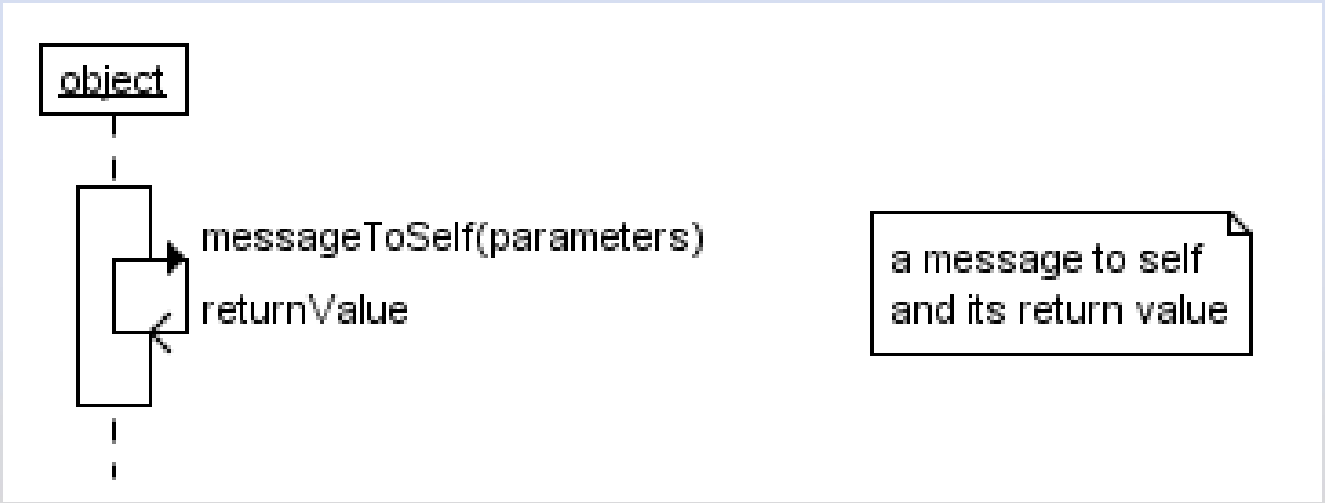
- Kod asinhrone poruke pošiljaoc **ne čeka primaoca da završi obradu poruke**, već odmah nastavlja sa slanjem drugih poruka
- Poruke koje u drugom procesu ili pozivu **započinju novu nit** su primeri asinhronih poruka
- Asinhrona poruka se prikazuje kao **otvorena strelica**



Poruke

Povratna poruka (*Message to self*)

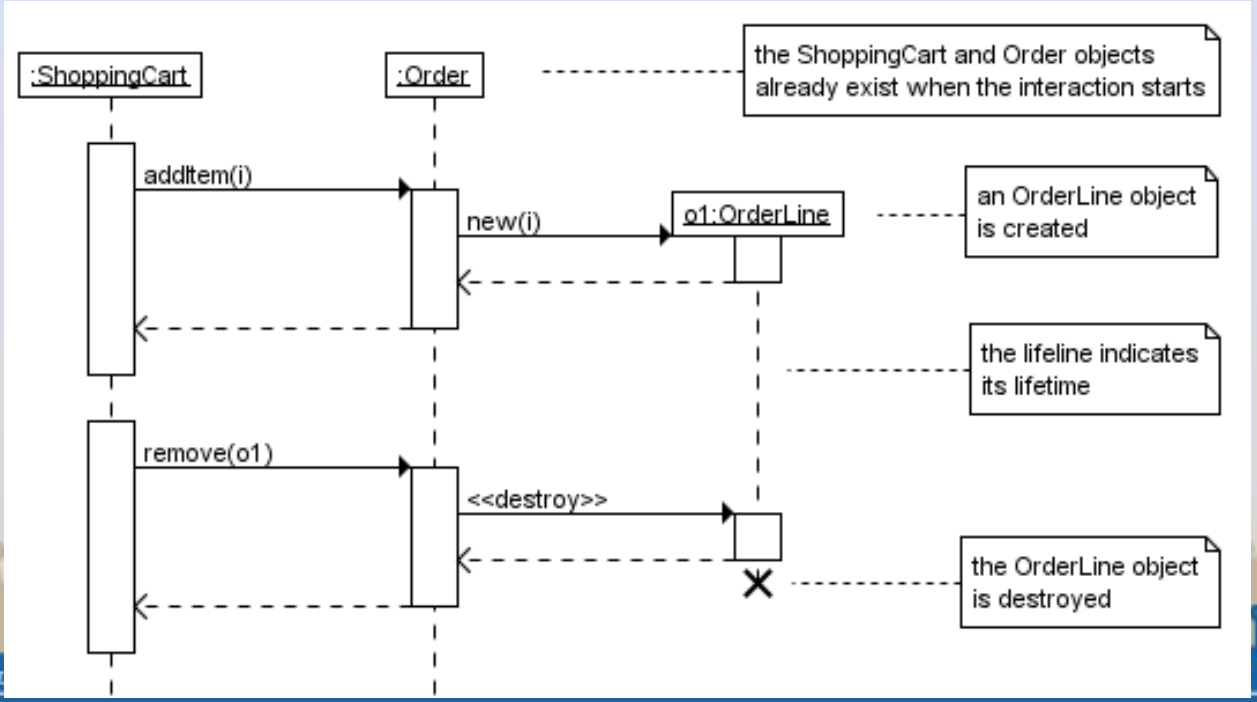
- Treba imati na umu da je svrha dijagrama sekvenci da prikaže interakciju između klasa, te s toga treba dobro razmisliti kada se dodaju povratne poruke na dijagram



Poruke

Kreiranje i uništenje (*Creation and destruction*)

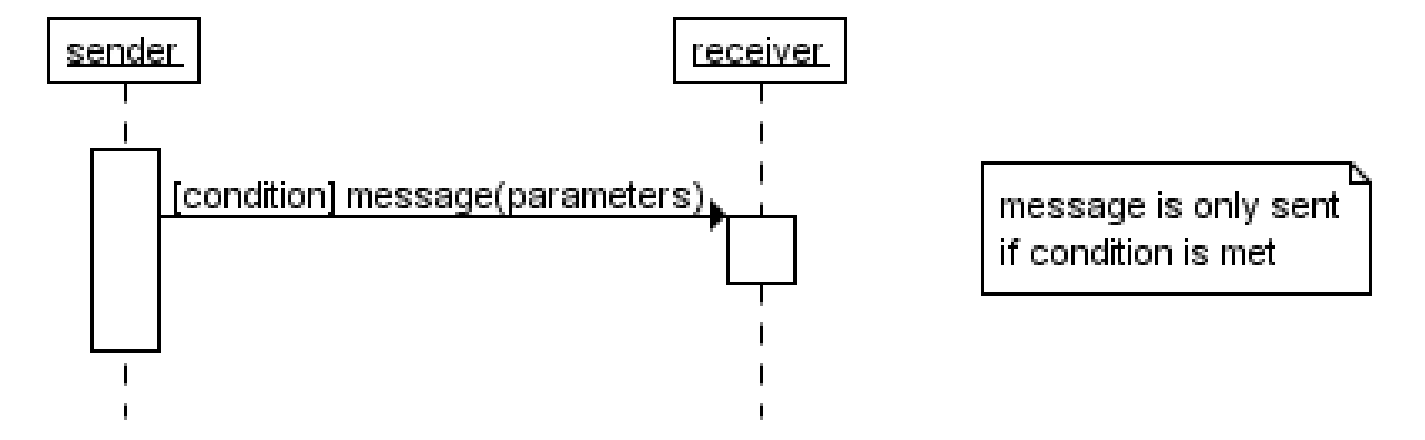
- Objekti koji se kreiraju na početku interakcije se stavljaju pri vrhu dijagrama, a svaki naredni se nastavljaju na prethodni, u trenutku njihovog nastajanja
- Životna linija objekta se proteže dokle god objekat egzistira
- Ukoliko se objekat tokom interakcije uništi, životna linija se prekida i stavlja se oznaka X



Poruke: Uslovne interakcije

Uslovne interakcije (*Conditional interaction*)

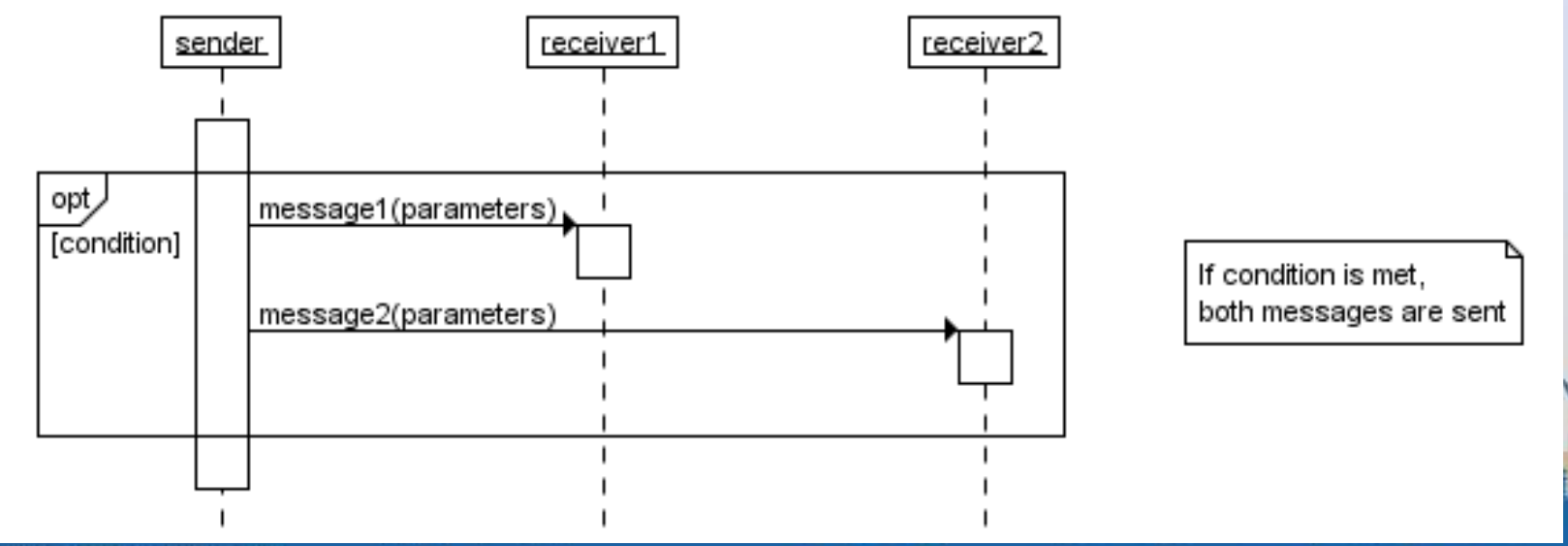
- Poruka može da uključi uslov što označava da se **poruka šalje samo ukoliko je ispunjen određeni uslov**
- Uslov se ispisuje u srednjoj zagradi ispred naziva poruke



Kombinovani fragment “opt”

Kombinovani fragment “opt” (*“opt” combined fragment*)

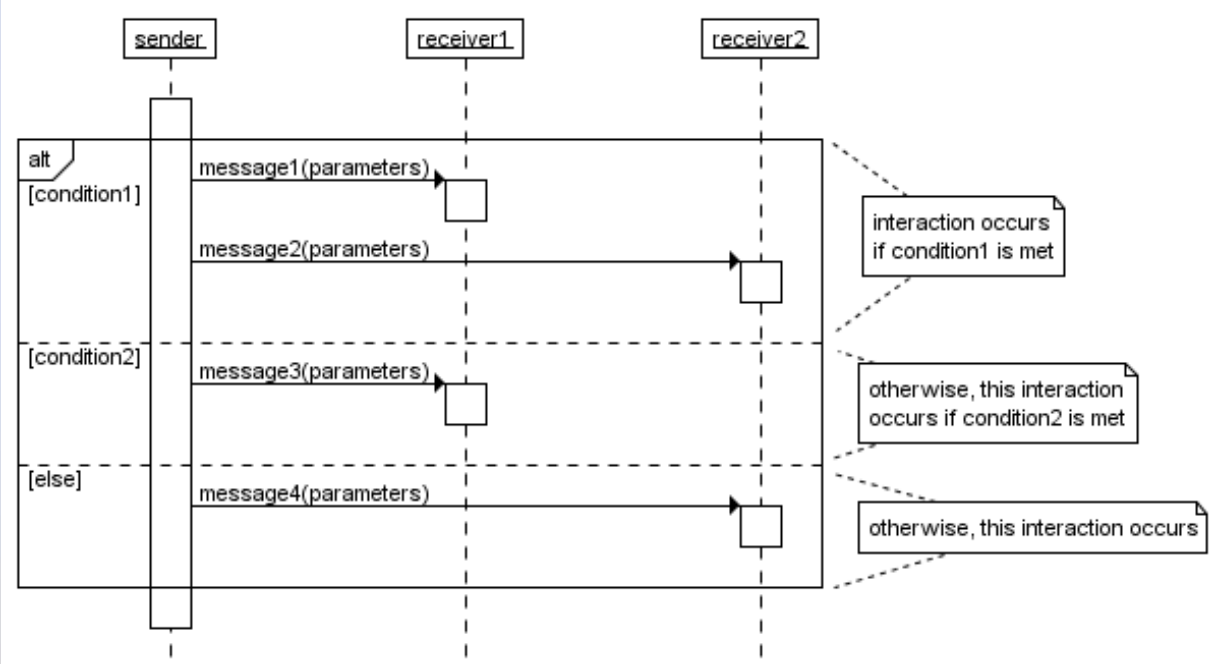
- Ukoliko se nekoliko poruka šalje pod istim uslovom onda se koristi “opt” fragment
- Prikazuje se kao veliki pravougaonik sa operatorom “opt” i uslovom i sadrži sve uslovne poruke koje treba da se dogode pod datim uslovom
- Uslovne poruke ili “opt” fragment je sličan *if* konstrukciji u programskim jezicima



Kombinovani fragment “alt”

Kombinovani fragment “alt”

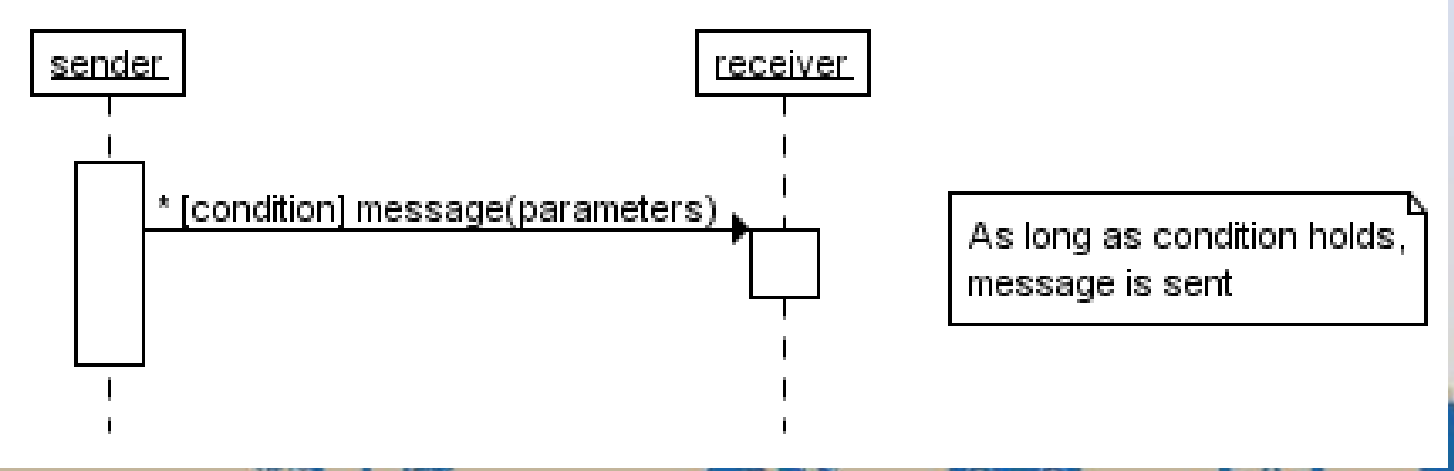
- Ukoliko se žele prikazati nekoliko alternativnih interakcija, koristi se “alt” kombinovani fragment
- Svaka alternativa ima uslov i sadrži interakcije koje se događaju kada se željeni uslov ispuni
- Najmanje jedan operand treba da se dogodi
- 'alt‘ fragment je sličan ugnježdenim *if-then-else* ili *switch/case* konstrukcijama kod programskih jezika



Ponavljajuća interakcija

Ponavljajuća poruka (*Repeated interaction*)

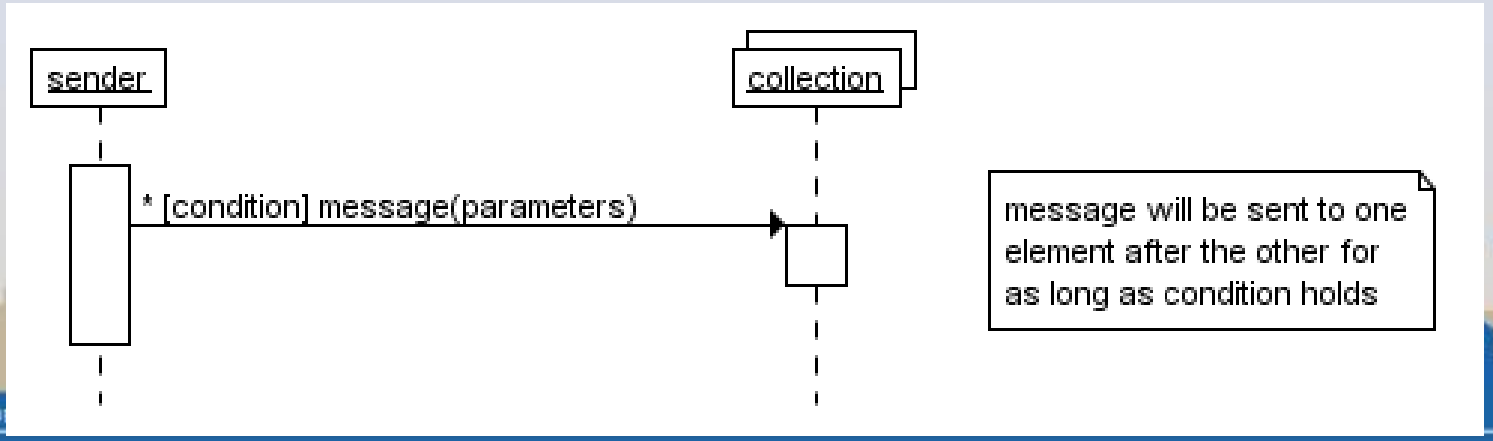
- Ukoliko ispred naziva poruke stoji simbol '*', znači da se poruka šalje uzastopnim ponavljanjem
- Uslov ukazuje pod kojim stanjem se poruka ponovo šalje
- Dok god je uslov ispunjem poruka se ponavlja



Ponavljajuća interakcija

Slanje ponavljajuće poruke elementima kolekcije

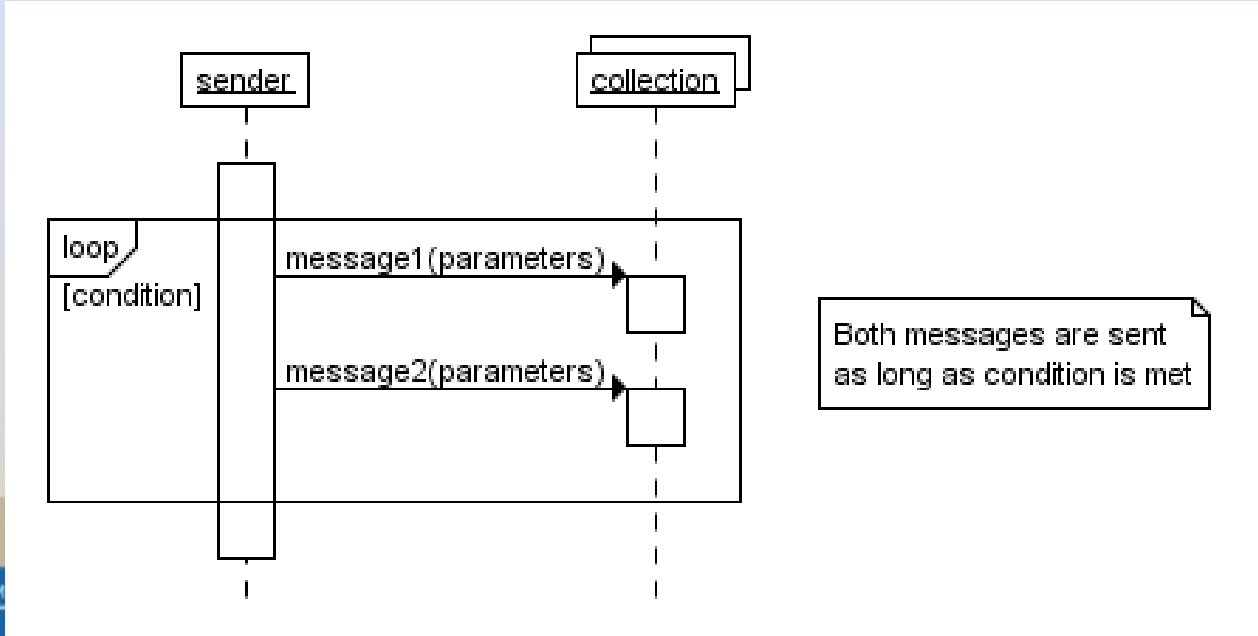
- Češće se koristi za slanje iste poruke različitim elementima u kolekciji
- U ovakvom scenariju, primaoc ponavljajuće poruke je višestruki objekat, a uslov ukazuje na stanje koje kontroliše ovo ponavljanje
- Svaki element u kolekciji prihvata poruku
- Za svaki element pre nego što mu se pošalje poruka proverava se stanje
- Obično, stanje predstavlja filter koji probira elemente iz kolekcije (npr., “novi klijenti”, “odrasli”, “svi” filteri za kolekciju objekata Osoba)



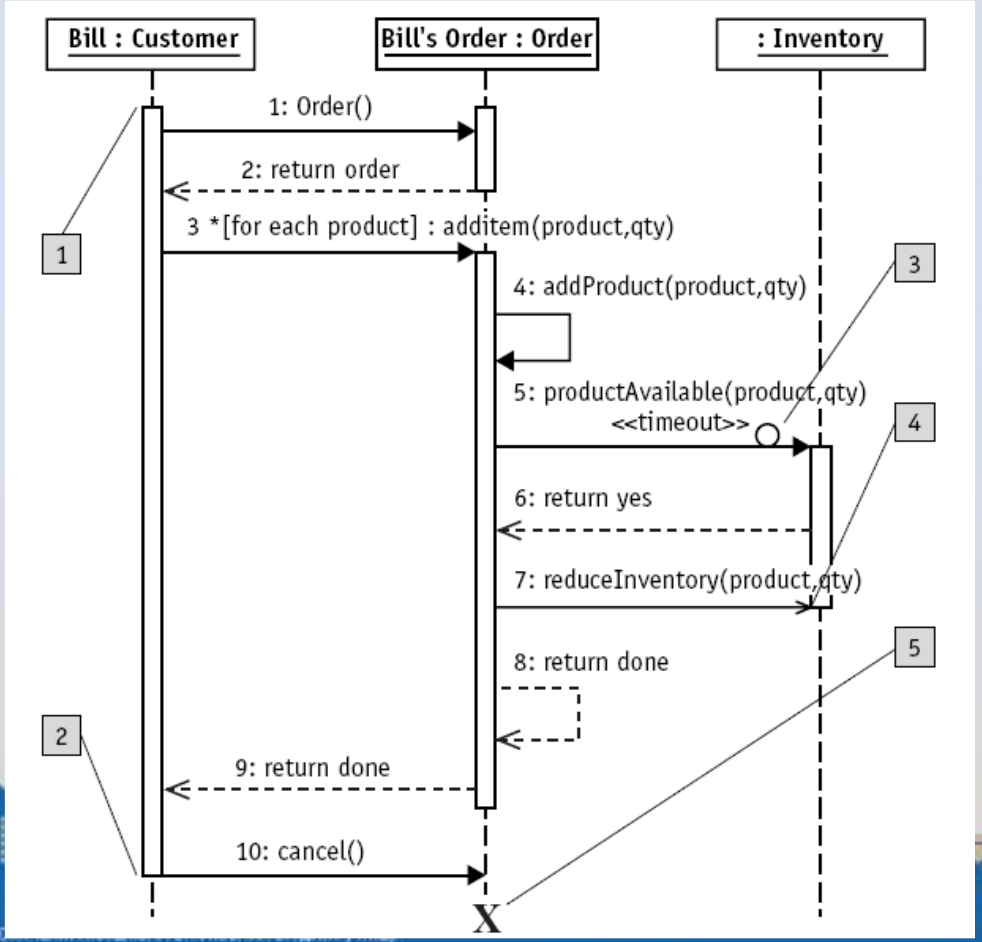
Ponavljajuća interakcija

“loop” kombinovani fragment

- Ukoliko se više poruka šalje u istoj iteraciji, koristi se “loop” kombinovani fragment
- Operator fragmenta loop (petlja), a uslov predstavlja stanje koje kontroliše petlju
- Primaoc ponavljajuće poruke je kolekcija, a uslov uglavnom predstavlja određeni filter za elemente kolekcije



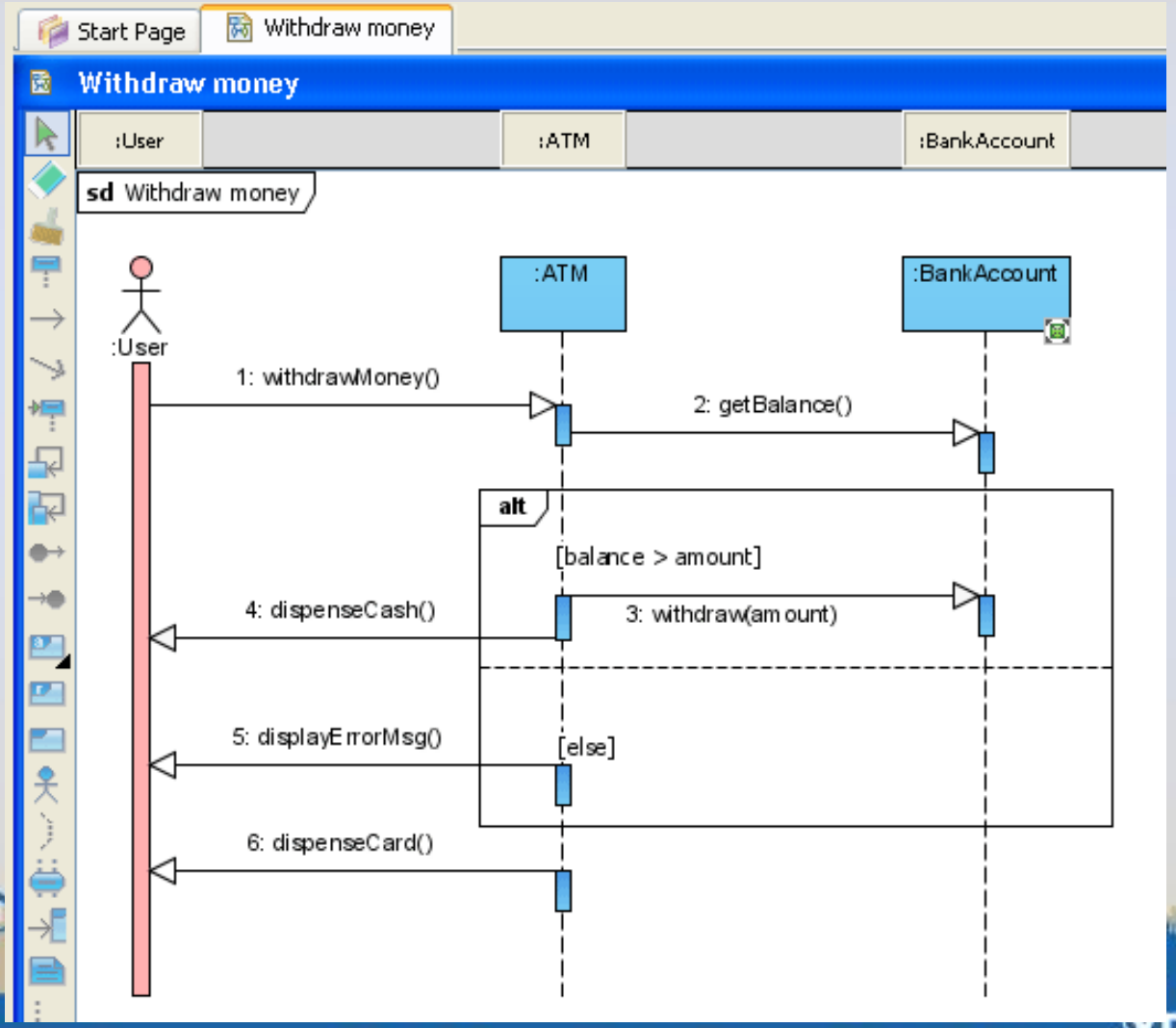
Definisanje proširenih notacija na dijagramu sekvenci



1. Activation: The start of the vertical rectangle, the activation bar
2. Deactivation: The end of the vertical rectangle, the activation bar
3. Timeout event: Typically signified by a full arrowhead with a small clock face or circle on the line
4. Asynchronous event: Typically signified by a stick arrowhead
5. Object termination symbolized by an X

ATM sistem

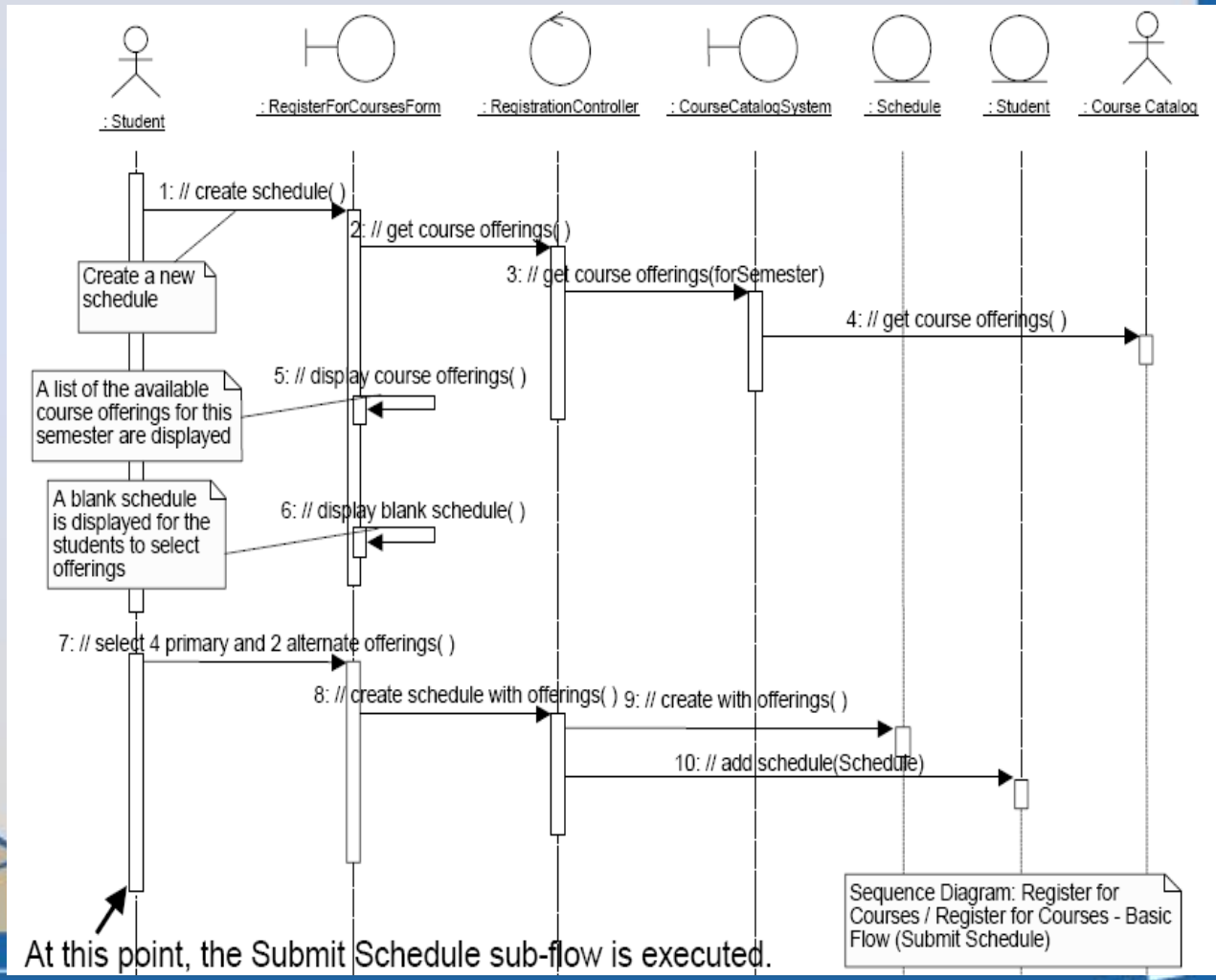
- Scenario: podizanje novca



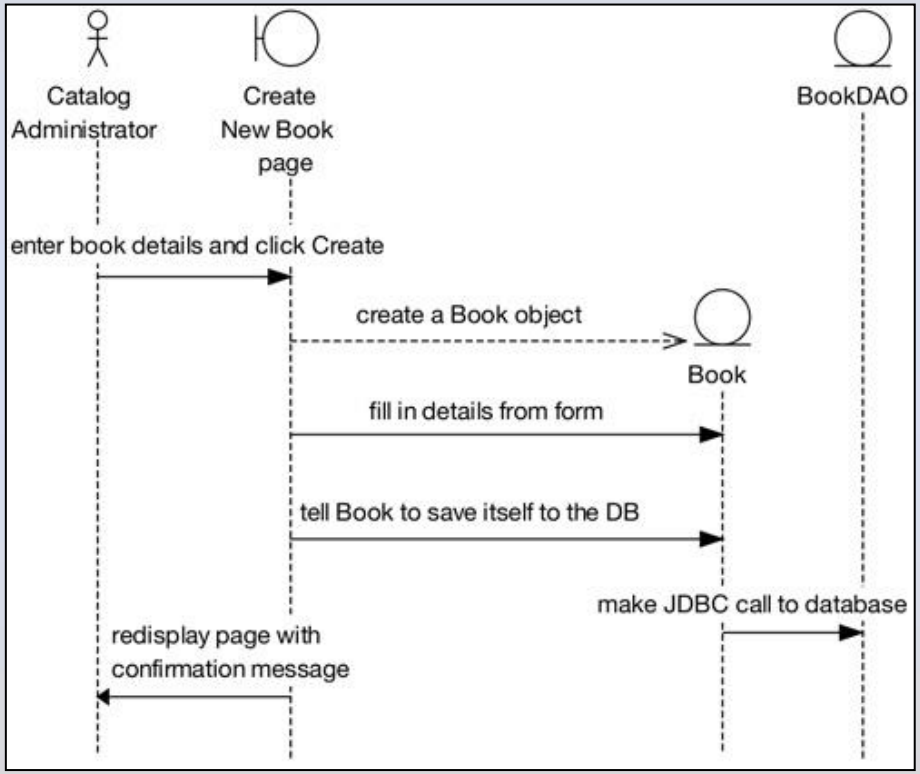
Primer dijagrama sekvenci

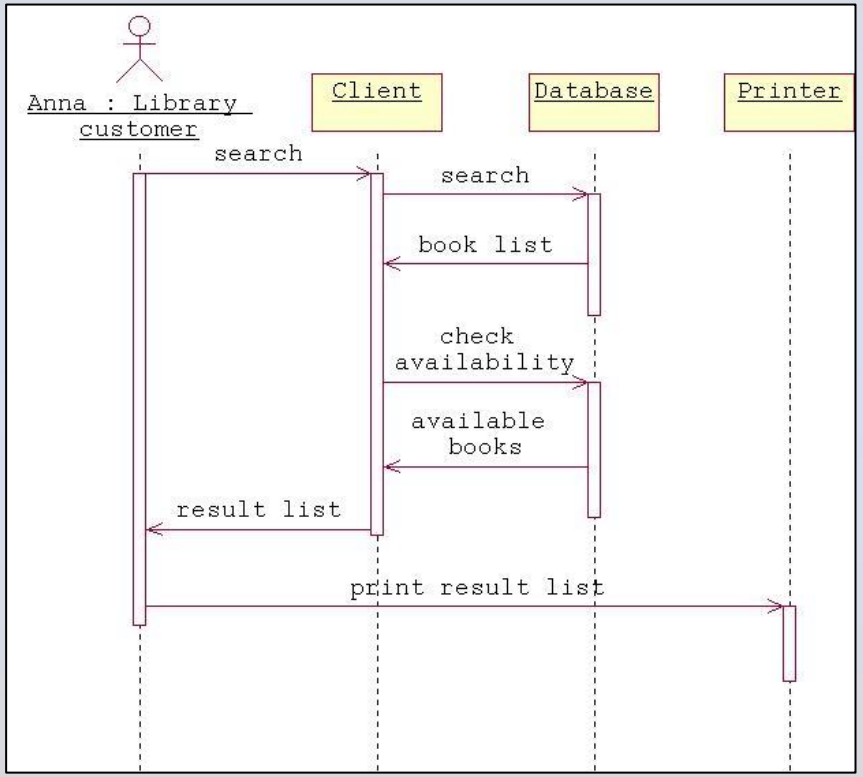
Scenario: Kreiranje rasporeda kurseva u okviru use case-a: Registracija na kurseve

- *RegisterForCoursesForm:*
  - zna koji podaci joj trebaju i kako da ih prikaže, ali ne zna gde da ih potraži, to je odgovornosti *RegistrationController*
  - u interakciji je sa akterom Student
- RegistrationController razume kako su povezani Students i Schedules
- Jedino je klasa *CourseCatalogSystem* u interakciji sa spoljnim *legacy* Course Catalog System



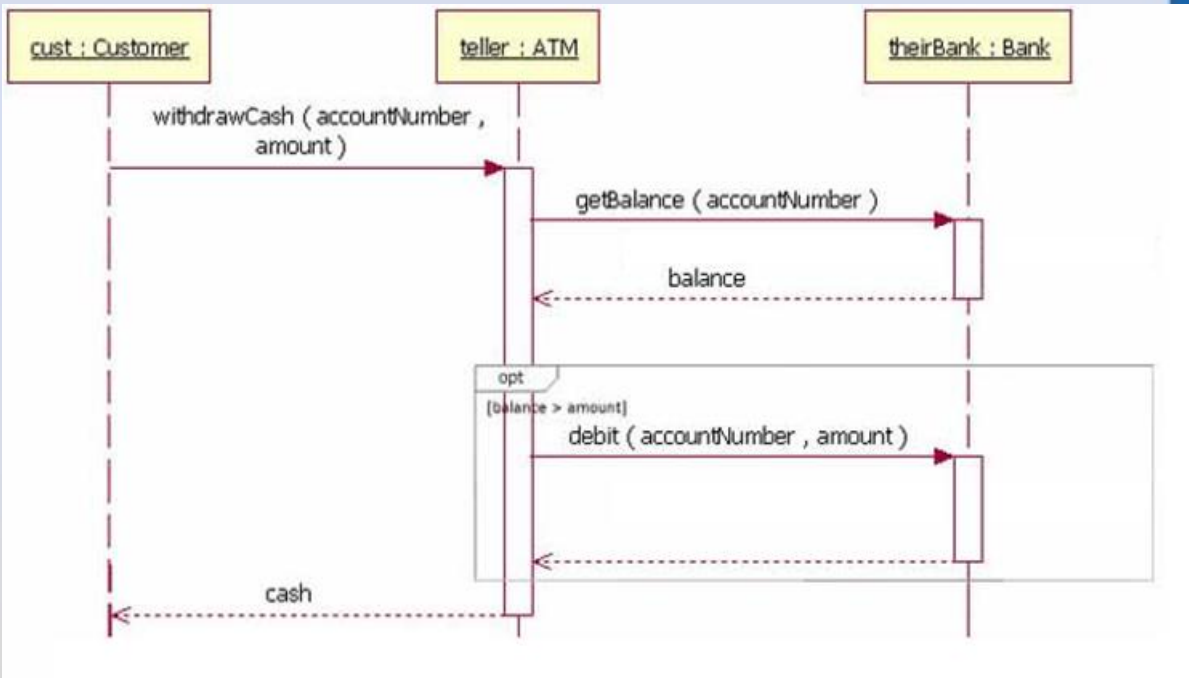
Primer:
Kreiranje zapisa i pretraga knjiga





Primer za vežbu: ATM

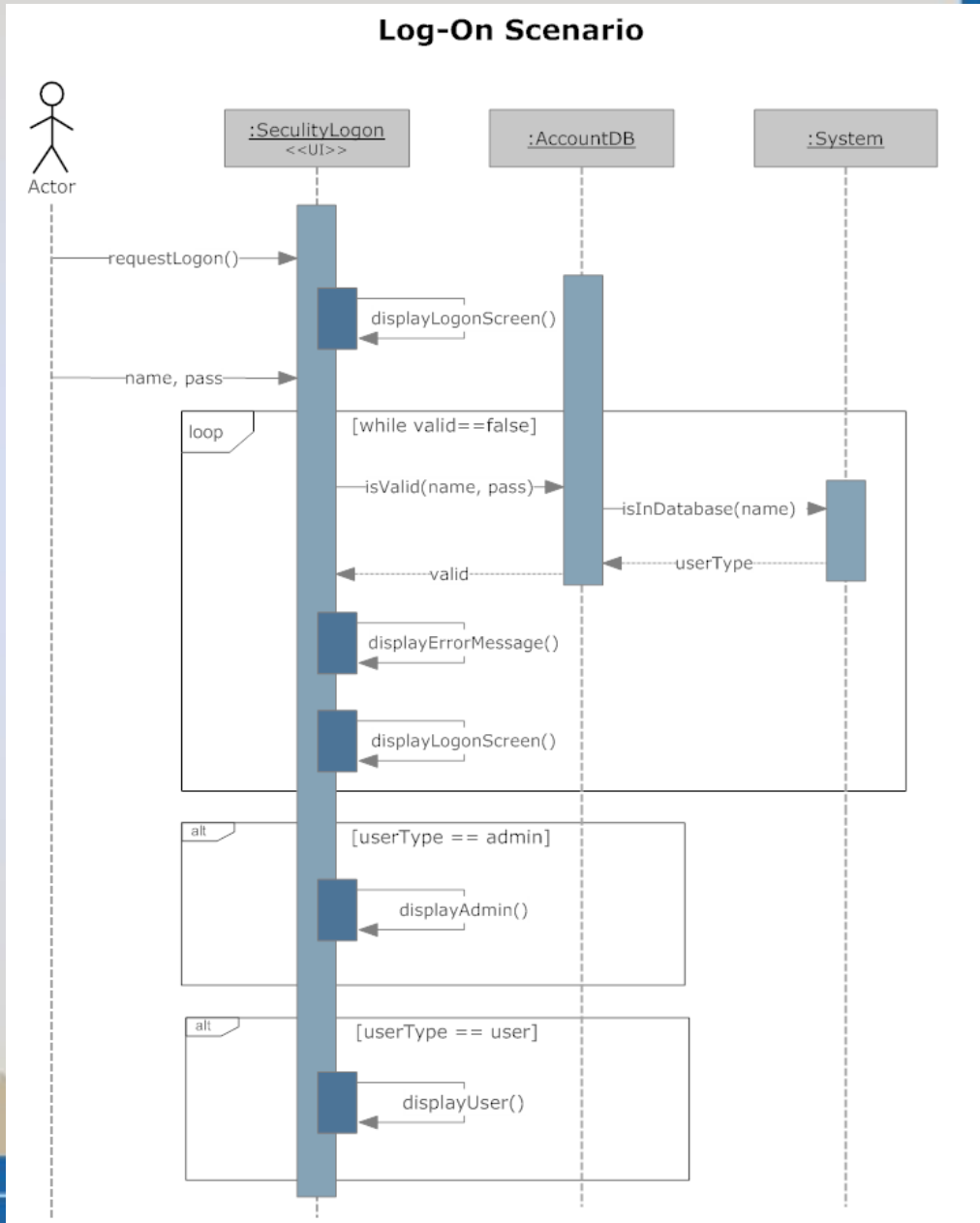
- Korisnik bira opciju podizanja novca (*withdrawCash*) – ova poruka sa parametrima (broj računa i iznos) se šalje ATM interfejsu
- ATM zahteva od sistema banke stanje na računu (getBalance) za dati broj računa
- Sistem banke vraća stanje (balance)
- Ukoliko je stanje veće od iznosa, treba da se zaduži dati račun za iznos koji je unešen za podizanje
- ATM izbacuje novac



Primer za vežbu

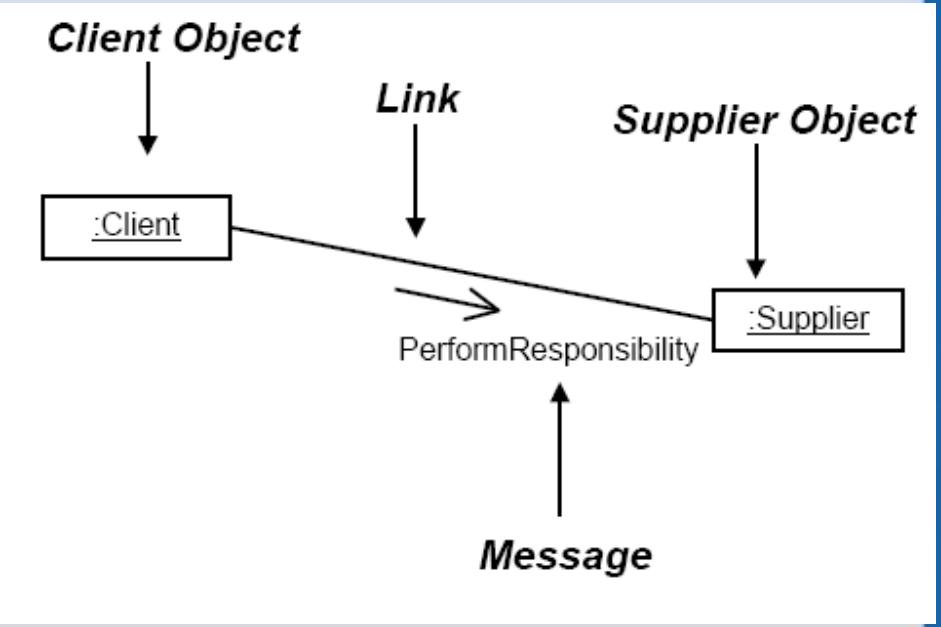
Scenario logovanja na sistem

- Klijent želi da se loguje - *requestLogon*()
- *SecurityLogon* prikazuje ekran za logovanje na sistem - *displayLogonScreen*()
- Klijent unosi *name, pass*
- Izvršava se petlja pod sledećim uslovom [*while valid==false*]:
  - *SecurityLogon* šalje na proveru *user* i *pass* (isValid(name, pass)) ka bazi korisničkih naloga (AccountDB)
  - *AccountDB* proverava da li naziv postoji u bazi (*isInDatabase(name*) u Sistemu (*System*)
  - Sistem vraća tip korisnika (*userType*)
  - *AccountDB* vraća info o validnosti (*valid*)
  - Interfejs *SecurityLogon* prikazuje poruku o grešci (displayErrorMessage())
  - Ponovo otvara tj. prikazuje Logon ekran (displayLogonScreen())
- SecurityLogon proverava uslov:
  - Ukoliko je [userType==admin] onda prikazuje ekran za logovanje administratora (displayAdmin())
  - Ukoliko je [userType==user] onda prikazuje ekran za logovanje korisnika (displayUser())



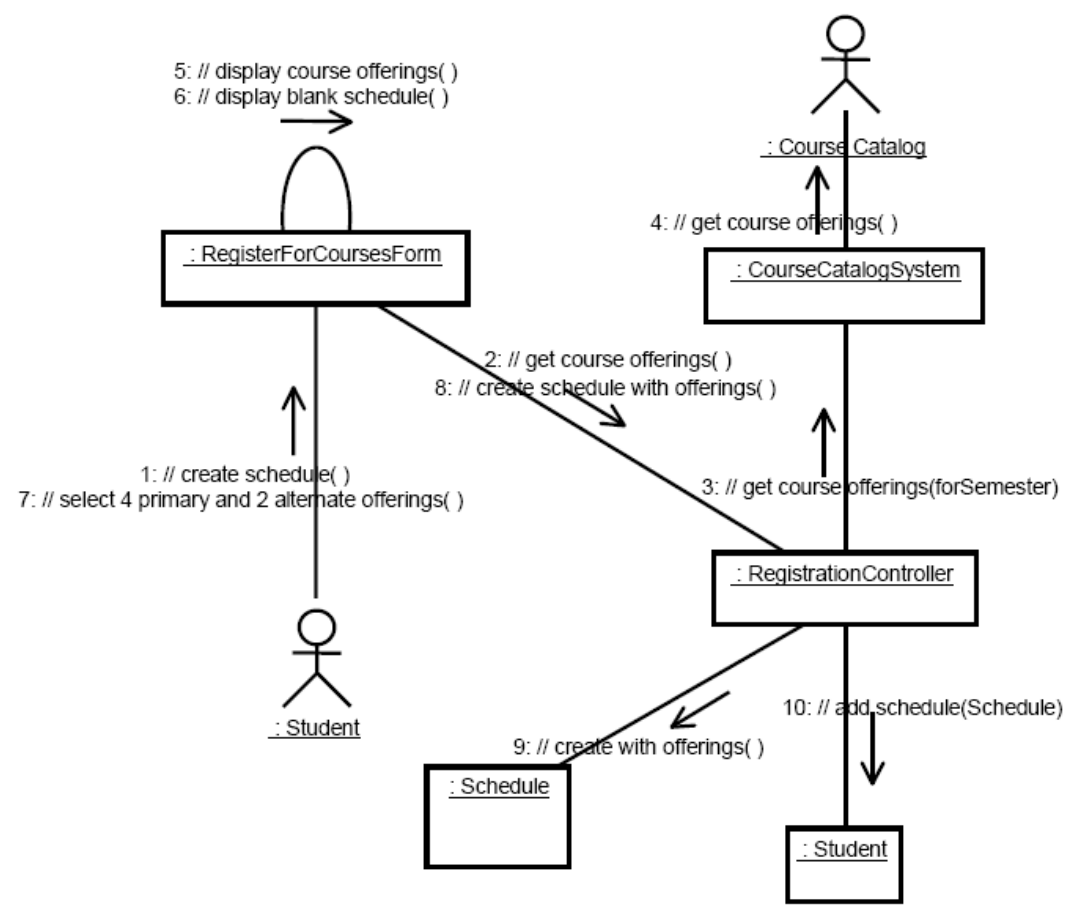
Dijagram komunikacije

- Opisuje interakcije između objekata prikazujući njihove veze i poruke koje mogu poslati jedni drugima
- **Objekat** se predstavlja na jedan od tri načina:
  - NazivObjekta:NazivKlase
  - NazivObjekta
  - :NazivKlase
- **Poruka** je komunikacija između objekata koja prenosi informaciju
  - Označava se kao imenovana strelica pored linka, što znači da se link koristi za prenos ili za implementaciju predaje poruke ciljnom objektu
  - Strelica ukazuje na pravac ciljnog objekta (onoga ko prima poruku)
  - Brojevi redosleda se koriste radi označavanja **redosleda poruka u interakciji**



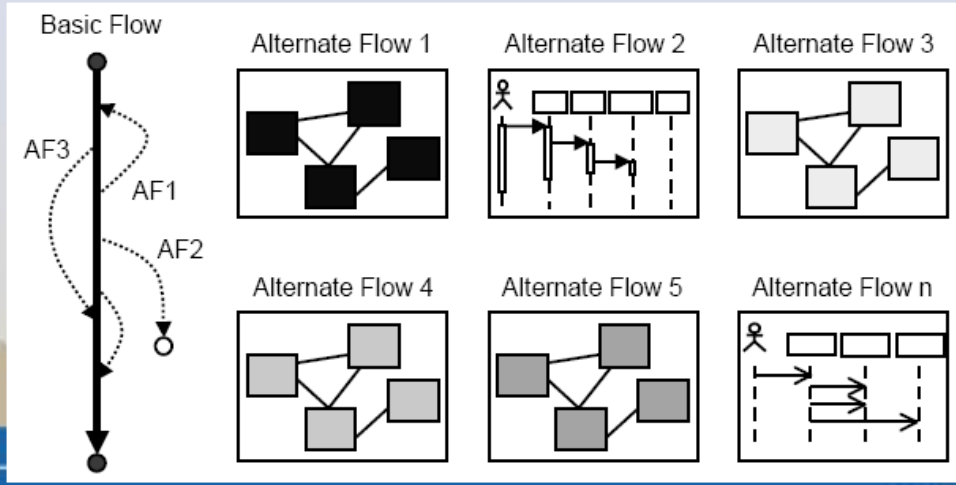
Primer dijagrama komunikacije

Scenario: Kreiranje rasporeda kurseva
u okviru use case-a: Registracija na kurseve



Jedan dijagram interakcije nije dovoljan

- Treba modelovati većinu tokova događaja kako bi se osiguralo da su identifikovani svi zahtevi za operacijama klasa
  - Počnite sa **opisivanjem osnovnog toka**, koji je opšti ili najvažniji tok događaja
  - Zatim opišite varijante kao što su tokovi izuzetaka
  - Trivijalne tokove, koji se odnose samo na jedan objekat, bi trebalo zanemariti
- Primeri tokova izuzetaka:
  - Rukovanje sa greškama: **Šta bi sistem trebao da radi kada bi se pojavila greška?**
  - Isticanje vremena: **Ukoliko korisnik ne odgovori u određenom periodu**, onda bi use case trebao da preuzme određene mere
  - **Pogrešni inputi**

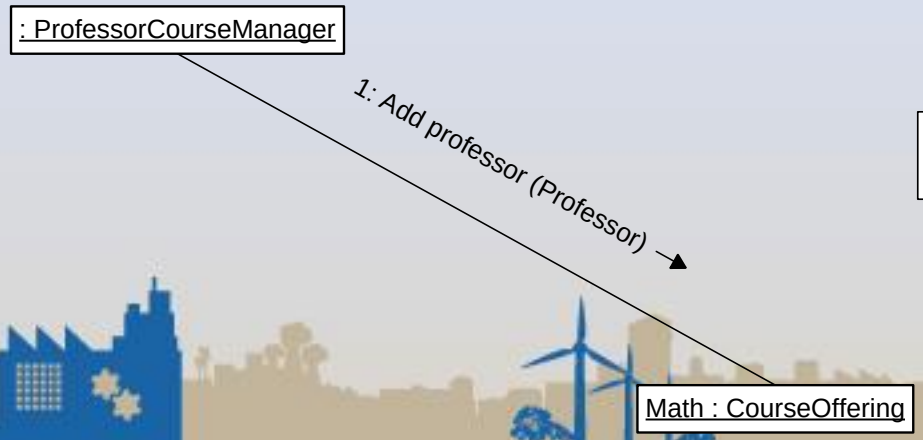


Dijagrami komunikacije *vs* Dijagrami sekvenci

- Dijagrami saradnje i sekvenci prikazuju slične informacije, ali na različite načine
- Predlo je da se dijagrami komunikacije koriste u fazi analize, a dijagram sekvenci kod projektovanja

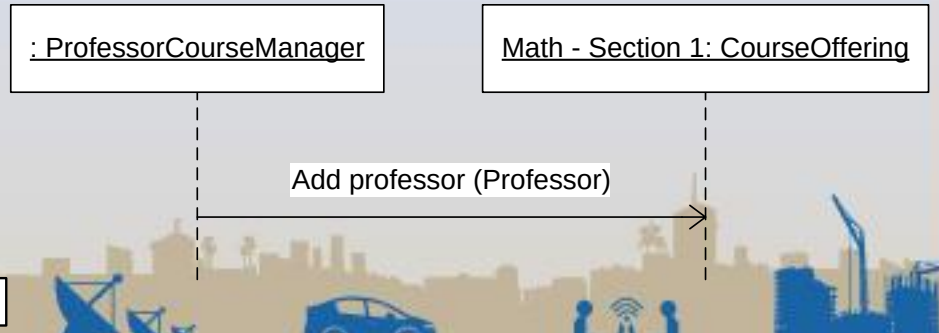
Dijagrami komunikacije

- Ističu strukturu saradnje objekata i pružaju jasniju sliku o relacijama i kontrolama koje postoje između objekata koji učestvuju u use case-u

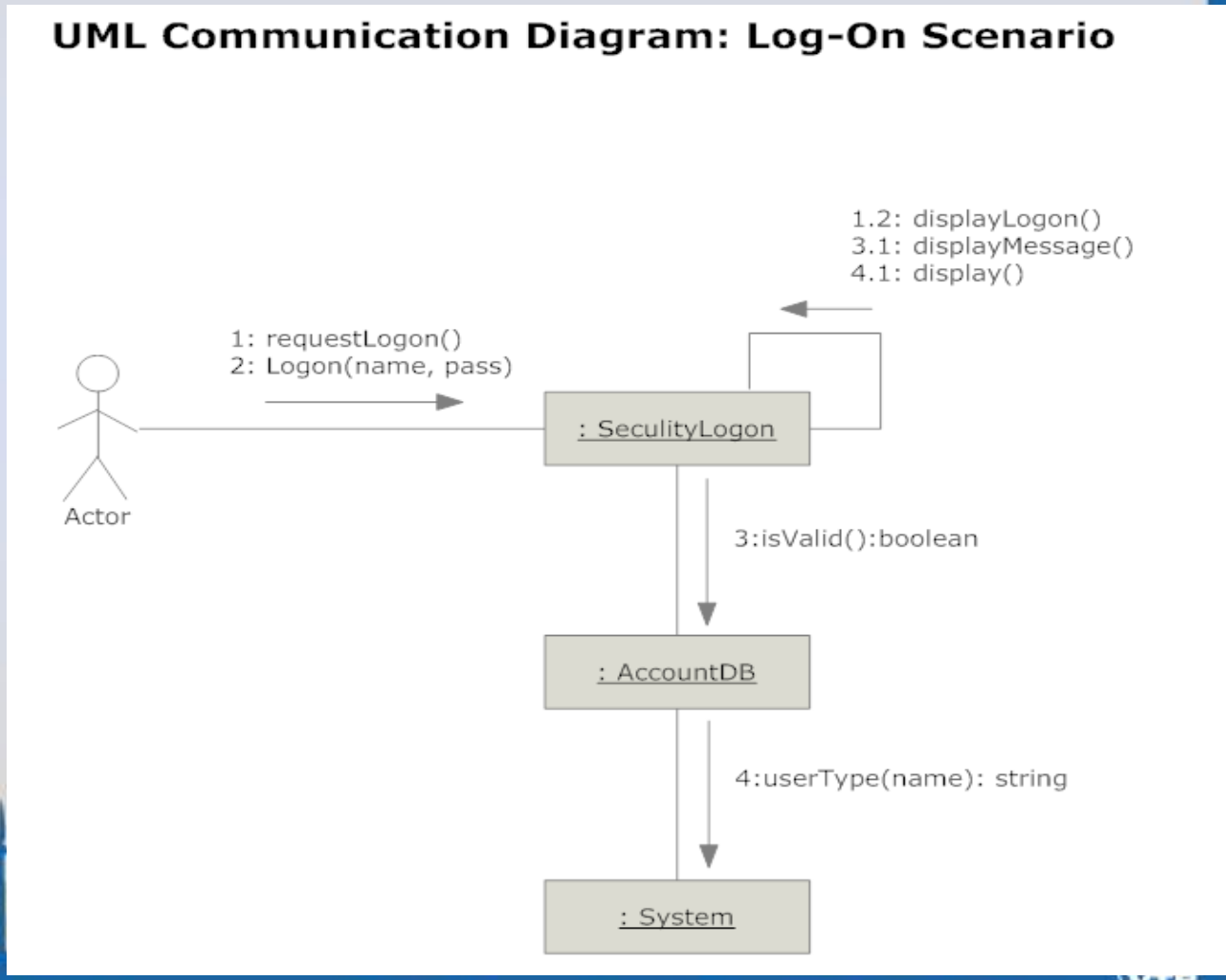


Dijagrami sekvenci

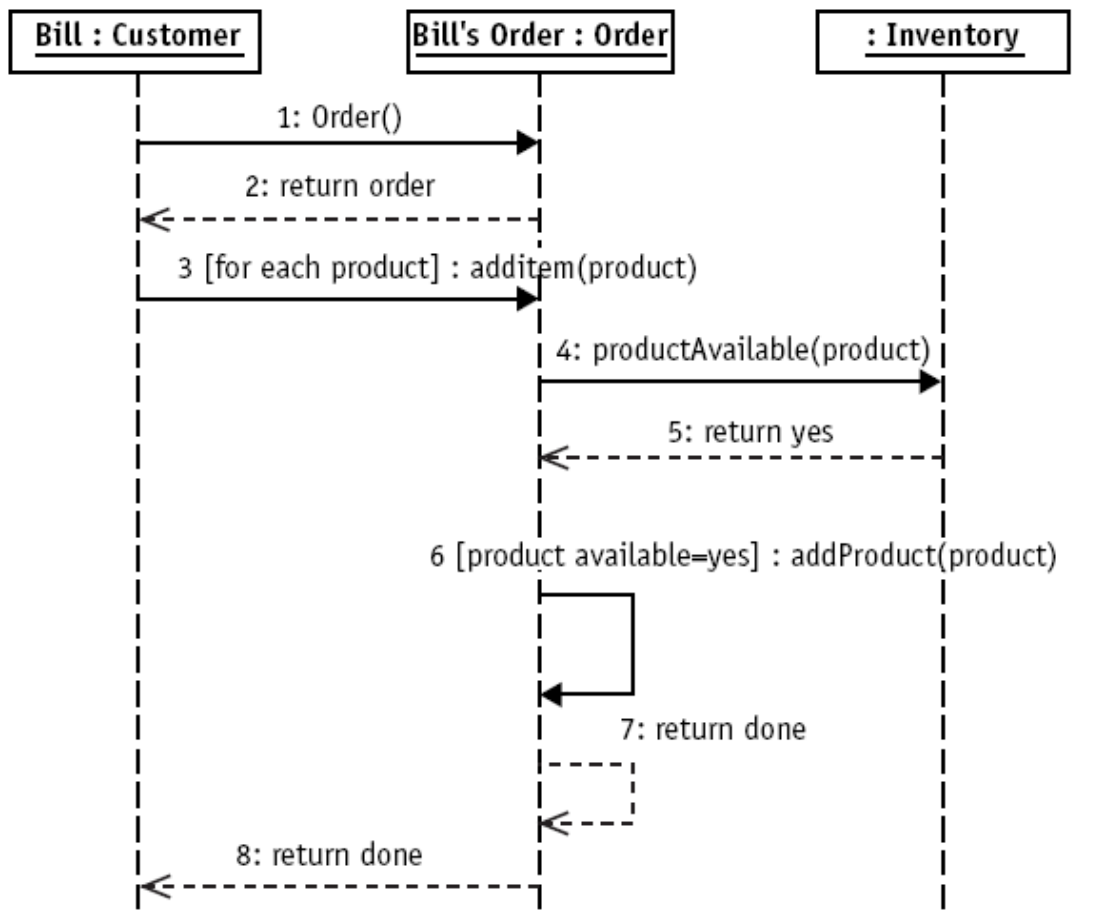
- Prikazuju jasnu sekvencu poruka i bolje su za složena scenarija u realnom vremenu
- Uključuju hronološku sekvencu
- Vremenska dimenzija je veoma laka za čitanje; operacije i parametre je lakše predstaviti i lakše se upravlja velikim brojem objekata



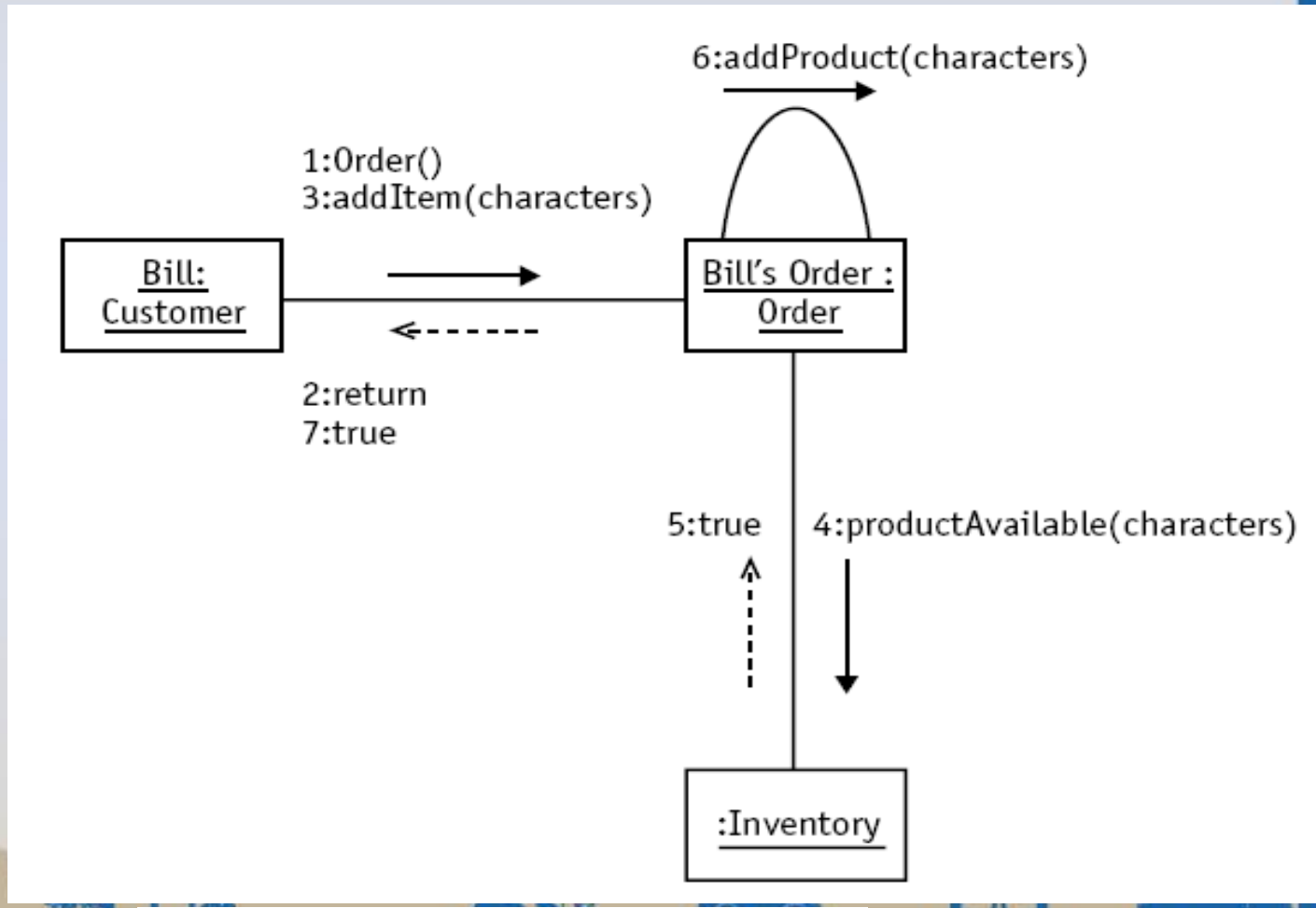
Dijagram komunikacije: log-on scenario



Vežba 1: Napraviti dijagram komunikacije

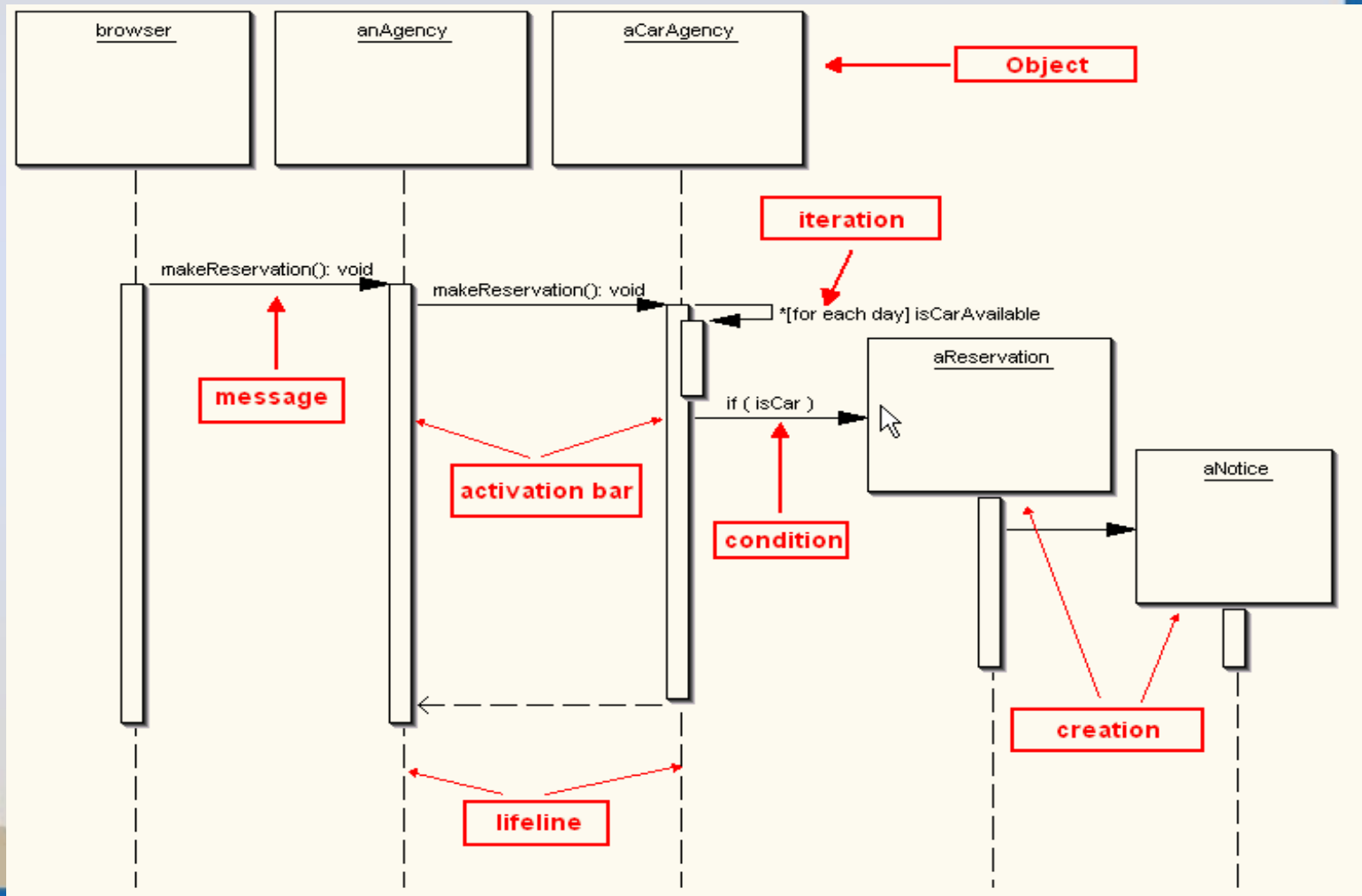


Vežba 1: Dijagram komunikacije

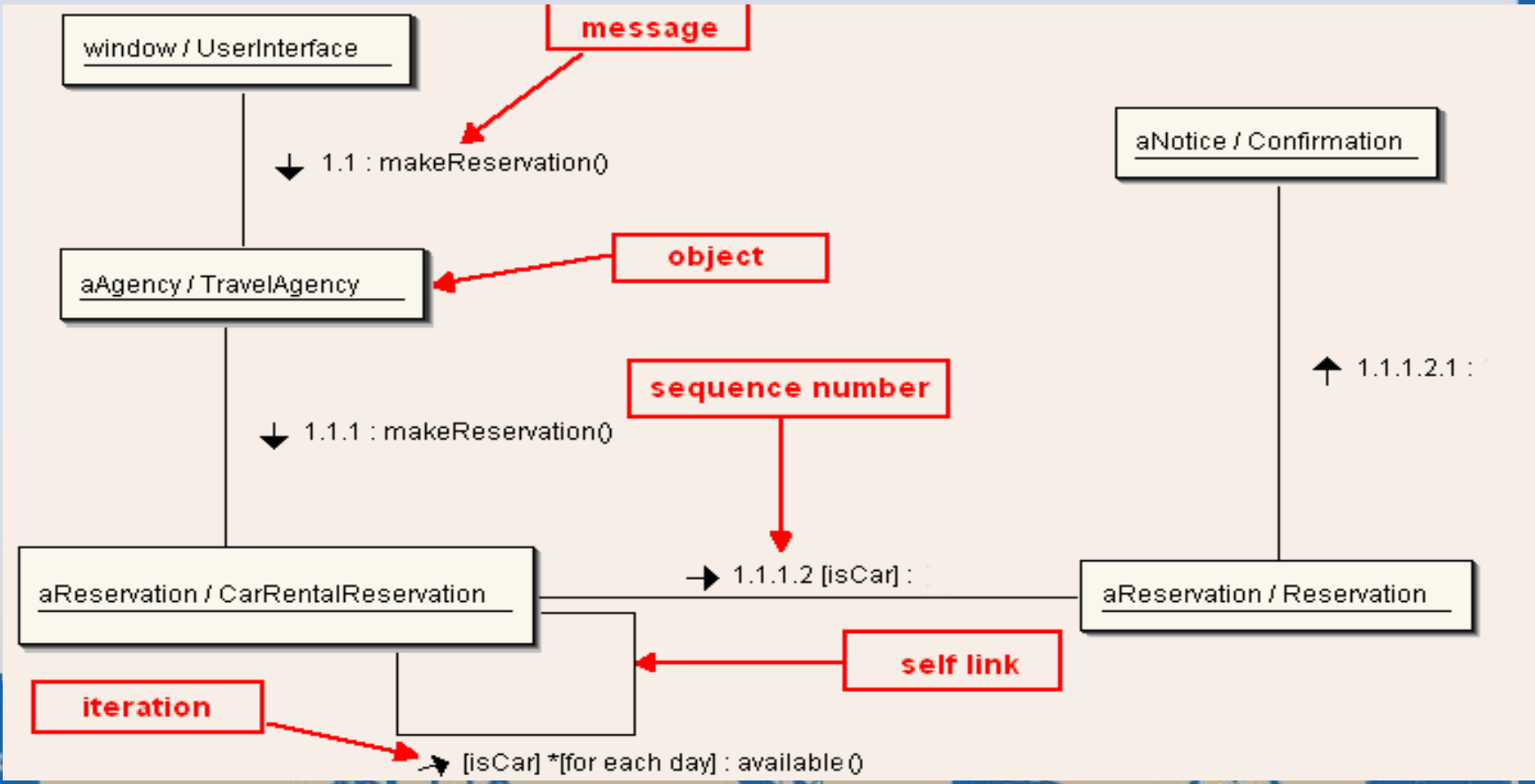


Collaboration diagram of Customer placing an Order

Vežba 2: Rent-a-car

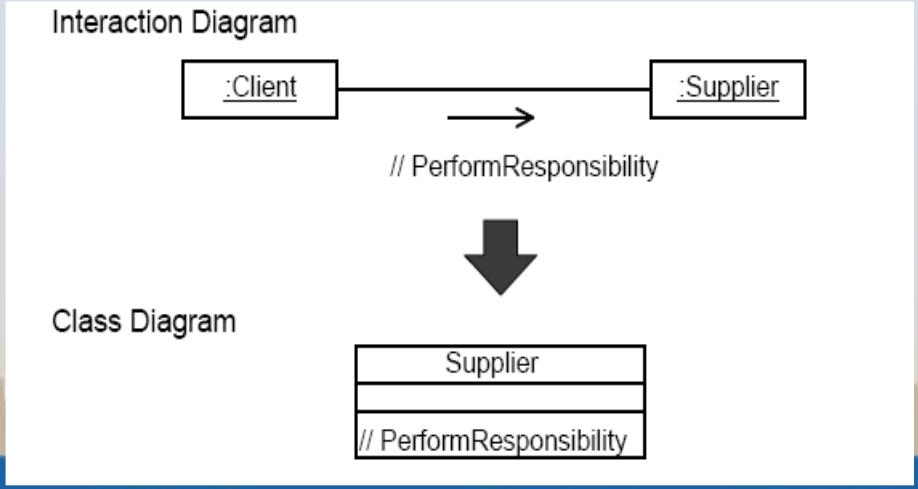


Vežba 2: Dijagram komunikacije

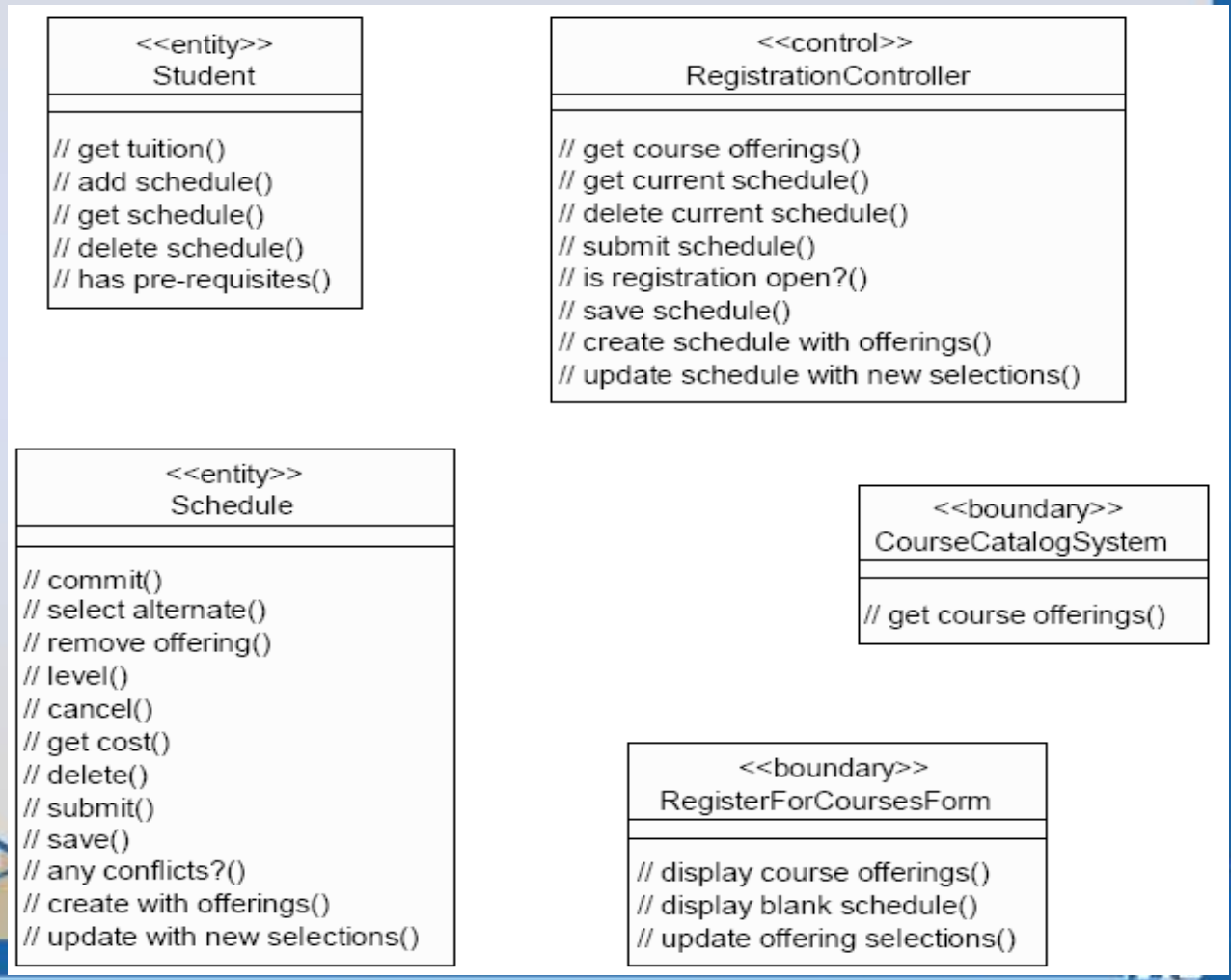


Konceptualni dijagram klasa

- Koje su odgovornosti (ponašanja) klase?
  - Akcije/Metode koje objekat može da izvršava
  - Znanje koje objekat održava i pruža drugim objektima
- Kako pronaći ponašanja?
  - Mogu se izvući iz poruka sa dijagrama interakcije
  - Druga ponašanja se mogu izvući iz nefunkcionalnih zahteva



Primer identifikovanja metoda klasa za *use case* registrovanje na kurs



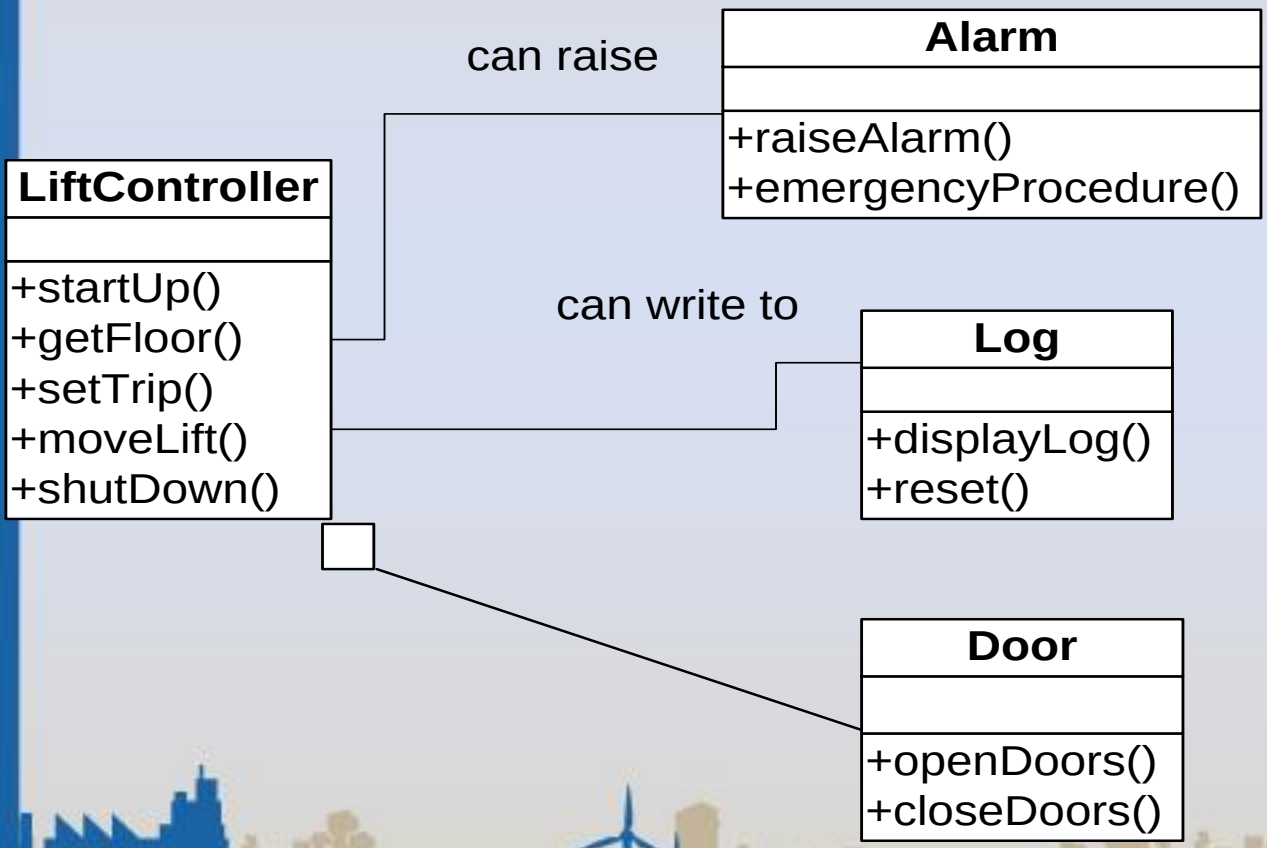
Šta su ključne apstrakcije?

- Ključna apstrakcija je koncept, koji nije pokriven u Zahtevima
- Nije cilj razviti kompletni model klasa, već samo definisati neke ključne apstrakcije i osnovne relacije
- Identifikovane klase će se najverovatnije menjati tokom projekta
- Definisanje ključnih apstrakcija:
  - Definisati klase i njihove relacije
  - Modelovati klase i relacije na dijagramu klasa
  - Mapirati klase ka mehanizmima analize

Identifikovanje ključnih apstrakcija na primeru klase Lift

- Da li je klasa dobro dizajnirana?
  - Obavlja puno posla, teško ju je održavati i nije dovoljno jasno šta klasa tačno radi
- Koliko klasa LiftController sadrži ključnih apstrakcija?
  - Svaka klasa treba da sadrži **samo jednu ključnu apstrakciju, tj. da predstavlja jednu stvar iz realnog života**
  - Klasa LiftController pokušava da modeluje bar 3 odvojene ključne apstrakcije, i to: Alarm, Vrata, Log…

Identifikovanje ključnih apstrakcija na primeru klase LiftController

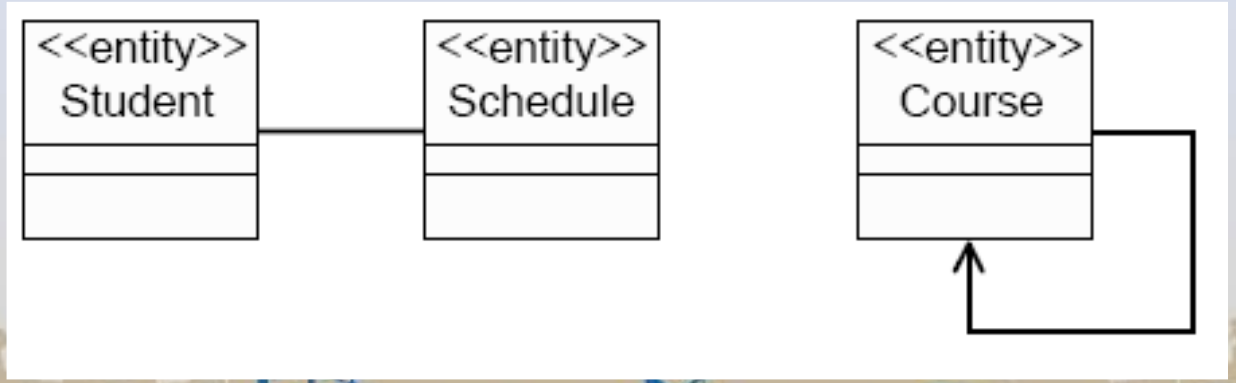


Održavanje konzistentnosti

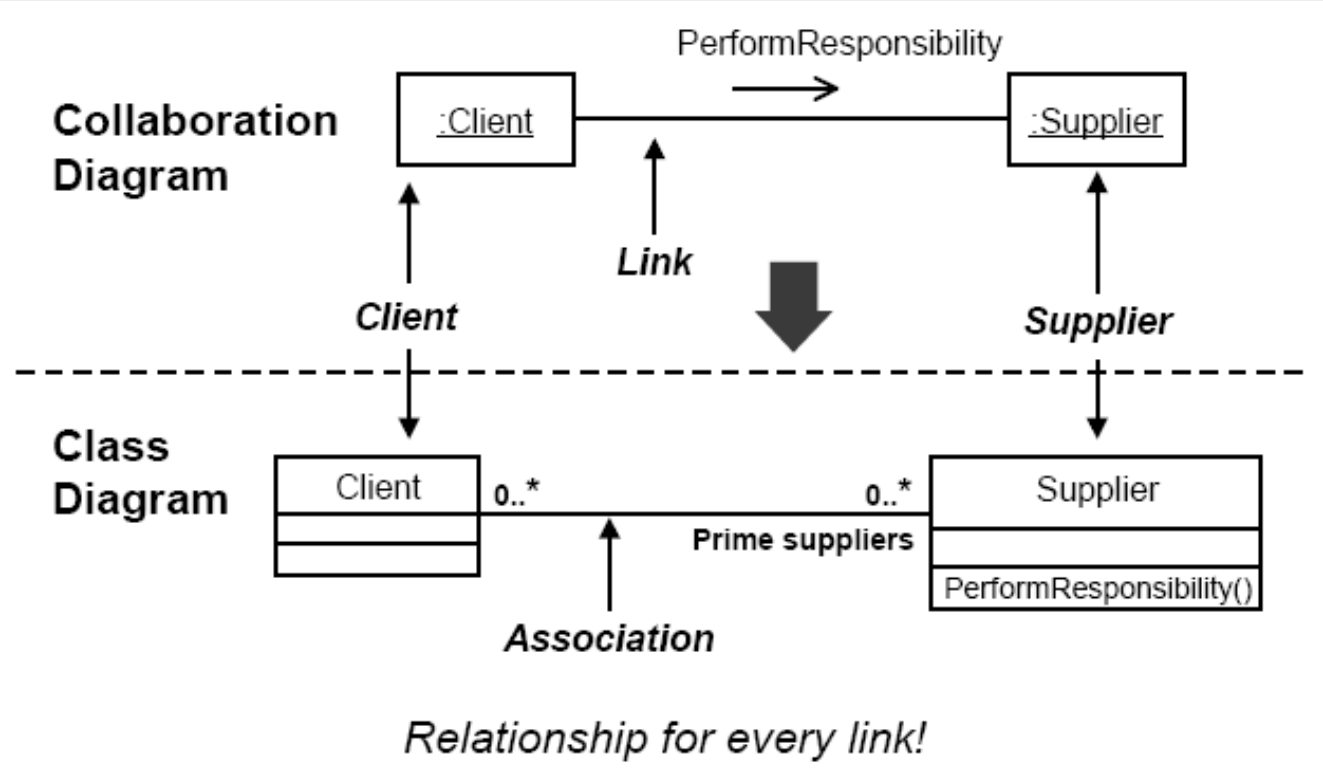
- Obratite pažnju na klase koje rade sve!
  - Svaka klasa bi trebala da ima nekoliko odgovornosti
  - Klasa sa samo jednom operacijom je verovatno suviše jednostavna i postavlja se pitanje čemu
  - Klasu koja ima previše operacija bi trebalo razdvojiti u nekoliko klasa
- Obezbediti da ne postoje dve klase sa sličnim odgovornostima
  - Treba ih kombinovati i ažurirati dijagram interakcije

Šta je asocijacija?

- Asocijacija predstavlja relaciju između dva ili više objekata različitih klasa
  - Većina asocijacija je jednostavna (između tačno dve klase) i prikazuje se kao puna linija između klasa
  - Ponekad klasa ima asocijaciju na samu sebe - uglavnom označava da jedna instanca klase ima asocijaciju ka drugoj instanci iste klase
  - Naziv asocijacije treba da bude glagol



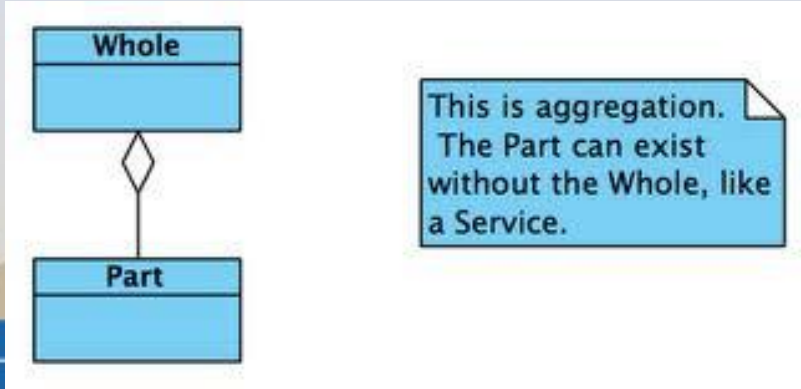
Pronalaženje relacija



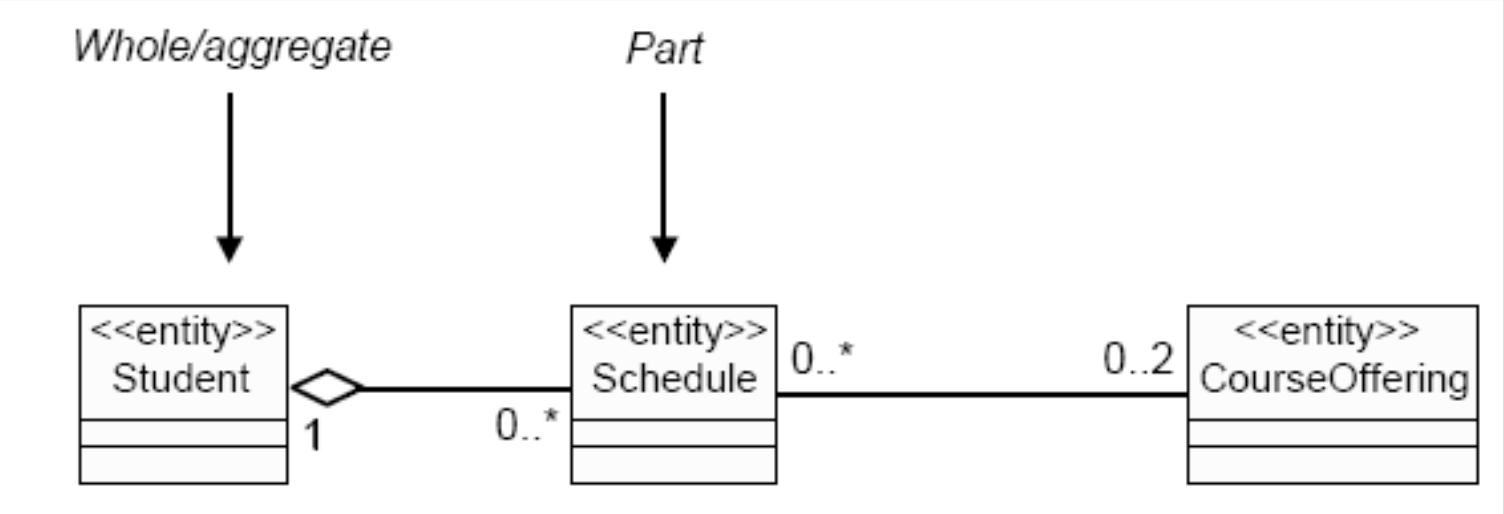
- Fokusirajte se samo na asocijacije koje su neophodne za realizaciju use case-a
- Svakoj asocijaciji treba dati nazive i multiplikativnosti

Šta je agregacija?

- Specijalni oblik asocijacije koja modeluje relacije između celine i njenih delova
  - Prazan romb je na strani celine ukazuje na relaciju agregacije
  - Kada je klasa u relaciji agregacije sa samom sobom, to znači da jedna instanca klase se sastoji od drugih instanci iste klase
- Relaciju agregacije bi trebalo koristiti kada je:
  - Jedan objekat fizički sačinjen od drugih objekata (npr., kola su fizički sačinjena od motora, točkova i dr.)
  - Jedan objekat logički sastavljen od drugih objekata (npr., porodica je sastavljena od roditelja i dece)
  - Jedan objekat fizički sadrži druge objekte (npr., avion fizički sadrži pilota)



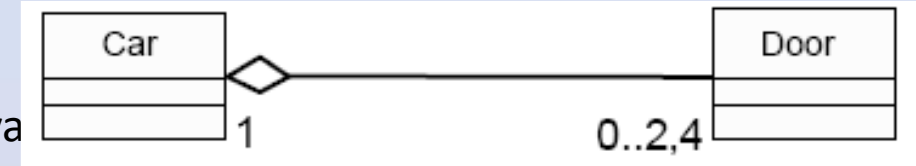
Primer



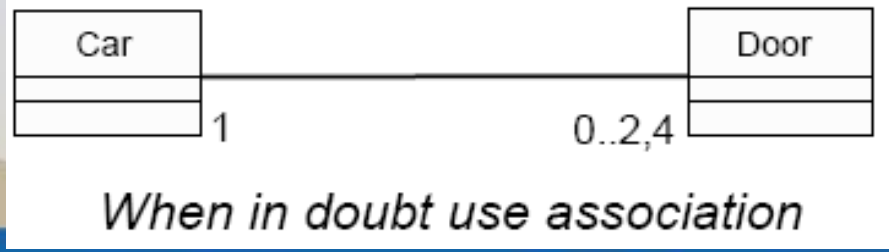
- Relacija između studenta i rasporeda (*Schedule*) je modelovana kao agregacija, jer je raspored nerazdvojivo vezan za određenog studenta
  - Raspored van konteksta studenta nema nikakvog smisla u sistemu Registrovanja na kurs (*Course Registration)*
  - Relacija od Rasporeda *(Schedule)* do Ponude kurseva (*CourseOffering)* je asocijacija jer kursevi mogu da se pojave na više rasporeda

Asocijacija ili agregacija?

- Ukoliko su dva objekta usko povezana relacijom celina-deo, onda je relacija agregacija
  - Ukoliko modelujete prodavnice kola, onda relacija između kola i vrata treba da bude modelovana kao agregacija, jer kola uvek dolaze sa vratima, a vrata se nikad samostalno ne prodaju

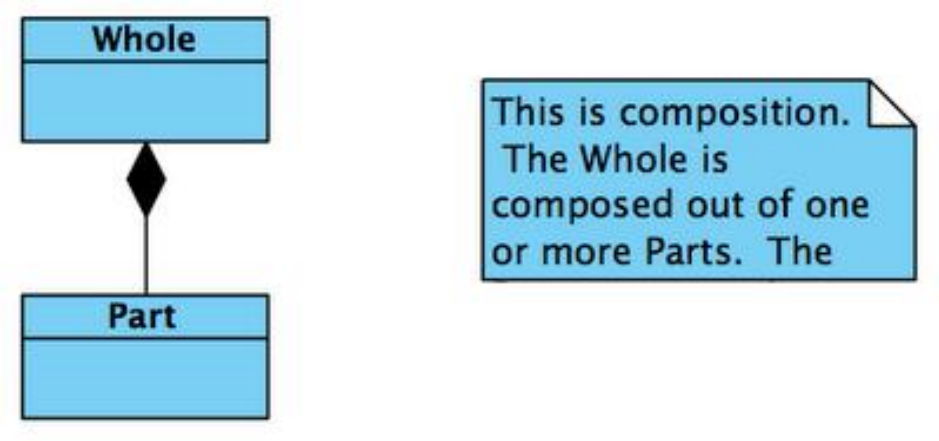


- Ukoliko su dva
  - Ukoliko modelujete prodavnicu auto delova, onda relacija između kola i vrata može da bude asocijacija, jer onda telo kola može da se pojavi nezavisno od vrata

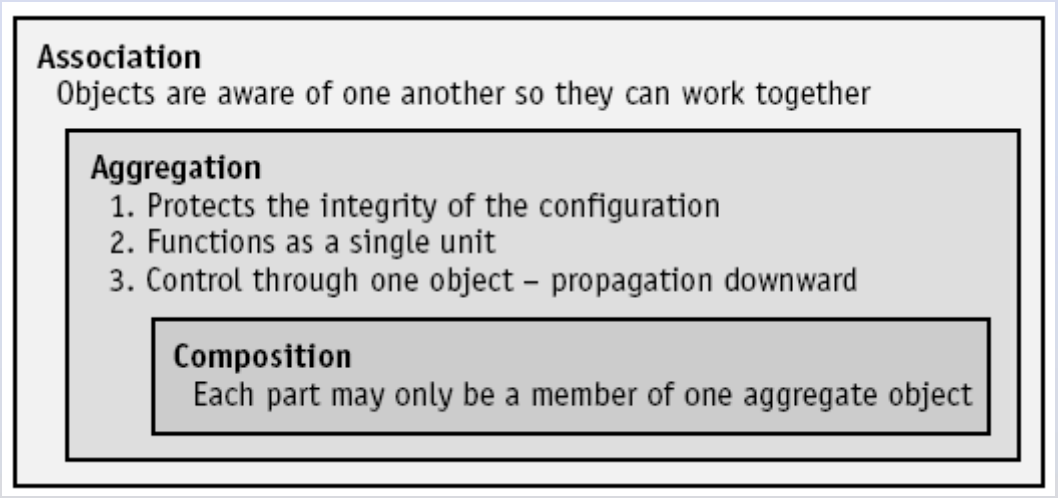


Kompozicija

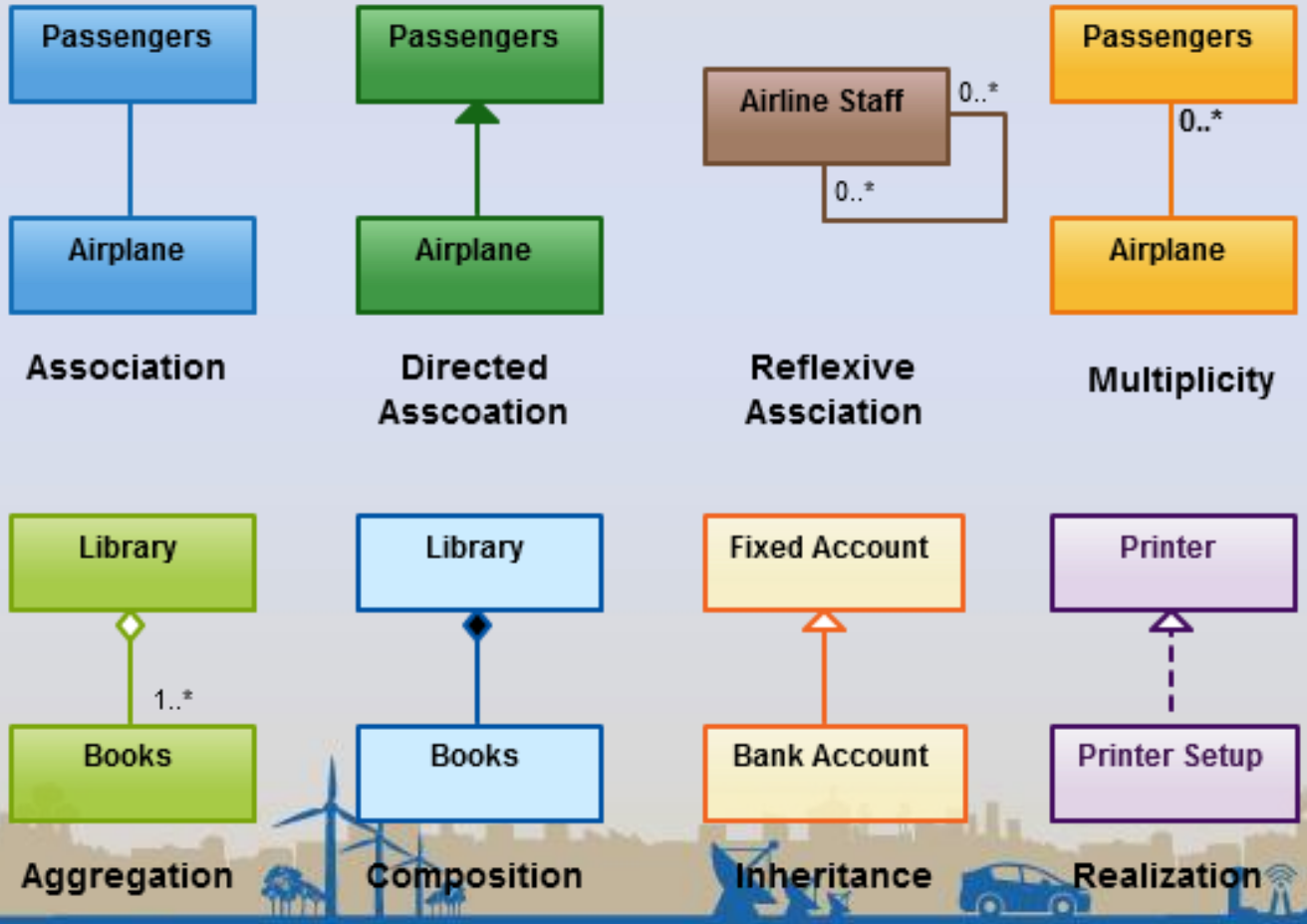
- Relacija kompozicije je slična relaciji agregacije samo što se kod uništavanja objekta uništava i klasa koja je deo tog objekta



Odnos relacija

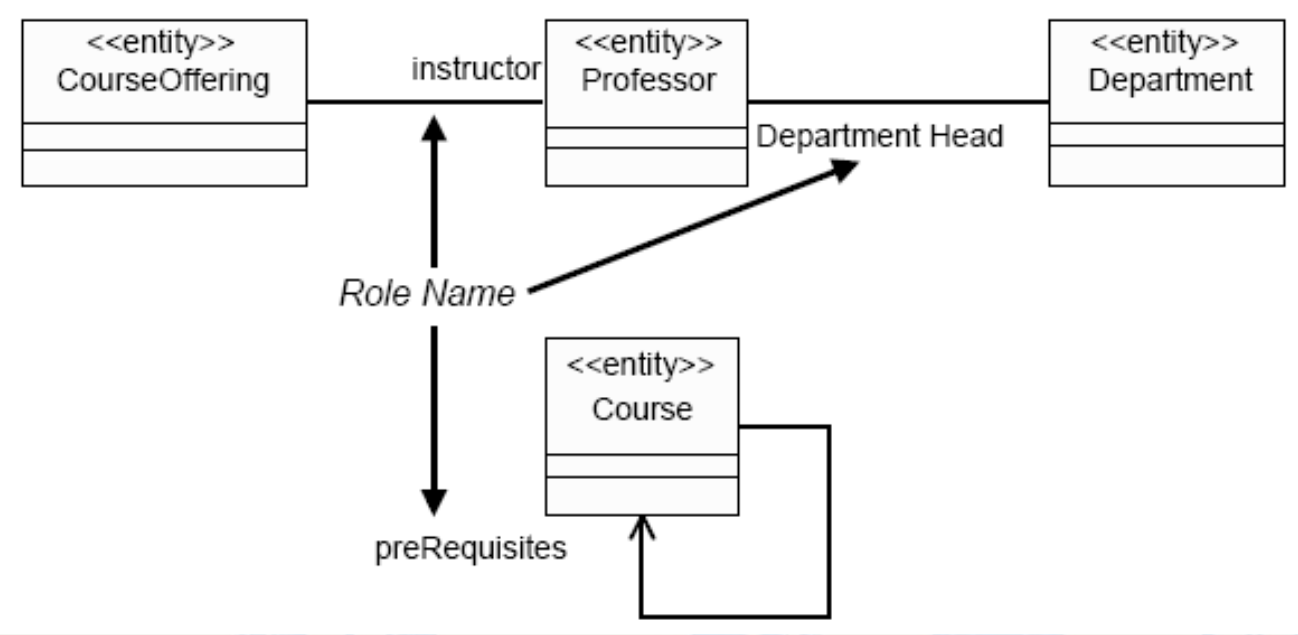


Moguće relacije na dijagramu klasa



Šta su role (uloge)?

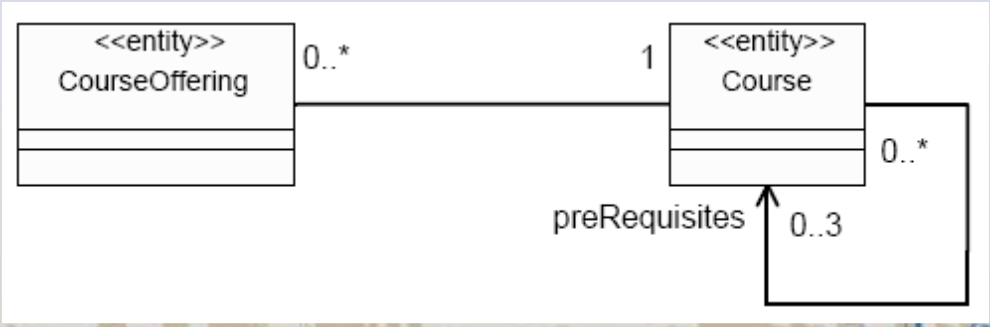
- Asocijacije sadrže neku ulogu u relaciji između klasa
  - Uloga ili rola se ispisuje na krajevima linije asocijacije
  - Uloga mora imati naziv (obično imenica)



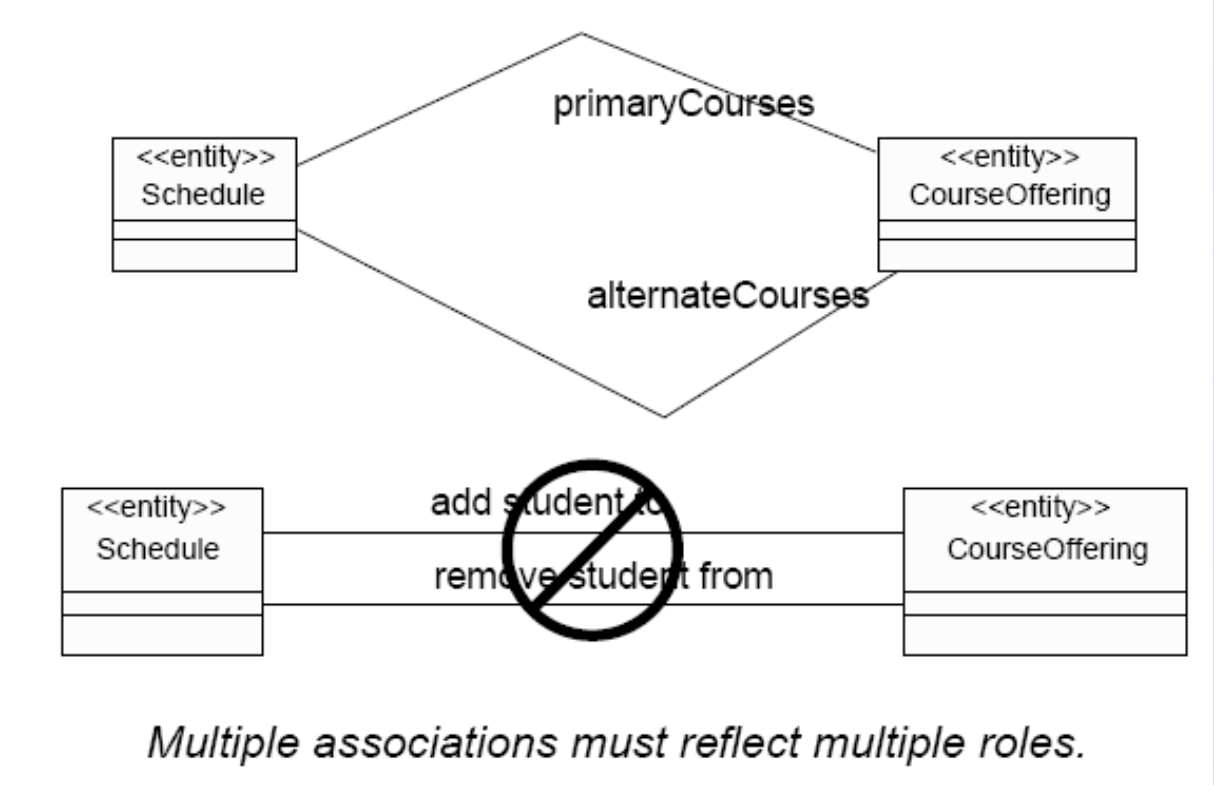
Multiplikativnost

- Za svaku rolu se navodi multiplikativnost klase
- Multiplikativnost je broj objekata klase koji se može pridružiti jednom objektu druge klase
- Beleži se na oba kraja relacije
- Multiplikativnost odgovara na dva pitanja:
  - Da li je asocijacija obavezna ili opciona? – ako je nula onda je takva asocijacija opciona
  - stanci koje mogu biti povezane ka drugoj instanci?

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or One (optional scalar role)	0..1
Specified Range	2..4
Multiple, Disjoint Ranges	2, 4..6



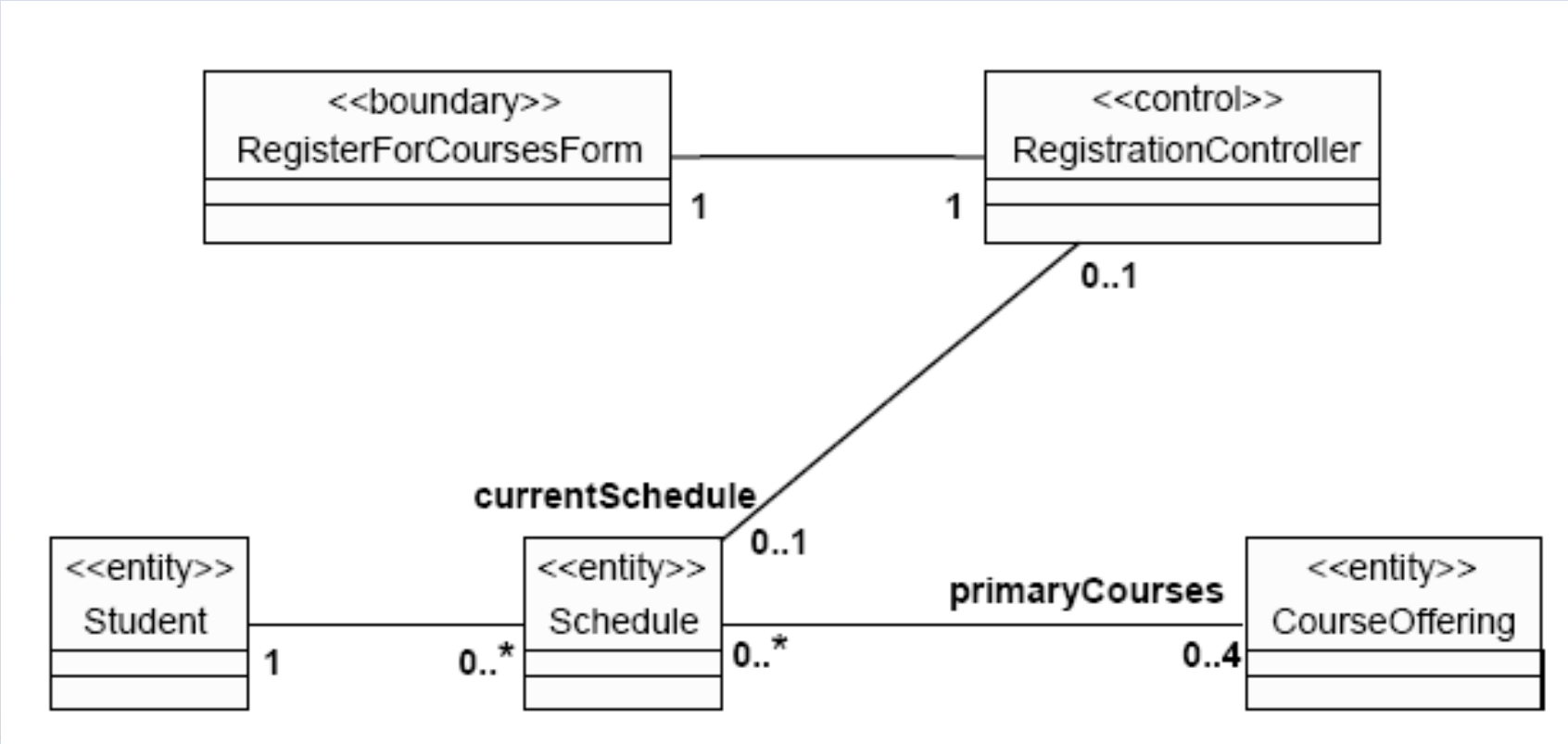
Primer višestruke asocijacije



Multiple associations must reflect multiple roles.

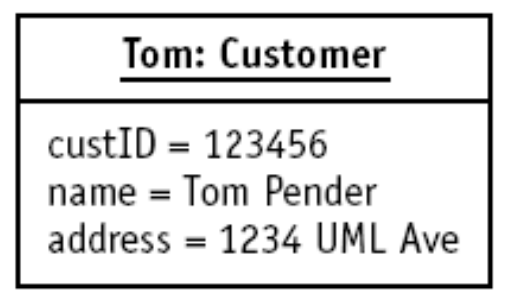
- Kada postoje višestruke asocijacije između dve klase, onda bi one trebalo da prikazuju različite uloge, a ne pozivanje različitih operacija

Primer pronalaženja relacija

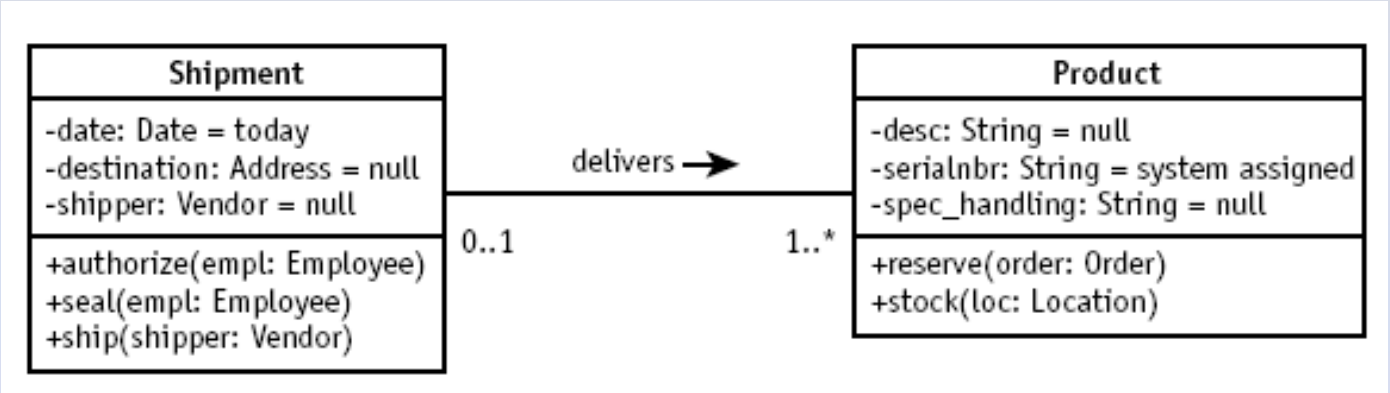


Objektni dijagram

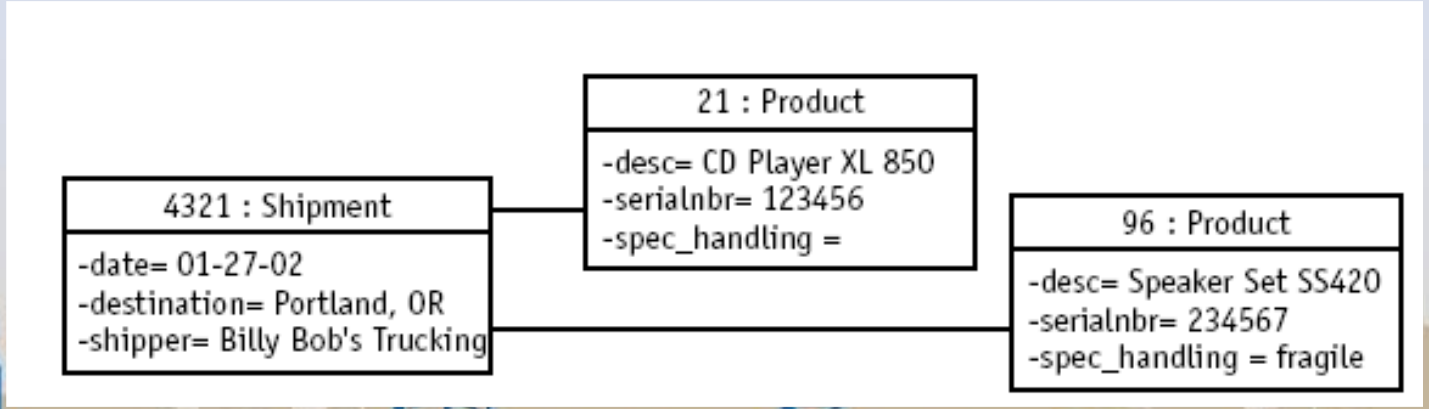
- Objektni dijagram je primarno alat za istraživanje i testiranje
- Može se koristiti i za razumevanje problema dokumentovanjem primera za određeni domen problema, kao i za verifikovanje tačnosti dijagrama klasa
- Objektni dijagram modeluje činjenice o određenim entitetima, dok dijagram klasa modeluje pravila za tipove entiteta
- UML notacija objekta:



Poređenje objektnog dijagrama i dijagrama klasa

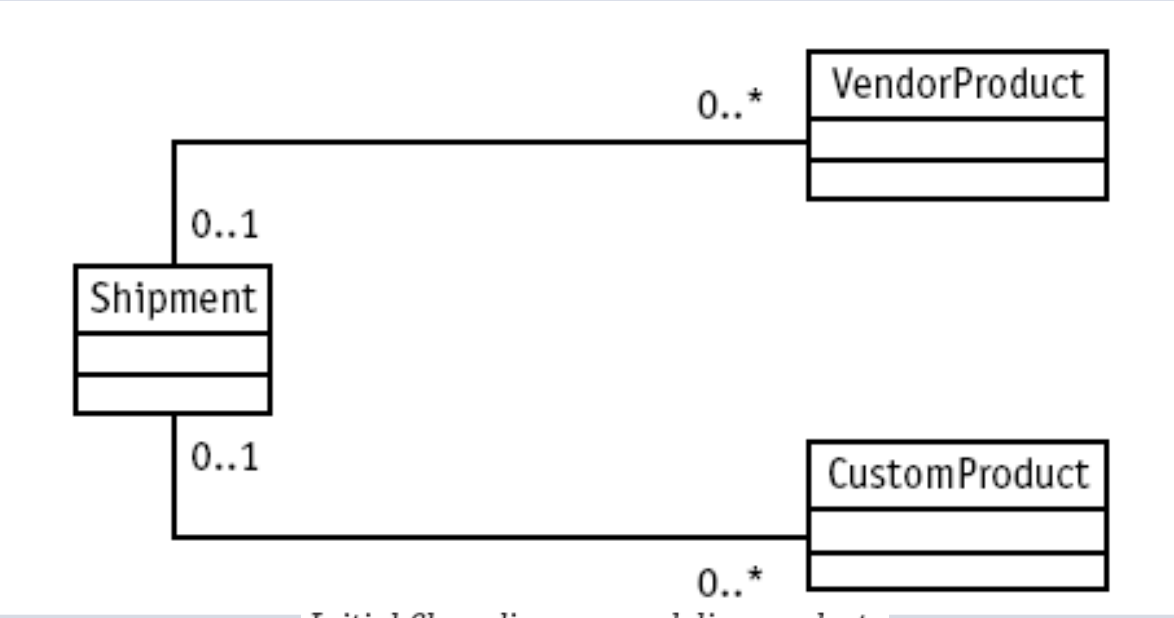


UML Class notation for the Shipment and Product



UML Object notation for a Shipment with two Products

Primena objektnog dijagrama za testiranje dijagrama klasa

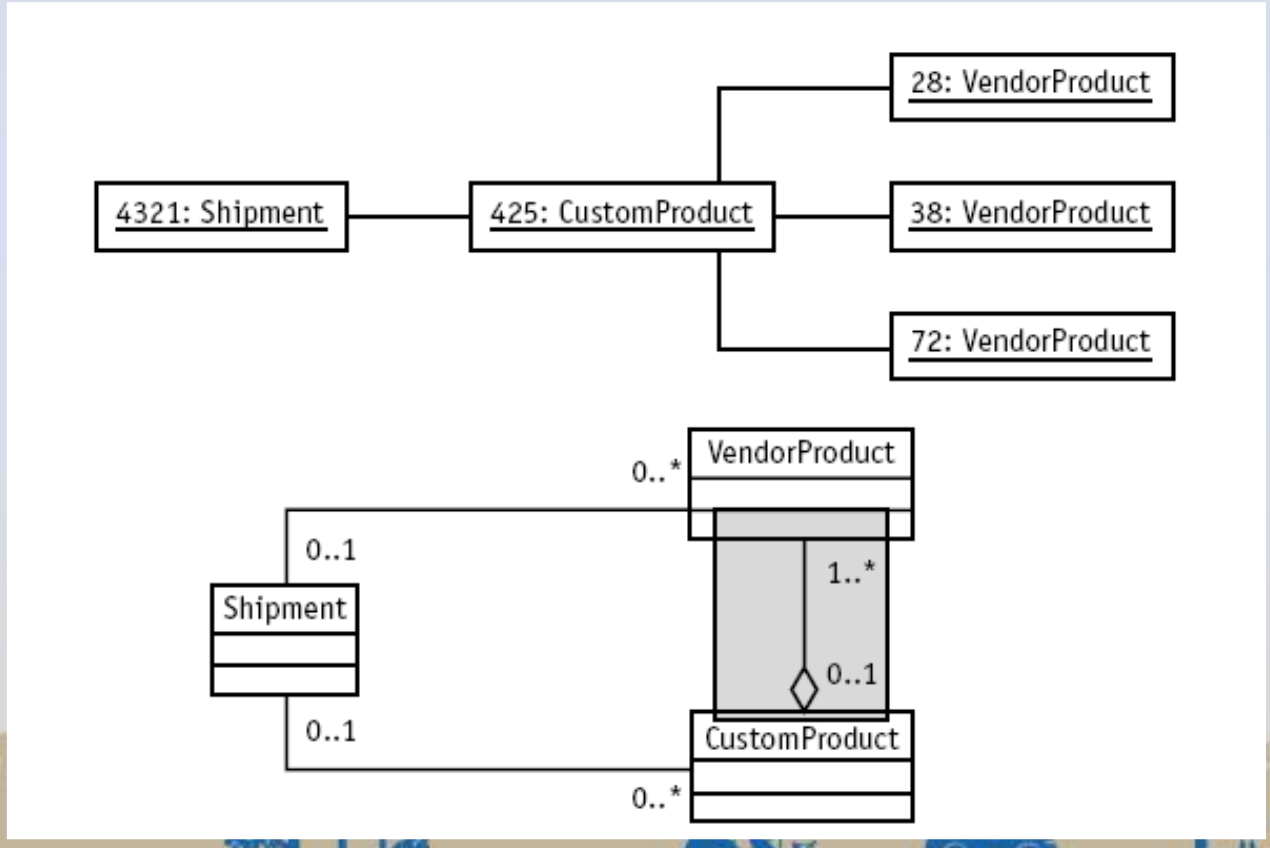


Initial Class diagram modeling products

- Svaka isporuka može da ima nula ili više Proizvoda dobavljača i nula ili više kastimiziranih proizvoda (proizvoda po narudžbi)
- Svaki tip proizvoda može, a i ne mora biti isporučen

Primena objektnog dijagrama za testiranje dijagrama klasa

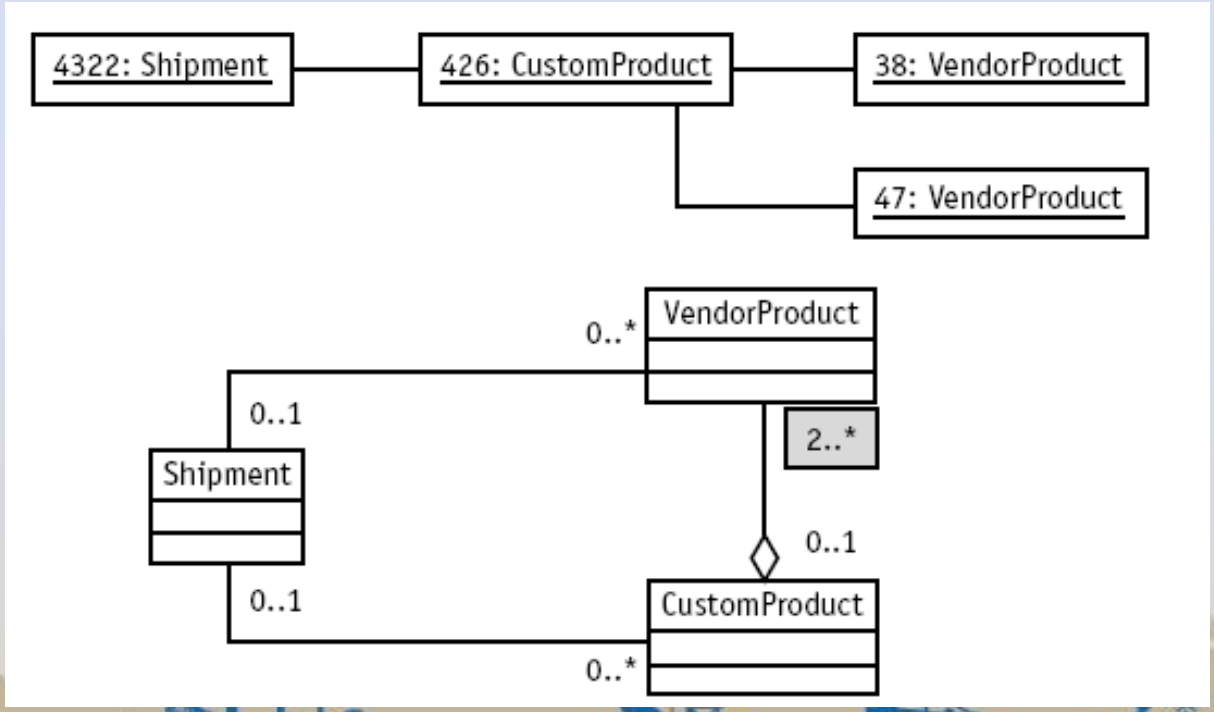
- Kastimiziran proizvod je sastavljen od proizvoda dobavljača. Proizvodi dobavljača 28, 38 i 72 kreiraju *custom* proizvod 425



The Object diagram (top) for Test Case 1 and the resulting updated Class diagram (bottom)

Primena objektnog dijagrama za testiranje dijagrama klasa - nastavak

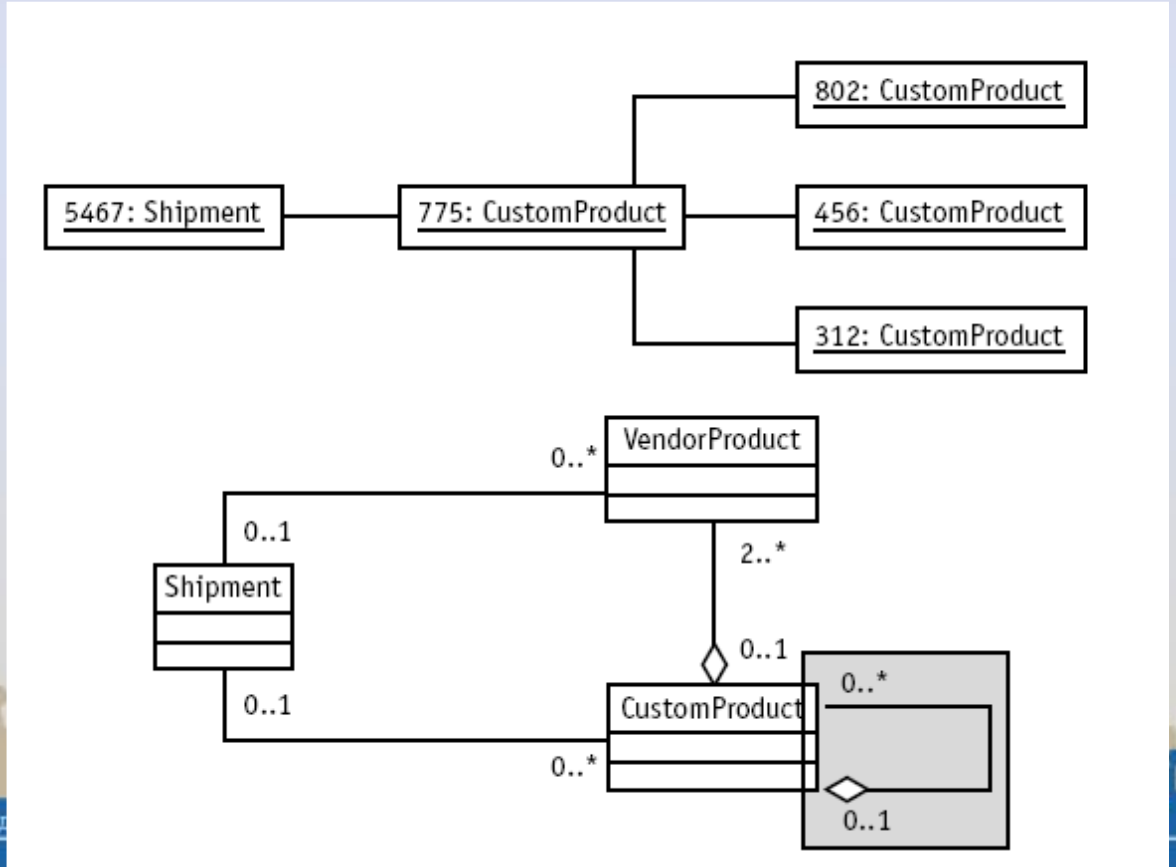
- Koji je najmanji broj objekata koji čini *Custom* proizvod? *Custom* proizvod mora da se sastoji od najmanje dva proizvoda dobavljača, inače ga drugačije nećemo moći razdvojiti



The Object diagram (top) and the resulting updated Class diagram (bottom)

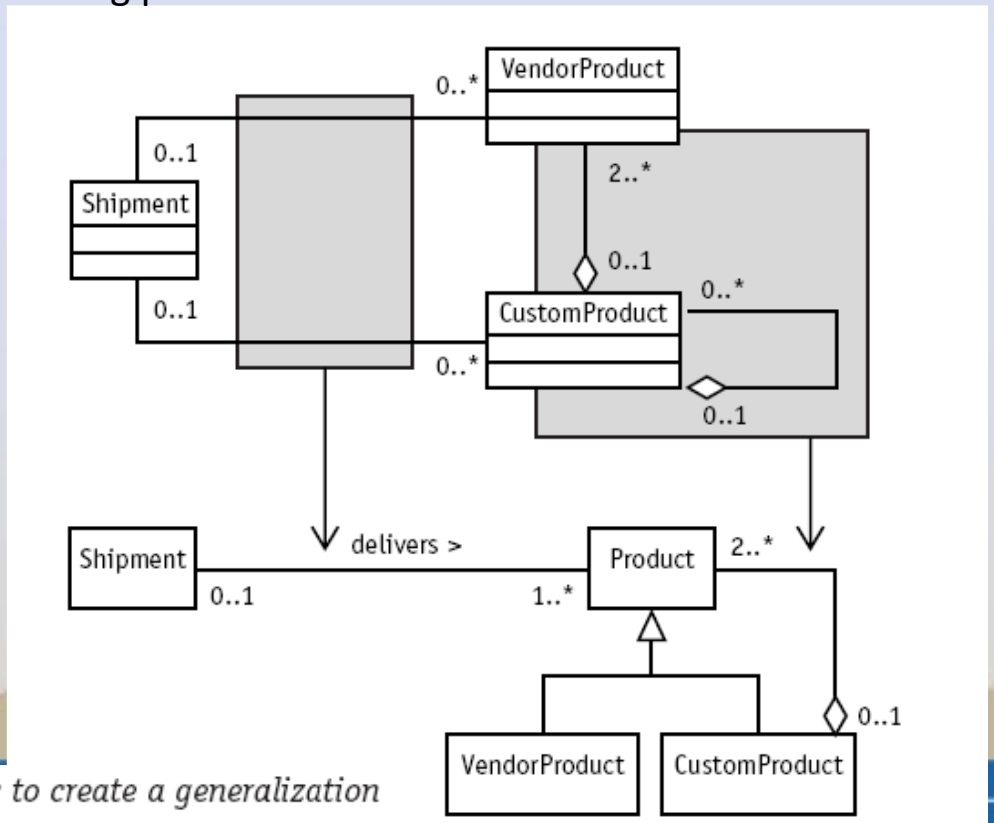
Primena objektnog dijagrama za testiranje dijagrama klasa - nastavak

- Da li postoji evidencija o tome da *Custom* proizvod može da bude ugrađen u drugi *Custom* proizvod?
- Mora da se podrži relacija agregacije između jednog i drugog *custom* proizvoda



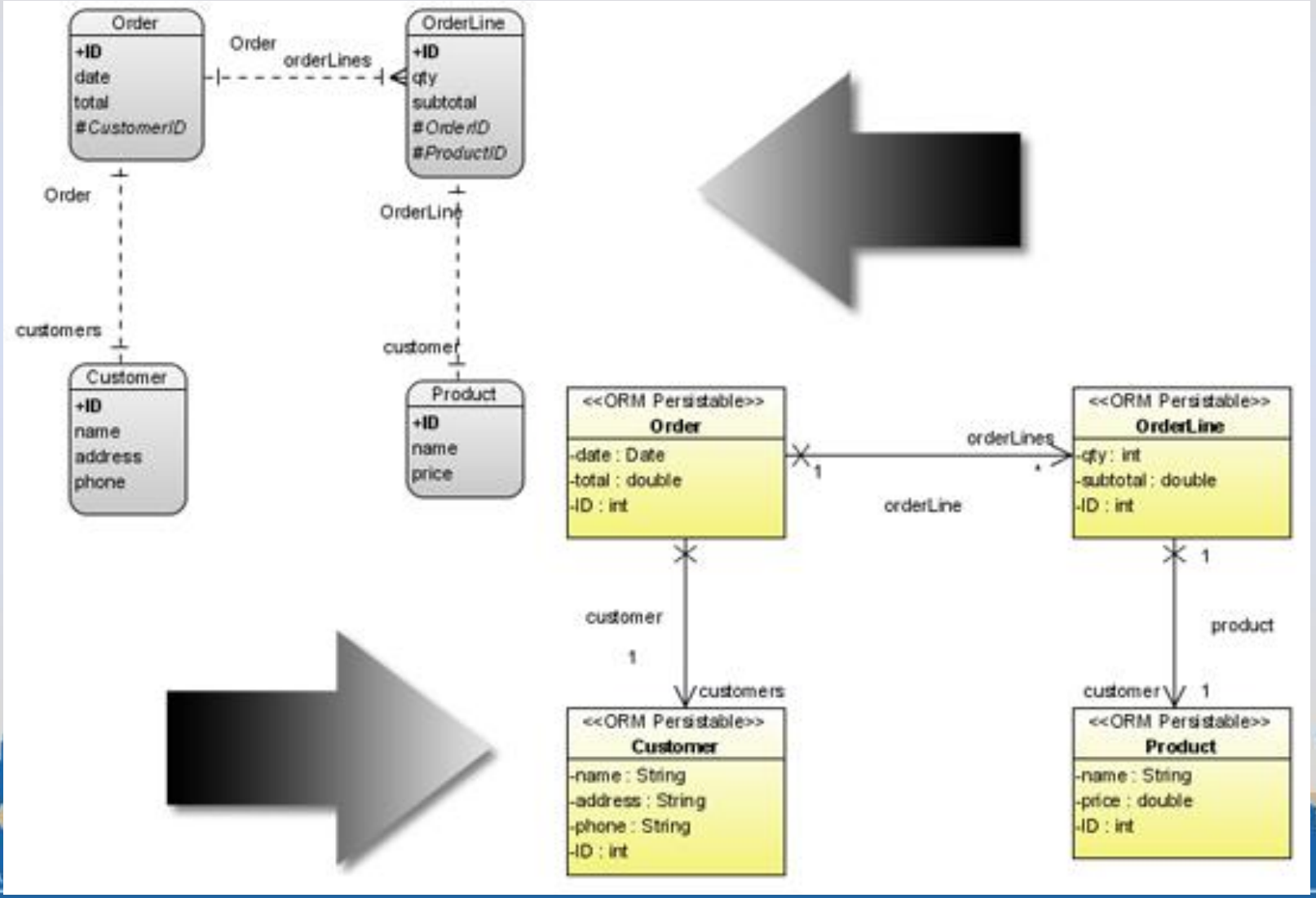
Primena objektnog dijagrama za testiranje dijagrama klasa - nastavak

- Da li postoje zajedničke karakteristike između objekata koji se koriste za konfigurisanje *custom* proizvoda?
- *Custom* proizvod i *vendor* proizvod mogu biti delovi *custom* prozvoda, tako da se generalizuju u proizvode što objašnjava da bilo koji tip proizvoda može da učestvuje u montaži kastimiziranog proizvoda

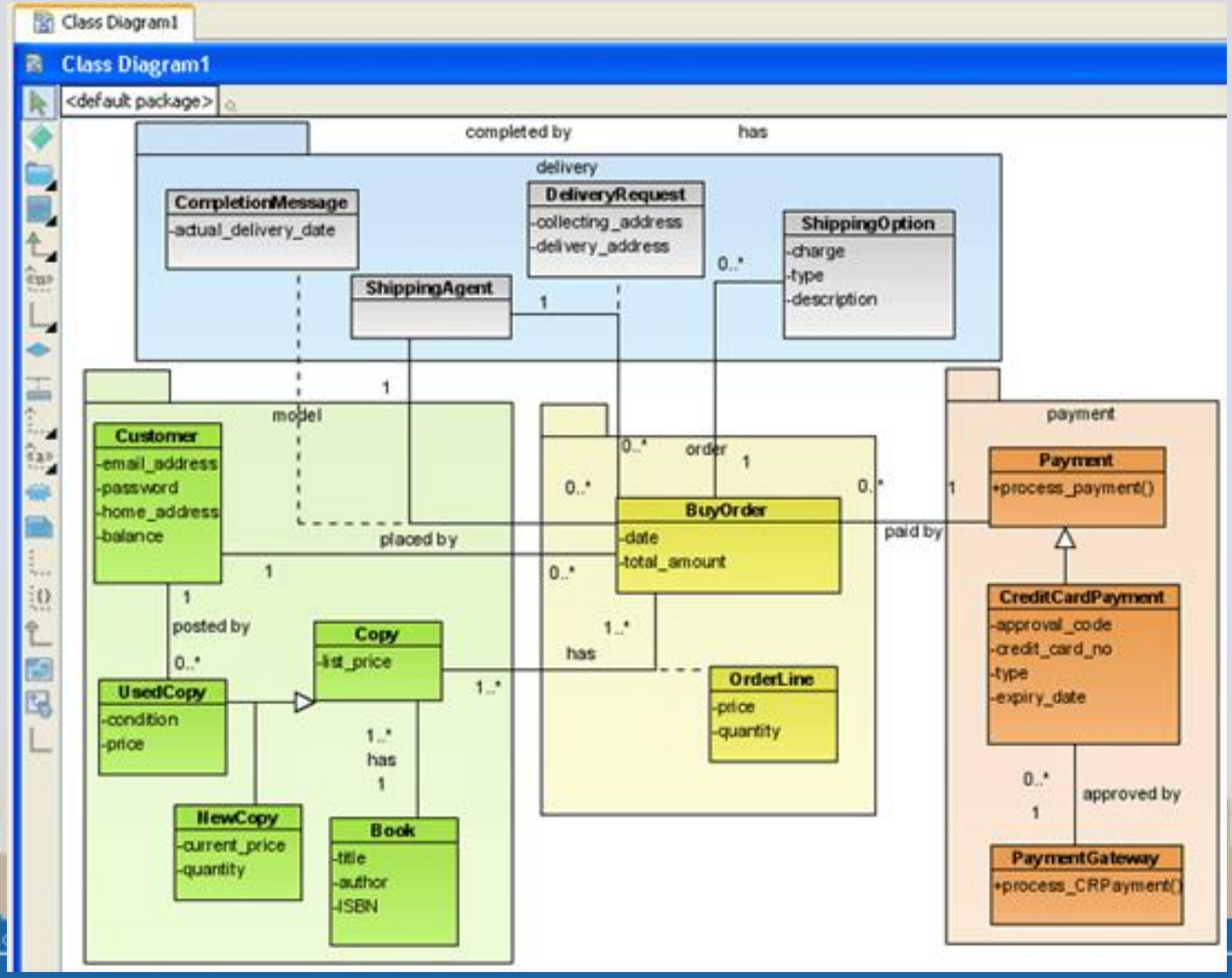


Identifying common properties to create a generalization

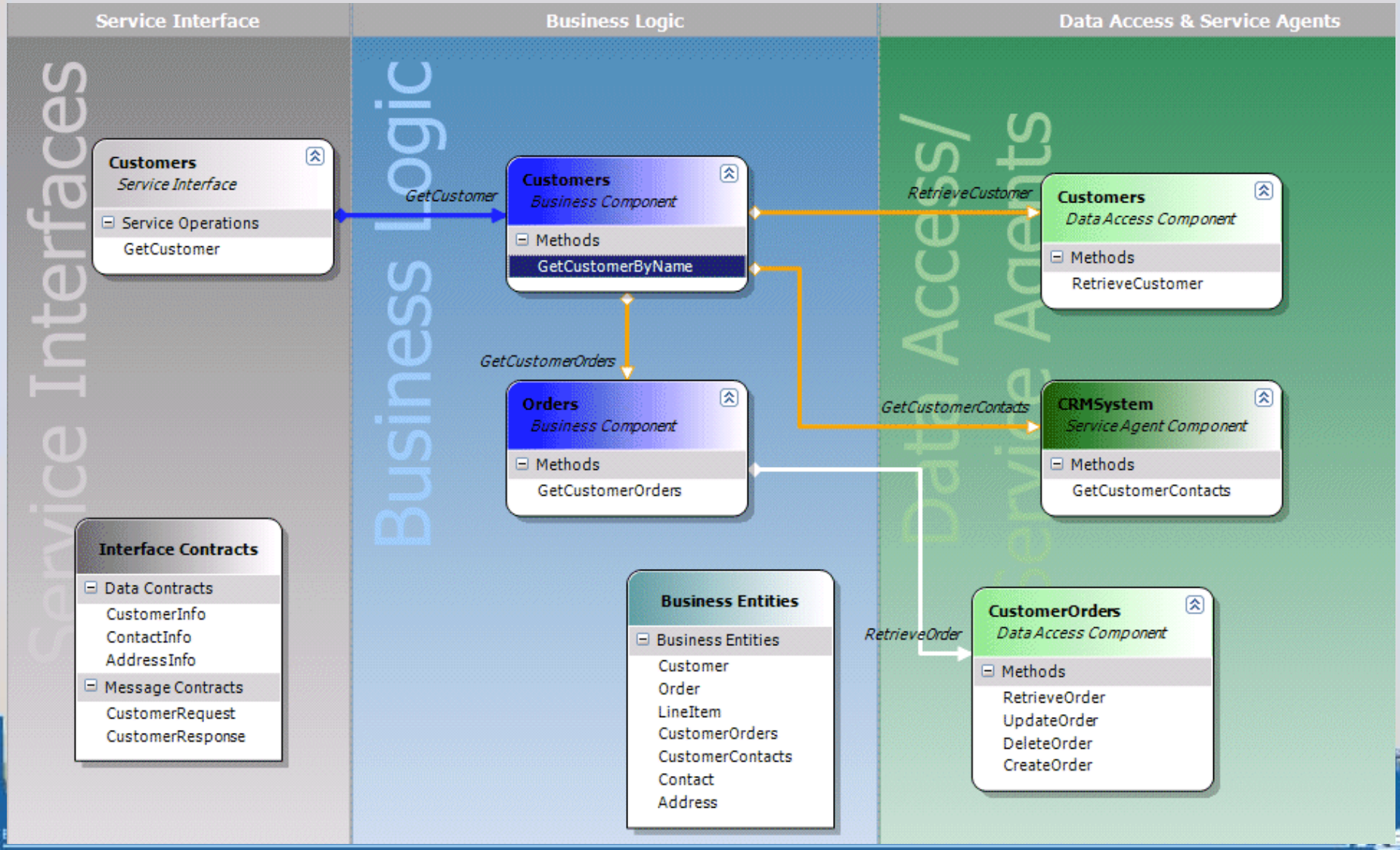
Poređenje modela podataka (ERD) i dijagrama klasa



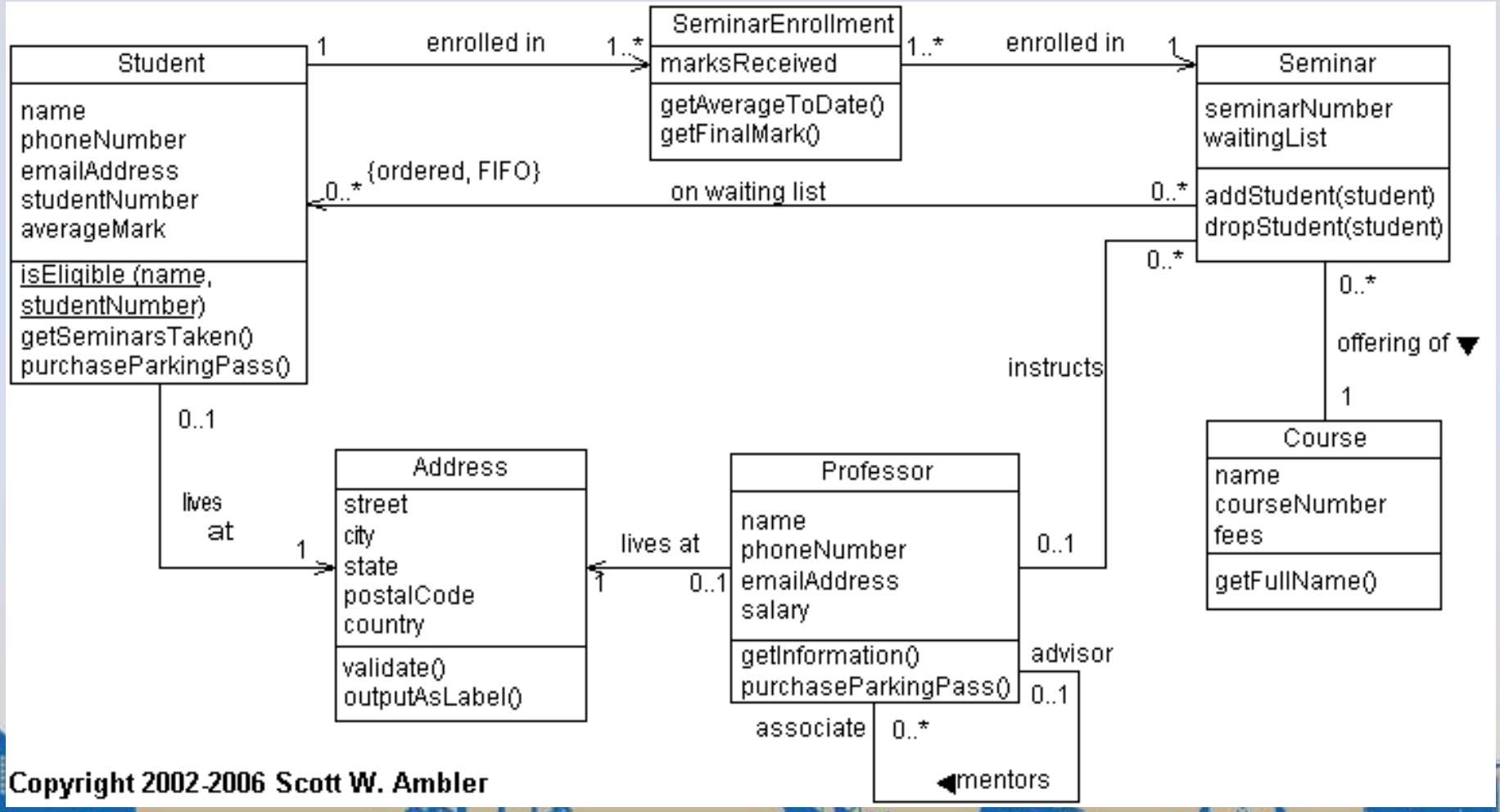
Primer grupisanja klasa u pakete



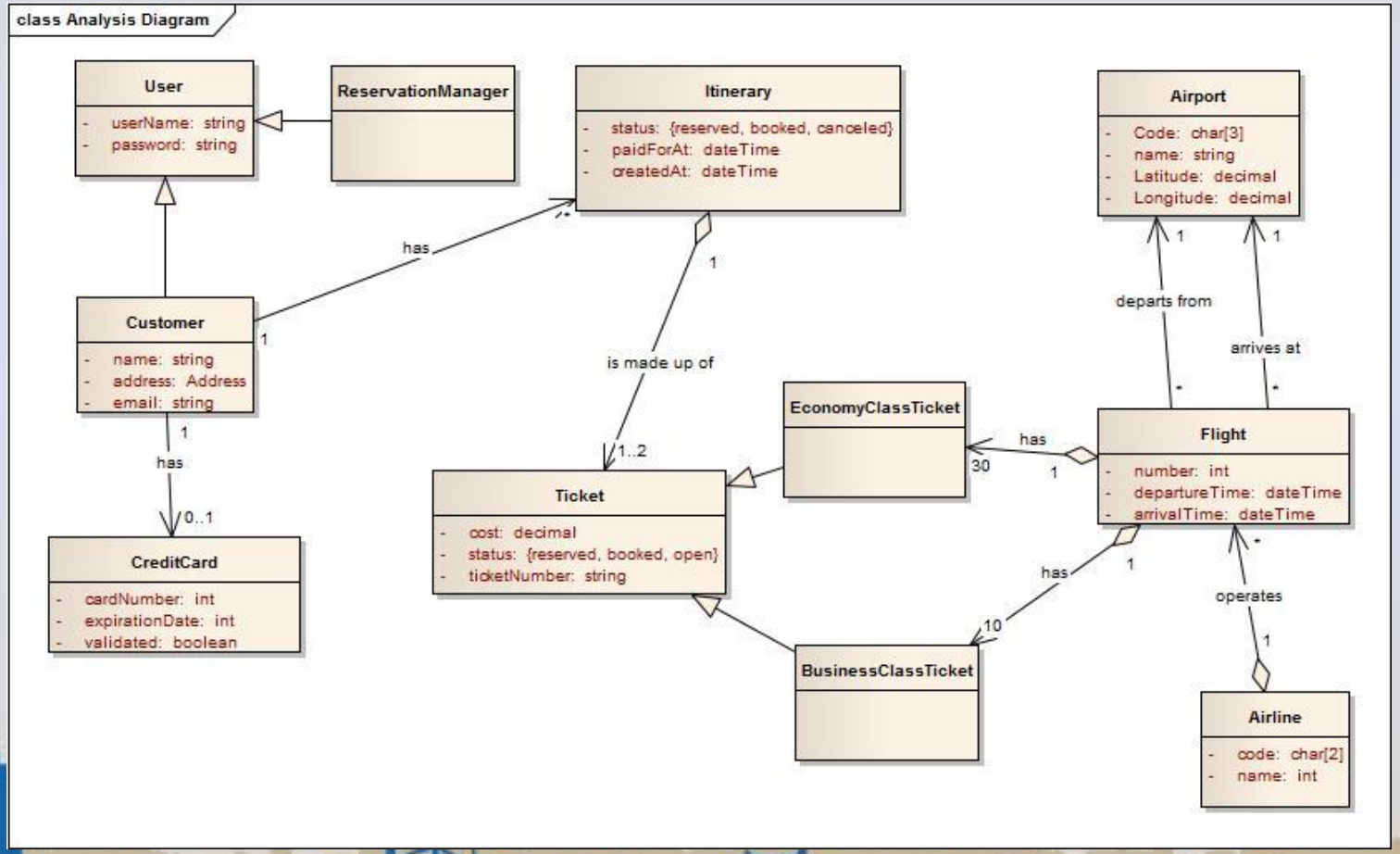
Dijagram klase prema slojevima arhitekture



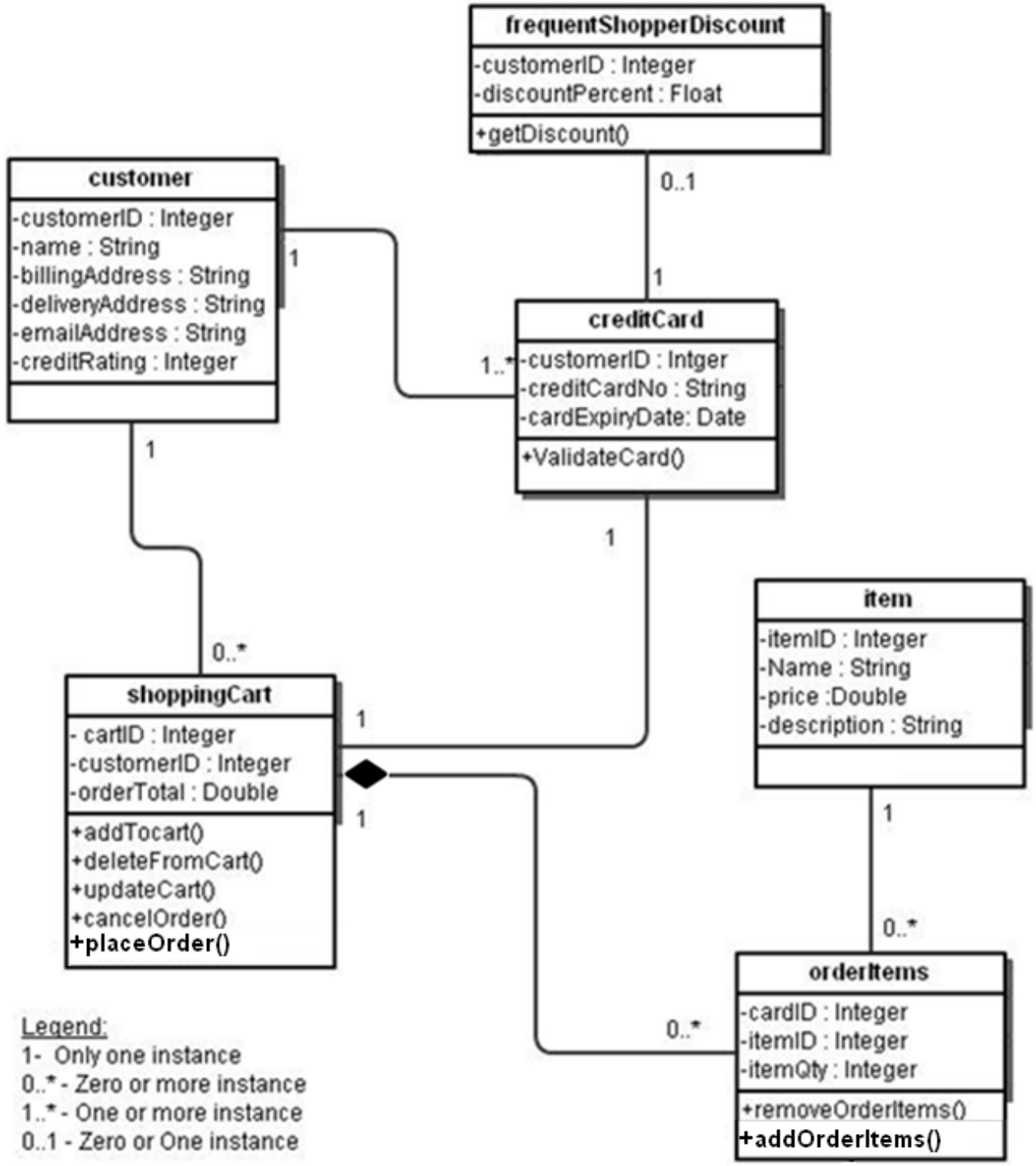
Konceptualni dijagram klasa - primer



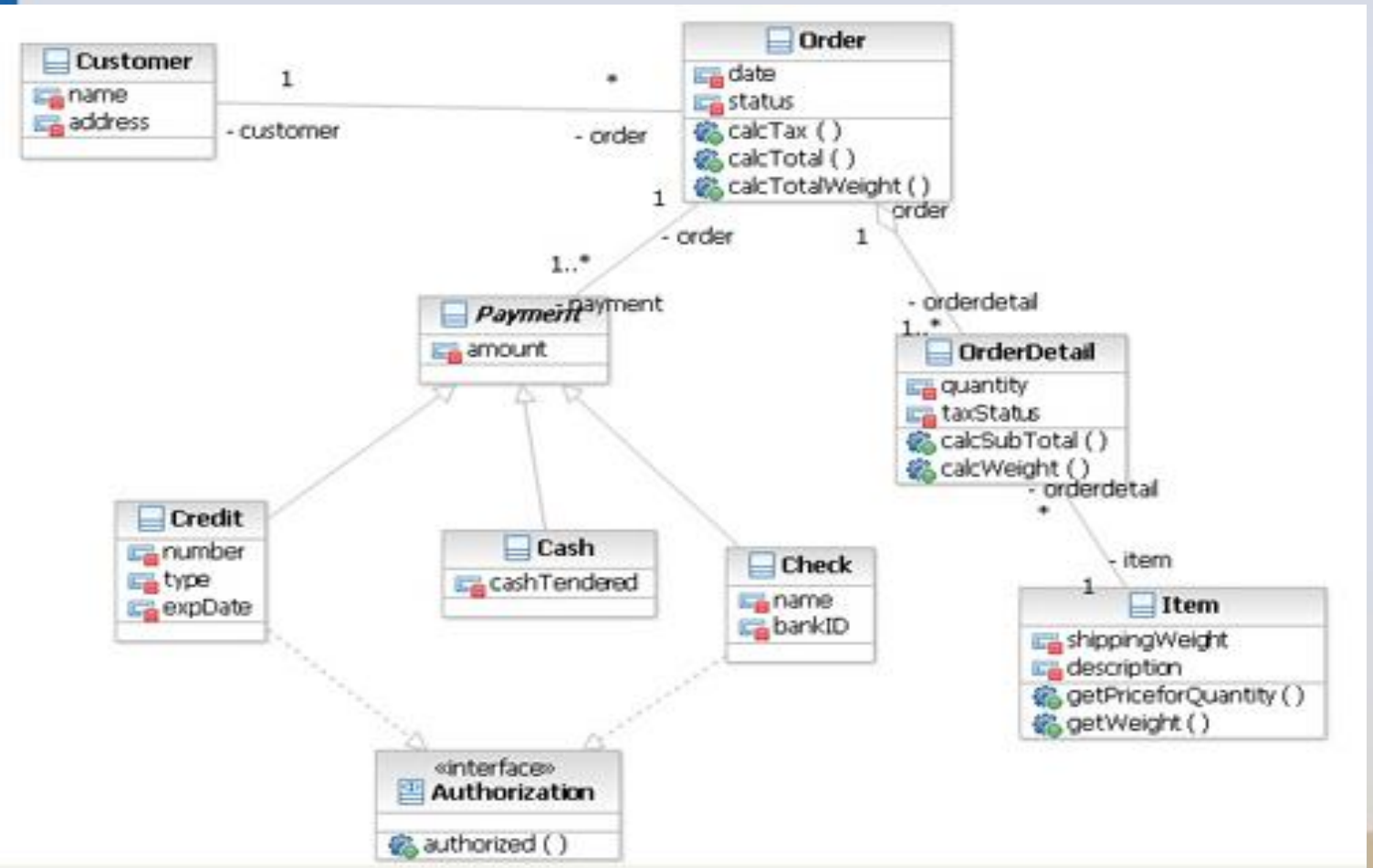
Primer: *Airline ticketing system*



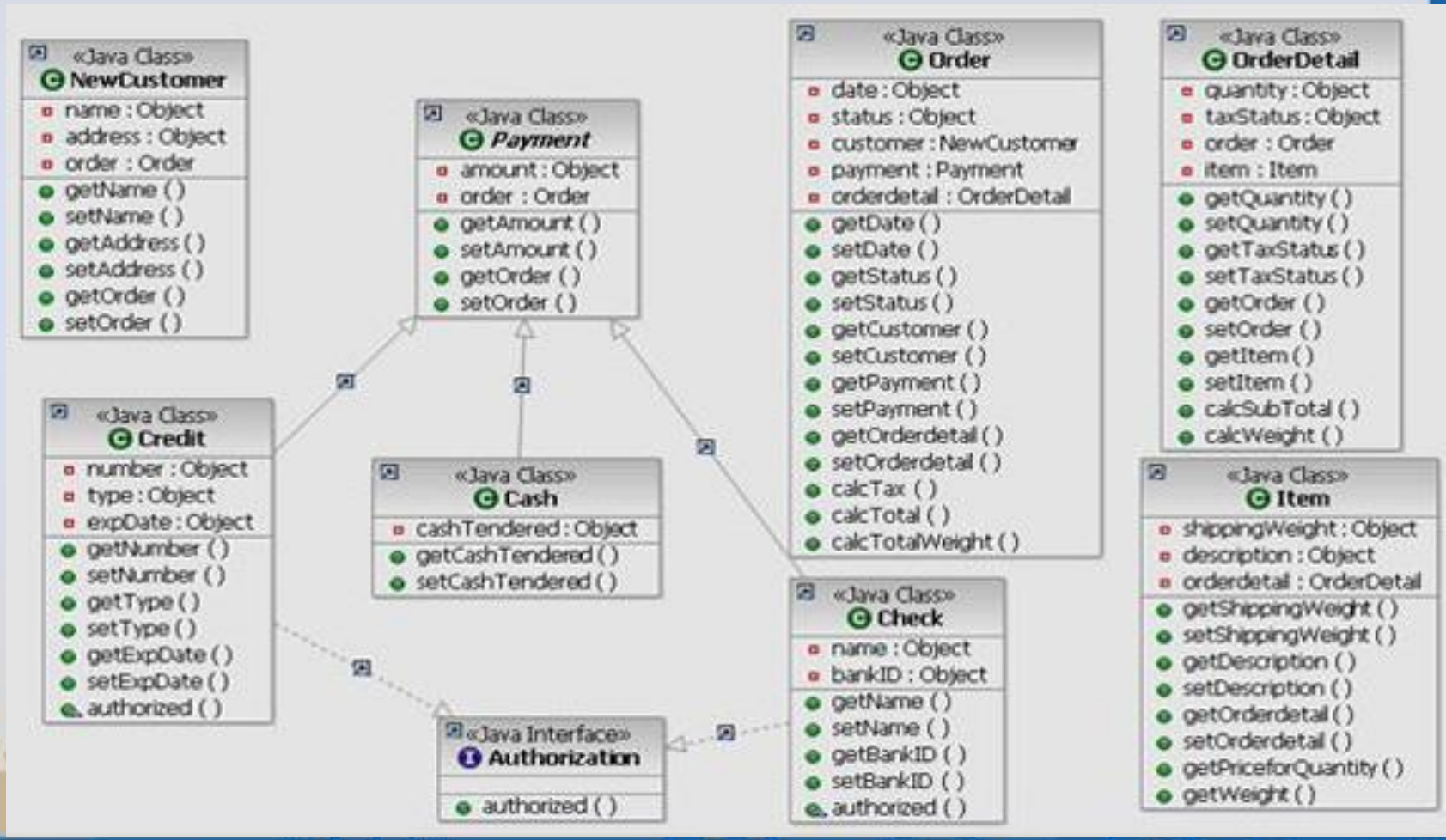
Primer:
Dijagram klase za šoping korpu



Poređenje konceptualnog i dijagrama klasa

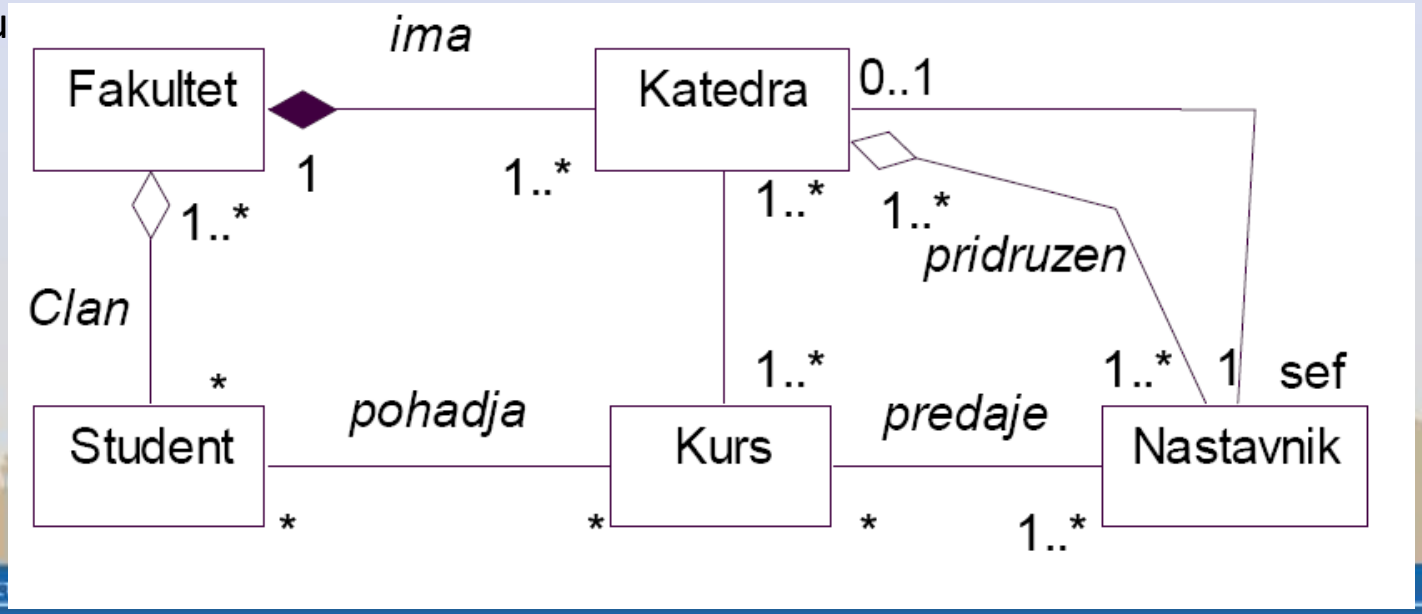


Dijagram klasa



Vežba 1: Nacrtati dijagram

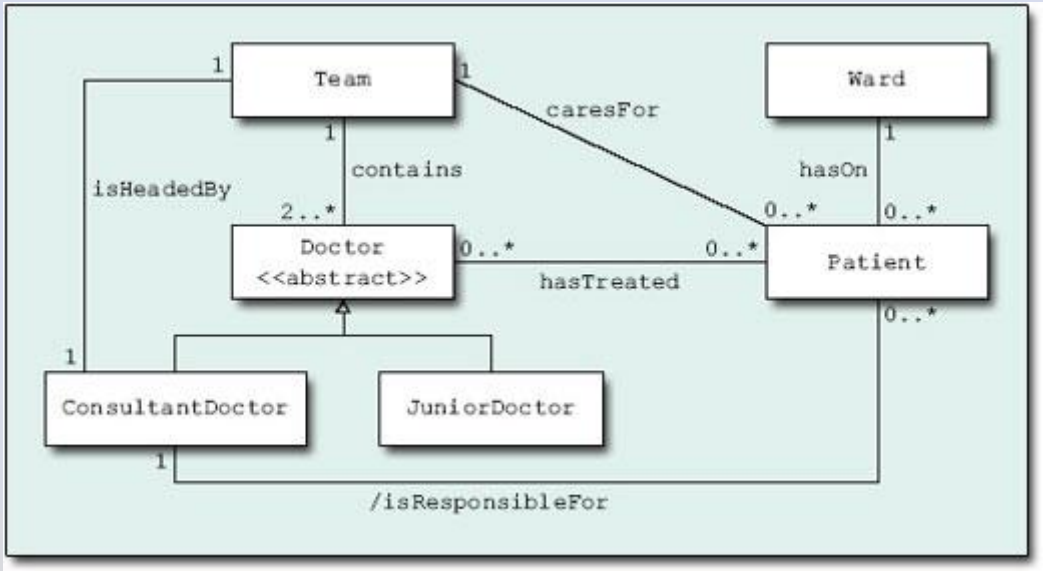
- Fakultet se sastoji od min. jedne i max. više Katedri
- Katedru čini min. jedan, a max. više nastavnika
- Samo jedan nastavnik može biti šef katedre
- Na katedri se drži min. jedan, a max. više kurseva
- Fakultet se sastoji od više studenata
- Stu



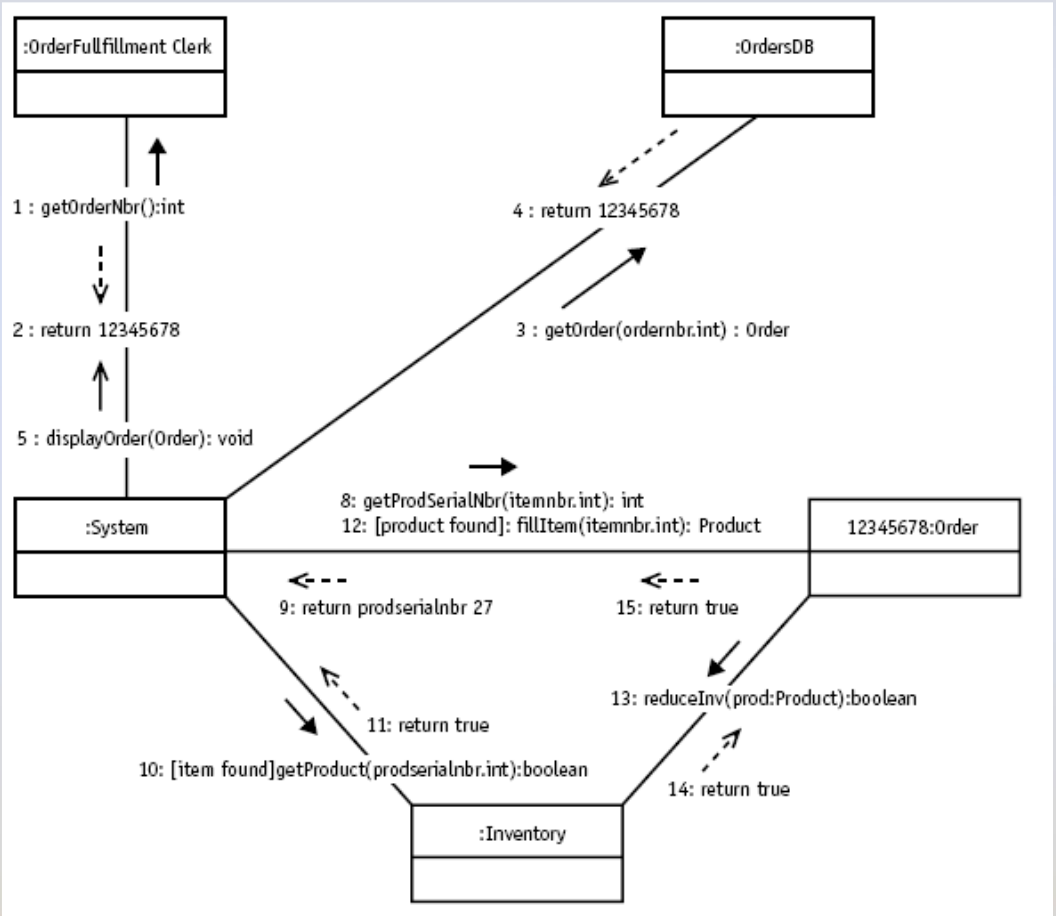
Vežba 2: Nacrtati konceptualni dijagram

- Jedan tim doktora (Team) se sastoji od minimum 2 ili više doktora
- Doktori mogu biti Junior doktor ili Doktor konsultant
- Jedan tim je predvođen jednim doktorom konsultantom
- Jedan tim brine o više pacijenata
- Doktor leči više pacijenata
- Doktor konsultant je odgovoran za više pacijenata
- U bolničkoj sobi može biti više pacijenata

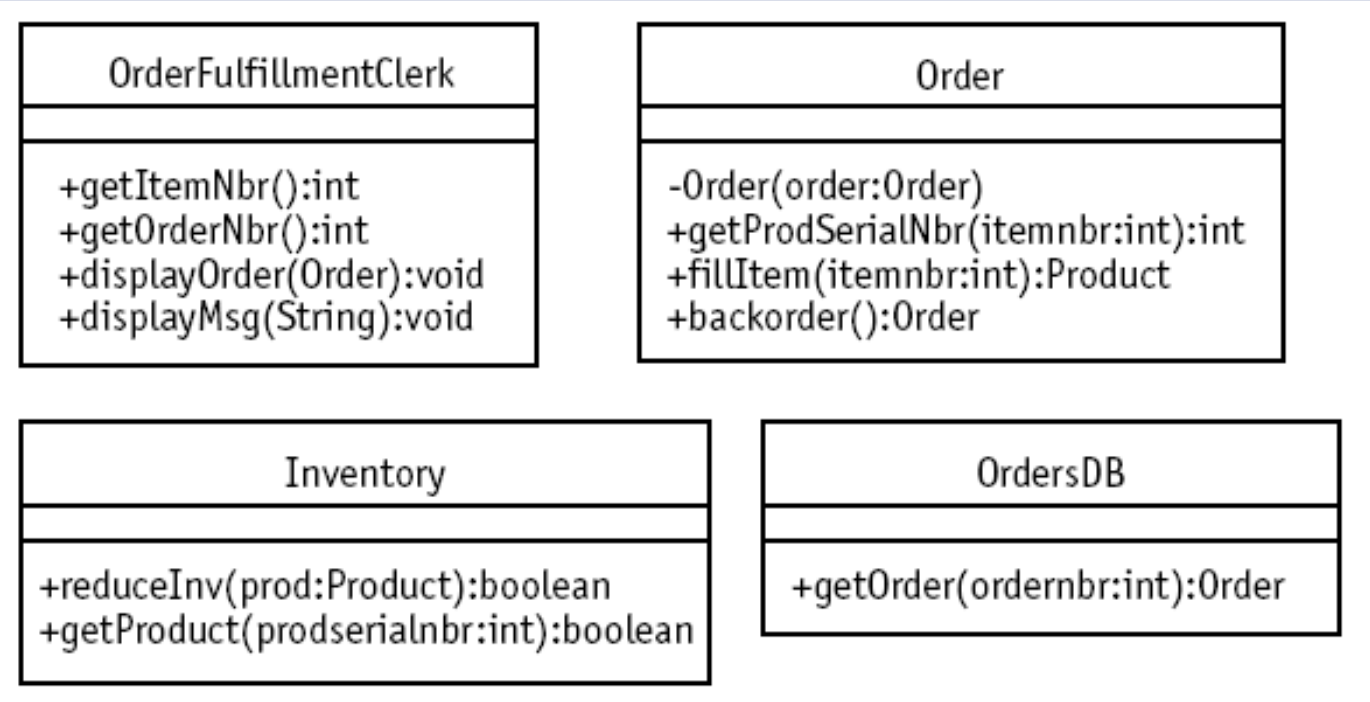
Konceptualni dijagram klasa



Vežba 3: Identifikovati klase i njihova ponašanja sa dijagrama komunikacije



Mapiranje elemenata dijagrama komunikacije u dijagram klase



Updated class operations

IBM Rational Software Architect

- http://publib.boulder.ibm.com/infocenter/rsahelp/v8/index.jsp?topic=/com.ibm.xtools.modeler.doc/topics/cclassd.html