

Oblikovanje programske potpore

2014./2015.

Uvod u programsko inženjerstvo

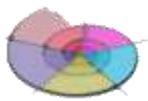


Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva
Zavod za elektroniku, mikroel., računalne i inteligentne sustave



Literatura

- Sommerville, I.: *Software Engineering*, Addison-Wesley, 2007.



Uvod u programsko inženjerstvo



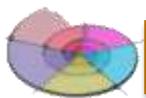
- Cilj: Upoznavanje s programskim inženjerstvom
- Prodiskutirati definicije i značaj PI
- Važnost etičkih i profesionalnih pitanja za programske inženjere



Programsko inženjerstvo



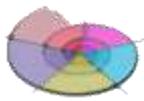
- većina složenih sustava (društvenih i tehničkih) upravljana je računalnim programima
- ovisnost ekonomije o računalnim programima
- potrošnja na računalne programe predstavlja značajan dio BDP-a svake zemlje
 - Potreba za računalnim programima je velika i raste.
- Veliki dio računalnih programa je neprihvatljive kakvoće, a situacija postaje sve gora.
- Kako bi se postigla prihvatljiva kakvoća moramo naučiti i rabiti “**inženjerski pristup**” u oblikovanju programske potpore.
- programsko inženjerstvo bavi se **teorijama, metodama i alatima** za profesionalno oblikovanje računalnih programa (programske potpore)



Karakteristike programske potpore



- Programska potpore je neopipljiva
 - teško je razumjeti napore procesa oblikovanja
- Jednostavna za reproducirati
 - bitno različita od svih drugih artefakata (do sada)
- Industrija programske potpore je izrazito radno intenzivna
 - teško je automatizirati u ključnim segmentima
- Lagano je unijeti izmjene
 - često i bez dubljeg razumijevanja posljedica
- Programska potpora se ne troši
 - ali se kvari ako se ne koristi kako je zamišljena ili ako postoje ugrađene pogreške



Cijena programske potpore



- Cijena programske potpore dominira u cijeni cjelokupnog sustava!!
- Cijena programske potpore sastoji se iz cijena:
 - *razvoja, oblikovanja, ispitivanja i održavanja.*
- Cijena održavanja programske potpore veća je od cijene razvoja i oblikovanja.
 - Kod dugo živućih sustava cijena održavanja je nekoliko puta veća od cijene razvoja.
- Cijena produkcije (kopiranja istovrsnih proizvoda) je zanemariva (to je bitno različito od svih drugih artifakta).
- Programsko inženjerstvo bavi se **cjenovno efikasnim** postupcima razvoja i oblikovanja programske potpore.

Pitanja programskog inženjerstva (1/2)

- Što je programska potpora?
- Što je programsko inženjerstvo?
- Koje je razlika između programskog inženjerstva i računarske znanosti?
- Koja je razlika između programskog inženjerstva i inženjerstva sustava?
- Što je proces programskog inženjerstva?
- Što je model procesa programskog inženjerstva?

Pitanja programskog inženjerstva (2/2)

- Koja je cijena programskog inženjerstva?
- Koje su metode programskog inženjerstva?
- Što je CASE (Computer-Aided Software Engineering)?
- Koje su značajke (atributi) dobre programske potpore?
- Koji su glavni izazovi i poteškoće u programskom inženjerstvu?
- Koje vrste projekata postoje u programskom inženjerstvu ?
- Što je profesionalna i etička odgovornost ?

Programska potpora

- računalni program, engl. **software**
- To je računalni program i pridružena dokumentacija: zahtjevi na sustav, modeli arhitekture i oblikovanja, korisnički priručnici.
- Računalni programi mogu se razvijati za
 - **ciljanog kupca** prema njegovim zahtjevima;
 - **skupinu kupaca** (npr. proračunske tablice za ručna računala upravljana određenim operacijskim sustavom);
 - **opće tržište** (engl. COTS - Commercial Off The Shelf).
- Programski produkti mogu se kreirati
 - oblikovanjem novih programa;
 - (re)konfiguracijom postojećih generičkih programa;
 - ponovnom uporabom postojećih programa (komponenti).



Programsko inženjerstvo



- Programsко inženjerstvo je inženjerska disciplina koja se bavi svim aspektima izgradnje programske potpore.
 - metodama i alatima za profesionalno oblikovanje i produkciju programske potpore uzimajući u obzir cjenovnu efikasnost
 - Programsko inženjerstvo
 - *sistematski i organiziran* pristup procesu izrade;
 - upotrebljavati *prikladne alate i tehnike* ovisno o *problemu* koji treba riješiti, *ograničenjima* u procesu izrade i postojećim *resursima*.
 - proces rješavanja problema kupaca i korisnika (ponekad je rješenje kupi, ne razvijaj)
- IEEE definicija:* the application of a *systematic, disciplined, quantifiable* approach to the development, operation, maintenance of software; that is, the application of engineering to software.



- Računarska znanost se bavi ***teorijom i temeljima***; programsko inženjerstvo se bavi ***praktičnim problemima razvoja, oblikovanja i isporuke korisnog računalnog programa***.
- Teorije u okviru računarske znanosti još uvijek su nedostatne za ukupnu podlogu programskom inženjerstvu
 - npr. razlika od fizike u elektrotehnici i elektronici.

- inženjerstvo sustava – engl. system engineering
 - Interdisciplinarno - sagledavanje cjeline, primjenjivo na sve složene sustave
- **Inženjerstvo računalnih sustava** (engl. computer system engineering) se bavi svim aspektima sustava zasnovanim na računalima (sklopolje, programska potpora, inženjerstvo procesa).
 - Programsко inženjerstvo je *dio* procesa usredotočeno na razvoj programske infrastrukture, upravljanje, primjenu i baze podataka u sustavu.
- Inženjeri sustava (računalnog) su uključeni u specificiranje sustava, oblikovanju arhitekture, integraciju i postavljanju u korisničko okruženje.



Proces programskog inženjerstva



- Skup aktivnosti čiji cilj je razvoj ili evolucija programskog produkta (podrazumijeva timski rad).
 - u primjeni velik broj procesa programskog inženjerstva
- Generičke aktivnosti u svim procesima programskog inženjerstva su:

Specifikacija	Temeljem analize zahtjeva odrediti što sustav treba činiti i koja su ograničenja u razvoju
Razvoj i oblikovanje	Izbor arhitekture i produkcija programskog sustava
Validacija	Provjera da li sustav čini ono što se od njega zahtijeva
Evolucija	promjene sukladno novim zahtjevima

- **Model** - pojednostavljeno predstavljanje procesa iz određene perspektive.
- Neki primjeri različitih perspektiva:
 - Tijek podataka (protok informacija);
 - Uloge i akcije (tko čini što);
 - Tijek posla (sekvenca aktivnosti).
- Generički modeli:
 - Vodopadni, spiralni, iterativni, komponentni, ...

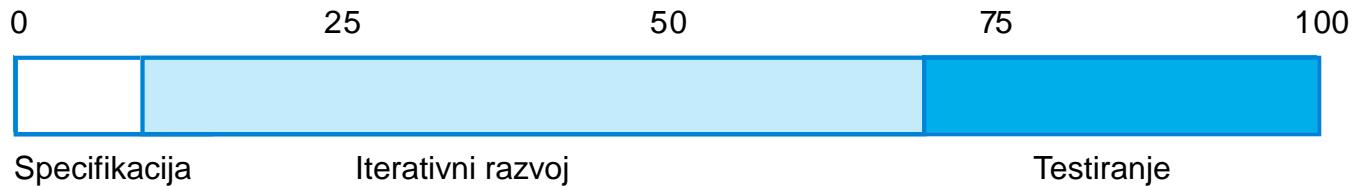
- Cijena programske potpore sadrži cijenu specifikacije, oblikovanja, ispitivanja i održavanja.
- Grubo 50% cijene je trošak razvoja, a 50% ispitivanja.
 - za jedinstvene programske produkte trošak evolucije (promjene, prilagodbe i održavanja) je veći od troška razvoja.
- Cijena uvelike ovisi o tipu sustava, zahtjevima, performansama i pouzdanosti.
- Razdioba troška ovisi o modelu procesa programskog inženjerstva.
 - Zahtjev: Korist mora biti veća od troška.
- Utjecaj okoline: Tržišno natjecanje u produkciji programske potpore (drugi mogu izgraditi brže i bolje).

Usporedba razdioba troškova

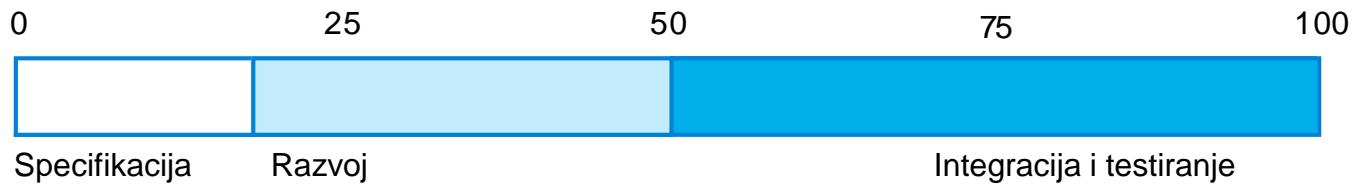
VODOPADNI eng. Waterfall model



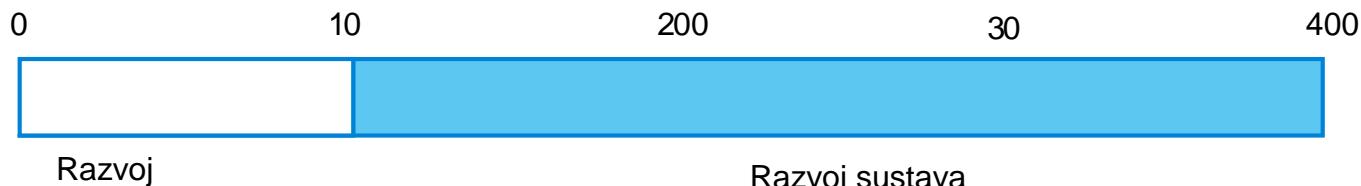
ITERATIVNI engl. Iterative development

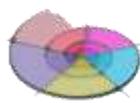


KOMPONENTNI engl. Component-based software engineering



DUGOVJEĆNI engl. Development and evolution costs for long-lifetime system





Metode programskog inženjerstva



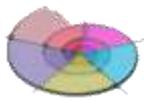
- strukturni(organizirani) pristup razvoju i oblikovanju programske potpore koji uključuje:
 - izbor modela sustava;
 - notaciju (označavanje);
 - pravila;
 - preporuke i naputci.
- Opisi modela
 - najčešće grafički
- Pravila
 - ograničenja primijenjena na modele sustava
- Preporuke
 - “dobra inženjerska praksa”
- Naputci o procesu
 - slijed aktivnosti

CASE (Computer-Aided Software Engineering)

- Programski produkti namijenjeni automatiziranoj podršci aktivnostima u procesu programskog inženjerstva.
- CASE sustavi često podupiru određenu metodu razvoja programskog produkta.
- **Viši CASE** (engl. *upper CASE*)
 - podupiru rane aktivnosti (npr. analiza zahtjeva, oblikovanje modela, ...)
- **Niži CASE** (engl. *lower CASE*)
 - podupiru kasnije aktivnosti (npr. programiranje – kodiranje, testiranje, ...)
- **Integrirani CASE**
 - podupire cijeloživotni ciklus produkta

Značajke dobrog programskog produkta

- Programska produkt mora *osigurati traženu funkcionalnost i performanse*, te mora biti **prihvatljiv korisniku, pouzdan** i mora se moći **održavati**.
- **Prihvatljivost** Acceptability (Usability)
 - razumljiv, koristan i kompatibilan s ostalim korisnikovim sustavima
- **Oslonljivost/Pouzdanost** Dependability
 - korisnik mora vjerovati u ispravnu funkcionalnost
- **Održavanje** Maintainability
 - evolucija produkta sukladno izmijenjenim i proširenim zahtjevima
- **Efikasnost/ekonomičnost**
 - Razumna uporaba resursa



Dionici programskog produkta

- dionici (engl. *Stakeholders*)

Kupac

Rješava problem uz prihvatljivu cijenu

Korisnik

Lagano učenje i prihvaćanje novog sustava, efikasnost u uporabi, pomaže da se posao obavi.



Programer/Osoba koja razvija

Jednostavno oblikovanje i održavanje, mogućnost ponovnog korištenja dijelova (engl. *reuse*).

Rukovoditelj razvoja

Prodati što više i zadovoljiti kupce uz minimalni trošak razvoja i održavanja.

Proturječja kakvoće programskog produkta



- Povećanje **efikasnosti** specijalizacijom čini sustav manje razumljivim i može smanjiti mogućnost održavanja ili ponovnog korištenja.
- Povećanje **lakoće korištenja** (npr. uključivanje uputa tijekom rada) može smanjiti efikasnost.
- Postavljanje razine kakvoće je ključna inženjerska aktivnost.
 - Produkt se razvija kako bi zadovoljio tu razinu. Nužni su kompromisi i optimizacija ograničenih resursa (npr. pouzdanost uz čvrsti proračun).
 - Dobra inženjerska praksa:
Avoid 'over-engineering' which wastes money.

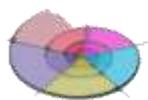
- **Heterogenost** engl. *Heterogeneity*
 - tehnike razvoja programske potpore za različite platforme okoline izvođenje.
- **Vrijeme isporuke** engl. *Delivery*
 - tehnike koje osiguravaju kratak interval od zamisli do stavljanja na tržište (engl time-to-market) uz prihvatljivu kakvoću
- **Povjerenje** engl. *Trust*
 - tehnike koje pokazuju da korisnici mogu imati povjerenja u programski sustav (npr. uporaba matematičkih, t.j. formalnih metoda).
- Postoji česta i iznenadna promjena zahtjeva kupaca.
- Programska potpora je složen sustav s velikim brojem detalja.



Vrste projekata



- Većina projekata je evolucijska i u svezi s održavanjem starijih sustava (engl. *legacy*).
- **Korektivni projekti** (otklanjanje pogrešaka).
- **Adaptivni projekti** (izmjene temeljem promjene operacijskog sustava, baza podataka, pravila i zakonskih odredaba, ...).
- **Unapređujući**, aditivni (dodavanje nove funkcionalnosti).
- **Re-inženjerstvo** (npr. unutarnje izmjene kako bi se olakšalo održavanje).
- **Sasvim novi projekti** (engl. *green-field*) – u manjini.
- **Integrativni** (oblikovanje novoga okruženja iz postojećih programskih komponenata i cjelina, engl. *framework*).



Profesionalna i etička odgovornost



- Programski inženjer se mora ponašati profesionalno korektno i etički odgovorno. Etičko ponašanje je više nego puko pridržavanje zakona.
- Povjerljivost prema poslodavcu i kupcu
 - Formalni ugovori CONFIDENTIALITY AND NON-DISCLOSURE AGREEMENT
- Kompetencije
 - prihvaćanje posla u okviru svojih kompetencija.
- Poštivanje prava intelektualnog vlasništva.
- Ne zloporabiti računalne sustave
 - npr. igranje za vrijeme rada, širenje virusa, spama ...
- ACM/IEEE Code of Ethics
 - http://www.unizg.fer.hr/ieee/upoznajte_ieee/kod_eticnost



Mi, članovi udruge IEEE, prepoznavajući **važnost tehnologija** i njihov utjecaj na kvalitetu života u cijelom svijetu, te prihvaćajući **osobnu obvezu prema vlastitom zanimanju, kolegama i zajednicama kojima služimo**, ovime se obvezujemo na najviše etičko i profesionalno ponašanje te smo odlučili:

- prihvati odgovornosti u donošenju odluka koje su u suglasju sa sigurnošću, zdravljem i općom dobrobiti i hitno otkriti čimbenike koji bi mogli ugroziti javnost ili okoliš;
 - izbjegavati stvarne ili uočene sukobe interesa kada god je to moguće i otkriti ih interesnim stranama ukoliko postoje;
 - biti pošteni i realni u iznošenju tvrdnji ili procjena temeljenih na dostupnim podacima;
 - odbiti mito i sve njegove oblike;
 - unaprijediti razumijevanje tehnologije, njenih prikladnih primjena i potencijalnih posljedica;
 - održavati i unaprjeđivati tehničku sposobnost i preuzeti tehnološke zadaće umjesto drugih samo zbog veće stručnosti i iskustva ili nakon potpunog iznošenja relevantnih ograničenja;
 - tražiti, prihvati i nuditi objektivnu kritiku tehničkog, u svrhu prihvatanja i ispravljanja pogrešaka te prikladno nagraditi tuđi doprinos;
 - jednako se ophoditi prema svim osobama bez obzira na rasu, vjeru, spol, invaliditet, godine ili nacionalnost;
 - izbjegavati ozljeđivanje drugi osoba, njihovog vlasništva, ugleda ili zaposlenja pogrešnim ili zlobnim radnjama;
 - pomagati kolegama i suradnicima u njihovom profesionalnom razvoju i podržavati ih da se pridržavaju ovog koda etičnosti.
- *Odobrio Upravni odbor IEEE Veljača 2006*

Sažetak

- Problemi razvoja programske podrške uzrokovana složenošću produkata i komunikacijskim problemima
- Dualnost programske podrške
- Osnovni cilj procesa razvoja programske podrške je osiguranje kvalitete
- Programsко инженерство је инженерска дисциплина која се бави примјеном систематског, дисциплинираног и мјерљивог приступа у свим fazama života programske podrške

Oblikovanje programske potpore

2014./2015.

Inženjerstvo zahtjeva u oblikovanju programske potpore



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

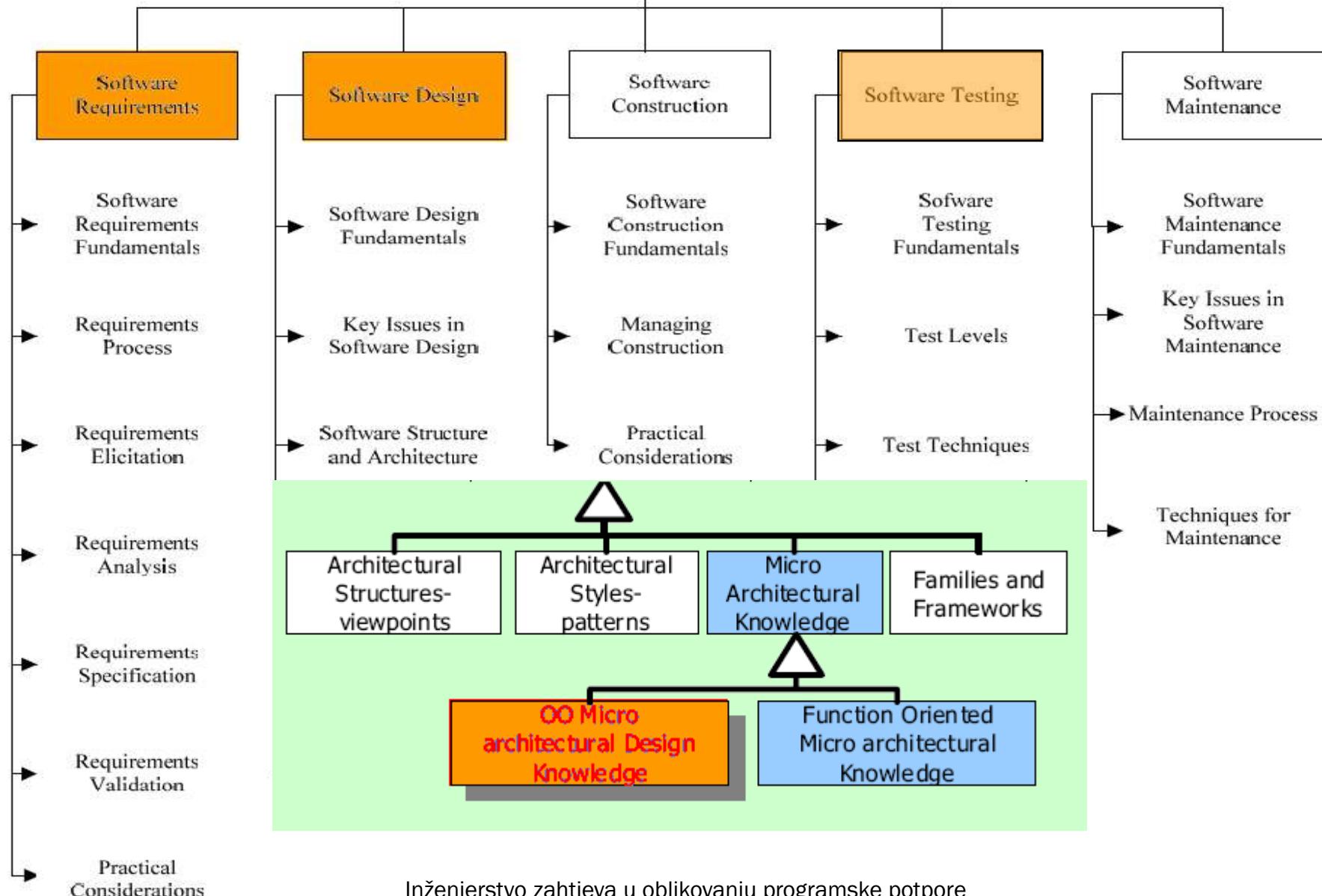


Osvrt

- Sadržaj programskog inženjerstva

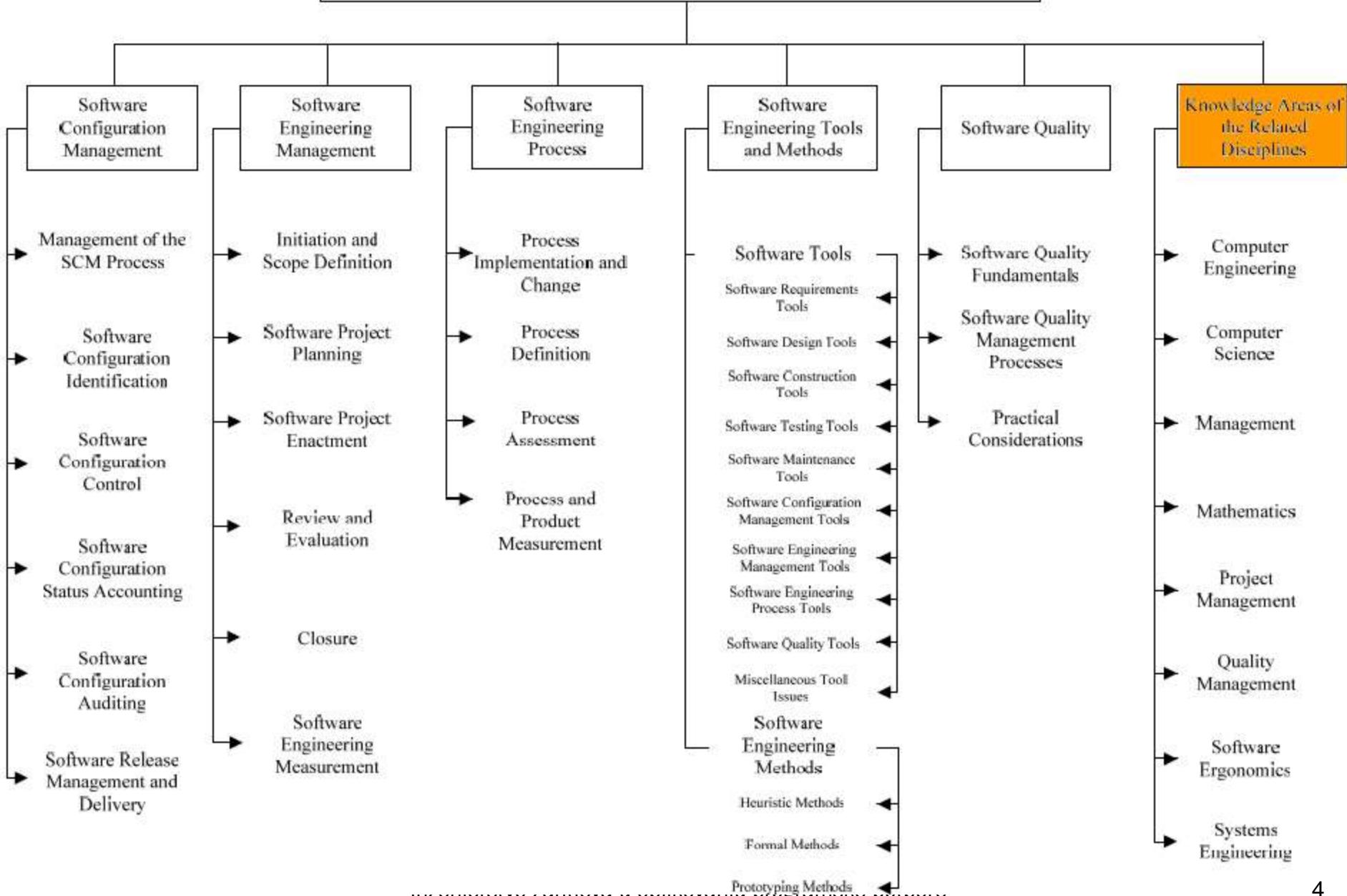
Guide to the Software Engineering Body of Knowledge

2004 Version



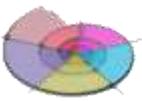
Guide to the Software Engineering Body of Knowledge

(2004 Version)



Tema

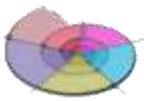
- Inženjerstvo zahtjeva (engl. *requirements engineering*)
- Koncept korisničkih zahtjeva i zahtjeva sustava
- Tipovi zahtjeva
- Organizacija dokumenata zahtijeva programske potpore
- Proces programskog inženjerstva
- Temeljne inženjerske aktivnosti **generiranja i dokumentiranja zahtjeva.**
- Tehnike izlučivanja i analize zahtjeva.*



Literatura



- Sommerville, I., ***Software engineering***, 8th ed., Addison-Wesley, 2007.
- Grady Booch, James Rumbaugh, Ivar Jacobson: ***Unified Modeling Language User Guide***, 2nd Edition, 2005
- Simon Bennett, John Skelton, Ken Lunn: ***Schaum's Outline of UML***, Second Edition, 2005
- WWW
- Rational Software Architect
 - <http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=/com.ibm.xtools.modeler.doc/topics/cextend.html>



Inženjerstvo zahtjeva



- engl. *requirements engineering, requirements gathering, requirements capture, requirements specification*
- To je postupak pronalaženja, analiziranja, strukturiranja, dokumentiranja i provjere **korisnički zahtijevanih usluga** sustava, te **ograničenja u uporabi**.
 - fokus: utvrđivanje ciljeva, funkcija i ograničenja sklopljiva i programske podrške.
- Zahtjevi sami za sebe su opisi usluga sustava i ograničenja koja se generiraju tijekom procesa inženjerstva zahtjeva.
 - klasifikacija zahtjeva obzirom na razinu detalja i sadržaj.

Klasifikacija zahtjeva prema razini detalja



■ *Korisnički zahtjevi*

- specifikacija visoke razine apstrakcije - u okviru ponude za izradu programskog produkta
- pišu se u prirodnom jeziku i grafičkim dijagramima.
 - Klijenti (rukovoditelji – manager)
 - Krajnji korisnici sustava
 - Klijenti (inženjeri)
 - Rukovoditelji za pisanje ugovora
 - Specijalisti za oblikovanje sustava (arhitekti)

■ *Zahtjevi sustava*

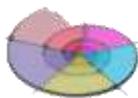
- vrlo detaljna specifikacija - uobičajeno nakon prihvatanja ponude, a prije sklapanja ugovora
- pišu se strukturiranim prirodnim jezikom, posebnim jezicima za oblikovanje sustava, dijagramima i matematičkom notacijom.
 - Krajnji korisnici sustava
 - Klijenti (inženjeri)
 - Specijalisti za oblikovanje sustava (arhitekti)
 - Specijalisti za razvoj programske potpore

■ *Specifikacija programske potpore*

- najdetaljniji opis koji objedinjuje korisničke i zahtjeve sustava
 - Klijenti (inženjeri) – možda ?
 - Specijalisti za oblikovanje sustava (arhitekti)
 - Specijalisti za razvoj programske potpore

KORISNIČKI ZAHTJEVI

- Odnosi se na korisničke zahtjeve i zahtjeve sustava
- **Funkcionalni zahtjevi** (engl. *Functional requirement*)
 - izjave o uslugama koje sustav mora pružati, kako će sustav reagirati na određeni ulazni poticaj, te kao bi se sustav trebao ponašati u određenim situacijama.
 - specifikacija rezultata rada: engl. "system shall do <requirement>"
- **Nefunkcionalni zahtjevi** (engl. *Non-functional requirement*)
 - ograničenja u uslugama i funkcijama, kao što su vremenska ograničenja, (ne)usvojeni standardi, ograničenja u procesu razvoja i oblikovanja i sl.
 - naglasak na karakteristikama: engl. "system shall be <requirement>"
- **Zahtjevi domene primjene**
 - zahtjevi (funkcionalni i nefunkcionalni) koji proizlaze iz domene primjene sustava kao i oni koji karakteriziraju tu domenu.



Primjer funkcionalnih zahtjeva: Sustav LIBSYS

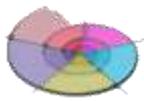


- LIBSYS: Hipotetski knjižničarski sustav koji pruža jedinstveno sučelje prema bazama članaka u različitim knjižnicama. Korisnik može pretraživati, spremati, ispisivati članke za osobnu potrebu.

Izvor: Sommerville, I., Software engineering

1. Korisnik mora moći pretraživati početni skup baza ili podskup.
2. Sustav mora sadržavati odgovarajuće preglednike koji omogućuju čitanje članaka u knjižnici.
3. Svakoj narudžbi mora se alocirati jedinstveni identifikator (ORDER_ID) koji korisnik mora moći kopirati u svoj korisnički prostor.

Primjer: <http://www.libsys.co.in/offerings-libsys7.html>



Poteškoće: Prirodni jezik



- Zahtjeva socijalne i komunikacijske vještine
- Nedostatak jasnoće
 - preciznost nije lako postići bez detaljiziranog i teško čitljivog dokumenta
- Miješaju se funkcionalni i nefunkcionalni zahtjevi.
- Nenamjerno objedinjavanje više zahtjeva u jednom.

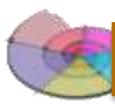
- Nejasno postavljeni zahtjevi mogu biti različito interpretirani od korisnika i razvojnih timova
 - uzrokuje probleme u procesu razvoja i kršenje ugovora

- Npr. u LIBSYS sustavu: “odgovarajući preglednik”:
 - korisnik – preglednici posebne namjene za svaki tip dokumenta.
 - razvojni tim – samo preglednik teksta kao bitnog sadržaja dokumenta.

Primjer:

- Zahtjev potporne rešetke (engl. *grid*) u grafičkom uređivaču:
- Za precizno postavljanje entiteta na crtež, korisnik može uporabom upravljačkog panela uključiti rešetku u centimetrima ili inčima.
Inicijalno je rešetka isključena. Rešetka se može uključiti i isključiti kao i mijenjati mjerne jedinice tijekom rada s editorom. Biti će osigurana opcija smanjivanja slike kako bi stala na zaslon, ali broj prikazanih crta rešetke će biti reduciran kako ne bi popunile manje slike.
- Miješaju se tri različita tipa zahtjeva:
 - Koncepcijski
 - potreba za rešetkom
 - Nefunkcionalni
 - mjerne jedinice rešetke
 - Nefunkcionalni ulazno izlazni zahtjevi
 - prebacivanje između tipova rešetki

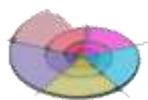
Izvor: Sommerville, I., Software engineering



Kompletност i konzistencija zahtjeva



- Kompletni zahtjevi:
 - sadrže opise svih zahtijevanih mogućnosti.
- Konzistentni zahtjevi:
 - ne smiju sadržavati konflikte ili kontradikcije u opisima zahtijevanih mogućnosti.
- U praksi je nemoguće postići *kompletan i konzistentan* dokument o zahtjevima složenih sustava.



Problemi dokumentiranja zahtjeva



- Različitost izražavanja specifikacije
 - različiti stilovi pisanja
 - različita razina iskustva
 - različiti formati
 - predetaljni ili nedovoljno specificirani zahtjevi
- Moguća poboljšanja:
 - jedan zahtjev po stavci
 - grupiranje (hijerarhijsko) povezanih zahtjeva
 - numeriranje
 - uporaba standarda

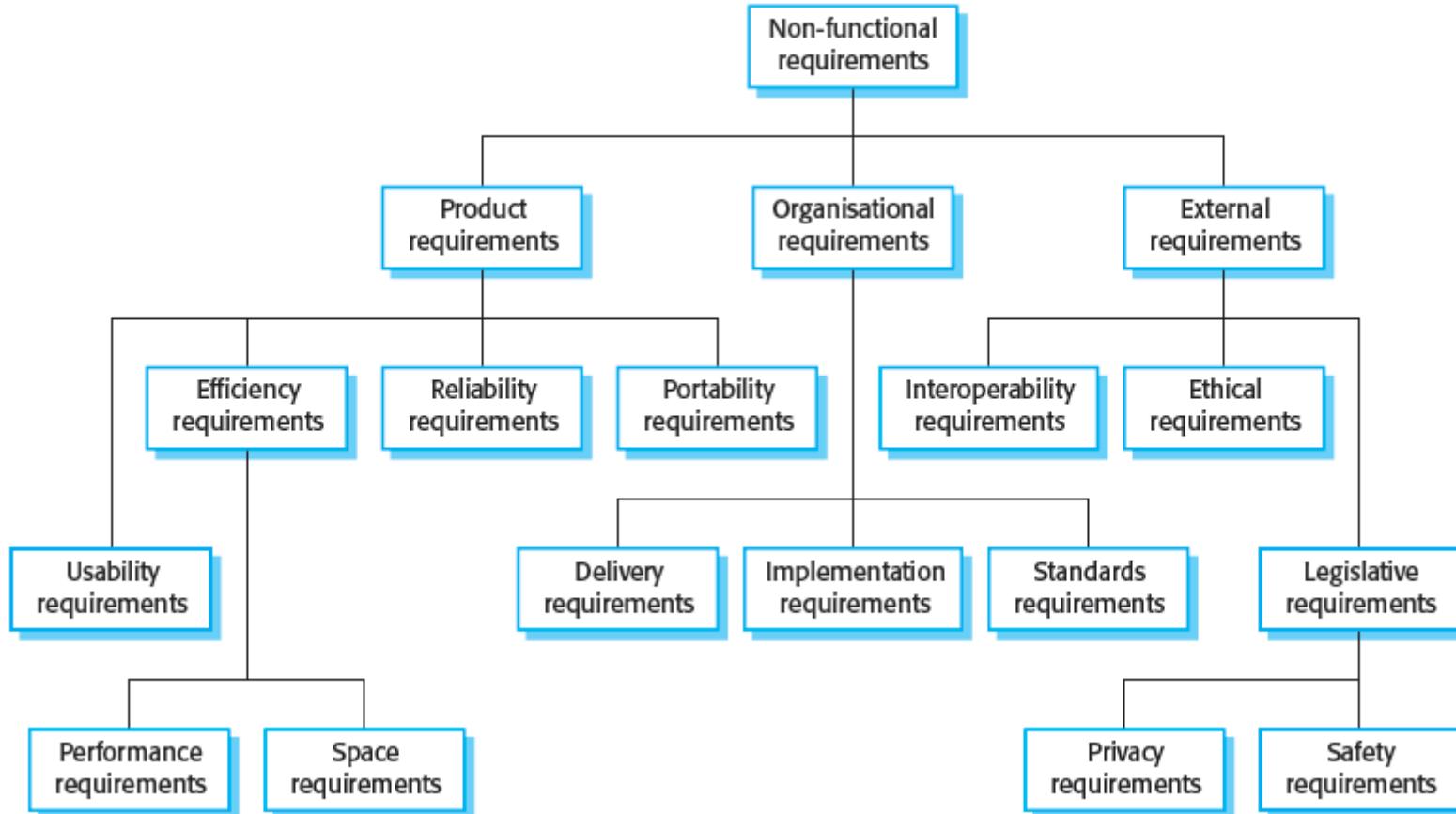


Klasifikacija nefunkcionalnih zahtjeva



- Zahtjevi programskog produkta:
 - zahtjevi koji specificiraju da se isporučeni produkt mora ponašati na osobit način (npr. vrijeme odziva).
- Organizacijski zahtjevi:
 - zahtjevi koji su rezultat organizacijskih pravila i procedura (npr. uporaba propisanog standardnog procesa razvoja, DoD ADA).
- Vanjski zahtjevi:
 - zahtjevi koji proizlaze izvan sustava i razvojnog procesa (međusobna operabilnost, legislativni zahtjevi i sl.).
- Moraju biti mjerljivi!

Primjer klasifikacije



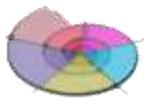
Izvor: Sommerville, I., Software engineering



Primjer: Nefunkcionalni zahtjevi



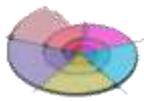
- Analiza primjera LIBSYS:
- Zahtjevi programskog produkta:
 - korisničko sučelje LIBSYS sustava biti će *implementirano kao jednostavni HTML* bez uporabe okvira ili Java apleta.
- Organizacijski zahtjevi:
 - proces razvoja sustava i isporučeni dokumenti moraju slijediti standard XYZCo-SP-STAN-95.
- Vanjski zahtjevi:
 - sustav neće operatorima otkriti osobne informacije o klijentima (osim njihovog imena i referentnog broja).



Zahtjevi domene primjene



- Zahtjevi domene primjene mogu biti ***novi funkcionalni zahtjevi ili ograničenja*** na postojeće zahtjeve.
- Problemi zahtjeva domene:
 - ***razumljivost***: programeri ne razumiju domenu primjene i traže detaljan opis zahtjeva.
 - ***implicitnost***: Specijalisti domene poznaju primjenu tako dobro da podrazumijevaju zahtjeve (koje tada eksplicitno ne određuju).
- Npr. LIBSYS zahtjevi domene primjene:
 - zbog ograničenja u pravima kopiranja neki dokumenti se po dolasku moraju odmah izbrisati.
 - ovisno o zahtjevu korisnika dokumenti se mogu ispisati lokalno kako bi se ručno dostavili korisniku.



Primjer: Standard ISO 9126



- ISO/IEC 9126 Software engineering - Definira osnovna svojstva programske potpore
- Funkcionalnost (*engl. functionality*)
 - suitability
 - accuracy
 - interoperability
 - security
 - functionality Compliance
- Pouzdanost (*engl. reliability*)
 - maturity
 - fault Tolerance
 - recoverability
 - reliability Compliance
- Uporabljivost (*engl. usability*)
 - understandability
 - learnability
 - operability
 - attractiveness
 - usability Compliance
- Efikasnost (*engl. efficiency*)
 - Time Behaviour
 - Resource Utilisation
 - Efficiency Compliance
- Održavanje (*engl. Maintainability*)
 - Analyzability
 - Changeability
 - Stability
 - Testability
 - Maintainability Compliance
- Prenosivost (*engl. Portability*)
 - Adaptability
 - Installability
 - Co-Existence
 - Replaceability
 - Portability Compliance

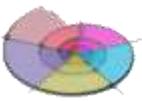
Primjer

- Tablični zapis funkcijskih zahtjeva
- tablični zapis nefunkcijsk zahtjeva

Zahtjev	Opis
F1	
F2	

Zahtjev	Opis
NF1	
NF2	
D1	

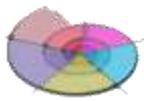
ZAHTJEVI SUSTAVA



Zahtjevi sustava



- Detaljnija specifikacija funkcija sustava, njegovih usluga i ograničenja nego zahtjevi korisnika.
 - uloga tih specifikacija je definiranje oblikovanja sustava.
 - mogu se uključiti u ugovor o isporuci sustava.
- Zahtjevi sustava mogu se definirati ili prikazati nekim od modela sustava.
 - model procesa programskog inženjerstva je apstraktna reprezentacija procesa.
- Odnos zahtjeva sustava i oblikovanja
 - u principu zahtjevi određuju **ŠTO** sustav mora raditi, a oblikovanje (dizajn) određuje **KAKO** će se to ostvariti.
 - u praksi su zahtjevi i oblikovanje neodvojivi.
 - Arhitektura sustava strukturira zahtjeve.
 - Sustav često mora radit u sinergiji s drugim sustavima koji generiraju zahtjeve na oblikovanje.
 - Uporaba specifičnog oblikovanja može biti zahtjev domene primjene.



Izražavanje zahtjeva sustava



■ *Strukturirani prirodni jezik*

- definiranje standardnih formulara i obrazaca u kojima se izražavaju zahtjevi (definicije, ulazni podaci i izvori, prethodni i posljedični uvjeti, popratni efekti, ...).
- prednost ovakve specifikacije je u zadržavanju izražajnosti prirodnog jezika, ali uz nametnutu izvjesnu uniformnost. Nedostatak je ograničena terminologija.
- u praksi nema usvojene globalne standardizacije.

■ *Jezik za opis oblikovanja (npr. SDL)*

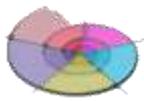
- poput programskog jezika, ali s više apstraktnih obilježja, definira se operacijski model sustava.

■ *Grafička notacija (npr. UML)*

- grafički jezik proširen tekstrom.

■ *Matematička specifikacija (FSM, teorija skupova i sl.)*

- notacija zasnovana na matematičkom konceptu. Najstrože definirana specifikacija. Korisnici je ne vole jer je ne razumiju.



Izražavanje zahtjeva sustava



- Strukturirani prirodni jezik
 - koristi se u zadavanju projekta.
- Jezici za opis oblikovanja (npr. SDL)
 - 1968 ITU study of stored program control systems
 - 1988 Blue Book SDL (SDL-88) Effective tools. Syntax well defined - formal definition. Language much as 1984.
 - 1999 SDL-2000, MSC-2000 podrška za OO modeliranje (UML), poboljšana podrška implementacija.
- Grafička notacija
 - UML – Unified Modeling Language
- Matematička specifikacija
 - kripke strukture, logika, vremenska logika.



Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose – the dose in insulin to be delivered
Destination	Main control loop
Action:	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin..
Post-condition	r0 is replaced by r1 then r1 is replaced by r2
Side-effects	None

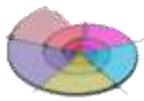
Izvor: Sommerville, I., Software engineering

Zahtjevi sustava: specifikacija sučelja



- Potrebno je specificirati sučelje prema korisniku i prema drugim sustavima. Postoje tri tipa sučelja:
 - proceduralno sučelje
 - skup usluga kroz sučelje – *primjensko programsko sučelje engl. Application Programming Interface - API* .
 - strukture podataka koje se izmjenjuju s drugim sustavima
 - npr. Entity-Relation-Attribute
 - predstavljanje podataka – značenje pojedinih podataka
- Formalna notacija je vrlo efikasan način specifikacije sučelja. Npr.:

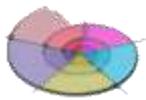
```
interface PrintServer {  
  
    // defines an abstract printer server  
    // requires:      interface Printer, interface PrintDoc  
    // provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter  
  
    void initialize ( Printer p ) ;  
    void print ( Printer p, PrintDoc d ) ;  
    void displayPrintQueue ( Printer p ) ;  
    void cancelPrintJob ( Printer p, PrintDoc d ) ;  
    void switchPrinter ( Printer p1, Printer p2, PrintDoc d ) ;  
}  
//PrintServer
```



Standardizacija zahtjeva



- IEEE 830 *Recommended Practice for Software Requirements Specifications*
 - http://standards.ieee.org/reading/ieee/std_public/description/se/index.html
- Definira općenitu strukturu dokumenta
 - uvod
 - opći opis sustava
 - specifičnosti zahtjeva
 - prilozi
 - indeks
- Primjer

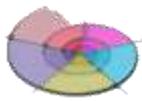


Struktura dokumenta zahtjeva



■ Predložak

1. predgovor
2. uvod
3. pojmovnik
4. definicija zahtjeva korisnika
5. arhitektura sustava
6. specifikacija zahtjeva sustava
7. model sustava
8. razvoj sustava
9. prilozi
10. indeks



Zaključci o zahtjevima



- Zahtjevi postavljaju što sustav treba raditi i definiraju ograničenja u implementaciji i radu sustava.
 - ne postoji jedinstveni standard
- Korisnički zahtjevi su izjave na višoj apstraktnoj razini što bi sustav trebao raditi.
 - prirodni jezik, tablice, dijagrami
- Zahtjevi sustava su detaljne specifikacije o funkcijama sustava.
 - strukturirani prirodni jezik, specifični jezici oblikovanja, grafička notacija, matematička specifikacija
- Funkcionalni zahtjevi definiraju usluge koje sustav osigurava.
- Nefunkcionalni zahtjevi postavljaju ograničenja na sustav ili na proces oblikovanja sustava.
- Dokument zahtjeva programskog produkta je usklađen skup izjava o svim zahtjevima na sustav.
- IEEE standard je korisna početna točka za definiranje detaljiziranog specifičnog načina pisanja dokumenta zahtjeva.

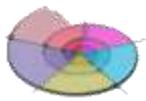
PROCESI INŽENJERSTVA ZAHTJEVA



Proces programskog inženjerstva



- Proces predstavlja strukturiran (organiziran) skup aktivnosti koji vodi nekom cilju.
- Proces inženjerstva zahtjeva je skup aktivnosti koje generiraju i dokumentiraju zahtjeve.
- Ciljevi:
 - opisati temeljne inženjerske aktivnosti i njihove odnose u **generiranju i dokumentaciji zahtjeva**.
 - upoznati se s tehnikama za izlučivanje i analizu zahtjeva.
 - opisati validaciju zahtjeva i ulogu recenzenta.
 - analizirati upravljanje zahtjevima (engl. *requirements management*) kao potporu procesu inženjerstva zahtjeva.

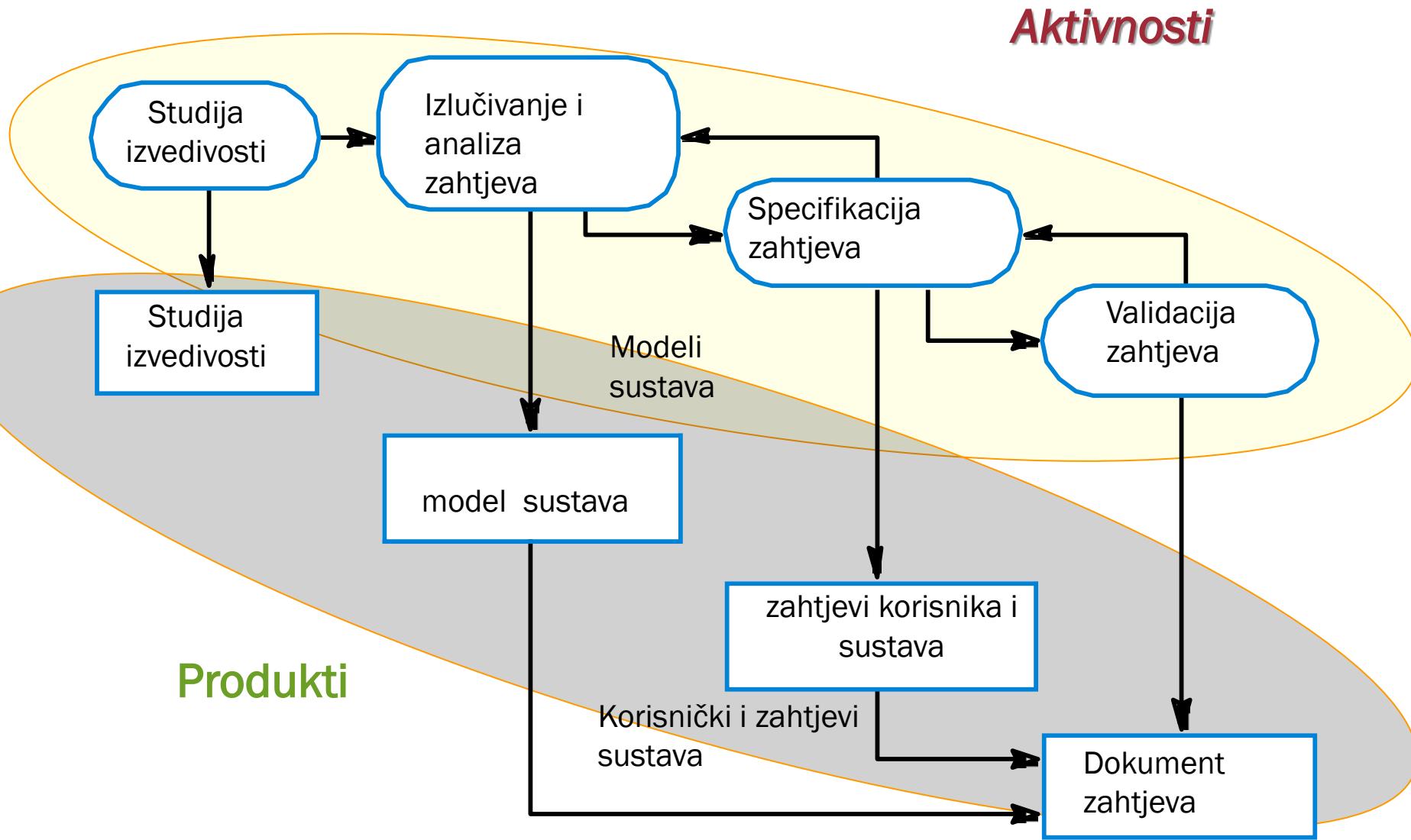


Procesi inženjerstva zahtjeva

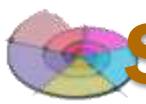


- Procesi koji su u upotrebi u inženjerstvu zahtjeva razlikuju se ovisno o domeni primjene, ljudskim resursima i organizaciji koja oblikuje zahtjeve.
 - nema jedinstvenog procesa inženjerstva zahtjeva
- Dva uobičajena **modela** procesa inženjerstva zahtjeva:
 - klasični i spiralni
- Generičke aktivnosti zajedničke aktivnostima inženjerstva zahtjeva:
 - studija izvedivosti (engl. *feasibility study*)
 - izlučivanje zahtjeva (engl. *requirements elicitation*) i analiza i specifikacija zahtjeva
 - validacija zahtjeva
 - upravljanje zahtjevima

Klasični model procesa inženjerstva zahtjeva



Izvor: Sommerville, I., Software engineering



Spiralni model procesa inženjerstva zahtjeva

Tro-stupanjska aktivnost

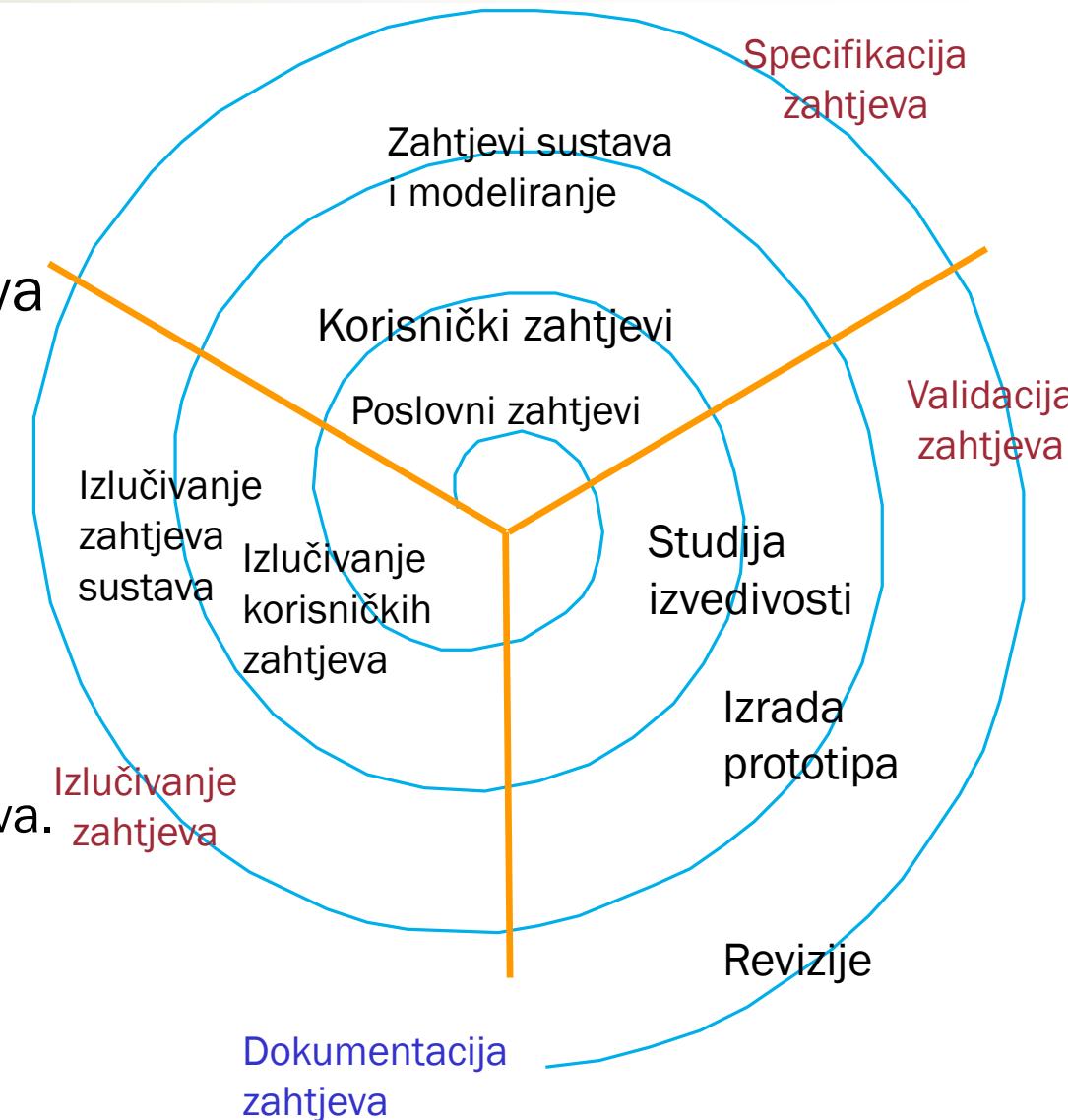
- specifikacija, validacija, izlučivanje.

Promatra proces inženjerstva zahtjeva kroz iteracije.

U svakoj iteraciji je različit intenzitet aktivnosti

- u ranim iteracijama fokus na razumijevanju poslovnog modela.
- u kasnijim modeliranje sustava.

Zahtjevi se u pojedinim iteracijama specificiraju s različitom razinom detalja.

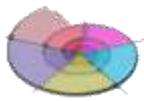


Generičke aktivnosti inženjerstva zahtjeva:

STUDIJA IZVEDIVOSTI

Studija izvedivosti

- engl. *feasibility study*
- Na početku procesa inženjerstva zahtjeva određuje da li se predloženi sustav isplati
 - ulaz predstavljaju preliminarni zahtjevi
 - da li je vrijedan uloženih sredstava?
- Kratka fokusirana studija koja provjerava:
 - *doprinose sustava* ciljevima organizacije u koju se uvodi.
 - *mogućnosti ostvarenja* postojećom tehnologijom i predviđenim sredstvima.
 - *mogućnosti integracije* predloženog sustava s postojećim sustavima organizacije u koju se uvodi.



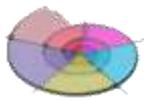
Provedba studije izvedivosti



- Temelji se na određivanju koje informacije su potrebne za studiju, prikupljanje informacija i pisanju izvješća.
- Pitanja za korisnike:
 - što ako se sustav ne implementira?
 - koji su trenutni problemi procesa organizacije?
 - kako bi predloženi sustav pomogao u poboljšanju procesa?
 - koji se problemi mogu očekivati pri integraciji novoga sustava?
 - da li je potrebna nova tehnologija ili nove vještine?
 - koje dodatne resurse organizacije traži implementacija novoga sustava?

Generičke aktivnosti inženjerstva zahtjeva:

IZLUČIVANJE I ANALIZA ZAHTJEVA



Izlučivanje i analiza zahtjeva



- engl. *Requirements elicitation*
- najznačajnija aktivnost u procesu inženjerstva zahtjeva
- Poznato i kao otkrivanje zahtjeva (engl. *discovery*).
- Uključuje stručno tehnički obrazovano osoblje koje u zajedničkom radu s kupcima i korisnicima:
 - razjašnjava domenu primjene;
 - definira usluge koje sustav treba pružiti;
 - određuje ograničenja u radu sustava.
- Može uključivati:
 - krajnje korisnike sustava, rukovoditelje, inženjere uključene u održavanje sustava, eksperte domene primjene, predstavnike sindikata i sl.
 ≡ dionici (engl. *stakeholders*)

Izlučivanja zahtjeva

- Osnovne aktivnosti izlučivanja
 - razumjevanje domene primjene
 - Opće znanje područja
 - razumjevanje problema
 - Detalji specifičnog problema korisnika
 - razumjevanje konteksta
 - Interakcije sustava i ukupnog cilja
 - razumjevanje potreba i ograničenja korisnika
- Aktivnosti izlučivanja zahtjeva
 - Identificiranje aktora
 - Tipovi korisnika, uloge, sustavi
 - Utvrđivanje scenarija
 - Interakcija korisnika i sustava
 - Utvrđivanje obrazaca uporabe
 - Poboljšanje obrazaca uporabe
 - Utvrđivanje odnosa obrazaca
 - Utvrđivanje nefunkcionalnih zahtjeva
 - Performanse, sigurnost, ...

Metode izlučivanja zahtjeva

- **Intervjuiranje** kao metoda izlučivanja
- **Scenarij** kao metoda izlučivanja
- Izlučivanje i specificiranje zahtjeva **obrascima uporabe** (UML “use cases”)
- Specificiranje **dinamičkih interakcija** u sustavu (UML sekvencijski dijagrami)
- Promatranje rada
- Izrada prototipa
- ...

Spiralni model izlučivanja i analize zahtjeva



- 4 osnovne aktivnosti
- **Izlučivanje/Otkrivanje zahtjeva**
 - interakcija s dionicima s ciljem otkrivanja njihovih zahtjeva. Zahtjevi domene primjene se također definiraju na ovom stupnju. Izvori informacija su dokumenti, dionici, slični sustavi.
- **Klasifikacija i organizacija zahtjeva**
 - grupiraju se srodnici zahtjevi i organiziraju u koherentne grozdove (klastere).
- **Ustanovljavanje prioriteta i pregovaranje**
 - zahtjevi se razvrstavaju po prioritetima i razrješuju konflikti.
- **Dokumentiranje zahtjeva**
 - zahtjevi se dokumentiraju i ubacuju u sljedeći ciklus spirale.



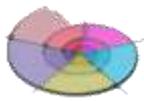
Analiza zahtjeva

■ Provjere

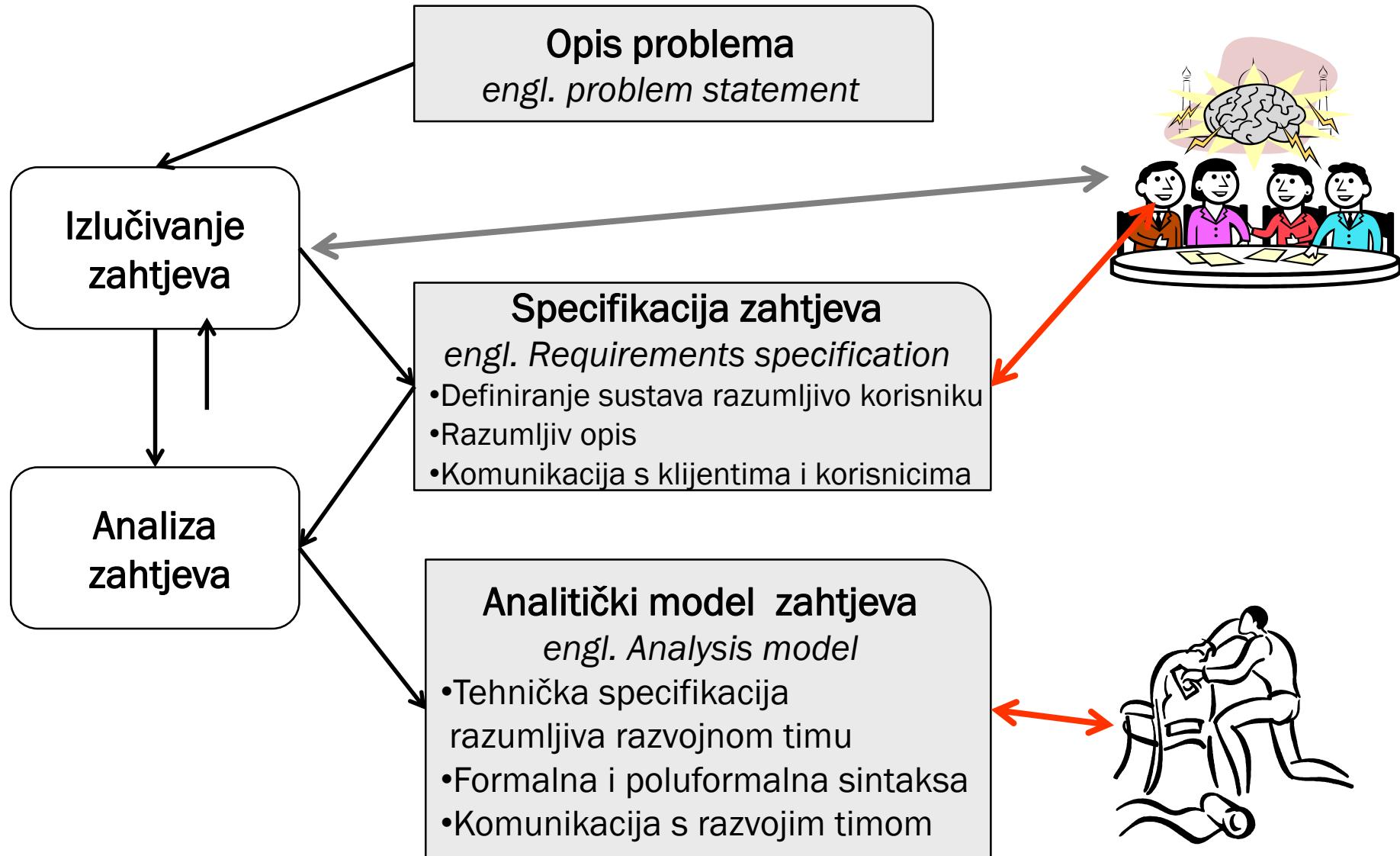
- neophodnosti zahtjeva
- konzistentnosti i kompletnosti
- mogućnosti ostvarenja

■ Problemi:

- dionici ne znaju što stvarno žele.
- dionici izražavaju zahtjeve na različite, njima specifične načine.
- različiti dionici mogu imati konfliktne zahtjeve.
- organizacijski i politički faktori mogu utjecati na zahtjeve.
- zahtjevi se mijenjaju za vrijeme procesa analize.
- pojavljuju se novi dionici uz promjenu poslovnog okruženja.



Rezultati procesa izlučivanja



Pogledi

- engl. *Viewpoints*
- Način strukturiranja zahtjeva tako da oslikavaju perspektivu i fokus različitih dionika
 - dionici se mogu razvrstati po različitim pogledima.
- Ova **više-perspektivna analiza** je značajna jer ne postoji jedan jedinstveni ispravan način u analizi zahtjeva sustava i omogućava razrješavanje konflikata.
- Tipovi pogleda:
 - **pogledi interakcije**
 - Ljudi i drugi sustavi koji izravno komuniciraju sa sustavom.
 - **indirektni pogledi**
 - Dionici koji ne koriste sustav izravno, ali utječu na zahtjeve.
 - **pogledi domene primjene**
 - Karakteristike domene i ograničenja koja utječu na zahtjeve.

Primjer: Bankomat

■ Bankomat

- čitač mag. Kartica
- tastatura, zaslon
- utor za umetanje omotnica
- spremnik novčanica
- pisač za ispis potvrda
- ključ za uključivanje/isključivanje
- komunikacija s bankom

Opis rada

- Posluživanje jednog korisnika
- Ubacivanje kartice + identifikacija PIN-om, podaci se šalju banci na validaciju tijekom svake transakcije
- Korisnik može obaviti jednu ili više transakcija
- Kartica se zadržava u bankomatu sve dok korisnik obavlja transakcije, nakon završetka kartica se vraća (postoji iznimke)



Usluge bankomata



- Korisnik može podići novce s računa kartice. Podizanje novac odobrava banka.
- Korisnik može uložiti novac na račun kartice (gotovina/ček)
 - korisnik upisuje uloženi iznos
 - operator ručno verificira iznos
 - banka odobrava prihvaćanje uplate
- Korisnik može prebacivati novce između računa
- Korisnik može pregledati stanje računa

Bankomat: dionici

- Dionici sustava bankomata:
 - bankovni klijenti
 - predstavnici drugih banaka
 - bankovni rukovoditelji
 - šalterski službenici
 - administratori baza podataka
 - rukovoditelji sigurnosti
 - marketing odjel
 - inženjeri održavanja sustava
 - sklopoljia i programske potpore
 - regulatorna tijela za bankarstvo

Bankomat: pogledi

■ *Pogledi interakcije*

- klijenti
- predstavnici banaka

■ *Indirektni pogledi*

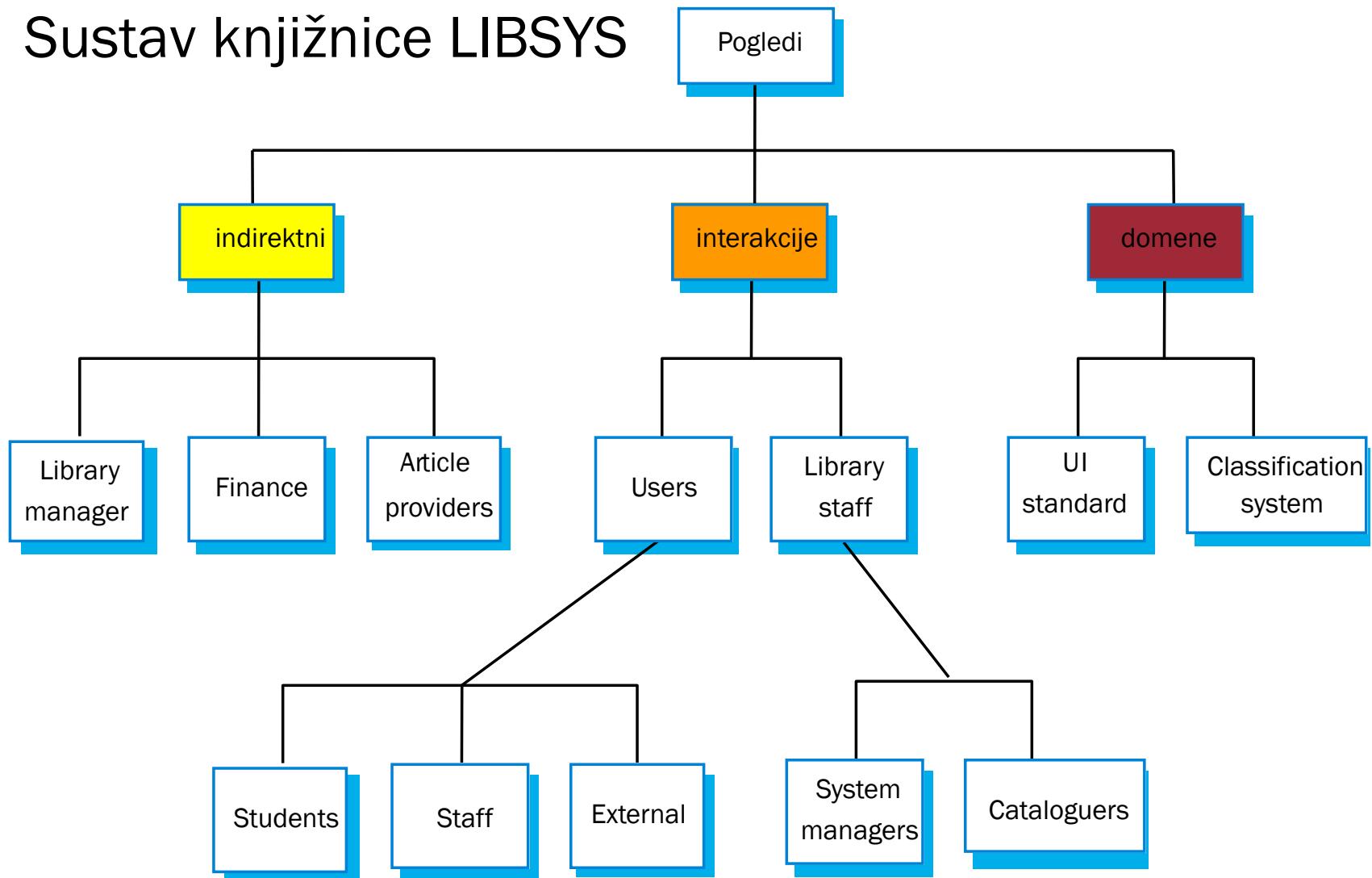
- rukovoditelji
- osoblje zaduženo za sigurnost
- šalterski službenici
- administratori podataka
- odjel marketinga

■ *Pogledi domene primjene*

- standardi u komunikaciji između banaka

Primjer pogleda: LIBSYS

Sustav knjižnice LIBSYS



Izvor: Sommerville, I., Software engineering

Generičke aktivnosti inženjerstva zahtjeva:

METODE IZLUČIVANJA ZAHTJEVA



Metode u izlučivanju i analizi zahtjeva

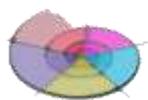
- Intervjuiranje kao metoda izlučivanja
- Scenarij kao metoda izlučivanja
- Izlučivanje i specificiranje zahtjeva *obrascima uporabe* (UML “use_cases”)
- Specificiranje *dinamičkih interakcija* u sustavu (UML sekvencijski dijagrami)

Intervjuiranje

- U formalnom i neformalnom intervjuiranju tim zadužen za *inženjerstvo zahtjeva* **ispituje** dionike o sustavu koji trenutno koriste te o *novo predloženom sustavu*.
- Tipovi intervjeta:
- **Zatvoreni intervju**
 - odgovara se na skup prije definiranih pitanja.
- **Otvoreni intervju**
 - ne postoje definirana pitanja, već se niz pitanja otvara i raspravlja s dionicima.
- U praksi intervjeti često ne daju dobre rezultate za *zahtjeve domene primjene*
 - inženjeri zahtjeva često ne razumiju specifičnu terminologiju domene
 - ekspertri domene toliko poznaju te zahtjeve da ih ne artikuliraju dobro.

Scenariji

- Primjeri iz stvarnog života o načinu korištenja sustava.
- Sadržaj scenarija:
 - opis početne situacije.
 - opis normalnog/standardnog tijeka događaja.
 - opis što se eventualno može dogoditi krivo.
 - informaciju o paralelnim aktivnostima.
 - opis stanja gdje scenarij završava.
- Izlučivanje zahtjeva
 - dionici diskutiraju i kritiziraju scenarij



Primjer scenarija za sustav LIBSYS (1):

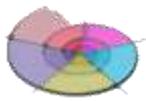


■ *Početno stanje*

- korisnik se prijavio u LIBSYS sustav, pronašao časopis u kojem se nalazi željeni članak.

■ *Normalan rad*

- korisnik odabire članak za kopiranje.
 - Sustav traži od korisnika informaciju o njegovim pravima (tip pretplate) ili načinu plaćanja.
 - Opcije plaćanja su kreditna kartica ili račun organizacije koja ima pretplatu.
- sustav traži da korisnik potpiše formular o pravima na kopiranje i ostali detaljima transakcije. To se daje LIBSYS sustavu.
- formular o pravima na kopiranje se provjerava, i ako je dozvoljeno članak u PDF formatu se proslijeđuje do korisničkog računala u sklopu LIBSYS sustava. Korisnik treba odabrati pisač i kopija članka se ispisuje. Ukoliko je članak tipa “samo za ispis”, članak se briše sa korisničkog računala nakon što korisnik potvrdi da je ispis završen.



Primjer scenarija za sustav LIBSYS (2):



■ *Greške u sustavu*

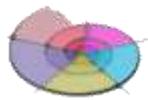
- korisnik nije ispravno popunio formular o pravima na kopiranje. U tom slučaju formular se mora ponovo dati korisniku na ispravak. Ako je ponovljeni formular krivo ispunjen, zahtjev korisnika se odbacuje.
- sustav može odbaciti način plaćanja. Korisnikov zahtjev se odbacuje.
- prijenos članka na korisnikovo računalo nije ispravno izveden. Treba ponavljati dok prijenos ne bude uspješan ili dok korisnik ne prekine transakciju.
- članak nije moguće ispisati. Ako članak nije tipa “samo za ispis” drži ga se u radnom prostoru LIBSYS sustava, a i u suprotnom članak se izbriše i korisnik kreditira u visini cijene članka.

■ *Paralelne aktivnosti*

- prijenos i obrada zahtjeva drugih korisnika LIBSYS sustava.

■ *Završno stanje*

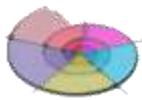
- korisnik i dalje prijavljen na sustav. Ako je članak “samo za ispis” briše se.



Metode izlučivanja zahtjeva



- Intervjuiranje kao metoda izlučivanja
- Scenarij kao metoda izlučivanja
- Izlučivanje i specificiranje zahtjeva ***obrascima uporabe*** (UML use cases)
- Specificiranje ***dinamičkih interakcija*** u sustavu (UML sekvencijski dijagrami)



Obrasci uporabe

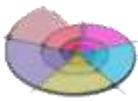
- engl. *Use cases*
- Obrasci uporabe predstavljaju tehniku preuzetu iz UML standarda. Opisuju se AKTORI u interakciji s uslugama sustava.
- Skup obrazaca uporabe opisuje **sve moguće interakcije sustava**.
- Uz obrasce uporabe, dodatno se mogu koristiti i drugi dijagrami sekvenci za detaljan opis tijeka događaja.
- Tri temeljna elementa u modelima obrazaca uporabe su: **obrasci uporabe, aktori i odnosi** (relacije, engl. *relations*) među njima.

AKTOR = korisnik



uporaba





Modeliranje obrascima uporabe



- **Model obrazaca** uporabe je pogled koji ističe ponašanje sustava kako ga vide vanjski korisnici.
- Model obrazaca uporabe razdjeljuje funkcionalnost sustava u
 - transakcije (“obrasce uporabe”)
 - razumljive korisnicima (“aktorima”).
- Pogodno za modeliranje:
 - korisničkih zahtjeva.
 - scenarija ispitivanja sustava (*engl. test scenarios*).

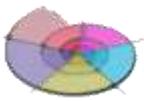
Modeliranje dinamičkih interakcija u sustavu



- Metoda izlučivanja zahtjeva modeliranjem dinamičkih interakcija u sustavu
- Modeliranje ponašanja, engl. *behavioral modeling*
- Detaljniji razvoj i prikaz scenarija u izlučivanju, analizi i dokumentiranju zahtjeva:
 - obrasci uporabe identificiraju individualne interakcije u sustavu.
 - dodatne informacije u inženjerstvu zahtjeva uz obrasce uporabe slijede iz dijagrama interakcije koji pokazuju aktore involvirane u interakciji, entitete (objekte, instancije) s kojima su u interakciji i operacije pridružene tim objektima.
- Osnovni tipovi dijagrama interakcija:
 - sekvencijski
 - kolaboracijski.

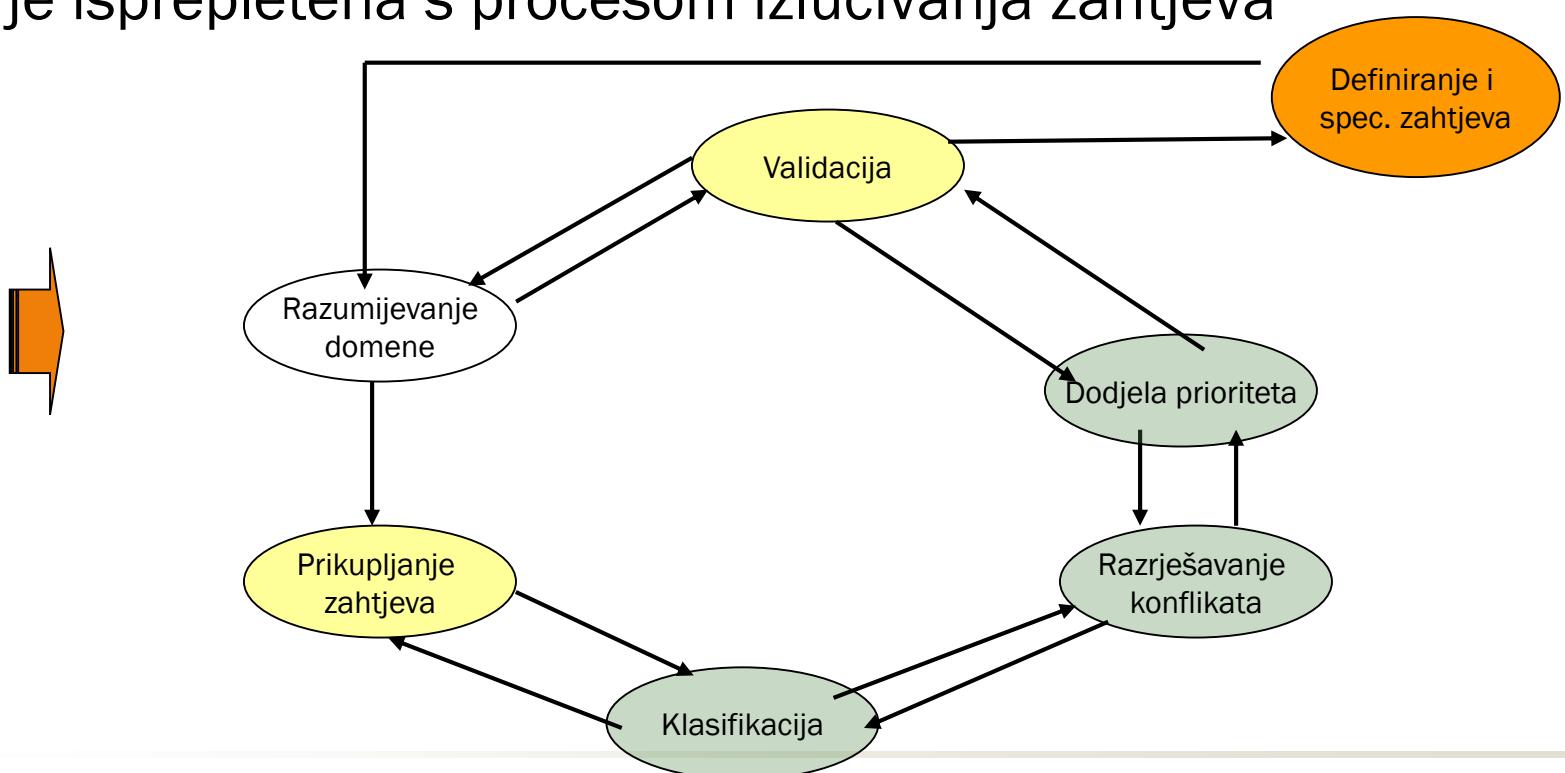
Interakcije

- Skup komunikacija između instancija elemenata sustava.
 - npr. u objektno usmjerenoj arhitekturi:
 - Pozivi operacija (procedura)
 - Kreacije instancija (objekata).
 - Destrukcije instancija (objekata).
 - komunikacije su djelomično uređene u vremenu.
- Modeliranje interakcije omogućava:
 - specificiranje međudjelovanje između elemenata sustava.
 - olakšava identifikaciju sučelja.
 - utvrđivanje potreba za raspodjelom zahtjeva.



Analiza zahtjeva

- Otkrivanje funkcije sustava, struktura sustava, ponašanje sustava.
- Cilj analize zahtjeva je utvrđivanje problema, nekompletnosti, i nejednoznačnosti u izlučenim zahtjevima.
 - dionici ih razrješavaju pregovorima.
- Analiza je isprepletena s procesom izlučivanja zahtjeva





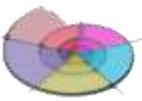
Konflikti i utvrđivanje prioriteta



- Utvrđivanje interakcije zahtjeva
 - pomaže pronalaženju problema
 - tablica
- **Zahtjevi visokog prioriteta** engl. *Core requirements*
 - obavezno razmotriti pri analizi, oblikovanju i impl.
 - neophodno demonstrirati klijentu pri preuzimanju
- **Srednji prioritet** engl. *Optional requirements*
 - obavezno razmotriti pri analizi i oblikovanju
 - uobičajeno se implementira u drugim iteracijama projekta
- **Nizak prioritet** engl. *Fancy requirements*
 - analizira se u obliku naprednih mogućnosti
 - pogodno za prikaz mogućnosti budućeg razvoja

Generičke aktivnosti inženjerstva zahtjeva:

VALIDACIJA ZAHTJEVA



Validacija zahtjeva



- Cilj validacije je pokazati da dokument zahtjeva predstavlja prihvatljiv opis sustava koji naručitelj doista želi.
 - naknadno ispravljanje pogreške u zahtjevima može biti višestruko skuplje od ispravljanja pogrešaka u implementaciji
- Tehnike validacije:
 - *recenzija zahtjeva*
 - sistematska ručna analiza
 - *izrada prototipa*
 - provjera na izvedenom sustavu
 - *generiranje ispitnih slučaja*
 - razvoj ispitnih sekvenci za provjeru zahtjeva
- Rezultati validacije
 - lista problema (pojašnjenja, nedostaci, konflikti, neostvarivost)
 - lista utvrđenih akcija za razrješavanje problema



Elementi provjere zahtjeva



■ *Razumljivost*

- Da li je dokument jasno napisan?

■ *Kompletnost*

- Da li sustav uključuje sve funkcije koje je korisnik tražio?

■ *Konzistencija*

- Da li postoji konflikt u zahtjevima?

■ *Valjanost*

- Da li sustav osigurava funkcije koje podupiru potrebe korisnika?

■ *Realnost*

- Da li se sve funkcije mogu implementirati uz danu tehnologiju i proračun?

■ *Provjerljivost*

- Da li se svi zahtjevi mogu provjeriti?

■ *Sljedivost*

- Da li je naveden izvor dokumenta i razlozi uvrštavanja zahtjeva?

■ *Adaptabilnost*

- Mogu li se zahtjevi mijenjati bez velikog utjecaja na druge zahtjeve?

Generičke aktivnosti inženjerstva zahtjeva:

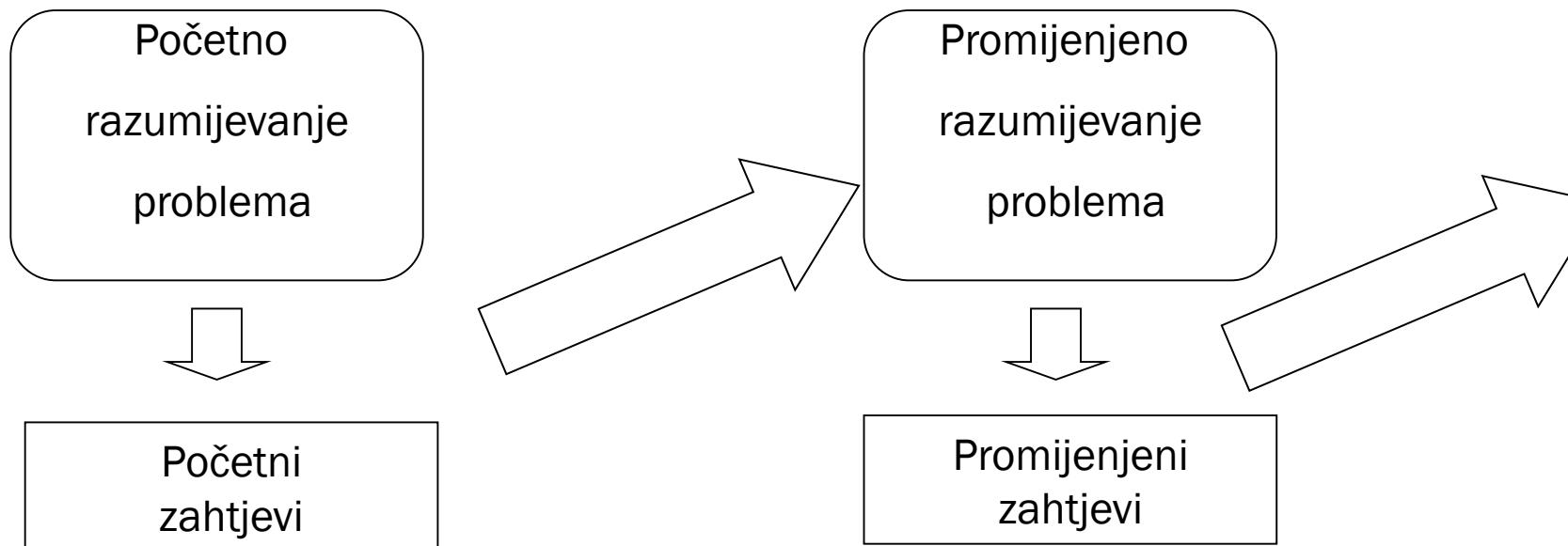
UPRAVLJANJE PROMJENAMA ZAHTJEVA



Upravljanje promjenama zahtjeva



- Upravljanje ili rukovanje engl. *requirements management*
- Promjene nastupaju zbog promijenjenog modela poslovanja, boljeg razumijevanja procesa tijekom razvoja ili konfliktnim zahtjevima u različitim pogledima.
- Evolucija zahtjeva:





Klasifikacija promjena zahtjeva



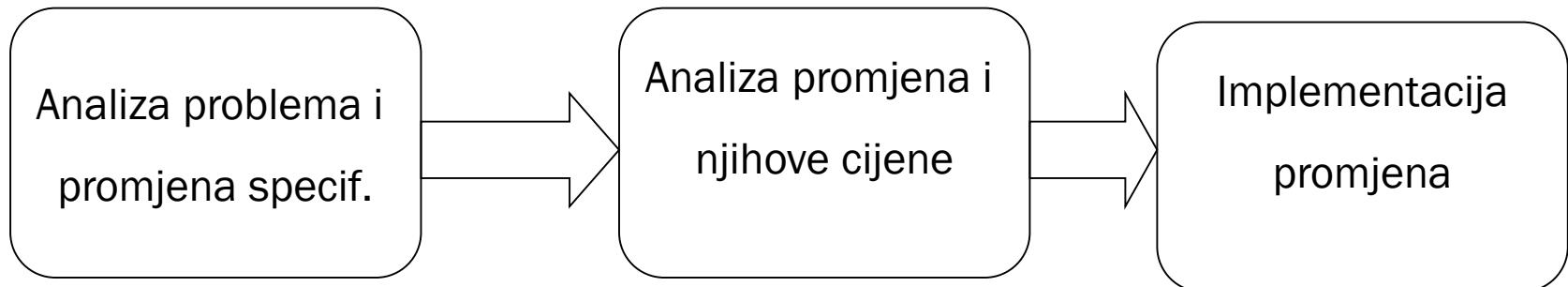
- Okolinom promijenjeni zahtjevi
 - promjena zbog promjene okoline u kojoj organizacija posluje (npr. bolnica mijenja financijski model pokrivanja usluga).
- Novonastali zahtjevi
 - zahtjevi koji se pojavljuju kako kupac sve bolje razumije sustav koji se oblikuje.
- Posljedični zahtjevi
 - zahtjevi koji nastaju nakon uvođenja sustav u eksploraciju, a rezultat su promjena procesa rada u organizaciji nastalih upravo uvođenjem novoga sustava
- Zahtjevi kompatibilnosti
 - zahtjevi koji ovise o procesima drugih sustava u organizaciji; ako se ti sustavi mijenjaju to traži promjenu zahtjeva i novo uvedeni sustav



Upravljanje procesom promjena



- Planiranje:
 - identifikacije zahtjeva (individualno identificiranje zahtjeva).
- Upravljanje procesom promjena
 - proces koji slijedi kada se utvrdi potreba za promjenom



- Sljedivost
 - koje informacije su potrebne za povezivanje zahtjeva
- Izbor CASE alata
 - automatiziranje skladištenja, unošenja promjena i povezivanje dokumenata

SOCIJALNI I ORGANIZACIJSKI ČIMBENICI



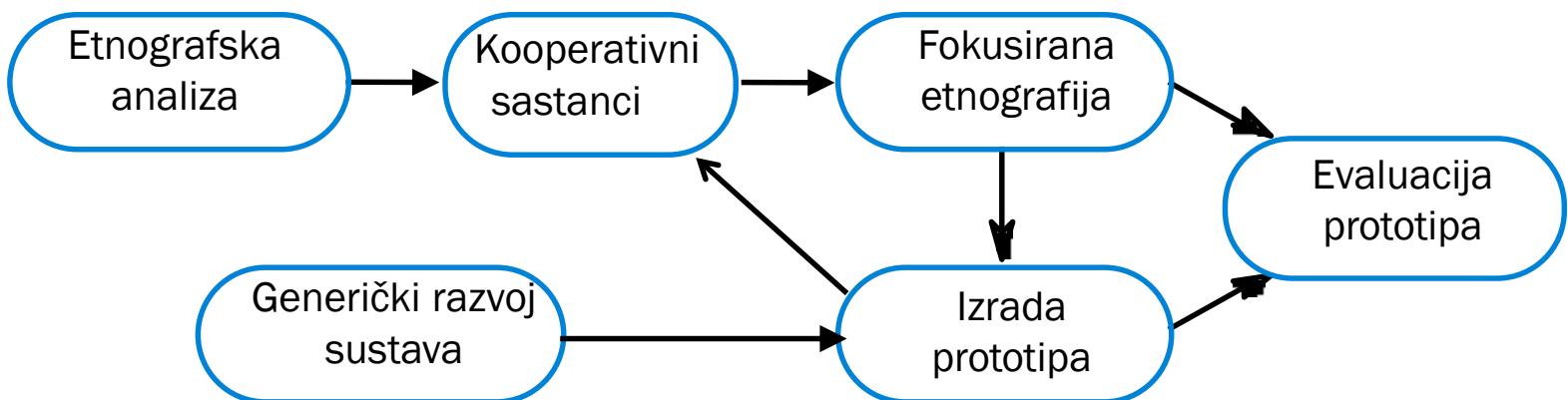
Socijalni i organizacijski čimbenici



- Programski produkti se uvijek koriste u socijalnom i organizacijskom kontekstu.
 - to uvelike utječe, a ponekad i dominira na zahtjeve sustava.
- Socijalni i organizacijski čimbenici nisu jedinstven pogled, već **utjecaj na sve poglede.**
 - oblikovanje programske potpore mora to uvažiti, ali trenutno ne postoji sistematski postupak kako se to može uključiti u analizu zahtjeva.
- Ne postoje jednostavni modeli za opis obavljanja nekog posla.
 - ljudima je često teško precizno opisati što rade!?
 - postojeći modeli zasnovani na prošlim, a ne obrascima ponašanja na novom poslu.
 - djelomično se tome može doskočiti izradom prototipa, te praćenjem rada na njemu.
- Za razumijevanje procesa korisna znanja iz područja društvenih znanosti

Etnografija

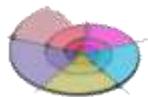
- Kvalitativno promatranje i opis ponašanja ljudi u društvu.
- Zahtjevi izvedeni temeljem istraživanja kako ljudi stvarno rade
 - ne kako definicija poslovnog procesa formalno propisuje!!!
- Zahtjevi izvedeni temeljem kooperacije i uzimajući u obzir aktivnosti drugih ljudi.
- Fokusirana etnografija (etnografija + prototip):



Izvor: Sommerville, I., Software engineering

Zaključak

- Proces inženjerstva zahtjeva uključuje:
 - studiju izvedivosti, izlučivanje zahtjeva i analizu;
 - specifikaciju zahtjeva i rukovanje (upravljanje) zahtjevima.
- Izlučivanje i analiza zahtjeva je ***iterativan proces*** koji uključuje razumijevanje domene primjene, prikupljanje, klasifikaciju, strukturiranje, sastavljanje prioriteta i validaciju zahtjeva.
 - sustavi imaju više dionika s različitim zahtjevima.
 - socijalni i organizacijski čimbenici utječu na zahtjeve sustava.
- Validacija zahtjeva bavi se valjanošću, konzistentnošću, kompletnošću, realizmom u izvedbi i provjerljivošću.
- Promjene načina poslovanja nužno mijenjaju zahtjeve
 - kontinuirani proces
- Rukovanje zahtjevima uključuje ***planiranje i upravljanje promjenama*** (engl. ***Requirements management***) zahtjevima.



Stvarnost



Izvor: <http://www.projectcartoon.com>

Diskusija

-
-
-
-

Oblikovanje programske potpore

2014./2015.

Primjena UML-a u inženjerstvu zahtjeva



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

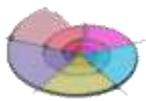


Tema

- Primjena UML-a (engl. Unified Modeling Language) u inženjerstvu zahtjeva
- Primjena UML-a
- Razvoj i elementi jezika
- Obrasci uporabe
- Dijagram interakcija

Literatura

- Sommerville, I., *Software engineering*, 8th ed., Addison-Wesley, 2007.
- Grady Booch, James Rumbaugh, Ivar Jacobson: *Unified Modeling Language User Guide*, 2nd Edition, 2005
- Simon Bennett, John Skelton, Ken Lunn: *Schaum's Outline of UML*, Second Edition, 2005
- WWW
- Rational Software Architect
 - <http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=/com.ibm.xtools.modeler.doc/topics/cextend.html>



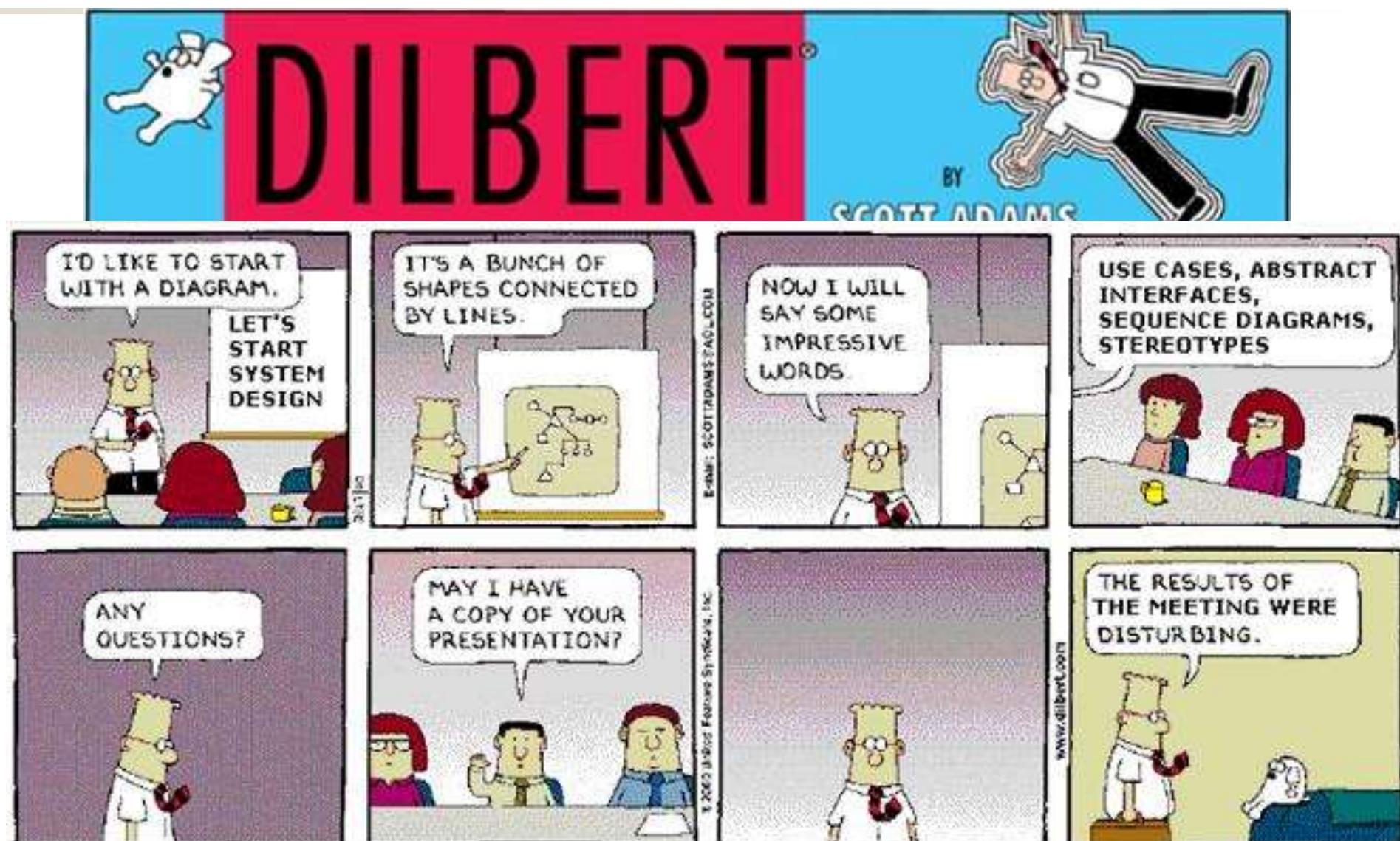
Unified Modeling Language



- UML je jezik za:
 - vizualizaciju;
 - specifikaciju;
 - oblikovanje i
 - dokumentiranje
- Artefakata programske potpore.
- Omogućava različite poglede na model
- ‘de facto’ standardni jezik programskog oblikovanja
- UML je posebice prikladan za specificiranje *objektno usmjerene* arhitekture programske potpore.
- Dijelovi UML-a pogodni su u specificiranju i drugih arhitektura.
- Omogućava
 - Višestruke međusobno povezane poglede
 - Poluformalnu semantiku izraženu kao metamodel
 - Jezik za opis formalnih logičkih ograničenja



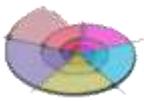
Percepcija UML-a



Copyright © 2000 United Feature Syndicate, Inc.

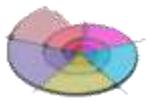
Povijest

- UML: Unified Modeling Language
 - Rumbaugh et al.: OMT 1991
 - Baze podataka i Entity Relation model
 - Jacobson: OOSE 1992
 - use cases / requirements
 - Booch: Booch notation 1994
 - Oblikovanje jezika, strukturni aspekt i nasljeđivanje
- “The Three Amigos” 1997
 - unified really means "joint effort makes more money compared to notational wars"



Potreba

- Osmisliti jezik koji osigurava jednostavan riječnik i pravila kombiniranja riječi u svrhu komunikacije.
- U jeziku modeliranja riječnik i pravila su usmjerena na konceptualnu i fizičku reprezentaciju sustava.
⇒ UML standard
- "A language provides a **vocabulary** and **the rules for combining words** [...] for the purpose of **communication**. A *modeling* language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modeling language such as the UML is thus a standard language for **software blueprints**."
 - *From "UML user guide"*



UML: osnovna svojstva

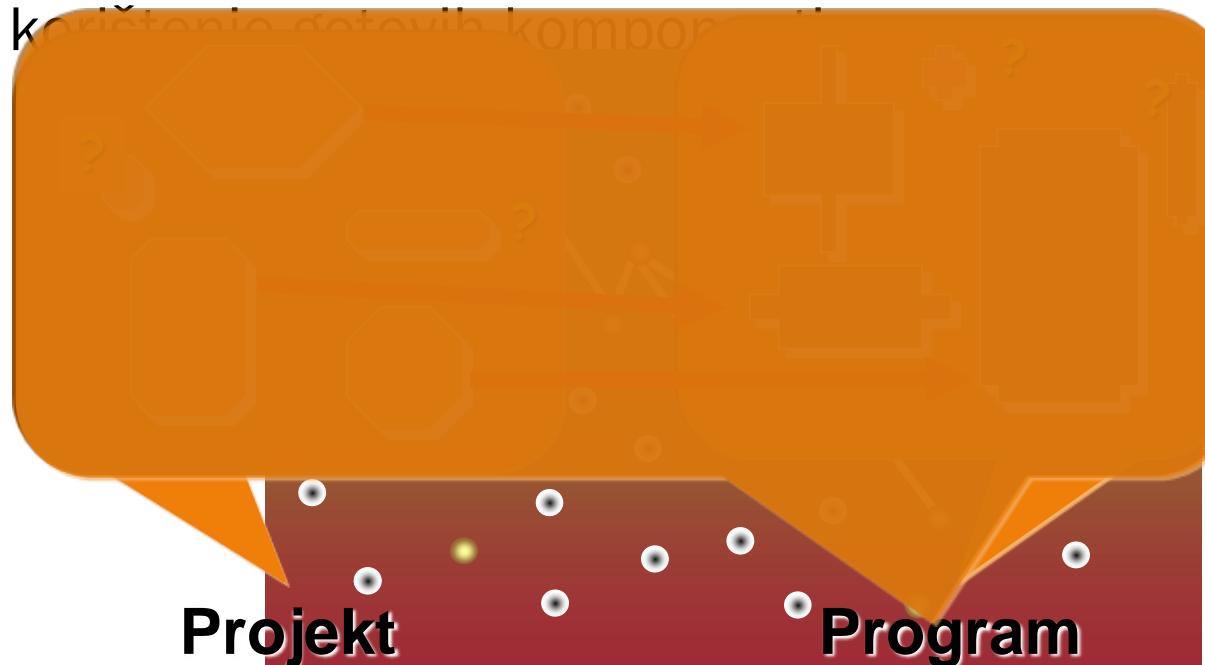


■ Osnovne ideje

- obuhvatiti i opisati poslovne procese
- poboljšati komunikaciju
- pomoći u borbi s kompleksnošću
- definirati logičku arhitekturu sustava
- omogućiti ponovno korištenje i rekorštiranje postojećih komponenti

■ Namjena:

- vizualizacija
- specificiranje
- konstrukcija
- dokumentiranje



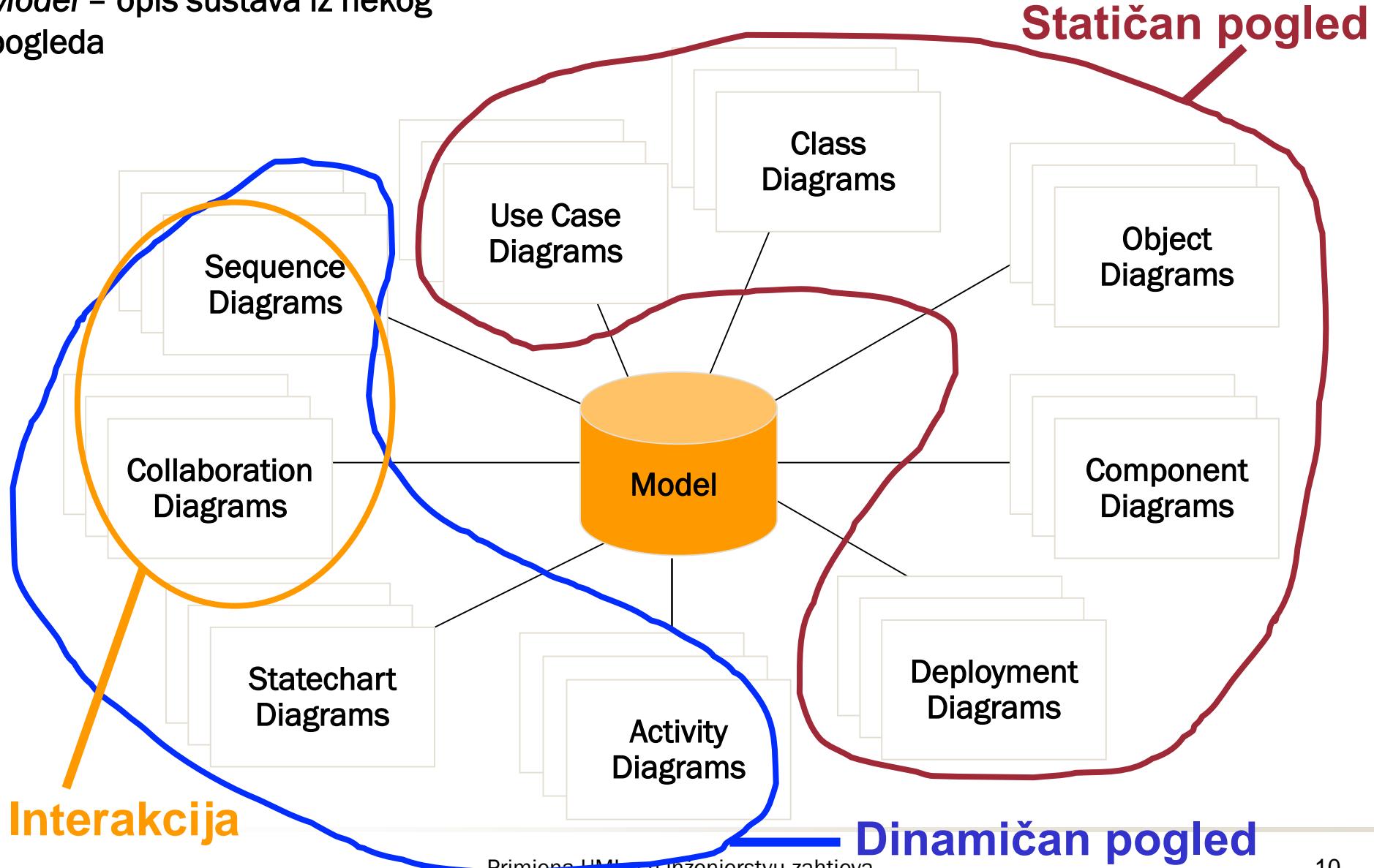


Osnovni elementi UML-a

- Stvari (engl. *things*)
 - strukturne stvari (engl. *structural things*)
 - klasa; sučelje; suradnja; obrasci uporabe; aktivna klasa; komponenta; čvor
 - aktori, signali, procesi i niti, aplikacije, dokumenti, datoteke, biblioteke, stranice, tablice
 - stvari ponašanja (engl. *behavioral things*)
 - interakcija; stanje
 - stvari grupiranja (engl. *grouping things*)
 - Organizacija elemenata u grupe (samo konceptualno, hijerarhija), paketi
 - stvari označavanja (engl. *annotation things*)
 - opisni elementi, formalni/neformalni opis
- Relacije (engl. *relationships*)
 - Asocijacije (eng. *association*)
 - Generalizacije (eng. *generalization*)
 - Realizacije (eng. *realization*)
 - Ovisnosti (eng. *dependency*)
- Dijagrami (engl. *diagrams*)
 - Dijagram klasa; objekata; slučajeva korištenja; toka; suradnji; stanja; aktivnosti; komponenata;....

Modeli, Pogledi, Dijagrami

Model – opis sustava iz nekog pogleda

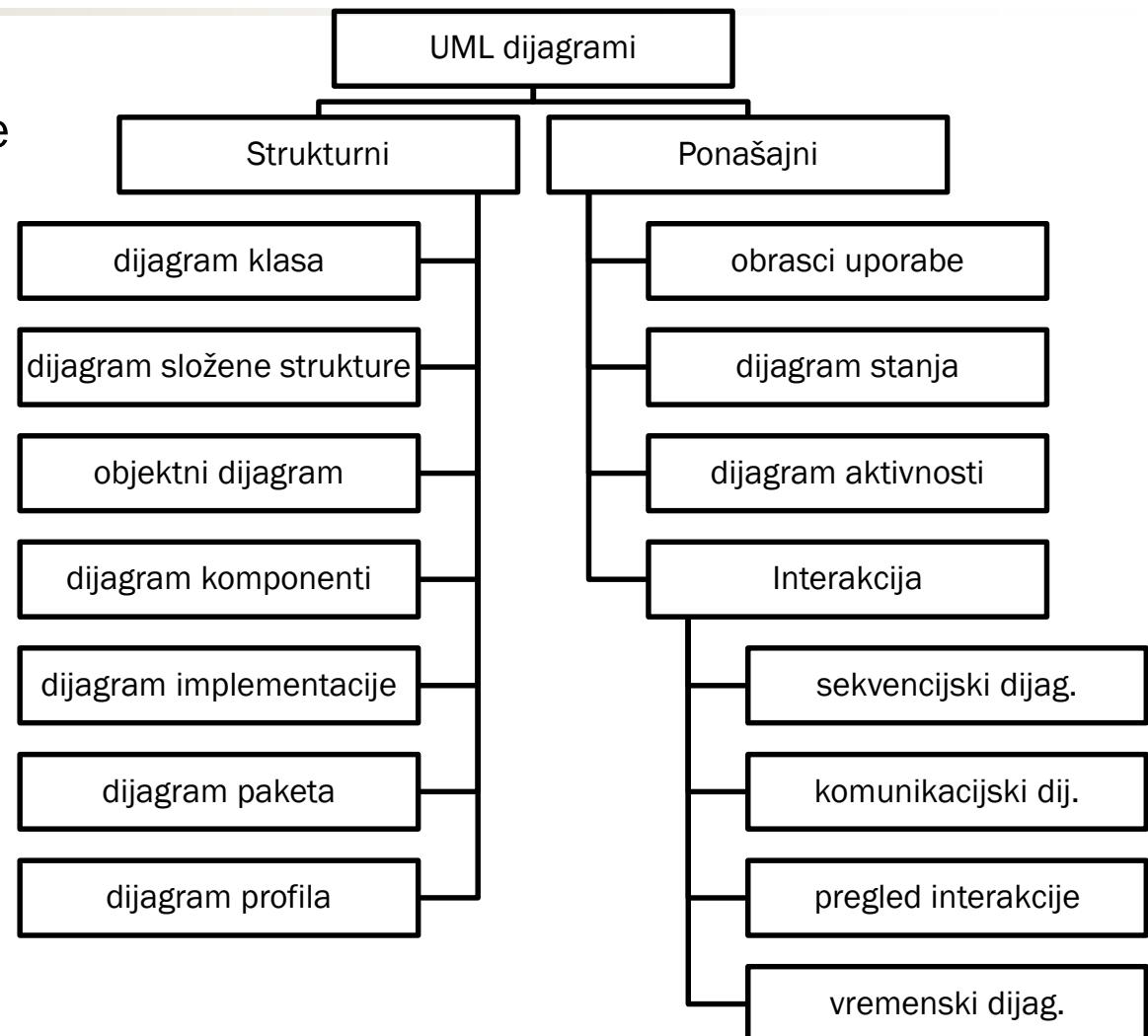


Dijagrami

- Pogled na model
 - Iz perspektive dionika
 - Djelomična reprezentacija sustava
 - Semantički konzistentan s ostalim pogledima
- Standardni dijagrami (UML 1.1)
 - **statični:** dijagram obrazaca uporabe, dijagram razreda, dijagram objekata, dijagram komponenata, dijagram razmještaja
 - *use case, class, object, component, deployment*
 - **dinamični:** sekvencijski dijagram, komunikacijski (kolaboracijski) dijagram, dijagram stanja (“statechart”), dijagram aktivnosti
 - *sequence, collaboration, statechart, activity*
 - uz manje promjene najčešće upotrebljavani dijagrami i u novijim inačicama UML-a
- Specifičnost dijagrama
 - Svaki ima vlastitu sintaksu i semantiku ☺ ?
- Evolucija UML-a
 - Veljača 2009. UML 2.2
 - Ožujak 2011. UML 2.4- beta
 - - razrada 2.5

UML 2.4

- Evoluiraju tijekom procesa oblikovanja, te se mijenjaju donošenjem odluka o oblikovanju i proširuju novim detaljima



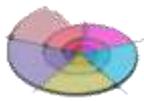


UML dijagrami

- | | |
|---------------------------------|-------------------------------------|
| ■ Obrasci uporabe | engl. use case diagrams |
| ■ Sekvencijski dijagram | engl. sequence diagrams |
| ■ Komunikacijski dijagram | engl. communication diagrams |
| ■ Dijagram stanja | engl. state machine diagrams |
| ■ Dijagram aktivnosti | engl. activity diagrams |
| ■ Dijagram komponentni | engl. component diagrams |
| ■ Dijagram implementacije | engl. deployment diagrams |
| ■ Dijagram paketa | engl. package diagrams |
| ■ Dijagram pregleda interakcije | engl. interaction overview diagrams |
| ■ Vremenski dijagram | engl. timing diagrams |
| ■ Dijagram profila | engl. profile diagram |
| ■ Dijagram klasa | engl. class diagrams |
| ■ Dijagram objekata | engl. object diagrams |
| ■ Dijagram složene strukture | engl. composite structure diagrams |

Obrasci uporabe

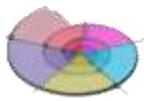
- engl. *Use cases*
- Pogled na sustav koji naglašava njegovo vanjsko ponašanje prema korisniku
- Dijeli funkcionalnost sustava u skup obrazaca uporabe (transakcija) koji su značajni korisniku (aktoru)
- Skup obrazaca uporabe opisuje **sve moguće interakcije sustava.**
- Uz obrasce uporabe, dodatno se mogu koristiti i dijagrami sekvenci kako bi se detaljno opisao tijek događaja.



Dijagram obrazaca uporabe

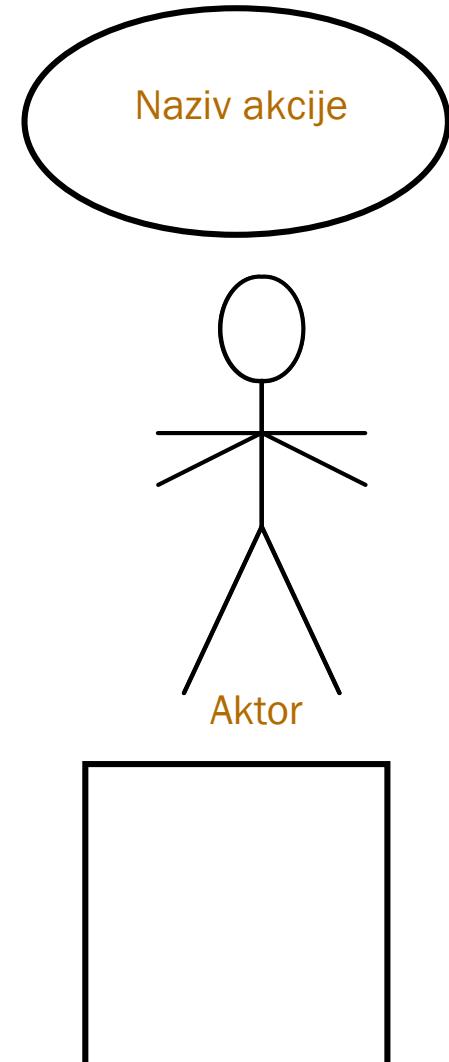


- Opisuje obrasce uporabe, aktore i njihove odnose.
- Jednostavan grafički prikaz granica sustava, tko/što koristi sustav, te prikazuje kako ga koristi
- Detalji unutar dijagrama obrazaca uporabe mogu se predstaviti tekstom ili dodatnim dijagramima interakcije između elemenata sustava.
- Vrste obrazaca uporabe:
 - Dijagram – uobičajeno u uporabi
 - Tekstualni opis



Elementi obrazaca uporabe

- Osnovni (jezgreni) elementi:
 - **Obrazac uporabe** engl. *use case*
 - Sekvenca akcija (uključujući varijante) koje sustav ili drugi entitet obavlja u interakciji s aktorima sustava (≡ funkcionalnost)
- **Aktor** – engl. *actor*
 - Vanjski objekt koji komunicira sa sustavom
 - Jedinstveno ime (+ po potrebi opis)
 - Koherentan skup uloga koje imaju korisnici u interakciji s obrascima uporabe.
 - Uloga korisnika u sustavu
 - Jedan korisnik – više aktora
 - Vanjski sustav
- **Granica sustava** – engl. *system boundary*
 - granica između fizikalnog sustava i različitih aktora koji su u interakciji s fizikalnim sustavom.



Aktori

- Za imenovanje upotrijebiti imenice
- Potrebno opisati njihovu ulogu u interakciji sa sustavom
- Definirati opseg sustava, identificirati elemente na rubu sustava i elemente o kojima sustav ovisi
- Obrasci uporabe se pišu iz perspektive aktora!
 - Što radi koji aktor?
 - Za aktivost aktora definira se obrazac uporabe
 - Opišite ju tekstom
 - Zadržite razinu apstrakcije

Obrazac uporabe

- Imenovanje: glagol-imenica
- Opisati uvjete početka, završetka, tijek razmjene inf., nefukcijska svojstva, ... – tekstualno + drugi UML dijagrami
- Definira doseg sustava i funkcionalnost koju podržava u sustavu te elemente o kojima ovisi
- Potpomaže aktorima u svrhu ostvarivanja ciljeva
 - Obrasci uporabe predstavljaju funkcionalnosti sustava i odgovornosti (*engl. responsibilities*)

Osnovne poveznice obrazaca uporabe



- Poveznice – jezgreni odnosi
 - (engl. *Communication relationship*)
- **Pridruživanje/asocijacija** – engl. *association*
 - komunikacija instancije aktora i instancije obrasca uporabe
- **Proširenje** – engl. *extend*
 - Odnos od proširene uporabe do osnovne uporabe
- **Obuhvaćanje/nužno sadrži** – engl. *include*
 - Odnos od osnovnog obrasca do uključenog obrasca
 - Definira ponašanje uključenog obrasca u odnosu na osnovni obrazac uporabe.
 - Osnovni obrazac sadrži ponašanje definirano u drugom obrascu.
- **Popćenje** – engl. *generalization*
 - odnos između općeg i specifičnog
- Opis višestrukih vrijednosti
 - Jednoznačan 1; 6; *
 - interval vrijednosti 0..1

<<extend>>
----->

<<include>>
----->

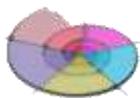




Elementi obrasca uporabe

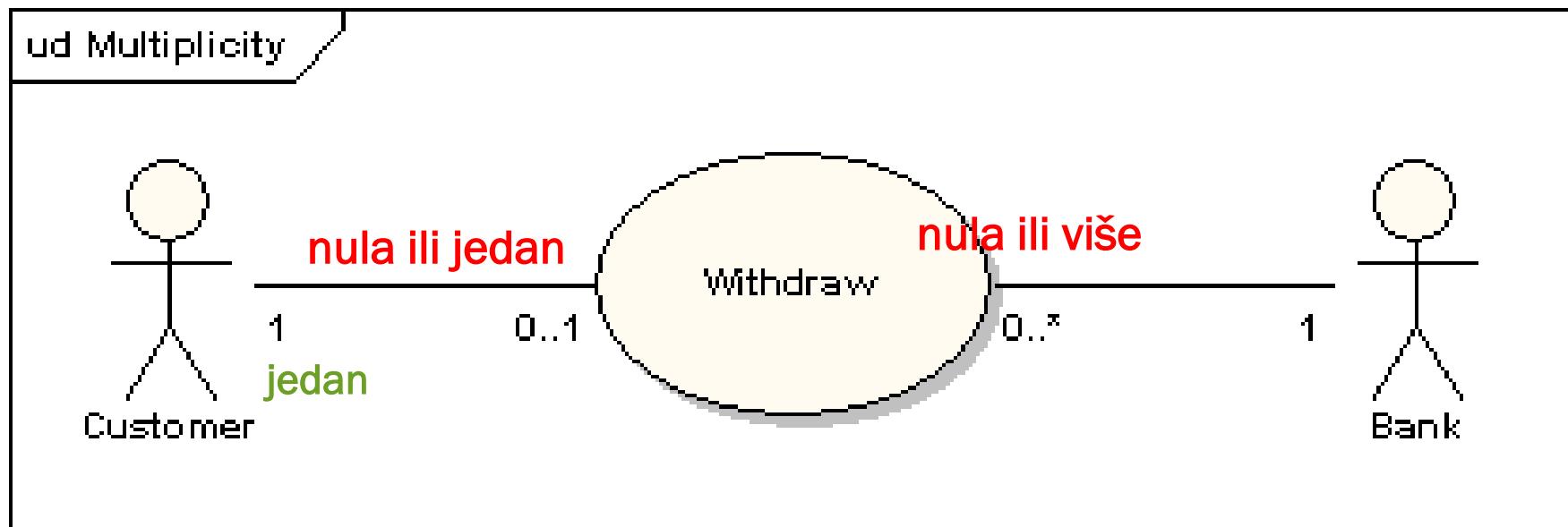


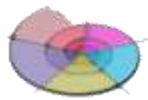
Element	Opis
Redni broj	Jedinstveni identifikacijski broj
Namjena	Koja je namjena sustava
Naziv	Jasno govoreće ime
Opis	Kratak opis
Glavni aktor	Tko je glavni aktor na kojeg se obrazac odnosi
Preduvjeti	Što mora biti zadovoljeno za aktivaciju obrasca
Pokretač	Događaj koji aktivira obrazac
Opis osnovnog tijeka	Opis idealnog tijeka događaja
Opis mogućih odstupanja	Najznačajnije alternative i iznimke



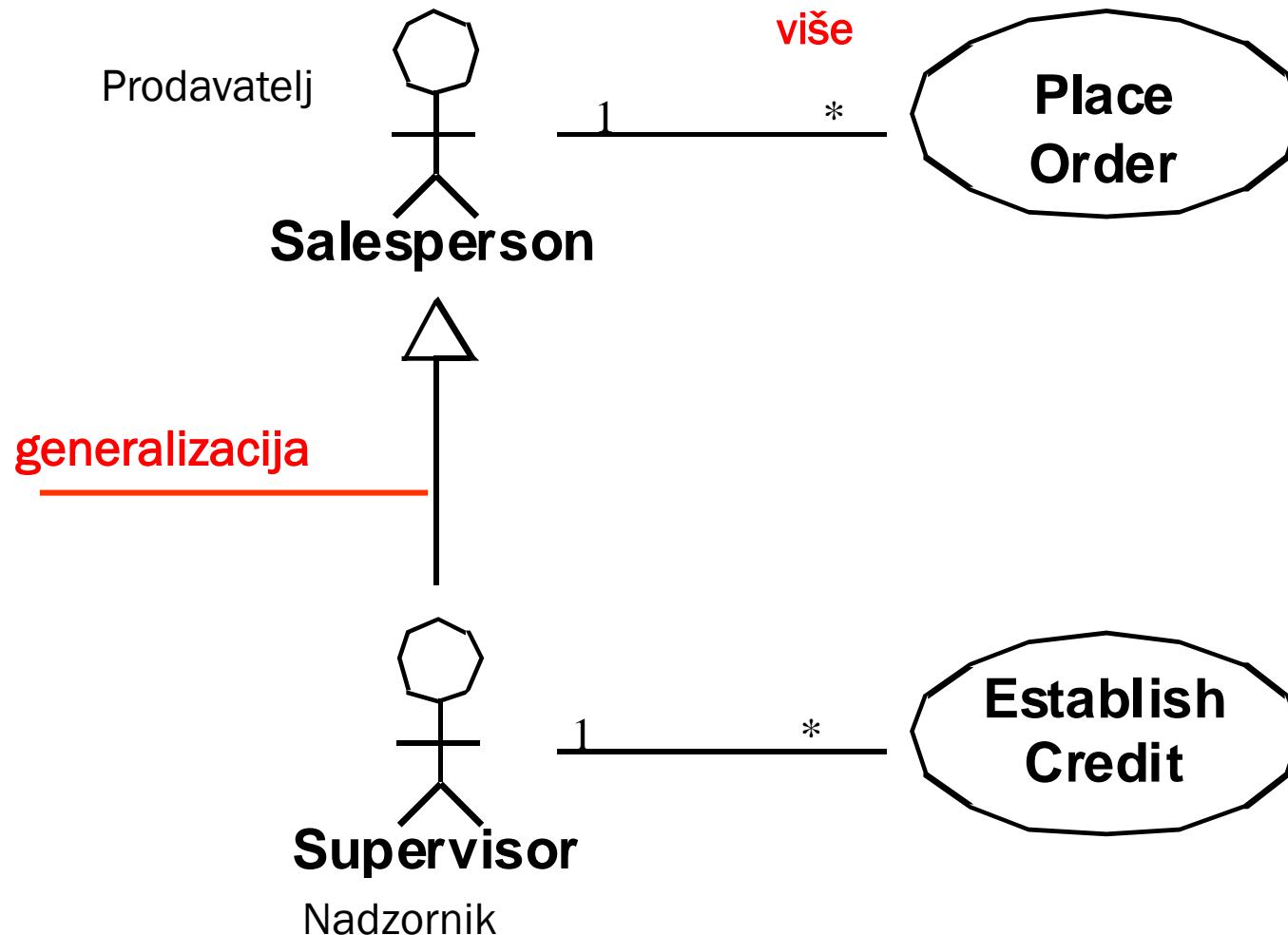
Višestrukost u obrascima uporabe

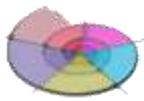
- Spojnica u obrascima uporabe može opcijски imati **višestrukost** (engl. *multiplicity values*) na svakom kraju.
 - Npr. klijent može imati najviše jednu isplatu u jednom času
 - banka može imati po volji broj transakcija (klijenata koji istovremeno obavljaju isplatu). (?)





Odnosi između aktora





Tekstovni obrazac uporabe



■ Primjer: Promjena rute leta

■ Aktori:

- putnik, baza računa klijenta (s planom puta), rezervacijski sustav avio kompanije.

■ Preduvjeti:

- Putnik se prijavio na sustav i odabroo opciju "promjena leta".

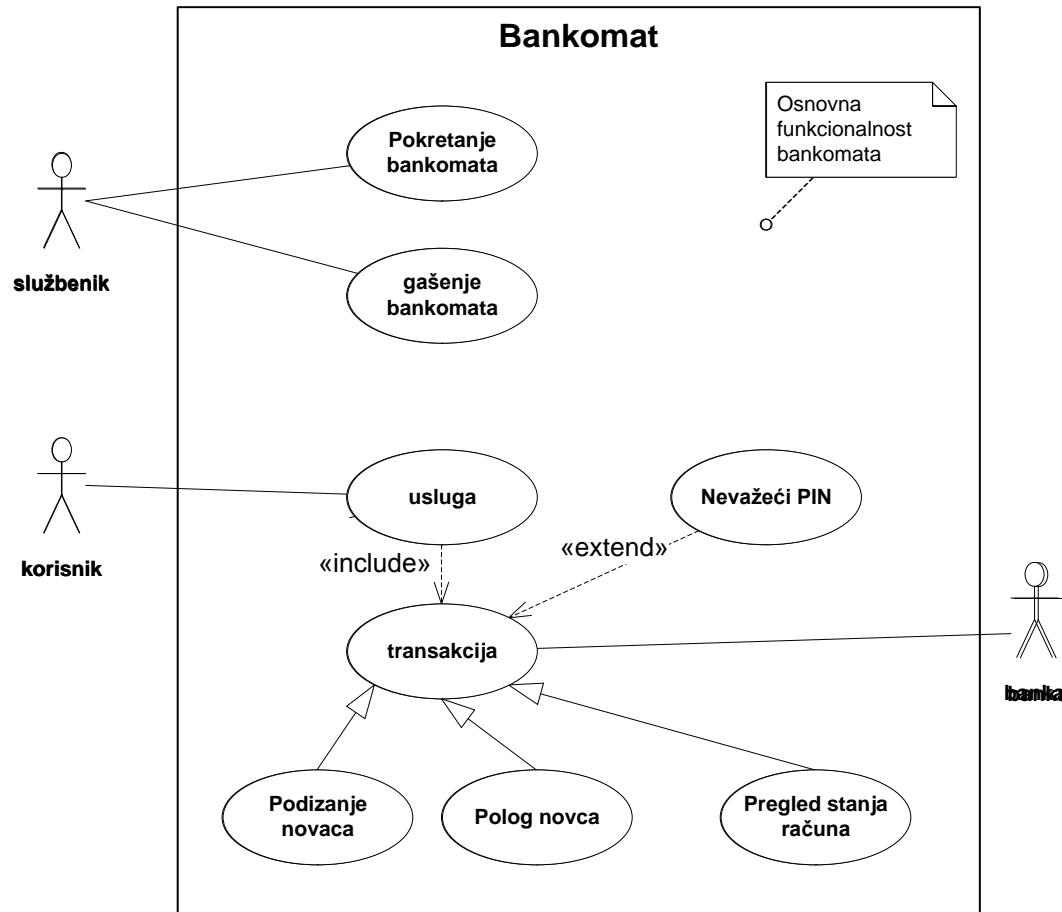
■ Temeljni tijek transakcija

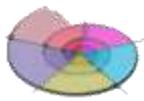
- Sustav dohvaća putnikov bankovni račun i plan puta iz baze.
- Sustav pita putnika da odabere dio plana puta koji želi mijenjati; putnik selektira segment puta.
- Sustav pita putnika za novi odlaznu i dolaznu destinaciju; putnik daje traženu informaciju.
- Ako je let moguć, tada ...
 - ...
- Sustav prikazuje sažetak transakcije..

■ Alternativni tijek transakcija

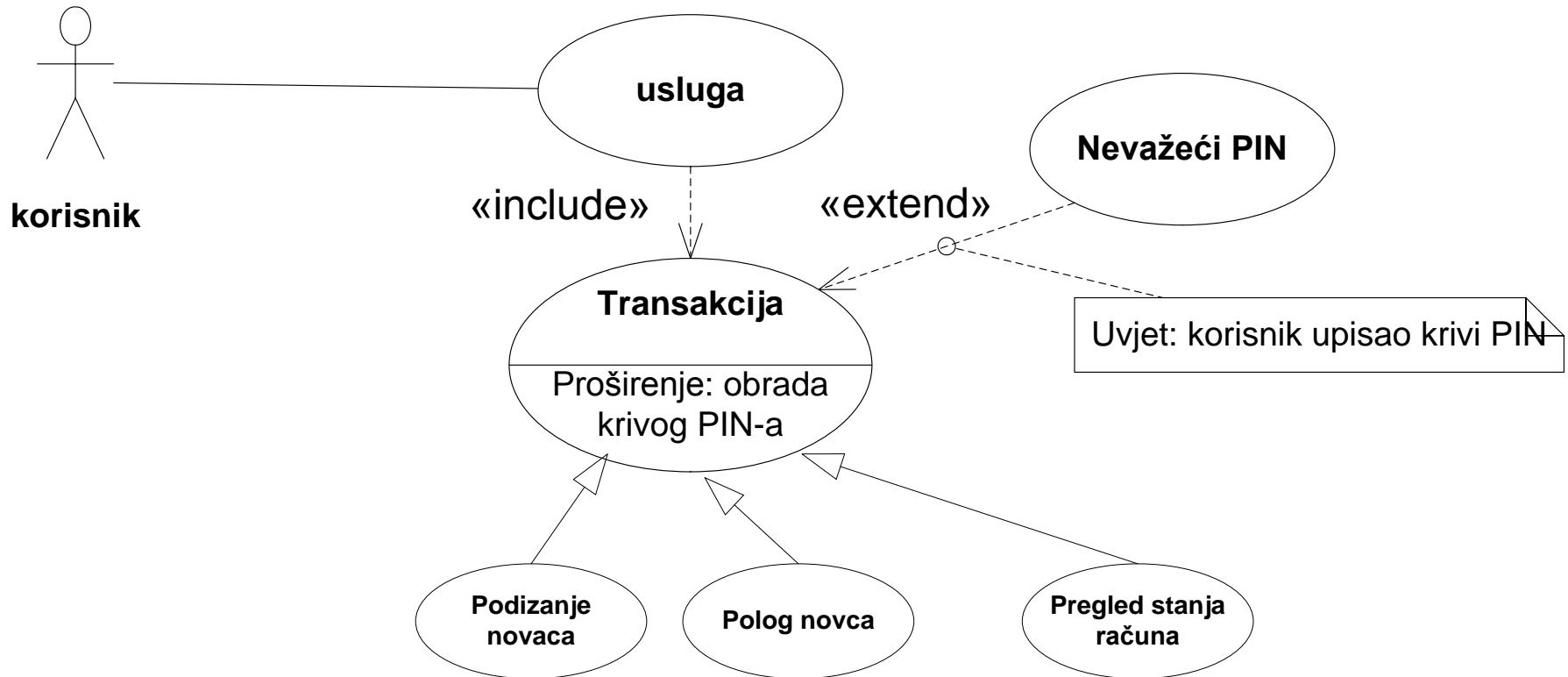
- Ako let nije moguć, tada ...

Primjer: Dijagram obrazaca uporabe





Relacije u obrascima uporabe



- **Zahtjevi korisnika**
- **Scenariji ispitivanja sustava** (engl. test scenarios)

- Za primjenu metode oblikovanja programske potpore zasnovanu na obrascima uporabe
 - Započni s obrascima uporabe i iz njih izvedi strukturne i ponašajne (engl. behavioral) modele sustava.
- Ili ako ne koristiš metodu oblikovanja programske potpore zasnovanu na obrascima uporabe
 - Osiguraj da su obrasci uporabe konzistentni s strukturnim i ponašajnim modelima.

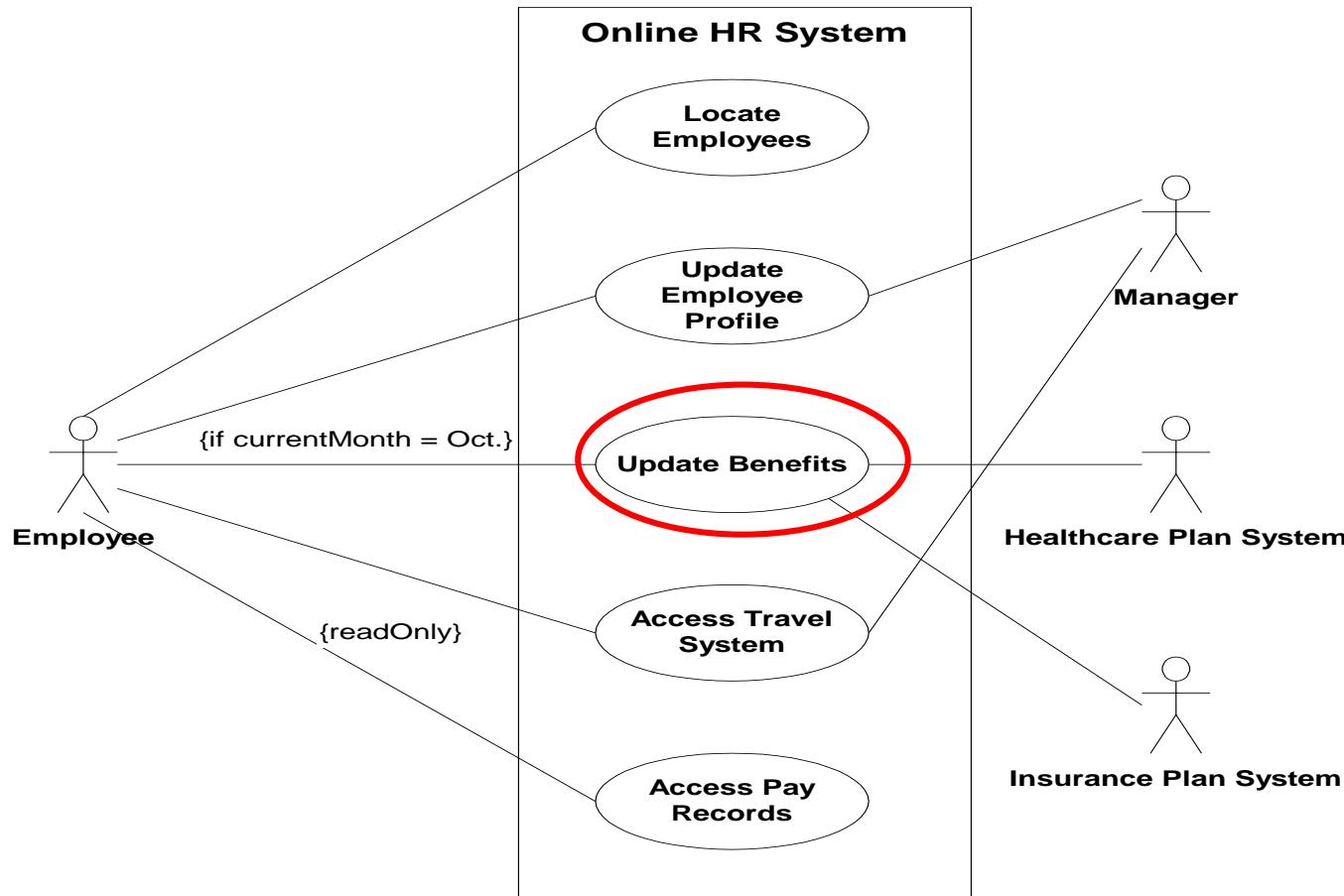
Preporuke u oblikovanju obrazaca uporabe



- Svaki obrazac uporabe mora sadržavati značajan dio uporabe sustava i biti razumljiv ekspertima domene i programerima.
- Ako se obrasci uporabe definiraju tekstom sve imenice i glagole treba koristiti razumljivo i konzistentno kako bi se kasnije mogli definirati ostali (UML) dijagrami.
- Ako su obrasci uporabe nužni, koristi <<include>>.
- Obrasci uporabe su kompletirani a postoje opcije, koristi <<extend>>.
- Dijagram obrazaca uporabe treba sadržavati obrasce uporabe jednake apstraktne razine.
- Uključiti samo zaista potrebne aktore.
- Veliki broj obrazaca uporabe treba organizirati u posebne odvojene dijagrame i pakete.

Primjer: Upravljanje ljudskim resursima

- “on-line” sustav za upravljanje ljudskim resursima





Primjer: Akcija - “osvježi pogodnosti”

Osvježi plan
zdravstvene
zaštite

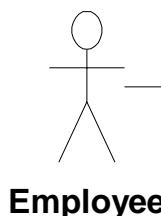
Update Medical
Plan

Osvježi zubnu zaštitu

Update Dental
Plan

Osvježi plan
osiguranja

Update
Insurance Plan



Zaposlenik

Update Benefits

Extension points
benefit options:

after required enrollments

Elect
Reimbursement
for Healthcare

Nadoknada
troškova

<<include>>

<<include>>

<<include>>

<<extend>>

employee requests
reimbursement option

<<extend>>

employee requests
stock purchase option

Točka proširenja

extension point
name and
location

Točka proširenja

extension
condition

Uvjet proširenja

Opcija
kupnje
dionica

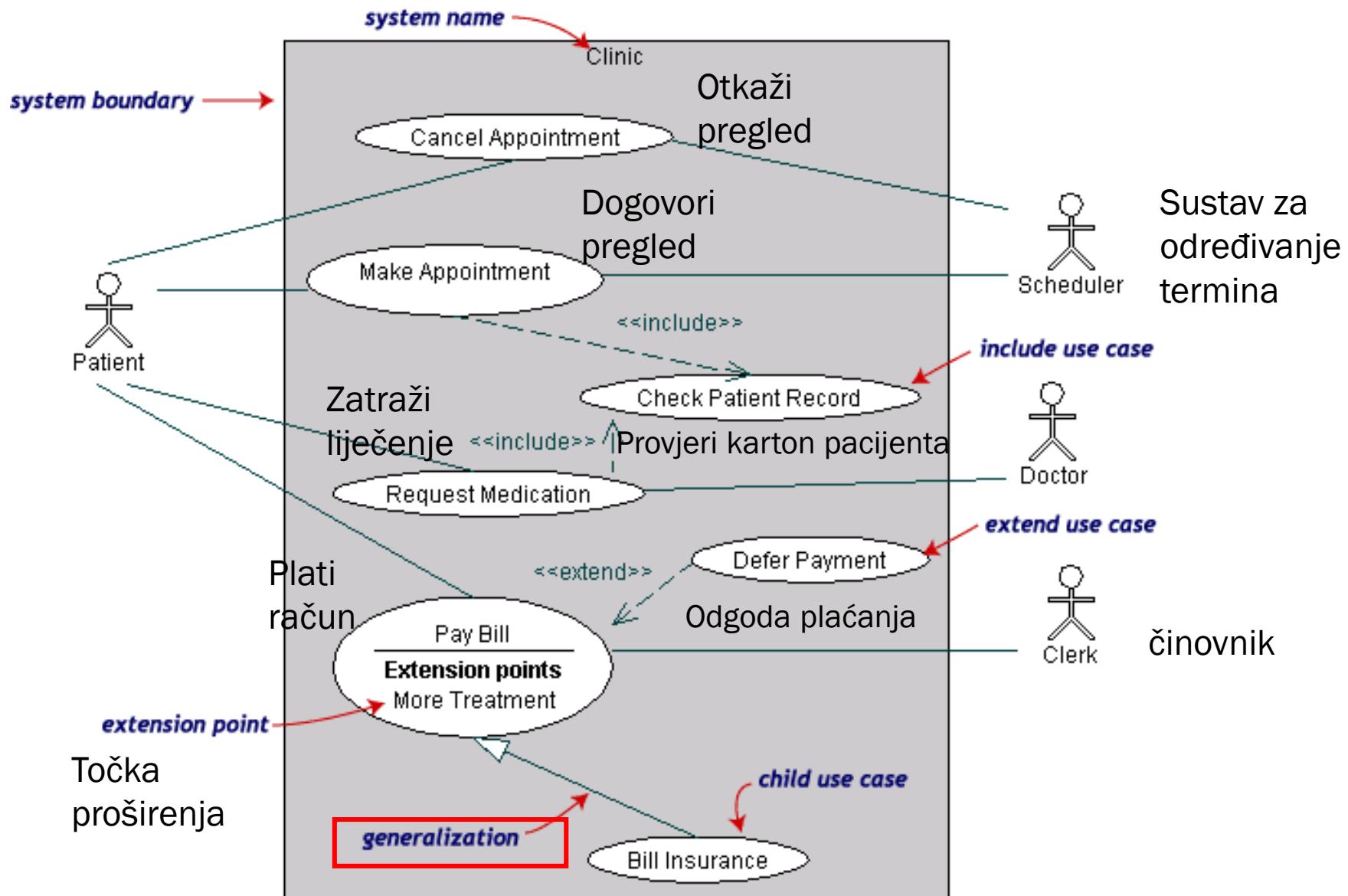


Primjer tekstualnog opisa



- **Aktori:** zaposlenik, baza računa zaposlenika, sustav zdravstvene zaštite, sustav (zdravstvenog) osiguranja.
- **Preduvjeti:**
 - Zaposlenik se prijavio na sustav i odabrao opciju: ‘`update benefits`’.
- **Temeljni slijed:**
 - Sustav dohvaća zaposlenikov račun iz baze.
 - Sustav traži da zaposlenik odabere tip plana zdravstvene zaštite; **include** `Update Medical Plan`.
 - Sustav traži da zaposlenik odabere tip plana zubne zaštite; **include** `Update Dental Plan`.
 - ...
- **Alternativni sljedovi:**
 - Ako tip plana zdravstvene zaštite kojeg je zaposlenik odabrao nije ponuđen u njegovom mjestu stanovanja, sustav traži odabir drugog plana zaštite.

Primjer: Ordinacija



Obrasci uporabe: pravila

- Opisuje tipičnu uporabu sustava
- Mora specifikirati najvažnije funkcionalne zahtjeve
- Detaljna specifikacija može dovesti do otkrivanja novih zahtjeva
- Opisuje značajke važne dizajneru i naručitelju
- “igrokaz”
- Kompletan opis početak, sadržaj i kraj
- Jednostavnost i izravnost
- Jednoznačnost
- Jedna stranica
- Specifikacija pogodna za razvoj i testiranje



Uporaba UML obrazaca uporabe



- UML dijagrami i tekstualni opisi obrazaca uporabe **modeliraju funkcionalne zahtjeve sustava.**
- Temeljeni su na ideji scenarija.
- Služe za izlučivanje zahtjeva prema **pogledu interakcije.**
- Pogodni su za modeliranje velikih i složenih programskih produkata.
- Jednostavni su za razumijevanje ali posjeduju i napredne značajke neophodne za ekspertne analitičare, arhitekte sustava i programere.
- Specificiraju sustave **neovisno o načinu implementacije.**
- Mali skup konstrukcija (10% - 20%) upotrebljavamo u 80%-90% mesta u sustavu.

DIJAGRAMI INTERAKCIJA



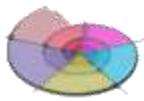
Dinamičke interakcije u sustavu



- Modeliranje ponašanja, engl. *behavioral modeling*
- Detaljniji razvoj i prikaz scenarija u izlučivanju, analizi i dokumentiranju zahtjeva:
- Obrasci uporabe identificiraju individualne interakcije u sustavu.
- Dodatne informacije u inženjerstvu zahtjeva uz obrasce uporabe slijede iz UML dijagrama koji pokazuju aktore uključene u interakciju, entitete (objekte, instancije) s kojima su u interakciji i operacije pridružene tim objektima.
- To su UML dinamički dijagrami interakcija:
 - sekvencijski
 - kolaboracijski.

Interakcije

- **Interakcija** – skup komunikacija između objekata, uključujući sve što utječe na objekte, kao što su pozivi funkcija, stvaranje i brisanje objekata.
- Komunikacije su (djelomično) vremenski uređene!
- **Kada (zašto) modelirati interakcije?**
 - Radi definiranja kako objekti međusobno djeluju (komuniciraju)
 - Radi identifikacije sučelja (objekata/modula)
 - Radi raspodjele zahtjeva (prema dijelovima sustava)

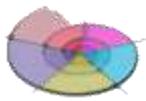


Modeliranje interakcije



■ Preporuke:

- Definirati okolinu interakcije
- Uključi samo relevantna obilježja objekta
- Akcije izražavaj od lijeva na desno te od vrha prema dolje
- Aktivne objekte postavi gore-lijevo, a pasivne dolje-desno
- Sekvencijske dijagrame (obavezno) koristi:
 - za prikaz uređenja među događajima (odvijanju akcija)
 - pri modeliranju sustava za rad u stvarnom vremenu

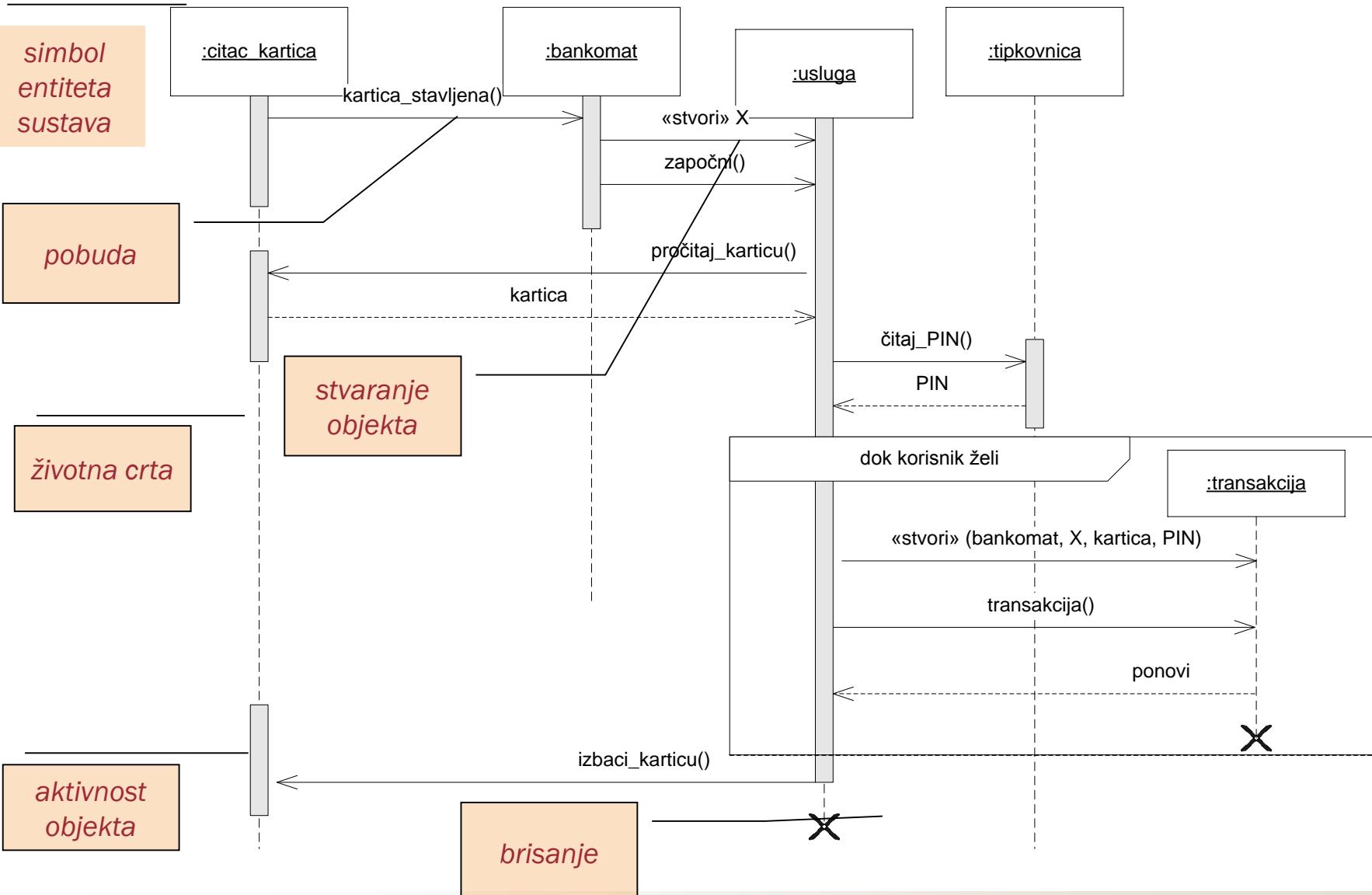


Sekvencijski dijagram



- **Sekvencijski dijagram** je oblik dijagraama interakcije koji pokazuje objekte kao **životne crte** koje idu prema dnu stranice.
- Interakcije u vremenu prikazuju se kao poruke korištenjem strelica od **životne crte** početnog objekta (koji pokreće interakciju/šalje poruku/poziva metodu) do životne crte odredišnog objekta (koji prima poruku...)
- Sekvencijski su dijagrami dobri za prikazivanje (identificiranje) objekata koji **međusobno komuniciraju** i koje poruke pokreću tu komunikaciju.
- Sekvencijski dijagrami **nisu** namijenjeni za prikaz **složene proceduralne logike** (algoritama).

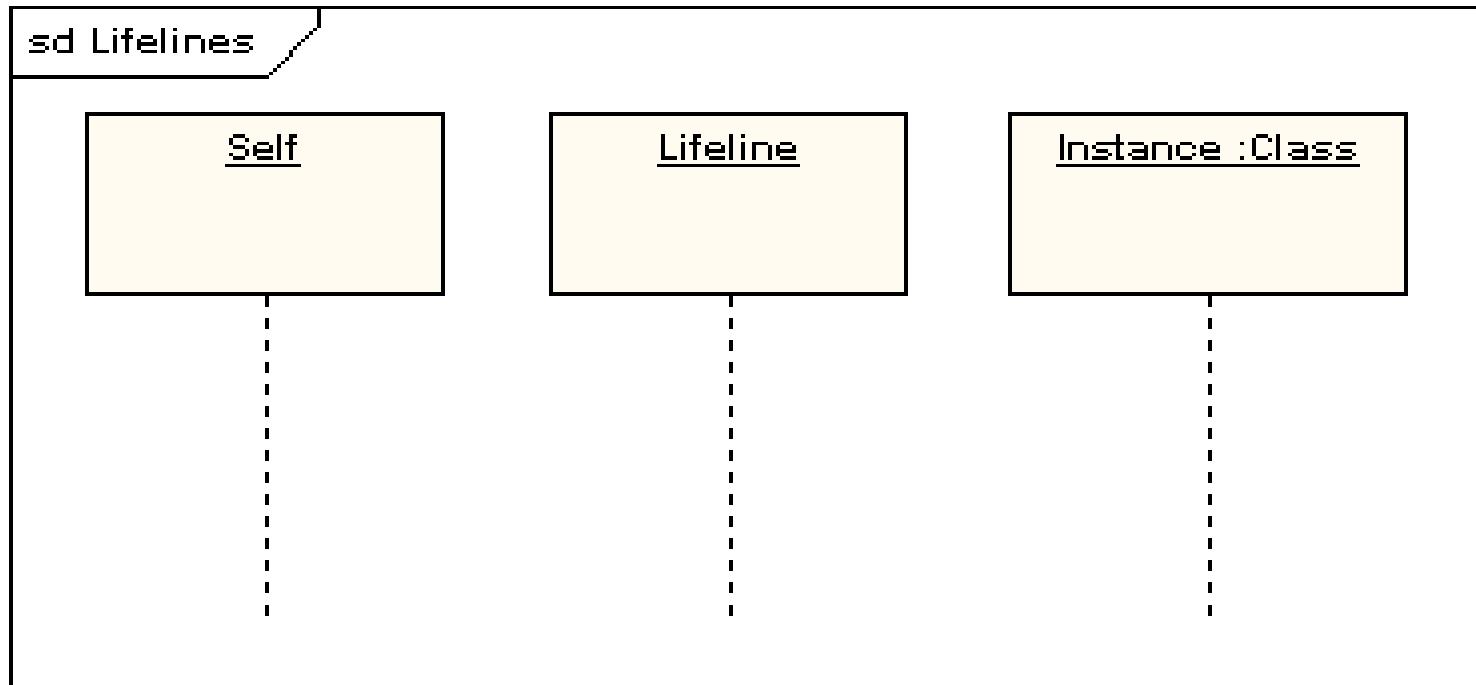
Primjer:

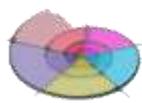




Životne linije (crte)

- engl. *Lifelines*
- Životna crta prikazuje pojedinačnog sudionika u sekveničkom dijagramu. Obično (na početku) sadrži pravokutnik s imenom objekta (entiteta).

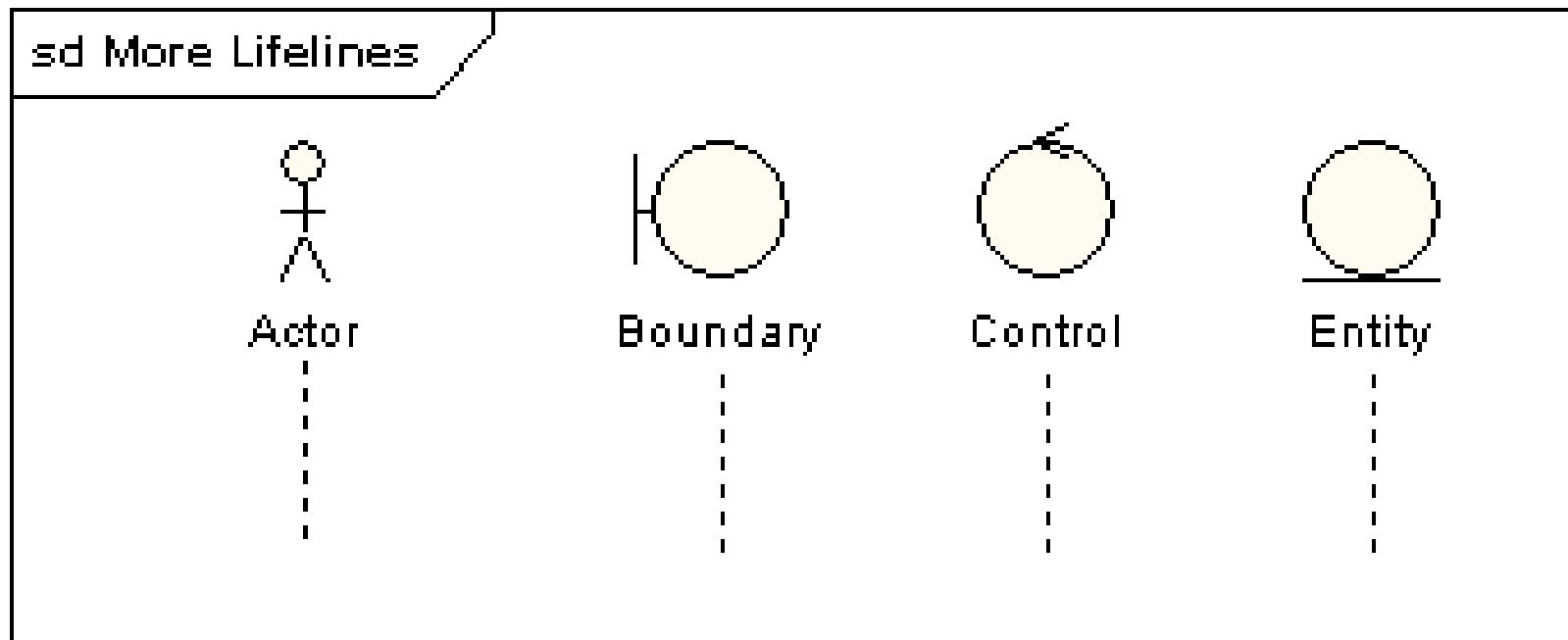




Životne linije (crte)



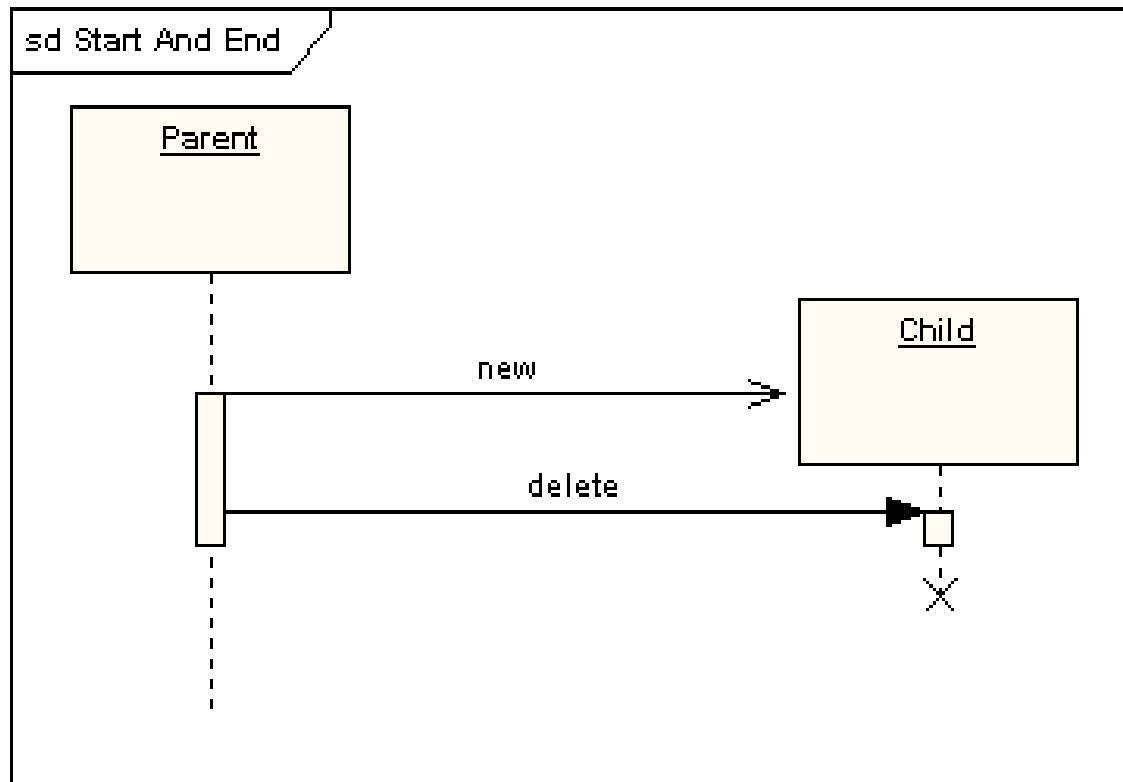
- Aktori se obično koriste kada sekvencijski dijagram pripada određenom obrascu uporabe.
 - Osim aktora, ostali elementi (Granični elementi i elementi upravljanja) mogu imati životnu crtu (vrlo rijetko!).





Početak i kraj životnih crta

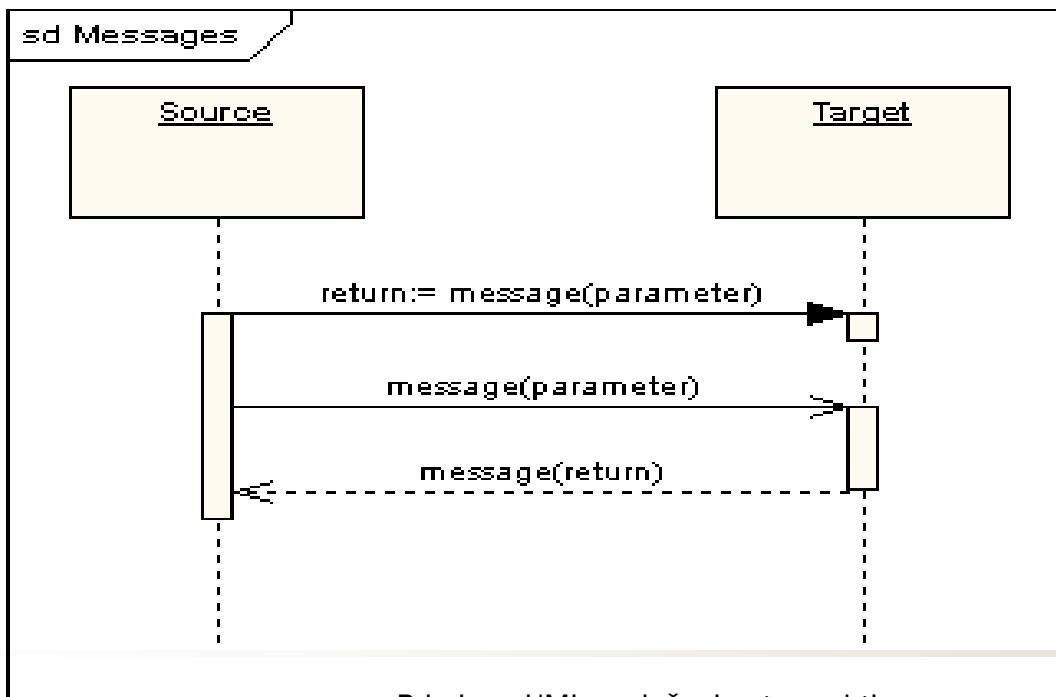
- engl. *Lifeline Start and End*
- Životna crta može biti stvorena ili prekinuta unutar jednog sekvencijskog dijagrama. Navedeni dijagram prikazuje stvaranje i brisanje objekta (nastanak i prekid životne crte).



Poruke

■ engl. *Messages*

- Poruke su prikazane strelicama. Poruke mogu biti **kompletne**, **izgubljene** ili **nađene**; **sinkrone** ili **asinkrone**; **poziv** ili **signal**. U prikazanom dijagramu prva je poruka sinkrona (prikazana ispunjenom strelicom) i kompletna, s **implicitnom povratnom porukom**; druga je asinkrona (obična strelica), a treća je asinkrona povratna poruka (prikazana isprekidanom linijom). (Navedena sintaksa se često ne koristi dosljedno.)



**sinkrona poruka =
pošiljatelj čeka na
odgovor (rezultat)**

Rekurzije

■ Poruka samom sebi - engl. *Self Message*

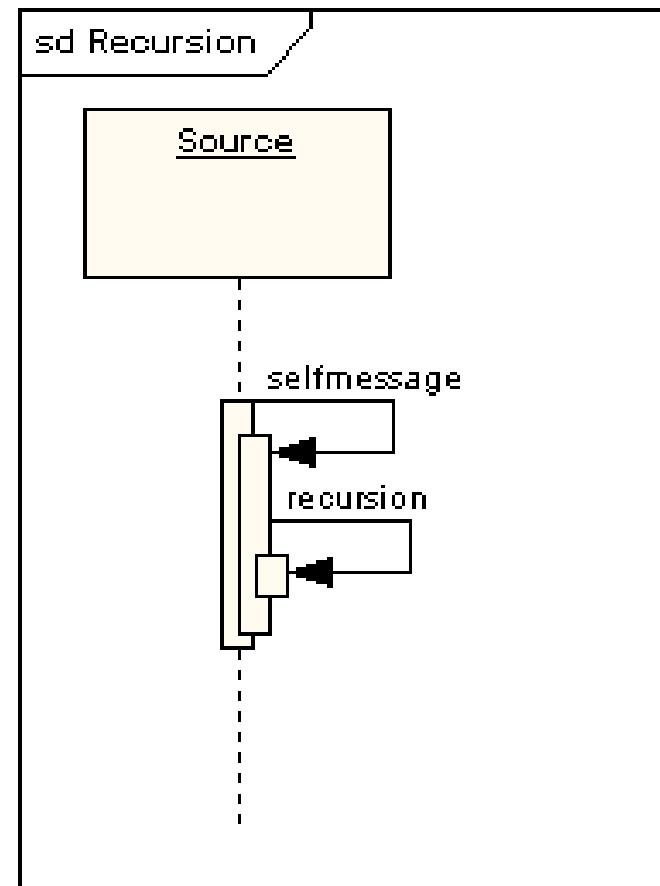
- Poruka samom sebi može predstavljati rekurzivni poziv, ili poziv druge metode istog objekta.

■ Poruka samom sebi:

- različite procedure, funkcije metode u objektu;

■ Rekurstija:

- ista procedura,
- funkcija,
- metoda.





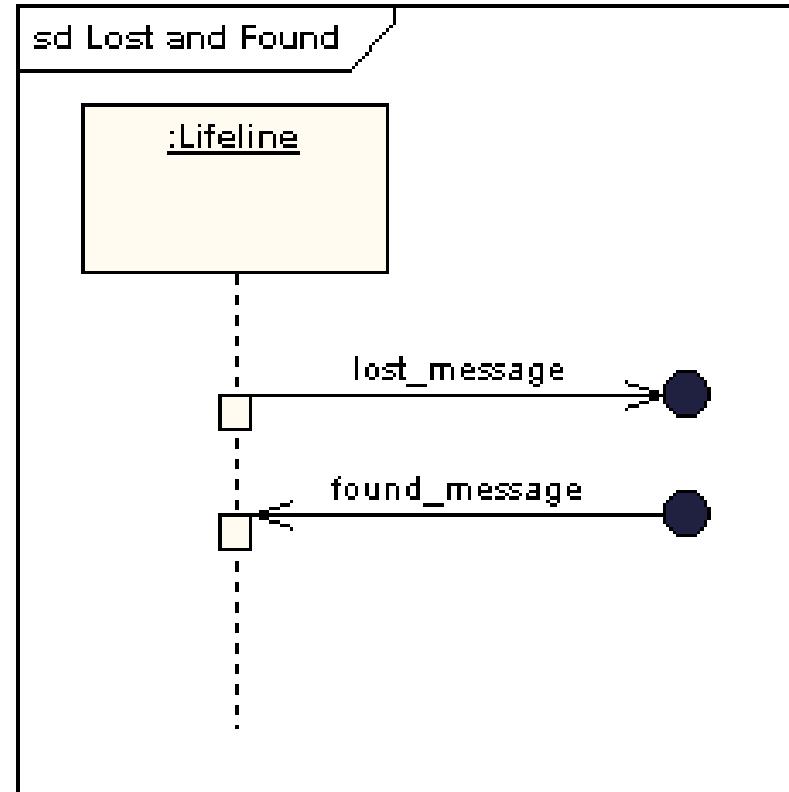
Izgubljene i nađene poruke

Izgubljene poruke

- poslane, ali nisu stigle do odredišta;
- poruke koje se šalju primateljima koji nisu prikazani na trenutnom dijagramu.

Nađene poruke

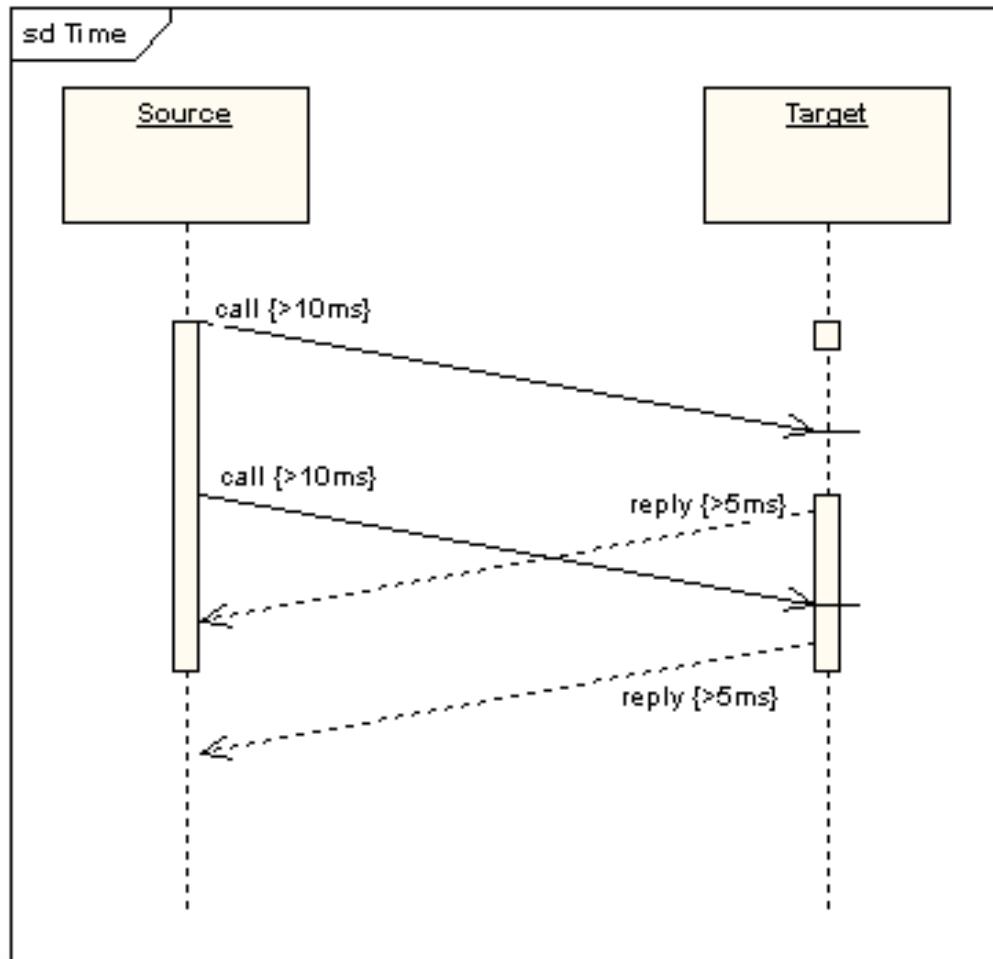
- dolaze od nepoznatog pošiljatelja;
- dolaze od pošiljatelja koji nisu prikazani na trenutnom dijagramu.





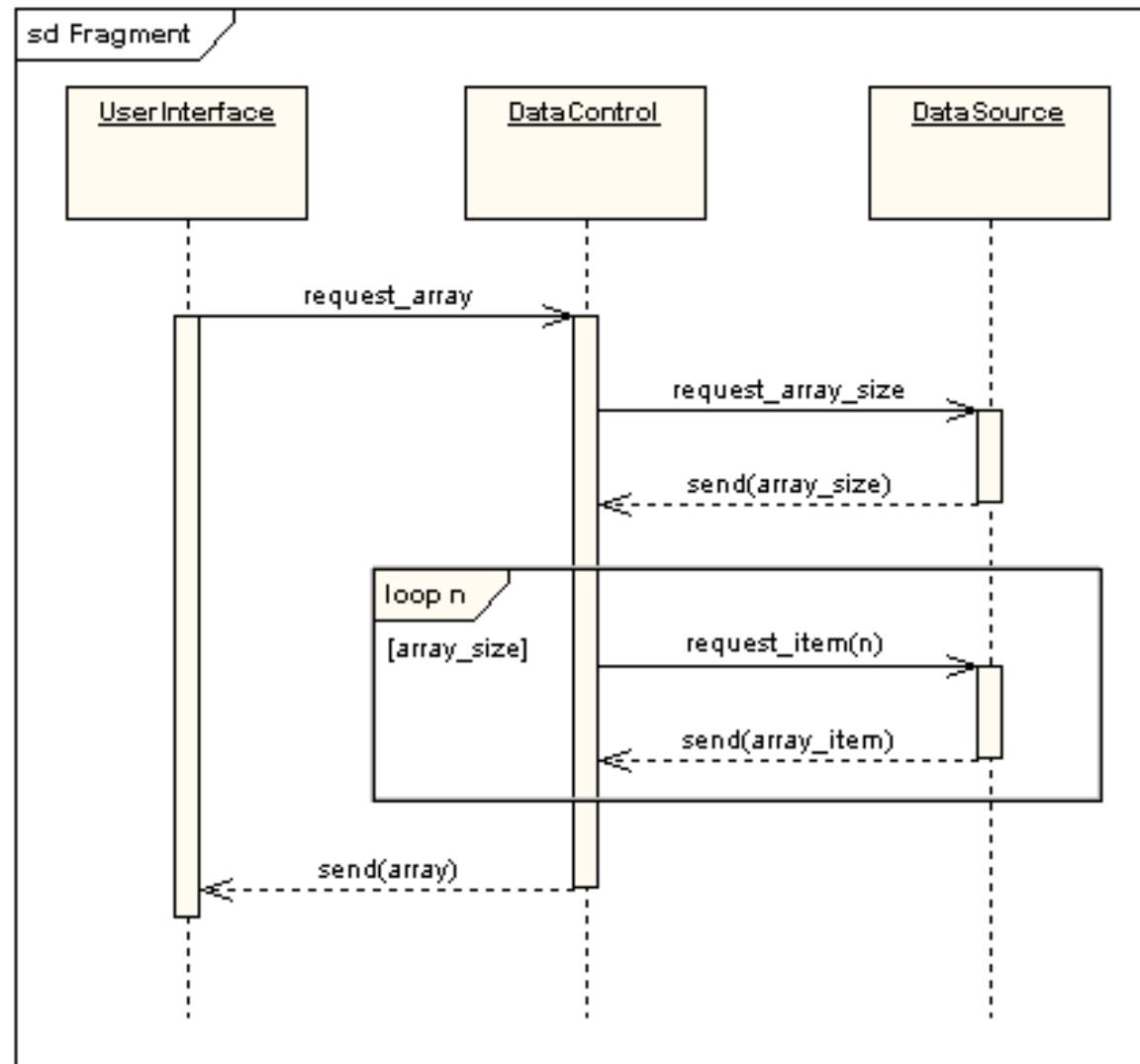
Ograničenja trajanja

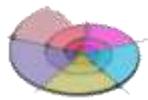
- Uobičajeno se poruka prikazuje kao vodoravna crta.
- Postojanje vremenskog ograničenja obilježava se nagibom.



Petlja

- engl. Loop
- Petljom se može predstaviti niz poruka koje se ponavljaju (određeni broj puta)



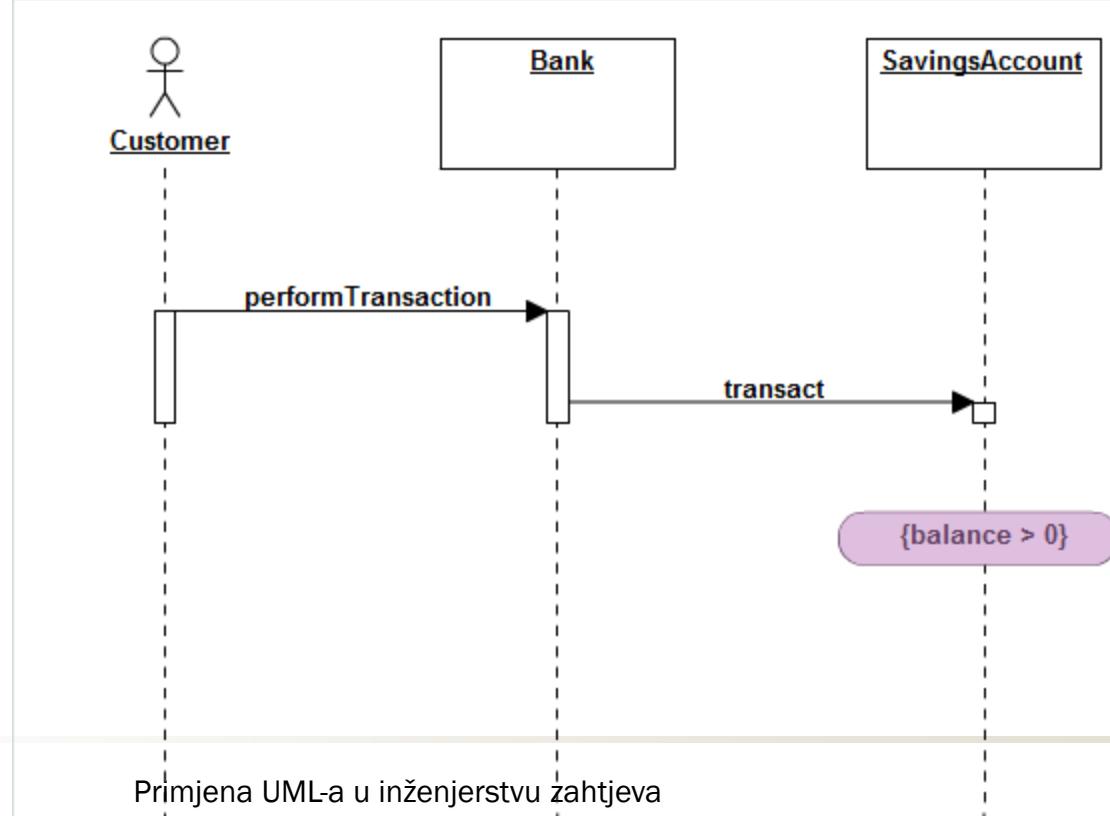


Invariante stanja



- engl. *State Invariant*
- Predstavlja ograničenja na životnoj crtici na kojoj se nalazi.
 - Izračunava se pri izvođenju (engl. *runtime*) i ako nije zadovoljeno poruka se smatra nevaljanom (ona koja prethodi ograničenju).

An account must always have a positive balance





Označavanje poveznica

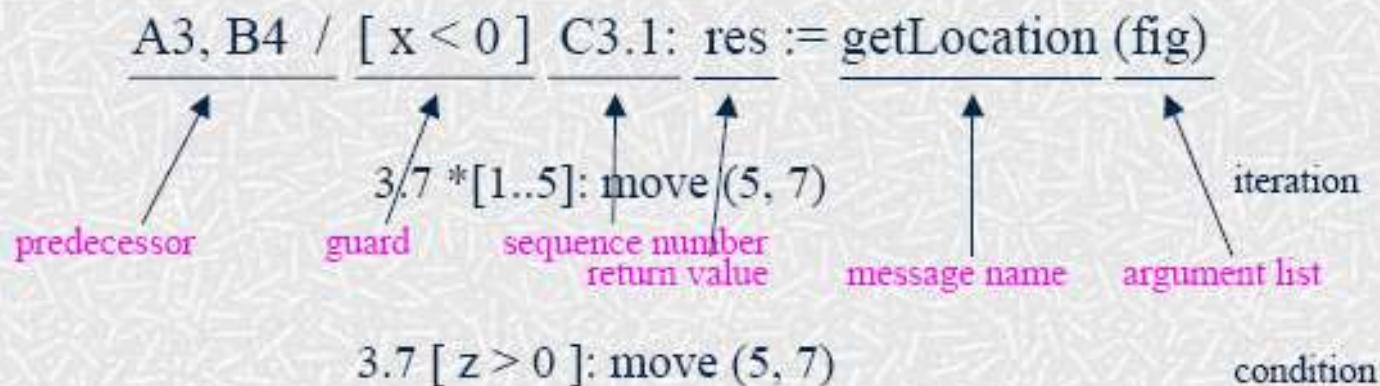
■ engl. Arrow label

prethodnik uvjet oznaka sekvence pov. vrijed. ime poruke lista arg.

predecessor guard-condition sequence-expression return-value := message-name argument-list

move (5, 7)

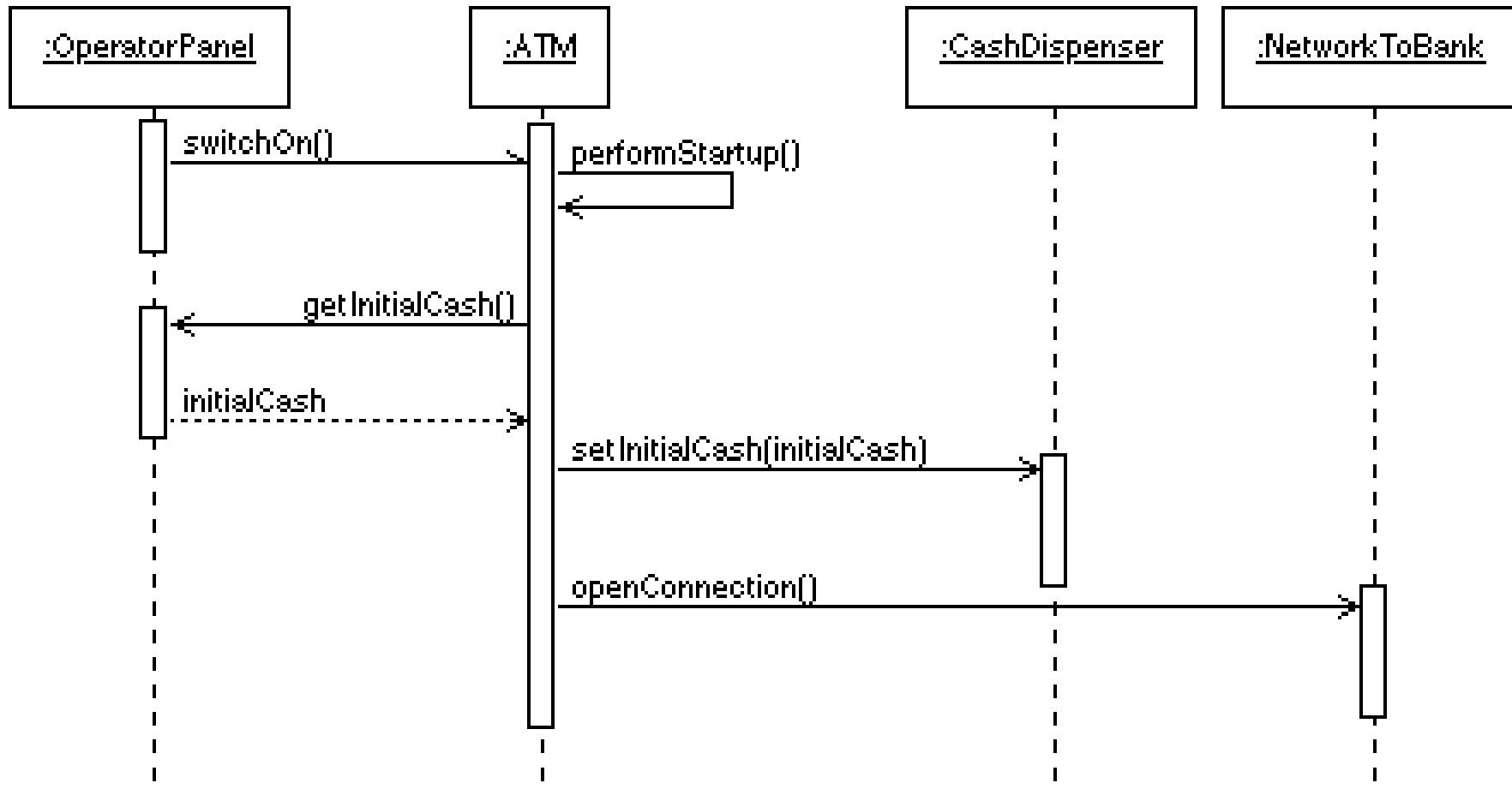
3.7.4: move (5, 7)

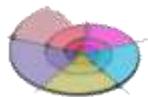




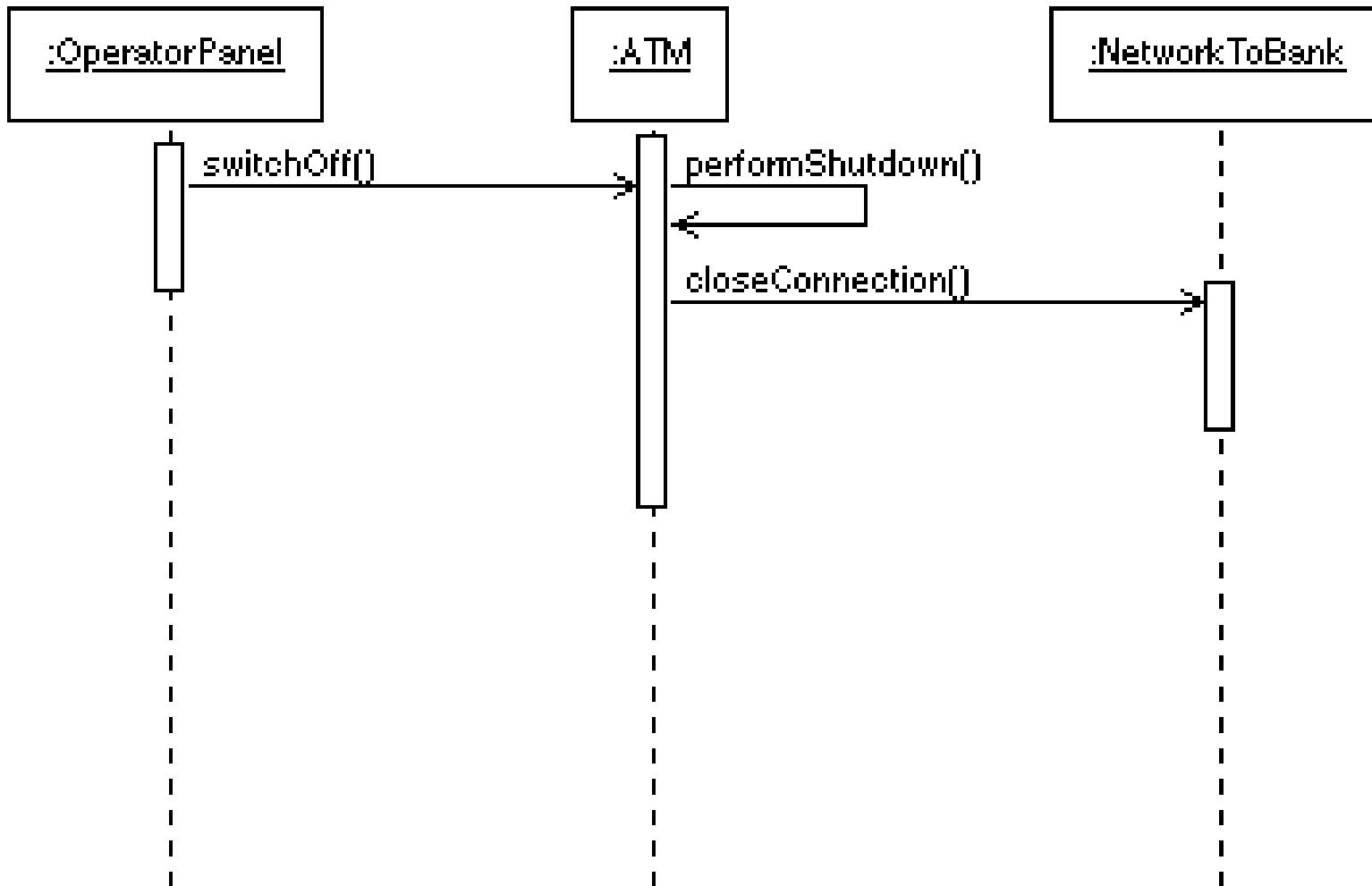
Primjer: bankomat

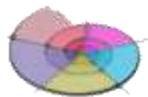
- Sekvencijski dijagram pokretanja



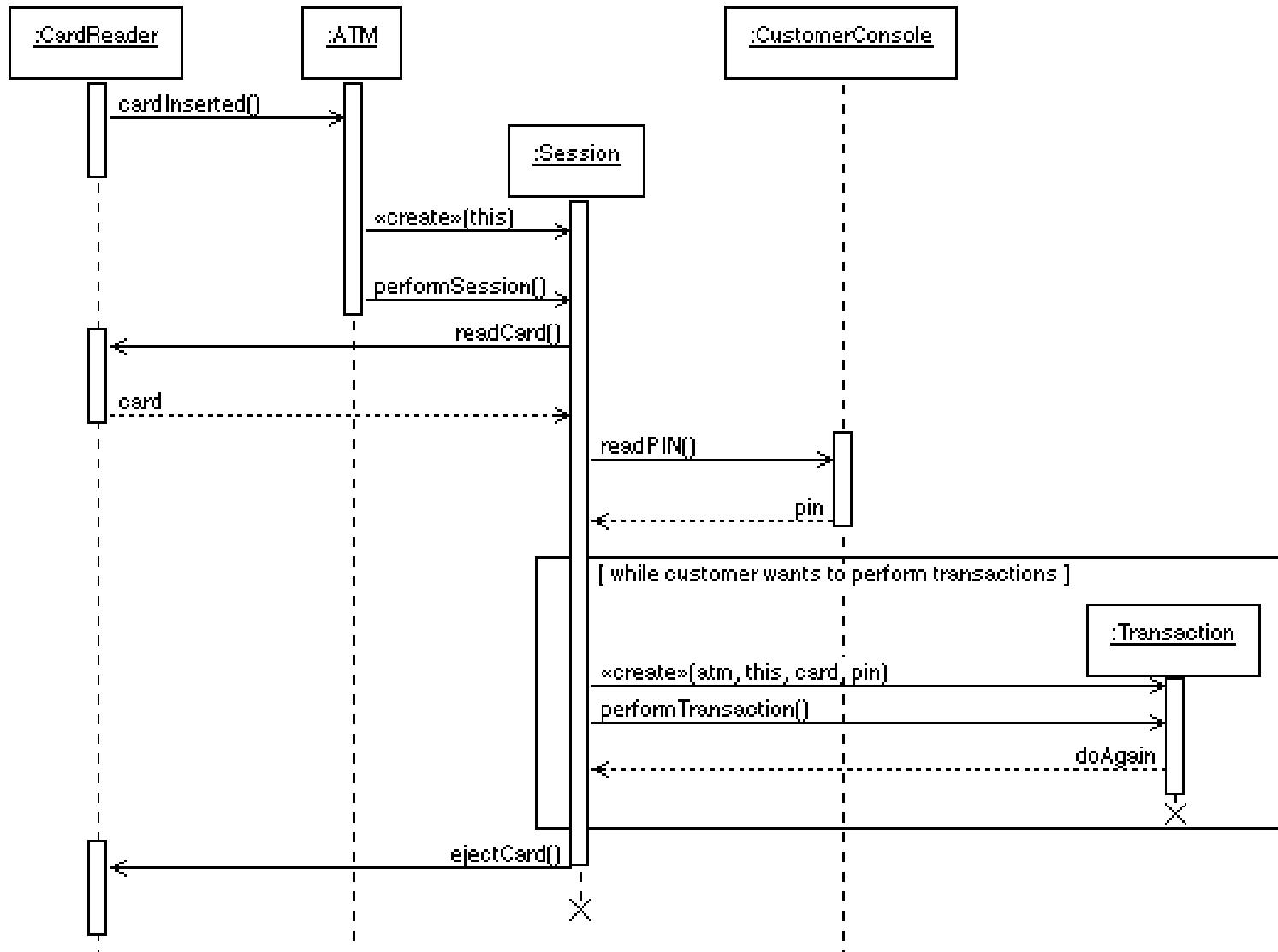


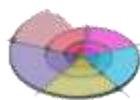
Sekvencijski dijagram gašenja



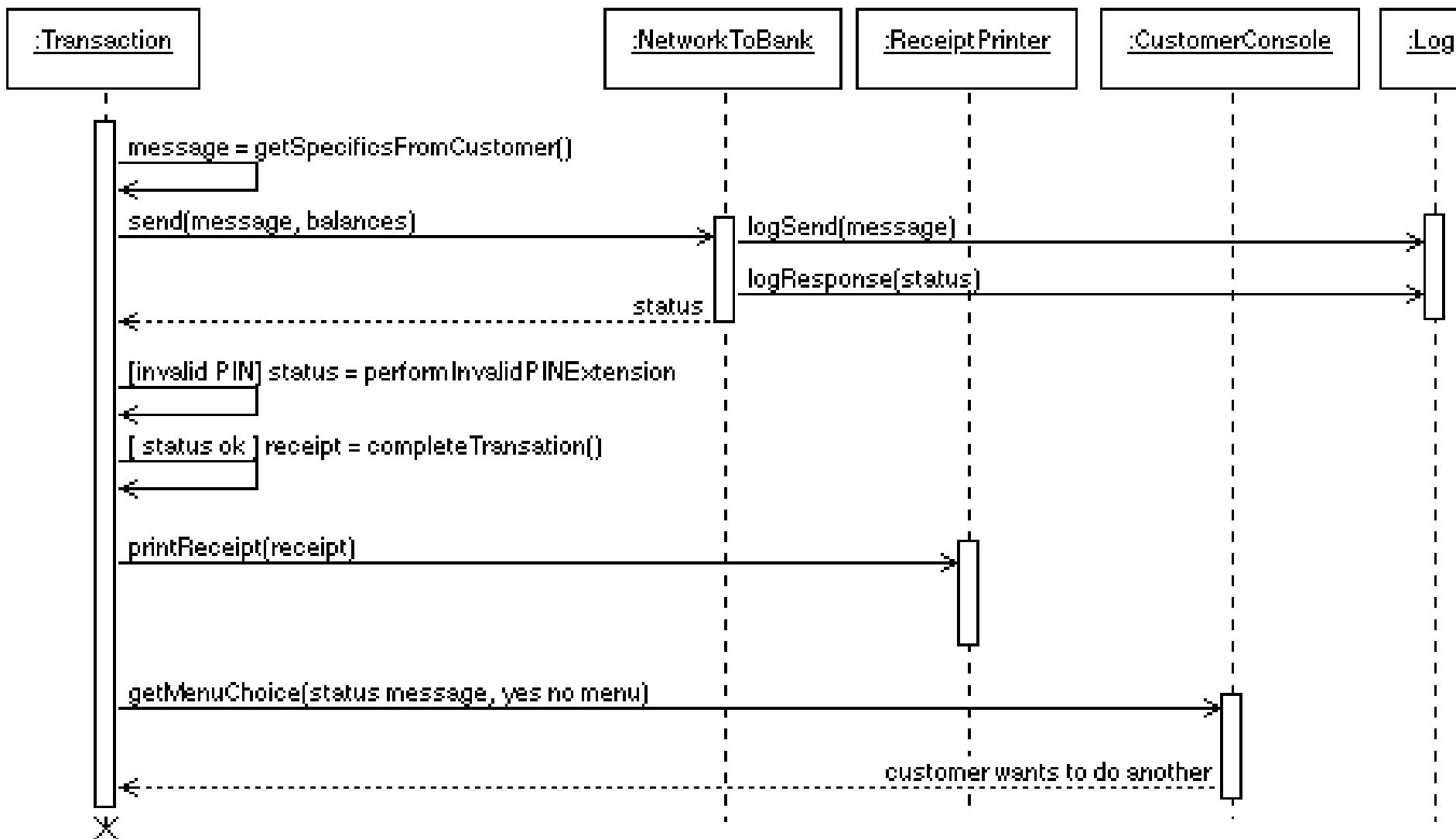


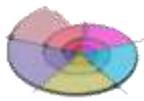
Sekvencijski dijagram usluge





Sekvencijski dijagram transakcije





Primjer: Promjena rute leta



■ Aktori:

- putnik, baza računa klijenta (s planom puta), rezervacijski sustav avio kompanije.

■ Preduvjeti:

- Putnik se prijavio na sustav i odabroao opciju "promjena leta".

■ Temeljni tijek transakcija

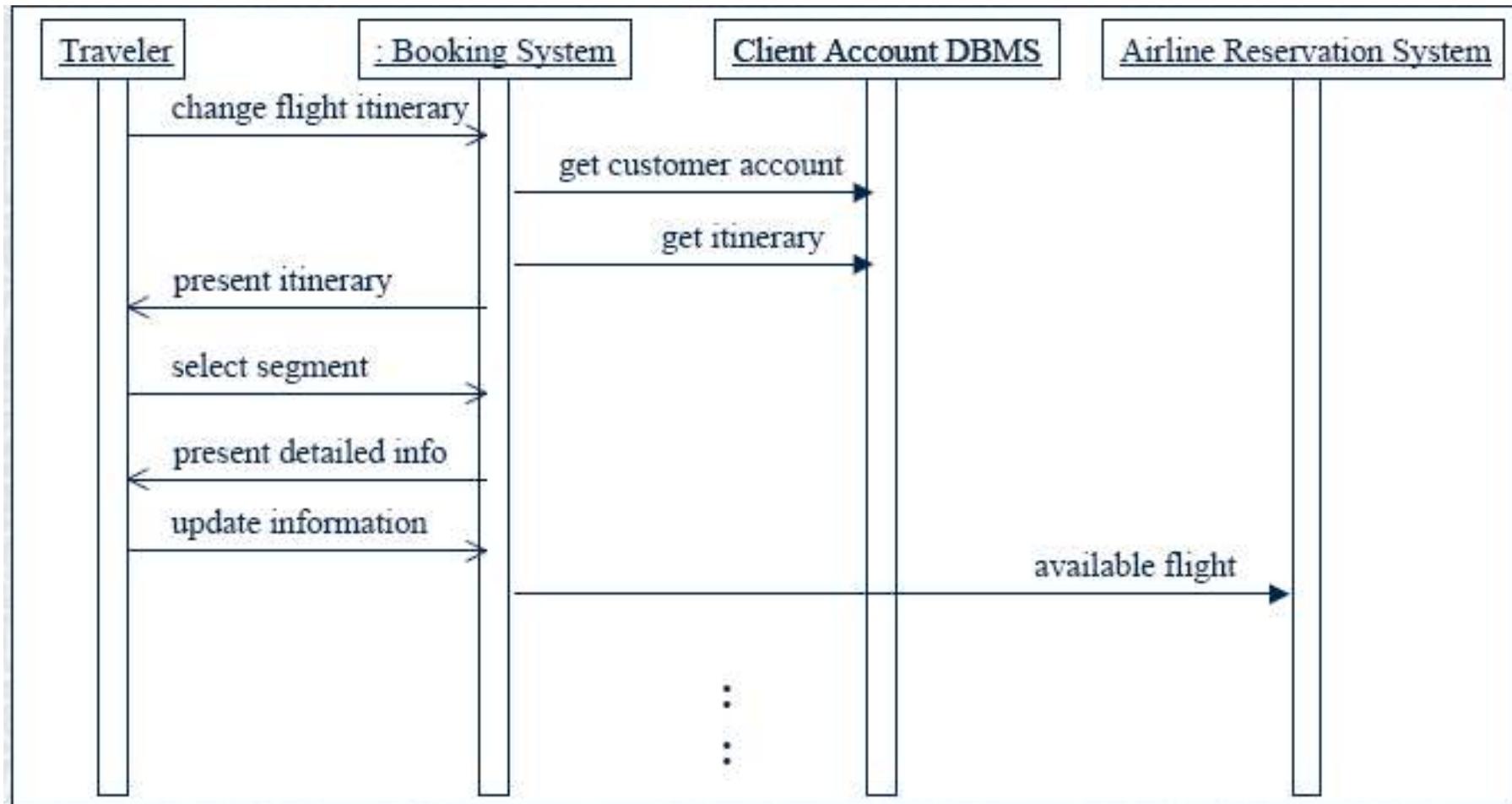
- Sustav dohvaća putnikov bankovni račun i plan puta iz baze.
- Sustav pita putnika da odabere dio plana puta koji želi mijenjati; putnik selektira segment puta.
- Sustav pita putnika za novi odlaznu i dolaznu destinaciju; putnik daje traženu informaciju.
- Ako je let moguć, tada ...
- ...
- Sustav prikazuje sažetak transakcije..

■ Alternativni tijek transakcija

- Ako let nije moguć, tada ...



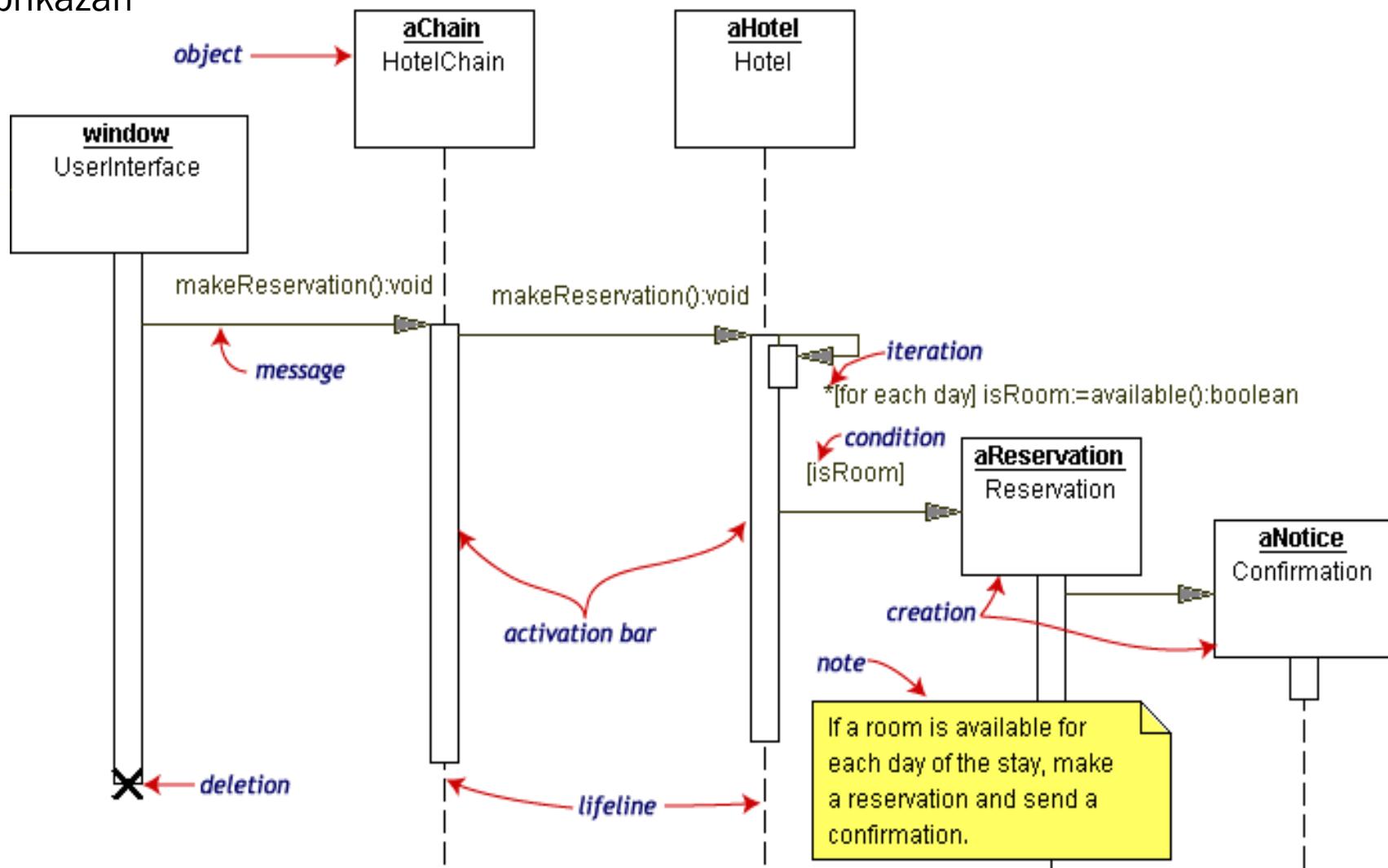
Sekvencijski dijagram

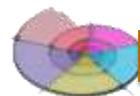




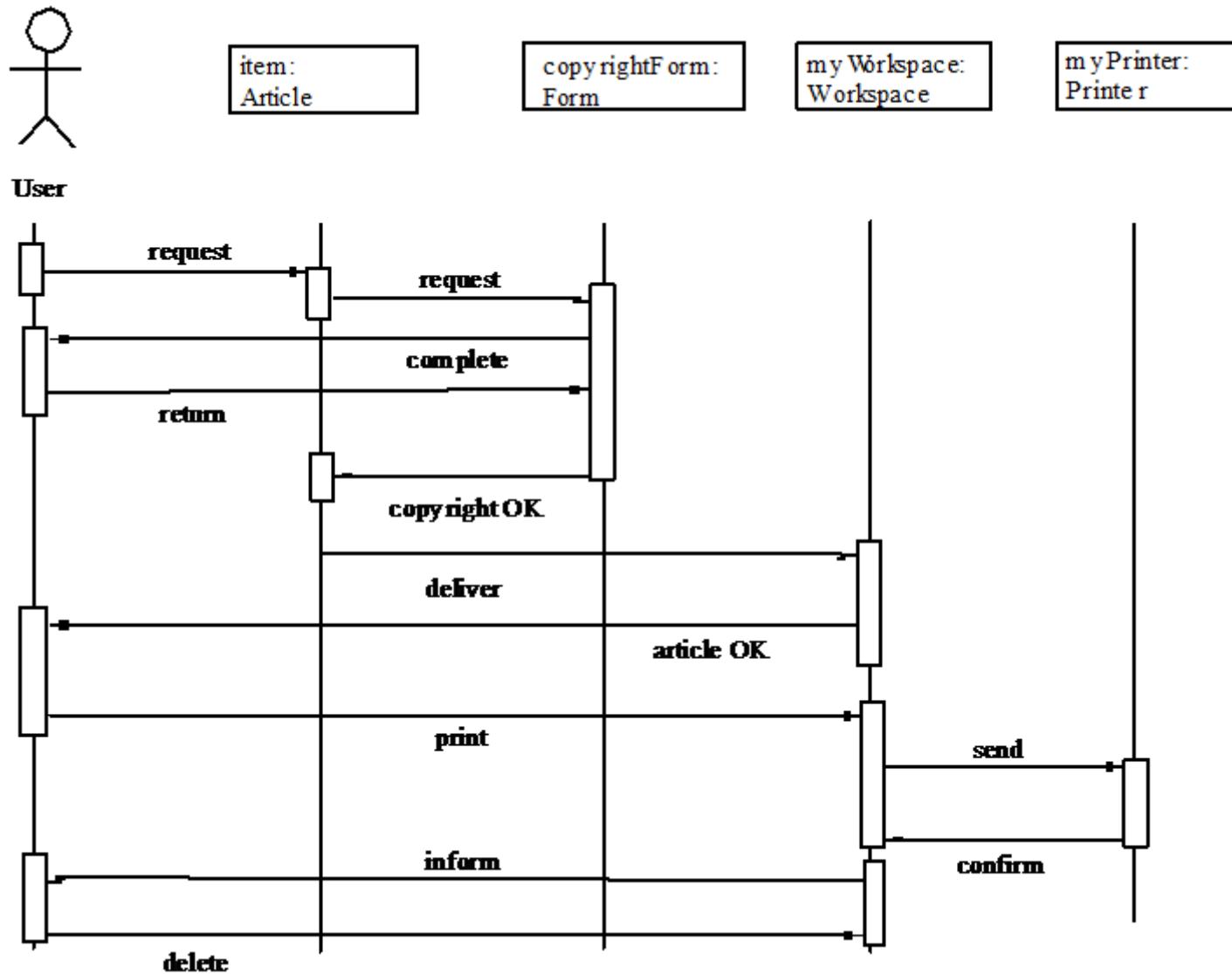
Primjer: Rezervacija hotela

- Početni objekt koji započinje niz razmjena poruka je `Reservation window` nije prikazan





Primjer: LIBSYS - dijagram sekvenci za ispis članka





Sekvencijski dijagram



- **Sekvencijski dijagram** pogodan za opis ponašanja interakcijama
- Interakcije u vremenu prikazuju se kao poruke korištenjem strelica od **životne crte** početnog objekta (koji pokreće interakciju/šalje poruku/poziva metodu) do životne crte odredišnog objekta (koji prima poruku...)
- Dobri za prikazivanje i identificiranje objekata koji **međusobno komuniciraju** i koje poruke pokreću tu komunikaciju.
- Pomažu kod pronalaženja nedostajućih objekata
- Vremenski zahtjevan proces izgradnje
- Omogućavaju prikaz paralelnosti u sustavu
- Sekvencijski dijagrami **nisu** namijenjeni za prikaz **složene proceduralne logike** (algoritama).



Zaključci

- UML osigurava izvrsnu notaciju za opis različitih potreba u razvoju programske potpore
- Izražajan ali i složen jezik
- Zamke: mogućnost izrade nečitljivih i pogrešno interpretiranih modela
- Obrasci uporabe pogodni za opis funkcionalnih modela
- Sekvencijski dijagrami pogodni za opis dinamičkih modela

Diskusija

-
-
-
-

Oblikovanje programske potpore

2014./2015.

Procesi programskog inženjerstva



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





Literatura

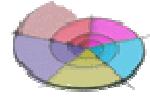


- Sommerville, I.: Software Engineering, Addison-Wesley, 2007.
- Lethbridge, T. C., Laganière: Object-Oriented Software Engineering: Practical Software Development Using UML and Java, McGraw-Hill, 2005.
- Pollice, G., Augustine L.; et al: Software Development for Small Teams: A RUP-Centric Approach, Addison-Wesley, 2003



Tema

- Upoznavanje i opis modela procesa programskog inženjerstva.
- Uloga iteracija u modelima procesa programskog inženjerstva.
- Generičke aktivnosti u procesima programskog inženjerstva:
 - osnove modela za analizu zahtjeva, oblikovanje programske potpore, validaciju i evoluciju
- Prikaz modela *Unified Process*
- Svojstva CASE tehnologija u procesima programskog inženjerstva



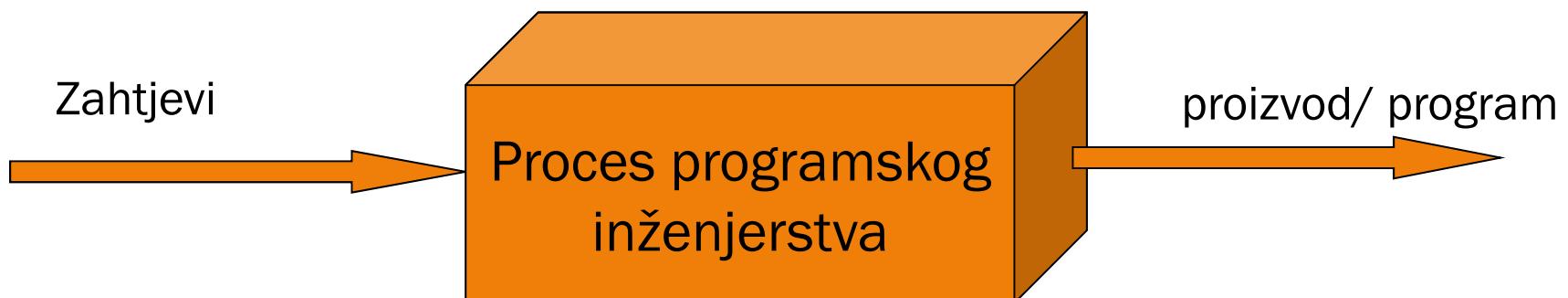
Procesi programskog inženjerstva

- *Podsjetnik:*
- Proces PI – strukturirani skup aktivnosti koji čini okvir neophodan za izvođenje sustavnog plana razvoja i oblikovanja programske potpore
- Generičke aktivnosti :
 - specifikacija;
 - oblikovanje i implementacija;
 - validacija i verifikacija ;
 - evolucija.
- Model procesa programskog inženjerstva je apstraktna reprezentacija procesa.
 - predstavlja opis procesa iz određene perspektive.



Uloga procesa

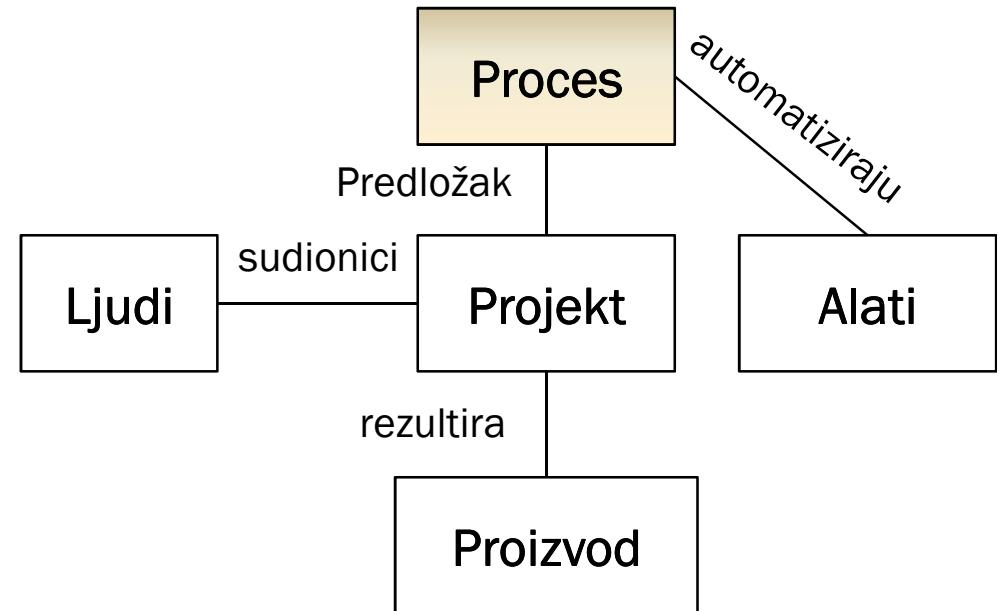
- Proces razvoja programske potpore osigurava:
 - potrebne informacije
 - u trenutku kada su potrebne
 - u upotrebljivom obliku
 - ni previše ni premalo
 - jednostavno pronalaženje
- Proces definira **TKO** radi **ŠTO, KADA** i kako postići željeni cilj.





Pogledi na razvoj PP

- Upravljenje projektima PP (engl. *Software Project Management*) je krovna aktivnost u području programskog inženjerstva
- Učinkovito programsko inženjerstvo fokusira se na:
 - ljudi – engl. *People*
 - najvažniji
 - projekt – engl. *Project*
 - rezultira proizvodom
 - proizvod – engl. *Product*
 - nije samo kôd
 - proces – engl. *Process*
 - upravlja projektom
 - alate – engl. *Tools*





Model životnog ciklusa



- engl. *Lifecycle model*
- U svrhu ostvarenja cilja provodi se niz povezanih aktivnosti koje nazivamo fazom životnog ciklusa.
- Model životnog ciklusa opisuje faze od začetka projekta do kraja životnog vijeka proizvoda
 - opisuje odnose glavnih točaka procesa i rezultata
 - određuje redoslijed odvijanja aktivnosti
- Životni ciklus razvoja programske potpore
 - engl. *software development life-cycle, software development process, software life cycle, software process*
 - predstavlja standardizirani format procesa programske potpore.
- Primjer standarda:
 - ISO/IEC 12207 *Systems and software engineering – Software life cycle processes*



Česti modeli

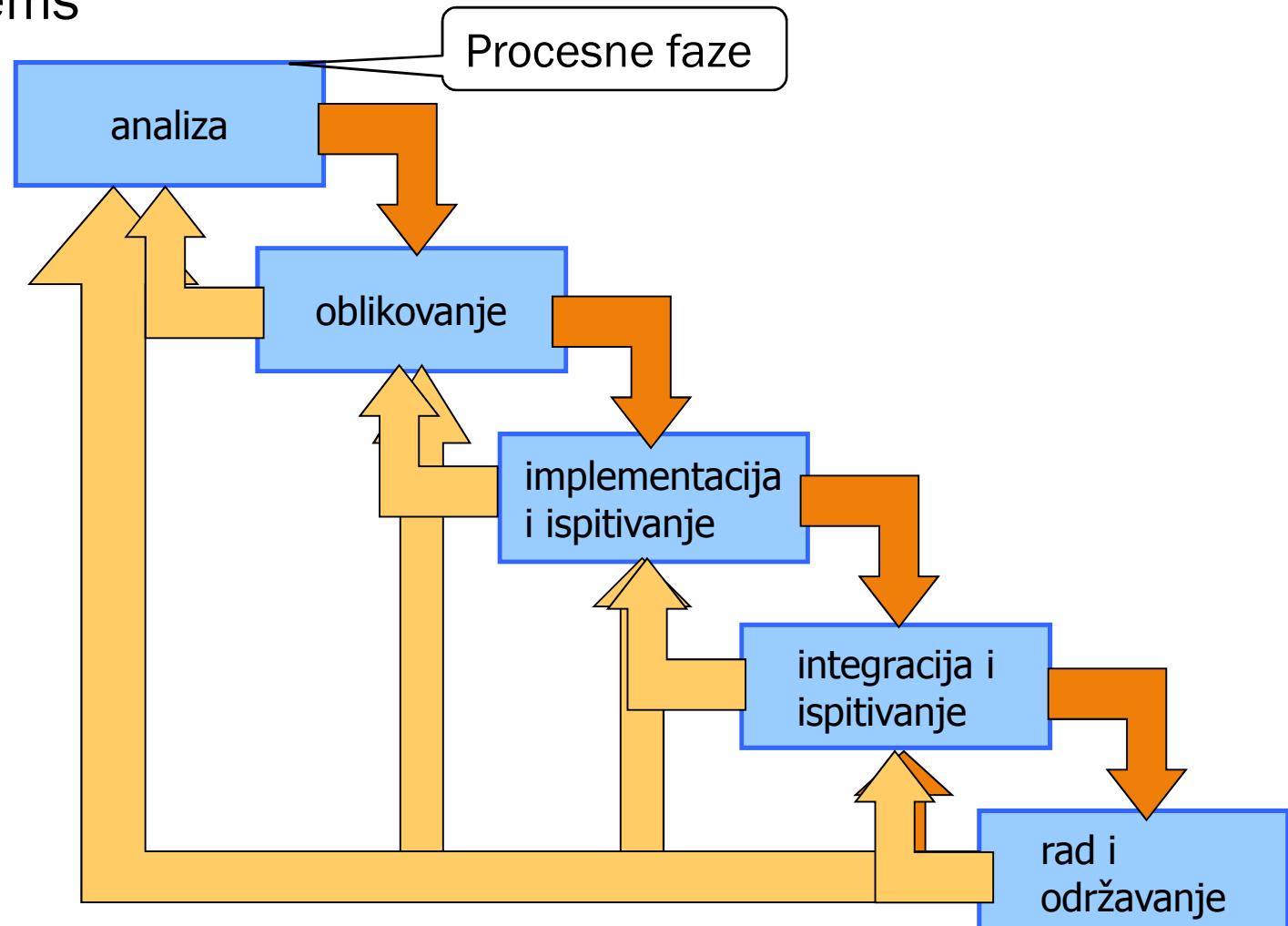
- Ad hoc model *engl. Code and fix life-cycle model*
- **Vodopadni model** *engl. Waterfall life-cycle model – temeljni generički*
 - odvojene i specifične faze specifikacije i razvoja
- Prototipni model *engl. Rapid prototyping life-cycle model*
- **Evolucijski model** *engl. Incremental/evolutionary – temeljni generički*
 - specifikacija, razvoj i validacija su isprepleteni
- **Komponentno usmjeren – temeljni generički**
 - sustav se gradi od postojećih komponenata
- **Unificirani proces** *engl. Unified process (UP)*
 - zasnovan na oblikovanju uporabom modela
 - *engl. Model Based Design*
- **Ubrzani proces** *engl. Agile processes*

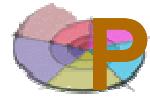
- Ovo su samo primjeri češće upotrebljavanjih modela od stotine postojećih te njihovih podvarijanti



Vodopadni model programskog inženjerstva

- 1970 Winston W. Royce "Managing the development of large software systems"



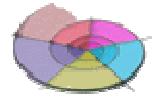


Procesne faze u vodopadnom modelu



- Analiza zahtjeva i definicije
- Razvoj i oblikovanje sustava i programske potpore
- Implementacija i ispitivanje (testiranje) modula
- Integracija i ispitivanje sustava
- Uporaba sustava i održavanje

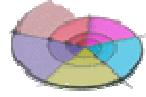
- Temeljna značajka vodopadnog modela:
 - pojedina faza ***se mora dovršiti*** prije pokretanja nove faze!!!



Pretpostavke vodopadnog modela



- Zahtjevi poznati prije oblikovanja
- Zahtjevi se rijetko mijenjaju
- Korisnik zna što treba i nisu mu potrebna pojašnjenja
- Oblikovanje se može provoditi odvojeno i rijetko dovodi do pogrešaka
- Tehnologija sama po sebi rješava neke zahtjeve
- Sustavi nisu složeni



Problemi vodopadnog modela

- Temeljni nedostatak vodopadnog modela
 - poteškoće ugradnje promjena nakon što je proces pokrenut
 - nefleksibilna podjela projekta u odvojene dijelove čini implementaciju promjena koje zahtijeva kupac vrlo teškim.
- Model je prikladan samo ako su zahtjevi dobro razumljivi i eventualne promjene svedene na minimum.
- Vrlo malo poslovnih sustava ima stabilne zahtjeve.
- Vodopadni model se uglavnom koristi za velike inženjerske projekte gdje se sustav razvija na više odvojenih mesta.



Evolucijski model razvoja i oblikovanja

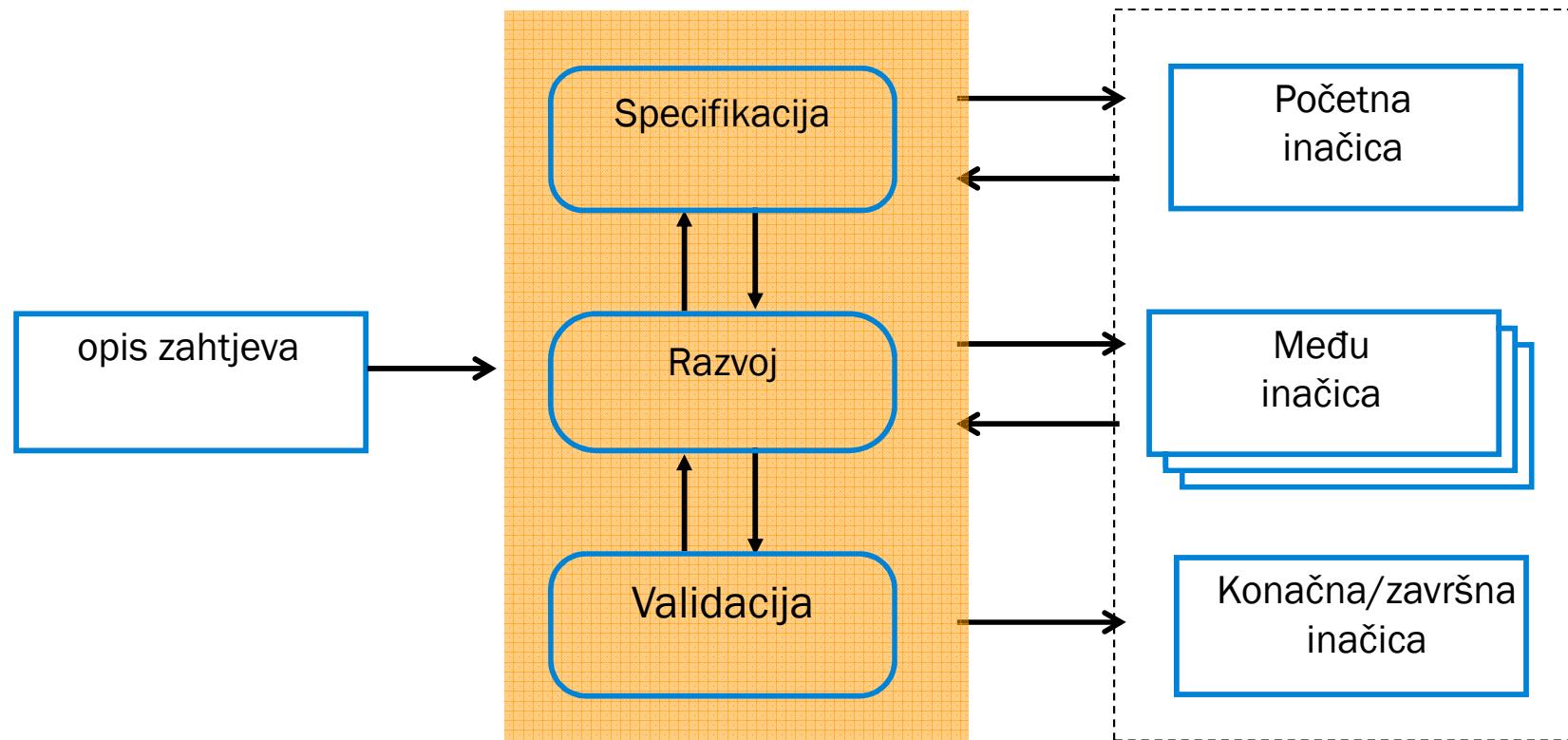


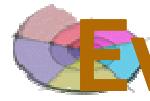
- Dva uobičajena postupka:
- *Metoda odbacivanja prototipa*
 - cilj je razumijevanje zahtjeva sustava.
 - započinje se s *grubo definiranim zahtjevima* koji se tijekom postupka razjašnjavaju u smislu što je doista potrebno.
 - prototip NIJE sustav koji radi
 - sučelja i kosturi stvarne funkcionalnosti
 - zašto korisnici misle da ga mogu odmah upotrijebiti?
- *Istraživački razvoj i oblikovanje*
 - cilj ovakvog pristupa je kontinuiran rad s kupcem na temelju inicijalne specifikacije.
 - započinje se s *dobro definiranim zahtjevima* a nove funkcionalnosti se dodaju temeljem prijedloga kupca.



Evolucijski model razvoja i oblikovanja

Paralelne aktivnosti
- ponavljaju se tijekom razvoja





■ *Problemi:*

- proces razvoja i oblikovanja nije jasno vidljiv.
- sustavi su često vrlo loše strukturirani.
- često su potrebne posebne vještine
 - npr. brzi razvoj prototipa – engl. *Rapid System Prototyping*

■ *Primjenljivost:*

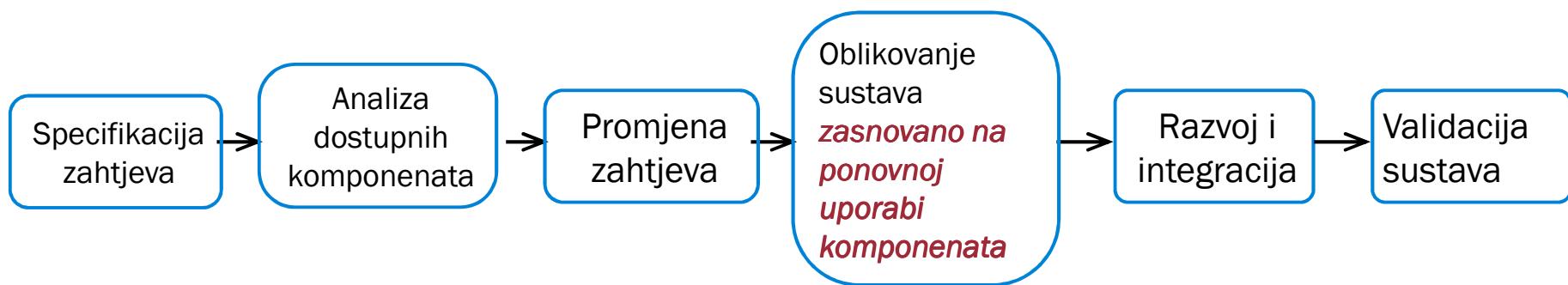
- za male i srednje interaktivne sustave
- za dijelove velikih sustava
 - npr. korisničko sučelje
- za sustave s kratkim vijekom trajanja



- engl. *Component-based software engineering* - CBSE
- Sustav se integrira višestrukom uporabom postojećih komponenata ili uporabom komercijalnih, gotovih komponenata (engl. *commercial-of-the-shelf COTS*).
- Stupnjevi procesa:
 - specifikacija i analiza zahtjeva
 - analiza komponenata
 - modifikacija zahtjeva
 - oblikovanje sustava s višestrukom uporabom komponenata (engl. *reuse*)
 - razvoj i integracija
- S povećanom standardizacijom komponenata CBSE
 - zrelo tržište
 - [International Symposium on Component Based Software Engineering \(CBSE\)](#)



- Višestruka uporaba komponenata
- engl. *Reuse-oriented development*





Unificirani proces

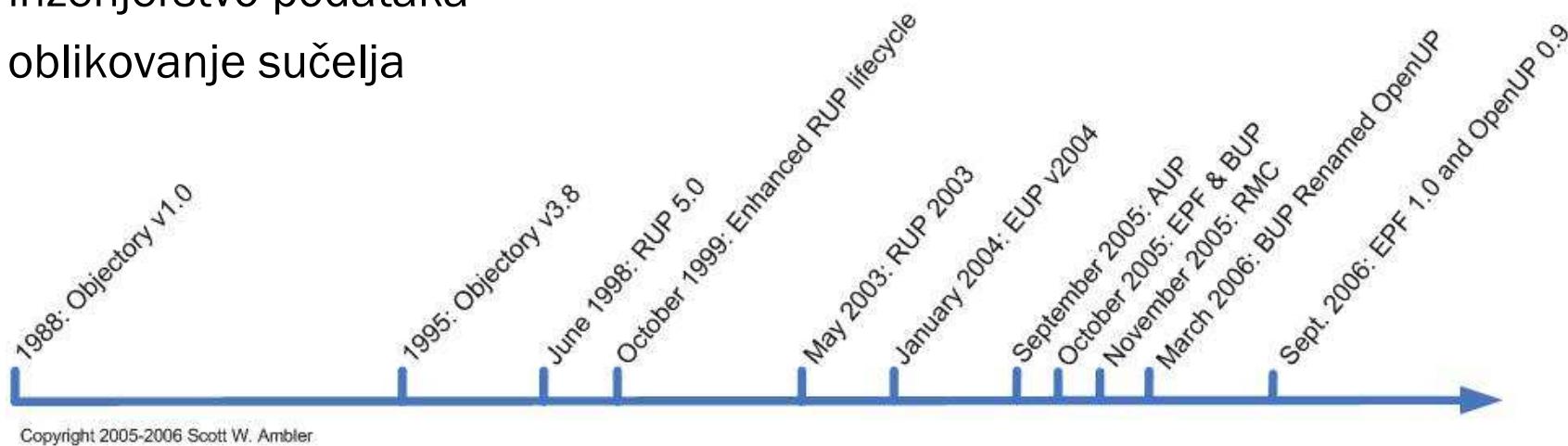


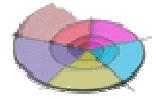
- engl. *Unified Process*
- Model procesa programskog inženjerstva izведен na temelju jezika za modeliranje UML-a (engl. *Unified Modeling Language*) i pridruženih aktivnosti.
- Svojstva
 - priznaje utjecaj korisnika
 - sugerira evolucijski pristup
 - podržava OO
 - prilagodljiv
- Najčešće opisan kroz tri perspektive:
 - *dinamička* perspektiva koja pokazuje slijed faza kroz vrijeme.
 - *statička* perspektiva koja pokazuje aktivnosti procesa.
 - *praktična* perspektiva koja sugerira aktivnosti kroz iskustvo i dobru praksu.



Povijesni razvoj

- 1999. Booch, Jacobson, Rumbaugh: “The Unified Software Development Process”
 - objedinjivanje tri različite metodologije ⇒ **Unified Process**
- Objedinjuje postupke:
 - inženjerstvo poslovnog procesa
 - rukovanje zahtjevima
 - rukovanje oblikovanjem i promjenama
 - ispitivanje
 - vrednovanje performansi
 - inženjerstvo podataka
 - oblikovanje sučelja





Prog. potpora za unificirani proces



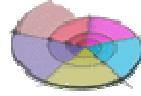
- Modeliranje razvoja OO programske potpore
- Rational Unified Process (RUP)
 - poznat kao Unified Process
 - [IBM Rational Unified Process](#)
- Object Oriented Software Process - OOOSP
- OPEN Process
 - [www.open.org.au](#)
- ICONIX Unified Object Modeling
 - [www.iconixsw.com](#)



The Rational Unified Process



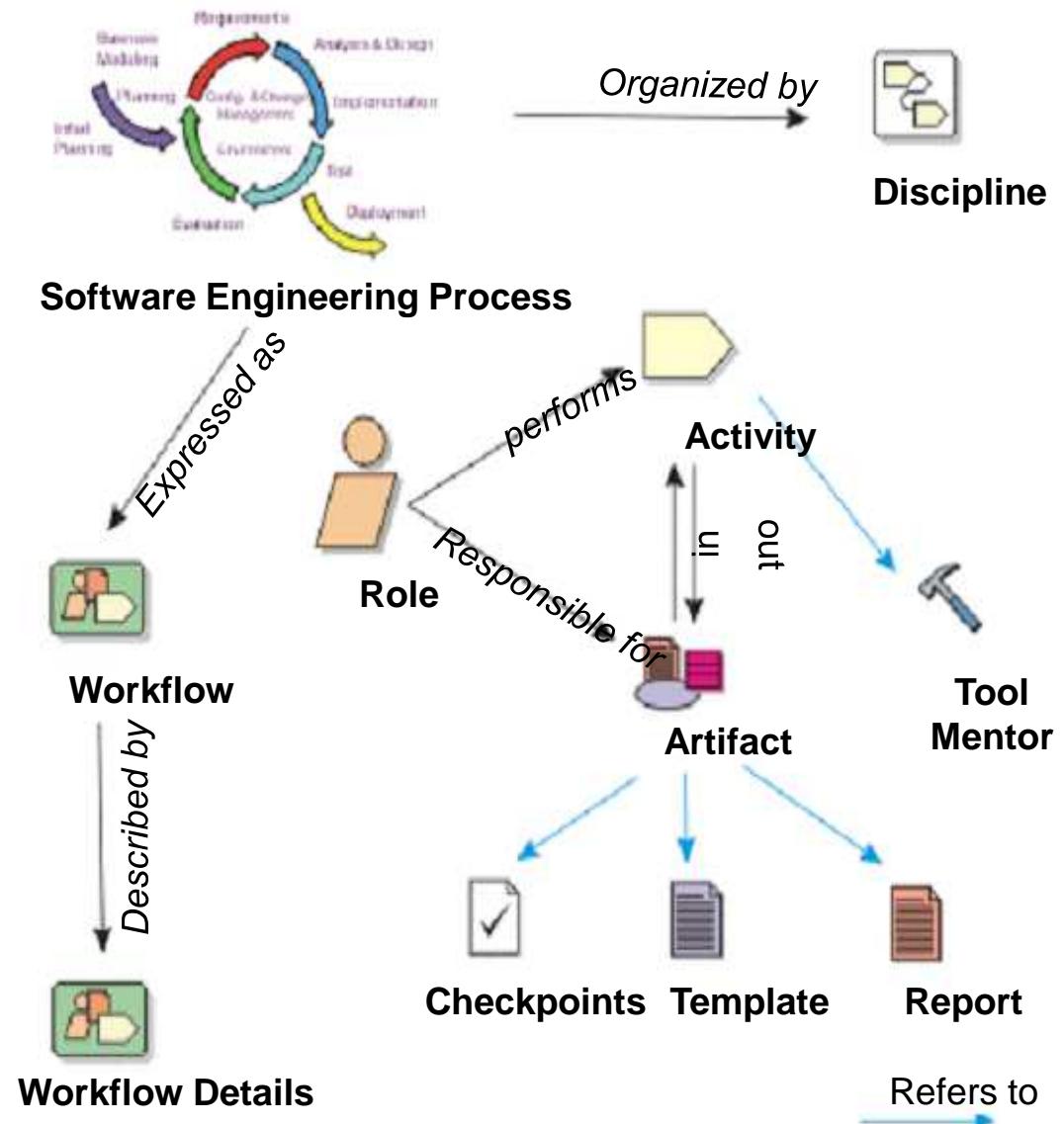
- Obilježja modernog modela procesa
 - promiče više iteracija na pojedinim inkrementima (dijelovima) programske potpore
 - temelji se na obrascima uporabe
 - predlošci scenarija
 - u fokusu je arhitektura sustava
- RUP predstavlja okosnicu procesa (*engl. process framework*) za razvoj i/ili prilagodbu programske potpore.
- RUP je proizvod koji razvija i održava IBM
 - metodologija *engl. process product*
 - integrirani skup programskih alata
 - <http://www-01.ibm.com/software/awdtools/rmc/>



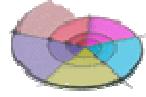
Struktura RUP-a



- Elementi RUP-a:
- Proces - engl. *process*
 - skup koraka
- Disciplina – engl. *discipline*
 - skup aktivnosti
- Uloga – engl. *role*
 - ponašanje i odgovornosti
- Aktivnosti – engl. *activity*
 - radnje daju neki rezultat
- Artefakt – engl. *artifact*
 - rezultat procesa (aktivnosti)
- Radni tijek – engl. *workflow*
 - slijed aktivnosti koji proizvede vidljiv rezultat

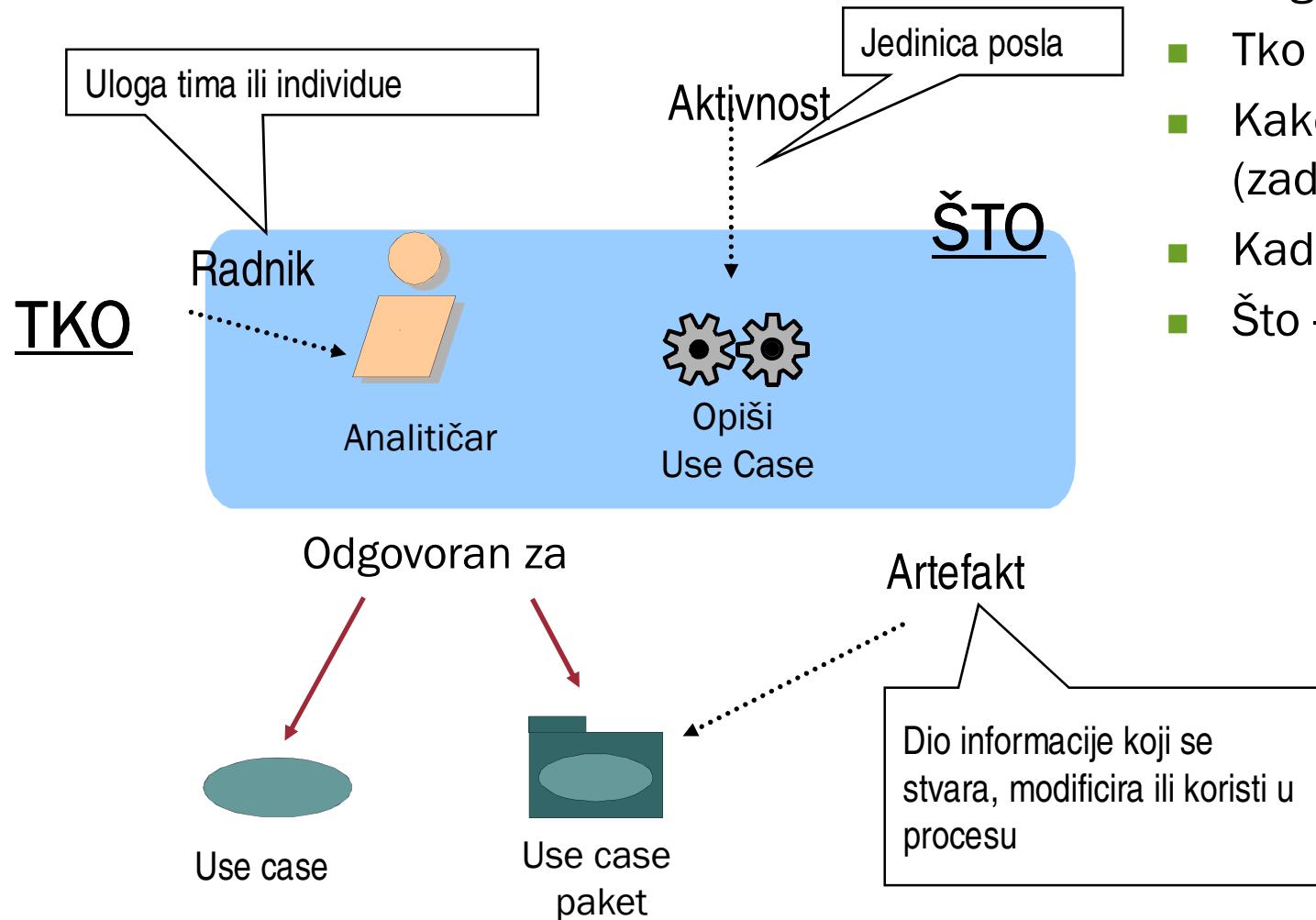


Izvor: Charbonneau S.:Software Project Management - A Mapping between RUP and the PMBOK



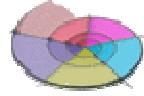
RUP kao proces

■ Predstavlja inženjerski pristup



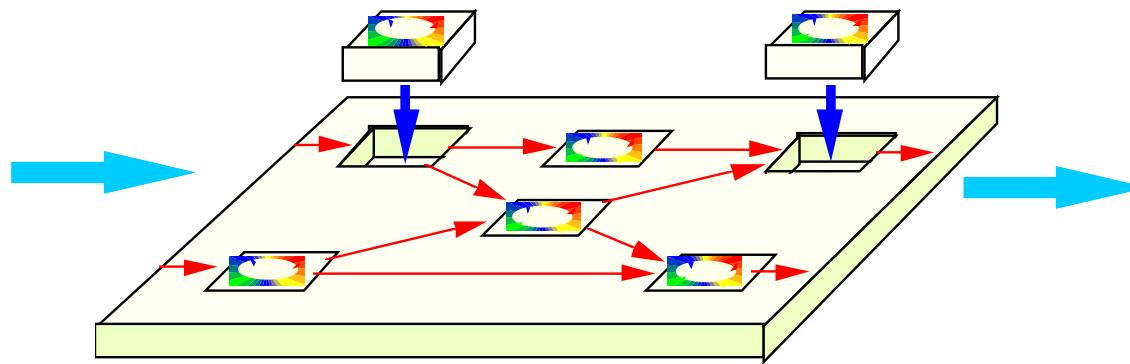
- Odgovori na pitanja:
- Tko -> ljudi (uloge)
- Kako -> aktivnosti (zadaci)
- Kada -> radni tijek
- Što -> artefakt



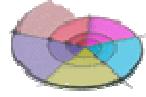


RUP kao okosnica

- RUP inherentno podržava fleksibilnost i proširenje
- Omogućuje razne strategije životnog ciklusa
- Odabire koje artefakte treba proizvesti
- Definira aktivnosti i radnike
- Modelira koncepte
 - engl. *Framework*

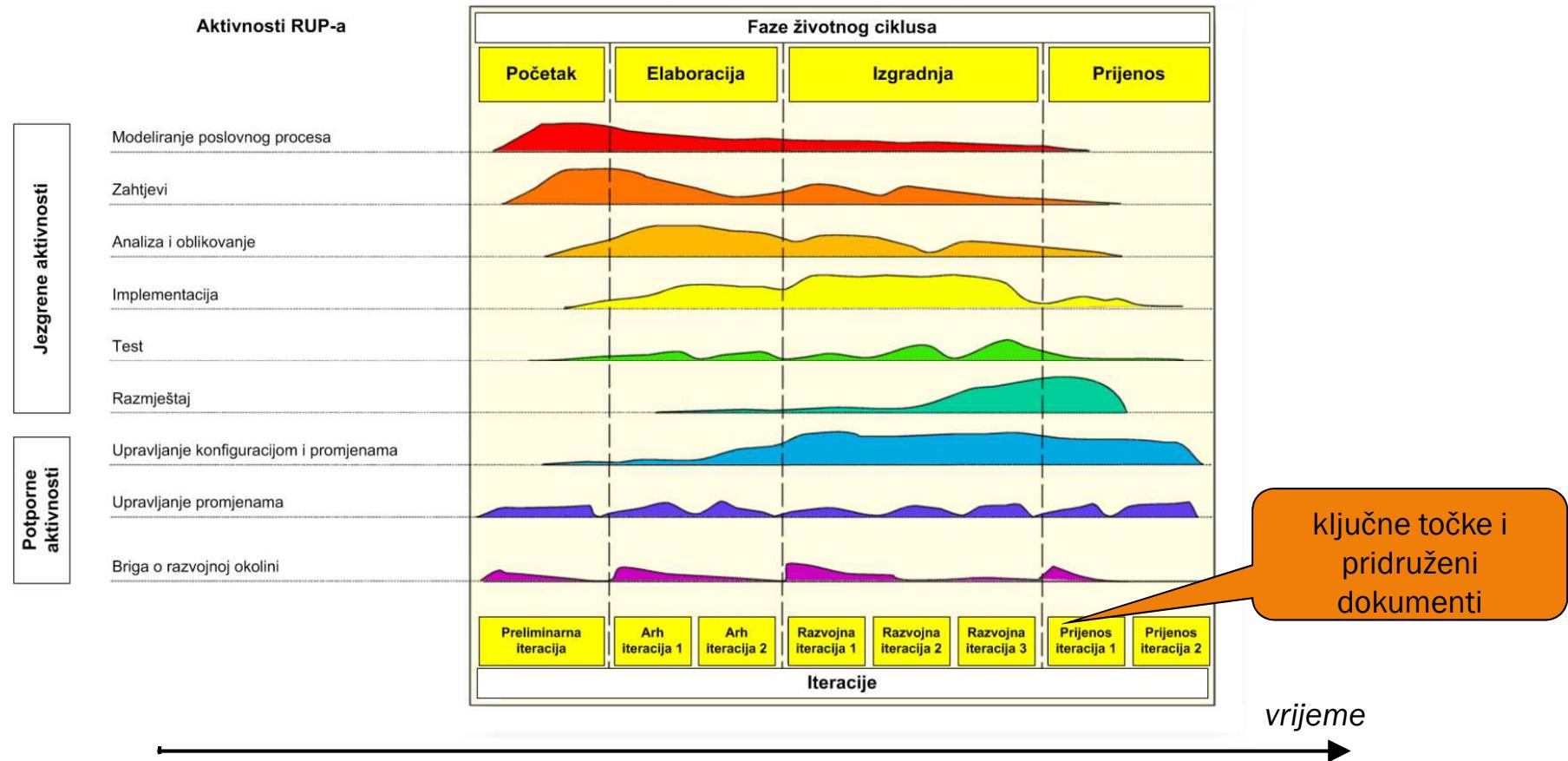


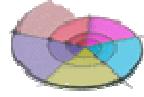
Ne postoji univerzalni proces oblikovanja programske potpore!!



RUP proces

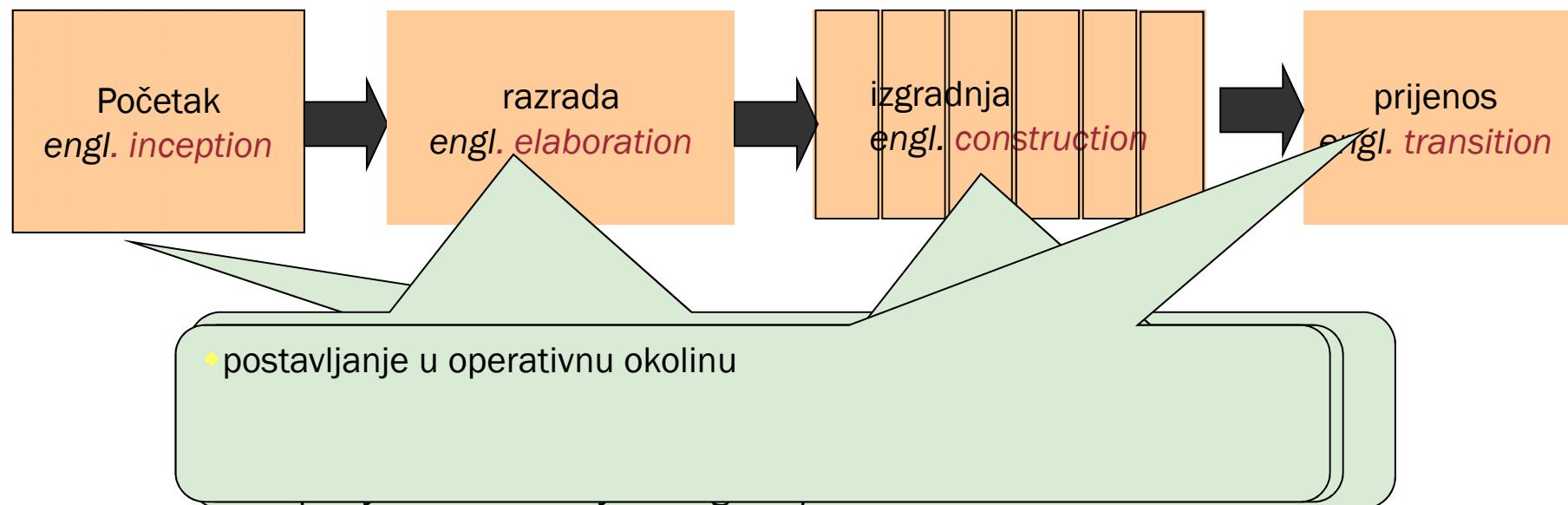
- Dvije dimenzije:
 - horizontalna - dinamika: ciklusi, faze, iteracije i ključne točke
 - vertikalna – statika: opis procesa: aktivnosti, discipline, uloge, artifakti
- Organizacija procesa u vremenu (fazama) i kontekstu (aktivnostima)

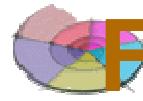




Pregled procesa razvoja

- **Faze:** Početak, razrada, izgradnja, prijenos
- **Iteracija** je sekvenca aktivnosti u okviru prihvaćenog plana i kriterija evaluacije.
 - rezultira u dokumentu, a kasnije i jednoj izvršnoj inačici programa (izdanju, engl. *Release*).

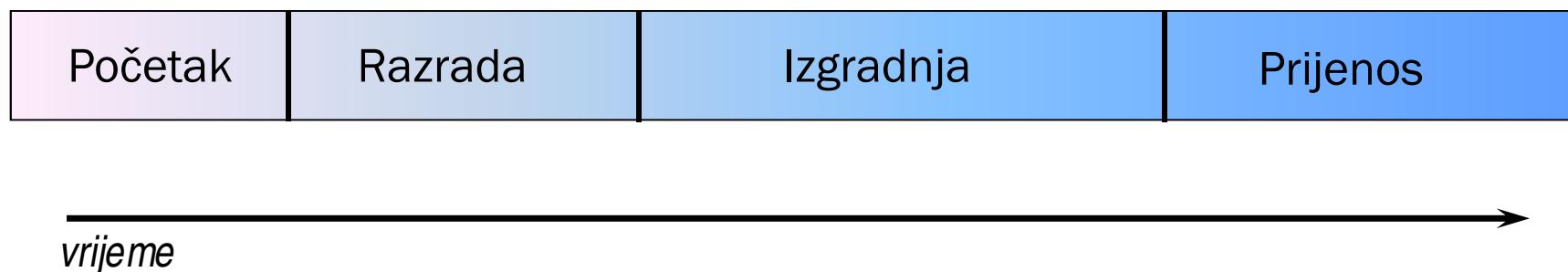


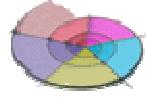


Faze u životnom ciklusu RUP procesa



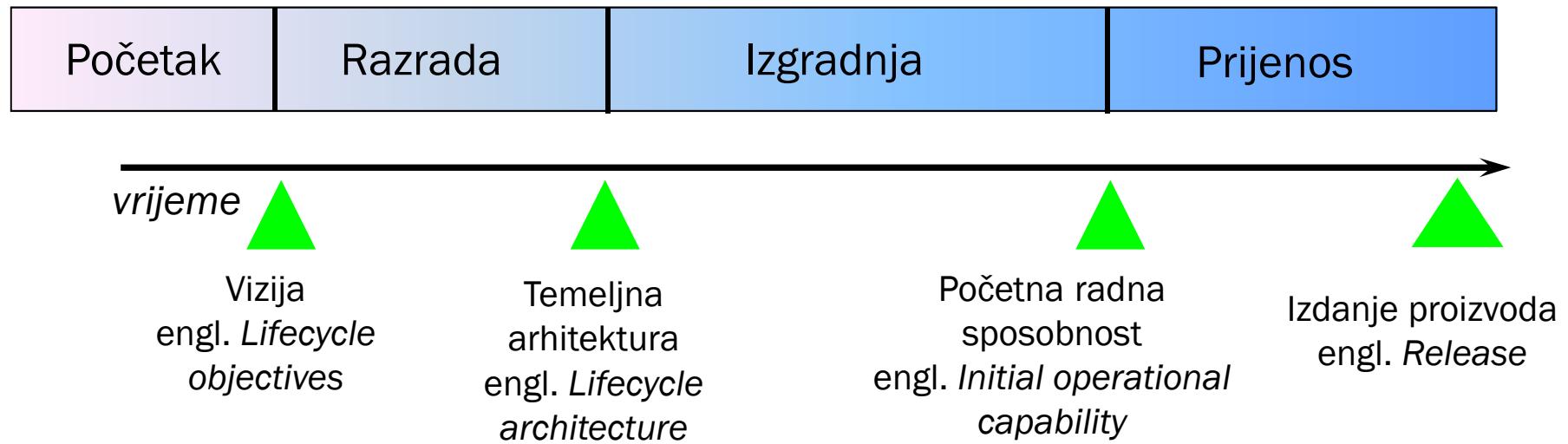
- Početak(engl. *Inception*)
 - definira doseg projekta, razvoj modela poslovnog procesa
- Razrada (engl. *Elaboration*)
 - obuhvaća plan projekta, specifikaciju značajki i temelje arhitekture sustava
- Izgradnja (engl. *Construction*)
 - izgradnja proizvoda (oblikovanje, programiranje, ispitivanje)
- Prijenos (engl. *Transition*)
 - prijenos proizvoda korisnicima (postavljanje u radnu okolinu)





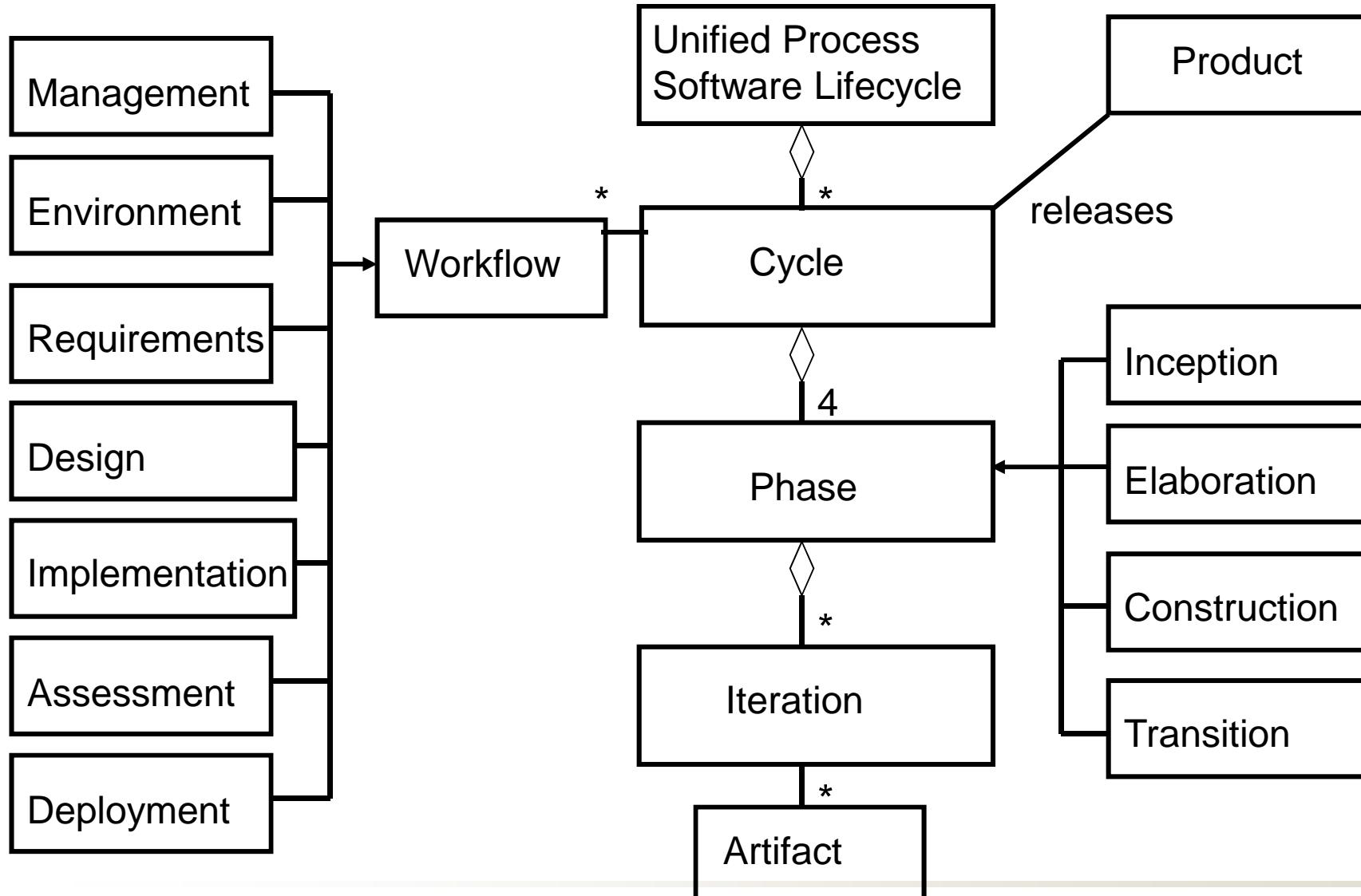
Ključne točke

- engl. *Milestones*
- Ključne točke definiraju pridružene dokumente ili aktivnosti



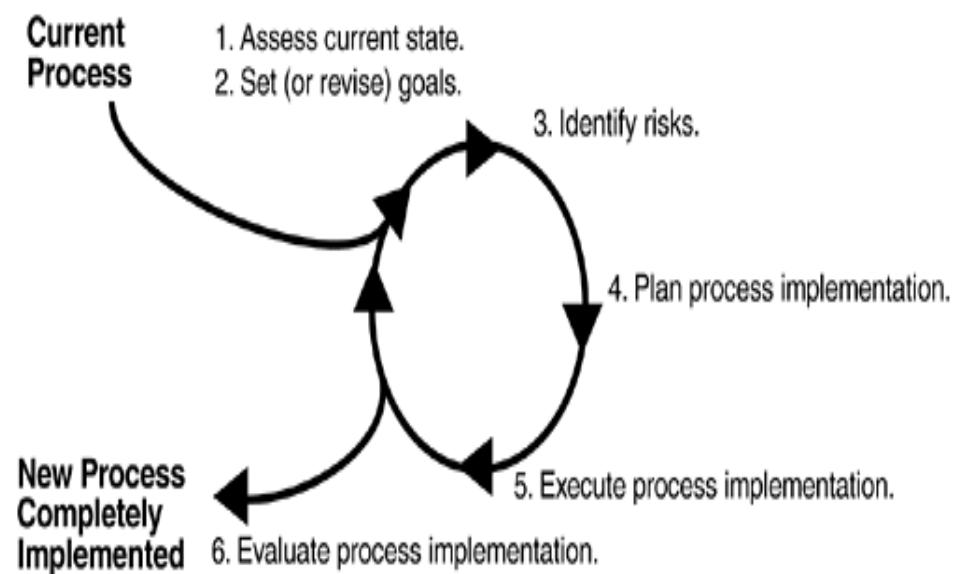


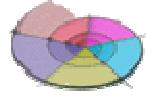
Formalna struktura



Implementacija Rational Unified Procesa

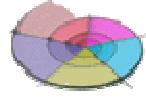
- 1: Procjena trenutnog stanja
- 2: Postavljanje/revizija ciljeva
- 3: Identifikacija rizika
- 4: Planiranje procesa implementacije
- 5: Implementacija
- 6: Vrednovanje implementacije





Prednosti uporabe UML-a

- Otvoren standard
- De facto industrijska norma
- Podupire cijeli životni ciklus oblikovanja programske potpore.
- Podupire različite domene primjene.
- Temeljen je na iskustvu i potrebama zajednice oblikovatelja i korisnika programske potpore.
- Razvijena dobra programska potpora



Mnogo dionika, mnogo pogleda

- Arhitekturu programske potpore različito vidi:
 - korisnik-kupac
 - rukovoditelj projekta
 - inženjer sustava
 - osoba koje razvija sustav
 - arhitekt
 - osoba koja održava sustav (*engl. Maintainer*)
 - drugi dionici
- Višedimenzionalna realnost
- Mnogo dionika
 - višestruki pogledi, višestruki nacrti



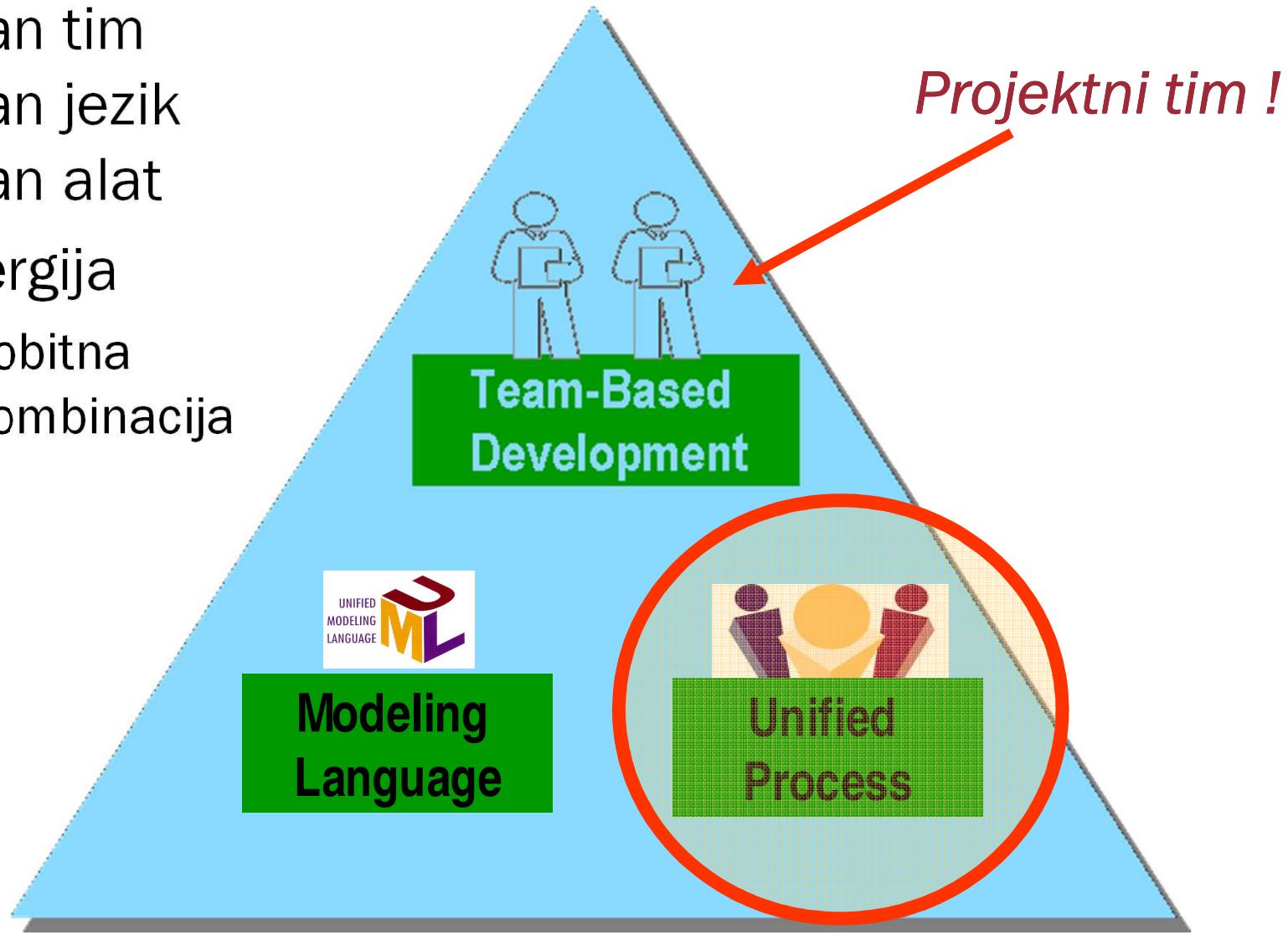
Koliko pogleda?

- Pogledi odgovaraju nekom kontekstu
 - svi sustavi ne trebaju sve poglede:
 - Jedan procesor, nije potreban nacrt razmještaja procesora.
 - Jedan proces, nije potreba nacrt razmještaja procesa.
 - Vrlo mali programi, nije potreban nacrt implementacije.
 - neki dodatni pogledi:
 - Pogled podataka, pogled sigurnosti u sustavu
- Dijagram
 - svaki pogled dokumentira se nacrtom
- MODEL
 - skup više pogleda (dijagrama) u okviru nekog konteksta
 - model je potpuni apstraktan opis sustava iz određene perspektive.
 - reduciran, pojednostavljen, smanjen, ograničen

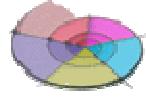


Tim + UML + RUP

- Jedan tim
- Jedan jezik
- Jedan alat
- Sinergija
 - dobitna kombinacija

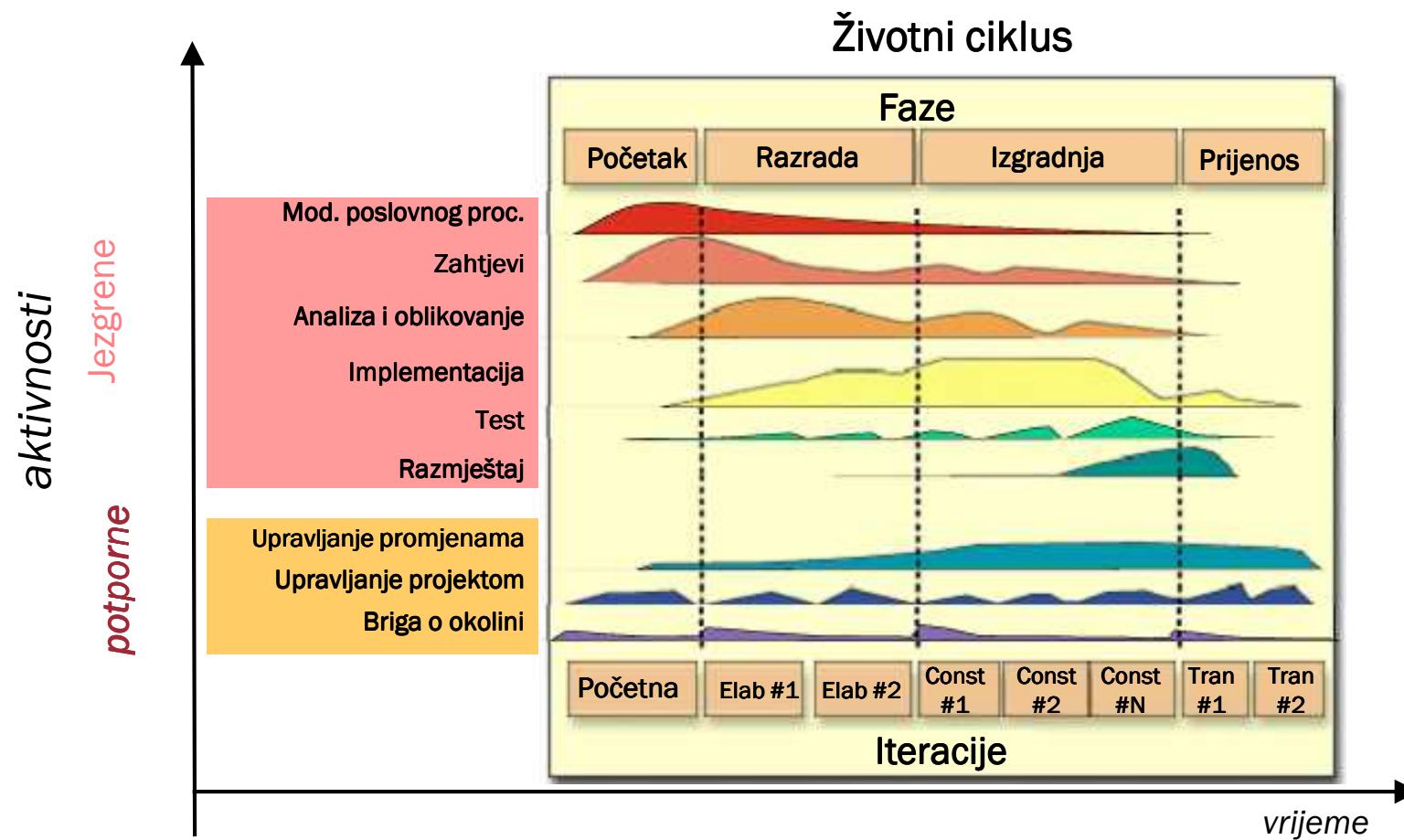


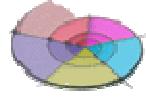
Izvor: I. Jacobson: UML i RUP



Iteracije i aktivnosti

■ Tijek rada, engl. *workflow*

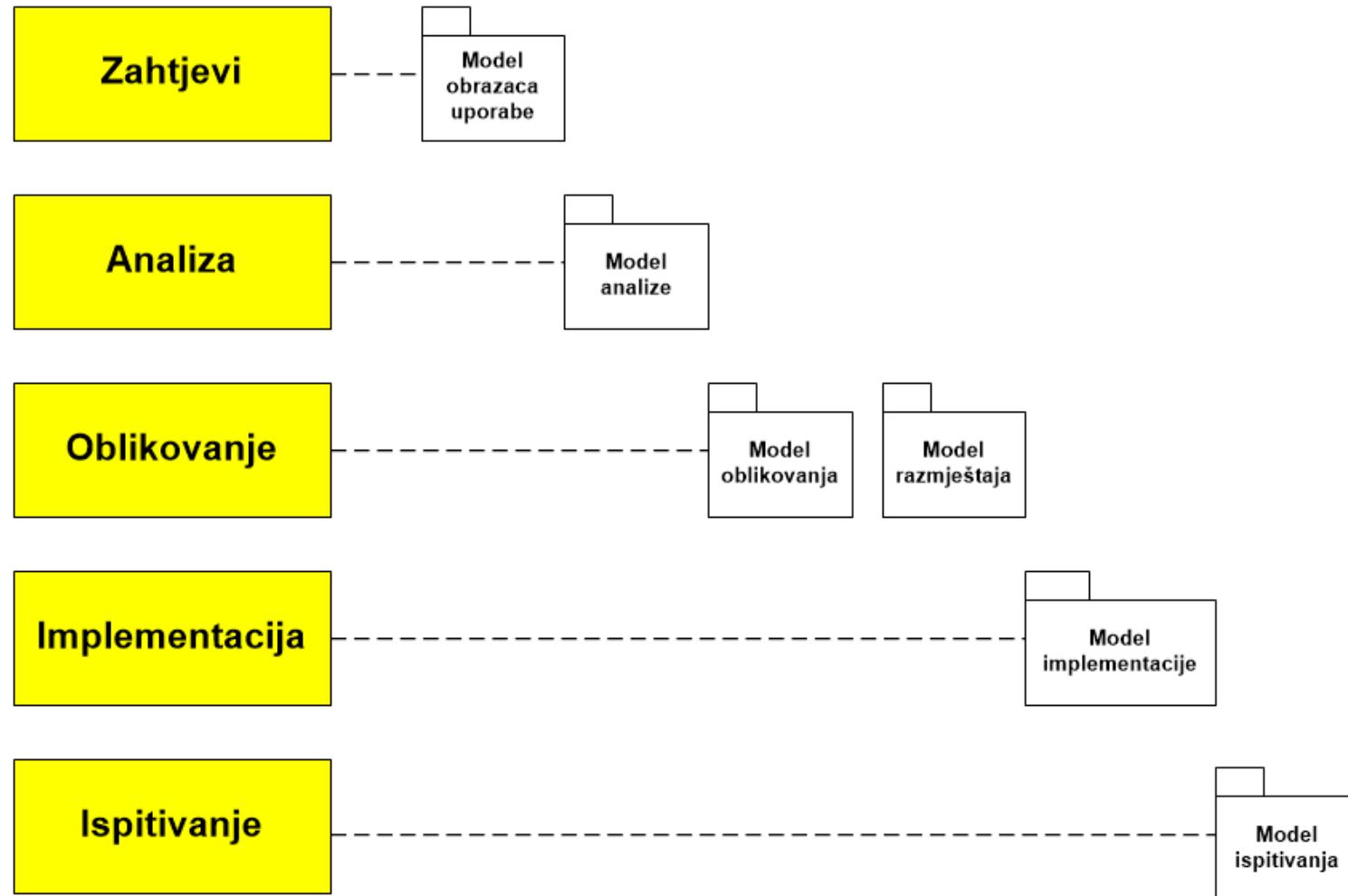


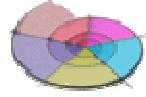


Modeliranje aktivnosti

Aktivnosti

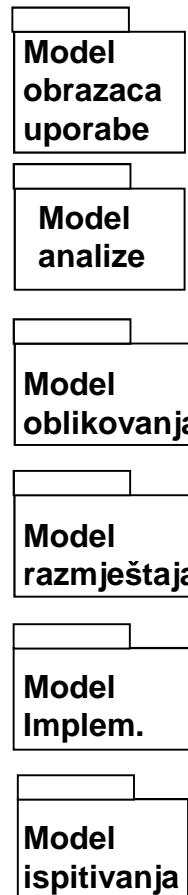
Pridruženi modeli (dokumentirani UML dijagramima)





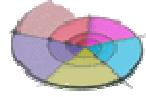
Primjer: Model obrazaca uporabe

Model



Dijagram

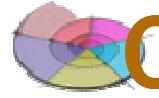




RUP i obrasci uporabe

- RUP se zasniva se na obrascima uporabe (tj. predlošku scenarija)
- Obrasci uporabe koriste se kroz sve faze

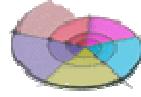




Obrasci uporabe - pokretači iteracija



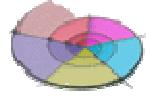
- Obrasci uporabe pokreću razne aktivnosti u životnom ciklusu oblikovanja programske potpore, kao npr.:
 - izrada i validacija arhitekture sustava
 - definicija ispitnih slučajeva, scenarija i procedura
 - planiranje iteracija
 - izrada korisničke dokumentacije
 - razmještaj (engl. *Deployment*) sustava
- Sinkroniziraju sadržaj različitih modela



RUP: arhitektura sustava

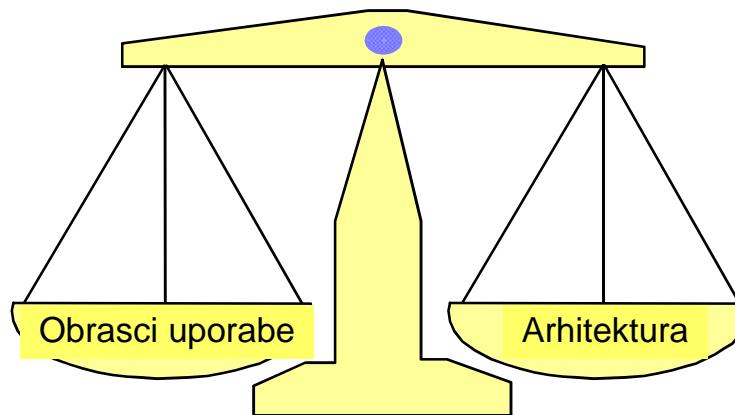
- **Arhitektura programske potpore** je struktura ili strukture sustava koja sadrži elemente, njihova izvana vidljiva obilježja i odnose između njih.
- Modeli su prijenosnici za vizualizaciju, specifikaciju, konstruiranje (oblikovanje, implementacija) i dokumentiranje arhitekture
- RUP propisuje sukcesivno rafiniranje od grubog modela do konačne izvršne arhitekture
- RUP promiče oblikovanje programske potpore zasnovano na modelima (engl. *Model Based Design - MBD*)

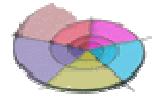




Obrasci uporabe i arhitektura

- Obrasci uporabe specificiraju funkcije
- Arhitektura specificira formu
- Obrasci uporabe i arhitektura moraju biti izbalansirani

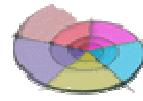




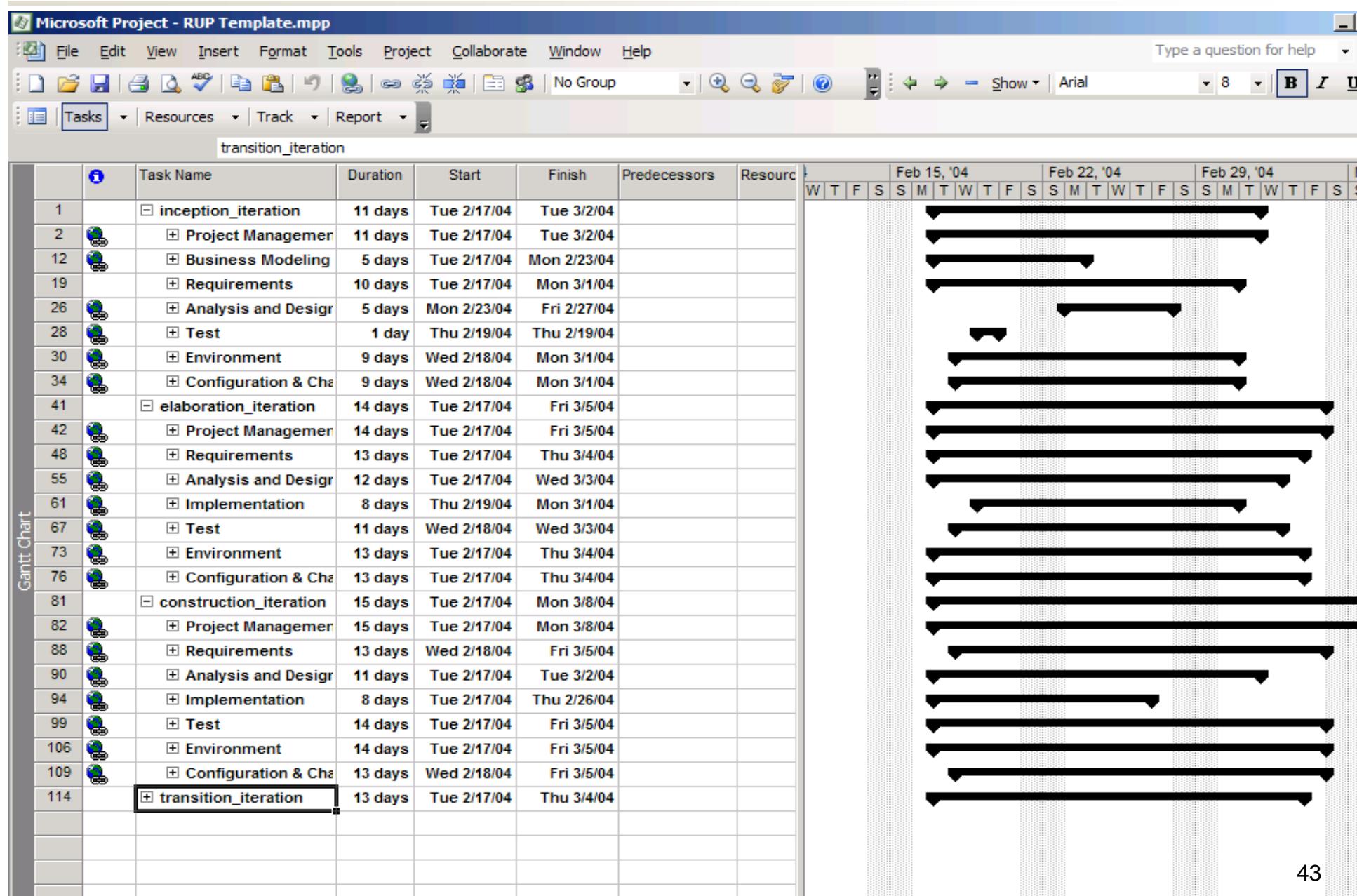
Najbolja praksa svojstvena RUP-u

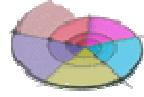


1. Iterativan razvoj
2. Upravljanje zahtjevima
3. Uporaba komponentno zasnovane arhitekture
4. Vizualno modeliranje (UML)
5. Kontinuirana verifikacija kvalitete
6. Upravljanje promjenama programske potpore



Primjer projektnog plana





Značajke RUP-a

- RUP posjeduje značajke *iterativnog i inkrementalnog* oblikovanja programske potpore.
- U središtu RUP procesa su *obrasci uporabe* sustava koji semantički povezuju sve aktivnosti.
- RUP definira i međusobno povezuje *faze i aktivnosti* procesa
- Za opis pojedinih *aktivnosti koriste se* odgovarajući modeli.
- *Modeli* su dokumentirani jednim ili više dijagrama
- *Dijagrami* su definirani UML normom.
- *Arhitektura sustava* sadrži skup pogleda u modele (tj. skup dijagrama)



Procesi programskog inženjerstva.

ITERACIJE U MODELIMA



Iteracije

- *Iteracije u modelima procesa programskog inženjerstva.*
- Zahtjevi na sustav uvijek evoluiraju i prate razvoj projekta:
 - iteracije procesa sastavni su dio velikih projekata jer se pojedini stupnjevi moraju ponovno oblikovati.
 - iteracije se mogu primijeniti na bilo koji generički model procesa programskog inženjerstva.
- Postoje dva međuvisna pristupa iteracijama:
 - *inkrementalni pristup*
 - *spiralni razvoj i oblikovanje*
- Primjena u raznim modelima od modificiranog, vodopadnog, unificiranog, ubrzanih modela

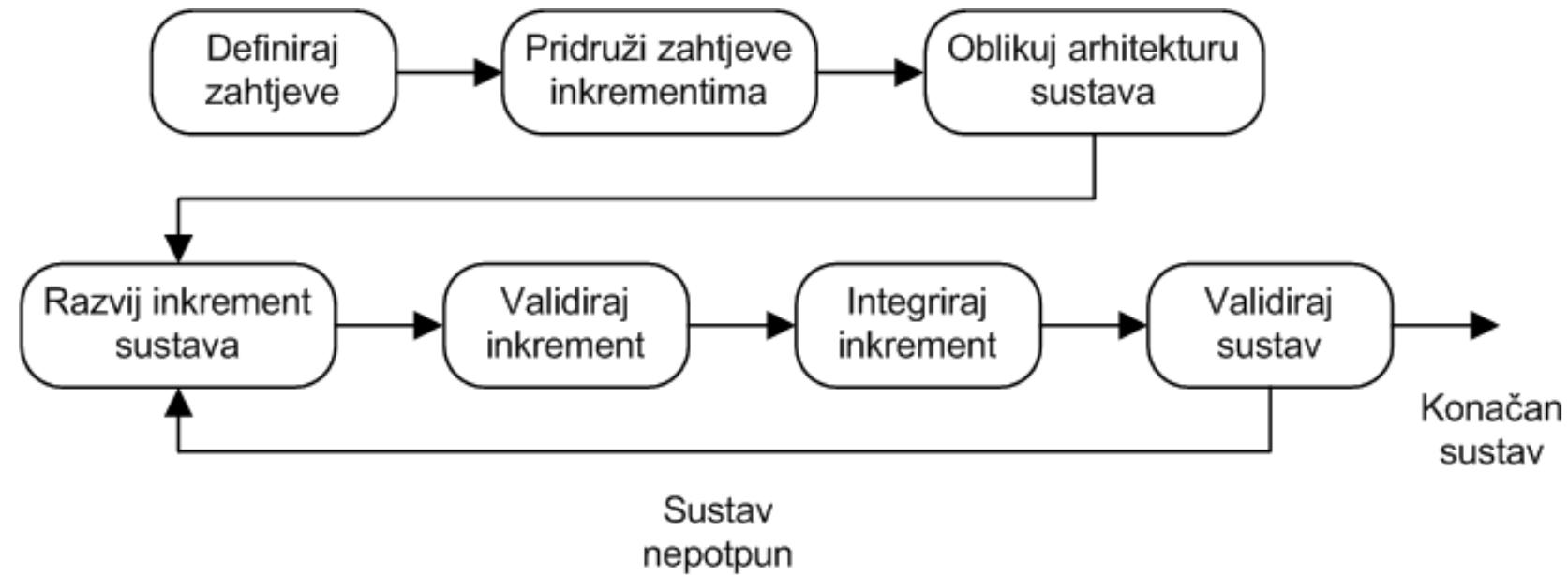


Inkrementalni pristup

- Sustav se ne isporučuje korisniku u cjelini
 - razvoj, oblikovanje i isporuka razbiju se u inkrementalne dijelove koji predstavljaju djelomične funkcionalnosti.
- Zahtjevi korisnika se svrstaju u prioritetne celine
 - dijelovi višega prioriteta isporučuju se u ranim inkrementima.
- S početkom razvoja pojedinog inkrementa njegovi zahtjevi se fiksiraju (zamrzavaju).
 - zahtjevi na kasnije inkremente nastavljaju evoluirati.



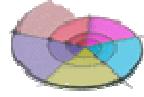
Inkrementalni razvoj, oblikovanje i isporuka





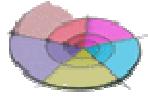
Prednosti inkrementalnog razvoja i isporuke

- Kupac dobiva svoju vrijednost sa svakim inkrementom. Funkcionalnost sustava se ostvaruje u ranim fazama projekta.
- Rani inkrementi služe kao prototipovi na temelju kojih se izlučuju zahtjevi za kasnije inkremente.
- Manji rizik za neuspjeh projekta.
- Prioritetne funkcionalne usluge sustava imaju mogućnost detaljnijeg ispitivanja (testiranja).

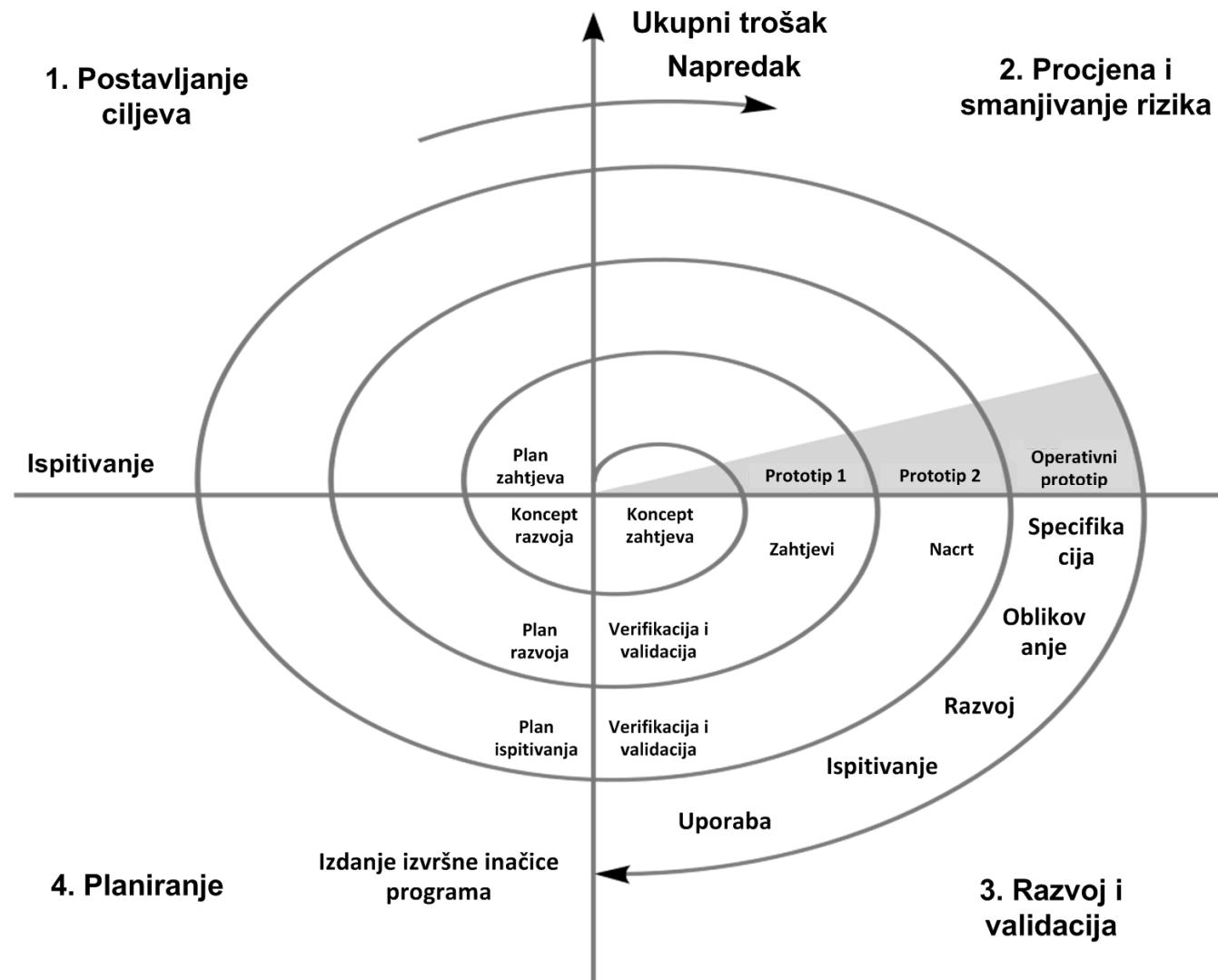


Spiralni razvoj i oblikovanje

- 1988. g. Barry Boehm "A Spiral Model of Software Development and Enhancement"
- Proces se predstavlja spiralom umjesto sekvencom aktivnosti s povratima.
- Svaka petlja u spirali predstavlja fazu procesa.
- Nema fiksnih faza
 - petlje u spirali izabiru se prema potrebnim zahtjevima.
- Rizici razvoja programskog produkta eksplicitno se određuju i razrješuju.



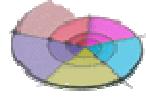
Spiralni pristup





Sektori u spiralnom modelu

- Slijede vodopadni model
- Postavljanje ciljeva
 - identifikacija specifičnih ciljeva sektora
- Procjena i smanjivanje rizika
 - procjenjuju se rizici i preslikavaju u aktivnosti koje ih smanjuju (npr. SWOT analize)
- Razvoj i validacija
 - odabire se model razvoja i oblikovanja
 - može biti bilo koji generički model
 - u ranim spiralama koncept, kasnije spirale nose sve detaljnije aktivnosti (specifikacija, oblikovanje, razvoj, ispitivanje, uporaba)
- Planiranje
 - projekt se kritički ispituje (revidira) i planira se sljedeća faza spirale



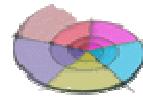
Karakteristike spiralnog pristupa

■ Prednosti

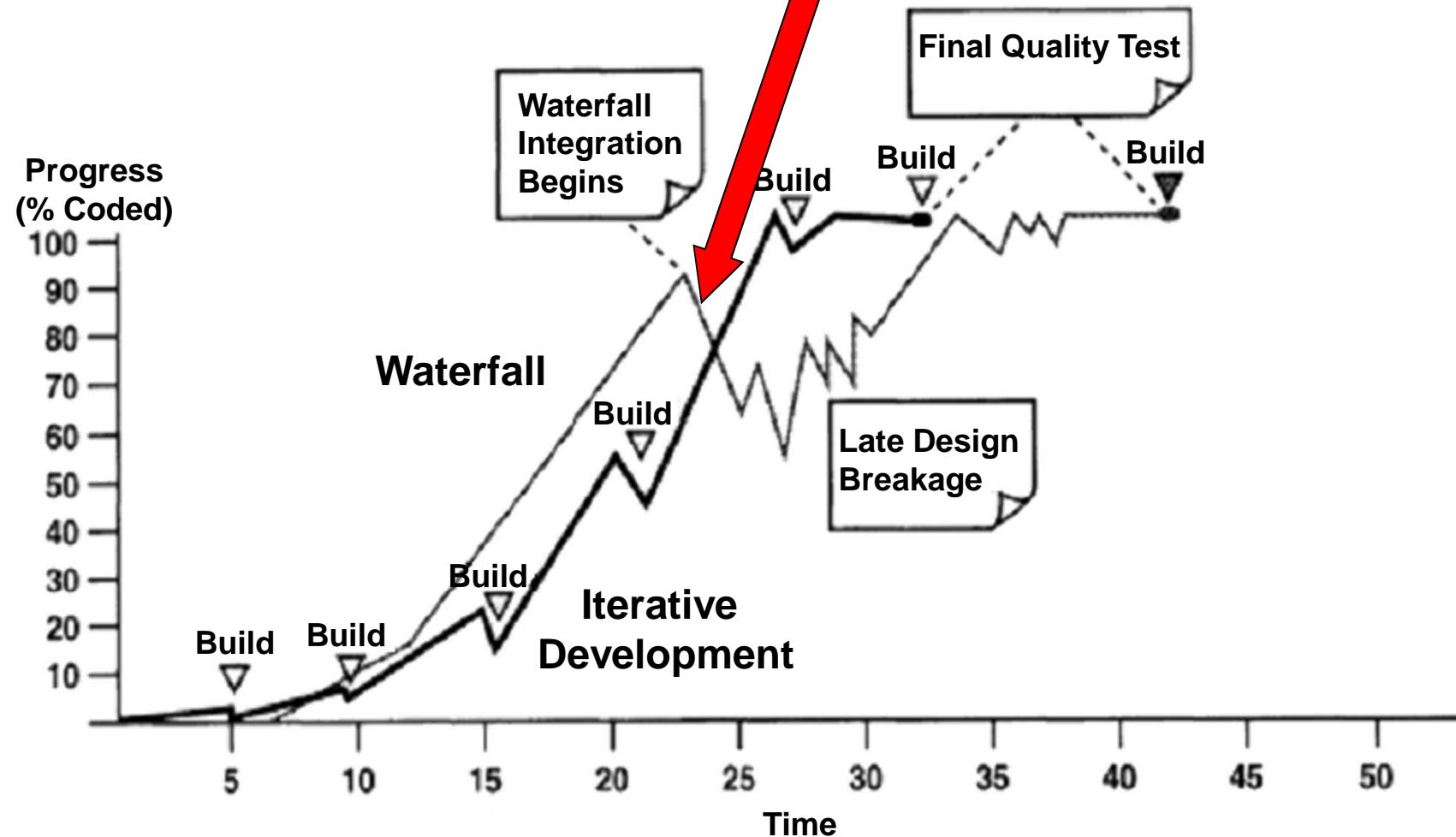
- odražava iterativnu prirodu razvoja programske podrške uzimajući u obzir nejasnoće zahtjeva
- prilagodljivo obuhvaća prednosti vodopadnog modela i brze izrade prototipa
- smanjuje rizik razvoja
- preglednost projekta

■ Nedostaci

- složen, veliko administrativno opterećenje
- zahtjeva poznavanje tehničke analize rizika
- nerazumljiv netehničkom rukovodstvu



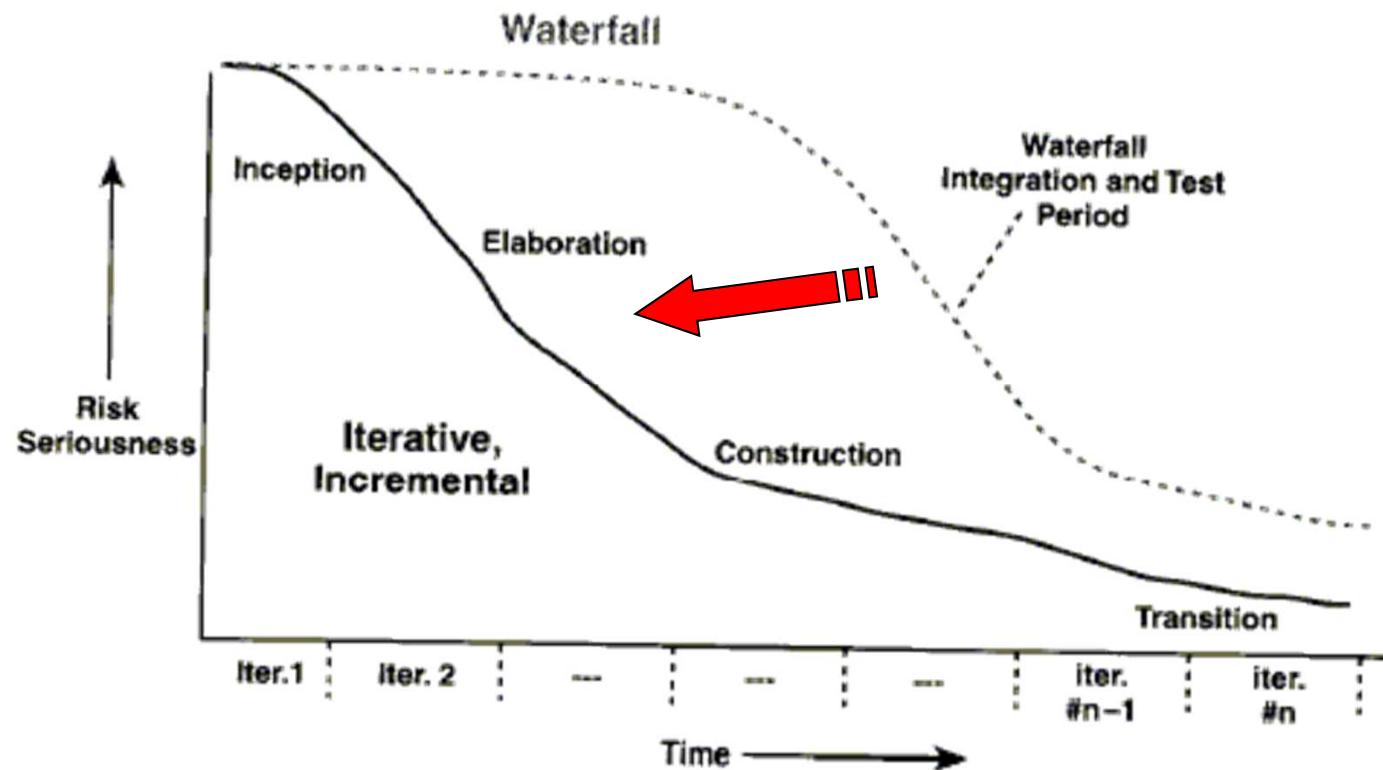
Usporedba vodopadnog modela i iterativnog razvoja



Izvor: I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process, 1999



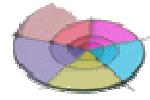
Upravljanje rizikom





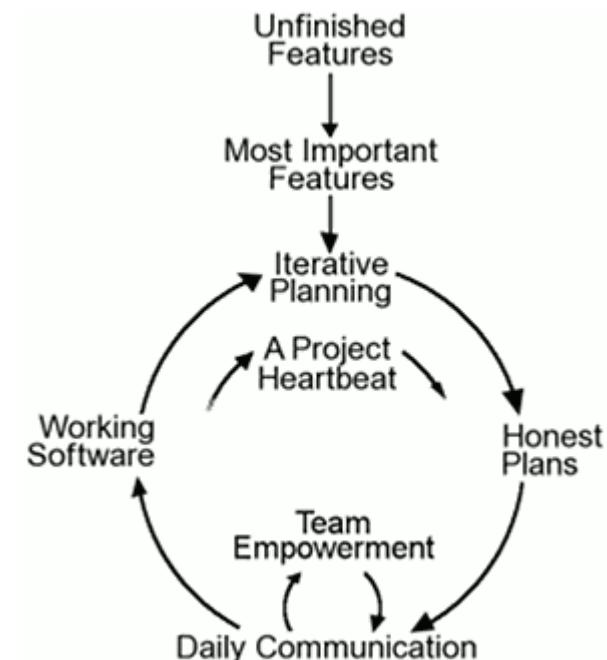
Ubrzani razvoj

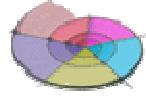
- engl. *Agile methods*
- Grupa metoda za razvoj programske potpore kojima je zajednički iterativni razvoj uz male inkremente te koje podržavaju brzi odziv na korisničke zahtjeve.
- Ovaj model razvoja programske potpore koristi se za brzi razvoj manjih i srednjih projekata u stalnoj interakciji s klijentima putem stalnog predočavanja novih poboljšanja, uz relativno slabo dokumentiranje.
- Fokus je na talentu i vještini pojedinaca
- Najpoznatije metode uključuju:
 - ekstremno programiranje (engl. *Extreme programming*)
 - čisti razvoj programske potpore (engl. *Lean software development*)
 - SCRUM
 - razvoj zasnovan na ispitivanju (engl. *Test-driven development*)
- <http://agilemanifesto.org/>, <http://agilemanifesto.org/iso/hr/>



Primjer: Ekstremno programiranje

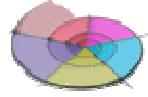
- engl. *extreme programming* - XP
- Kao vrsta inkrementalnog postupka razvoja i isporuke
 - sredinom 1990-tih - odgovor na probleme strukturiranih procesa programskog inženjerstva.
 - Smatra se da vodopadni model unosi previše birokracije
- Pristup se bazira na razvoju, oblikovanju i isporuci vrlo malih inkremenata funkcionalnosti.
 - jedina mjera napretka je funkcionalni programski produkt
- Kontinuirano poboljšanje koda
- Sudjelovanje korisnika u razvojnog timu
- Programiranju u paru
 - engl. *pairwise programming*
 - jedno radno mjesto, međusobno provjeravanje
- [Http://www.extremeprogramming.org/](http://www.extremeprogramming.org/)
- Nedostatak: nerazumljivost, ne podupire ponovnu uporabu rješenja.
- Evolucija: Industrial XP





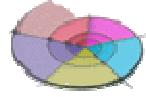
Lean razvoj programske potpore

- engl. *Lean software development (LSD)*
- Primjenjuje "Lean" načela proizvodnje i upravljanja preuzeta iz Toyote
- Cilj je razviti u što kraćem vremenu sustav koji će zadovoljiti korisnike
- Koristi se 7 principa oblikovanja:
 - eliminacija svega suvišnoga
 - koda, nejasnih zahtjeva, birokracije, spore interne komunikacije
 - naglašeno učenje (engl. *Create knowledge/Amplify Learning*)
 - stalna komunikacija s klijentom, stalna testiranja i brze nadogradnje
 - odlaganje odluke (engl. *Defer commitment*)
 - predlaganjem opcija klijentu, skupljanjem činjenica
 - brza isporuka (engl. *Deliver fast*)
 - niz metoda, uglavnom mali inkrementi, više timova radi isto
 - uvažavanje tima (engl. *Respect people*)
 - Motivacija i suradnja unutar tima
 - ugradnja kvalitete (engl. *Build quality in*)
 - Ugradnja cjelovitosti u sustav (kupac mora biti zadovoljan sa sustavom u cjelini: funkcionalnošću, intuitivnošću korištenja, cijenom)
 - optimizacija cjeline (engl. *Optimize the whole*)
 - svaki član tima mora znati kako i zašto čisti razvoj programske potpore treba funkcionirati



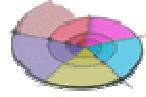
GENERIČKE AKTIVNOSTI U PROCESU PROGRAMSKOG INŽENJERSTVA

SPECIFIKACIJA PROGRAMSKOG PROIZVODA



Specifikacija programskog produkta

- “*The hardest single part of building software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, machines, and to other SW systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.*”
 - Frederick Brooks, The Mythical Man-month essays on software engineering, Addison-Wesley, 1987
- Proces određivanja potrebnih usluga i ograničenja u radu i razvoju sustava.
 - specifikacija programskog proizvoda određuje se procesom inženjerstva zahtjeva (engl. Requirements engineering).
 - proces rezultira dokumentom u kojem se navode potrebne usluge i ograničenja u radu i razvoju sustava.
- Proces inženjerstva zahtjeva (engl. *Requirements engineering*)
 - Studija izvedivosti
 - Izlučivanje i analiza zahtjeva
 - Specifikacija (zapisivanje) zahtjeva
 - Validacija zahtjeva

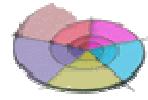


GENERIČKE AKTIVNOSTI U PROCESU PROGRAMSKOG INŽENJERSTVA

OBLIKOVANJE I IMPLEMENTACIJA PROGRAMSKOG PROIZVODA

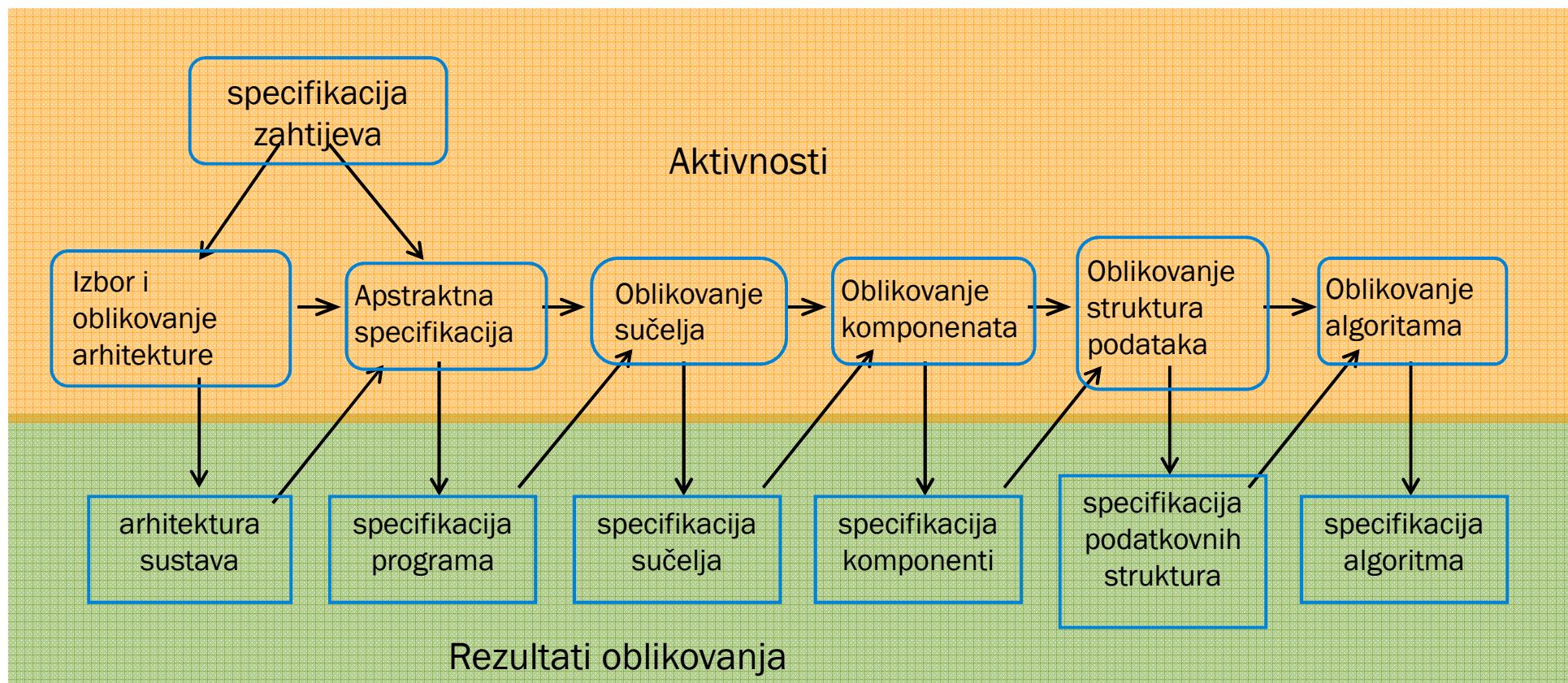


- Proces preslikavanja specifikacije u stvarni, realni sustav.
- *Oblikovanje programske potpore*
 - oblikovanje strukture sustava koja realizira specifikaciju (izbor i modeliranje arhitekture).
- *Implementacija*
 - preslikavanje strukture u izvršni program.
- Aktivnosti oblikovanja i implementacije su povezane i mogu biti isprepletene.



Aktivnosti procesa oblikovanja

- Izbor i oblikovanje arhitekture
- Apstraktna specifikacija
- Oblikovanje sučelja
- Oblikovanje komponenata
- Oblikovanje struktura podataka
- Oblikovanje algoritama





Izbor arhitekture



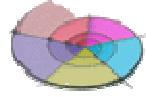
- Prva aktivnost
 - temelji se na neformalnoj analizi i primjeni dobre inženjerske prakse
 - nema formalnog postupka
- Sistematski pristupi oblikovanja programskog proizvoda provodi se na temelju odabране arhitekture
- Arhitektura je dokumentirana skupom modela
 - najčešće grafički dijagrami
- Tipovi arhitekture programske potpore:
 - protok podataka (engl. *data-flow*)
 - objektno usmjerena arhitektura
 - repozitorij podataka
 - ...



Implementacija programskog proizvoda

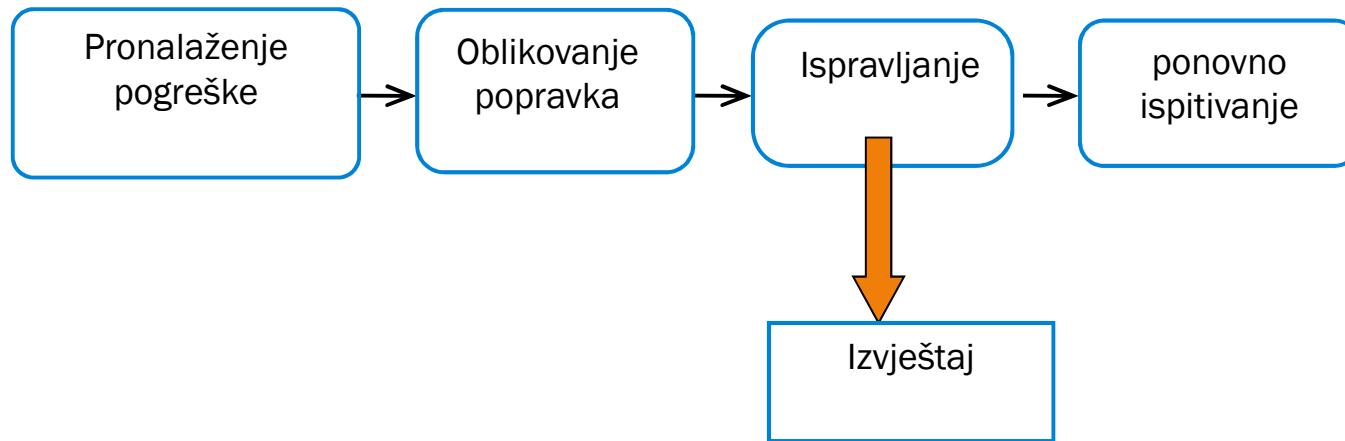


- Implementacija
 - programiranje i otklanjanje pogrešaka
- Preslikavanje dokumentiranog oblikovanja u program i otklanjanje pogrešaka u programu
- Programiranje je osobna aktivnost
 - nema generičkog proces programiranja
- Programeri izvode neke aktivnosti ispitivanja (testiranja) s ciljem otkrivanja pogrešaka u programu i njihovog otklanjanja (*engl. debugging*), oblikovanje i programiranje ispitnih programa

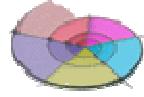


Proces otklanjanja pogrešaka

- U okviru implementacije – programiranja



- Ispitivanje integrirane i cjelovite programske potpore je posebna aktivnost koja ne spada u generičke aktivnosti implementacije!



Zaključak

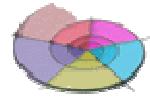
- *Procesi u programskom inženjerstvu*
 - aktivnosti usmjereni na proizvodnju i evoluciju programske potpore.
- *Generičke aktivnosti*
 - specifikacija, oblikovanje i implementacija, validacija i evolucija.
- *Modeli procesa*
 - apstraktna reprezentacija
- *Generički modeli procesa*
 - opisuju njihovu organizaciju
 - vodopadni, evolucijski, komponentni, RUP, ubrzani.
- *Iteracije*
 - opisuju proces kao ciklus različitih aktivnosti unutar generičkih modela procesa (različiti intenzitet)
- Nema univerzalnog i standardnog procesa za sve slučajeve!



Zaključak



- **Inženjerstvo zahtjeva**
 - proces izrade specifikacije programskog produkta.
- **Procesi oblikovanja i implementacije**
 - preslikavaju specifikaciju u arhitekturu i radni/izvršni program.
- **Validacija i verifikacija**
 - provjerava da li sustav zadovoljava specifikaciju i potrebe korisnika.
- **Evolucija**
 - bavi se modifikacijama sustava tijekom njegove uporabe.
- **Rational Unified Process (RUP)**
 - generički model procesa koji odvaja aktivnosti od faza izvođenja.
- **CASE-tehnologija**
 - podupire aktivnosti tijekom procesa programske inženjerstva.



Diskusija



Oblikovanje programske potpore

2014./2015.

Arhitektura programske potpore



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



Tema

- Definicija arhitekture programske potpore.
- Proces donošenja odluke izbora i oblikovanja arhitekture programske potpore.
- Kriteriji izbora arhitekture programske potpore.
- Struktura i sadržaj dokumenata oblikovanja arhitekture programske potpore.

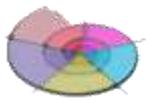
Literatura

- Sommerville, I., *Software engineering*, 8th ed., Addison-Wesley, 2007.
- Timothy C. Lethbridge, Robert Laganière, *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, Second Edition, McGraw Hill, 2001
- Mary Shaw, Paul Clements: *The golden age of software architecture*, IEEE Software, vol 23, no 2, March/April 2006.
- *Software Engineering Institute*
 - Carnegie Mellon University, Pittsburgh, PA, USA
 - utemeljen 1984, zapošljava >300 ljudi (Pittsburgh, Pennsylvania, Arlington, Virginia, i Frankfurt)
 - <http://www.sei.cmu.edu/>



Software Engineering Institute

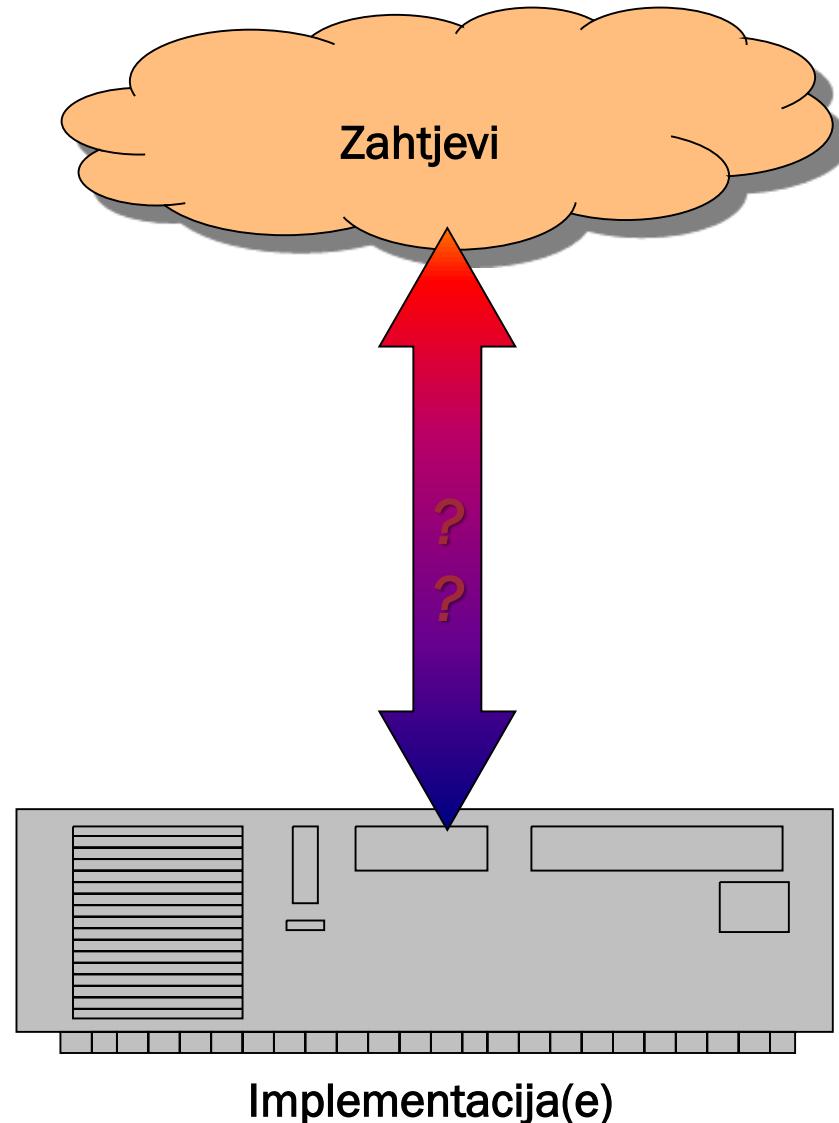
| Carnegie Mellon



Od zahtjeva do koda

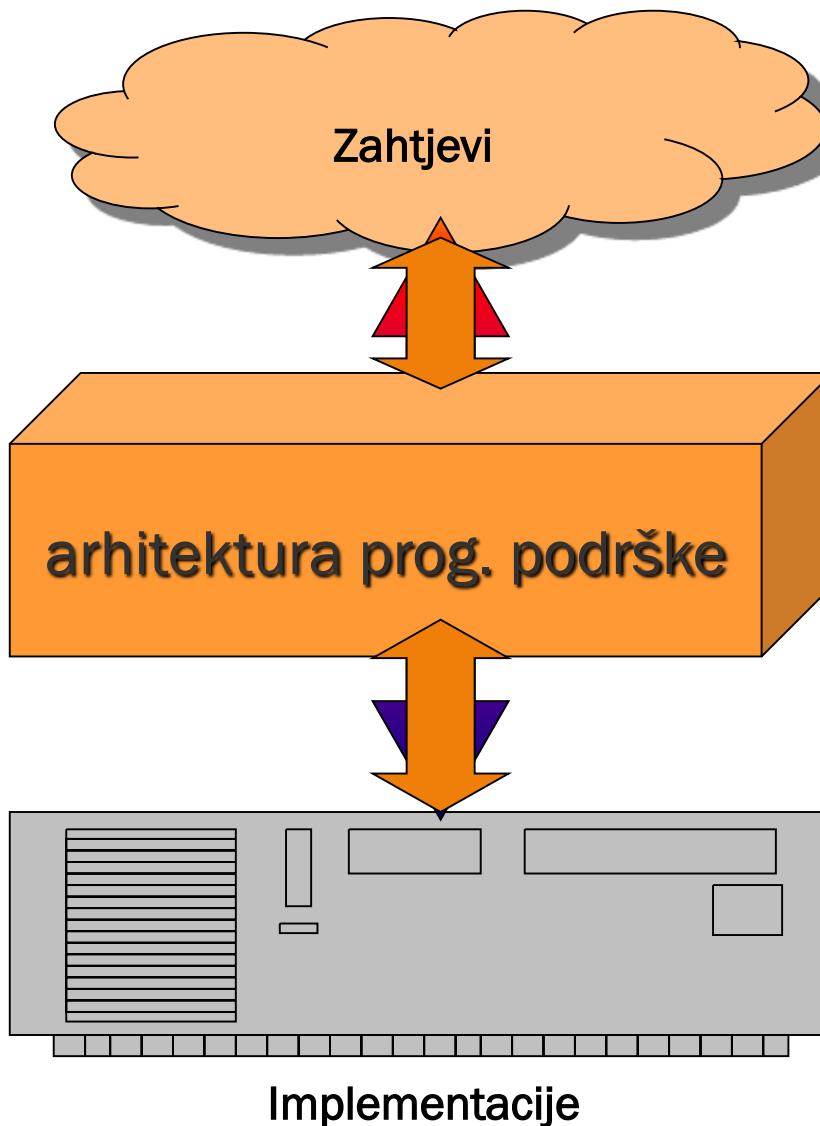


- Veliki raskorak između problema i rješenja

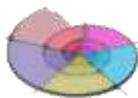




Uloga arhitekture prog. potpore



- Apstrakcija sustava na visokom nivou
 - komponente, konektori, ..
- Osnovni nositelj kvalitete sustava
- Strukturira razvojni projekt i samu programsку podršku
- Kapitalna investicija koja se može ponovno koristiti
- Osnova za komunikaciju dionika
- Izgrađuje se prije detaljne specifikacije



Razvoj arhitekture prog. potpore

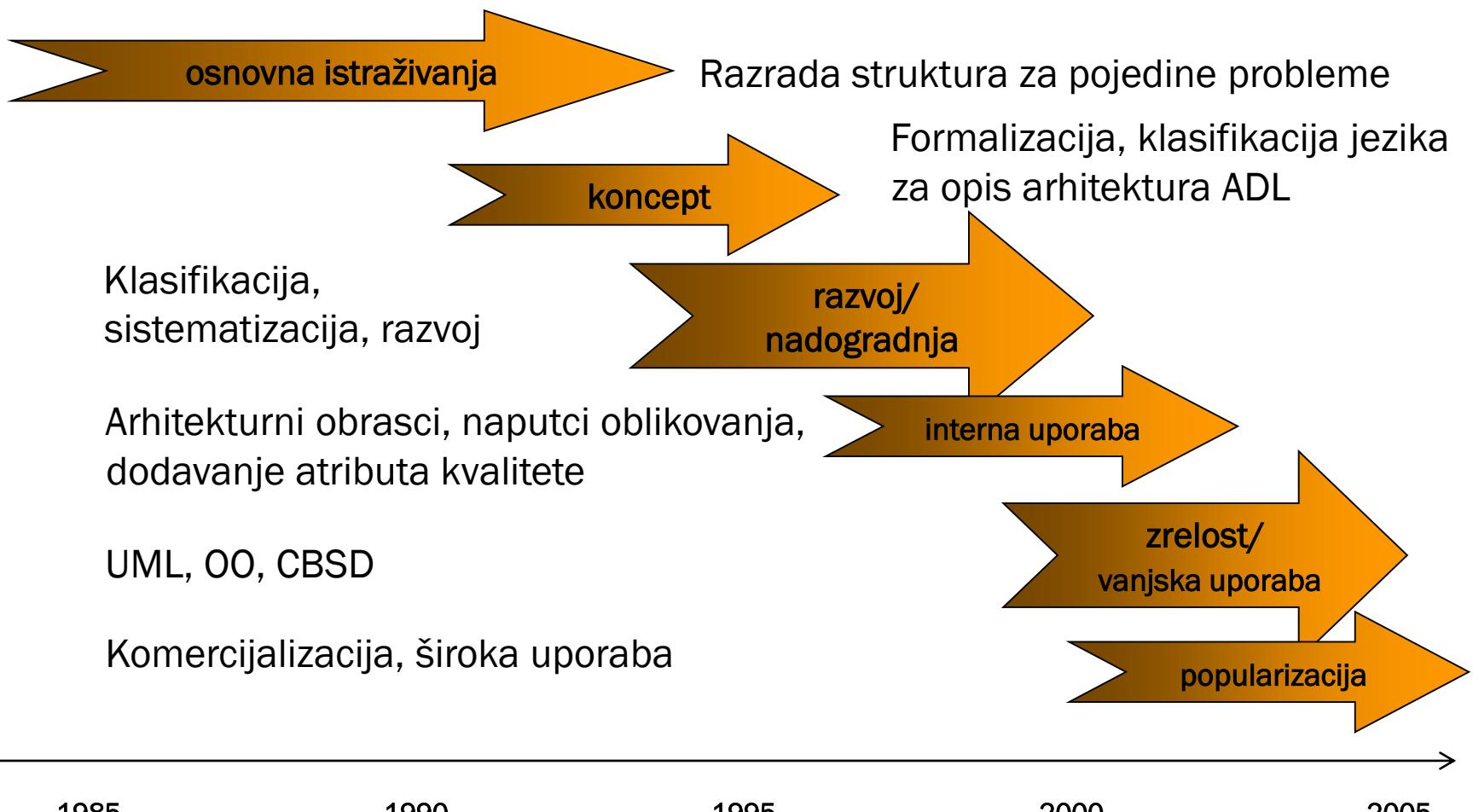


- 1968 - **Conwayov zakon** (Melvin Conway),
 - “It concerns the structure of organizations and the corresponding structure of systems (particularly computer software) designed by those organizations.”
- Krajem 1980-tih - istraživanja u području arhitekture programske podrške s ciljem analize velikih programskih sustava
- U počecima temeljeno na kvalitativnim opisima empirički promatranih organizacija sustava, kasnije se razvija i obuhvaća formalne opise, alate i tehnike analize.
 - od interpretacije do prakse
 - konkretni naputci za analizu, oblikovanje i razvoj složene programske podrške
 - danas predstavlja osnovni element oblikovanja i izrade programskih sustava



Primjer: OO arhitektura prog. potpore

Osnove: sakrivanje informacija, abstraktni tipovi podataka, strukturiranost,...





Razvoj programske potpore



- 1950 – programiranje na bilo koji način
- 1960 – potprogrami i zasebno prevodenje dijelova (engl. *programming in the small*). (Code-and-fix, spaghetti coding)
- 1970 – apstraktni tipovi podataka, objekti, skrivanje informacije (engl. *programming in the large*).
- 1980 – razvojne okoline, cjevovodi i filtri
- 1990 – objektno usmjereni obrasci (engl. *patterns*), integrirana razvojna okruženja.
- 2000 – arhitektura programske potpore, jezici za oblikovanje (npr. *UML*) i metode oblikovanja (npr. modelno oblikovanje arhitekture – engl. *model driven architecture MDA*).



Oblikovanje arhitekture programske potpore



- Proces identificiranja i strukturiranja podsustava koji čine cjelinu te okruženja za upravljanje i komunikaciju između podsustava.
 - rezultat procesa oblikovanja je opis/dokumentacija **arhitekture programske potpore**.
- 1969 NATO Software Engineering conference: I.P. Sharp:
“I think we have something in addition to software engineering... This is the subject of software architecture. Architecture is different from engineering.”



Prednosti definiranja arhitekture

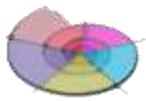


- Smanjuje cijenu oblikovanja, razvoja i održavanja programskog proizvoda.
- Omogućuje ponovnu uporabu rješenja (engl. *re-use*).
- Poboljšava razumljivost.
- Poboljšava kvalitetu proizvoda.
- Razjašnjava zahtjeve.
- Omogućuje donošenje temeljnih inženjerskih odluka.
- Omogućuje ranu analizu i uočavanje pogrešaka u oblikovanju.



Uloga arhitekta

- “Just as good programmers **recognized useful data structures** in the late 1960s, good **software system designers** now **recognize useful system organizations.**”
 - Garlan & Shaw: An Introduction to Software Architecture
- Dobar arhitekt:
 - razumije potrebe poslovnog modela i zahtjeve projekta.
 - svjestan različitih tehničkih pristupa u rješavanju danog problema.
 - vrednuje dobre i loše strane tih pristupa.
 - preslikava potrebe i vrednovane zahtjeve u tehnički opis arhitekture programske potpore.
 - vodi razvojni tim u oblikovanju i implementaciji.
 - koristiti “meke” vještine kao i tehničke vještine.
- Pogled arhitekta na programsку potporu :
 - struktura kao skup implementacijskih zahtjeva
 - struktura i odnosi elemenata tijekom dinamičke interakcije
 - odnosi programskih struktura i okoline

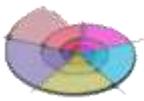


Definicija arhitekture



- Arhitektura programske potpore je struktura ili strukture sustava koji sadrži elemente, njihova izvana vidljiva obilježja i odnose između njih.
- Opis arhitekture programske potpore je skup dokumentiranih pogleda raznih dionika.
- **Pogled** predstavlja djelomično obilježje razmatrane arhitekture programa i dokumentiran je **dijagramom** (nacrtom) koji opisuje strukturu sustava i sadrži:
 - elementi/komponente: dijelovi sustava
 - odnose između elemenata : topologija
 - vanjski vidljiva obilježja
 - Veličina, performanse, sigurnost, API, ...

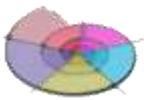




Model arhitekture



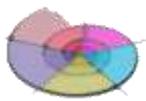
- Više pogleda u okviru nekog konteksta predstavljaju **model arhitekture programske potpore**.
- Klasifikacija:
 - statičan strukturni model - Moduli
 - engl. **module viewtype**
 - pokazuju kompoziciju/dekompoziciju sustava
 - dinamički procesni model
 - engl. **component and connector viewtype**
 - komponente u izvođenju (engl. **runtime**)
 - alocirani elementi
 - engl. **allocation viewtype**
 - Dokumentacija odnosa programske potpore i razvojne/izvršne okoline
- Arhitektura programske potpore opisuje se modelima koji svaki sadrži jedan ili više pogleda (dijagrama).



Stilovi arhitektura programske potpore



- Familije, engl. *Architecture style*
- U pojedinim modelima mogu se prepoznati često upotrebljavane forme i oblici
 - skupovi srodnih arhitektura.
 - u jednom programskom proizvodu može postojati kombinacija više stilova.
- Opisuju se:
 - rječnikom (tipovima komponenata i konektora).
 - topološkim ograničenjima koja moraju zadovoljiti svi članovi stila.
- Primjeri stilova:
 - protok podataka (engl. *data-flow*)
 - objektno usmjereni stil
 - repozitorij podataka
 - upravljan događajima
 - ...



Klasifikacija arhitekture po dosegu



■ **Koncepcijska** (engl. *Conceptual Architecture*)

- usmjeravanje pažnje na pogodnu dekompoziciju sustava
- komunikacija s netehničkim dionicima (uprava, prodaja, korisnici)
- prikaz: UML arhitekturni dijagrami (dijagram paketa, komponenata na visokoj razini, neformalna specifikacija komponenti: CRC kartice)

■ **Logička** (engl. *Logical Architecture*)

- precizno dopunjena koncepcijska arhitektura
- detaljan nacrt pogodan za razvoj komponenti
- prikaz: UML arhitekturni dijagrami sa sučeljima, specifikacije komponenti i sučelja, komunikacijski dijagrami, potrebna objašnjenja diskusije

■ **Izvršna** (engl. *Execution Architecture*)

- namijenjena raspodijeljenim i paralelnim sustavima
- pridruživanje procesa fizičkom sustavu



Problem dokumenata arhitekture

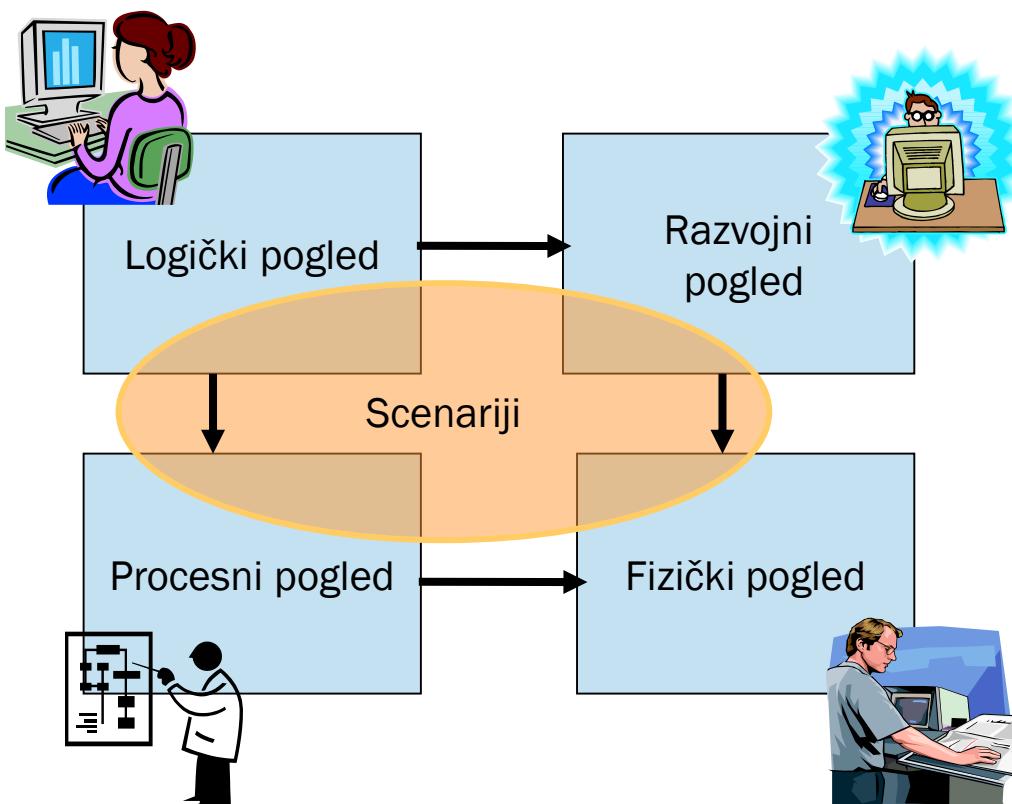


- Prenaglašavanje jedne dimenzije razvoja
- Programski inženjeri pokušavaju sveobuhvatno prikazati problem na jednom nacrtu
 - previše složen dokument
 - nerazumljiv dokument

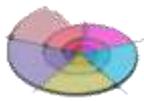


Primjeri arhitekturnih pogleda

- 4+1 pogled
 - 1995 Kruchen, P.: Architectural Blueprints—The “4+1” View Model of Software Architecture, IEEE Software
- Logički pogled – engl. Logic View
 - ponašanje sustava i dekompozicija
- Procesni pogled – engl. Process View
 - opis procesa i njihove komunikacije
- Fizički pogled – engl. Physical View
 - instalacija i izvršavanje u mrežnom okruženju
- Razvojni pogled – engl. Development View
 - opisuje module sustava
- Podatkovni pogled – engl. Data View
 - tijek informacija u sustavu
-



Izvor: Kruchen P.: Architectural Blueprints—The “4+1” View Model of Software Architecture

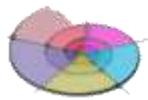


Pogled: Komponente i konektori

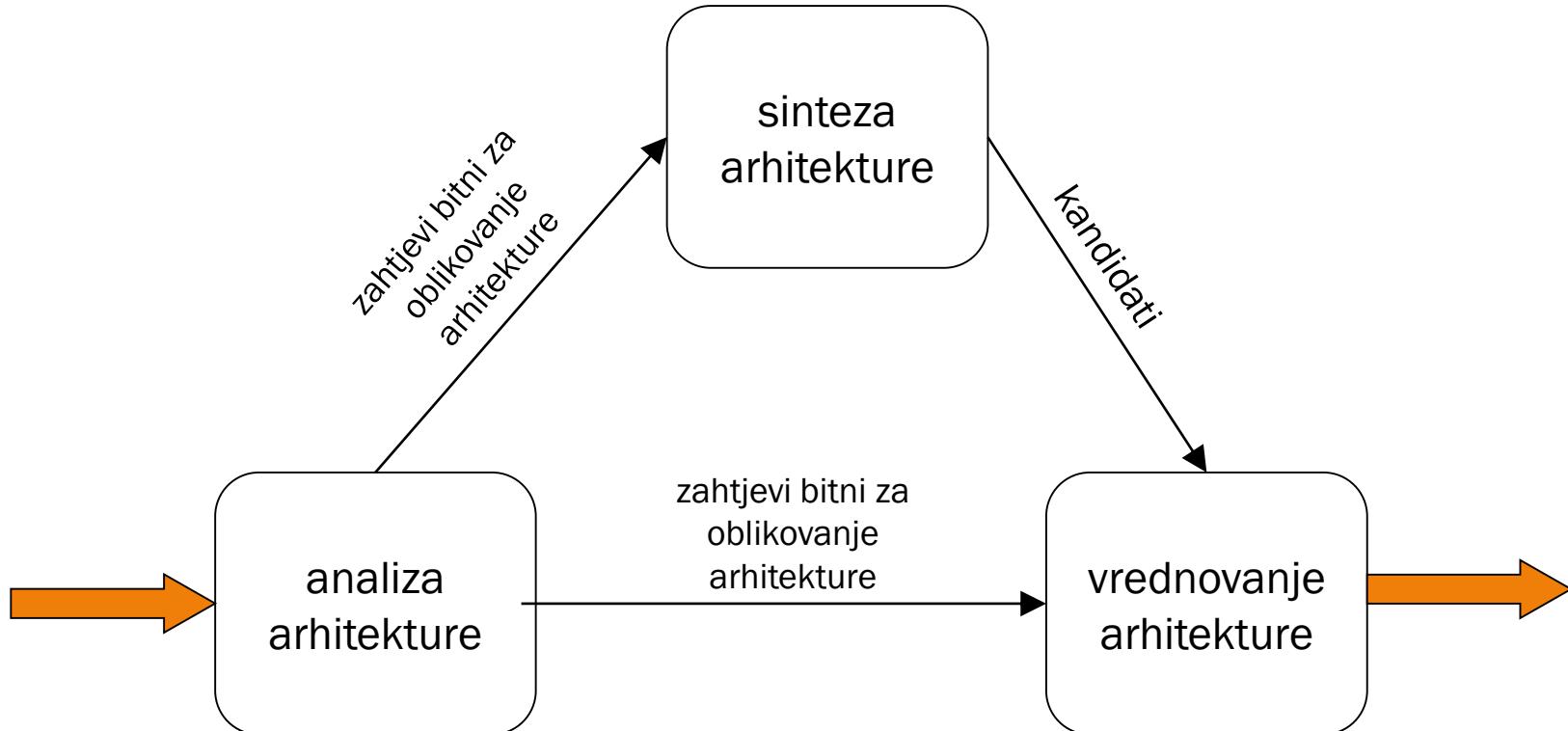


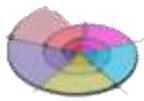
- engl. *Connector and Component*
- Opisuje ponašanje tijekom izvršavanja
- Dekompozicija sustava u komponente (tipično hijerarhijska dekompozicija).
 - komponente:
 - osnovna jedinica izračunavanja i pohrane podataka
 - npr. objekti, procesi, klijent-poslužitelj.
 - konektori:
 - apstrakcija interakcije između komponenata
 - cjevovodi, repozitoriji, utičnice (engl. *socket*), udaljeni poziv procedure, posrednici (engl. *middleware*)
 - uporaba stila arhitekture:
 - oblikovanje kompozicije komponenata i konektora.
- Uvažiti ograničenja i moguće invarijante.

PROCES IZBORA I VREDNOVANJA ARHITEKTURE PROGRAMSKE POTPORE



Aktivnosti oblikovanja





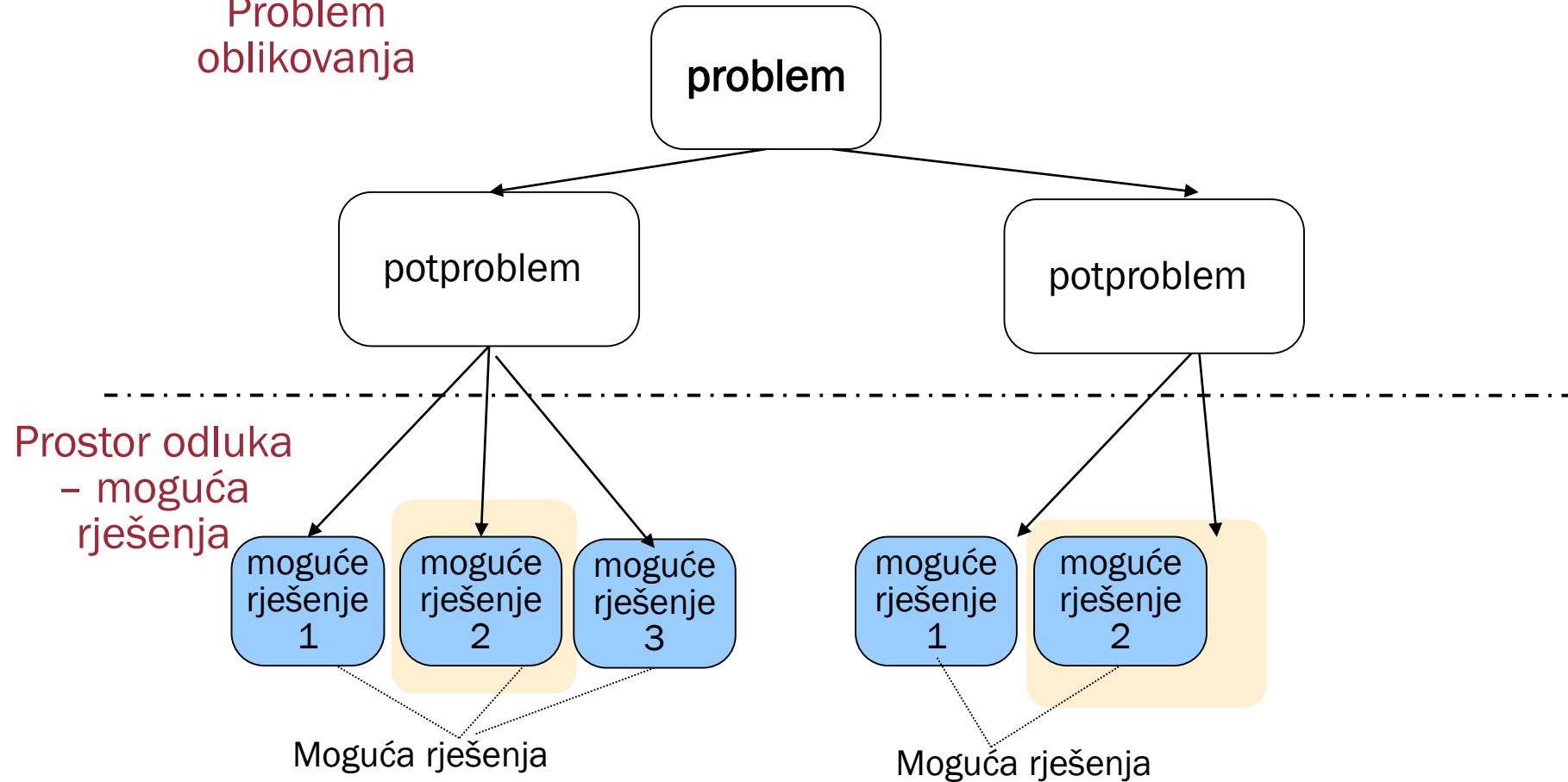
Proces izbora i vrednovanja



- Proces izbora i vrednovanja arhitekture ≡ proces donošenja odluka
- Alternativni stilovi arhitekture programske potpore
⇒ Oblikovanje kao niz odluka
- Dizajner se sučeljava s rješavanjem niza problema (engl. *design issues*)
 - potproblemi ukupnog problema
- Više inačica rješenja
 - engl. *design options*
- Dizajner donosi odluke (engl. *design decision*) za rješavanje problema
 - odabir najbolje opcije između više mogućih rješenja problema

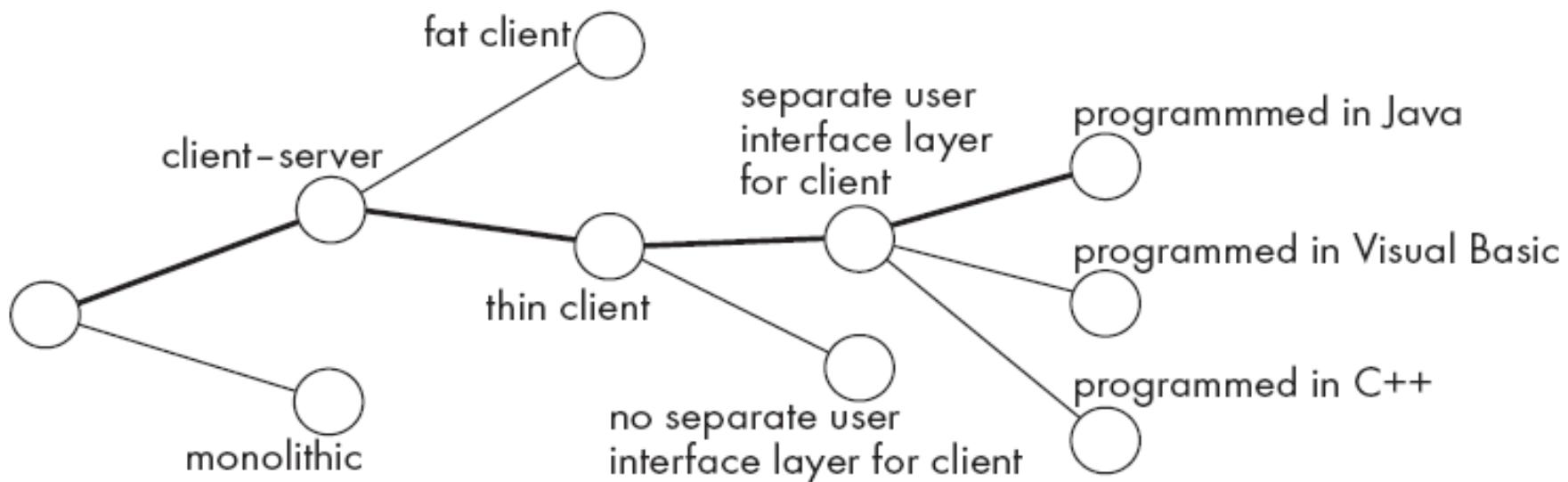
Odluke

Problem
oblikovanja

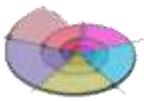


Prostor oblikovanja

- Prostor oblikovanja je **skup opcija** koja su na raspolaganju uporabom različitih izbora, engl. **design space**
- Primjer:



Izvor: Sommerville, I., Software engineering



Donošenje odluka



- Za donošenje odluka potrebna znanja
 - zahtjeva
 - trenutno oblikovana arhitektura
 - raspoloživa tehnologija
 - principi oblikovanja i najbolja praksa (*engl. best practices*)
 - dobra rješenja iz prošlosti
- Zadaće donošenja odluka
 - postavljanje prioriteta sustava
 - dekompozicija sustava
 - definiranje svojstava sustava
 - postavljanje sustava u kontekst
 - cjelovitost sustava
- Tehnička i netehnička pitanja su isprepletena!



Oblikovanje od vrha prema dolje



■ engl. *Top-down design*

- oblikuje najvišu strukturu sustava
- postepeno razrađuj detalje
- na kraju detaljne odluke:
 - Format podataka;
 - Uporaba/izbor algoritama



Oblikovanje od dna prema vrhu

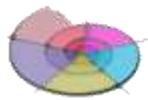


■ *engl. Bottom-up design*

- donošenje odluka o komponentama za ponovnu uporabu
- odluke o njihovoj uporabi za stvaranje komponenti više razine

■ *Hibridno oblikovanje*

- uporaba obje metode:
- oblikovanje od vrha prema dolje
 - Dobra struktura sustava
- oblikovanje od dna prema vrhu
 - Stvaranje komponenti pogodnih za ponovnu uporabu



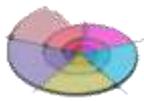
Svojstva oblikovanja



- Oblikovanje arhitekture
 - podjela u podsustave i komponente
 - Način povezivanja?
 - Način međudjelovanja?
 - Sučelja?
- *Oblikovanje korisničkog sučelja*
- *Oblikovanje algoritma*
 - za izračunavanje, upravljanje ..
- *Oblikovanje protokola*
 - komunikacijski protokoli



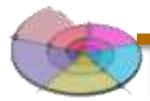
*ne u ovom
kolegiju*



Razvoj modela arhitekture



- Započni s grubom skicom arhitekture zasnovanoj na osnovnim zahtjevima i obrascima uporabe.
- Odredi temeljne potrebne komponente sustava.
- Izaberi između raznih stilova arhitekture
 - savjet: nekoliko timova nezavisno radi grubu skicu arhitekture, a potom se spoje najbolje ideje.
- Arhitekturu dopuni detaljima tako da se:
 - identificiraju osnovni načini komunikacije i interakcije između komponenata.
 - odredi kako će dijelovi podataka i funkcionalnosti raspodijeliti između komponenata.
 - pokušaj identificirati dijelove za ponovnu uporabu
 - vrati se na pojedini obrazac uporabe i podesi arhitekturu.



■ *Uporaba prioriteta i ciljeva za odabir alternativa*

1. pobroji i opiši alternative odluka oblikovanja
2. pobroji prednosti i nedostatke svake alternative u odnosu na prioritete i ciljeve
3. odredi sukob alternative koje su u sukobu s ciljevima
4. odaberite alternative koje najbolje zadovoljavaju ciljeve
5. prilagodi prioritete za daljnje donošenje odluka

- Strukturne odluke (engl. *structural decisions*)
 - stvaranje podsustava, nivoa, komponenata ...
- Ponašajne odluke (engl. *behavioral decisions*)
 - formiranje interakcija u sustavu
- Odluke o svojstvima (naputci)
 - vodilje (engl. *design rules or guidelines*)
 - ograničenja (engl. *design constraints*)
- Izvršne odluke
 - poslovne odluke koje utječu na metodologiju razvoja, ljudi, alate

Primjer prioriteta i ciljeva

- U području oblikovanja ***računalnog sustava***:
- ***Sigurnost/Security***: Podaci se ne smiju moći dešifrirati poznatim tehnikama za < 100sati na 400Mhz Intel procesoru.
- ***Održavanje/Maintainability***: Nema
- ***Performanse/CPU efficiency***: Odziv < 1s na 400MHz Intel procesoru.
- ***Mrežno opterećenje/Network bandwidth efficiency***: $\leq 8\text{KB}$ po transakciji.
- ***Memorijski resursi/Memory efficiency***: $\leq 20 \text{ MB RAM}$.
- ***Prenosivost/Portability***: Windows XP, Linux



Analiza mogućih opcija



- Npr. 5 različitih opcija

	<i>Security</i>	<i>Maintainability</i>	<i>Memory efficiency</i>	<i>CPU efficiency</i>	<i>Bandwidth efficiency</i>	<i>Portability</i>
Algorithm A	High	Medium	High	Medium; DNMO	Low	Low
Algorithm B	High	High	Low	Medium; DNMO	Medium	Low
Algorithm C	High	High	High	Low; DNMO	High	Low
Algorithm D	—	—	—	Medium; DNMO	DNMO	—
Algorithm E	DNMO	—	—	Low; DNMO	—	—

Izvor: Sommerville, I., Software engineering



Uporaba analize troškova i koristi za odabir



- engl. *Cost-Benefit Analysis*
- Utvrđivanje troškova novog sustava – engl. Cost
 - razvoj – engl. *capital expenditure* - CAPEX
 - redovni rad – engl. *operating expenditure* - OPEX
- Utvrđivanje koristi novog sustava – engl. *Benefits*
 - mjerljive i direktne - engl. *tangible*
 - teško mjerljive, posredne – engl. *intangible*
- Procjena cijene/troškova uključuje:
 - cijena rada programskog inženjera, uključujući održavanja
 - cijena uporabe razvojne tehnologije
 - cijena krajnjih korisnika i potpore
- Procjena koristi/dobiti uključuje:
 - uštedu vremena programskog inženjera
 - dobrobiti mjerene kroz povećanu prodaju ili ostale financijske uštede



Oblikovanje stabilne arhitekture



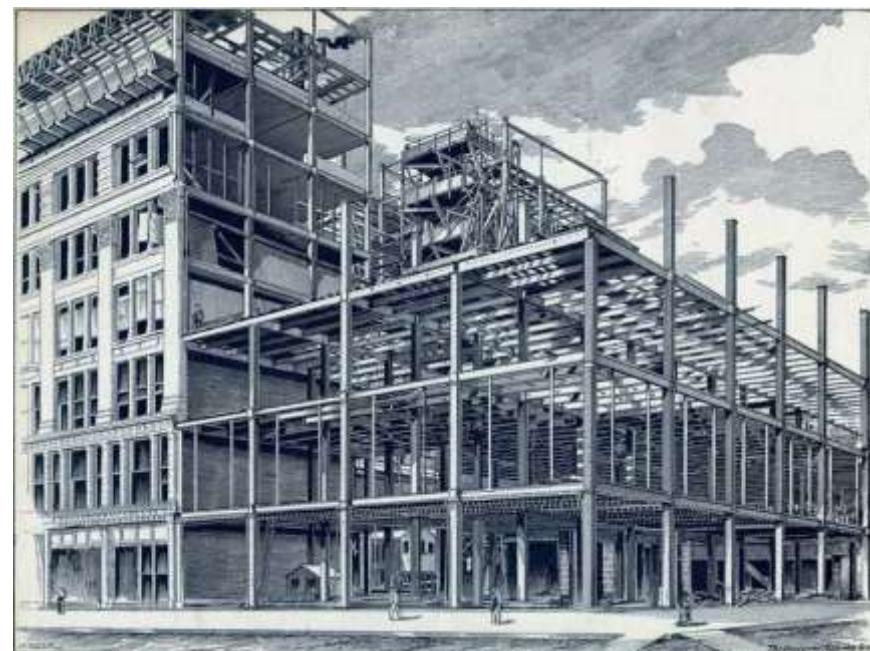
- Za osiguranje održavanja i pouzdanosti, oblikovana arhitektura mora biti stabilna
 - dodavanje novih karakteristika treba biti jednostavno i treba donijeti minimalne promjene u arhitekturi

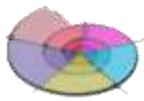


Razvoj modela arhitekture



- Započeti izradom kostura
 - koristiti osnovne zahtjeve i oblikovne obrasce
 - odrediti neophodne osnovne komponente
 - odabrati jedan od uzoraka arhitekture
- *Preporuka:*
 - koristiti različite timove
 - uzeti najbolje ideje





Razvoj modela arhitekture



- Poboljšanje arhitekture
 - utvrđivanje osnovnih načina interakcije komponenti i odgovarajućih sučelja
 - odluka o raspoređivanju dijelova podataka i funkcionalnosti po komponentama
 - utvrđivanje mogućnosti ponovne uporabe postojećih arhitektura, mogućnost izgradnje nove arhitekture
- Posudba oblikovnih obrazaca i prilagodba arhitekture za njihovo ostvarenje
- Zrelost arhitekture

PRINCIPI OBLIKOVANJA ARHITEKTURE

Principi oblikovanja

1. **Podijeli pa vladaj** – engl. *Divide and conquer*
2. **Povećaj koheziju** - engl. *Increase cohesion where possible*
3. **Smanji međuovisnost** - engl. *Reduce coupling where possible*
4. **Zadrži (višu) razinu apstrakcije** - engl. *Keep the level of abstraction as high as possible*
5. **Povećaj ponovnu uporabivost** - engl. *Increase reusability where possible*
6. **Povećaj uporabu postojećeg** - engl. *Reuse existing designs and code where possible*
7. **Oblikuj za fleksibilnost** - engl. *Design for Flexibility*
8. **Planiraj zastaru** - engl. *Anticipate Obsolescence*
9. **Oblikuj za prenosivost** - engl. *Design for Portability*
10. **Oblikuj za ispitivanje** - engl. *Design for Testability*
11. **Oblikuj konzervativno** - engl. *Design Defensively*
12. **Oblikuj po ugovoru** - engl. *Design by Contract*

1: Podijeli pa vladaj

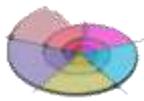
- engl. *Divide and conquer*
- Jednostavniji rad s više malih dijelova
 - odvojeni timovi rade na manjim problemima
 - omogućava specijalizaciju
 - manje komponente – povećana razumljivost
 - olakšana zamjena dijelova
 - bez opsežne intervencije u cijeli sustav



Primjeri programskih sustava



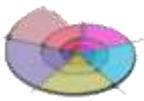
- Raspodijeljeni sustavi – klijenti i poslužitelji
- Podjela sustava u podsustave
- Podjela podsustava u pakete
- Podjela paketa u razrede



2: Povećanje kohezije



- engl. *Increase cohesion where possible*
- Podsustav ili modul ima veliku koheziju ako grupira međusobno povezane elemente, a sve ostalo stavlja izvan grupe
- Olakšava razumijevanje i promjene u sustavu
 - klasifikacija
 - Funkcijska – engl. *Functional*;
 - Razinska – engl. *Layer*;
 - Komunikacijska – engl. *Communicational*;
 - Sekvencijska – engl. *Sequential*;
 - Proceduralna - engl. *Procedural*;
 - Vremenska – engl. *Temporal*;
 - Korisnička - engl. *Utility*.



Funkcijska kohezija



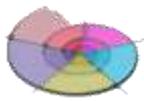
- engl. *Functional cohesion*
- Kôd koji obavlja pojedinu operaciju je grupiran, sve ostalo izvan
 - npr. modul obavlja jednu operaciju, vraća rezultat bez popratnih efekata
- Prednosti:
 - olakšano razumijevanje
 - povećana ponovna uporabljivost modula
 - lakša zamjena
- Nefunkcionalna kohezija:
 - modul mijenja bazu podatka, stvara datoteku, interakcija s korisnikom



Razinska kohezija



- Kohezija u istoj razini, engl. *Layer cohesion*
- Svi resursi za pristup skupu povezanih usluga na jednom mjestu, sve ostalo izvan
 - razine formiraju hijerarhiju
 - Viša razina može pristupiti uslugama niže razine
 - Niža razina ne pristupa višoj
 - skup procedura kojima pojedina razina omogućava pristup servisima naziva se aplikacijsko programsko sučelje – engl. *application programming interface (API)*
 - moguća zamjena pojedine razine bez utjecaja na ostale (više ili niže) razine



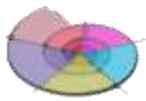
Komunikacijska kohezija



- engl. *Communicational cohesion*
- Svi moduli koji pristupaju ili mijenjaju određene podatke su grupirani, sve ostalo izvan
 - razred ima dobru komunikacijsku koheziju
 - Sadrži sve sistemske usluge neophodne za rad sa podacima
 - Ne obavlja ništa drugo osim upravljanja podacima
 - prednost:
 - Prilikom promjene podatka sav kod na jednom mjestu

Sekvencijska kohezija

- engl. *Sequential cohesion*
- Grupiranje procedura u kojoj jedna daje ulaz sljedećoj, sve ostale izvan
 - postizanje sekvencijske kohezije provodi se nakon svih prethodnih kohezijskih tipova.



Proceduralna kohezija



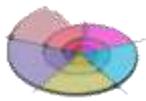
- engl. *Procedural cohesion*
- Procedure koje se upotrebljavaju jedna nakon druge
 - ne moraju razmjenjivati informacije
 - slabija od sekvencijske



Vremenska kohezija



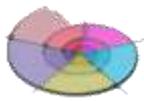
- engl. *Temporal Cohesion*
- Operacije koje se obavljaju tijekom iste faze rada programa su grupirane, sve ostalo izvan
 - podizanje sustava ili inicijalizacija
 - slabije od proceduralne kohezije



Kohezija pomoćnih programa



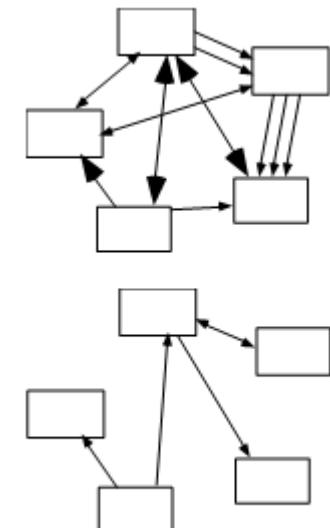
- engl. *Utility cohesion*
- Povezani pomoćni programi (engl. *utilities*) koji se logički ne mogu smjestiti u ostale grupe
 - procedura ili razred koji je široko primjenjiv za različite sustave
 - npr. **java.lang.Math**

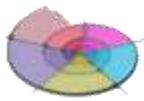


3: Smanjivanje međuovisnosti

- engl. *Reduce coupling where possible*
- *Povezivanje se javlja u slučaju međuovisnosti modula*
 - Međuovisnost \Rightarrow promjene na jednom mjestu zahtijevaju i promjene drugdje
 - Kod velike međuovisnosti teško je jasno raspoznati rad komponente.
 - Tipovi međuovisnosti:

- Međuovisnost sadržaja - engl. Content
- Opća međuovisnost - engl. Common
- Upravljačka međuovisnost - engl. Control
- Međuovisnost u objektnom oblikovanju - engl. Stamp
- Podatkovna međuovisnost - engl. Data
- Povezivanje poziva procedura - engl. Routine Call
- Međuovisnost tipova - engl. Type use
- Međuovisnost uključivanjem - engl. Inclusion/Import
- Vanjska međuovisnost - engl. External





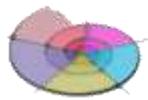
Međuovisnost sadržaja



- engl. *Content coupling*
- Jedna komponenta prikriveno mijenja interne podatke druge komponente
- OO oblikovanje:
 - za smanjivanje međuovisnosti sadržaja koristi se enkapsulacija svih instanci varijabli
 - Deklarirati ih kao *private*
 - Osigurati *get* i *set* metode
 - najgori oblik međuovisnosti sadržaja javlja se kod izravne promjene varijable od instancirane varijable (višerazinska međuovisnost).

Opća međuovisnost

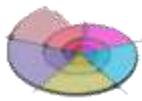
- engl. *Common coupling*
- Pri uporabi globalne varijable
 - sve komponente koje ju upotrebljavaju postaju povezane
 - slabiji oblik međuovisnosti je pristupanje varijabli unutar istog paketa
 - Npr. Java package
 - globalne varijable mogu biti prihvatljive za postavljanje tipičnih vrijednosti sustava (engl. *default*).



Upravljačka međuovisnost



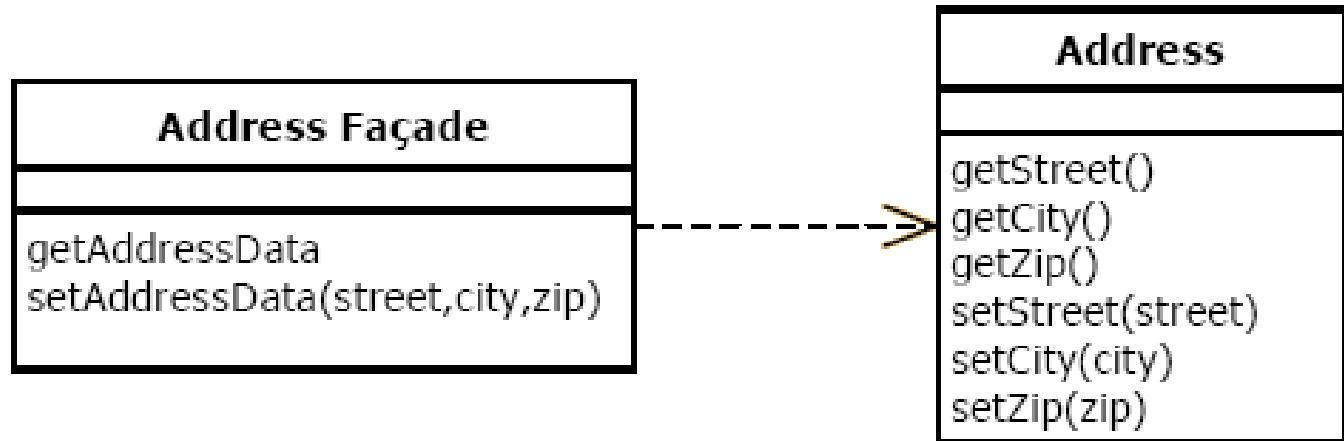
- engl. *Control coupling*
- Izravna kontrola rada druge procedure uporabom zastavice ili naredbe
 - za promjenu potrebno mijenjati obje procedure
 - izbjegavanje
 - uporabom polimorfnih operacija u objektnom pristupu. Tijekom rada određuje se koju proceduru treba pozvati i kako se ona izvodi.
 - *look-up* tablice
 - Pridruživanje naredbi odgovarajućoj metodi

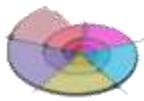


Vanjska međuovisnost



- engl. *External coupling*
- Predstavlja ovisnost modula o OS, knjižnici, HW, ...
 - minimizirati broj mesta na kojima se javlja takva povezanost
 - u objektnom pristupu, oblikovati malo sučelje prema vanjskim komponentama engl. *The Façade design pattern*





4: Zadrži razinu apstrakcije



- engl. *Keep the level of abstraction as high as possible*
- Osigurati da oblikovanje omogući sakrivanje ili odgodu razmatranja detalja, te na taj način smanji složenost
 - dobra apstrakcija podrazumijeva skrivanje informacija (engl. *information hiding*)
 - predstavlja temelj Objektno usmjerenog pristupa.
 - 1972. Parnas D.L.: *On Criteria to be Used in Decomposing Systems Into Modules*.
 - omogućava razumijevanje suštine podsustava bez poznavanja nepotrebnih detalja.



Apstrakcije u objektnom oblikovanju



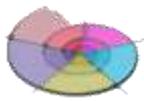
- engl. *Abstraction and classes (OO design)*
- Razredi (engl. *Class*) su podatkovne apstrakcije koje sadrže proceduralne apstrakcije (metode)
 - razina apstrakcije se povećava definiranjem privatnih varijabli
 - one su nedostupne izvan dosega
 - apstrakcija se poboljšava smanjivanjem broja javnih metoda
 - nadrazredi/nasljeđivanja i sučelja (engl. *Superclass, interface*) povećavaju razinu apstrakcije
 - atributi i pridruživanja predstavljaju podatkovne apstrakcije
 - dvije vrste varijabli primjeraka
 - metode predstavljaju proceduralne apstrakcije
 - Apstrakcija se povećava sa smanjivanjem broja argumenata procedura (metoda).



5: Povećaj ponovnu uporabivost



- engl. *Increase reusability where possible*
- Oblikovanje različitih aspekata sustava tako da može pridonijeti ponovnoj uporabi
 - poopćavanje oblikovanja u što većoj mjeri
 - uporaba prethodnih principa oblikovanja *Povećaj koheziju, Smanji povezanost, Zadrži razinu apstrakcije*
 - oblikuj sustav tako da sadrži kopče/sučelje (engl. *hooks*) koje omogućava pristup u program dodatnom korisničkom kodu
 - Korisnik vidi kopče kao otvore u kodu koji su dostupni u trenutku pojave nekog događaja ili zadovoljenja nekog uvjeta.
 - maksimalno pojednostavi oblikovanje



6: Povećaj uporabu postojećeg



- engl. *Reuse existing designs and code where possible*
- Princip je komplementaran povećanju ponovne uporabivosti.
 - što veća aktivna ponovna uporaba komponenti
 - smanjuje trošak i povećava stabilnost sustava
 - korištenje prethodnih investicija
 - Kopiranje i umetanje, “kloniranje” (engl. *cloning*) se ne razmatra kao uporaba postojećeg dijela koda

7: Oblikuj za fleksibilnost

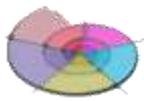
- engl. *Design for flexibility*
- Aktivno predviđaj buduće moguće promjene i provedi pripremu za njih
 - smanji povezivanje (međuvisnost) i povećaj koheziju
 - stvaraj apstrakcije
 - ne upotrebljavaj izravno umetanje podataka ili konfiguracija u izvorni programski kôd (engl. *hard code*)
 - ostavi otvorene opcije za eventualne modifikacije
 - upotrebljavaj postojeći kod
 - Za kôd koji radiš teži da zadovolji što laksu i veću ponovnu uporabu

8: Planiraj zastaru

- engl. *Anticipate obsolescence*
- Planiraj promjene u tehnologiji ili okolini na taj način da program može raditi ili biti jednostavno promijenjen
 - izbjegavaj uporabu novih tehnologija ili njihovih novih inačica (engl. *release*)
 - izbjegavati knjižnice namijenjene specifičnim okolinama
 - izbjegavaj nedokumentirane ili rijetko upotrebljavane dijelove knjižnica
 - izbjegavaj SW/HW bez izgleda za dugotrajniju podršku
 - uporaba tehnologija i jezika podržanih od više dobavljača

9: Oblikuj za prenosivost

- engl. *Design for Portability*
- Omogućiti rad na što većem broju različitih platformi
 - izbjegavaj specifičnosti neke okoline
 - Npr. specifičnosti okoline razvojnog okruženja



10: Oblikuj za ispitivanje



- engl. *Design for Testability*
- Olakšaj ispitivanje
 - oblikuj program za automatsko ispitivanje
 - omogući odvojeno pokretanje svih funkcija uporabom vanjskih programa (npr. bez grafičkog sučelja)
 - npr. u Javi, u svakom razredu stvori metodu `main()` za jednostavnije ispitivanje drugih metoda.

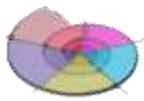
11: Oblikuj konzervativno

- engl. *Design Defensively*
- Ne koristiti pretpostavke kako će netko koristiti oblikovanu komponentu
 - obradi sve slučajeve u kojima se komponenta može neprikladno upotrijebiti
 - provjeri valjanost ulaza u komponentu provjerom definiranih pretpostavki
 - Pretjerano obrambeno oblikovanje – često dovodi do nepotrebnih provjera

12. Oblikuj po ugovoru

- engl. *Design by Contract*
- Tehnika koja omogućava učinkovit i sustavni pristup konzervativnom oblikovanju
 - osnovna ideja
 - Sve metode imaju ugovor s pozivateljima
 - ugovaratelj ima skup zahtjeva
 - Preduvjete (engl. *preconditions*) koje mora ispuniti pozvana metoda kada započinje izvođenje
 - Završne uvjete (engl. *postconditions*) koje pozvana metoda mora osigurati kod završetka izvođenja
 - Invarijante (engl. *invariants*), varijable na koje pozvana metoda neće djelovati pri izvođenju

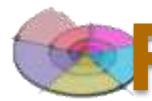
DOKUMENTI OBLIKOVANJA ARHITEKTURE PROGRAMSKE POTPORE



Dokumentiranje arhitekture



- Zašto dokumentirati?
 - zabilježiti odluke arhitekta. Kompletnost i jednoznačnost.
 - priopćavanje arhitekture. Komunikacija dionika.
- Potrebno zbog rane analize sustava
- Temeljni nositelj obilježja kvalitete
- Ključ za održavanje, poboljšanja i izmjene PP
- Dokumentacija ne zastarijeva
 - dugoročno govori umjesto arhitekta (ukoliko je sustav oblikovan, održavan i mijenjan sukladno dokumentaciji).
- U praksi je danas dokumentacija često nejednoznačna i kontradiktorna.
 - najčešće samo pravokutnici i linije koje mogu značiti: A šalje upravljačke signale do B, A šalje podatke do B, A šalje poruku do B, A kreira B, A dobavlja vrijednost od B, ...



- Dokumenti oblikovanja su pomoć izradi boljih sustava
 - traže izričitost i obrađuju najvažnija pitanja prije faze implementacije
 - omogućuju vrednovanje oblikovanja i poboljšanje
 - sredstvo komunikacije
 - tima za implementaciju
 - budućih osoba zaduženih za održavanje, razvoj ili promjene
 - osobama uključenih u oblikovanje povezanih sustava ili podsustava
 - pisati ih s gledišta čitatelja
 - upotrebljavati standardnu organizaciju

- Referentna specifikacija - engl. *Reference Specification*
 - potpuni skup dokumentiranih pokretača arhitekture (engl. *architecture drivers*), pogleda te pomoćne dokumentacije poput matrica odluka.
- Pregled za upravu - engl. *Management Overview*
 - pregled visokog nivoa, vizija sustava, poslovni motivi, koncepti arhitekturnih dijagrama, poveznice poslovne i tehničke strategije
- Dokumentacija komponenti - engl. *Component Documents*
 - za komponente se osigurava pogled na razini sustava (engl. *Logical Architecture Diagram*), specifikacija komponente, specifikacija sučelja te dijagrami suradnje



Struktura dokumenta oblikovanja



- A. Svrha - engl. *Purpose*
 - koji sustav ili dio sustava ovaj dokument opisuje
 - označene poveznice prema dokumentu zahtjeva na koji se ovaj dokument oslanja - slijedivost
- B. Opći prioriteti - engl. *General priorities*
 - opiši prioritete koji su vodili proces oblikovanja (npr. 4+1 pogled ...)
- C. Skica sustava - engl. *Outline of the design*
 - navedi opis sustava s najviše razine promatranja kako bi čitatelj razumio osnovnu ideju
- D. Temeljna pitanja u oblikovanju - engl. *Major design issues*
 - diskutiraj osnovne probleme koji su se morali razriješiti
 - navedi razmatrane opcija rješenja, konačnu odluku i razloge za njeno donošenje
- E. Detalji oblikovanja - engl. *Other details of the design*
 - predviđati sve ostale detalje koji su čitatelju zanimljivi a u dokumentu još nisu razmatrani



Pisanje dokumentacije



- Preporuka u sadržaju dokumenta oblikovanja:
 - izbjegavati dokumentiranje informacija očitih iskusnim programerima i dizajnerima
 - ne pisati detalje koji su dio komentara koda
 - ne pisati detalje koji su vidljivi u strukturi koda
- Arhitektura programske podrške mora rezultirati dokumentacijom koja ima slijedeća svojstva:
 - ***dobra***
 - Doseže potrebe i ciljeve ključnih dionika
 - ***ispravna***
 - Tehnički ispravna i jasno prezentirana
 - ***uspješna***
 - Upotrebljava se u stvarnom razvoju sustava kojim se postižu strateške prednosti

Diskusija

-
-
-
-
-

Oblikovanje programske potpore

ak.god. 2014./2015.

Modularizacija i objektno usmjerena arhitektura



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



Tema

- Podsjetnik
 - metode programskog inženjerstva
- Programske paradigmе
- Objektno usmjerena paradigma
- Koncepti objektnog usmjerjenja
 - Objekt, Razred, Nasljeđivanje, Polimorfizam

Literatura

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



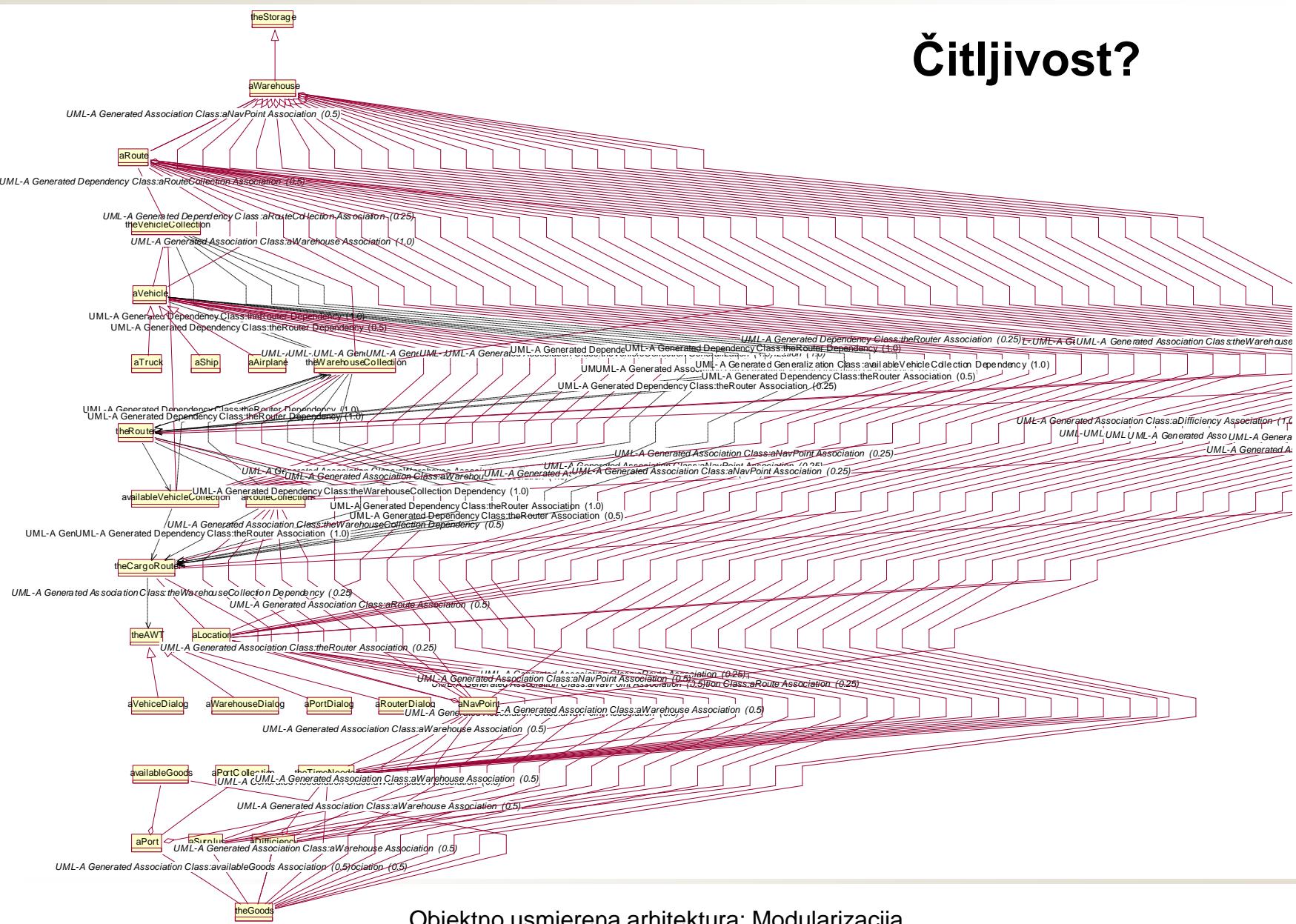
Problemi u oblikovanju programske potpore



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

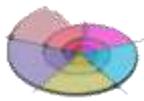
Primjer programa

Čitljivost?



Modularizacija

- Moguće rješenje problema:
- Modularizacija
 - jednostavnije upravljanje sustavom
 - princip podijeli pa vladaj
 - evolucija sustava
 - promjene jednog dijela ne utječu na druge dijelove
 - razumijevanje
 - sustav se sastoji od razumno složenih dijelova
- Koje kriterije koristiti za modularizaciju?
- Što je to modul?
 - dio koda
 - jedinica kompilacije, koja uključuje deklaracije i sučelje.
 - d. Parnas, Comm. ACM, 1972. : "Jedinica posla."



Povijest modularizacije



Glavni program i potprogrami

- dekompozicija u procesne korake s jednom niti izvođenja.

Funkcijski moduli

- agregacija (skupljanje) procesnih koraka u module.

Apstraktni tipovi podataka

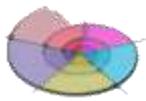
- engl. *Abstract Data Types - ADT*
- zatvaranje podataka i operacija, skrivanje predstavljanja.

Objekti i objektno usmjerena arhitektura

- procedure (metode) se povezuju dinamički, polimorfizam, nasljedivanje.

Komponente i oblikovanje zasnovano na komponentama (engl. *Component based design CBD*)

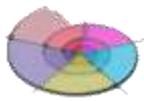
- višestruka sučelja, posrednici, binarna kompatibilnost.



Glavni program i potprogrami



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

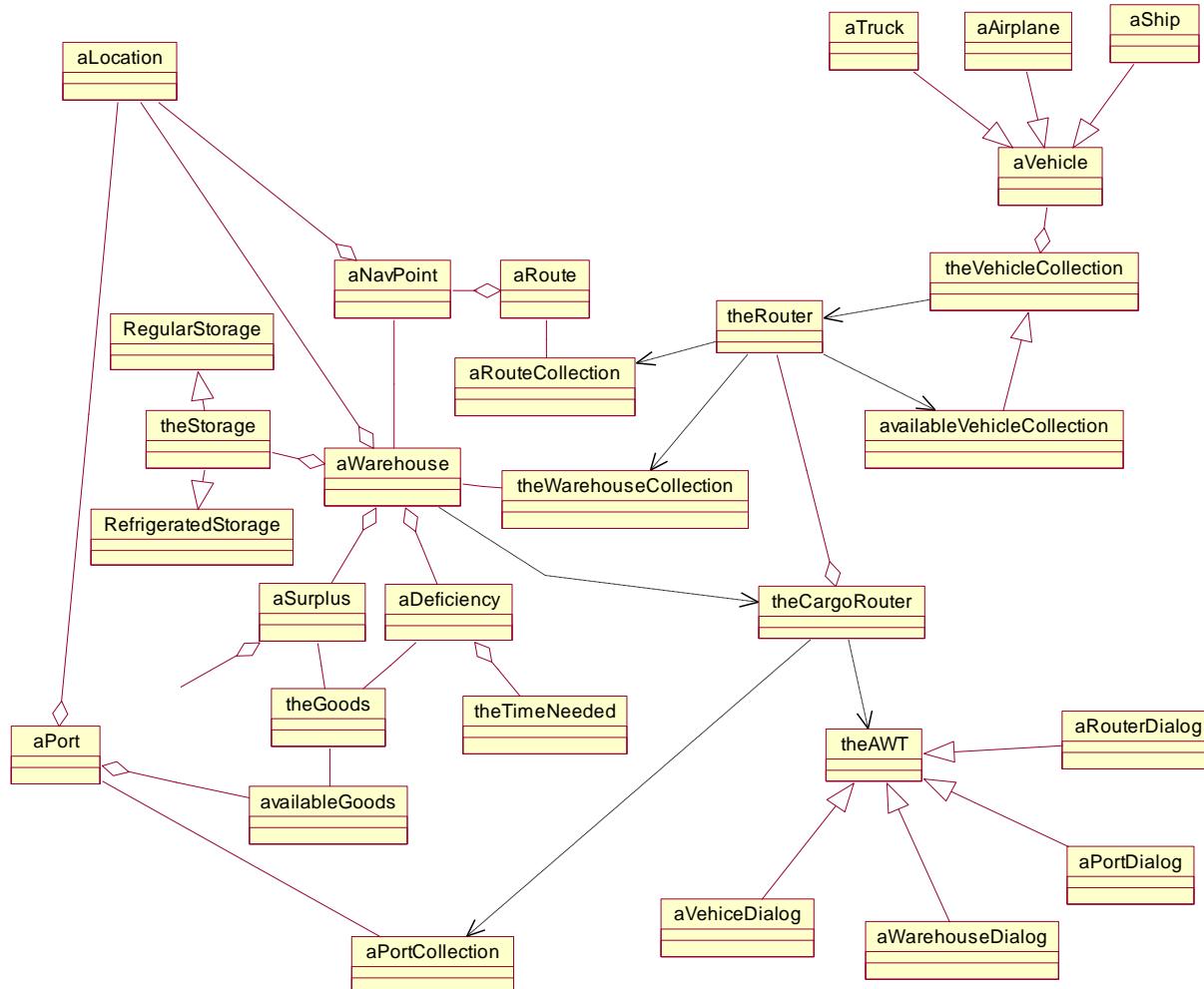


Apstraktni tipovi podataka



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

Primjer programa: Apstrakcija



Bolje?



Primjer: Apstraktni tipovi podataka



neka k označuje cijeli broj (*integer*).

ADT integer

podatak – engl. *Data*

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

takov jedan cijeli broj s predznakom neka je N .

operacije – engl. *Operations*

constructor – kreira novi cijeli broj.

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

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

set(k) – Posljedica je $N=k$.

...

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



Primjer: Apstraktni tip podatka



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

Primjer programa koji eliminira duplike u listi L:

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

- Prednost
 - nezavisnost o strukturi podataka \Rightarrow moguća uporaba u bilo kojoj implementaciji liste
 - promjena implementacije liste ne utječe na ovaj kod.



Proceduralna paradigma



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



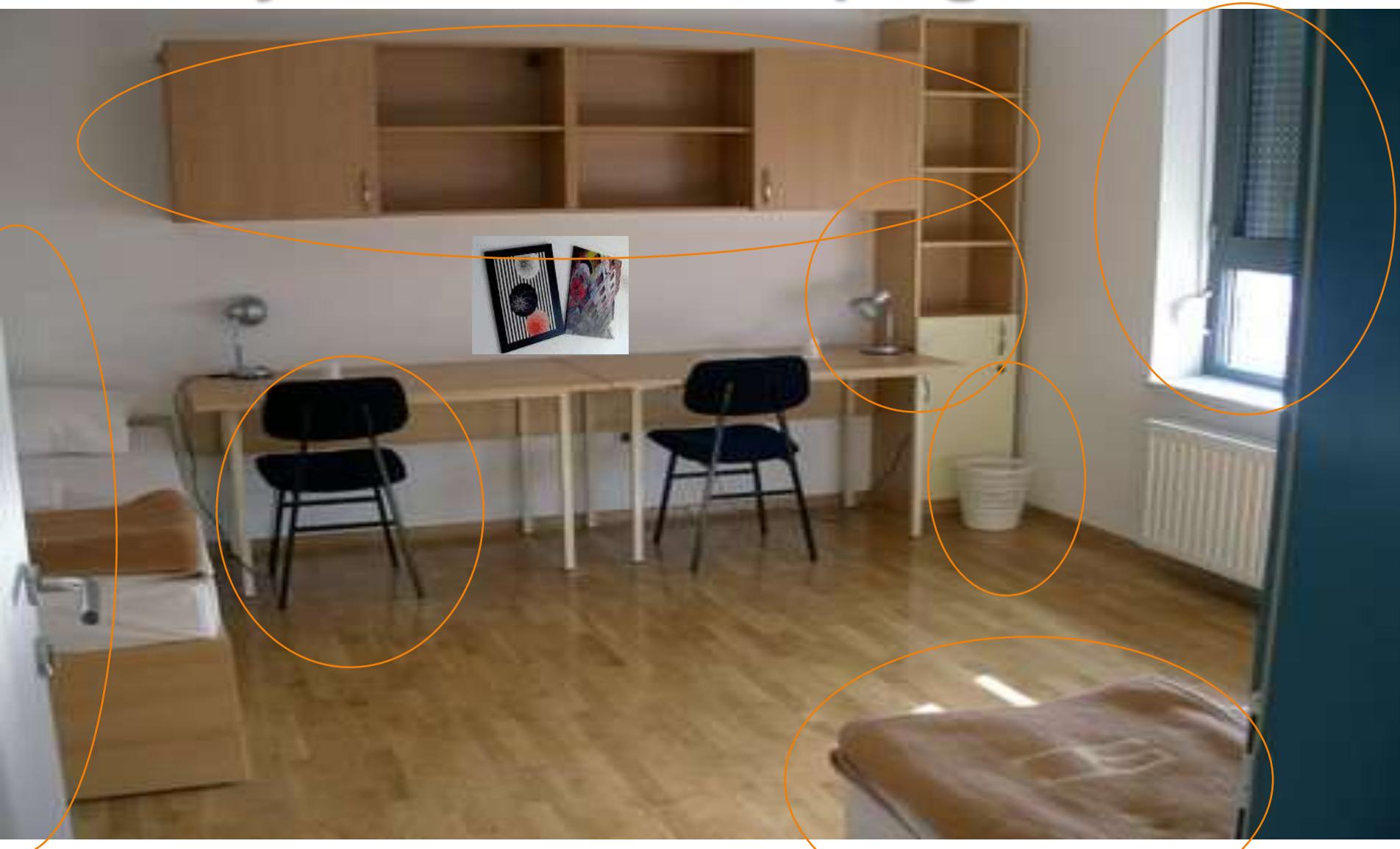
Objektno usmjerena paradigma



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



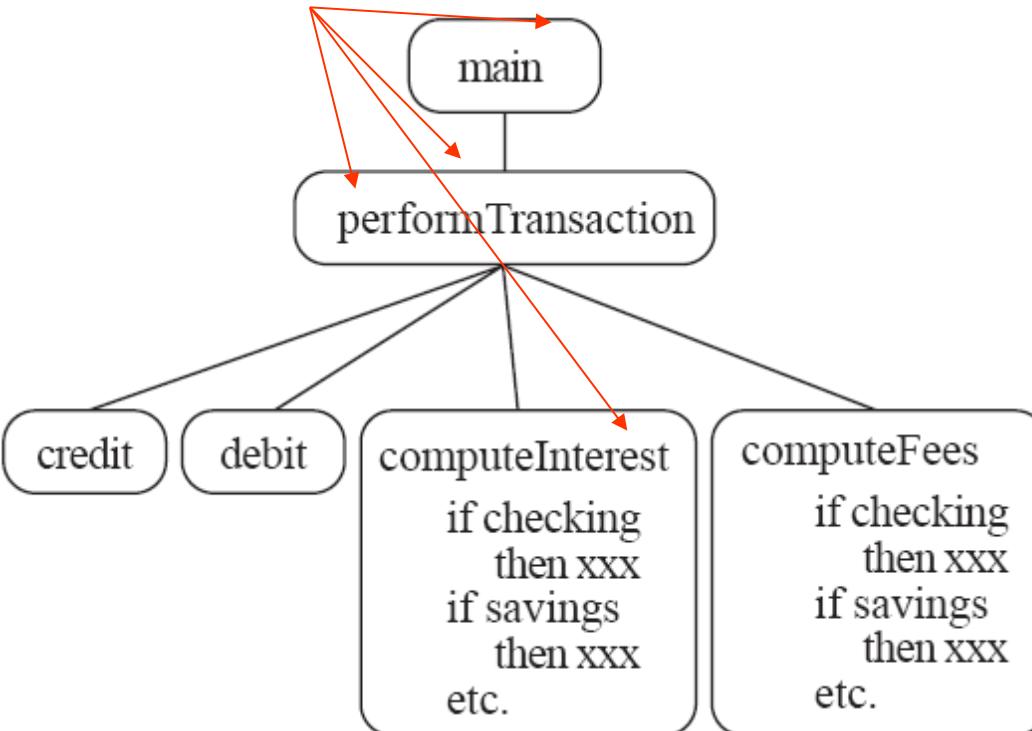
Objekti kako ih vidi programer...



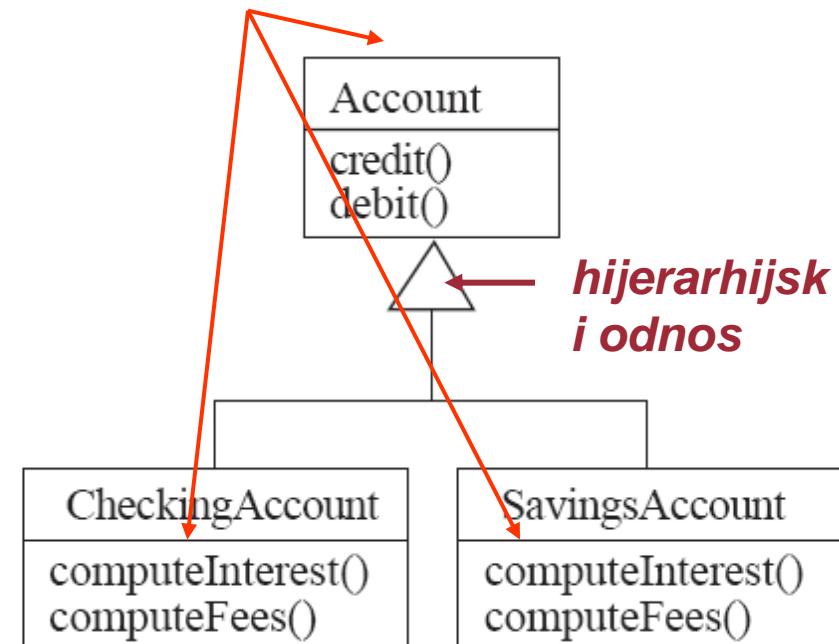
Izvor: <http://www.sczg.unizg.hr/>

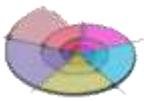
Primjer paradigm

Procedure manipuliraju različitim tipovima podataka



Procedure su svezane (zatvorene) s podacima





Koncepti objektnog usmjerenja



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

Osnovni principi objektnog usmjerenja

■ Apstrakcija

- olakšava savladavanje složenih problema
- objekt <- nešto u realnom svijetu
- razred <- objekti (instancije)
- nadrazred <- podrazred
- operacija <- metode
- atributi i pridruživanje (asocijacije) <- varijable instanci

■ Učahurivanje/Enkapsulacija

- detalji mogu biti skriveni u razredima.
- potiče skrivanje informacija (*engl. information hiding*):
 - Programeri ne moraju znati sve detalje razreda.

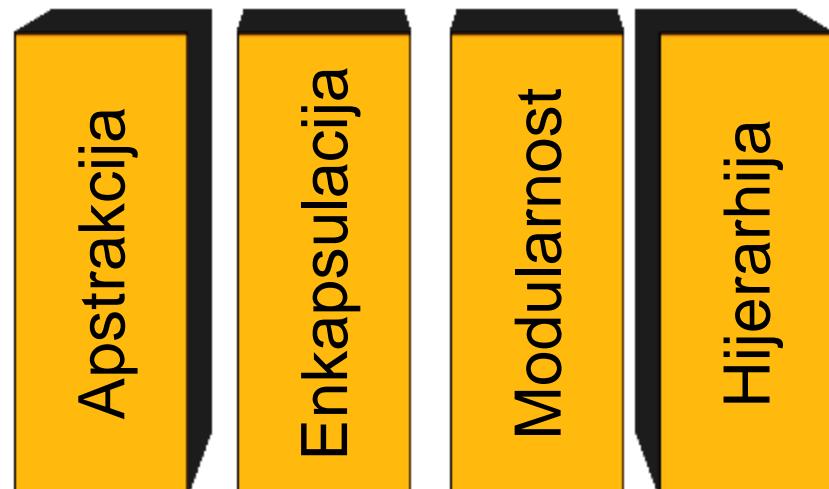
■ Modularnost

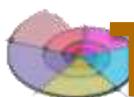
- program se može oblikovati samo iz razreda (bez globalnih varijabli?).

■ Hijerarhija

- elementi istog hijerarhijskog nivoa moraju biti na istom nivou apstrakcije

Objektno usmjereno

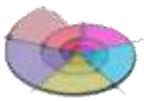




Terminologija objektnog oblikovanja



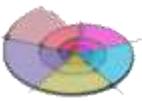
- Objekt – engl. *Object*
- Razred – engl. *Class*
- Atribut – engl. *Attribute*
- Metoda – engl. *Method*
- Operacija – engl. *Operation*
- Sučelje – engl. *Interface* (*Polymorphism*)
- Komponenta – engl. *Component*
- Paket – engl. *Package*
- Podsustav – engl. *Subsystem*
- Pridruživanja – engl.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



Objekt



- Objekt predstavlja osnovni entitet izvođenja u objektno-orientiranom sustavu
 - ima svoje stanje predstavljeno kao niz atributa i definirani skup operacija
 - objekt zauzima prostor u memoriji i ima pridruženu adresu
- Objekti su rezultat instanciranja razreda
 - instanciranje je proces uzimanja predloška (nacrta) i definiranja svih pridruženih atributa i ponašanja
 - pri tome se rezervira memorijski prostor za smještaj atributa i pridruženih metoda
 - stvaranje objekta: operator new + konstruktor razreda:
 - `new Ball();`
 - pri instanciranju vraća se referenca na objekt
 - `Ball b = new Ball();`
- Program u radu se može sagledati kao **skup objekata** koji u međusobnoj kolaboraciji obavljaju dani zadatak.



Svojstva objekta



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

objekt: bankovni_račun

atributi:
model
tip
oprema
...



Razlika instance - objekt

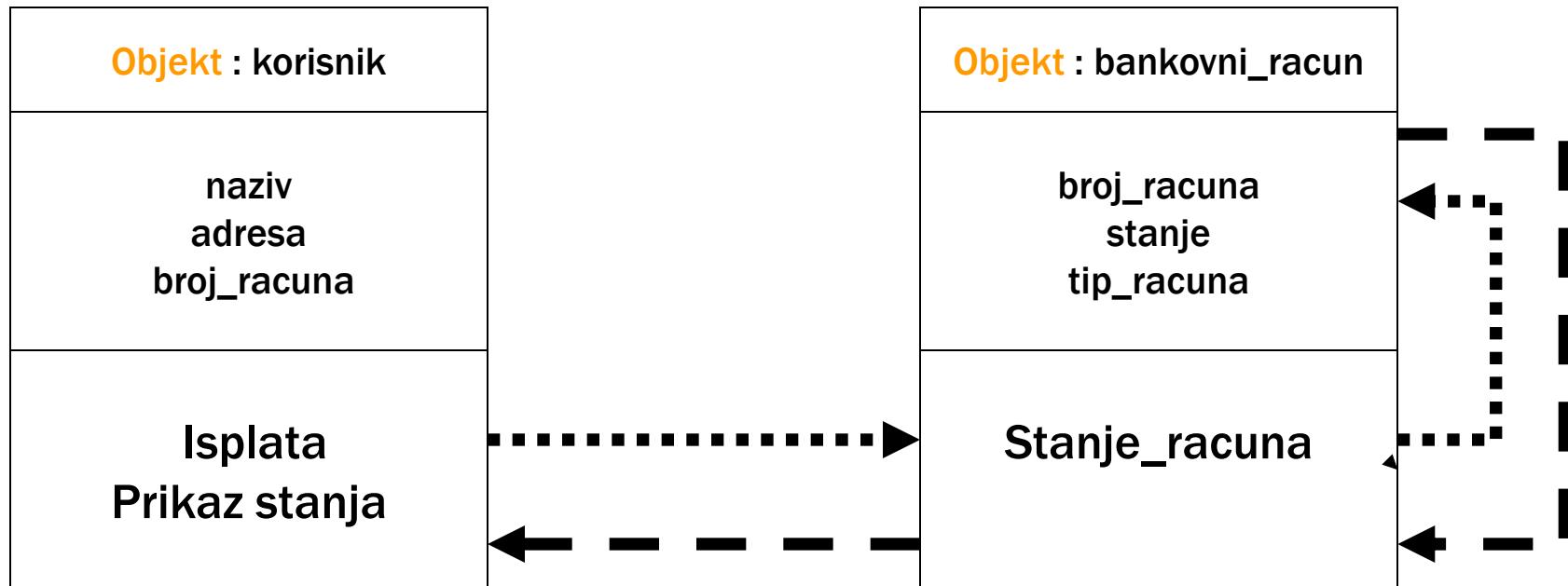


- Nema razlike, odnosi se na istu jedinku (entitet)
- Razlika je nastala u korištenju prirodnog jezika.
 - Npr. kćer – djevojka
 - “Imala je sedam kćeri” (ne djevojaka).
 - “Vidio sam prekrasnu djevojku” (ne kćer).
- Implementacijske razlike:
- Objekt:
 - memorija koja sadrži informacije o objektu
- Instanca:
 - referenca na objekt
 - pokazuje na početnu adresu na kojoj je objekt pohranjen
 - dvije instance mogu pokazivati na isti objekt
 - životni vijek instance i objekta nije povezan
 - kada su sve instance koje pokazuju na objekt obrisane briše se i objekt.

Primjer

■ Interakcija objekata

- razmjenom poruka (engl. *Message passing*)



Interakcija objekata

- razmjena poruka (engl. *Message passing*)
- Objektno usmjereni program sastoji se od skupa objekata koji komuniciraju razmjenom poruka
- Uobičajeni koraci:
 - Stvaranje razreda koji definiraju objekte i njihovo ponašanje
 - Stvaranje objekata iz razreda
 - Uspostavljanje komunikacije među objektima
- Objekti međusobno komuniciraju slanjem i primanjem poruka
 - poruka objektu je zahtjev za izvođenjem metode
 - objekt koji primi poruku poziva proceduru i generira rezultat
 - za razmjenu poruka neophodno specificirati:
 - Ime objekta
 - Ime metode.
 - prijenos informacija u obliku parametara.

Primjer: Objekti u bankovnom sustavu

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

Naziv objekta

Obilježje

Jane:
dateOfBirth="1955/02/02"
address="99 UML St."
position="Manager"

Savings account 12876:
balance=1976.32
opened="1999/03/03"

Greg:

dateOfBirth="1970/01/01"
address="75 Object Dr."

Margaret:

dateOfBirth="1984/03/03"
address="150 C++ Rd."
position="Teller"

Instant teller 876:

location="Java Valley Cafe"

Mortgage account 29865:

balance=198760.00
opened="2003/08/12"
property="75 Object Dr."

Transaction 487:

amount=200.00
time="2001/09/01 14:30"

RAZREDI

Razred

- Razred/Klasa – engl. **Class**
- Objektno usmjereno razmišljanje započinje definiranjem razreda (opći opis, predložak, nacrt)
 - korisnički definiran tip podataka
 - jedinica apstrakcije u objektno usmjerenoj paradigmi.
- Razredi predstavljaju slične objekte
 - opis/predložak za stvaranje objekata
 - objekti su **instance** razreda (proizvoljan broj).
- U objektno orijentiranim programskim jezicima razred je vrsta programskog modula koji sadrži:
 - opis strukture objekta, tj. **obilježja** (engl. **properties**)
 - podatci koji implementiraju obilježja i početne vrijednosti
 - sadrže **metode** (procedure) koje **implementiraju ponašanje** objekata.
 - npr. procedure, funkcije za promjenu obilježja

Razred

- Nivo detalja u specificiranju razreda ovisi o stanju razvojnog procesa
- Razred specificira naziv, atribute stanja i pridružene metode
 - umotava engl. *wrapping up*
 - podaci nisu dostupni izvan razreda
 - razredi predstavljaju apstraktni tip podataka

Class Name
attribute: Type = initialValue
....
method(arg list): return type
....



Odnos razreda i instance



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

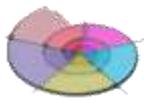
Imenovanje razreda

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

Primjer razreda

- Primjer razreda **Circle** u Javi:

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



Konstruktori i destruktori



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

```
class rectangle { // jednostavan razred
    int height;
    int width;

public:
    rectangle(void); // konstruktor
    ~rectangle(void); // destruktur
};

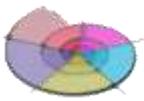
// inicijalizacija objekta
rectangle::rectangle(void) // konstruktor
{ height = 6; width = 6; }
```

Varijable instanci

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

Varijable instanci

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



Varijable i objekti



- Varijable i objekti su zasebni i različiti koncepti
 - PAZI: uobičajena zabuna
- Tip varijable
- Određuje objekte razreda na koje se varijabla referencira.
 - npr. u Javi postoje dva tipa varijabli:
 - **primitive** - sadrži jednu vrijednost, nije objekt, evaluira se u vrijednost koju sadrži.
 - **reference / object** - evaluira se u adresu objekta (slično pokazivaču ali u širem kontekstu)
 - u različitim trenucima može se odnositi na različite objekte.
 - jedan objekt može u isto vrijeme referencirati više različitih varijabli.
 - referenciranje je općenitiji pojam od "sadrži". U varijabli tipa "reference" se smješta adresa objekta na koji se referencira, a ne sam objekt.
 - objekt je dostupan preko varijable koja ga referencira!



Primjer: Odnos varijable i objekta



Neka postoji razred **Ball**.

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

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

Zadatak: Kreirati objekt iz razreda **Ball** tako da varijabla b1 referencira baš taj objekt:

```
ball b1 = new Ball();
      // rutina Ball() konstruira objekt iz razreda Ball.
                  // engl. constructor
```

Taj isti objekt može biti referenciran i od druge varijable:

```
ball b2 = b1;
```

Obje varijable referenciraju isti objekt iz razreda **Ball**.

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

Varijable razreda

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

Primjer: Varijable razreda

- Primjer varijable razreda:

```
public class Circle{  
    public static int num_circles; // varijabla razreda  
    public double x,y;           // varijsable instanci  
    public double r;              // varijsable instance  
    public double opseg(){return 2 * 3.14 *r};  
    public double povrsina(){return 3.14 * r *r} }
```

- Kako se pristupa varijabli razreda ?
- Preko naziva razreda (a ne preko objekta – tj. varijsable koja referencira taj objekt):

```
system.out.println("No of circles:" +  
                    circle.num_circles);
```

Metode razreda

- engl. *class methods, static methods*
- Metode definirane unutar nekog razreda i deklarirane ključnom riječi **static**
 - analogno varijablama razreda
- Primjer poziva takve metode:

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

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



Lokalne varijable



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

Parametri

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

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

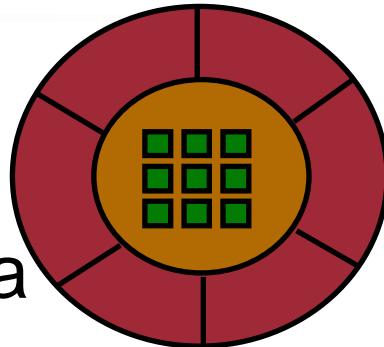
- **args** varijabla je parametar procedure (metode) **main**.

- Parametri se uvijek klasificiraju kao "varijable".
- To se odnosi i na druge konstrukcije koje prihvataju parametre (npr. *engl. constructors and exception handlers*).
- Broj parametara metoda u objektno usmjerenoj paradigmi treba biti što manji.

METODE

Metoda

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



Metode - primjer

Neka su u razredu Ball deklarirane i definirane dvije metode:

```
setSpeed(); // postavlja brzinu na int vrijednost  
getSpeed(); // očitava brzinu u int
```

Svi stvoren objekti iz Ball imaju te dvije metode.

Stvaranje objekta referenciranog varijablama b1 i b2:

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

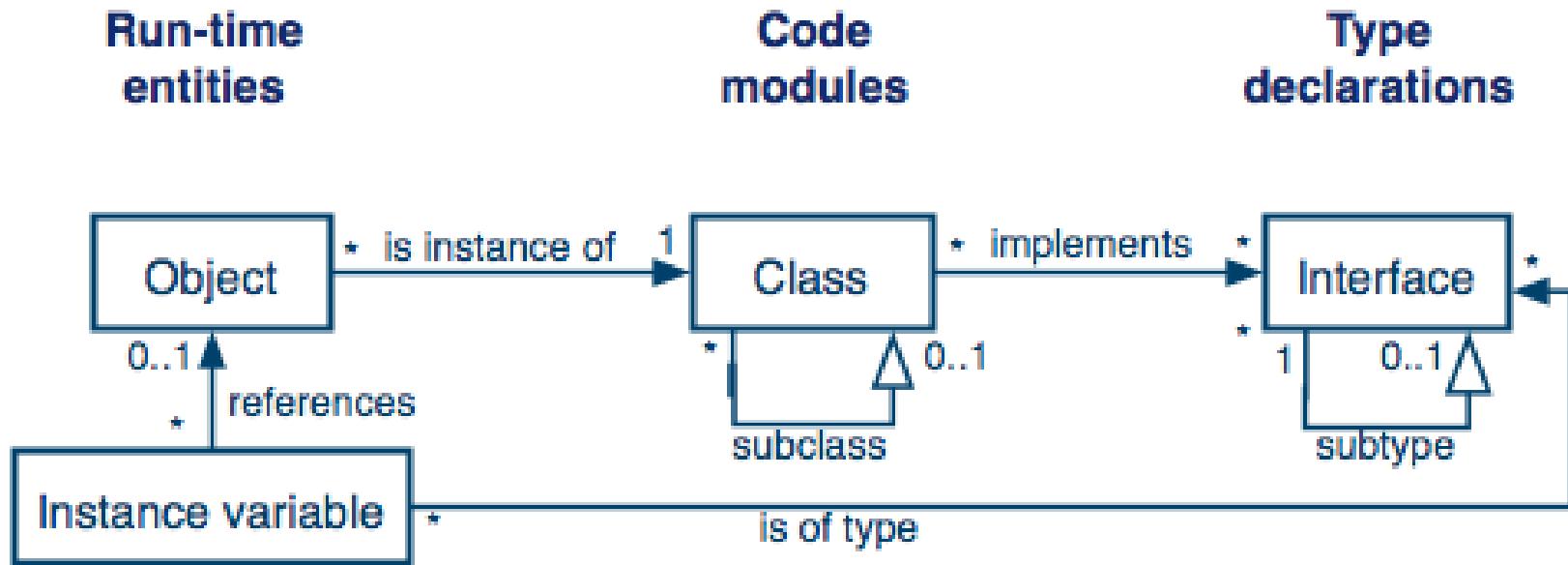
```
b1.setspeed(50) // u objektu kojega referencira  
                  // b1 postavlja brzinu na 50  
b2.setSpeed(100) // u objektu kojega referencira  
                  // b2 postavlja brzinu na 100
```

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

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

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

Pregled odnosa



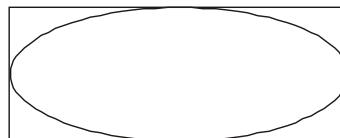
Operacija

- Proceduralna apstrakcija više razine nego metoda.
- Operacija specificira tip ponašanja.
- Jedna operacija može biti implementirana s više metoda.
- Neovisna je o kodu koji implementira njeni ponašanji.
 - npr. Izračun_površine

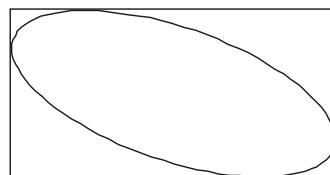
Primjer: operacije na grafičkim objektima

- operacije ništa ne govore o implementaciji

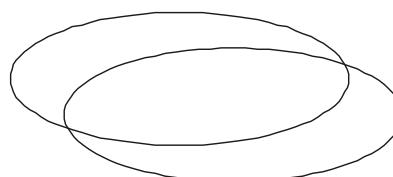
Original objects
(showing bounding rectangle)



Rotated objects
(showing bounding rectangle)



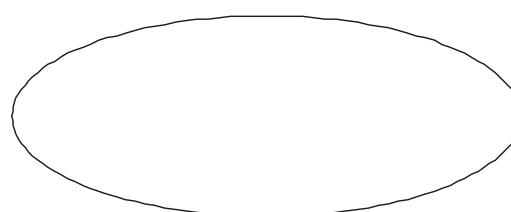
Translated objects
(showing original)



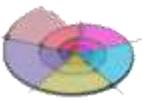
Scaled objects
(50%)



Scaled objects
(150%)



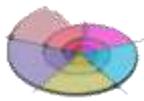
Izvor: T.C. Lethbridge, R. Laganière: Object-Oriented Software Engineering: Practical Software Development using UML and Java



Vidljivost operacija



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



Više metoda istog naziva



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

Primjer

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

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

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

Nasljeđivanje

- engl. *Inheritance*
- Način za podržavanje principa ponovne uporabe objekata
 - razredi mogu nasljeđivati značajke drugih podrazreda
- Proces kod kojeg se jedan razred stvara na temelju drugog tako da se dodaju specifična obilježja i ponašanje
- Omogućava dodavanje novih obilježja postojećem razredu bez modificiranja

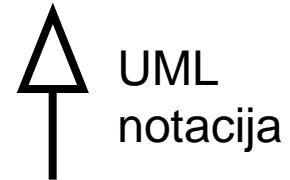


Organizacija razreda u hijerarhije



- nadrazredi (engl. *Superclasses*) (u C++ “base class”)
 - sadrže značajke zajedničke jednom skupu razreda.
- Hijerarhije nasljeđivanja (engl. *Inheritance hierarchies*)
 - pokazuju odnos između nadrazreda i podrazreda (engl. *superclasses i subclasses*).
 - oznaka trokuta pokazuje generalizaciju.
- Nasljeđivanje (engl. *Inheritance*)
 - svi podrazredi *implicitno* posjeduju značajki koje su definirane u nadrazredu.

```
public class MortgageAccount extends Account
{
    // dodatne naredbe obilježja i ponašanja (proširenje
    // Account-a)
}
```



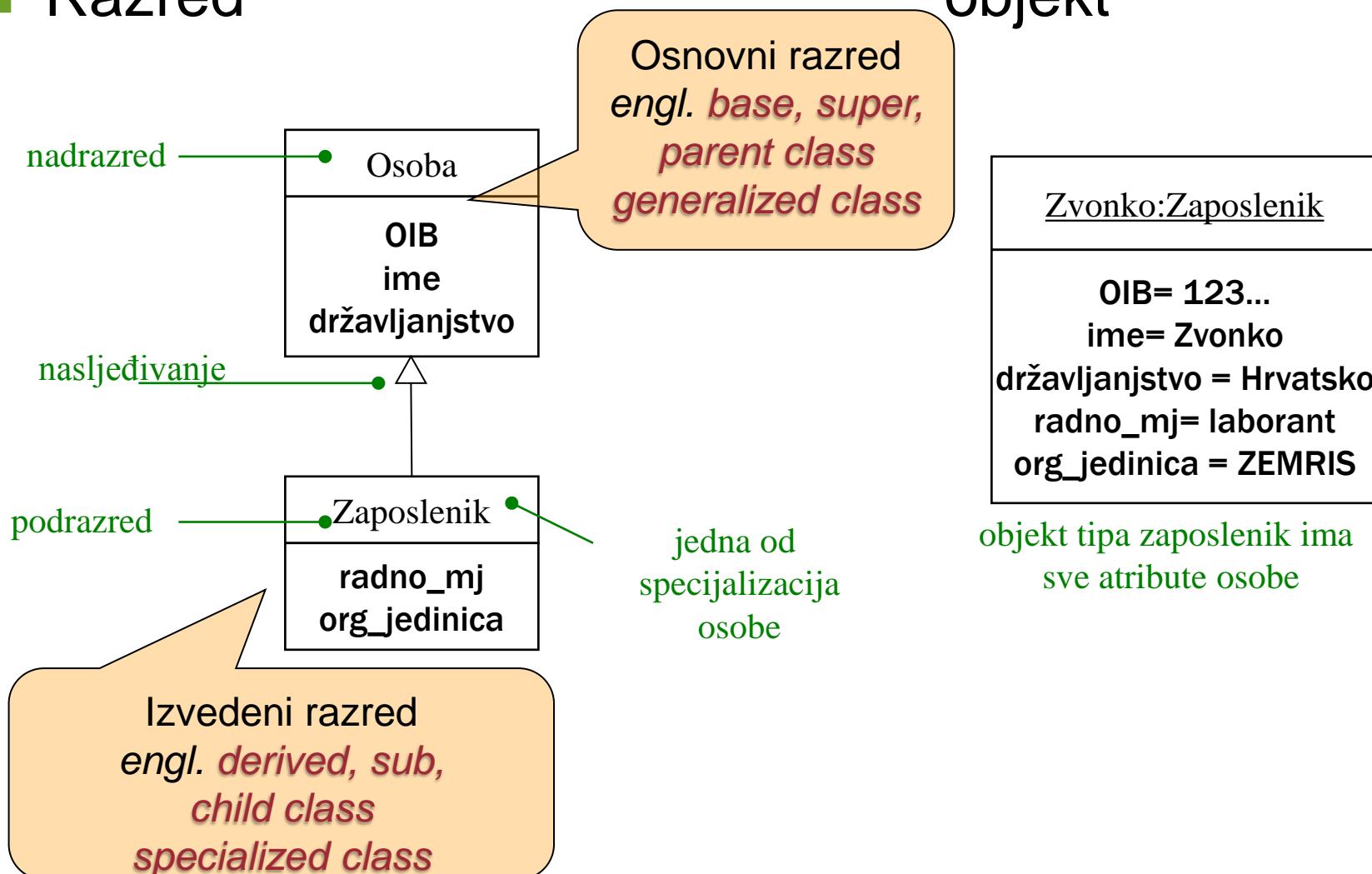
UML
notacija

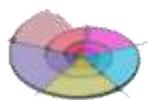


Primjer: hijerarhija nasljeđivanja



Razred





Pravila nasljeđivanja: “is-a” pravilo

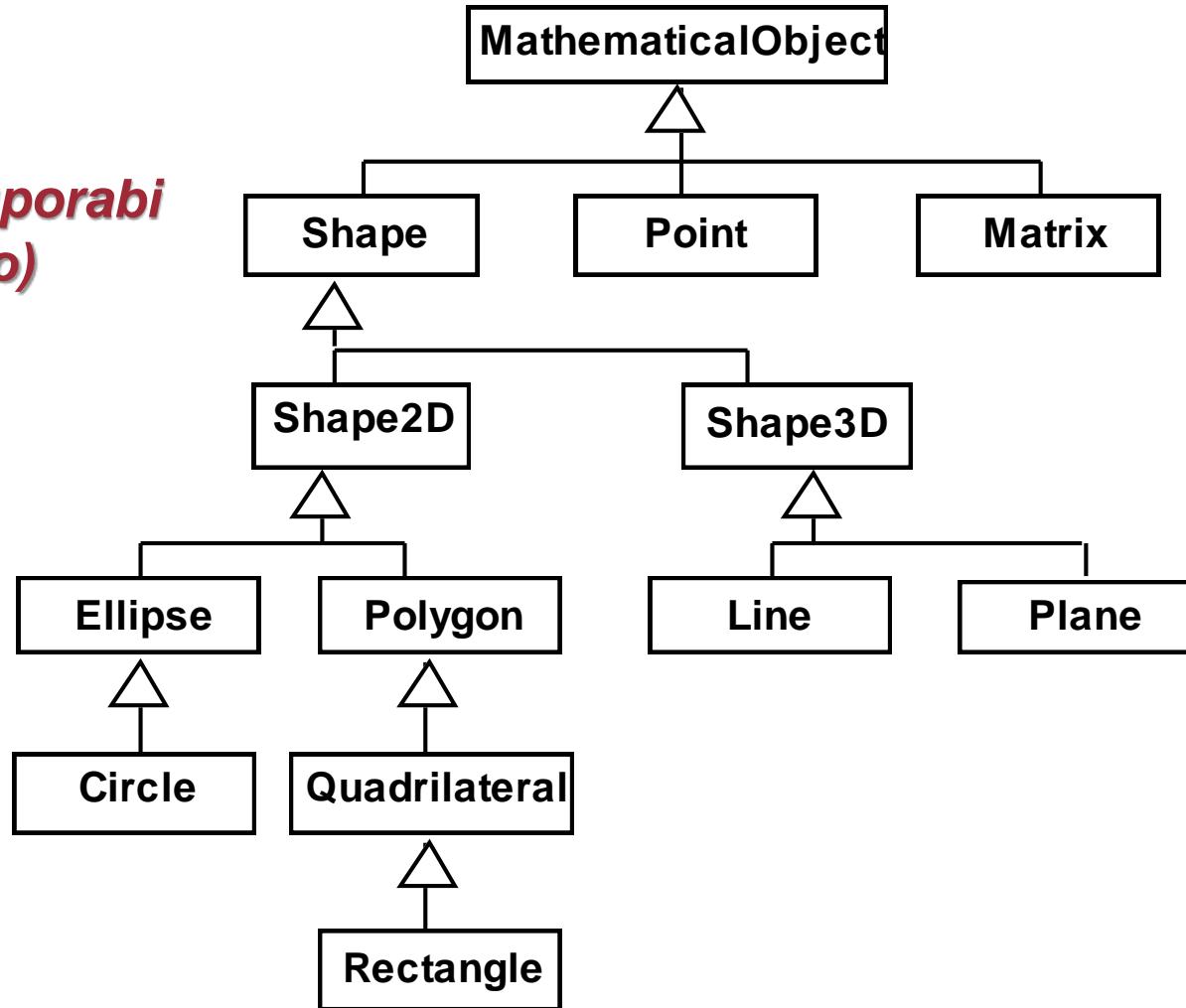


- engl. ‘*is a*’ *Hierarchy* - *ISA*
- Uvijek provjeri da li generalizacija zadovoljava “is a” pravilo (pravilo “je”, odnos podskup → skup).
 - “checking account is an account”
 - “village is a municipality”
- Da li bi “Županija” bila podrazredom “Država” ?
 - ne, jer ne zadovoljava “is-a” pravilo:
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

Primjer

- Moguća hijerarhija nasljeđivanja matematičkih objekata

*(U provjeri uporabi
“is-a” pravilo)*

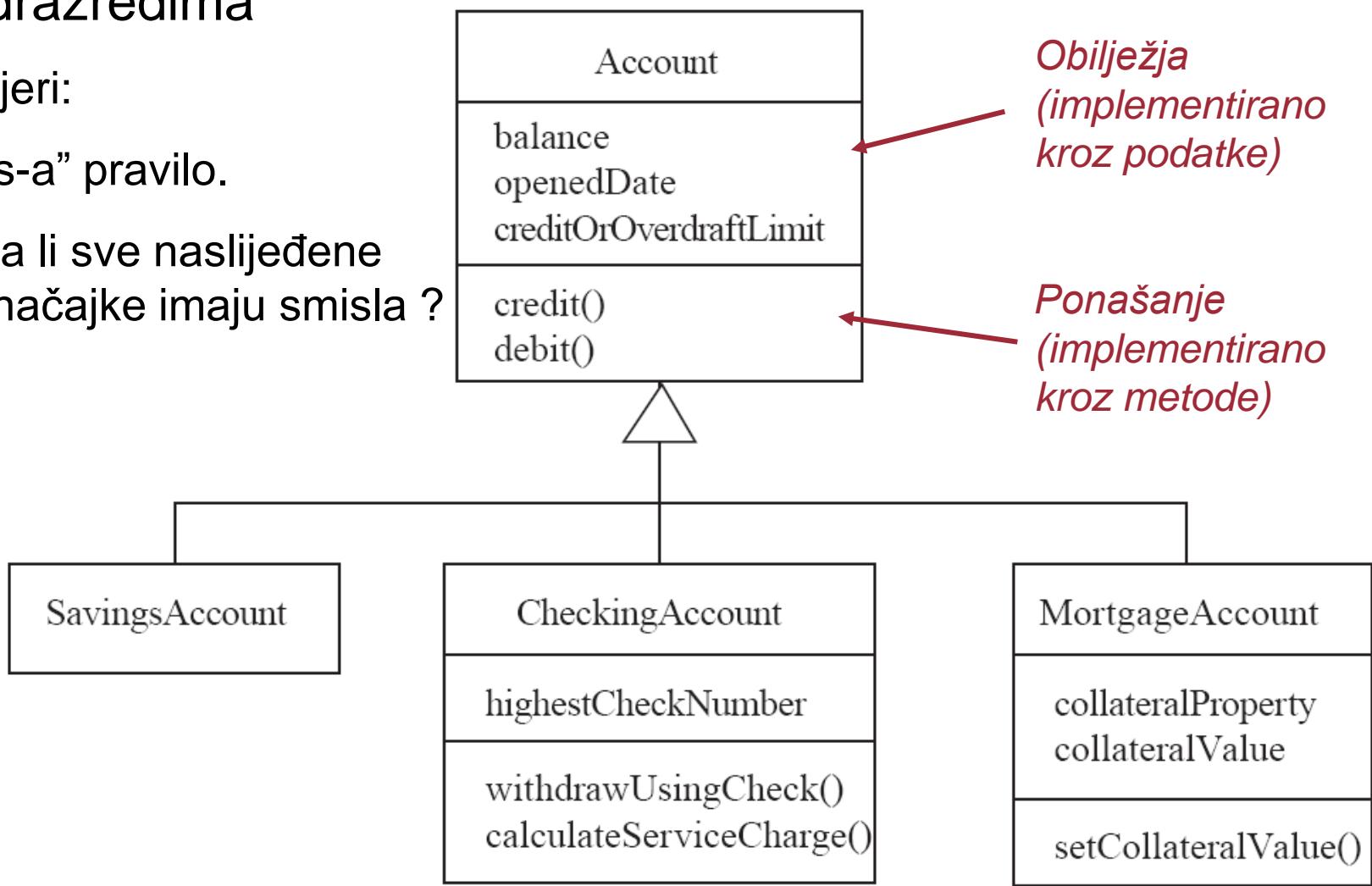


Primjer:

- Osiguraj da sve naslijedene značajke imaju smisla u podrazredima

Provjeri:

1. “is-a” pravilo.
2. Da li sve naslijedene značajke imaju smisla ?





Provjera ispravnosti nasljeđivanja



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



Svojstva nasljeđivanja

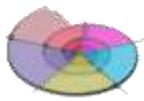


■ Prednosti

- mehanizam apstrakcije pogodan za organizaciju
- mehanizam ponovne uporabe u oblikovanju i implementaciji
- organizacija znanja o domeni i sustavu

■ Nedostaci

- razredi nisu samodostatni i ne mogu se razumjeti bez poznavanja nadrazreda
- nasljeđivanja uočena u fazi analize mogu dati neučinkovita rješenja
 - potrebno zasebno promatrati



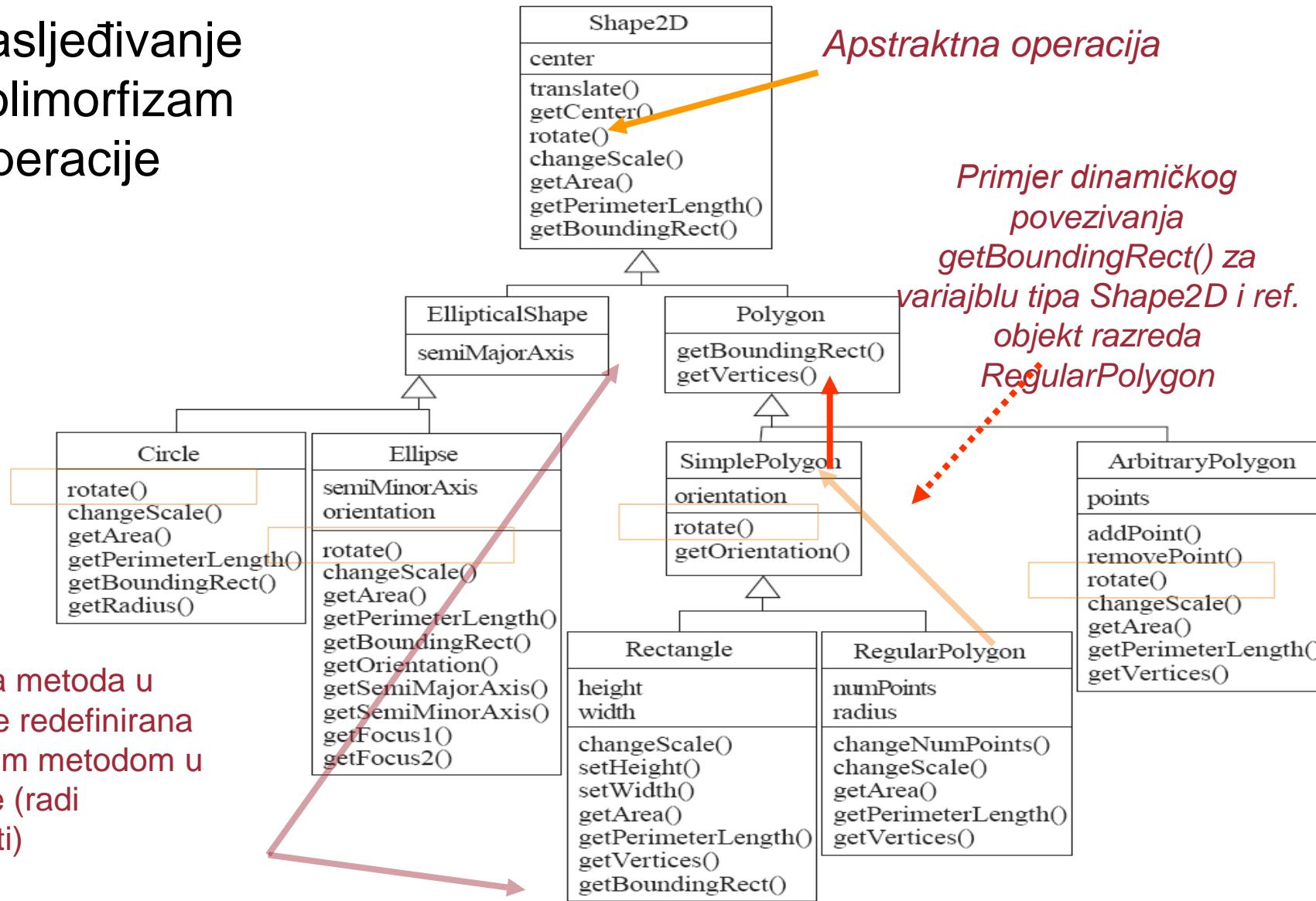
Apstraktni razredi i metode

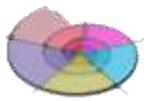


- Pojedina operacija treba biti *deklarirana* da postoji u najvišem hijerarhijskom razredu gdje ima smisla.
- Na toj razini operacija može biti *apstraktna* (bez implementacije)
 - npr. **rotate** u **Shape2D** (vidi sliku hijerarhije matematičkih objekata, gdje će podrazredi će imati svoje specifične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

Primjer ...

- Nasljeđivanje
- Polimorfizam
- Operacije





Višeobličje/Polimorfizam



- Moć poprimanja više oblika, *engl. polymorphism*
- Svojstvo objektno usmjerенog programa da se jedna apstraktna operacija može izvesti na različite načine u različitim razredima.
 - operacija pokazuje drugačije ponašanje ovisno o instanci
 - npr. `add(2,4)`, `add("dobro ", "jutro") ..`
- Polimorfizam zahtijeva da postoji više metoda istog naziva.
 - izbor koja metoda će se izvesti ovisi o razredu objekta koji se nalazi u varijabli.
 - polimorfizam smanjuje potrebu za kodiranjem velikog broja **if-else** ili **switch** naredbi



Primjer: Višeobličje

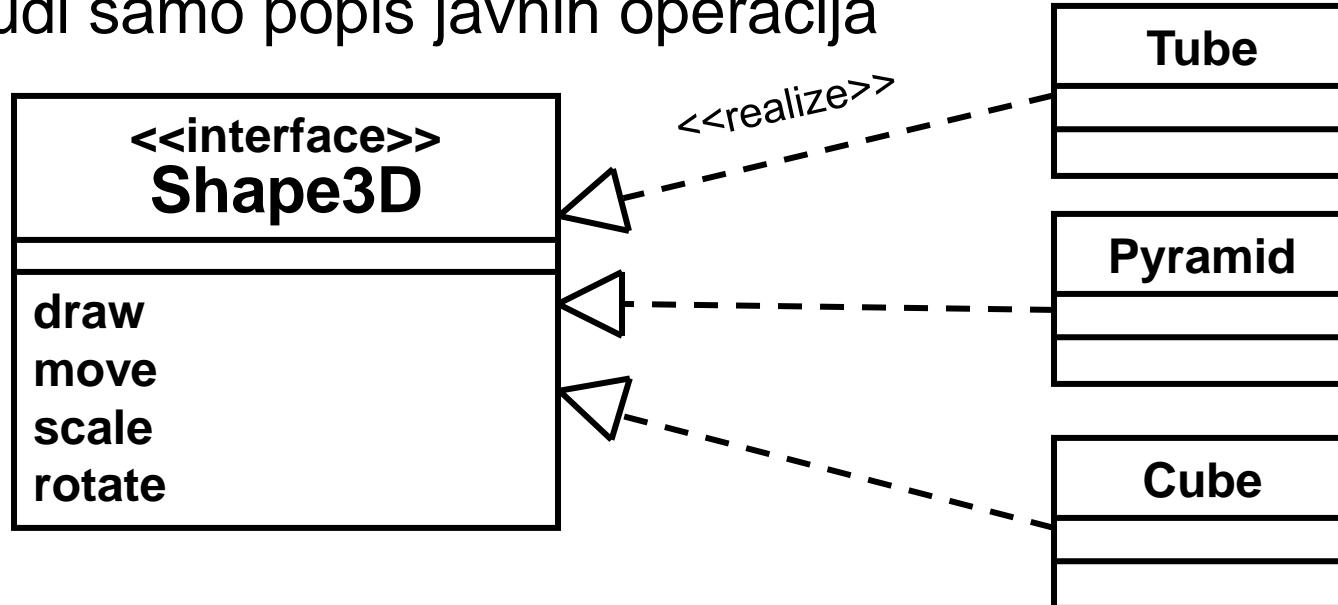


```
class Animal {  
    abstract string makeNoise (); } // apstraktna operacija  
  
class Cat extends Animal {           // podrazred Cat  
    string makeNoise() {return "Meow"; } } // konkretna metoda  
  
class Dog extends Animal {           // podrazred Dog  
    string makeNoise() {return "Bark"; } } // konkretna metoda  
  
main () {  
    Animal animal = zoo.getAnimal(); // animal je tipa Animal  
    Console.WriteLine (animal.makeNoise()); }
```

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

Sučelje

- *engl. Interface*
- formalizira polimorfizam
- podržava uključi i koristi koncept (engl. plug-and-play)
- spada u podskup apstraktnih razreda koji ne sadrže implementaciju metoda
 - nudi samo popis javnih operacija

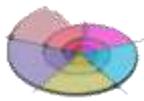




Nadjačavanje



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



Odabir metode za izvođenje



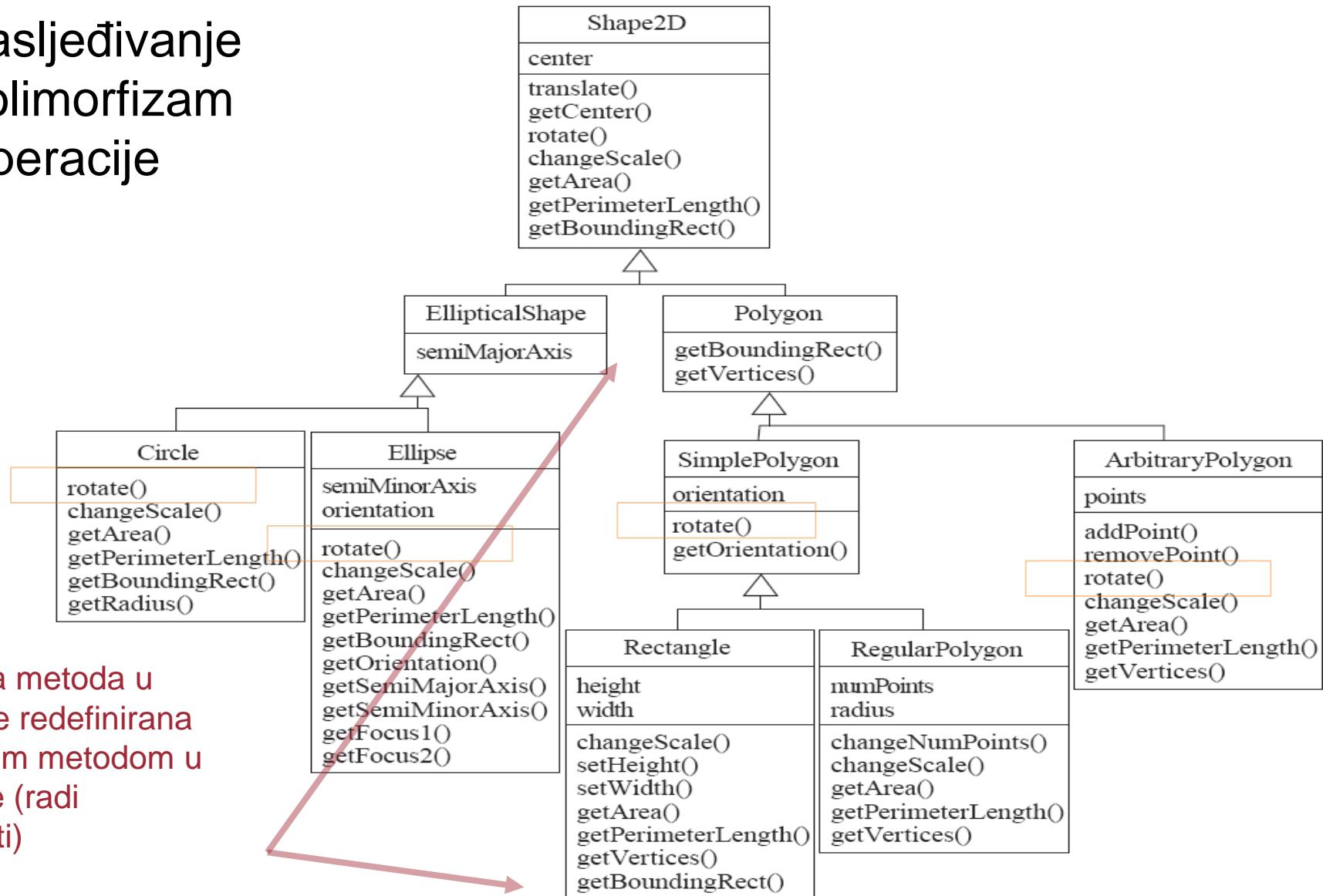
- Poziva se neka metoda iz objekta (varijable koja “sadrži” objekt **b** iz nekog razreda; tip varijable je taj razred).

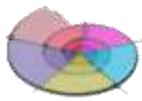
```
resultVariable = b.methodName(argument);
```

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

Primjer ...

- Nasljedivanje
- Polimorfizam
- Operacije





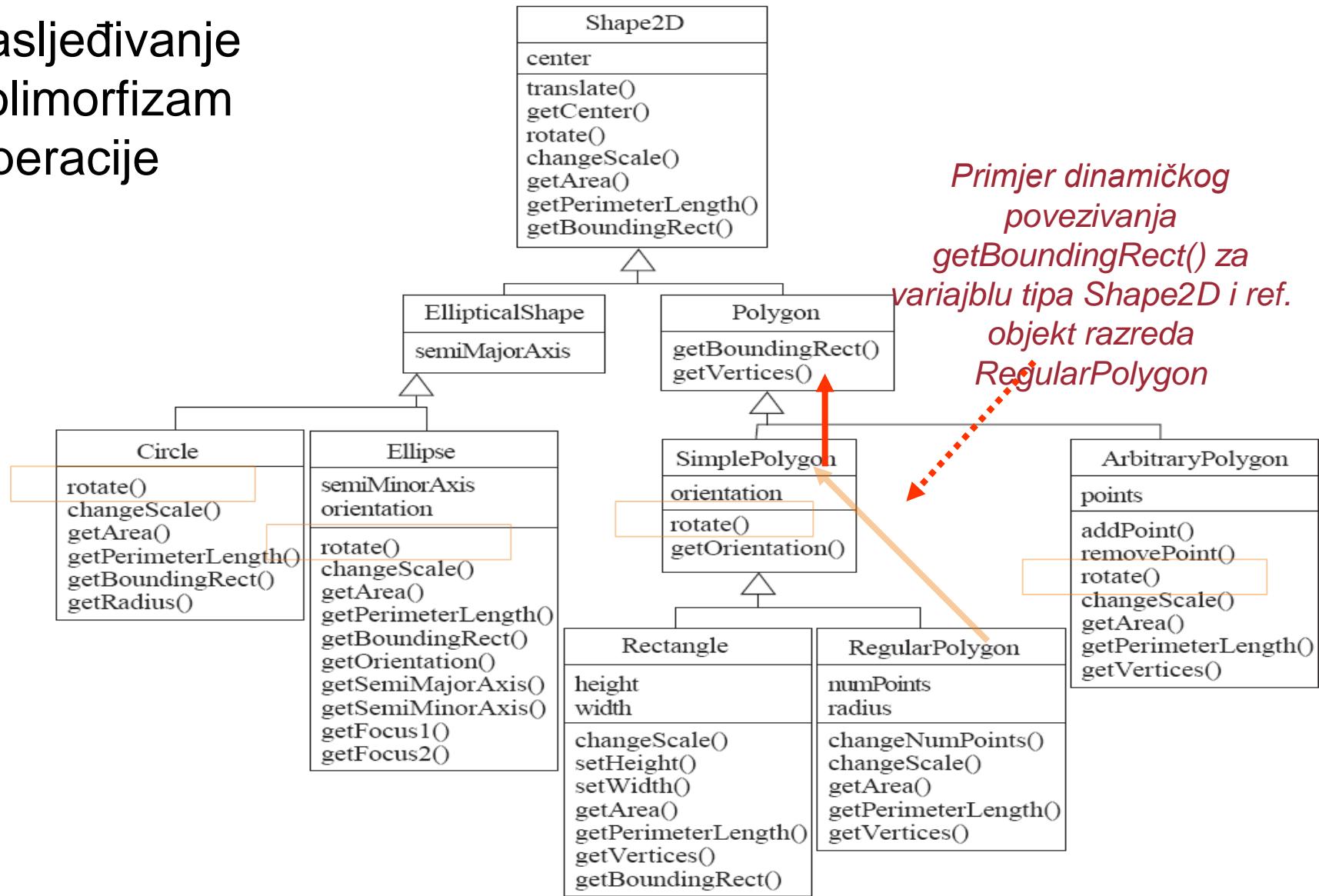
Dinamičko povezivanje

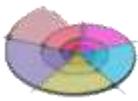


- engl. *Dynamic binding*
- Povezivanje se odnosi na ostvarivanje poveznice (engl. *linking*) poziva metode i koda koji se izvodi
- Dinamičko povezivanje (engl. *late binding*) pojavljuje se u slučaju kada se odluka o izboru konkretne metode donosi za vrijeme izvođenja programa (engl. *at run-time*) tj. nije poznato do trenutka izvođenja.
 - posljedica polimorfizma i nasljeđivanja
- To je potrebno kada:
 - varijabla je deklarirana da je tipa nadrazreda (tj. postoji hijerarhija podrazreda toga tipa variable).
 - postoji više polimorfnih metoda koje se mogu izvesti u sklopu hijerarhije razreda određene nadrazred tipom variable.
 - npr.:
 - neka postoji varijabla **aShape**, a njen tip je **Shape2D**.
 - to znači da **aShape** može sadržavati objekt iz bilo kojeg konkretnog razreda u hijerarhiji razreda **Shape2D**.
 - pretraživanje i odabir konkretne metode započinje pregledom razreda čiji objekt se stvarno nalazi u varijabli **aShape**.
 - ako metoda nije pronađena u tom razredu, pretražuje se sukcesivno hijerarhija razreda idući prema gore do nadrazreda **Shape2D** koji je naveden kao tip varijable.
 - ako metoda nije pronađena deklarira se pogreška.

Primjer ...

- Nasljedivanje
- Polimorfizam
- Operacije





Primjer dinamičkog povezivanja



■ Shape2D hijerarhija

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

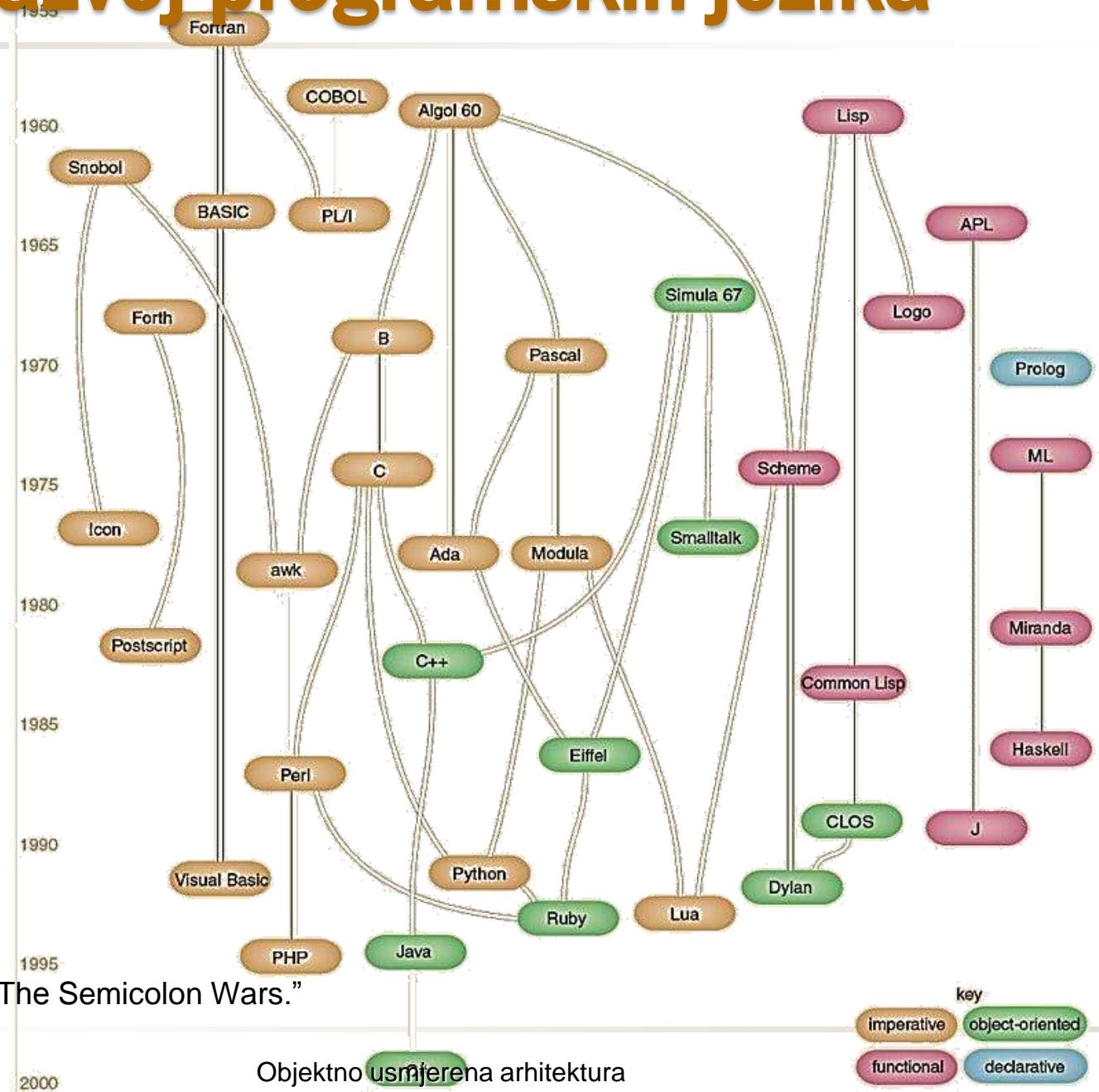


Dobra praksa objektnog usmjerjenja



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

Razvoj programskih jezika



- 2006, Brian Hayes, "The Semicolon Wars."



Objektno usmjereni programski jezici



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

Svojstva OO

Proceduralna

Dekompozicija problema u funkcije

Odvojeno modeliranje podataka i funkcija

Velika međuvisnost komponenti – teškoće održavanja

Komponente slabo odgovaraju stvarnom problemu – teško u slučaju rješavanja složenih problema

Često neprilagodljiv i linearan proces razvoja

OO

Dekompozicija u skupove objekata

Podaci i povezane operacije na jednom mjestu

Neovisnost komponenti

Blisko ljudskom rješavanju složenih problema

Pogodno za iterativan i inkrementalan razvoj



Sažetak



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

Diskusija

-
-
-
-
-

Oblikovanje programske potpore

ak.god. 2014./2015.

Modeliranje objektno usmjerene arhitekture UML-om



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



Tema

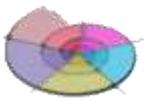
- Koraci objektno usmjerenog oblikovanja
- UML dijagrami razreda i objekata
- Stvaranje dijagrama razreda

Literatura

- Timothy C. Lethbridge, Robert Laganière, **Object-Oriented Software Engineering: Practical Software Development using UML and Java**, Second Edition, McGraw Hill, 2001
- O'Docherty, Mike: **Object-oriented analysis and design : understanding system development with UML 2.0** / Mike O'Docherty, John Wiley & Sons Ltd, 2005
- Grady Booch. Robert A. Maksimchuk. Michael W. Engle. Bobbi J. Young: **Object-Oriented Analysis and Design with Applications**. Third Edition, Addison-Wesley Professional, 2007
- Šribar, J.; Motik, B.: **Demistificirani C++**, Element, 2001 ("Dobro upoznajte protivnika da biste njime ovladali")

Koraci objektno usmjerenog oblikovanja

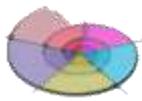
1. Identifikacija/pronalaženje razreda i osnovnih veza među njima
2. Poboljšavanje razreda specificiranjem operacija koje obavljaju
 - klasifikacija operacija: konstruktori, destruktori,
 - voditi računa o međuovisnostima, kompletnosti, ..
3. Poboljšavanje razreda specificiranjem ovisnosti o drugim razredima
 - moguća nasljeđivanja
 - sastavljanje
 - uporaba
4. Specifikacija sučelja:
 - podjela na privatne i javne operacije
 - specifikacija tipa operacija
 - izdvajanje javnih operacija u sučelja



Utvrđivanje razreda



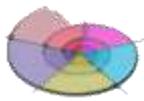
- Dobro oblikovanje obuhvaća i modelira stvarnost
- Uobičajeno:
 - imenice odgovaraju razredima
 - glagoli predstavljaju funkcije
 - imeničke fraze – atributi
 - glagolske fraze - veze
- Poželjne konzultacije s ekspertima domene
- Razred obuhvaća:
 - stanja objekata – sve informacije koje se čuvaju u objektu
 - ponašanje – skup operacija koje se mogu obavljati s objektom
 - uobičajeno mijenja stanje objekta.



Utvrđivanje operacija



- Razmatranja kako se objekti razreda stvaraju, umnažaju, brišu
- Neophodno definirati minimalan skup operacija koje zahtijeva razmatrani razred
- Razmatranje dodatnih operacija u svrhu zadovoljavanja konvencija i rada s objektima
- Razmatranje dodavanja apstraktnih metoda
- Klasifikacija operacija
 - osnovne: konstruktori i destruktori (engl. *constructors , destructors*)
 - pomoćne (engl. *Assessors, Selectors, Getters*)
 - ne mijenjaju stanje objekta
 - modificirajuće (engl. *Modifiers, Setters*)
 - utječu na promjenu stanja.
 - pretvorba (engl. *Conversion*)
 - stvaraju objekt drugog tipa na osnovu trenutnog stanja
 - ponavljajući (engl. *Iterators*)
 - obrađuju skupine objekata



UML dijagrama razreda



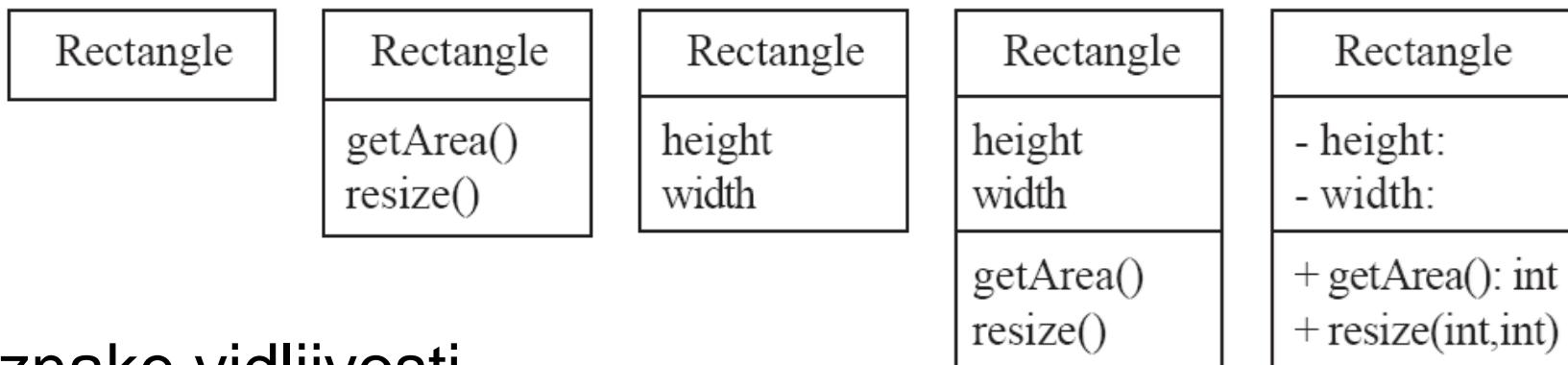
- engl. *class diagram*
- Osnovni simboli prikazani na dijogramima razreda su:
 - razredi
 - Predstavljaju tip podataka (podatkovne apstrakcije koje sadrže procedure).
 - pridruživanja (engl. *Associations*)
 - predstavljaju vezu između instanci razreda.
 - atributi
 - jednostavni podaci sadržani u razredima i njihoviminstancama.
 - operacije
 - predstavljaju funkcije koje izvode razredi i njihove instance.
 - generalizacije
 - grupiranje razreda u hijerarhiju nasljeđivanja.

Razred

- U UML-u razred je predstavljen naziv razreda pravokutnikom
 - Grafički simbol razreda može vidljivost prikazati atribute i operacije.
 - naziv razreda:
[packet_name]:: name
 - atributi
[visibility] name [[multiplicity]] [: type] [=initial value] [{property}]
 - operacije
[visibility] name [(parameter-list)] [: return-type] [{property}]
parameter-list:==[direction] name : type [= default-value]
direction ::= in| out| inout
-
- The diagram illustrates the structure of a UML class. It consists of several horizontal compartments separated by thin lines. At the top is a red compartment containing the stereotype '<< stereotype >>' and the class name 'Ime razreda'. Below this is a blue compartment listing attributes: '- x : int' and '- y : int'. The next compartment is light blue and lists operations: '+ getX() : int', '+ setX(rX : int) : void', '+ getY() : int', and '+ setY(rY : int) : void'. The bottom compartment is yellow and labeled 'odgovornosti' (responsibilities). Four green arrows point from text labels to specific parts of the class structure: one arrow points from 'param. predloška' to the top red compartment; another from 'atributi' to the blue compartment; a third from 'operacije' to the light blue compartment; and a fourth from 'pravo pristupa' to the yellow compartment.

Primjer: Razredi

- U različitim fazama – različita razina detalja
 - ovisno o pogledu!!!



- Oznake vidljivosti

`visibility ::= '+' | '#' | '-'| '˜' | '/' | '_'`

+ public – dostupno svima

protected - dostupni od podrazreda

- private - dostupni unutar razreda

~ package – dostupno unutar istog paketa

_ static – samo jedna vrijednost za sve instance

/ derived – izvedena vidljivost atributa tijekom izvođenja (može se kombinirati)

Tipovi pridruživanja

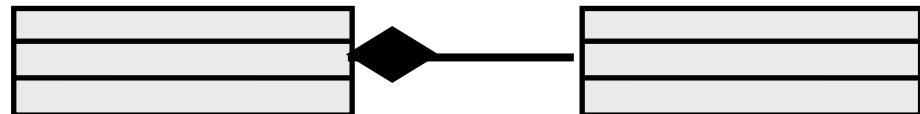
■ Povezivanje razreda

- pridruživanje
- kompozicija
- agregacija
- generalizacija
- ovisnost

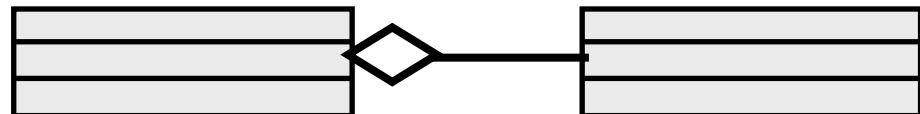
association



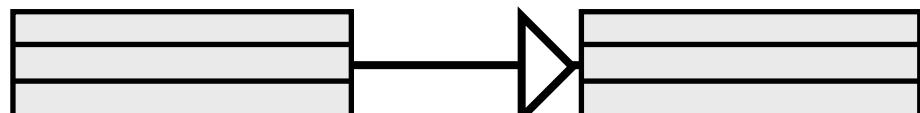
composition



aggregation



generalization



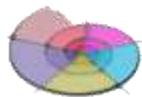
dependency



Pridruživanje i brojnost (višestrukost)

- Pridruživanje ili asocijacija pokazuje postojanje odnosa između dva razreda.
 - postoji veza između instanciranih objekata
- Brojnost pokazuje broj instanci razreda.
- Simboli koji pokazuju brojnost smješteni su na svakom kraju pridruživanja.

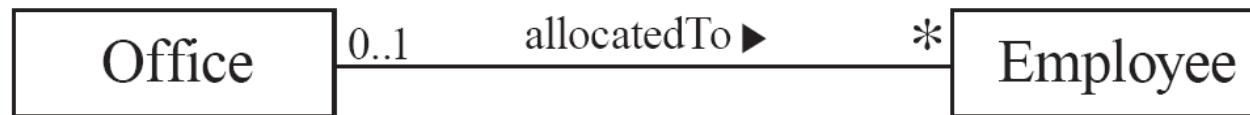




Označavanje pridruživanja



- engl. *role name*
- Svako pridruživanje može se imenovati nazivom uloge
 - poboljšano razumijevanje



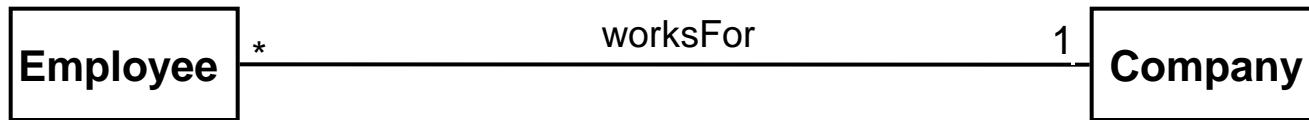


Analiza i validacija pridruživanja



■ Više/mnogo - jedan

- jedna kompanija ima mnogo zaposlenika.
- jedan zaposlenik može raditi samo za jednu kompaniju.
 - Ta kompanija nema podataka o drugom zaposlenju
- kompanija ima nula zaposlenika.
 - Npr. *početna registracija, ili krovna kompanija.*
- nije moguće biti zaposlenik ako ne radiš za neku kompaniju.





Analiza i validacija pridruživanja



■ Više - više

- jedan asistent može raditi za mnogo menadžera.
- jedan menadžer može imati mnogo asistenata.
- asistenti mogu biti organizirani u skupove (engl. *pool*).
- menadžeri mogu imati grupu asistenata.
- neki menadžeri ne moraju imati asistenata.



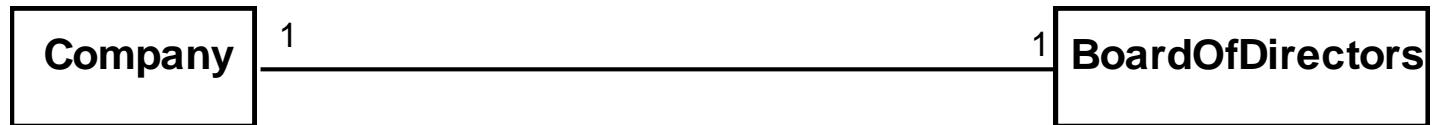


Analiza i validacija pridruživanja



■ Jedan - jedan

- u svakoj kompaniji postoji točno jedan odbor direktora.
- odbor direktora je odbor samo jednoj kompaniji.
- kompanija mora uvijek imati odbor direktora.
- odbor direktora je uvijek odbor u nekoj kompaniji.



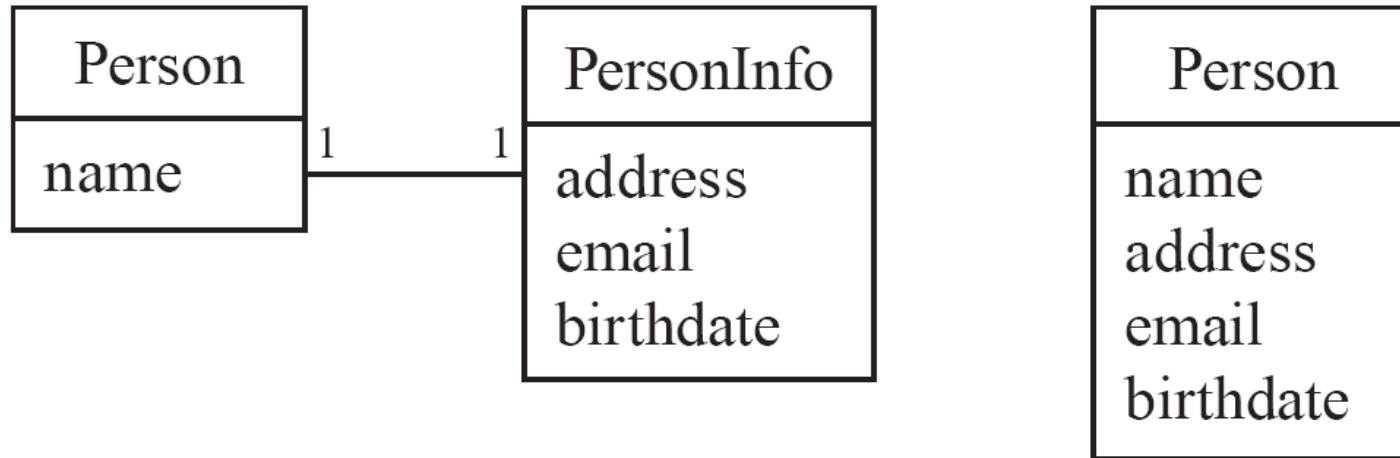


Analiza i validacija pridruživanja



- Izbjegavaj nepotrebna pridruživanja jedan – jedan.

izbjegavati

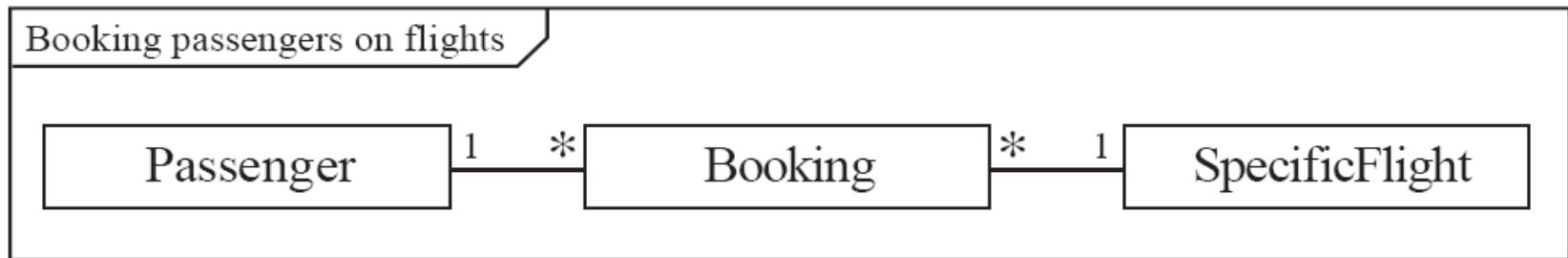




Primjer pridruživanja: rezervacija leta

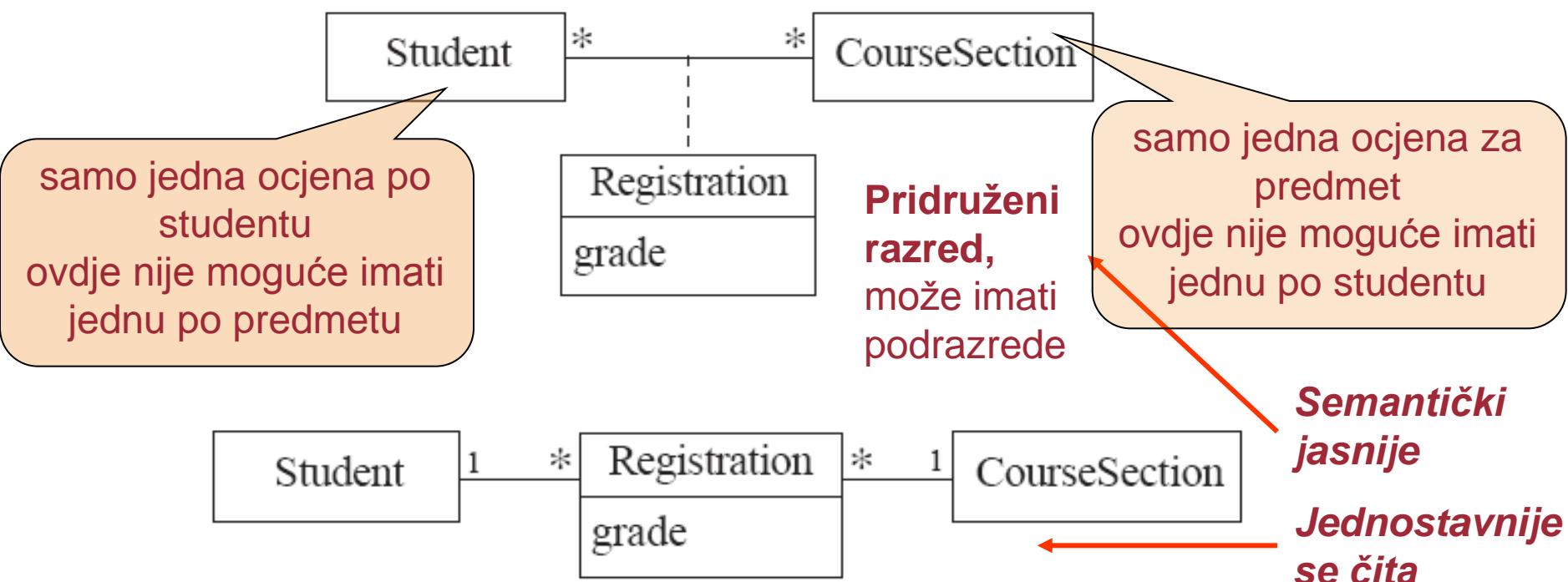


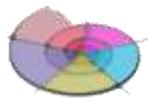
- **Rezervacija je uvijek samo za jednog putnika.**
 - nema rezervacije za nula putnika.
 - rezervacija *nikada* ne uključuje više od jednog putnika.
- **Putnik može imati bilo koji broj rezervacija.**
 - putnik uopće ne mora imati neku rezervaciju.
 - putnik može imati više od jednu rezervaciju.



Pridruženi razredi

- Ponekad se atribut koji se tiče više razreda ne može smjestiti niti u jedan od navedenih razreda.
- Postoje dva ekvivalentna označavanja **pridruženih razreda**:

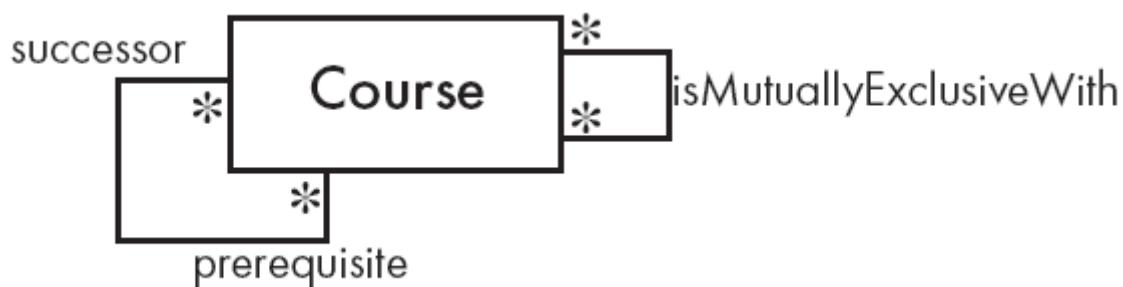


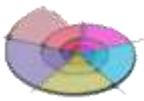


Refleksivno pridruživanje



- engl. *Reflexive associations*
- Moguće je da se pridruživanje spaja na isti razred.
- Primjer:
 - predmet može imati druge predmete kao preuvjete
 - spriječiti upis vrlo sličnih predmeta

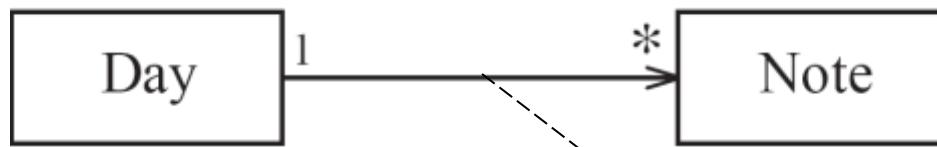




Smjer pridruživanja



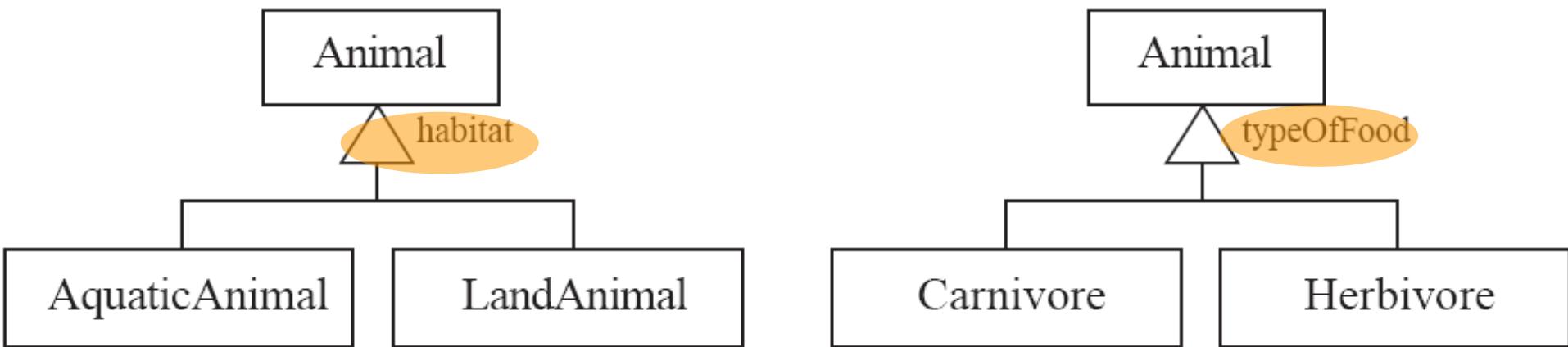
- Pridruživanja su u osnovici dvosmjerna (bidirekcijska).
- Moguće je ograničiti smjer pridruživanja dodavanjem strelice na jednom kraju.
 - ako nije naznačen veza je dvosmjerna ili nije specificirana



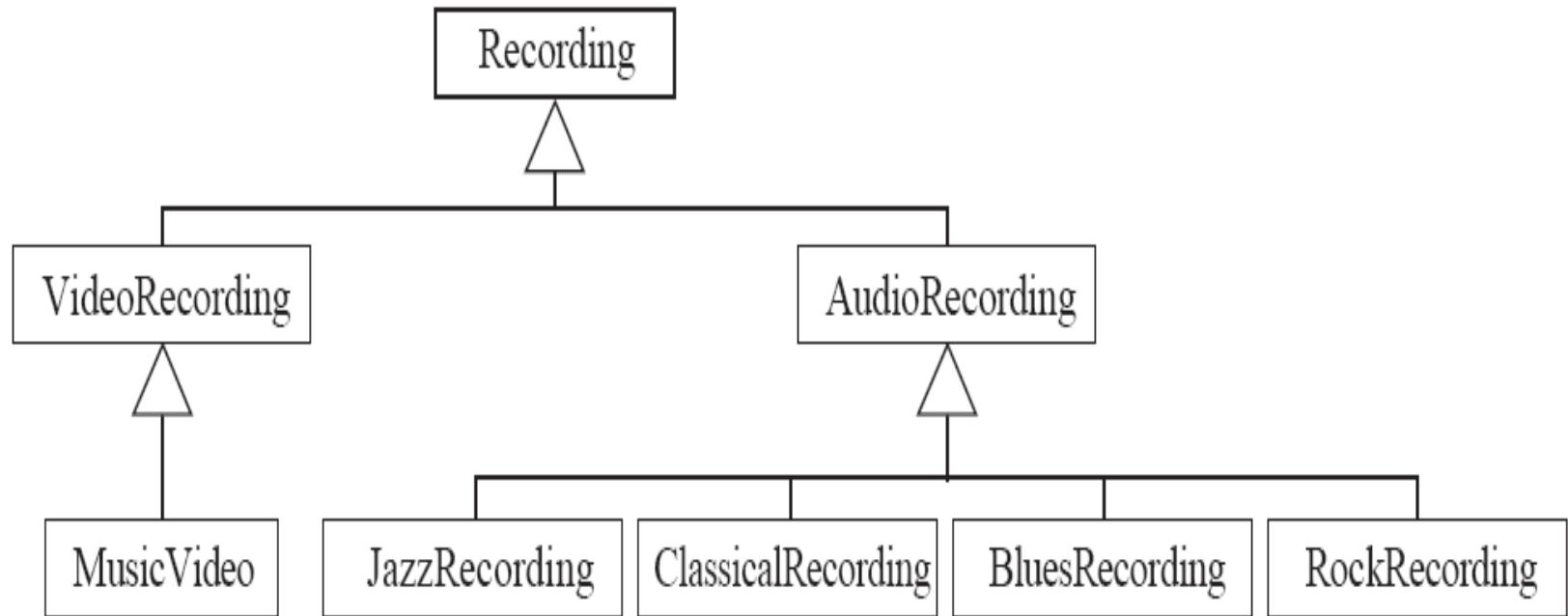
za odabrani dan pogledaj
bilješku, ali ne i obratno

Generalizacija

- engl. *generalization/specialization*
- Specijalizacija nadrazreda u jedan ili više podrazreda.
 - Generalizacijski skup (engl. *generalization set*)
 - označena grupa generalizacija sa zajedničkim nadrazredom.
 - Oznaka (katkad nazvana diskriminator)
 - opisuje kriterije za specijalizaciju.

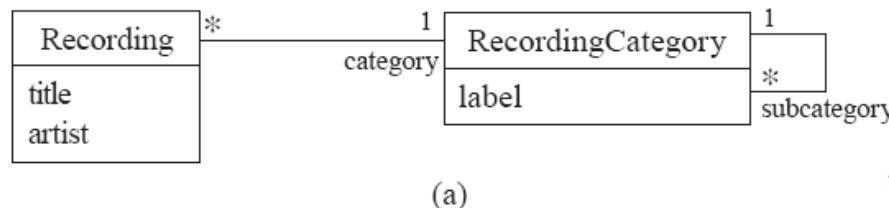


Izbjegavanje nepotrebne generalizacije

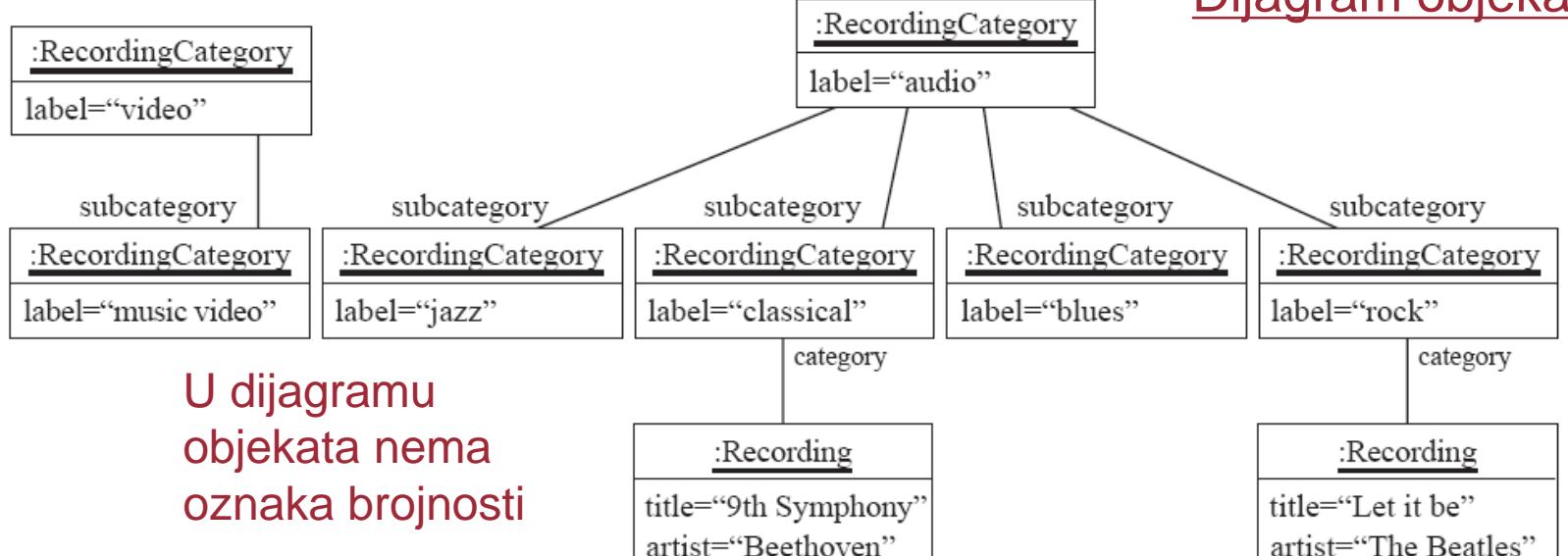


Nepogodna hijerarhija razreda,
Trebali bi biti instance.

Izbjegavanje nepotrebne generalizacije



Dijagram razreda

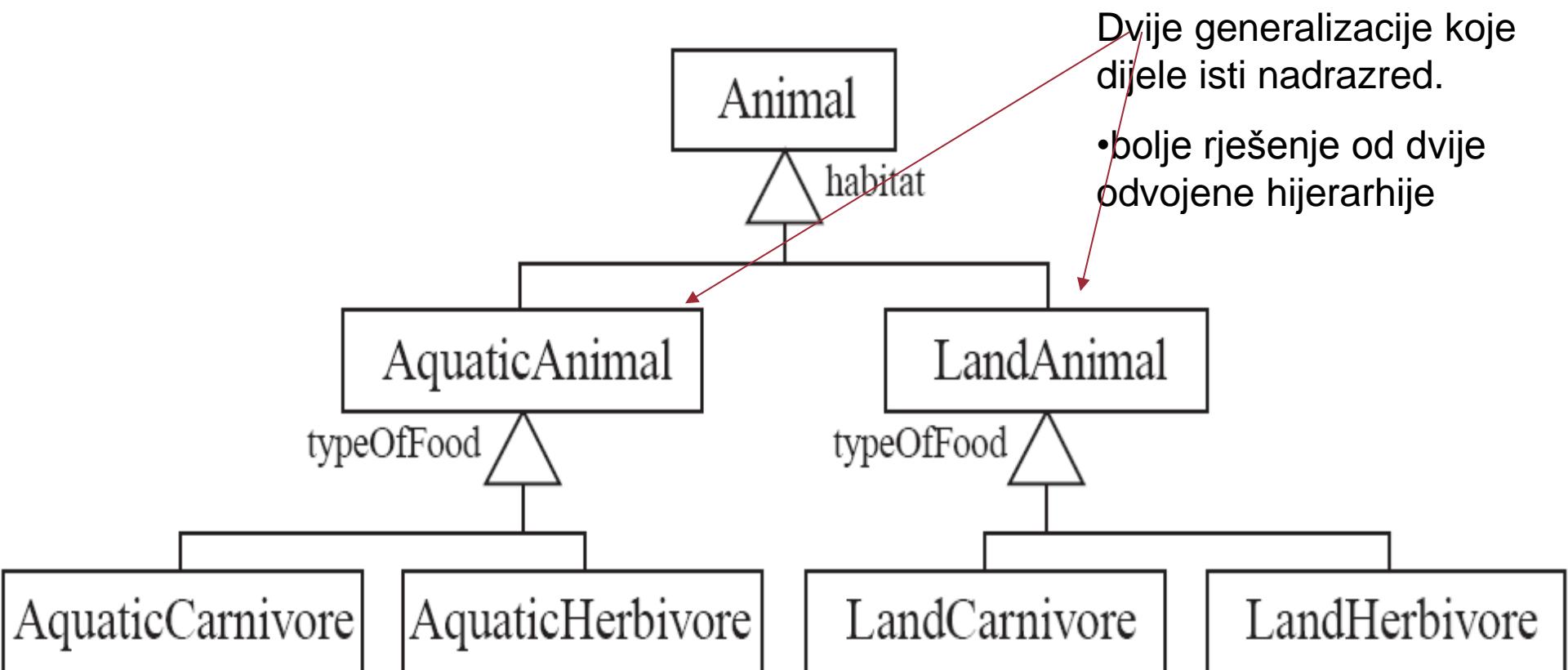


Dijagram objekata

Slika pokazuje poboljšani **dijagram razreda** uz pridruženi **dijagram objekata** (instanci). Razredi iz prethodnog dijagrama su instance jednog razreda *RecordingCategory*

Rukovanje višestrukim diskriminatorima

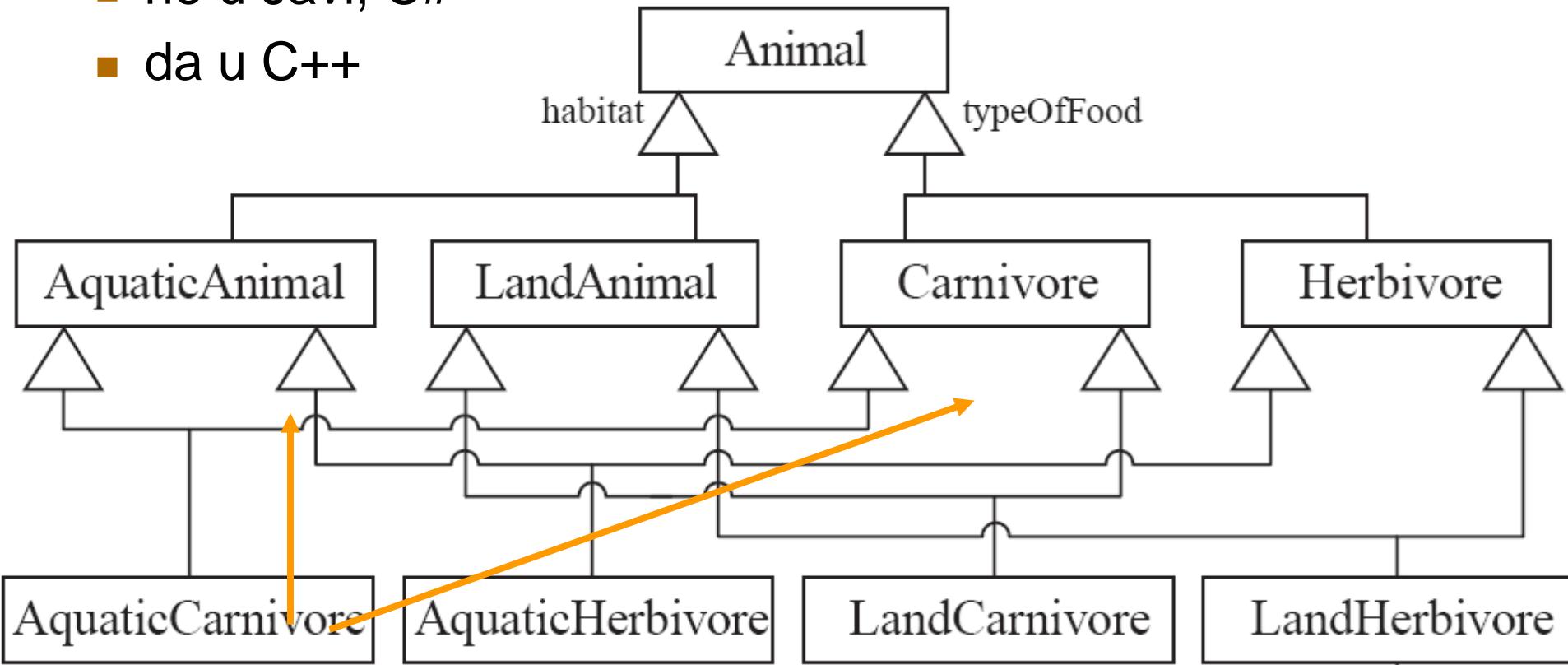
■ Kreiranje generalizacije više razine.



Rukovanje višestrukim diskriminatorima

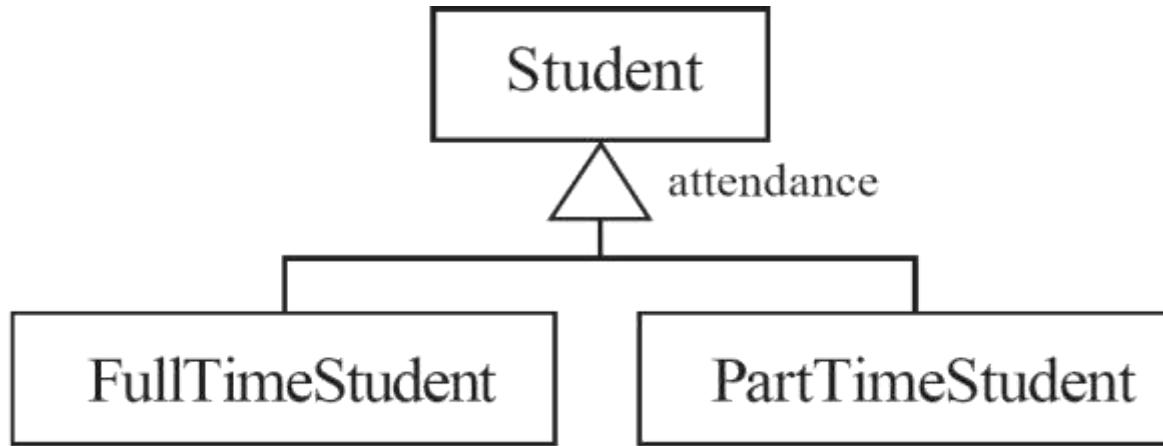
■ Višestruko nasljeđivanje

- ne u Javi, C#
- da u C++

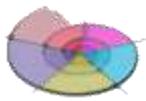


Izbjegavanje promijene razreda Instanci

- Instanca nikad ne smije imati potrebu promjene razreda.



- Studentov status se može promijeniti \Rightarrow taj objekt (instanca) mora promijeniti razred
 - traži destrukciju – kreaciju novog objekta
- Jedno rješenje: uključi atribut *attendanceStatus* u razred *Student* te ne koristiti podrazrede
 - tako se gubi mogućnost polimorfizma - specifičnih metoda u različitim podrazredima



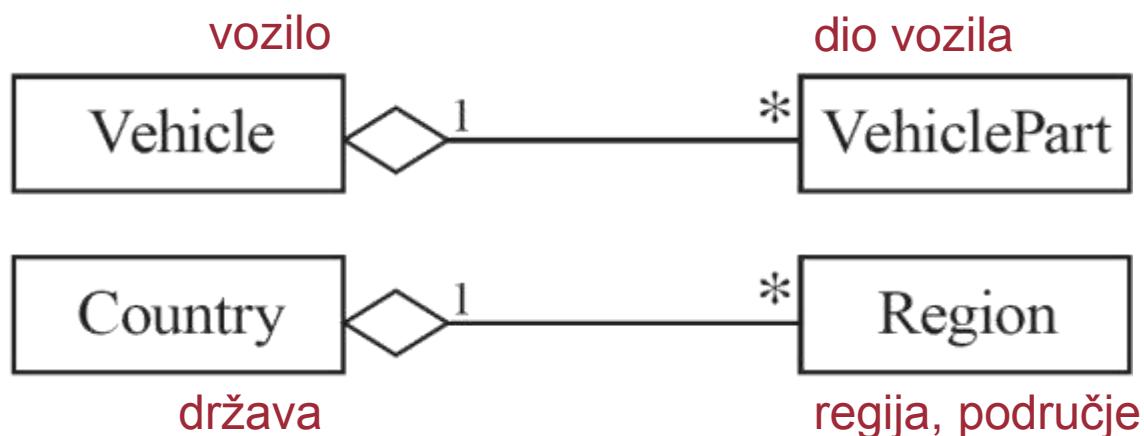
O pridruživanju i generalizaciji

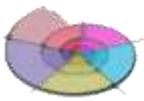


- **Pridruživanje** (asocijacija) opisuje odnos između *instanci* koji će nastupiti za vrijeme izvođenja programa.
 - kada se dijagram objekata generira iz dijagrama razreda, u dijagramu objekata postojat će instance oba razreda spojene pridruživanjem.
- **Generalizacija** opisuje odnos između *razreda* u dijagramu razreda.
 - generalizacija se ne pojavljuje u dijagramu objekata.
 - jedna instanca nekog razreda ujedno je i instanca svakog njegovog nadrazreda.

Agregacija/Sadržavanje

- engl. *Aggregation, is part of*
- Predstavljaju specifičnu vrstu pridruživanja koja predstavlja odnos “**cjelina-dio**.
 - “Cijeli dio se često naziva **agregat** (zbor, skup).
 - UML simbol označuje pridruživanje “ **je dio od** (engl. *isPartOf*).





Uporaba agregacije



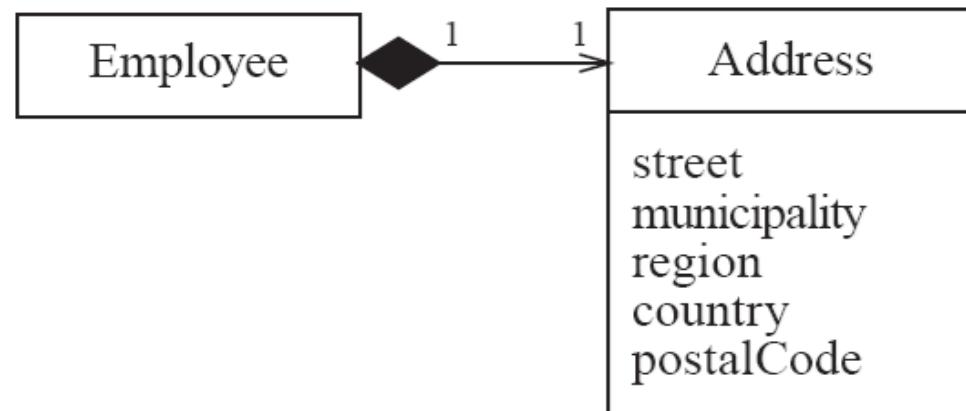
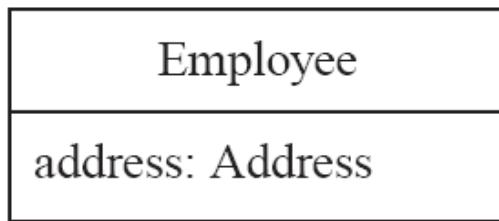
- Kao opće pravilo, agregacija se u pridruživanju koristi kada:
 - može se tvrditi:
 - Dijelovi su dio agregata.
 - ili agregat je sastavljen od dijelova.
 - kada je netko ili nešto vlasnik agregata (ili upravlja njime) tada je ujedno i vlasnik (upravlja) njegovim dijelovima.

Kompozicija

- Kompozicija (*engl. Composition*) je jaki tip agregacije.
 - Ako se agregat uništi, tada se uništavaju i njegovi dijelovi.



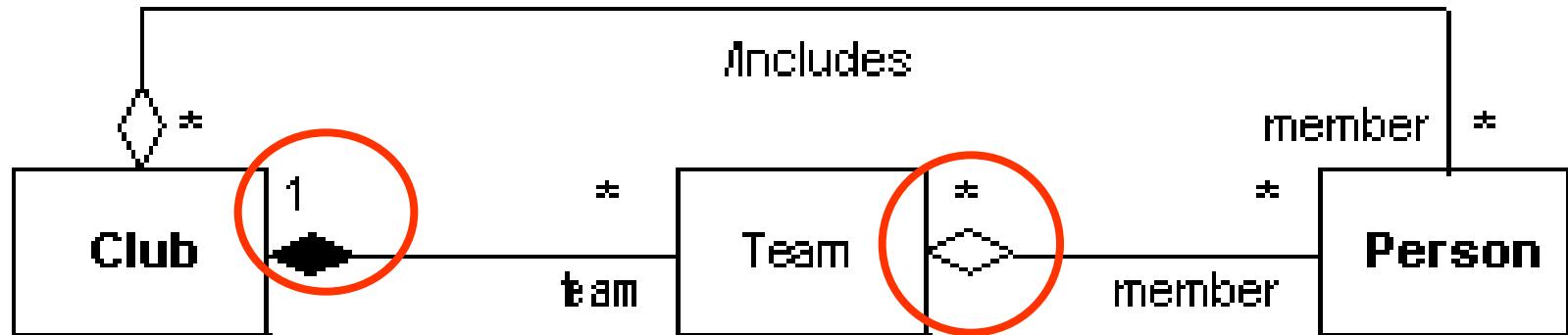
- Alternativni načini modeliranja adresa:



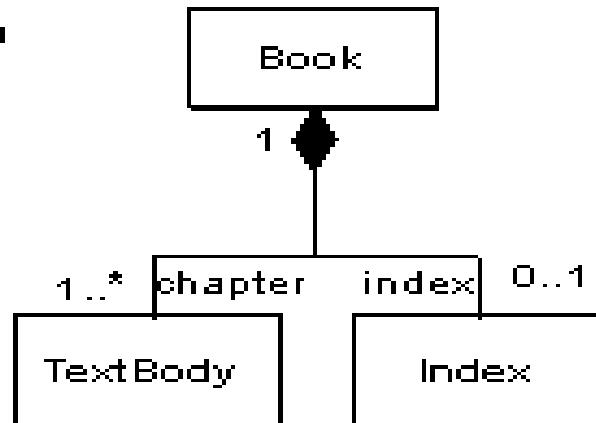


Primjeri agregacije i kompozicije

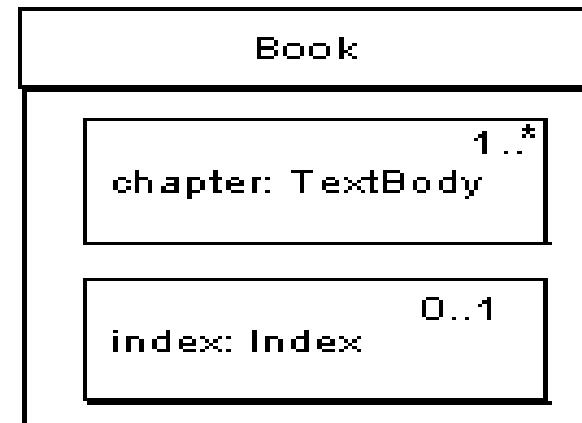
{ Club.member = Club.team.member }



(a)

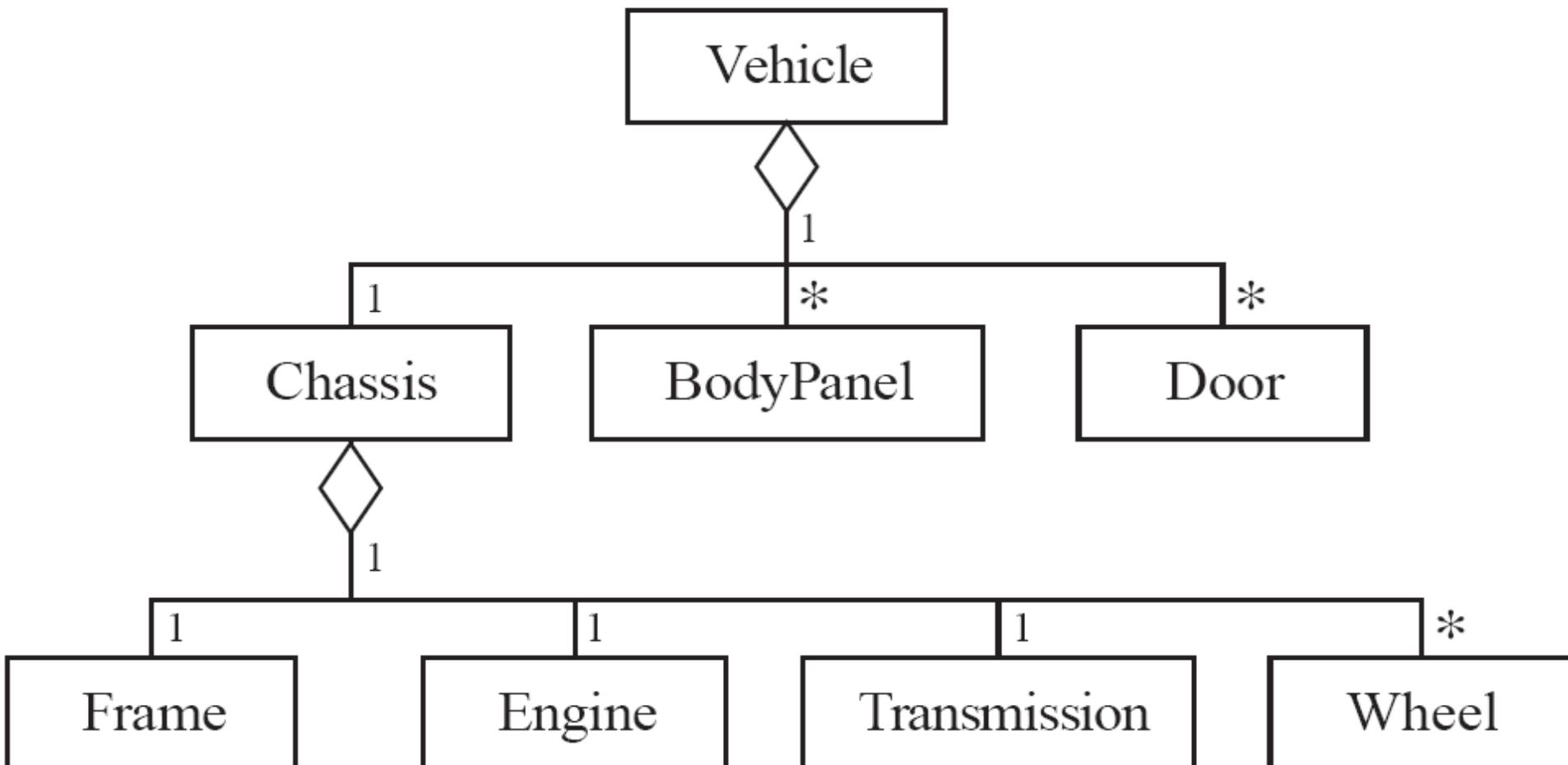


(b)





Agregacijska hijerarhija





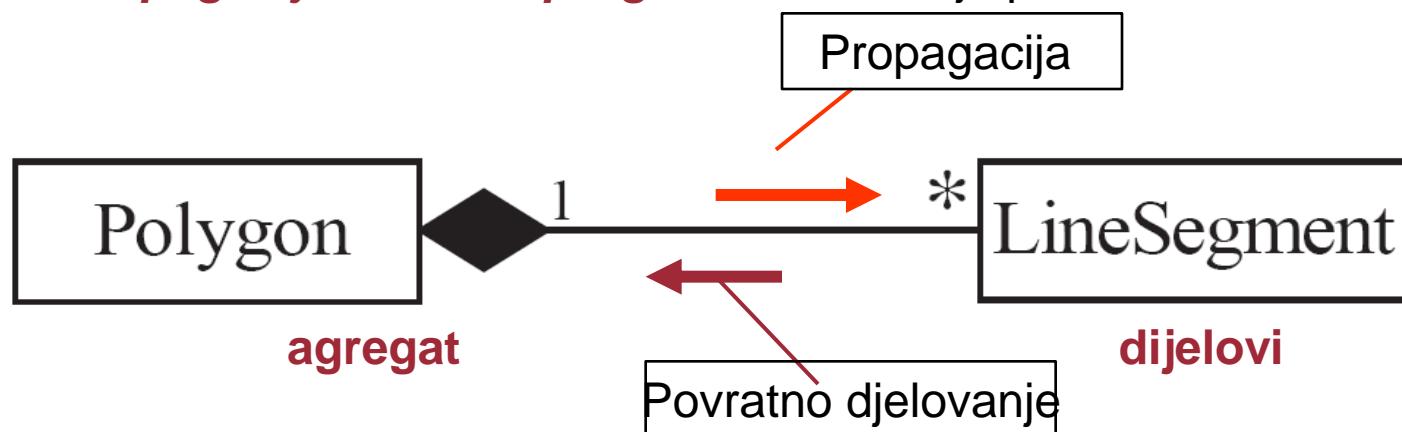
Propagacija u agregaciji



- Propagacija je mehanizam kojim se implementira operacija u agregatu tako da djeluje na njegove dijelove.
- U isto vrijeme, značajke dijelova propagiraju natrag prema agregatu.
- Propagacija u agregaciji predstavlja isto što i nasljeđivanje u generalizaciji.

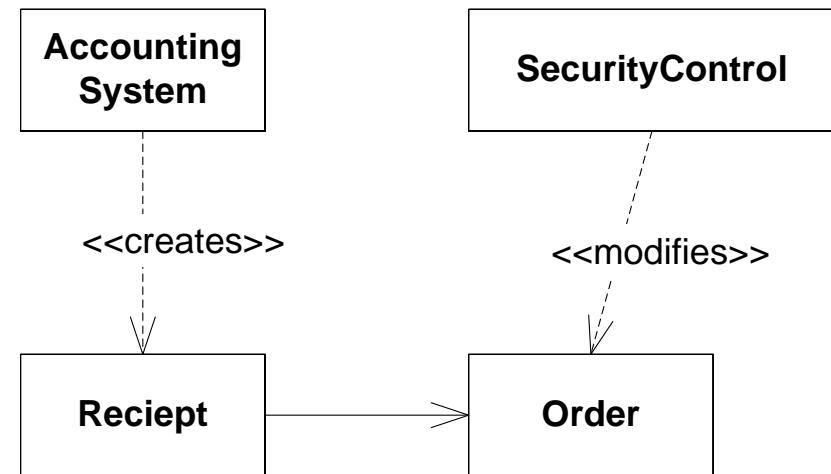
- temeljna razlika:

- Nasljeđivanje je *implicitan* mehanizam.
- *Propagacija se mora programirati* kada je potrebna.



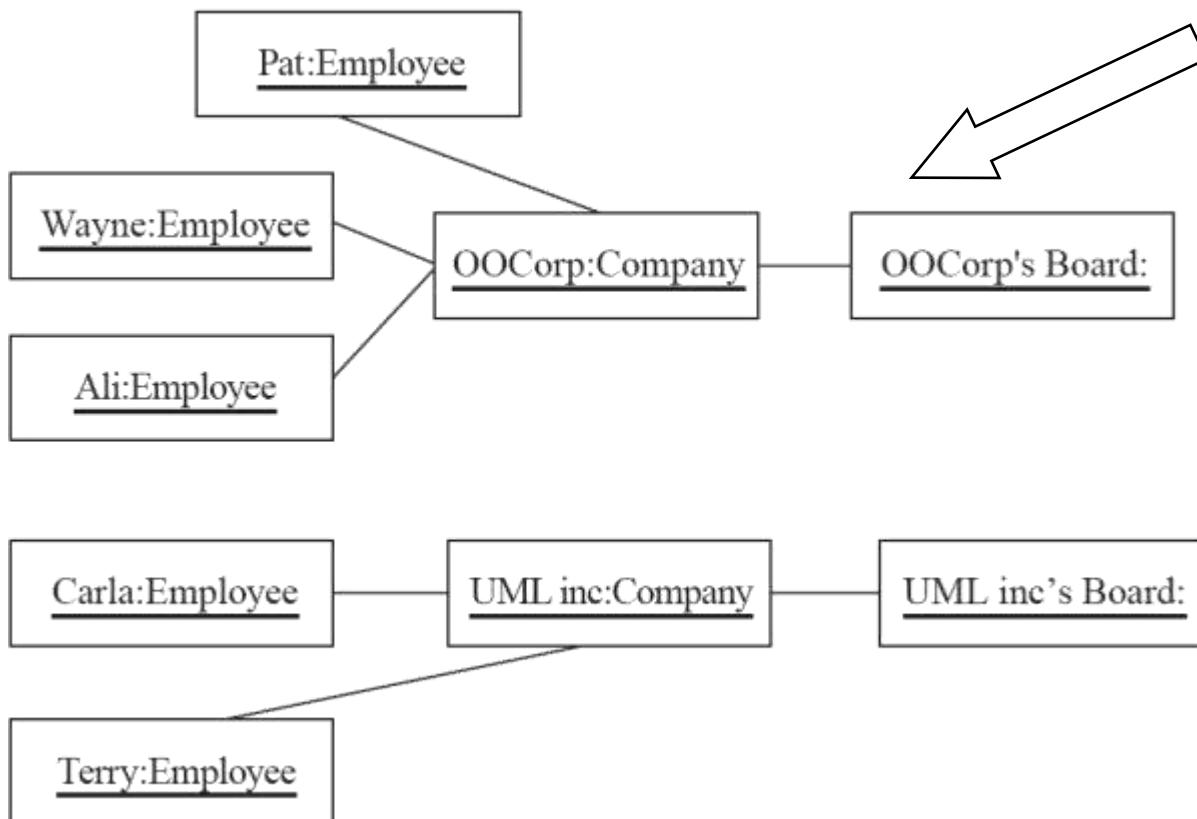
Ovisnost

- engl. *Dependency* ----->
- Najopćenitiji odnos razreda
 - označava utjecaj jednog objekta na drugi
- Uvijek usmjereni, bez brojnosti
- Tipovi:
 - «call»
 - «create»
 - «modifies»



Dijagram objekata

- engl. *Object Diagrams*
- Prikazuje instance i veze u **jednom trenutku izvođenja sustava**.
- **Veza** (engl. *link*) je instanca pridruživanja.
 - Na isti način kako je objekt instanca razreda.



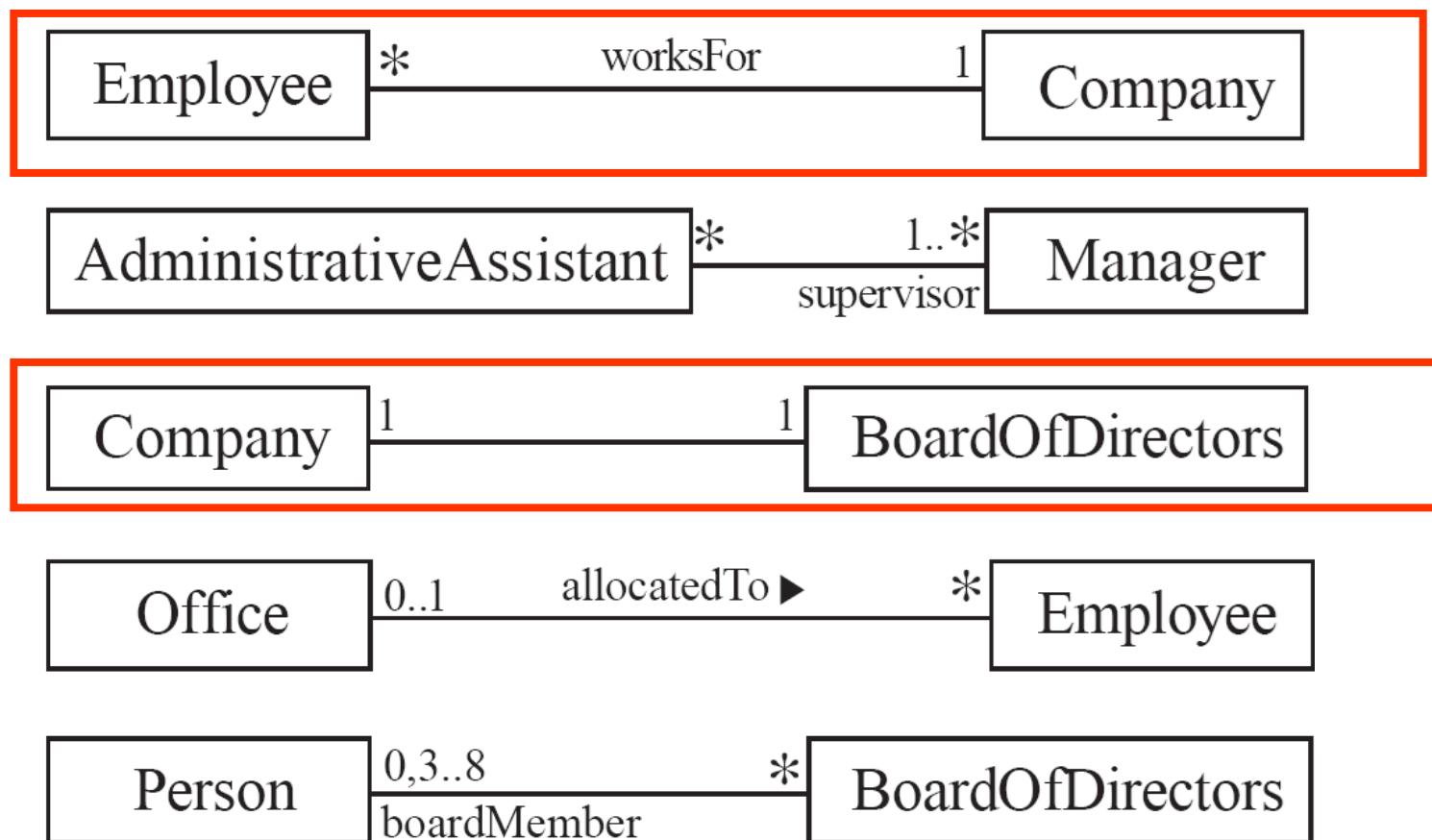
—**nema oznake brojnosti** (radi se o individualnim objektima).



Generiranje dijagrama objekata

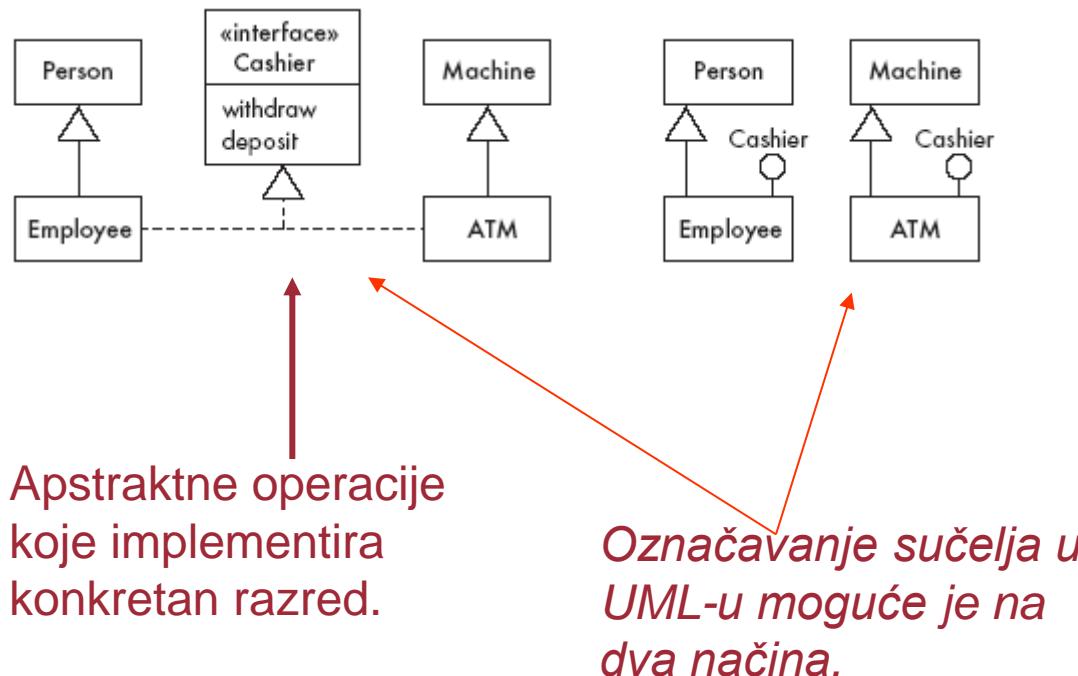


- Dijagrami objekata generiraju se iz dijagrama razreda
- Primjer: Dijagram razreda upotrijebljen za stvaranje prethodnog dijagrama objekata:



Sučelja

- engl. *Interfaces*
- Sučelje opisuje *dio vidljivog ponašanja skupa objekata*.
 - Jedno sučelje slično je razredu, osim što mu nedostaju varijable instanci i implementacija metoda. *To je lista apstraktnih operacija. Sučelje može biti tip varijable, te u tom slučaju u varijablu se može staviti bilo koja instanca nekog konkretnog razreda koji ostvaruje* to sučelje. Moguće je i dinamičko povezivanje.



Primjer sučelja

- Vrijednost *sučelja* je u specifikaciji operacija koje polimorfno implementiraju različiti razredi. Ti razredi (koji implementiraju operacije) ne moraju biti ni u kakvom međusobnom odnosu.
- Neki razred može realizirati više sučelja.
- *Sučelje* može biti tip neke varijable. Uporabom te varijable može se izvesti bilo koja operacija koja je podržana sučeljem. Pozivanje ispravne metode osigurano je dinamičkim povezivanjem.
- Primjer sučelja *Ownable* :

```
public interface Ownable
{
    public abstract String getOwner();
    public abstract void setOwner(String name);
}
```

- Implementacija sučelja u razredu:

```
public class BankAccount implements Ownable
{
...
    public String getOwner()
    {
        return accountHolder;
    }
    public void setOwner(String name)
    {
        accountHolder = name;
    }
...
}
```

Bilješke i opisni tekst

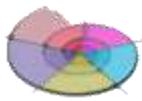
- Opisni tekst i drugi dijagrami
 - uključi dijagrame u veći dokument (vidi projekt u okviru predmeta Oblikovanje programske potpore).
 - tekst može objasniti bilo koji aspekt sustava uporabom sasvim slobodne (nestandardizirane) notacije.
 - ističe i proširuje opis važnih značajki sustava, te navodi argumente u donošenju odluka oblikovanja.
- Bilješke (*engl. Notes*):
 - bilješka je mali blok teksta uključen u UML dijagram.
 - ima istu ulogu kao i komentar u programskom jeziku.



Proces razvoja dijagrama razreda



- UML modeli mogu se kreirati u različitim fazama oblikovanja programske potpore. s različitim ciljem (svrhom) i s različitom razinom detalja.
 - **istraživački (engl. *exploratory*) model domene primjene:**
 - malo detalja
 - razvija se tijekom analize domene s ciljem razumijevanja te domene.
 - **sistemska model domene:**
 - više detalja
 - modelira aspekt domene koji će biti predstavljen programskim sustavom.
 - **model sustava:**
 - najviše detalja
 - uključuje razrede objekata potrebnih u izgradnji arhitekture sustava i korisničkog sučelja.



Modeli i razine detalja



	Elementi koji predstavljaju domenu	Elementi domene koji će stvarno biti implementirani	Elementi (ne domene) potrebni za izgradnju sustava
Istraživački model	Da	-	-
Sistemski model domene	Da	Da	-
Model sustava	Da	Da	Da



Tehnike identifikacije OO elemenata



- Klasični pristup – engl. *classical object-oriented analysis*
 - temeljem kategorizacije “opipljivih stvari npr. Coad and Yourdon, Ross..”
- Analiza ponašanja – engl. *behavior analysis*
 - naglašava dinamičko ponašanje kao izvor za pronalaženje razreda
- Analiza domene engl. *domain analysis*
 - koncentrira se na elemente važne ekspertima (npr. Moore i Bailin)
- Analiza obrazaca uporabe – engl. *use case analysis*
 - organizira prethodne pristupe u proces
- CRC kartice – engl. *Class-Responsibility-Collaboration cards*
 - učinkovit način analize scenarija
- Analiza neformalnog opisa
 - imenice, glagoli ,..

- ***Sistemska model domene*** može izostaviti razrede potrebne u izgradnji cjelovitog programskog sustava.
 - može sadržavati manje od polovice od ukupnog broja razreda u sustavu.
 - izgrađuje se i koristi neovisno o skupu
 - razreda koji modeliraju korisničko sučelje
 - razreda koji modeliraju arhitekturu sustava
- Cjeloviti ***model sustava*** sadrži:
 - sistemski model domene
 - razrede koji modeliraju korisničko sučelje
 - razrede koji modeliraju arhitekturu sustava
 - pomoćne razrede



Preporučena sekvenca aktivnosti

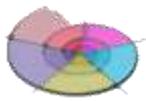


1. Identificiraj početni skup kandidata za razrede.
2. Dodaj pridruživanja (*engl. Associations*) i attribute.
3. Pronađi generalizacije.
4. Popiši temeljne **odgovornosti** (*engl. Responsibilities*) svakog razreda.
5. Odluči se za specifične operacije.
6. Iteriraj proces dok ne dobiješ zadovoljavajući model
 - dodaj ili izbriši razrede, pridruživanja, attribute, generalizacije, odgovornosti ili operacije.
 - identificiraj razrede sučelja.

Odgovornost je nešto što sustav mora obaviti.

To je viša razina apstrakcije ponašanja od operacije.

Hijerarhija ponašanja: **odgovornost** \Leftarrow **operacija** \Leftarrow **metoda**.



Identifikacija razreda



- Tijekom razvoja modela domene **otkrij** razrede.
- Kada si usredotočen na korisničko sučelje ili na arhitekturu sustava razredi se **osmišljavaju** (engl. *Invent*)
 - cilj je riješiti određen problem u oblikovanju.
 - osmišljavanje je dopustivo i tijekom razvoja modela domene.
- Obrati pažnju na mogućnost ponovne uporabe razreda (engl. *Reuse*).

Tehnika otkrivanja razreda

- Pogledaj u izvorne dokumente opisa zahtjeva.
- Izdvoji **imenice** i **imeničke izraze**.
- Ukloni imenice koje:
 - su redundantne
 - predstavljaju instance
 - nejasne su i vrlo općenite
 - nepotrebne u primjeni
- Obrati pažnju na razrede u domeni koji predstavljaju tipove korisnika ili druge aktore.

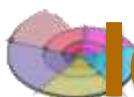
Primjer:

- Označavanja razreda: dobri, *loši* i razredi za koje nismo sigurni.
 - **sustav** osigurava temeljne usluge za rukovanje bankovnom računima u banci koja se zove **OOBank**. Ta banka ima više poslovnica, koje imaju svoje adrese i **oznaku poslovnice**. Pojedini klijent otvara račun u poslovnici banke.

Sustav – previše općenit razred

OOBank – očito instanca

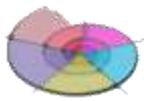
oznaka poslovnice – očito instanca



Identificiranje pridruživanja i atributa



- Započni s razredima koji su središnji i najvažniji.
- Odluči o jasnim i očiglednim podacima koje ti razredi moraju sadržavati, te o njihovim odnosima s drugim razredima.
- Potraži informacije koje svaki razred mora podržavati.
- Neke imenice koje su odbačene kao razredi, sada mogu biti atributi.
- Atribut bi općenito trebao sadržavati jednostavnu vrijednost
 - npr. string, broj
- Proširuj dijagram iznutra prema van do razreda koji su manje važni.
- Izbjegavaj veliki broj pridruživanja i atributa u pojedinom razredu.
 - sustav je jednostavniji ako rukuje s manje informacija.



Pridruživanja između razreda

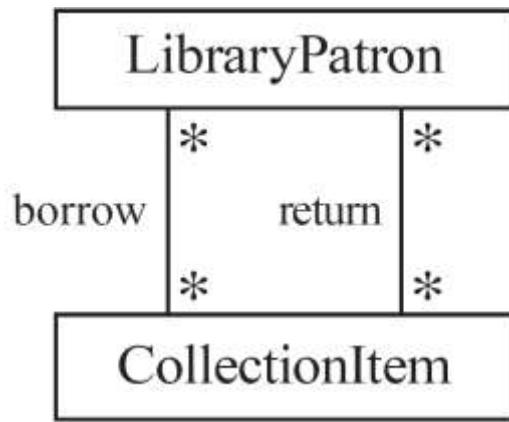


- Identificiranje valjanih pridruživanja između razreda
 - uporaba sekvencijskih dijagrama
 - pridruživanje postoji ako razred:
 - *posjeduje (ili ovlađava)*
 - *upravlja*
 - *spojen je s*
 - *odnosi se prema (engl. is related to)*
 - *dio je od*
 - *ima dijelove*
 - *član je*
 - *ima članove*
- Specificiraj brojnost na oba kraja pridruživanja.
- Jasno označi pridruživanje.

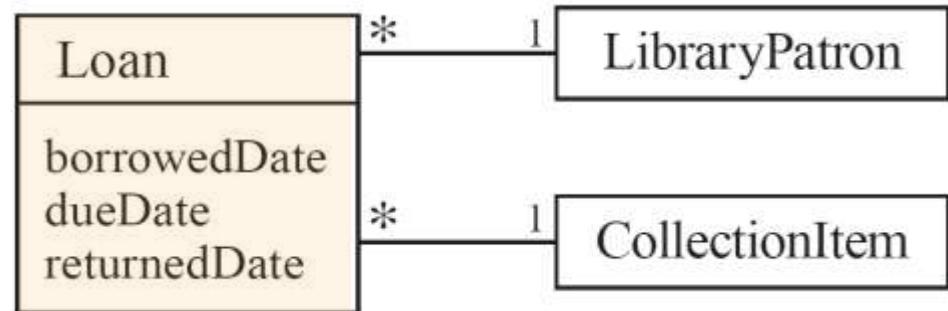
nekog drugog razreda u modelu

Akcije i pridruživanja

- Česta pogreška je predstaviti akcije kao pridruživanja.



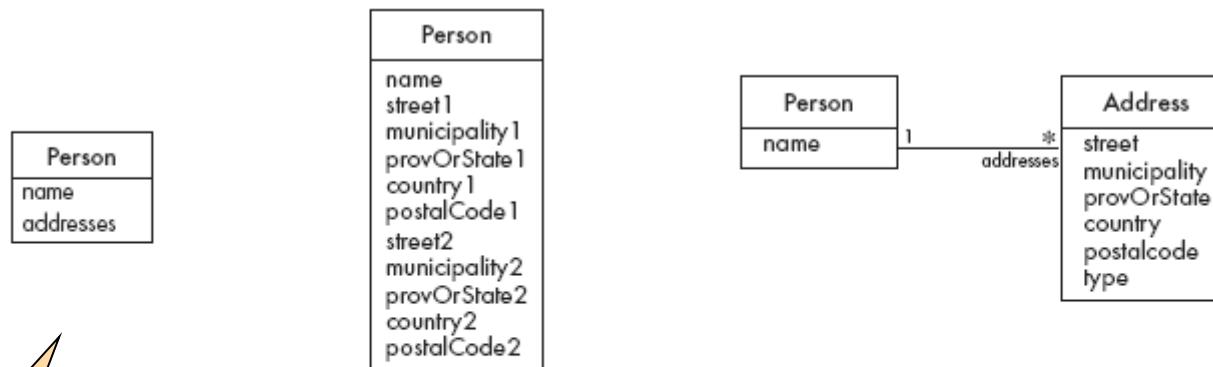
Loše, akcije su predstavljene kao pridruživanja



Bolje, operacija **borrow** kreira **Loan**.
Operacija **return** postavlja **returnedDate** atribut.
Obrati pažnju da su **borrow** i **return** **operacije**.

Identifikacija i specificiranje valjanih atributa

- Nije dobro imati mnogo kopija atributa.
- Ako neki podskup atributa u pojedinom razredu čini koherentnu grupu kreiraj poseban razred koji sadrži te atribute.



Bad, due to
a plural attribute

Loše, atribut
u množini.

Bad, due to too many
attributes, and the
inability to add more
addresses

Loše,
Previše
atributa.

Good solution. The type indicates whether it
is a home address, business address etc.

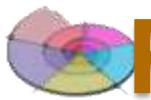
Dobro, Tip označuje da li je to
adresa stanovanja ili posla.



Identificiranje generalizacija i sučelja



- Postoje dva načina identificiranja generalizacija:
 - odozdo prema gore
 - Grupiraj slične razrede i kreiraj novi nadrazred
 - odozgo prema dolje
 - Najprije potraži općenitije razrede pa ih specijaliziraj ako je potrebno.
- Kreiraj sučelje (*engl. Interface*) umjesto nadrazreda ako postoji potreba za varijablom koja bi morala držati instance nekoliko razreda, a pri tome:
 - razredi su vrlo različiti osim nekoliko zajedničkih operacija.
 - jedan ili više razreda već imaju svoj nadrazred (u većini jezika nema višestrukog nasljeđivanja).
 - želi se ograničiti operacije s varijablom (koja je deklarirana da sadrži instance više razreda) samo na operacije dostupne u sučelju.



Primjer: Sustav rezervacije avio-leta



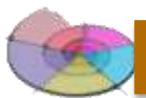
The reservation system keeps track of passengers who will be flying in specific seats on various flights, as well as people who will form the crew.

For the crew the system needs to track what everyone does and who supervises whom.

Sustav za rezervaciju vodi računa i pohranjuje podatke o putnicima koji će letjeti na odabranom sjedištu na raznim letovima, te podatke o posadi aviona. Što se posade tiče sustav treba voditi računa tko radi te tko nadgleda koga.

■ Postupak identificiranja razreda:

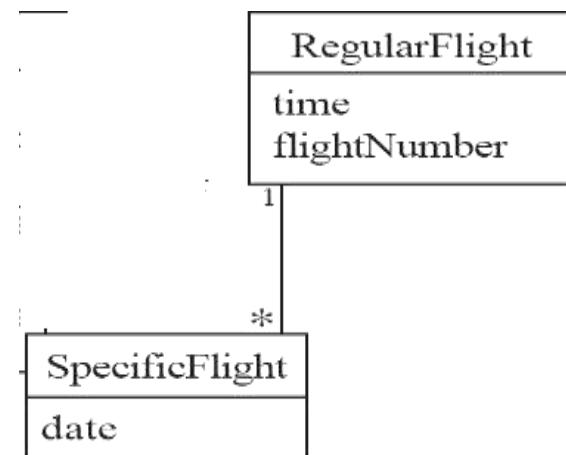
- imenice na početnoj listi razreda: **Flight**, **Passenger**, **Employee**
- ostale imenice:
 - “reservation system – nije razred, već dio sustava
 - “seat – atribut razreda Flight
 - “crew – Employee je mnogo fleksibilnije

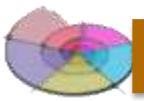


Primjer: Sustav rezervacije avio-leta



- **Flight** – središnji razred (sadrži *date*, *time*, *flightNumber*).
 - letovi koji polijeću dnevno u isto vrijeme imaju isti broj leta (*flight number*).
 - podijeli **Flight** u **RegularFlight** (sadrži *time* i *flight number*) i **SpecificFlight** (odlazi na određen dan).
 - pridruživanje između dva razreda: jedan - mnogo.
 - koliko letova u danu?



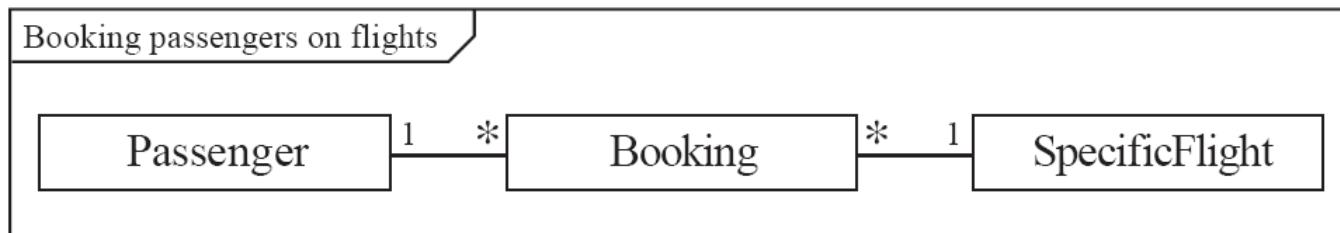


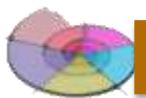
Primjer: Sustav rezervacije avio-leta



Putnici i rezervacije leta:

- **passenger** ima atribut ime putnika ali i neki ID broj.
- između **Passenger** i **SpecificFlight** općenito postoji odnos *mnogo-mnogo*. Ranije pokazano da je između **Passenger** i **SpecificFlight**, potrebno ubaciti pridruženi razred **Booking** sa **seatNumber** kao atributom.
- instanca **Passenger** se kreira prije ili istovremeno s **Booking**.
- **booking** je uvijek za jednog **Passenger-a**.
- **passenger** može imati više **Bookinga** (može i 0 **Bookinga**).
- za svaki **Booking** postoji samo jedan **SpecificFlight**.
- svaki **SpecificFlight** može imati veći koji broj **Bookinga** (od 0 do kapaciteta aviona).

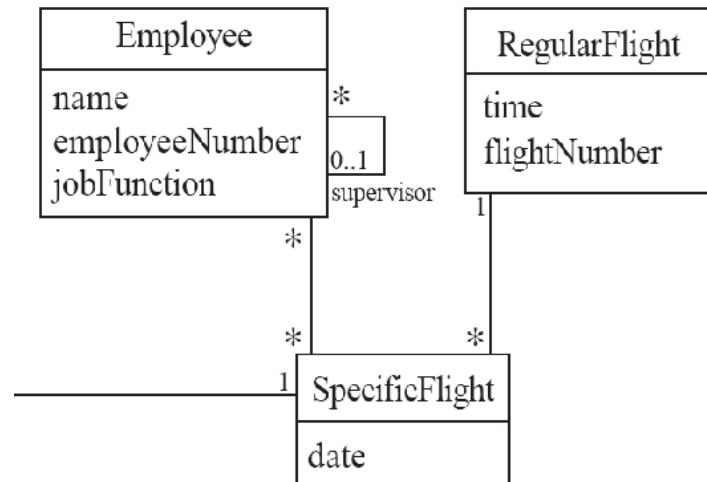




Primjer: Sustav rezervacije avio-leta



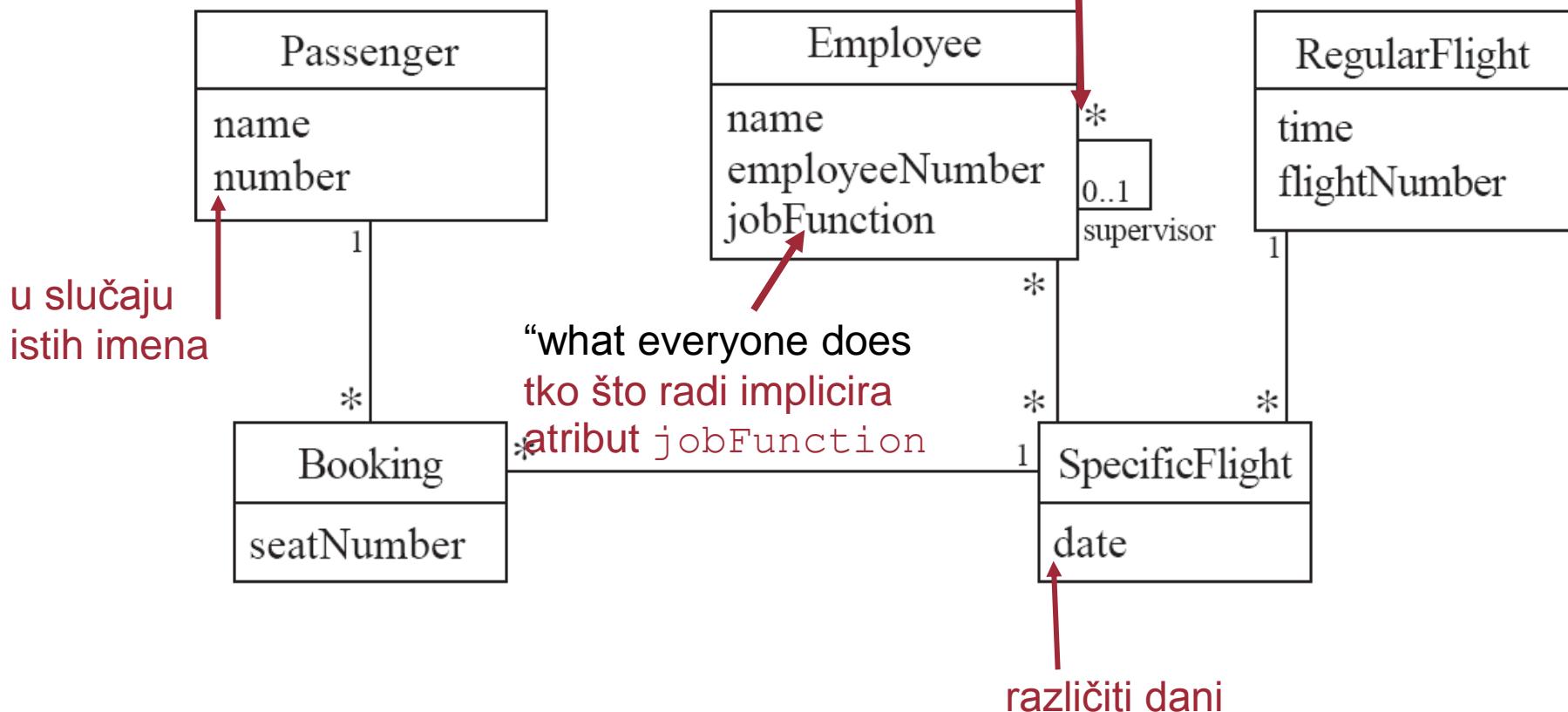
- Razred zaposlenik (*Employee*):
- Atributi: *name*, *employeeID*, *jobFunction* (tko što radi).
- *employee* može biti *supervisor* drugim *Employee*. Potrebno je uvesti refleksivno pridruživanje.
- *employee* je pridružen *SpecificFlight* asocijacijom *mnogo-mnogo*.

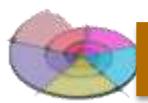




Primjer: Sustav rezervacije avio-leta

■ atributi i pridruživanja





Primjer: Sustav rezervacije avio-leta



- Identificiranje generalizacija
- Razredi **Passenger** i **Employee** dijele zajedničku informaciju.
- Definiranjem nadrazreda **Person** nije dovoljno dobro jer jedna osoba (**person**) može biti oboje: i putnik (**passenger**) i zaposlenik (**employee**).
- ***Problem: Instanca može nastati i postojati samo od jednog razreda !***
- Rješenje: uvesti razred **PersonRole** i pridruživanja koje dozvoljava da svaka osoba može imati dvije uloge.

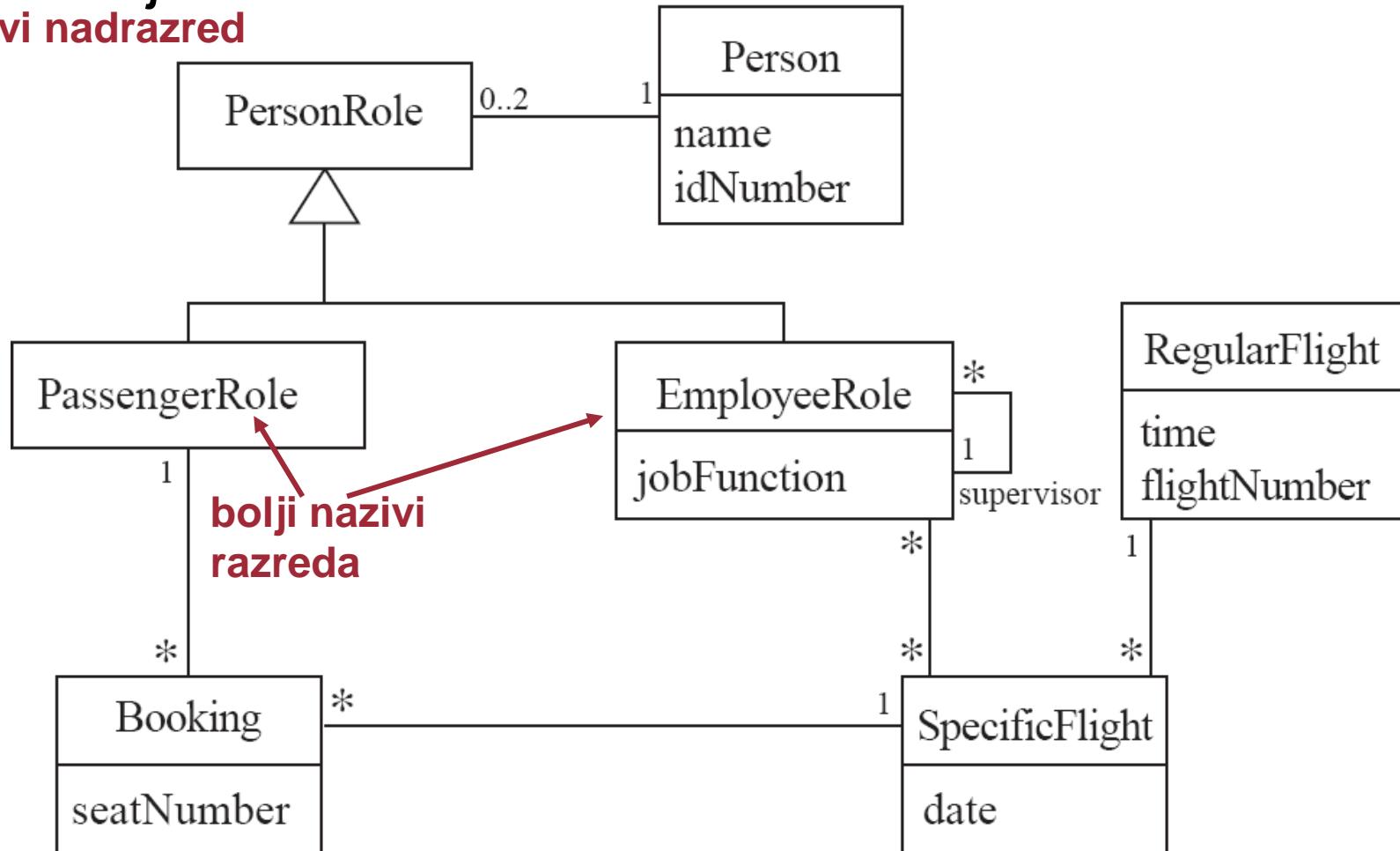


Primjer: Sustav rezervacije avio-leta



generalizacija

novi nadrazred

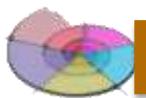


Alociranje odgovornosti razredima

- **Odgovornost** (engl. *responsibility*) je nešto što sustav mora izvršiti.
 - svaki **funkcionalni zahtjev** mora se pripisati nekom razredu.
 - Sve odgovornosti jednog razreda moraju biti *jasno povezane*.
 - Ako jedan razred ima previše odgovornosti, razmotri *podjelu* toga razreda u različite razrede.
 - Ako razred nema odgovornosti, tada je vjerojatno *beskoristan*.
 - Ako se neka odgovornost ne može pripisati niti jednom od postojećih razreda, mora se kreirati *novi* razred.
 - za određivanje odgovornosti:
 - Analiziraj **obrasce uporabe** (engl. *use case*).
 - U opisu sustava potraži **glagole i imenice koje opisuju akcije**.

Kategorije odgovornosti

- Postavljanje i dohvaćanje vrijednosti atributa.
- Kreiranje i inicijalizacija novih instanci.
- Upis i čitanje iz trajne pohrane podataka.
- Dokidanje (uništavanje) instanci.
- Dodavanje i brisanje veza pridruživanja između instanci.
- Kopiranje, konverzija, preslikavanje, prijenos, izlaz podataka.
- Izračunavanje numeričkih vrijednosti.
- Navigacija i pretraživanje.
 - npr. pronaći sve instance koje odgovaraju nekom kriteriju
- Drugi specijalizirani poslovi.



Primjer: Sustav rezervacije avio-leta



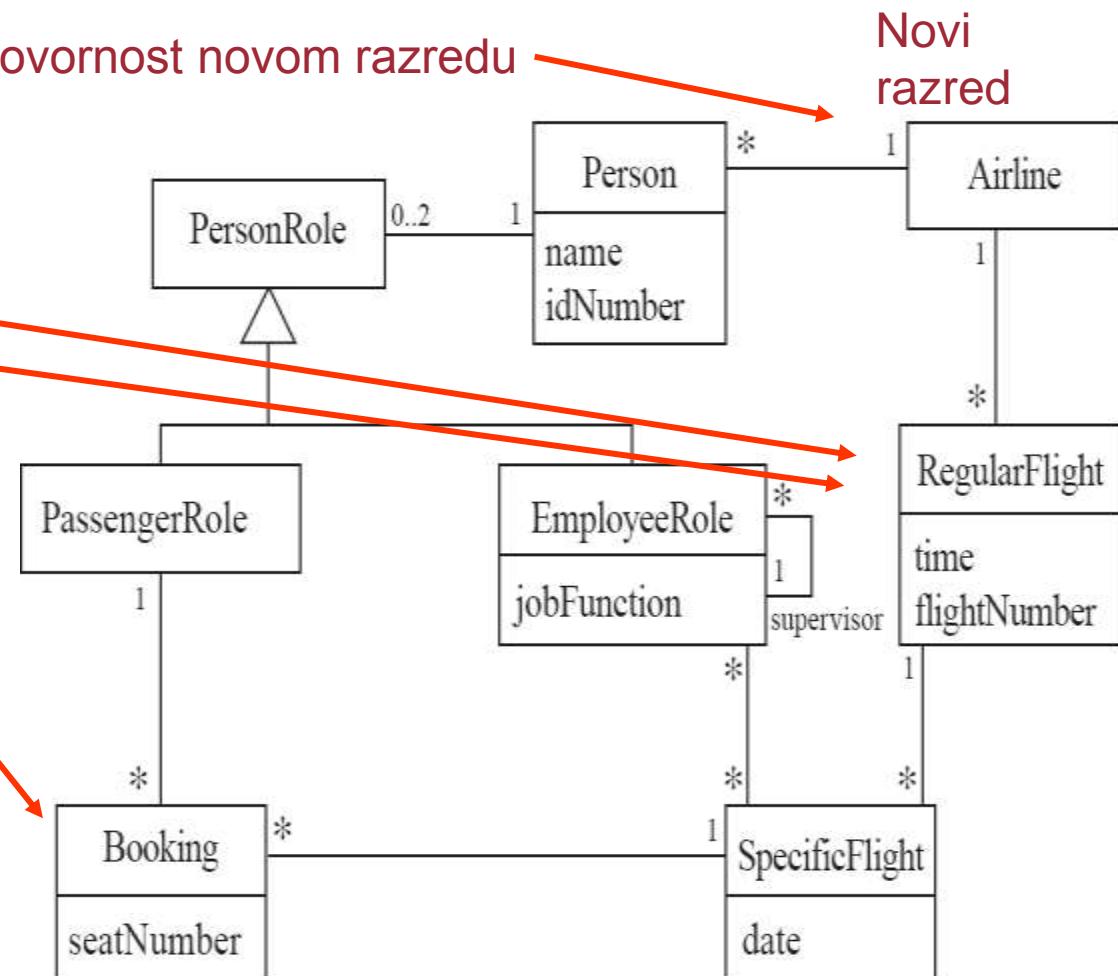
■ odgovornosti

- Kreiranje novog regularnog leta
- Pronalaženje leta
- Modificiranje atributa leta
- Kreiranje specifičnog leta
- Rezervacija za putnika
- Otkazivanje rezervacije

Pripiši odgovornost novom razredu

Novi razred

Može se pripisati u Booking, ali Booking još nije kreiran u trenutku kad se ova odgovornost inicijalizira





Primjer alociranja odgovornosti:



- Dinamičko kreiranje **RegularFlight** prepušta se objektu. Stoga se uvodi novi razred **Airline** čiji objekt će kreirati instancu od **RegularFlight**. Vjerojatno će postojati samo jedan objekt od razreda **Airline**.
- Kreiranje **RegularFlight** prepušta se objektu. Stoga se uvodi novi razred **Airline**. Vjerojatno će postojati samo jedan objekt toga novoga razreda.
- U pronalaženju određenog **RegularFlight**, odgovornost za održavanje kolekcije objekata **RegularFlight** prepušta se također razredu **Airline**.
- Modificiranje atributa u **RegularFlight** je odgovornost tog istog razreda **RegularFlight**.
- Kreiranje **SpecificFlight** može biti odgovornost razreda **RegularFlight**, jer **RegularFlight** postoji kada se ta odgovornost inicira.
- Otkazivanje leta **SpecificFlight** može biti odgovornost toga istog razreda.
- Odgovornost za rezervaciju (engl. *Booking a passenger*) je objašnjena na prethodnoj slici.
- Otkazivanje rezervacije je prirodna odgovornost razreda **Booking**.



Identificiranje operacija i metoda



- Hijerarhija ponašanja:
 - ***Odgovornost* \Leftarrow *operacije* \Leftarrow *metode***
- Operacije ostvaruju odgovornosti pojedinog razreda i implementiraju se metodama.
 - može postojati nekoliko operacija i metoda koje realiziraju jednu odgovornost.
 - temeljne operacije (metode) koje implementiraju neku odgovornost normalno se deklariraju kao **public**.
 - druge operacije (metode) koje surađuju u realizaciji odgovornosti trebaju biti što je moguće više deklarirane kao **privatne**.
 - Inače, ako bi sve operacije mogle biti izravno pozivane (public), sustav bi se mogao naći u nestabilnom stanju (u sredini neke odgovornosti).



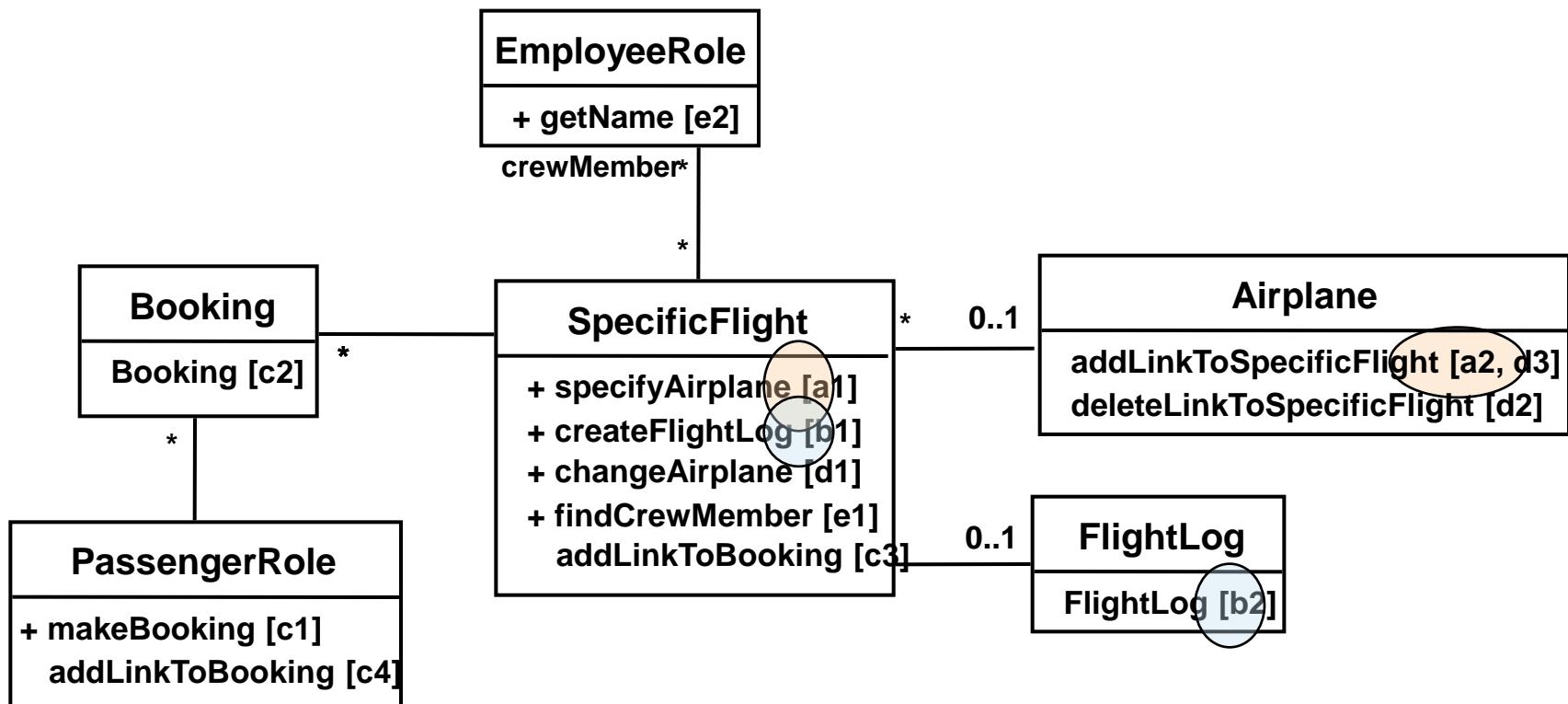
Tipične jednostavne odgovornosti



- Implementacija odgovornosti najčešće traži kolaboraciju nekoliko razreda.
- Tipične jednostavne odgovornosti su:
 - a) Povezati dva postojeća objekta
 - b) Kreirati objekt i povezati ga s nekim postojećim objektom.
 - c) Kreirati pridružen razred (*engl. association class*).
 - d) Promijeniti odredište veze (*engl. link*) između objekata.
 - e) Pretražiti skup pridruženih objekata

Primjer

- Realizacija odgovornosti koje su pripisane razredu **SpecificFlight** (dodani su i novi razredi **Airplane** i **FlightLog**)



Izvor: T C. Lethbridge and R. Laganière <http://www.lloseng.com>

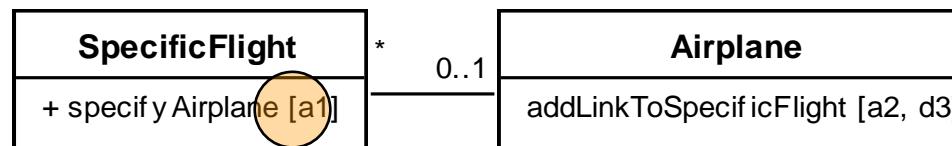
- Temeljni zadatak: Povezivanje dva postojeća objekta.
 - to je u objektnom dijagramu kreiranje veze (*engl. link*), a u implementaciji definiranje varijable u koju su upisani objekti.
 - varijable instanci u kojoj će biti referenciran objekt. Tako da jedan objekt zna za drugoga.
 - Dvosmjerna veza razbija se na dvije jednosmjerne.
- Asocijacija se implementira kao veza (*engl. link*) kroz varijablu instanci.
- Dvosmjerna asocijacija/veza se rastavlja na dvije jednosmjerne asocijacije/veze.



Povezivanje dva postojeća objekta



- Primjer: Odgovornost a: dodavanje dvosmjerne veze između instance **SpecificFlight** i instance od **Airplane** (postojeći objekti).
 - a1 - (public) instanca od **SpecificFlight**
 - izgradi jednosmjernu vezu prema instanci od Airplane
 - zatim zove operaciju a2.
 - a2 - (non-public)* instanca od **Airplane**
 - izgradi jednosmjernu vezu natrag do instance od SpecificFlight.



- Ako je brojnost veze višestruka upotrebljava se za tip varijable razreda npr. lista:

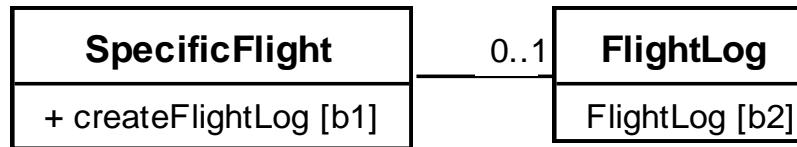
private List bookings;

* Ako ništa nije navedeno pod vidljivost, podrazumjeva se **package** razina vidljivosti



Kreiranje objekta i povezivanje s postojećim objektom

- Primjer: Odgovornost b - Kreiranje objekta **FlightLog** i povezivanje sa **SpecificFlight**:



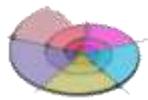
- B1 – (public) instanca od **SpecificFlight**
 - poziva konstruktor iz **FlightLog** (operacija b2)
 - nakon konstrukcije, ostvaruje se jednosmjerna veza prema tom novom objektu iz **FlightLog**.
- B2 – (non-public) konstruktor razreda **FlightLog**
 - pored drugih akcija, ostvaruje jednosmjernu vezu na **SpecificFlight**.
- Primjer:
 - **regularFlight** kreira više objekata (više specifičnih letova) i povezuje se sa **SpecificFlight**.

RegularFlight kreira objekt SpecificFlight

```
class RegularFlight {  
    private List specificFlights; // var tipa razred List  
    ...  
    // glavna metoda koja zove konstruktor u SpecificFlight  
    public void addSpecificFlight(Calendar aDate) {  
        specificFlight newSpecificFlight; // var za novi objekt  
        newSpecificFlight = new SpecificFlight(aDate, this);  
        specificFlights.add(newSpecificFlight); // dodaj u listu  
    }...}
```

*odnosi se na trenutni
objekt*

```
class SpecificFlight {  
    private Calendar date;  
    private RegularFlight regularFlight; // veza s RegularFlight  
    ...  
    specificFlight(Calendar aDate, RegularFlight aRegularFlight) {  
        date = aDate; // na određen dan  
        regularFlight = aRegularFlight; } ... }
```



Primjer: SpecificFlight

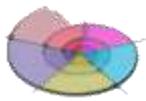


```
class SpecificFlight
{
    private Calendar date;
    private RegularFlight regularFlight;
    private TerminalOfAirport destination;
    private Airplane airplane;
    private FlightLog flightLog;

    private ArrayList crewMembers;
    // of EmployeeRole
    private ArrayList bookings
    ...
}
```

// konstruktor od SpecificFlight
smijemo pozvati samo iz
addSpecificFlight

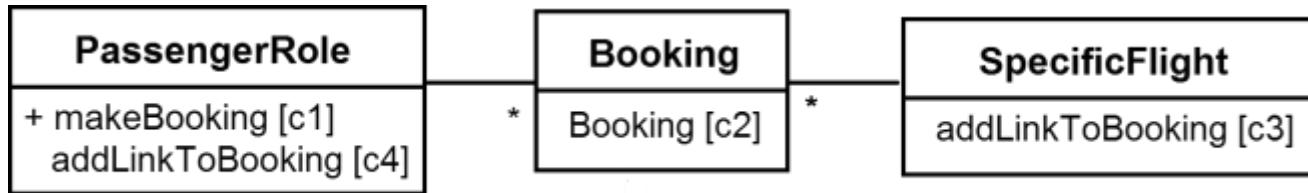
```
SpecificFlight(
    Calendar aDate,
    RegularFlight aRegularFlight)
{
    date = aDate;
    regularFlight = aRegularFlight;
}
```



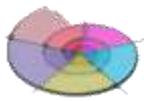
Kreiranje pridruženog razreda



- Primjer: Odgovornost c) - Kreiranje instance **Booking** koja povezuje **PassengerRole** i **SpecificFlight**



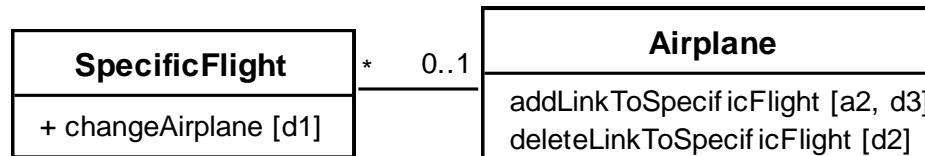
- C1 – (public) instanca od **PassengerRole** zove konstruktor od **Booking** (c2).
- C2 – (non-public), konstruktor iz **Booking**, uz druge akcije stvara:
 - jednosmjernu vezu natrag na **PassengerRole**.
 - jednosmjernu vezu prema **SpecificFlight**
 - poziva operacije c3 i c4
- C3 – (non-public), instanca od **SpecificFlight** stvara jednosmjernu vezu s instancom od **Booking**
- C4 – (non-public), instanca od **PassengerRole** stvara jednosmjernu vezu prema instanci od **Booking**



Promjena odredišta veze

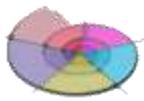


- Primjer: Odgovornost d) - Promjena postojećih aviona (objekata) *airplane1* u *airplane2* kao instanci iz razreda *Airplane*



Uobičajene
UML
oznake za
objekte

- D1- (public), instance of *SpecificFlight*
 - briše vezu na *airplane1*
 - ostvaruje jednosmjernu vezu na *airplane2*
 - zove operaciju d2, a zatim operaciju d3.
- D2 - (non-public), *airplane1*
 - briše jednosmjernu vezu do instance do *SpecificFlight*.
- D3 - (non-public) *airplane2*
 - ostvaruje jednosmjernu vezu prema instanci od *SpecificFlight*.



Pretraživanje skupa objekata



- Primjer: Odgovornost e) - Pretraživanje i pronalaženje po imenima članova posade za određeni objekt iz razreda **SpecificFlight**



- E1 - (public), instanca od **SpecificFlight**
 - kreira sekvencijski upit (Iterator) koji iterira preko svih svojih veza tipa **crewMember**.
 - za svaku od tih veza zove operaciju e2, dok se ne pronađe podudarnost.
- E2 – (public), instanca od **EmployeeRole** vraća ime člana posade.

Oblikovanje dijagrama razreda na papiru

- Nakon identifikacije razreda, svakom razredu se dodijeli mali komad papira (karticu) s nazivom razreda
- Te se kartice nazivaju CRC kartice (*engl. Class-Responsibility-Collaboration*)
- Nakon što se identificiraju atributi i odgovornosti, popišu se sustavno na CRC kartici određenog razreda.
 - tehnika modeliranja zasnovana na odgovornostima (*engl. Responsibility-based modeling*)
- Ako se sve odgovornosti ne mogu ispisati na jednoj CRC kartici:
 - to sugerira da se razred treba podijeliti u dva međusobno povezana razreda.
- Premještajući i pomicajući kartice oblikuje se dijagram razreda.
- Povlačenje linja između kartica predstavlja asocijacije i generalizacije.

Modeliranje zasnovano na odgovornostima



- Pogodno za oblikovanje dijagrama razreda i podjelu sustava na podsustave
- Pretpostavke:
 - ljudi mogu intuitivno donoditi dobre prosudbe o pridruživanju odgovornosti
 - osnovne odluke o podjeli sustava donose se temeljem odgovornosti dijelova prema cjelini(sustavu)
 - mora li ovaj objekt obrađivati odgovornost?
- Tipovi odgovornosti
 - aktivno obraditi nešto
 - pružiti informaciju

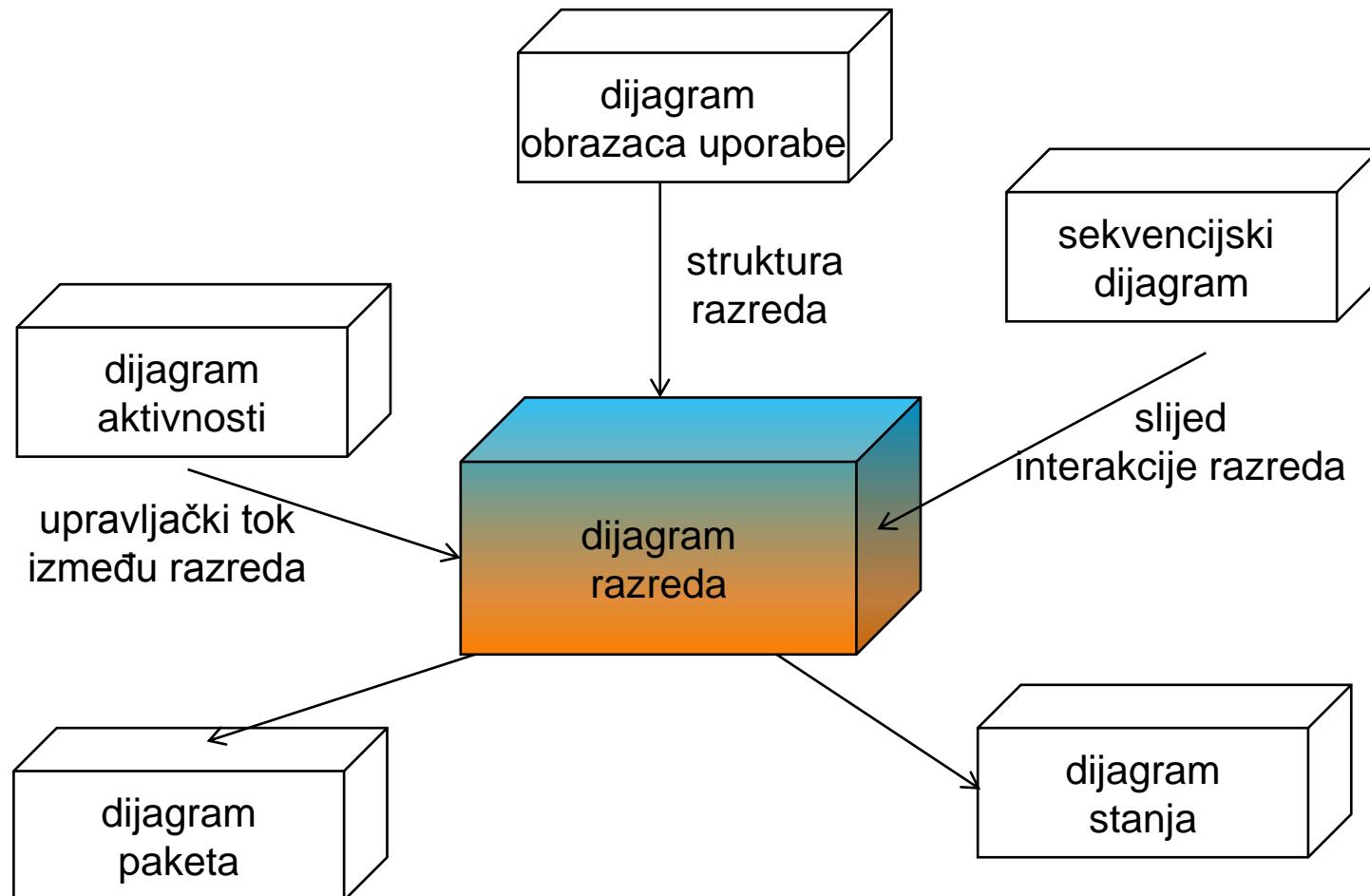
CRC kartice

- Tehnika objektno usmjerene analize
- Pogodno za analizu scenarija
- Preporučeno u ekstremnom programiranju...
 - primjer: <http://www.extremeprogramming.org/rules/crccards.html>

Class Name:	Person	Superclass:	Subclasses:
Responsibilities		Collaborations	
Inicijalizacija (name, address, telephone, e-mail)		name, address, telephone, e-mail	
Print		name, address, telephone, e-mail	
Class Name:	Superclass:	Subclasses:	
Responsibilities		Collaborations	



Uloga dijagrama razreda



- Dijagram razreda – osnova za analizu, oblikovanje i implementaciju

- Modeliranje je posebno teška vještina.
 - čak i izvrsni programeri imaju poteškoća razmišljati na odgovarajućoj razini apstrakcije.
 - obrazovanje se tradicijski više fokusira na programiranje nego na modeliranje.
- Rješenje:
 - osiguraj da članovi tima imaju adekvatno obrazovanje.
 - imaj u timu jednu ili više iskusnih osoba za modeliranje.
 - temeljito recenziraj sve modele.

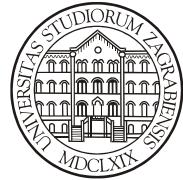
Diskusija

-
-
-
-
-

Oblikovanje programske potpore

2014./2015.

UML dijagrami



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



Tema

- Pregled UML dijagrama
- UML dijagrami interakcija
 - komunikacijski dijagram
- UML dijagrami stanja i aktivnosti
- UML komponentni dijagrami
- UML dijagrami razmještaja
- Ostali UML dijagrami

Literatura

- Sommerville, I., ***Software engineering***, 8th ed., Addison-Wesley, 2007.
- Grady Booch, James Rumbaugh, Ivar Jacobson: ***Unified Modeling Language User Guide***, 2nd Edition, 2005
- OMG; ***OMG Unified Modeling Language, Superstructure Version 2.2***; URL:
www.omg.org/spec/UML/2.2/Superstructure/PDF
- Simon Bennett, John Skelton, Ken Lunn: ***Schaum's Outline of UML***, Second Edition, 2005

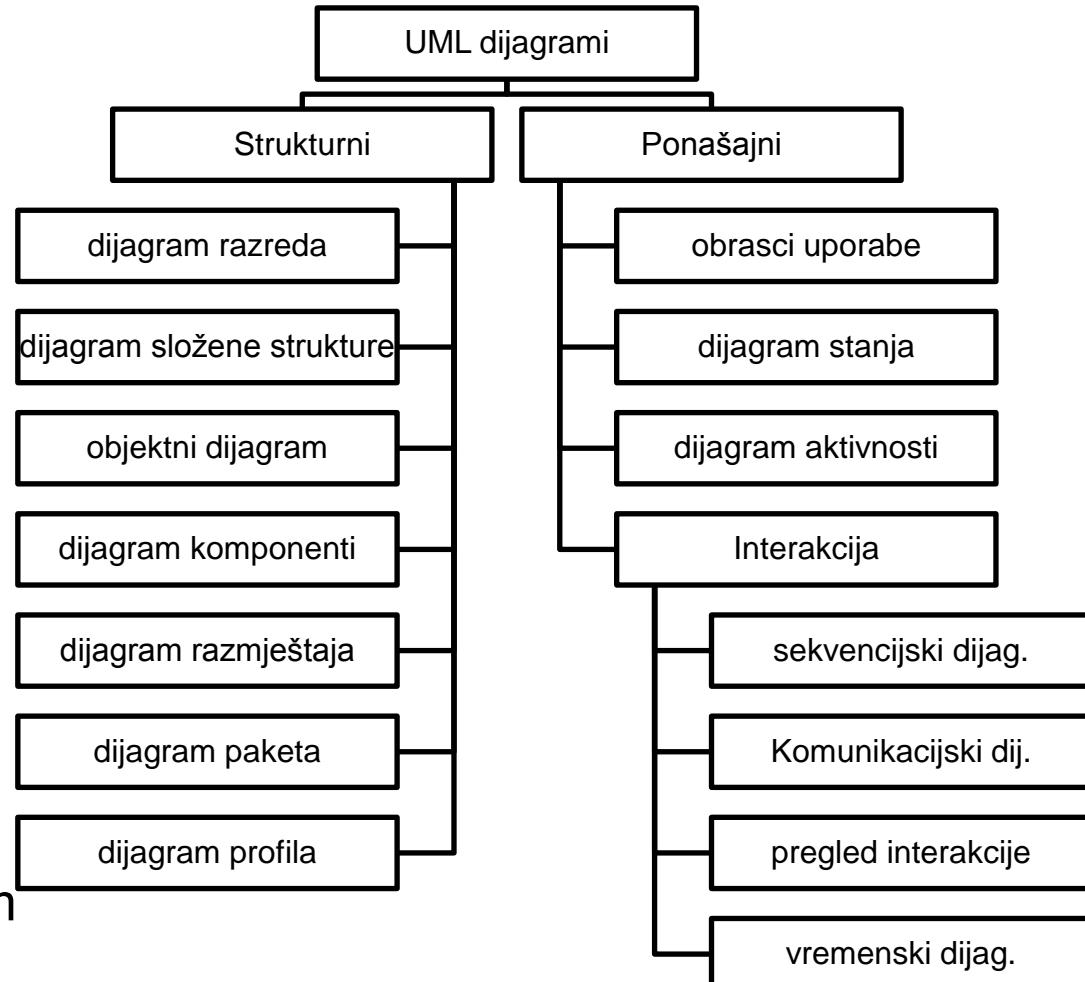
UML dijagrami

■ Pogled na model

- iz perspektive dionika
- djelomična reprezentacija sustava
- semantički konzistentan s ostalim pogledima

■ Specifičnost dijagrama

- svaki ima vlastitu sintaksu i semantiku
- evoluiraju tijekom procesa oblikovanja, te se mijenjaju donošenjem odluka o oblikovanju i proširuju novim detaljima



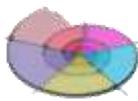
Dinamički?

UML dijagrami

- Dijagram obrazaca uporabe engl. *use case diagrams*
- Sekvencijski dijagram engl. *sequence diagrams*
- Komunikacijski dijagram engl. *communication diagrams*
- Dijagram stanja engl. *state machine diagrams*
- Dijagram aktivnosti engl. *activity diagrams*
- Dijagram komponenti engl. *component diagrams*
- Dijagram razmještaja engl. *deployment diagrams*
- Dijagram paketa engl. *package diagrams*
- Dijagram pregleda interakcije engl. *interaction overview diagrams*
- Vremenski dijagram engl. *timing diagrams*
- Dijagram profila engl. *profile diagram*
- Dijagram razreda engl. *class diagrams*
- Dijagram objekata engl. *object diagrams*
- Dijagram složene strukture engl. *composite structure diagrams*

UML dijagrami

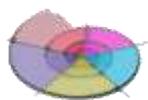
- Dijagram obrazaca uporabe
- Sekvencijski dijagram
- ***Komunikacijski dijagram***
- Dijagram stanja
- Dijagram aktivnosti
- Dijagram komponenti
- Dijagram razmještaja
- Dijagram paketa
- Dijagram pregleda interakcije
- Vremenski dijagram
- Dijagram profila
- Dijagram razreda
- Dijagram objekata
- Dijagram složene strukture



Dinamičke interakcije u sustavu



- Prikaz interakcija instanci modela
 - grafički prikaz instanci i podražaja
 - stvaranje i brisanje instanci
- Interakcije se modeliraju ako se želi:
 - specificirati kako instance uzajamno djeluju.
 - identificirati sučelja (*engl. interfaces*).
 - raspodijeliti zahtjeve (*engl. distribute requirements*).
- UML dinamički dijagrami interakcija:
 - sekvencijski – **vrijeme**
 - eksplicitno uređenje vremenskih odnosa između podražaja
 - modeliranje sustava za rada u stvarnom vremenu
 - dijagram komunikacije/kolaboracije – **struktura**
 - upotreba za opis struktura interakcije
 - usredotočeno na učinke instanci



Osnovni elementi interakcija



■ Objekti – engl. *Objects*

- različite uloge

<u>Ime</u>
atributi

■ Veze - engl. *Associations*

- prikazuju povezanost objekata koji komuniciraju

■ Poruke – engl. *Messages*

- komunikacija između instanci
- struktura:

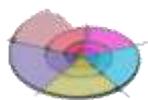
[broj_sekvence]{*[petlja]}{[uvjet]} [:]Ime(parametri) [: povratne vrijednosti]

[sequenceNumber]{*[loop]}{[condition]} {} methodName(parameters)[:
returnValue]

- oznaka sekvence prema Deweyom sustavu. npr. 1.1, 1.2, 1.3
- asinkrone poruke

opis
→

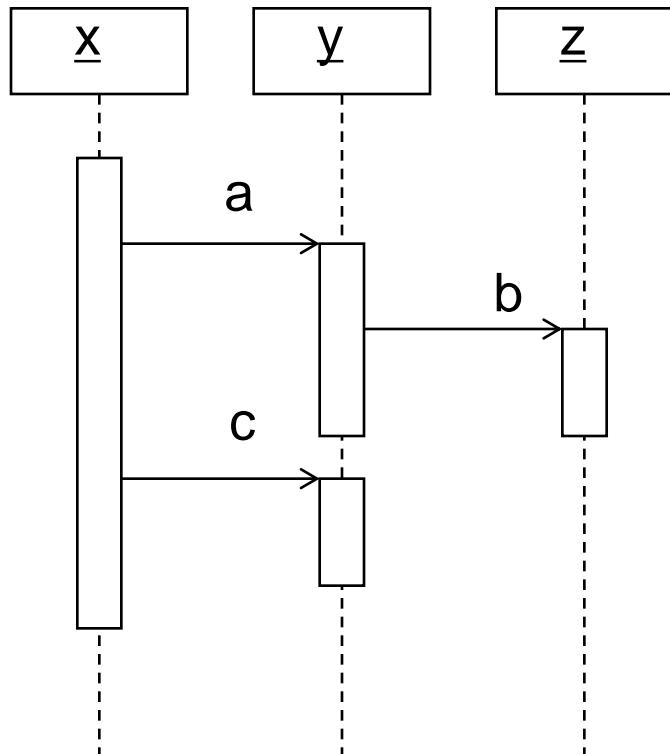
opis
→



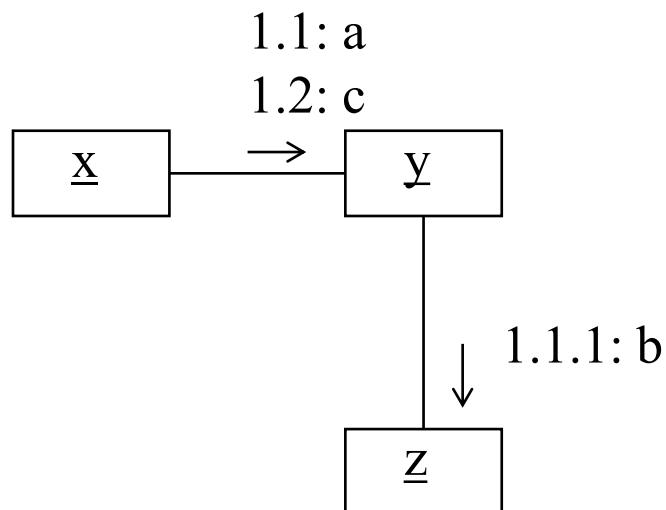
Dijagrami interakcija

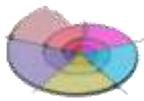


- Sekvencijski dijagram



- Komunikacijski dijagram



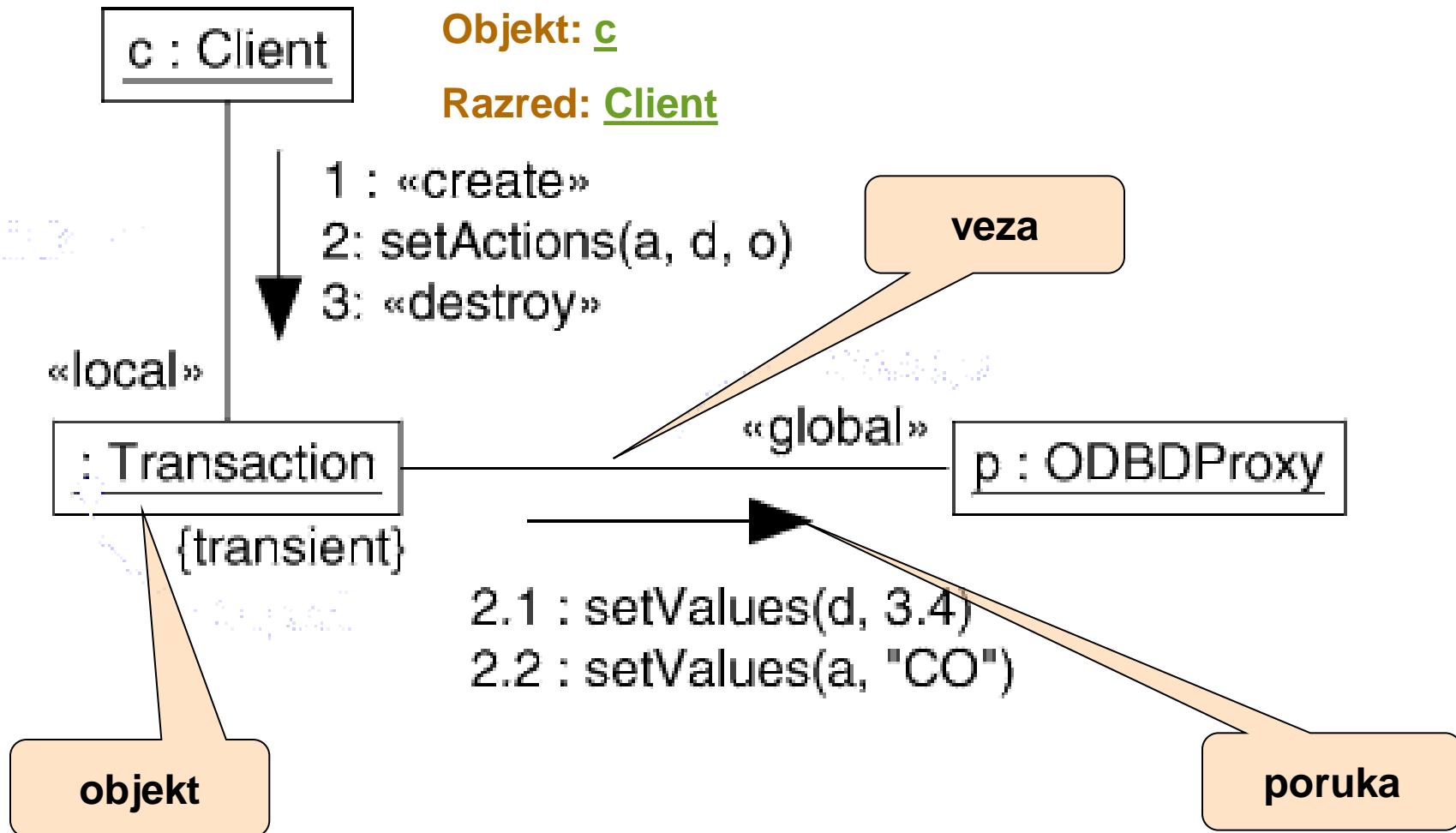


Komunikacijski dijagram

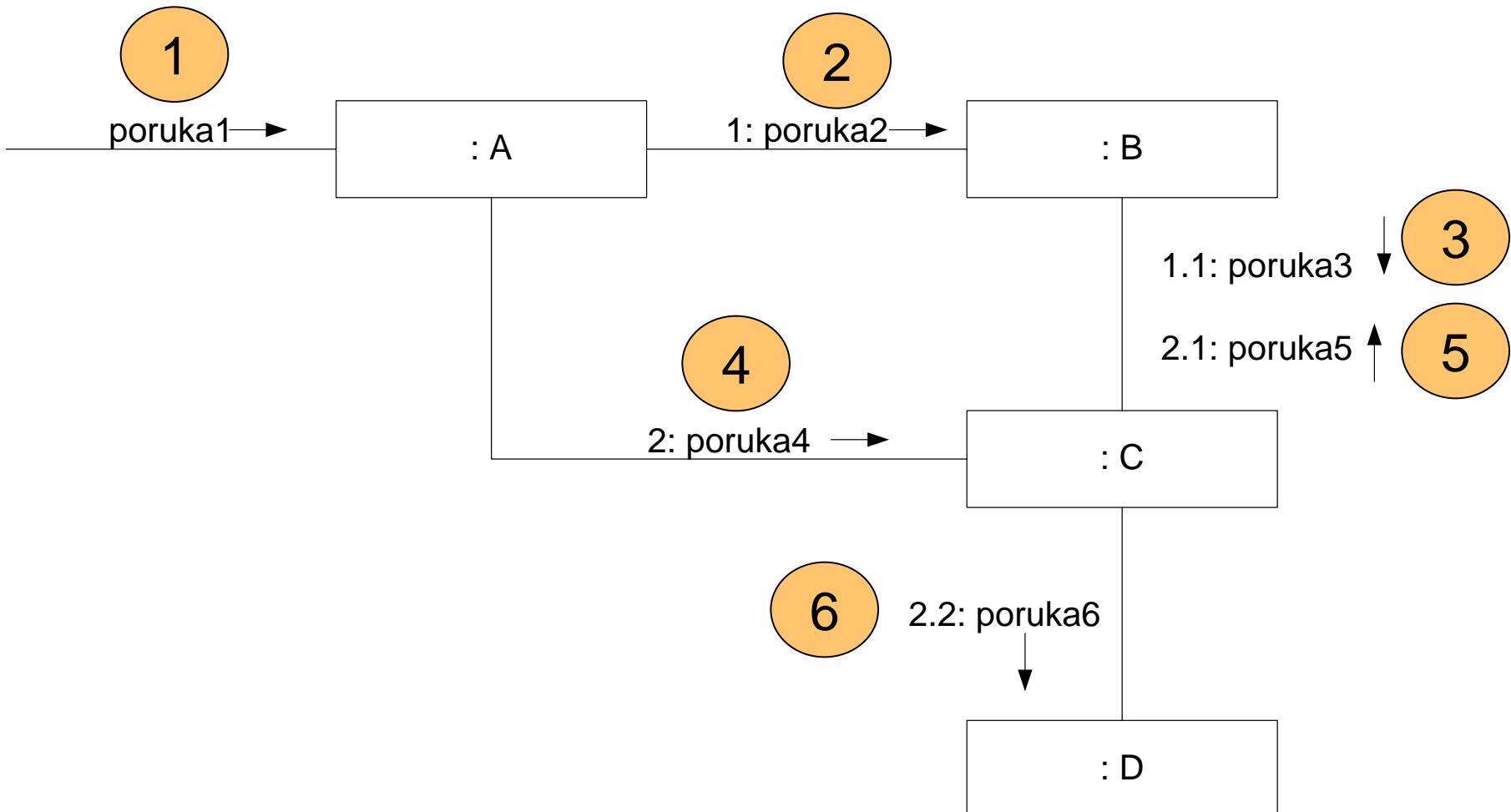


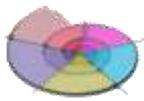
- engl. *Communication Diagrams*
- Prijašnje inačice = kolaboracijski dijagram, engl. *Collaboration Diagram*
- Obuhvaća dinamičko ponašanje
 - poruke – tko šalje kome; uređen redoslijed.
 - definira uloge instanci tijekom obavljanja nekog zadatka
 - ne prikazuje vremenske odnose.
- Modelira upravljački tok
 - prikaz koordinacije
 - nije izravno vidljiv
 - specificira tijek komunikacije između instanci tijekom suradnje

Primjer:



Primjer: Označavanje i redoslijed poruka



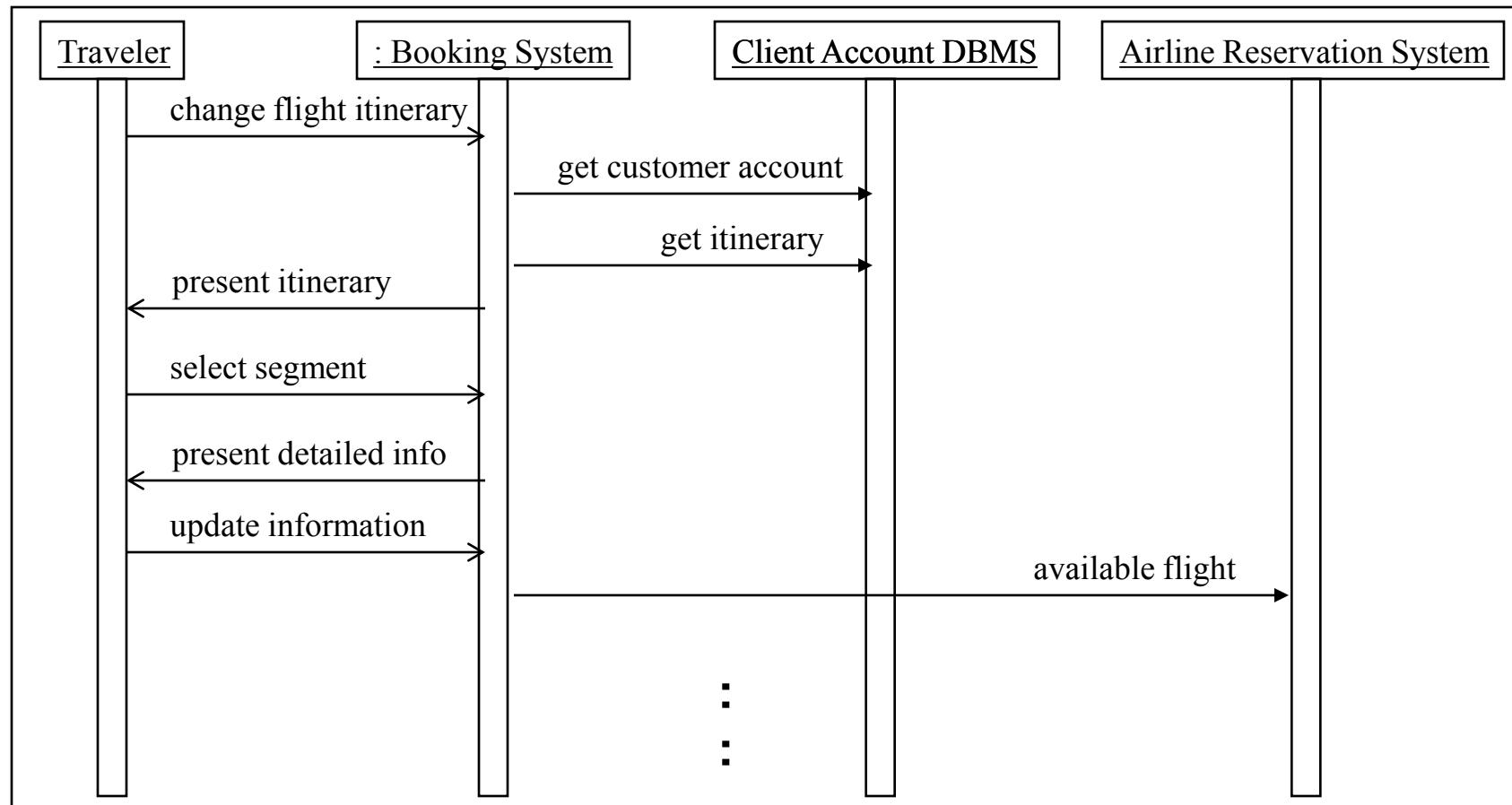


Primjer: Promjena rute leta

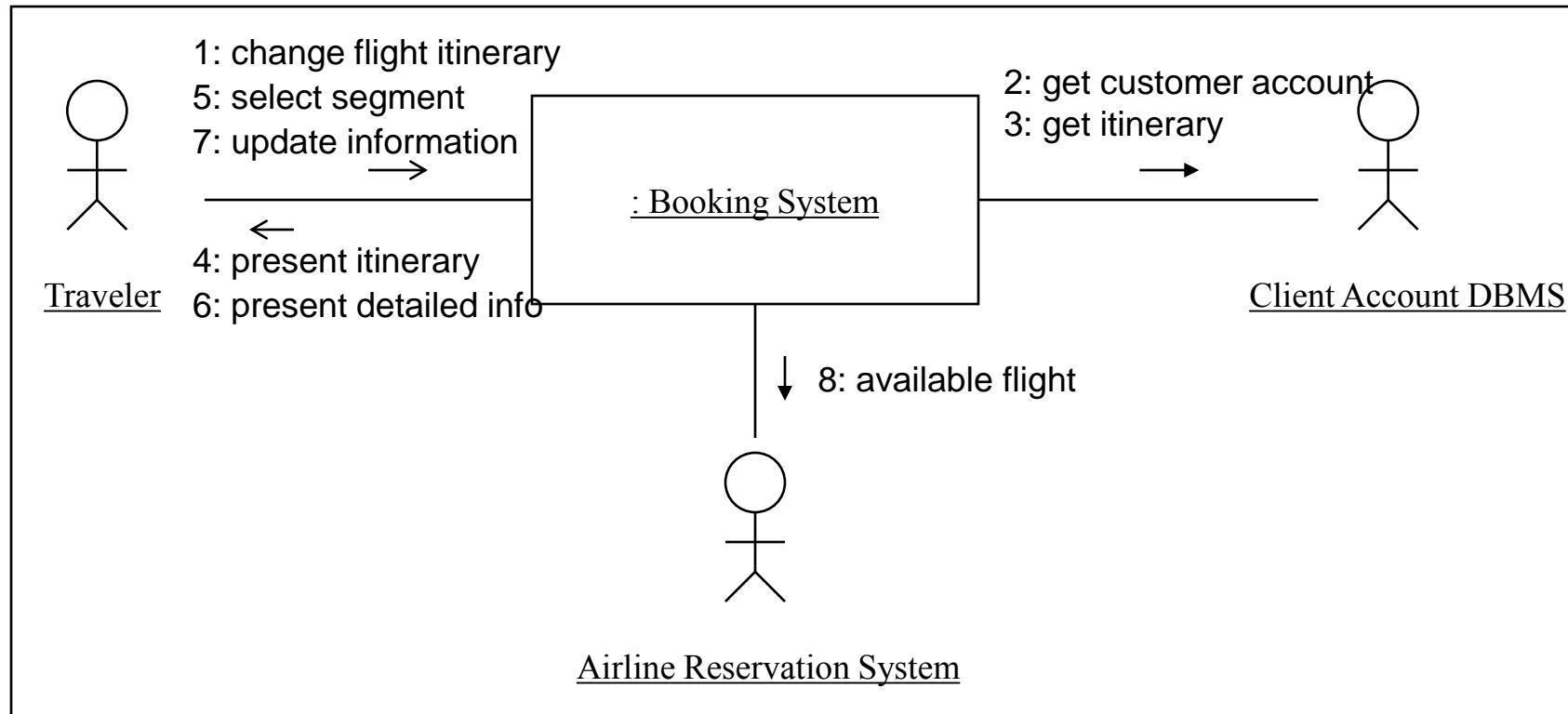


- **Aktori:**
 - putnik, baza računa klijenta (s planom puta), rezervacijski sustav avio kompanije.
- **Preduvjeti:**
 - putnik se prijavio na sustav i odabroa opciju "promjena leta".
- **Temeljni tijek transakcija**
 - sustav dohvaća putnikov bankovni račun i plan puta iz baze.
 - sustav pita putnika da odabere dio plana puta koji želi mijenjati; putnik selektira segment puta.
 - sustav pita putnika za novu odlaznu i dolaznu destinaciju; putnik daje traženu informaciju.
 - ako je let moguć, tada ...
 - ...
 - sustav prikazuje sažetak transakcije.
- **Alternativni tijek transakcija**
 - ako let nije moguć, tada ...

Primjer: sekvencijski dij. promjene rute leta

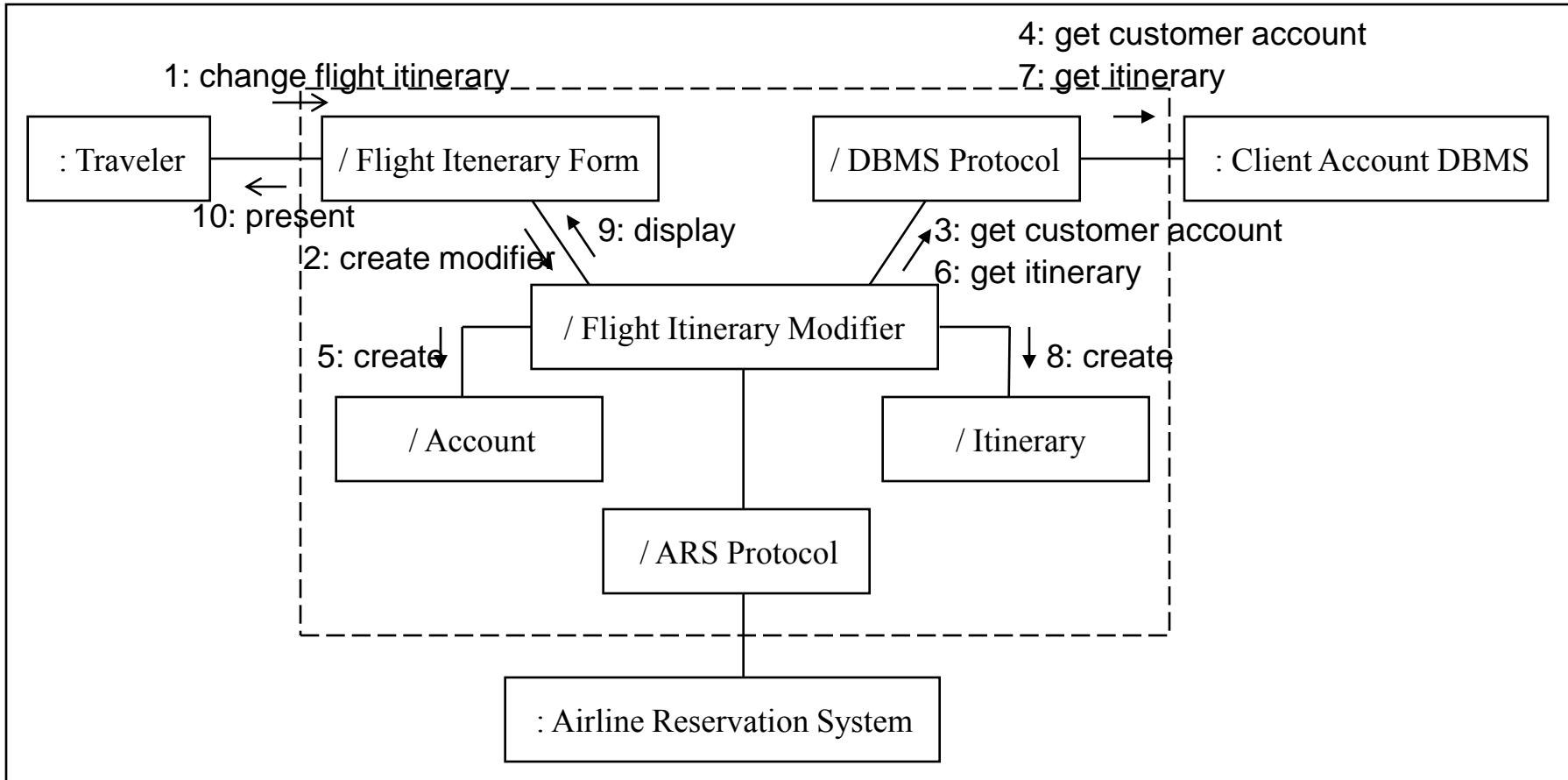


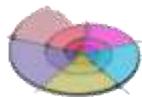
Promjena rute leta – komunikacijski dijagram





Promjena rute leta – komunikacijski dijagram



Primjer: Izdavanje gotovine

1.1: from := readMenuChoice(
 "Account to withdraw from",
 availableAccounts menu)

[while not valid amount] 1.2:
 amount := amountValues [
 readMenuChoice(
 "Amount to withdraw",
 withdrawal amounts menu)]

:CustomerConsole

1: message = getSpecificsFromCustomer()
2: receipt = completeTransaction()

« self »

1.3: validAmount :=
 checkCashOnHand(amount)

2.1: dispenseCash(amount)

:CashDispenser

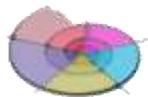
:Withdrawal

1.4 « create »

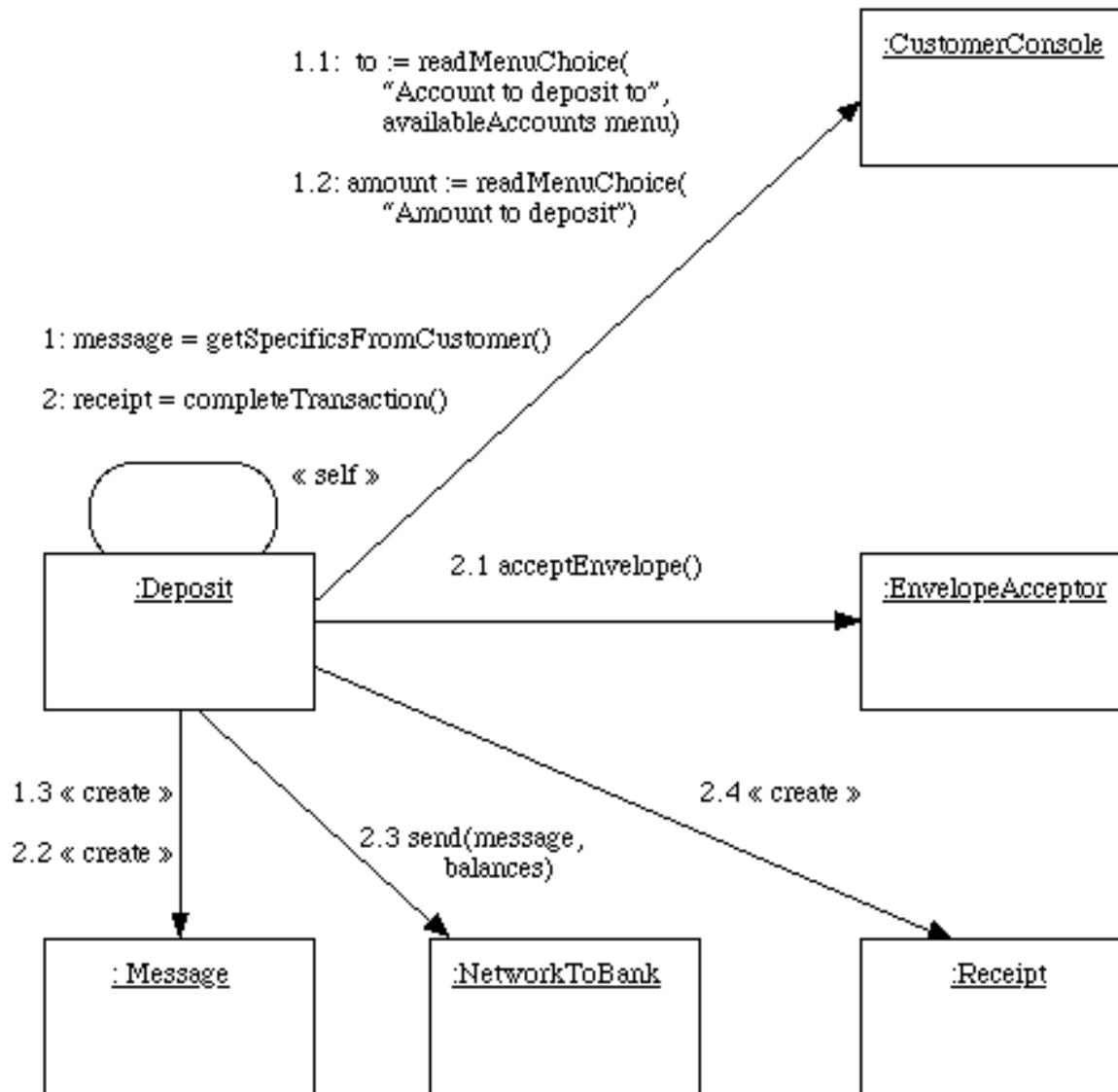
2.2 « create »

:Message

:Receipt



Primjer: polog na račun



Preporuke

- Odaberite ime koje prikazuje namjenu
- Pri crtanju minimizirajte presijecanje linija
- Upotrijebite samo nužne elemente
- Važne elemente smjestite u centar dijagrama
- Započnite s porukom koja započinje interakciju i nastavite sistematično
- Jedan dijagram interakcije prikazuje samo jedan upravljački tok!

UML dijagrami

- Dijagram obrazaca uporabe
- Sekvencijski dijagram
- Komunikacijski dijagram
- ***Dijagram stanja***
- Dijagram aktivnosti
- Dijagram komponenti
- Dijagram razmještaja
- Dijagram paketa
- Dijagram pregleda interakcije
- Vremenski dijagram
- Dijagram profila
- Dijagram razreda
- ***Dijagram objekata***
- Dijagram složene strukture

Dijagrami stanja

- Nedostatak standardnih dijagrama stanja
 - velik broj stanja složenijih sustava, nepreglednost
- 1987. Harel, D.: Statecharts: A visual formalism for complex systems.
 - vizualni formalizam za opis stanja i prijelaza s naglaskom na modularnost, grupiranje, ortogonalnost, konkurentnost i poboljšanja
 - modeliranje jednog reaktivnog objekta
 - osnova UML dijagrama stanja s modifikacijom semantike i terminologije

Dijagrami stanja

- engl. *State Machine Diagram*
- Opisuje dinamičko ponašanje jednog objekta u vremenu
 - pogodno za opis značajnijeg dinamičkog ponašanja objekta
 - objekt se promatra izolirano od ostalih
 - izlaz ne ovisi samo o trenutnim ulazima nego i o povijesti
 - pogodno za opis diskretnog ponašanja (engl. *discrete-event*)
- Prikazuje sekvencu **stanja objekta** te **prijelaze** iz jednog stanja u drugo temeljene na **događajima**
- Stanje objekta
 - opis stanja (okolnosti) u kojem se objekt nalazi kada zadovoljava određene uvjete
 - vrijednosti jednog ili više atributa objekta
 - u jednom stanju objekt može obavljati tri grupe aktivnosti:
 - **do** – aktivnosti koje se izvode za vrijeme dok je objekt u tom stanju
 - **entry** – akcije koje se izvedu pri ulasku u stanje
 - **exit** – akcije koje se izvedu pri izlasku iz stanja

Dijagrami stanja

- početno stanje – jedno
- krajnje stanje - može imati više njih
- Prijelaz
 - sve dozvoljene promjene stanja iz trenutnog u novo
 - novo stanje može biti to isto stanje
 - inicirani su događajima i uvjetima
 - **[uvjet] događaj/akcija**
 - događaji:
 - interakcija
 - asinkroni prijem signala - primljene poruke – *engl. signal*
 - sinkroni poziv objekta – *engl. call*
 - vremenski – *engl. time*
 - proteklo vrijeme – istek vremenskog intervala
 - apsolutno vrijeme – def. *trenutak, takt ...*
 - Ispunjeni uvjeti – *engl. change*
 - **mogu prenositi parametre!**
 - trajanje izvođenja prijelaza je 0 i ne može se prekinuti

Aktivnost i akcije

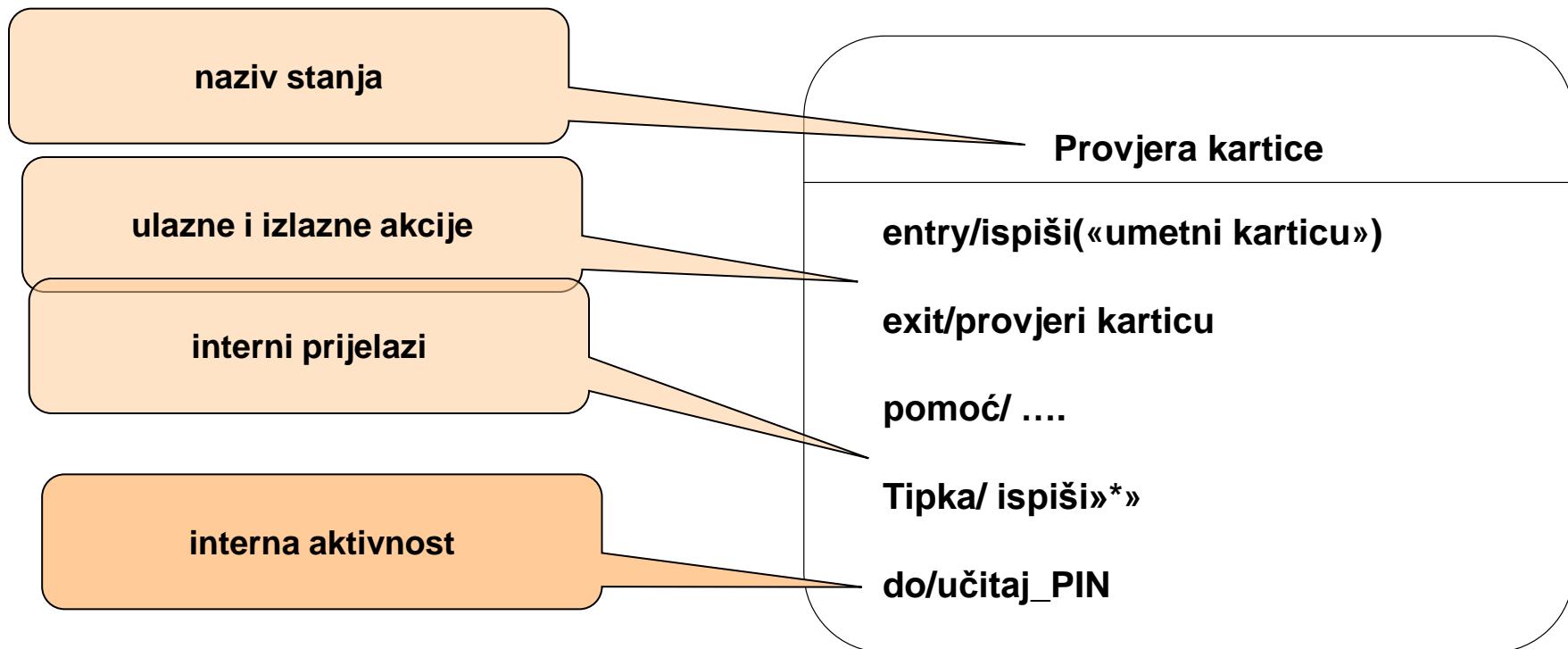
- Akcija – engl. *Action*
 - kratkotrajno, neprekidivo ponašanje
- Aktivnost – engl. *Activity*
 - obavlja se sve dok je stanje aktivno
 - ovisi o dolaznim događajima



Stanje



- Akcija:
 - naziv_događaja/Akcija
- Aktivnost
 - do/Aktivnost

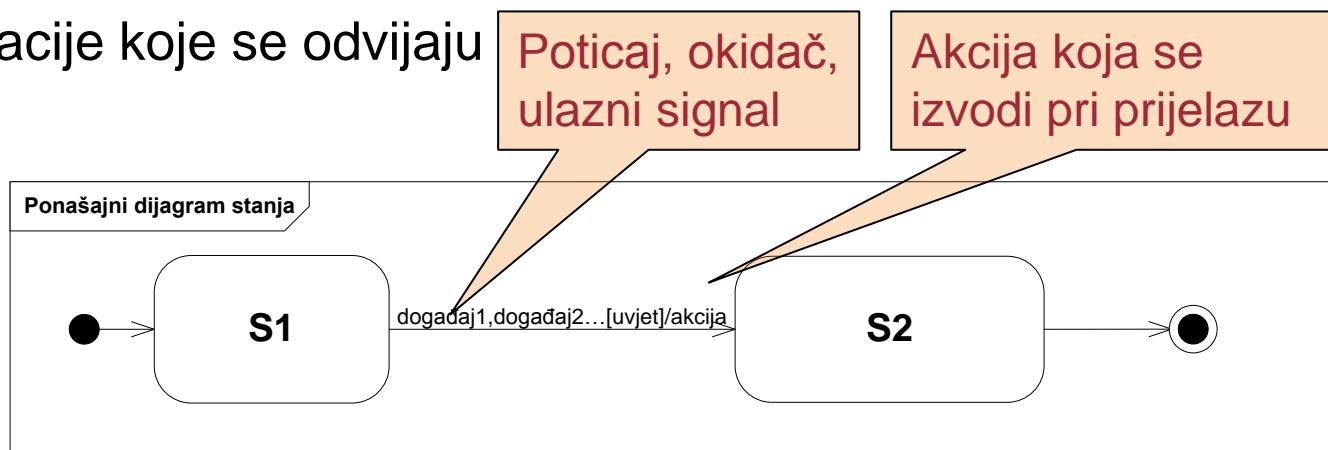




Prijelazi



- Ponašajni dijagrami - *engl. behavioral state diagrams*
- Događaji
 - vanjska ili unutarnja pojavljivanja događaja koja pokreću prijelaz
- Uvjet
 - izraz koji kada je ispunjen(istinit) dopušta odvijanje prijelaza (naravno uz pojavu događaja)
- Akcija
 - operacije koje se odvijaju

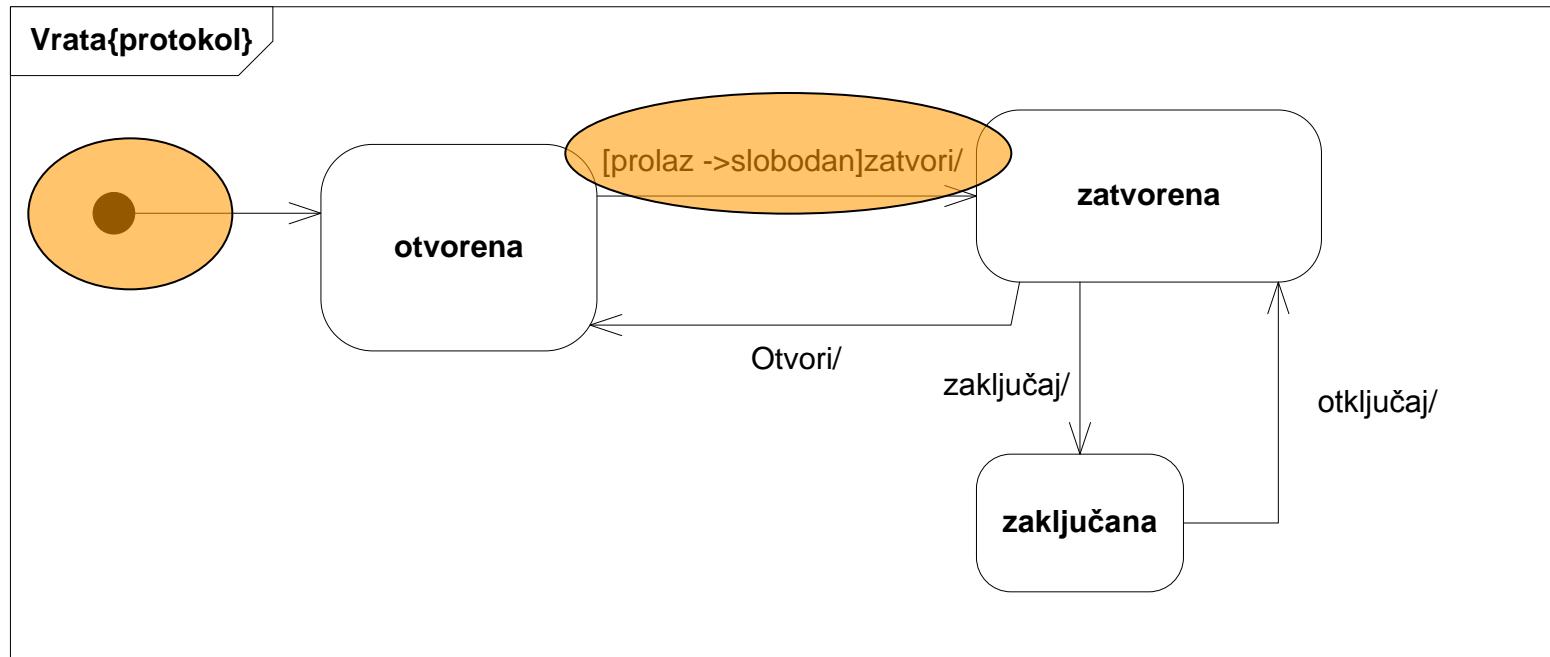




Primjer

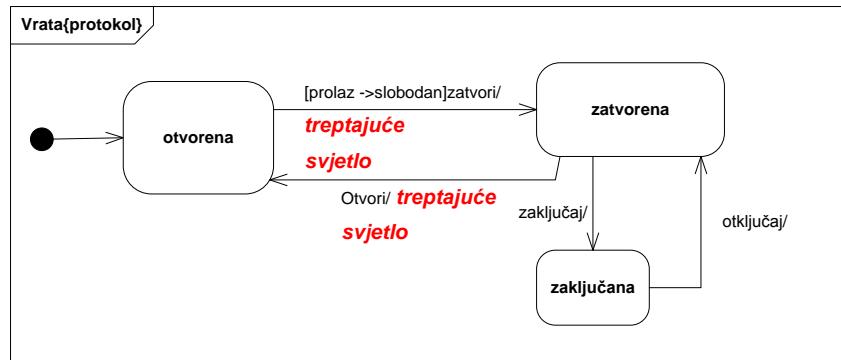


■ Grafički prikaz ponašanja ulaznih vrata

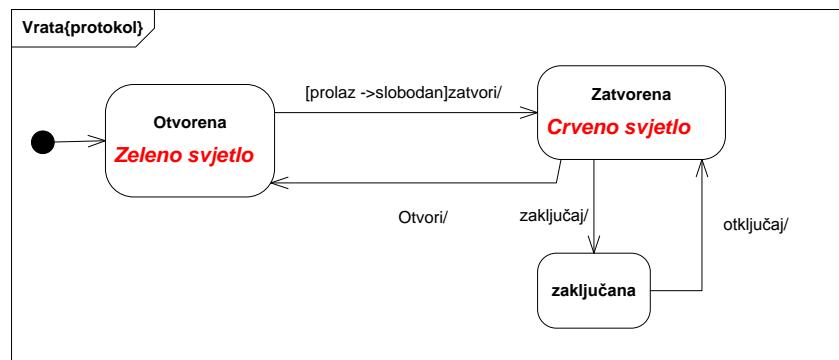


Promjena stanja i akcije

- Automati s izlazom - može generirati akciju
 - mealyev automat - izlaz je funkcija ulaza i trenutnog stanja

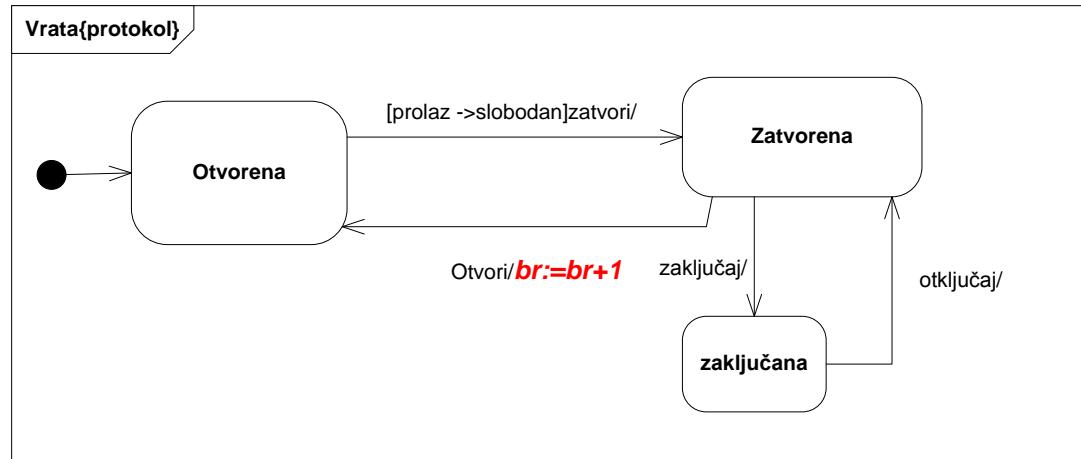


- mooreov automat - izlaz je funkcija trenutnog stanja



Proširenje stanja

Dodatne varijable stanja



Definiranje automata

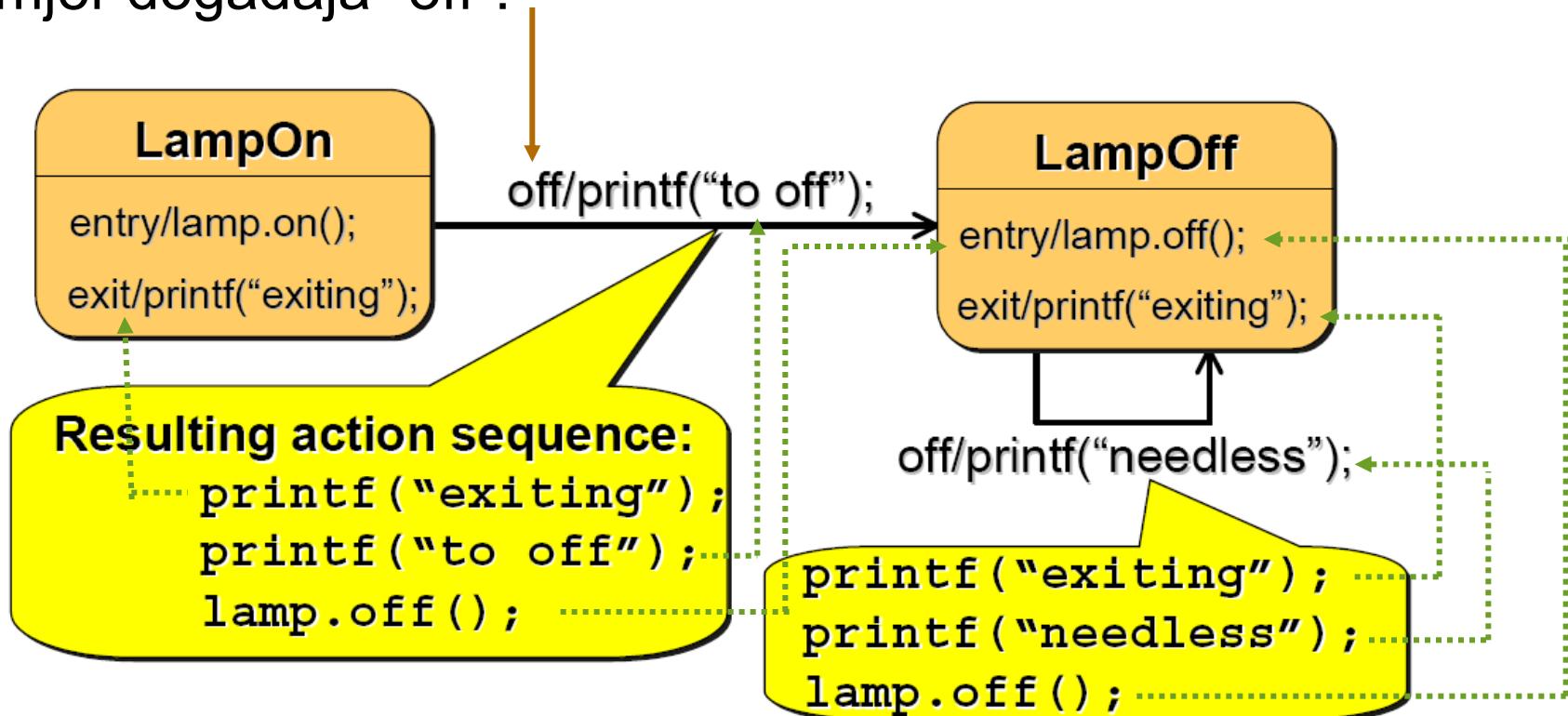
- konačan skup ulaznih signala
- konačan skup izlaznih signala
- konačan skup stanja
- skup prijelaza
 - Signali okidanja
 - Akcije koje se izvode pri prijelazu
- konačan skup proširenih varijabli stanja
- početno stanje
- skup konačnih stanja



Dijagrami stanja – slijed akcija



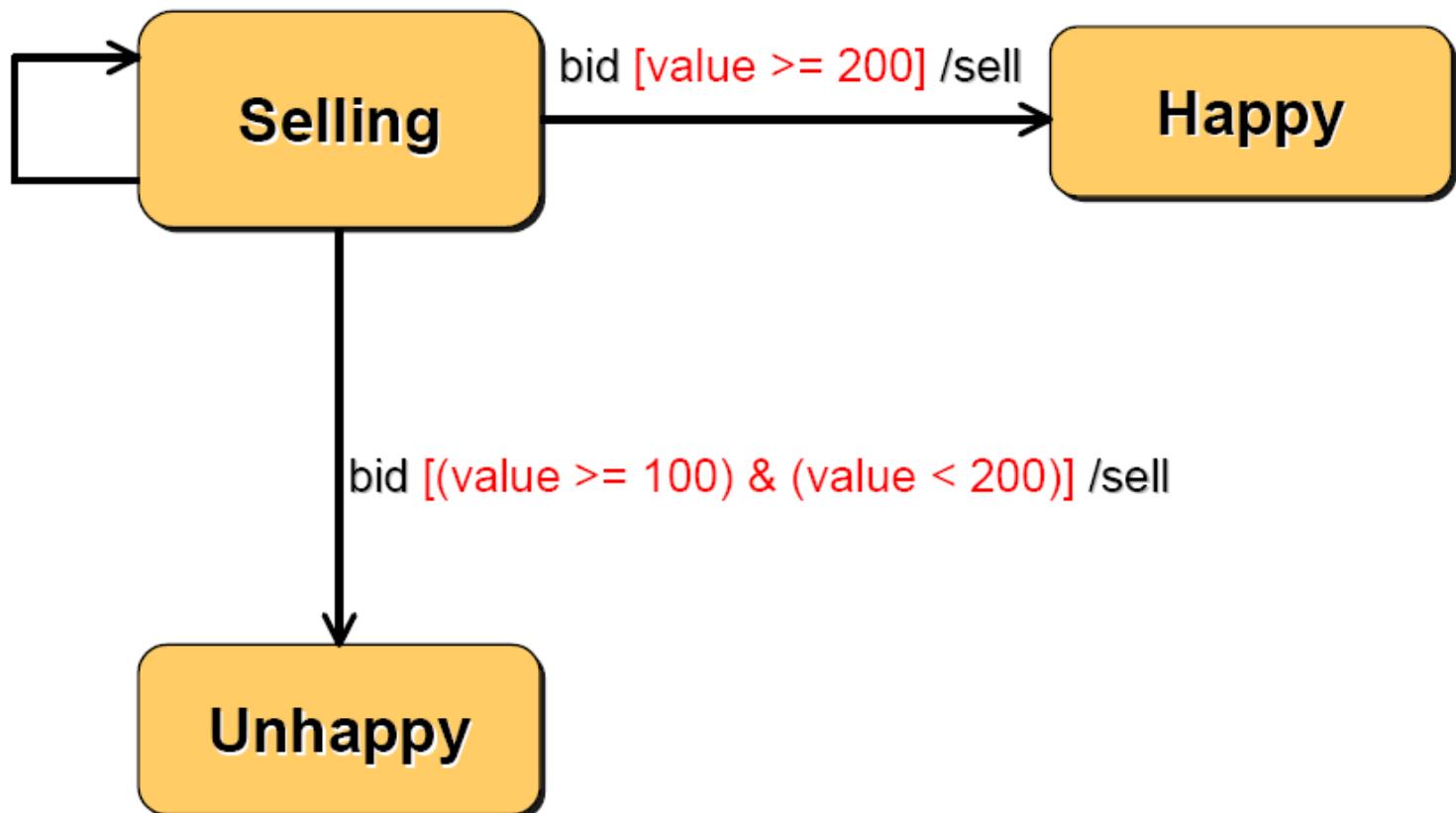
- Akcije pri izlazu iz stanja (izlazne akcije) izvode se prije akcije prijelaza.
- Akcije pri ulazu u stanje (ulazne akcije) izvode se nakon akcija prijelaza.
- Primjer događaja “off”:



Uvjeti

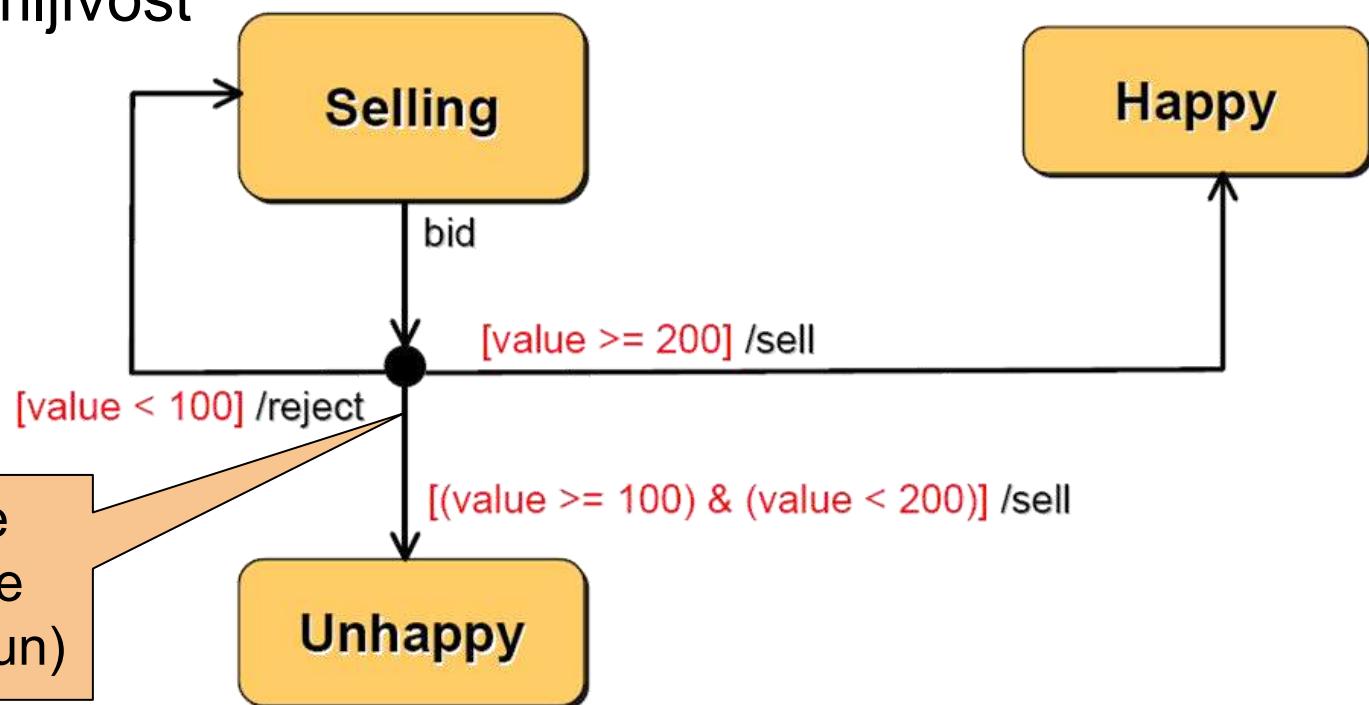
- Uvjetno izvođenje prijelaza
- Npr: prodaja dionica kada je ispunjen uvjet

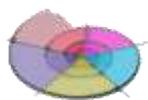
bid [value < 100] /reject



Uvjetna grananja

- Statičko uvjetovanje grananja
 - grafički prikaz odlučivanja
 - uvjeti su poznati prije grananja
- Pojednostavljivanje grafičkog prikaza
 - veća razumljivost

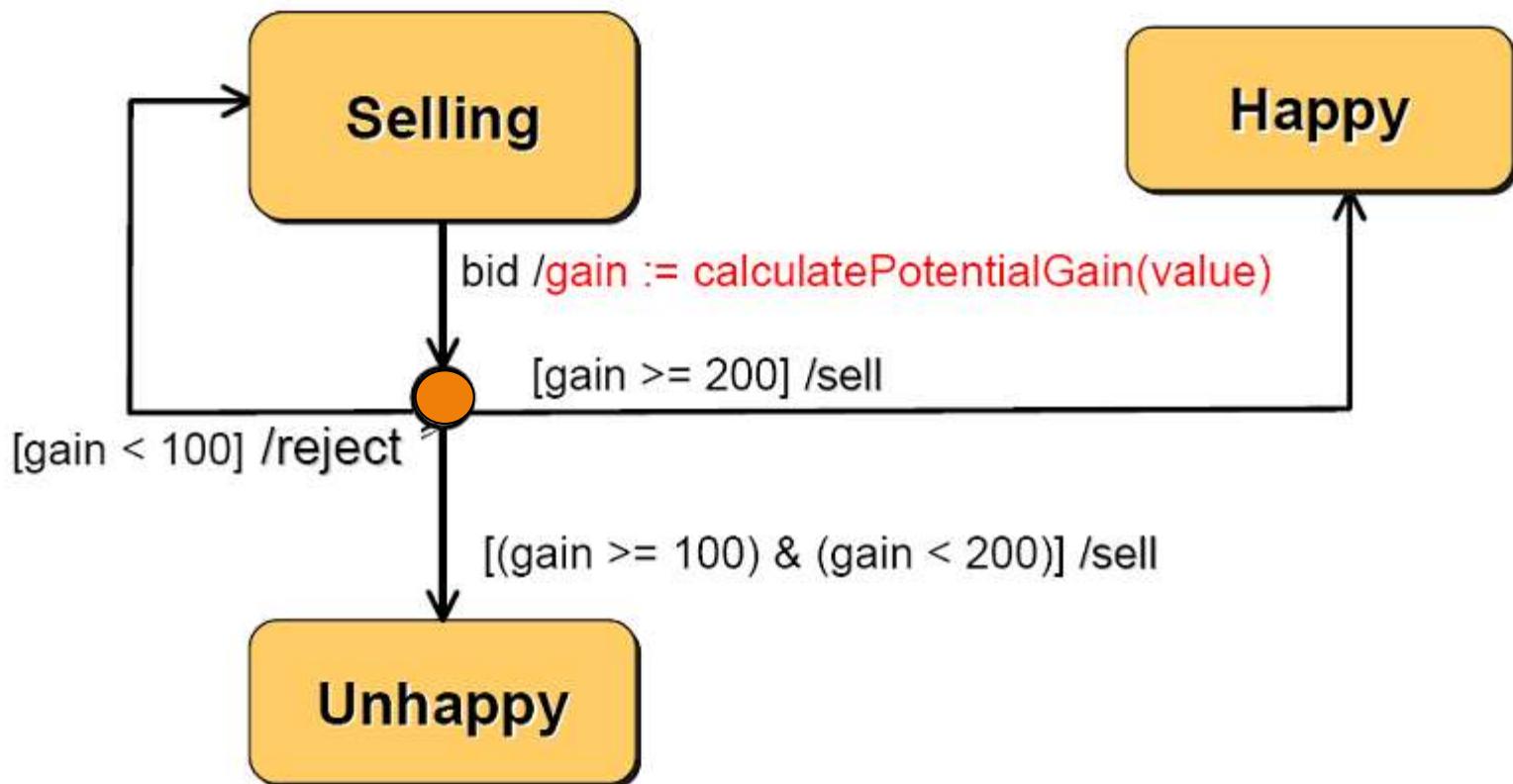




Dinamička uvjetna grananja



- Dinamičko uvjetovanje grananja
 - uvjeti se izračunavaju pri izlasku iz stanja.
 - Nisu poznati unaprijed





Pseudostanja UML dijagrama

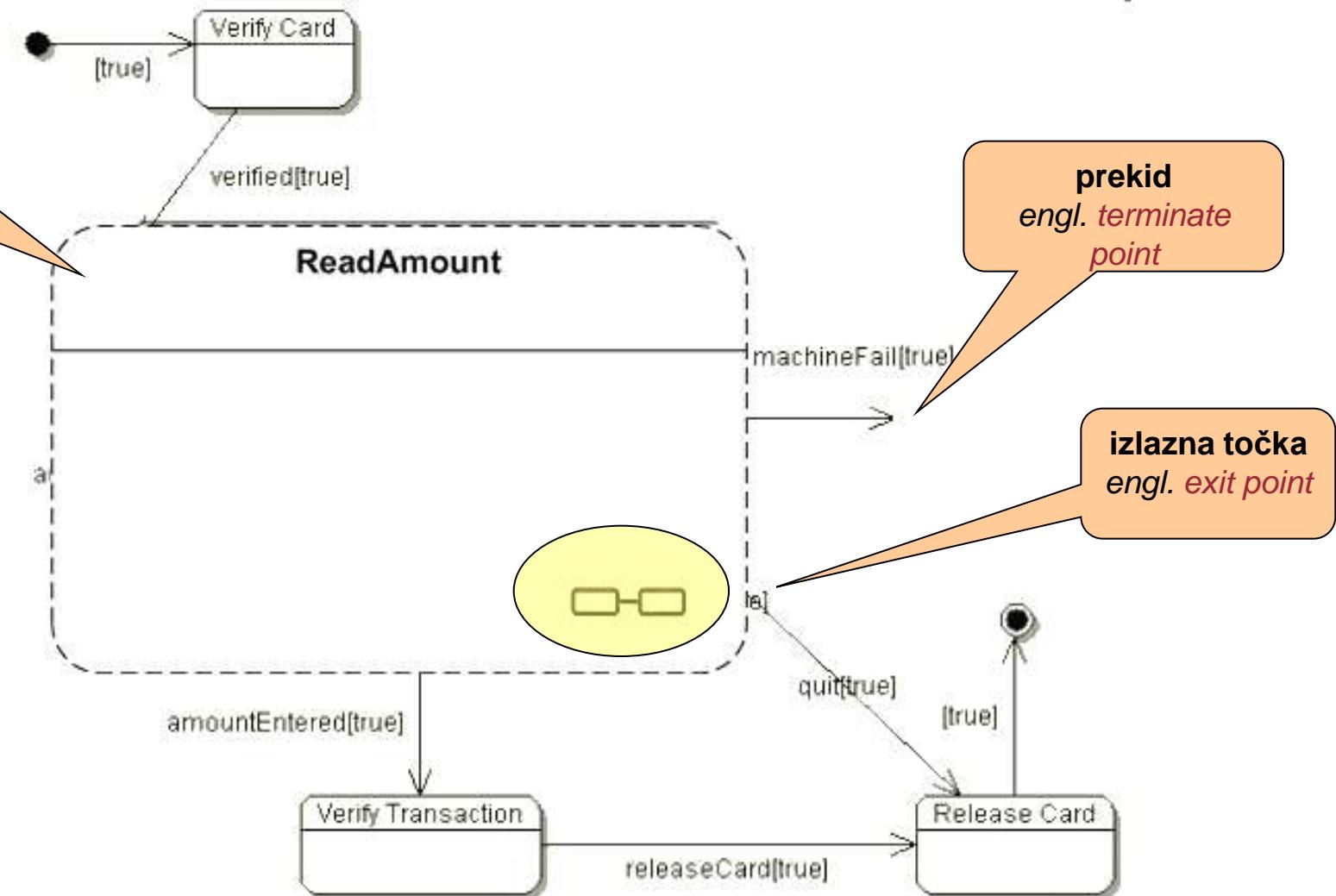


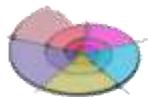
- vrsta stanja u UML metamodelu koje predstavlja točku prijelaza unutar dijagrama stanja

	Početno stanje - <i>Initial State</i>
	Završno stanje(a) - <i>Final State</i>
	Povijest - <i>History</i>
	Duboka povijest – <i>Deep History</i>
	Ulazna točka - <i>Entry Point</i>
	Izlazna točka - <i>Exit Point</i>
	Spajanje - <i>Junction Pseudo-State</i>
	Izbor - <i>Choice Pseudo-State</i>
	Završetak - <i>Terminate Pseudo-State</i>
	Grananje/Račvanje - <i>Fork</i>
	Spajanje/skupljanje - <i>Join</i>

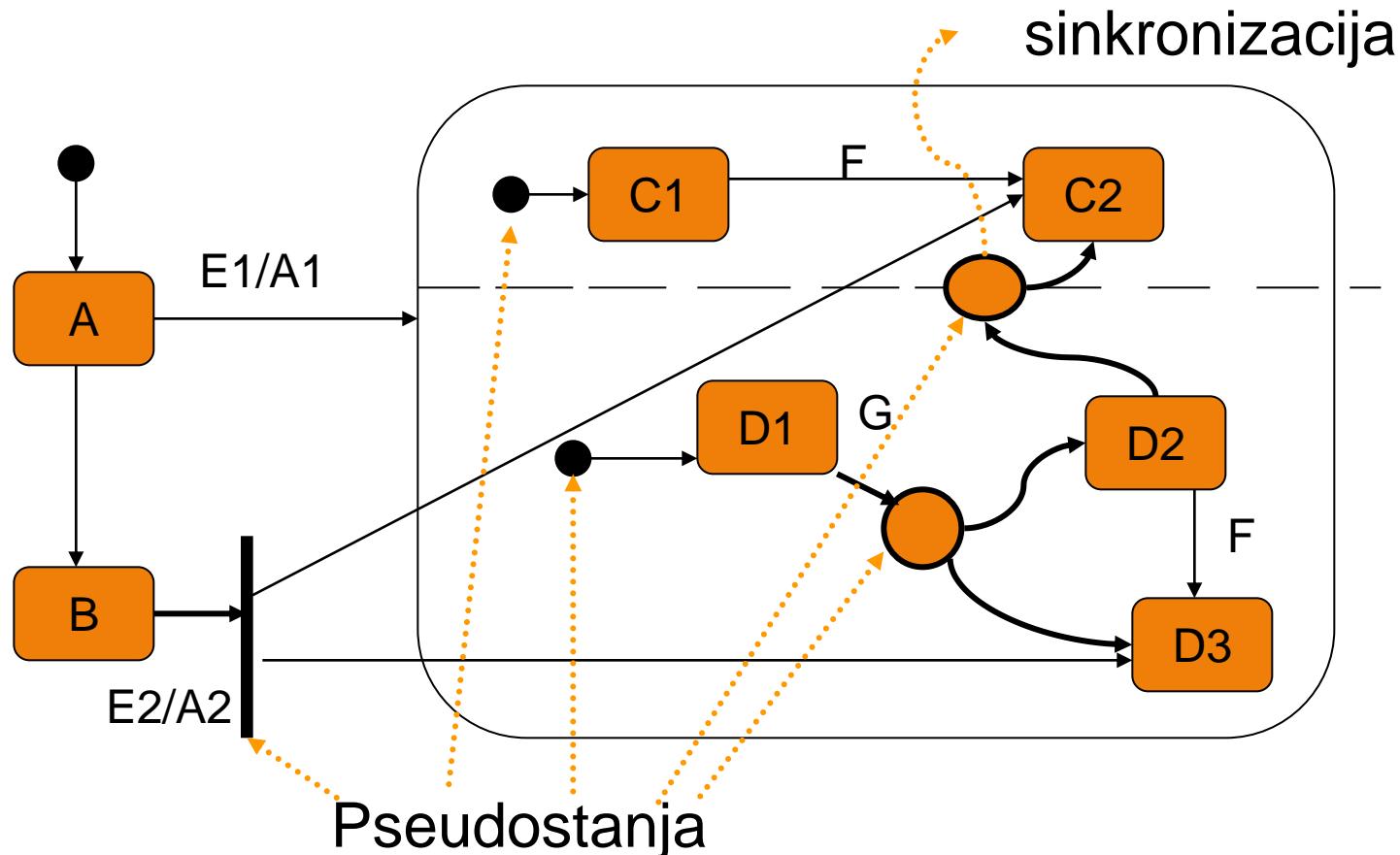
Hijerarhija stanja

- Olakšava prikaz složenih problema





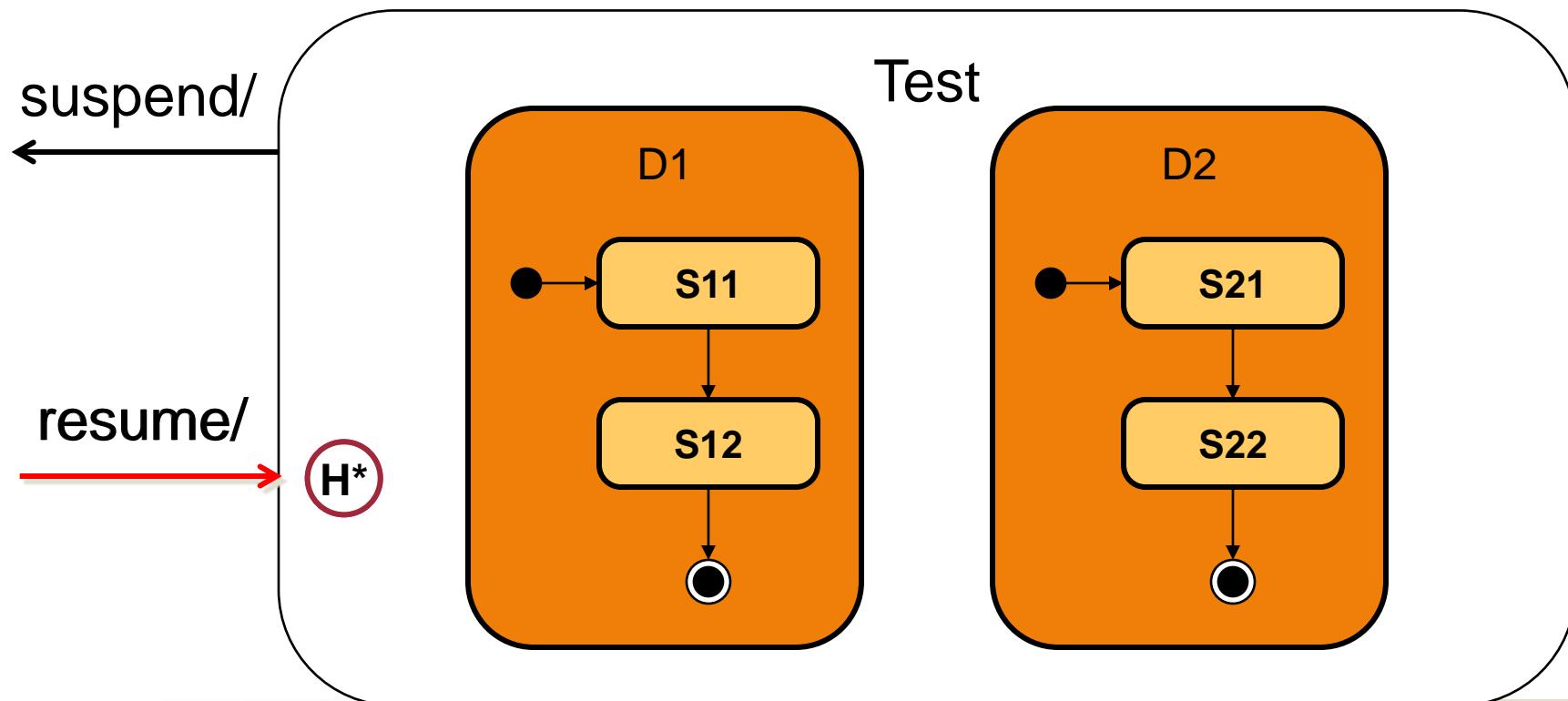
Primjer pseudostanja



Povijest

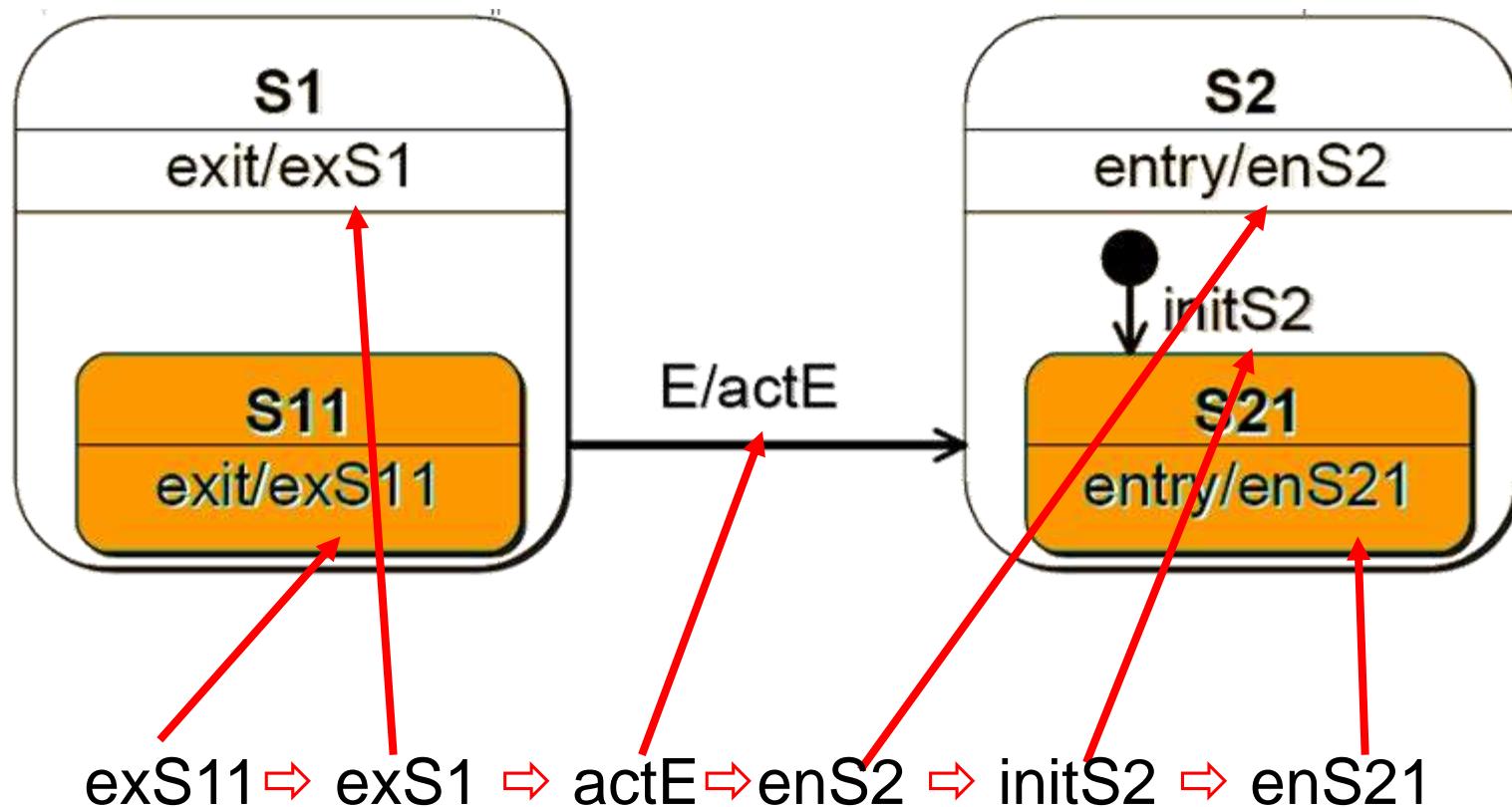
Mogućnost povratka na prethodno stanje

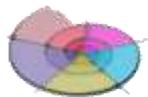
- povijest – engl. *Shallow History (H)*
 - povratak na posljednje stanje na istoj razini
- duboka povijest - engl. *Deep History (H*)*
 - povratak na posljednje stanje (bez obzira na kojoj razini)



Prioritet akcija

- Ulag: izvana prema unutra
- Izlaz: iznutra prema van

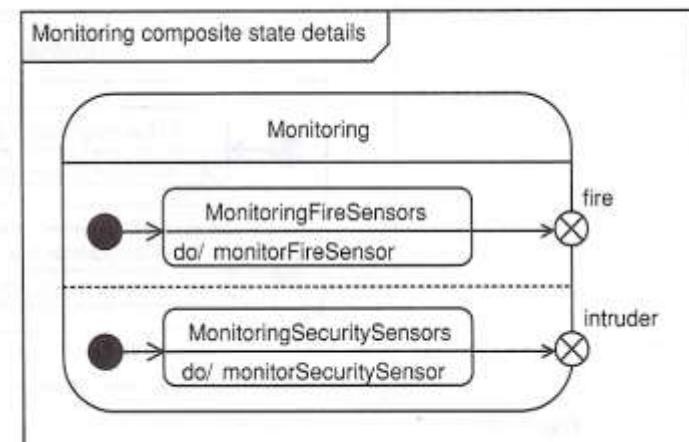
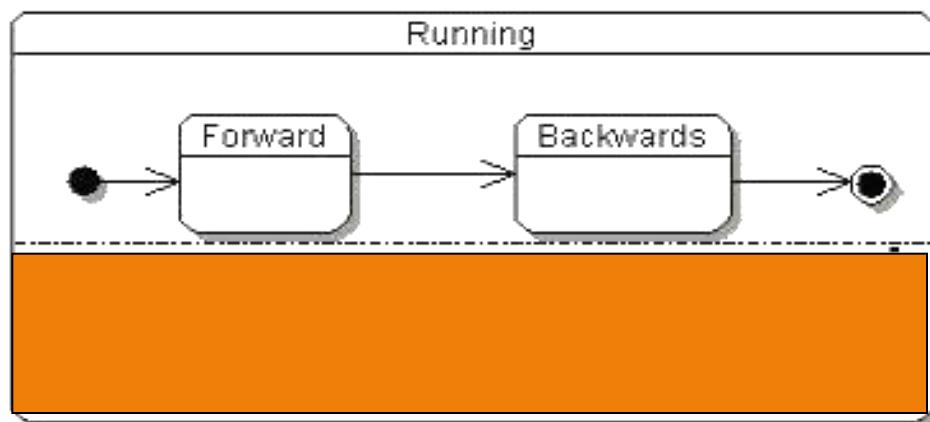




Konkurentna podstanja



- Jedno stanje može imati više konkurentnih podstanja
 - višestruka perspektiva jednog objekta
 - smanjuje broj stanja





Paralelna stanja - Ortogonalnost



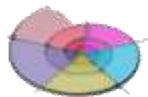
- Višestrukva perspektiva istog objekta *engl. orthogonality*

- paralelna stanja

- ~~Istodobno reagiraju na iste događaje~~



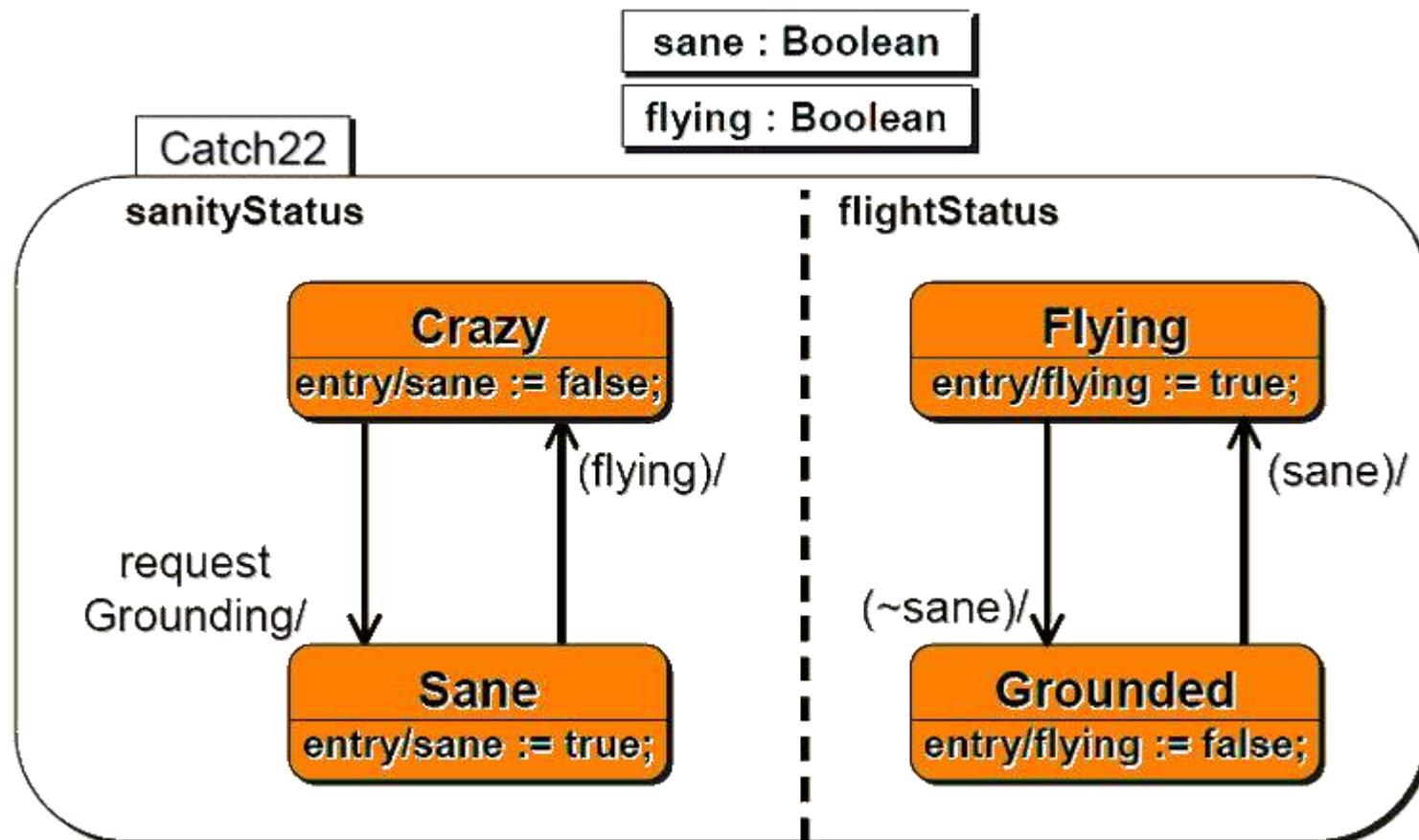
- Ne modelirati neovisne objekte primjenom ortogonalnih regija!

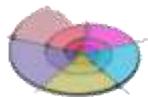


Interakcija između područja



- Uporabom dijeljenih varijabli

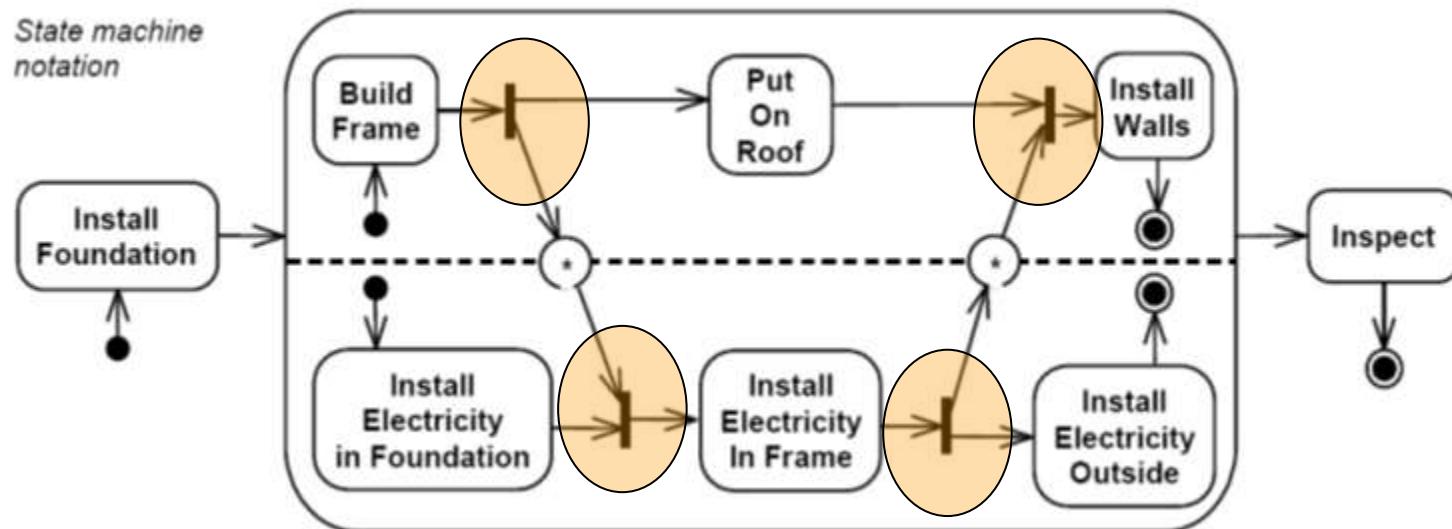




Račvanje i skupljanje

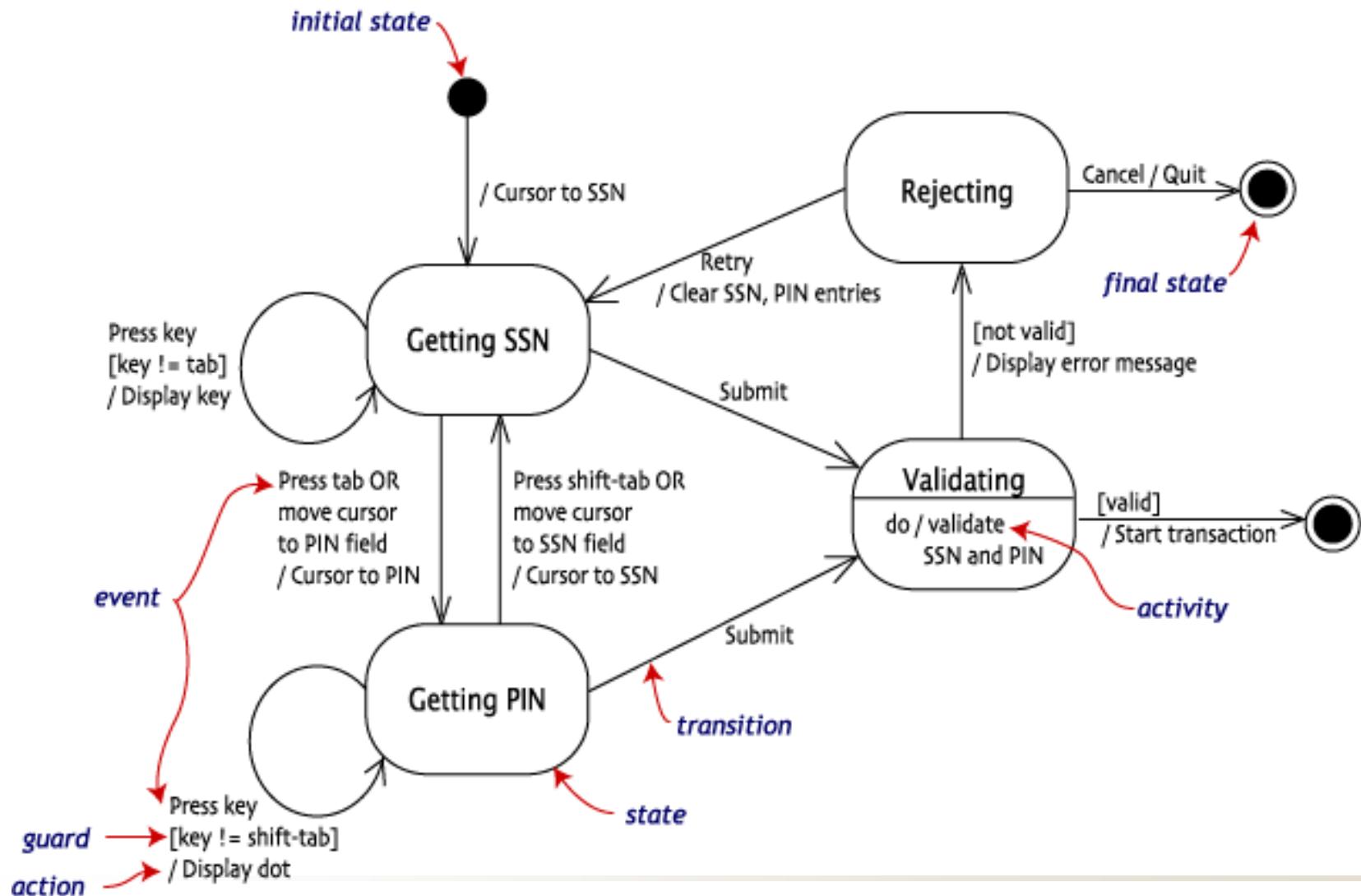


- Prikaz račvanja i skupljanja paralelnih prijelaza koristi se za opis prijelaza u i iz ortogonalnih paralelnih regija.

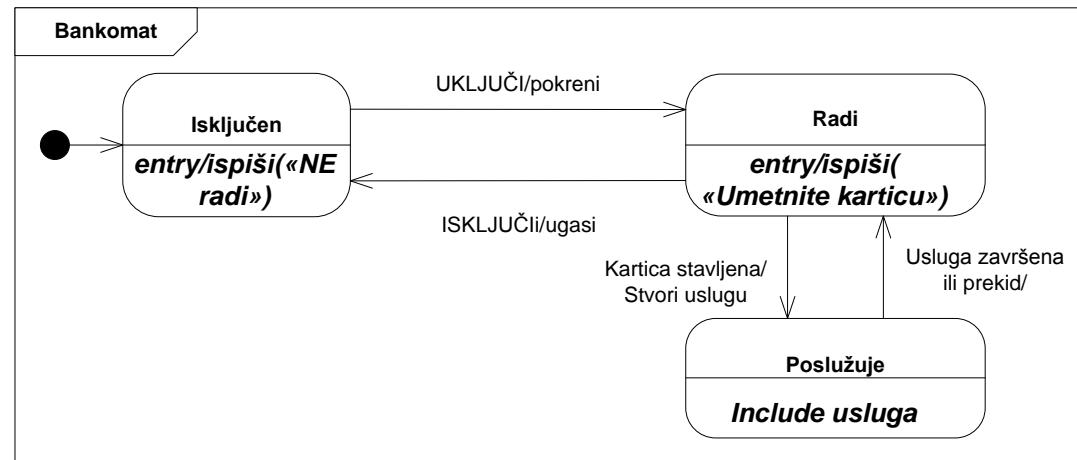


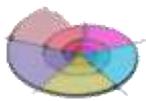
Primjer: Prijava na bankarski sustav

Prijava: unos OIB-a i PIN-a



Primjer: Dijagram stanja bankomata





Dijagrami stanja – sažetak



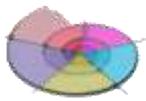
- Pogodni za modeliranje događajima poticanog ponašanja sustava (*engl. event driven behavior*).
 - u objektno usmjerenoj arhitekturi - opis ponašanja aktivnih objekata.
 - Sustavi se prikazuju kao mreža međusobnog djelovanja (kolaboracije) strojeva stanja.
- U okviru UML jezika dijagrami stanja sadrže napredne značajke:
 - hijerarhijsko modeliranje stanja.
 - modeliranje paralelnih (ortogonalnih) stanja.
 - aktivnosti u stanjima.
 - akciju pri ulasku/izlasku iz stanja.
 - statičko i dinamičko uvjetno grananje.

UML dijagrami

- Dijagram obrazaca uporabe
- Sekvencijski dijagram
- Komunikacijski dijagram
- Dijagram stanja
- ***Dijagram aktivnosti***
- Dijagram komponenti
- Dijagram razmještaja
- Dijagram paketa
- Dijagram pregleda interakcije
- Vremenski dijagram
- Dijagram profila
- Dijagram razreda
- **Dijagram objekata**
- Dijagram složene strukture

Dijagrami aktivnosti

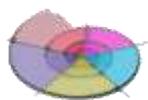
- engl. *Activity diagrams*:
- U prvoj inačici UML-a predstavljali su samo specifičan prikaz dijagrama stanja
- UML 2 definira novu semantiku zasnovanu na Petrijevim mrežama
 - povećana prilagodljivost modeliranju različitih tipova tijeka
 - jasno razdvajanje od dijagrama stanja
 - usmjereni na opis tijeka izvođenja i ponašanja sustava
- Namjena pobliže opisivanje obrazaca uporabe, dijagrama razreda, komponenti, sučelja i operacija
- Pogodni za:
 - prikaz proceduralnog tijeka procesa
 - modeliranje poslovnih zahtjeva (procesa)
 - prikaz paralelnosti



Primjena dijagrama aktivnosti



- Primjenjuju se za opis modela toka upravljanja (*engl. control flow*) ili toka podataka (objekata – engl. *object flow*).
- Ne primjenjuju se za modeliranje događajima poticanog ponašanja.
- Tipična primjena: opis poslovnog modela u kojem želimo modelirati tijek zadataka i poslova.
- Temeljna razlika između dijagrama aktivnosti i komunikacijskih dijagrama te dijagrama stanja je način iniciranja pojedinog koraka a posebice kako koraci dobivaju ulazni signal ili podatke (u komunikacijskom dijagramu to su poruke, u dijagramu stanja to su događaji).
- U modeliranju toka upravljanja svaki novi korak poduzima se nakon završenog prethodnog (neovisno da li su ulazi dostupni, ispravni ili potpuni u tom času).
 - tzv. “pull” način djelovanja (povlačenje).
- U modeliranju toka podataka (objekata) sljedeći korak se poduzima kada su svi ulazni podaci dostupni.
 - tzv. “push” način djelovanja (guranje).
- Dijagrami aktivnosti vrlo su slični dijagramima stanja, ali s drugačijom semantikom elemenata s kojima se modelira.



Primjena dijagrama aktivnosti



- Za opis upravljačkog i podatkovnog toka
- Analiza
 - oblikovanje tijeka obrazaca uporabe
 - oblikovanje tijeka između različitih obrazaca uporabe
- Oblikovanje
 - detalji operacija
 - detalji algoritama
- Modeliranje poslovnih procesa



Elementi dijagrama aktivnosti



- Prilagođeni opisu tijeka upravljanja i podataka
- Čvorovi – *engl. Activity nodes, nodes:*
 - čvor akcije – *engl. Action nodes*
 - nedjeljive aktivnosti
 - upravljački čvorovi – *engl. Control nodes*
 - upravljanje tijekom
 - objekti – *engl. Object nodes*
 - predstavljaju objekte u aktivnostima
- Veze – *engl. Activity edges*
 - upravljački tijek – *engl. Control flows*
 - predstavlja tijek upravljanja aktivnostima
 - tijek objekta – *engl. Object flows*
 - tijek objekta kroz aktivnosti

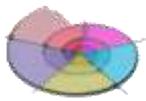
Aktivnosti

- Aktivnost – engl. *Activity*
 - obuhvaća više čvorova i veza koji predstavljaju odgovarajući slijed zadataka
- Akcija – engl. *Action*
 - kratkotrajno, neprekidivo ponašanje unutar čvora akcije
- Za sve aktivnosti i akcije mogu biti definirani odgovarajući uvjeti prije izvođenja (engl. *preconditions*) i nakon izvođenja (engl. *postconditions*)
- Značke – engl. *Tokens*
 - dio semantike bez grafičkog prikaza
 - značka može predstavljati:
 - upravljački tijek
 - objekt
 - podatak
 - kreću se od izvorišta prema odredištu vezama ovisno o:
 - ispunjenim uvjetima izvornog čvora
 - postavljenim uvjetima veza (engl. *Edge guard conditions*)
 - preduvjetima ciljnog čvora

Aktivnost

- Modelira ponašanje kao niz akcija
- Izvođenje se modelira uporabom znački
 - proizvode ih čvorovi, kreću se po vezama

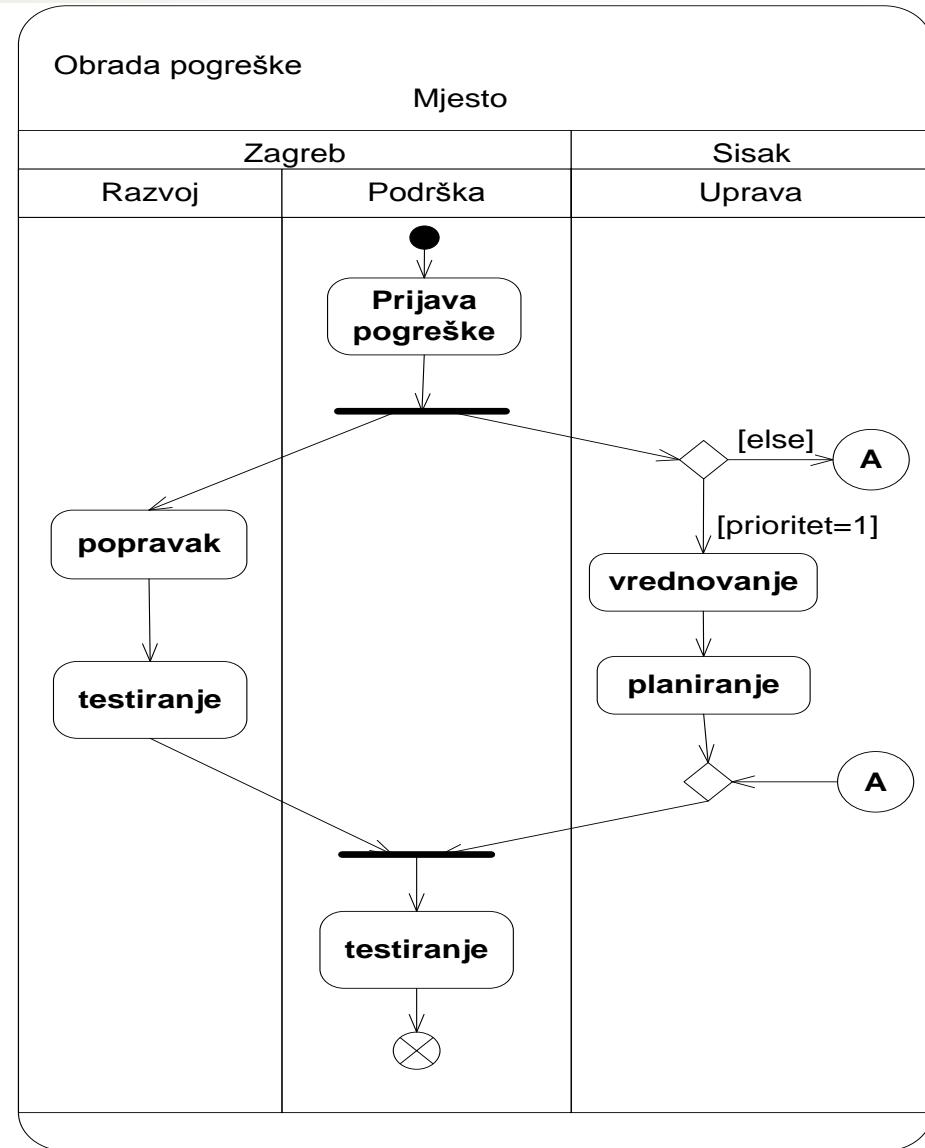




Podjela aktivnosti - particije

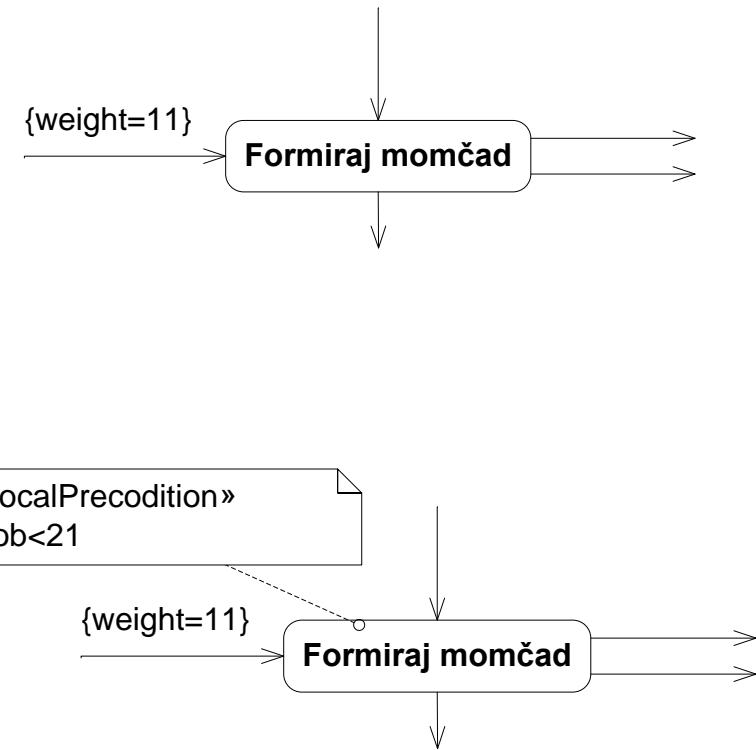


- Dijagram aktivnosti može biti podijeljen u particije (engl. *swim lanes*)
 - horizontalno, vertikalno, proizvoljno
 - hijerarhija
- Ne utječu na tijek odvijanja aktivnosti
 - način grupiranja



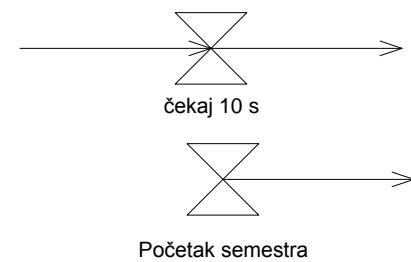
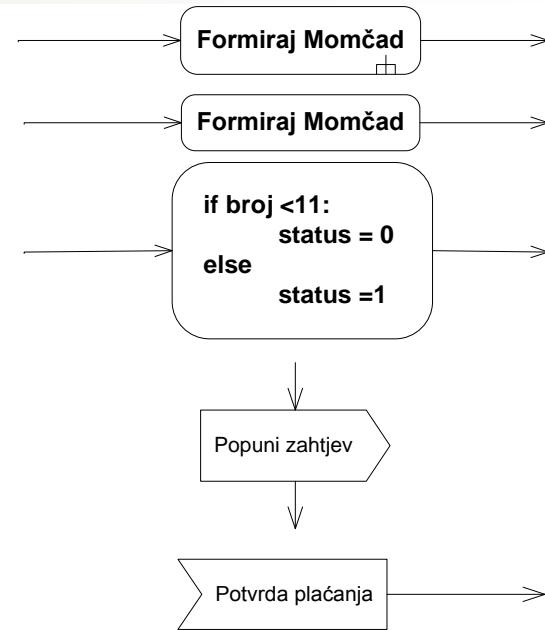
Čvor akcije

- Započinje izvođenje kada:
 - postoji odgovarajući broj znački na svim ulazima
 - zadovoljeni su svi lokalni preduvjeti akcije
- Nakon izvođenja provjerava se zadovoljavanje izlaznih uvjeta te prosljeđuju značke na sve izlaze
- Tipovi:
 - obrade osnovnih operacija, pozivanje složenih aktivnosti ili ponašanja - engl. *call action*
 - komunikacijske akcije
 - slanje signala
 - manipuliranje objektima



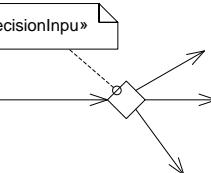
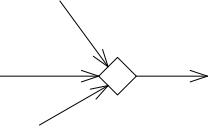
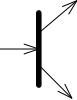
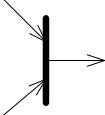
Tipovi čvorova akcije

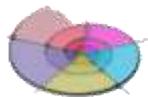
- Obrada osnovnih operacija, pozivanje složenih aktivnosti ili ponašanja - engl. *call action*
- Slanje signala – engl. *send signal*
 - asinkroni signal
- Prihvaćanje događaja – engl. *accept event*
 - čekanje na neki događaj
- Vremenski događaj – engl. *time event*
 - definirana vremenskim izrazom



Upravljački čvorovi

- Rukuju upravljanjem aktivnostima
 - početak i završetak
 - odluke
 - paralelnost

	Početni čvor - Initial node
	Završni čvor aktivnosti – Activity final node
	Završetak toka – Final flow node
	Čvor odluke – Decision node
	Spajanje čvorova – Merge node
	Granjanje toka- Fork node
	Spajanje – Join node Sinkronizira više paraljenih tokova

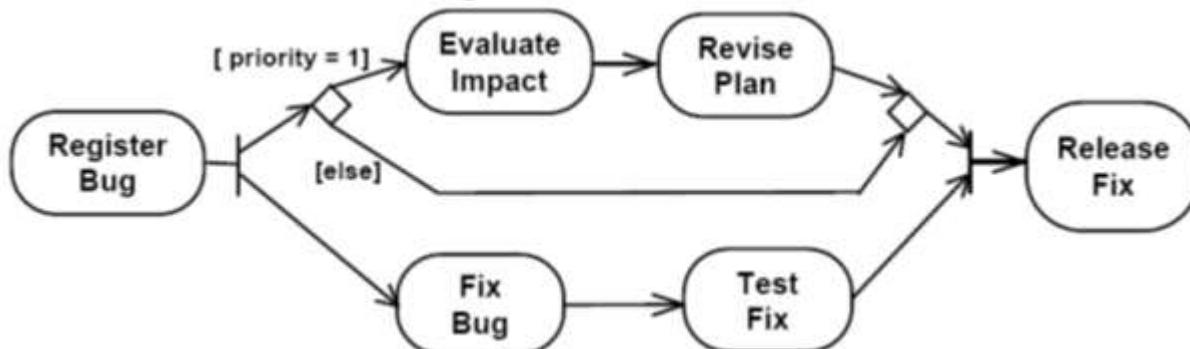


Koordiniranje koraka

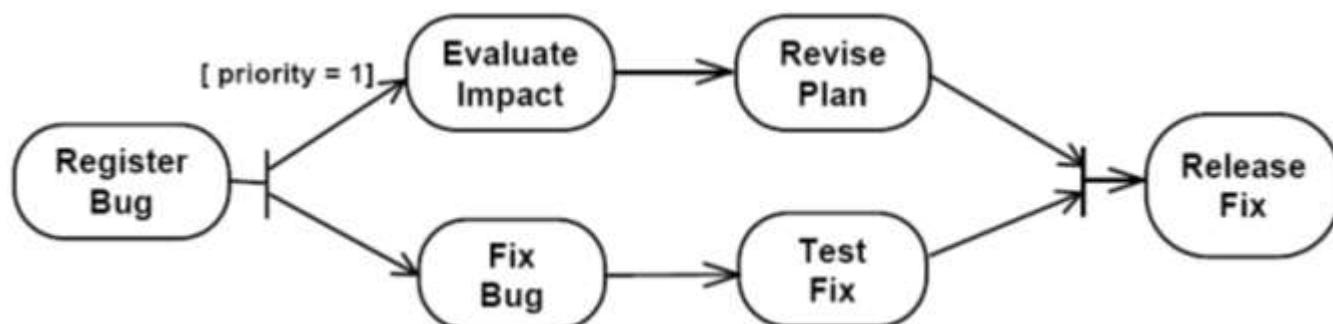


- Uvjetovanje toka aktivnosti može se izvesti na dva načina:

- u točkama odlučivanja



- na račvanju
 - sažetije

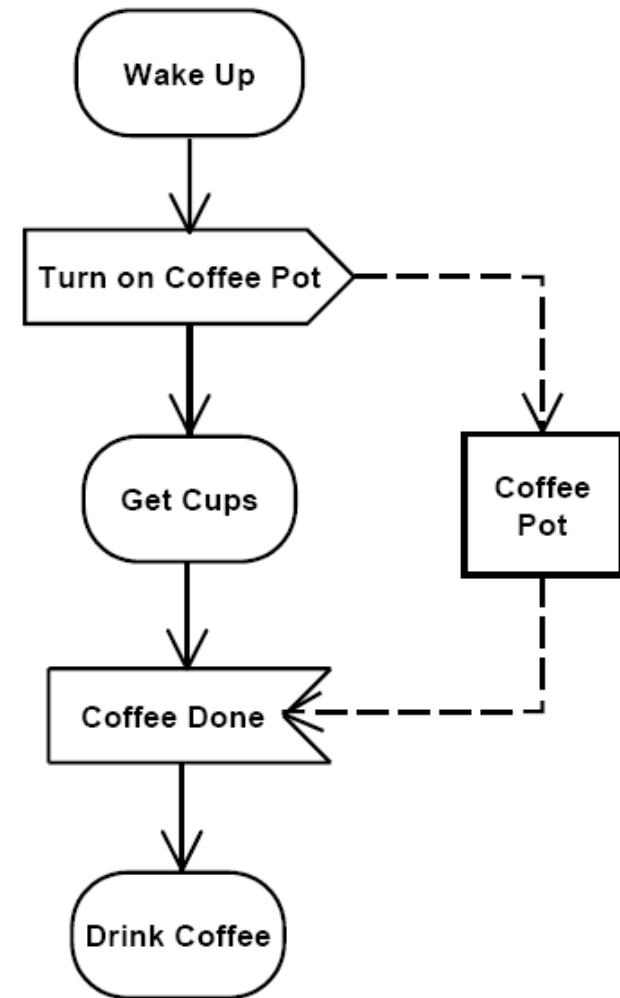




Primjer uporabe signala

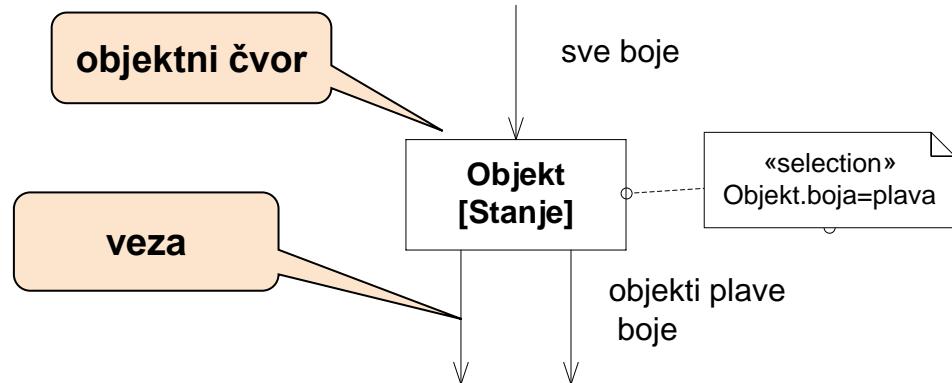


- U dijagramu aktivnosti katkada je potrebno poslati ili primiti signal.
- Pošalji (engl. *send*) signal preslikava se u prijelaz i akciju slanja signala.
- Primi (engl. *receive*) signal preslikava se u stanje čekanja na signal bez akcije.



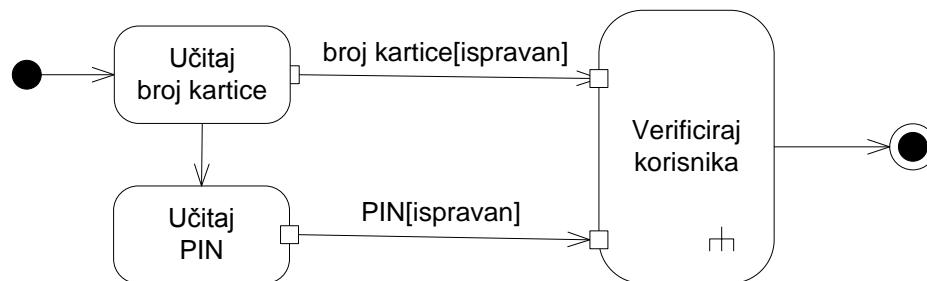
Objektni čvorovi

- Prikazuju podatke i objekte
 - objektni čvor ukazuje na raspoloživost u danoj točki aktivnosti
 - uobičajeno označeni imenom razreda i predstavljaju instancu
- Ulazne i izlazne veze predstavljaju tok objekta
 - opisuju kretanje objekta unutar aktivnosti
- Objekte stvaraju i upotrebljavaju akcijski čvorovi
- Objektne značke
 - kada objektni čvor primi značku, nudi ju na svim izlaznim vezama koje se natječu za značku. Značku dobiva prva veza koja ju je spremna prihvatiti
- Objektni čvorovi se ponašaju kao međuspremniци (*engl. buffer*)
 - pohranjuju objektne značke

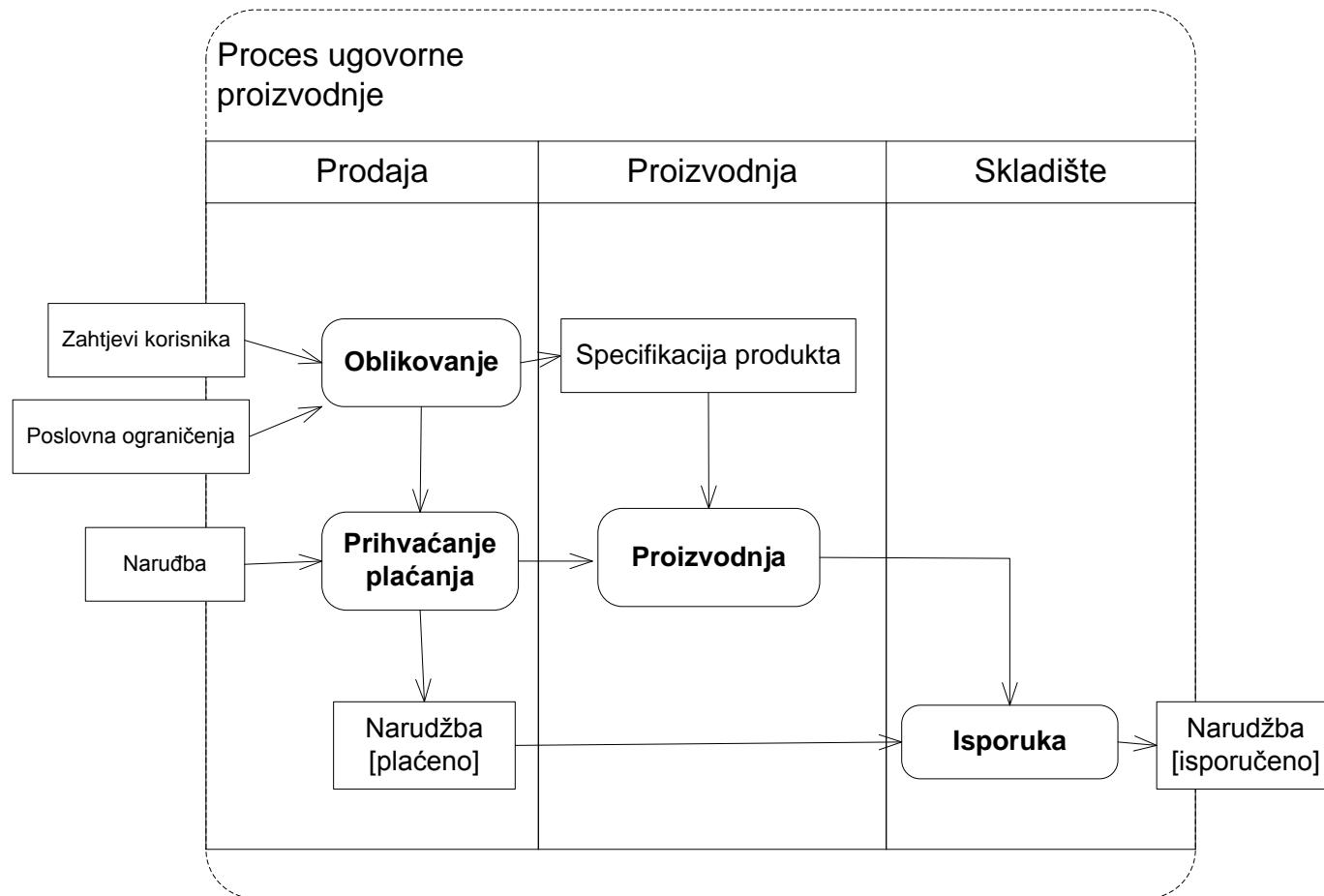


Priključnice

- engl. *Pins*
- Objektni čvor s jednim ulazom ili izlazom prema čvoru akcije
- Pojednostavljaju grafički prikaz dijagrama aktivnosti s većim brojem objektnih čvorova

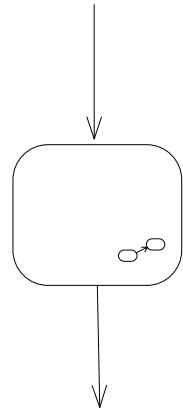


Primjer:



Podaktivnost

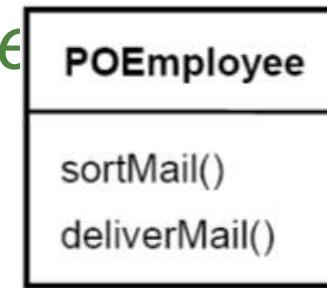
- engl. *Subactivity*
- Namjena funkcionalne dekompozicije
- Ugniježđenje drugog dijagrama aktivnosti
- Ulaskom u podaktivnost pokreće se njegova početna akcija
- Po završetku podaktivnosti, nastavlja se izvođenje prethodne aktivnosti



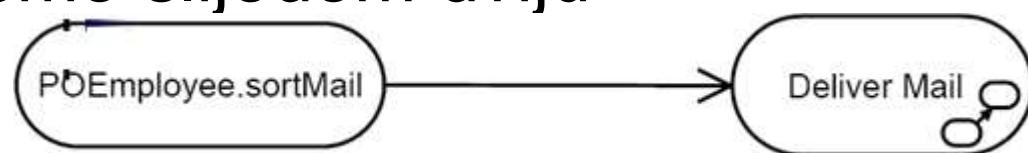


Primjer:

- Primjer: Objekt iz razreda *POEmployee* izvodi operacije *sortMail()* i *deliver()*

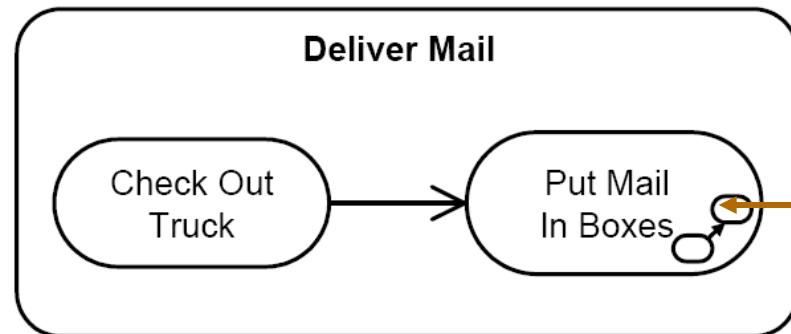


- Operacije opisujemo slijedom dviju aktivnosti:

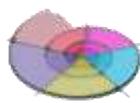


ima
podaktivnosti

- Aktivnost *Deliver Mail* ima podaktivnosti koje se razjašnjava



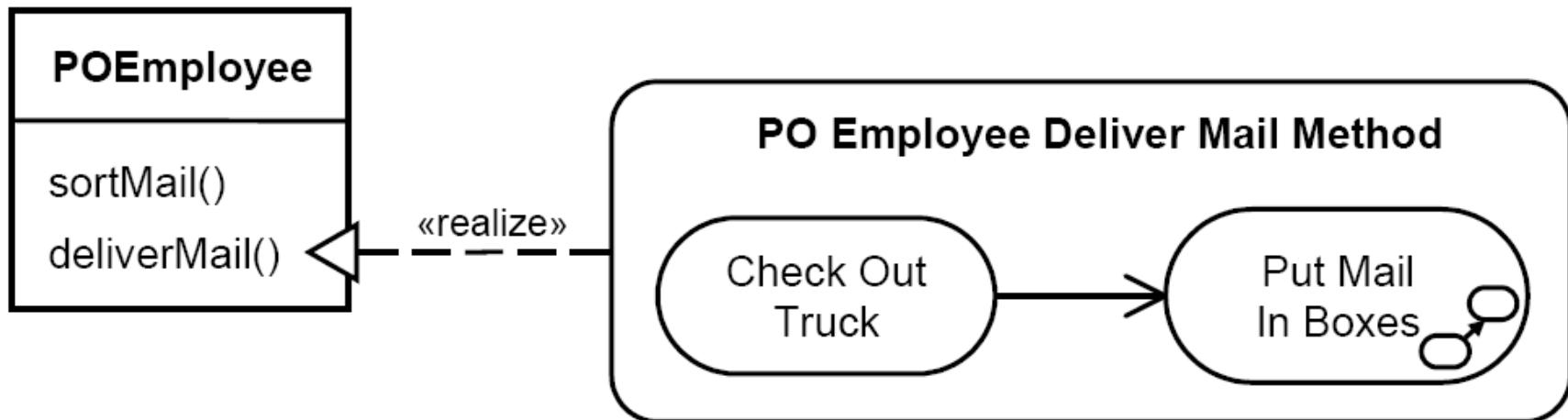
ima daljnje
podaktivnosti

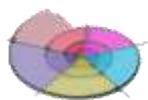


Dijagram aktivnosti kao “metoda”



- Programska potpora opisana dijagramima aktivnosti je potpuno objektno usmjerena ako svi dijagrami aktivnosti realiziraju metode (procedure) kao implementaciju pojedinih operacija.
- Na slici dijagram aktivnosti realizira (oznaka «*realize*») operaciju *deliverMail()*.
- Pokazuje se povezivanje dijagrama razreda i dijagrama aktivnosti.





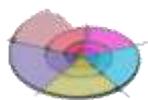
Dinamički paralelizam



- Neka aktivnost ili podaktivnost može se izvoditi više puta (paralelno).
- Broj izvođenja ovisi o rezultatu evaluacije nekog izraza tijekom izvođenja.
- Unutar simbola aktivnosti/podaktivnosti može se umetnuti UML oznaka brojnosti koja određuje dopustiv maksimalan broj paralelnog izvođenja (npr. osiguranje od beskonačne petlje).
- Nakon završetka svih paralelnih aktivnosti inicira se prijelaz na novi korak.

ovdje $0..*$, odnosno *

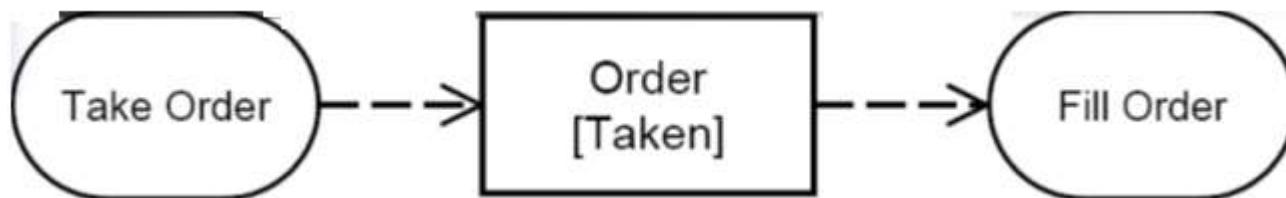
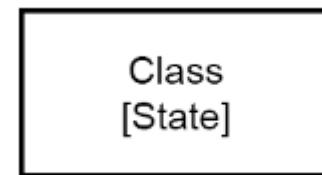
Action/Subactivity *

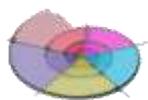


Objekt kao uvjet prijelaza



- Ponekad je potrebno prijelaz s aktivnosti na drugu aktivnost uvjetovati postojanjem određene vrste objekta (npr. ulazno/izlaznih parametara).
 - u prijelaz se umjeće simbol:
 - u tom stanju nema nikakve aktivnosti.
 - tok se nastavlja do sljedećeg koraka nakon zadovoljenja uvjeta.
- Npr. aktivnost *Take Order* mora generirati izlazni parametar (objekt određenog tipa), a aktivnost *Fill Order* mora imati ulazni parametar (objekt određenog tipa).
- U modeliranju uvjetnog prijelaza koriste se crtkane oznake.





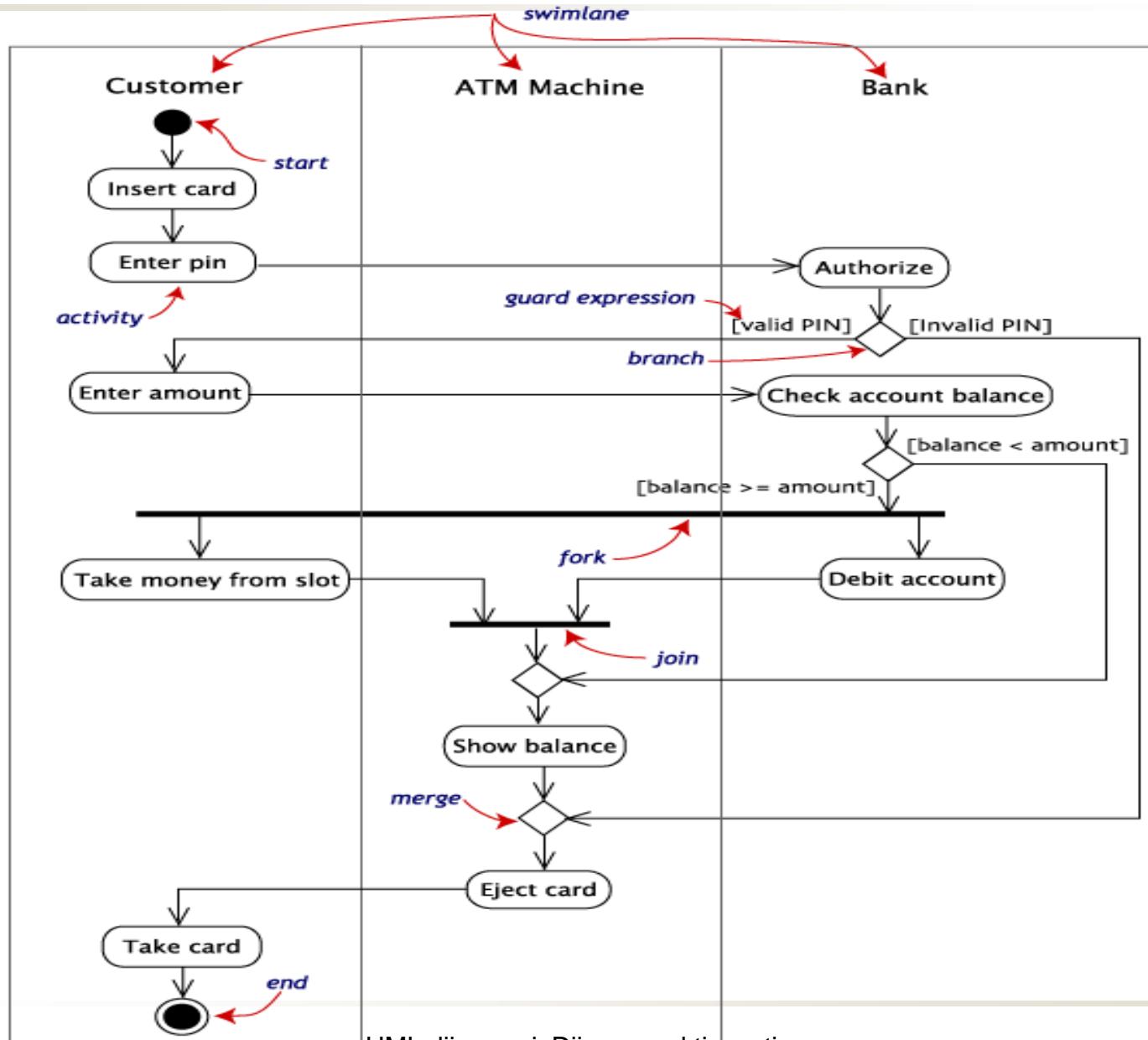
Primjena dijagrama aktivnosti



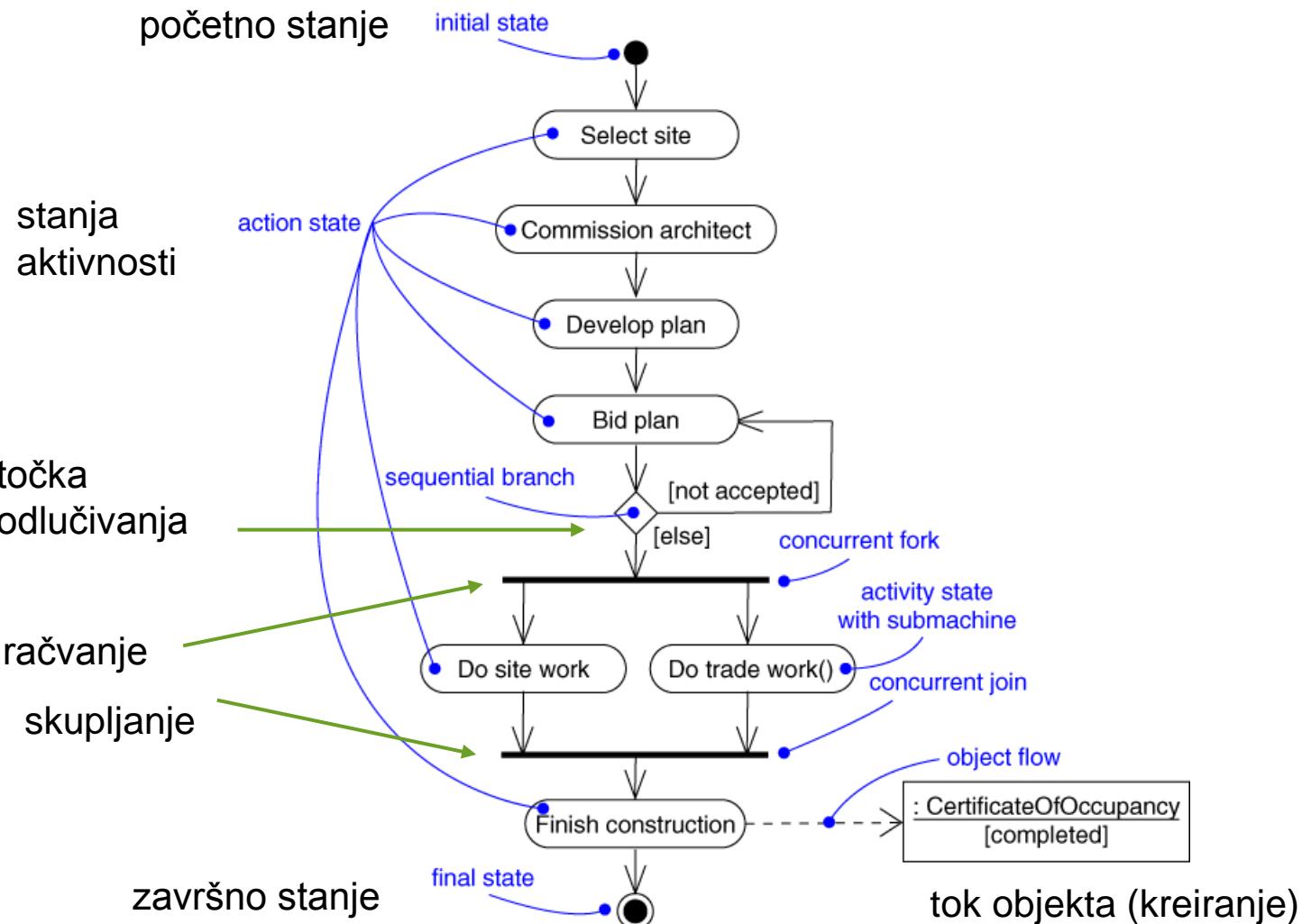
■ Pogodno za slučajeve:

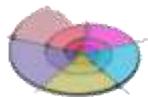
- kada ponašanje ne zavisi o velikom broju vanjskih događaja
- postoje definirani koraci bez prekida
- kada imamo tijek objekata između koraka
- kada želimo pojasniti aktivnosti

Primjer: bankomat



Primjer: Gradnja kuće

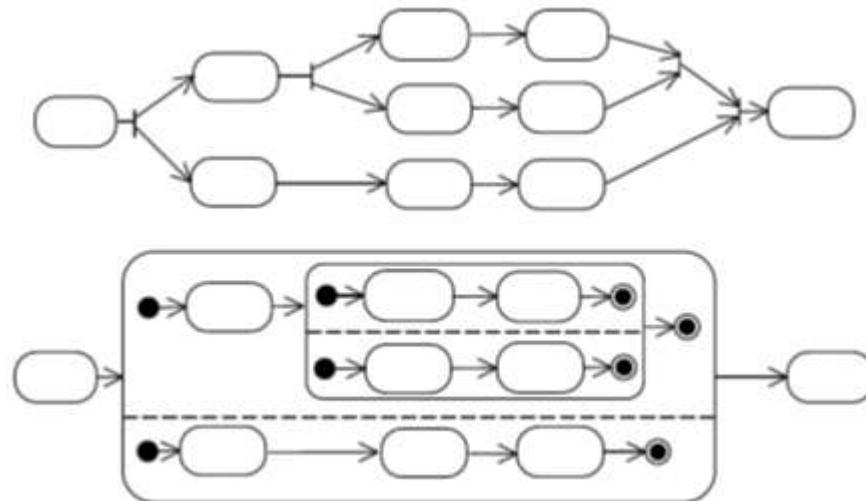




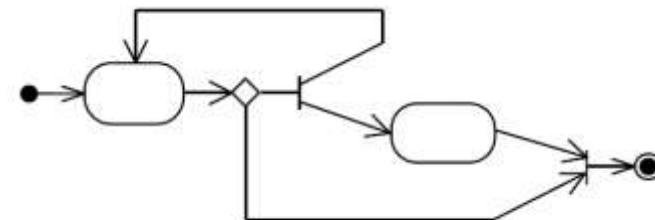
Dijagrami aktivnosti

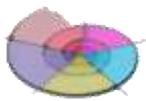


- Dijagrami aktivnosti naslijedili su iz dijagrama stanja potrebu za dobrom strukturu i ugniježđivanjem složenih stanja.
- Dobro ugniježđivanje:



- Loše ugniježđivanje:





Dijagrami aktivnosti – sažetak



- Dijagrami aktivnosti koriste se za modeliranje ponašanja koje nije jako ovisno o vanjskim događajima.
 - primarno na modeliranje poslovnog procesa (a ne na npr. oblikovanje ugrađenih sustava zasnovanih na računalima).
- Dijagrami aktivnosti slični su dijagramima stanja (do UML v1.3 nisu bili posebno deklarirani, u UML v2.0 potpuno su odvojeni).
- Dijagrami aktivnosti posjeduju korake koji se izvode do završetka (nisu prekidani događajima).
- Dijagrami aktivnosti izrađuju se u slučaju kad je usredotočenost ponašanja na pojedinim aktivnostima i njihovom slijedu a ne na to koji objekti su odgovorni za te aktivnosti.
- Između koraka postoji tok/razmjena objekata.
- Upravljački tok i podatkovni/objektni tok nemaju različitu već jedinstvenu semantiku.

UML dijagrami

- Dijagram obrazaca uporabe
- Sekvencijski dijagram
- Komunikacijski dijagram
- Dijagram stanja
- Dijagram aktivnosti
- ***Dijagram komponenti***
- Dijagram razmještaja
- Dijagram paketa
- Dijagram pregleda interakcije
- Vremenski dijagram
- Dijagram profila
- Dijagram razreda
- Dijagram objekata
- Dijagram složene strukture

Programske komponente

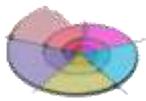
- engl. *software component*
 - “are binary units of independent production, acquisition, and deployment that interact to form a functioning system”, C. *Szyperski*
 - zamjenjivi, ponovo iskoristivi dijelovi koda.
- U programskom inženjerstvu zasnovanom na komponentama sustav se integrira
 - višestrukom uporabom postojećih komponenata,
 - uporabom komercijalnih, gotovih komponenata (engl. *commercial-off-the-shelf* COTS) ili
 - modificiranim komercijalnim komponentama (engl. *modified-off-the-shelf* MOTS).



Dijagrami komponenti



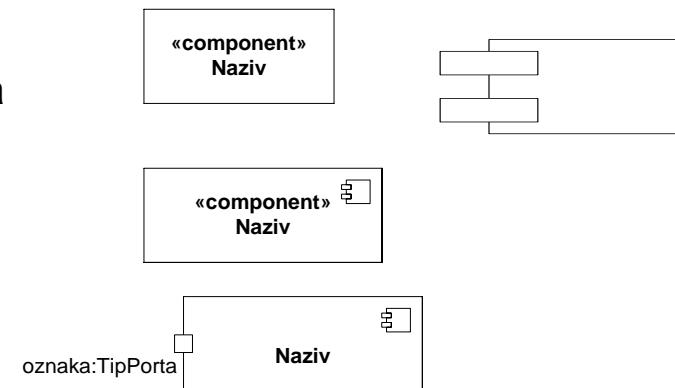
- Dijagrami komponenti predstavljaju statički pogled na sustav.
- Opisuju organizaciju i međuovisnost između implementacijskih komponenata programske potpore.
- Dijagrami komponenti dio su specifikacije arhitekture programske potpore.
- Dijagrame komponenti oblikuju arhitekti programske potpore i programeri.
- Vrste komponenata:
 - izvorni kod
 - binarni (objektni) kod
 - statičke ili dinamičke knjižnice programskih komponenata
 - izvršne komponente programske potpore (aka “also known as” exe)
 - tablice
 - druge datoteke
 - baze podataka
 - ...

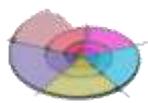


Dijagram komponenti



- engl. *Component Diagram*
- UML komponente predstavljaju **fizičke** modularne, zamjenjive jedinke s dobro definiranim sučeljem
 - obuhvaća neki sadržaj kojem se pristupa putem sučelja
 - definira ponašanje kroz ponuđena i zahtijevana sučelja
 - **razred** - logička apstrakciju
 - atribute i operacije
 - može im se pristupati izravno
 - **komponenta** - fizička stvar, fizičko obuhvaćanje logičkih apstrakcija
 - operacije
 - može im se pristupati samo kroz sučelja
- Namjena komponentnog dijagrama je prikaz komponenata, interne strukture i odnosa prema okolini
- Komponente mogu sadržavati druge komponente te na taj način prikazivati internu strukturu.
- Mogu imati definirane portove (*engl. port*)
 - točka interakcije komponente s okolinom
- Instanca komponente
 - ime_komponente: Tip_komponente





Sučelja/Konektori



- engl. *Interface*
- Imenovan skup javno vidljivih atributa i apstraktnih operacija
 - implementaciju osigurava komponenta/razred
- Komponente i razredi ostvaruju sučelja
 - uključivanjem atributa i implementacijom operacija

Tipovi sučelja:

- Ponuđeno/implementirano sučelje – engl. *provided interface*
 - ostvaruje ga razred ili komponenta
 - usluga koja se nudi
- Zahtijevano sučelje – engl. *required interface*
 - potrebno klasi ili komponenti
 - ono što je potrebno za njezin rad

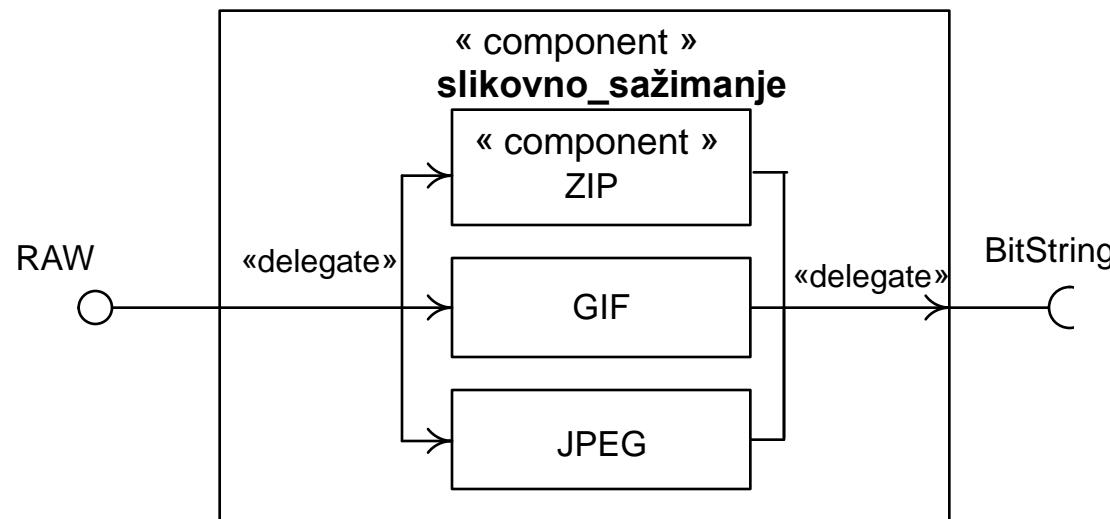


Tipovi konektora:

- Spojnica - engl. *assembly connector*
 - povezuje dvije komponente
 - “ball-and-socket”, “lollipop”
- Delegacija - engl. *delegation connector*
 - povezuje sučelje komponente s internom strukturom

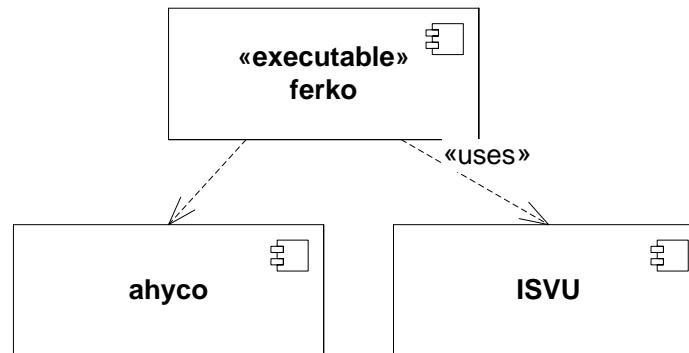


Primjer interne strukture komponente



Ovisnost

- engl. *dependency*
- Ova relacija između komponenti upotrebljava se kada jedna komponenta zahtijeva uporabu druge radi potpunog ostvarenja implementacije
- Predstavlja se crtkanom strelicom od ovisne komponente



Primjer

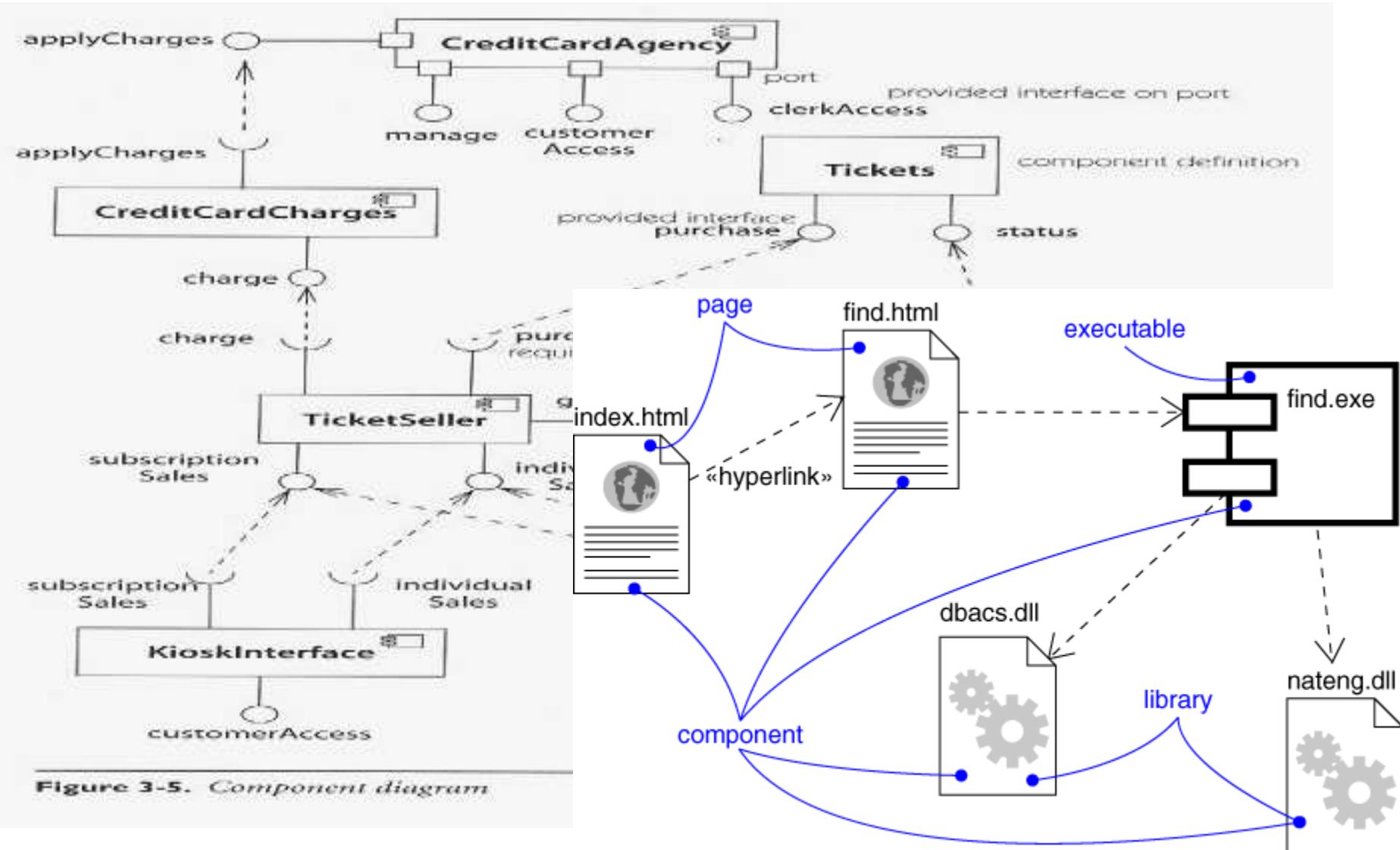


Figure 3-5. Component diagram

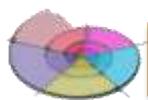
Artefakti i stereotipovi

■ Artefakti

- iz razvojnog procesa: izvorni kod, podaci ...
 - engl. *Source Component*
- binarne komponente za isporuku
 - engl. *Binary Component (object code file, static or dynamic libraries)*
- izvršne komponente
 - engl. *Executable component*

■ Stereotipovi

- *executable* - komponenta koja se može izvršavati
- *library* - statička ili dinamička biblioteka
- *file* - datoteka s proizvoljnim sadržajem
- *document* - dokument
- *script* - skript
- *source* - datoteka sa izvornim kodom



Primjena komponentnih dijagrama



- Statički model programskih komponenti
 - ponovna uporaba i zamjena komponenti dijelova
- Oblikovanje programskih komponenti
 - arhitekturni model
 - detalji oblikovanja
 - odnos s okolinom
- Oblikovanje interne strukture komponenti

UML dijagrami

- Dijagram obrazaca uporabe
- Sekvencijski dijagram
- Komunikacijski dijagram
- Dijagram stanja
- Dijagram aktivnosti
- Dijagram komponenti
- ***Dijagram razmještaja***
- Dijagram paketa
- Dijagram pregleda interakcije
- Vremenski dijagram
- Dijagram profila
- Dijagram razreda
- Dijagram objekata
- Dijagram složene strukture



Dijagram razmještaja



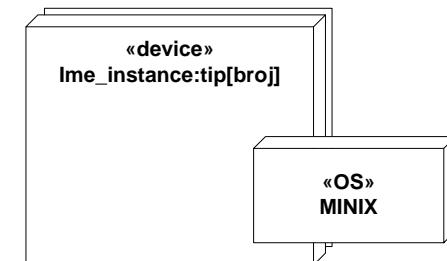
- engl. Deployment diagram
- Opisuju topologiju sustava i usredotočeni su na odnos sklopoških i programske dijelove
 - ostvarenje u obliku koda i podataka koji se nalaze i izvršavaju na računalnim resursima
 - prikaz sklopoških komponenti, komunikacijskih putova te smještaja i izvođenja programskih artifakta
 - naznaka gdje i kako implementirati komponente

Sklopoške komponente:

- Čvorovi – engl. nodes
 - uređaj – engl. device
 - stvarni i virtualni uređaji
 - procesna jedinka npr. *računalo*
 - oznaka «device»
 - okolina izvođenja – engl. execution environment
 - programski sustav npr. *virtualni stroj, OS, interpreter*
 - oznaka «execution environment»
- Spojevi (vezice) – engl. connections
 - Komunikacijski putovi

Programske komponente (artifakti):

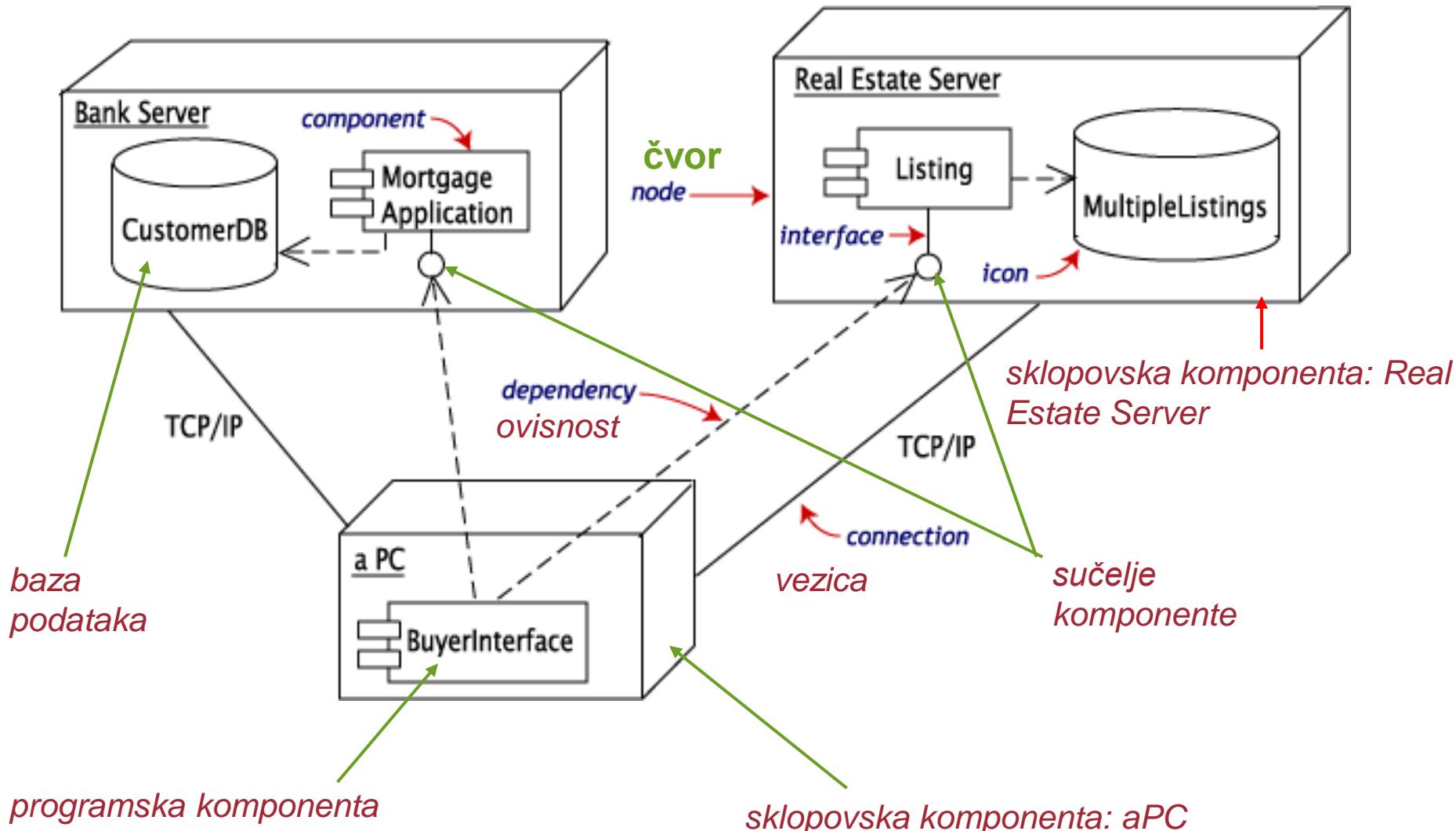
- Komponente – implementirani moduli i podaci
- Ovisnosti – engl. Dependencies
 - prikazuju odnos između komponenti



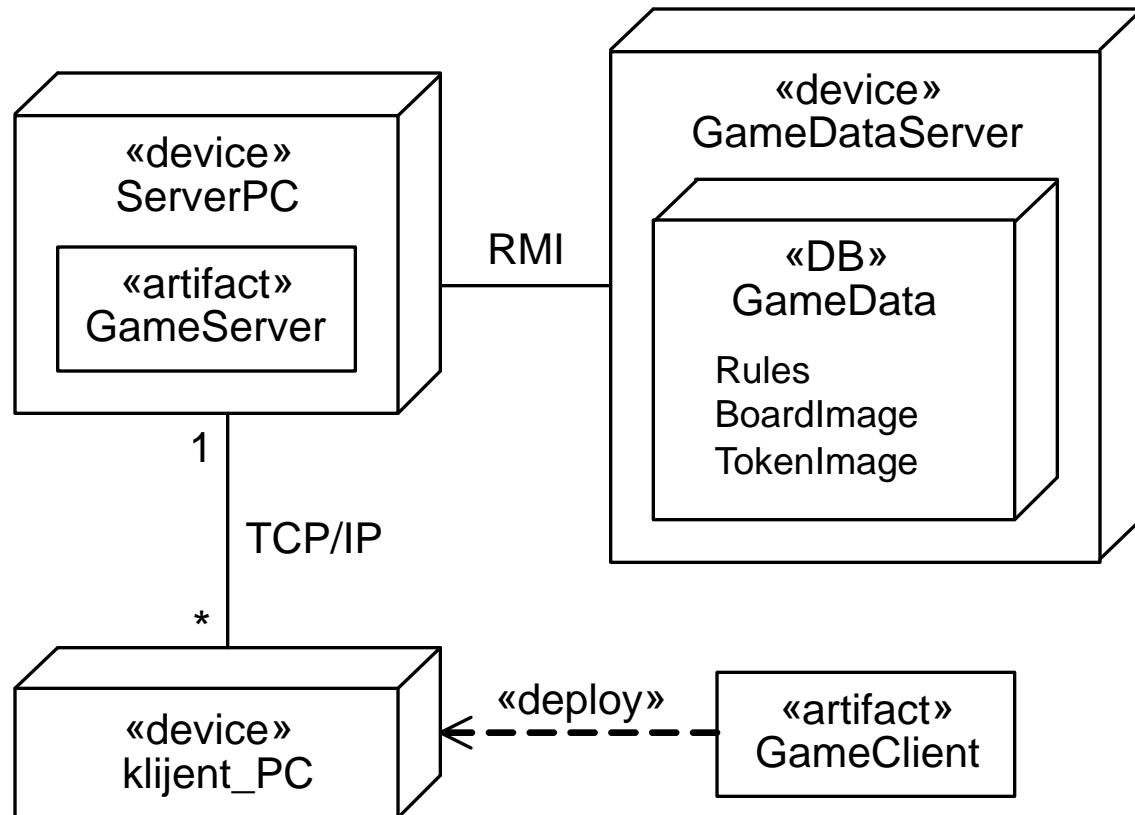
Vezice - punom crtom
Ovisnost - crtkanom strelicom



Primjer: kupnja nekretnine



Primjer:

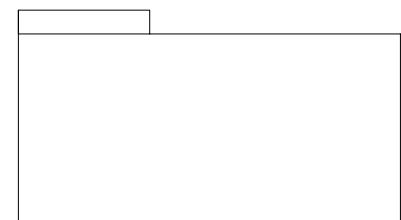


UML dijagrami

- Dijagram obrazaca uporabe
- Sekvencijski dijagram
- Komunikacijski dijagram
- Dijagram stanja
- Dijagram aktivnosti
- Dijagram komponenti
- Dijagram razmještaja
- ***Dijagram paketa***
- ***Dijagram pregleda interakcije***
- ***Vremenski dijagram***
- ***Dijagram profila***
- Dijagram razreda
- Dijagram objekata
- ***Dijagram složene strukture***

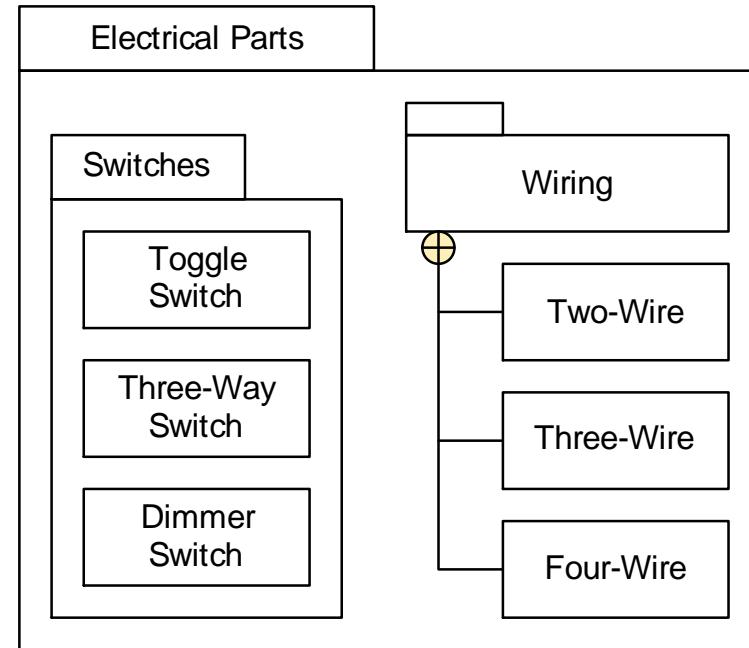
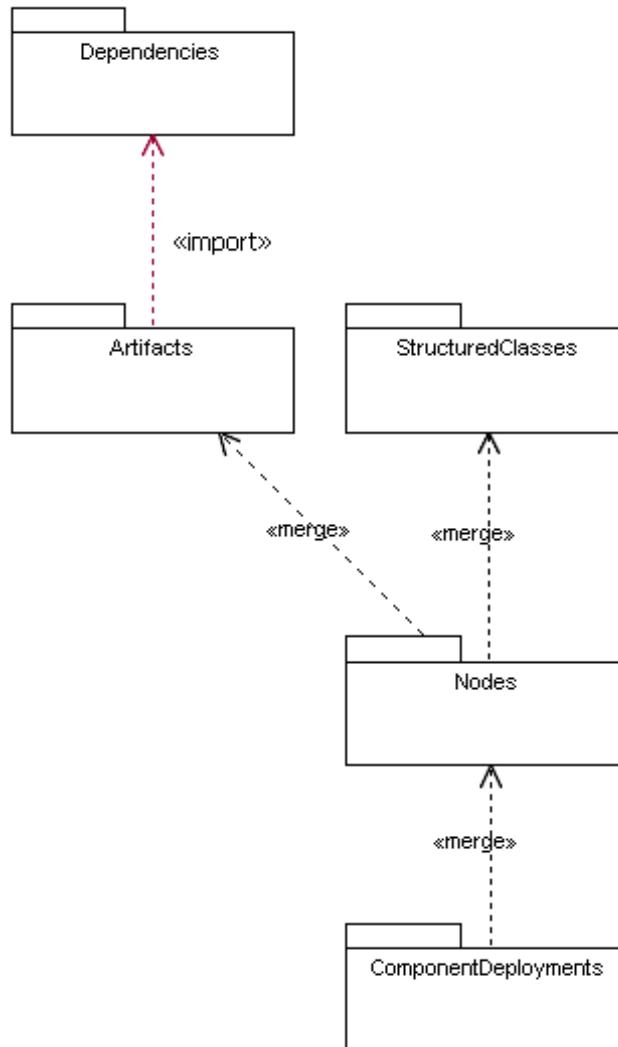
Dijagram paketa

- engl. *Package Diagram*
- Paket je mehanizam organiziranja elemenata u grupe
 - nema utjecaj na izvođenje
- Dobro definiran paket povezuje semantički bliske elemente koji imaju tendenciju zajedničkih promjena
- U jednostavnim aplikacijama nema potrebe za uvođenjem paketa
- Dijagram paketa prikazuje dekompoziciju u organizirane grupe i njihove međuodnose
 - hijerarhija
- Vidljivost – moguća su tri nivoa:
 - javni element (engl. *public*)
 - sadržaj vidljiv svim paketima koji koriste paket (+)
 - zaštićeni element (engl. *protected*)
 - mogu ga vidjeti samo djeca tog paketa (#)
 - privatni element (engl. *private*)
 - nije vidljiv izvan paketa (-)

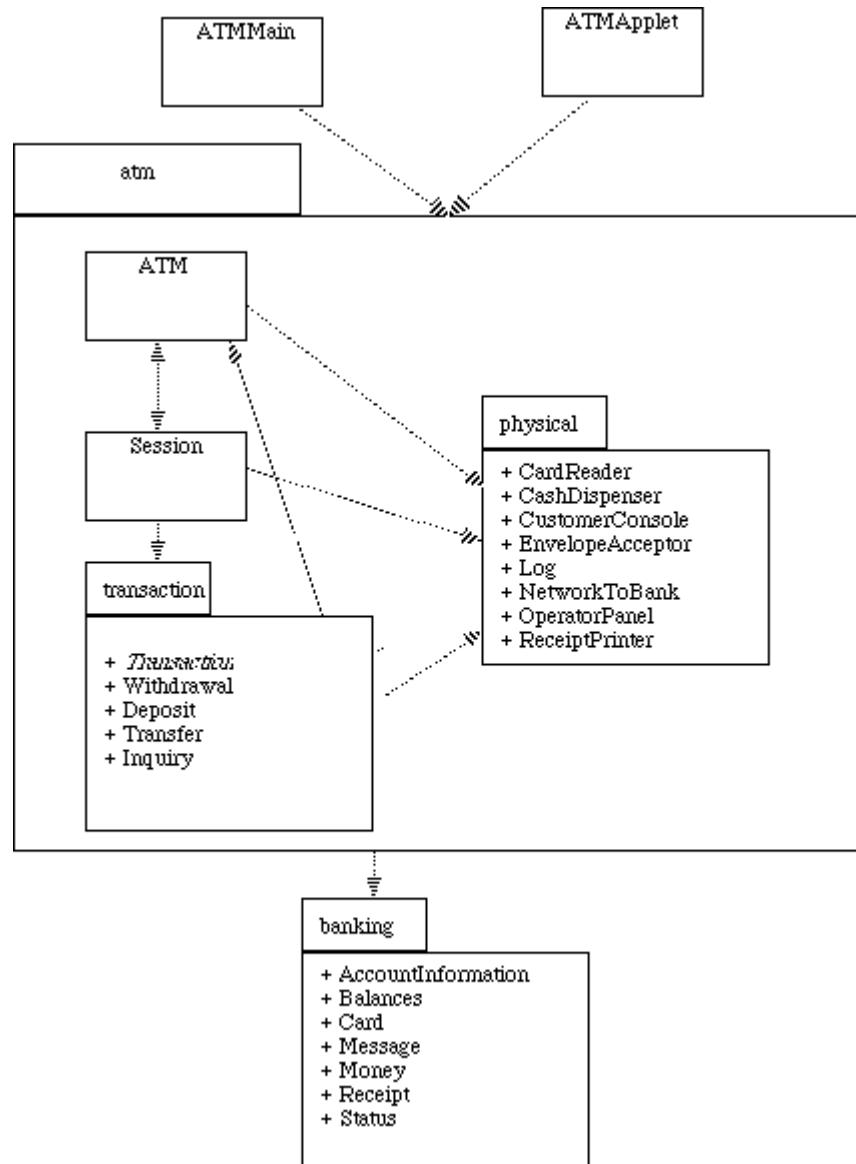




Primjer: Dijagrami paketa



Primjer: bankomat

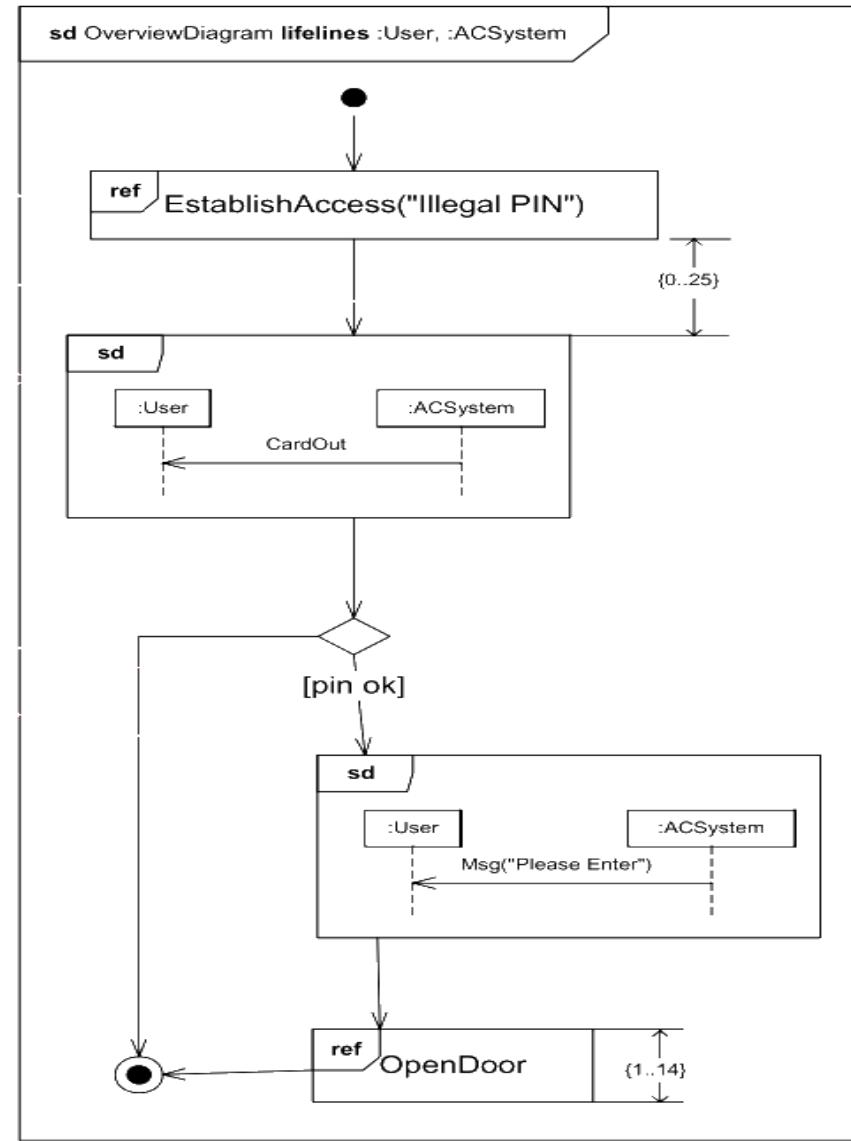




Dijagram pregleda interakcije

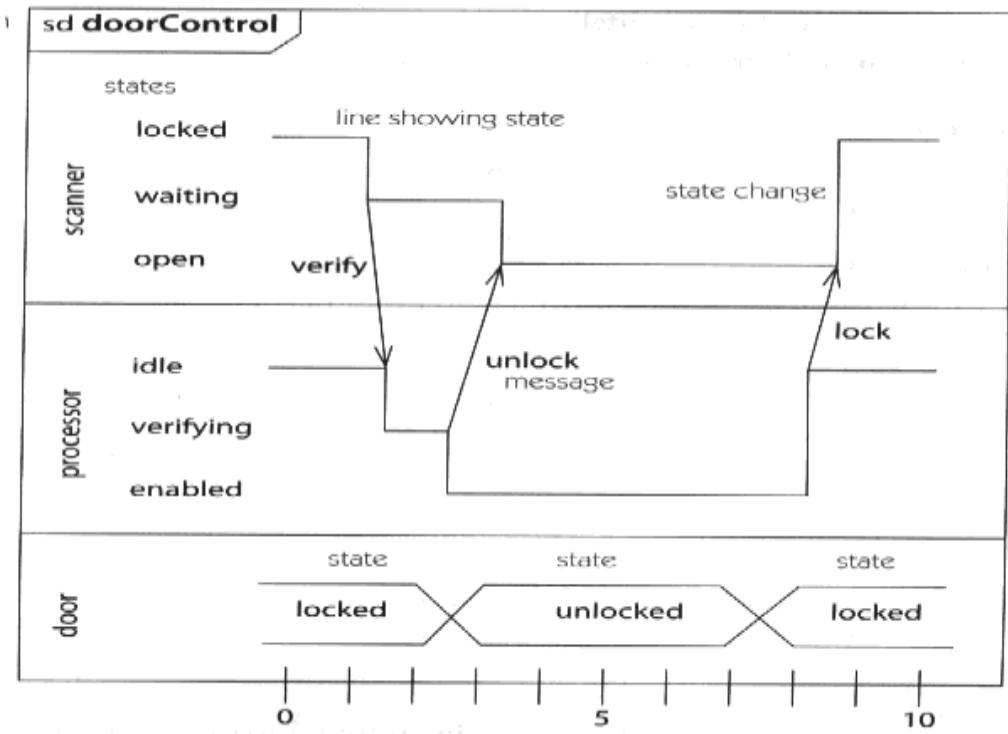


- engl. *Interaction Overview Diagram*
- Kombinacija dijagrama aktivnosti i sekvenčnog dijagrama
 - dijagram aktivnosti s dijelovima sekvenčnog dijagrama i kontrolom tijeka
 - notacija odluka i grananja iz dijagrama aktivnosti



Vremenski dijagram

- engl. *Timing Diagram*
- Vrsta dijagrama interakcije
- Prilagođen za izričit prikaz stvarnih vremena
 - točniji zapis sekveničkog dijagrama (vidljiv samo relativan odnos poruka)
 - prikazuje promjene stanja na liniji života u vremenu
- Pogodan za primjenu u sustavima za rad u stvarnom vremenu (engl. *real-time applications*)



Dijagram profila

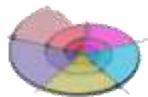
- engl. *Profile Diagram*
- Namjena proširenje jezika za stvaranje novih dijagrama
- Statički dijagrami strukture koji pokazuju proširenje postojećih ili nove elemente modeliranja.
- Omogućava definiranje korisničkih stereotipova, vrijednosti, ikona za specifična područja primjene, tehnologije ili metode.
- UML Profil
 - skup predefiniranih stereotipova
 - NE proširuje sam UML, već pobliže opisuje specifične primjene
 - pojednostavljuje primjenu UML-a u drugim domenama primjene



Primjeri UML profila

- UML profile for CORBA
- UML profiles for System-on-a-chip
- UML testing profiles

- Više na:
http://www.omg.org/technology/documents/profile_catalog.htm



Primjeri UML profila



Concept	Mapping to existing UML	UML extension	Description
Vulnerability	Object	<< vulnerability >>  *{vulnerability}	This stereotype is used to distinguish objects containing vulnerabilities from normal objects. The tag vulnerability is used to specify the vulnerability. If more than one vulnerability is present, several vulnerability tags should be used (and numbered).
Unwanted incident	Message	<< unwanted incident >> 	This stereotype is used to distinguish unwanted incidents from normal messages.
Misuser	Actor instance	<< misuser >>  *{type} *{intention}	This stereotype is used to distinguish misusers from normal users (actors). The tag type refers to whether the misuser is insider or outsider. The tag intention refers to whether the misuse is intended or unintended.
Threat	Message	<< threat >> 	This stereotype is used to distinguish threats from normal and abnormal messages.
Asset	Object	<< asset >> 	Stereotype from CORAS UML profile.

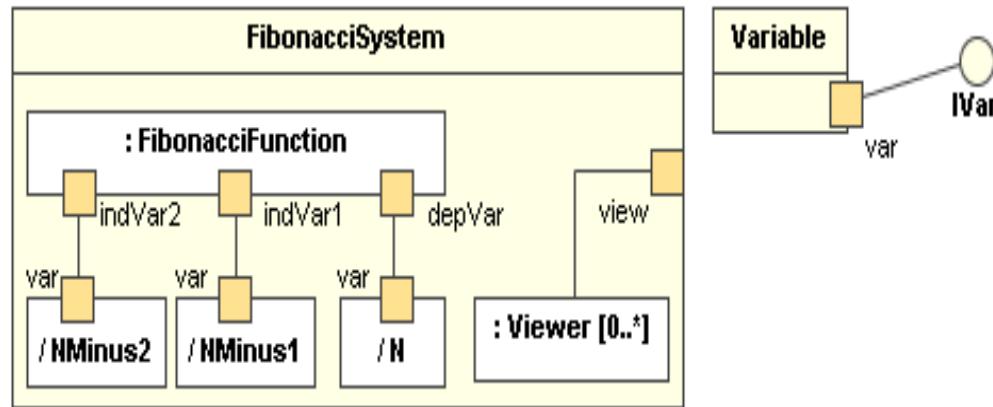
Izvor: Houmb S.H., Hansen K.K.: Towards a UML Profile for Security Assessment



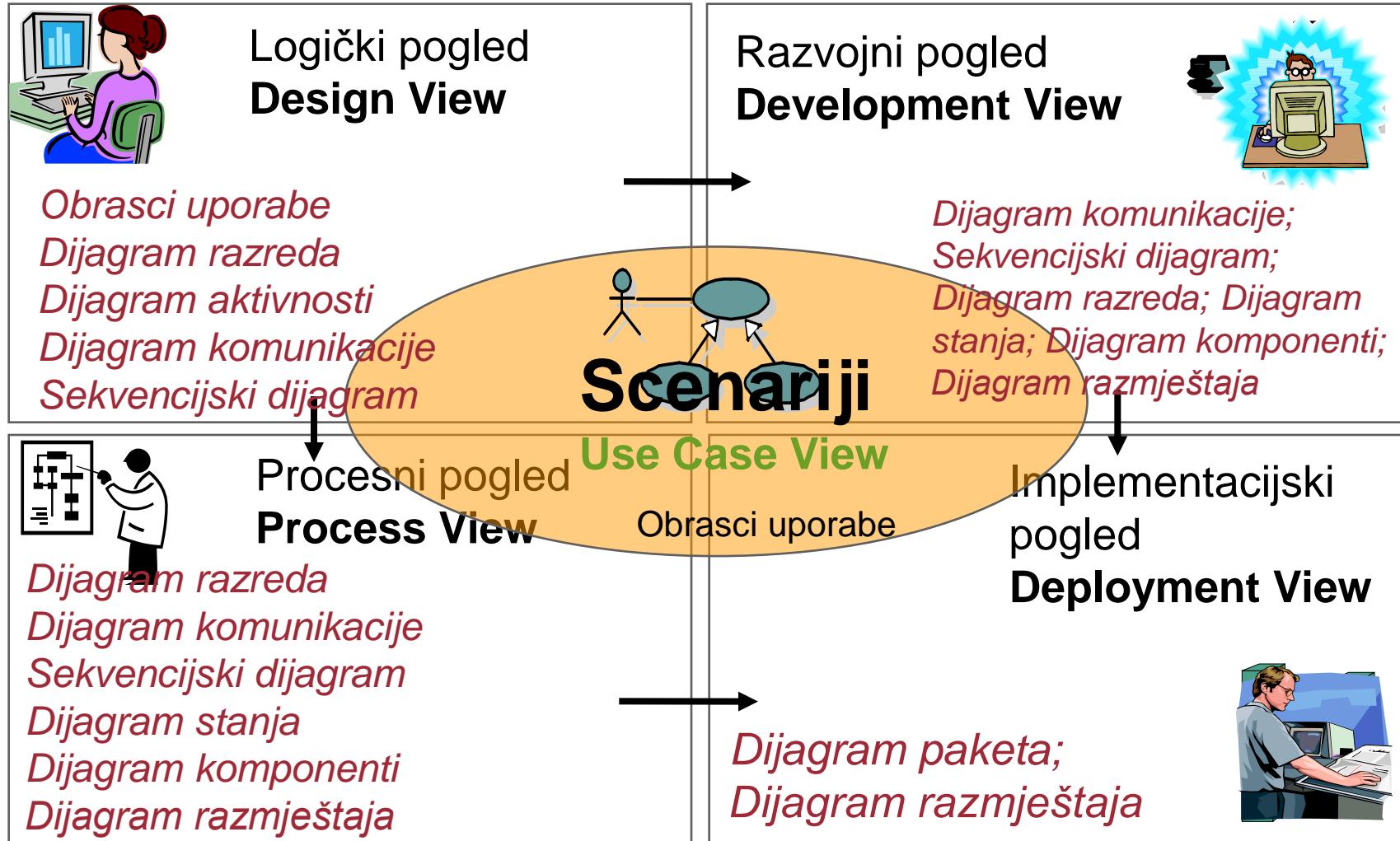
Dijagrami složene strukture



- engl. *Composite Structure Diagram*
- Opisuje konfiguraciju i odnose dijelova koji zajednički ostvaruju neko ponašanje
- Opisuju internu strukturu razreda i suradnje koje ta struktura omogućuje.
 - uključuju interne dijelove i sučelja (engl. ports) za međusobnu interakciju dijelova i interakciju s vanjskim svijetom, te konektore.



Uporaba UML dijagrama



Diskusija

-
-
-
-
-

Oblikovanje programske potpore

ak.god. 2014./2015.

Arhitekturni obrasci



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



Tema

- Podsjetnik
- Ponovna uporaba programskih komponenti
- Arhitekturni obrasci
 - raspodijeljeni sustavi
 - arhitektura klijent – poslužitelj
 - primjer objektnog radnog okvira klijent-poslužitelj
 - arhitektura zasnovana na događajima
 - obrazac model-pogled-nadglednik
 - posrednička arhitektura
 - uslužno usmjerena arhitektura – SOA
 - arhitektura sustava zasnovanih na komponentama
- Implementacija programskog proizvoda

Literatura

- Timothy C. Lethbridge, Robert Laganière: ***Object-Oriented Software Engineering: Practical Software Development using UML and Java***, McGraw Hill, 2001
 - <http://www.lloseng.com>
- Sommerville, I., ***Software engineering***, 8th ed, Addison Wesley, 2007.

Pripremio i prilagodio: Nikola Bogunović, Vlado Sruk

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

U pripremi materijala osim literature upotrijebljeni su i drugi izvori, te zahvaljujem autorima.

Podsjetnik

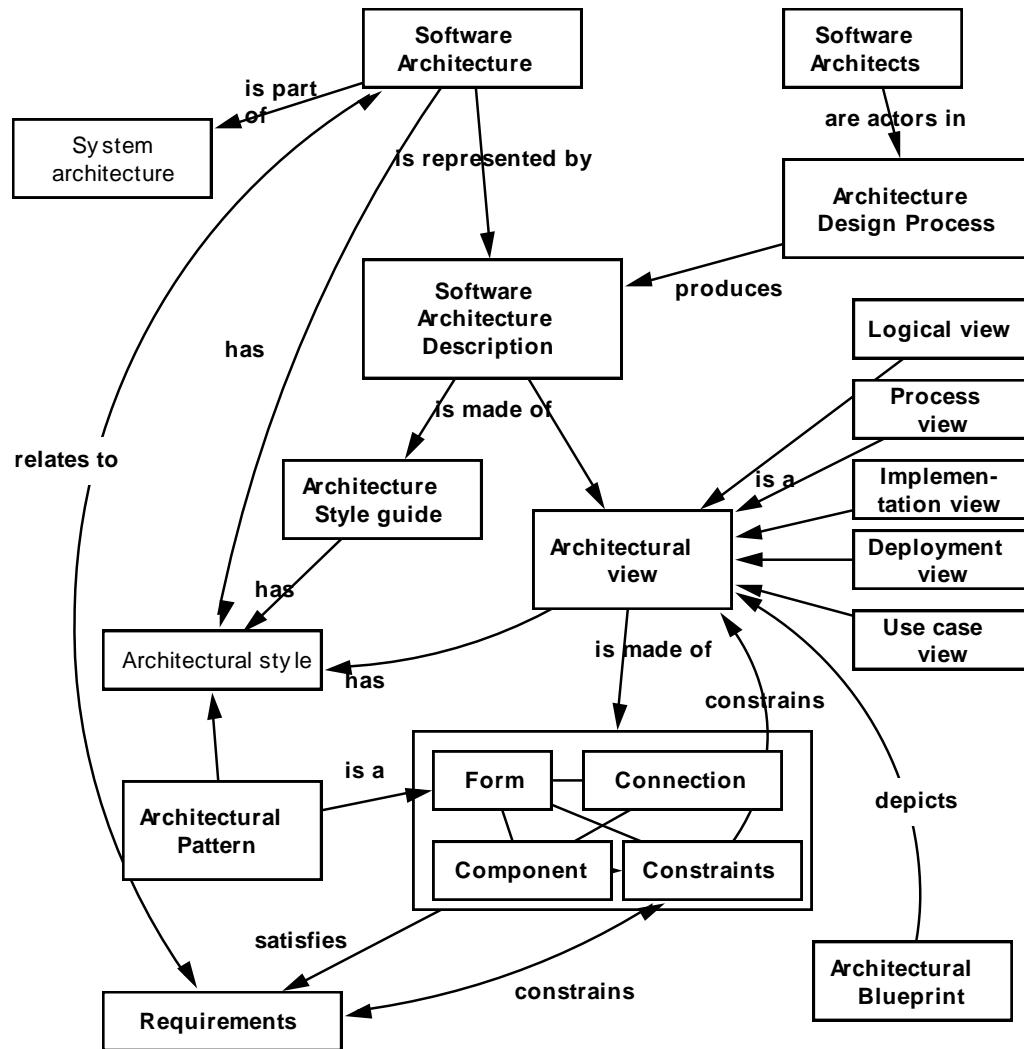
- Modeliranje obrazaca uporabe
- Modeliranje interakcija u sustavu
- Objektno usmjereni model sustava
 - dijagrami razreda
 - razredi i poveznice
 - modeliranje hijerarhije - generalizacija
 - modeliranje agregacije
 - modeliranje ponašanja (engl. *Behavioral models*)
 - modeliranje stanja (engl. *State machine models*)

Podsjetnik

- Arhitektura programske podrške
 - oblikovanje arhitekture programske podrške
 - fokus kako donositi ispravne odluke oblikovanja
 - odluke oblikovanja arhitekture (engl. *Architectural design decisions*)
 - stilovi arhitekture programske potpore (engl. *Architectural style*)
 - perspektive oblikovanja arhitekture
 - 4+1 pogled
 - ponovna uporaba (engl. *design reuse*)
 - procjena arhitekture
 - procjena atributa arhitekture programske podrške i utjecaja na proces donošenja odluka
 - npr. Architecture Tradeoff Analysis Method - ATAM
 - dokumentiranje arhitekture
 - daje naputak za dokumentiranje arhitekture
 - ne obuhvaća eksplicitno način donošenja odluka
- Proces oblikovanja objektno usmjerenih sustava
 - engl. *object-oriented design process*
 - utvrđivanje razreda, sučelja, ...

Arhitekturni stilovi i obrasci

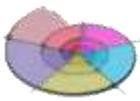
■ Metamodel arhitekture



Arhitektturni obrasci

- engl. *architectural patterns*
- skup odluka oblikovanja arhitekture primjenjiv na česte probleme
 - često parametriziran u svrhu šire uporabljivosti
- namjena: pojednostavljene odluke oblikovanja arhitekture na visokom nivou i organizaciji programa
- oblikovni obrasci (engl. *design patterns*)
 - opisuju detalje na nižim razinama implementacije

RADNI OKVIRI

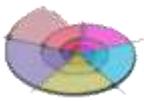


Koncept ponovne uporabe



- Oblikovanje temeljem iskustva drugih
- Inženjeri koji oblikuju programsku potporu trebaju izbjegavati razvoj programske potpore koja je već bila razvijena.
- Tipovi ponovne uporabe:
 - ponovna uporaba znanja
 - engl. *Reuse of expertise*
 - ponovna uporaba standardnog oblikovanja i algoritama
 - engl. *Reuse of standard designs and algorithms*
 - ponovna uporaba knjižnice razreda ili procedura
 - engl. *Reuse of libraries of classes or procedures*
 - ponovna uporaba značajnih naredbi jezika i OS-a
 - engl. *Reuse of powerful commands built into languages and operating systems*
 - ponovna uporaba okvira
 - engl. *Reuse of frameworks*
 - ponovna uporaba cijelih aplikacija
 - engl. *Reuse of complete applications*

Zašto?



Oblikovni obrasci



- engl. *design pattern*
- Dokazano dobar način ponovne uporabe znanja o čestom problemu i načinu rješavanja
- Obrazac (engl. *pattern*) daje opis problema i osnovni predložak rješenja primjenjiv u različitim situacijama
 - općenito rješenje za česte probleme
 - predlošci s rješenjem problema
 - nije kompletno rješenje
 - često upotrebljava nasljeđivanje i polimorfizam
- Značajno unapređenje objektno usmjerenog oblikovanja
- Preduvjeti za uporabu oblikovnih obrazaca
 - shvaćanje da problem ima moguće rješenje primjenom obrasca
- E.Gamma je 1995. identificirao 23 oblikovna obrasca. Danas znatno više.

“Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to the problem, in such a way that you can use this solution a million times over, without doing it the same way twice.” Christopher Alexander, 1977.

Elementi obrasca

- Objektno usmjereni obrasci uporabe (engl. *object-oriented design pattern*) sustavno imenuju, pojašnjavaju i vrednuju značajne često upotrebljavane obrasce u objektno usmjerenim sustavima
- Predlošci oblikovnih obrazaca nisu normirani!
- Minimalni elementi obrasca:

Element	Opis
Naziv	Razumljivo ime
Opis	Opis problema
Rješenje	Opis predloška koji može biti upotrijebljen na različite načine
Posljedice	Rezultati i pogodnosti primjene obrasca uporabe



Predložak OO oblikovnog obrasca

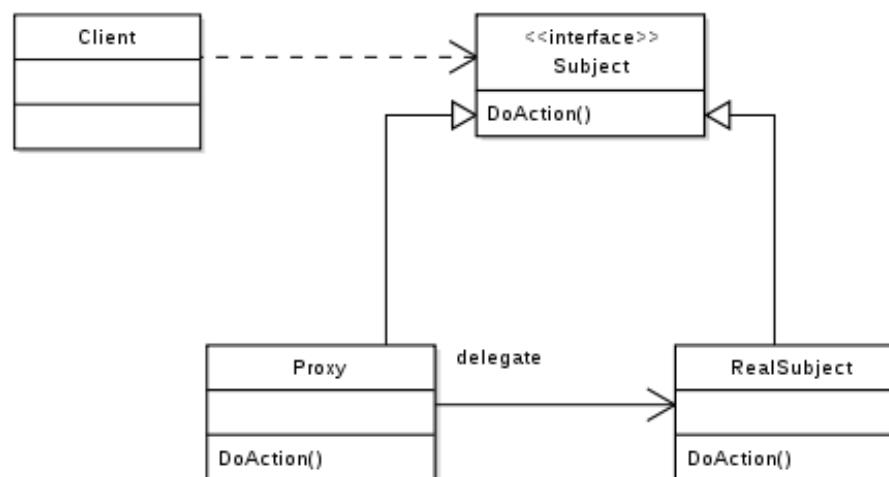


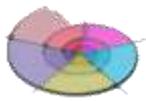
Element	Opis
Naziv Pattern Name	Izražajno kratko ime
Namjena Intent	Opis namjene obrasca
Sinonim Also Known As	Lista sinonima za isti obrazac
Motivacija	Primjer problema i način rješavanja uporabom obrasca
Primjenjivost	Popis slučajeva pogodnih za uporabu
Struktura	Skup dijagrama razreda i objekata koji opisuju obrazac
Elementi Participants	Opis razreda i objekata te njihovih odgovornosti
Međudjelovanje Collaborations	Opis kako se izvršavaju odgovornosti
Posljedice Consequences	Opis uvjeta obrasca, prednosti i ustupaka



Primjer: Oblikovni obrazac “Proxy”

Element	Opis
Naziv	Proxy
Opis	Osigurava zamjenski objekt (surogat) koji upravlja pristupom cilnjom objektu Dodatna razina indirekcije: dijeljeni, kontrolirani ili pametni pristup
Primjenjivost	Omogućava uobičajen pristup inače nedostupnim objektima iz npr. drugih procesa, arhiva i sl.
Posljedice	Transparentan pristup





Tipovi oblikovnih obrazaca

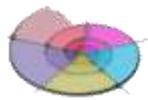


- Velik broj obrazaca
 - stvaralački (engl. *creational*)
 - usredotočeni na načine stvaranja objekata
 - npr. builder, factory, prototype, singleton
 - strukturni (engl. *structural*)
 - usredotočeni na odnose razreda i objekata
 - npr. adapter, bridge, composite, decorator, façade, flyweight, proxy
 - ponašajni (engl. *behavioral*)
 - usredotočeni na međudjelovanja razreda i objekata
 - npr. command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor
- Specifični
 - npr. J2EE Patterns – opisuje prezentacijski nivo
- Više: http://sourcemaking.com/design_patterns

Radni okviri

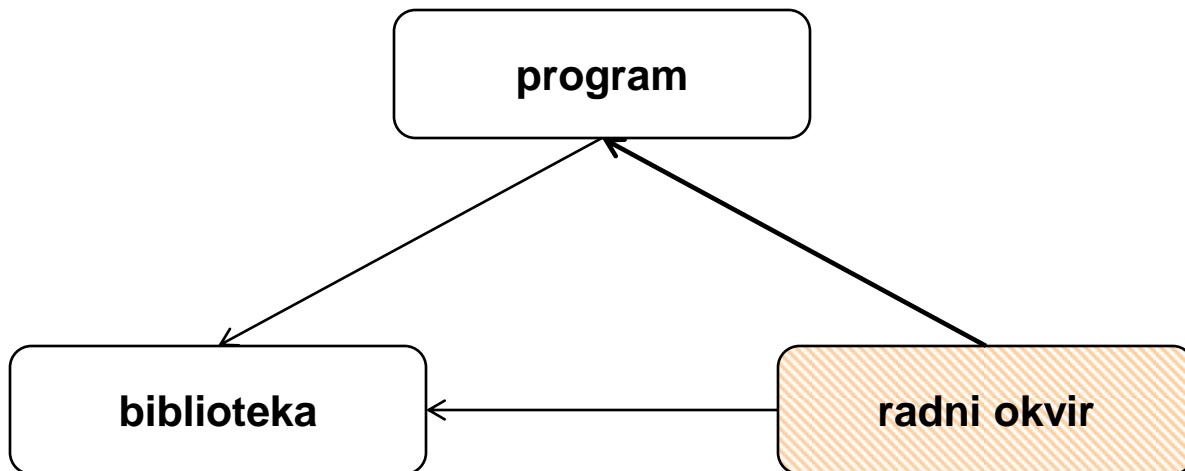
- engl. *Framework*
- Platforma za razvoj programske potpore
- Radni okvir je skup integriranih komponenti koji omogućava ponovnu uporabu arhitekture za učestalo korišten dio programske potpore
 - osigurava opća (zajednička) sredstva koja se mogu uporabiti u različitim primjenskim programima.
- Princip ponovne uporabe:
 - aplikacije/primjenski programi namijenjeni obavljanju različitih poslova slične namjene slično su oblikovani
- Okviri su u osnovi nepotpuni
 - postoje razredi ili metode koje se nužno trebaju implementirati ako se žele koristiti (engl. *slot*)
 - implementacija nekih funkcionalnosti nije obvezna
 - Prema potrebi mogu se dodati tijekom razvoja (engl. *hook*)
 - pružaju određene usluge
 - Application Program Interface (API)
- Složeni – zahtijevaju vrijeme učenja (npr. .NET, DirectX ...)

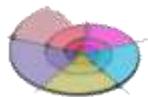
U čemu je razlika API-a i
okvira?



Uporaba radnih okvira

- Program se iz radnih okvira razvija:
 - nasljeđivanjem komponenti
 - instanciranjem parametriziranih komponenti
 - razvojem funkcija koje nisu implementirane
- Uspješno primjenjiv za ciljanu domenu primjene
- Radni okvir implementira osnovnu logiku aktiviranja komponenti

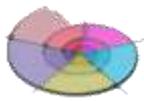




Objektno usmjereni radni okviri



- engl. *Object Oriented Frameworks*
- U objektno usmjerenoj paradigmi radni **okvir se sastoji iz knjižnica razreda**.
- Primjensko sučelje definirano je kao skup svih javnih (engl. *public*) metoda tih razreda
 - engl. *Application programming interface – API*
 - neki od tih razreda su apstraktni.
 - zahtijevaju implementaciju



Radni okviri i linije proizvoda



■ engl. *software product lines* -SPL

“a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are **developed from a common set of core assets in a prescribed way**”

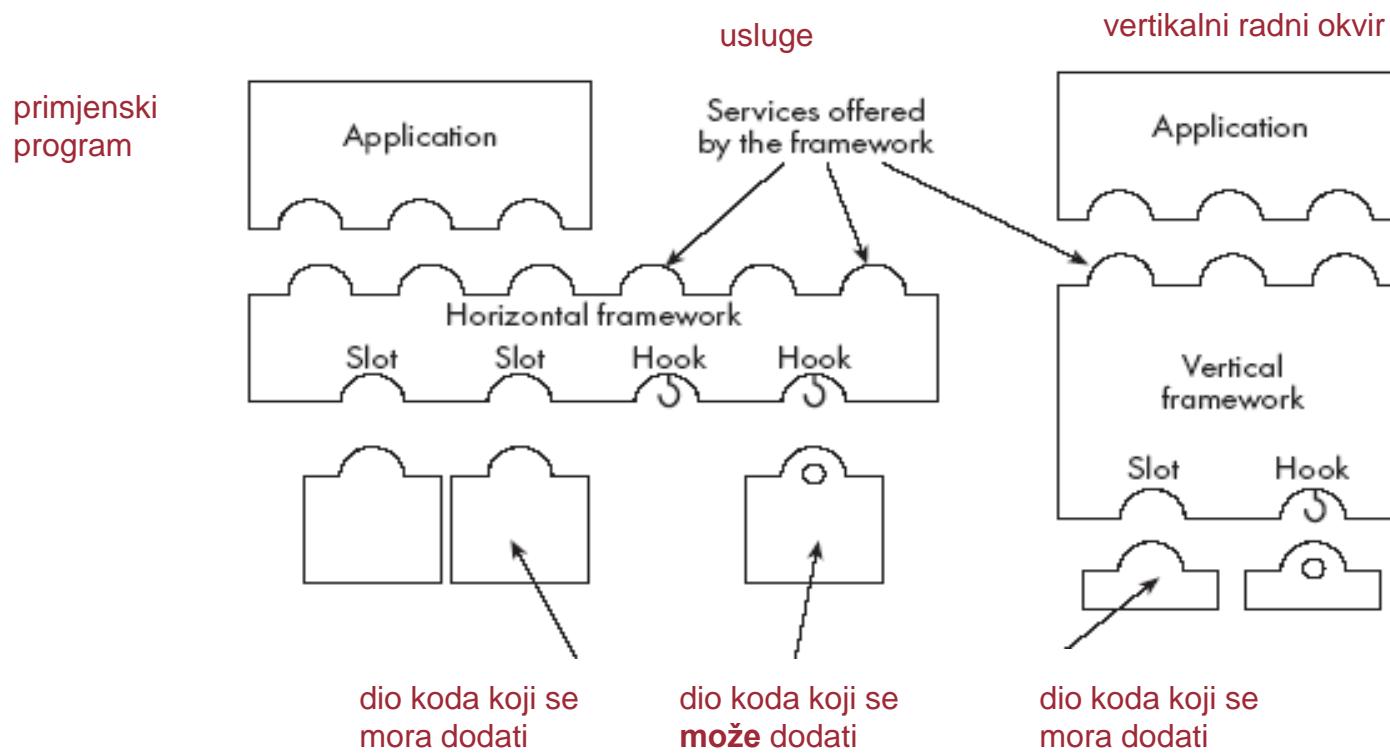
www.sei.cmu.edu/productlines

- Linije proizvoda (ili porodica proizvoda) je skup svih produkata izrađenih na zajedničkoj osnovnoj tehnologiji.
- Različiti produkti u liniji proizvoda imaju različite značajke kako bi zadovoljili različite segmente tržišta.
 - npr. “demo”, “pro”, “lite”, “enterprise” i sl. verzije.
 - lokalizirane verzije su također linije proizvoda.
- Programska tehnologija zajednička svim proizvodima u liniji proizvoda uključena je u radni okvir (engl. *framework*).
- Svaki proizvod izrađen je temeljem radnog okvira u kojem su popunjena odgovarajuća prazna mjesta.

Koji su poticaji i ograničenja?

Arhitekture radnih okvira

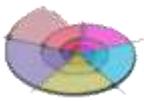
- **Horizontalni radni okvir** osigurava opća i zajednička sredstva koja mogu koristiti mnogi primjenski programi (aplikacije).
- **Vertikalni radni okvir** je mnogo cjelevitiji ali još uvijek traži popunu nekih nedefiniranih mesta kako bi se prilagodio specifičnoj primjeni.



Primjeri radnih okvira

- Radni okvir za obradu plaća.
- Radni okvir za rukovanje čestim kupcima.
- Radni okvir za upis i odabir predmeta na fakultetu.
- Radni okvir za komercijalno web sjedište (e-dućan).
- Radni okvir za sigurno slanje e-pošte

ARHITEKTURA KLIJENT – POSLUŽITELJ



Raspodijeljeni sustavi



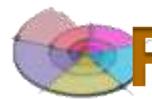
- engl. *Distributed Systems*
- Značajke raspodijeljenih sustava:
 - Obradu podataka i izračunavanja obavljaju **odvojeni programi**.
 - Uobičajeno je da su ti odvojeni programi na odvojenom sklopolju (računalima, čvorovima, stanicama).
 - Odvojeni programi **međusobno komuniciraju** računalnom mrežom.
- Primjer raspodijeljenog sustava:
- **Klijent - poslužitelj:**
 - **Poslužitelj (engl. server)**
 - Program koji dostavlja uslugu drugim programima koji su spojeni na njega preko komunikacijskog kanala.
 - **Klijent (engl. client):**
 - Program koji pristupa poslužitelju (ili više njih) tražeći uslugu
 - Poslužitelju mogu pristupiti mnogi klijenti istovremeno



Primjeri raspodijeljenih sustava



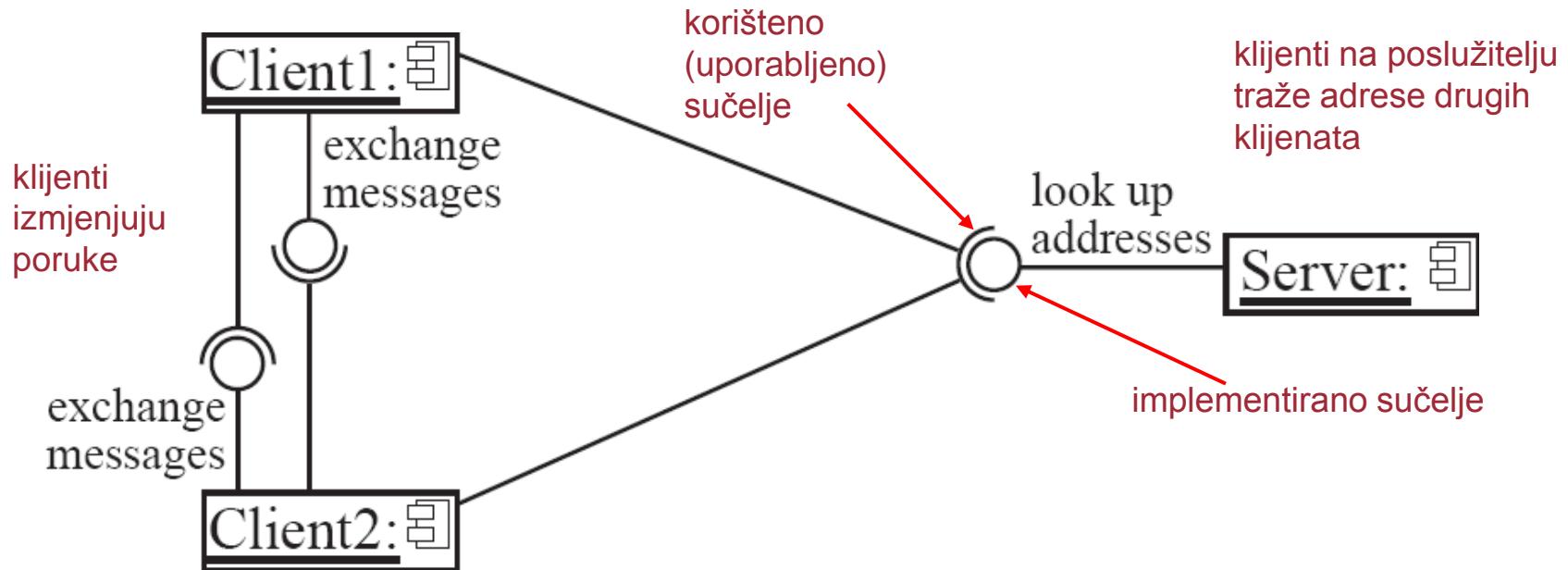
- Sustavi ravnopravnih sudionika (engl. *Peer-to-Peer*)
 - svaki čvor u sustavu ima jednake mogućnosti i odgovornosti (čvor je istovremeno poslužitelj i klijent).
 - snaga obrade podataka i izračunavanja u P2P sustavu ovisi o pojedinim krajnjim čvorovima, a ne o nekom skupnom radu čvorova.
- Srodne socijalne mreže (engl. *affinity communities*)
 - jedan korisnik se povezuje s drugim korisnikom u cilju razmjene informacija (MP3, video, slike itd.).
- Kolaborativno izračunavanje (engl. *collaborative computing*)
 - neiskorišteni resursi (npr. CPU vrijeme, prostor na disku) mnogih računala u mreži kombiniraju se u izvođenju zajedničkog zadatka (GRID computing, SETI@home, ...).
- Slanje poruka u stvarnom vremenu (engl. *Instant Messaging*)
 - izmjena tekstualnih poruka između korisnika u stvarnom vremenu.
- Upravljanje automobilom



- Kako se alociraju i pokreću funkcije poslužitelja?
- Kako se definiraju i šalju parametri između klijenta i poslužitelja?
- Kako se rukuje neuspjesima (pogreškama) u komunikaciji?
- Kako se postavlja i rukuje sa sigurnošću?
- Kako klijent pronađe poslužitelja?
- Koje strukture podataka koristiti i kako rukovati s njima?
- Koja su ograničenja u istovremenom radu dijelova raspodijeljenog sustava?
- Kako se uopće skupina komponenata usuglašava oko zajedničkih pitanja?

Primjer

- Primjer raspodijeljenog sustava u koji omogućava izravnu komunikaciju klijenata.





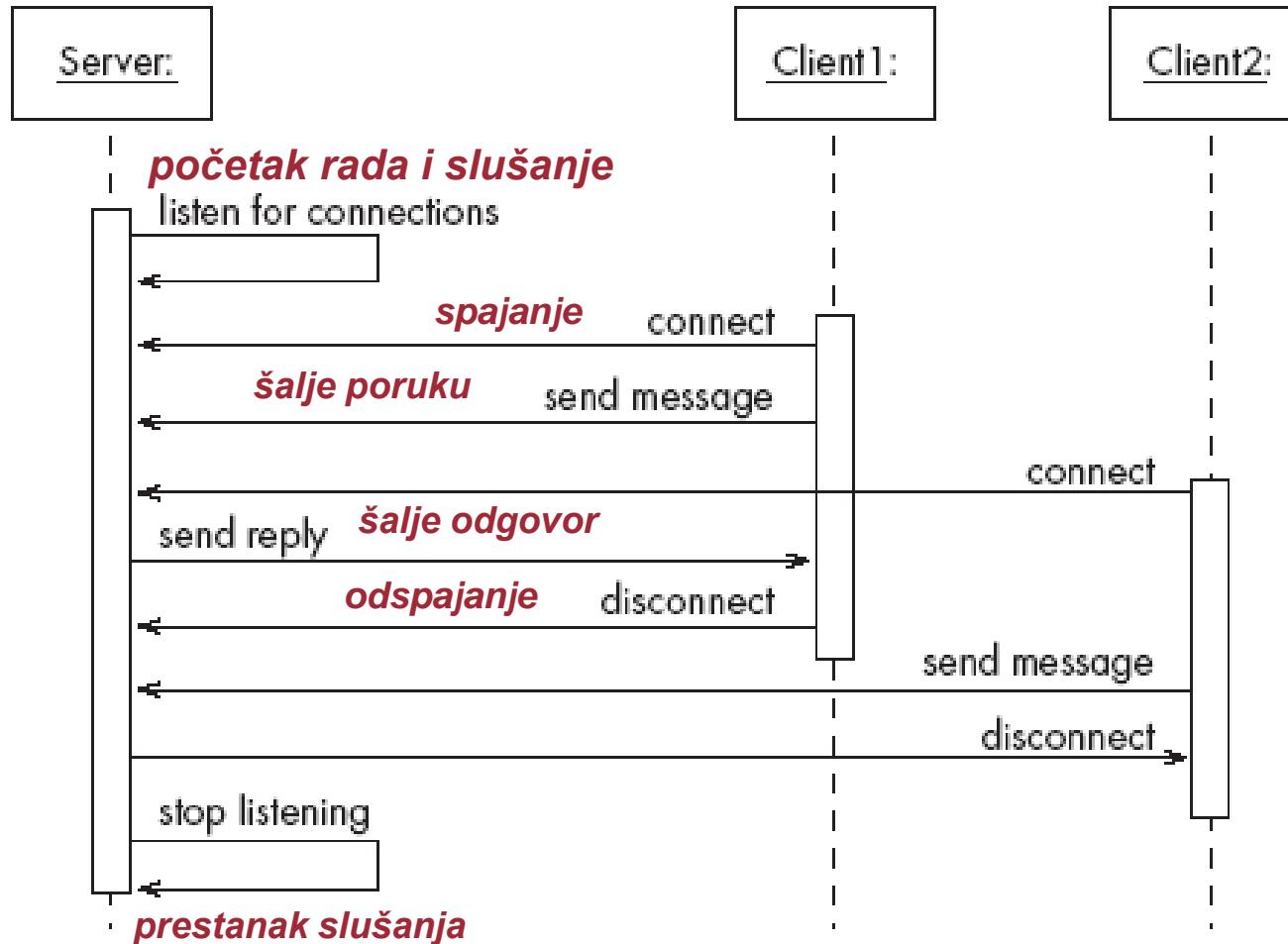
Arhitektura klijent – poslužitelj



- Sekvenca aktivnosti:
 1. Poslužitelj započinje s radom
 2. Poslužitelj čeka na dolazak klijentskog zahtjeva
 - poslužitelj sluša
 3. Klijenti započinju s radom i obavljaju razne operacije,
 - neke operacije traže zahtjeve i odgovore s poslužitelja
 4. Kada klijent pokuša spajanje na poslužitelja, poslužitelj mu to omogući (ako su ispunjeni uvjeti/želi)
 5. Poslužitelj čeka na poruke koje dolaze od spojenih klijenata
 6. Kada pristigne poruka nekog klijenta poslužitelj poduzima akcije kao odziv na tu poruku
 7. Klijenti i poslužitelj nastavljaju s navedenim aktivnostima sve do odspajanja ili prestanka rada



Poslužitelj u komunikaciji s dva klijenta





Alternative klijent – poslužitelj arhitekture



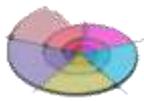
- Implementacija jednog programa na jednom računalu koji obavlja sve (tzv. monolitna programska potpora)
- Računala nisu spojena u mrežu, već svako računalo obavlja svoj posao odvojeno
- Ostvariti neki drugi mehanizam (osim klijent-poslužitelj) kako bi računala u mreži razmjenjivala informacije
 - npr. jedan program upisuje u bazu podataka a drugi čita



Prednosti klijent – poslužitelj arhitekture



- Posao se može *raspodijeliti* na više računala (strojeva)
- Klijenti *udaljeno pristupaju* funkcionalnostima poslužitelja
- Klijent i poslužitelj mogu se *oblikovati odvojeno*
- Oba entiteta mogu biti *jednostavnija*
- Svi podaci mogu se držati na *jednom mjestu* (na poslužitelju)
- Obrnuto, podaci se mogu *distribuirati* na više udaljenih klijenata i poslužitelja
- Poslužitelju može *istodobno* pristupiti više klijenata
- Klijenti mogu ući u *natjecanje* za uslugu poslužitelja (a i obrnuto)



Rizici klijent – poslužitelj arhitekture



- Posebice značajni obzirom na primjenu i održavanje:
- Sigurnost
 - sigurnost je veliki problem bez savršenog rješenja
 - Potrebno uporabiti enkripciju, zaštitne zidove (*engl. firewalls*) i sl.
- Potreba za adaptivnim održavanjem
 - budući da se programska potpora za klijente i poslužitelja oblikuje odvojeno potrebno je osigurati da sva programska potpora bude
 - kompatibilna prema unatrag (*engl. backwards*),
 - prema unaprijed (*engl. forwards*),
 - te kompatibilna s drugim verzijama klijenata i poslužitelja.



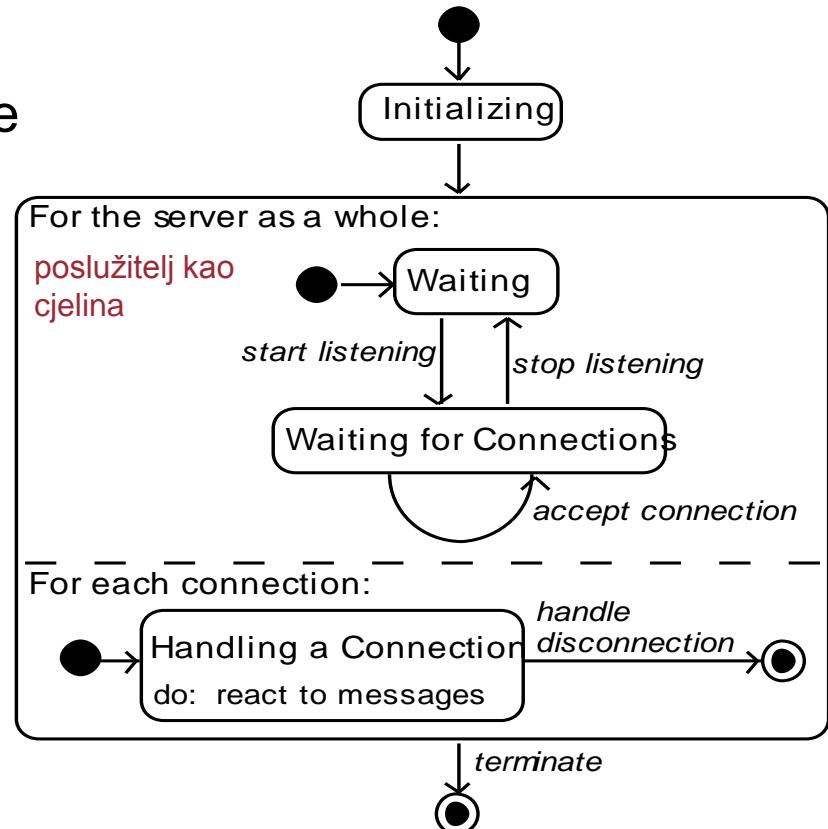
Primjeri arh. klijent – poslužitelj



- The World Wide Web
- E-mail
- Network File System -NFS
- Transaction Processing System
- Remote Display System
- Communication System
- Database System
-
- Poslužitelji:
 - Application Servers
 - Audio/Video Servers
 - Chat Servers
 - Fax Servers
 - FTP Servers
 - IRC Servers
 - Mail Servers
 - News Servers
 - Proxy Servers
 - Web Servers
 - Z39.50 Servers

Funkcionalnosti poslužitelja

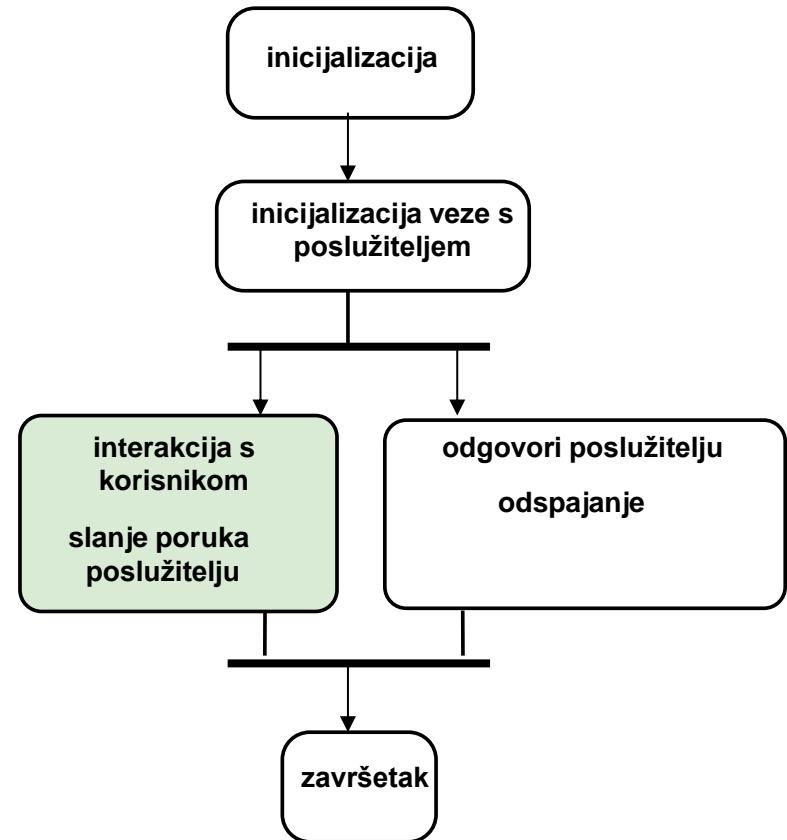
- Inicijalizacija poslužitelja
- Započinje slušati klijentska spajanja
- Rukuje sljedećim tipovima događaja koje potiču klijenti:
 1. Prihvaća spajanje
 2. Odgovara na poruke
 3. Rukuje odspajanjem klijenta
- Može prestati slušati
- Mora čisto završiti rad!



za svako spajanje rukuje spajanjem, reagira na poruku

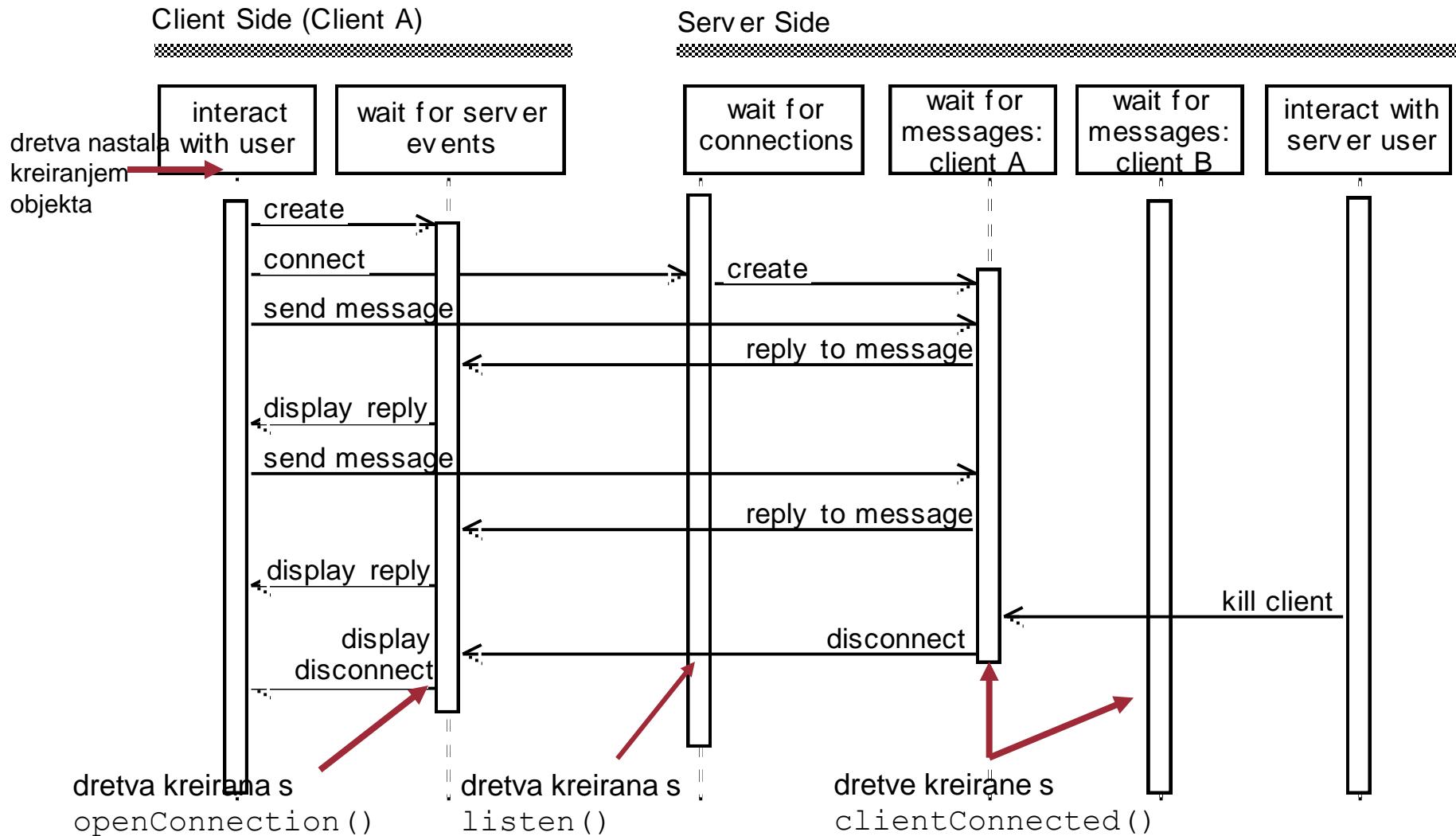
Funkcionalnosti klijenta

- Inicijalizira klijenta
- Inicijalizira spajanje na poslužitelja
- Šalje poruke
- Rukuje sljedećim tipovima događaja koje potiče poslužitelj:
 - odgovara na poruke
 - rukuje odspajanjem od poslužitelja
- Mora čisto završiti rad



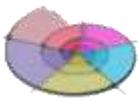
Sekvencijski dijagram

- Paralelne dretve/niti izvođenja u sustavu klijent - poslužitelj



Komunikacijski protokoli

- Poruke koje klijenti šalju poslužitelju formiraju jedan jezik
 - poslužitelj mora biti programiran da razumije taj jezik
- Poruke koje poslužitelj šalje klijentima također formiraju jedan jezik
 - klijenti moraju biti programirani da razumiju taj jezik
- Kada klijent i poslužitelj komuniciraju oni razmjenjuju poruke uporabom ta dva jezika
- Ta *dva jezika i pravila konverzacije* čine zajedničkim imenom **protokol**



Primjer: Internet protokoli



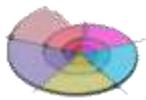
- Internet Protocol (IP)
 - određuje put poruka od jednog računala do drugog
 - dugačke poruke se cijepaju u manje dijelove
- Transmission Control Protocol (TCP)
 - nalazi se između primjenskog programa ("aplikacije") i IP protokola
 - razbija veliku poruku na manje dijelove i šalje dijelove uporabom IP protokola
 - osigurava da je poruka uspješno primljena (pojedini IP paketi ispravno sastavljeni)
 - TCP je pouzdan protokol
- Svako računalo (čvor) u mreži ima jedinstvenu IP adresu i ime (engl. *host name*)
 - nekoliko poslužitelja mogu raditi na jednom čvornom računalu u mreži
 - svaki poslužitelj na računalu je identificiran preko jedinstvenog broja **ulaznog porta** (engl. *port number*) u rasponu 0 to 65535
 - kako bi započeo komunikaciju s poslužiteljem klijent mora znati host name i port number
 - brojevi ulaznih portova 0 – 1023 su rezervirani (npr. port 80 za Web poslužitelj)

Smijete li vi upotrijebiti taj port za nešto drugo?



- Preporuke:
 - Oblikuj temeljne poslove poslužitelja i klijenta
 - Odredi kako će se posao raspodijeliti
 - tanki nasuprot debelog klijenta
 - Oblikuj detalje skupa poruka koje se razmjenjuju
 - komunikacijski protokol
 - Oblikuj mehanizme:
 - Inicijalizacije
 - Rukovanja spajanjima
 - Slanja i primanja poruka
 - Završetka rada
- U oblikovanju koristi princip “**Povećaj uporabu postojećeg**” (princip br. 6)

OBJEKTNI RADNI OKVIR KLIJENT- POSLUŽITELJ



Objektni radni okvir klijent-poslužitelj



- engl. *Object Client-Server Framework - OCSF*
- Metoda oblikovanja arhitekture klijent-poslužitelj temeljena na ponovnoj i višestrukoj uporabi komponenata (engl. *reuse*)
- Pravila uporabe:
 - ne mijenjati apstraktne razrede u OCSF;
 - kreirati podrazrede;
 - konkretizirati metode u podrazredima;
 - ponovo definirati (engl. *override*) neke metode u podrazredima;
 - napisati kod koji kreira instance i inicira akcije.
- Izvorni kod OCSF u Javi nalazi se na web stranicama knjige
 - T.C.Lethbridge, R.Laganiere: Object-Oriented Software Engineering, 2nd ed., McGraw-Hill, 2005.
 - **potrebno proučiti :**
<http://www.site.uottawa.ca/school/research/lloseng/supportMaterial/source/>

Programski jezik Java

- Uporaba *java.net* paketa
 - implementacija TCP/IP veze
- Za uspostavu veze nužno da poslužitelj bude spreman za osluškivanje na definiranom portu:

```
ServerSocket serverSocket = new ServerSocket(port);  
Socket clientSocket = serverSocket.accept();
```

- Ostvarenje veze klijenta s poslužiteljem :

```
Socket clientSocket= new Socket(host, port);
```

- Razmjena informacija
 - uporaba paketa *java.io* (*InputStream*, *OutputStream*)

```
output = new OutputStream(clientSocket.getOutputStream());  
Input = new InputStream(clientSocket.getInputStream());
```

Razmjena poruka

■ Izravno

```
output.write(msg);  
msg = input.read();
```

Formatirani podatci *DataInputStream/DataOutputStream*

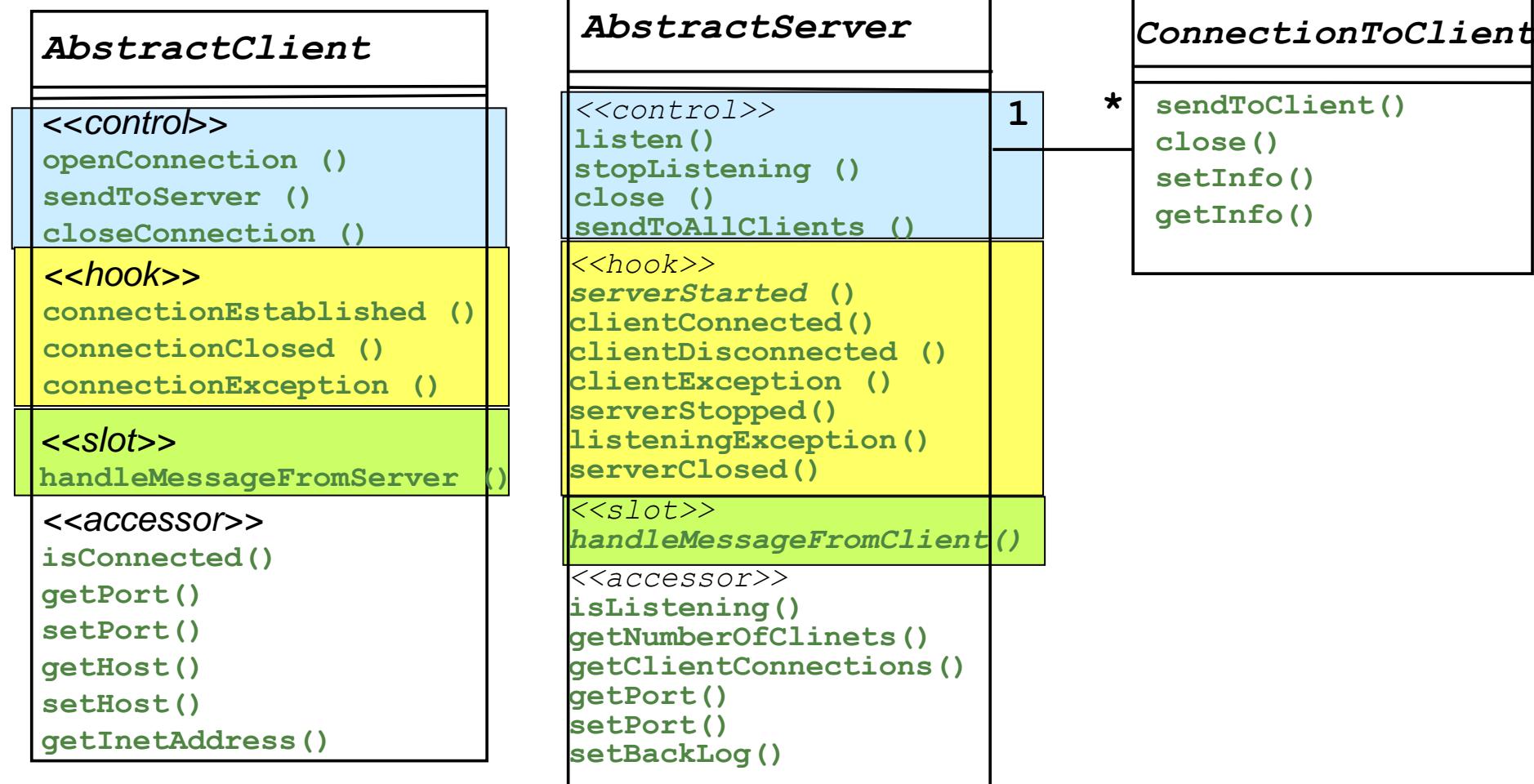
```
output.writeDouble(msg);  
msg = input.readDouble();
```

Objekti *ObjectInputStream / ObjectOutputStream*

```
output.writeObject(msg);  
msg = input.readObject();
```



Object Client-Server Framework (OCSF)





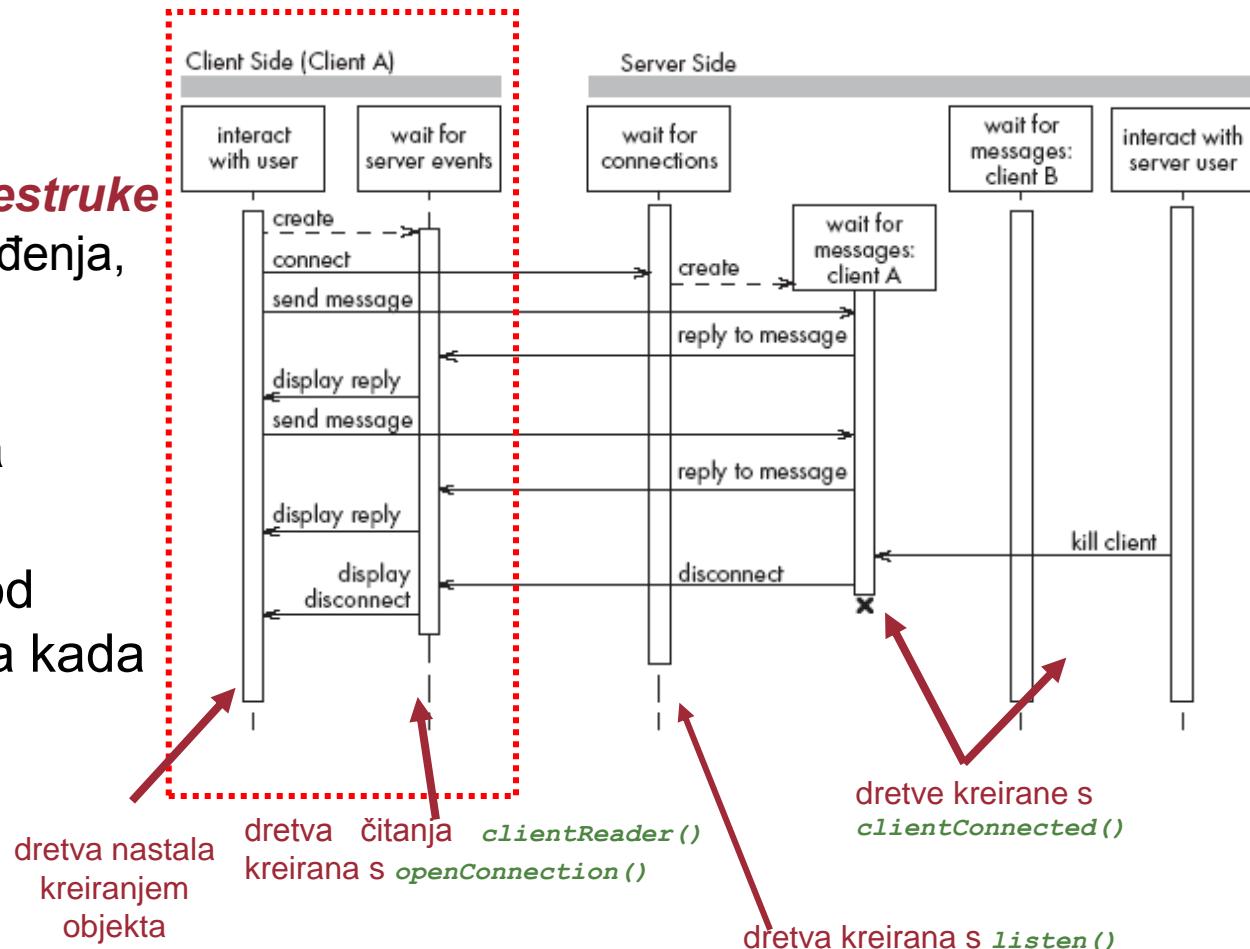
Uporaba objektnih radnih okvira

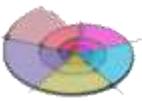


- Programski inženjeri u uporabi radnih okvira ***nikada ne modificiraju ključne razrede***
 - u primjeru OCSF-a: *AbstractClient*, *AbstractServer*,
ConnectionToClient
- Umjesto modifikacije izvornih razreda treba:
 - kreirati podrazrede apstraktnih razreda u radnom okviru i implementirati metode
 - zvati javne metode koje uključuje radni okvir
 - te metode su usluge koje pruža radni okvir.
 - redefinirati neke metode posebno označene u kategorije <<*hook*>> i <<*slot*>> koje su eksplicitno namijenjene da budu redefinirane.
- Zašto?
 - implementirane (konkretne) metode su rigorozno provjerene (nadamo se)
 - nisu predviđene za redefiniranje

Klijent

- engl. *Client Side*
- paralelne aktivnosti na klijentskoj strani – implementirane kao **višestruke dretve izvođenja** (nit izvođenja, engl. *thread*)
- čekanje na interakciju s korisnikom i odgovor na interakciju
 - čekanje na poruku od poslužitelja i reakcija kada poruka stigne





OO Klijent



Sadrži jedan apstraktan razred: ***AbstractClient***

- iz razreda ***AbstractClient*** moraju se izvesti
 - svaki podrazred mora osigurati implementaciju metode:
handleMessageFromServer()
 - navedena u skupu <<slot>> oznake
 - zadužena za odgovarajuću akciju po primitku poruke od poslužitelja
- ***AbstractClient*** implementira rad s poslužiteljem.
 - za čitanje poruka postoji posebna dretva instance (objekta) - vidi sekvensijski dijagram.
 - dretva započinje izvođenje nakon što upravljačka metoda *openConnection()* pozove *start()* dretve naziva ***clientReader***
 - ona pokreće *run()* metodu (glavni program za dretvu).
 - *run()* metoda sadrži petlju koja se izvodi tijekom životnog ciklusa dretve
 - prima poruke i zove metodu za rukovanje s porukama

Dretve u Javi

- Svaka dretva mora imati **run** metodu

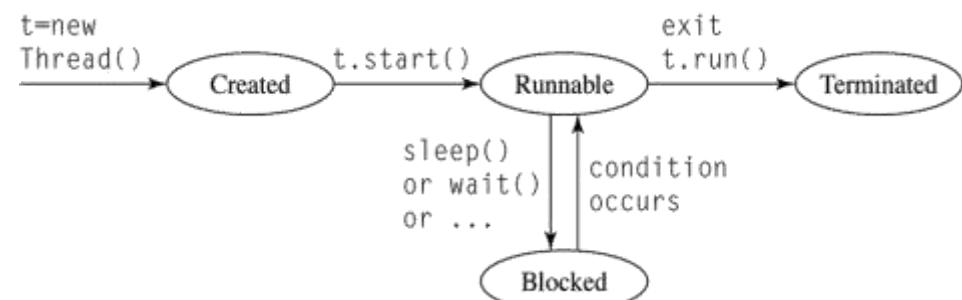
- u Javi dretva je objekt razreda Thread i uvijek se mora stvoriti:

dretva = new Thread();

- započinje pozivom metode **start**

dretva.start()

- ona aktivira **run()** metodu koja implementira njezine funkcionalnosti
- završava završetkom **run** metode



Primjer: Dretva ReadThread

```
public class ReadThread implements Runnable
{
    ...
    final public void run()
    {
        object msg;
        clientSocket= new Socket(host, port);
        output = new ObjectOutputStream(clientSocket.getOutputStream());
        input = new ObjectInputStream(clientSocket.getInputStream());
        while(!readyToStop)
        {
            msg = input.readObject();
            handleMessageFromServer(msg);
        }
    }
}
```



Klijentska strana - apstraktan razred



AbstractClient

```
<<control>>
openConnection ()
sendToServer ()
closeConnection ()

<<hook>>
connectionEstablished ()
connectionClosed ()
connectionException ()

<<slot>>
handleMessageFromServer ()

<<accessor>>
isConnected()
getPort()
setPort()
getHost()
setHost()
getInetAddress()
```

- mora imati podrazred koji:
implementira
handleMessageFromServer ()
- obavlja akcije na poruke pristigle od poslužitelja
- implementira kod za pokretanje dretvi
 - u Javi *Runnable* interface

```
public class Test implements
Runnable {
```

```
    public void run () {
```

```
        ....
```

```
    }
```

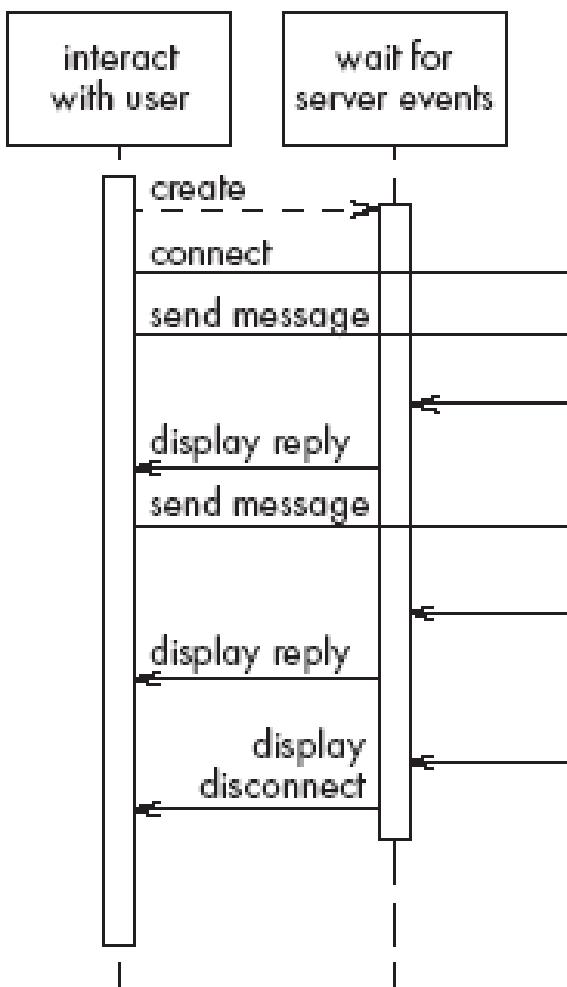
```
}
```
- izvodi *run ()* metodu za vrijeme života dretve



Sekvencijski dijagram klijenta



Client Side (Client A)



```
main() // korisničko sučelje - GUI, konzola
Ui = new ClientConsole()

ClientConsole()
c = new ChatClient() //instanciraj klijenta
ChatClient()
this.openConnection()

openConnection() //otvorи komunikacijski kanal
run()
Loop:

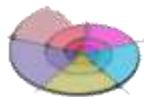
Msg = input.ReadObject()
handleMessageFromServer(msg)

handleMessageFromServer ()
Ui.display(msg)

Ui.Accept() // čekaj upis s konzole
accept()
msg = fromConsole.readLine();
c.handleMessageFromClientUI(msg)

handleMessageFromClientUI()
sendToServer(msg)

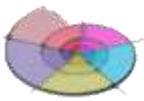
sendtoserver()
output.writeObject(msg)
```



Uporaba razreda *AbstractClient*



1. Kreiraj podrazred od *AbstractClient*
2. U podrazredu implementiraj *handleMessageFromServer <<slot>>* metodu
3. Napiši kod koji:
 - kreira instancu novoga podrazreda (start prve dretve)
 - pozove *openConnection* (start druge dretve)
 - šalje poruku poslužitelju uporabom *sendToServer* metode
4. Implementiraj *connectionEstablished* povratnu metodu
 - npr. izvijesti korisnika
5. Implementiraj *connectionClosed* povratnu metodu
 - npr. izvijesti korisnika o čistom završetku veze
6. Implementiraj *connectionException* povratnu metodu
 - npr. kao odziv na kvar na mreži

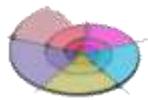


Klijentska strana



AbstractClient se spaja i šalje poruke poslužitelju.

- Za čitanje poruka postoji posebna dretva instance (objekta) - vidi sekvencijski dijagram.
 - dretva započinje izvođenje nakon što upravljačka metoda *openConnection()* pozove *start()* dretve *clientReader* koja pokreće njenu *run()* metodu.
 - *run()* metoda sadrži petlju koja se izvodi tijekom životnog ciklusa dretve
 - prima poruke i zove metodu za rukovanje s porukama



Klijentska strana - *openConnection*



- Otvaranje veze i komunikacije input - output

```
final public void openConnection() {  
    // ne redefinirati  
    // kreiraj socket i data streams (odlazne i povratne poruke)  
clientSocket= new Socket(host, port);  
output = new ObjectOutputStream(clientSocket.getOutputStream());  
input = new ObjectInputStream(clientSocket.getInputStream());  
    // kreiraj dretvu čitanja poruka s poslužitelja  
clientReader = new Thread(this);  
    // makni zabranu  
readyToStop = false;  
    // startaj dretvu koja aktivira run metodu  
clientReader.start(); }
```

Gdje je definiran start?

Klijentska strana – run

```
public class ReadThread implements Runnable
{
    ...
final public void run() {
    object msg;           // varijabla za poruku
    try {
        while(!readyToStop) {
            // dohvati poruke s poslužitelja i poziv metode za
            // obradu poruke
            // dretva čeka poruku na sljedećoj naredbi
            try{ msg = input.readObject();
                // i zove obavezno implementiranu metodu
                // handleMessageFromServer za obradu poruke
                if (!readyToStop) {
                    handleMessageFromServer(msg);
                }
            }
            catch (Exception exception) { ... } }
    }
}
```

U ovom skraćenom prikazu nije prikazana obrada iznimke u `catch` bloku `try-catch` za manipulaciju iznimkama: `catch (Exception ex) {}`

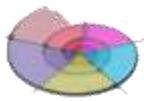
Slanje poruka poslužitelju

- Nije funkcija `clientReader` dretve !

```
public void sendToServer(Object msg)
{
    output.writeObject(msg);
    // writeObject je definiran u razredu
    // ObjectOutputStream
    // output preko clientSocket zna kome šalje
}

final public void closeConnection() {
    // zaustavljanje dretve
    readyToStop = true;
    try {
        // closeAll() je implementirana u radnom okviru
        closeAll();
    }
    finally {
        // poziv metode
        connectionClosed();
    }
}
```

```
private void closeAll() {
    try {
        // zatvori socket
        if (clientSocket != null)
            clientSocket.close();
        // zatvori output stream
        if (output != null)
            output.close();
        // zatvori input stream
        if (input != null)
            input.close();
    }
    finally {
        output = null;
        input = null;
        clientSocket = null;
    }
}
```



Privatni dijelovi razreda AbstractClient



- Osobe zadužene za oblikovanje programske potpore ne moraju znati te detalje, ali može pomoći u razumijevanju.
- Varijable instanci:
 - *clientSocket* (tipa *Socket*)
 - drži sve informacije o vezi s poslužiteljem.
 - dva niza tipa *ObjectOutputStream* i *ObjectInputStream* koriste za slanje i prijam objekata uporabom varijable *clientSocket*.
 - *clientReader* tipa *Thread*
izvodi se *run* metodom objekta razreda *AbstractClient*. Dretva započinje kada *openConnection* pozove *start* koja pozove *run*. Petlja unutar *run* čeka na poruku koja dolazi od poslužitelja. Kada je poruka primljena, *run* zove metodu *handleMessageFromServer*.
- varijabla *boolean readyToStop*
 - za signalizaciju zaustavljanja dretve čitanja poruka servera.
 - dvije varijable koje čuvaju *host* i *port* adresu poslužitelja.

Klijentska strana - *AbstractClient*

Varijable instance:

```
// kanal za komunikaciju operacijskog  
// sustava  
private Socket clientSocket;  
// nizovi za manipulaciju izlaza - ulaza  
private ObjectOutputStream output;  
private ObjectInputStream input;  
// dretva čitanja  
private Thread clientReader;  
// indikacija kada dretva može završiti  
private boolean readyToStop;  
// ime i broj porta poslužitelja  
private String host;  
private int port;
```

Javno sučelje

- Konstruktor ***AbstractClient*** pri stvaranju objekta inicijalizira ***host*** i ***port*** varijable na koje će se klijent spojiti.

Upravljačke (<<***control***>>) metode (dekl. ***final***, ne redefinirati):

openConnection (spaja se na poslužitelj, koristi ***host*** i ***port*** varijable koje postavlja konstruktor ili može koristiti metode ***setHost***, ***setPort***, uspješna veza starta dretvu)

- ***sendToServer*** (šalje poruku koja može biti bilo koji objekt)
- ***closeConnection*** (zaustavlja rad dretve u petlji i završava)

Pristupne (<<***accessor***>>) metode daju informaciju ili mijenjaju vrijednost

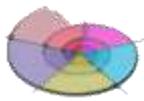
- ***isConnected*** (ispituje da li je klijent spojen)
- ***getHost*** (ispituje koji ***host*** je spojen)
- ***setHost*** (omogućuje promjenu ***hosta*** dok je klijent odspojen)
- ***getPort*** (ispituje na koji ***port*** je klijent spojen)
- ***setPort*** (omogućuje promjenu ***porta*** dok je klijent odspojen).
- ***getInetAddress*** (dobavlja neke detaljnije informacije o vezi)

Konstruktor za *AbstractClient*:

```
public AbstractClient(String host, int port)
{
    // inicijalizacija varijabli objekta-klijenta
    // host i port su adrese poslužitelja
    this.host= host;
    this.port= port;
}
```

Uporaba razreda *AbstractClient*

```
public class MyClient extends AbstractClient {  
    . . . // varijable instanci  
    // konstruktor instance  
    public MyClient(String host, int port) {  
        super(host, port); // konstruktor superrazreda  
        openConnection();  
    .  
    . // metode instance  
    .  
    // glavna konkretizirana metoda  
    public void handleMessageFromServer(Object msg) {  
        .  
        // obrada poruke  
        . . .  
    }  
}
```



Povratne metode u *AbstractClient*



- engl. *Callback*
- Metode koje se **mogu** redefinirati
 - označene su kao skupina «*hook*»
 - ne zovu se izravno
 - npr. *u C++ zovu se preko pokazivača*
 - najčešće se povratne metode zovu kao odziv na neki asinkroni događaj
 - npr. *sva moderna grafička korisnička sučelja zasnovana su na povratnim metodama*
 - ako podrazred ima namjeru poduzeti neke akcije kao odziv na događaj
 - ***connectionEstablished*** (poziva se po uspostavi veze s poslužiteljem)
 - ***connectionClosed*** (poziva se nakon završetka veze)
 - ***connectionException*** (poziva se kada nešto pođe krivo, npr. poslužitelj prekida vezu).

Metoda koja se **mora** implementirati (najvažniji dio koda):
handleMessageFromServer()

- definira se u podrazredima i pozove kada je primljena poruka od poslužitelja

Uporaba razreda *AbstractClient*

Povratne metode :

```
// Aktivira se kad je uspostavljena veza  
// "Default" implementacija ne čini ništa  
// Može se ponovo redefinirati ako potrebno  
protected void connectionEstablished() {}  
// slično za završetak veze  
protected void connectionClosed() {}  
// kao i za obradu iznimke  
protected void connectionException(Exception exception) {}
```



Poslužiteljska strana

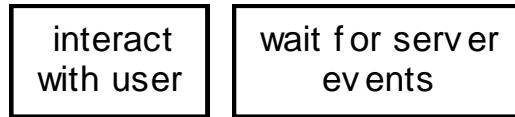
- Poslužiteljska strana sadrži dva razreda
- Potrebne su dvije dretve:
 - jedna koja sluša novo spajanje klijenta,
 - druga ili više njih koje rukuju vezama s klijentima
 - jedna ili više njih
 - sekvencijski dijagram
- Dodatna dretva je potrebna za interakciju s korisnikom na poslužitelju
- Implementacija:
 - instanca razreda ***AbstractServer***
 - sluša novo spajanje klijenta
 - rukuje s porukom.
 - ne šalje poruku pojedinom klijentu
 - to čini instanca od ***ConnectionToClient()***
 - može poslati istu poruku svim spojenim klijentima metodom ***sendToAllClients()***
- jedna ili više instanci razreda ***ConnectionToClient***
 - rukuje vezana s klijentima



Paralelne dretve klijent - poslužitelj



Client Side (Client A)



create
connect
send message

display reply
send message

display reply
display disconnect

Server Side



create
reply to message

reply to message

disconnect

kill client

dretva
listen()

kreirana

s

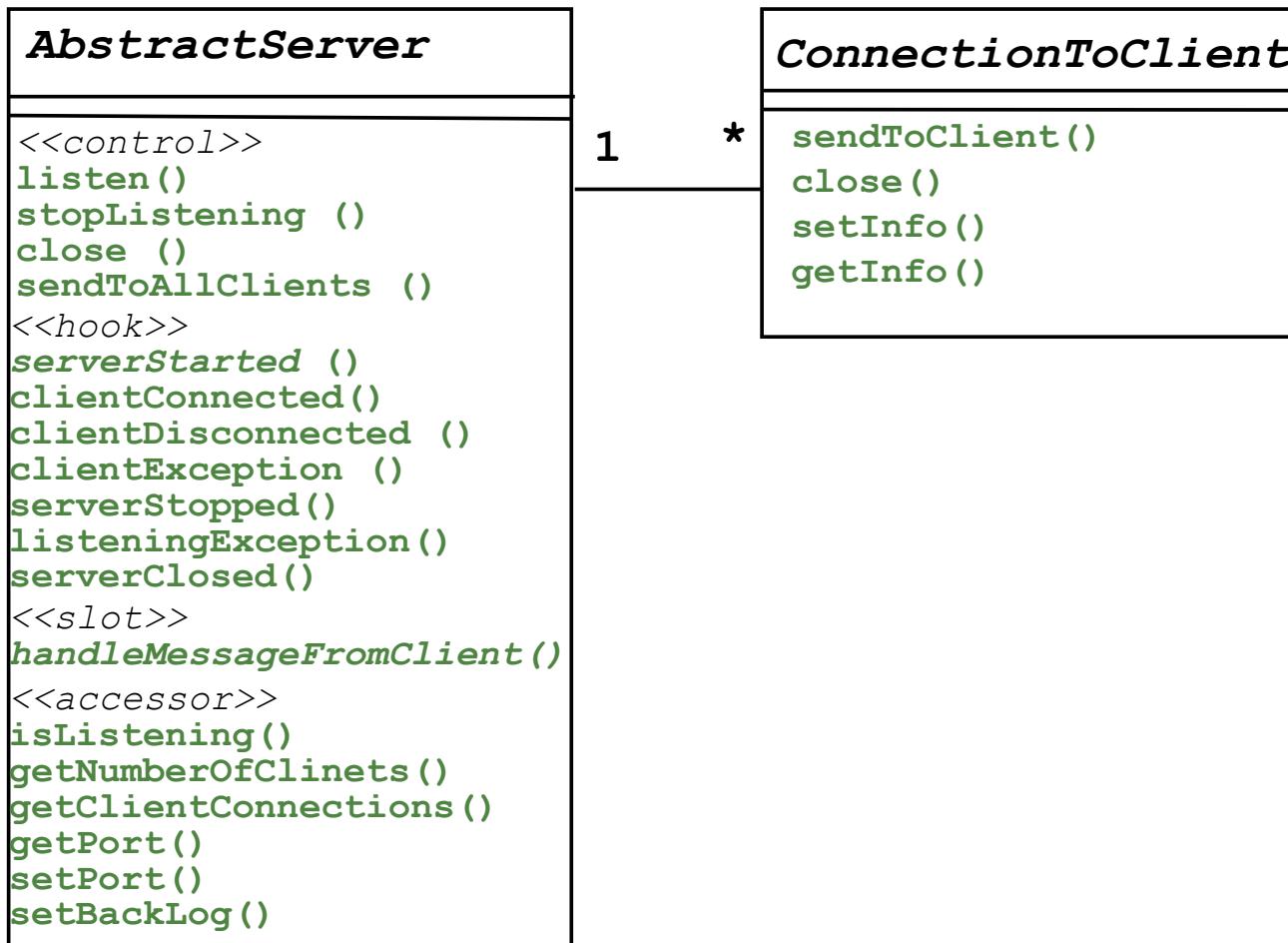
Dretve
ConnectionToClient()

kreirane

kada

i

Poslužiteljska strana





Javno sučelje *AbstractServer*



■ Konstruktor *AbstractServer*:

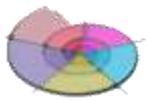
- stvara objekt s brojem porta na kojem poslužitelj sluša. Kasnije se može promijeniti sa *setPort*.

Upravljačke (`<<control>>`) metode:

- *listen* (kreira varijablu *serverSocket* tipa *Socket* koja će slušati na portu specificiranom konstruktorom, inicira dretvu, a *run* metoda čeka na spajanje klijenta) – vidi raniji sekvencijski dijagram.
- *stopListening* (signalizira *run* metodi da prestane slušati. Spojeni klijenti i dalje komuniciraju).
- *close* (kao *stopListening* ali odspaja sve klijente)
- *sendToAllClients* (šalje poruku svim spojenim klijentima)!

■ Pristupne (`<<accessor>>`) metode:

- *isListening* (vraća da li poslužitelj sluša)
- *getClientConnections* (može se iskoristiti za neki rad sa svim klijentima)
- *getPort* (vraća na kojem portu poslužitelj sluša)
- *setPort* (koristi se za sljedeći *listen()*, nakon zaustavljanja)
- *setBacklog* (postavlja veličinu repa čekanja)



Povratne metode u *AbstractServer*



- Metode koje **mogu** biti redefinirane
 - ukoliko su konkretni podrazredi zainteresirani za poseban odziv na događaj).
 - **serverStarted** (poziva se kada poslužitelj započinje prihvati spajanja).
 - **clientConnected** (poziva se kada se novi klijent spoji, sadrži instancu razreda **ConnectionToClient** kao argument) – vidi OCSF.
 - **clientDisconnected** (poziva se kada poslužitelj odspoji klijenta, argument je instanca razreda **ConnectionToClient**) .
 - **clientException** (poziva se kada se klijent sam odspoji ili kada nastupi kvar u mreži)
 - **serverStopped** (poziva se kada poslužitelj prestane prihvati povezivanje s klijentima a kao rezultat **stopListening**).
 - **listeningException** (poziva se kada poslužitelj prestane slušati zbog nekog kvara).
 - **serverClosed** (poziva se nakon završetka rada poslužitelja).
- Metoda koja se **mora** implementirati (najvažniji dio koda):
handleMessageFromClient (argumenti su primljena poruka te instanca razreda **ConnectionToClient**).



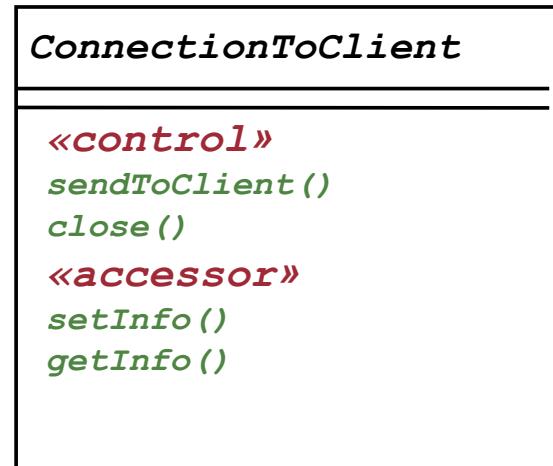
Javno sučelje *ConnectionToClient*

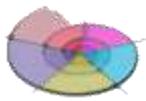


■ Tko ga stvara?

Početna aktivacija metode *listen()* u *AbstractServer* pokreće dretvu koja u svojoj *run()* metodi preko metode *accept()* "socketa" poslužitelja stvara objekt razreda *ConnectionToClient* (s pripadnom dretvom)

- po jedan za svako spajanje klijenta





- Za svakog klijenta postoji instanca razreda *ConnectionToClient* za vrijeme dok je klijent spojen na poslužitelja.
- *connectionToClient* je konkretni razred.
 - korisnici ne moraju kreirati podrazrede.

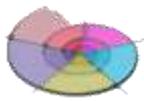
Upravljačke (<<*control*>>) metode:

- *sendToClient* (središnja metoda, koristi se za komunikaciju s klijentom).
- *close* (odspaja klijenta)

Pristupne (<<*accessor*>>) metode:

- *getInetAddress* (dobavlja Internet adresu klijenta)
- *setInfo* (omogućuje spremanje proizvoljnih informacija o klijentu. Npr. posebne privilegije)
- *getInfo* (omogućuje čitanje informacija o klijentu).

1. Kreiraj podrazred od ***AbstractServer*** razreda.
2. U podrazredu implementiraj ***handleMessageFromClient***.
3. Napiši kod koji:
 - kreira instancu podrazreda od ***AbstractServer*** poziva ***listen()*** metodu odgovara na "call back" ***clientConnected()*** slanjem poruke uporabom metoda u ovom objektu:
getClientConnections() ili ***sendToAllClients()*** (nisu callback metode)
odnosno metodom ***sendToClient()*** u objektu razreda ***ConnectionToClient***
4. Po potrebi implementiraj jednu ili više drugih povratnih metoda kao odzive na zanimljive događaje.

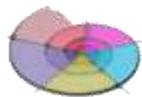


Sinkronizacija rada s više klijenata



- Mnoge metode na poslužiteljskoj strani su sinkronizirane.
Budući da postoji više ***ConnectionToClient*** dretvi, koje mogu istovremenu mijenjati podatke na poslužitelju, sinkronizacija garantira da se kritične operacije odvijaju jedna po jedna, te se tako čuva integritet podataka.
 - u Javi ključna riječ: **synchronized**
- Kolekcija objekata ***ConnectionToClient*** koje održava ***AbstractServer*** smješteni su u poseban Java razred ***ThreadGroup***.
 - taj razred će automatski maknuti instancu kada njena dretva završi
- Poslužitelj mora povremeno (npr. svakih 500ms)
 - provjeriti da li je pozvana metoda ***stopListening()***
 - ako nije, nastavlja s radom.

Zašto?



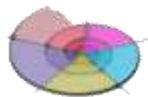
Implementacija *AbstractServer*



- privatni dijelovi s početnim vrijednostima

Varijable instanci

```
// pristupni socket poslužitelja  
private ServerSocket serverSocket = null;  
  
// dretva slušanja na spajanje klijenta  
private Thread connectionListener = null;  
  
// broj porta  
private int port;  
  
// povremeni prekid za provjeru na "stop slušanja"  
private int timeout = 500;  
  
// duljina repa čekanja klijenata na spajanje  
private int backlog = 10;  
  
// grupa dretvi pridružena grupi spojenih klijenata  
private ThreadGroup clientThreadGroup;  
  
// indikacija da li je čitanje spremno za zaustavljanje  
private boolean readyToStop = true;
```



Implementacija *AbstractServer*



Konstruktor:

```
public AbstractServer(int port) {  
    // koji port  
    this.port= port;  
    // struktura za klijente  
    this.clientThreadGroup= new ClientThreadGroup("ConnectionToClient Threads");  
}
```

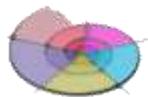
Metode instance:

```
final public void listen() {  
    // definiranje konekcije i start dretve slušanja  
    serverSocket= new ServerSocket(getPort(), backlog);  
    connectionListener= new Thread(this);  
    connectionListener.start(); }  
    // ovaj start pokreće run metodu dretve - vidi nastavak
```

Dijelovi implementacije

Metoda `run()`

```
final public void run() {  
    // poslužitelj započinje s radom  
    readyToStop= false;  
    // poziv callback metode za neku opciju akciju  
    serverStarted();  
    // čekaj na spajanje klijenta  
    while(!readyToStop)  
        socket clientSocket = server.Socket(accept);  
        // kad je klijent spojen, kreiraj novu instancu  
        // ConnectionToClient koja se izvodi kao dretva  
        // vidi konstruktor objekta ConnectionToClient  
        new ConnectionToClient(this.clientThreadGroup, clientSocket, this); }
```

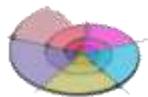


Implementacija *ConnectionToClient*



Varijable instanci:

```
// klijent mora znati za poslužitelja - asocijacija  
private AbstractServer server;  
// komunikacijski kanal klijenta  
private Socket clientSocket;  
// nizovi za čitanje i pisanje  
private ObjectInputStream input;  
private ObjectOutputStream output;  
// indikacija da li je dretva spremna na zaustavljanje  
private boolean readyToStop;
```

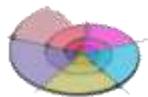


Implementacija *ConnectionToClient*



Konstruktor objekta *ConnectionToClient*:

```
protected ConnectionToClient(ThreadGroup group,  
socket clientSocket, AbstractServer server) {  
// inicijalizacija varijabli ConnectionToClient (bit će pozvan kada se  
neki klijent spoji (varijabla clientSocket dobiva podatke o klijentu).  
  
this.clientSocket = clientSocket;  
this.server = server;  
// inicijalizacija nizova  
input= new ObjectInputStream(clientSocket.getInputStream());  
output= new ObjectOutputStream(client.Socket.getOutputStream());  
// start dretve i run metode, čekanje na podatke... vidi  
nastavak  
readyToStop= false;  
start(); } //aktivira run() metodu
```



Implementacija *ConnectionToClient*



Metoda *run()*:

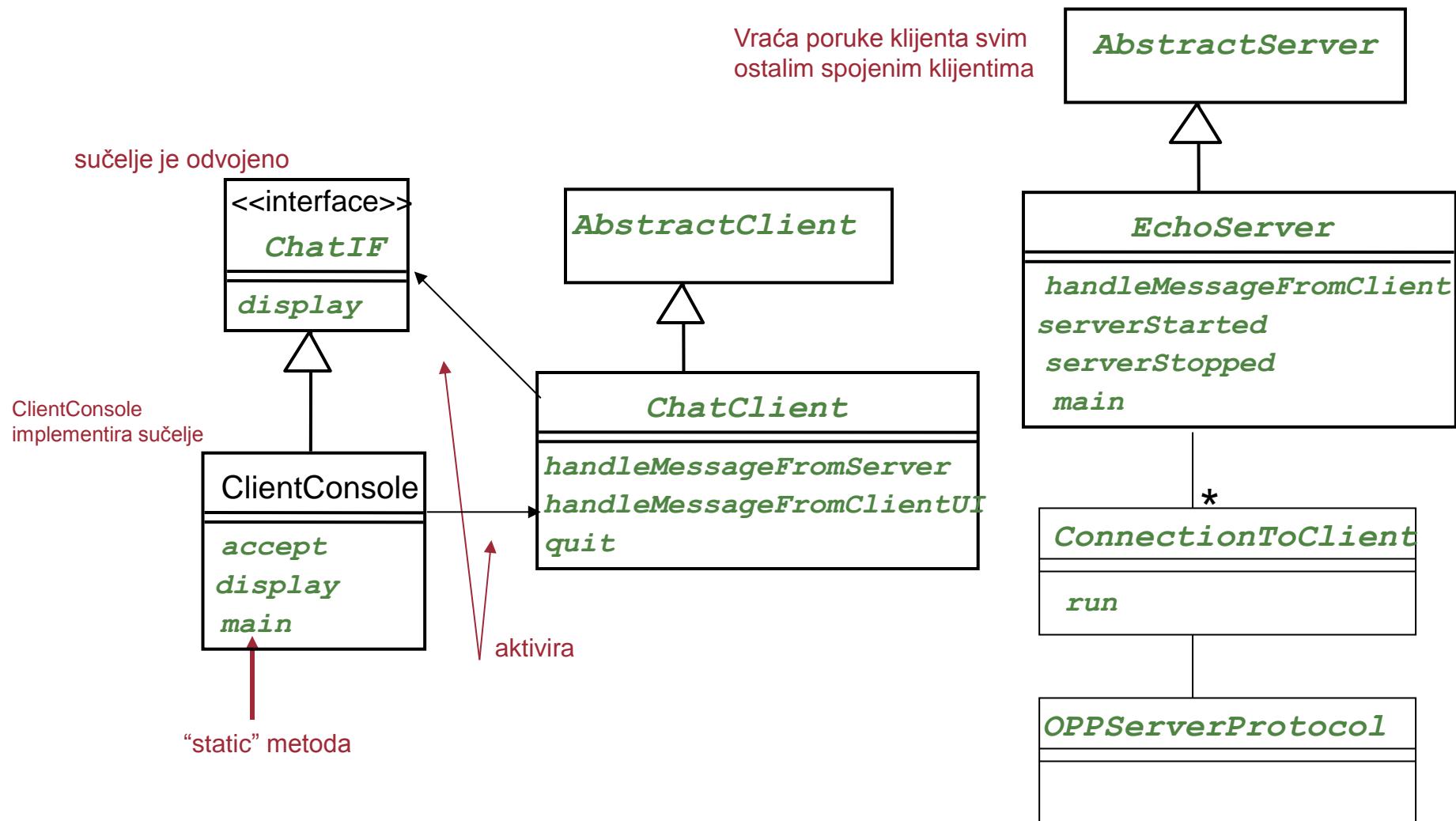
```
final public void run() {  
    // callback u poslužitelju za akciju  
    server.clientConnected(this);  
    // petlja čitanja i slanja poruke poslužitelju  
    Object msg;  
    while(!notReadyToStop) {  
        msg = input.readObject();  
        // manipulacija porukom  
        server.handleMessageFromClient(msg, this);  
    . . . } // razni odzivi na iznimke  
}
```

slanje poruke klijentu (nije dio run() metode):

```
public void sendToClient(Object msg) {  
    output.writeObject(msg) }
```

Primjer OCSF-a : Chat

- Jednostavan klijent-poslužitelj sustav razmjene poruka u stvarnom vremenu.
- Poslužitelj vraća poruku primljenu od nekog klijenta svim spojenim klijentima.

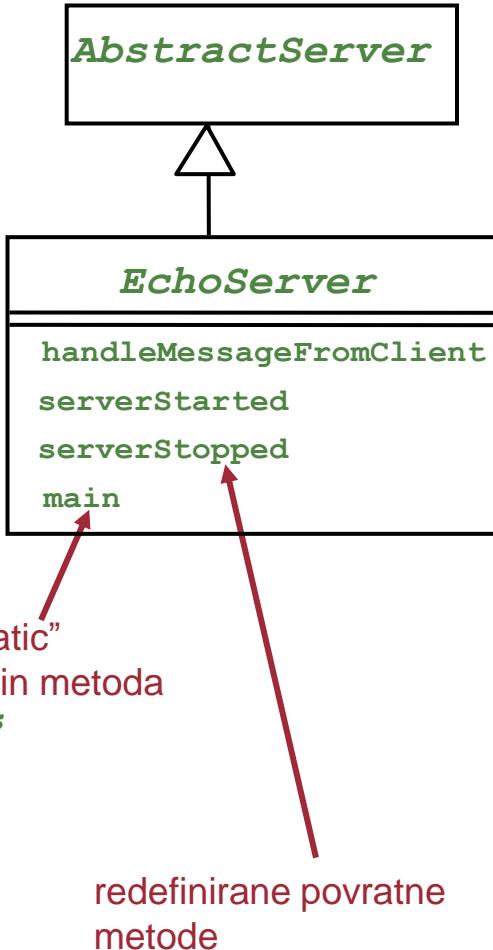


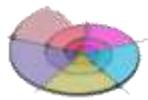


Chat poslužitelj



- `echoServer` je podrazred od `AbstractServer` razreda
 - `main` metoda kreira novu instancu i pokreće ju.
 - Instanca razreda `EchoServer` sluša spajanje klijenata (pozivom `listen` metode) i rukuje s vezama sve dok se poslužitelj ne zaustavi.
 - `EchoServer` nema korisničkog sučelja.
 - tri povratne metode samo ispisuju poruku korisniku.
`handleMessageFromClient`,
`serverStarted`
`serverStopped`
- metoda `handleMessageFromClient` poziva `sendToAllClients`
 - time se proslijeđuju poruke svim klijentima.
 - od `ConnectionToClient` metode postoji instance za spojene klijente
ne koriste se njihove metode `sendToClient()`, već metoda `sendToAllClients()`





Dio implementacije EchoServer



Redefinirana metoda `handleMessageFromClient`:

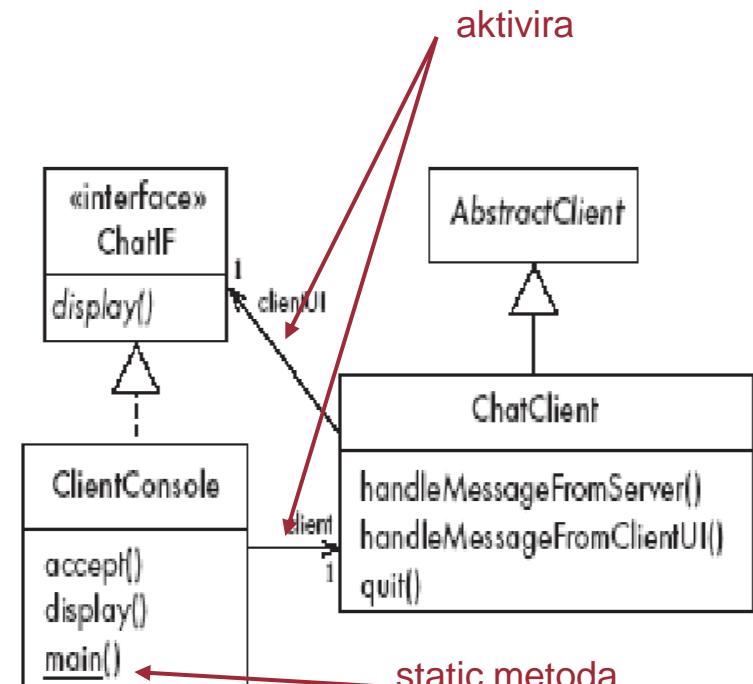
```
// stvaranje podrazreda
public class EchoServer extends AbstractServer {
final public static int DEF_PORT = 5555;
// konstruktor objekta s definiranim
portom slušanja
// na koji se spajaju klijenti
public EchoServer(int port) { super(port); }

// metoda za rukovanje porukom mora se
redefinirati
public void handleMessageFromClient
(
    Object msg, ConnectionToClient client)
{
    System.out.println(
        "Message received: "
        + msg + " from " + client);
    this.sendToAllClients(msg);
}
```

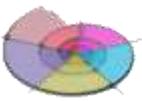
```
// početna metoda u poslužitelju
public static void main(String[]
args)
{
    port = DEF_PORT;
    // kreiranje objekta
    poslužitelja sv
    EchoServer sv = new
    EchoServer(port);
    // započinjanje slušanja
    sv.listen();
} // kraj main metode
} // kraj razreda EchoServer
```

Chat klijent

- *ChatClient* je podrazred od *AbstractClient*.
- *chatClient* redefinira metodu *handleMessageFromServer* koja samo inicira prikaz poruke korisniku.
- Druge dvije metode u *ChatClient* pozivane su od strane korisničkog sučelja.
- *ChatIF* *ChatIF* je sučelje odvojeno od funkcionskog dijela, tj. od *ChatClient*.
- Implementaciju sučelja (samo jedne metode *display*) obavlja razred *ClientConsole*.



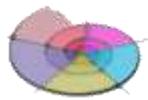
ClientConsole
implementira sučelje
ChatIF



Chat klijent



- Klijentski program započinje s radom *main* metodom u *ClientConsole*. Ona kreira instance dva razreda (*ChatClient* i *ClientConsole*) koje se izvode u dvije odvojene dretve :
 - *chatClient*
 - podrazred od *AbstractClient*
 - redefinira *handleMessageFromServer*
 - jer samo zove *display* metodu korisničkog sučelja
 - *clientConsole* (nije podrazred od *AbstractClient*)
 - korisničko sučelje kao razred je odvojeno je od funkcionalnog dijela klijenta. *ClientConsole* implementira ovo sučelje, tj. implementira *display* metodu koja prikazuje na konzoli
 - prihvata ulaz korisničkih podataka pozivom *accept* metode koja šalje sve ulazne podatke objektu razreda *ChatClient* pozovom metode *handleMessageFromClientUI*
 - metoda *handleMessageFromClientUI* zove metodu *sendToServer* (javna metoda u *AbstractClient*)



Dio implementacije *ChatClient*



`ChatIF clientUI; // clientUI je varijabla tipa sučelja ChatIF.`

Slanje poruke poslužitelju:

```
public void handleMessageFromClientUI(String message) {
    try {
        sendToServer(message); // prijenos do poslužitelja
    }
    catch (IOException e) // rukovanje iznimkom
    {
        clientUI.display (
            "Could not send message. " +
            "Terminating client.");
        quit(); } }
```

Primanje poruke od poslužitelja:

```
public void handleMessageFromServer(Object msg)
{
    clientUI.display(msg.toString());
}
```



Dio implementacije razreda *ClientConsole* (1)



```
public class ClientConsole implements ChatIF {  
    // odredi port, varijabla razreda  
    final public static int DEFAULT_PORT = 5555;  
    // varijabla instance, asocijacija s ChatClient  
    chatClient client;  
    // konstruktor implementira objekt razreda ChatClient  
    public ClientConsole(String host, int port) {  
        client = new ChatClient(host, port, this); }  
    // metoda za prihvati poruke od korisnika  
    public void accept() {  
        BufferedReader fromConsole = new BufferedReader(new  
            InutStreamReader(System.in));  
        String message;  
        while (true) {  
            message = fromConsole.readLine();  
            client.handleMessageFromClientUI(message); }  
    }  
}
```

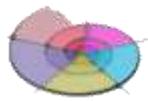


Dio implementacije razreda *ClientConsole* (2)



```
// metoda display() sučelja je implementirana ovdje
public void display(String message) {
    System.out.println("> " + message); }

// početna metoda razreda za stvaranje ovoga objekta
// klijentska strana započinje rad pozivom ove metode
public static void main(String[] args) {
    host = "local host";
    clientConsole chat =
        new ClientConsole(host, DEFAULT_PORT);
    // prihvati poruke od korisnika
    chat.accept();
}
} // kraj razreda ClientConsole
```

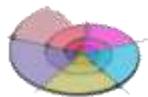


Chat klijent – sučelje ChatIF



```
// redefinira se u razredu koji to implementira  
// prikazuje poruku pozivom metode display
```

```
public interface ChatIF  
{  
    public abstract void display(String message);  
}
```



Osnovni kod *Chat* klijenta



```
public void handleMessageFromClientUI(String message)
{
    try
    {
        sendToServer(message);
    }
    catch (IOException e)
    {
        clientUI.display (
            "Could not send message. " +
            "Terminating client.");
        quit();
    }
}

public void handleMessageFromServer(Object msg)
{
    clientUI.display(msg.toString());
}
```



1. Podijeli pa vladaj:

- Podjelom sustava na klijenta i poslužitelja je uspješan način optimalne podjele.
- klijent i poslužitelj mogu se oblikovati odvojeno.

2. Povećaj koheziju:

- Poslužitelj osigurava kohezijski spojenu uslugu klijentima.

3. Smanji međuvisnost:

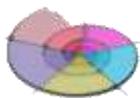
- Uobičajeno je da postoji samo jedan komunikacijski kanal preko kojega se prenose jednostavne poruke.

4. Povećaj apstrakciju:

- Odvojene raspodijeljene komponente su dobar način povećanja apstrakcije.

5. Povećaj uporabu postojećeg:

- često je moguće pronaći odgovarajući radni okvir temeljem kojega se oblikuje raspodijeljeni sustav. Međutim, klijent-poslužitelj arhitektura je često specifična s obzirom na primjenu.



6. Oblikuj za fleksibilnost:
 - Raspodijeljeni sustavi se često vrlo lako mogu rekonfigurirati dodavanjem novih poslužitelja ili klijenata.
7. Oblikuj za prenosivost:
 - Klijenti se mogu oblikovati za nove platforme bez promjene poslužiteljske strane.
8. Oblikuj za ispitivanje:
 - Klijenti i poslužitelji mogu se ispitivati neovisno.
9. Oblikuj konzervativno:
 - U kod koji rukuje porukama mogu se ugraditi stroge provjere (npr. rukovanje iznimkama).

Diskusija

-
-
-
-
-

Oblikovanje programske potpore

ak.god. 2014./2015.

Primjeri arhitekturnih obrazaca



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



Tema

- Podsjetnik
- Ponovna uporaba programskih komponenti
- Arhitekturni obrasci
 - raspodijeljeni sustavi
 - arhitektura klijent – poslužitelj
 - primjer objektnog radnog okvira klijent-poslužitelj
 - arhitektura zasnovana na događajima
 - obrazac model-pogled-nadglednik
 - posrednička arhitektura
 - uslužno usmjerena arhitektura – SOA
 - arhitektura sustava zasnovanih na komponentama
- Implementacija programskog proizvoda

Literatura

- Timothy C. Lethbridge, Robert Laganière: ***Object-Oriented Software Engineering: Practical Software Development using UML and Java***, McGraw Hill, 2001
 - <http://www.lloseng.com>
- Sommerville, I., ***Software engineering***, 8th ed, Addison Wesley, 2007.

Pripremio i prilagodio: Nikola Bogunović, Vlado Sruk

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

U pripremi materijala osim literature upotrijebljeni su i drugi izvori, te zahvaljujem autorima.

POSREDNIČKA ARHITEKTURA



Posrednička arhitektura



- Predstavlja proširenje raspodijeljene arhitekture klijent – poslužitelj.
- Proširenje se ogleda kroz:
 - slojevitu organizaciju klijenata i poslužitelja
 - Više razina (naslaga, slojeva) - engl. *tier*
 - uvođenje međusloja
 - Posrednici - engl. *middleware*
 - uvođenje zastupnika
 - engl. *broker*



- *engl. n-tier architecture*
- **Uvodjenje više razina**
 - klijenti i poslužitelji organiziraju se u **razine** (slojeve, naslage).
- Svaka razina pruža uslugu razini iznad.
- Svaka razina oslanja se na razinu ispod.
- Razina zatvara (skriva, enkapsulira) skup usluga i implementacijske detalje niže razine o kojoj ovisi.
- Često niža razina predstavlja “virtualni stroj” za razinu iznad.
- Višerazinska arhitektura **nije** u posebnom smislu **slojevita** (*engl. layered*).
 - **slojevita arhitektura** odnosi se na strukturu modula, dakle statički pogled,
 - **višerazinska arhitektura** (*engl. tiered*) odnosi na organizaciju u izvođenju (*engl. run-time*), dakle dinamički pogled.



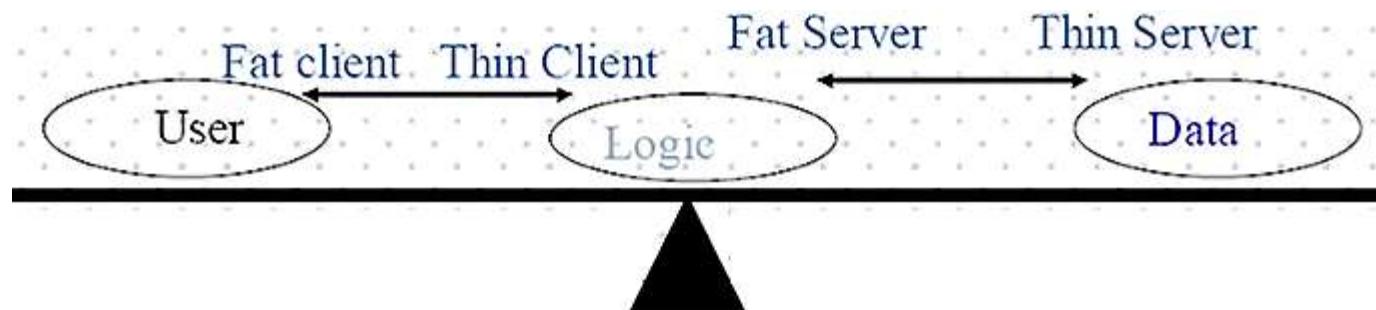
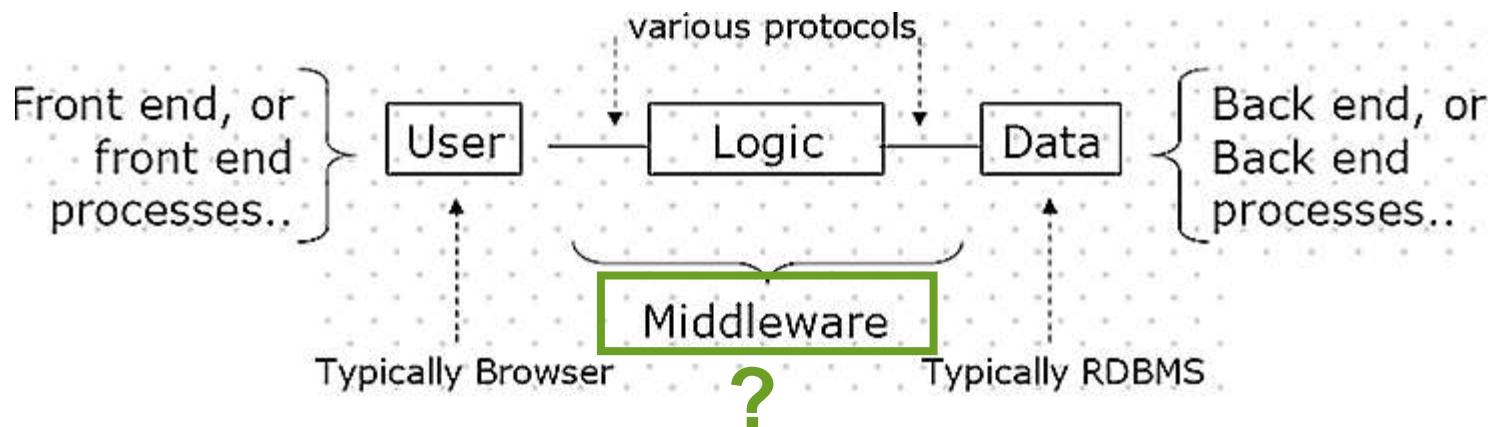
Trorazinska klijent-poslužitelj arhitektura



- engl. *Three-tier (layer) architecture*
 - Predstavlja vrstu arhitekture klijent-poslužitelj
 - Znatno bolje promovira skalabilnost i mogućnost jednostavnije modifikacije.
 - Nastoji otkloniti neke nedostatke klasične klijent-poslužitelj arhitekture, a posebice nastoji povećati performanse, raspoloživost i sigurnost.
 - Općenito trorazinska arhitektura sadrži
 - korisničku razinu
 - engl. *user; presentation; client tier*
 - logičku ili poslovnu razinu
 - engl. *business; logic; middle tier*
 - podatkovnu razinu
 - engl. *data tier*
-
- ```
graph TD; A[Korisnička razina] --> B[Logička razina]; B --> C[Razina podataka]
```

# Primjer

- Postoje mnoge varijacije trorazinske arhitekture, ovisno koliko se funkcionalnosti pridodaje svakoj razini.



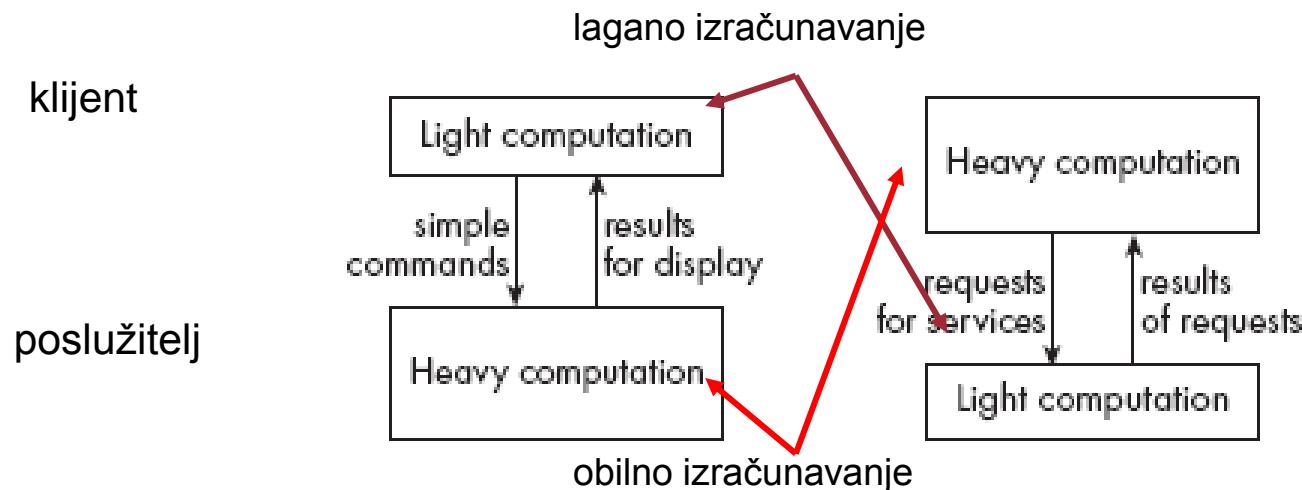
# Tanki i debeli klijent

## ■ Sustav tankog klijenta (engl. *Thin-client*)

- klijent je oblikovan da bude što je moguće manji i jednostavniji.
- većina posla obavlja se na poslužiteljskoj strani.
- oblikovnu strukturu klijenta i izvršni kod jednostavno se preuzima preko računalne mreže.

## ■ Sustav debelog klijenta (engl. *Fat-client*)

- što je moguće više posla delegira se klijentima.
- poslužitelj na taj način može rukovati s više klijenata.

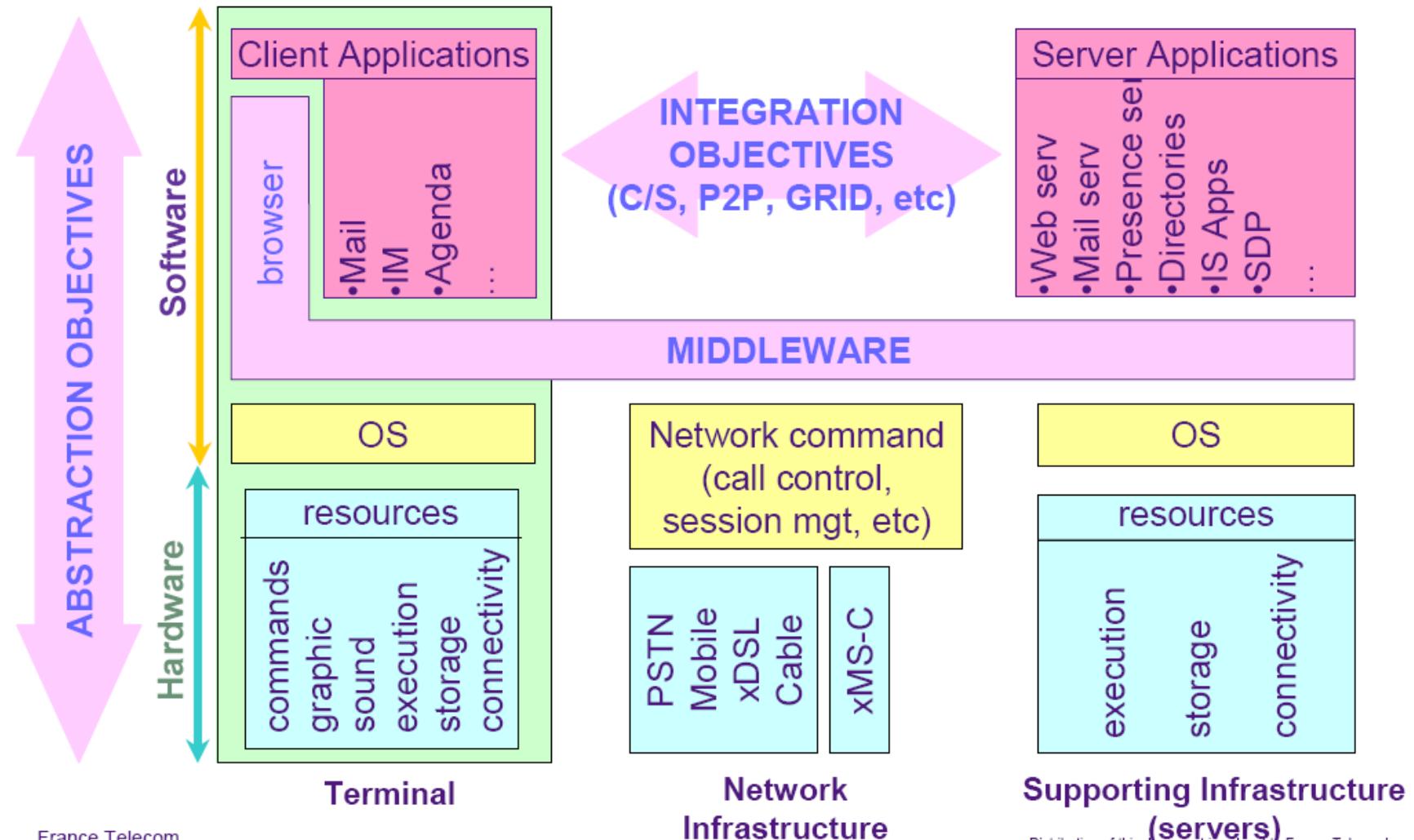


# Posrednička arhitektura

- *engl. Middleware*
- Uvođenje posredničke razine
  - nalazi se između klijenta i poslužitelja.
- Programska podrška koja omogućava uzajamno djelovanje aplikacija bez potrebe za poznavanjem i kodiranjem operacija nužnih za implementaciju usluge.
  - skriva osobama koje oblikuju raspodijeljeni sustav detalje operacijskog sustava i druge specifičnosti implementacije.
  - posrednička razina preuzima detalje komunikacijske mreže.
  - omogućuje osobama koje oblikuju raspodijeljeni sustav da se usredotoče na primjenski (aplikacijski dio).
- Olakšava oblikovanje i razvoj raspodijeljenih sustava

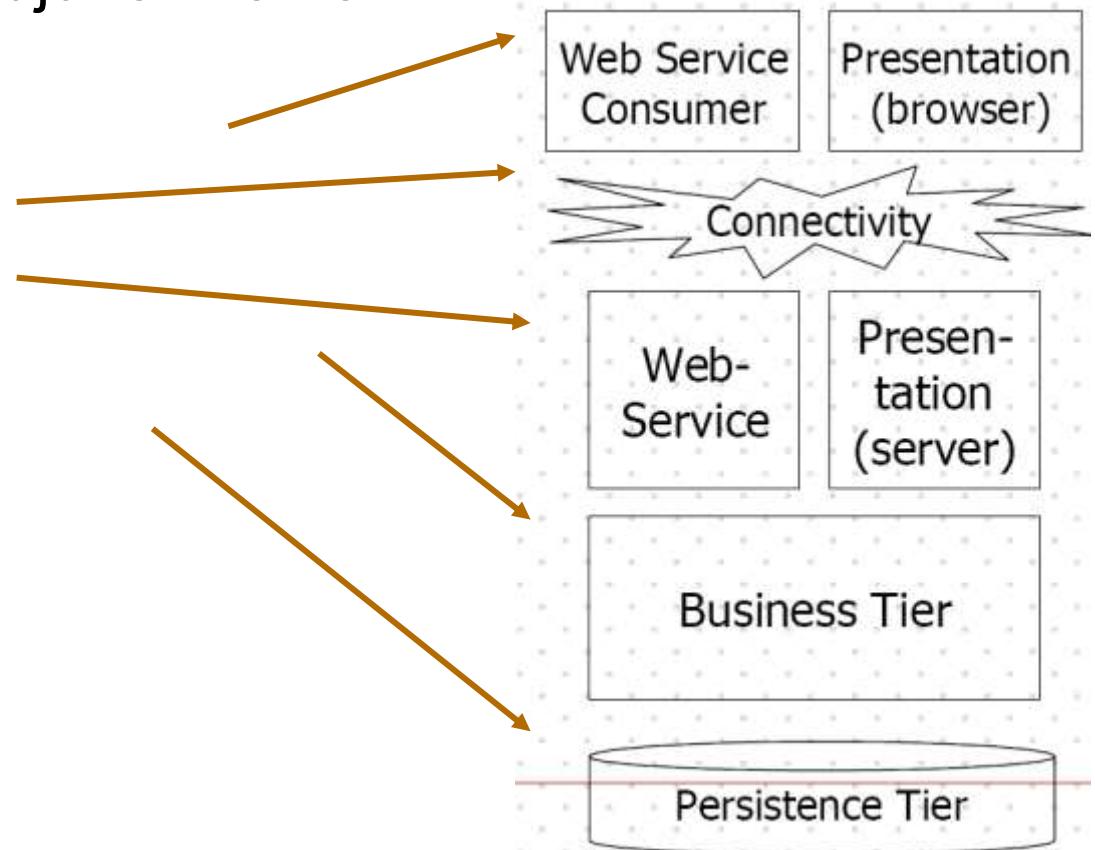


# Primjer posredničke razine



# Arhitektura s više razina

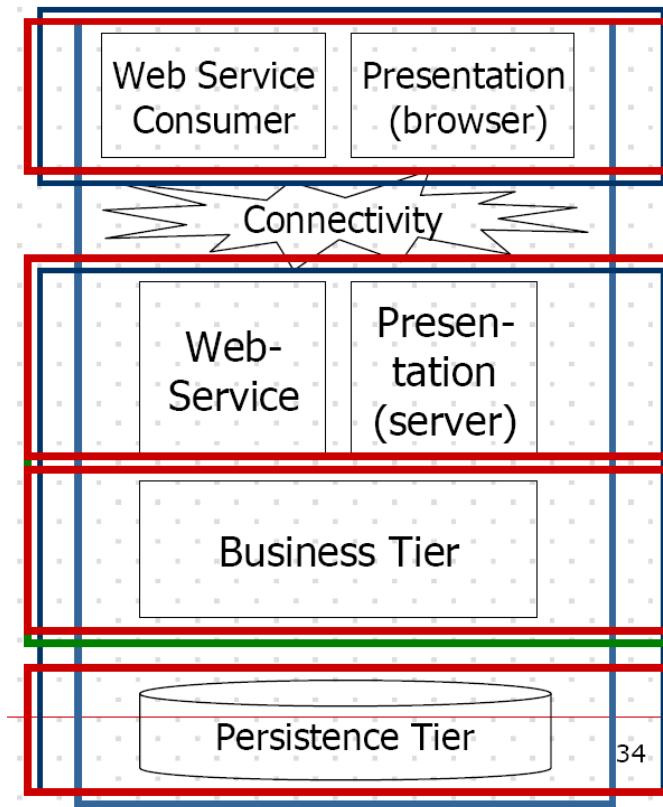
- engl. *N-tier*
- Funkcionalne odgovornosti (najviša razina apstrakcije funkcionalnih zahtjeva u objektno usmjerenoj paradigmi) se raščlanjuju i pridjeljuju razinama.
- Primjer web programa:
  - prezentacijska razina
  - povezivanje
  - web usluge
  - logička (poslovna) razina
  - podatkovna razina  
(trajno čuvanje)



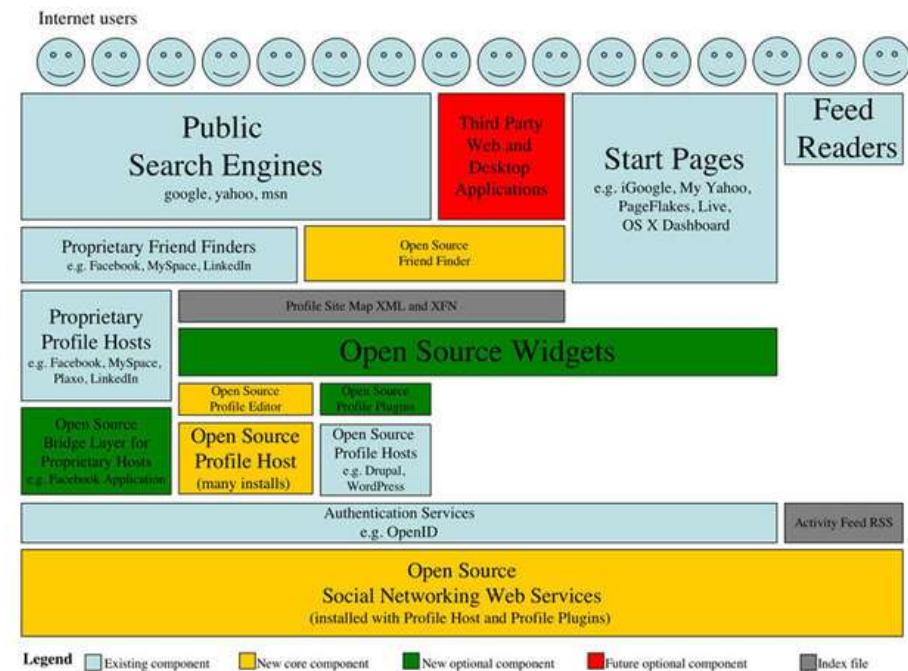
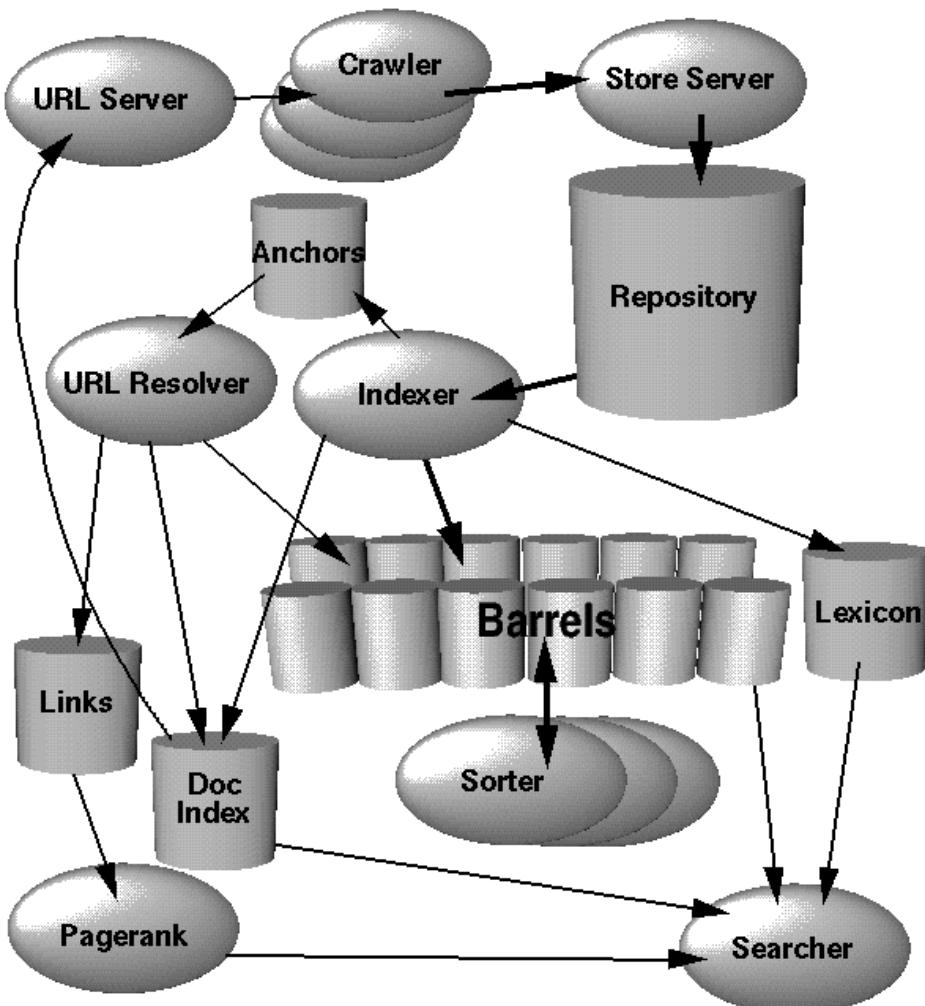


# Arhitektura s više razina

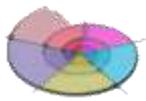
- Potrebno je razlikovati alociranje odgovornosti od skupa komponenata u izvođenju.
- Preslikavanje odgovornosti na komponente u izvođenju može se izvesti na više načina.



# Primjer:



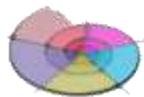
Izvor: Jeff Reifman: Open source social networking architecture



# Posrednička arhitektura

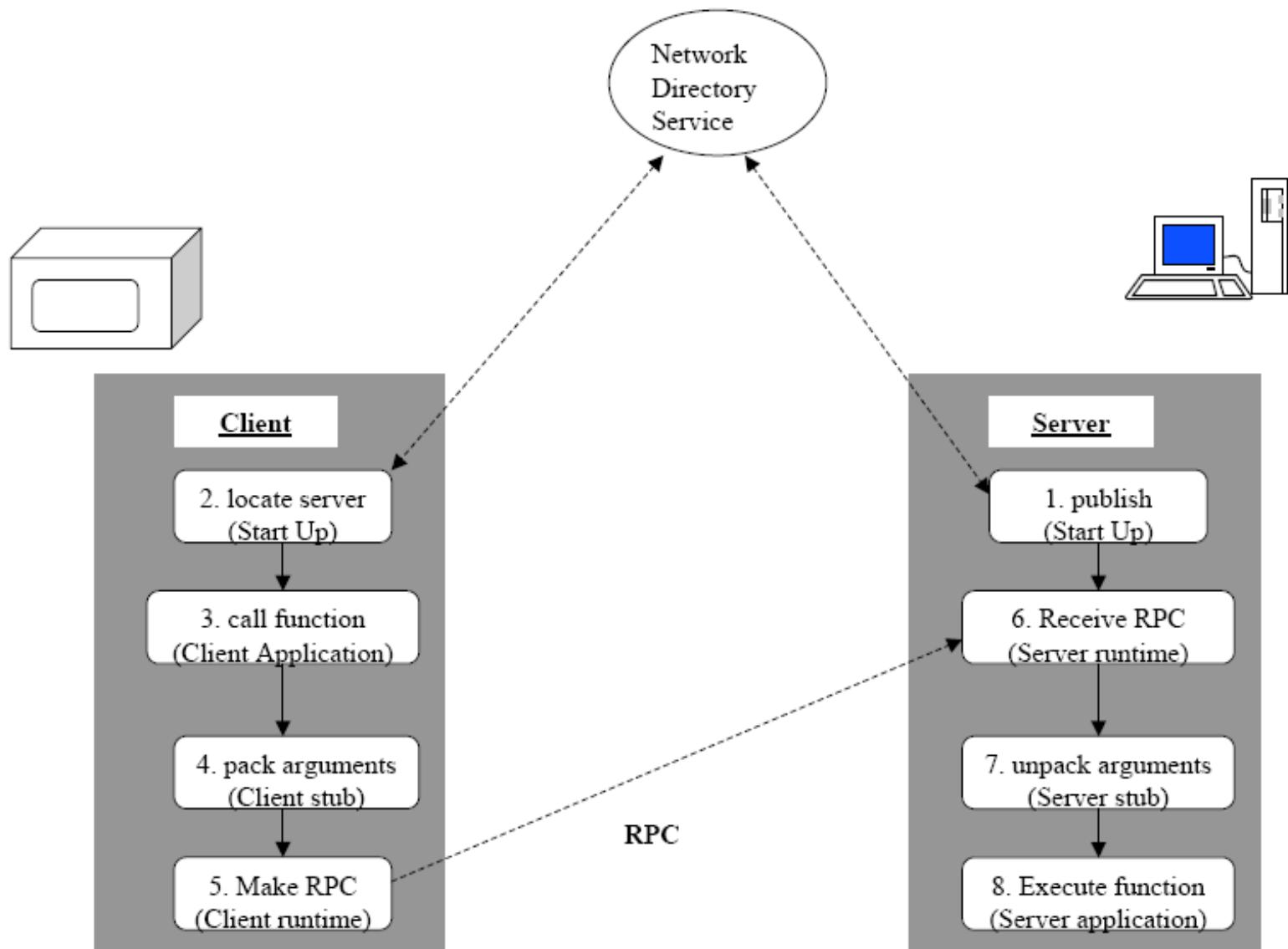


- Postoje tri vrste posredničkih arhitektura:
  - Transakcijski usmjerena (komunikacija s bazama podataka).
  - Zasnovana na porukama (pouzdana, asinkrona komunikacija).
  - Objektno usmjerena (sinkrona komunikacija između raspodijeljenih OBJEKATA).
- Najpopularnije objektno usmjerene arhitekture posredničke arhitekture obuhvaćaju:
  - **CORBA**
  - **DCOM, DotNET**
  - **EJB** (Enterprise JavaBeans, zasnovana na aktiviranju poruka, engl. *Remote Message Invocation - RMI*)
  - **MPI** (engl. *Message Passing Interface*)
- Svi ovi posrednički sustavi (referencirani kao Objektno usmjerena posrednička arhitektura) su zasnovani na udaljenom pozivu procedura (engl. *Remote Procedure Call - RPC*)
  - te arhitekture proširuju RPC uvođenjem mehanizama iz objektno usmjerene paradigmе.



# Udaljeni poziv procedura

## RPC Mechanism





## ■ Prednosti:

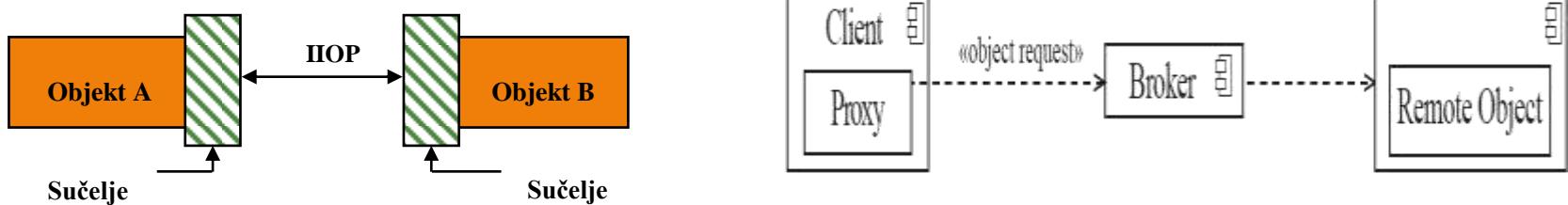
- oblikovanje temeljem više razine apstrakcije.
- podupire povećanje i poboljšanje sustava
  - promjena na jednoj razini utječe samo još na razinu ispod i iznad
- podupire ponovnu uporabu, prenosivost i sl.

## ■ Nedostaci:

- teško je odrediti optimalno preslikavanje odgovornosti na razine.
- ponekad se izračunavanje i funkcionalnosti sustava ne mogu razbiti na razine.
- ako se želi poboljšati performanse, mora se preskakati ili “tunelirati” kroz razinu.

# Arhitektura zastupnika

- posrednik, zastupnik *engl. broker*
- U objektno usmjerenom stilu arhitekture uvodi se specifična posrednička razina (*engl. specific middleware*) koja omogućuje interoperabilnost u *heterogenim* sustavima (različiti operacijski sustavi, različiti programski jezici) u *raspodijeljenom* okruženju.
- To je infrastruktura za ***raspodijeljene objekte***. Time se transparentno alociraju pojedini aspekti programske potpore na pojedine čvorove u mreži.



- Opća arhitektura posrednika zahtjeva za objektima - Common Object Request Broker Architecture - CORBA je jedna vrlo popularna i standardna implementacijska posrednička razina za oblikovanje raspodijeljenih objektno usmjerениh sustava.
  - **CORBA** standard razvija **OMG** (*Object Management Group*)
  - <http://www.omg.org/spec/index.htm>



# CORBA klijent-poslužitelj

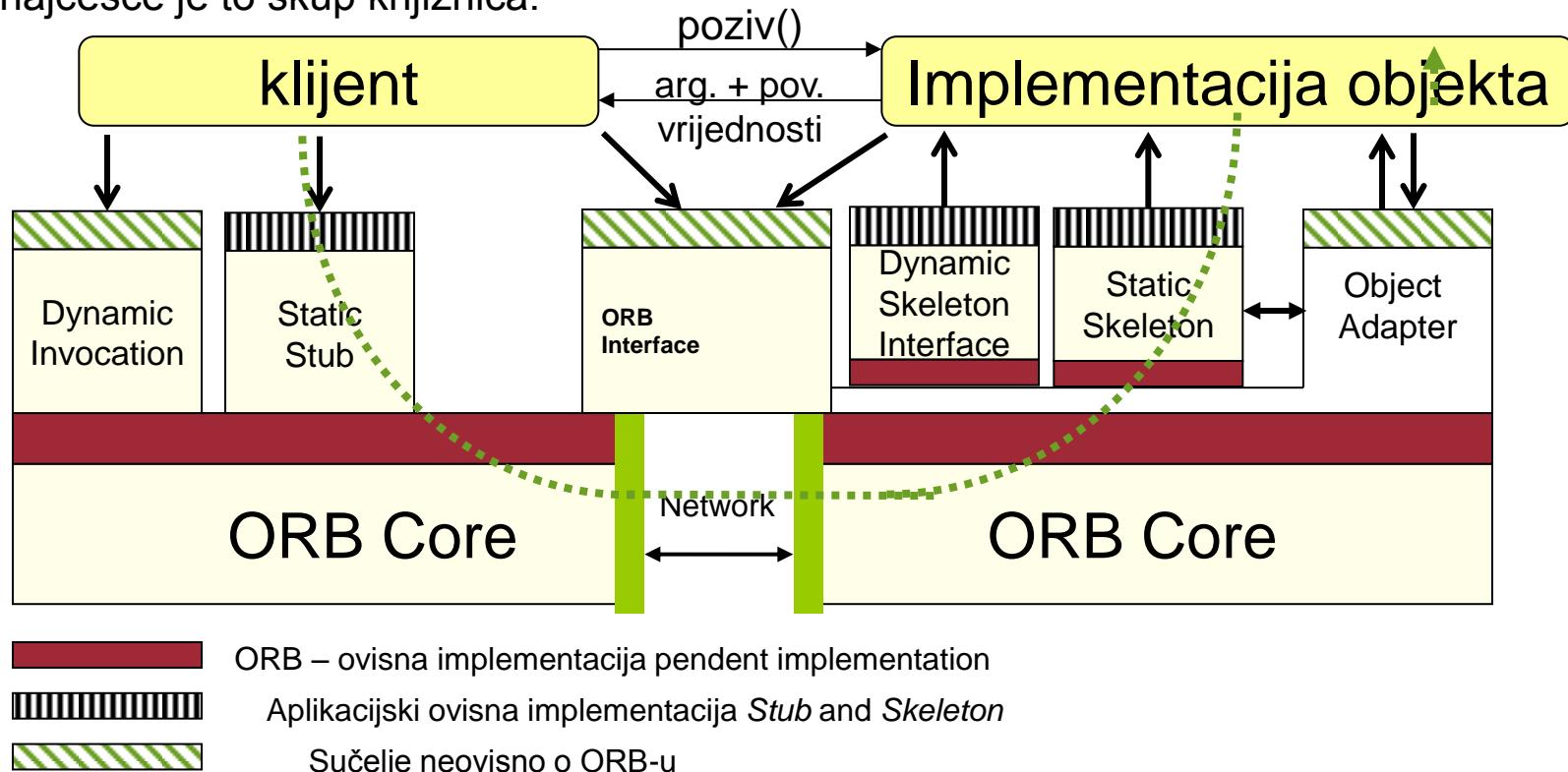


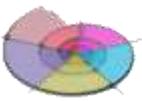
- Poslužitelji posjeduju sučelje koje je neovisno o bilo kojem programskom jeziku. Dostupno je preko jezika **IDL** koji se prevodi u standardne programske jezike (Java, C++, ...).
- Klijenti i poslužitelji mogu se programirati različitim programskim jezicima (ali svaki takav programski jezik mora imati kopče za CORBA-u).
  - **jezična transparentnost** na razini izvornog koda (ne na binarnoj/izvršnoj razini).
- **Posrednik ORB** (Object Request Broker), u obliku programske knjižnice na strani klijenta i poslužitelja prenosi prevedeni zahtjev klijenta do poslužiteljskog objekta.
- Klijenti i poslužitelji ne brinu se o lokaciji jednog ili drugog.
  - **lokacijska transparentnost** - Svi objekti su dostupni preko standardnog sučelja i reference na objekt (engl. *object reference*) ma gdje bili.
  - mogu biti implementirani na različitim sklopovskim platformama i operacijskim sustavima i još uvijek razgovarati međusobno standardnim protokolom.
- CORBA nema raspodijeljeno upravljanje memorijom (engl. *garbage collection*).



# Posrednička arhitektura: CORBA model

- ORB – Object Request Broker, je središnja povezujuća posrednička arhitektura za transparentno komuniciranje između objekata u **raspodijeljenim sustavima**.
  - oslanja se i instalira izravno na operacijski sustav i mrežne servise.
  - najčešće je to skup knjižnica.





# CORBA objekti



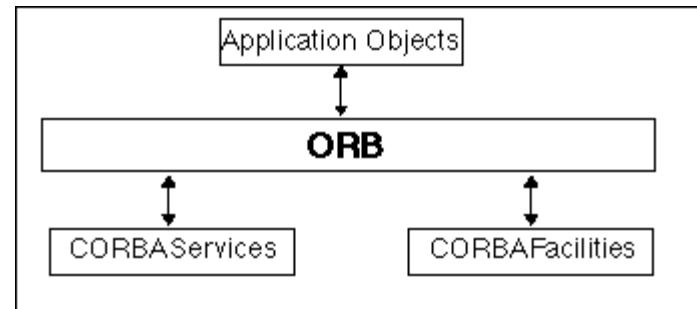
- Zatvoreni i pristupa im se samo preko dobro definiranog sučelja.
- Sučelje i implementacija su razdvojeni.
  - za jedno sučelje može postojati više implementacija, a jedna implementacija može podržavati više sučelja.
- Identitet objekta se proširuje kako bi sadržavao dodatnu informaciju kao što je lokacija objekta (npr. host i port broj poslužitelja).
  - sve to zajedno čini referencu na objekt (*engl. object reference*)
  - kako bi se locirao objekt za vrijeme izvođenja, klijent mora znati njegovu referencu.
  - referenca je tekstovni niz kodiran na specifičan način (klijent ga dekodira) i sadrži dovoljno informacija da se:
    - Uputi zahtjev ispravnom poslužitelju (host, port)
    - Locira ili kreira objekt (ime razreda, podaci/atributi)
- Referenca objekta se može dobiti na više načina u okviru standardnih ORB servisa.
  - Najčešće se čita datoteka u koju je poslužitelj upisao referenciju.



# Object Management Architecture (OMA)



- CORBA usluge
  - engl. *CORBAServices*
  - rukovanje asinkronim događajima, transakcije, paralelnost, imenovanje, odnosi, ...
- Objekti
  - engl. *Application Objects*
  - usluge aplikaciji ili skupu aplikacija
- CORBA procesi
  - CORBA Facilities
  - osiguravaju zajedničke horizontalne i vertikalne usluge
  - npr. *korisničko sučelje, upravljanje informacijama, sustavom, standarde domene...*



# Primjer IDL-a

- Primjer IDL specifikacije sučelja "Accounting" nekog objekta
- (IDL je jezik sličan pojednostavljenom C++):

```
module Money { // u jednom modulu može biti više sučelja

interface Accounting {
 float get_outstanding_balance();};};
```

- Uporaba sučelja u klijentu– pisano u Javi

```
import org.omg.CORBA.*; // uvezi sve potrebno

public class Client {

public static void main(String args[]) {

// inicijaliziraj ORB – Object Request Broker

ORB orb = ORB.init(args, null);

// poveži lokalnu varijablu "acc" s objektom "Account"

money.Accounting acc = Money.AccountingHelper.

 bind(orb, "Account");

// dohvati stanje računa UDALJENI POZIV!!

float balance = acc.get_outstanding_balance();

// ispiši stanje računa

System.out.println("The balance is $" + balance); } }
```

## Poslužitelj:

```
class AccountingImpl extends _AccountingImplBase
{
public float get_outstanding_balance()
{
 float bal = (float) ??????; // Implementacija stvarne funkcije db ...
 return bal;
}

public static void main(String[] args)
{
 try {
 // Initialize the ORB and BOA
 ORB orb = ORB.init(args, null);
 BOA boa = orb.BOA_init();

 // Instanciranje AccountingImpl
 AccountingImpl impl = new AccountingImpl("Account");
 boa.obj_is_ready(impl);

 // čekanje poziva
 boa.impl_is_ready();

 }
 catch(SystemException e) {

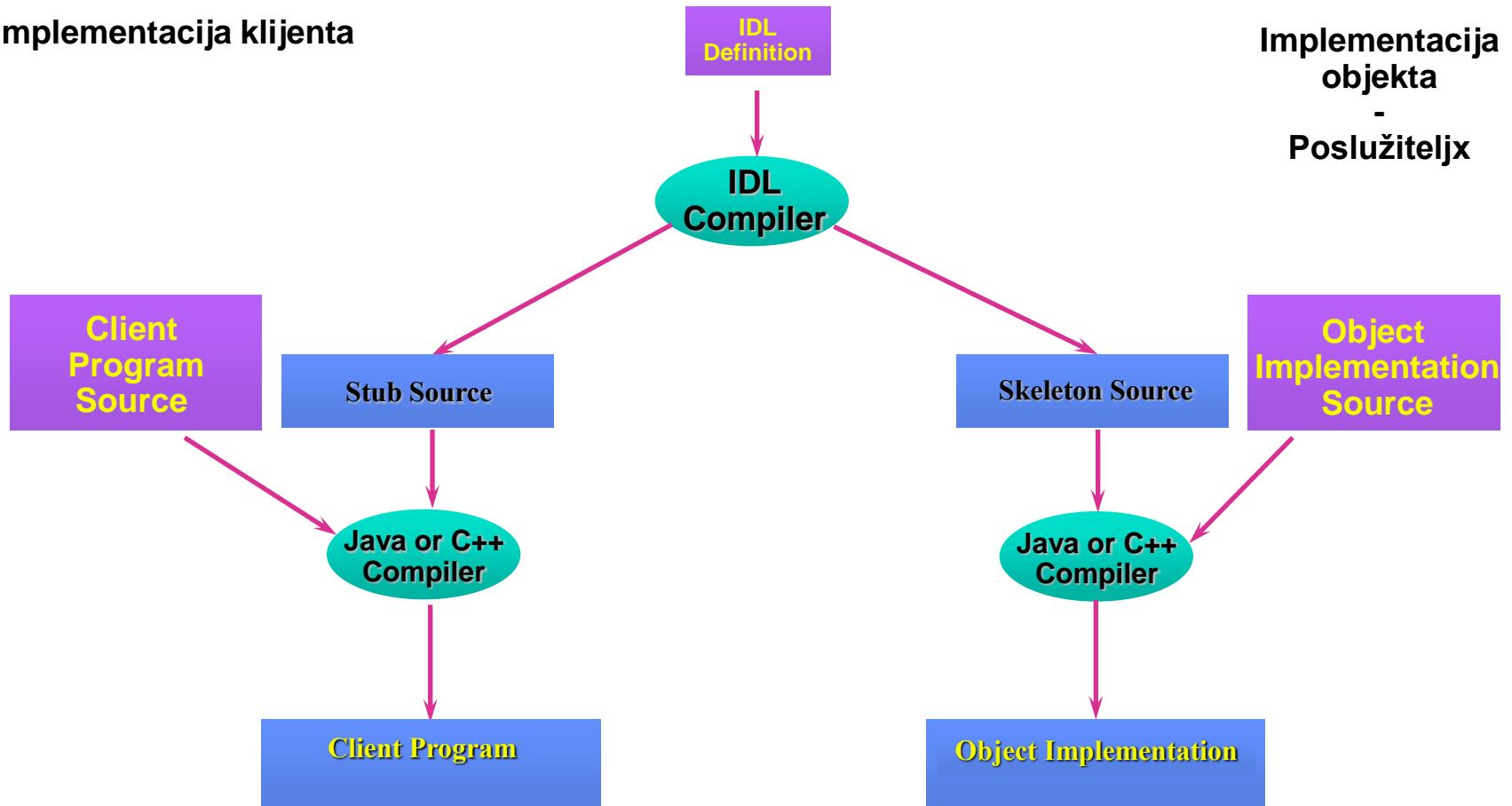
 }
}
}
```



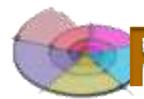
# Primjer: Uporaba IDL-a u razvojnom procesu



Implementacija klijenta



Implementacija objekta  
-  
Poslužiteljx



## 1. Podijeli pa vladaj:

- udaljeni objekti mogu biti neovisno oblikovani.

## 5. Povećaj ponovnu uporabu:

- često je moguće oblikovati udaljene objekte tako da ih i drugi sustavi mogu koristiti.

## 6. Povećaj uporabu postojećeg

- mogu se koristiti objekti koje su drugi oblikovali.

## 7. Oblikuj za fleksibilnost:

- posrednik (broker) se može ažurirati.
- objekti se mogu zamijeniti.

## 9. Oblikuj za prenosivost:

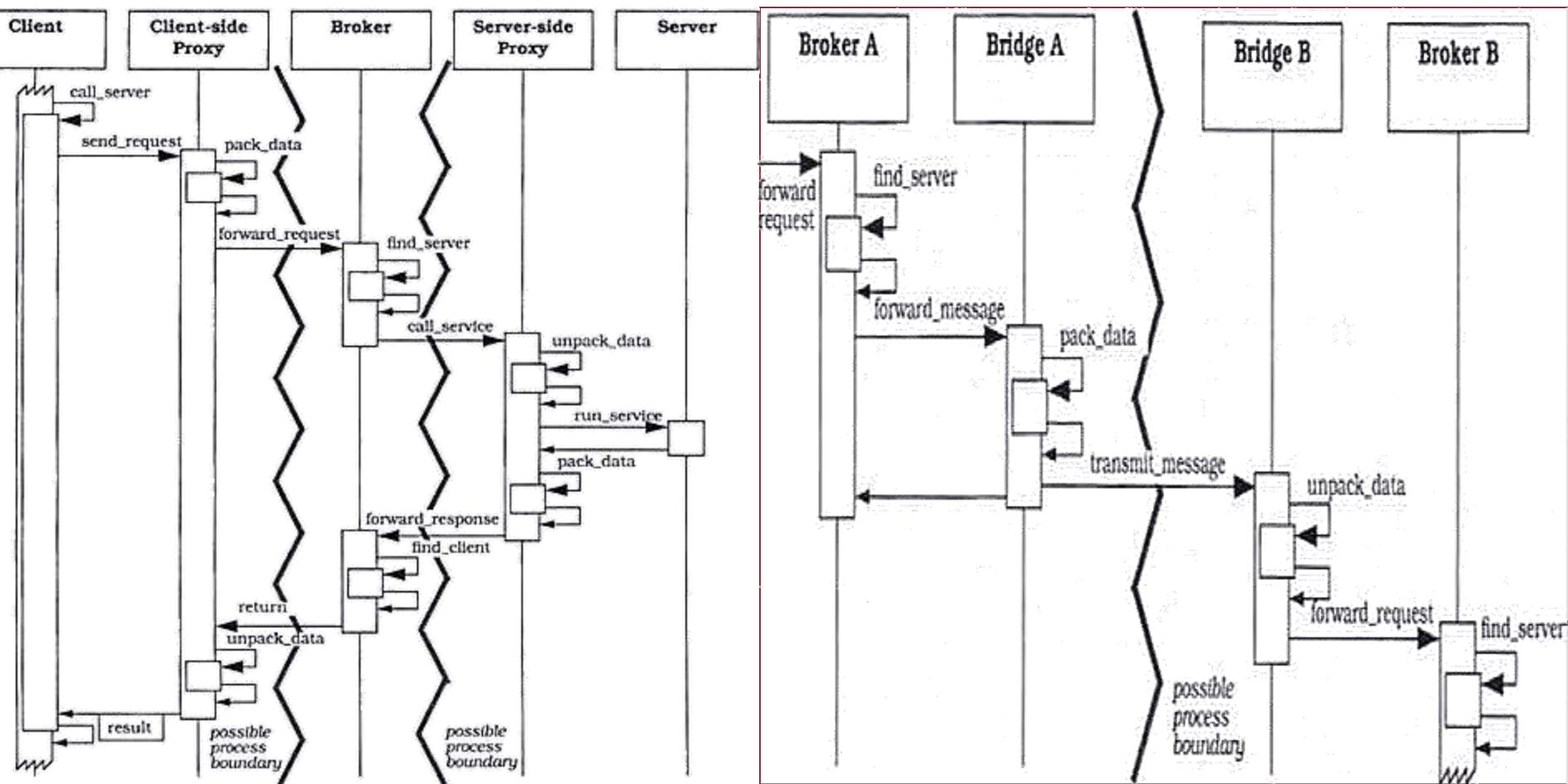
- može se oblikovati klijente za novu platformu i još uvijek pristupati postojećim brokerima i udaljenim klijentima na drugim platformama.

## 11. Oblikuj konzervativno:

- možeš se rigorozno provjeriti nedvosmisленo ponašanje udaljenih objekata.
- provjera istinitosti obilježja udaljenih objekata



# Primjeri



# Prednosti i nedostaci

## ■ Prednosti

- transparentnost lokacija;
- izmjenjivost i proširivost komponenti;
- prenosivost;
- interoperabilnost različitih sustava Brokera;
- ponovna uporaba.

## ■ Nedostaci

- smanjena efikasnost
- veća osjetljivost na pogreške
  - engl. *fault-tolerance*

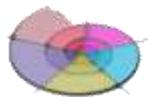
# ARHITEKTURA ZASNOVANA NA DOGAĐAJIMA



# Arhitektura zasnovana na događajima



- engl. *event based architecture, implicit invocation*
- Temeljne značajke:
  - komponente se međusobno ne pozivaju eksplisitno.
  - neke komponente generiraju signale = događaje.
  - neke komponente su zainteresirane za pojedine događaje, te se prijavljuju na strukturu za povezivanje komponenata.
  - komponente koje objavljaju događaj nemaju informaciju koje će sve komponente reagirati i kako na događaj.
  - asinkrono rukovanje događajima.
  - nedeterministički odziv na događaj.
  - model izvođenja: Događaj se javno objavljuje te se pozivaju registrirane procedure.



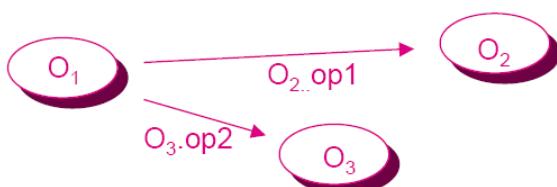
# Arhitektura zasnovana na događajima



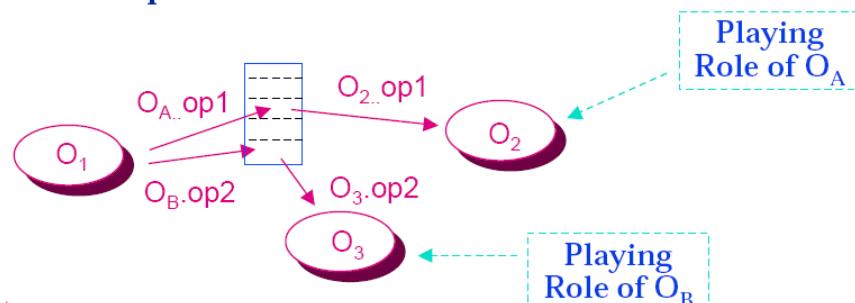
## ■ Tipovi povezivanja:

- izričito/eksplicitno (engl. *explicit*)
- implicitno (engl. *implicit*)

### Explicit Direct Invocation



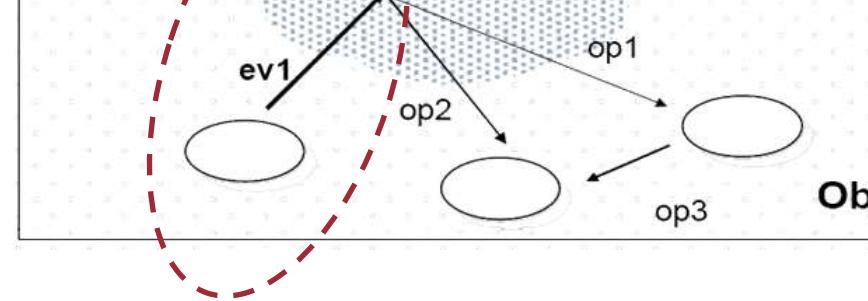
### Explicit Indirect Invocation



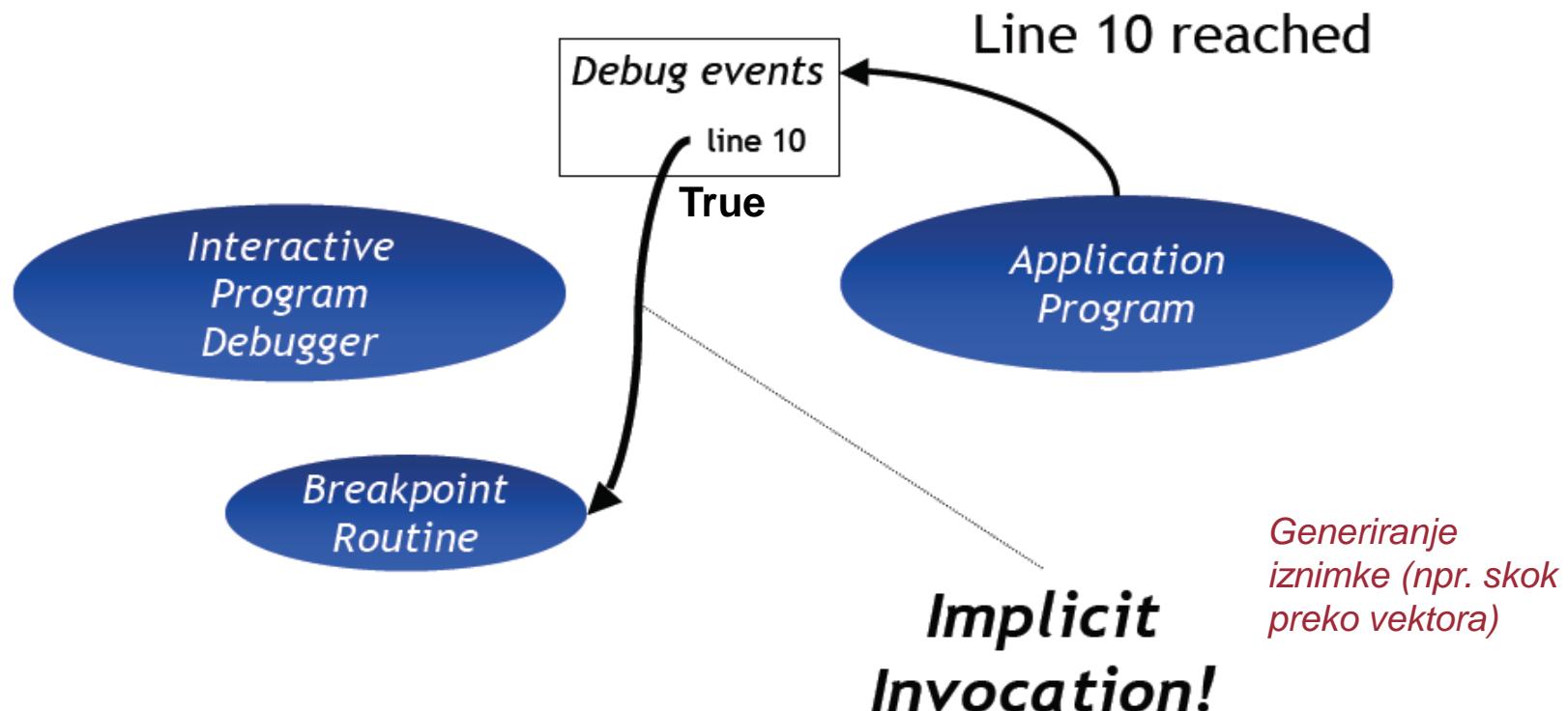
lectures

### Implicit Invocation

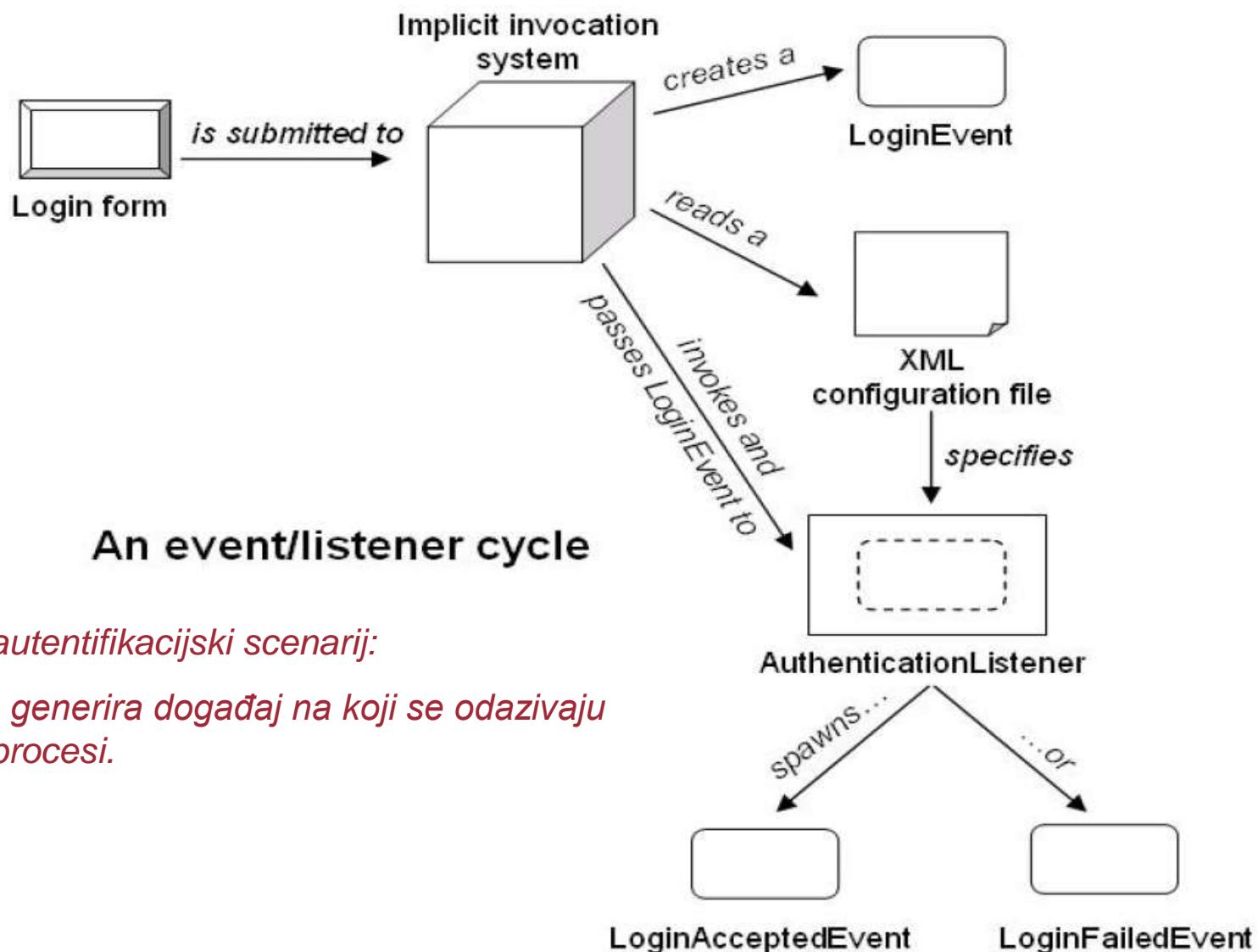
*Struktura za povezivanje*

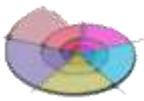


- Primjenski program ne poziva rutinu za obradu događaja izravno niti neizravno.
  - Prekidna rutina (odziv na iznimku) se aktivira preko prekidnog vektora.



# Primjer 2:

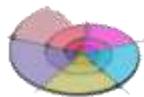




# Prednosti i nedostaci



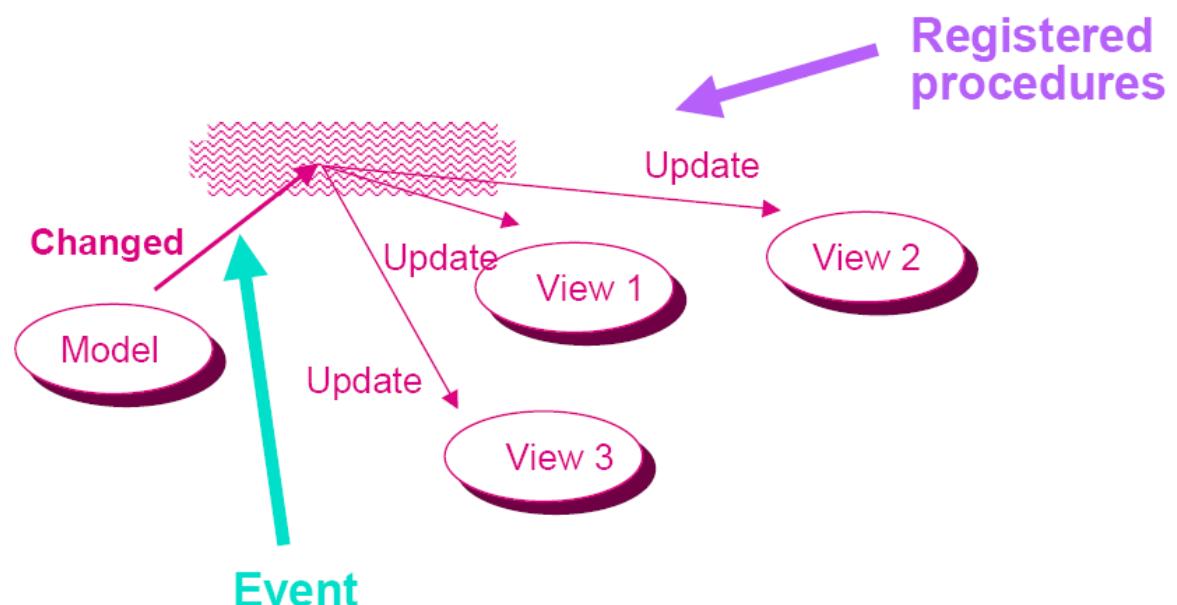
- Prednosti:
  - omogućuje razdvajanje i autonomiju komponenata.
  - snažno podupire evoluciju i ponovno korištenje.
  - jednostavno se uključuju nove komponente bez utjecaja na postojeće.
- Nedostaci:
  - komponente koje objavljuju događaje nemaju garancije da će dobiti odziv.
  - komponente koje objavljuju događaje nemaju utjecaja na redoslijed odziva.
  - apstrakcija događaja ne vodi prirodno na postupak razmjene podataka (možda je potrebno uvođenje globalnih varijabli).
  - teško rasuđivanje o ponašanju komponenata koje objavljuju događaje i pridruženim komponentama koje su registrirane uz te događaje (nedeterministički odziv)

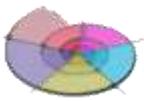


# Obrazac model-pogled-nadglednik

- MVC obrazac, engl. *Model View Controller*
- Stilistička varijacija arhitekture zasnovane na događajima

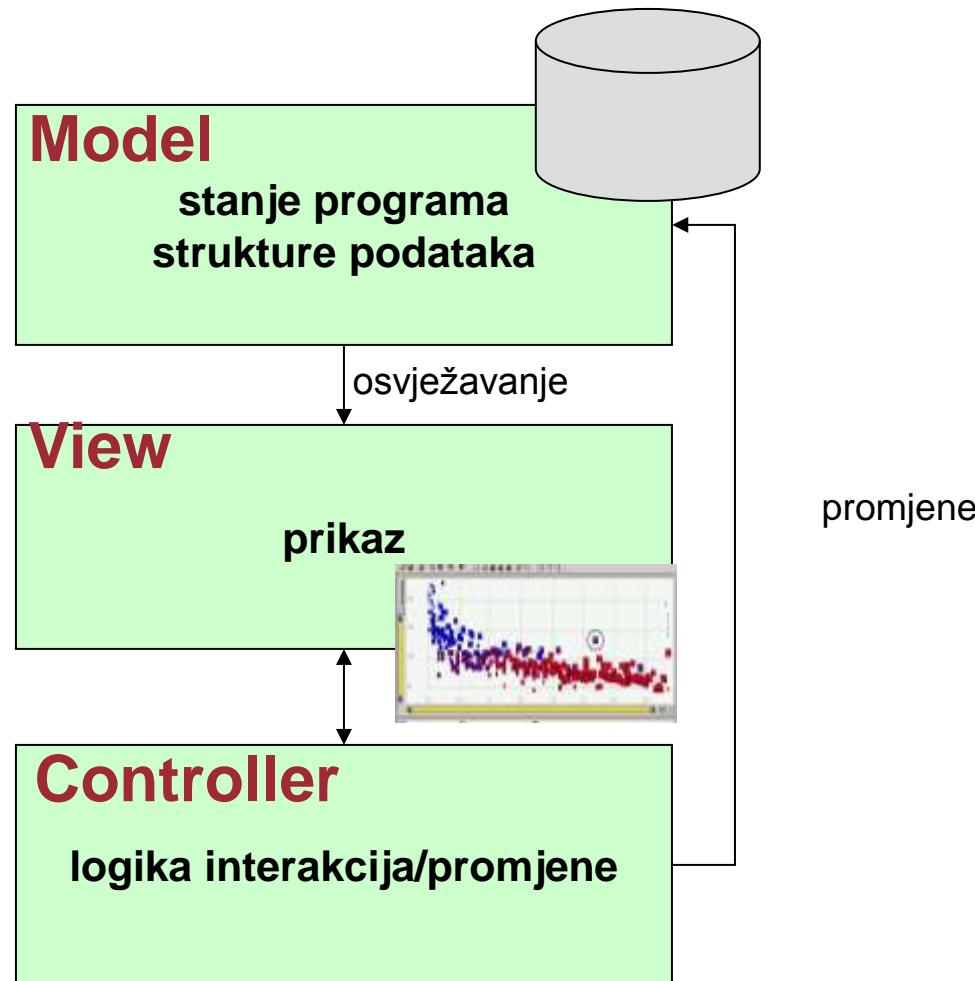
## Smalltalk-80 Model-View-Controller (MVC)





# Model-pogled-nadglednik

- Model – modelira problem neovisno od pogleda i nadglednika
  - ali osigurava usluge za njihov rad



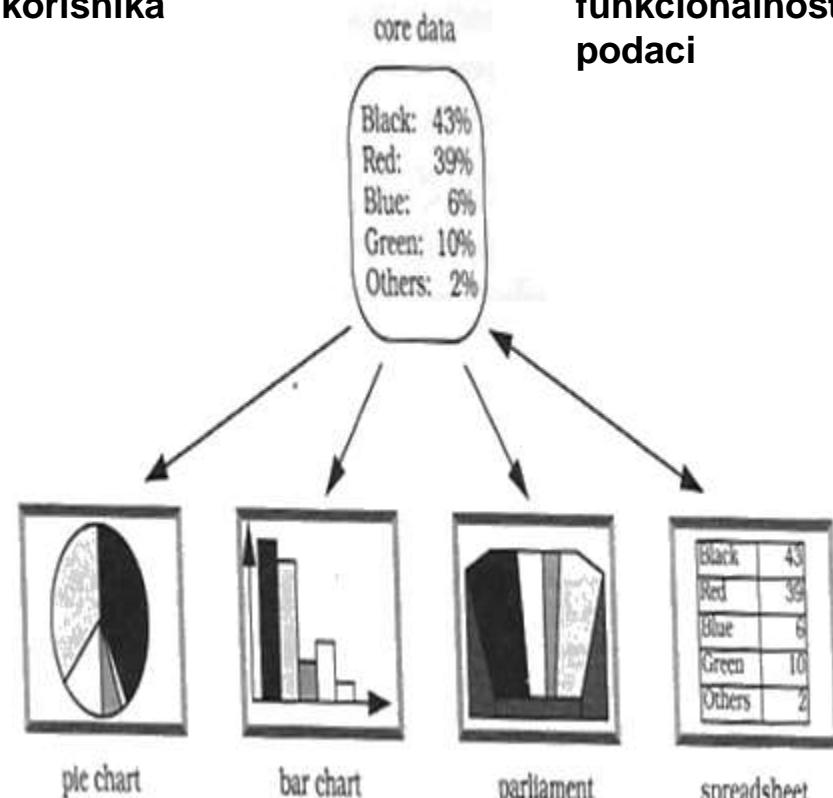


# Primjer: Proračunske tablice

- Sustav proračunskih tablica za upis i pregled podataka, npr. *Excel*.
- Bez značajnih promjena cijelokupnog sustava postoji potreba za:
  - dodavanjem nove neinteraktivne vizualizacije podataka.
  - dodavanjem nove interaktivne vizualizacije podataka

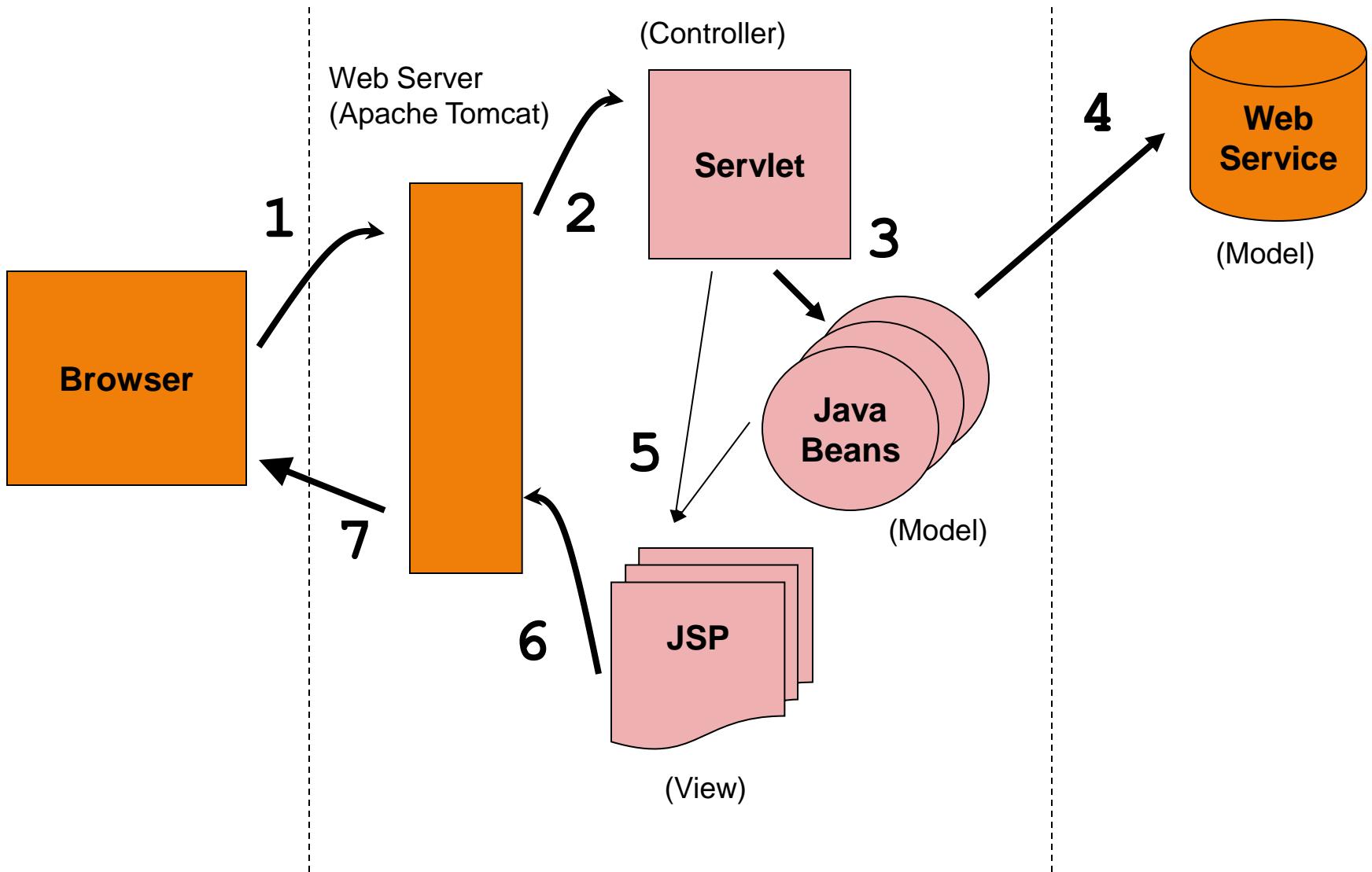
Controller: Rukovanje ulaznim podacima korisnika

Model: Jezgrene funkcionalnosti i podaci



View. *Prikaz informacija*

# Primjer Web aplikacija



# MVC obrazac

1. Akcija klijenta generira zahtjev za određenom URL stranicom.
2. Apache Tomcat otprema Java servlet koji rukuje s URL-om.
3. Servlet kao “Controller” interpretira akciju klijenta i šalje upit ili traži promjenu u Java Beans (“Model”).
4. Java Beans koristi vanjske resurse kao što je npr. Web usluga.
5. Ovisno o stanju modela servlet odabire prikladnu vizualizaciju i prosljeđuje Java Beans do Java Servlet Pages (JSP), tj. do “View” komponente.
6. Apache Tomcat konvertita JSP u HTML.
7. HTML se prosljeđuje klijentu.

# Zaključak

- Vrlo značajna arhitektura programske potpore.
- Intenzivna primjena u procesorskim sustavima s više jezgara (*engl. multicore*).
- Uključena u najnoviju specifikaciju UML-a (UML 2.0).

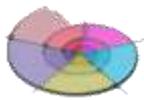
# USLUŽNO USMJERENA ARHITEKTURA



# Uslužno usmjerena arhitektura

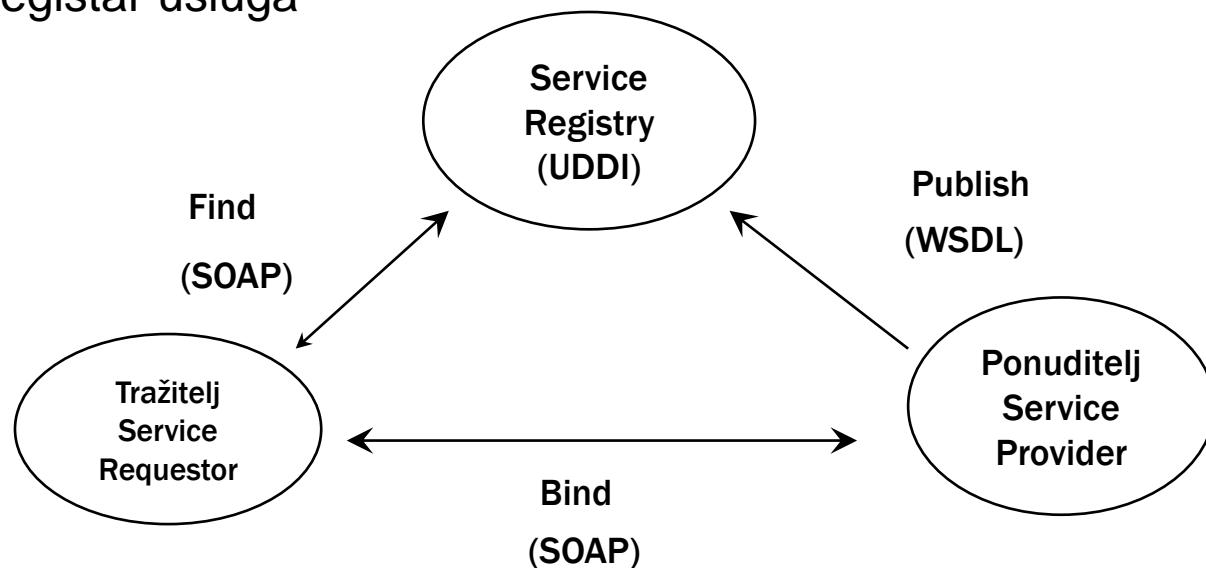


- engl. *Service oriented architecture (SOA), Service oriented architectural pattern (SOAP)*
- **Uslužno usmjerena arhitektura** organizira primjenski program (cjelovitu aplikaciju) kao kolekciju usluga koje međusobno komuniciraju uporabom dobro definiranih **sučelja**.
- Npr. Internet okruženje – Web usluge (engl. *Web services*)
  - web usluga = aplikacija dostupna putem Interneta, u svrhu izgradnje složenog sustava omogućava integraciju s ostalim Web uslugama
    - Dobro definirana funkcija
    - Samodostatna
    - Ne ovisi o drugim uslugama (okolina ili stanje)
  - komunikacija – standardnim protokolima razmjene podataka npr. XML



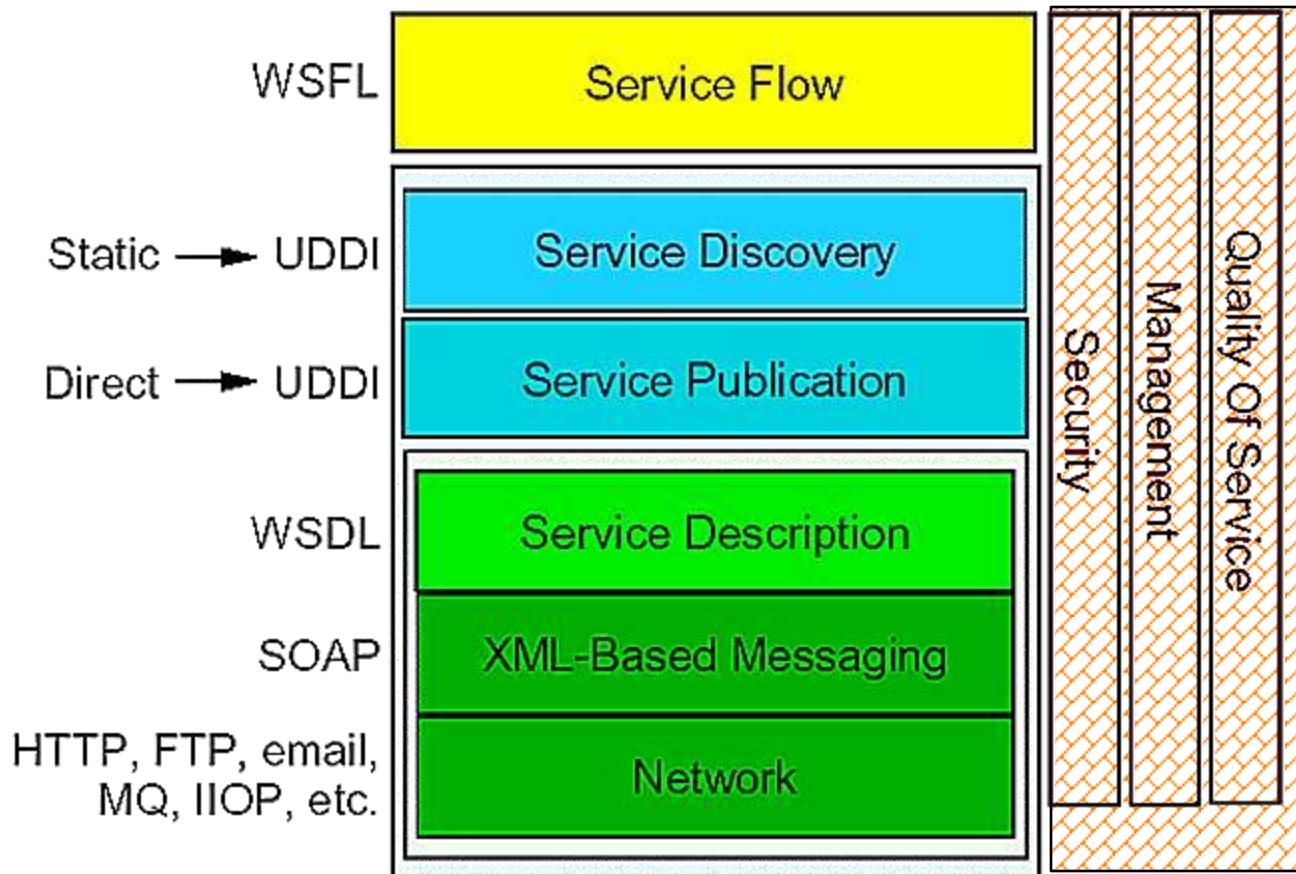
# Model web uslužne arhitekture

- Oglašavanje, otkrivanje selekcija i uporaba web usluge
  - tehnologije zasnovane na XML-u
- Simple Object Access Protocol
  - razmjena poruka Web usluga
- Web Service Definition Language
  - definira web uslugu
- Universal Description, Discovery, and Integration
  - API za registar usluga



# Web usluga

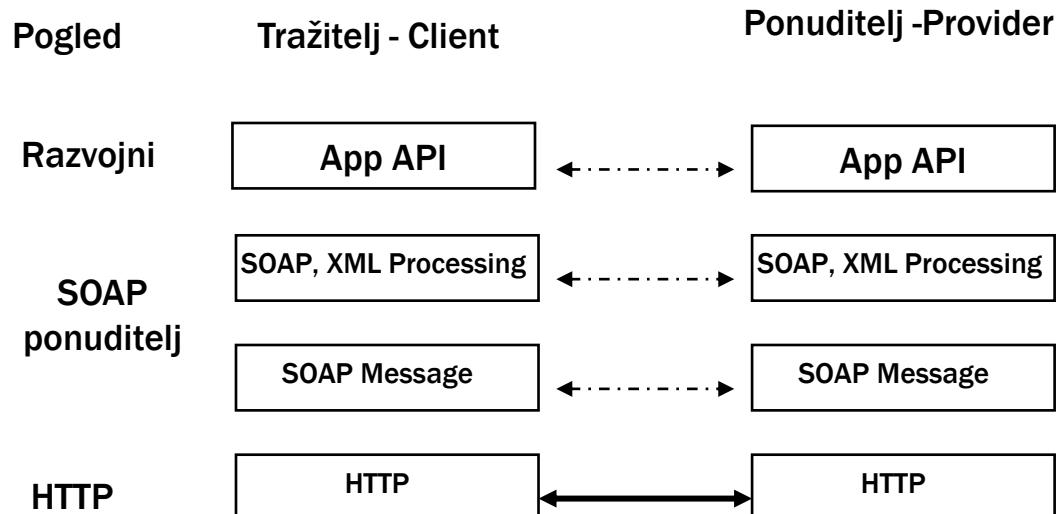
## ■ Struktura Web usluge



# SOAP poruka

## ■ SOAP je komunikacijski protokol

- neovisan o platformi
- zasnovan na XML-u
- namijenjen komunikaciji aplikacija
- zamjenjuje ranije razvijene protokole u Internet okruženju



```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Envelope>
```

```
<Header>
```

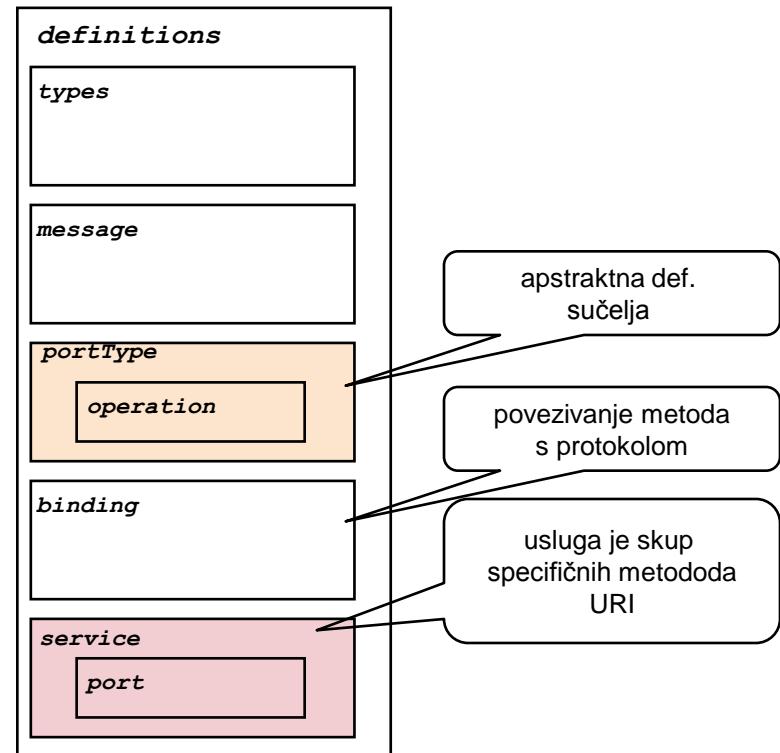
```
</Header>
```

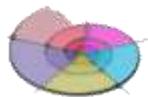
```
<Body>
```

```
</Body>
```

# WSDL

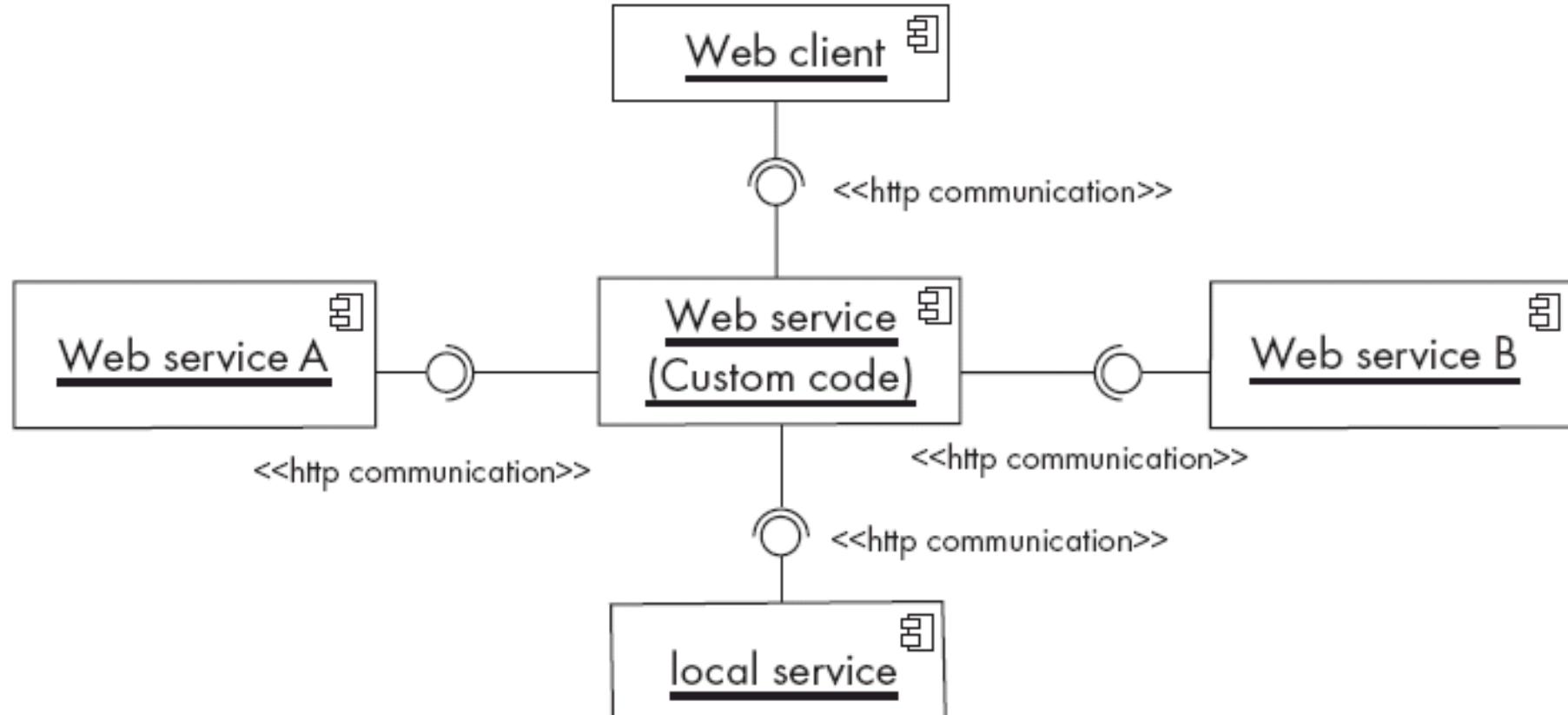
- XML dokument prema WSDL specifikaciji
- Opisuje
  - što radi servis
    - Operacije (metode) koje može pružiti
  - način pristupa
    - Format podataka i opis protokola
  - smještaj/lokaciju usluge
    - URL adresa
- Alati za automatsko generiranje koda iz WSDL-a





# Primjer: Opis Web usluge

- UML dijagram komponenata



# Utjecaj Web usluga

- Pojednostavljaju integraciju
  - unutar i između organizacija
- Faze uvođenja Web usluga i uslužno usmjerene arhitekture
  1. eksperimentiranje
    - Upoznavanje, vanjske web usluge, razmjena podataka, školovanje
  2. prilagodba postojećih sustava
  3. uklanjanje međuovisnosti
  4. uspostava interne arhitekture zasnovane na principima uslužno usmjerene arhitekture
  5. uključivanje vanjskih Web usluga
  6. pružanje usluga drugima
- Proces razvoja i normiranje uklanja nepovjerenje
- Podizanje razine kvalitete
- Iskorištavanje novih mogućnosti



1. Podijeli pa vladaj: Cijeli primjenski program sastoje se iz neovisno oblikovanih komponenata - usluga.
2. Povećaj koheziju: Web usluge su strukturirane kao slojevi i imaju dobru funkcionalnu koheziju.
3. Smanji međuovisnost; Web zasnovani primjenski programi su slabo vezani i oblikovani su spajanjem raspodijeljenih komponenata.
5. Povećaj ponovnu uporabu: Web usluga je posebice značajno ponovno uporabiva komponenta.
6. Povećaj uporabivost postojećeg: Web zasnovane usluge su oblikovane ponovnom uporabom postojećih web usluga.
8. Planiraj zastaru: Zastarjele usluge mogu se zamijeniti novim implementacijama bez utjecaja na primjenske programe koje ih koriste.

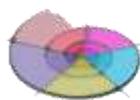
# ARHITEKTURA PROGRAMSKE POTPORE ZASNOVANA NA KOMPONENTAMA



- engl. *Component Based Design - CBD*
- Pouke procesa oblikovanja programske potpore s fokusom na princip ponovne uporabe dijelova.

	Objektno usmjeren pristup	Oblikovanje zasnovani na komponentama
Sastavi komponente u jedinstveni produkt	<u>Teško</u> , traži se vještina objektnog programiranja	<i>Može izvesti vješt korisnik</i>
Oblikuj komponente	<u>Teško</u> , traži se vještina objektnog programiranja	<u>Jako teško</u> , mora se voditi računa o mnogo korisnika

Izvor: A. *R. Newton, UC Berkeley*



- U fokusu arhitekture zasnovane na komponentama je njihova ponovna i višestruka uporaba.
- Princip višestruke uporabe u oblikovanju programske potpore manifestirao se tijekom vremena kroz:
  - ponovna uporaba *konzistencije* (programske jezice).
  - ponovna uporaba *fragmenata rješenja* (knjižnice).
  - ponovna uporaba *dijelova arhitekture* (arhitekturni obrasci i oblikovni obrasci – engl. *architectural patterns, design patterns*).
  - ponovna uporaba radnih okvira – engl. *frameworks*.
  - ponovna uporaba *cjelokupne arhitekture sustava*.



## ■ **Programski jezik:**

- uvodi kulturu kako se oblikuje program.
- često uvodi dogmu kako neke stvari treba napraviti.
- programski jezik ne osigurava ispravno oblikovanje ali isključuje mnoge stvari koje unose pogreške.

## ■ **Knjižnice:**

- prvi programski jezici imali su uključene sve funkcionalnosti.
- jezik C jezik (1978) izvlači specifične rutine u knjižnice (npr. U/I, matematičke,..).
  - od tada postoji tendencija odvajanja posebnih funkcionalnosti.

## ■ **Arhitekturni oblikovni obrasci:**

- nastoji se izgraditi katalog najmanjih ponavljačih dijelova arhitekture (obrascima) u objektno usmjerenom programiranju.
- obrazac je definiran imenom, opisom problema (s uvjetima) koji rješava, elementima (razredi, odnosu između razreda, odgovornosti, kolaboracije), te navođenjem dobrih strana uporabe obrasca.



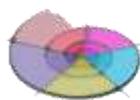
## ■ *Radni okviri:*

- skup razreda i interakcija.
- radni okviri traže oblikovanje podrazreda i implementaciju nekih operacija.
- radni okviri najčešće nisu posebno prilagođeni pojedinim primjenama već oslikavaju određene koncepte.

## ■ Cjelokupna arhitektura:

## ■ Uvodi zasebne stilove, npr.:

- cjevovodi i filtri
- događajno usmjerena
- repozitorij podataka
- virtualni strojevi
- ...



## ■ ***Definicija programske komponente***

- programska komponenta je jedinica kompozicije s ugovorno specificiranim sučeljem i kontekstnom ovisnošću.
- programska komponenta može se nezavisno razmjestiti u sustavu kojega oblikuju drugi dionici.
- programska komponenta je binarna jedinica kompozicije nezavisno proizvedena.
- programska komponenta nastoji se potvrditi na tržištu komponenata.

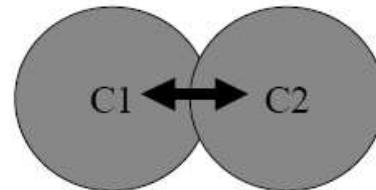
## ■ ***Razlika između objekta i komponente***

- definicija objekta je tehnička; ne uključuje pojam nezavisnosti.
- kompozicija objekata nije namijenjena širokom krugu korisnika.
- ne postoji niti će postojati tržište objekata.
- objekti ne podržavaju paradigmu priključi-i-radi (*engl. plug-and-play*)

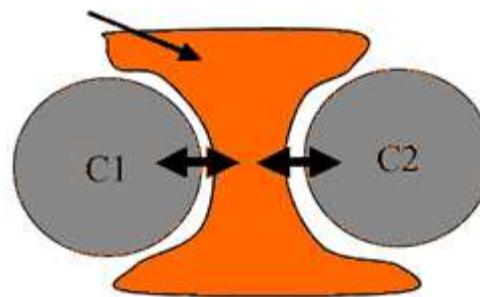


# Arhitektura zasnovana na komponentama

- Temeljni problem u širokoj uporabi komponenata je nepostojanje zajedničke platforme (okruženja) koja objedinjuje komponente.
- Danas komponente surađuju preko definiranog sučelja:



- Potreba za zajedničkom platformom (novim "operacijskim sustavom"):



- U arhitekturi sustava zasnovanog na web uslugama:
  - web = Operacijski sustav

# Najvažnije tehnologije

- Najznačajnija uloga u omogućavanju distribuirane komponentne arhitekture:
  - Object Management Group
    - CORBA - Common Object Request Broker Architecture
  - Oracle/Sun Microsystems
    - Java Beans, EJB(Enterprise Java Beans)
  - Microsoft – COM
    - Component Object Model
    - DCOM -Distributed COM
    - .NET Remoting
  - MPI Forum (40 organizacija)
    - MPI – Message Passing Interface

# CORBA

- Dobro: standard (OMG grupa), transparentna komunikacija između objekata koji su oblikovani različitim programskim jezicima i žive na različitim strojevima, u heterogenoj raspodijeljenoj okolini.
- Loše: Definirano na razini izvornog koda, a ne na binarnoj razini.
  - jako usporava rad jer se komunikacija odvija na visokoj razini definiranih protokola.
  - programski jezici moraju imati implementirane kopče za CORBA sučelje.

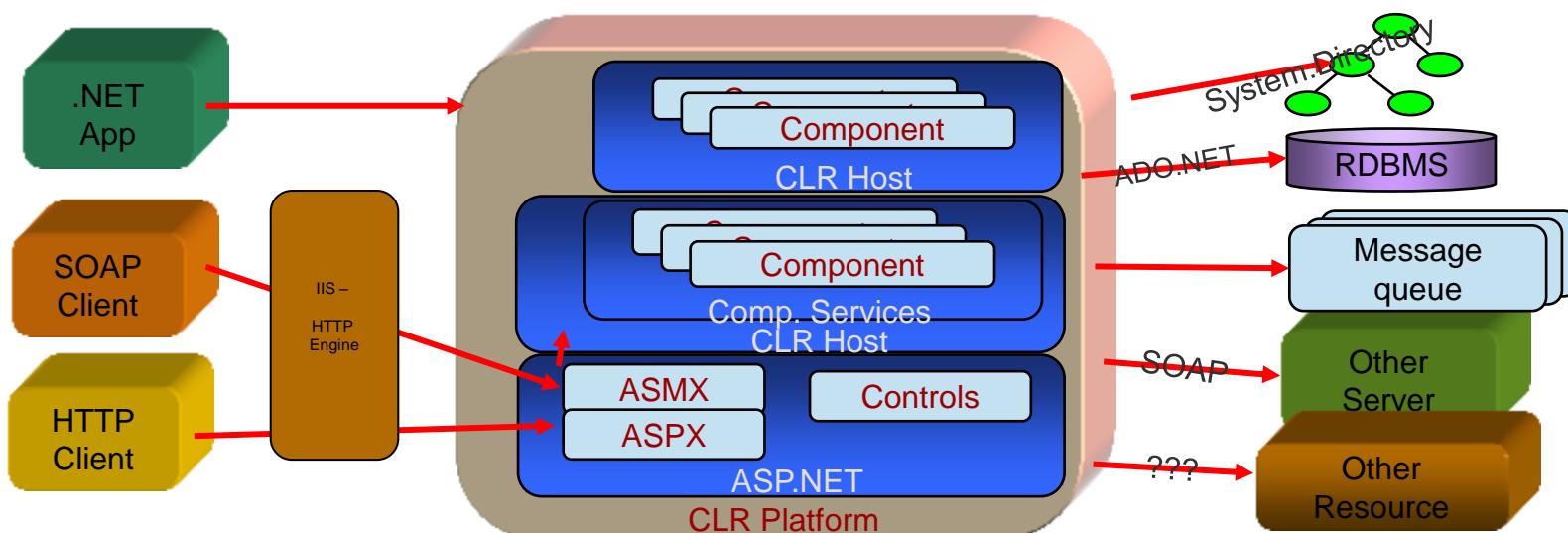
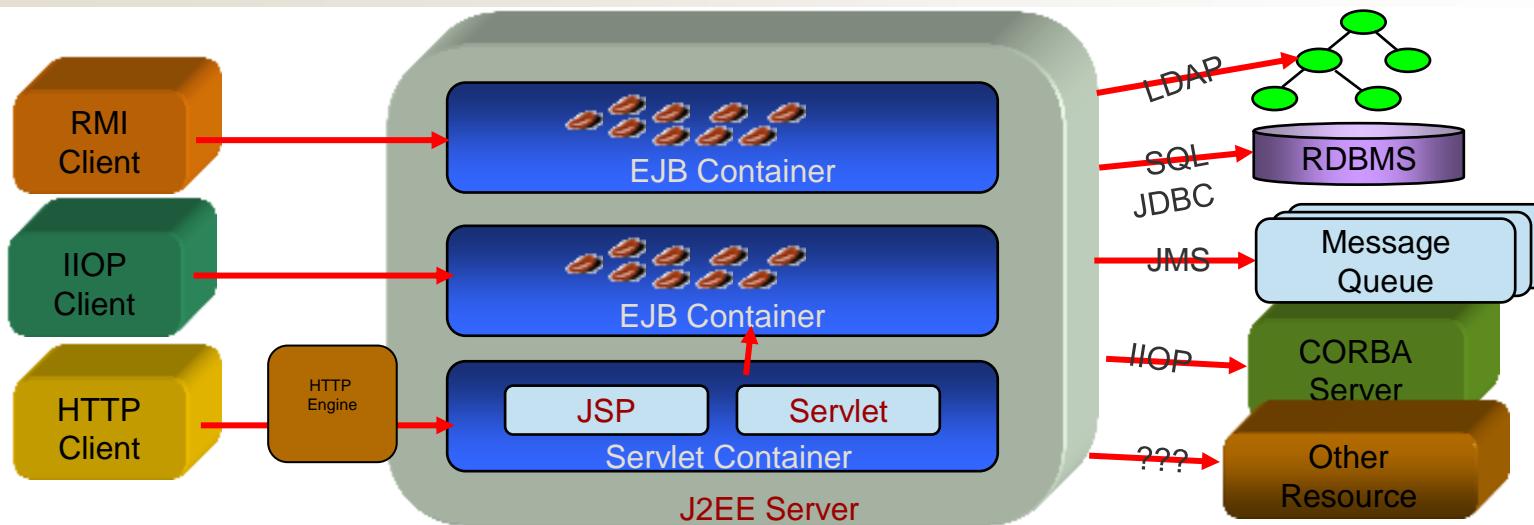
## Java/JavaBeans:

- Dobro: neovisnost o radnoj platformi,
  - kao reakcija na događaj Beans komponenta može komunicirati i spajati se s ostalim Beans komponentama,
  - Beans komponenta se može prilagođavati specijalnoj primjeni, dostupan izvorni kod.
- Loše: pretpostavka virtualnog stroja usporava rad,
  - otkriva se unutarnja struktura.

## .NET

- Dobro: binarni i mrežni standard za komunikaciju između objekata, primjena u više programskih jezika (C#, VB, Javascript, VisualC++),  
moguća je implementacija više globalno poznatih sučelja (prva metoda u sučelju je ***queryInterface()***, vraća oznaku ako sučelje nije podržano).
- Loše: upravljanje memorijom, kompatibilnost (Microsoft Windows).

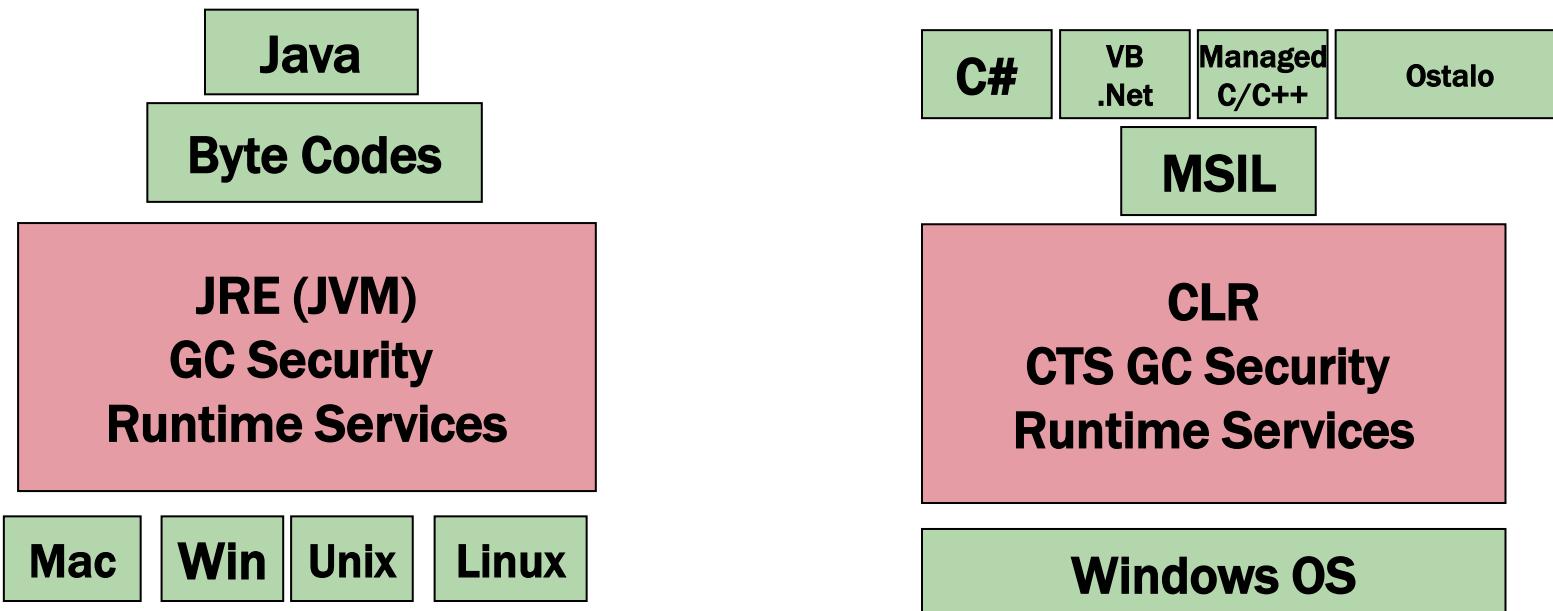
# Primjer: Arhitektura Java i .Net



# Usporedba JVM i CLR-a

## ■ Kako odabrat?

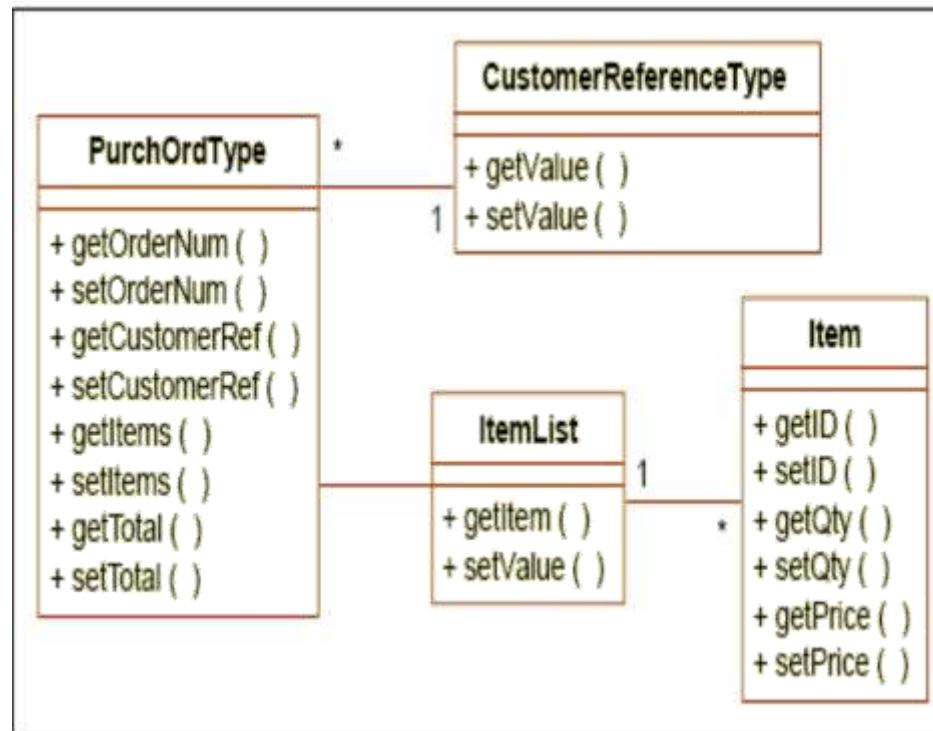
- netehnički parametri presudni!





# Primjer komponentnog modela

- **Komponentama** su pridruženi entiteti i pravila (npr. obuhvaća poslovni model)



- **Usluga** je građena od niza komponenti koje osiguravaju poslovnu funkciju predstavljenu uslugom

# Koraci oblikovanja zasnovanog na komponentama



## activities/

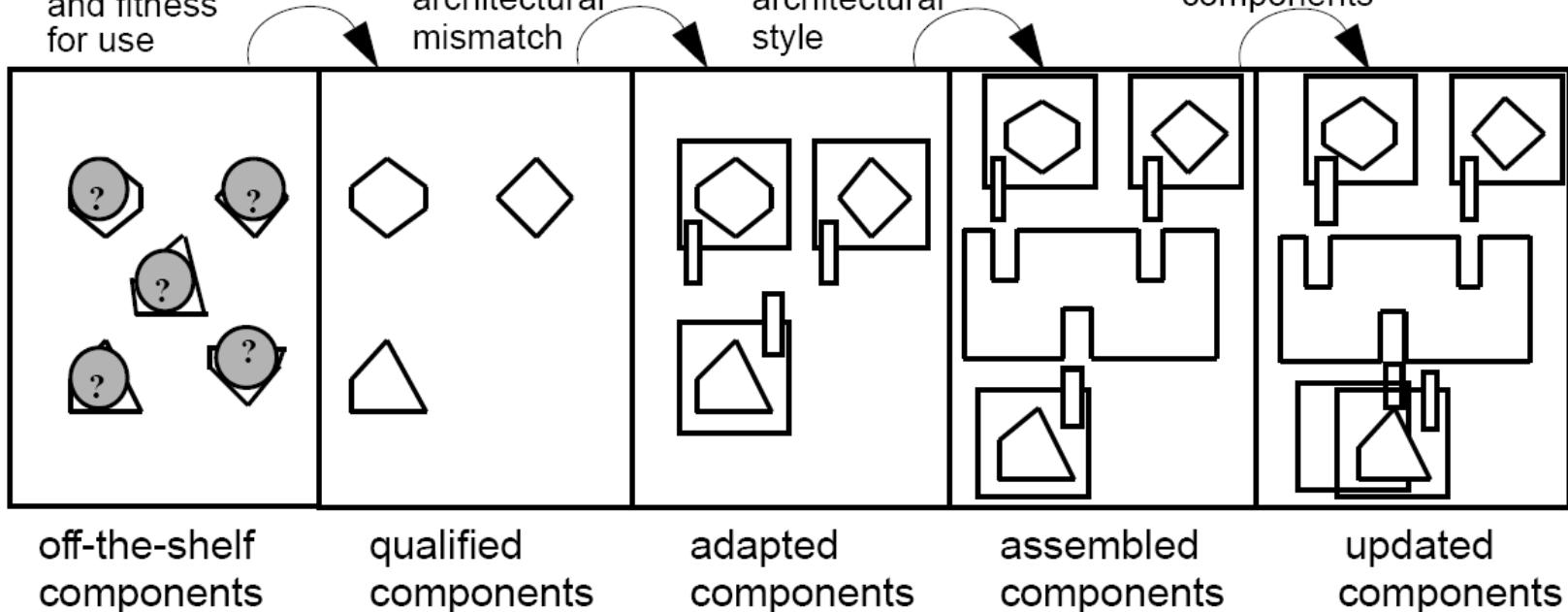
## transformations

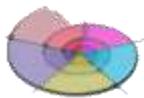
qualification to  
discover  
interface  
and fitness  
for use

adaptation to  
remove  
architectural  
mismatch

composition into  
a selected  
architectural  
style

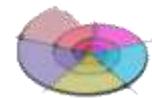
evolution to  
updated  
components



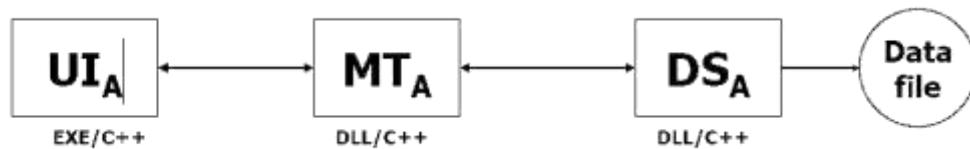


# Primjer komponentnog razvoja

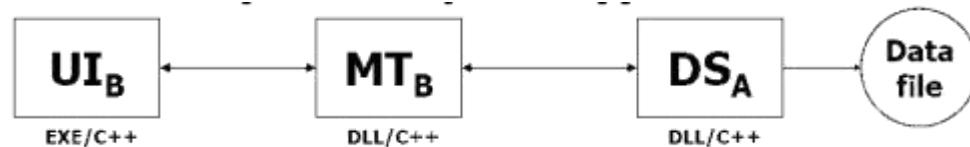
- Npr. razvoj programske potpore sustava naručivanja
- Komponente:
  - korisničko sučelje (UI)
  - poslovna logika (MT)
  - pristup podacima (DA)
- Komponente mogu imati inačice
- ⇒ inačice aplikacije tijekom razvoja
  - a) 1 stavka
  - b) više stavaka
  - c) GUI
  - d) jednostavan zapis podataka → Baza
  - e) klijent-poslužitelj CORBA, .....



a) 1 stavka



b) Više stavaka



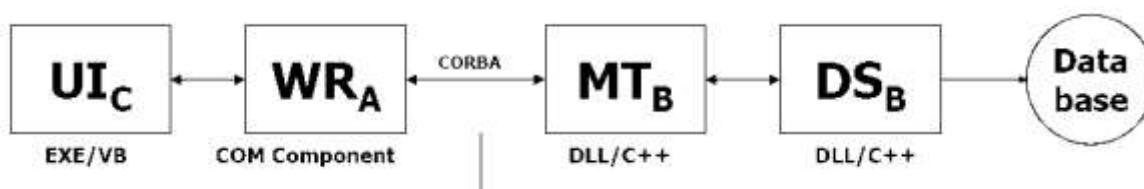
c) Gui



d) Jednostavan zapis podataka → Baza



e) Klijet-poslužitelj CORBA, .....





# Objektno usmjerena arhitektura



- U procesu oblikovanja programske potpore teži se povećanju **razine apstrakcije**.
- Objektno usmjerena arhitektura povećava razinu apstrakcije uvođenjem koncepta razreda i njihovih različitih odnosa te dinamičkih akcija i reakcija njihovih instanci (objekata).
- Još višu razinu apstrakcije moguće je postići uporabom **radnih okvira** (engl. *frameworks*) koji predlažu skup apstraktних razreda (vidi klijent-poslužitelj radni okvir) primјeren pojedinoj primjeni.
- Sljedeća, viša razina apstrakcije obuhvaća **posredničku i uslužnu arhitekturu**, te uporabu gotovih komponenata.
- Visoka razina apstrakcije postiže se uporabom gotovih **programskih obrazaca** (engl. *Design patterns*).
- Mnogi drže da se najviša razina apstrakcije postiže **na razini korisnika** koji bi mogli oblikovati primjenski program.
  - već danas postoje radna okruženja za modeliranje na toj razinu
  - engl. *Domain specific modeling*
  - *Consumer Computing* <http://ccl.fer.hr/>

# Trendovi

## ■ Microsoft:

- staro: Visual Basic, C++, COM, DCOM, DLLs, ...
- novo: .NET, VB.NET, C#, ASP.NET
- prednost - interoperabilnost

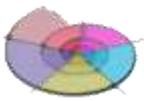
## ■ Java:

- trend EJB, J2EE, JAWS, JAX, ....
- prednost: jasnoća i kvaliteta

## ■ CORBA:

- web i XML podrška
- trend





# Sadašnje stanje



- U programskom inženjerstvu postoje programske utvrde (engl. *software fortress*)
  - programski sustavi namijenjeni općoj namjeni upravljeni od strane organizirane grupe pojedinaca
  - rade u uskoj sprezi na pružanju jednoznačne i smislene funkcionalnosti u neprijateljskom okruženju
- Osnovni tipovi: utvrde poslovnih aplikacija, utvrde prezentacijskog sučelja, utvrde tehnologija (npr. Web servisa), utvrde ugovornih odnosa, servisne utvrde, pravne utvrde,....
- Različiti neovisno razvijeni sustavi
- Složenost odnosa
  - Tehnološki ratovi (Windows ↔ Linux, .NET ↔ Java, ...)
  - Nepovjerenje (korisnika i razvojnih timova)
  - Nered podataka
  - Povjerenje u Internet – sigurnost
  - Bojazan promjene naslijeđenih aplikacija (engl. *legacy application*)

# Budućnost?

- Arhitektura programske podrške je više od strukture, ponašanja i pridruženih podataka
  - jednako važni upravljački, ugovorni i finansijski aspekti
- Napredne tehnike oblikovanja, povećanje udjela arhitekturnih informacija, kompleksniji gradivni blokovi ⇒
  - usmjerenost na arhitekturu
  - usmjerenost na ljude
    - današnja razina arhitekture naglašava utjecaj ljudskog čimbenika
      - Ispravan rad sustava;
      - Izrada u okviru troškova;
      - Omogućavanje timskog rada;
      - Pomoći održavanju sustava;
      - Razumijevanje sustava za sve dionike.

# Zaključci

- U procesu oblikovanja programske potpore teži se povećanju razine apstrakcije.
- Objektno usmjerena arhitektura povećava razinu apstrakcije uvođenjem koncepta razreda i njihovih različitih odnosa te dinamičkih akcija i reakcija njihovih instanci (objekata).
- Još višu razinu apstrakcije moguće je postići uporabom radnih okvira (engl. *frameworks*) koji predlažu skup apstraktnih razreda (vidi klijent-poslužitelj radni okvir) primјeren pojedinoj primjeni.
- Sljedeća, viša razina apstrakcije obuhvaća posredničku i uslužnu arhitekturu, tu uporabu gotovih komponenata.
- Visoka razina apstrakcije postiže se uporabom gotovih programskih obrazaca (engl. *design patterns*).
- Mnogi drže da se najviša razina apstrakcije postiže na razini korisnika koji bi mogli oblikovati primjenski program. Već danas postoje radna okruženja za modeliranje na toj razini (engl. *Domain Specific Modeling*).

GENERIČKE AKTIVNOSTI U PROCESU PROGRAMSKOG INŽENJERSTVA

# IMPLEMENTACIJA PROGRAMSKOG PROIZVODA

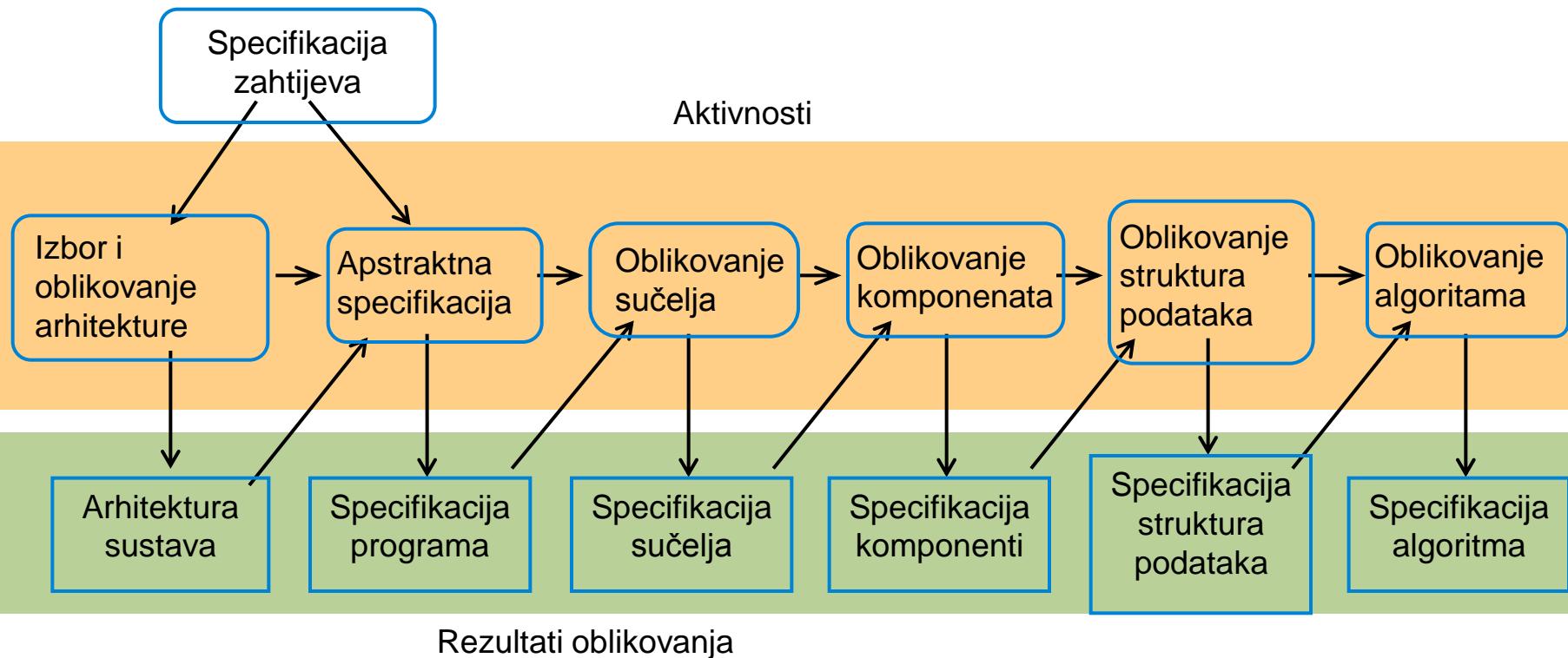


- Proces preslikavanja specifikacije u stvarni, realni sustav.
- ***Oblikovanje programske potpore***
  - oblikovanje strukture sustava koja realizira specifikaciju (izbor i modeliranje arhitekture).
- ***Implementacija***
  - preslikavanje strukture u izvršni program.
- Aktivnosti oblikovanja i implementacije su povezane i mogu biti isprepletene.



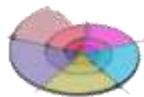
# Aktivnosti procesa oblikovanja

- Izbor i oblikovanje arhitekture
- Apstraktna specifikacija
- Oblikovanje sučelja
- Oblikovanje komponenata
- Oblikovanje struktura podataka
- Oblikovanje algoritama





- Implementacija
  - programiranje i otklanjanje pogrešaka
- Preslikavanje dokumentiranog oblikovanja u program i otklanjanje pogrešaka u programu
- Programiranje je osobna aktivnost
  - nema generičkog proces programiranja
- Programeri izvode neke aktivnosti ispitivanja (testiranja) s ciljem otkrivanja pogrešaka u programu i njihovog otklanjanja (engl. *debugging*), oblikovanje i programiranje ispitnih programa



# Implementacijski zahtjevi



## Dodatni zahtjevi implementacije

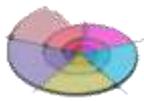
- Obratiti pažnju na zahtjeve ponovne uporabe
- Upravljanje konfiguracijom (engl. *Configuration management*)
  - konzistentno i automatizirano upravljanje verzijama
  - podržavanje timskog rada i integracije
  - evidencija i obrada pogrešaka
- Konfiguracija razvojne i ciljne okoline
  - razvoj i izvođenje se odvija na drugačijim sustavima
    - različite arhitekture i parametri sklopolja
    - različiti operacijski sustavi

## Uporaba integriranih razvojnih okolina

- engl. *Integrated development environments*
- skup razvojnih programskih alata koji podržavaju razvoj programa uporabom zajedničkog radnog okvira i korisničkog sučelja

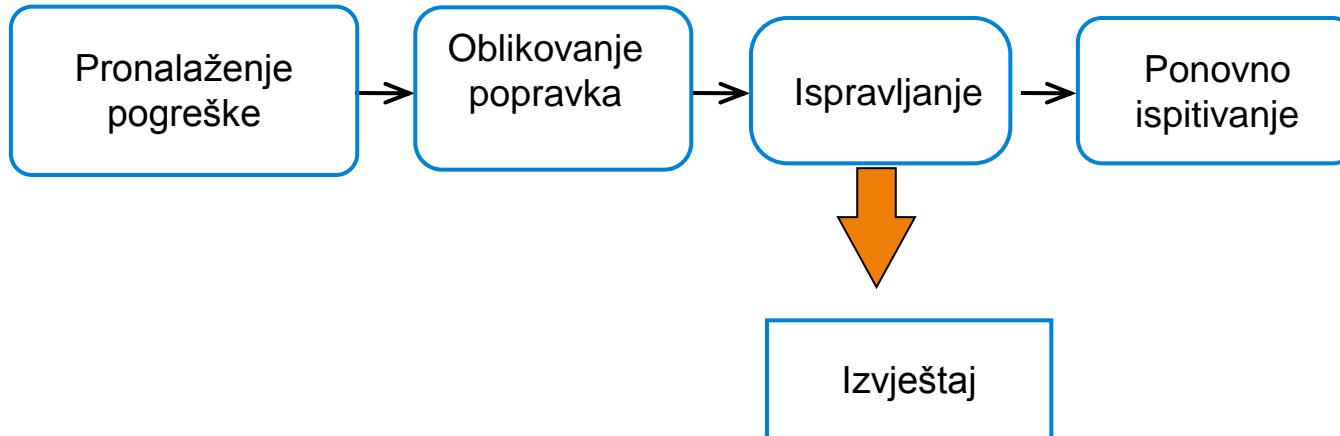
# Otvoreni kod

- engl. *Open source development*
- Pristup razvoju u kojem je izvorni kod javno objavljen i svi mogu doprinositi razvojnom procesu (predlagati promjene i poboljšanja)
  - Free Software Foundation: [www.fsf.org](http://www.fsf.org)
- Sve više kompanija prihvata takav pristup
  - poslovni model se na zasniva na prodaji programa nego na prodaji podrške
- Glavna otvorena pitanja:
  - kada i kako koristiti komponente otvorenog koda?
  - može li takav pristup biti uspješan?
- Modeli licenciranja
  - The GNU General Public License – GPL
    - recipročan pristup – sve otvoreno
  - The GNU Lesser General Public License – LGPL
    - nije potrebno objaviti izvorni kod svih komponenti
  - The Berkley Standard Distribution License - BSD
    - ne postoji obaveza objave promijenjenog i/ili dodanog koda
    - slobodna komercijalna prodaja



# Proces otklanjanja pogrešaka

- U okviru implementacije – programiranja



- Ispitivanje integrirane i cjelovite programske potpore
  - zasebna aktivnost koja ne spada u generičke aktivnosti implementacije!

# Diskusija

- 
- 
- 
- 
-

# Oblikovanje programske potpore

ak.god. 2014./2015.

## *Ispitivanje programske potpore*



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



# Tema

- Definicija i ciljevi ispitivanja programske podrške
- Klasifikacija ispitivanja
- Upoznavanje procesa ispitivanja
- Strategije ispitivanja
- Principi ispitivanja sustava i komponenti
- Funkcionalno i struktorno ispitivanje
- Generiranje ispitnih slučajeva
- Automatizacija procesa ispitivanja
  
- Cilj:
  - upoznavanje tehnika ispitivanja kao jedne od mogućih tehniku verifikacije programske podrške
  - razumijevanje terminologije, procesa i različitosti tehnika ispitivanja

# Literatura

- Sommerville, I., ***Software engineering***, 8th ed, Addison Wesley, 2007.
- Glen Myers, ***The art of software testing***, Second ed., John Wiley & Sons, New Jersey, 2004.
- S. Siegel, ***Object-Oriented Software Testing: A hierarchical Approach***, John Wiley & Sons, New Jersey, 1996.

Pripremio: Vlado Sruk

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

U pripremi materijala osim literature upotrijebljeni su i drugi izvori, te zahvalujem autorima.

GENERIČKE AKTIVNOSTI U PROCESU PROGRAMSKOG  
INŽENJERSTVA

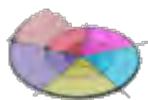
# **VALIDACIJA I VERIFIKACIJA PROGRAMSKOG PROIZVODA**



# Validacija i verifikacija programskog proizvoda



- Potrebno pokazati da sustav odgovara specifikaciji i da zadovoljava zahtjeve kupca i korisnika.
- **Validation** – “Are we building the right system?”
  - zadovoljava li sustav funkcijalne zahtjeve
  - provodi se ispitivanjem sustava
- **Verification** – “Are we building the system right?”
  - zadovoljava li sustav zahtjeve na ispravan način
  - uključuje **provjeru** poželjno zasnovanu na formalnim (matematičkim i logičkim) metodama.



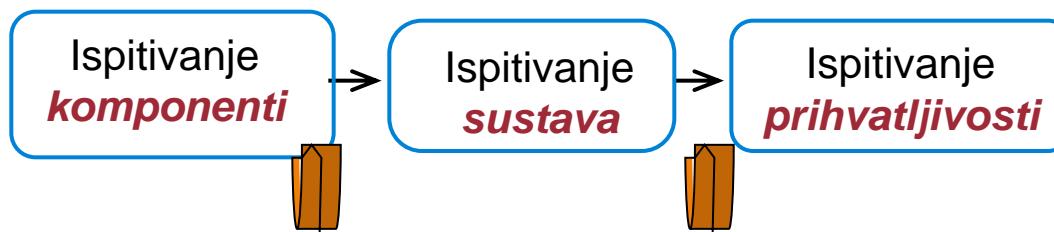
# Ispitivanje programske potpore

- Ispitivanje je nezaobilazna aktivnost u procesima programskog inženjerstva.
  - ispitivanje se dopunjuje formalnom verifikacijom.
- Ispitivanje sustava temelji se na radu sustava s ispitnim ulaznim parametrima (podacima) koji se izvode iz specifikacije realnih podataka koje sustav treba prihvati.
  - skupo (resursi, vrijeme)
- Pretpostavka za ispitivanje:
  - bez specifikacije nema ispitivanja!
- Ispitivanje znači usporedbu stvarnih rezultata s postavljenim standardima.

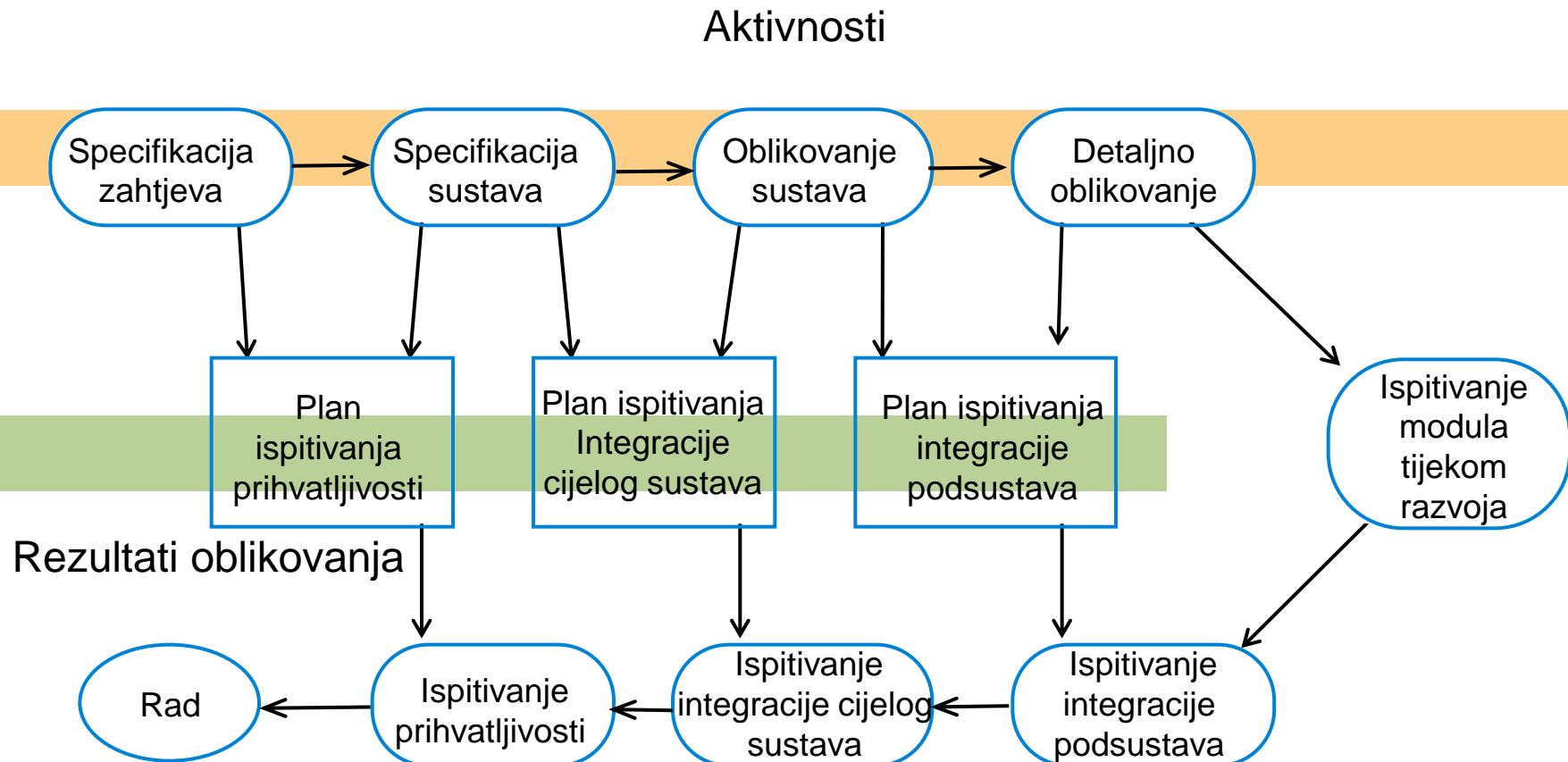


# Proces ispitivanja sustava

- Ispitivanje ***komponenti i modula***
  - individualne komponente se ispituju nezavisno.
  - komponente mogu biti funkcije, objekti ili koherentne skupine tih entiteta.
- Ispitivanje ***sustava***
  - ispitivanje cjelovitog sustava. Posebice je važno ispitivanje novih i nenadanih svojstava.
- Ispitivanje ***prihvatljivosti***
  - značajki na temelju kojih kupac prihvata i preuzima sustav (*engl. acceptance*).



# Aktivnosti i proizvodi procesa ispitivanja



## ■ Knowledge Areas

- Software Requirements
- Software Design
- Software Construction
- **Software Testing**
- Software Maintenance
- Software Configuration Management
- Software Engineering management
- Software Engineering Process
- Software Engineering Tools And Methods
- Software Processes and Product Quality

## Software Testing

1. Software Testing Fundamentals
  - 1.1. Testing-related terminology
  - 1.2. Key issues
  - 1.3. Relationships of testing to other activities
2. Test Levels
  - 2.1. The target of the test
  - 2.2. Objectives of Testing
3. Test Techniques
  - 3.1. Based on the software engineer's intuition and experience
  - 3.2. Specification-based techniques
  - 3.3. Code-based techniques
  - 3.4. Fault-based techniques
  - 3.5. Usage-based techniques
  - 3.6. Techniques based on the nature of the application
  - 3.7. Selecting and combining techniques
4. Test-related measures
  - 4.1. Evaluation of the program under test
  - 4.2. Evaluation of the tests performed
5. Test Process
  - 5.1. Practical considerations

# ISPITIVANJE



# Što je ispitivanje?

- Cilj ispitivanja je pokazati da program ispravno obavlja željene funkcije.
  - Ispitivanje je proces uspostave povjerenja ispravnog rada.
  - Ispitivanje je proces pokazivanja odsustva pogrešaka.
- 
- ***Ispitivanje je proces izvođenja programa sa svrhom pronalaženja pogrešaka.***
    - Može pokazati postojanje, ali ne i nepostojanje pogreške
- 1972 Dijkstra: “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence”
- Ispitivanje programa može biti vrlo učinkovit način za pokazati prisutnost pogrešaka, ali je beznadno neadekvatno za pokazati njihovu odsutnost
- Potreba za formalnom verifikacijom

# Ispitivanje

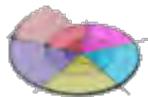
- Aktivnost s ciljem ***otkrivanja informacija*** o ispravnosti i kvaliteti, te ***poboljšanja*** pronalaženjem kvarova i problema ispitivane programske podrške.

IEEE: Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.

- Otkrivanje informacija
  - organizirana i detaljna potraga za relevantnim informacijama
  - aktivan proces istraživanja
    - postavljanje pitanja i analiza rezultata
- Za ostvarenje cilja upotrebljava:
  - eksperimentiranje;
  - logika, matematika;
  - modeli;
  - alati (programi, mjerni instrumenti, analizatori ...)

# Ciljevi ispitivanja

- Glavni ciljevi ispitivanja:
  - otkrivanje što većeg broja pogrešaka u danom vremenu u:
    - zahtjevima
    - oblikovanju sustava
    - implementaciji (programskom kodu)
    - resursima sustava
- Ispitivanje uobičajeno obuhvaća više ciljeva, a time zahtijeva primjenu različitih strategija i ispitivanja, dokumentaciju i daje različite rezultate



# Dodatni ciljevi ispitivanja

- Osigurati pouzdanost, ispravnost, otkrivanje pogrešaka
- Definirati način sigurne uporabe
- Omogućiti parametre za procjenu kvalitete
- Minimizirati rizike
  - provjeriti sukladnost rada (engl. *interoperability*) s ostalim komponentama
  - pomoći u donošenju odluke o puštanju u rad/prodaju
  - zaustaviti prerano puštanje u rad/prodaju (engl. *premature product releases*)
  - minimizirati troškove tehničke podrške
  - procijeniti sukladnost specifikacijama
  - sukladnost s normama (zakonske, tehničke, ...)
  - minimizirati rizik tužbi obzirom na sigurnost

# Primjer: Microsoft EULA<sup>1</sup>

18. EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, ***IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER*** (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) ***ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE*** THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THE GUARANTEE - THE SOFTWARE IS DESIGNED AND OFFERED AS A GENERAL-PURPOSE SOFTWARE, NOT FOR ANY USER'S PARTICULAR PURPOSE. YOU ACCEPT THAT NO SOFTWARE IS ERROR FREE AND YOU ARE STRONGLY ADVISED TO BACK-UP YOUR FILES REGULARLY. ....

B) ANY SUPPORT SERVICES PROVIDED BY MICROSOFT SHALL BE SUBSTANTIALLY AS DESCRIBED IN APPLICABLE WRITTEN MATERIALS PROVIDED TO YOU BY MICROSOFT AND MICROSOFT SUPPORT ENGINEERS WILL USE REASONABLE EFFORTS, CARE AND SKILL TO SOLVE ANY PROBLEM ISSUES.

<sup>1</sup>EULA = End User Licence Agreement

# The GPL 1

## 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. ***THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.*** SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

## 16. Limitation of Liability.

***IN NO EVENT*** UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ***ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY*** WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE ***LIABLE TO YOU FOR DAMAGES***, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ***ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM*** (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## 17. Interpretation of Sections 15 and 16.

IF THE DISCLAIMER OF WARRANTY AND LIMITATION OF LIABILITY PROVIDED ABOVE CANNOT BE GIVEN LOCAL LEGAL EFFECT ACCORDING TO THEIR TERMS, REVIEWING COURTS SHALL APPLY LOCAL LAW THAT MOST CLOSELY APPROXIMATES AN ABSOLUTE WAIVER OF ALL CIVIL LIABILITY IN CONNECTION WITH THE PROGRAM, UNLESS A WARRANTY OR ASSUMPTION OF LIABILITY ACCOMPANIES A COPY OF THE PROGRAM IN RETURN FOR A FEE.

<sup>1</sup> **GPL = General Public License (GNU GPL)**

# Zadaci ispitivača

- Pronalaženje mogućih pogrešaka u sustavu
  - Važna uloga u pronalaženju problema i poboljšanju kvalitete sustava
  - Mora poznavati ograničenja koja mogu dovesti do pogrešaka
  - U različitim fazama oblikovanja progrsmske podrške potrebna su specifična znanja
- 
- Poželjne karakteristike dobrog ispitivača:
    1. Tehničko razmišljanje
      - mogućnost modeliranja i razmjevanje uzroka i posljedica
    2. Kreativnost
      - generiranje novih ideja za ispitivanje i razmatranje najboljih opcija
    3. Kritičnost
      - evaluacija ideja i donošenje zaključaka
    4. Provođenje
      - provođenje ideja u praksi



# Svojstva ispitivanja

- Ispitivanje programske podrške zasniva se na **dinamičkoj verifikaciji ponašanja programa u izvođenju** na **konačnom broju ispitnih slučajeva**, pogodno odabranih iz uobičajeno beskonačne domene izvođenja, obzirom na očekivano ponašanje.
  - temelji se na provjeri rada sustava s ispitnim ulaznim podacima
    - uobičajeno se odabiru temeljem specifikacije stvarnih podataka koje sustav treba prihvati.
  - **apsolutno detaljno ispitivanje je nemoguće**
    - praktična ograničenja resursa i vremena (cijena)
- Tehnike ispitivanja se razlikuju u načinu odabira pogodnih kriterija i ispitnih slučajeva.
- Ispitivanje mora omogućiti donošenje odluke o prihvatljivosti i očekivanim rezultatima.
  - usporedba stvarnih rezultata rada s prethodno utvrđenim rezultatima

# Problem ispitivanja

- **Bez specifikacije nema ispitivanja!!**
- Ispitivanje znači usporedbu stvarnih rezultata s pretpostavljenim očekivanim rezultatima na temelju specifikacija
- *The Institute of Electrical and Electronics Engineers (IEEE) definira:*
  - **ispitni slučaj/test** - jedan ili više ispitnih slučaja/scenarija (*engl. Test case*)
  - **ispitivanje/testiranje** - proces analize programskog koda sa svrhom pronašlaska razlike između postojećeg i zahtijevanog stanja (pogreška, kvar, bug), te vrednovanja svojstava programa  
⇒ Odgovornost za verifikaciju i validaciju
  - **Verification:** "Are we building the product right"
    - *Ponašanje programa prema specifikacijama (odsustvo kvarova)*
    - *Unit testing, integrating testing, formal verification*
  - **Validation:** "Are we building the right product"
    - *Program mora odgovarati zahtjevima (zadovoljstvo korisnika)*
    - *prototipovi, ispitivanje prihvatljivosti (engl. acceptance testing)*

# Verifikacija i ispitivanje

- Tehnike verifikacije programa
  - staticka verifikacija
    - ispitivanja strukture (engl. *Structure Walkthroughs*)
    - provjere ispravnosti
  - dinamička verifikacija
    - ispitivanje izvođenjem programa sa stvarnim podacima
  - formalna verifikacija
    - primjena formalnih metoda matematičke logike za dokaz ispravnosti programa

# Statička verifikacija

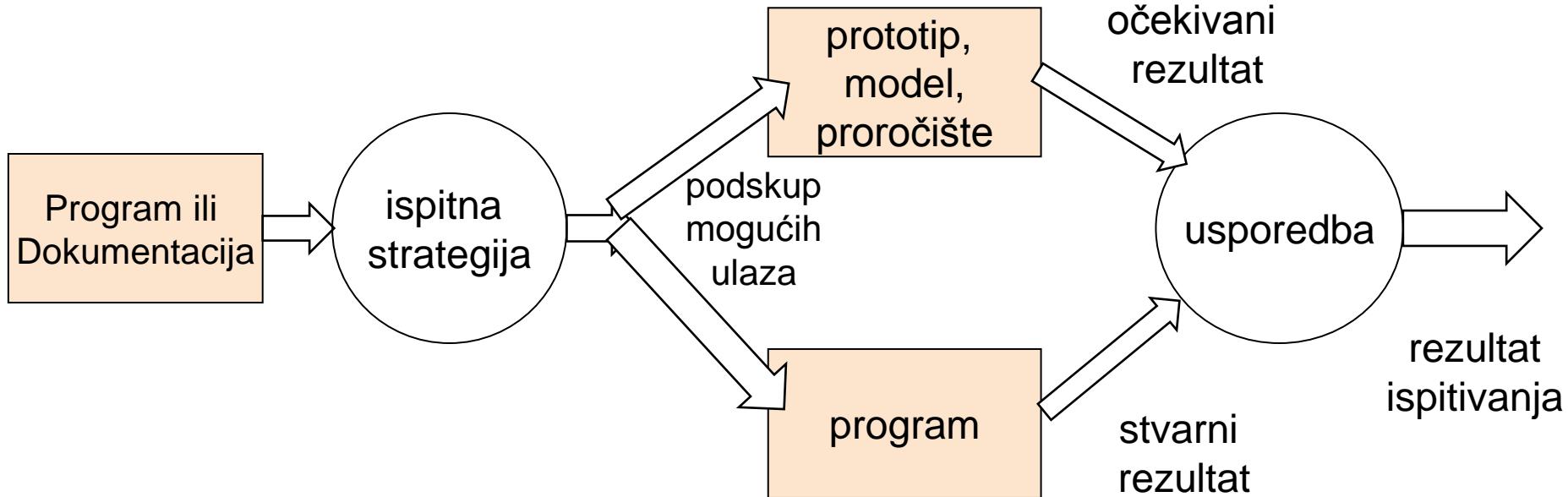
- Statička verifikacija se provodi na specifikaciji zahtjeva, raznim nivoima oblikovanja sustava i programskom kodu
  - nadzor izvornog koda engl. *Software inspections, Walkthroughs*
  - analizatori programa engl. *Static analyzers*
    - otkrivanje nepravilnosti npr. *LINT, Jtest, StyleCop*
  - formalne metode
- Tehnike ispitivanja i staticke verifikacije su međusobno komplementarne i nadopunjaju se
- Efikasan način pronalaženja potencijalnih grešaka koje mogu dovesti do zatajenja



# Nadzor izvornog koda

- Analiza statičkih artefakata s ciljem otkrivanja problema
- Postupak davanja ekspertnih mišljenja, recenzija programskog produkta
  - ljudi provjeravaju izvorne artefakte
  - ne zahtijeva izvođenje
- U praksi često upotrebljavane i efikasne tehnike
  - provjera usklađenost sa specifikacijama
  - ne može provjeravati nefunkcionalna svojstva
- Prolazak, češljanje (*engl. Walkthroughs*)
  - neformalan nadzor i inspekcija izvornog koda ili dokumentacije, (često) inicirana od strane autora
- Nadzor, inspekcija (*engl. Software inspections*)
  - svrha je utvrđivanje usklađenosti sa standardima ili zahtjevima
  - usporedba dokumenata oblikovanja, koda i ostalih artefakata
  - zahtjeva planiranje i raspodjelu zadaća, formalno bilježenje i obradu rezultata

# Proces ispitivanja



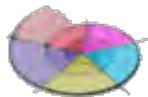
Ispitivanja koja se provode bez poznatog razloga uobičajeno nisu od koristi!

- uočene pogreške
- sukladnost zahtjevima
- performanse
- potvrda kvalitete

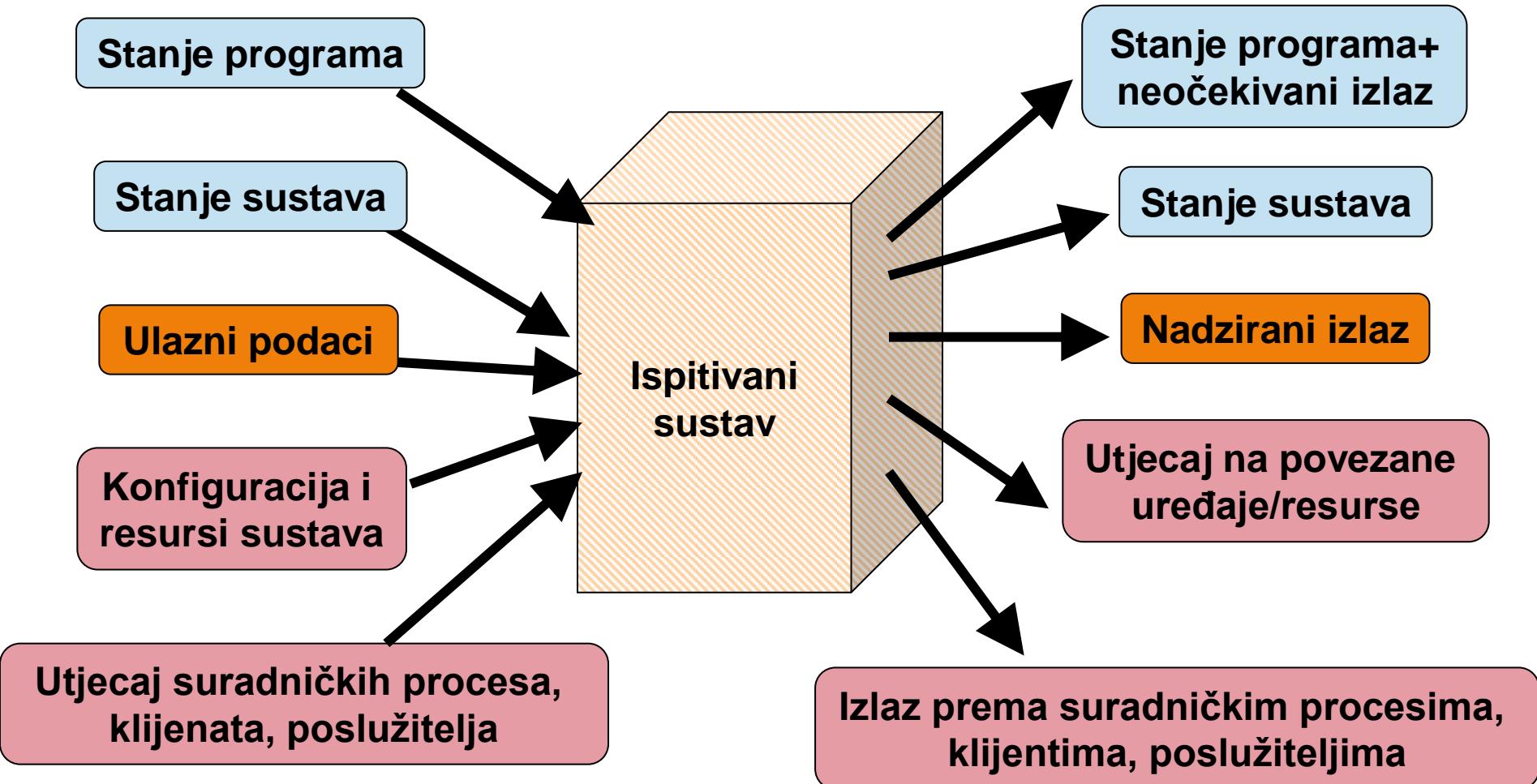
# Cijena ispitivanja

- Programi definiraju ponašanje ključne infrastrukture
  - velika odgovornost
    - širina primjene, složenost, broj korisnika
- Troškovi ispitivanja dosežu ~ 50% (htjeli mi to ili ne!!)
  - u stvarnosti ispitivanje često započinje tek po završetku oblikovanja i implementacije
  - preskakanje ranog ispitivanja uobičajeno podiže troškove
  - složeni sustavi i sklopovsko programski sustavi zahtijevaju više ispitivanja
- Koja je cijena ne provođenja ispitivanja?
- Stupanj ispitivanja izravno utječe na mogućnost pružanja odgovornosti za proizvod (engl. *product liability*)
  - kako se osigurati?

# KVAROVI I ZATAJENJA PROGRAMSKE PODRŠKE



# Uzročnici zatajenja programa





# Kvar, pogreška, zatajenje

## ■ **Kvar** (engl. *fault*)

- proglašeni uzročnik kvara – staticki fizikalni svijet
- može biti prikiven neko vrijeme

## ■ **Pogreška** (engl. *error*)

- dio stanja sustava odgovorno za stvaranje zastoja
- manifestacija kvara

## ■ **Zatajenje** (engl. *failure*)

- sustav ne zadovoljava specifikacije
  - problem vidljiv izvan sustava
  - neispravno ponašanje obzirom na zahtjeve ili opis očekivanog ponašanja

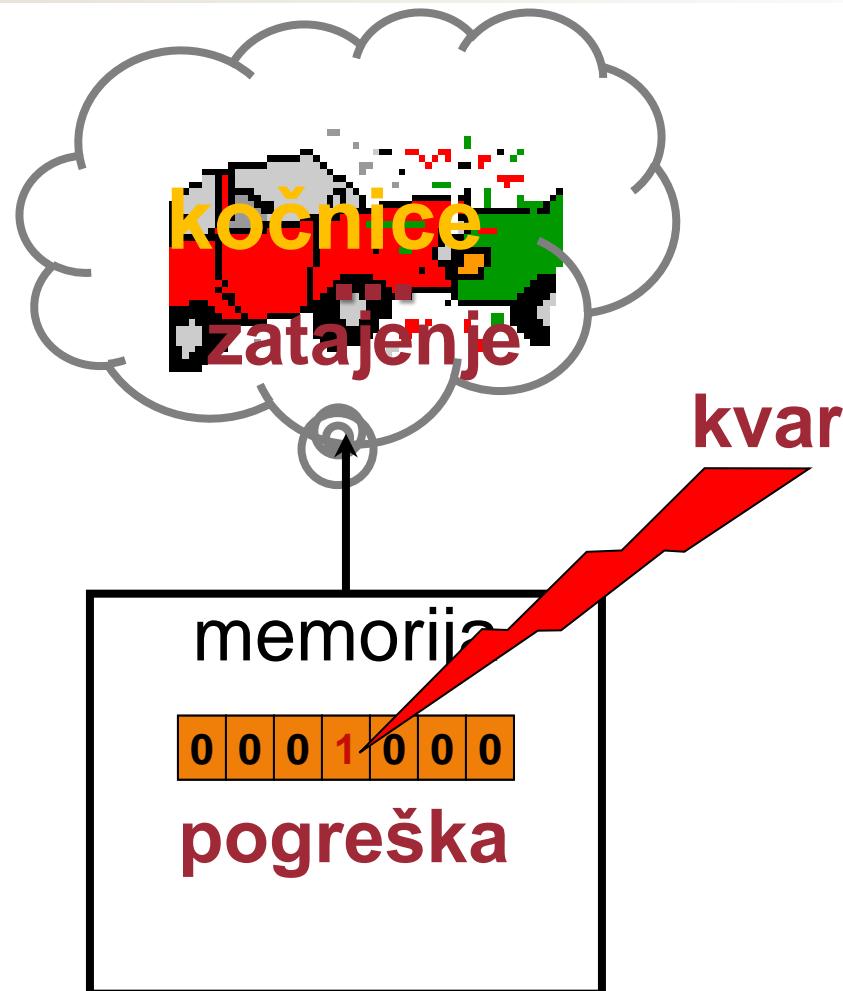


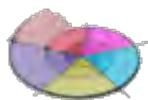
fizikalni svijet

informacija

vanjska  
pojavnost

# Primjer





# Odnos pogreška i zatajenja

## ■ Kvar

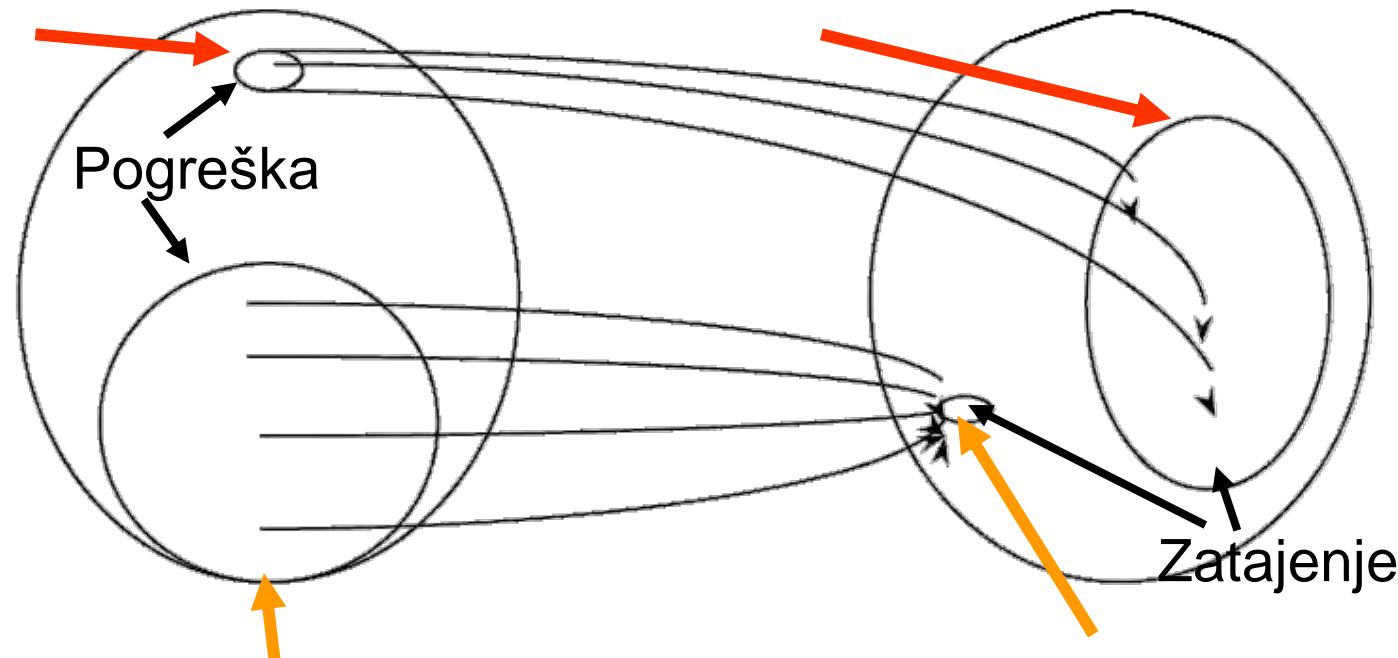
- unosi se pri oblikovanju ili programiranju

## ■ Pogreška uvođenje kvara u programsку potporu (dokumentacija/program)

- uzrokuje pogrešku obrade/izvođenja programa i dovodi do zatajenja

## ■ Paretov princip (engl. *Pareto principle, Law of the vital few*)

- mali broj pogrešaka dovodi do velikog broja zatajenja (20/80)





# Primjer: kvar i pogreška

- Kvar specifikacije sučelja
  - nepodudaranje formata poruka klijenta i poslužitelja
  - nepodudaranje zahtjeva i implementacije
- Kvar u algoritmima
  - engl. *Algorithmic Faults*
  - nedostatak inicijalizacija
  - pogrešna grananja
  - zanemarivanje rukovanja nul vrijednostima
- Mehanički kvar
  - engl. *Mechanical Faults*
  - dokumentacija nije sukladna stvarnom stanju
- Pogreške
  - pogreška izazvana gubitkom poruka pri opterećenu
  - pogreške izazvana ograničenjima memorije
  - vremenske pogreške
  - ...
- *primjeri vrsta kvarova:*
  - aritmetički
  - logički
  - sintaksni
  - memorijski
  - dretveni
  - sučelja
  - performansi
  - timski
  - ...



# Zatajenje programske potpore

- *engl. software failure*
- Mogući slučajevi:
  - ne ispunjava očekivanja zahtjeva
  - zadovoljava zahtjeve no ne i očekivanja korisnika
- Razlozi zatajenja:
  - zahtjevi su nepotpuni, nekonzistentni, nemogući za implementaciju
  - pogrešna interpretacija zahtjeva
  - kvar u oblikovanju arhitekture
  - kvar u oblikovanju programa
    - upotrijebljen neodgovarajući algoritam
  - kvar u programskom kodu
    - problem implementacije
  - dokumentacija nekorektno opisuje ponašanje sustava

# Posljedice pogrešaka

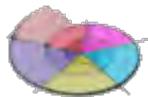


- *Tipovi pogrešaka*
  - blage (engl. *mild*)
  - dosadne (engl. *annoying*)
  - uznemiravajuće (engl. *disturbing*)
  - ozbiljne (engl. *serious*)
  - granične (engl. *extreme*)
  - katastrofalne (engl. *catastrophic*)
  - zarazne (engl. *infectious*)
- *Kategorije pogrešaka*
  - funkcijeske, sistemske, podatkovne, pogreške kodiranja, projektiranja, dokumentacije, .....
- *IEEE std. 1044.1 IEEE guide to classification for software anomalies*

# Uvjeti zatajenja

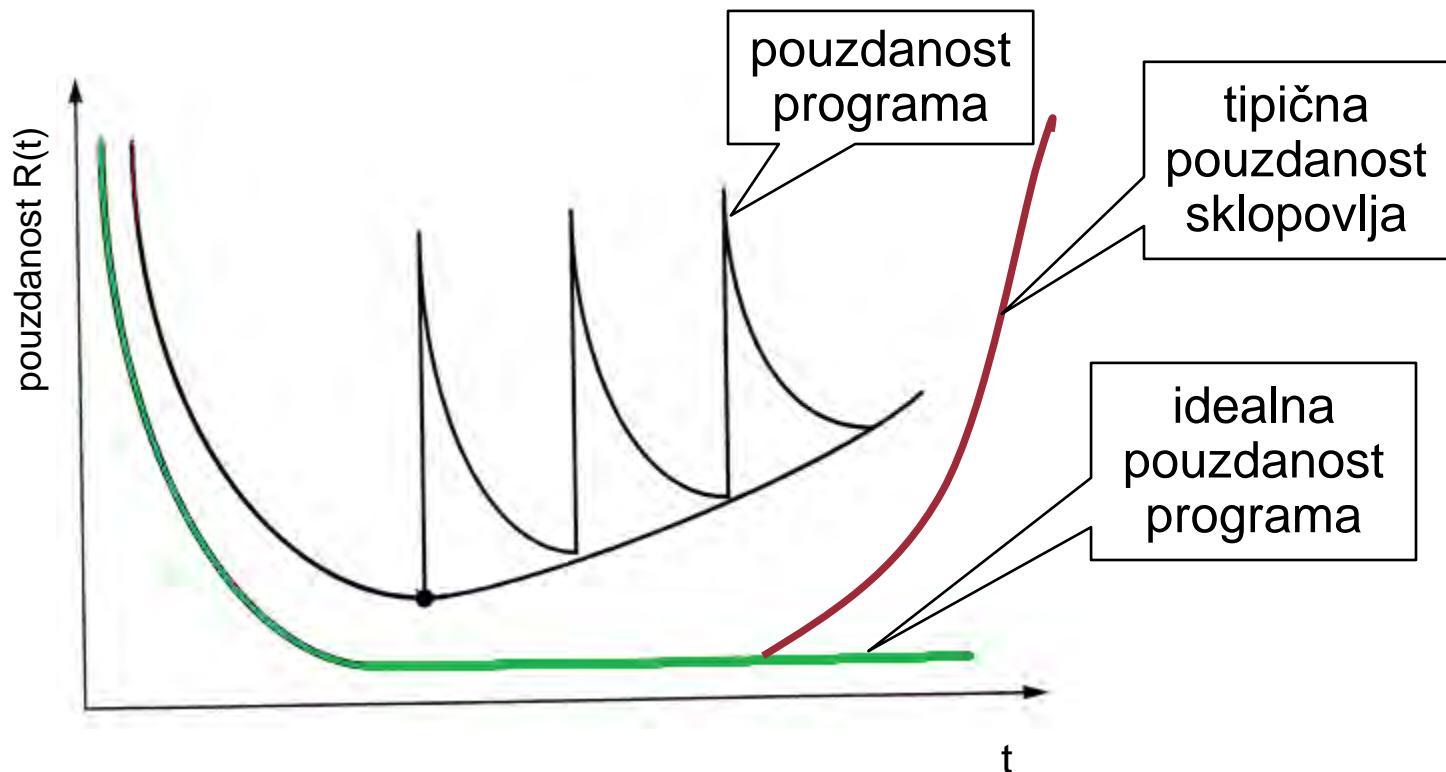
- Za manifestaciju zatajenja potrebno je ispuniti uvjete:

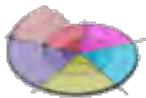
1. Doseg (engl. *reachability*)
  - mjesto kvara u programu mora biti dohvaćeno
2. Infekcija (engl. *infection*)
  - stanje programa mora biti neispravno
3. Propagacija (engl. *propagation*)
  - inficirano stanje mora uzrokovati promjenu nekog izlaza programa



# Pouzdanost programa

- Ispravljanje programa i uvođenje promjena programa ili sklopolja dovodi do odstupanja od idealne krivulje pouzdanosti

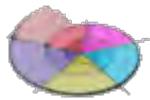




# Obrada pogrešaka

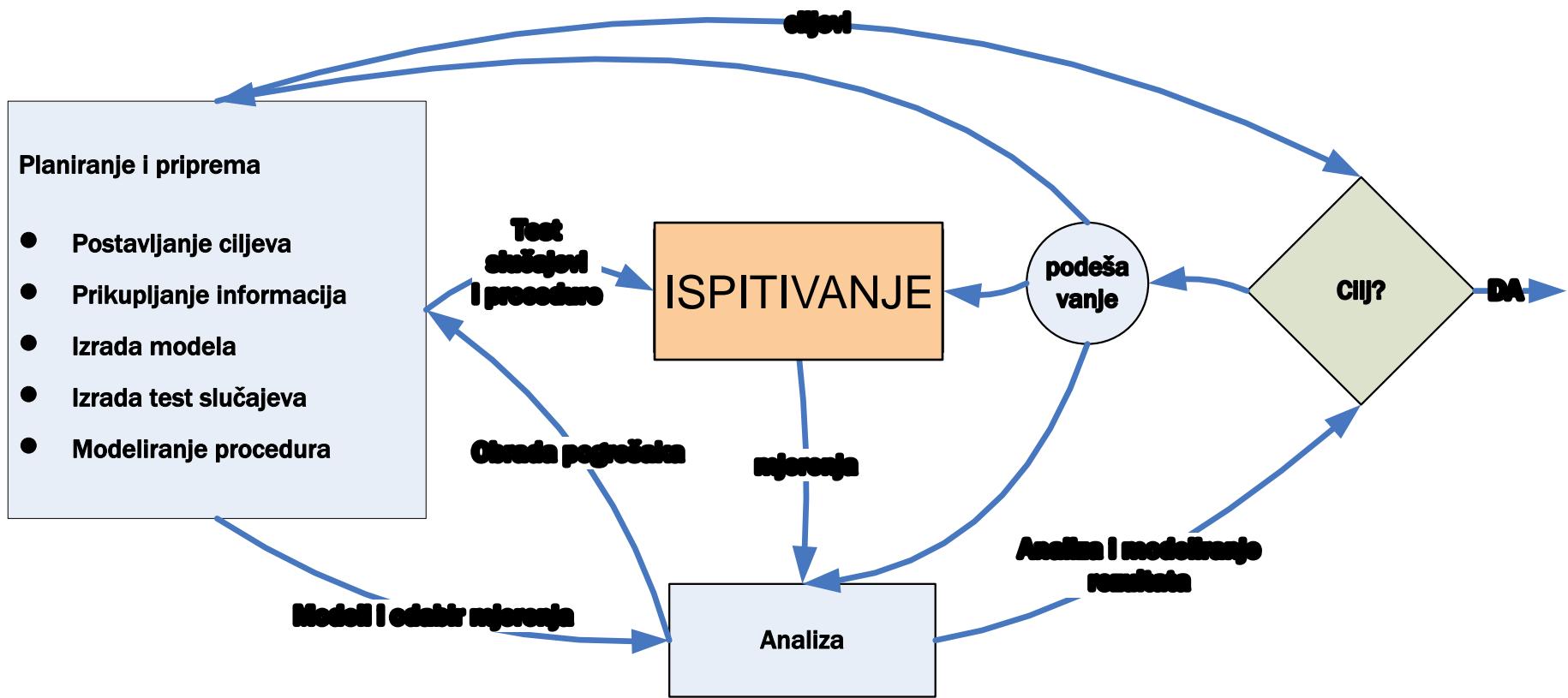
- Pogreške u složenim programskim sustavima su neizbjegne te tražimo načine njihove minimizacije
- **Prevencija** – engl. *Error prevention*
  - uporaba pogodnih metoda oblikovanja za smanjenje složenosti
  - sprečavanje nekonzistentnosti - npr. *CVS*, ...
  - primjena verifikacije za sprječavanje kvarova u algoritmima
- **Detekcija** - engl. *Error detection*
  - tijekom rada
    - *Ispitivanje*
    - Ispravljanje pogrešaka – engl. *Debugging*
    - Nadzor rada – engl. *Monitoring*
- **Oporavak** - engl. *Error recovery*
  - u radu programa
    - npr. *djeljenje s nulom*
  - baze podataka
    - ponavljanje transakcija
  - modularna zalihost - redundancija

# PROCES ISPITIVANJA



# Aktivnosti ispitivanja

## Prikaz osnovnog tijeka procesa ispitivanja





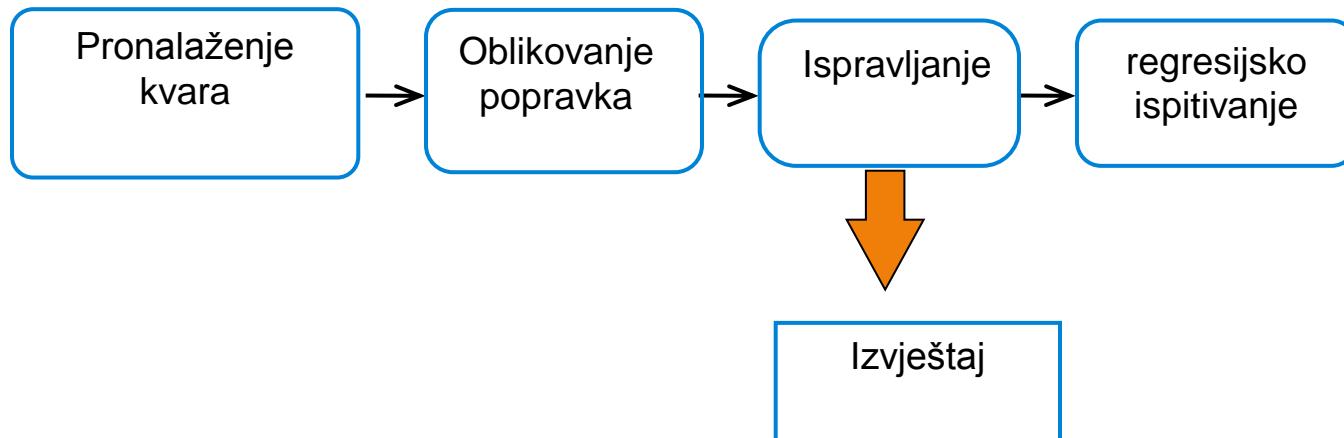
# Aktivnosti ispitivanja

- Neophodna različita znanja i vještine
  - postavljanje ciljeva ispitivanja
  - oblikovanje ispitnih slučajeva
  - izrada (pisanje) ispitnih slučajeva
  - ispitivanje svih ispitnih slučajeva
- Četiri osnovne kategorije:
- **Oblikovanje ispitivanja (engl. *Test design*)**
  - oblikovanje ispitnih vrijednosti sa svrhom zadovoljenja ciljeva ispitivanja
  - analogno oblikovanju arhitekture programske podrške
- **Automatizacija ispitivanja (engl. *Test automation*)**
  - programiranje
- **Ispitivanje (engl. *Test execution*)**
  - provođenje ispitivanja i bilježenje rezultata
- **Valorizacija ispitivanja (engl. *Test evaluation*)**
  - poznavanje domene i postupaka ispitivanja



# Otkrivanje kvara

- Pokreće proces otklanjanja kvara (*engl. debugging*)
  - u okviru implementacije – programiranja
- Zahtijeva analizu i otkrivanje neispravnosti
- Odgovarajuće promjene s ciljem uklanjanja neispravnosti



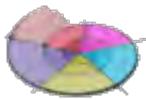


# Regresijsko ispitivanje

- engl. *Regression Testing*
- Ponovljeno ispitivanje nakon promjene/popravka
- Ispitivanje ispravljenih programa s ciljem potvrđivanja ispravnosti promjena i ne postojanja negativnog utjecaja na nepromijenjene dijelove programa
- Utjecaj ispravaka programa na ispitivanje:
  - neki ispitni slučajevi postaju nepotrebni
  - promjene očekivanih rezultata ispitnih slučajeva
  - izrada novih ispitnih slučajeva
- Provoditi:
  - tijekom integracije
  - nakon nadogradnji

# Struktura plana ispitivanja

- Opis procesa (odabrani standard, razlozi, ...)
- Sljedivost zahtjeva (*engl. Requirements traceability*)
  - omogućavanje sustavnog praćenja pojedinih zahtjeva i njegovih promjena
- Elementi ispitivanja (*engl. Tested items*)
  - što ispitujemo, što ne ispitujemo + razlozi
- Vremenik ispitivanja (*engl. Testing schedule*)
  - strategija, prioriteti, resursi ...
- Procedura bilježenja rezultata (*engl. Test recording procedures*)
  - baza podataka, elementi, automatizacija...
- Zahtjevi okoline (*engl. HW & SW requirements*)
- Ograničenja
  - opis planiranih ograničenja ispitivanja



# Povijest ispitivanja

razina

- Evolucija koncepta ispitivanja:
- – 1956 Prva faza: **ispravljanje pogrešaka** (engl. *Debugging*)
  - ispitivanje = provjera rada programa i ispravljanje
  - engl. *check-out & debugging*0
- 1957 – 1978 Druga faza: **Orijentacija na demonstraciju** (engl. *Demonstration*)
  - **razdvajanje provjere rada programa i ispravljanja pogrešaka;**
  - ispitivanje pokazuje ispravan rad (tipični ulazi), sukladnost specifikaciji1
- 1979 – 1982 Treća faza: **Orijentacija na razaranje** (engl. *Destruction*)
  - ispitivanje = izazivanje zatajenja sa svrhom otkrivanja pogrešaka
  - uspješno ispitivanje otkriva zatajenje2
- 1983 – 1987 Četvrta faza: **Orijentacija na evaluaciju** (engl. *Evaluation*)
  - ispitivanje = dio validacije i verifikacije
  - rano otkrivanje pogrešaka u zahtjevima, oblikovanju i implementaciji3
- 1988 – 2000 Peta faza: **Orijentacija na prevenciju** (engl. *Prevention*)
  - ispitivanje = jedan od načina pokušaja izbjegavanja pogrešaka
  - prevencija pogrešaka u zahtjevima, oblikovanju i implementaciji4
- 2000 – Šesta faza: **Orijentacija na razvoj prog. potpore** (engl. *Discipline*)
  - engl. *test-driven software development*
  - rezultira programskim kodom podobnim za ispitivanje5

# Standardi ispitivanja

- U uporabi se nalazi velik broj definiranih standarda ispitivanja
- Standardi osiguranja kvalitete *engl. Quality Assurance standards*
  - definiraju koja ispitivanja je nužno provesti
  - npr. *ISO 9003*
- Industrijski standardi *engl. Industry-specific standards*
  - specificiraju razine ispitivanja
  - npr. *DO178b*
- Standardi ispitivanja programske potpore *engl. Testing standards*
  - specificiraju kako provesti ispitivanje



# Standardi ispitivanja programske potpore

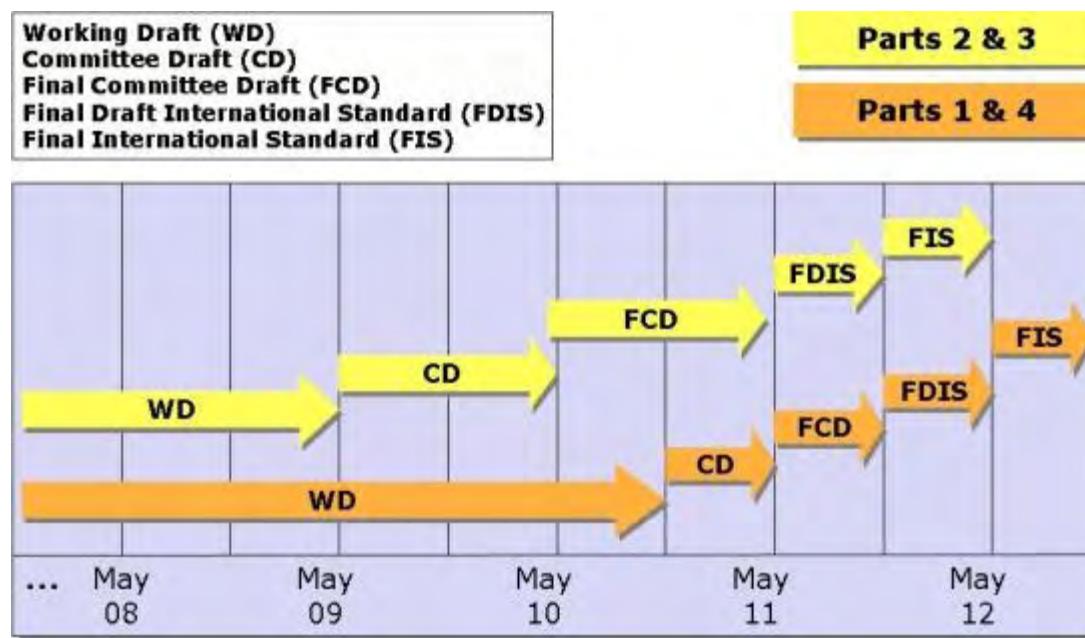


- **IEEE Standard 830-1998** - Software Requirements Specifications
- **IEEE Standard 829-1998** - Software Test Documentation
  - P829/D11, Feb 2008 Draft IEEE Standard for software and system test documentation (Revision of IEEE 829-1998)
- **IEEE Standard 1008-1987** - Software Unit Testing
- **IEEE Standard 1012-1986** - Software Verification and Validation Plans
- **ISO 9126** - Standard for the evaluation of software quality
- **ISO 25000:2005** - Software product Quality Requirements and Evaluation
- **ISO/IEC 90003:2004 - Software engineering** - Guidelines for the application of ISO 9001:2000 to computer software
  - quality management standard for computer software and related services.



# Standard ISO/IEC 29119

- ISO/IEC 29119 Software Testing
  - ISO/IEC JTC1/SC7 Working Group 26
- Razvoj započeo 2007. godine
- Cilj je razviti jedan standard koji će pokrivati cijeli životni ciklus ispitivanja programske podrške



# Primjer dokumenata

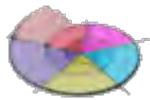
- IEEE Std. 829-2008 *Standard for Software Test Documentation*  
*Contents*

1. Plan ispitivanja – engl. *Test Plan*
  - Glavni plan ispitivanja i osnovne razine.
  - *plan provođenja ispitivanja*
2. Oblikovanje ispitivanja - engl. *Test Design Specification*
  - Svojstva, pristup i rezultata ispitivanja.
  - *odлука što treba ispitivati*
3. Ispitni slučajevi - engl. *Test Case Specification*
  - Opis ispitnih slučajeva.
  - *izrada ispitnih slučajeva*
4. Procedure ispitivanja- engl. *Test Procedure Specification*
  - Koraci provođenja ispitivanja.
  - *kako se izvodi ispitivanje*
5. Bilješke ispitivanja - engl. *Test Log*
  - Zapis tijeka provođenja ispitivanja.
6. Izvješća odstupanja - engl. *Test Incident Report*
  - Bilješke anomalija u zahtjevima, oblikovanju, kodu ili provođenju ispitivanja.
7. Sažetak izvješća ispitivanja - engl. *Test Summary Report*
  - Završno izvješće



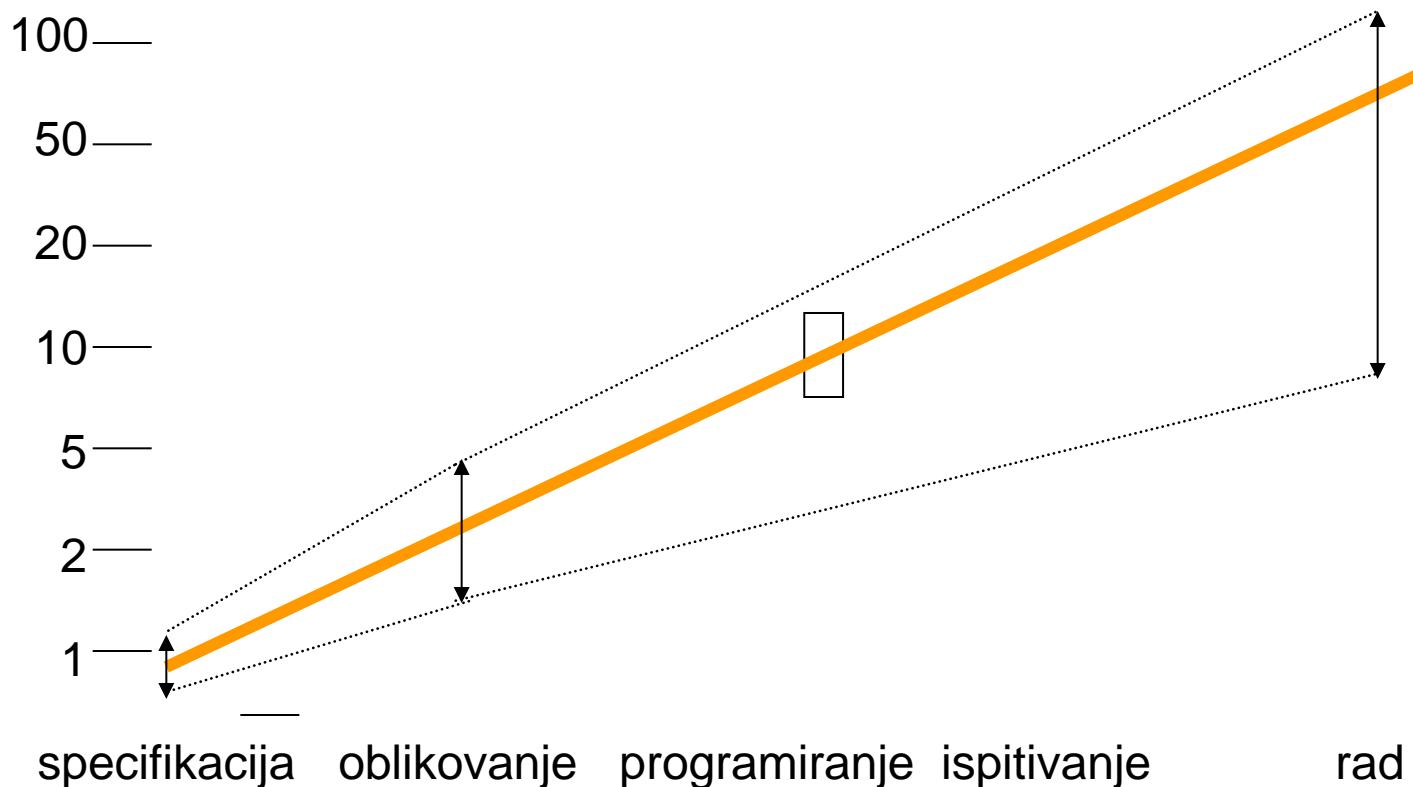
# Struktura tima za ispitivanje

- Voditelj projekta (*engl. Test Manager*)
  - upravlja procesom ispitivanja i potrebnim resursima
- Analitičar
  - analiza poslovnih procesa, zahtjeva i funkcijalne specifikacije
  - planiranje ispitivanja
- Voditelj upravljanja kvalitetom (*engl. Quality Assurance*)
  - definiranje standarda ispitivanja
  - praćenje i osiguranje sukladnosti procesa ispitivanja
- Voditelj ispitivanja
  - analiza zahtjeva ispitivanja
  - oblikovanje strategije i metodologije ispitivanja
  - oblikovanje ispitnih slučajeva i podatka
- Profesionalni ispitivači (*engl. Software Test Engineers*)
  - priprema ispitivanja
  - provođenje ispitivanja
  - pronalaženje pogrešaka
- Korisnici



# Trošak ispravljanja pogreške

- Ispravljanje uočenih pogrešaka u ranim fazama izgradnje programske potpore najjeftinije
  - raste s gotovošću programa



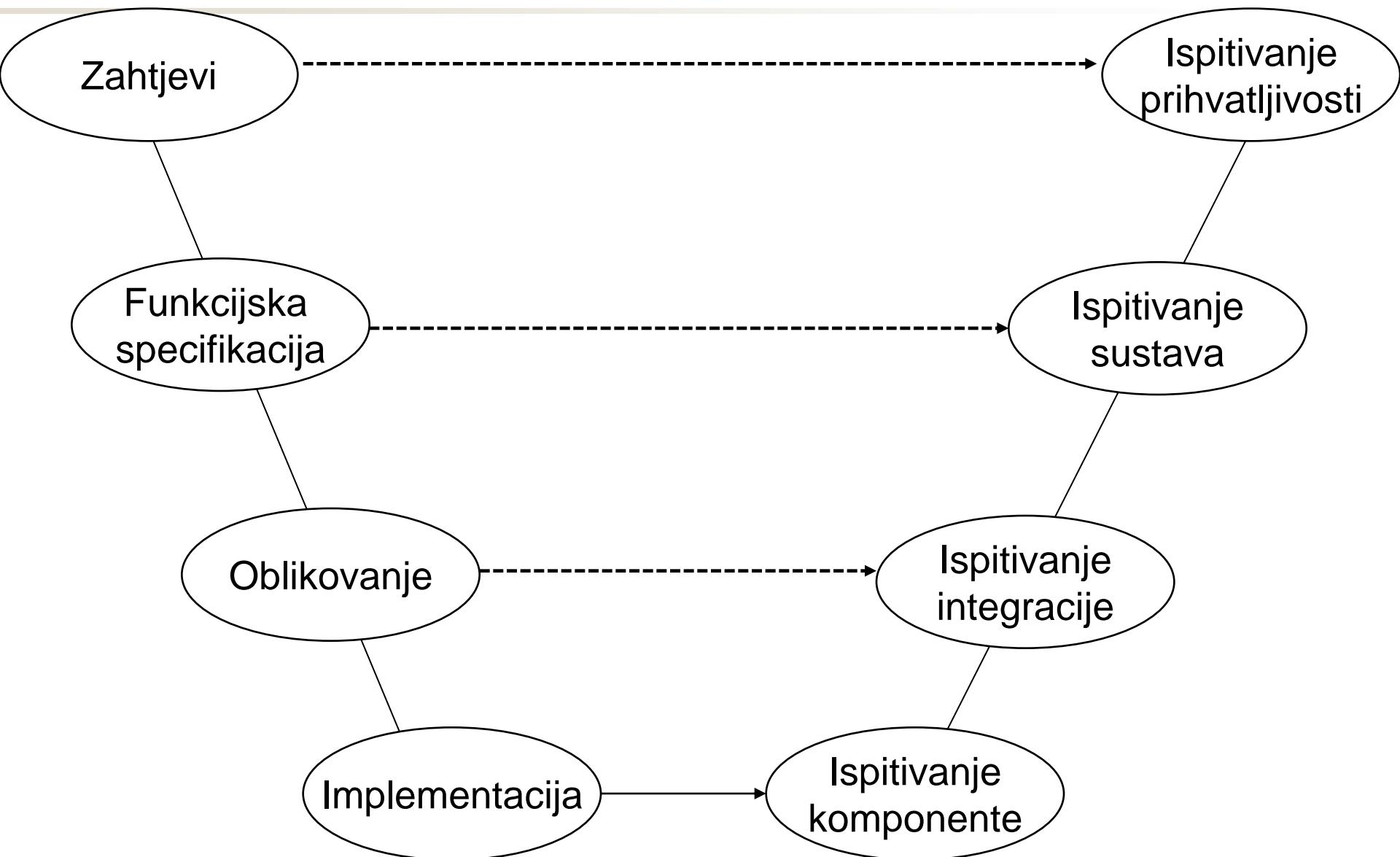
# Uloga ispitivanja

- Nije aktivnost koja započinje nakon završetka kodiranja s ograničenom svrhom otkrivanja pogrešaka
- Aktivnost objedinjena u procesu razvoja i održavanja i važan dio izgradnje produkta
- Započinje u ranim fazama analize zahtjeva
  - planovi ispitivanja i procedure moraju biti sistematično i kontinuirano razvijane, te prilagođavane sukladno razvoju
- Trošak ispitivanja predstavlja značajni udio cijene programskog produkta
  - bez obzira na primjenjenu metodologiju

# Organizacija ispitivanja

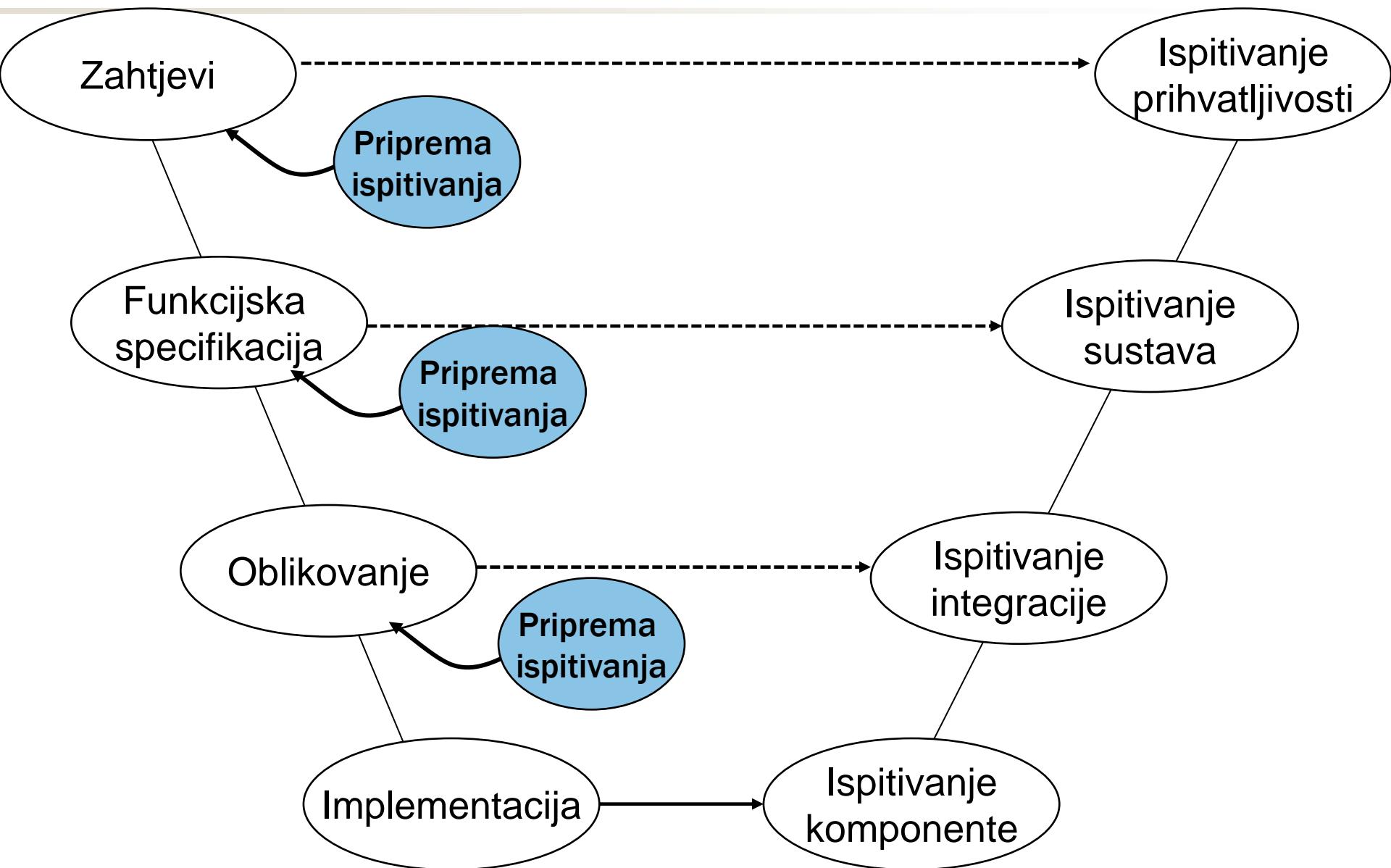
- Ovisi o svojstvima programske potpore
- Za složene programske sustave provodi se u više faza
- Različita za različite modele razvoja programske potpore
- Primjeri:
  - v-model ispitivanja
  - v-model ispitivanja s ranom pripremom
  - w-model ispitivanja

# V-model ispitivanja

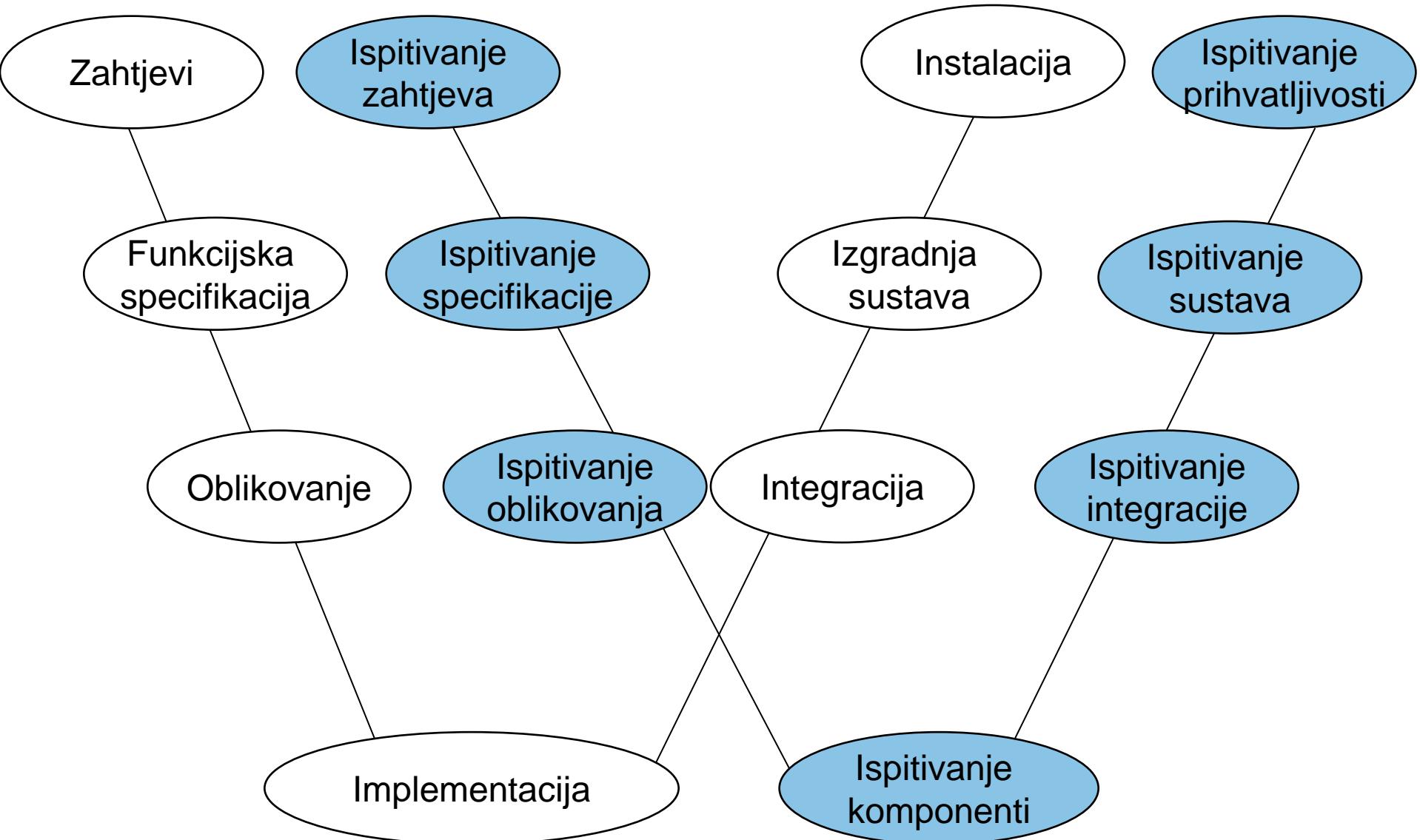


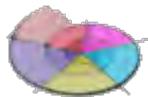


# V-model ispitivanja s ranom pripremom



# W-model ispitivanja





# Svojstva ispitljivih programa

- **engl. Software Testability**
- *Ispitljivost je mjeru mogućnosti jednostavnog ispitivanja programa*
  - omogućava primjenu bržeg i jednostavnijeg procesa ispitivanja
- **Osmotrivost (engl. observability)**
  - jednostavna identifikacija rezultata, različiti izlazi za različite ulaze,
  - jednostavno uočavanje neispravnih rezultata
- **Upravlјivost (engl. controllability)**
  - jednostavnost upravljanja tijekom provođenja ispitivanja
  - mogućnost pogodne specifikacije, automatizacije i ponovne uporabe ispitivanja
- **Dekompozicija (engl. decomposability)**
  - neovisno ispitivanje modula
- **Jednostavnost (engl. simplicity)**
  - odnosi se na složenost arhitekture, logike programa i kodiranja
- **Stabilnost (engl. stability)**
  - promjene programa tijekom ispitivanja utječu na rezultate provedenih ispitivanja
- **Razumljivost (engl. understandability)**
  - dobre informacije o strukturama, međuvisnostima i organizacija tehničke dokumentacije



# Uporaba informacija u ispitivanju



- Specifikacije
  - formalne
  - neformalne
  - za odabir, generiranje, provjeru (ispitnih slučajeva)
- Informacije oblikovanja
  - za odabir, generiranje, provjeru
- Programski kod
  - za odabir, generiranje, provjeru
- Uporaba
  - povijest, model
- Iskustvo ispitivanja

# Osnovni koraci ispitivanja

## 1. Odabir što ispitivati

- analiza: kompletност zahtjeva
- oblikovanje ispitivanja
- implementacija

## 2. Odluka kako ispitivati

- statička verifikacija
  - npr. nadzor izvornog koda *engl. Software or code inspection*
- odabir ispitivanja komponenti (*engl. Black-box, white box*)
- odabir integracijskog ispitivanja (*engl. big bang, bottom up, top down, sandwich*)

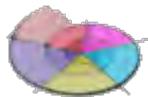
## 3. Razvoj ispitnih slučajeva

## 4. Predikcija rezultata

- za sve definirane ispitne slučajeve

# Kako ispitivati?

- Ispitivanje je spoznajna aktivnost
  - zahtjeva kreativnost
- Preduvjeti efikasnog ispitivanja
  - dubinsko poznavanje sustava
  - znanje tehnika ispitivanja
  - vještine primjene tehnika na efikasan način
- Najbolje koristiti neovisne timove za ispitivanje
  - programeri za ispitivanje **nesvjesno** koriste podatke na kojima program radi
  - povećan rizik sukoba s razvojnim timom
- Program najčešće ne radi kada ga upotrebljava netko drugi!!!
  - najgora situacija?
    - kada to otkriva korisnik

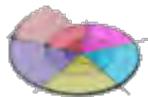


# Problemi zatajenja programa

- Ugrađena dijagnostika kao pomoć u analizi
  - umetanje provjera u kod programa
  - negativi efekti (struktura podataka, resursi, vrijeme,...)
- Zatajenje zbog efekta “nenamjernog sljepila”
  - ljudi: Ne vide ono što nije u centru pažnje
  - programi: Uvijek ne vide ono što im nije naglašeno da paze
    - Precizniji i manje prilagodljivi
- Neponovljiva zatajenja
  - razmatranje krivih uvjeta
    - nemoguće analizirati sve
- *Ispitni slučajevi u praksi ne mogu pokriti sve mogućnosti pojavljivanja pogrešaka* ☹

# Terminologija

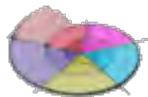
- Ispitni podaci (I)
  - ulazi odabrani za provođenje određenog ispitnog slučaja
- Očekivani izlaz (O)
  - zabilježen **prije** provođenja ispitivanja
- Ispitni slučaj
  - uređeni par (I, O)
  - kako ispitivati modul, funkciju, ...
  - sadrži opis stanja prije ispitivanja, funkcije koja se ispituje
- Stvarni izlaz
  - rezultat dobiven provođenjem ispitivanja
- Kriterij prolaza ispitnog slučaja
  - kriterij usporedbe očekivanog i stvarnog izlaza određen prije provođenja ispitnog slučaja



# Primjer ispitnog slučaja

- Ispitivanje funkcije “*get PIN*” za ukradene kartice
- Ulaz: 4 znamenkasti broj
- Očekivani rezultat: “Ukradena kartica”

R.br.	Ulaz	Očekivani rezultat	Stvarni rezultat	Status
1	Pročitaj karticu	Prihvati karticu i traži PIN, ako je kartica neispravna izbacи ju	Prihvaćena kartica i traži PIN	<i>U redu</i>
2	Pročitaj oštećenu karticu	Poruka “Neispravna kartica” i izbacivanje iz bankomata	Prihvaćena kartica i traži PIN	<i>Ne zadovoljava</i>
3	Unos nevaljanog PIN-a > 3	Poruka “ukradena kartica”, zadržana kartica	Prihvaćena kartica i traži PIN	<i>Ne zadovoljava</i>



# Primjer: Ispitni slučaj za OCSF



## General Setup for Test Cases in the 2000 Series

System: SimpleChat/OCSF      Phase: 2

### Instructions:

1. Install Java, minimum release 1.2.0, on Windows 95, 98 or ME.
2. Install Java, minimum release 1.2.0, on Windows NT or 2000.
3. Install Java, minimum release 1.2.0, on a Solaris system.
4. ~~Install the SimpleChat - Phase 2 on each of the above platforms.~~

## Test Case 2001

System: SimpleChat      Phase: 2

Server startup check with default arguments

Severity: 1

### Instructions:

1. At the console, enter: java EchoServer.

### Expected result:

1. The server reports that it is listening for clients by displaying the following message:  
Server listening for clients on port 5555
2. The server console waits for user input.

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

1. Hit CTRL+C to kill the server.

# Izvješće o incidentu

- engl. *Test Incident Report*
- IEEE 829 IEEE Standard for Software Test
- Dobar sustav praćenja mora omogućiti jednostavan pristup svim relevantnim informacijama
- Tko je odgovoran za prijavu incidenta?
  - SVI koji ga uoče!
  - što je s korisnicima?
- Po zatvaranju mora ostati dostupno

Test incident report identifier	
Unique name or number	
Associated test plan reference (if needed)	
Summary (include references to specifications and test log)	
Incident Description	
Inputs	Opis stvarno primijenjenog ulaza
Expected results	Očekivani rezultat (iz ispitnog slučaja)
Anomalies	razlike u odnosu na stvarno + relevantni podaci
Date and time	
Procedure step	korak u kojem je došlo do incidenta
Environment	okolina ispitivanja, npr. Korisnička ispitni poslužitelj ABC
Attempts to repeat	
Testers	
Observers	
Expected impact	



# Sažeto izvješće ispitivanja

- engl. *Test Summary Report*
- IEEE 829 IEEE Standard for Software Test
- Predstavlja sažetak tijeka procesa ispitivanja

Test summary report identifier Unique name or number	
Associated test plan reference (if needed)	
Summary Item tested (including version and environment)	
Summary	
Document references	
Variances Odstupanja od planiranog ispitivanja I stvarno provedenog	
Comprehensiveness assessment	
Summary of results Razriješeni problemi Nerazriješeni problemi Metrika ispitivanja – npr. pokrivenost ....	
Preporuke ....	
Evaluation Evaluacija pojedinog ispitivanja temeljem njihovih rezultata I ograničenja Stabilnost ispitivanja Modeliranje pouzdanosti	
Summary of activities Pregled aktivnosti ispitivanja: uključeno osoblje, utrošeni resursi ....	
Approvals	
Name	Job Title



# Oblikovanje ispitnog slučaja

- Oblikovanje ispitnog slučaja obuhvaća određivanje ulaza i odgovarajućeg rezultata kojeg upotrebljavamo za ispitivanje sustava
- Uobičajeno predstavlja skup ispitivanja
  - težimo efikasnom otkrivanju pogrešaka
- Cilj: otkrivanje pogrešaka
- Kriterij: kompletnost
- Ograničenje: minimum resursa i vremena
- Problem:
  - kako odrediti mesta ispitivanja?





- **Ispitivanje zasnovano na pokrivenosti** (engl. *Coverage-based testing*)
  - sve naredbe moraju biti izvršene barem jednom
  - zahtjevi ispitivanja su specificirani obzirom na pokrivenost ispitivanog programa
- **Ispitivanje zasnovano na pogreškama** (engl. *Fault-based testing*)
  - ispitni slučajevi koji omogućavaju otkrivanje pogrešaka
  - umjetno ubacivanje pogrešaka i otkrivanje u kojoj mjeri ih ispitivanje otkriva
- **Ispitivanje zasnovano na kvarovima** (engl. *Error-based testing*)
  - usmjereni na kvarove graničnih vrijednosti ili maksimalnog broja elemenata
  - ispitni slučajevi zasnovani na poznavanju tipičnih mesta izloženih kvarovima (posebice u krivoj uporabi).
- **Funkcijsko ispitivanje** (engl. *Black box, function, specification*)
  - ispitni slučajevi izgrađuju se temeljem specifikacije
- **Struktorno ispitivanje** (engl. *White box, structure, program based*)
  - ispitivanje uzima u obzir strukturu programa
- **Ispitivanje pod pritiskom** (engl. *Stress Based*)
  - naglasak na robusnost u kritičnim uvjetima rada

# Trajanje ispitivanja

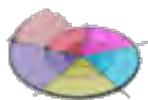
- Potpunost ispitivanja (engl. *Complete testing*)
- Problem određivanja trajanja procesa ispitivanja
- Značenje potpunog ispitivanja?
  - ispitivanje svake linije/grane/puta?
  - ne pronalaženje novih pogrešaka?
  - kraj plana ispitivanja?
- Po završetku postupka ispitivanja znamo da nema neotkrivenih pogrešaka.
  - ako postoje možemo ih pronaći novim ispitivanjem  $\Rightarrow$  Ispitivanje nije potpuno
- Pojednostavljenje problema uvođenjem pojma **potpunog pokrivanja** (engl. *complete coverage*).

# Potpuno ispitivanje

- Za ***potpuno ispitivanje*** neophodno provesti:
  - ispitivanje svih mogućih vrijednosti varijabli (ulazne/izlazne/među)
  - ispitivanje svih mogućih kombinacija ulaza
  - ispitivanje svih mogućih sekvenci izvođenja programa
  - ispitivanje svih mogućih HW/SW konfiguracija
  - ispitivanje svih mogućih načina uporabe programa
- Primjer:
  - MASPAC (Massively Parallel Computer, 64K paralelnih procesora primjena u NASI)
  - korijen 32-bitnog cijelog broja -  $\sqrt{4.294.967.296}$ 
    - Ispitivanje - 6 minuta, usporedba tablicama
    - pronađene 2 pogreške
  - koliko traje ispitivanje za slučaj 64-bitnog cijelog broja?
    - $18.446.744.073.709.551.616 - 49029$  godina

# Pokrivanje ispitivanja

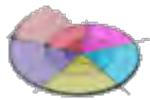
- engl. *Complete coverage*
- Stupanj gotovosti ispitivanja pojedinih atributa ili dijelova programa u odnosu na broj mogućih ispitnih slučajeva.
- Metrika:
  - najčešće pokrivenost koda ispitnim slučajevima
    - Postotak programskih elemenata koji su izvedeni
    - Pokrivenost linija koda
    - Pokrivenost grana
    - Pokrivenost putova
- Nedostatak: previše pojednostavljenja
  - problem prekida, paralelnosti ..
  - provjera zanimljivih podataka i kombinacija
  - nedostatak koda (engl. *black box*)
- Dobar alat za procjenu udaljenosti od potpunog ispitivanja
- Loše za procjenu gotovosti



# Problemi ispitivanja ulaznih varijabli



- Velik broj kombinacija
- Uređivani ulazi podataka (*engl. edited inputs*)
  - složeni unos podataka preko raznih uređaja
- Varijacije u vremenu unosa
  - brzo, sporo
  - prije, tijekom, nakon obrade događaja
- Obrada nevaljalih ulaza (nedopustivih ?)
- Nedokumentirani odziv - Uskršnja jaja (*engl. Easter Eggs*)
  - na neobične (neočekivane) ulaze – kako otkriti ?
- Čest problem ispitivanja:
  - razmišljanje: “niti jedan korisnik to ne bi napravio”
  - interpretacija: “niti jedan korisnik *kojeg mogu zamisliti i koji mi odgovara*, to ne bi *namjerno* napravio”



# Kombinacijsko ispitivanje

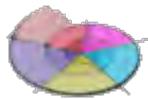
- engl. *Combination testing*
- Međutjecaj varijabli
- Za N varijabli ( $V_1, V_2 \dots V_N$ ) broj mogućih kombinacija
  - $V_1 \times V_2 \times \dots \times V_N$
- Primjer:
  - broj kombinacija dvije varijable definirane kao jednoznamenkasti dekadski broj
    - $10 \times 10 = 100$
  - koliki je broj mogućih kombinacija prva četiri poteza u šahu?
    - 318.979.564.000



# Primjer: Ispitivanje ulaznih varijabli



- Koliko kombinacija ulaznih vrijednosti trebamo ispitati u slučaju programa za zbrajanja dva dvoznamenkasta cijela broja unesena preko tipkovnice čiji se rezultat ispisuje na zaslon ekrana?
  - ulazne vrijednosti -99..99
  - 39.601 kombinacija ispravnih vrijednosti ulaza
  - što je s neispravnim vrijednostima ulaza?



# Ispitivanje – sažetak

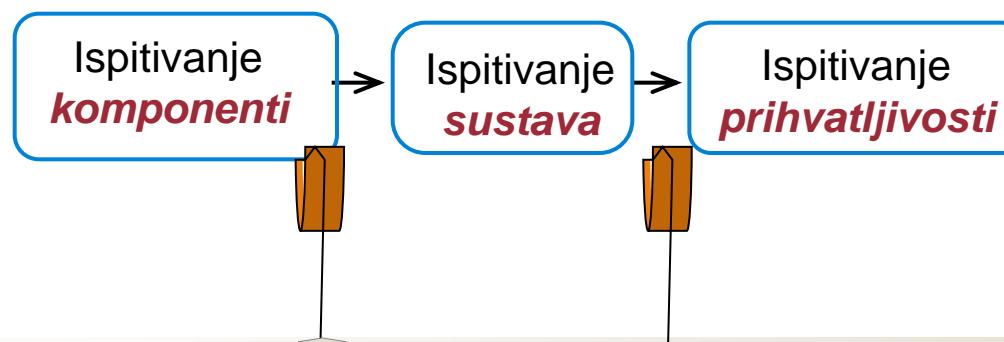
- Def.: Ispitivanje je proces izvođenja programa sa svrhom pronalaženja pogrešaka (dinamička provjera).
- IEEE standardi, dokument ispitivanja.
- Terminologija I: kvar (engl. *error*), pogreška (engl. *fault*), zatajenje (engl. *failure*).
- Terminologija II: ulazni podaci za ispitivanje, očekivani izlaz, ispitni slučaj (scenarij), stvarni izlaz, kriterij prolaza ispitivanja.
- Potrebno za ispitivanje:
  - specifikacija, informacije o oblikovanju, programski kod, uporaba programa, iskustvo.
- **Ispitivanja praktično ne mogu pokriti sve mogućnosti pogrešaka.**
- Potpuno ispitivanje (kriterij ?), potpuno pokrivanje.
- Kada ispitivati: V-model, V-model + priprema, W-model.
- Aktivnosti ispitivanja:
  - oblikovati, automatizirati, provoditi, evaluirati rezultate.
- Klasifikacija obzirom na odabir scenarija ispitivanja:
  - pokrivenost, pogreške, kvarovi.
- Klasifikacija obzirom na izvor informacija za scenarij ispitivanja:
  - crna kutija (kod nedostupan), bijela kutija (kod dostupan).

# ORGANIZACIJA ISPITIVANJA

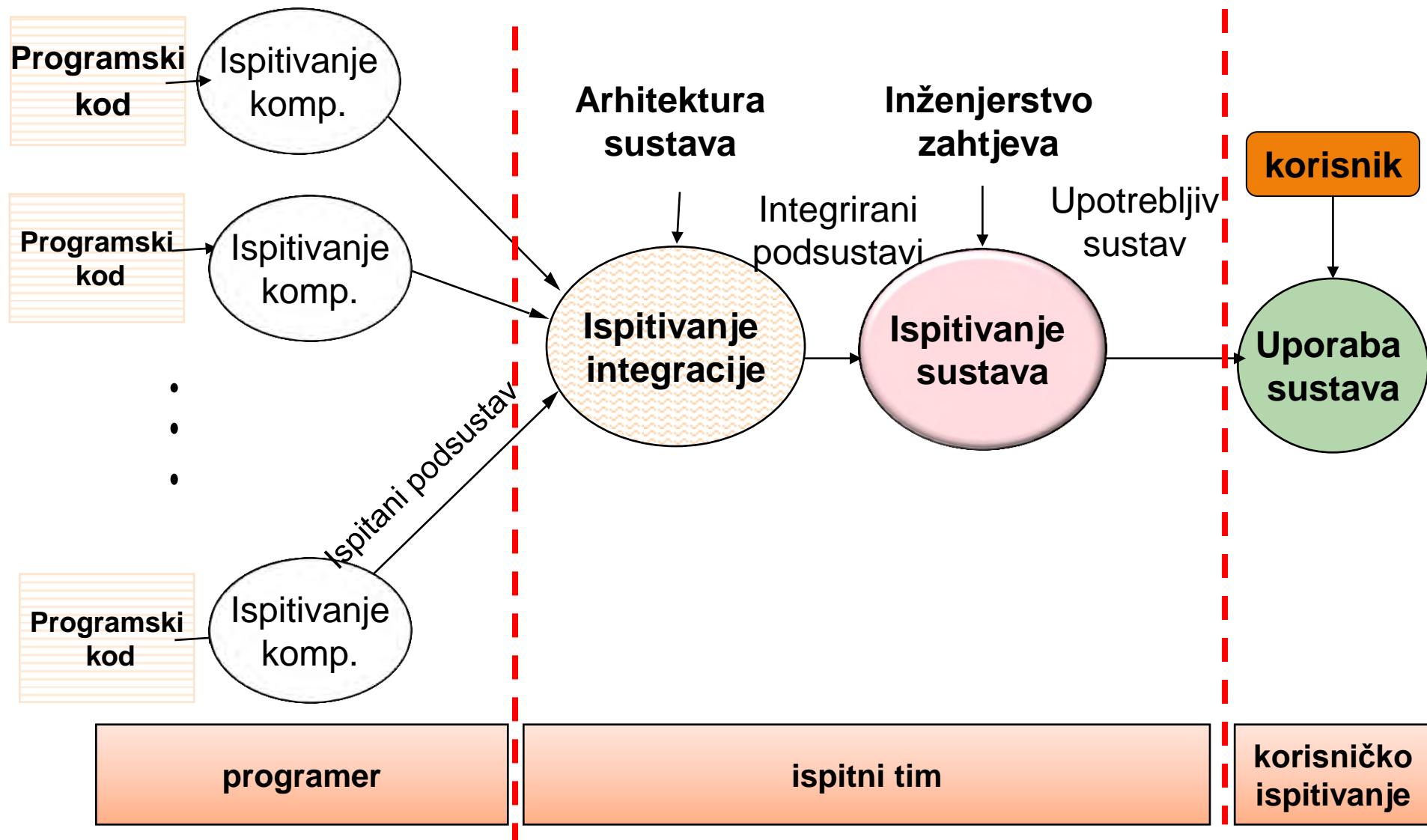


# Proces ispitivanja sustava

- Ispitivanje ***komponenti i modula***
  - individualne komponente se ispituju nezavisno.
  - komponente mogu biti funkcije, objekti ili koherentne skupine tih entiteta.
- Ispitivanje ***sustava***
  - ispitivanje cjelovitog sustava. Posebice je važno ispitivanje novih i nenadanih svojstava.
- Ispitivanje ***prihvatljivosti***
  - značajki na temelju kojih kupac prihvata i preuzima sustav (engl. *acceptance*).



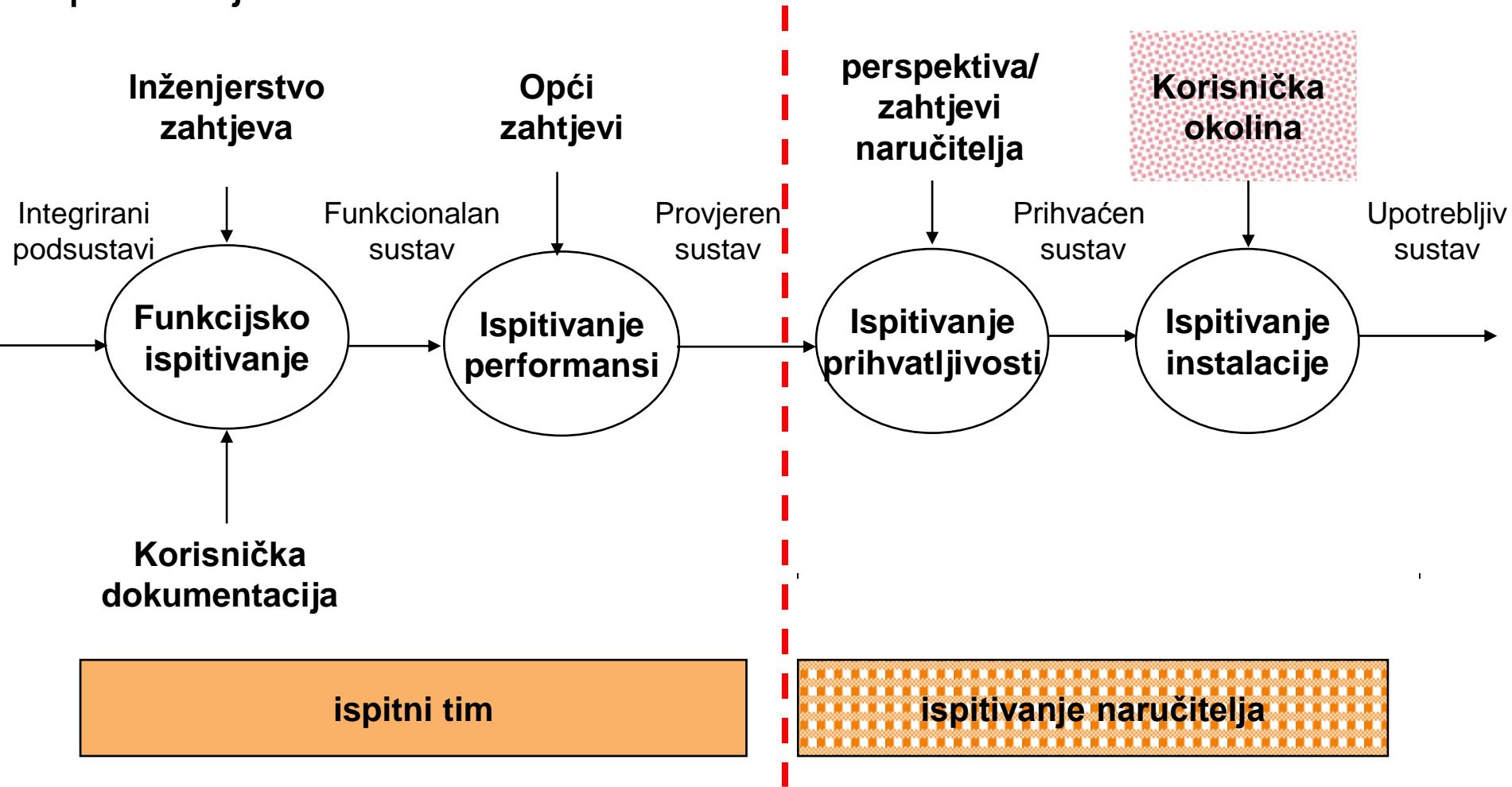
# Proces ispitivanja (1)





# Proces ispitivanja (2)

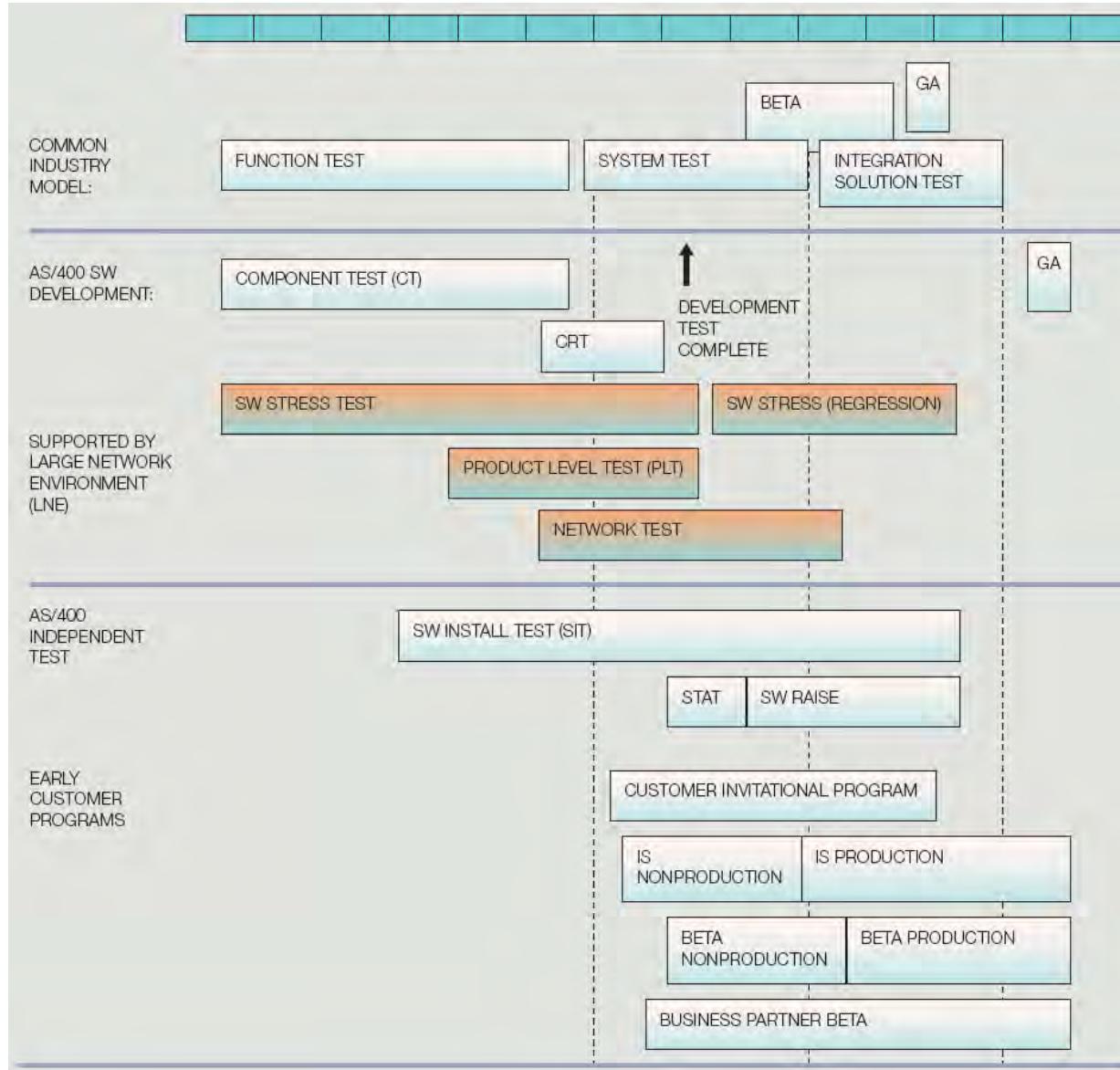
- Ispitivanje sustava verificira funkcione i nefunkcione zahtjeve te prihvatljivost sustava



# Organizacija ispitivanja

- Ispitivanje komponenti
  - engl. *Component/Unit testing*
- Integracijsko ispitivanje
  - engl. *Integration testing*
- Ispitivanje sustava
  - engl. *System/Release testing*
- Ispitivanje prihvatljivosti
  - engl. *Acceptance testing*
- Ispitivanje instalacije
  - engl. *Installation testing*
- Alfa ispitivanje
- Beta ispitivanje

# Primjer procesa ispitivanja



Izvor: S. H. Kan, J. Parrish D. Manlove: *In-process metrics for software testing AS/400 software testing cycle*

# ISPITIVANJE KOMPONENTI

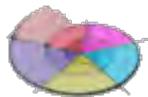
# Ispitivanje komponenti

- Ispitivanje koda na pogreške u algoritmima, podacima i sintaksi
- Oblikovanje ispitnih slučajeva
  - sučelje
  - strukture podataka
  - granični uvjeti
  - različiti tokovi izvođenja
  - obrada pogrešaka
  - ...
- Ispitivanje komponente provodi se u kontekstu specifikacije zahtjeva



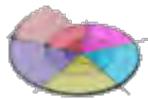
# Ispitivanje komponenti

- Verificira rad programskih dijelova koje je moguće neovisno zasebno ispitati:
  - pojedinačne funkcije ili metode unutar objekta
  - klase objekata s više atributa i metoda
  - složene komponente s definiranim sučeljem za pristup njihovim funkcijama
- Postoji pristup programskom kodu i upotrebljavaju se alati za ispravljanje pogrešaka
- Tko radi?
  - najčešće programer komponente
  - ispitivanja proizlaze iz iskustva



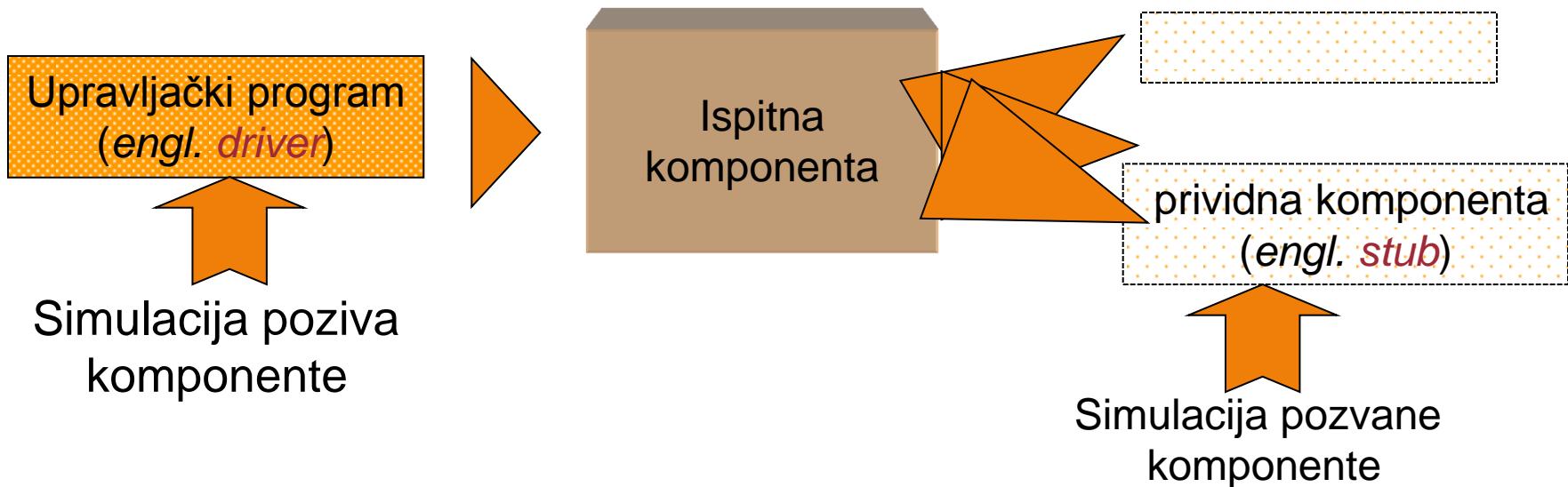
# Ispitivanje komponenti

- engl. *Unit test*
- Tijekom kodiranja
  - inkrementalno kodiranje
  - izuzetno značajno za skraćenje vremena ispitivanja
- Staticko ispitivanje
  - prolazno (neformalne prezentacije drugima)
  - nadzor koda (formalno predstavljanje drugima)
  - automatizirani alati za provjeru
    - sintaktička i semantička pogreške
    - odstupanje od standarda
- Dinamička ispitivanje:
  - funkcionalno ispitivanje, crna kutija
  - strukturno ispitivanje, crna kutija



# Okolina ispitivanja komponenti

- Postupak izolacije komponente u svrhu ispitivanja
- Upravljački program (*engl. driver*) – kod koji upravlja procesom izvođenja jedne ili više komponenata
  - uobičajeno upotrebljava postojeće sučelje komponente, a može zahtijevati i stvaranje novih
- Prividna (krnja) komponenta (*engl. stub*) – kod koji simulira pozivanu komponentu
  - identično sučelje stvarnoj komponenti međutim





# Elementi ispitivanja komponenti

- Sučelje
  - osigurava ispravno prihvaćanje i pružanje informacija
- Podaci struktura:
  - osigura integritet podataka
- Rubni uvjeti
  - provjera rada u graničnim slučajevima
- Nezavisni putovi
  - sve putove kroz kontrolne strukture
- Iznimke
  - provjera ispravne obrade iznimaka



# Ispitivanje komponenti OO sustava

- Što je komponenta u OO sustavu?
  - za potrebe ispitivanja najčešće se uzima razred/klasa
- Ispitivanje razreda obuhvaća
  - ispitivanje svih operacija (Kako tretirati hijerarhiju?)
  - postavljanje i ispitivanje svih atributa objekta
  - ispitivanje objekta u svim stanjima (simulacija događaja koji mijenjaju stanje)
- Ispitivanje grupa razreda predstavlja oblik integracijskog ispitivanja
- Poteškoće ispitivanja razreda uzrokuju osnovni koncepti OO (enkapsulacija, nasljeđivanje, polimorfizam, ...)
  - npr. Informacije koje treba ispitati nisu lokalizirane



# Koraci ispitivanja OO sustava

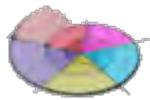
1. Ispitni slučaj jedinstveno označiti i eksplicitno povezati s ispitivanim **razredom**
2. Definirati namjenu ispitnog slučaja
3. Razraditi korake ispitivanja:
  1. definirati **stanja objekata** koja se ispituje
  2. definirati **poruke i operacija** koje se ispituju te njihove posljedice
  3. definirati listu **iznimaka** koje mogu proizaći tijekom ispitivanja
  4. definirati stanje **okoline** pri ispitivanju

# Slučajno ispitivanje

- engl. *Random testing*
- Koraci
- Identificirati operacije primjenjive na razred
- Definirati ograničenja na njihovo korištenje
- Identificirati minimalni ispitni slučaj
  - slijed operacija koji definira minimalnu životni vijek instanciranog objekta
- Generirati niz ispravnih slučajnih ispitnih sekvenci

# Ispitivanje particija

- engl. *Partition testing*
- Smanjuje broj ispitnih slučajeva potrebnih za ispitivanje razreda
  - princip ekvivalencije particija
  - particija stanja
    - engl. *state-based partitioning*
  - kategorizirati i ispitati operacije temeljem utjecaja na promjenu stanja objekta
  - particije atributa
    - engl. *attribute-based partitioning*
  - kategorizirati i ispitati operacije temeljem svojstava atributa
- Particije kategorija
  - engl. *category-based partitioning*
  - dekompozicija funkcijске specifikacije i utvrđivanje generičkih karakteristika
  - kategorizirati i ispitati operacije na temelju generičkih funkcija koje obavljaju

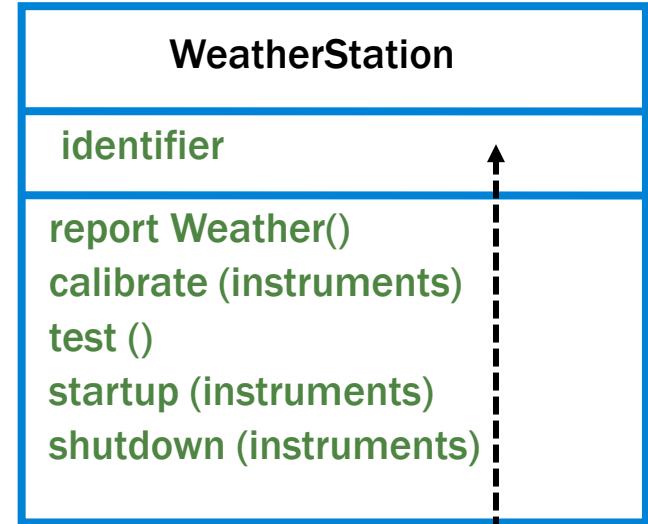


# OO ponašajno ispitivanje

- engl. *OO behavioral testing*
- Ispitni slučajevi moraju pokriti sva stanja objekta
- Pogodna uporaba UML dijagrama stanja
- Velike mogućnosti automatizacije

# Primjer

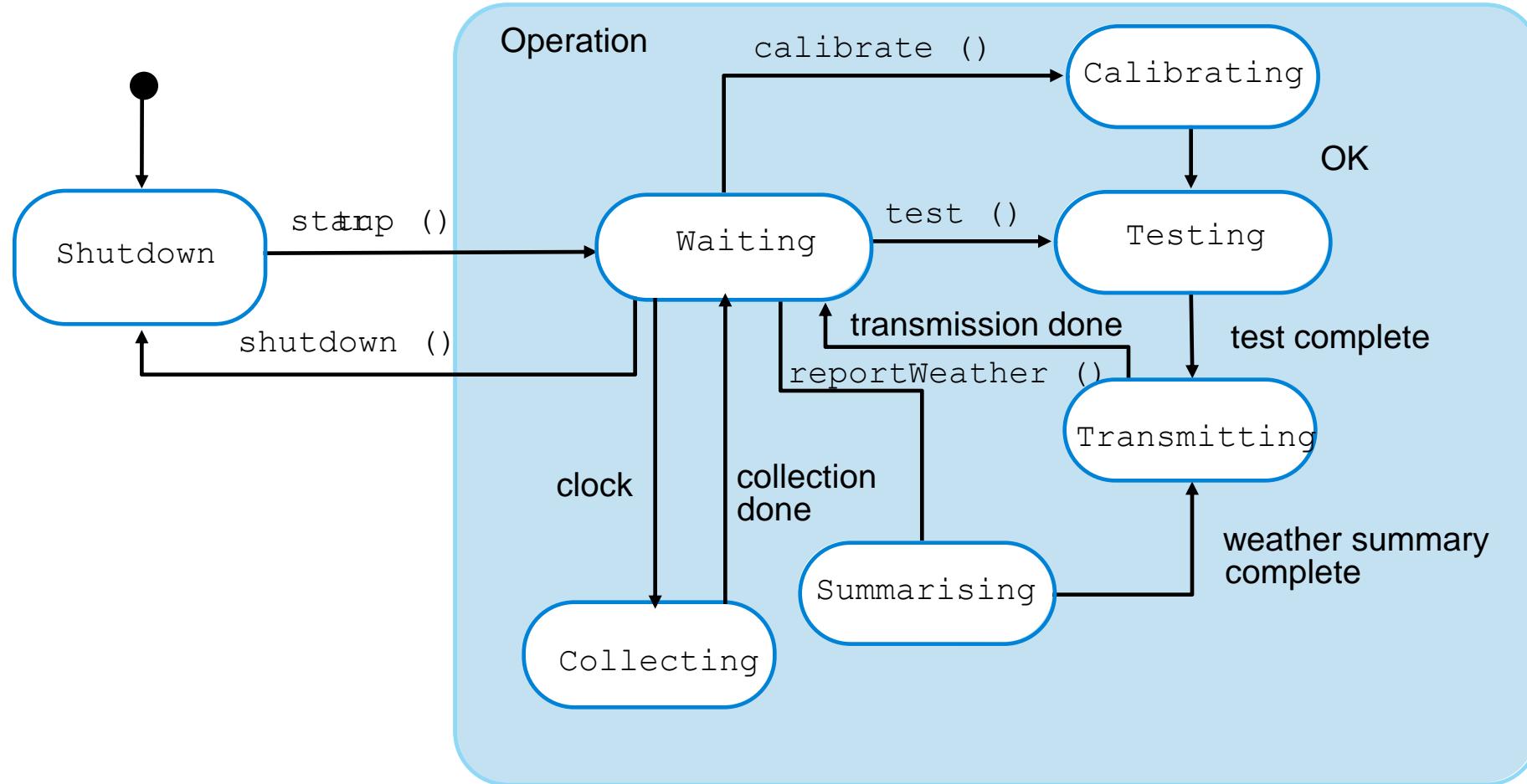
- Razred ***Weather station***
- Definirati ispitne slučajeve za:
  - ***reportWeather, calibrate, test, startup, shutdown.***
- Uporabom dijagrama stanja pronaći:
  - sekvence prijelaza za ispitivanje i
  - odgovarajuće sekvence događaja koje ih uzrokuju prema dijagramu stanja



ispitivanje atributa  
***identifier***

•npr. ***provjeriti da li je postavljen***

# Primjer: Dijagram stanja Weather station



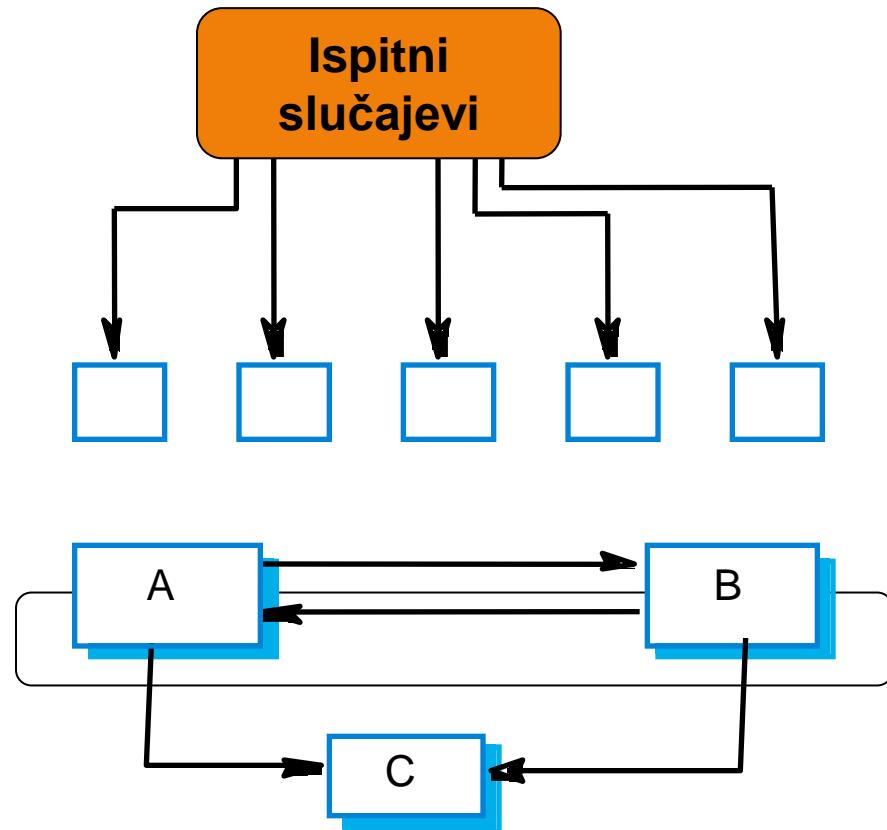
- Waiting → Calibrating → Testing → Transmitting → Waiting
- Shutdown → Waiting → Shutdown
- Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

Izvor: Sommerville, I., Software engineering

# Ispitivanje sučelja

- Cilj je otkriti pogreške uzrokovane kvarovima sučelja ili pogrešnim prepostavkama o sučelju
- Posebno značajno za OO sustave
  - objekti (i komponente) su definirani sučeljima
  - pogreške nastaju kao posljedica interakcija

- Složena komponenta



(ispitni slučaj za cijeli podsustav  
odnosno složenu komponentu)

Izvor: Sommerville, I., Software engineering



# Tipovi sučelja programskih komponenata



- **Parametarsko sučelje** (engl. *Parameter interfaces*)
  - podaci i funkcije se prenose pozivima procedure
- **Dijeljena memorija** (engl. *Shared memory interfaces*)
  - procedure dijele zajednički memorijski prostor
- **Proceduralno sučelje** (engl. *Procedural interfaces*)
  - komponente obuhvaćaju skup procedura koje pozivaju ostali podsustavi (npr. objekti)
- **Sučelje zasnovano na porukama** (engl. *Message passing interfaces*)
  - podsustavi traže usluge od ostalih podsustava slanjem poruke (npr. klijent-uslužitelj)

# Kvarovi sučelja

- **Pogrešna uporaba** (engl. *Interface misuse*)
  - komponenta koja poziva drugu pogrešno upotrebljava njezino sučelje
    - Npr. Pogrešan redoslijed parametara
- **Nerazumijevanje sučelja** (engl. *Interface misunderstanding*)
  - komponenta koja poziva drugu pogrešno prepostavlja specifikaciju njezinog ponašanja (npr. binarno pretraživanje na neuređenom nizu)
- **Vremenske pogreške** (engl. *Timing errors*)
  - pozvana i pozivajuća komponenta rade **različitom brzinom** (npr. zajednička memorija te različita brzina čitanja i pisanja)
- Naputak za ispitivanje sučelja:
  - oblikuj ispitne slučajeve tako da parametri poprime **ekstremne vrijednosti**
  - uvijek ispitaj pokazivače (ako se šalju sučeljem) s **null vrijednostima**
  - oblikuj ispitni slučaj proceduralnog sučelja tako da **zataji komponentu**
  - sustave s razmjenom poruka ispitaj na **stres** (generiraj više pouka nego što je normalno potrebno)
  - sustave s dijeljenom memorijom ispitaj s **različitom redoslijedom** aktiviranja komponenti

# Oblikovanje programske potpore

ak.god. 2014./2015.

## *Ispitivanje programske potpore*



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



# Tema

- Definicija i ciljevi ispitivanja programske podrške
- Klasifikacija ispitivanja
- Upoznavanje procesa ispitivanja
- Strategije ispitivanja
- Principi ispitivanja sustava i komponenti
- Funkcionalno i strukturno ispitivanje
- Generiranje ispitnih slučajeva
- Automatizacija procesa ispitivanja
  
- Cilj:
  - upoznavanje tehnika ispitivanja kao jedne od mogućih tehniku verifikacije programske podrške
  - razumijevanje terminologije, procesa i različitosti tehnika ispitivanja



# Literatura



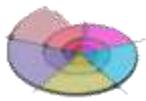
- Sommerville, I., ***Software engineering***, 8th ed, Addison Wesley, 2007.
- Glen Myers, ***The art of software testing***, Second ed., John Wiley & Sons, New Jersey, 2004.
- S. Siegel, ***Object-Oriented Software Testing: A hierarchical Approach***, John Wiley & Sons, New Jersey, 1996.

Pripremio: Vlado Sruk

Ovaj dokument namijenjen je isključivo za osobnu upotrebu studentima Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu.

U pripremi materijala osim literature upotrijebljeni su i drugi izvori, te zahvalujem autorima.

# INTEGRACIJSKO ISPITIVANJE



# Integracijsko ispitivanje



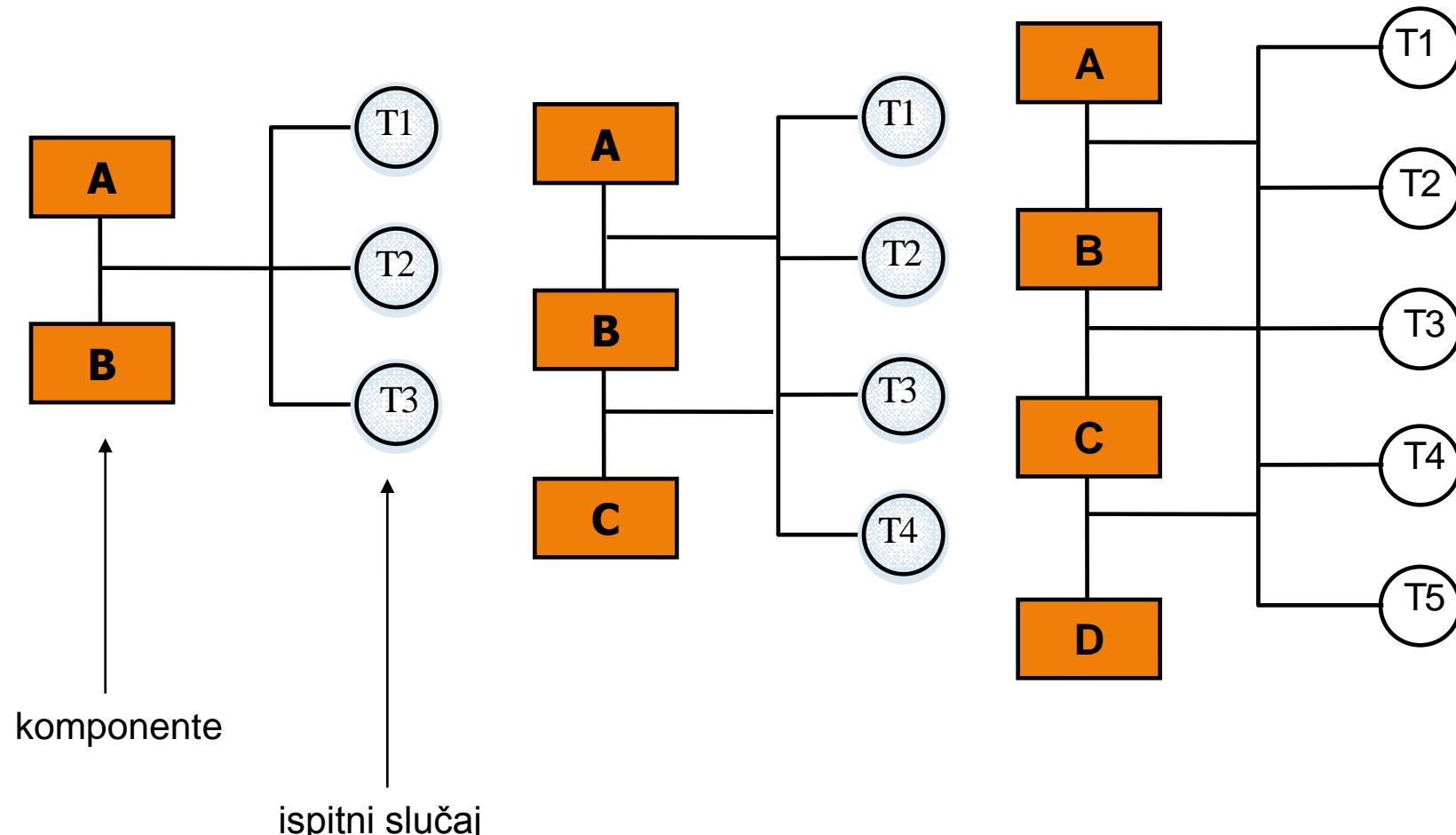
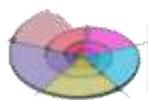
- Proces verifikacije interakcije programskih komponenti
  - s dodatnim kodom za zajednički rad
- Sistem se ispita tijekom integracije komponenti s ciljem uočavanja problema integracije
- Cilj je osigurati zajednički rad grupe komponenti prema specifikaciji dokumentaciji zahtjeva
- Osnovni problem predstavlja lokalizacija pogrešaka
  - zbog složenih interakcija



# Pristupi integracijskom ispitivanju



- ***Veliki prasak*** (engl. *Bing bang*)
  - integrirati sve komponente bez prethodnog ispitivanja
  - u slučaju pogrešaka problem predstavlja otkrivanje mesta pogreške
- ***Poboljšani veliki prasak*** (engl. *Enhanced bing bang*)
  - integracija svih komponenti nakon provedenog ispitivanja
  - u slučaju pogrešaka i dalje prisutan problem otkrivanja mesta
- ***Inkrementalni pristup***
  - integracija i ispitivanje sustava dio po dio
  - uobičajen pristup u praksi
  - efikasan u lokalizaciji mesta pogreške



## ■ Kako pristupiti ispitivanju?

Izvor: Sommerville, I.: Software engineering

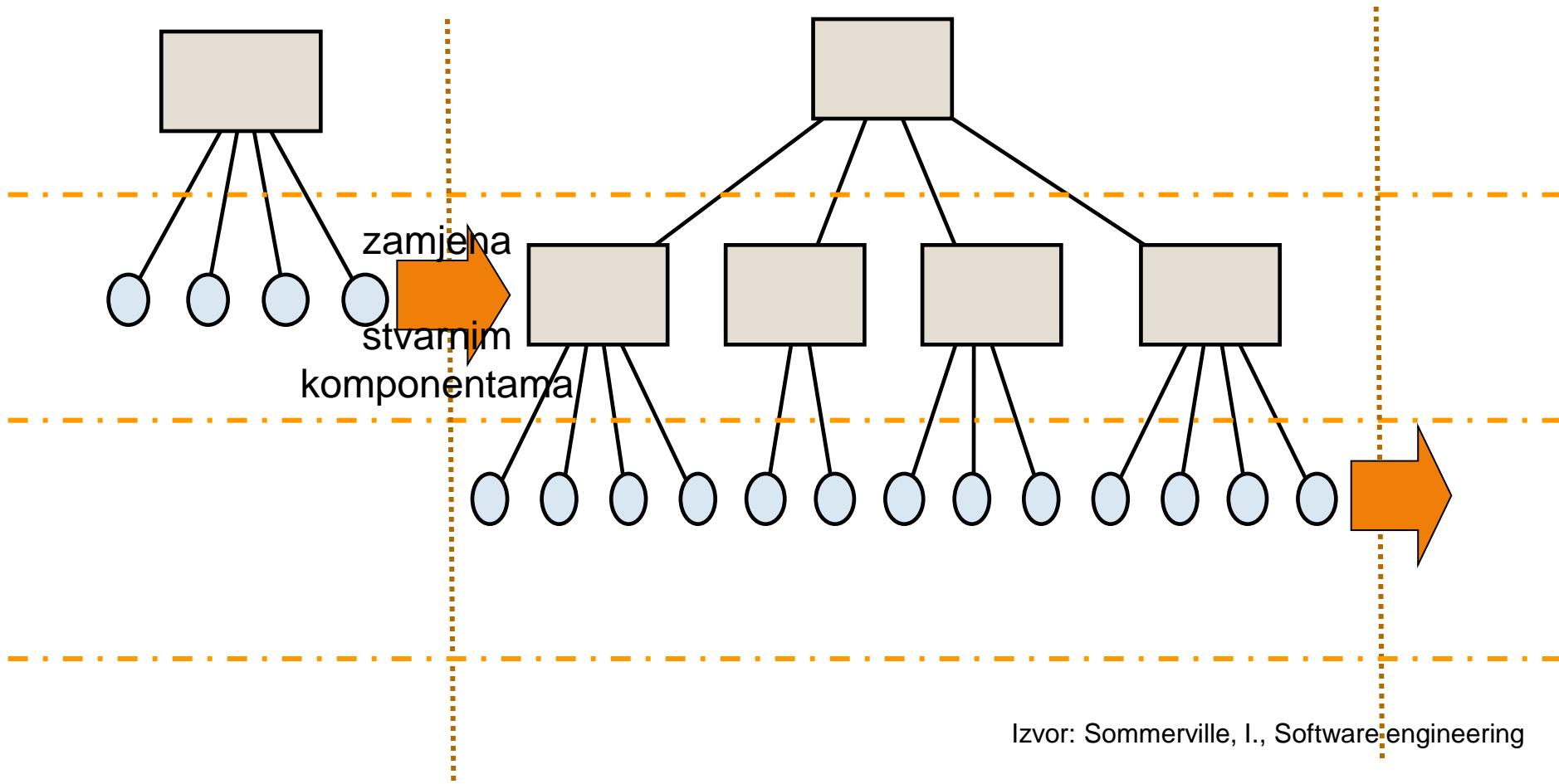


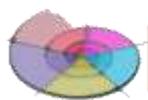
# Inkrementalno integracijsko ispitivanje



## ■ Odozgo na dolje (engl. *Top down integration*)

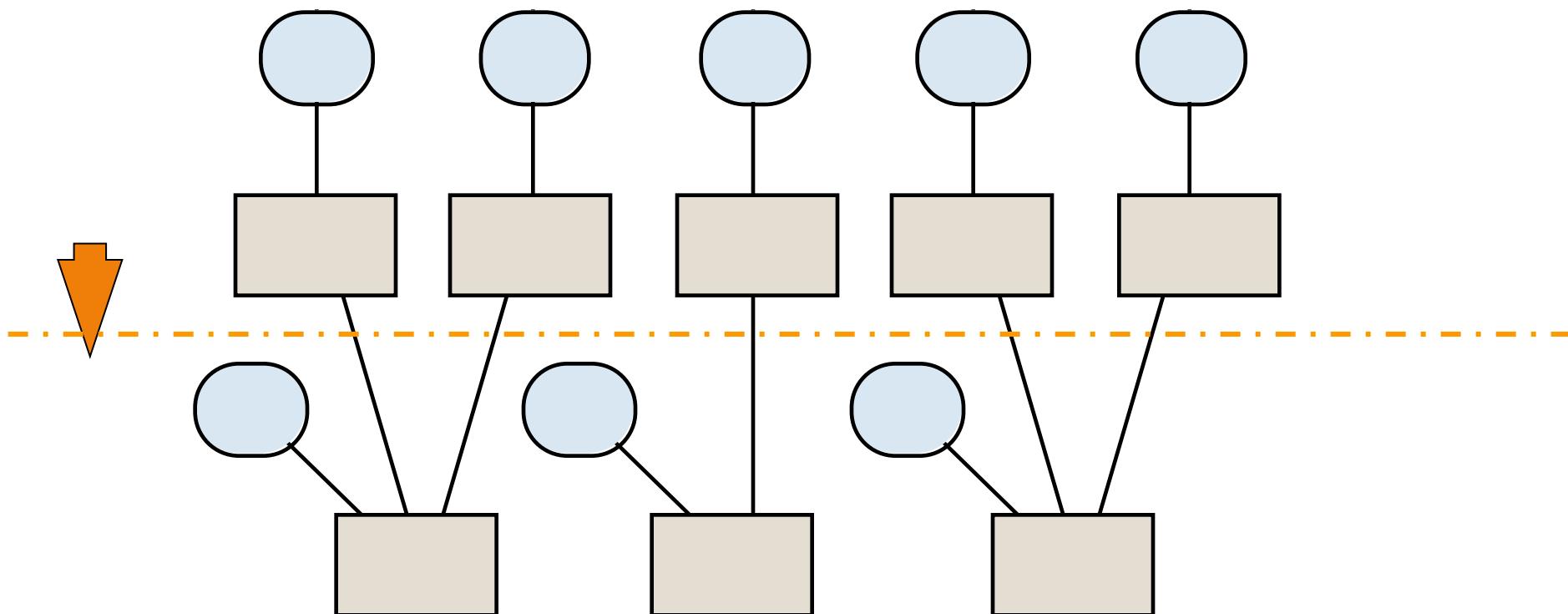
- Razviti kostur sustava i popuniti ga komponentama
- ne treba razvijati upravljačke programe





## ■ Odozdo na gore (engl. *Bottom up integration*)

- prvo integrirati neke komponente (najvažnije i najčešće funkcionalnosti), te dodati preostale
- ne treba razvijati prividne komponente



Izvor: Sommerville, I., Software engineering

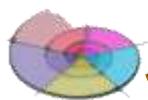
- Funkcijska integracija (*engl. Sandwich testing*)
  - integriranje komponenti u konzistentne funkcije bez obzira na hijerarhijsku strukturu
  - kombinacija odozgo-prema-dolje i odozdo-prema-gore
  - najčešći postupak u praksi



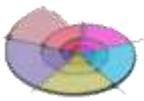
# Integracijsko ispitivanje u OO



- engl. *Inter-class testing*
- U objektno orijentiranim sustavima pogodno je iskoristiti prepoznata grupiranja razreda
  - međuvisne klase su dobri kandidati za početak inkrementalne integracije (engl. *tightly coupled*).
- Kandidati:
  - razredi grupirane u pakete
  - razredi u hijerarhijskoj ovisnosti
  - razredi povezani dijagramima interakcija pridruženim obrascima uporabe
- Najčešći pristupi
  - pristup odozdo prema gore
  - primjena obrazaca uporabe



- engl. *Inter-class testing*
- Integracijsko ispitivanje razreda može se grupirati prema razinama apstrakcije
  - ispitivanje zasnovano na dretvama
    - engl. *thread-based testing*
    - integrira skup razreda obzirom na poticaje ulaza ili događaja
  - ispitivanje zasnovano na uporabi
    - engl. *use-based testing*
    - integrira skup razreda koje zahtijeva obrazac uporabe
  - ispitivanje zasnovano na grupiranju
    - engl. *cluster testing*
    - integrira skup razreda koje su neophodni za ostvarenje suradnje razreda



# Tijek ispitivanja



1. Uobičajeni postupak je da se za svaki podrazred, koristiti popis operatora za generiranje niza slučajnih ispitnih sekvenci.
  - operatori šalju poruke drugim razredima
2. Za svaku poruku koja se generira, određuje se pozvani (suradnički) razred i odgovarajući operator u poslužitelja objekta.
3. Za svaki operator u pozvanom objektu razreda utvrditi nove poruke koje odašilje.
4. Za svaku od poruka, odrediti sljedeću razinu koja se poziva
  - izraditi odgovarajuće ispitne slučajeve



# Pristupi integracijskom ispitivanju OO

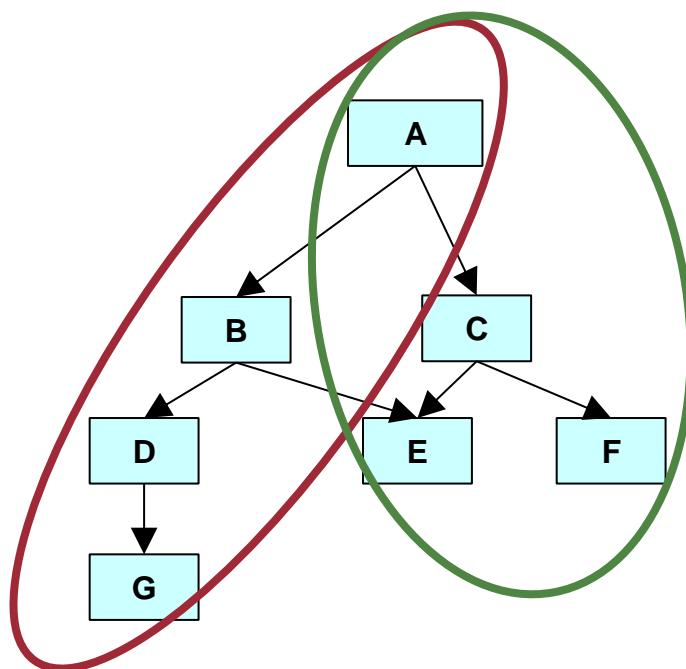


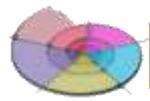
## ■ Pristup odozgo prema gore

- najčešća tehnika
- integracija razreda započevši od krajnjih podrazreda prema hijerarhiji ovisnosti

## ■ Primjena obrazaca uporabe

- obrasci opisuju interakciju razreda
  1. u hijerarhiji razreda identificirati suradnju
  2. odabratи kriterij sekvenci ispitivanja
  3. oblikovati ispitivanje

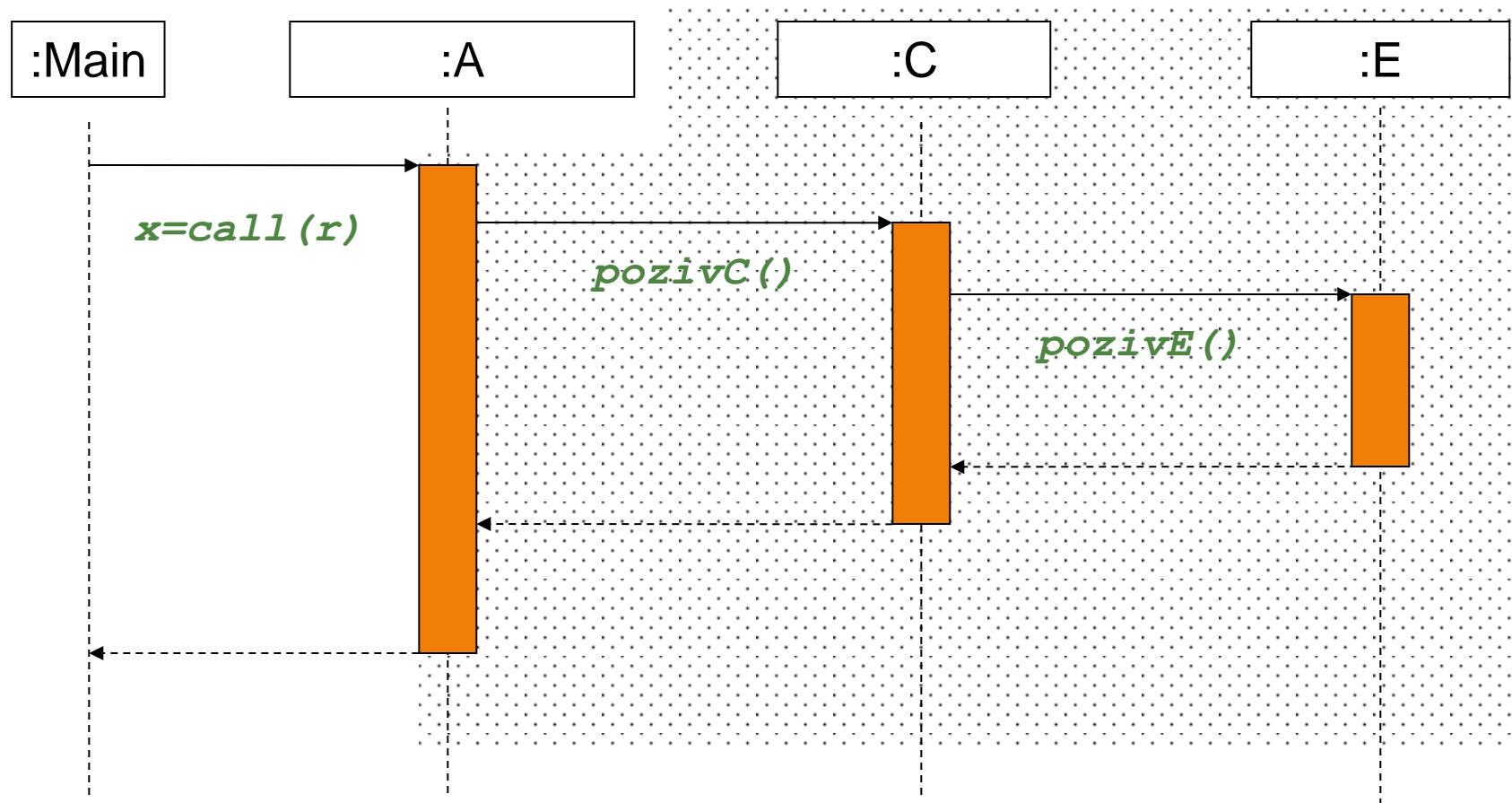




# Primjer: Uporaba dijagrama interakcija

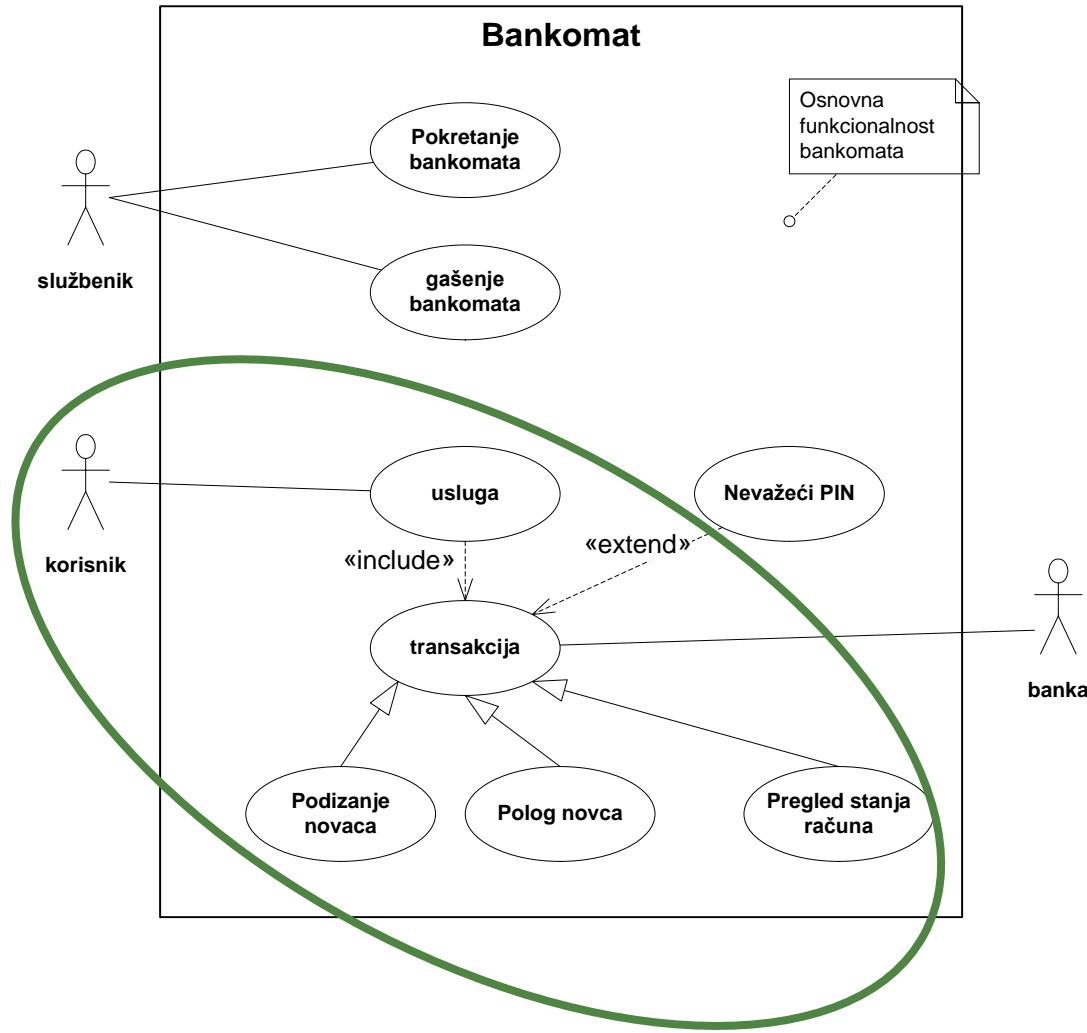


- Ispitivanje metode `call` s ulazom `r` i očekivanim rezultatom `x`



# Primjer: Obrasci uporabe

## ■ Bankomat:



# Prednosti i nedostaci

## Ispitivanje odozdo prema gore

- Omogućuje rano ispitivanje
- Moduli mogu biti integrirana u različitim grupama
- Glavni naglasak je na funkcionalnosti i performansama modula
- Nije potrebno izgrađivati prividne module
- Lagana organizacija provođenja
- Pogreške u kritičnim modula nalaze rano
  
- Nedostaci
  - potrebna izgradnja upravljačkih programa
  - dugotrajan proces u slučaju složenih hijerarhija
  - kasno otkrivanje pogreške u sučeljima (kritično)

## Ispitivanje odozgo prema dolje

- Upravljački (glavni) program je ispitani prvi
- Moduli su integrirani istodobno
- Glavni naglasak je postavljen na ispitivanje sučelja
- Nije potrebno izgrađivati upravljačke programe
- brzo se dobiva radni prototip
  - rano otkrivanje pogrešaka u sučeljima (kritično)
- Sučelje pogreške otkrio rano
- Modularnost pospješuje ispravljanje pogrešaka
- Nedostaci
  - potrebno je izgrađivati prividne module
  - sporo uključivanje većeg tima
  - pogreške u kritičnim modula na niskim razinama nalaze se kasno

# ISPITIVANJE SUSTAVA

# Ispitivanje sustava

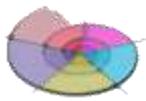
- engl. *System/Release testing*
- Proces ispitivanja inačice namijenjene distribuciji korisniku
- Osnovni cilj provjera podudarnosti sustava s **funkcijskim** zahtjevima
- Uobičajeno okruženje funkcionskog ispitivanja
  - ispitivači nemaju znanje o detaljima implementacije
  - veća vjerojatnost pronalaženja kvarova koje nisu uočili programeri
  - neovisni ispitivači uklanjaju sukob interesa razvoja i ispitivanja
- Dvije faze
  - integracijsko ispitivanje – ispitivači imaju pristup izvornom kodu
  - ispitivanje gotovog sustava - ispitivači vide sustav kao crnu kutiju



# Ispitivanje performansi



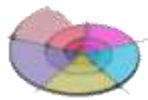
- Prije isporuke potrebno je ispitati **svojstva sustava kao cjeline u radu** (engl. *emergent properties* – svojstvo sustava koje proizlazi od interakcije sastavnih dijelova)
- Različite namjene:
  - standardno ispitivanje performansi (engl. *benchmarks*)
  - pokazati zadovoljenje performansi pod različitim radnim opterećenjima
  - pokazati proširivost sustava
  - ispitivanje se planira s povećanjem opterećenja sve dok performanse ne postanu nezadovoljavajuće
- **Ispitivanje pod pritiskom** (engl. *Stress testing*)
  - namjena određivanje stabilnosti sustava
    - Robusnost, raspoloživost i obrada pogrešaka na graničnim vrijednostima opterećenja npr. broja podataka, učestalost zahtjeva, veličina i sl.
    - npr. transakcija u sekundi, u raspodijeljenim sustavima broj korisnika, DoS napadi,...
  - istjerivanje pogrešaka



# Ispitivanje prihvatljivosti



- engl. *Acceptance testing*
- Provjerava ponašanje sustava u odnosu na zahtjeve naručitelja (verifikacija je značajnija)
- Najčešće se provodi **zajednički s timom naručitelja**
- Provodi se kao funkcionalno ispitivanje (ispitivanje crne kutije)
- Uobičajeno se provodi prije isporuke programa
  - moguće i nakon isporuke prema dogovoru s kupcem
- Cilj je pokazati da sustav zadovoljava zahtjeve i spreman je za uporabu



# Instalacijsko, α i β ispitivanje



## ■ *Instalacijsko ispitivanje*

- provodi se nakon ispitivanja prihvatljivosti na instalaciji u radnoj okolini
- identično ispitivanju sustava prema zahtjevima sklopovske konfiguracije

## ■ *Alfa / Beta ispitivanje*

- prije završne isporuke, program pokusno upotrebljava ciljana skupina korisnika
  - Alfa – unutar tvrtke, konzorcija – interno uz prisustvo razvojnog tima
  - Beta – vanjski korisnici

# Strategije ispitivanja

## ■ **Iscrpno ispitivanje** (engl. *Exhaustive testing*)

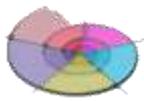
- ispitivanje svih mogućih scenarija
- moguće samo za ograničene primjere

## ■ **Slučajno ispitivanje** (engl. *Random testing*)

- nije isto kao i "ad hoc" ispitivanje
- odabir ispitnih slučajeva temeljem vjerojatnosti
  - Uniformna razdioba
  - Razdioba temeljena na prethodno prikupljenim podacima

## ■ **Sistematsko ispitivanje**

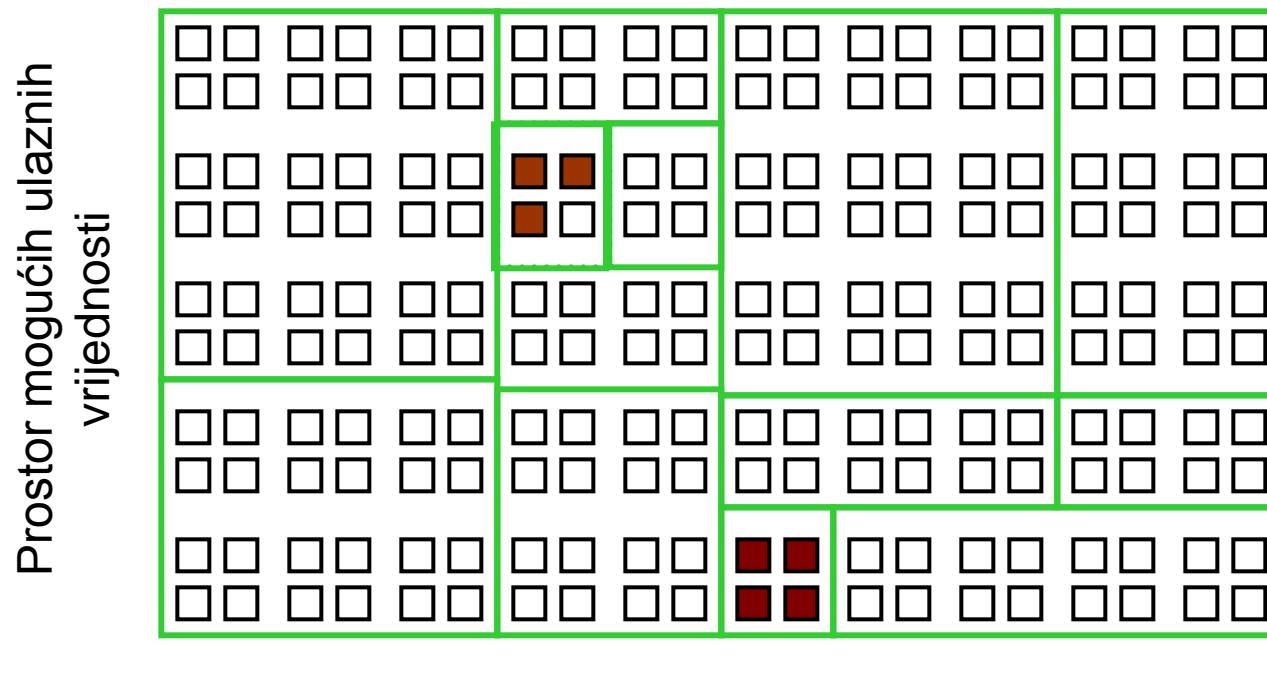
- podjela ulaznih podataka u poddomene (particije) i odabir ispitnih slučajeva temeljem različitih svojstava
  - svojstva koda, specifikacija, rizik, ...

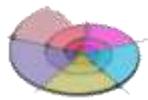


# Sistematsko ispitivanje particija



- Uobičajeno 30-85 pogrešaka u 1000 LOC
- Dobro ispitani programi 0.5 – 3 pogreške
- Pogreške su često grupirane
- Cilj je izolirati područja s vjerojatnom pojавom pogreške

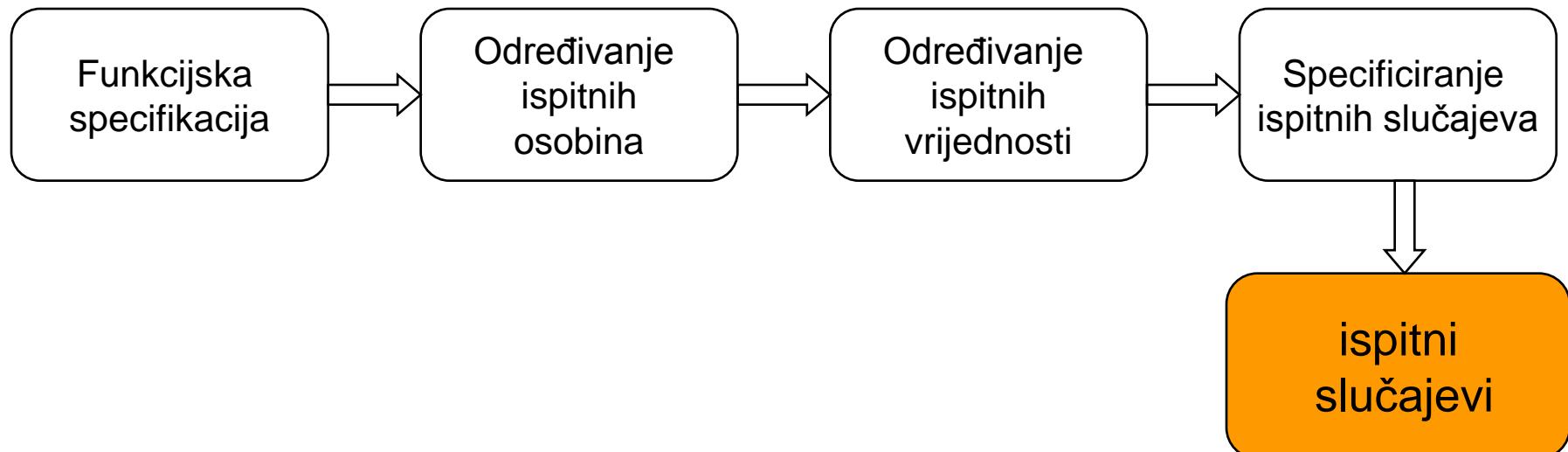




# Sistematsko ispitivanje

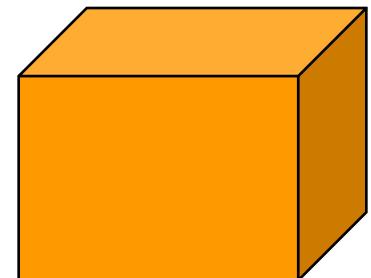


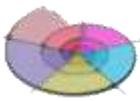
## ■ Osnovni koraci:



Tehnike ispitivanja

# FUNKCIJSKO ISPITIVANJE



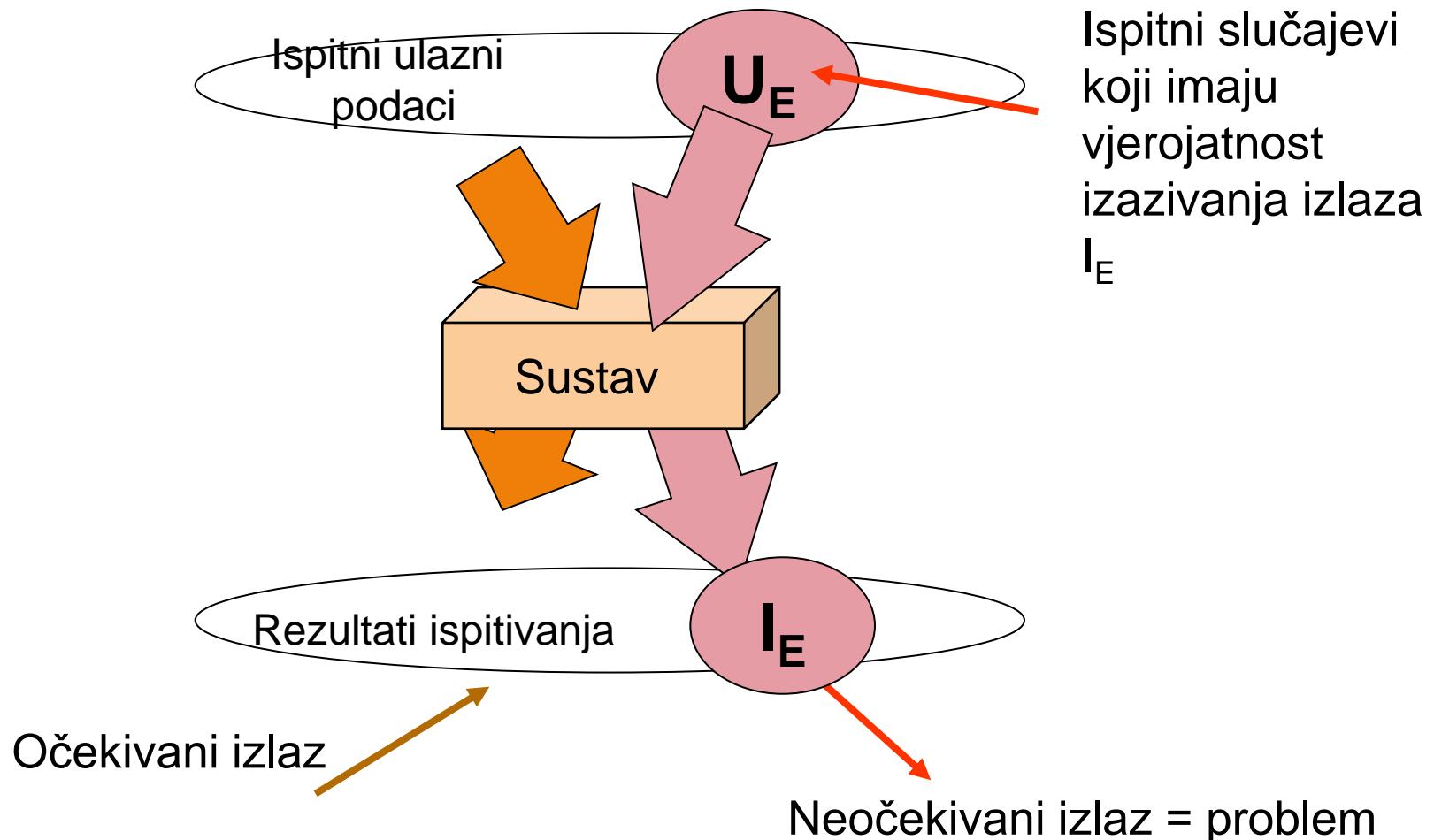


# Funkcijsko ispitivanje



- engl. *Black box testing*
- Prepostavlja da *nema znanja programskog koda* ili oblikovanja sustava
  - koncentracija na U/I ponašanje
  - ispitivanje samo prema zahtjevima i specifikacijama
- Prepostavka je da za ulazne podatke možemo predvidjeti izlaz
  - gotovo nemoguće generirati sve moguće ulaze!
- Cilj: Smanjiti broj ispitnih slučajeva ekvivalentnom podjelom ulaza i analizom graničnih vrijednosti
- Oblikovanje ispitnih slučajeva (engl. *test case design*)
  - zasnovano na specifikaciji sustava
  - podjela vrijednosti ulaznih podataka
  - podjela ulaznih podataka u klase
  - odabir ispitnih slučajeva za svaku klasu
  - analiza graničnih vrijednosti

# Funkcijsko ispitivanje



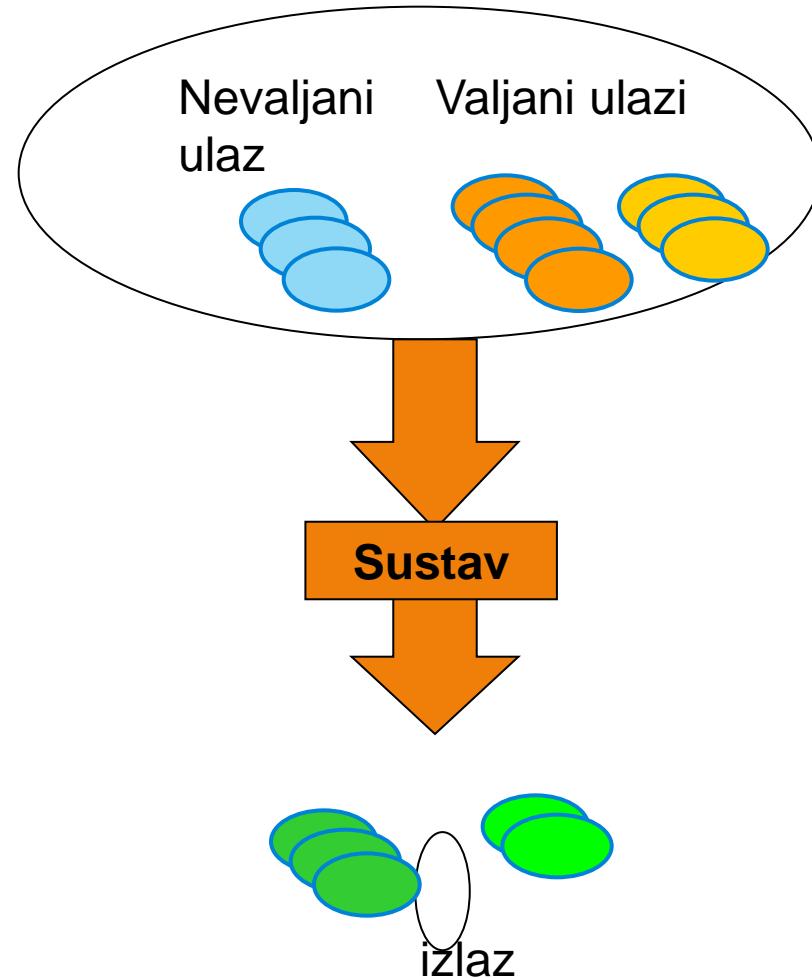
# Ispitivanje particija

- Ulazni podaci i rezultati često se mogu grupirati u različite klase u kojima je ponašanje članova slično (princip: *Podijeli i vladaj*).
- Svaka od tih klasa predstavlja ekvivalentnu particiju (engl. *equivalence partition*) u kojoj se program ponaša na isti način za sve njegove članove
- Ispitni slučajevi moraju pokrivati sve ekvivalentne particije
  - posebice granične vrijednosti svake particije jer su to najčešći uzroci kvara
- Odabir ekvivalentnih particija za ispitivanje
  - podijeliti ulazne ekvivalentne particije na:
    - Valjana vrijednost
    - Nevaljana vrijednost
  - vrijednosti ulaza su valjane u intervalu –odabratи 3 ispitna slučaja
    - Vrijednost ispod intervala
    - Vrijednost u intervalu
    - Vrijednost iznad intervala

**PRETPOSTAVKA**

# Ekvivalentne podjele

- Ekvivalentne podjele ulaza:
  - Valjani ulazi
  - Nevaljani ulazi
- Izlazne ekvivalentne podjele
  - zajedničke karakteristike



# Koraci ispitivanja

1. Odredi particije za sve ulazne varijable.
2. Za sve particije odaberि vrijednosti ispitivanja.
3. Definiraj ispitne slučajeve koristeći odabrane vrijednosti.
4. Odredi očekivane izlaze za odabrane ispitne slučajeve i provedi ispitivanje.

# Primjer funkciskog ispitivanja

- Zadatak: *Ispitati program za odlučivanje o pravu na glasovanje.* Pravo glasovanja ovisi o starosti osobe, minimalna dob je 18 godina.
  - jedna varijabla - dob
  - svi ulazni podaci mogu se podijeliti u **dvije ekvivalentne podjele**
    - 0-17 - ne smije glasovati
    - $\geq 18$  - smije glasovati
  - ispitni slučaj 1
    - Odabir reprezentativnih podataka iz svake podjele
  - ispitni slučaj 2
    - Odabir podataka na granicama podjela

Ispitni slučaj 1

Test	Podatak	Rezultat
1	12	NE
2	21	DA

Ispitni slučaj 2

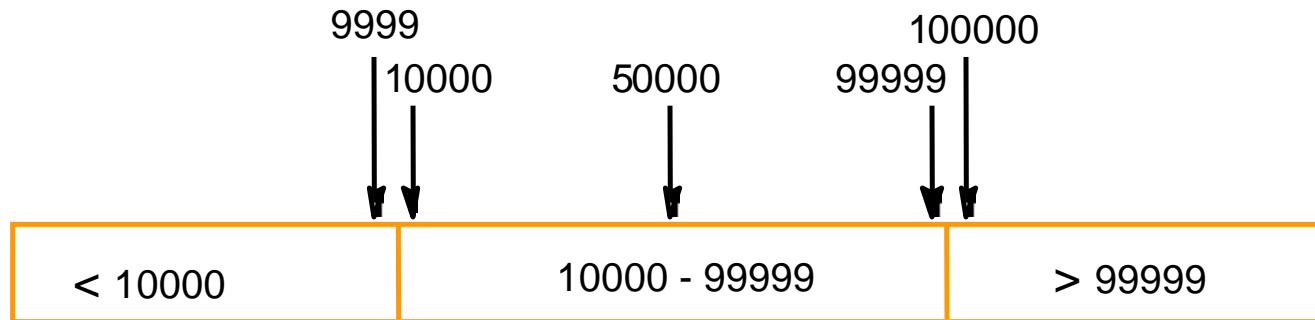
Test	Podatak	Rezultat
3	17	NE
4	18	DA



# Ekvivalentne podjele



- Primjer: Ulaz 5-znamenkasti broj vrijednosti između 10000 i 99999
  - <10,000
  - 10,000-99,999
  - >99,999





# Primjer: komponenta za pretraživanje



- Pretražuje se sekvencija (niz) elemenata za dani element (ključ - key) zadana slijedećom **apstraktnom specifikacijom** (ne implementacijom):

```
public static void trazi (int [] element, int kljuc,
 bool nadjen, int indeks)
```

**Preduvjet** - istinit prije poziva komponente

- rutina radi (pretražuje) samo za neprazne sekvenčne - ulazni niz ima najmanje jedan element

*element'first <= element'last*

**Rezultat** - istinit nakon izvođenja komponente

- varijabla *nadjen = T* ako je traženi element u nizu (na mjestu L):

*nadjen and element(L) = kljuc*

Ili

- traženi element se ne nalazi u nizu:

*not nadjen and not (exists i, element'first >= i <= element'last, element (i) = kljuc)*



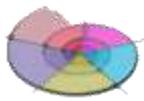
# Primjer: podjela particija



- Podjela particija temeljem specifikacije:
  - ulazi koji zadovoljavaju preuvjete
  - Ulazi kod kojih je ključ element zadanog niza
  - Ulazi kod kojih je ključ nije element zadanog niza
  - ulazi koji ne zadovoljavaju preuvjete
    - Ulazi kod kojih je ključ element zadanog niza
    - Ulazi kod kojih je ključ nije element zadanog niza
- Naputak za provedbu ispitivanja:
  - ispitaj pretraživanje na slijedećim nizovima:
  - niz je jedan broj (programeri obično pretpostavljaju više)
  - niz ima paran broj vrijednosti
  - niz ima neparan broj vrijednosti
- Preporučeni slijed ispitivanja
  1. Ispitati jednostrukе vrijednosti
  2. Sekvence (nizove) različitih duljina neophodno ispitati zasebno
  3. Ispitne slučajeve oblikovati na taj način da obuhvaćaju prvi, srednji i zadnji element sekvene (niza)
  4. Ispitati sekvene (nizove) duljine 0

# Primjer:

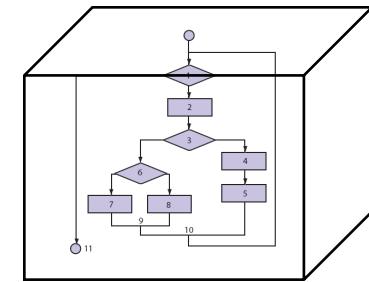
element ( <b>ulazni niz</b> )	kljuc	nadjen, <b>L=mjesto</b>	Opis
17	17	T, 1	1 element niza Traženi element je u nizu
17	0	F, ?	1 element niza Traženi element nije u nizu
17, 21, 23, 29	17	T, 1	Više elementa niza Traženi element prvi u nizu
9, 16, 18, 30, 31, 41, 45	45	T, 1	Više elementa niza Traženi element zadnji u nizu
17, 18, 21, 23, 29, 38, 41	23	T, 4	Više elementa niza Traženi element u sredini niza
17, 18, 21, 23, 29, 38, 41	21	T, 3	Više elementa niza Traženi element u sredini niza
12, 18, 21, 23, 32	23	T, 4	Više elementa niza Traženi element u sredini niza
21, 23, 29, 33, 38	25	F, ?	Više elementa niza Traženi element nije u nizu



# Funkcijsko ispitivanje - zaključci



- Izražen problem kombinatorne eksplozije ispitnih slučajeva
  - posebice izraženo kod potrebe ispitivanja valjanih i nevažećih podataka
- Često nije jasno da li su odabrani ispitni slučajevi dobro oblikovani za otkrivanje ciljane pogreške
  - nepoznavanje unutarnje strukture!
- Jednostavno za uporabu
- Brzi razvoj ispitnih slučajeva
- Ispitni slučajevi se rade iz perspektive korisnika
- Neovisno o jeziku implementacije

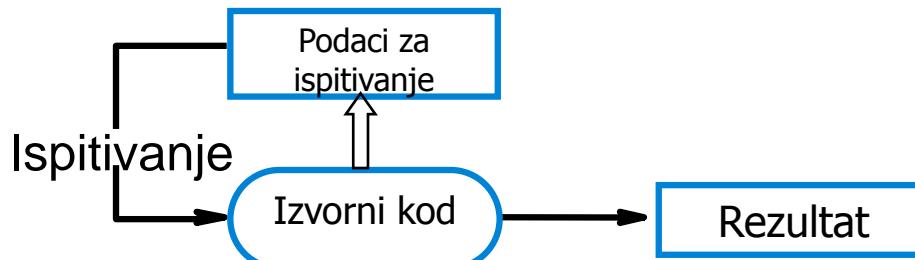


Tehnike ispitivanja

# STRUKTURNO ISPITIVANJE

# Struktурно ispitivanje

- engl. *Structural testing, White Box testing*
- Ispitivanje očekivanog ponašanja zasnovano na svojstvima programa ili oblikovanju (tj. *strukturi programa*)



- Cilj ispitivanja je pokrivanje izvođenja svih mogućih naredbi i uvjeta programa najmanje jednom
- Primjer:
  - ispitivanje komponenti, struktурно ispitivanje (engl. *statement coverage, branch coverage, unit testing, i sl.*)
- Oblikovanje ispitnih slučajeva (engl. *test case design*)
  - zasnovano na strukturi programa (poznamo strukturu)

# Prikaz programa

- apstrakcija programskog koda
- Graf tijeka programa
  - engl. *control flow graph*
  - Graf. *reprezentacije tijeka programa*
- Čvorovi – procesi ○, programske odluke ○ ili ◇
- Lukovi – kontrola tijeka

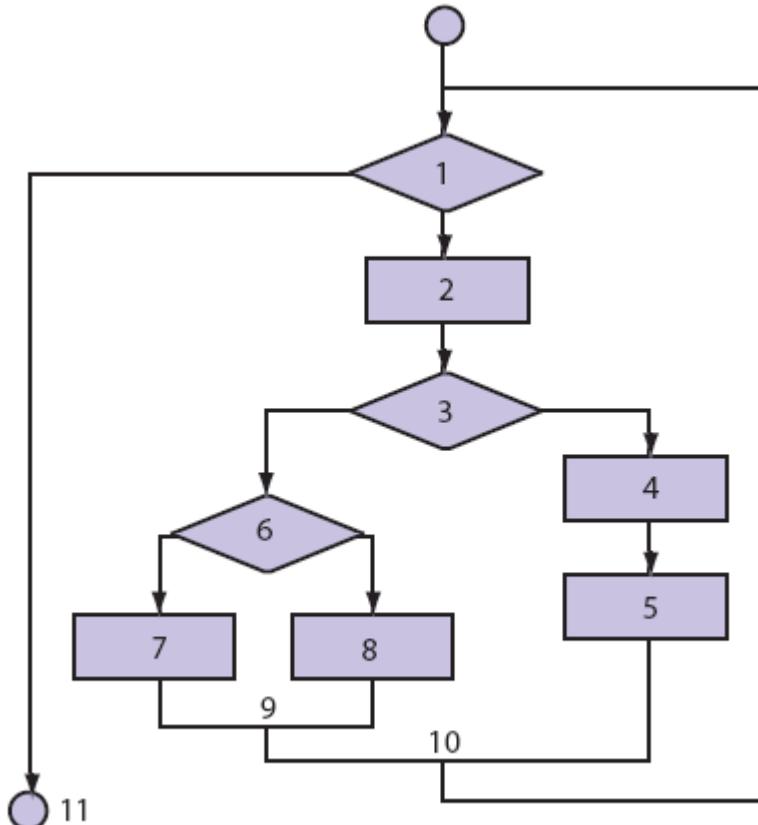
```
public roots (double a, b, c)
{
 double q = b * b - 4 * a * c;
 if (q > 0 && a != 0) {
 ...
 }
 else if (q == 0) {
 x = (-b) / (2 * a);
 }
 else {
 ...
 }
 else {
 ...
 }
}
```



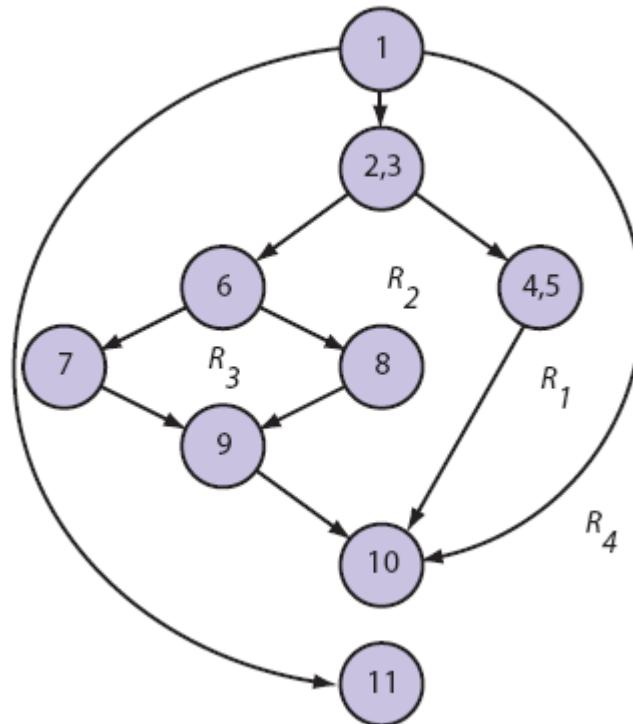
# Primjer: Dijagram toka i grafa tijeka programa

- Dijagram toka

- engl. *flowchart*



- Graf tijeka programa



# Primjer struktturnog ispitivanja

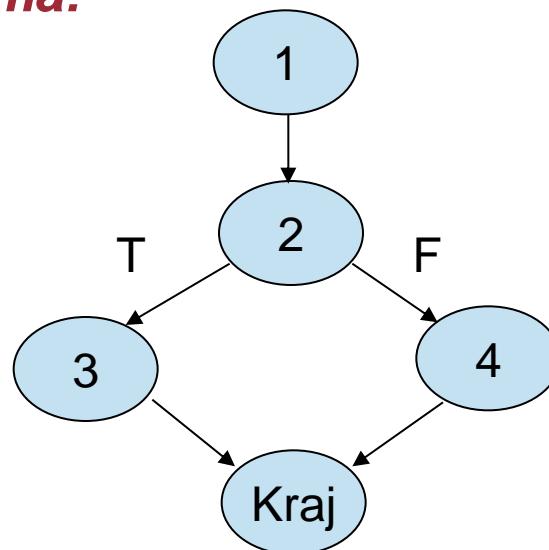
- Provjera prava glasovanja

```

učitaj (dob);
2 -----> ako (dob>= 18) tada {
 ispiši ("smije glasovati");
}
inače {
 ispiši ("ne smije glasovati")
}

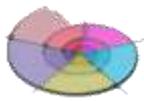
```

- **Graf tijeka programa:**



Mogući putovi:

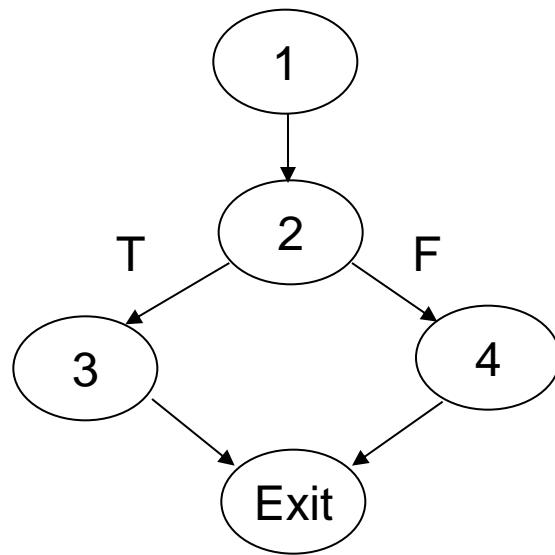
1. 1 → 2 → 3 → kraj
2. 1 → 2 → 4 → kraj



# Primjer struktturnog ispitivanja



## Graf tijeka programa



Ispitni slučaj 1: osigurati izvođenje oba puta tijekom ispitivanja

Test	Podatak	Rezultat
1	12	NE
2	21	DA

## Analiza putova

Mogući putovi:

1. 1 → 2 → 3 → kraj
2. 1 → 2 → 4 → kraj

Ispitni slučaj 2: osigurati ispravan rad *if* naredbe na graničnim vrijednostima

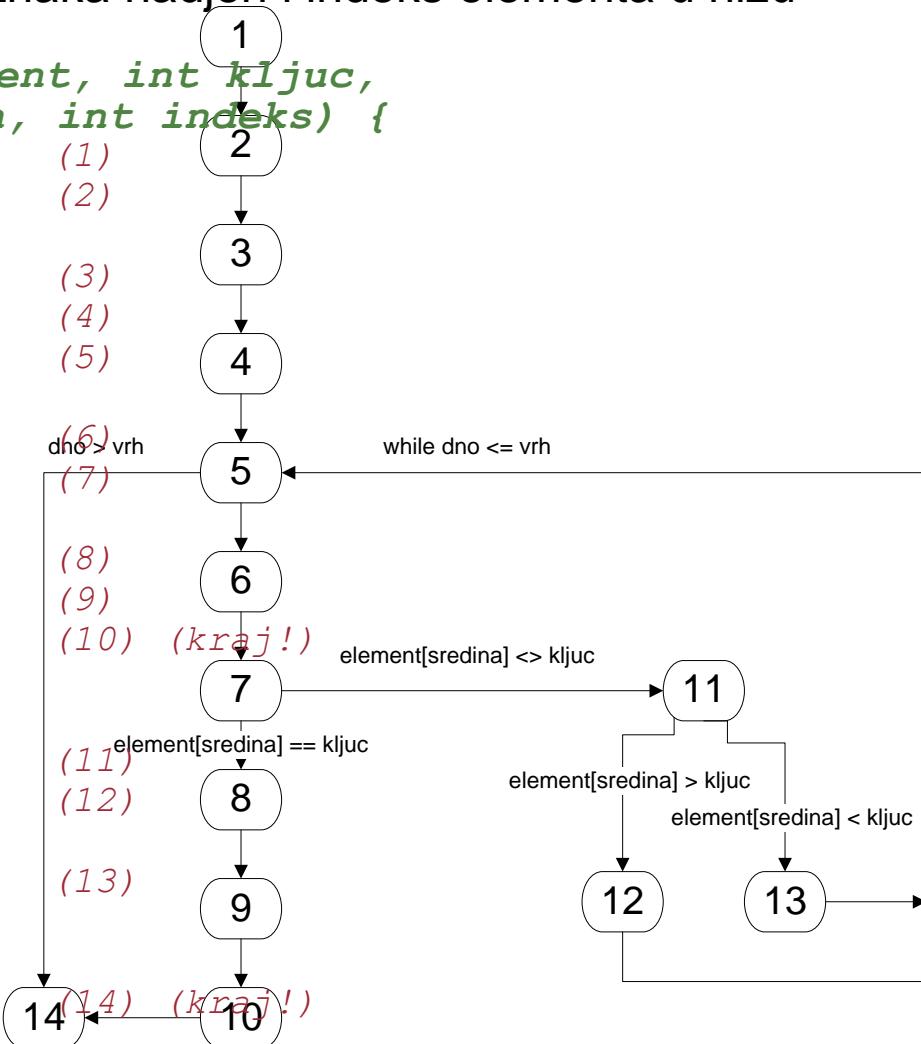
Test	Podatak	Rezultat
3	17	NE
4	18	DA



# Primjer: binarno pretraživanje

- Ulaz uređen niz elemenata i ključ, izlaz oznaka nadjen i indeks elementa u nizu

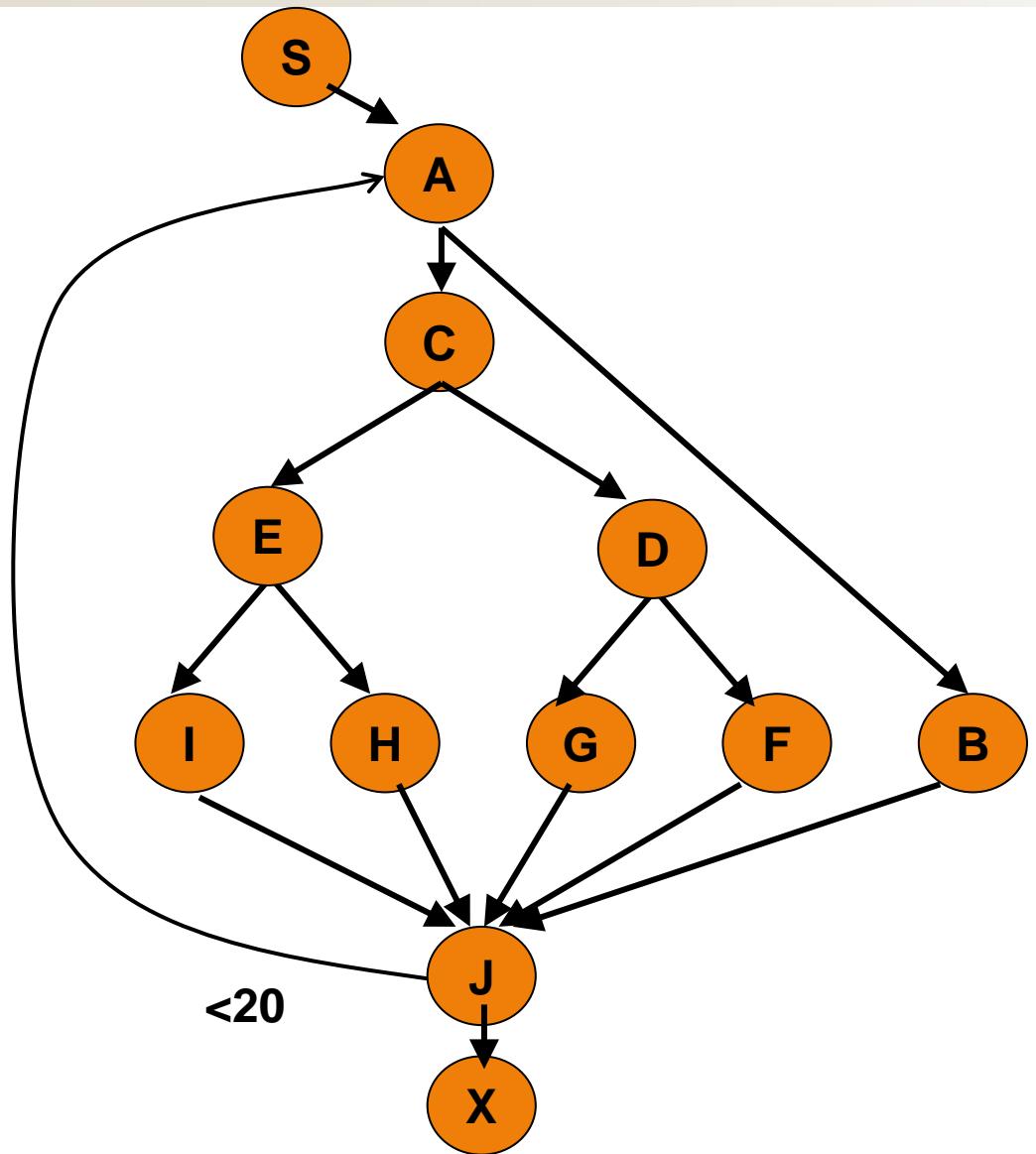
```
class Pretraga {
public static void trazi (int [] element, int kljuc,
 bool nadjen, int indeks) {
 int dno = 0 ;
 int vrh = element.length - 1 ;
 int sredina ;
 nadjen = false ;
 indeks = -1 ;
 while (dno <= vrh)
 {
 sredina = (vrh + dno) / 2 ;
 if (element [sredina] == kljuc)
 {
 indeks = sredina ;
 nadjen = true ;
 return ;
 }
 else {
 if (element [sredina] < kljuc)
 dno = sredina + 1 ;
 else
 vrh = sredina - 1 ;
 } //while
 } // void trazi
} // class Pretraga
```



# Analiza putova

- Problem predstavlja postojanje programske petlje
  - veliki broj putova
  - putovi koji prolaze kroz petlju više puta su ekvivalentni
  - uobičajeno velik broj mogućih kombinacija
- Ispituju se:
  - svi neovisni putovi
  - logički izrazi (T,F)
  - petlje (rubovi, sredina)
  - interni podaci - struktura
- Izvođenje ispitnog slučaja
  - osigurava prolaz kroz određeni put

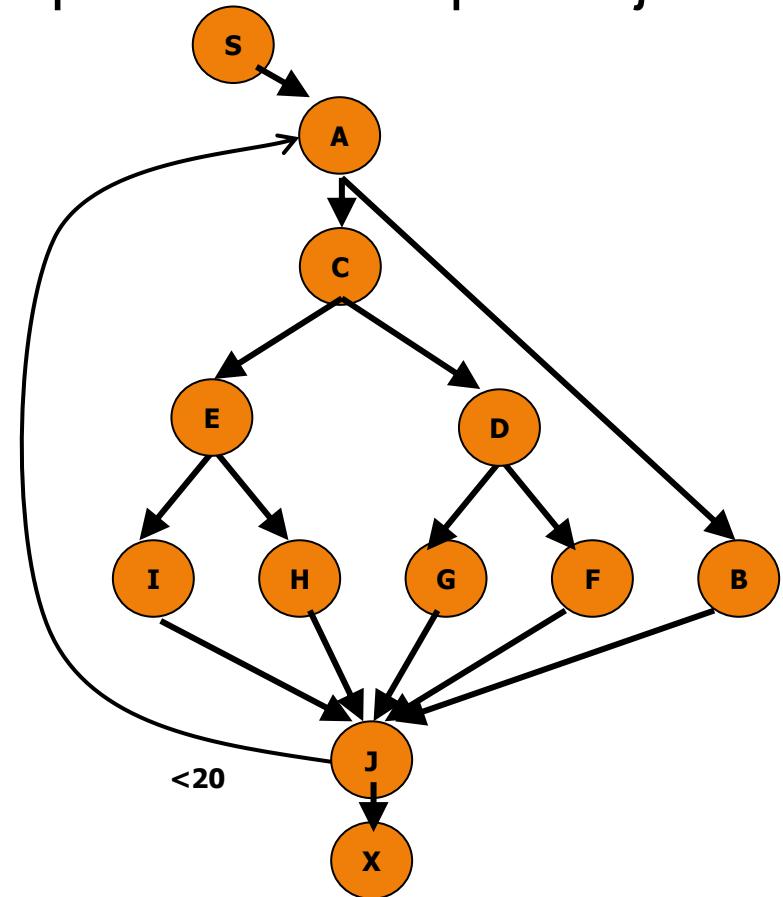
# Primjer sekvenci



- Putovi od  $S \Rightarrow X : 5$
- Cilj ispitati petlju od 1 do 20 puta sa **SVIM** kombinacijama putova u ponavljanju petlje
  - $5 \times 5 \times 5 \dots \times 5$
  - $5^1 + 5^2 + \dots + 5^{19} + 5^{20} = 10^{14}$
- Iscrpno ispitivanje
- engl. *exhaustive testing*
- Ispituje sve moguće kombinacije
  - da li je ostvarivo?
- **Trajanje??**
  - ako ispitni slučaj traje 1 ms
    - $\Rightarrow 3170$  godina

# Selektivno ispitivanje

- Ispitivanje svih kombinacija je najčešće nemoguće
- Pokrivanje koda (engl. *Code coverage*)** predstavlja pokrivenost izvornog koda programa provedenim ispitivanjima
- Suzujemo prostor ispitivanja na:
  - ispitivanje temeljnih putova
    - engl. *Basis path testing*
  - ispitivanje uvjeta
    - engl. *Condition testing*
  - ispitivanje petlji
    - engl. *Loop testing*
  - ispitivanje protoka podataka
    - engl. *Dataflow testing*





# Ispitivanje temeljnih putova



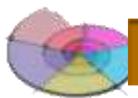
- Temeljni skup (*engl. basis set*)
  - Skup putova koji minimalno jednom pokrivaju izvođenje svih naredbi i uvjeta
    - Ne mora biti jednoznačan
- Teorija grafova – izračun broja linearne neovisnih putova
  - CV(G) - Ciklomatička složenost (*engl. Cyclomatic complexity*)
  - Broj neovisnih putova u temeljnog skupu
- Primjena za mjerjenje količine logike odlučivanja u programskom modulu – 1974.g. McCabe
  - $CV(G) = \text{Lukovi} - \text{Čvorovi} + 2^*P$ 
    - Lukovi = broj lukova u grafu
    - Čvorovi = Broj čvorova u grafu
    - P = Broj povezanih komponenti (potprograma, prekidnih rutina i sl.)
  - gornja granica broja ispitnih slučajeva koja garantira potpuno pokrivanje svih naredbi programa



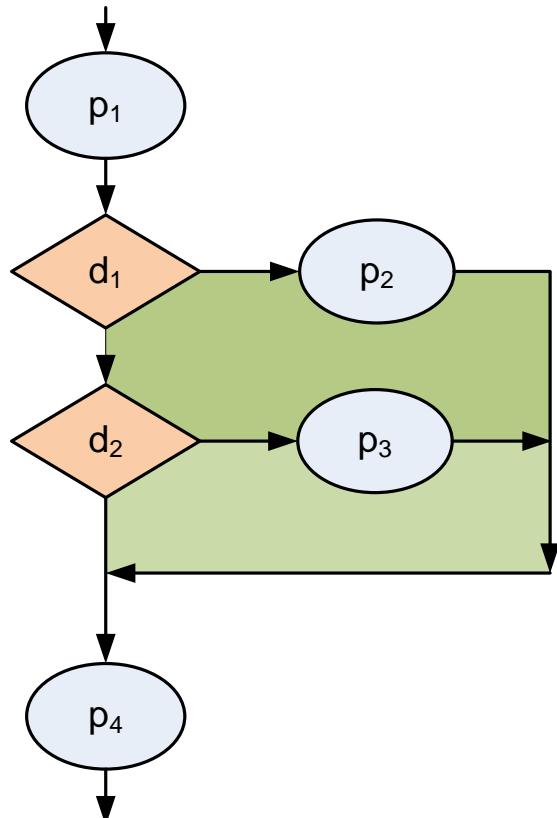
# Ciljevi pokrivanja



- Ispitivanje nema za cilj povećati postotak pokrivanja
- Pokrivanje nam govori u kojoj smo fazi ispitivanja
  - veći postotak ne dokazuje da u programu nema kvarova
- Nikada se ne traže načini povećanja pokrivanja samo radi pokrivanja
  - često se može postići relativno jednostavnim ispitivanjima za koje znamo da nemaju veliku šansu otkrivanja kvara
  - u nekim slučajevima ipak može biti dio zahtjeva
  - npr. ako postoji kvar koji izaziva zatajenje pri svakom izvođenju nema smisla ponavljati ispitivanja
- Za svako pokrivanje mora postojati obrazloženje
  - tj. obrazložiti otkrivanje kojih kvarova se propušta nepokrivanjem



# Primjer: Izračun ciklomatičke složenosti

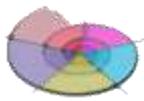


- Procesi=4
- Odluke=2
- Lukovi=7
- Čvor=  $4+2=6$
- $P = 1$  (nema dodatnih komp.)

$$CV(g)=7-6+2=3$$

Preporuka:

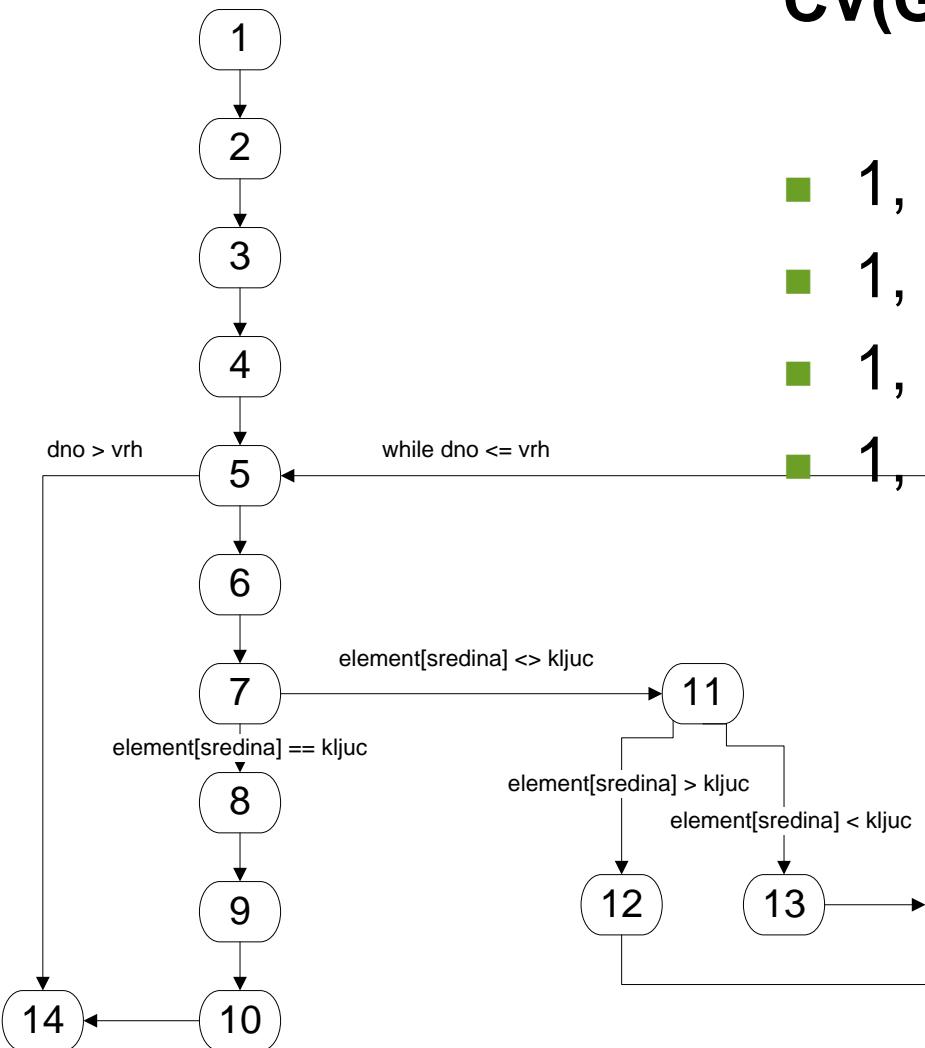
- Ciklomatička složenost modula < 10



# Primjer: Binarno pretraživanje



$$\begin{aligned} \text{CV(G)} &= \text{Lukovi} - \text{Čvorovi} + 2 * P \\ &= 16 - 14 + 2 = 4 \end{aligned}$$



Čvor je linija u programu

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...

Ako se obiđu svi navedeni putovi:

- svaka naredba je ispitana najmanje jednom
- ispitano je svako grananje

# Ispitivanje uvjeta

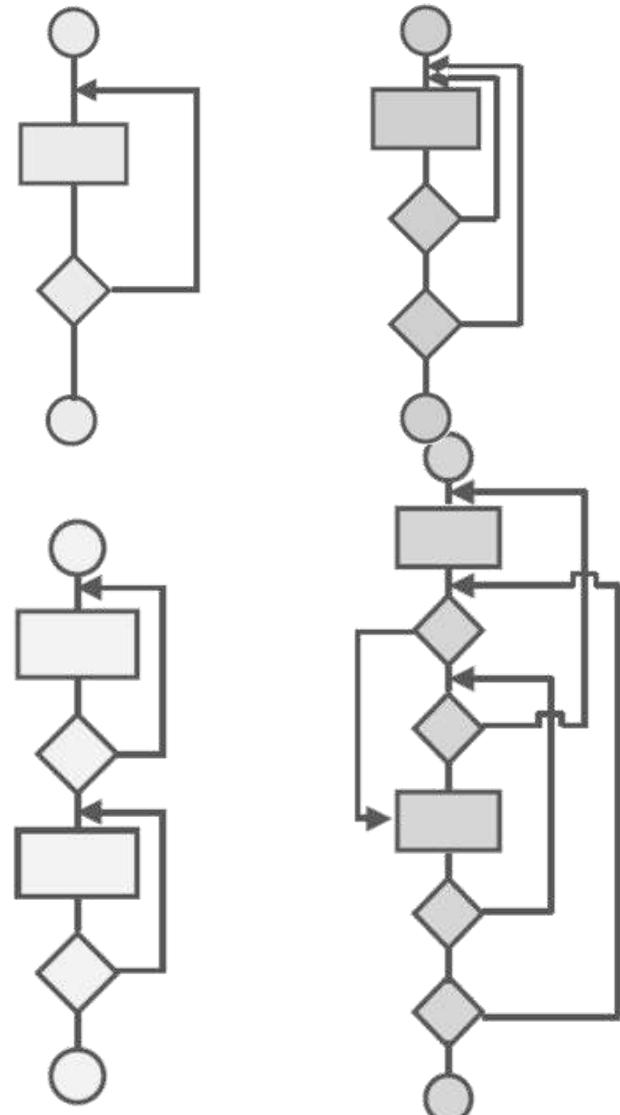
- Ispitivanje svih logičkih uvjeta u modulu
  - svaki uvjet poprima vrijednosti istinitosti:  $\text{True}$  i  $\text{False}$
  - jednostavni uvjeti
    - Booleve varijable  $\text{True}$ ,  $\text{False}$
    - Jednostavni relacijski izrazi  $a < b$ ,  $a \geq b$  .....
  - složeni izrazi
    - $( (a=b) \ \& \ (c>d) )$
- Metode ispitivanja
  - ispitivanje grana (svaka grana svakog uvjeta ispituje se bar jednom)
  - ispitivanje domene
    - Npr. za relaciju  $a < b$ , treba provesti 3 ispitna slučaja:  $a < b$ ,  $a = b$ ,  $a > b$
    - Booleov izraz sa  $n$  varijabli  $\Rightarrow 2^n$  ispitnih slučaja

# Ispitivanje petlji

- Kritične točke programa
- Fokus na valjanost primjene petlje

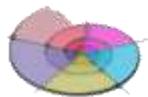
```
while x < 20 loop
 { . . . }
end loop;
```

- Klasifikacija petlji
  - a) Jednostavna (engl. *simple loop*)
    - $n = \text{max broj prolaza}$
  - b) Ugnježđena (engl. *nested loop*)
    - Iznutra prema van (geom. složenost)
  - c) Ulančana (engl. *concatenated loop*)
    - Pristup kao jednostavne ili ugnježdene
  - d) Nestrukturirana (engl. *unstructured loops*)
    - loše programiranje



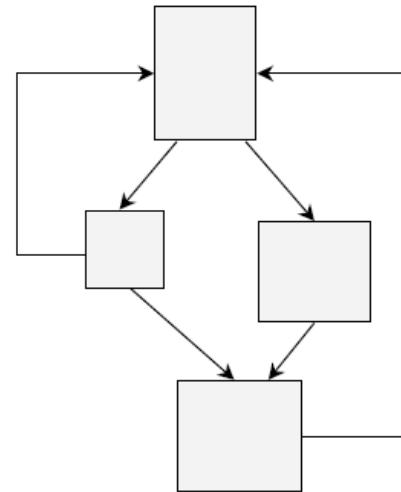
# Primjer ispitivanja petlji

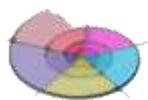
- Ispitni slučaj za jednostavnu petlju:
  - odrediti  $n =$  maksimalno dozvoljeni broj prolaza
    1. preskočiti petlju
    2. jedan prolaz
    3. dva prolaza
    4.  $m$  prolaza ( $m < n$ )
    5.  $n-1$  prolaz
    6.  $n+1$  prolaz
- Kako ispitati ugnježdene petlje?
- Kako ispitati ulančane petlje?



# Ispitivanje protoka podataka

- Ispitivanje upravljačkog toka obzirom na uporabu varijabli i uočavanje kvarova provjerom uzorka uporabe podataka
- Za svaku cjelinu odrediti uporabu varijabli
  - **definiranje** (*D*) - deklaracija varijable (objekta), konstruktor, pridjeljivanje vrijednosti
  - **uporaba** (*U*) - bez mijenjanja vrijednosti
  - **brisanje** (*K*) - varijabla postaje nedefinirana, oslobođanje memorije (*engl. garbage collection*)
  - **nebitno** (*X* – prijašnje akcije i *X~* nebitno nakon)
- Analiza slijeda akcija nad varijablama
- Npr. ispravni sljedovi – DD, DU, UU, UD, UK

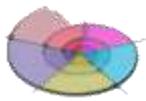




# Struktурно испитивање -закључци



- Potencijalno beskonačan broj putova za испитивање
- Često испituje ono što je implementirano, umjesto što bi trebalo biti
- Увид у извор погрешке
- Пovećana ponovna uporaba испитних slučajeva
- Mogućnost ciljanog испитивања критичних dijelova
  
- Osjetljivo na promjene izvornog koda

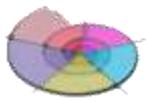


# Ispitivanje sive kutije



- engl. *Gray box testing*
- Funkcijsko i strukturno ispitivanje predstavljaju ekstremne slučajeve
- Uobičajeno se funkcijsko ispitivanje kombinira s poznavanjem koda za potvrdu očekivanih rezultata
  - najčešće interne strukture podataka i algoritmi implementacije

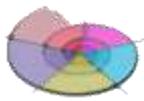
# AUTOMATIZACIJA ISPITIVANJA



# Automatizacija ispitivanja



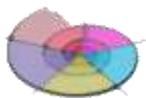
- Ispitivanje i ispravljanje pogrešaka skupo i vremenski zahtjevno
  - 1995. g. *“50% of my company employees are testers, and the rest spends 50% of their time testing!” -Bill Gates*
- Obzirom na prirodu programa pogodi su za ispitivanje primjenom drugih programima
  - engl. *Computer Aided Software Testing CAST*
  - razvijeni mnogobrojni specijalizirani alati radne okoline
- Prednosti automatizacije
  - brže i jeftinije
  - poboljšanje točnosti
  - stabilno ispitivanje kvalitete
  - automatizirano dokumentiranje prijava pogrešaka i izvješćivanje
  - smanjenje ljudskog rada



# Programska podrška ispitivanju



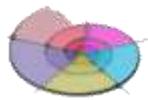
- Automatizacija na različitim razinama ispitivanja:
- Komponenta
  - npr. *izgradnja upravljačkih programa i, analiza pokrivanja*
- Integracije
  - sučelja i protokoli
- Sustava
  - automatizirani izvođenje ispitivanja i performanse alata
- Prihvatljivosti
  - analiza zahtjeva, implementacije, upotrebljivosti ...
- Pomoćne aktivnosti ispitivanja
  - praćenje kvarova (*engl. Bug Tracking Tools*)
  - upravljanje procesom ispitivanja (*engl. Test Management Tools*)



# Automatizacija ručnog ispitivanja



- Visoka cijena ručne provedbe ispitivanja
  - Ponavlja se svakim ispitivanjem, potrebno ponoviti nakon svakog ispravljanja
- Automatizacija ručnog ispitivanja
  1. generiranje ulaznih podataka i očekivanih rezultata
  2. izvođenje ispitivanja
  3. evaluacija
  - jedinično 3-30 puta cijene ručnog ispitivanja
  - zahtjeva formalizirani ručni proces ispitivanja
  - cijena ovisi o postavljenom dosegu ispitivanja
    - Postupak kodiranja ispitnih slučajeva
  - cijena ponavljanja  $\approx 0$
- Prednosti
  - Povećana pouzdanost
  - Povećana kvaliteta ispitivanja
    - automatizacija procesa
  - Kraće vrijeme izvođenja ispitnih slučajeva
  - Automatska analiza rezultata ispitnih slučajeva
- Nedostatci
  - Cijena
  - Vrijeme pripreme



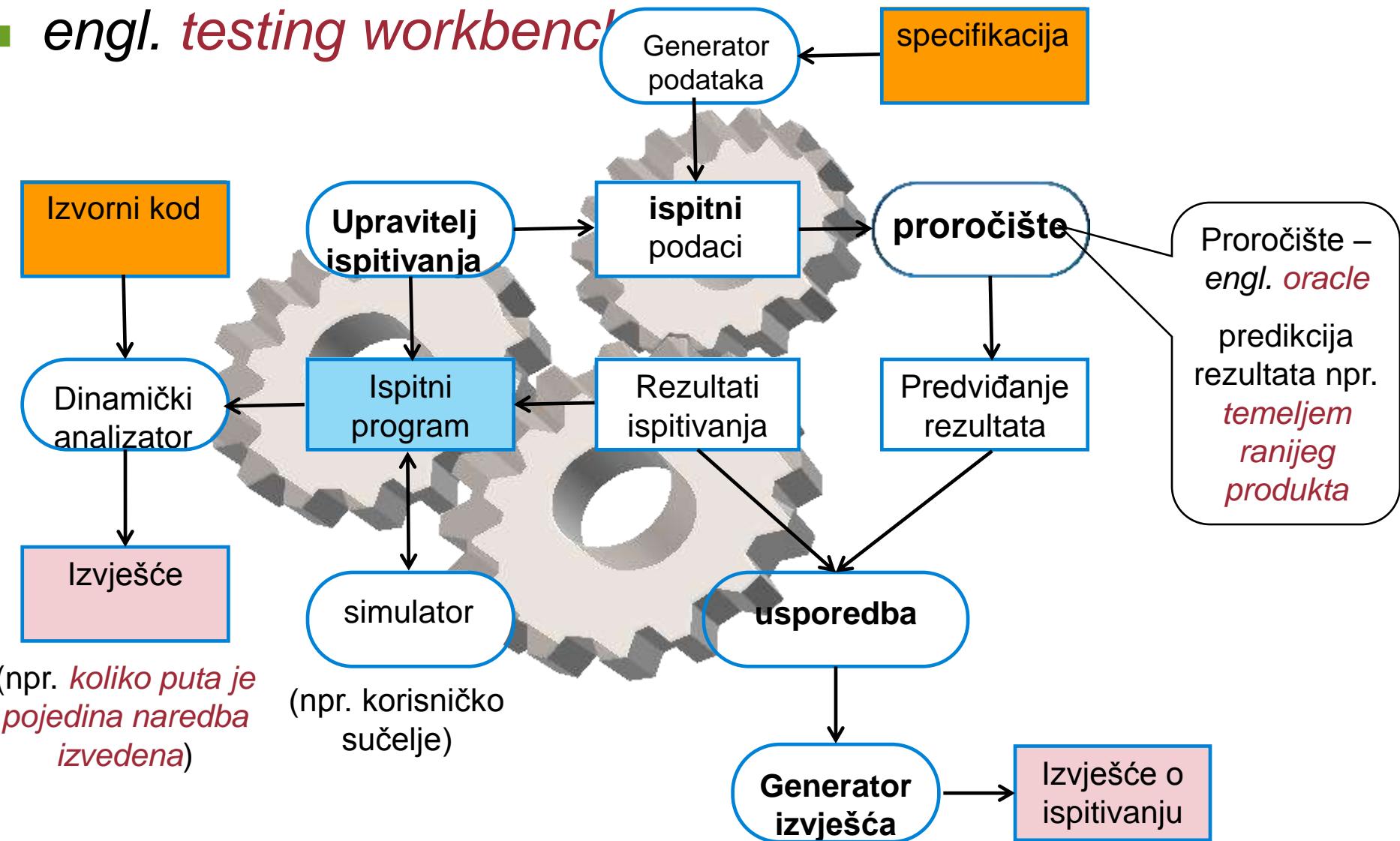
# Razine automatizacije ispitivanja



- Osnovna automatizacija
  - *metodologija snimanja i reprodukcije*
  - *engl. Record&Playback Methodology*
    - skripte generirane bilježenjem korisnikovih akcija
  - *podatkovno upravljana metodologija*
  - *engl. Data-Driven Methodology*
    - podaci nisu ukodirani u skriptu - čitaju se iz datoteka
- Napredna automatizacija
  - funkcionalna dekompozicija
    - male skripte predstavljaju ispitne module i funkcije
  - radni okviri - *Framework*
    - *Test Framework*
    - *Process Framework*
    - *Hybrid Framework*

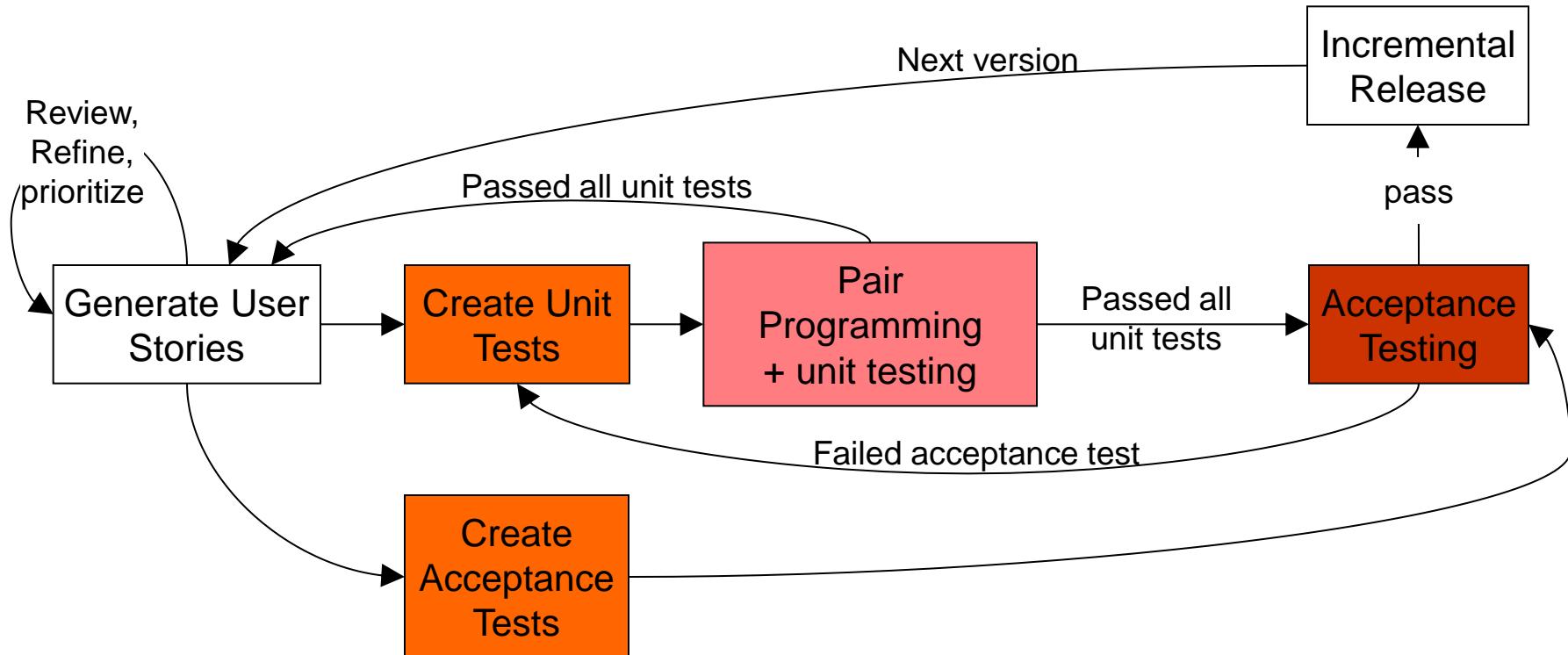
# Ispitna radna klupa

## ■ engl. *testing workbench*





# Primjer: Extreme Programming (XP)



# Primjer: JUnit

- JUnit – framework
  - okvir za ispitivanje Java programa
- Princip: “Code a little, test a little”
  - Erich Gamma, Kent Beck
  - zasnovan na Javi
    - Java programi mogu ispitivati vlastiti kod
  - JUnit
    - Definira i izvodi ispitne slučajeve
    - Formalizira zahtjeve
    - Pisanje i ispravljanje koda
    - Integrira se s ispitnim slučajem
  - razvojne okoline
    - BlueJ, JBuilder, Eclipse, NetBeans
  - više: JUnit.org <http://www.junit.org/>
    - Resources for Test Driven Development
    - “JUnit is a simple framework for writing and running automated tests. As a political gesture, it celebrates programmers testing their own software.”

# Primjer: JUnit

```
public class Primjer {
 static public int Zbroji(int a, int b) {
 return a + b;
 }
}
```

```
import junit.framework.Test ;
import junit.framework.TestSuite ;

public class testPrimjer{
 @Before public void setUp() {
 System.out.println("Testiranje počinje");
 }
 @After public void tearDown() {
 System.out.println("Završeno!!");
 }
 @Test public void testZbroji(){
 int i = Primjer.zbroji(2,3);
 assertEquals(5,i);
 }
 @Test public void testZbrojiL(){
 int i = Primjer.Zbroji(3,5);
 assertTrue("Pogrešna suma",i!=5);
 }
}
```

← izvorni kod

dodaci za ispitivanje

```
<junit printsummary="yes"
 haltonfailure="yes"
 showoutput="yes">

<classpath>
 <pathelement path="${build}" />
</classpath>

<batchtest fork="yes"
 todir="${reports}/raw/">
 <formatter type="xml"/>
 <fileset dir="${src}">
 <include name="**/*Test*.java"/>
 </fileset>
</batchtest>

</junit>
```



# Statistika uklanjanja kvarova



- Analize izvora pogrešaka ukazuju na tipičnu raspodjelu: 60% pogreška oblikovanja, a 40% pogrešaka kodiranja
- Efikasnost uklanjanja pogrešaka ovisi o kvaliteti razvojnog tima i primijenjenoj metodologiji

metoda	Potencijal pogrešaka	Efikasnost uklanjanja	Pogreške*
TSP	2.70	97%	0.08
CMMI 5	3.00	96%	0.12
RUP	3.90	95%	0.20
CMMI 3	4.50	93%	0.32
XP	4.50	92%	0.38
Agile	4.70	91%	0.42
CMMI 1	5.00	85%	0.75

Tip	Isporučeno pogrešaka*	KESLOC
System Software	0.4	1
Commercial Software	0.5	8
Information Software	1.2	10
Military Software	0.3	<1

Izvor: Jones, C.: Applied Software Measurement: Global Analysis of Productivity and Quality

\*Defects per function point – broj pogrešaka u odnosu na ukupan broj korisničkih funkcionalnosti

Capability Maturity Model Integration (CMMI) – definira razine zrelosti organizacije u primjeni metodologije poboljšanja procesa

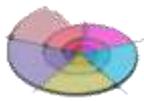
Team Software Process (TSP) - definira procese organizacije s ciljem poboljšanja razine kvalitete i produktivnosti tima

# Zaključak

- Ispitivanje je složena i zahtjevna aktivnost oblikovanja programske podrške
  - provodi se **na svim razinama** i planira kao **sastavni dio razvojnog procesa**
  - **kritični dio** razvoja sustava
    - troši ~50% vremena, a zahtjevi za pouzdanošću rastu
  - nemogućnost potpunog ispitivanja
  - **problem strategije**
    - veliki broj ciljeva  $\Rightarrow$  oblikovanje strategije za postizanje ciljeva
    - **problem odluke** – engl. *The Oracle Problem*
      - Da li je program prošao ispitivanje ili ne?
    - loše oblikovani sustavi otežavaju ispitivanje
- Ispitivanje može pokazati postojanje pogreške, a ne može dokazati njihovo nepostojanje!
- Postoje razvijena pravila i razne dobre prakse
  - Upotrebljavati iskustvo i smjernice za oblikovanje ispitnih slučajeva
- Trend predstavlja integracija ispitivanja i razvoja
- Alternativa: uporaba formalnih metoda

# Diskusija

- 
- 
- 
- 
-



# Ispitni slučajevi za SimpleChat



## Test Case 2002

System: SimpleChat Phase: 2

Client startup check without a login

Severity: 1

Instructions:

1. *At the console, enter: java ClientConsole.*

Expected result:

1. *The client reports it cannot connect without a login by displaying:*

ERROR - No login ID specified. *Connection aborted.*

2. *The client terminates.*

Cleanup: (if client is still active)

1. Hit CTRL+C to kill the client.

## Test Case 2003

System: SimpleChat Phase: 2

Client startup check with a login and without a server

Severity: 1

Instructions:

1. At the console, enter: java ClientConsole <loginID>  
where <loginID> is the name you wish to be identified by.

Expected result:

1. The client reports it cannot connect to a server by displaying:  
cannot open connection. Awaiting command.
2. The client waits for user input

Cleanup: (if client is still active)

1. Hit CTRL+C to kill the client.



# Ispitni slučajevi za SimpleChat



## Test Case 2007

System: SimpleChat Phase: 2

Server termination command check

Severity: 2

Instructions:

1. Start a server (Test Case 2001 instruction 1) using default arguments.
2. Type #quit into the server's console.

Expected result:

1. The server quits.

Cleanup (If the server is still active):

1. Hit CTRL+C to kill the server.

## Test Case 2013

System: SimpleChat Phase: 2

Client host and port setup commands check

Severity: 2

Instructions:

1. Start a client without a server (Test Case 2003).
2. At the client's console, type #sethost <newhost> where  
<newhost> is the name of a computer on the network
3. At the client's console, type #setport 1234.

Expected result:

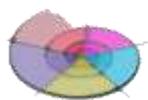
1. The client displays

Host set to: <newhost>

Port set to: 1234.

Cleanup:

1. Type #quit to kill the client.



# Ispitni slučajevi za SimpleChat



## Test Case 2019

System: SimpleChat Phase: 2

Different platform tests

Severity: 3

Instructions:

1. Repeat test cases 2001 to 2018 on Windows 95, 98, NT or 2000, and Solaris

Expected results:

1. The same as before.

# Oblikovanje programske podrške

ak.god. 2014./2015.

## *Temelji formalne verifikacije*

## *Logika*



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



# Sadržaj

- Formalna matematička logika
  - propozicijska
  - predikatna
- Preslikavanje formula predikatne logike u normalizirane klauzule
- Postupci formalne verifikacije računalnih sustava
  
- Cilj:
  - Osnove logike i formalnog promatranja apstraktno i simbolički
  - Motiviranje rasuđivanja sposobnost i razvoj logičnog razumijevanja problema
  - Promocija uporabe logika i njene primjenjivosti za rješavanje problema u računarstvu
  - Osigurati osnovne elemente logičkih argumenta, analize i formulacije procesa, bitne u razumijevanju i primjeni računarstva



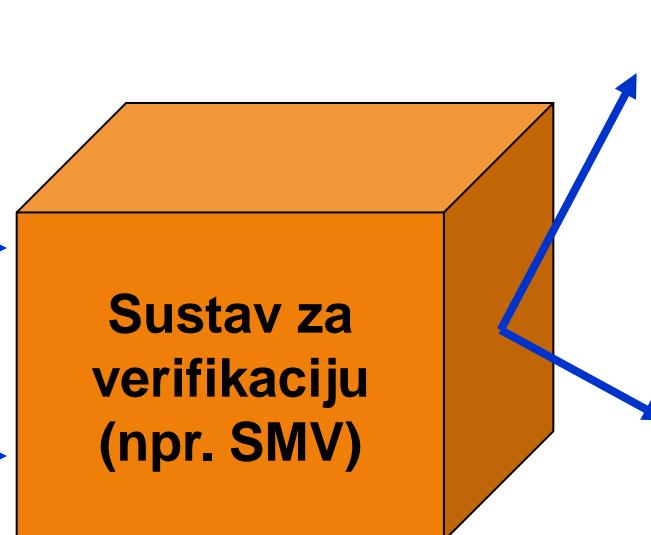
# Formalna verifikacija



- Formalna verifikacija programske potpore metodom provjere modela (engl. *Model checking*)
- Postupak provjere da *formalni model* izvedenog sustava (*I*), odgovara *formalnoj specifikaciji* (*S*) s matematičkom izvjesnošću

*I* = Implementacija (model sustava koji se verificira). Izraženo povezanim strojevima s konačnim brojem stanja (FSM).

*S* = Specifikacija (željeno ponašanje). Izraženo u vremenskoj logici.



$$I \neq S$$

**DA** = model sustava **logički zadovoljava** specifikaciju

**NE** - ispis pogrešnog izvođenja programa



# Formalna verifikacija



- Formalna verifikacija nastoji dokazati logičku zadovoljivost (tj. da model zadovoljava specifikaciju) te za njezinu primjenu su potrebna osnovna znanja iz područja:
  - Formalna (matematička) logika
    - posebice definicija “logičke zadovoljivosti” i sl.
  - Modeliranje implementacije strojevima s konačnim brojem stanja.
  - Izražavanje specifikacije (tj. željenog ponašanja) vremenskom logikom kao proširenjem klasične matematičke logike.

# Primjer

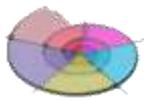
```
void merge (int a[], a_len, b[], b_len, *c)
{
 int i = 0, j = 0, k = 0;
 while (k < a_len+b_len) {
 if (a[i] < b[j]) {
 c[k] = a[i];
 i++;
 }
 else {
 c[k] = b[j];
 j++;
 }
 k++;
 }
}
```

*Specifikacija?*

*Program/Implementacija?*



# FORMALNA (MATEMATIČKA) LOGIKA



# Formalna (matematička) logika

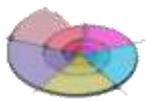


- Različiti tipovi logike se razlikuju po sadržaju svojih “primitiva”.
- Dva su temeljna pogleda na logiku:
  - *Ontološki*: Što postoji u svijetu.
  - *Epistemološki*: Kakvo je stanje znanja (što agent vjeruje).

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief 0...1
Fuzzy logic	degree of truth	degree of belief 0...1

*Klasična logika zasniva se na pojmu istinitosti.*

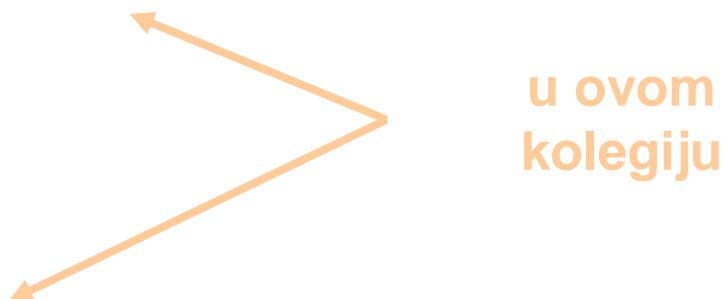
*naš interes*

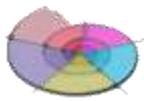


# Formalna (matematička) logika



- Logike su **formalni jezici** koji predstavljaju informaciju na način da se mogu automatizirano izvoditi zaključci.
- *Sintaksa* definira strukturu rečenice u jeziku.
- *Semantika* definira značenje rečenica (definira istinitost rečenice u svijetu u kojem ju promatramo).
- Postoji mnogo logika:
  - *Propozicijska i predikatna logika*
  - Logike višega reda
  - Modalne logike
    - Epistemička logika
    - *Vremenska logika*
    - ...
  - Opisna logika
  - Nemonotona logika
  - ...





# Formalna (matematička) logika



- Logika određuje postupke ispravnog rasuđivanja.

- Primjer 1:

Pretpostavka (premisa) 1: 1. Svaki čovjek je smrtan.

Pretpostavka (premisa) 2: 2. Sokrat je čovjek.

Zaključak: 3. Sokrat je smrtan.

(Ako su istinite rečenice 1 i 2, "logički slijedi" rečenica 3.)

- Primjer 2:

1. Svaki  $\alpha$  ima obilježje  $\beta$ .

2.  $\gamma$  je  $\alpha$ .

3.  $\gamma$  ima obilježje  $\beta$ .

■ **Zaključak 3** "Logički slijedi" samo na temelju oblika (forme), a ne na temelju sadržaja (konteksta).

■ **Matematička ili formalna logika** daje sustav zaključivanja u kojem je "logički izведен" zaključak barem tako dobar kao polazne pretpostavke.

■ Temelj: formalan sustav (definicija formalnog sustava slijedi kasnije).

■ Niti jedan **formalan sustav ne može osigurati istinite** polazne pretpostavke.



# Propozicijska logika



- Logika sudova, iskaza, tvrdnji
- engl. *propositional logic, propositional calculus*
- Sintaksa:
  - Logika iskaza preslikava deklarativne rečenice (koje mogu biti istinite ili lažne) u sustav simbola.
  - Npr.: "Sokrat je mudar." preslikava se u simbol P.
- Sustav propozicijske logike sastoji se od:
  - PS:            P, Q, ...     PS je prebrojiv skup atoma, simboličkih varijabli, simbola
  - Logički operatori (vezice):
    - $\neg$         (ne, not,  $\sim$ )                      negacija
    - $\wedge$         (i, and,  $\&$ )                      konjunkcija
    - $\vee$         (ili, or,  $|$ )                      disjunkcija
    - $\Rightarrow$       (ako, if,  $\supset$ ,  $\rightarrow$ )              implikacija
    - $\Leftrightarrow$     (akko, iff,  $\equiv$ ,  $\leftrightarrow$ )              ekvivalencija
    - Rezervirani simboli:
      - F            (false,  $\emptyset$ , 0,  $\perp$ )                      konstanta (neistinitost)
      - T            (true, 1)                              konstanta (istinitost)
      - $(\cdot)$ ,  $,$   $.$                                       znakovi zagrada, zareza i točke
  - Def. (rekurzivno) ispravno formiran složeni iskaz, ili formula (engl. *well-formed formula - wff* ) :
    - 1. Svaki atom je formula.
    - 2. Ako su P i Q formule, onda su formule:  $(\neg P)$ ,  $(\neg Q)$ ,  $(P \wedge Q)$ ,  $(P \vee Q)$ ,  $(P \Rightarrow Q)$ ,  $(P \Leftrightarrow Q)$ .



# Semantika

- Pridruživanje obilježja istinitosti (T, F) atomičkim simbolima = *Interpretacija*
  - $I: PS \rightarrow \text{BOOL}$

Gdje je  $\text{BOOL} = \{ T, F \}$ , tj. funkcija s kodomenom T ili F (istinito ili lažno).
- Semantika *dvaju složenih atomičkih simbola* prikazuje se istinitosnom tablicom.
  - $2 \text{ suda} = 2^2 = 4$  interpretacije,  $2^4 = 16$  istinitosnih tablica
- Neke važnije tablice istinitosti za povezivanje dva simbola:

	implikacija		ekvivalencija	kontradikcija tautologija		
	P	Q	$(P \Rightarrow Q)$	$(P \Leftrightarrow Q)$	$(\perp)$	T

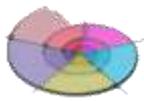
$I_1 :$	T	T	T	T	F	T
$I_2 :$	T	F	F	F	F	T
$I_3 :$	F	T	T	F	F	T
$I_4 :$	F	F	T	T	F	T



# Svojstva implikacije ( $P \Rightarrow Q$ )



- To je *materijalna implikacija* i nije potpuno intuitivna prirodnom jeziku. Namjera materijalne implikacije je modelirati *uvjetnu konstrukciju, (a ne uzročno-posljedičnu vezu)*, tj...:
- “ako P tada Q”, tj. ako je P istinit, tada je ( $P \Rightarrow Q$ ) istinito samo ako je Q istinito.
- Primjeri koji pokazuju *neintuitivni aspekt* materijalne implikacije:  
 $(2 + 2 = 4) \Rightarrow (\text{"Zagreb je glavni grad Hrvatske"})$
- Je istinita formula jer su prethodna (P) i posljedična (zaključna) (Q) tvrdnja istinite.  
 $(2 + 2 = 4) \Rightarrow (\text{"London je glavni grad Hrvatske"})$
- Je neistinita formula jer je posljedična tvrdnja (Q) neistinita.



# Svojstva implikacije ( $P \Rightarrow Q$ )

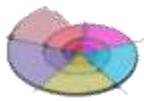


- Što je s formulama gdje je prethodna tvrdnja neistinita, a zaključna istinita ili neistinita:

$(2 + 2 = 5) \Rightarrow (\text{"Zagreb je } \text{glavni grad Hrvatske"})$

$(2 + 2 = 5) \Rightarrow (\text{"London je } \text{glavni grad Hrvatske"})$

- U prirodnom jeziku mogli bi ovakvima formulama implikacije pridijeliti bilo istinitost ili neistinitost, a možda čak i tvrditi da ako je prethodna tvrdnja ( $P$ ) neistinita, implikacija ne mora biti ni istinite ni neistinite.
- U formalnoj logici prihvaćena je **konvencija**:
- *Ako je  $P$  neistinit, tada je implikacija ( $P \Rightarrow Q$ ) istinita, neovisno o istinitosti  $Q$ .*



# Svojstva implikacije ( $P \Rightarrow Q$ )



- Zašto ima smisla implikaciju proglašiti istinitom ako je  $P$  neistinit ?
- Koje su moguće opcije:
- Za  $P = \text{istinito}$  suglasni smo s istinitosti implikacije
  - istinita ili neistinita ovisno o  $Q$ .
- Za  $P = \text{neistinito}$  postoje 4 moguće tablice:

P	Q	1:	2:	3:	4:	
<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>T</i>	<i>T</i>	
<i>F</i>	<i>T</i>	<i>F</i>	<i>T</i>	<i>F</i>	<i>T</i>	<i>što izabratи ?</i>
<i>T</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	<i>F</i>	
<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	<i>T</i>	nije sporno!

- Prva tri slučaja identična su drugim operacijam (1.- konjunkcija, 2. -  $Q$ , 3.- ekvivalencija).
  - Dakle preostaje jedino 4. tablica.

# Semantička pravila

- Izračunavanje istinitosti složene formule (evaluacija):
- Primjer 1:  $P_1, P_2$  istinite,  $Q_1, Q_2$  neistinite, a bilo koja formula (istinita ili ne).
- Istinite* su formule:

$\neg Q_1$   
 $(P_1 \wedge P_2)$   
 $(P_1 \vee A)$   
 $(A \vee P_1)$   
 $(A \Rightarrow P_1)$   
 $(Q_1 \Rightarrow A)$   
 $(P_1 \Leftrightarrow P_2)$   
 $(Q_1 \Leftrightarrow Q_2)$

*Neistinite* su formule:

$\neg P_1$   
 $(Q_1 \wedge A)$   
 $(A \wedge Q_1)$   
 $(Q_1 \vee Q_2)$   
 $(P_1 \Rightarrow Q_1)$   
 $(P_1 \Leftrightarrow Q_1)$   
 $(Q_1 \Leftrightarrow P_1)$   
*() - prazna formula*

- Primjer 2: Izračunavanja istinitosti složene formule  $(Q \vee (((\neg Q) \wedge P) \Rightarrow R))$ 
  - ima 3 propozicijska simbola  $P, Q, R$ :
- Interpretacija* (jedan od mogućih svjetova, ovdje  $2^3$  mogućih interpretacija)
- Neka je jedna interpretacija  $I$ :  $P=T, Q=F, R=F$ ,
  - izračunavanje (evaluacija) istinitosne vrijednosti daje formuli:
  - $(Q \vee (((\neg Q) \wedge P) \Rightarrow R))$  *neistinitu* vrijednost.
- Semantika uključuje interpretaciju i evaluaciju.**

# Pravila ekvivalencije

- Definicija: Dvije formule su semantički ekvivalentne ili jednake
  - ako imaju jednaku (istu) istinitosnu vrijednost za svaku interpretaciju I.
- Ekvivalencija u slijedećim pravilima može se provjeriti tablicom istinitosti za sve interpretacije. Provjera koincidencije istinitosnih tablica **nije u općem slučaju dovoljna**, ali definicija je ispravna.

$$(A \wedge \neg A) = ()$$

$$(\neg(\neg A)) = A$$

$$(A \wedge A) = A$$

$$(A \vee A) = A$$

$$(A \vee B) = (B \vee A)$$

$$(A \wedge B) = (B \wedge A)$$

$$((A \vee B) \vee C) = (A \vee (B \vee C))$$

$$((A \wedge B) \wedge C) = (A \wedge (B \wedge C))$$

$$(A \wedge (B \vee C)) = ((A \wedge B) \vee (A \wedge C))$$

$$(A \vee (B \wedge C)) = ((A \vee B) \wedge (A \vee C))$$

$$(\neg(A \vee B)) = ((\neg A) \wedge (\neg B))$$

$$(\neg(A \wedge B)) = ((\neg A) \vee (\neg B))$$

$$\underline{(A \Rightarrow B) = ((\neg A) \vee B)}$$

$$\underline{(A \Leftrightarrow B) = ((A \Rightarrow B) \wedge (B \Rightarrow A))}$$

$$\underline{(A \Rightarrow B) = ((\neg B) \Rightarrow (\neg A))}$$

kontradikcija

dvostruka negacija

jednaka važnost (idempotencija)

jednaka važnost

komutativnost

komutativnost

asocijativnost

asocijativnost

distributivnost

distributivnost

De Morganov zakon

De Morganov zakon

eliminacija uvjeta

eliminacija dvostrukog uvjeta

transpozicija

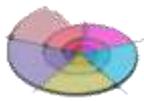
# Formalan sustav

- Definiramo formalan sustav kao dvojku:  $\{\Gamma, L\}$  gdje je
  - $\Gamma$  - konačan skup ispravno definiranih (formiranih) formula (wff)
  - $L$  – konačan skup pravila zaključivanja
- Neka temeljna **pravila zaključivanja** (jedan mogući skup  $L$ )
- Generiraju dodatne **istinite formule** (mehanički) bez razumijevanja konteksta (značenja).
  - Pogodna za strojnu primjenu.
  - Semantički korespondiraju sa semantikom "istinitosti".

Ako $P=T$ , $Q=T$ , generiraj $(P \wedge Q) = T$	(uvodenje konjunkcije)
Ako $P=T$ , $(P \Rightarrow Q)=T$ , generiraj $Q = T$	("modus ponens")
Ako $\neg Q=T$ , $(P \Rightarrow Q)=T$ , generiraj $\neg P$	("modus tolens")
Ako $(P \wedge Q)=T$ , generiraj $(Q \wedge P) = T$	(komutativnost $\wedge$ )
Ako $(P \wedge Q)=T$ , generiraj $P=T$ , $Q=T$	(eliminacija $\wedge$ )
Ako $P=T$ (odnosno $Q=T$ ), generiraj $(P \vee Q) = T$	(uvodenje disjunkcije)
Ako $[\neg(\neg P)]=T$ , generiraj $P=T$	(eliminacija negacije)

# Definicije obilježja u formalnom sustavu

- Sekvencija formula  $\{\omega_1, \omega_2, \dots, \omega_n\}$  ili pojedina formula  $\omega_i$  je
  - *teorem (dokaz, dedukcija)* iz skupa formula  $\Gamma$ , ako je u skupu  $\Gamma$ ,
  - ili se može izvesti iz  $\Gamma$  korištenjem pravila zaključivanja  $L$ .
  - $\Gamma \vdash_L \{\omega_1, \omega_2, \dots, \omega_n\}$  sekvencija formula je teorem
  - $\Gamma \vdash_L \omega_i$  formula  $\omega_i$  je teorem
- Zarez označava konjunkciju
- Npr. (skup  $\Gamma$  sadrži dvije *istinite* formule):  $\Gamma = \{ P, (P \Rightarrow Q) \}$ 
  - Korištenjem pravila "Modus ponens" (iz skupa dopustivih pravila  $L$ ), izvodimo da je istinita nova formula  $Q$ ,
  - te je ta formula  $Q$  *teorem (dokaz, dedukcija) skupa  $\Gamma$* .
- Skup  $\Gamma$  je *konzistentan* akko ne sadrži formule na temelju kojih bi  $\omega_i$  i  $\neg\omega_i$  (istovremeno) bili teoremi.
  - $\Gamma = \{ P, (P \Rightarrow Q) \}$  je *konzistentan*.
- Primjer:
- $\Gamma = \{ P, \neg P, (P \Rightarrow Q) \}$  je *nekonzistentan ili kontradiktoran* jer su  $P$  i  $\neg P$  istovremeno teoremi (nalaze se u samom skupu  $\Gamma$ ).
- $\Gamma = \{ P, \neg Q, (P \Rightarrow Q) \}$  je *nekonzistentan* jer sadrži  $\neg Q$ , a pravilom "Modus ponens" može se izvesti  $Q$ , dakle  $\neg Q$  i  $Q$  bi istovremeno bili teoremi.



# Obilježja u formalnom sustavu



- Za formalni sustav  $\{\Gamma, L\}$  izvodi se teorem  $\omega_i$ , tj. tražimo odgovor da li je  $\omega_i$  teorem ili ne.
- Formalni sustav  $\{\Gamma, L\}$  je:
  - *odrediv* ili odlučljiv (engl. *decidable*), akko postoji algoritam koji će u konačnom vremenu odrediti ili ne teorem  $\omega_i$  (dati u konačnom vremenu dati odgovor da li teorem  $\omega_i$  postoji ili ne).
  - *poluodrediv* ili poluodlučljiv (engl. *semidecidable*), akko postoji algoritam koji će u konačnom vremenu odrediti teorem ako on postoji.
    - Algoritam završava u konačnom vremenu s odgovorom DA (za teorem  $\omega_i$ ),
    - ali ne mora završiti u konačnom vremenu s odgovorom NE (ako  $\omega_i$  nije teorem).
  - *neodrediv* ili neodlučljiv (engl. *undecidable*) ako nije odrediv ni poluodrediv.



# Interpretacija i evaluacija



- Semantika u formalnom sustavu povezana je s
  - *interpretacijom (pridruživanjem)* istinitosti atomima) i
  - *evaluacijom (izračunavanjem)* istinitosti složene formule).
- Neka *interpretacija je model* formalnog sustava ako evaluira *sve njegove formule u istinito* (vrijedi i za svaku formulu pojedinačno).
  - Npr.: interpretacija I:  $\{P=T, Q=F, R=F\}$  formule  $(Q \vee (((\neg Q) \wedge P) \Rightarrow R))$  *nije model*
    - jer ta interpretacija formuli daje neistinu vrijednost.
- Skup formula je *zadovoljiv* (engl. *satisfiable*) ako ima model (*barem jedan*).
  - Vrijedi i za pojedinačne formule. (SAT problem (zadovoljivost) - temeljni NP problem!!)
- Sukladno ranijoj definiciji, *nezadovoljiv* (nekonzistentan, kontradiktoran) skup formula *nema nijedan model*.
- Skup formula  $\Gamma$  *implicira* ili *povlači* (engl. *entails*) formulu  $\omega$ , ako je *svaki model* od  $\Gamma$  ujedno i model od  $\omega$ .
  - Formula  $\omega$  je tada *logička posljedica* skupa formula  $\Gamma$ .
  - $\Gamma \models \omega$  (svaki model od  $\Gamma$  je model formule  $\omega$ )
- Formula je *valjana* ili *tautologija* (engl. *valid*)
  - ako je istinita za svaku interpretaciju i evaluaciju.
  - $\models \omega$  (svaka interpretacija je model formule  $\omega$ )



# Primjeri logičkih posljedica



- Svaka interpretacija koja lijevoj strani od znaka  $\models$  daje istinitost mora i desnoj strani dati istinitost.
- $(P \wedge Q) \models P$   
Lijeva strana = T samo za ( $P=T, Q=T$ ), a to daje i desnoj strani =T
  - gornji izraz vrijedi ( $P$  je *logička posljedica*  $(P \wedge Q)$ ).
- $(P \vee Q) \models P$   
Lijeva strana je istinita za ( $P=F, Q=T$ ;  $P=T, Q=F$ ;  $P=T, Q=T$ ), ali desna za interpretaciju ( $P=F, Q=T$ ) nije istinita, te  $P$  *nije logička posljedica*  $(P \vee Q)$ .
- $\{\neg Q, (P \vee Q)\} \models P$       (zarez predstavlja konjunkciju  $\wedge$ )  
Skup  $\Gamma$  na lijevoj strani je istinit samo za  $Q=F, P=T$ , a to daje istinitost i desnoj strani, te je  $P$  *logička posljedica* navedenog skupa  $\Gamma$ .
- $P \models (Q \vee \neg Q)$   
Također vrijedi, jer za svaku interpretaciju za koju je lijeva strana istinita ( $P=T$ ) i desna stana je istinita (desna strana je uvijek istinita).



# Primjeri logičkih posljedica



$$\Gamma = (A \vee C) \wedge (B \vee \neg C) = \text{Knowledge Base} = \text{KB}$$

- dvije konjunkcijom povezane formule (umjesto  $\wedge$  može se koristiti zarez).

Neka je:  $\alpha = (A \vee B)$

KB  $\models \alpha$  ?

$A$	$B$	$C$	$A \vee C$	$B \vee \neg C$	$KB$	$\alpha$
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>



# Ispravnost i kompletnost form. sust.



- Formalan sustav  $\{\Gamma, L\}$  je *ispravan* (engl. *sound*)
  - ako  $\Gamma \models \omega_i$  kadgod je  $\Gamma \vdash L \omega_i$ ,
  - tj. svaka pravilima dokazana formula je ujedno i logička posljedica skupa  $\Gamma$ .
$$\Gamma \vdash L \omega_i \text{ implicira } \Gamma \models \omega_i$$
- Formalan sustav  $\{\Gamma, L\}$  je *kompletan* (engl. *complete*)
  - ako  $\Gamma \vdash L \omega_i$  kadgod je  $\Gamma \models \omega$ ,
  - tj. svaku logičku posljedicu skupa  $\Gamma$  moguće je dokazati pravilima  $L$ .
$$\Gamma \models \omega \text{ implicira } \Gamma \vdash L \omega_i$$
- U *ispravnom i kompletном* formalnom sustavu  $\{\Gamma, L\}$  vrijedi:
$$\Gamma \models \omega = \Gamma \vdash L \omega_i$$
- Većina interesantnih formalnih sustava je nekompletno, a vrlo malo ih je određivo.
- *Propozicijska logika je ispravna, kompletna i odrediva* (npr. preslikavanjem u tablicu istinitosti), jer operira s *konačnim* skupom simbola.

# Proriteti operatora

- Primjeri:
- Prioritet logičkih operatora:
- Najviši:



$\neg$	negacija
$\wedge$	konjunkcija
$\vee$	disjunkcija
$\Rightarrow$	implikacija
$\Leftrightarrow$	ekvivalencija

- Najniži:

- Formule:

1.  $P$
2.  $(P \vee \neg P)$
3.  $(P \wedge \neg P)$
4.  $\emptyset$
5.  $P \Rightarrow (Q \Rightarrow P)$
6.  $(P \wedge Q)$

## Obilježja:

zadovoljiva ali ne i valjana (interpretacija  $P=T$  je model, dok interpretacija  $P=F$  nije model).

valjana (tautologija), sve interpretacije (dvije)  $P=T$ ,  $P=F$ , su modeli (formula je istinita).

kontradiktorna (nezadovoljiva), nema modela.

*kontradiktorna (nezadovoljiva).*

valjana (tautologija), sve interpretacije (ima ih 4: FF, FT, TF, TT) su modeli.

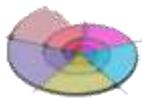
zadovoljiva. Ima samo jedan model:  $P=T$ ,  $Q=T$ .



# Semantička ekvivalencija



- **Ranija definicija:** Dvije formule su semantički *ekvivalentne* ili *jednake*
  - ako imaju *jednaku (istu) istinitosnu vrijednost za svaku interpretaciju I.*
- Npr. Da li su ekvivalentne dvije formule:  $((P \wedge Q) \Rightarrow P)$  i  $(R \vee \neg R)$ ?
  - *DA!* sukladno gornjoj definiciji (obje su valjane -- tautologije), ali usporedba istinitosnih tablica nema smisla (simboli i tablice su različite).
- *Definicija ekvivalencije uporabom pojma logičke posljedice ( $\Vdash$ )*
- Ako dvije ekvivalentne formule imaju jednaku istinitosnu vrijednost za svaku interpretaciju, može se definirati:
- *Dvije formule  $\alpha$  i  $\beta$  su semantički ekvivalentne (oznake  $(\alpha \Leftrightarrow \beta)$  ili  $(\alpha \equiv \beta)$ ) akko vrijedi:  $(\alpha \Vdash \beta)$  i  $(\beta \Vdash \alpha)$ .*
- Ranija tablica pravila ekvivalencije daje:  $(\alpha \Leftrightarrow \beta) = (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$ .
  - Ako su  $\alpha$  i  $\beta$  ekvivalentne, formula  $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$  mora biti **uvijek** istinita:  
 $\Vdash (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
  - *Semantička* ekvivalencija je na taj način identična *dokazivoj* ekvivalenciji.
- Ako želiš dokazati ekvivalentnost, dokaži da je  $((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$  tautologija, tj.  
*da je njena negacija nezadovoljiva*
  - Primjer ekvivalentne formule:  $((P \wedge Q) \Rightarrow R) \Leftrightarrow (P \Rightarrow (Q \Rightarrow R))$



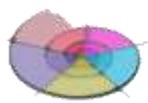
# Teorem dedukcije

- Dokazivanje logičke posljedice preko (ne)zadovoljivosti
- Formula  $\psi$  je *logička posljedica* formule  $\varphi$ , tj.  $\varphi \models \psi$ , akko je formula  $(\varphi \Rightarrow \psi)$  tautologija (valjana).
- Dokaz:
- Akko je  $(\varphi \Rightarrow \psi)$  tautologija, onda iz tablice za implikaciju proizlazi da kada je  $\varphi$  istinit i  $\psi$  mora biti istinit. (To je upravo definicija logičke posljedice.)

$\varphi$	$\psi$	$(\varphi \Rightarrow \psi)$
F	F	T
F	T	T
T	F	F
T	T	T

- Budući da  $(\varphi \Rightarrow \psi)$  mora bit tautologija, to njena negacija  
 $\neg(\varphi \Rightarrow \psi) = \neg(\neg\varphi \vee \psi) = (\varphi \wedge \neg\psi)$  mora biti nezadovoljiva. Dakle:

■  $\varphi \models \psi$  akko je  $(\varphi \wedge \neg\psi)$  nezadovoljiva



# Primjena teorema dedukcije



- Dokazivanje obaranjem - 1
- $\varphi$  možemo zamisliti kao konjunkciju istinitih formula (bazu formula, bazu znanja).
  - To su početne pretpostavke – aksiomi nekog problema.

Npr. neka  $\varphi$  predstavlja skup istinitih formula (povezanih konjunkcijom):

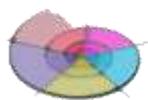
$$\begin{aligned} A_1 \wedge \\ A_2 \wedge \\ \dots \wedge \\ A_p \end{aligned}$$

*Tražimo dokaz:*

*Da li je neka formula  $\psi$  logička posljedica skupa formule danim sa  $\varphi$  ?*

Teorem dedukcije kaže: Ako želimo dokazati da je neka formula  $\psi$  logička posljedica formule  $\varphi$ , moramo dokazati *nezadovoljivost* formule  $(\varphi \wedge \neg\psi)$ .

- Dakle formulu  $\psi$  negiramo i dodamo formulama  $\varphi$ , te uporabom dopustivih pravila pokušavamo pokazati kontradiktornost (nezadovoljivost) u  $(\varphi \wedge \neg\psi)$ .
- *Nezadovoljivost možemo dokazati ako primjenom pravila uspijemo generirati praznu formulu "( )"* (jer je ona sigurno nezadovoljiva).



# Normalni oblici logičkih formula



- Svaka propozicijska formula može se preslikati (ekvivalentna je) formuli u *disjunkcijskom normalnom obliku (DNF)* :

$$\blacksquare (k_1_1 \wedge \dots \wedge k_1_n) \vee (k_2_1 \wedge \dots \wedge k_2_m) \vee \dots \vee (k_p_1 \wedge \dots \wedge k_p_r)$$

- Svaka propozicijska formula može se preslikati (ekvivalentna je) formuli u *konjunkcijskom normalnom obliku (CNF)* :

$$\blacksquare (k_1_1 \vee \dots \vee k_1_n) \wedge (k_2_1 \vee \dots \vee k_2_m) \wedge \dots \wedge (k_p_1 \vee \dots \vee k_p_r)$$

- *CNF* = konjunkcija klauzula

- Gdje su:

- $k_i$  = *literal* (negirani ili nenegirani atomički simbol - atom)
- *klauzula* = disjunkcija literala. Npr.:  $(k_2_1 \vee \dots \vee k_2_m)$

# Konverzija propozicijske formule u CNF oblik

- Svaka formula u propozicijskoj logici može se preslikati u konjunkciju klauzula (CNF):
- Npr:  $\neg(P \Rightarrow Q) \vee (R \Rightarrow P)$ 
  1. Eliminiraj implikaciju uporabom ekvivalentnog " $\vee$ " oblika:  
 $\neg(\neg P \vee Q) \vee (\neg R \vee P)$
  2. Ograniči doseg negacije (pomak u desno) uporabom DeMorganovih pravila, te eliminiraj dvostrukе negacije:  
 $(P \wedge \neg Q) \vee (\neg R \vee P)$
  3. Pretvori u CNF asocijativnim i distribucijskim pravilima:  
 $(P \vee \neg R \vee P) \wedge (\neg Q \vee \neg R \vee P),$
- Te dalje:  
 $(P \vee \neg R) \wedge (\neg Q \vee \neg R \vee P) = \text{CNF oblik}$



# Postupci pojednostavljivanja klauzula



- 1. Uporaba temeljnih pravila:
  - Npr.:

$(P \vee R \vee P)$  pojednostavi (spoji) u  $(P \vee R)$

$(P \vee \neg P \vee Q)$  izostavi cijelu jer je evidentno valjana (T)
- 2. Podrazumijevanje (*engl. subsumption*) *klauzula*
  - Klauzula  $\omega_1$  podrazumijeva klauzulu  $\omega_2$ , ako su literali u  $\omega_1$  podskup literalova u  $\omega_2$ .
  - Npr.:

$(P \vee R)$  podrazumijeva klauzule  $(P \vee R \vee Q)$



# Usporedba CNF i DNF oblika



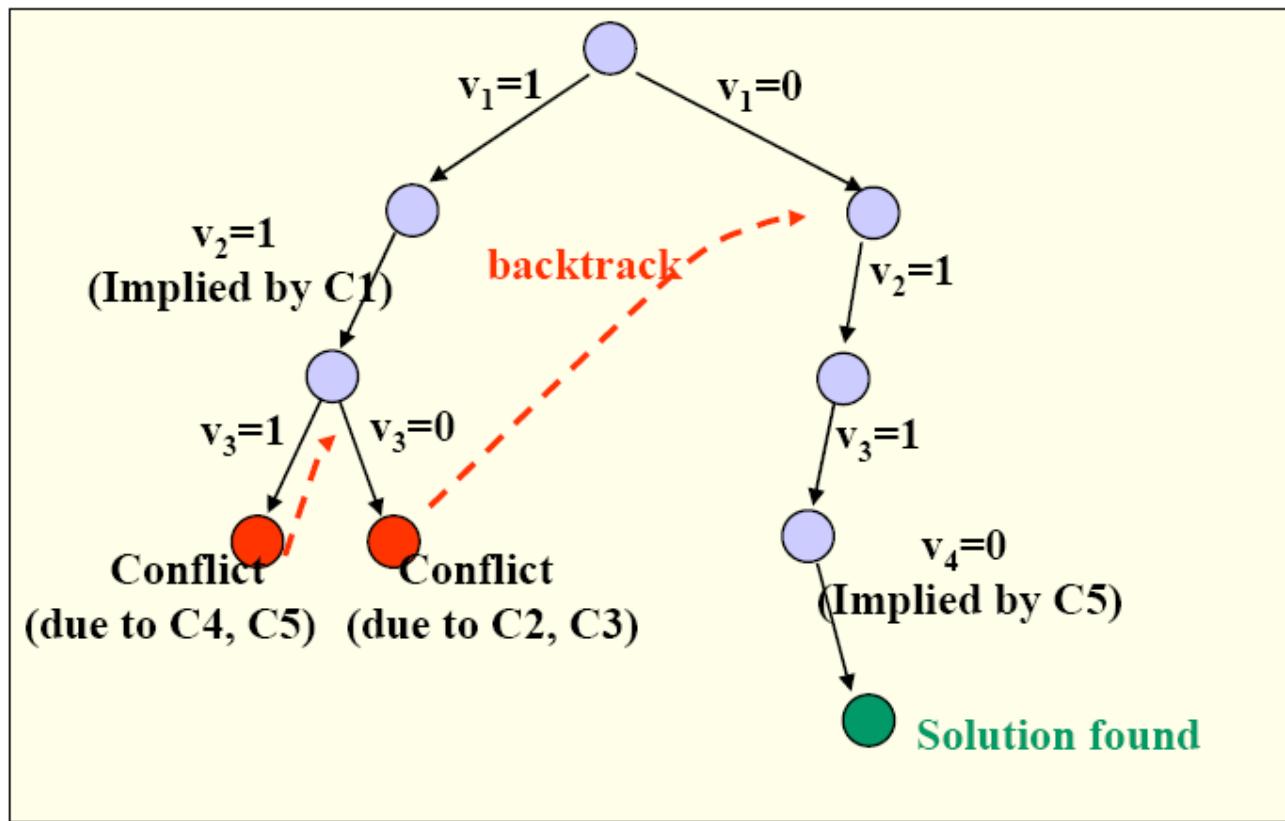
- Mogu dati brze odgovore na česta pitanja:
- DNF nam govori da li je **formula zadovoljiva**.
  - Ako su sve *disjunkcije neistinite* ( $\perp$ ), ili sve sadrže komplementarne literale (npr.  $(A \wedge \neg A)$ ) ne postoji niti jedan model za taj DNF. Inače je formula zadovoljiva.
- CNF nam govori da li je ili nije **formula tautologija** (valjana, uvijek istinita).
  - Ako sve klauzule sadrže istinitost ( $T$ ) ili sve sadrže komplementarne literale (npr.  $(A \vee \neg A)$ ) formula je tautologija.
  - Inače, za formulu postoji barem jedna interpretacija (pridruživanje istinitosti atomičkim simbolima) koja nije model (ne zadovoljava formulu) pa formula nije tautologija.
- Preslikavanje iz CNF u DNF oblik i obrnuto je računalno vrlo skupo (vremenski i prostorno)

# SAT problem

- Problem zadovoljivosti - temeljni NP problem
- Tražimo model skupa formula  $\Gamma$  - interpretaciju koja evaluira **sve** formule u skupu  $\Gamma$  u istinito.
  - To je ekvivalentno traženju modela *jedne* složene formule koja se sastoji iz *konjunkcije svih formula u  $\Gamma$* .
- $\Gamma$  skup formula je najčešće dan u **CNF** obliku:  
$$(k_1 \vee \dots \vee k_p) \wedge (k_2 \vee \dots \vee k_r) \wedge \dots \wedge (k_s \vee \dots \vee k_t)$$
- Iscrpna procedura rješavanja **CNF** SAT problema sistematski pridjeljuje istinitosne vrijednosti atomičkim propozicijskim simbolima.
  - Za  $n$  atoma  $2^n$  pridruživanja.
  - Eksponencijalna složenost, računalno neizvedivo u općem slučaju.
- Za **DNF – polinomska složenost** jer postoji konačan broj literala, a dovoljno je pronaći zadovoljivost u samo jednom disjunkcijskom članu.
- Cnf 2SAT      - polinomska kompleksnost (do 2 literala u klauzuli)
- **CNF 3SAT**      - **NP kompletno** (3 literali u klauzuli)
- *Zadovoljivost formule u CNF obliku s 3 i više literali je NP kompletno.*
- Mnogi stohastički algoritmi troše eksponencijalno vrijeme u najgorem slučaju, ali polinomsko u srednjem (očekivanom).

# Primjer: Donošenja odluke o zadovoljivosti

Conjunctive Normal Form (CNF)  
CNF Clause Literal

$$(\bar{v}_1 + v_2)(\bar{v}_1 + v_3 + v_4)(\bar{v}_2 + v_3 + \bar{v}_4)(\bar{v}_1 + \bar{v}_3 + v_4)(\bar{v}_2 + \bar{v}_3 + \bar{v}_4)$$




# Primjena teorema dedukcije



- **Dokazivanje SAT rješavačem**
- Neka *istinite* formule predstavljaju skup  $\Gamma$ :
  1.  $P$
  2.  $(P \Rightarrow Q)$
  3.  $(Q \Rightarrow S)$
- U CNF obliku:  $\Gamma = [ (P) \wedge (\neg P \vee Q) \wedge (\neg Q \vee S) ]$
- Da li je neka formula  $S$  logička posljedica skupa  $\Gamma$ :  $\Gamma \models S$  ?
- Prema teoremu dedukcije:
  1.  $S$  je logička posljedica  $\Gamma$  ako je  $(\Gamma \wedge \neg S)$  nezadovoljiva.
  2. Skupu  $\Gamma$  **dodajemo negaciju formule** koju želimo dokazati ( $\neg S$ ):  
 $[ (P) \wedge (\neg P \vee Q) \wedge (\neg Q \vee S) \wedge (\neg S) ]$
  3. Sat sustavom pokušamo naći bar jedan model (zadovoljivost).  
Ako SAT sustav pokaže da formulu **nije moguće zadovoljiti** (nema modela), zaključujemo:  
*S je doista logička posljedica skupa  $\Gamma$ .*

# PREDIKATNA LOGIKA



# PREDIKATNA LOGIKA



- Logika predikata prvoga reda – *FOPL* (engl. *predicate logic, predicate calculus, first order predicate logic*)
  1. P: Svi ljudi su smrtni.
  2. Q: Sokrat je čovjek.
  3. R: Sokrat je smrtan.
- U propozicijskoj logici nikako se iz 1 i 2 ne može zaključiti 3.
- Fopl uvodi objekte, relacije, obilježja, funkcije
  - pobliži opis izjave

# Sintaksa predikatne logike

## ■ Atomički predikat:

- pred\_simbol: osnovno obilježje u rečenici (predikat)
- $t_i$  = članovi: objekti ili odnosi u rečenici
- dva načina zapisa:
  - (pred\_simb  $t_1 t_2 \dots t_n$ ) – infiks notacija (LISP)
  - pred\_simb( $t_1 t_2 \dots t_n$ ) – prefiks notacija (Prolog)

1. Konstanta je član.

2. Varijabla je član.

3. Ako je fun\_simb funkcijski simbol sa  $n$ -argumenata, a  $t_1, t_2, \dots, t_n$  su članovi, tada je (fun\_simb  $t_1 t_2 \dots t_n$ ) član.

## ■ Logički operatori (vezice): $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

## ■ Članovi ( $t_i$ ):

- Konstante: objekti u nekom svijetu (blok1, sokrat, ...).
- Rezervirane konstante: T, F.
- Varijable: razred objekata ili obilježja; mogu poprimiti vrijednosti iz svoje domene;
  - (Npr.: X, Y, ...).

■ **Kvantifikacijski simboli** (uz varijable, pobliže određuju istinitost rečenice):

- $\exists$  (postoji, za\_neki, exist) - egzistencijski ili partikularni kvantifikator (barem jedan).
- $\forall$  (za\_svaki, svi, for\_all) - univerzalni kvantifikator (svi), ima središnju ulogu u izražavanju generalizacije.

## ■ Funkcije:

- veza između objekata - (fun\_simb  $t_1 t_2 \dots t_n$ )
- Npr.: (cos X), (otac\_od abel kain)

## ■ Formalna def. člana:

# Ispравно definiran složeni predikat ili formula

- 1. svaki atomički predikat je formula.
- 2. ako je  $S_i$  formula, tada su formule:  
 $(\neg S), (S_1 \wedge S_2), (S_1 \vee S_2), (S_1 \Rightarrow S_2), (S_1 \Leftrightarrow S_2)$ .
- 3. ako je  $X$  varijabla, a  $S$  formula, tada su formule:  $\exists X S(X), \forall X S(X)$ .  
(oznaka  $S(X)$  = formula  $S$  u kojoj postoji varijabla  $X$ )
- Negirani ili nenegirani atomički predikat naziva se *literal*.
- *Dopuna pravilima ekvivalencije* ( $P(X), Q(X)$  su wff s varijablom  $X$ ):  
 $\exists X P(X) = \exists Y P(Y)$  - simbol varijable nije bitan, ali je bitan  
 $\forall X P(X) = \forall Y P(Y)$  doseg, uvijek unutar jedne formule
- $(\neg(\forall X P(X))) = \exists X (\neg P(X))$  - analogno De Morgan - Negacija mijenja kvantifiaktor  
 $(\neg(\exists X Q(X))) = \forall X (\neg Q(X))$
- Primjer ispravno definirane složene formule u infiks notaciji :  
 $(\forall X \forall Y (((otac X Y) \vee (majka X Y)) \Rightarrow (roditelj X Y)))$

# Pravila ekvivalencije

- $P(X)$  je ispravno definirana formula s varijablom  $X$ .
- Neka je okvir razmatranja (domena  $X$ ):  $U = \{1, 2, 3\}$
- Formula  $\forall X P(X)$  je ekvivalentna  $[ P(1) \wedge P(2) \wedge P(3) ]$   
$$\forall X P(X) \equiv [ P(1) \wedge P(2) \wedge P(3) ]$$
  - $\forall X P(X)$  je istinita ako su stinite **sve** supstitucije iz domene.
- Formula  $\exists X P(X)$  je ekvivalentna  $[ P(1) \vee P(2) \vee P(3) ]$   
$$\exists X P(X) \equiv [ P(1) \vee P(2) \vee P(3) ]$$
  - $\exists X P(X)$  je istinita ako je istinita **bar jedna** supstitucija



# Dopuna pravila ekvivalencije



- 1. Univerzalni kvantifikator  $\forall$
- $X$  – šahovska figura

$\forall X P(X)$  – "Sve figure su bijele." ,     $\forall X Q(X)$  – "Sve figure su crne."

$$\forall X P(X) \wedge \forall X Q(X) = \forall X (P(X) \square Q(X))$$

$\forall x \Phi(x)$  je konjunkcija  $\Phi(d)$  svih elemenata  $d$  iz domene.

- Za disjunkciju to ne vrijedi:

$$\forall X P(X) \vee \forall X Q(X) \neq \forall X (P(X) \square Q(X)) \quad - \text{nije ekvivalentno}$$

- 2. Egzistencijski kvantifikator  $\exists$

$\exists X P(X)$  – "Netko voli plivanje.",  $\exists X Q(X)$  – "Netko voli tenis."

$$\exists X P(X) \vee \exists X Q(X) = \exists X (P(X) \square Q(X))$$

$\exists x \Phi(x)$  je disjunkcija supstitucija instancije  $\Phi$ .

- Za konjunkciju to ne vrijedi:

$$\exists X P(X) \square \exists X Q(X) \neq \exists X (P(X) \square Q(X)) \quad - \text{nije ekvivalentno}$$

# Primjer

- Neka su okvir razmatranja: cijeli brojevi.
- Neka  $P(X)$  znači  $X > 5$ ,  $Q(X)$ :  $X < 3$
- $(\exists X P(X)) \wedge (\exists X Q(X))$  znači:  
“Postoji broj koji je veći od 5 i postoji broj koji je manji od 3.”  
To je istinito je npr. broj 6 daje istinitost za prvu izjavu, a broj 2 za drugu.
- $\exists X (P(X) \wedge Q(X))$  znači:  
“Neki broj je veći od 6 i (isti broj) manji od 2.”  
To je naravno neistinito.

Slijedi:

$$(\exists X P(X)) \wedge (\exists X Q(X)) \neq \exists X (P(X) \wedge Q(X))$$

Formule nisu ekvivalentne.

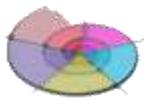
# Dopuna pravila ekvivalencije

$$\forall x(P(x) \wedge Q(x)) \equiv \forall xP(x) \wedge \forall xQ(x)$$

$$\exists x(P(x) \vee Q(x)) \equiv \exists xP(x) \vee \exists xQ(x)$$

$$\forall x(P(x) \vee Q(x)) \not\equiv \forall xP(x) \vee \forall xQ(x)$$

$$\exists x(P(x) \wedge Q(x)) \not\equiv \exists xP(x) \wedge \exists xQ(x)$$



# Dopuna pravila ekvivalencije



- Permutacija kvantifikatora

- Formule :

$$\forall X \exists Y P(X,Y) \neq \exists Y \forall X P(X,Y)$$

- Nisu ekvivalentne !

- Primjer:

- Neka  $P(X, Y)$  znači: "Y je majka od X."

- $\forall X \exists Y P(X,Y)$  znači: "Svatko ima majku."

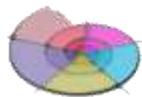
- $\exists Y \forall X P(X,Y)$  znači: " Postoji osoba koja je majka svima."



# Semantika predikatne logike



- Skup ispravno definiranih složenih predikata ili formula (engl. *well-formed formula - wff*) odnosi se na neku domenu razmatranja D.
- Interpretacija I je proces preslikavanja elemenata iz domene D svakoj pojedinoj konstanti, varijabli, i funkciji, te atomičkom predikatu, tako da:
  - Simbolu T uvijek je pridružena istinita vrijednost.
  - Simbolu F uvijek je pridružena neistinita vrijednost.
  - Svakoj konstanti pridruži se jedan element iz D.
  - Svakom funkcijском simbolu pridruži se jedan element iz D.
  - Svakoj varijabli se pridruži neprazan podskup iz D (dozvoljene supstitucije).
- Svaka funkcija f, sa m argumenata, definira *interpretacijom i evaluacijom* preslikavanje iz  $D^m$  u D, tj.:  $f: D^m \rightarrow D$  (pridruživanje jednog elementa iz D).
- Svaki predikat P, s brojem članova n, definira *interpretacijom i evaluacijom* svojih članova preslikavanje iz  $D^n$  u  $\{T, F\}$ , tj.  $P: D^n \rightarrow \{T, F\}$  (istinito ili ne).
- Vrijednosti wff formula složenih logičkim operatorima date su odgovarajućim istinitosnim tablicama.
- Vrijednost  $\forall X P(X)$  je T, ako  $P(X)$  je T, za sve vrijednosti X date sa I, a F inače.
- Vrijednost  $\exists X P(X)$  je T, ako  $P(X)$  je T, barem za jednu vrijednost X danoj sa I, a F inače.



# Semantika predikatne logike



- Određivanje istinitosti wff svodi se na *interpretaciju + evaluaciju*
- Primjeri pridruživanja istinitosti:
  - 1. (*prijatelj ivan ana*)
    - predikat je T, ako u D postoji objekt Ana koja je prijatelj Ivanu.
  - 2. X je domena prirodnih brojeva
    - $\forall X (\text{veci } X \ 10)$  atomički predikat je F
    - $\exists X (\text{veci } X \ 10)$  atomički predikakt je T
- $\forall$  - u određivanju istinitosti potrebne sve supstitucije varijable
  - (problem ako je domena beskonačna)
- $\exists$  - u određivanju istinitosti potrebna je jedana supstitucija za koju vrijedi T
  - (problem ako je domena beskonačna i predikat F)
- Skup svih istinitih predikata iz domene D = *stanje svijeta*
  - engl. *state of the world*

# Primjeri preslikavanja

- Preslikavanja prirodnog jezika u formule predikatne logike:

1. *“Nitko nije savršen.” (infiks notacija)*

- $\neg \exists X (\text{savršen } X)$ , ili
- $\forall X (\neg (\text{savršen } X))$ , tj. "svi su nesavršeni"

2. *“Svi košarkaši su visoki.” (infiks notacija)*

- $\forall X ((\text{kosarkas } X) \Rightarrow (\text{visok } X))$
- "za svaki  $X$  u domeni razmatranja vrijedi da ako je  $(X)$  košarkaš tada je visok"

■ *Ispravna uporaba univerzalnog kvantifikatora  $\forall$  (1)*

Neka je okvir razmatranja (skup objekata): { Garfield, Feliks, računalo }

Preslikaj u pred. logiku: "Sve mačke su sisavci."

Za sve objekte u okviru razmatranja vrijedi: ako su mačke tada su sisavci.

$\forall x [ \text{mačka}(x) \Rightarrow \text{sisavac}(x) ]$  (prefiks notacija)

# Ispравна uporaba univerzalnog kvantifikatora

- $\forall x [ \text{mačka}(x) \Rightarrow \text{sisavac}(x)]$  (vrijedi za sve objekte  $x$ )
- Dokaz: Supstitucija svih objekata u formulu (konjunkcija formula jer  $\forall$ ):  
 $[ \text{mačka(Garfield)} \Rightarrow \text{sisavac(Garfield)}] \wedge [ \text{mačka(Feliks)} \Rightarrow \text{sisavac(Feliks)}] \wedge [ \text{mačka(računalo)} \Rightarrow \text{sisavac(računalo)}] \wedge \dots$  (ostali objekti ako ih ima)  
prva [ ]: T (vidi tablicu za  $\Rightarrow$ )  
druga [ ]: T (vidi tablicu za  $\Rightarrow$ )  
treća  $[F \Rightarrow T] =$  T (vidi tablicu za  $\Rightarrow$ ) pa je i treća formula =T !!!!!
- Ako bi preslikali:  $\forall x [ \text{mačka}(x) \wedge \text{sisavac}(x)]$
- Supstitucija svih objekata daje:  
 $[ \text{mačka(Garfield)} \wedge \text{sisavac(Garfield)}] \wedge [ \text{mačka(Feliks)} \wedge \text{sisavac(Feliks)}] \wedge [ \text{mačka(računalo)} \wedge \text{sisavac(računalo)}] \wedge \dots$  (ostali objekti ako ih ima)
- $\text{Mačka(računalo)} = F$  - daje neistinitu cijelu formulu !!

# Ispравна uporaba egzistencijskog kvantifikatora $\exists$



- Neka je okvir razmatranja (kao i prije): { Garfield, Feliks, računalo }
- Preslikaj u predikatnu logiku: "Garfield ima brata koji je mačka."
- Postoji **barem jedan** (neki) objekt i takav da su mu obilježja istinita.  
 $\exists x [brat(x, \text{Garfield}) \wedge \text{mačka}(x)]$
- Dokaz supstitucijom svih objekata u formulu (disjunkcija formula jer  $\exists$ ):  
  
[brat(Garfield, Garfield)  $\wedge$  mačka(Garfield)]  $\vee$   
[brat(Feliks, Garfield)  $\wedge$  mačka(Feliks)]  $\vee$   
[brat(računalo, Garfield)  $\wedge$  mačka(računalo)]  $\vee$  ... (ostali ako ih ima)
- Prva [ ] neistinita, ali idemo dalje jer su [ ... ] povezane disjunkcijom.
- Drugi red istinit, **cijela formula je istinita** (dalje ne moramo ispitivati).

# Ispравна uporaba egzistencijskog kvantifikatora $\exists$



- Ako bi preslikali:  $\exists x [brat(x, Garfield) \Rightarrow mačka(x)]$
- Supstitucija svih objekata u disjunkciju formula daje:  
 $[brat(Garfield, Garfield) \Rightarrow mačka(Garfield)] \vee$   
 $[brat(Feliks, Garfield) \Rightarrow mačka(Feliks)] \vee$   
 $[brat(računalo, Garfield) \Rightarrow mačka(računalo)] \vee \dots$  (ostali objekti ako ih ima)
- Implikacija je istinta ako je atomički izraz na lijevoj strani neistinit !
- Npr. ako je:  $[brat(računalo, Garfield) \Rightarrow mačka(računalo)]$  istinito,
  - cijela je formula istinita !!
- Egzistencijski kvantificirana implikacijska formula je istinita ako
  - u okviru razmatranja postoji barem jedan objekt
  - za koji je premla implikacije neistinita (desna strana može biti T ili F).
- Takva rečenica **ne daje nikakvu potvrđnu informaciju.**
- Zaključak:
  - $\forall$  ide uz  $\Rightarrow$
  - $\exists$  ide uz  $\wedge$

# Primjer: predikatne logike

- Preslikaj u predikatnu logiku: "Niti jedan student ne sluša sve predmete."
- Rješenje: Definiramo predikate (formalna logika ne definira predikate):

$$\begin{aligned}
 S(x) & - x \text{ je student (prefiks notacija)} \\
 L(x) & - x \text{ je predmet} \\
 B(x, y) & - x \text{ sluša } y
 \end{aligned}$$

- Prioriteti logičkih operatora:  $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ .
  - Preporuka: koristi zagrade za prioritet i doseg kvantifikatora
- A) "Ne postoji  $x$  koji je student i takav da sluša sve predmete."

$$\neg \exists x [ S(x) \wedge \forall y (L(y) \Rightarrow B(x, y)) ]$$

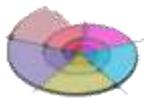

= obilježje

- B) "Za svaki  $x$  vrijedi: ako je student postoji predmet (bar jedan) koji ne sluša"

$$\forall x [ S(x) \Rightarrow (\exists y (L(y) \wedge \neg B(x, y))) ]$$


= obilježje

- Evidentno:
  - $\exists$  ide uz  $\wedge$ ,  $\forall$  ide uz  $\Rightarrow$
  - pomicanjem negacije u a) slijedi b) - DeMorgan



# Obilježja predikatne logike



- Zadovoljivost
- Model
- Logička posljedica
- Kontradiktornost
- Pravila zaključivanja



kao u propozicijskoj logici

## ■ Logika predikata višega reda

- Predikatna logika:
  - Kvantifikacija samo varijabli (objekata u domeni D), a ne na odnose (predikatni ili funkcijski simbol) u domeni D.
- Logika višega reda:
  - kvantifikacija na predikatnom (ili funkcijskom) simbolu.
  - Primjer:  $\forall(voli) (Voli \ ivo \ ana)$

- Većina interesantnih formalnih logičkih sustava je nekompletna, a vrlo malo ih je određivo.
- Propozicijska logika je:
  - *Ispravna, kompletan i odrediva* (npr. preslikavanjem u tablicu istinitosti), jer operira s konačnim skupom simbola.
- Predikatna logika je:
  - *Poluodrediva* (ako teorem postoji, dokazat će se, a ako ne postoji može se ali i ne mora dokazati).
  - "Čista" predikatna logika (npr. bez aritmetike) je **ispravna i kompletan** (Gödel).

## 1. Deduktivni pristup

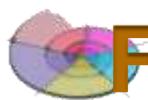
- Opis sustava (implementacija) dana skupom formula  $\Gamma$ . Treba dokazati da je formula  $\varphi$  (specifikacija sustava) logička posljedica skupa  $\Gamma$ .

$$\Gamma \models \varphi \quad \text{dokaži da su } \textit{svi modeli} \Gamma \text{ ujedno i modeli } \varphi$$

ili

$$\Gamma \vdash_L \varphi \quad \text{dokaži (izvedi) } \varphi \text{ uporabom skupa pravila L} \\ (\text{simboličko izvođenje programa})$$

- Obilježja:
  - Problem predstavljanja.
  - Zahtijeva stručno vođenje (strategije, jednakost, ...).
  - Primjena ograničena na Ulazno/Izlazne sustave (terminirajuće).
  - Može se koristiti za sustave s beskonačnim brojem stanja.

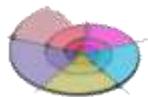


## 2. Provjera modela (engl. “model checking”)

- $\Gamma \models \varphi$  provjeravamo isključivo zadovoljivost (da li je  $\varphi$ -specifikacija istinita u jednom modelu  $\Gamma$ -implementacije, tj. *za jednu interpretaciju*)
- Obilježja:
  - Ograničeno na modele s konačnim brojem stanja = FSM.
  - Primjena u reaktivnim sustavima (neterminirajućim).
  - Automatizirano izvođenje.
- Pazi :
- Višestruka semantika (“overloading”) znaka  $\models$ :
  - *logička posljedica* (formalna logika) i
  - *zadovoljivost* (provjera modela)
  - Treba uvijek navesti kontekst u kojem se koristi ova oznaka.

# Provjera modela

- 1981. *Clarke & Emerson, Model Checking*
- Fokus: strojevi s konačnim brojem stanja i prijelazi
  - nije "općenito" dokazivanje teorema
- Zasniva se na uporabi:
  - Linearne vremenska logike
    - engl. *Linear Temporal Logic*
    - Boolean + always, until, eventually, ....
  - vremenska logika s grananjem
    - engl. *Computation Tree Logic*
    - + "for all futures"; "for some futures"



# Tipične primjene formalne logike



1. *Matematika (dokazivanje teorema)*
  2. Formalna logika (dopuna teorije)
  3. Zagonetke (imitacija racionalnog rasuđivanja)
  4. Oblikovanje računalnih sustava
  5. Automatizirano upravljanje temeljem istinitih formula
  6. ...
- 
- Izraz "formalne metode" odnosi se na razne matematičke tehnike koje se koriste za formalnu specifikaciju i razvoj programske potpore.

# Diskusija

- 
- 
- 
- 
-

# Primjer dokaza IF naredbe

- { PRE:  $\text{true}$  }
 

```
if (a>b) then
 max = a;
else
 max = b;
```
- { POST:  $(\text{max}=a \vee \text{max}=b) \wedge (\text{max} \geq a \wedge \text{max} \geq b)$  }

1. pokazati {  $\text{true} \wedge a > b$  }  $\text{max} = a$ ; { POST }

supstitucija ( $\text{max} = a$ ) u POST daje:

$$\{a = a \vee a = b\} \wedge \{a \geq a \wedge a \geq b\}$$

$$\{\text{true}\} \wedge \{\text{true} \wedge a \geq b\}$$

$$a \geq b$$

■ istinito za slučaj kada je a maksimalan tj.

za  $\{\text{true} \wedge a > b\}$  vrijedi  $\{a \geq b\}$

2. pokazati {  $\text{true} \wedge \text{not}(a > b)$  }  $\text{max} = b$ ; { POST }

supstitucija ( $\text{max} = b$ ) u POST daje:

$$\{b = a \vee b = b\} \wedge \{b \geq a \wedge b \geq b\}$$

$$\{\text{true}\} \wedge \{b \geq a \wedge \text{true}\}$$

$$b \geq a$$

■ istinito za slučaj kada je b maksimalan tj.

za  $\{\text{true} \wedge \text{not}(a > b)\}$  vrijedi  $\{a \geq b\}$

# Oblikovanje programske podrške

ak.god. 2014./2015.

## *Temelji formalne verifikacije* *Formalna verifikacija*



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



# Sadržaj

- Osnove modalne i vremenske (temporalna) logike
- Definicije Kripke struktura
- Vremenska logika s grananjem
- Specifikacija ponašanja sustava CTL formalizmom

# Sadržaj

- Formalna matematička logika
  - propozicijska
  - predikatna
- Preslikavanje formula predikatne logike u normalizirane klauzule
- Postupci formalne verifikacije računalnih sustava

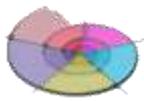


# Modalna logika



- **Modalna logika** - proširenje klasične logike "modalitetima" istinitosti (subjektivnim konceptima), kao što su npr. "što mora biti istinito" i "što može biti istinito".
- Npr.:  $p$  = atomički propozicijski simbol  
Neka je:  $p = F$  (neistinit) u sadašnjem svijetu (stanju stvari).  
Tada u modalnoj logici vrijedi:
  - (*moguće*  $p$ ) = T      ako postoji bar jedan drugi svijet
    - ◊  $p = T$ 
      - neka druga situacija, neki drugi scenarij, neka druga baza znanja u kojoj je  $p = T$ .
  - (*nužno*  $p$ ) = F      jer (*nužno*  $p$ ) = T samo akko je  $p$  istinit u svim svjetovima (što ovdje nije slučaj).
    - $p = F$

U klasičnu propozicijsku i predikatnu logiku dodaju se *modalni operatori*.



# Vremenska (temporalna) logike



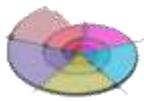
- Prema tipu modalnosti razlikujemo logike:
  - Aletička logika potrebitost, mogućnost
  - Deontička logika: obligatornost (nužnost), dozvoljivost
  - Epistemička logika: znanje, vjerovanje
  - **Vremenska logika:** uvijek, konačno, što\_je\_bilo, što\_je\_sad, što\_će\_bitи
  - ...
- **Vremenska logika** (engl. *Temporal Logic - TL*) - višestruki pogledi:
- Propozicijska vremenska logika :
  - klasična propozicijska logika proširena vremenskim operatorima.
  - najviša razina apstrakcije u rasuđivanju.
- Vremenska predikatna logika prvoga reda (varijable, funkcije, predikati, kvantifikatori):
  - različiti tipovi vremenske logike prvoga reda
  - interpretirana-neinterpretirana (prepostavlja ili ne strukturu),
  - globalne i lokalne varijable,
  - kvantifikacija preko vremenskih operatora ili ne.



# Vremenska logika: višestruki pogledi



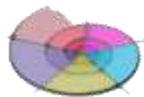
- Globalna ili modularna:
  - Endogena i egzogena. Rasuđivanje o kompletnom sustavu ili ne.
- Vremenska logika linearog vremena:
  - U svakom trenutku postoji samo jedan budući trenutak (jedna vremenska crta).
- Vremenska logika s grananjem vremena:
  - U svakom trenutku može postojati više različitih budućih vremenskih crta.
- Diskretno ili kontinuirano vrijeme.
  - U računarstvu uobičajeno diskretno vrijeme (sekvence stanja).
- Prošlo i buduće vrijeme.
  - Izvorno vremenska logika obuhvaća oba vremena.
  - U digitalnim sustavima uobičajeni su samo operatori budućeg vremena.
- Odabiremo: propozicijska, globalna, grananje, buduće vrijeme



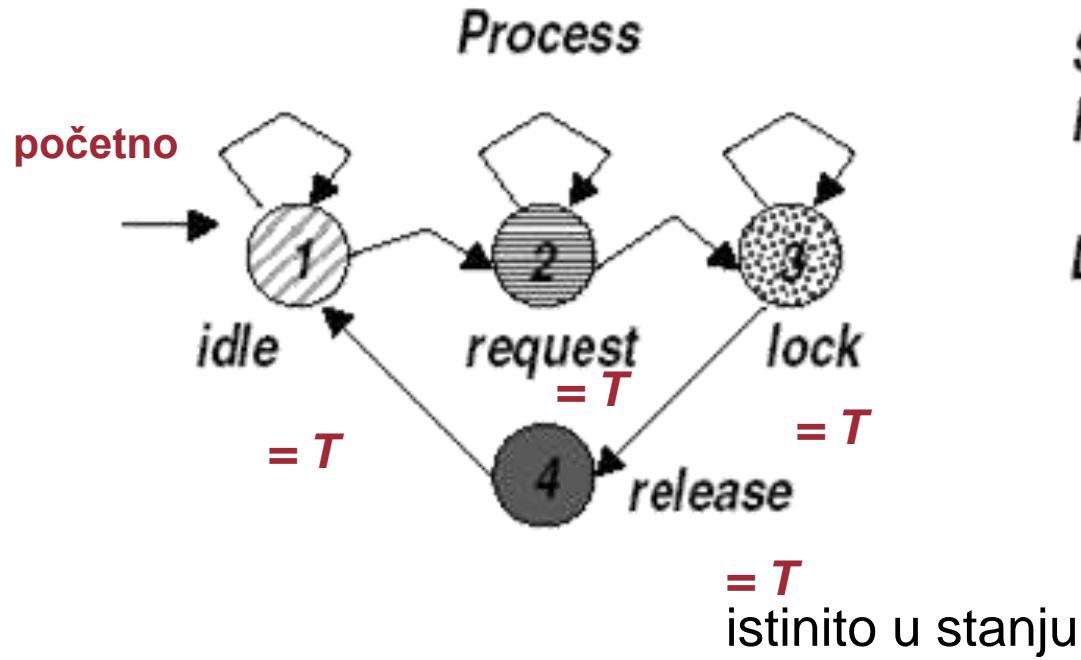
# Linearna vremenska struktura



- Linearna vremenska struktura opisana uređenom trojkom model implementacije ( / ) = *Kripke struktura M*
- Kripke struktura:  $I = M = (S, R, L)$  (1971. Saul Kripke)
- $S$  : skup stanja: - skup mogućih svjetova (stanja).
- $R$  : relacija prijelaza: -  $R \subseteq S \times S$  između svjetova (stanja).
  - $\forall s \in S \ ( \exists t \in S \mid (s, t) \in R )$
  - totalna binarna relacija
- $L : S \rightarrow 2^{AP}$  - funkcija označavanja stanja:
  - daje interpretaciju svih simbola iz skupa AP za stanje s (engl. *labeling*).
  - AP: skup atomičkih propozicijskih simbola
- Analogno i formalno jednako modelu automata (stroju stanja).
- U Kripke strukturi oznake na čvorovima grafa (kod automata oznake su na lukovima).



# Tipičan primjer Kripke strukture



$$S = \{ 1, 2, 3, 4 \}$$

$$R = \{ (1,1), (1, 2), (2, 2), (2, 3), \\ (3, 3), (3, 4), (4, 1) \}$$

$$L: L(1) = \{ \text{idle} \}$$

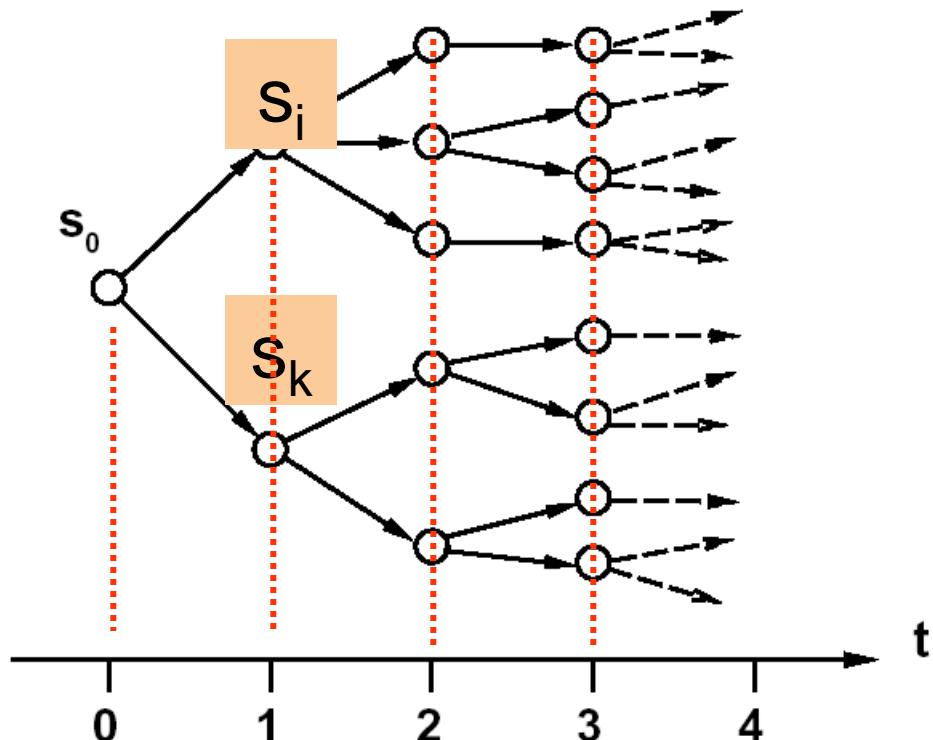
$$L(2) = \{ \text{request} \}$$

$$L(3) = \{ \text{lock} \}$$

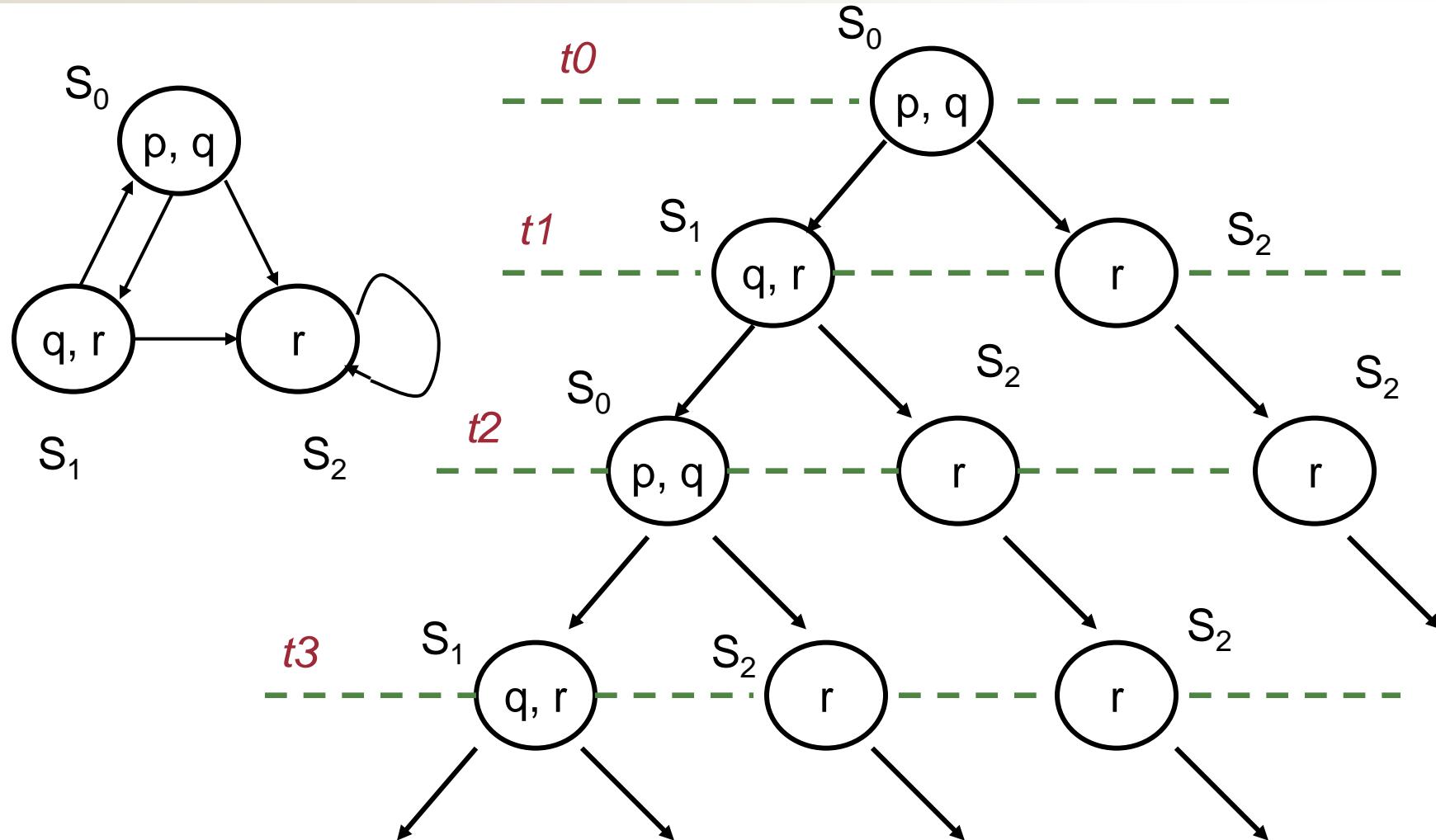
$$L(4) = \{ \text{release} \}$$

# Kripke struktura M

- Tip nedeterminističkog stroja s konačnim brojem stanja
  - 1963 Saul Kripke
- Može se promatrati kao beskonačno stablo izvođenja sustava
  - “odmota” se počevši od promatranog stanja  $s_0$
- To je *vremenska logika s grananjem*
  - *engl. Computation Tree Logic - CTL*



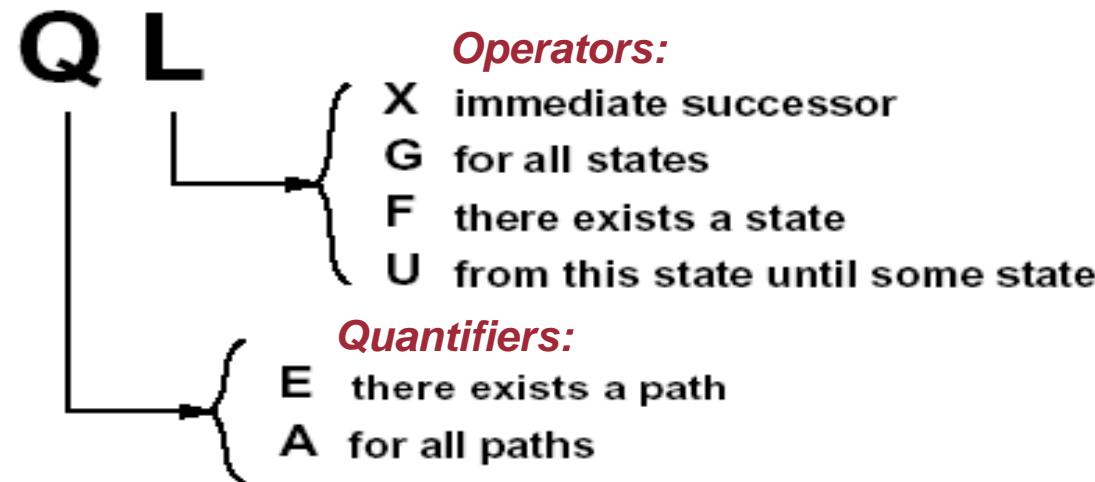
# Primjer modela



$p, q, r$  - propozicijski simboli

$S_0$  - početno stanje ili stanje koje nas zanima

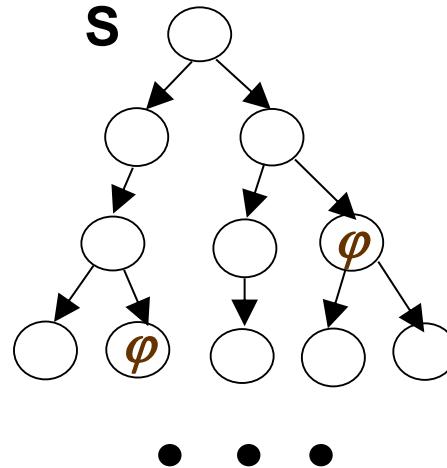
- Promatrano u kontekstu Kripke strukture
- Operatori i kvantifikatori dolaze u parovima



EX	AX
EG	AG
EF	AF
EU	AU

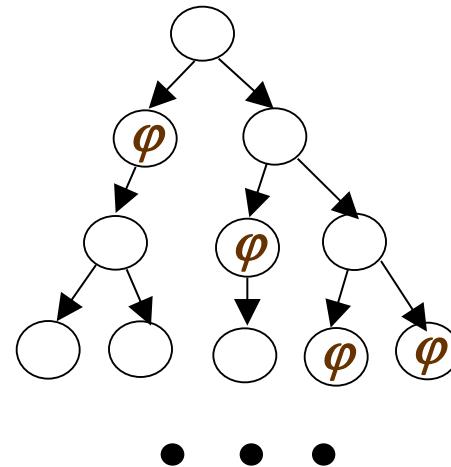
# Primjeri CTL formula

- ***M – model***
- ***S – stanje***
- ***$\varphi$  - formula***
- Postoji put takav da je  $\varphi$  eventualno istinita



**EF (exists future) -  $(EF \varphi) = T$**

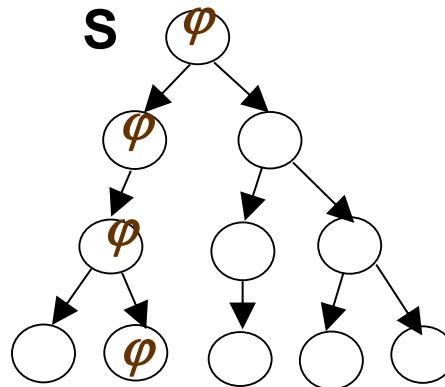
- Za sve putove vrijedi da je  $\varphi$  eventualno istinita



**AF (all future) -  $(AF \varphi) = T$**

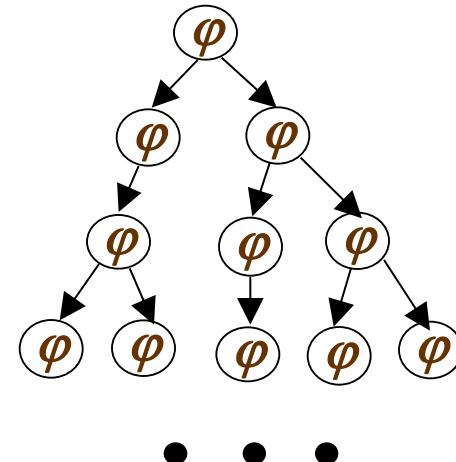
# Primjeri CTL formula

- Postoji put takav da je  $\varphi$  istinito u svim stanjima



**EG (exists globally)** -  $(EG \varphi) = T$

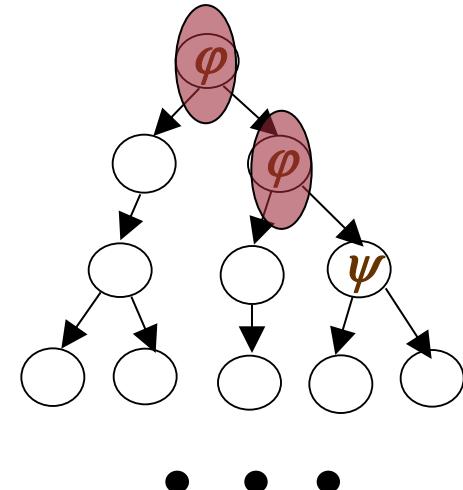
- Za sve putove vrijedi da je  $\varphi$  istinita



**AG (allways globally)** -  $(AG \varphi) = T$

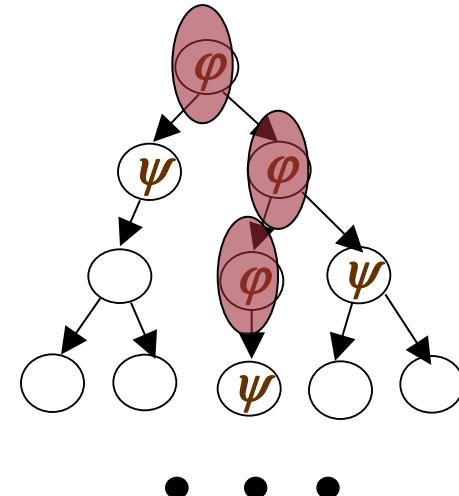
# Primjeri CTL formula

- postoji put takav da je  $\varphi$  istinita do ispunjenja  $\psi$



**EU (exists until) -  $E(\varphi U \psi) = T$**

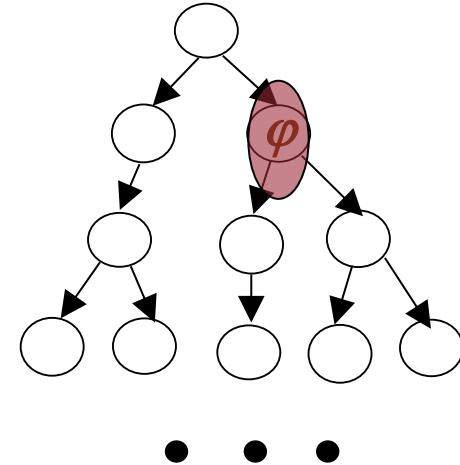
- za sve puteve vrijedi da je  $\varphi$  istinita do  $\psi$ 
  - kakva je vrijednost  $\varphi$ ?



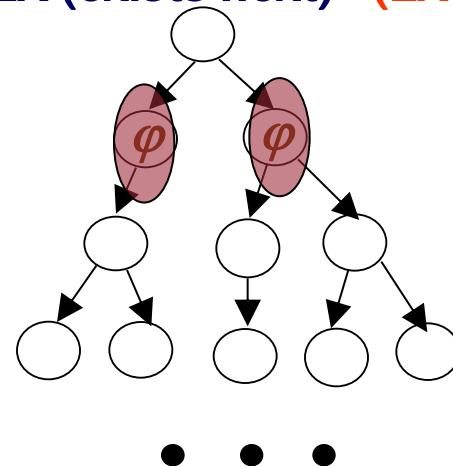
**AU (all until) -  $A(\varphi U \psi) = T$**

# Primjeri CTL formula

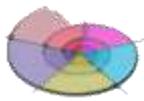
- postoji put takav da je  $\varphi$  istinita u sljedećem stanju



**EX (exists next) -  $(\text{EX } \varphi) = T$**



**AX (all next) -  $(\text{AX } \varphi) = T$**



# Formalna sintaksa CTL logike:



- Propozicijska formula  $\in L$  = atomička formula
- 1. Svaka atomička formula je *formula stanja*.
- 2. Ako su  $f, g$  formule stanja, to su i  $\neg f$ ,  $(f \wedge g)$ , (ostale se izvode)
- 3. Ako je  $f$  *formula puta*,  $E f$ ,  $A f$  su *formule stanja*.
- 4. Ako su  $g, h$  *formule stanja*, tada su  $X g$ ,  $(g U h)$  *formule puta*,
  
- Formule stanja se evaluiraju u stanjima.
- Formule puta se evaluiraju duž puta.
  
- Svi drugi operatori (npr.  $EG$ ) se mogu izraziti pravilima 1 - 4.
- AU, EU - binarni operatori
- Ostali - unarni operatori
- U CTL logici formule puta ne mogu biti ugnježđene!!
  - one traže uporabu operatora E ili A da bi postale formule stanja (pravilo 3).

# Primjeri CTL sintakse

- Ispravno/Dobro definirane CTL formule:

$AG (q \Rightarrow EG r)$

$EG p$

$E (p \cup q)$

$A (p \cup EF p)$

$AG (p \Rightarrow A [p \cup (\neg p \wedge A [\neg p \cup q] ) ] )$

- Krivo definirane formule CTL formule:

$FG p$  ; F i G slijede iza E ili A

$EF (r \cup q)$  ; U se može upariti samo sa A ili E

; Ex.:  $EF E(r \cup q)$ ,  $EF A(r \cup q)$

$AF [(r \cup q) \wedge (p \cup r)]$  ; ispravan oblik je  $A(\alpha \cup \beta)$

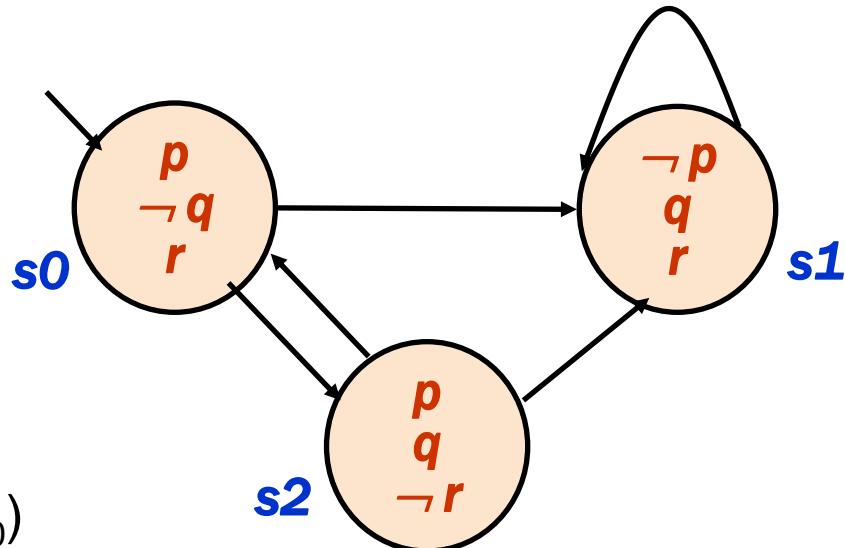
; F se ne može ovdje miješati

;  $\wedge$  može biti samo unutar  $\alpha$  ili  $\beta$

; Ex:  $A [(p \wedge q) \cup (\neg r \Rightarrow q)]$

# Primjer CTL

- Koja svojstva vrijede za zadani primjer?



- $(\text{EX } p)(s_0)$
- $(A[p \cup q])(s_0)$
- $(\text{EX } AF p )(s_0)$
- $(A[\neg p \cup q])(s_0)$

# CTL semantika

- $M = (S, R, L)$  - model sustava (Kripke struktura)
- $M, s \models \varphi$  formula vrem. Logike  $\varphi$  je *istinita* u modelu M za stanje s
- $M, s \not\models \varphi$  formula vrem. logike  $\varphi$  nije *istinita* u modelu M za stanje s

1.  $M, s \models p$  *istinita* akko  $p \in L(s)$  ; *p je propozicijski atomički simbol*

2.  $M, s \models (\varphi_1 \wedge \varphi_2)$  akko  $M, s \models \varphi_1$  i  $M, s \models \varphi_2$

3.  $M, s \models (\varphi_1 \vee \varphi_2)$  akko  $M, s \models \varphi_1$  ili  $M, s \models \varphi_2$

4.  $M, s \models (\varphi_1 \Rightarrow \varphi_2)$  akko  $M, s \not\models \varphi_1$  ili  $M, s \models \varphi_2$

5.  $M, s \models AX \varphi$  akko za sve  $s_i$  takve da  $s \rightarrow s_i$

Vrijedi  $M, s_i \models \varphi$  (u svakom slijedećem stanju)

6.  $M, s \models EX \varphi$  ako za neki  $s_i$  takav da  $s \rightarrow s_i$ , vrijedi  $M, s_i \models \varphi$   
*(u nekom slijedećem stanju)*



7.  $M, s \models AG \varphi$  akko za sve putove  $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ,

Gdje  $s = s_1$  i za svaki  $s_i$  duž puta, vrijedi  $M, s_i \models \varphi$

(za sve putove koji započinju u  $s$ , obilježje  $\varphi$  vrijedi globalno duž puta)

8.  $M, s \models EG \varphi$  akko postoji put  $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ,

Gdje  $s = s_1$  i za svaki  $s_i$  duž puta, vrijedi  $M, s_i \models \varphi$

(postoji put koji započinje u  $s$ , takav da obilježje  $\varphi$  vrijedi globalno duž puta)

9.  $M, s \models AF \varphi$  akko za sve putove  $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ,

Gdje  $s = s_1$ , postoji neki  $s_i$  duž puta, vrijedi  $M, s_i \models \varphi$

(za sve putove koji započinju u  $s$ , postoji neko buduće stanje u kojem vrijedi obilježje  $\varphi$ )



10.  $M, s \models EF \varphi$

akko postoji put  $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ,  
 Gdje  $s = s_1$  i za neki  $s_i$  duž puta, vrijedi  $M, s_i \models \varphi$   
*(postoji put koji započinje u s takav da  
 Obilježje  $\varphi$  vrijedi u nekom budućem stanju)*

11.  $M, s \models A(\varphi_1 \cup \varphi_2)$

akko za sve puteve  $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ,  
 Gdje  $s = s_1$ , taj put zadovoljava  $(\varphi_1 \cup \varphi_2)$ .  
 $\varphi_1$  je kontinuirano istinita  
 Dok se ne pojavi ( $\varphi_2 = \text{True}$ ) nekom stanju.

*Formula zahtijeva da bude ( $\varphi_2 = \text{True}$ )  
 U nekom budućem stanju.*

12.  $M, s \models E(\varphi_1 \cup \varphi_2)$

akko postoji put  $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ,  
 Gdje  $s = s_1$ , i taj put zadovoljava  $(\varphi_1 \cup \varphi_2)$ .  
 $\varphi_1$  je kontinuirano istinita  
 Dok se ne pojavi ( $\varphi_2 = \text{True}$ ) u nekom stanju.

*Formula zahtijeva da bude ( $\varphi_2 = \text{True}$ )  
 U nekom budućem stanju.*



11. i 12. :  $\varphi_1$  može biti istinit ili ne u i nakon stanja u kojem  $\varphi_2 = \text{True}$  (semantika "until" je različita od prirodnog jezika).  
 $\varphi_2$  može biti istinit i prije početnog stanja s.

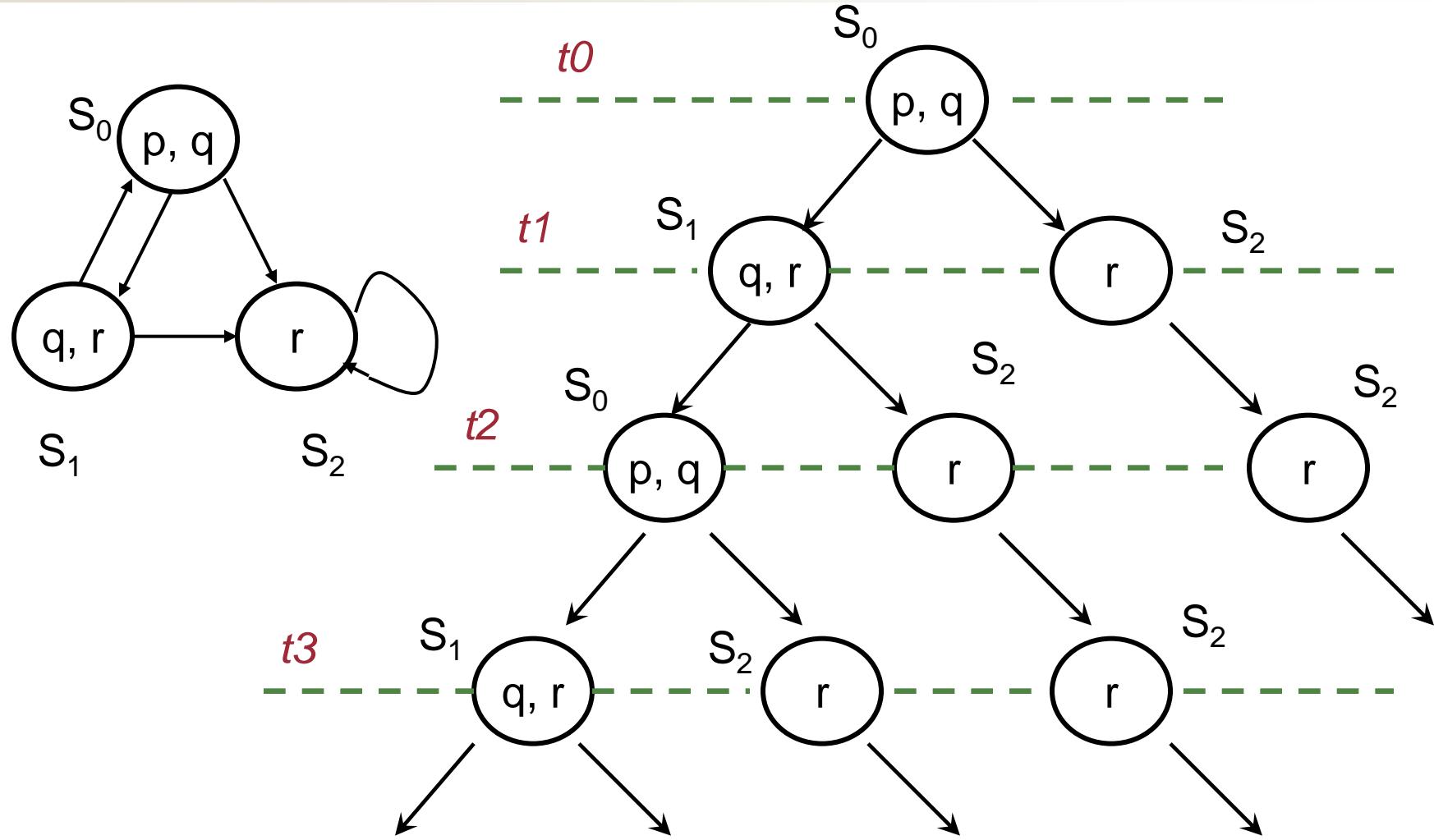
Za 7. do 12. : Skup budućih stanja uključuje i sadašnje stanje (konvencija)

Posljedica:

$$\begin{aligned} P \Rightarrow EF p & \quad (\text{ako } p \text{ vrijedi sada, } EF p \text{ također vrijedi}) \\ (AG p) \Rightarrow p & \\ P \Rightarrow A(q \cup p) & \end{aligned}$$

Ove formule su istinite u svakom stanju svakog modela.

# Primjer modela



$p, q, r$  - propozicijski simboli

$S_0$  - početno stanje ili stanje koje nas zanima



# Istinite vremenske formule za model i stanje $s_0$



- $M, s_0 \models (p \wedge q)$  ; atomi p i q su istiniti u stanju  $s_0$
- $M, s_0 \models \neg r$  ; atom r nije istinit u stanju  $s_0$
- $M, s_0 \models \text{EX } (q \wedge r)$  ; postoji put gdje je za slijedeće stanje vrijedi  $(q \wedge r)$
- $M, s_0 \models \neg \text{AX } (q \wedge r)$  ; postoji jedan put na kojem ne vrijedi za  
Slijedeće stanje  $(q \wedge r)$
- $M, s_0 \models \neg \text{EF } (p \wedge r)$  ; nema puta sa stanjem za koje vrijed  $(p \wedge r)$
- $M, s_0 \models \text{AF } r$  ; duž svih putova možemo dosegnuti stanje za  
Koje vrijedi r
- $M, s_0 \models \text{E } [(p \wedge q) \vee r]$  ; postoji put iz  $s_0$  na kojem u svim stanjima  
 $(p \wedge q) = \text{True}$ , dok  $r = \text{True}$  (npr. do  $s_2$ )
- $M, s_0 \models \text{A } [p \vee r]$  ; na svim putovima vrijedi  $[p \vee r]$

# CTL ekvivalencije

$\neg AF \varphi = EG \neg \varphi$	; de Morgan
$AF \varphi = \neg EG \neg \varphi$	
$EG \varphi = \neg AF \neg \varphi$	
$AG \neg \varphi = \neg EF \varphi$	; de Morgan
$AG \varphi = \neg EF \neg \varphi$	
$\neg AX \varphi = EX \neg \varphi$	; X je vlastiti dual
$AX \varphi = \neg EX \neg \varphi$	
$AF \varphi = A (\text{True} U \varphi) = \neg EG \neg \varphi$	
$EF \varphi = E (\text{True} U \varphi)$	
$EG \varphi = \neg [A (\text{True} U \neg \varphi)]$	

EG je nedjeljiv, tj.  $E \neg G$  nije ispravna CTL formula

Notacija:

$$A[p \cup q] = [p AU q]$$

$$E[p \cup q] = [p EU q]$$

Temeljem gornjih ekvivalencija, za izračun svih CTL formula dovoljno je imati postupke za izračun EX, EG, EU = *adekvatni skup*

- engl. *adequate set*
- Postoji više adekvatnih skupova.

# Primjeri: Preslikavanja prirodnog jezika u CTL

1. Moguće je doći u stanje gdje  $start=T$  i  $ready=F$ .

$EF (start \wedge \neg ready)$

2. Za svako stanje, ako se postavi zahtjev (za nekim resursom) biti će konačno prihvaćen (kad-tad).

$AG (zahtjev \Rightarrow AF \text{ prihvaćen})$

3. U svakom slučaju, određeni proces će *konačno* biti stalno zaustavljen

$AF (AG \text{ zaustavljen})$

4. Iz svakog stanja moguće je doći do stanja "restart".

$AG (EF \text{ restart})$

5. Na putu prema gore, dizalo na drugom katu neće promijeniti smjer gibanja, ako postoji putnik koji želi na peti kat.

$AG[ (kat=2 \wedge smjer=gore \wedge pritisnuta\_tipka\_5) \Rightarrow$

$A (smjer=gore \cup kat=5) ]$



6. Dizalo može ostati stalno stajati na trećem katu sa zatvorenim vratima.

$AG [ (kat=3 \wedge stoji \wedge vrata=zatvoreno) \Rightarrow$

$EG (kat=3 \wedge stoji \wedge vrata=zatvoreno) ]$

7. Kad god in = 1, nakon dva takta uvijek out = 1

$AG (in \Rightarrow AX AX out)$

8. Uvijek vrijedi: ako se pojavi signal "send" onda konačno signal "receive" postaje istinit, te do tog trenutka "send" mora ostati istinit

$AG (send \Rightarrow A(send \vee receive))$

# CTL provjera modela

- engl. *CTL model checking*
- Za danu Kripke strukturu (usmjereni označeni graf)
  - I određen *skup početnih stanja*  $S_0$ ,
  - Provjeri da CTL formula zadovoljava za ta stanja:

Formalno:

$$M, S_0 \models \phi, \text{ tj.}$$

$$\forall s_0 \in S_0 \quad M, s_0 \models \phi \text{ (za svako stanje iz } S_0)$$

**Postupak:**

Potrebno je pronaći sva stanja koja **zadovoljavaju** CTL formulu  $\phi$ , i **ispitati** da li je ~~željeni podskup~~  $S_0$  uključen.

*Ctl provjera modela  $\Rightarrow$  manipulacija skupovima stanja.*

# Primjer razrješavanja ugniježđenih operatora



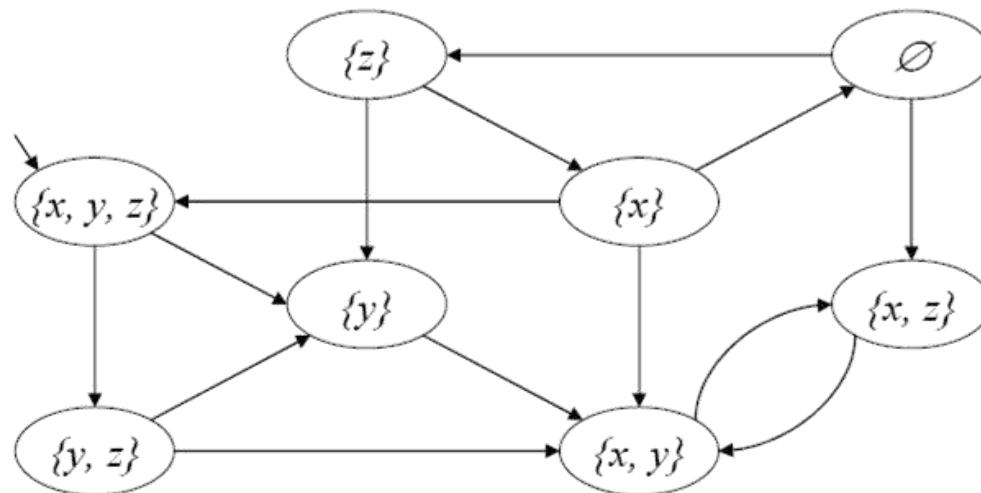
## Algoritam:

1. izračunaj stanja koje zadovoljava najugneždenija formula
  - princip iznutra prema van
2. uporijebi rezultate za izračun drugog novog formula
3. ponavljam 2

## Primjer izračuna $S_K(\text{AF AG } x)$

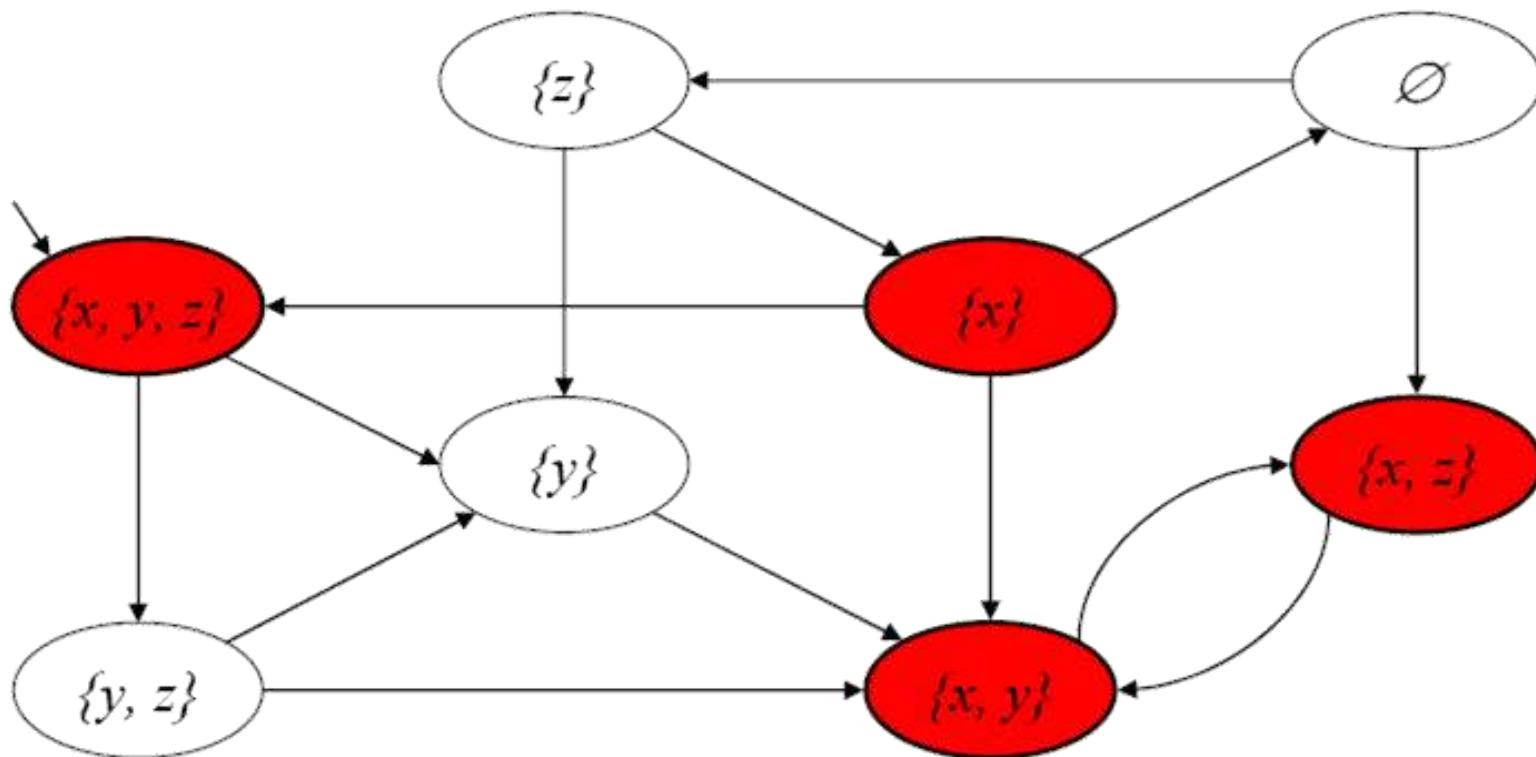
### Example

For  $S_K(\text{AF AG } x)$   
compute successively  
-  $S_K(x)$ ,  
-  $S_K(\text{AG } x)$ , and  
-  $S_K(\text{AF AG } x)$



# 1. Izračunati $S_K(x)$

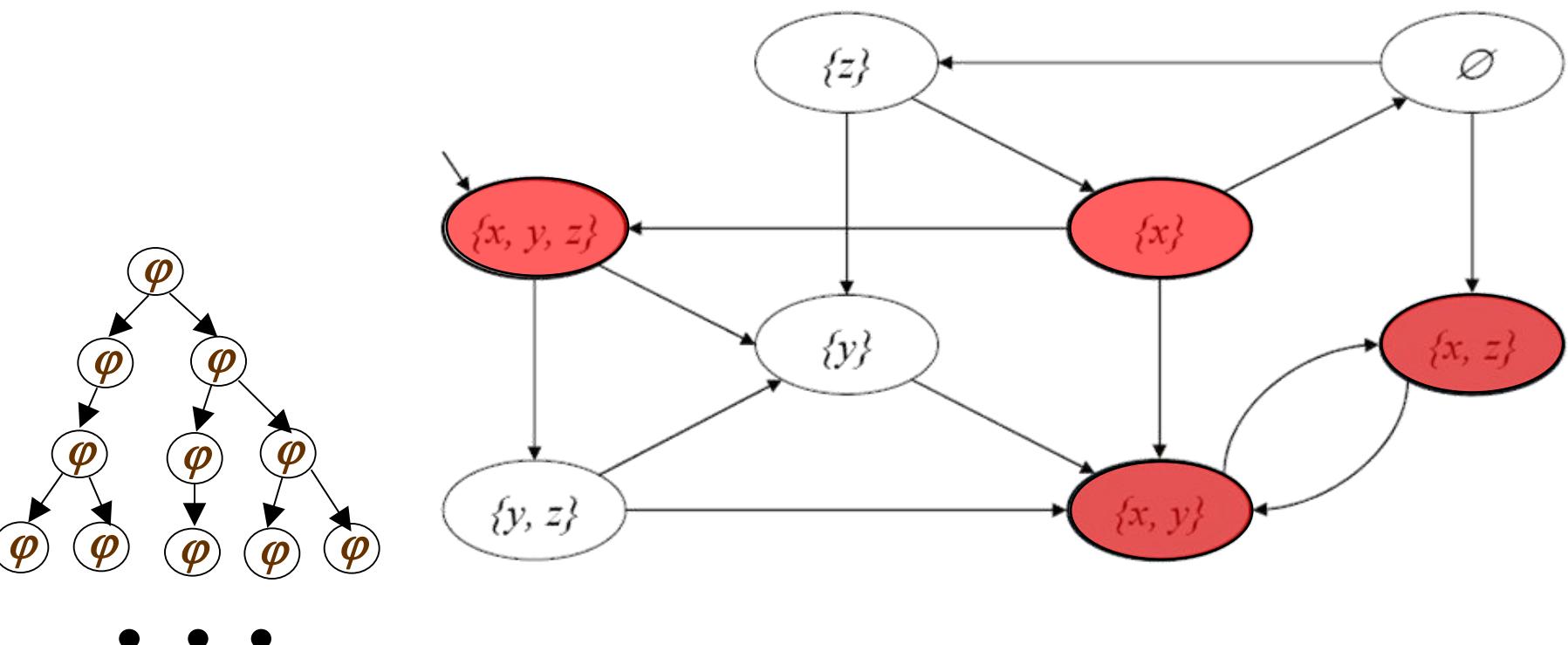
- Sva stanja u kojima postoji  $x$



## 2. Izračunati $S_K(AG\ x)$

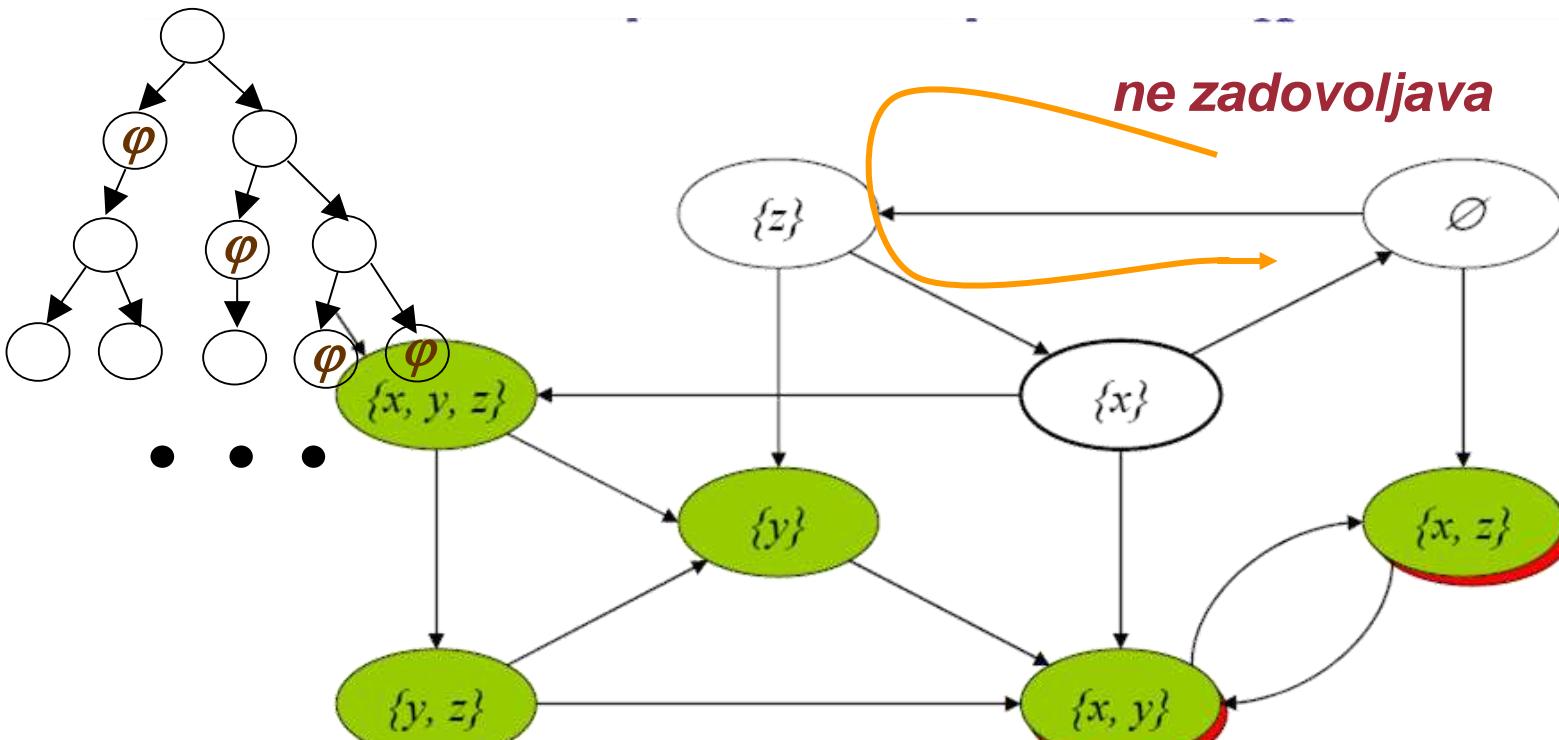
$AG(x)$  - za sve putove vrijedi da je  $x$  istinita

**Uključuje sadašnje stanje!**



UNESCO math&dev.  
TUNIS - février 2008

### 3. Izračunati $S_k(\text{AF AG } x)$



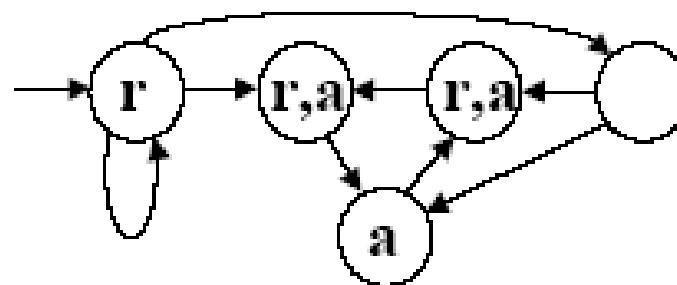
Pronaći sva stanja iz kojih se može doći do stanja AG x

NAPOMENA:

Iz tih stanja uvijek se mora moći doći do stanja AG x !

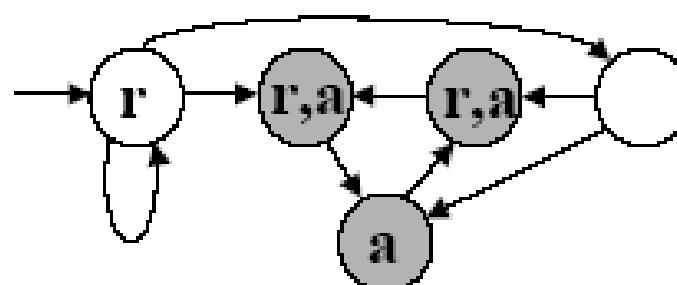
# Primjer 2

Determine the set of states in machine satisfying the  
CTL formula:  $AG(r \rightarrow AF a)$  = specifikacija



= model (implementacija)

Step (1) Set of states satisfying “ a ”





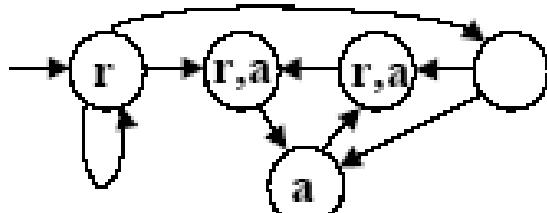
## Step (2) Set of states satisfying “AF a”

Note: a state  $s$  satisfies “AF a” if either: on all paths

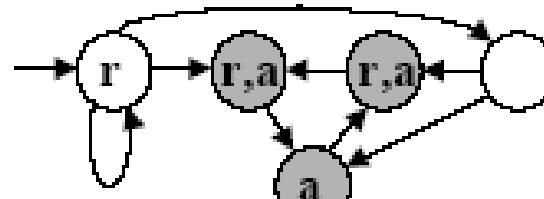
- (i) “ a ” holds in  $s$ , or
- (ii) “AF a” holds in every successor state of  $s$

Obtain approximations iteratively till fixpoint is reached.

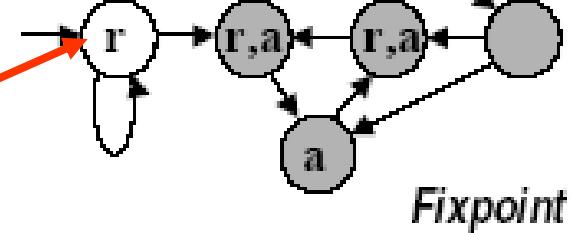
*Initial approximation:*



*Next approximation:*

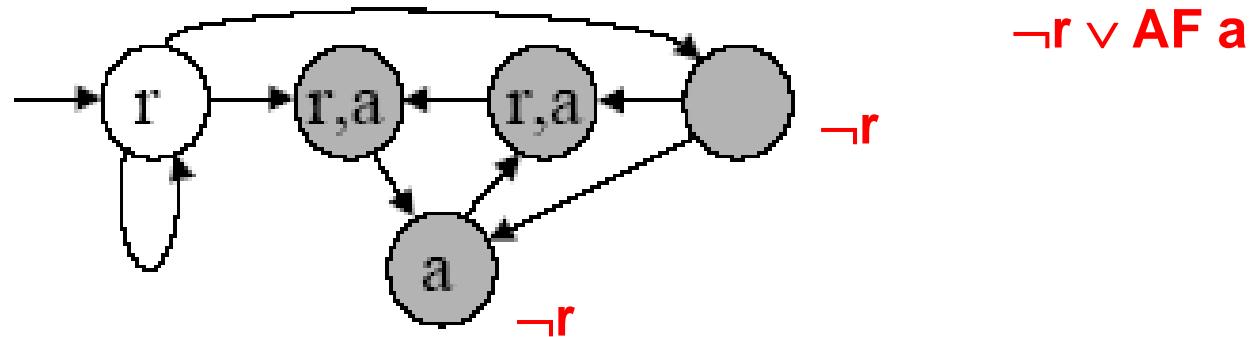


*Final approximation:*

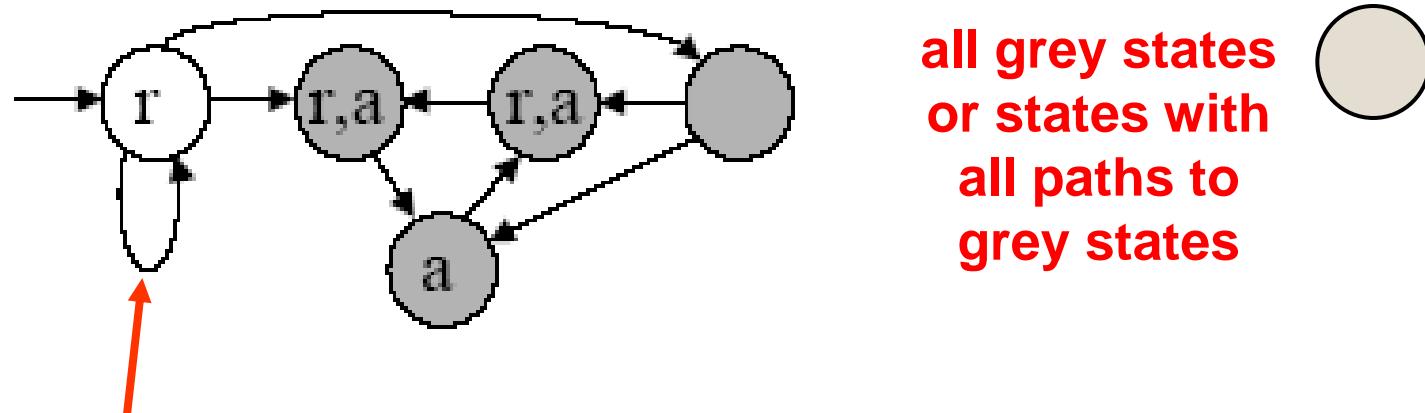


**Does not hold  
because of this path**

## Step (3) Set of states satisfying $r \Rightarrow AF a$



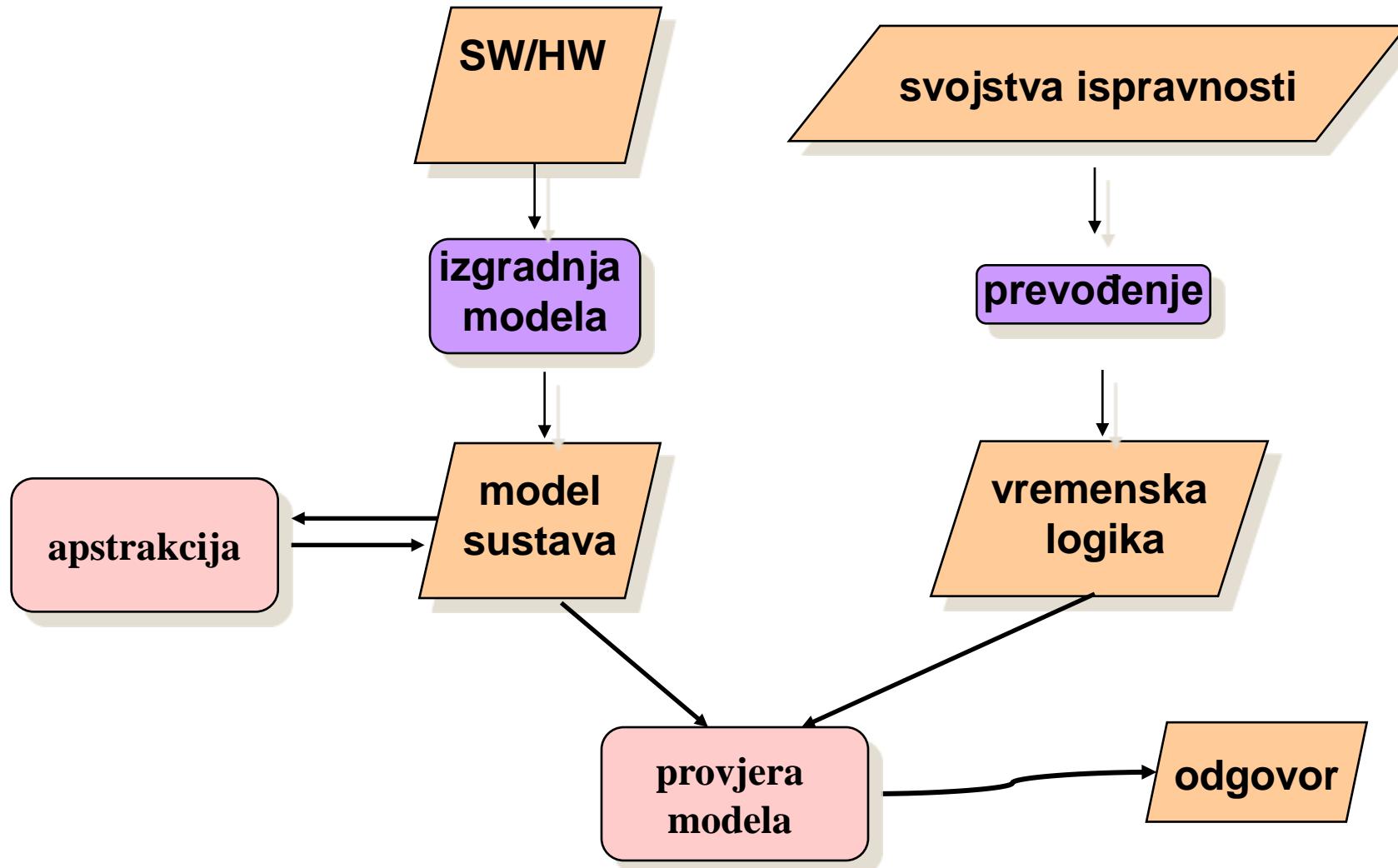
## Step (4) Set of states satisfying $AG(r \Rightarrow AF a)$



Property is not true in the initial state !

Counter-example: initial state infinitely often (not always)

# Sustavi automatske verifikacije





# Formalna specifikacija



- Velik broj razvijenih formalnih specifikacija, najrasprostranjenije:
- VDM-SL (Vienna Development Method – Specification Language),  
IBM Research Laboratory in Vienna
  - <http://www.vienna.cc/e/vdm.htm>
  - Cliff B. Jones: Systematic Software Development Using VDM, by, 2nd edition, Prentice Hall, 1990.
  - Fitzgerald, J.S., Larsen, P.G., Mukherjee, P., Plat, N. and Verhoef, M., Validated Designs for Object-oriented Systems. Springer Verlag 2005
- Z, PRG (Programming Research Group), University of Oxford, UK
  - <http://czt.sourceforge.net/>
  - Jim Woodcock, Jim Davies: "Using Z: Specification, Refinement, and Proof", Prentice Hall, 1996.
- B-Method, Jean-Raymond Abrial, France
  - <http://www.methode-b.com/>
  - J-R Abrial: The B-Book: Assigning Programs to Meanings, Cambridge University Press, 1996
  - ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 - September 2, 2000
  - ...

# Zaključci

- Pokazana je samo jedna od mnogih *formalnih metoda*.
  1. Implementacija sustava modelira se Kripke struktrom. Sustavi za verifikaciju (npr. SMV, VIS, SPIN, ...) traže opis Kripke strukture u posebnim programskim jezicima.
  2. Specifikacija željenog ponašanja izražava se CTL vremenskom logikom (u nekim sustavima i drugim vremenskim logikama).
  3. Sustav za verifikaciju prolazi kroz sva stanja modela i provjerava da li model implementacije logički zadovoljava specifikaciju (*engl. model checking*).
- Poteškoće:
  - Precizno izraziti željeno ponašanje i modelirati strukturu.
  - Sustav za verifikaciju provjerava uvijek samo jedno željeno ponašanje.
  - Programski produkti imaju ogroman skup stanja, pa je moguća provjera samo pojedinih kritičnih dijelova.

# Diskusija