UVOD

TEMELJNA PITANJA OKO PROGRAMSKOG INŽENJERSTVA

1. Što je programska potpora (software) ?
PP je računalni program i pridružena dokumentacija.

2. Što je programsko inženjerstvo ?
Disciplina koja se bavi metodama i alatima za profesionalno oblikovanje i produkciju programske potpore uzimajući u obzir cijenu.

3. Koje je razlika između programskog inženjerstva i računarske znanosti ?
RZ se bavi teorijskim dijelom, PI se bavi praktičnom primjenom.

4. Koja je razlika između programskog inženjerstva i inženjerstva sustava ?
Inženjerstvo sustava se bavi svim aspektima sustava dok PI se bavi razvojem programske infrastrukture, upravljanja te primjene.

5. Što je proces programskog inženjerstva ?
Skup aktivnosti čiji je cilj razvoj ili evolucija programskog produkta. (generičke aktivnosti: ***specifikacija*** – analizom zahtjeva specificirati što sustav treba činiti i koja su ograničenja razvoj; ***razvoj i oblikovanje*** – izbor arhitekture i produkcija sustava; ***validacija i verifikacija*** – provjera da li sve radi kako treba; ***evolucija*** – rukovanje promjenama sukladno novih zahtjevima)

6. Što je model procesa programskog inženjerstva ?
Pojednostavljeno predstavljanje procesa iz određene perspektive.

7. Kakva je struktura cijene (troška) u programskom inženjerstvu ?
Cijena sadrži cijenu razvoja, oblikovanja, ispitivanja i održanja.

8. Što su metode programskog inženjerstva ?
To su strukturni pristupi u razvoju i oblikovanju programske potpore što uključuje izbor modela sustava, pravila, preporuke i naputke.

9. Što je CASE (engl. *Computer-Aided Software Engineering*) ?
To su programski produkti namijenjeni automatiziranoj podršci aktivnostima u procesu programskog inženjerstva.

10. Koje su značajke (atributi) dobre programske potpore ?
Prihvatljivost – razumljiv koristan i kompatibilan o ostalim sustavima, pouzdanost – korisnik mora vjerovati u pouzdanost sustava, održavanje – evolucija je sukladna izmijenjenim zahtjevima.

11. Koje su osnovne poteškoće i izazovi u programskom inženjerstvu ?
Heterogenost – različite tehnike i metode razvoja
Vrijeme isporuke, povjerenje.

12. Koje vrste projekata postoje u programskom inženjerstvu ?
Korektivni, adaptivni, re-inženjerstvo, integrativni, unaprjeđujući**.**

13. Što je profesionalna i etička odgovornost ?
Povjerljivost, kompetencije, poštivanje prava intelektualnog vlasništva, ne zlorabiti računalne sustave.

- oblikovanje sustava zasnovano na modelima (apstrakcija, razumljivost, predvidljivost, jednostavnost...)
- formalna verifikacija: postupak provjere da formalni model izvedenog sustava (I), odgovara formalnoj specifikaciji (S) s matematičkom izvjesnošću
    - S = specifikacija, što sustav treba raditi
    - I = implementacija, kako sustav radi
- validacija = da li smo napravili ispravan sustav (zadovoljava li sustav funkcionalne zahtjeve)
- verifikacija = da li smo ispravno napravili sustav (odsustvo kvarova)

INŽENJERSTVO ZAHTJEVA U OBLIKOVANJU PROGRAMSKE POTPORE

- proces pronalaženja, analiziranja, dokumentiranja i provjere zahtijevanih usluga, te ograničenja u uporabi
- zahtjevi sami za sebe su specifikacija
- kompletni zahtjevi = sadrže opise svih zahtijevanih mogućnosti
- konzistentni zahtjevi = ne smiju sadržavati konflikte ili kontradikcije u opisima zahtijevanih mogućnosti
- zahtjevi postavljaju što sustav treba raditi i definiraju ograničenja u implementaciji i radu sustava
- dokument zahtjeva programskog produkta: usklađen skup izjava o svim zahtjevima na sustav (sadrži: uvod, opći opis sustava, specifičnosti zahtjeva, priloge, indeks)

- klasifikacija obzirom na razinu detalja i sadržaj:

a) Zahtjevi obzirom na razinu detalja:

1. korisnički zahtjevi (visoka razina apstrakcije) – moraju biti razumljivi ne-tehničkom osoblju, pišu se prirodnim jezikom te grafovima; predlažu se u okviru ponude za izradu programskog produkta
2. zahtjevi sustava (vrlo detaljna specifikacija) – pišu se strukturnim prirodnim jezikom, jezikom za oblikovanje sustava i matematičkom notacijom; uobičajeni su nakon prihvaćanja ponude, a prije sklapanja ugovora
3. specifikacija programske potpore – najdetaljniji opis i objedinjuje korisničke te sustavske zahtjeve

b) Zahtjevi obzirom na sadržaj (odnosi se na korisničke i zahtjeve sustava)**:**

1. funkcionalni – izjave o uslugama koje sustav mora sadržavati, kako sustav reagira na poticaje te kako se mora ponašati u određenim situacijama (system shall do)
2. nefunkcionalni – ograničenja u uslugama i funkcijama (system shall be)
3. zahtjevi domene primjene – zahtjevi koji proizlaze iz domene sustava kao i oni koji karakteriziraju tu domenu; mogu biti novi funkcionalni zahtjevi ili ograničenja na postojeće zahtjeve; problemi su razumljivost i implicitnost

Klasifikacija nefunkcionalnih zahtjeva:

1. zahtjevi programskog produkta – ponašanje na određen način (npr. vrijeme odziva)
2. organizacijski zahtjevi - rezultat organizacijskih pravila i procedura
3. vanjski zahtjevi – zahtjevi izvan sustava i razvojnog procesa (međusobna operabilnost, legislativni zahtjevi...)

Zahtjevi sustava

- uloga = definiranje oblikovanja sustava
- izražavanje zahtjeva sustava:
    - strukturirani prirodan jezik: definiranje standardnih formulara i obrazaca u kojima se izražavaju zahtjevi; prednost = zadržavanje izražajnosti prirodnog jezika; nedostatak = ograničena terminologija
    - jezik za opis oblikovanja: definira se operacijski model sustava (npr. SDL)
    - grafička notacija: grafički jezik proširen tekstom (npr. UML)

- matematička specifikacija: notacija zasnovana na matematičkom konceptu; najstrože definirana specifikacija (npr. FSM, teorija skupova, logika...)

- specifikacija sučelja
  1) proceduralno sučelje – API = primjensko programsko sučelje
  2) struktura podataka koje se izmjenjuju s drugim sustavima
  3) predstavljanje podataka = značenje pojedinih podataka

Procesi inženjerstva zahtjeva (i.z.)

- skup aktivnosti koje generiraju i dokumentiraju zahtjeve
- nema jedinstvenog procesa i.z. – razlikuju se ovisno o domeni primjene, ljudskim resursima i organizaciji koja oblikuje zahtjeve
- modeli procesa i.z.:
  - klasični model
  - spiralni model
    - trostupanjska aktivnost: specifikacija, validacija, izlučivanje
    - promatra proces i.z. kroz iteracije – u svakoj iteraciji različit intenzitet aktivnosti i različita razina detalja (ranije iteracija imaju fokus na razumijevanju poslovnog modela, a kasnije na modeliranje sustava)
- generičke aktivnost:
  1) **studija izvedivosti**
     - na početku se određuje da li se predloženi sustav isplati
     - provjerava se:
       i. doprinos sustava ciljevima organizacije u koju se uvodi
       ii. mogućnost ostvarenja postojećom tehnologijom i sredstvima
       iii. mogućnost integracije s postojećim sustavima organizacije
     - provedba: određivanje koje informacije su potrebne za studiju
  2) **izlučivanje zahtjeva**
     - najznačajnija aktivnost u procesu i.z. u kojoj se zajedno s kupcima i korisnicima razjašnjava domena primjene, definiraju se usluge koje sustav treba pružiti s kojim ograničenjima
     - problemi: dionici ne znaju što stvarno žele, izražavaju zahtjeve na specifične načine, imaju konfliktne zahtjeve, zahtjevi se mijenjaju tokom procesa analize, pojavljuju se novi dionici...
     - isprepleteno s analizom zahtjeva (otkrivanje funkcija, struktura i ponašanja sustava)
     - pogledi: način strukturiranja zahtjeva tako da oslikavaju perspektivu i fokus različitih dionika čime omogućava razrješavanje konflikata; tipovi pogleda:
       i. **pogledi interakcije** (ljudi i sustavi koji izravno komuniciraju sa sustavom)
       ii. **indirektni pogledi** (dionici koji ne koriste sustav izravno, ali utječu na zahtjeve)
       iii. **pogledi domene primjene** (karakteristike domene i ograničenja koja utječu na zahtjeve)
     - metode izlučivanja zahtjeva:
       i. **intervjuiranje**
          - tim zadužen za i.z. ispituje dionike o sustavu koji trenutno koriste te o novo predloženom sustavu
          - tipovi: zatvoreni (pitanja prije definirana) i otvoreni (ne postoje definirana pitanja nego se raspravlja s dionicima)
          - u praksi ne daje dobre rezultate (različit rječnik inženjera i korisnika)

ii. **scenarij**
    - primjeri iz stvarnog života o načinu korištenja
    - dionici diskutiraju i kritiziraju scenarij
    - sadržaj: opis početne situacije, opis normalnog tijeka događaja, opis eventualnih pogrešaka, informacije i paralelnim aktivnostima, opis stanja gdje scenarij završava
iii. **obrasci uporabe (use cases)**
    - tehnika preuzeta iz UML standarda
    - skup obrazaca uporabe opisuju sve moguće interakcije sustava
    - 3 temeljna elementa: obrasci uporabe, aktori, odnosi
    - pogled kako ga vide vanjski korisnici
    - pogodno za korisničke zahtjeve i scenarije ispitivanja sustava
iv. **specificiranje dinamičkih interakcija u sustavu**
    - sekvencijski i kolaboracijski dijagrami
    - omogućava specificiranje međudjelovanja između elemenata sustava
v. **promatranje rada, izrada prototipa...**

- spiralni model u izlučivanju i analizi zahtjeva; 4 osnovne aktivnosti:
    i. <u>izlučivanje/otkrivanje zahtjeva</u>
    ii. <u>klasifikacija i organizacija zahtjeva</u>
    iii. <u>ustanovljenje prioriteta i pregovaranje</u>
        - zahtjevi visokog prioriteta (neophodno demonstrirati klijentu pri preuzimanju)
        - srednji prioritet (obavezno razmotriti pri analizi i oblikovanju)
        - nizak prioritet (analizira se u obliku naprednih mogućnosti, prikaz budućeg razvoja)
    iv. <u>dokumentiranje zahtjeva</u>

3) <u>validacija zahtjeva</u>

- cilj: pokazati da zahtjevi odgovaraju sustavu koji naručitelj doista želi (naknadno ispravljanje je puno skuplje)
- tehnike validacije:
    i. recenzije zahtjeva (sistemska ručna analiza)
    ii. izrada prototipa (provjera na izvedenom sustavu)
    iii. generiranje ispitnih slučajeva
- elementi provjere: valjanost, konzistencija, komplementnost, realnost, provjerljivost, razumljivost, sljedivost, adaptabilnost

4) <u>upravljanje promjenama zahtjeva</u>

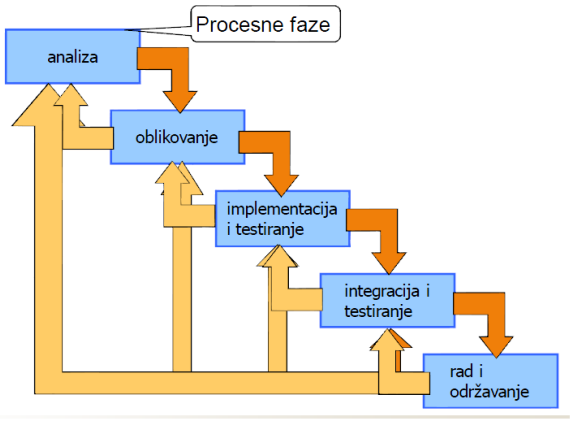
- klasifikacija promjena zahtjeva:
    i. okolinom promijenjeni zahtjevi
    ii. novonastali zahtjevi
    iii. posljednični zahtjevi (nastaju nakon uvođenja sustava u eksploataciju, a rezultat su promjena procesa rada u organizaciji nastalih uvođenje novog sustava)
    iv. zahtjevi kompatibilnosti (zahtjevi koji ovise o procesima drugih sustava)

- socijalni i organizacijski čimbenici:
    - utječu na sve poglede – programski produkti se uvijek koriste u socijalnom i organizacijskom kontekstu
    - etnografija – zahtjevi izvedeni temeljem istraživanja kako ljudi stvarno rade, a ne kako bi definicija poslovnog procesa to propisivala

PROCESI PROGRAMSKOG INŽENJERSTVA

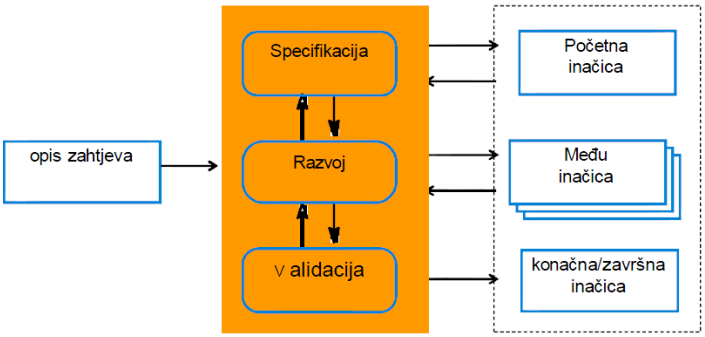
- strukturirani skup aktivnosti potreban da se oblikuje i razvija programski produkt
- generičke aktivnosti procesa programskog inženjerstva:
    - specifikacija
    - razvoj i oblikovanje
    - validacija i verifikacija
    - evolucija

- model procesa programskog inženjerstva: apstraktna reprezentacija procesa, predstavlja opis procesa iz određene perspektive
- model životnog ciklusa opisuje faze od začetka projekta do kraja životnog vijeka proizvoda
- generički modeli programskog inženjerstva:
    - **<u>vodopadni</u>**

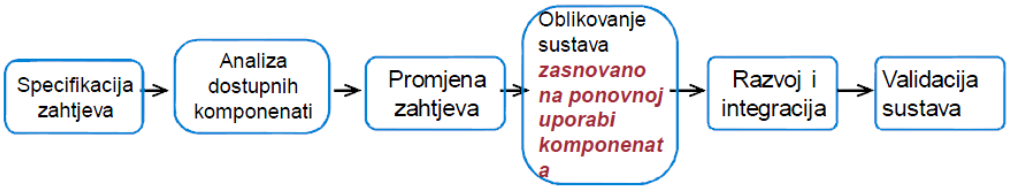


- odvojene faze specifikacije i razvoja
- procesne faze:
    - analiza zahtjeva i definicije
    - razvoj i oblikovanje sustava i programske potpore
    - implementacija i ispitivanje (testiranje) modula
    - integracija i testiranje sustava
    - uporaba sustava i održavanje
- pojedina faza u vodopadnom se mora dovršiti prije pokretanja nove faze
- pretpostavke:
    - zahtjevi poznati prije oblikovanja
    - zahtjevi se rijetko mijenjaju
    - korisnik zna što želi i bez objašnjenja
    - oblikovanje se može provoditi odvojeno i rijetko dovodi do pogrešaka
    - jednostavniji sustavi
- problemi:
    - poteškoće ugradnje promjena nakon što je proces pokrenut
    - nefleksibilna podjela projekta u odvojene dijelove
    - model prikladan samo za dobro razumljive zahtjeve
- uglavnom se koristi za velike projekte koji se razvijaju na nekoliko odvojenih mjesta

- **evolucijski**



  - specifikacija, razvoj i validacija su isprepleteni
  - dva uobičajena postupka:
    - metoda odbacivanja prototipa
      - cilj je razumijevanje zahtjeva sustava
      - započinje grubo definiranim zahtjevima koji se tijekom postupka razjašnjavaju (što je doista potrebno?)
    - istraživački razvoj i oblikovanje
      - cilj je kontinuirani rad s kupcem na temelju inicijalne specifikacije
      - započinje dobro definiranim zahtjevima, a novi se dodaju na temelju prijedloga kupca
  - problemi:
    - proces razvoja i oblikovanja nije jasno vidljiv
    - često loša struktura sustava
    - često potrebne posebne vještine
  - primjena: mali i srednji interaktivni sustavi, dijelovi velikih sustava, sustavi s kratkim vijekom trajanja
- **komponentno usmjereni (CBSE)**

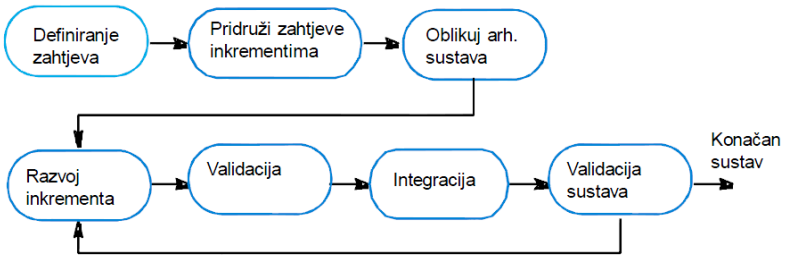


  - sustav se integrira višestrukom uporabom postojećih komponenti
  - stupnjevi procesa:
    - specifikacija i analiza zahtjeva
    - analiza komponenti
    - modifikacija zahtjeva
    - oblikovanje sustava s višestrukom uporabom komponenti
    - razvoj i integracija
- **RUP proces**
  - zasnovan na oblikovanju uporabom modela
  - use case osnova, njime se semantički povezuju sve aktivnosti
  - u fokusu je arhitektura sustava
  - izveden na temelju jezika za modeliranje UML-a i pridruženih aktivnosti
  - svojstva:
    - priznaje utjecaj korisnika
    - sugerira evolucijski pristup
    - podržava OO
    - prilagodljiv
  - najčešće opisan kroz 3 perspektive:
    - dinamička – slijed faza kroz vrijeme
    - statička – pokazuje aktivnosti procesa
    - praktična – sugerira aktivnosti kroz iskustvo i dobru praksu

- dvije dimenzije RUP procesa:
  - horizontalna (dinamika):ciklusi, faze, iteracije, ključne točke
  - vertikalna (statika): aktivnosti, discipline, uloge, artifakti
- faze RUP procesa:
  - **početak** – definira opseg projekta, razvoj modela poslovnog procesa
  - **razrada** (elaboracija) – obuhvaća plan projekta, specifikaciju značajki i temelje arhitekture sustava
  - **izgradnja** – izgradnja produkta (oblikovanje, programiranje, ispitivanje)
  - **prijenos** – prijenos produkta korisnicima
- ključne točke – definiraju pridružene dokumente ili aktivnosti
- jezgrene aktivnosti:
  - zahtjevi
  - analiza
  - oblikovanje i implementacija
  - test
- potporne aktivnosti:
  - mgmt. konfiguracijom
  - management
  - briga o okolišu
- svakoj aktivnosti pridružen je jedan ili više modela (model=apstraktan opis sustava iz određene perspektive)
- RUP posjeduje značajke iterativnog i inkrementalnog oblikovanja programske potpore
- arhitektura sustava sadrži skup pogleda u modele (skup dijagrama)

Iteracije u modelima

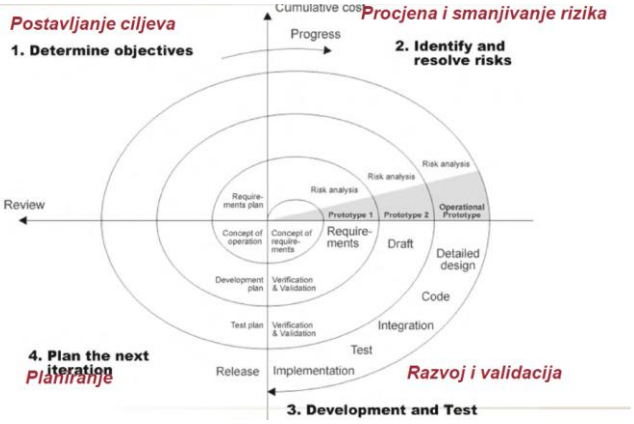
- 2 međuovisna pristupa iteracijama:
  - **inkrementalni pristup**



- sustav se ne isporučuje korisniku u cjelini nego se razvoj, oblikovanje i isporuka razbijaju u inkrementalne dijelove
- zahtjevi korisnika se svrstavaju u prioritetne cjeline i oni više prioriteta se isporučuju ranije
- s početkom razvoja pojedinog inkrementa njegovi zahtjevi se fiksiraju
- prednosti:
  - kupac dobiva svoju vrijednost sa svakim inkrementom
  - funkcionalnost sustava se ostvaruje u ranim fazama
  - 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)

- primjer: ekstremno programiranje:
    - bazira se na razvoju, oblikovanju i isporuci vrlo malih inkremenata funkcionalnosti
    - kontinuirano poboljšavanje koda
    - sudjelovanje korisnika u razvojnom timu
    - programiranje u paru
    - spada u agilne/ubrzane modele

- **spiralni razvoj i oblikovanje**



- proces se predstavlja spiralom umjesto sekvencom s povratima
- svaka petlja u spirali predstavlja fazu procesa
- nema fiksnih faza → petlje se u spirali izabiru prema potrebnim zahtjevima
- rizici razvoja programskog produkta eksplicitno se određuju i razrješuju
- sektori:
    - postavljanje ciljeva – identifikacija specifičnih ciljeva
    - procjena i smanjivanje rizika
    - razvoj i validacija – odabire se model razvoja i oblikovanja
    - planiranje – projekt se kritički ispituje i planira se sljedeća spiralna faza
- 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

Generičke aktivnosti u procesu programskog inženjerstva

1. **specifikacija programskog produkta**
    - proces određivanja potrebnih usluga i ograničenja u radi i razvoju sustava
    - određuje se procesom inženjerstva zahtjeva
    - rezultira dokumentom u kojem se navode potrebne usluge i ograničenja u radu i razvoju sustava

2. **oblikovanje i implementacija programskog produkta**
    - proces preslikavanja specifikacije u stvarni, realni sustav
    - oblikovanje programskog produkta – oblikovanje strukture sustava koja realizira specifikaciju (izbor i modeliranje arhitekture)
    - implementacija – preslikavanje strukture u izvršni program
    - aktivnosti procesa oblikovanja:
        - izbor i oblikovanje arhitekture
        - apstraktna specifikacija
        - oblikovanje sučelja
        - oblikovanje komponenti
        - oblikovanje struktura podataka
        - oblikovanje algoritama
3. **validacija i verifikacija programskog produkta**
    - potrebno je pokazati da sustav odgovara specifikaciji i zadovoljava zahtjevima kupca i korisnika
    - ispitivanje sustava temelji se na radu sustava s ispitnim ulaznim parametrima koji se izvode iz specifikacije realnih podataka koje sustav treba prihvatiti
    - testiranje - usporedba stvarnih rezultata s postavljenim standardima
    - proces ispitivanja sustava:
        - ispitivanje komponenti i modula
        - ispitivanje sustava
        - ispitivanje prihvatljivosti
    - vrste ispitivanja programske potpore:
        - ispitivanje komponenti
        - integracijsko ispitivanje
        - ispitivanje sustava
        - test prihvatljivosti
        - test instalacije
        - alpha test
        - beta test
4. **evolucija programskog produkta**
    - programski produkt je inherentno fleksibilan i može se mijenjati
    - kako se mijenjaju zahtjevi na sustav, tako se i programski produkt mora mijenjati

Računalom podržano programsko inženjerstvo (CASE)

- CASE (Computer-aided software engineering)
- programski produkti namijenjeni podršci aktivnostima u procesu programskom inženjerstva (specifikacija, oblikovanje i evolucija)
- poboljšanje i kontrola procesa
- alati za automatizaciju oblikovanja:
    - grafički editori za razvoj modela sustava
    - rječnici i zbirke za upravljanje entitetima u oblikovanju
    - grafičko okruženje za oblikovanje i konstrukciju korisničkih sučelja
    - alati za pronalaženje pogrešaka u programu
    - automatizirani prevoditelji koji generiraju nove inačice programa
- CASE slabo podupire timski rad
- klasifikacija CASE alata:
    - funkcionalna perspektiva – alati se klasificiraju prema specifičnoj funkciji
    - procesna perspektiva – alati se klasificiraju prema aktivnostima koje podupiru u procesu

    - integracijska perspektiva – alati se klasificiraju prema njihovoj organizaciji u integrirane cjeline
        - alati
            - podupiru individualne zadatke u procesu
        - radne klupe
            - podupiru pojedine faze procesa
            - integriraju više alata u jedinstvenu okolinu
        - razvojne okoline
            - skupina alata i radnih klupi
            - podupiru cijeli ili značajni dio procesa programskog inženjerstva
- prednosti CASE alata:
    - veća produktivnost
    - bolja dokumentacija
    - veća točnost
    - poboljšana kvaliteta
    - smanjeni troškovi održavanja
    - utjecaj na organizaciju rada
- nedostaci:
    - velika cijena
    - vrijeme učenja
    - potreba za različitim alatima

ARHITEKTURA PROGRAMSKE POTPORE

- struktura ili strukture sustava koji sadrži elemente, njihova izvana vidljiva obilježja i odnose između njih
- uloga arhitekture programske potpore:
    - apstrakcija na visokom nivou
    - osnovni nositelj kvalitete sustava
    - kapitalna investicija koja se može ponovno koristiti
- prednosti definiranja arhitekture:
    - smanjuje cijelu oblikovanja, razvoja i održavanja programskog produkta
    - omogućuje ponovnu uporabu rješenja
    - poboljšava razumljivost
    - poboljšava kvalitetu produkta
    - razjašnjava zahtjeve
    - omogućuje donošenje temeljnih inženjerskih odluka
    - omogućuje ranu analizu i uočavanje pogrešaka u oblikovanju
- oblikovanje arhitekture programske potpore:
    - proces identificiranja i strukturiranja podsustava koji čine cjelinu te okruženja za upravljanje i komunikaciju između podsustava
    - rezultat = opis arhitekture programske potpore
- opis arhitekture je skup dokumentiranih pogleda raznih dionika
    - pogled je djelomično obilježje razmatrane arhitekture programa i dokumentarin je dijagramom koji opisuje strukturu sustava, a sadrži elemente, odnose među elementima i vanjski vidljiva obilježja

- arhitektura programske potpore opisuje se modelima od kojih svaki sadrži barem jedan pogled
    - klasifikacija modela:
        - statični stukturni – pokazuje kompoziciju/dekompoziciju sustava
        - dinamički procesni model – komponente u izvođenju
        - alocirani elementi – dokumentacija odnos programske pogpore i razvojne okoline
- klasifikacija arhitekture po dosegu
    1. **koncepcijski**
        - usmjeravanje pažnje na pogodnu dekompoziciju sustava
        - komunikacija s netehničkim dionicima
    2. **logički**
        - precizno dopunjena
        - detaljan nacrt pogodan za razvoj komponenti
    3. **izvršni**
        - namijenjena distribuiranim i paralelnim sustavima
        - pridruživanje procesa fizičkom sustavu

Proces izbora i evaluacije arhitekture programske potpore

- odabir najbolje opcije između više mogućih rješenja
- prostor oblikovanja = skup opcija koje su na raspolaganju uporabom različitih izbora
- oblikovanje od vrha prema dolje (**Top-down design**)
    - oblikuje najvišu strukturu sustava pa postepeno razrađuje detalje
    - dobra struktura sustava
- oblikovanje od dna prema vrhu (**Bottom-up design**)
    - stvaranje komponenti pogodnih za ponovnu uporabu
- hibridno oblikovanje
    - uporaba obje metode
- tehnike donošenja dobrih odluka oblikovanja:
    - **uporaba prioriteta i ciljava za odabir alternativa**
        1. prebroji i opiši alternative odluka oblikovanja
        2. svakoj alternativi pribroji prednosti i nedostatke u odnosu na ciljeve i prioritete
        3. odredi alternative koje su u sukobu s ciljevima
        4. odredi alternative koje najbolje zadovoljavaju ciljeve
        5. prilagodi prioritete za daljnje donošenje odluka
    - **uporaba analize troškova i koristi za odabit**
        - utvrđivanje troškova novog sustava (razvoj i redovni rad)
            - cijena rada programskog inženjera (+održavanje), cijena uporabe razvojne tehnologije i cijena krajnjih korisnika i potpore
        - utvrđivanje koristi novog sustava (mjerljive i direktne; teško mjerljive i posredne)
            - ušteda vremena programskog inženjera
            - dobrobiti mjerene kroz povećanu prodaju ili ostale financijske uštede

Kriteriji za izbor arhitekture

- **principi oblikovanja**
    1. **podijeli i vladaj**
        - jednostavniji rad s više malih dijelova
        - odvojeni timovi rade na manjim problemima – omogućena specijalizacija
        - manje komponente = veća razumljivost
        - olakšana zamjena dijelova

2. **povećaj koheziju**
    - grupiranje međusobno povezanih elemenata (olakšano razumijevanje i promjene)
    - funkcijska
        - kod koji obavlja pojedinu operaciju je grupiran, sve ostalo izvan
        - olakšano razumijevanje, povećana ponovna uporabljivost modula, lakša zamjena
    - razinska
        - svi resursi za pristup skupu povezanih servisa na jednom mjestu, sve ostalo izvan
        - razine formiraju hijerarhiju (više mogu pristupiti nižoj razini, obrnuto ne)
        - API (skup procedura kojima pojedina razina omogućava pristup servisima)
        - mogućnost zamjene pojedine razine bez utjecaja na druge
    - komunikacijska
        - svi moduli koji pristupaju ili mijenjaju određene podatke su grupirani, sve ostalo izvan
        - npr. klase
    - sekvencijska
        - grupiranje procedura u kojoj jedna daje ulaz sljedećoj
    - proceduralna
        - procedure koje se upotrebljavaju jedna nakon druge
    - vremenska
        - operacije koje se obavljaju tijekom iste faze rada programa su grupirane
    - korisnička (kohezija pomoćnih programa)
        - povezani pomoćni programi koji se logički ne mogu smjestiti u ostale grupe
3. **smanji međuovisnost**
    - međuovisnost sadržaja
        - jedna komponenta prikriveno mijenja interne podatke druge komponente
        - OO: enkapsulacija (deklarirati kao private; korištenje get i set)
    - opća međuovisnost
        - pri uporabi globalne varijable
    - upravljačka međuovisnost
        - izravna kontrola rada druge procedure uporabom zastavice ili naredbe
        - izbjegavanje: uporabom polimorfnih operacija u objektnom pristupu; look-up tablice
    - međuovisnost u OO oblikovanju
        - javlja se kad je jedna klasa deklarirana kao tip argumenta metode
        - jedna klasa upotrebljava drugu te na taj način otežava promjene sustava
        - smanjenje: uporaba sučelja, prijenos jednostavnijih varijabli
    - podatkovna međuovisnost
        - javlja se kad je tip metode argumenata primitiv ili jednostavna klasa biblioteke
    - povezivanje poziva procedura
        - javlja se kada procedura ili metoda u OO sustavu poziva drugu
    - međuovisnost tipova
        - javlja se kada modul koristi podatkovni tip definiran u drugom modulu

- međuovisnost uključivanjem
  - javlja se kada komponenta importira paket
  - jedna komponenta uključuje drugu
- vanjska međuovisnost
  - predstavlja ovisnost modula o OS, biblioteci, HW...

4. **zadrži (višu) razinu apstrakcije**
   - osigurati da oblikovanje omogući sakrivanje ili odgodu razmatranja detalja te na taj način smanji složenost
   - omogućava razumijevanje suštine podsustava bez poznavanja nepotrebnih detalja
   - u OO oblikovanju: klase/razredi su podatkovne apstrakcije koje sadrže proceduralne apstrakcije (metode), a apstrakcija se povećava korištenjem private varijabli, nasljeđivanja, sučelja
5. **povećaj ponovnu uporabivost**
   - oblikovanje različitih aspekata sustava tako da može pridonijeti ponovnoj uporabi
   - poopćavanje oblikovanja u što većoj mjeri
   - oblikuj sustav tako da sadrži kopče/sučelje koje omogućava pristup u program dodatnom korisničkom kodu (korisnik vidi kopče kao otvore u kodu koji su dostupni u trenutku pojave nekog događaja ili zadovoljavanja nekog uvjeta)
6. **povećaj uporabu postojećeg**
   - što veća aktivna ponovna uporaba komponenti
   - korištenje prethodnih investicija
7. **oblikuj za fleksibilnost**
   - aktivno predviđaj buduće moguće promjene i provedi pripremu za njih
     - smanji povezivanje
     - stvaraj apstrakcije
     - ne upotrebljavaj izravno umetanje podataka u kod
     - ostavi otvorene opcije za modifikacije
     - upotrebljavaj postojeći kod
8. **planiraj zastaru**
   - planiraj promjene u tehnologiji ili okolini na taj način da program može raditi ili biti jednostavno promijenjen
   - izbjegavaj SW/HW bez izgleda za dugotrajniju podršku
9. **oblikuj za prenosivost**
   - omogući rad na što većem broju različitih platformi
10. **oblikuj za ispitivanje**
    - olakšaj ispitivanje
    - omogući odvojeno pokretanje svih funkcija uporabom vanjskih programa
11. **oblikuj konzervativno**
    - ne koristiti pretpostavku kako će netko koristiti oblikovanu komponentu
    - obradi sve slučajeve u kojima se komponenta može neprikladno upotrijebiti
12. **oblikuj po ugovoru**
    - tehnika koja omogućava efikasan i sistemski pristup konzervativnom oblikovanju
    - ugovaratelj ima skup zahtjeva:
      - preduvjete koje mora ispuniti pozvana metoda kada započinje izvođenje
      - uvjete koje pozvana metoda mora osigurati kod završetka izvođenja
      - invarijante na koje pozvana metoda neće djelovati pri izvođenju

Dokumenti oblikovanja arhitekture programske potpore

- dokumentacija potrebna zbog rane analize sustava
- ključ za održavanje, poboljšanje i izmjene PP
- temelj obilježja kvalitete
- dokumentacija ne zastarijeva
- minimalna dokumentacija arhitekture:
    - referentna specifikacija – potpuni skup dokumentiranih pokretača arhitekture, pogleda te pomoćne dokumentacije
    - pregled za upravu – pregled visokog nivoa, vizija sustava, poslovni motivi, koncepti arhitekturnih dijagrama...
    - dokumentacija komponenti – pogled na nivou sustava, specifikacija komponenti, sučelja
- struktura dokumenta oblikovanja:
    - svrha
    - opći prioriteti
    - skica sustava
    - temeljna pitanja u oblikovanju
    - detalji oblikovanja
- dokumentacija mora biti:
    - dobra – tehnički ispravna i jasno prezentirana
    - ispravna – doseže potrebe i ciljeve ključnih dionika
    - uspješna – upotrebljava se u stvarnom razvoju sustava kojm se postižu strateške prednosti

MODULARIZACIJA I OBJEKTNO USMJERENA ARHITEKTURA

- problemi u oblikovanju programske potpore:
    - ranjivost na globalne – široko dijeljene varijable
    - nenamjerno otkrivanje interne strukture
    - prodiranje odluka o oblikovanju
    - disperzija koda koji se odnosi na jednu odluku
    - povezane odluke o oblikovanju
- modularizacija:
    - moguće rješenje problema:
        - jednostavnije upravljanje sustavom (podijeli i vladaj)
        - evolucija sustava (promjena jednog dijela ne utječe na druge dijelove)
        - razumijevanje (sustav se sastoji od razumno složenih dijelova)
    - modul = dio koda; jedinica kompilacije koja uključuje deklaracije i sučelje
    - povijest modularizacije:
        - glavni program i potprogrami
            - hijerarhijska dekompozicija u procesne korake s jednom niti izvođenja
            - hijerarhijsko rasuđivanje – ispravno izvođenje programa ovisi o ispravnom izvođenju podprograma koji se poziva
            - implicitna struktura podsustava – podprogrami tipično skupljeni u module
        - funkcijski moduli
            - agregacija procesnih koraka u module

- apstraktni tipovi podataka (ADT)
    - skup dobro definiranih elemenata i skup pridruženih operacija na tim elementima definiranih s matematičkom preciznošću neovisno o implementaciji
    - skup elemenata jedino dohvatljiv preko skupa operacija
    - skup operacija je sučelje
    -
- objektni i objektno usmjerena arhitektura
- komponente i oblikovanje zasnovano na komponentama (CBD)
    - višestruka sučelja, posrednici, binarna kompatibilnost

Objektno usmjerena paradigma

- tehnika mmmodeliranja koja promatra svijet kroz objekte
- organiziranje proceduralnih apstrakcija u kontekstu podatkovnih apstrakcija
- pristup rješenju problema u kojem se sva izračunavanja obavljaju u kontekstu objekata
- objekti su instance programskih konstrukcija koje nazivamo razredi (klase)
- razred
    - podatkovna apstrakcija
    - sadrži proceduralne apstrakcije koje izvode operacije na objektima
- program u radu -> skup objekata koji u međusobnoj kolaboraciji obavljaju dani zadatak
- koncepti objektnog usmjerenja = nužna obilježja koja definiraju sustav ili programski jezik tako da bismo ga smatrali objektno usmjerenim
    - **objekt** (apstrakcija)
        - svaki objekt je jedinstven i može se referencirati (adresom)
        - 2 objekta mogu imati iste podatke, ali su jedinstveni
        - rezultat instanciranja razreda (instanciranje = proces preuzimanja perdloška i definiranja svih pridruženih atributa i ponašanja)
        - stvaranje objekta: operator 'new' + konstruktor razreda: new Ball();
        - pri instanciranju se vraća referenca na objekt: Ball b = new Ball();
        - može predstavljati sve iz stvarnog svijeta čemu se mogu pridružiti *obilježja* i *ponašanje*
        - svojstva objekta: stanje, ponašanje, jedinstvena identifikacija
        - interakcija objekata – razmjenom poruka
    - **razredi** (apstraktni tip podataka)
        - programski kod koji je organiziran uporabom koncepta razreda, koji svaki za sebe opisuje skup objekata
        - jedinica apstrakcije u objektno usmjerenoj paradigmi
        - sadrži *obilježja* (opis strukture instanca) i *metode* koje implementiraju ponašanje objekata
    - **nasljeđivanje** (ponovna uporaba)
        - mehanizam u kojem se značajke podrazreda implicitno nasljeđuju od superrazreda
    - **polimorfizam** (dinamičko povezivanje)
        - mehanizam u kojem postoji više metoda istog naziva koje različito implementiraju istu apstraktnu operaciju
- osnovni principi objektnog usmjerenja:
    - **apstrakcija**
        - olakšava savladavanje složenih problema
        - objekt -> nešto iz realnog svijeta; razred -> objekt; superrazred -> podrazred; operacija -> metode; atributi i pridruživanje -> varijable instanci

- **enkapsulacija**
  - detalji mogu biti skriveni u razredima
  - potiče skrivanje informacija
- **modularnost**
  - program moguće oblikovati samo iz razreda
- **hijerarhija**
  - elementi istog hijerarhijskog nivoa moraju biti na istom nivou apstrakcije

- razlika instanca – objekt:
  - nema razlike, odnosi se na istu jedinku
  - objekt = memorija koja sadrži informacije o objektu
  - instanca = referenca na objekt (pokazuje na početnu adresu na kojoj je objekt pohranjen)
    - 2 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
- **varijable instanci**:
  - definirane unutar razreda
  - varijabla je mjesto gdje se smještaju podaci
  - varijable definirane unutar razreda odgovaraju podacima koji se nalaze u svakoj instanciji toga razreda
  - 2 skupine varijabli instanci:
    - atributi (jednostavni podaci; obilježja objekata)
    - asocijacije između instanci različitih razreda (odnosi prema drugim instancama drugih razreda)
- varijabla – objekt
  - zasebni i različiti koncepti
  - tip varijable određuje koje razrede objekata može sadržavati
    - primitive: sadrži jednu vrijednost, nije objekt, evaluira se u vrijednost koju sadrži
    - reference: slično pokazivaču, ali u širem kontekstu, u različitim trenucima može se odnositi na različite objekte
  - jedan objekt može se u isto vrijeme referencirati u više različitih varijabli
    - objekt je dostupan preko varijable koja ga referencira
  - primjer: 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).
    - Sada želimo kreirati objekt iz razreda Ball i želimo da varijabla b1 referencira baš taj objekt:
      ```
      Ball b1 = new Ball();
      ```
      ➔ rutina Ball() konstruira objekt iz razreda Ball. Zove se "konstruktor"
    - Taj isti objekt može biti referenciran i od druge varijable:
      ```
      Ball b2 = b1;
      ```
      ➔ obje varijable referenciraju isti objekt iz razreda `Ball`.
- **varijable razreda**
  - identificiraju se ključnom riječi *static*
  - varijabla razreda može sadržavati vrijednost, ali tu vrijednost dijele sve instance tog razreda – ako jedna instanca upiše vrijednost u varijablu razreda, sve instance toga razreda vide izmijenjenu vrijednost → uvijek postoji samo jedna vrijednost te varijable

    - varijabli razreda se pristupa preko naziva razreda (ne preko objekta, tj varijable koja ga referencira)
- **metode razreda**
    - metode definirane unutar nekog razreda i deklarirane ključnom riječi *static*
    - metode se pozivaju preko naziva razreda, a ne preko objekta (tj varijable koja ga referencira)
- **lokalne varijable**
    - doseg je unutar zagrada procedure (metode)
    - vidljiva je samo u metodi u kojoj je deklarirana (nije joj moguće pristupiti iz ostatka razreda)
- **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
    - pozivaju se razmjenom poruka
    - nekoliko različitih razreda može imati metodu istog naziva
- **operacija**
    - proceduralna apstrakcija više razine nego metoda
    - specificira tip ponašanja
    - neovisna je o kodu koji implementira njeno ponašanje
    - vidljivost:
        - public (**+**) – dostupna svima
        - protected (**#**) – dostupna unutar hijerarhije razreda u kojem je deklarirana
        - private (**-**) – dostupna unutar razreda u kojem je deklarirana
- **višestrukost metoda / overloading**
    - postojanje više metoda istog naziva, ali različitog broja, tipova i mjesta parametara
- **nasljeđivanje**
    - podržavanje principa ponovne uporabe objekata
    - razredi mogu nasljeđivati značajke drugih podrazreda
    - svi podrazredi implicitno posjeduju značajke koje su definirane u superrazredu
    - LSP (Liskov princip zamjene)
        - ako postoji varijabla čiji tip je superrazred, program se mora korektno izvoditi ako se u varijablu pohrani instancija tog superrazreda ili instancija bilo kojeg podrazreda (podrarazredi nasljeđuju sve od superrazreda)
    - prednosti:
        - apstrakcija pogodna za organizaciju
        - ponovna uporaba u oblikovanju i implementaciji
        - organizacija znanja o domeni i sustavu
    - nedostaci:
        - razredi nisu samodostatni i ne mogu se razumjeti bez poznavanja superrazreda
        - nasljeđivanja uočena u fazi analize mogu dati neefikasna rješenja
- **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)
    - ako neka operacija nema implementacije, cijeli razred je „apstraktan“ i ne može kreirati instance – suprotno, ako postoje sve implementacije razred je „konkretan“
    - ključne riječi: *abstract* ili *virtual*

    - ako superrazred ima apstraktnu operaciju, tada na nekoj nižoj razini hijerarhije mora postojati konkretna metoda za tu operaciju – krajnji razredi moraju biti konkretni
- **polimorfizam**
    - moć poprimanja više oblika
    - svojstvo objektno usmjerenog programa da se jedna apstraktna operacija može izvesti na različite načina u različitim razredima
    - zahtjeva da postoji više metoda istog naziva
- **sučelje**
    - formalizirani polimorfizam
    - podržava „plug-and-play“ koncept
    - ne prestavlja apstraktni razred – ne pruža opis ponašanja metoda
- **nadjačavanje / overriding**
    - vezano uz hijerarhiju i nasljeđivanje razreda
    - iako je metoda definirana u superrazredu i može se naslijediti, redefinira se u podrazredu (podrazred sadrži inačicu metode)
    - koristi se za: restrikciju, proširenje, optimizaciju
- **dinamičko povezivanje**
    - odluka o izvođenju konkretne metode donosi se za vrijeme izvođenja programa
    - potrebno u slučaju:
        - varijabla je deklarirana da je tipa superrazred
        - postoji više polimorfnih metoda koje se mogu izvesti u sklopu hijerarhije razreda određene superrazred tipom varijable
- **konstruktori i destruktori**
    - posebne metode koje se pozivaju kada se ili kreira ili definira objekt
    - konstruktor inicijalizira objekt i njegove varijable
    - naziv konstruktorske metode je isti kao i naziv njenog razreda

Proceduralna	**OO**
Dekompozicija problema u funkcije	Dekompozicija u skupove objekata
Odvojeno modeliranje podataka i funkcija	Podaci i povezane opreracije na jednom mjestu
Velika međuovisnost komponenti – teškoće održavanja	Neovisnost komponenti
Komponente slabo odgovaraju stvarnom problemu – teško u slučaju rješavanja složenih problema	Blisko ljudskom rješavanju složenih problema
Često neprilagodljiv i linearan proces razvoja	Pogodno za iterativan i inkrementalan razvoj

OBJEKTNO USMJERENA ARHITEKTURA

RASPODIJELJENI SUSTAVI:

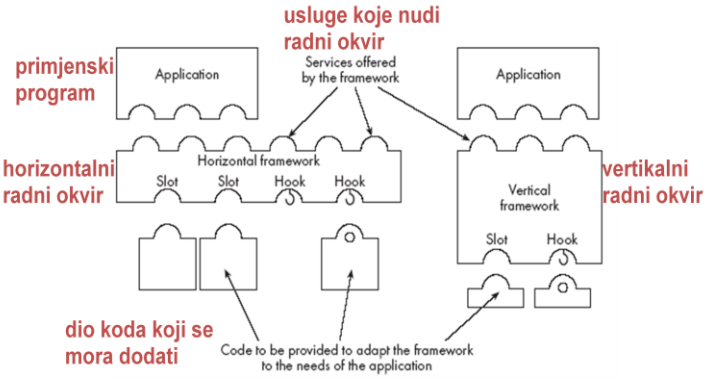
- obradu podataka i izračunavanja obavljaju odvojeni programi (ugl na odvojenom sklopovlju) koji međusobno komuniciraju po računalnoj mreži
- primjeri:
    - Klijent – poslužitelj
        - poslužitelj (server) -> program koji dostavlja uslugu drugim programima koji su spojeni na njega preko komunikacijskog kanala
        - klijent -> program koji pristupa poslužitelju/ima tražeći uslugu (više klijenata može istovremeno pristupiti istom poslužitelju)
    - Peer-to-Peer (P2P)
        - svaki čvor u sustavu ima jednake mogućnosti i odgovornosti (istovremeno i klijent i server)
        - snaga obrade podataka i izračunavanja ovisi o pojedinim i krajnjim čvorovima, a ne o skupnom radu
    - Afinitetna (društvena) mreža
        - jedan korisnik se povezuje s drugim u cilju razmjene informacija
    - Kolaborativno izračunavanje
        - neiskorišteni resursi mnogih računala (CPU vrijeme, prostor na disku...) kombiniraju se u izvođenju zajedničkog zadatka (GRID computing...)
    - Instant Messaging
        - izmjena tekstovnih poruka između korisnika u stvarnom vremenu

Arhitektura klijent – poslužitelj

- sekvenca aktivnosti
    1. Poslužitelj započinje s radom.
    2. Poslužitelj aktivira slušanje i čeka na dolazak klijentskog zahtjeva (*poslužitelj sluša*).
    3. Klijenti započinju s radom i obavljaju razne operacije,
    4. Neke operacije traže zahtjeve (i odgovore) s poslužitelja.
    5. Kada klijent pokuša spajanje na poslužitelja, poslužitelj mu to omogući (ako poslužitelj želi).
    6. Poslužitelj čeka na poruke koje dolaze od spojenih klijenata.
    7. Kada pristigne poruka nekog klijenta poslužitelj poduzima akcije kao odziv na tu poruku.
    8. Klijenti i poslužitelj nastavljaju s navedenim aktivnostima sve do odspajanja ili prestanka rada.
- prednosti:
    - posao se može raspodijeliti na više računala
    - klijenti udaljeno pristupalju funkcionalnostima poslužitelja
    - klijenti i poslužitelj mogu se oblikovati odvojeno
    - svi podaci mogu se držati na jednom mjestu (server) i distribuirati na više udaljenih mjesta
    - više klijenata može simultano pristupati poslužitelju
    - klijenti mogu ući u natjecanje za uslugu servera (i obrnuto)

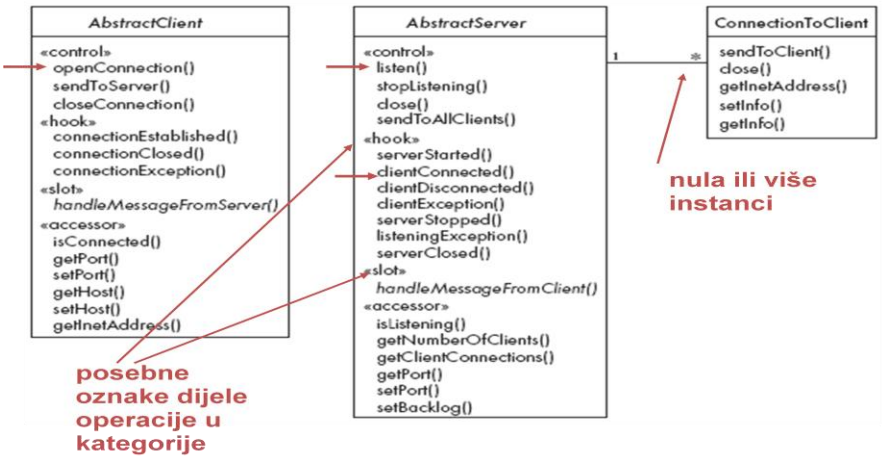
- funkcionalnost poslužitelja:
    1. poslužitelj se inicijalizira
    2. započinje slušati
    3. rukuje sljedećim tipovima događaja klijenata:
        - prihvaća spajanje (rukuje spajanjem i reagira na poruku)
        - odgovara na poruke
        - rukuje odspajanjem klijenata
    4. može prestati slušati
    5. mora čisto završiti rad
- funkcionalnost klijenta:
    1. klijent se inicijalizira
    2. inicijalizira spajanje na poslužitelja
    3. interakcija s korisnikom i šalje poruke na poslužitelj
    4. rukuje sljedećim tipovima događaja poslužitelja:
        - odgovara na poruke
        - rukuje odspajanjem poslužitelja
    5. mora čisto završiti rad
- **tanki klijent**
    - klijent oblikovan da bude što je moguće manji i jednostavniji
    - većina posla obavlja se na poslužitelju
    - oblikovna struktura klijenta i izvršni kod jednostavno se preuzimaju preko računalne mreže
- **debeli klijent**
    - što je moguće više posla obavlja se na klijentima
    - poslužitelj može rukovati s više klijenata
- internetski protokoli (pravila konverzacije kako bi poslužitelj razumio jezik na kojem mu klijent šalje poruke i obrnuto)
    - IP = određuje put poruka od jednog do drugog računala temeljem IPv4 ili IPv6 adrese
    - TCP = između aplikacijskog sloja i IPa; razbija poruku na manje dijelove i šalje ih uporabom IP protokola osiguravajući ispravnost primljene poruke (pouzdaniji od UDPa)
    - DNS = preslikavanje jedinstvene IP adrese računala u simboličko ime (host name)
        - nekoliko poslužitelja može raditi na jednom čvornom računalu u mreži, ali je svaki jedinstveno određen preko ulaznog porta (0 – 65535)
        - za početak komunikacije s poslužiteljem, klijent mora znati IP i port servera
        - 0 – 1023 su rezervirani portovi (dobro poznata vrata, npr 80 za web)
- oblikovanje klijent – poslužitelj arhitekture:
    - specificiraj temeljne poslove klijenta i poslužitelja
    - odredi kako će se posao raspodijeliti (tanki ili debeli klijent)
    - oblikuj detalje skupa poruka koje se razmjenjuju (protokoli)
    - oblikuj mehanizme:
        - incijalizacije
        - rukovanje spajanjima
        - slanje/primanje poruka
        - završetak rada
    - u oblikovanju koristi princip „uporabi postojeće gotove komponente)
- radni okviri (framework):
    - učestalo korišten dio programske potpore koji implementira generičko rješenje problema
    - osigurava opća sredstva koja se mogu uporabiti u različitim primjenskim programima

- OO paradigma: radni okvir sastoji se od knjižice razreda i sadrži:
  - primjensko sučelje (API) – skup svih javnih metoda tih razreda
  - apstraktni razredi - moraju se implementirati u podrazredima
- vrste:
  - **horizontalni radni okvi** – osigurava veći broj općih i zajedničkih sredstava koja mogu koristiti mnogi primjenski programi (aplikacije)
  - **vertikalni radni okvir** (primjenski) – cjelovitiji, ali još uvijek traži dopunu nekih nedefiniranih mjesta kako bi se prilagodio specifičnoj primjeni



- **linije proizvoda** = skup svih produkata izrađenih na zajedničkoj osnovnoj tehnologiji
  - različiti produkti u liniji proizvoda imaju različite značajke kako bi zadovoljili različite segmente tržišta
  - programska tehnologija zajednička svim proizvodima u liniji proizvoda uključena je u radni okvir
    - svaki proizvod izrađen je temeljen radnog okvira u kojem su popunjena odgovarajuća prazna mjesta (iz jednog okvira može nastati više linija proizvoda)

- **<u>uporaba objektnog klijent – poslužitelj radnog okvira (OCSF)</u>**
  - postupak uporabe:
    - ne mijenjati apstraktne razrede u OCFS
    - kreirati podrazrede
    - konkretizirati metode u podrazredima
    - nanovo definirati (override) neke metode u podrazredima (neke u hook i slot)
    - napisati kod koji kreira instance i inicira akcije



  - nikada ne modificirati tri navedena razreda!

- klijent = **AbstractClient**

  - paralelne aktivnosti na klijentskoj strani izvođene kao višestruke niti izvođenja (dretve):
    - čekanje na interakciju s korisnikom i odgovor na interakciju
    - čekanje na poruku poslužitelja i reakcija kada poruka stigne
  - u Javi dretva je objekt razreda Thread i uvijek se mora stvoriti i pokrenuti:
    dretva=new Thread(); //stvaranje
    dretva.start(); //pokretanje (metoda koja dalje pokreće run() koja je main metoda u dretvi)
  - iz razreda AbstractClient moraju se izvesti podrazredi
    - svaki podrazred mora osigurati implementaciju operacije: **handleMessageFromServer()**
  - AbstractClient se spaja i šalje poruke poslužitelju
  - za čitanje poruka postoji posebna dretva koja započinje izvođenje nakon što metoda **openConnection()** pozove start() dretve clientReader koja pokreće njenu run() metodu
    - run() sadrži petlju koja se izvodi tijekom životnog ciklusa dretve (prima poruke i zove metodu za rukovanje porukama)
    - openConnection:

```
// final = ne redefinirati
final public void openConnection(){
// priprema i otvaranja veze prema serveru
// kreira objekt "socket" za vezu prema serveru
// parametri host i port servera u konstruktoru
// ili se mogu postaviti posebnim metodama
 clientSocket = new Socket(host, port);
// kreiraj objekte ua izlaz i ulaz podataka
 output = new
  ObjectOutputStream(clientSocket.getOutputStream());
 input = new
  ObjectInputStream(clientSocket.getInputStream());
// varijable output i input su tipa-razreda objekata
// ObjectOutputStream i ObjectInputStream
// kreiraj dretvu clientReader za čitanje poruka
// s poslužitelja
// varijabla clientReader referencira tu dretvu
  clientReader = new Thread(this);
// makni zabranu
  readyToStop = false;
// startaj dretvu koja aktivira run metodu
  clientReader.start();
}
```

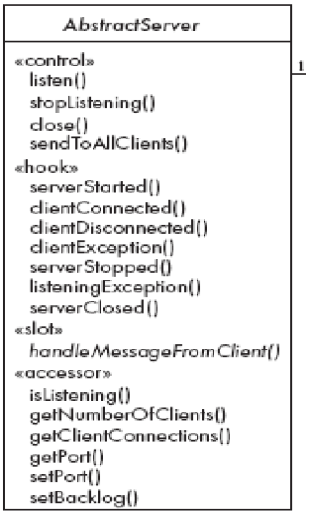
- slanje poruka i završetak:

```
public void sendToServer(Object msg)
{
output.writeObject(msg); }
// writeObject je definiran u razredu
// ObjectOutputStream
// output preko clientSocket zna kome šalje
final public void closeConnection() {
   readyToStop = true;
   closeAll(); }
// closeAll() je implementirana u "frameworku"
```

- privatni dijelovi razreda AbstractClient
  - varijable instanci
    - **clientSocket** – sadrži sve informacije o vezi s poslužiteljem
    - **ObjectOutputStream** i **ObjectInputStream** – 2 niza koja se koriste za slanje i prijam objekata msg uporabom varijeble **clientSocket**
    - dretva **clientReader** tipa Thread – započinje kada openConnection pozove start() koja inicira run()
    - varijabla boolean **readyToStop** – signalizira zaustavljanje dretve čitanja poruka servera
- javno sučelje AbstractClient
  - konstruktor AbstractClient() – pri stvaranju obketa inicijalizira host i port varijable poslužitelja na koje će se klijent spojiti
  - <<control>> metode
    - **openConnection()** – spoj na poslužitelj, koristi host i port dobivene konstruktorom ili preko **setHost() setPort()**, uspješna veza starta dretvu **clientReader**
    - **sendToServer()** – šalje poruku koja može biti bilo koji objekt
    - **closeConnection()** – zaustavlja rad dretve u petlji i završava
  - <<accessor>> metode (daju info ili mijenjaju vrijednost)
    - **isConnected()** (ispituje je li je klijent spojen)
    - **getHost()** (ispituje koji ***host*** – server je spojen)
    - **setHost()** (omogućuje promjenu ***hosta*** - dok je klijent odspojen)
    - **getPort()** (ispituje na koji ***port*** servera je klijent spojen)
    - **setPort()** (omogućuje promjenu ***porta*** dok je klijent odspojen).
    - **getInetAddress()** (dobavlja neke detaljnije informacije o vezi)
- **Callback** metode u AbstractClient
  - metode koje se mogu redefinirati
  - označene su kao <<hook>> skupina
  - **connectionEstablished()** (aktivira se uspostavom veze s poslužiteljem)
  - **connectionClosed()** (aktivira se nakon završetka veze)
  - **connectionException()** (aktivira se kada nešto pođe krivo, npr. poslužitelj prekida vezu).
  - općenito funkcije koje se ne zovu izravno, najčešće kao odziv na neki asinkroni događaj
- uporaba razreda AbstractClient
  - kreiraj podrazred od AbstractClient
  - implementiraj handleMessageFromServer()
  - napiši kod koji:
    - kreira instancu klijenta (start 1. dretve)
    - pozove openConnectio() (start 2. dretve)
    - šalje poruku serveru uporabom sendToServer()

- implementiraj connectionEstablished()
- implementiraj connectionClosed()
- implenentiraj connectionExepction()

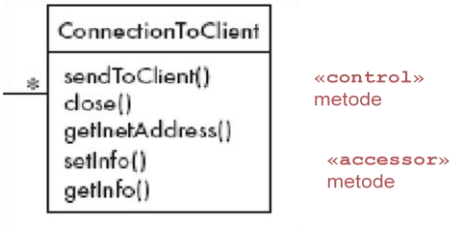
- server
  - sadrži 2 razreda
  - potrebne min 2 niti izvođenja:
    - jedna koja sluša novo spajanje klijenta
      - instanca razreda **AbstractServer**
      - rukuje s porukom
      - ne šalje poruku pojedinom klijentu, ali može poslati istu poruku svim spojenim klijentima metodom **sendToAllClients()**
    - jedna (ili više) koja rukuje vezama s klijentima
      - jedna ili više instani razreda **ConnectionToClient**
  - **AbstractServer**



    - javno sučelje AbstractServer
      - **AbstractServer()** – konstruktor, stvara objekt s brojem porta na kojem poslužitelj sluša (kasnije se može promijeniti **setPort()** metodom)
      - <u><<control>> metode</u>:
        - **listen()** – kreira varijablu **serverSocket()** tipa ServerSocket koja će slušati na portu specificiranom konstruktorom, kreira i inicira dretvu, a run() metoda dretve čeka na spajanje klijenta
        - **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, jedina metoda iz ove skupine koja se može redefinirati
      - <u><<accessor>> (pristupne) metode</u> (ispituju stanje servera):
        - **isListening()** – vraća informaciju o tome sluša li poslužitelj
        - **getClientConnections()** – vraća niz instanci razreda za vezu s klijentima ConnectionToClient, 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 klijenata

- ostale metode u AbstractServer
  - metoda koja se mora implementirati
    - **handleMessageFromClient()** – argumenti: primljena poruka + tko šalje (instanca razreda ConnectionToClient)
  - **Callback** metode koje se mogu redefinirati (aktiviraju se kao odziv na neki događaj)
    - **serverStarted()** – aktivira se kada poslužitelj započinje prihvaćati spajanja
    - **clientConnected()** – aktivira se kada se novi klijent spoji
    - **clientDisconnected()** – aktivira se kada poslužitelj odspoji klijenta, argument je instanca razreda **ConnectionToClient**
    - **clientException()** – aktivira se kada se klijent sam odspoji ili kada nastupi kvar u mreži
    - **serverStopped()** – aktivira se kada poslužitelj prestane prihvaćati povezivanje s klijentima, a kao rezultat **stopListening()**
    - **listeningException()** – aktivira se kada poslužitelj prestane slušati zbog nekog kvara
    - **serverClosed()** – aktivira se nakon završetka rada poslužitelja

- **ConnectionToClient**



- početna aktivacija **listen()** u AbstractServer pokreće dretvu koja u svojoj run() metodi preko metode accept() „socketa“ poslužitelja čeka na spajanje i kreira objekt/instancu razreda ConnectionToClient (s pripadnom dretvom) – po jedan za svako spajanje klijenta
- javno sučelje ConnectionToClient:
  - za svakog klijenta dok je spojen na poslužitelja kreira se instanca razreda ConnectionToClient -> to je konkretan razred i korisnici ne moraju kreirati podrazrede
  - <u><<controll>> metode:</u>
    - **sendToClient()** – središnja metoda, koristi se za komunikaciju s klijentom
    - **close()** – odspaja klijenta
  - <u><<accessor>> metode:</u>
    - **getInetAddress()** – dobavlja Internet adresu klijenta
    - **setInfo()** – omogućuje spremanje proizvoljnih informacija o klijentu (npr posebne privilegije za neke klijente)
    - **getInfo** – omogućuje čitanje informacija spremljenih sa setInfo()

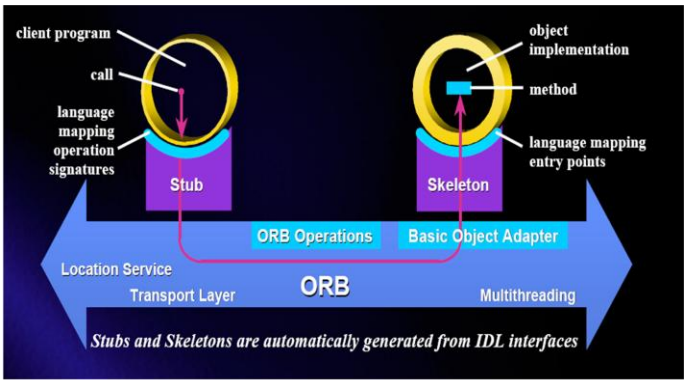
- uporaba AbstractServer i ConnectionToClient
  - kreiraj podrazred od AbstractServer razreda
  - u podrazredu implementiraj handleMessageFromClient()
  - napiši kod koji:
    - kreira instancu podrazreda AbstractServer
    - poziva listen() metodu
    - eventualno odgovara na „call back“ clientConnected() slanjem poruke uporabom metoda iz AbstractServer:
      - getClientConnections() ili sendToAllClients()
  - implementiraj jednu ili više drugih callback metoda kao odzive na zanimljive događaje
- sinkronizacija rada s više klijenata:
  - postoji više ConnectionToClient objekata koje mogu istovremeno mijenjati podatke na poslužitelju, sinkronizacija garantira da se kritične operacije odvijaju jedna po jedna (čuvanje integriteta podataka)
  - poslužitelj mora povremeno provjeriti je li pozvana metoda stopListening() jer je slušanje rad u samo jednoj dretvi
- rizici u primjeni klijent – poslužitelj arhitekture
  - sigurnost (potrebna enkripcija, zaštitni zidovi...)
  - potreba za adaptivnim održavanjem
    - sva programska potpora za klijenta i poslužitelja oblikuje se odvojeno pa je potrebno osigurati da sva programska potpora bude kompatibilna prema unatrag i prema unaprijed (također i s drugim verzijama klijenta i poslužitelja)
- principi oblikovanja:
  - ***Podijeli i vladaj***: podjela sustava na klijenta i poslužitelja je uspješan način optimalne podjele – klijent i poslužitelj mogu se oblikovati odvojeno.
  - ***Povećaj koheziju***: poslužitelj osigurava kohezijski spojenu uslugu klijentima
  - ***Smanji međuovisnost***: uobičajeno je da postoji samo jedan komunikacijski kanal preko kojega se prenose jednostavne poruke
  - ***Povećaj apstrakciju***: odvojene raspodijeljene komponente su dobar način povećanja apstrakcije
  - ***Povećaj uporabu postojećeg***: često je moguće pronaći odgovarajući radni okvir temeljem kojega se oblikuje raspodijeljeni sustav (klijent-poslužitelj arhitektura je često specifična obzirom na primjenu)
  - ***Oblikuj za fleksibilnost***: raspodijeljeni sustavi se često vrlo lako mogu rekonfigurirati dodavanjem novih poslužitelja ili klijenata
  - ***Oblikuj za prenosivost***: llijenti se mogu oblikovati za nove platforme bez promjene poslužiteljske strane
  - ***Oblikuj za ispitivanje***: klijenti i poslužitelji mogu se ispitivati neovisno
  - ***Oblikuj konzervativno***: u kod koji rukuje porukama mogu se ugraditi vrlo stroge provjere (npr. rukovanje iznimkama)

Posrednička arhitektura

- uvođenje više razina
  - klijenti i poslužitelji organiziraju se u razine (gornja se oslanja na donju, a donja pruža uslugu gornjoj)
  - razina enkapsulira skup usluga i implementacijske detalje niže razine u o kojoj ovisi
  - prednosti:
    - oblikovanje temeljem više razine apstrakcije

        - podupire lagano prošitenje i promjene sustava (promjena na jednoj razini utječe samo na razinu ispod i iznad)
        - podupire ponovno korištenje, prenosivost...
    - nedostaci
        - teško 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 tuneliti kroz razinu
- <u>trorazinska klijent – poslužitelj arhitektura</u>
    - generalizacija klijent – poslužitelj arhitekture
    - bolja skalabilnost i mogućnost jednostavnije modifikacije
    - nastoji ukloniti nedostatke klasične klijent – poslužitelj (povećati performanse, raspoloživost, sigurnost)

- <u>uvođenje posredničke razine</u>
    - između klijenta i psolužitelja
    - skriva osobama koje oblikuju raspdijeljeni sustav detalje OSa i druge specifičnosti implementacije i tako omogućuje da se usredotoče na primjenski dio
    - preuzima detalje komunikacijske mreže
    - lakše oblikovanje i razvijanje raspodijeljenog sustava
    - 3 vrste posredničke arhitekture
        - **transakcijski usmjerena** (komunikacija s bazama podataka)
        - **zasnovana na porukama** (pouzdana, asinkrona)
        - **objektno usmjerena** (sinkrona između raspodijeljenih objekata)
            - najpopularnija: CORBA , DCOM, DotNET
            - zasnovani na udaljenom pozivanju procedura (Remote Procedure Call RPC)
            - proširuju RPC uvođenjem mehanizama iz objektno usmjerene paradigme
            - specifična posrednička razina koja omogućuje interoperabilnost u heterogenim sustavim u raspodijeljenom okruženju
- CORBA
    - poslužiteljski objekti posjeduju sučelje koje je neovisno o programskom jeziku
    - klijent rukuje podacima i metodama u poslužiteljskom objektu samo preko sučelja (sveo stalo mu je skriveno)
    - klijent i poslužitelj mogu se programirati različitim programskim jezicima – jezična transparentnost na razini izvornog koda
    - klijent i poslužitelj ne brinu se o međusobnim lokacijama –lokalna transparentnost (mogu biti implementirani na različitim sklopovskim platformama i OSovima)
    - CORBA objekti mogu se relocirati po čvorovima u mreži
    - CORBA objekti su zatvoreni i pristupa im se samo preko dobro definiranog sučelja (sučelje i imeplementacija totalno razdvojeni)
    - „object reference“ – identitet objekta se proširuje dodatnim informacijama (npr host...)
    - kako bi se locirao objekt za vrijeme izvođenja, klijent mora znati njegov „reference“ – tekstovni niz kodiran na specifičan način i sadrži dovoljno informacija da se:
        - uputi zahtjev ispravnom poslužitelju (host, port)
        - locira ili kreira objekt (ime razreda, podaci/atributi)



Stub i **Skeleton** su dijelovi CORBA arhitekture koji preslikavaju programski jezik klijenta i poslužitelja u pozivanje odgovarajuće uslužne metode. Generiraju se automatski programiranjem sučelja IDL jezikom. IDL spaja i pomiruje različite objektne modele i programske jezike.

- principi oblikovanja i posrednička arhitektura:
  1. ***Podijeli i vladaj***: Udaljeni objekti mogu biti neovisno oblikovani.
  5. ***Povećaj ponovnu uporabu***: Moguće je i poželjno oblikovati udaljene objekte tako da ih i drugi sustavi mogu koristiti.
  6. ***Povećaj uporabu postojećeg***: Možeš koristiti objekte koje su drugi oblikovali.
  7. ***Oblikuj za fleksibilnost***: Posrednik (broker) se može ažurirati. Objekti se mogu zamijeniti.
  9. ***Oblikuj za prenosivost***: Možeš oblikovati klijente za novu platformu i još uvijek pristupati postojećim brokerima i udaljenim klijentima na drugim platformama.
  11. ***Oblikuj konzervativno***: Možeš rigorozno provjeriti nedvosmisleno ponašanje udaljenih objekata (Java: **try – catch** iznimke).

Uslužno usmjerena arhitektura

- organizira primjenski program kao kolekciju usluga koje međusobno komuniciraju uporabom dobro definiranih sučelja (web usluge ukoliko je riječ o Internetu)
- web usluga = primjenski program kojem se može pristupiti preko Interneta i koji se može integrirati s drugim uslugama na web-u kako bi se oblikovao cjeloviti sustav
- različite komponente u web uslužnom sustavu komuniciraju uporabom otvorenog standarda kao što je XML
- principi oblikovanja i uslužna arhitektura:
  1. ***Podijeli i vladaj***: Cijeli primjenski program sastoji 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, a 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

- poruke temeljem procesa oblikovanja programske potpore s fokusom na princip ponovnog korištenja dijelova koje se manifestira kroz:
  - Ponovno korištenje konzistencije (programski jezici).
  - Ponovno korištenje fragmenata rješenja (knjižnice).
  - Ponovno korištenje dijelova arhitekture (arhitekturni obrasci – *engl. architectural patterns, design patterns*).
  - Ponovno korištenje arhitekture (radni okviri – *engl. frameworks*).
  - Ponovno korištenje cjelokupne arhitekture sustava.

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

- **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 “plug-and-play”.
- temeljni problem u širokoj uporabi komponenata je nepostojanje zajedničke platforme koja objedinjuje komponente
- potencijalne tehnologije kao temelji oblikovanja programske potpore zasnovane na komponentama:
  - CORBA
    - **dobro**: standard (OMG grupa), transparentna komunikacija između objekata koji su oblikovani u različitim programskim jezicima i žive na različitim strojevima, u heterogenoj raspodijeljenoj okolini
    - **loše**: Definirani su na razini izvornog koda, a ne na binarnoj razini. To jako usporava rad jer se komunikacija odvija na visokoj razini definiranih protokola. Svi programski jezici moraju imati 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 (MSWindows).

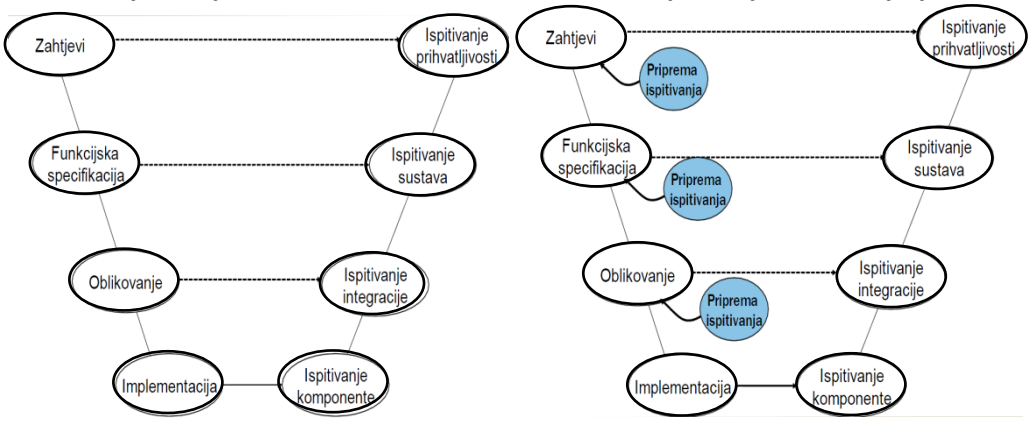
ISPITIVANJE PROGRAMSKE POTPORE

- ispitivanje je proces izvođenja programa sa svrhom pronalaženja pogrešaka
- ispitivanje programske podrške zasniva se na dinamičkoj verifikaciji ponašanja programa u izvođenju na konačnom broju ispitnih slučajeva, pogodno odabranih iz uobičajeno beskonačne domene izvočenja, obzirom na očekivano ponašanje
- ispitivanje mora omogućiti donošenje odluke o prihvatljivosti i očekivanim rezultatima (usporedba stvarnih rezultata s prethodno utvrđenim rezultatima na temelju specifikacija)
- **ispitni slučaj/test** – jedan ili više ispitnih slučajeva/scenarija
- **ispitivanje/testiranje** – proces analize programskog koda sa svrhom pronalaska razlike između postojećeg i zahtijevanog stanja te vrednovanja svojstva programa
- tehnike verifikacije programa:
    - dinamička verifikacija – ispitivanje/testiranje
    - statička verifikacija – ispitivanje strukture, provjera ispravnosti
    - formalna verifikacija – primjena formalnih metoda matematičke logike za dokaz ispravnosti programa
- statička verifikacija
    - provodi se na specifikaciji zahtjeva, raznim nivoima oblikovanja sustava i programskom kodu:
        - **nadzor izvornog koda**
            - analiza statičkih artefakata s ciljem otkrivanja problema
            - postupak davanja ekspertnih mišljenja, recenzija programskog produkta
                - ljudi provjeravaju izvorne artefakte bez izvođenja
            - ne može provjeravati nefunkcionalna svojstva
            - **prolazak/češljanje/walkthroughs**
                - neformalni nadzor i inspekcija izvornog koda ili dokumentacije, ugl indicirana od strane autora
            - **nadzor/inspekcija/software inspections**
                - svrha je utvrđivanje usklađenosti sa standardom ili zahtjevima
                - usporedba dokumenata oblikovanja, koda i ostalih artefakata
                - zahtjeva planiranje i raspodjelu zadaća, formalno bilježenje i obradu rezultata
        - **analizatori programa**
        - **formalne metode**
- ciljevi ispitivanja:
    - pronaći i ispraviti greške, osigurati pouzdanost, ispravnost
    - minimizirati rizike:
        - provjeriti sukladnost rada s ostalim komponentama
        - pomoći u donošenju odluke o puštanju u rad/prodaju – zaustaviti prerano puštanje
        - minimizirati troškove tehničke podrške
        - procijeniti sukladnost specifikacijama i normama

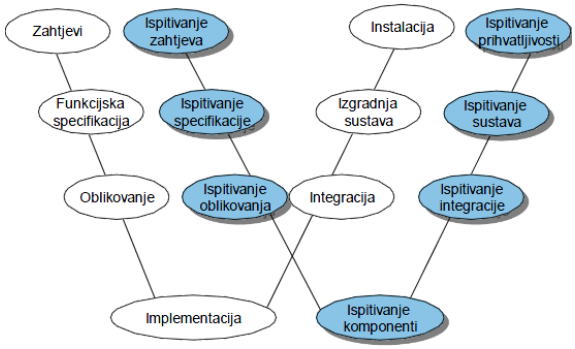
    - definirati način sigurne uporabe
    - procjena kvalitete
- kvar, pogreška, zatajenje:
    - **kvar** – fizikalni svijet
        - krivi rad pri oblikovanju ili programiranju
        - kvar specifikacije sučelja (nepodudranja formata klijenta i poslužitelja, nepodudaranje zahtjeva i implementacije)
        - kvar u algoritmima (nedostatak inicijalizacija, pogrešna grananja, zanemarivanje null vrijednosti)
        - mehanički kvar (dokumentacija nije sukladna stvarnom stanju)
        - aritmetički, logički, sintaksni, memorijski...
    - **pogreška** – informacija
        - manifestacija kvara, uvođenje kvara u programsku potporu
        - uzrokuje pogrešku obrade/izvođenja programa i dovodi do zatajenja
        - Paretov princip – mali broj pogrešaka, dovodi do velikog broja zatajenja
        - pogreška izazvana gubitkom poruka pri opterećenju, pogreška izazvana ograničenjima memorije, vremenske pogreške...
        - tipovi pogrešaka (od najmanjeg do najvećeg stupnja štete):
            - blage
            - dosadne
            - uznemiravajuće
            - ozbiljne
            - granične
            - katastrofalne
            - zarazne
        - kategorije: funkcijske, sistemske, podatkovne, pogreške kodiranja, projektiranja...
        - upravljanje pogreškama:
            - prevencija – uporaba pogodnih metoda oblikovanja za smanjenje složenosti, sprečavanje nekonzistentnosti, primjena verifikacije za sprječavanje kvarova u algoritmima
            - detekcija – tijekom rada (ispitivanje, ispravljanje pogrešaka, nadzor rada)
            - oporavak – u radu programa, baze podataka, modularna zalihotnost
    - **zatajenje** – vanjska pojavnost
        - mogući slučajevi: ne ispunjava očekivanje zahtjeva / korisnika
        - razlozi zatajenja:
            - zahtjevi nepotpuni, nekonzistentni, nemogući za implementaciju
            - pogrešna implementacija zahtjeva
            - kvar u oblikovanju arhitekture, programa, programskog koda
            - dokumentacija nekorektno opisuje ponašanje sustava
- evolucija ispitivanja po fazama:
    1. ispravljanje pogrešaka
    2. orijentacija na demonstraciju
    3. orijentacija na razaranje
    4. orijentacija na evaluaciju
    5. orijentacija na prevenciju
    6. orijentacija na razvoj programske potpore
- standardi ispitivanja:
    - standadrdi osiguranja kvalitete (koja ispitivanje je nužno provesti)
    - industrijski standardi (razine ispitivanja)

    - standardi ispitivanja programske potpore (kako provesti ispitivanj)
- tim za ispitivanje:
    - voditelj projekta – upravlja procesom ispitivanja i posebnim resursima
    - analitičar – analiza poslovnih procesa, zahtjeva i funkcijske specifikacije; planiranje ispitivanja
    - voditelj upravljanja kvalitetom – definiranje standarda ispitivanja, praćenje i osiguranje sukladnosti procesa ispitivanja
    - voditelj ispitivanja – analiza zahtjeva ispitivanja; oblikovanje strategije i metodologije ispitivanja, ispitnih slučajeva i podataka
    - profesionalni ispitivači – priprema i provođenje ispitivanja, pronalaženje pogrešaka
    - korisnici
- <u>aktivnosti ispitivanja</u>
    - osnovne kategorije:
        - **oblikovanje ispitivanja** – oblikovanje ispitnih vrijednosti sa svrhom zadovoljenja ciljeva ispitivanja
        - **automatiziranje ispitivanja** – programiranje
        - **ispitivanje** – provođenje ispitivanja i bilježenje rezultata
        - **valorizacija ispitivanja** – poznavanje domene i postupaka ispitivanja
    - **planiranje ispitivanja** – planiranje procesa, aktivnosti; raspoređivanje resursa; utvrđivanje zahtjeva, strategija i alata
    - **oblikovanje ispitnih slučajeva**
    - **izrada ispitnih slučajeva**
    - **provođenje ispitivanja** – ručno + automatizirano
    - **analiza rezultata i izvješćivanje**
    - **upravljanje ispitivanjem** – upravljanje aktivnostima, kontrola plana, praćenje troškova
    - **automatizacija ispitivanja** – definiranje, razvoj i prilagodba alata
    - **upravljanje ispitivanjem** – upravljanje i dokumentiranje životnim ciklusom ispitnih slučajeva, inačicama ispitnih okolina i sl.
- <u>regresijsko ispitivanje</u> – ponovljeno ispitivanje nakon popravka/promjene s ciljem potvrđivanja ispravnosti promjena i ne postojanja negativnog utjecaja na nepromijenjene dijelove programa (provodi se tijekom integracije i nakon nadogradnji)
- <u>plan ispitivanja:</u>
    - opis procesa
    - sljedivost zahtjeva – sustavno praćenje pojedinih zahtjeva i njihovih promjena
    - elementi ispitivanja – što ispitujemo, a što ne + razlozi
    - vremenik ispitivanja – strategija, prioriteti, resursi...
    - procedura bilježenja rezultata – baza podataka, elementi, automatizacija...
    - zahtjevi okoline – HW, SW
    - ograničenja – opis planiranog ograničenja ispitivanja
- ispravljanje je najjeftinije u ranim fazama
- ispitivanje je objedinjeno u procesu razvoja i održavanja (ne samo nakon završetka)

- modeli ispitivanja
  - **v-model ispitivanja** **v-model ispitivanja s ranom pripremom**



  - **w-model ispitivanja**



- svojstva ispitljivih programa (ispitljivost=mjera mogućnosti jednostavnog ispitivanja programa):
  - **osmotrivost** – jednostavna identifikacija rezultata, različiti izlazi za različite ulaze
  - **upravljivost** – jednostavnost upravljanja tijekom provođenja ispitivanja (automatizacija, ponovna uporabivost)
  - **dekompozicija** – neovisno ispitivanje modula
  - **jednostavnost** – složenost arhitekture, logike programa i kodiranja
  - **stabilnost** – promjene programa tijekom ispitivanja utječu samo na rezultate provedenih ispitivanja
  - **razumljivost** – dobre informacije o strukturama, međuovisnostima i organizaciji tehničke dokumentacije
- uporaba informacija u ispitivanju:
  - specifikacije (formalne, neformalne; za odabir, generiranje, provjeru)
  - informacije oblikovanja (za odabir, generiranje, provjeru)
  - programski kod (za odabir, generiranje, provjeru)
  - uporaba (povijest, model)
  - iskustvo ispitivanja
- osnovni koraci ispitivanja:
  1. Što ispitivati?
     - kompletnost zahtjeva, oblikovanje ispitivanja, imeplementacija
  2. Kako ispitivati?
     - statička verifikacija
     - odabir ispitivanja komponenti (black box/white box)
     - odabir integracijskog ispitivanja (big bang, bottom up, top down, sandwich)
  3. Razvoj ispitnih slučajeva
  4. Predikcija rezultata
- ispitivati prije kodiranja + za vrijeme kodiranja + nakon kodiranja

- najbolje koristiti neovisne timove za ispitivanje (programeri nesvjesno koriste one podatke na kojima sustav radi + sustav najčešće ne radi kada ga upotrebljava netko drugi
- terminologija:
    - ispitni podaci (I) – ulazi odabrani za provođenje određenog ispitnog slučaja
    - očekivani izlaz (O) – zabilježen **prije** provođenja ispitivanja
    - ispitni slučaj (I,O) – uređeni par koji sadrži opis stanja prije ispitivanja
    - stvarni izlaz – rezultat dobiven provođenjem ispitivanja
    - kriterij prolaza ispitnog slučaja – kriterij usporedbe očekivanog i dobivenog izlaza određen prije provođenja ispitnog slučaja
- oblikovanje ispitnog slučaja – obuhvaća određivanje ulaza i odgovarajućeg rezultata kojeg upotrebljavamo za ispitivanje sustava
    - cilj: otkrivanje pogrešaka
    - kriterij: kompletnost
    - ograničenje: minimum resursa i vremena
- <u>pristupi ispitivanja programske podrške</u>

  (prva 3 obzirom na scenarij ispitivanja, druga 2 obzirom na izvor informacije)
    - **ispitivanje zasnovano na pokrivenosti**
        - sve narebne moraju biti izvršene barem jednom
        - zahtjevi ispitivanja specificirani obzirom na pokrivenost ispitivanog programa
    - **ispitivanje zasnovano na pogreškama**
        - ispitni slučajevi koji omogućavaju otkrivanje pogrešaka
        - umjetno ubacivanje pogrešaka i otkrivanje u kojoj mjeri ih ispitivanje otkriva
    - **ispitivanje zasnovano na kvarovima**
        - usmjereno na kvarove graničnih vrijednosti ili max br elemenata
        - ispitni slučajevi zasnovani na poznavanju tipičnih mjesta izloženih kvarovima
    - **funkcijsko ispitivanje (black box)**
        - ispitni slučajevi izgrađuju se temeljem specifikacije
    - **strukturno ispitivanje (white box)**
        - ispitivanje uzima u obzir strukturu programa
    - **ispitivanje pod pritiskom**
        - naglasak na robusnost u kritičnim uvjetima rada
- potpuno ispitivanje – po završetku postupka ispitivanja znamo da nema neotkrivenih pogrešaka
    - za potpuno ispitivanje neophodno provesti ispitivanje svih mogućih:
        - vrijednosti varijabli
        - kombinacija ulaza
        - sekvenci izvođenja programa
        - HW/SW konfiguracija
        - načina uporabe programa
- pokrivanje ispitivanja:
    - stupanj gotovosti ispitivanja pojedinih atributa ili dijelova programa u odnosu na broj mogućih ispitnih slučajeva
        - postotak programskih elemenata koji su izvedeni
        - pokrivenost linija koda
        - pokrivenost grana
        - pokrivenost putova
    - nedostatak: previše pojednostavljenja
    - loše za procjenu gotovosti
- za ispitivanje potrebno: specifikacije, informacije o oblikovanju, programski kod, uporaba programa, iskustvo

Organizacija ispitivanja

- proces ispitivanja sustava:
  - ispitivanje **komponenti i modula**
    - individualne komponente ispituju se nezavisno (funkcije, objekti...)
  - ispitivanje **sustava**
    - ispitivanje cjelovitog sustava
  - ispitivanje **prihvatljivosti**
    - značajke na temelju kojih kupac prihvaća i preuzima sustav
- faze u ispitivanju sustava:
  - **ispitivanje komponenti**
    - provodi programer u kontekstu specifikacije zahtjeva
      - tijekom kodiranja (inkrementalno kodiranje, značajno za skraćenje vremena ispitivanja)
      - statičko ispitivanje (prolazno – neformalno, nadzor koda – formalno, automatizirani alati za provjeru – sintaksa i semantika + odstupanja od standarda)
    - ispitivanje koda na pogreškama u algoritmima, podacima i sintaksi
    - verificira rad programskih dijelova koje je moguće neovisno ispitati (pojedinačne funkcije ili metode unutar objekata, klase objekata s više atributa i metoda, složene komponente s definiranim sučeljem za pristup njihovim funkcijama)
    - okolina ispitivanja komponenti
      - postupak izolacije komponenata u svrhu ispitivanja
      - upravljački program – kod koji upravlja procesom izvođenja 1 ili više komponenti
      - prividna (krnja) komponenta – kod koji simulira pozvanu komponentu
    - elementi ispitivanja komponenti:
      - sučelje - osigurava ispravno prihvaćanje i pružanje informacija
        - **parametarsko sučelje** – podaci i funkcije prenose se pozivima proc.
        - **dijeljena memorija** – procedure dijele zajednički mem. prostor
        - **proceduralno sučelje** – komponente obuhvaćaju skup procedura koje pozivaju ostali podsustavi
        - **sučelje zasnovano na porukama** – podsustavi traže usluge od ostalih podsustava slanjem poruke (klijent-poslužitelj)
        - naputak:
          - oblikuj ispitne slučajeve tako da parametri poprime ekstremne vrijednosti
          - ispituj pokazivače s null vrijednostima
          - oblikuj ispitni slučaj proceduralnog sučelja tako da zataji komponenta
          - sustave s razmjenom poruka ispitaj na stres
          - sustave s dijeljenom memorijom ispitaj s različitim redoslijedom aktiviranja komponenti
        - kvarovi sučelja:
          - pogrešna uporaba
          - nerazumijevanje sučelja
          - vremenske pogreške
      - podaci strukutra – osigurava integritet podataka
      - rubni uvjeti – provjera rada u graničnim slučajevima
      - nezavisni putovi – sve putove kroz kontrolne skupine
      - iznimke – provjera ispravne obrade iznimke

- ispitivanje komponenti u OO sustavu
  - komponenta = ugl. klasa
  - obuhvaća ispitivanje svih operacija, postavljanje i ispitivanje svih atributa objekata i ispitivanje objekata u svim stanjima
  - ispitivanje grupa razreda predstavlja oblik integracijskog ispitivanja
  - koraci ispitivanja:
    1. ispitni slučaj jedinstveno označiti i eksplicitno povezati s ispitivanim razredom
    2. definirati namjenu ispitnog slučaja
    3. razraditi korake ispitivanja
       i. definirati stanja objekata koja se ispituju
       ii. definirati poruke i operacije koje se ispituju i njihove posljedice
       iii. definirati listu iznimaka koje mogu proizaći tijekom ispitivanja
       iv. definirati stanje okoline pri ispitivanju
  - **slučajno ispitivanje:** koraci:
    1. identificirati opercije primjenjive na razred
    2. definirati ograničenja na njihovo korištenje
    3. identificirati minimalni ispitni slučaj (slijed operacija koji definira minimalni životni vijek instanciranog objekta)
    4. generirati niz ispravnih slučajnih ispitnih sekvenci

  - **ispitivanje particija**
    - smanjuje broj ispitnih slučajeva potrebnih za ispitivanje razreda (princip ekvivalencije particija)
    - particije stanja – kategorizirati i ispitati operacije temeljem utjecaja na promjenu stanja objekata
    - particije atributa – kategorizirati i ispitati operacije temljem svojstva atributa
    - particije kategorija – dekompozicija funkcijske specifikacije i utvrđivanje generičkih karakteristika -> kategorizirati i ispitati operacije na temelju generičkih funkcija koje obavljaju
  - **ponašajno ispitivanje**
    - ispitni slučajevi moraju pokriti sva stanja objekta
    - velika mogućnost automatizacije i uporabe UML dijagrama stanja

- **integracijsko ispitivanje**
  - provodi ispitni tim
  - proces verifikacije interakcije programskih komponenti
- **ispitivanje sustava**
  - provodi ispitni tim
- **ispitivanje prihvatljivosti**
  - provodi ispitivanje naručitelja
- **ispitivanje instalacije**
  - provodi ispitivanje naručitelja
  -
- **alfa ispitivanje**
- **beta ispitivanje**

fali ostatak lekcije ispitivanj

ARHITEKTURA PROTOKA PODATAKA

- dominantno pitanje – kako se podaci pomiču kroz kolekciju modula za izračunavanje
- temelji se na skupu procesnih modula (automata – aktora) koji razmjenjuju podatke i paralelno (konkurentno) obavljaju obradu
- osnovna prednost:
    - razdvajanje procesnih dijelova programa
    - minimazicija dijelova programa koji se odnose na eksplicitno povezivanje varijabli, funkcija, modula...
- **komponenta**
    - ima definirane ulaze+izlaze -> čita podatke s ulaza i stvara niz podataka na izlazima
    - ulaze transformira slijedno i lokalno
    - radi neovisnosti često se nazova filter – ne poznaje ostale filtre, a ako svi filtri u sustavu obrađuju sve podatke u jednom koraku tada je riječ o **slijedno sekvencijskoj**
- **konektor**
    - medij za prijenos podataka – cjevovod
    - specijalizacija
        - pipeline
        - bounded
- primjena: Unix prevodioci, obrada signala, paralelni sustavi, distribuirani sustavi
- svojstva:
    - **programibilna** računala sa sklopovljem optimiziranim za izračunavanje upravljano podacima
    - **fina podjela** na nivou instrukcija
    - **upravljanje podacima** – paralelizam uvjetovan samo raspoloživošću podataka; programi predstavljeni grafovima
- terminologija
    - aktor – procesni elementi
        - izvode izračunavanje, često bez stanja
        - funkcijski
        - uvjetni
        - spajanje
    - podaci
    - lukovi
    - ulazi
    - onemogućavanje
    - okidanje
    - izlaz podataka (ulazni+okidanje)
- prednosti:
    - izražavanje paralelizma
    - tolerancija kašnjenja
    - mehanizmi fine sinkronizacije
- nedostaci:
    - gubitak lokalnosti (ispreplitanje instrukcija)
    - rasipanje resursa
    - zauzeće prostora
- obilježja sustava:
    - raspoloživost podataka upravlja izračunavanjem
    - struktura arhitekture je određena redoslijedom pomicanja podataka od komponente do komponente
    - oblik strukture je eksplicitan

- prijenos podataka je jedini način komunikacije između komponenata u sustavu

- mehanizmi upravljanja protokom podataka
  - guranje (push) – izvor podataka gura podatke od sebe
    - aktivni pisatelj incira prijenos podataka pozivom primajuće procedure receive(data) pasivnog čitača (podaci se prenose kao parametar procedure)
    - upravljački tok i tok podataka u istom smjeru
  - povlačenje (pull) – ponor podataka uvlači podatke
    - aktivni čitatelj inicira prijenos pozivom procedure data_supply() pasivnom pistalju (podaci se prenose kao povratna vrijednost procedure)
  - guranje/povlačenje – filtar aktivno povlači podatke iz niza, obavlja obradu i gura podatke dalje
  - pasivni mehanizam – ne čini ništa, podaci tonu
- primjer: LabVIEW
  - svaka komponenta izvršava zadatak kada su svi ulazni uvjeti zadovoljeni (paralelizam/pseudo paralelizam)
  - temeljna komponenta = prevoditelj za G programski jezik
    - homogen – G aktori proizvode i konzumiraju pojedinačne značke na svakom luku graha
    - dinamičan – G sadrži konstrukcije koje dopuštaju da se dijelovi grafa izvode uvjetno
    - višedimenzijski – G podržava višedimenzijske podatke
- stilovi arhitekture protoka podataka:
  - **skupno sekvencijski**
    - svaki procesni korak je nezavisan program i izvodi se do potpunog završetka, prije prijelaza na sljedeći korak
    - podaci se između koraka prenose u cijelosti
    - primjene:
      - poslovne = diskretne transakcije unaprijed određenog tipa koje se pojavljuju u periodičnim intervalima, periodični ispisi i dopune, obrada koja nije pod strogim vremenskim ograničenjima itd. – npr. obračun plaća
      - transformacijska analiza podataka = sakupljanje i analiza sirovih podataka u koračnom i skupnom modu
    - primjer:
      - kompilatori – početno mehanizam preslikavanja izvornog koda više razine u objektni kod
      - CASE – početno okolina za pisanje i komiliranje
  - **cjevovodi i filtri**
    - alternativa skupno sekvencijskom sustavu kod potrebe kontinuirane obrade
    - magnetska vrpa u skupnom sekvencijskom sustavu poprima oblik jezika i konstruktora u operacijskom sustavu
    - obilježja i uloge (čistih) filtra:
      - čita niz podataka na ulazu i generira niz podataka na izlazu
      - transformira niz u niz
        - obogaćuje dodajući nove informacije
        - rafinira podatke izbacujući nerelevantne informacije
        - transformira podatke promjenom njihovog značenja
      - inkrementalno transformira podatke
        - podaci se obrađuju po dolasku

        - filtri su nezavisni entiteti
            - nema konteksta u obradi niza – nema čuvanja stanja između pojedinih instanciranja sustava
            - nema znanja o neposrednim susjedima
        - podaci se prenose u jednom smjeru (eventualno dopušteno upravljanje u oba smjera)
    - primjer: UNIX
        - UNIX procesi koji preslikavaju stdin u stdout nazivaju se filtri
        - često konzumiraju sve ulazne podatke prije nego generiraju izlaz (narušavanje inkrementalnosti)
        - datoteke tretiraju kao filtre te kao ulazne i izlazne datoteke – one su pasivne i nije moguće povezivanje po volji
        - pretpostavlja da cjevovodi prenose ASCII znakove

Skupno sekvencijska	***Cjevovodi i filtri***
Gruba zrnatost	Fina zrnatost
Visoka latencija	Rezultati s početkom obrade
Vanjski pristup ulazima	Lokalizirani ulazi
Nema paralelizma	Moguć paralelizam
Nema interaktivnosti	Interaktivnost moguća (ali nespretno)

- razlozi za arhitekturu protoka podataka:
    - zadatkom dominira dobavljivost podataka
    - podaci se mogu prediktivno prenositi od procesa do procesa
    - cjevovodi i filtri su dobar izbor u mnogim primjenama protoka podataka zbog:
        - ponovna upotreba i rekonfiguracija filtra
        - rasuđivanje o cijelom sustavu olakšano
        - reducira se ispitivanje sustava
        - mogućnost inkrementalne i paralelne obrade
    - mogućnost reduciranja performansi
- evaluacija arhitekture protoka podataka
    1. *Podijeli i vladaj:* odvojeni procesi mogu se oblikovati nezavisno
    2. *Povećaj koheziju*: procesi posjeduju funkcionalnu koheziju
    3. *Smanji međuovisnost*: procesi imaju mali broj ulaza i izlaza
    4. *Povećaj apstrakciju*: cjevovodne komponente su dobra apstrakcija jer skrivaju unutarnje detalje
    5. *Oblikuj da se omogući ponovna uporabivost dijelova*: procesi se često koriste u mnogim različitim kontekstima
    6. *Uporabi postojeće komponente*: često se mogu pronaći gotove komponente za uključenje u sustav
    7. *Oblikuj za fleksibilnost*: postoji više različitih ostvarenja fleksibilnosti sustava Komponente se mogu jednostavno izbacivati i nadomještati. Nekim komponentama se može mijenjati redoslijed u sustavu
    10. *Oblikuj za jednostavno ispitivanje*: uobičajeno je jednostavnije ispitivanje pojedinačnih komponenti
    11. *Oblikuj konzervativno*: rigorozno se provjeravaju ulazi svake komponente ili se mogu postaviti jasne preduvjeti i post-kondicije svake komponente

OSTALE ARHITEKTURE PROGRAMSKE POTPORE

Arhitektura zasnovana na događajima

- temeljne značajke
    - komponente se međusobno ne pozivaju eksplicitno
    - neke komponente generiraju signale – događaje, a one komponente koje su zainteresirane za njih prijavljuju se na strukturu za povezivanje komponenata
    - komponente koje objavljuju događaj nemaju informaciju koje komponente i kako će reagirati na taj događaj
    - asinkrono rukovanje događajima
    - nedeterministički odziv na događaj
    - model izvođenja: događaj se javno objavljuje te se pozivaju registrirane procedure
- tipovi povezivanja:
    - eksplicitno
    - implicitno
        - primjenski program ne poziva rutinu za obradu događaja ni izravno ni neizravno
            - prekidna rutina aktivira se preko vektora
        - primjer: login scenarij -> prijava generira događaj na koji se odazivaju brojni procesi
- 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 događaj nemaju garancije da će dobiti odziv niti imaju utjecaj na redoslijed odziva
    - apstrakcija događaja ne vodi prirodno na postupak razmjene podataka
    - teško rasuđivanje o ponašanju komponenata koje objavljuju događaje i pridruženim komponentama koje su registrirane uz te događaje
- primjer: MVC
    - model: jezgrene funkcionalnosti i podaci
    - view: prikaz informacija
    - controller: rukovanje ulaznim podacima korisnika
- intenzivna primjena u procesorskim sustavima s više jezgara

Arhitektura repozitorija podataka

- obuhvaća načine prikupljanja, rukovanja i očuvanja velike količine podataka
- dvije vrste komponenti:
    - središnji repozitorij podataka
    - kolekcija nezavisnih komponenti koje operiraju nad središnjim repozitorijem
- temeljna prednost: jednostavno dodavanje i povlačenje proizvođača i korisnika podataka
- problemi:
    - sinkronizacija
    - konfiguracija i upravljanje shemama strukture podataka
    - atomičnost
    - konzistencija
    - očuvanje
    - performanse

- **BAZA PODATAKA (DB)**
  - komponente modificiraju ili čitaju podatke iz baze
  - baza skladišti perzistentne podatke između različitih transakcija
  - nema fiksnog uređenja redoslijeda između transakcija
  - komponenta „Control“ upravlja paralelnim pristupima DB
  - E-R model
    - koncepcijski model podataka koji opisuje podatkovne zahtjeve u informacijskom sustavu na jednostavan i razumljiv grafički način
    - služi za modeliranje scenarija
  - E-R model vs UML dijagram (razreda)
    - E-R dozvoljava n-struku relaciju, dijagram razreda samo relacija između 2 razreda
    - E-R dozvoljava atribute s više vrijednosti
    - E-R specificira identifikatore
    - dijagram razreda omogućuje dinamičku klasifikaciju -> tijekom izvođenja objekt može promijeniti razred (u E-R modelu to nije niti predviđeno)
    - UML razredi imaju pridružene metode, ograničenja, pre/post uvjete
- **OGLASNA PLOČA (blackboard)**
  - mnogi eksperti promatraju rješavanje zajedničkog problema na ploči pri čemu svaki ekspert dodaje svoj dio u rješavanju cjelokupnog problema
  - osnovni dijelovi:
    - **izvori znanja (knowledge sources KS)** – odvojeni i nezavisni dijelovi primjenskog znanja (eksperti)
      - međusobno surađuju samo putem oglasne ploče (ne izravno)
    - **oglasna ploča** – podaci o stanju problema, organizirani prema hijerarhiji
      - promjena sadržaja na ploči -> KS mijenjaju pojedine sadržaje -> rješenje
    - **upravljanje stanjem** oglasne ploče
      - KS se odazivaju oportunistički kada se promjene na oglasnoj ploči na njih odnose
  - DB vs oglasna ploča
    - baza podataka: vanjsi procesi iniciraju promjenu sadržaja
    - oglasna ploča: promjena sadržaja inicira vanjske procese (KS)
  - problemi:
    - nema izravnog algoritamskog rješenja – višestruki pristup rješavanju
    - neizvjesnost – pogreške u pdoacima, srednji ili niski omjer signal prema šum u podacima
    - aproksimativno rješenje je dovoljno dobro – ne postoji jedan diskretan odgovor na problem ili ispravan odgovor može varirati
  - primjena:
    - obrada signala
    - planiranje, logistika, dijagnostika
    - optimizacija prevodioca programskih jezika
  - nije unaprijed specificirano:
    - kako je predstavljeno znanje / struktura znanja
    - kako KS dohvaćaju, zapisuju i koriste podatke s oglasne ploče
    - kako se određuje kvaliteta rješenja
    - kako se upravlja ponašanjem KS
  - znanje može biti prestavljeno na različite načine (jednostavnim funkcijama, logičkim izrazima...)
  - predstavljeno znanje izaziva odgovarajuću akciju na oglasnoj ploči

- mehanizmi upravljanja:
  - upravljano događajima ili signalima (prekidima)
  - upravljanje pozivanjem procedura
- strategija upravljanja
  - podatkovno inicirano – sljedeći korak je definiran stanjem podataka te se poziva odgovarajući KS
  - inicirano ciljem – upravljačka funkcija poziva KS koji najviše doprinosi pomaku prema cilju
  - kombinacija – korišteno u praksi
- prednosti i nedostaci:
  - intencijsko i reaktivno ponašanje – kako se napreduje prema rješenju
    + jednostavno integriranje različitih autonomnih sustava
    - oglasna ploča može biti vrlo složena arhitektura, posebice ako su komponente prirodno međuzavisne
  - predstavljanje i obrada neizvjesnog znanja
    + oglasna ploča je dobra arhitektura za ovaj tip problema
  - sigurnost i tolerancija na kvarove
    + podsustavi mogu promatrati oglasnu ploču i obratiti pažnju na potencijalne poteškoće
    - oglasna ploča kritičan resurs
  - fleksibilnost
    + oglasna ploča inherentno fleksibilna
- primjer: Hearsay
  - sustav za raspoznavanje govora s velikim vokabularom
- primjer: prevoditeljl
  - inicijalno: prevođenje izvornog u objektni kod

Slojevita arhitektura

- primjenski program ili OS se strukturira tako da se dekomponira u podskupove modula koji čine različite razine apstrakcije
- dekompozicija je na razini statičkog izvornog koda
  - ako je na razini izvođenja radi se o razinama/naslagama
- temeljni elementi
  - komponente = slojevi
  - konektori = protokoji koji definiraju interakciju između slojeva
- hijerarhijska organizacija:
  - samo susjedni komuniciraju – svaki sloj daje uslugu sloju iznad i postaje klijent sloju ispod
  - svaki sloj skriva slojeve ispod
- prednosti:
  - sloj djeluje kao koherentna cjelina
  - vrlo jednostavna zamjena sloja novijom inačicom
  - jednostavno održavanje jer svaki sloj ima dobro definiranu ulogu
  - minimizirana je međuovisnost komponenata
  - podupire oblikovanje programske potpore na visokoj apstraktnoj razini
  - podupire ponovno korištenje (*engl, reuse*)
- nedostaci:
  - smanjene performanse jer gornji slojevi samo indirektno dohvaćaju donje slojeve
  - teško se pronalazi ispravna apstrakcija (posebice u sustavima s većim brojem slojeva)
  - svi sustavi se ne preslikavaju izravno u slojevitu arhitekturu (djelokrug programske jedinice može zahvaćati više slojeva)

Virtualni strojevi (VM)

- računalno okruženje čiji je skup resursa i funkcionalnosti izgrađen (kroz programski sloj) iznad nekog drugog programskog okruženja (apstrakcija računalnih resursa)
- temeljna prednost: odvajanje primjenskog programa od ostatka sustava
- nedostatak: slabljenje performansi
- VM može simulirati funkcionalnosti nepostojeće sklopovske platforme – pomoć u razvoju
- konfiguracije:
    - **hipervizorski VM**
        - VM kao dodatni sloj odmah iznad sklopovlja
        - na njega se oslanjaju jedan ili više OS gdje svaki smatra sebe jedinim u računalu
        - primjer: Vmware ESX server
    - **udomaćeni VM**
        - VM, kao i svaki drugi primjenski program, izvodi se iznad OS-a
        - poseban proces
        - domaćinski OS osigurava pristup sklopovlju
        - primjer: Vmware GSX server, Linux user mode
    - **primjenski VM**
        - slično udomaćenom, ali na primjenskoj razini
        - primjer: Java VM
    - **paralelni VM**
        - pretpostavlja se postojanje međusloja
            - međusloj egzistira kao uslužni program, zajedno s pozivima u skup rutina
            - međusloj omogućuje da su rutine raspodijeljene u mreži računala