

UNIVERZA NA PRIMORSKEM

FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN INFORMACIJSKE TEHNOLOGIJE

Peter Rogelj

SKRIPTA ZA PREDMET

PROGRAMSKO INŽENIRSTVO

NA ŠTUDIJSKEM PROGRAMU

RAČUNALNIŠTVA IN INFORMATIKE NA 1. STOPNJI

UP FAMNIT

PRVA IZDAJA

Koper, 2014

Predgovor

Študijska literatura, ki je pred vami, je nastala iz prosojnic predavanj pri predmetu Programsko inženirstvo na Fakulteti za matematiko, naravoslovje in informacijsko tehnologijo Univerze na Primorskem. Njen namen, kot tudi namen predmeta, je bralca/študenta seznani z osnovami programskih procesov. Obravnava celotni življenjski cikel programskih produktov. Ukvarjamo se torej z zakonitostmi razvoja programskih produktov - softvera, ne glede na to za kakšne vrste programov gre; čemu so namenjeni, na katerih platformah delujejo, kdo so njegovi uporabniki ali kakšne naj bodo njihove lastnosti. Torej, življenje programov spremljamo oziroma vodimo od ideje za njihov nastanek preko razvoja, njihove uporabe, do prenehanja vzdrževanja in opustitve programskega produkta. Celotni programski proces je razdeljen na več faz in pri vsaki od njih je poudarjen njen pomen, tipične težave in napotki za zagotavljanje kar največje uspešnosti programskih produktov. O uspešnosti programskih produktov priča obseg njihove uporabe, ki je odvisen od vrste dejavnikov, med katerimi je eden najbolj očitnih prisotnost programskih napak. Tem se sicer v popolnosti nikoli ne moremo izogniti, s sistematičnim pristopom pri razvoju in vzdrževanju pa lahko zmanjšamo njihovo število in njihovo kritičnost.

Za popestritev vsebine so v literaturo vključeni tudi zabavnejši pogledi na problematiko, ki bralca spodbudijo k širšemu razmišljanju; običajno so označeni s simbolom smeška »☺«.

Ljubljana, Januar 2014

Peter Rogelj

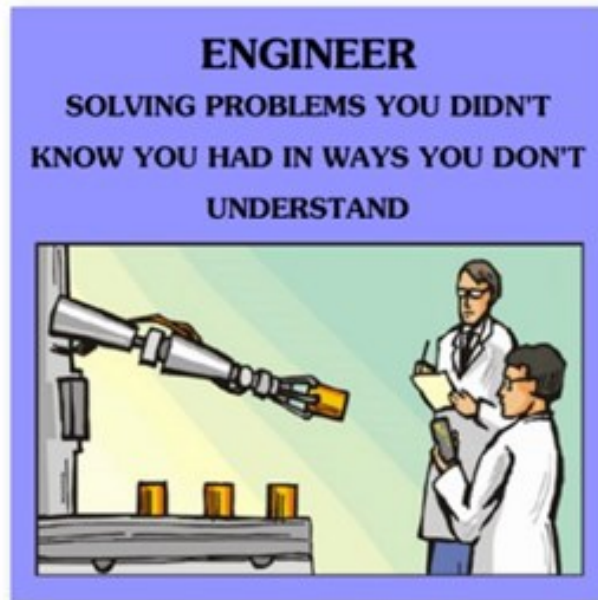
Kazalo

1	Uvod v programsko inženirstvo	4
2	Programski proces.....	20
3	Agilni pristopi	60
4	Diagramске tehnike	82
5	Analiza in definiranje zahtev	104
6	Načrtovanje sistema	135
7	Načrtovanje komponent	179
8	Testiranje	209
9	Vzdrževanje.....	245
10	Uporabniški vmesniki	266
11	Pravne zadeve	294

1 Uvod v programsko inženirstvo

Programsko inženirstvo je inženirska disciplina, ki zahteva interdisciplinarna znanja. Njen prvotni namen je izboljšati kvaliteto razvoja programskih produktov, tako da bi bili le ti razviti pravočasno, v predvidenem obsegu sredstev in da ne bi vsebovali napak. Dandanes se zavedamo, da zanesljive rešitve za to ni, z uporabo sistematičnega pristopa pa te probleme lahko močno omilimo. Kljub vsemu ostaja razvoj softvera ena najmanj uspešnih inženirskih disciplin, saj je neuspešnih kar okrog $1/4$ programskih projektov, resne težave pa zaznamo pri vsaj $1/3$. Razlog je v interni kompleksnosti softvera, ki je večja od kompleksnosti klasičnih produktov. Programski produkti so odraz človekovega ustvarjalnega udejstvovanja, ki ga zaradi kompleksnosti reševanih problemov ni mogoče avtomatizirati in je zato podvrženo človeškim napakam. Pristopi programskega inženirstva le te lahko omilijo oziroma omogočajo njihovo razkritje še preden lahko postanejo usodne.

Inženirstvo ☺



Kaj je programsko inženirstvo

❖ IEEE definicija (1993):

The application of a systematic, disciplines, quantifiable approach to the development, operation, and maintenance of software; that is the application of engineering to software.

Umestitev programskega inž.



Programski produkt

- ❖ **Računalniški programi** (programska oprema), ki omogočajo, da z njihovim izvajanjem dosežemo željen učinek.
- ❖ **Podatkovne strukture**, ki programom omogočajo manipulacijo (obdelavo) podatkov (informacij).
- ❖ **Dokumenti**, ki opisujejo delovanje in uporabo programov

Delitev programske opreme

- ❖ Sistemska
- ❖ Za delo v realnem času (real-time)
- ❖ Poslovna
- ❖ Znanstvena
- ❖ Vgrajena (embedded)
- ❖ Osebna (za osebno uporabo na osebnih rač.)
- ❖ Mrežna
- ❖ Umetna inteligenca
- ❖ ...

Ne glede na vrsto programske opreme veljajo enake osnovne zakonitosti njenega razvoja.

Začetki programskega inž.

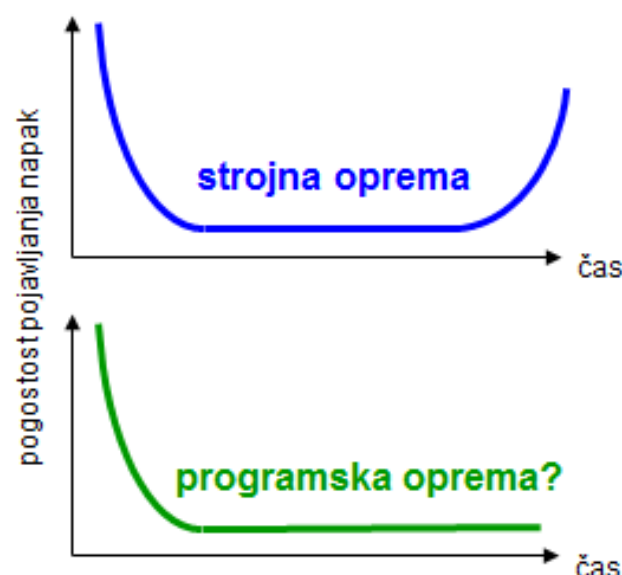
- ❖ 1945 – 1965: za razvoj softvera je opaženo, da pogosto:
 - ♦ zahteva več sredstev, kot je sprva predvideno,
 - ♦ se ne zaključi ob predvidenem času,
 - ♦ je neuspešen v smislu prisotnosti napak v končnem produktu.
- ❖ 1968: NATO organizira dve konferenci na temo razvoja programskih produktov kjer
 - ♦ probleme razvoja softvera definirajo kot „softversko krizo“ (ang. The Software Crisis),
 - ♦ definirajo novo vejo inženirstva - programsko inženirstvo, ki naj bi rešilo omenjene težave.
- ❖ 1968 - 1985: nadaljevanje „softverske krize“, dokler po vrsti neuspešnih poskusov in zaradi dolgotrajnosti njenega trajanja sčasoma ne postane jasno, da za razvoj programskih produktov ni enoznačne rešitve.
 - ♦ Nadaljuje se z iskanjem rešitev, ki bi problem omilile.

Vir: http://en.wikipedia.org/wiki/History_of_software_engineering

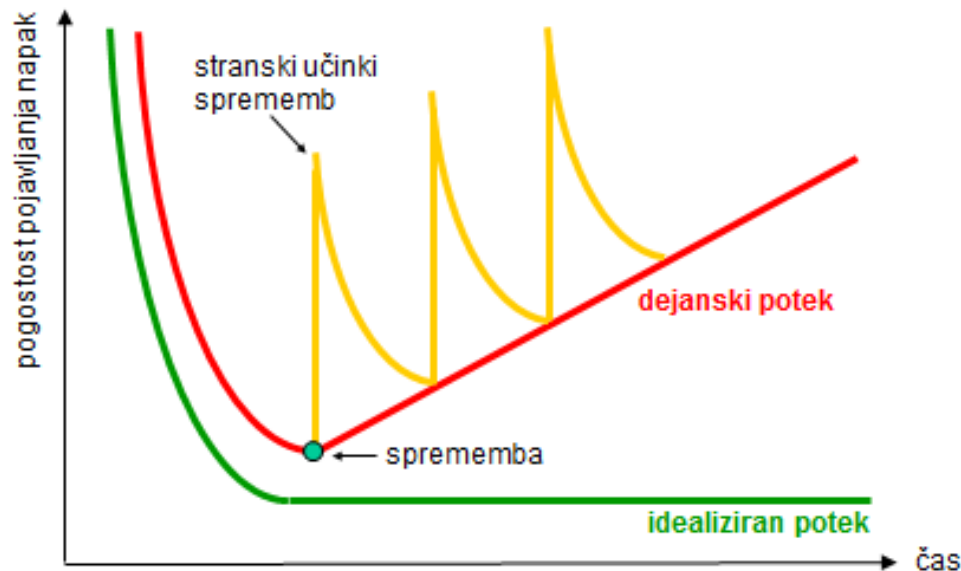
Posebnosti programskega inženirstva

- ❖ Programska oprema ima visoko notranjo kompleksnost - vsebuje skrita stanja: veliko število nevidnih spremenljivk.
- ❖ Programske opreme ne "proizvajamo" v klasičnem pomenu besede, pač pa jo razvijamo.
- ❖ Programska oprema je v večini primerov izdelana po naročilu.
- ❖ Programska oprema je pogosto najdražji del celotnega sistema.
- ❖ Težavno merjenje napredka pri razvoju.
- ❖ Vzdrževanje pomeni popravke, izboljšave in dodatne funkcionalnosti.
- ❖ Programska oprema se z uporabo ne iztroši. V splošnem pa njena kvaliteta z vzdrževanjem in s spreminjanjem upada.

Stopnja okvar



Stopnja okvar programske opreme



Programska oprema in napake

- ❖ Programska oprema tipično vsebuje napake.
- ❖ Programsko inženirstvo nudi tehnike za omejitev oziroma zmanjšanje števila in usodnosti napak.
- ❖ Pa vendar se napakam težko (nikoli?) popolnoma izognemo...
 - ♦ *"...building software will always be hard. There is inherently no silver bullet."* F. Brooks

Primeri usodnih napak

❖ Ariane 5

- ♦ Samouničenje rakete 39 sekund po vzletu (vključno s 4 dragimi in nezavarovanimi raziskovalnimi sateliti).
- ♦ Nepotreben popravek smeri letenja, katere vzrok je bila aritmetična napaka.
- ♦ Napaka se je zgodila v delu kode, ki za Ariane 5 ni bila potrebna in je podedovana iz Ariane 4.
- ♦ Programska izjema pri pretvorbi 64-bitnega števila 16-bitno...

Vir: <http://www.around.com/ariane.html>

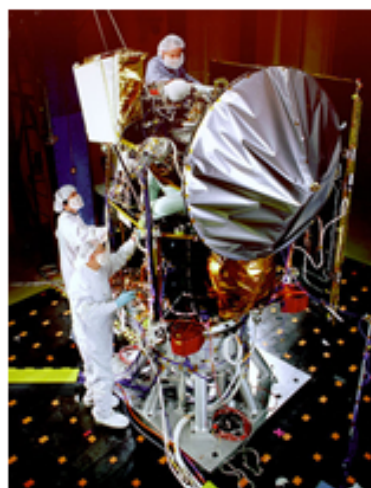


Primeri usodnih napak

❖ Mars Climate Orbiter (1999)

- ♦ Uničen ob vstopu v Marsovo atmosfero zaradi prehitrega spusta.
- ♦ Programska oprema je bila podedovana od predhodnika.
- ♦ Vzrok je bil v programski napaki, ki je podcenila učinek raketnih pogonov za faktor 4,45, kar je razmerje med enotami Angleškega in enotami metričnega sistema.
- ♦ Razvojne ekipe so uporabljale različne sisteme enot in niso bile usklajene. Testiranje ni bilo zadostno.

Vir: http://en.wikipedia.org/wiki/Mars_Climate_Orbiter



Primeri usodnih napak

- ❖ Therac-25 radioterapevtska obsevalna naprava
 - ♦ Zaradi programske napake je prišlo do 100x prevelikega obsevanja pacientov zaradi česar so trije pacienti umrli.
 - ♦ Vzrok je bil v softverski napaki zaradi katere je bil določenih situacijah namesto nizko-energijskega snopa elektronov uporabljen visoko-energijski rentgenski vir.

Vir:
<http://fauxdurbeyfield.wordpress.com/2011/11/03/therac-25-because-there-isnt-enough-radiation-in-this-world/>



Uspešnost programskih produktov

- ❖ Uspešnost programskih produktov je v splošnem manjša od uspešnosti projektov na drugih inženirskih področjih!
- ❖ Programska rešitev je lahko uspešna če
 - ♦ je zaključen razvoj
 - ♦ je uporabna
 - ♦ je uporabljena

Razlogi za neuspeh

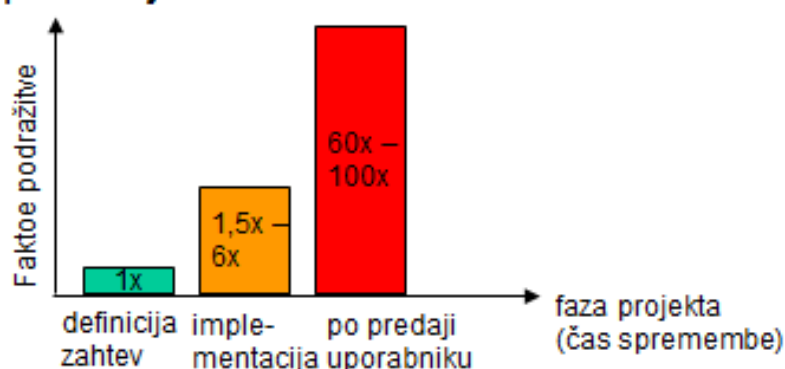
- ❖ Slaba kvaliteta - razviti sistem vsebuje napake
- ❖ Nedoseganje ključnih rokov (čas dobave)
- ❖ Preseganje omejenih stroškov
- ❖ Nezadovoljevanje uporabnikovih potreb
- ❖ Nezmožnost ali težavnost vzdrževanja
- ❖ Cena razvoja in vzdrževanja je bistveno višja od pričakovane (ocenjene)

Razlogi za neuspeh

- ❖ Neustrezno planiranje razvoja
- ❖ Nedefinirani cilji projekta
- ❖ Slabo razumevanje uporabnikovih potreb
- ❖ Nezadostno preverjanje in kontrola kvalitete
- ❖ Nezadostno tehnično znanje članov razvojnega tima
- ❖ Slabo razumevanje potrebnih razvojnih vložkov in cene, na strani razvijalca in uporabnika.

Cena programskih produktov

- ❖ Največji del cene razvoja programskih produktov predstavlja cena dela ljudi.
- ❖ Spremembe v času izvedbe projekta le tega močno podražijo.



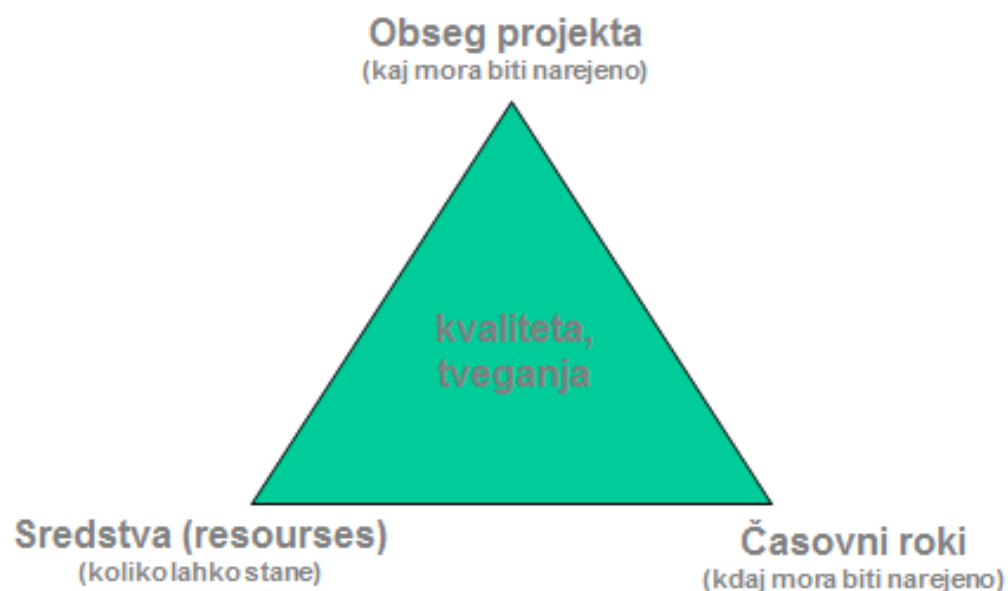
Lastnosti dobrih programskih produktov

- ❖ **Funkcionalnost** - ki zadosti potrebe uporabnika
- ❖ **Zmožnost vzdrževanja** – možnost razvoja glede na spreminjajoče potrebe.
- ❖ **Zanesljivost** – programska oprema mora biti vredna zaupanja.
- ❖ **Učinkovitost** – uporabe sistemskih zmogljivosti.
- ❖ **Sprejemljivost** – sprejeta mora biti s strani uporabnikov za katere je bila načrtovana – biti mora razumljiva, uporabna in združljiva z drugimi sistemi.

Inženirski pristop

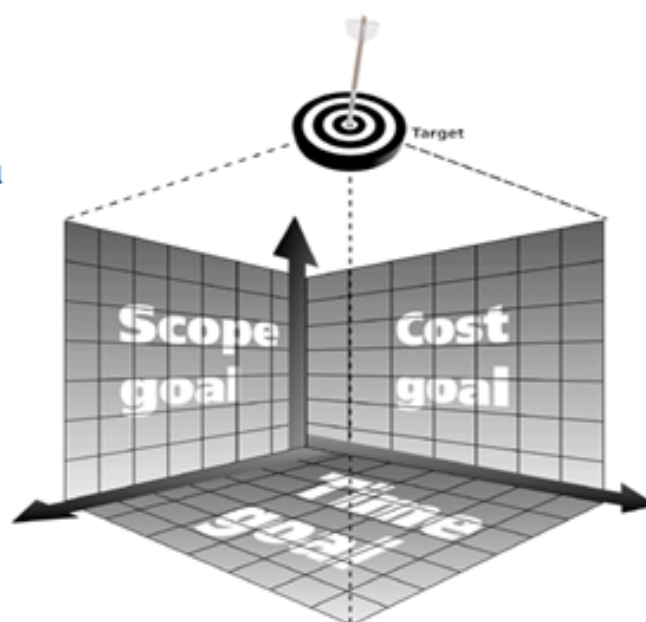
- ❖ Ocenjevanje stroškov/vložkov
- ❖ Časovno načrtovanje dela
- ❖ Vključitev uporabnikov ob definiranju zahtev
- ❖ Določitev faz v razvoju
- ❖ Določitev končnih produktov
- ❖ Spremljanje napredovanja projekta
- ❖ Načrtovanje pregledov in kontrole kvalitete
- ❖ Obsežno testiranje

Omejitve in cilji projektov



Omejitve in cilji projektov

Uspešen projekt mora zadovoljiti vse tri omejitve oz. cilje (obseg - scope, čas - time, in stroški - cost). Za to mora skrbeti vodja projekta!



Vir: Kathy Schwalbe, Information Technology Project Management, Fourth Edition

Izzivi velikih/kompleksnih projektov

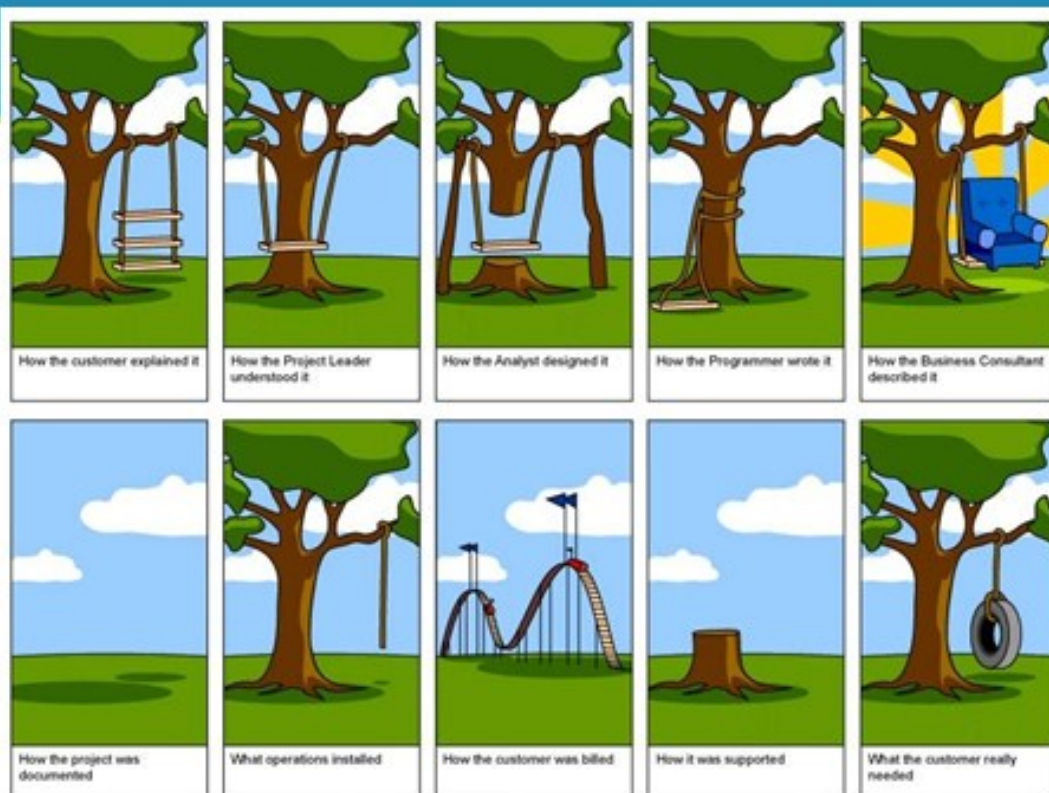
- ❖ Razvoj zahteva veliko vloženega dela (truda)
- ❖ Visoka cena razvoja
- ❖ Razvoj traja daljše časovno obdobje
- ❖ Spreminjajoče potrebe uporabnikov
- ❖ Visoka tveganost, možnost neuspeha (negativen odziv uporabnikov, nezadostne performance, oteženo vzdrževanje...)

Velikost šteje!



Akterji v (programskih) projektih

- ❖ sponzor projekta
- ❖ vodja projekta (project manager)
- ❖ projektni tim
- ❖ podporno osebje
- ❖ stranke
- ❖ uporabniki
- ❖ dobavitelji
- ❖ nasprotniki projekta



Izzivi projektne vodje

❖ Upoštevanje uporabnikov

- ♦ Nejasno definirane zahteve
- ♦ Bistvenega pomena je enostavnost uporabe in odzivnost.

❖ Upoštevanje članov razvojnega tima

- ♦ Pozornost posvečena načinu razvoja, tehnični rešitvi.

❖ Upoštevanje managementa

- ♦ Bistvena je ekonomska uspešnost, povrnitev vložkov ter upoštevanje časovnih rokov.

Izzivi programskega inženirja

❖ Heterogenost

- ♦ Razvoj tehnik za gradnjo programske opreme, ki je primerna za različne platforme in okolja.

❖ Hitrost razvoja

- ♦ Razvoj tehnik ki vodijo v hitrejši razvoj programske opreme in zgodnejšo dostavo uporabniku.

❖ Zaupanje

- ♦ Razvoj tehnik, ki opravičijo zaupanje uporabnikov.

Profesionalna in etična odgovornost

- ❖ Programski inženir ima širše odgovornosti kot le uporabo tehničnega znanja:
 - ♦ **Zaupnost.** Spoštovanje zaupnosti in varovanje podatkov delodajalcev in strank.
 - ♦ **Kompetence.** Spoštovanje nivoja svojih pristojnosti.
 - ♦ **Intelektualne pravice.** Varovanje intelektualnih pravic delodajalca, strank in partnerjev.
 - ♦ **Zlorabe.** Računalnikov in tehničnih znanj se ne sme zlorabljati. Obseg zlorab je širok – od nameščanja neodobrenih programov na službeni računalnik ali igranja iger, do razširjanja virusov in vdorov v sisteme.

Etične dileme

- ❖ Nestrinjanje s principi nadrejenih.
- ❖ Delodajalec neetično izda kritično programsko opremo brez zadostnega testiranja.
- ❖ Sodelovanje pri razvoju vojaških ali nuklearnih sistemov.
- ❖ ...

Miti

- ❖ Za uspešnost razvoja programske opreme je ključna sodobna strojna oprema.
- ❖ Če smo s projektom v zaostanku, lahko zamujeno nadoknadimo z dodatnimi sredstvi (koncept Mongoljske horde).
- ❖ Izražena želja stranke je dovolj za začetek kodiranja – detajle lahko upoštevamo kasneje
- ❖ Projektne zahteve se stalno spreminjajo, a vendar jih z lahkoto upoštevamo, kajti programska oprema je prilagodljiva.
- ❖ Ko končamo kodiranje in program začne delovati, je naše delo zaključeno.
- ❖ Dokler program ne deluje, ne moremo oceniti njegove kvalitete.
- ❖ Edini produkt programskega projekta je delujoč program.
- ❖ Programsko inženirstvo zahteva prekomerno dokumentiranje, kar upočasnjuje razvoj.

2 Programski proces

Programski proces je proces gradnje programskega produkta, od zaznane potrebe po programskem produktu do konca njegovega vzdrževanja. V tem procesu se soočamo s številnimi izzivi, ne le tehničnimi pač pa tudi organizacijskimi. V vsakem programskem procesu moramo opraviti naloge, ki jih lahko razdelimo v več faz in jih lahko vnaprej načrtujemo. Tudi če katere od faz ne načrtujemo, moramo naloge te faze kljub temu opraviti; govorimo o neformalnem izvajanju faze. Glavna pomanjkljivost neformalnega izvajanja je nepreglednost, ki prepogosto vodi do slabe kakovosti končnih programskih produktov. V primeru formalnega izvajanja lahko vsako od faz razdelimo v več korakov. Vsak korak mora imeti jasno zastavljene cilje, pogoje za začetek izvajanja in za zaključek ter nuditi tudi informacije ki omogočajo spremljanje njegovega izvajanja.

Zaporedje faz v programskem procesu je lahko različno. Četudi nekaterih faz ne moremo izvesti pred predhodno opravljenimi drugimi fazami, pa se deloma lahko prekrivajo ali se izvajajo večkrat, v več delih ali iteracijah. Glede na zaporedje izvajanja faz (in njihovih korakov) poznamo različne razvojne modele. Najbolj razumljiv je linearni model, ki pa ima kljub svoji dobri preglednosti tudi svoje pomanjkljivosti, ki onemogočajo kasnejše prilagajanje programskega procesa. Potrebe po prilagajanju lahko nastanejo zaradi spremenjenih pogojev/zahtev ali ugotovljenih napak/pomanjkljivosti v predhodnih fazah.

V primeru nejasnosti v projektu se pogosto poslužujemo razvoju prototipov. Tradicionalno jih uporabljamo za razjasnitev konkretnih nejasnosti in ne vključujejo vseh funkcionalnosti končnega programskega produkta, niti niso razviti z upoštevanjem vseh zahtev in omejitev. Vse pogostejša pa je uporaba prototipov na način, da le te postopno nadgrajujemo, dokler jih na koncu ne razvijemo v ciljni programski produkt. Razvoj takšnih prototipov mora biti bolj skrben že od samega začetka, saj neupoštevanje vseh potrebnih lastnosti sistema vodi do prisotnosti napak in težav pri vzdrževanju.

Enake razvojne modele lahko uporabimo za razvoj več (podobnih) programskih produktov. V tem primeru govorimo o programskih ogrodjih. Te lahko postopno izboljšujemo v smislu njihove učinkovitosti. Učinkovitost programskih procesov lahko merimo s pet stopenjsko lestvico. Če želimo, da so programski procesi optimalni pa jih moramo stalno prilagajati tehnološkim in organizacijskim spremembam.

Programski proces

- ❖ Programski proces je proces gradnje programskega produkta od zaznane potrebe po programskem produktu, do konca vzdrževanja.
- ❖ V programskem procesu se soočamo s tehničnimi in organizacijskimi izzivi.
- ❖ Rezultat programskega procesa je programski produkt.
- ❖ Različni programski produkti zahtevajo različne programske procese.

Temeljne predpostavke

- ❖ Dober programski proces vodi do dobrega programskega produkta.
- ❖ Dober programski proces zmanjšuje tveganja.
- ❖ Dober programski proces je bolj pregleden.

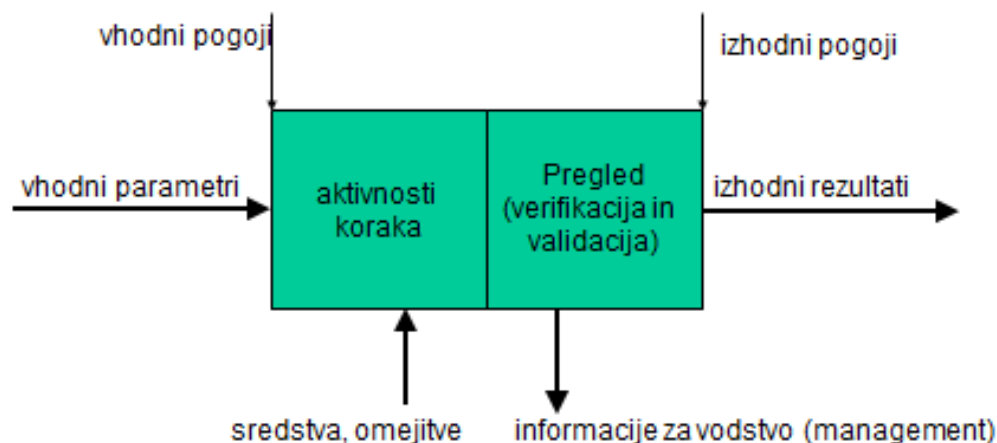
Struktura programskega procesa

- ❖ Programski proces sestoji iz aktivnosti/korakov, ki jih je potrebno izvesti v ustreznem vrstnem redu.
- ❖ Vsak posamezen korak
 - ♦ mora imeti jasno zastavljene cilje
 - ♦ zahteva ljudi s specifičnimi znanji
 - ♦ temelji na jasnih vhodnih parametrih
 - ♦ rezultira v dobro definiranih izhodnih rezultatih
 - ♦ definiran je čas začetka (tudi začetni pogoji) ter čas konca (končni pogoji)
 - ♦ uporablja specifične tehnike, orodja, smernice, dogovore, standarde...

Korak programskega procesa

- ❖ Mora biti izvršen po projektnem planu, ki določa trajanje, sredstva (resources), omejitve...
- ❖ Dajati mora informacije za vodstvo (management), ki omogoča korektivne ukrepe (npr. dodatna sredstva).
- ❖ Korak se konča s pregledom (V&V):
 - ♦ Verifikacija – preveri konsistentnost izhodnih rezultatov z vhodnimi parametri
 - ♦ Validacija – preveri konsistentnost s potrebami uporabnika

Korak programskega procesa



Faze programskega procesa ☺



Faze programskega procesa

- ❖ Definicija problema
- ❖ Študija izvedljivosti
- ❖ Analiza in definiranje zahtev
- ❖ Načrtovanje sistema
- ❖ Načrtovanje programa
- ❖ Izvedba
- ❖ Testiranje
- ❖ Predaja produkta
- ❖ Obratovanje in vzdrževanje

Definicija problema ^{1/4}

- ❖ Odgovor na vprašanja:
 - ♦ **Kaj je problem?**
 - ♦ Kje se problem pojavi in kdo ga občuti?
 - ♦ Zakaj se problem sploh pojavi?
 - ♦ Kako se problem lahko natančneje razišče?
 - ♦ Uporabniki in management se morajo strinjati o problemu.
 - ♦ Izogibati se je potrebno dvournim definicijam (možnosti različnih interpretacij).
- ❖ Ta korak je kratek in tipično traja dan ali dva.

Definicija problema 2/4

- ❖ Če problem obstaja, če ga je potrebno rešiti in če so za njegovo reševanje na voljo sredstva, potem to lahko postane projekt.
 - ♦ Potrebna je ocena stroškov za naslednji korak – študijo izvedljivosti.
 - ♦ Potrebna je groba ocena celotnih stroškov projekta, s čimer uporabnik dobi občutek o obsežnosti projekta. Boljša ocena se pripravi v kasnejših korakih.

Definicija problema 3/4

- ❖ Možne so različna izhodišča, npr.:
 - ♦ Nezaustojljiva prepustnost obstoječega sistema, slaba odzivnost.
 - ♦ Obstoječa rešitev ni ekonomična (stroški obratovanja)
 - ♦ Nezaustojljiva točnost obstoječe rešitve.
 - ♦ Nezaustojljiva obstoječe rešitve.
 - ♦ Nezaustojljiva varnost obstoječe rešitve.
 - ♦ Obstoječa rešitev ne omogoča pridobivanja ključnih informacij.
 - ♦ Potreba po novi funkcionalnosti.
- ❖ Kaj je problem in kaj rešitev!?
 - ♦ "Računalniški sistem za..." je rešitev in ne problem!

Definicija problema 4/4

❖ Rezultat definicije problema je dokument - opis problema (1-2 strani):

- ♦ naziv projekta,
- ♦ jedrnat opis problema,
- ♦ cilji projekta,
- ♦ morebitne začetne ideje o možnih rešitvah,
- ♦ potreben čas in stroški za naslednji korak – študijo izvedljivosti,
- ♦ groba ocena celotnih stroškov projekta.

Študija izvedljivosti 1/5

❖ Odgovori na vprašanja:

- ♦ Ali je projekt tehnično izvedljiv?
- ♦ Ali je projekt ekonomsko upravičen?
- ♦ Ali je projekt operativno izvedljiv (zakoni, kultura, dogovori)?
- ♦ Kakšne prednosti prinaša projekt?
- ♦ Kakšen je obseg predlaganega projekta?

❖ Predlaganih je lahko več alternativ izvedbe. Med njimi se odločamo glede na definirane (in dokumentirane) kriterije.

❖ Študija izvedljivosti vodi v odločitev za začetek projekta.

Študija izvedljivosti 2/5

❖ Tehnična izvedljivost:

- ♦ Za vsako alternativo izvedbe je potrebno okvirno proučiti tehnične načine izvedbe vseh nadaljnjih faz projekta (analiza zahtev, načrtovanje, izvedba, obratovanje) in podati tehnično oceno tveganja.
- ♦ Vključiti je potrebno tehnično osebje (ki kasneje lahko postane del projektnega tima).

Študija izvedljivosti 3/5

❖ Ekonomska upravičenost

- ♦ **Cena (costs):**
 - Enkratni stroški opreme (HW+SW), izobraževanj, programske opreme, konzultacije...
 - Variabilni stroški plač, sredstev, vzdrževanja, najemnin...
- ♦ **Prednosti (benefits):**
 - Prihranki (prihranki plač, materiala, povečanje proizvodnje, zmanjšanje stroškov obratovanja...)
 - Doprinosi (nudenje izboljšanih storitev strankam, izboljšan nadzor poslovanja (proizvodnja, inventar, finance...) zmanjšanje okvar, prisotnosti napak...

Študija izvedljivosti 4/5

❖ Kako oceniti stroške projekta?

- ♦ Z razdružitvijo sistema v več komponent. Posamezne komponente je lažje ocenjevati kot celoten sistem.
- ♦ Z uporabo zgodovinskih podatkov (npr. predhodni projekti)
- ♦ Z ocenitvijo stroškov osebja (za razvoj in obratovanje) stroški so časovno odvisni, zato je potreben predhodni časovni načrt.
- ♦ Z uporabo standardov na nivoju organizacije, za ocenitev dodatnih stroškov (vodenje, tajništvo, prostori, elektrika...)

Študija izvedljivosti 5/5

- ❖ Rezultat študije izvedljivosti je poročilo, ki se ga predstavi vodstvu in uporabnikom. Ti ocenijo:
 - ♦ Ali so alternative sprejemljive?
 - ♦ Ali rešujejo pravi problem?
 - ♦ Ali katera od rešitev ponuja bistvene prednosti in zagotavlja uspešnost projekta? Upoštevati je potrebno življenjsko dobo sistema, ki je običajno 5-7 let.
- ❖ Uporabniki in vodstvo izberejo eno od alternativ. To je osnova za začetek projekta.
- ❖ Veliko število (nesojenih) projektov tu 'umre'.

Analiza in definiranje zahtev 1/6

- ❖ Namen analize in definiranja zahtev je v celoti opisati obnašanje sistema:
 - ♦ Funkcijske zahteve: primeri uporabe (ang. use cases), za vse interakcije **uporabnika** s sistemom.
 - ♦ Ostale (nefunkcijske) zahteve, ki omejujejo načrtovanje ali izvedbo sistema
 - Obratovalne lastnosti (varnost, uporabljivost, zahtevana prepustnost sistema...)
 - Evolucijske lastnosti (možnost testiranja, možnost vzdrževanja, razširljivost, nadgradljivost...)
- ❖ Analizo in definiranje zahtev izvaja sistemski analitik.
- ❖ Napačna, nepopolna nekonsistentna ali dvoumna specifikacija zahtev so pogosto vzrok za neuspeh projektov in spore.

Za primere nefunkcijskih zahtev glej: http://en.wikipedia.org/wiki/Non-functional_requirement

Analiza in definiranje zahtev 2/6

- ❖ Izzivi:
 - ♦ Uporabniki pogosto ne poznajo točno dejanskih potreb in ne vejo kako lahko računalniški sistem izboljša njihove procese.
 - ♦ Uporabniki s časom spreminjajo svoje želje.
 - ♦ Uporabniki imajo lahko neskladne zahteve.
 - ♦ Uporabniki težko ločujejo med tem kaj je (cenovno) izvedljivo in kaj ne.
 - ♦ Uporabniki pogosto ne ločujejo med željami in zahtevami
 - ♦ Sistemski analitik ima običajno omejeno znanje iz področja uporabe sistema.
 - ♦ Pogosto uporabnik in stranka (naročnik) nista ista oseba.

Analiza in definiranje zahtev 3/6

❖ Proces analize in definiranja zahtev:

- ♦ Pridobivanje znanja iz področja uporabe.
- ♦ Seznanjanje s problemom (intervjuji, vprašalniki, iskanje soodvisnosti, identifikacija vrzeli).
- ♦ Študija obstoječih sistemov.
- ♦ Pogosto analiza poteka od zunaj navznoter – od potrebnih izhodnih podatkov do vhodnih podatkov, potrebnih procesov, potreb po hranjenju podatkov.
- ♦ Izdelava specifikacije (opisi sistema z naravnim jezikom, diagrami...)
- ♦ Recenzija rezultatov analize skupaj z uporabnikom.
- ♦ Po potrebi nova iteracija analize in definiranja zahtev.

Analiza in definiranje zahtev 4/6

❖ Rezultat analize in definiranja zahtev je dokument **Specifikacija zahtev**, ang. Software requirements specifications (SRS), (tudi funkcijske specifikacije -FS)

- ♦ Uporabniku nudi informacije na osnovi katerih lahko presodi o pravilnosti razumevanja problema s strani organizacije, ki bo za njih razvijala sistem.
- ♦ Problem razgradi na manjše komponente.
- ♦ Uporabi se za načrtovanje sistema.
- ♦ Uporabi se pri validaciji sistema.

Analiza in definiranje zahtev 5/6

- ❖ Vsebina specifikacije zahtev se mora prilagajati potrebam projekta, tipično pa vsebuje:
 - ♦ Uvod (namen, obseg, definicije, reference na druge dokumente)
 - ♦ Splošni opis (relacije z drugimi sistemi, pregled funkcij/komponent, značilnosti uporabnikov, splošne omejitve)
 - ♦ Opis zahtev vsake posamezne komponente (opis, vhodi, obdelava, izhodi)
 - ♦ Zahteve zunanjih vmesnikov (formati, strojna oprema, povezave z drugimi programi ali sistemi)
 - ♦ Zahteve po prepustnosti sistema (performance)
 - ♦ Omejitve pri načrtovanju (standardi, omejitve strojne opreme...)
 - ♦ Ostale zahteve

Analiza in definiranje zahtev 6/6

- ❖ Faza analize in definiranja zahtev se konča z recenzijo specifikacije zahtev
 - ♦ Pregled vseh zahtev:
 - Veljavnost. Ali sistem nudi funkcije, ki najbolj zadostijo potrebam uporabnika?
 - Konsistentnost. Ali so kakšne zahteve med seboj v konfliktu?
 - Popolnost. Ali so upoštevane vse funkcije, ki jih zahteva uporabnik?
 - Realnost. Ali so zahteve lahko realizirane s predvideno tehnologijo in v okviru omejitev stroškov?
 - ♦ Preverjanje posameznih zahtev:
 - Preverljivost. Ali je zahtevo realno mogoče preveriti?
 - Razumljivost. Ali je zahteva pravilno razumljena?
 - Sledljivost. Ali je izvor zahteve jasno naveden?
 - Prilagodljivost. Ali je zahtevo mogoče spremeniti brez velikega vpliva na druge zahteve?

Načrtovanje sistema ^{1/2}

- ❖ V fazi načrtovanja sistema določimo **arhitekturo** programskega sistema:
 - ♦ strukture programskih komponent,
 - ♦ zunanjih lastnosti komponent in
 - ♦ relacij med komponentami.
- ❖ Razdelitev zahtev celotnega sistema med posamezne komponente (določitev funkcij).

Načrtovanje sistema ^{2/2}

- ❖ Delitev sistema na komponente (dekompozicija)
 - ♦ Razdeli sistem tako, da se doseže čim večja neodvisnost komponent (bistveno za vzdrževanje).
 - ♦ Dekompozicija poteka vertikalno in horizontalno:
 - **Particije** vertikalno delijo sistem v več neodvisnih (ali šibko povezanih) podsistemov, ki nudijo storitve na istem nivoju abstrakcije.
 - **Nivoji** (layer) so podsistemi, ki uporabljajo storitve nižjega nivoja in nudijo storitve višjemu nivoju.
 - ♦ Komponente sistema so procesne in podatkovne.

Načrtovanje programa 1/2

- ❖ Glede na izbran načrt izvedbe sistema se načrtuje posamezne programske komponente.
 - ♦ Celotna struktura programa
 - ♦ Moduli, ki jih je potrebno izvesti (kodirati)
 - ♦ Načrt podatkovne baze in datotečnih formatov.
- ❖ V tem koraku se vsaka komponenta definira tako podrobno, da na osnovi tega lahko sledi izvedba programa.

Načrtovanje programa 2/2

- ❖ Izdelki načrtovanja programa so:
 - ♦ Specifikacije programa (npr. diagrami, psevdokoda, opisi)
 - ♦ Načrt datotek (organizacija, način dostopa...)
 - ♦ Specifikacija strojne opreme
 - ♦ Načrt testiranja
 - ♦ Časovni načrt izvedbe
- ❖ Konča se s tehničnim pregledom.

Izvedba 1/2

❖ V fazi izvedbe na osnovi načrta programa zgradimo programski produkt:

- ♦ Množico programov ali programskih enot, ki so lahko
 - napisane v ta namen
 - pridobljene od drugod
 - spremenjene obstoječe programske enote
- ♦ Podatkovne strukture, ki programom omogočajo manipulacijo (obdelavo) podatkov (informacij).
- ♦ Dokumentacijo, ki opisuje delovanje (in uporabo) programov.

Izvedba 2/2

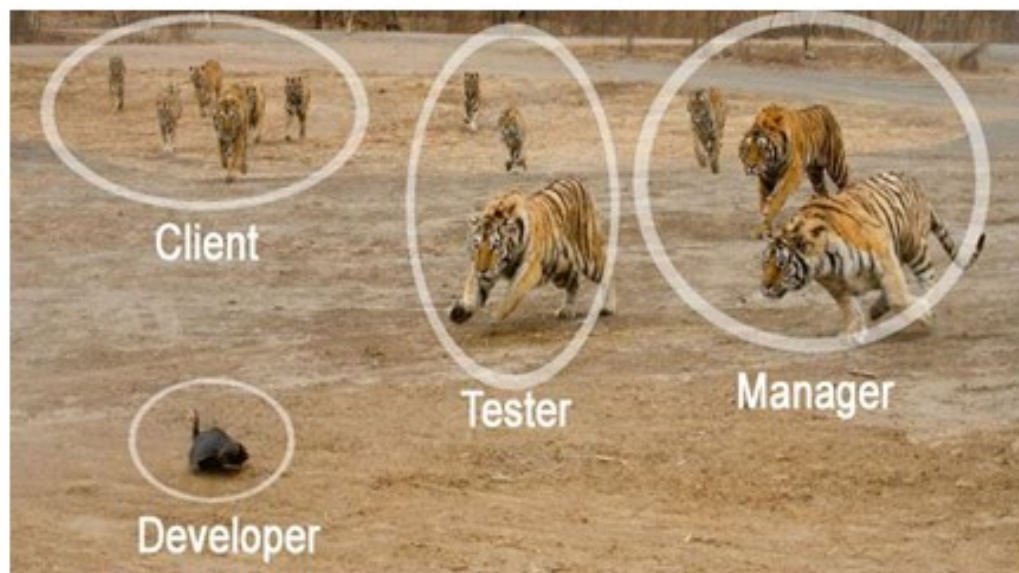
❖ Izvedba se konča z verifikacijo:

- ♦ Posamezne komponente se testira glede na skladnost s specifikacijami programa.
- ♦ Posamezne komponente se združi v programski sistem (faza integracije) in testira glede na specifikacije sistema.

Testiranje

- ❖ V fazi testiranja se izvaja validacija, s katero se preveri ali programski produkt ustreza potrebam uporabnikov.
 - ♦ Poteka po vnaprej pripravljenem načrtu, ki izhaja iz specifikacije zahtev.
 - ♦ Po odpravi (morebitnih) ugotovljenih nepravilnosti se testiranje ponovi.
- ❖ V času testiranja se izdelava/dodela navodila za uporabo sistema.

Testiranje 😊



Predaja produkta

❖ Prevzem s strani uporabnika

- ♦ Skupaj z uporabnikom se preveri delovanje celotnega sistema. Uporabnik ob tem potrdi ali zavrne ustreznost sistema.
- ♦ Testiranje se lahko izvrši pri razvijalcu (FAT) ali pri uporabniku (SAT).

❖ Predaja produkta naročniku

- ♦ Celotni sistem je dostavljen naročniku in predan v obratovanje.
- ♦ Uporabnikom so na voljo navodila, organizirana je pomoč uporabnikom...

Obratovanje in vzdrževanje

- ❖ Sistem mora izpolnjevati uporabnikove potrebe – pravilno in neprestano.
- ❖ Vzdrževanje pomeni spreminjanje programskega produkta:
 - ♦ Korektivne spremembe (odpravljanje napak).
 - ♦ Adaptivne spremembe (prilagajanje spremenjenim pogojem, npr. razširitev funkcionalnosti, spremembe vmesnikov...)
 - ♦ Perfektivne spremembe (izboljšanje programskega produkta, npr. prenos na nove platforme, dodatne funkcionalnosti...)
 - ♦ Preventivne spremembe (odpravljanje poslabševanja programskega produkta)
- ❖ Življenjski cikel programskih produktov se konča s prenehanjem uporabe.

Generične faze prog. procesa

❖ Definicija problema	Definicija
❖ Študija izvedljivosti	Analiza
❖ Analiza in definiranje zahtev	
❖ Načrtovanje sistema	Načrtovanje
❖ Načrtovanje programa	
❖ Izvedba	Izvedba
❖ Testiranje	
❖ Predaja produkta	
❖ Obratovanje in vzdrževanje	Obratovanje

Razvojni modeli programskega procesa

- ❖ Razvojni model programskega procesa je načrt izvajanja faz programskega procesa.
- ❖ Vsak programski proces vsebuje vse faze programskega procesa, ki pa so lahko izvedene **formalno** ali **neformalno** in v različnem zaporedju.
 - ♦ Študija izvedljivost se na primer ne more izvajati brez predhodne definicije problema.
 - ♦ "Ad hoc" pristop pomeni, da določeno/e faze programskega procesa izvajamo neformalno.
 - ♦ Načrtovanje programa običajno razkrije pomanjkljivosti specifikacije zahtev.
 - ♦ ...

Prednosti formalnih procesov

- ❖ Formalno izvajanje faz programskega procesa prinaša vrsto prednosti.
 - ♦ Boljši nadzor nad sredstvi (resources).
 - ♦ Boljši odnos s stranko.
 - ♦ Krajši čas razvoja.
 - ♦ Nižji stroški.
 - ♦ Višja kvaliteta in zanesljivost.
 - ♦ Večja produktivnost.
 - ♦ Boljša organizacija in koordinacija dela.
 - ♦ Manj stresno delo.
- ❖ Prednosti so bolj izražene pri večjih projektih.

Prednosti neformalnih procesov?

- ❖ Pri manjših projektih so prednosti manj izražene in pogosto pride ugibanj o potrebnosti formalnega izvajanja faz:
 - ♦ Manj potreb po dokumentiranju?
 - ♦ Manj potrebnih sredstev za organizacijo in koordinacijo?
 - ♦ Večja svoboda in motiviranost razvijalcev?
 - ♦ ...
- ❖ Neformalno izvajanje določenih faz je lahko vzrok za težave v kasnejših fazah!

Razvojni modeli prog. procesov



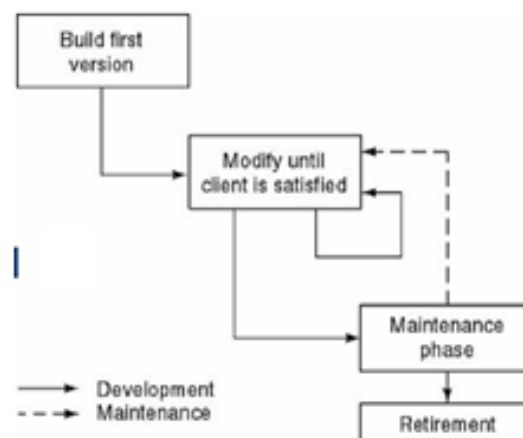
“Build and fix”

❖ Težave

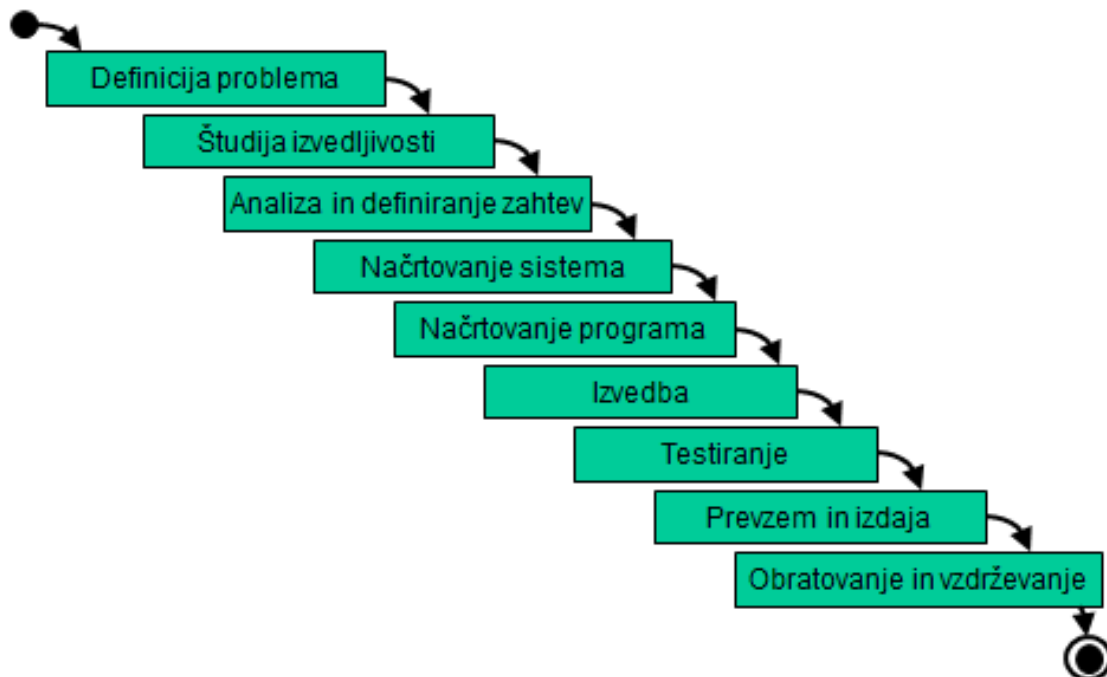
- ♦ Ni specifikacij
- ♦ Ni načrta izvedbe
- ♦ Ni določenih faz
- ♦ Ni mejnikov
- ♦ ...

❖ Neprimeren pristop

- ♦ Nemogoč nadzor nad tekom projekta
- ♦ Težavno zagotavljanje kvalitete
- ♦ Visoka obremenjenost stranke, padec zaupanja v produkt...



Linearni model ^{1/3}



Linearni model ^{2/3}

❖ Linearni (klasični, kaskadni) model, ang. waterfall model:

- ♦ Faze si sledijo ena za drugo in se ne prekrivajo. Naslednja faza se začne po zaključeni predhodni fazi.
- ♦ Vsaka faza se zaključi s pregledom oziroma testiranjem.
- ♦ Možen le v primeru dobro določenih zahtev/ciljev in jasnega načina izvedbe, saj je upoštevanje kasnejših sprememb težavno.
- ♦ Zahteva potrpljenje stranke – delujoča verzija ji je na voljo šele ob predaji končnega produkta

Linearni model ^{3/3}

❖ Prednosti:

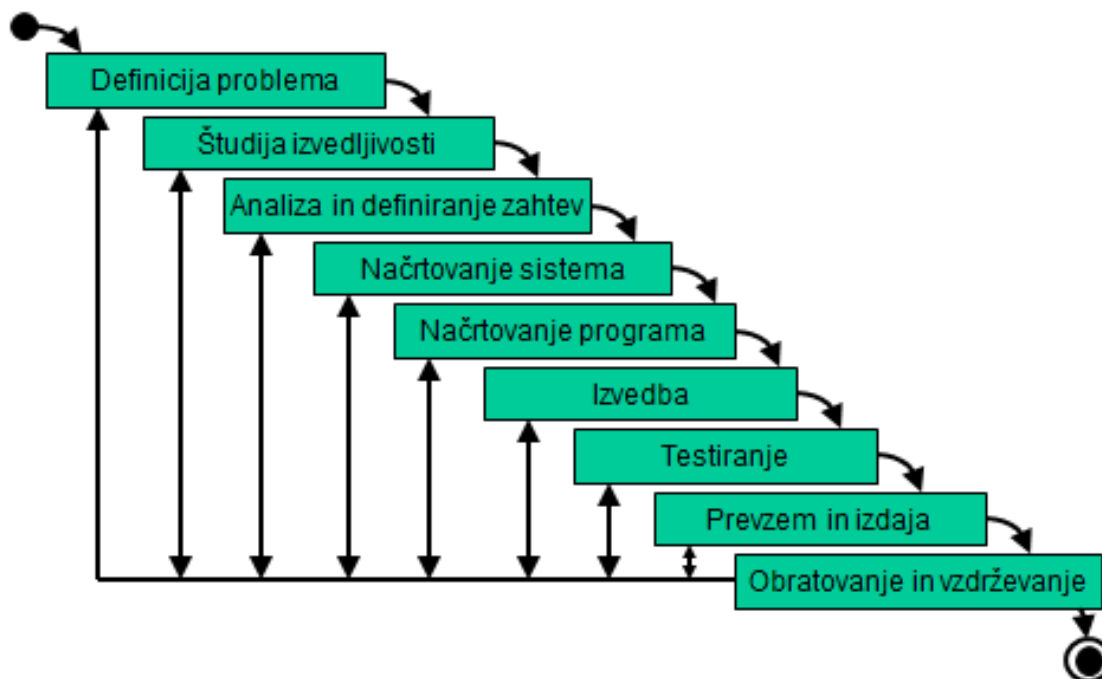
- ♦ preglednost,
- ♦ ločenost faz,
- ♦ stalen nadzor nad kvaliteto,
- ♦ dober nadzor nad stroški.

❖ V praksi šele vsak naslednji korak omogoča širše/boljše razumevanje prejšnjih faz, zato so pogosto nujne revizije predhodnih korakov – Linearni model tega ne omogoča.

Linearni model s povratki

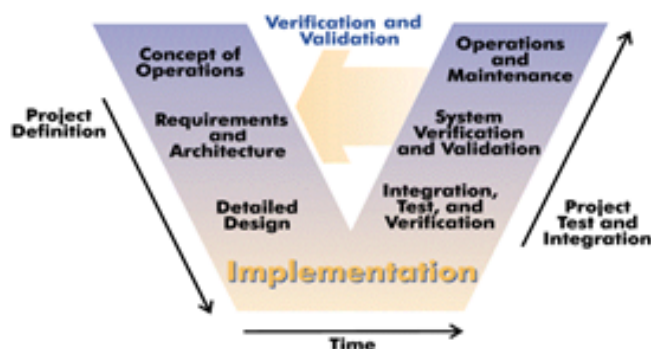
- ❖ Omogoča revizije predhodnih korakov.
- ❖ Možna je izvedba z delnim prekrivanjem faz.

Linearni model s povratki

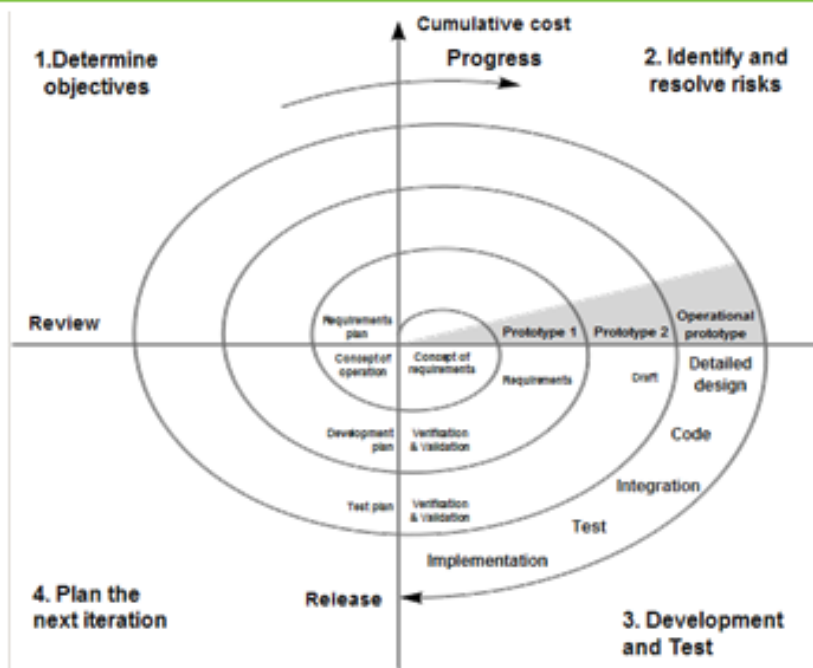


“V” model

- ❖ Razširitev linearnega modela.
- ❖ Vsaka faza razvoja je povezana s pripadajočo fazo testiranja.
- ❖ V primeru ugotovljenih nepravilnosti je potrebna vrnitev v pripadajočo fazo razvoja.



Spiralni model 1/9



Vir: http://en.wikipedia.org/wiki/Spiral_model

Spiralni model 2/9

- ❖ Vsak cikel vsebuje faze linearnega modela.
- ❖ V vsakem ciklu se izdela prototip.
- ❖ Vsak cikel sestoji iz štirih kvadrantov:
 - ♦ Določitev ciljev, alternativ in omejitev.
 - ♦ Ocenjevanje alternativ, identifikacija tveganj.
 - ♦ Razvoj produkta trenutnega cikla.
 - ♦ Načrtovanje naslednjega cikla.

Spiralni model 3/9

- ❖ Določitev ciljev, alternativ in omejitev:
 - ♦ **Cilji:** funkcionalnost, prepustnost/zmogljivost, določitev vmesnikov, faktorji za uspeh projekta...
 - ♦ **Alternative:** razvoj, uporaba že razvitih komponent, nakup komponent/produkta, podizvajalci...
 - ♦ **Omejitve:** stroški, časovne omejitve, vmesniki...

Spiralni model 4/9

- ❖ Ocenjevanje alternativ, identifikacija tveganj:
 - ♦ Presoja alternativ glede na cilje in omejitve
 - ♦ Identifikacija tveganj (pomanjkanje izkušenj, nove tehnologije, kratki časovni roki...)
 - ♦ Ocenitev tveganj (možnost neuspeha, smiselnost nadaljnjega razvoja...)

Spiralni model 5/9

- ❖ Razvoj produkta trenutnega cikla:
 - ♦ Cikli linearnega modela: načrtovanje, izvedba, testiranje.
 - ♦ Kontrola kvalitete (pregledi, verifikacije, validacija)

Spiralni model 6/9

- ❖ Načrtovanje naslednjega cikla:
 - ♦ Projektno planiranje naslednjega cikla
 - ♦ Upravljanje konfiguracije (identifikacija, organiziranje in kontrola sprememb razvoja programske opreme v projektni skupini)
 - ♦ Načrtovanje testiranj
 - ♦ Načrtovanje namestitev sistema

Spiralni model 7/9

❖ Prednosti:

- ♦ Omogoča zgodnjo identifikacijo nepremostljivih ovir – z majhnimi stroški.
- ♦ Uporabniki zgodaj dobijo vpogled v sistem (prototipi)
- ♦ Najprej se razvijejo najbolj kritične funkcije sistema
- ♦ Nepopolno načrtovanje je sprejemljivo
- ♦ Uporabniki imajo vpogled v razvoj sistema
- ♦ Zgodnja povratna informacija od uporabnikov

Spiralni model 8/9

❖ Slabosti:

- ♦ Izguba časa zaradi večkratnega ponovnega načrtovanja, predefiniranja zahtev, analize tveganj in izgradnje prototipov.
- ♦ Model je kompleksen in ne določa konca projekta (število ciklov).
- ♦ Razvijalci v nekaterih kvadrantih niso izkoriščeni.

Spiralni model 9/9

❖ Kdaj ga uporabiti?

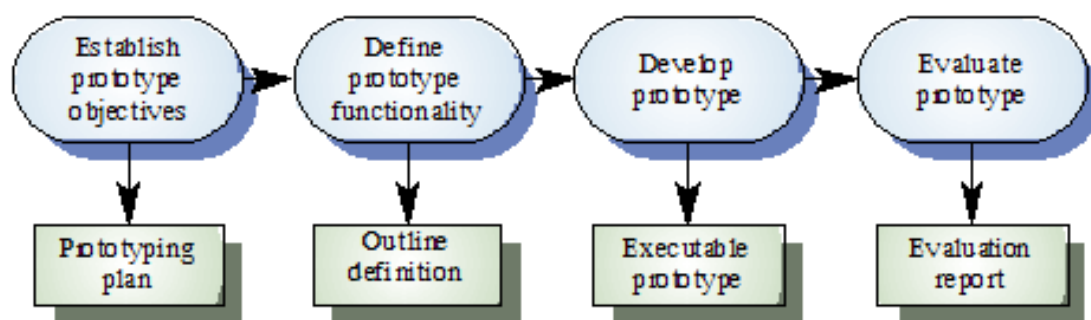
- ♦ Ko je upravičena izgradnja prototipov
- ♦ Ko je pomembno ocenjevanje tveganj – za bolj tvegane projekte.
- ♦ Ko je dopustno daljše časovno obdobje razvoja.
- ♦ V primeru kompleksnih zahtev.
- ♦ Ko so pričakovane večje spremembe zahtev.

Razvoj z uporabo prototipov 1

- ❖ Razvoj z uporabo prototipov je način hitrega razvoja sistema (ang. rapid development).
- ❖ Prvotni namen prototipa je pomoč stranki in razvijalcem pri razumevanju zahtev sistema.
 - ♦ Uporabniki lahko preverijo kako se odražajo njihove zahteve.
 - ♦ Prototip razkrije napake in manjkajoče zahteve.
- ❖ Razvoj prototipov lahko razumemo kot aktivnost zmanjševanja rizika pri specificiranju zahtev.
- ❖ Prototip je na voljo zgodaj v programskem procesu in lahko služi tudi za usposabljanje uporabnikov in testiranje sistemov.

Razvoj z uporabo prototipov 2

❖ Proces razvoja prototipa



Vir: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

Razvoj z uporabo prototipov 3

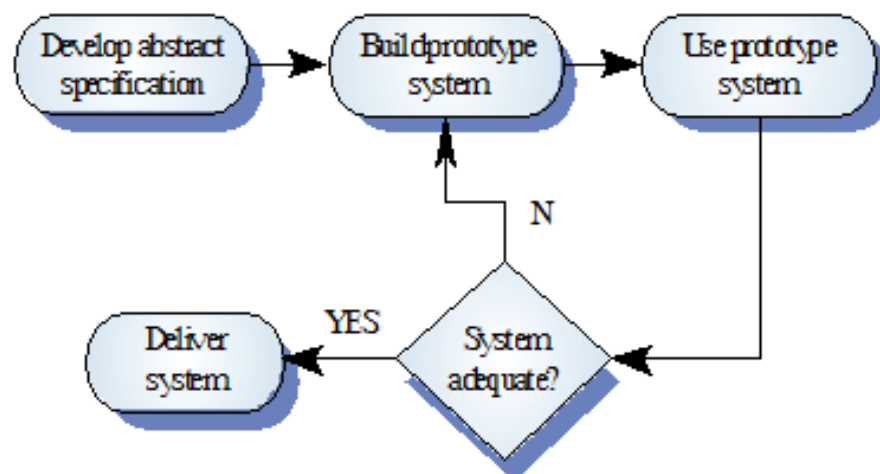
❖ V preteklosti je prototip veljal za manjvrednega zato je izgradnji prototipa vedno sledil nadaljnji razvoj. Danes poznamo v programskih procesih dve možnosti uporabe prototipov:

- ❖ Evolucijski proces (npr. spiralni model)
 - Začetni prototip je postopno izboljššan in tako preide v končni sistem.
 - Namen prototipa je zgraditi končni sistem.
 - Razvoj se začne pri zahtevah, ki so najbolj jasne.
- ❖ Razvoj z opustitvijo prototipa
 - Prototip je namenjen izključno odkrivanju zahtev in se po specificiranju zahtev opusti. Končni sistem se razvije neodvisno.
 - Razvoj se začne pri zahtevah, ki so najbolj nejasne.

Evolucijski razvoj 1/4

- ❖ Evolucijski razvoj je potreben takrat, kadar specifikacije ne morejo biti izdelane pred izvedbo sistema (prototipom), npr. sistemi umetne inteligence.
- ❖ Verifikacija ni mogoča, ker ni specifikacij. Validacija se izvaja kot demonstracija primernosti sistema.
- ❖ Evolucijski razvoj temelji na tehnikah, ki omogočajo hiter razvoj prototipov.

Evolucijski razvoj 2/4



Vir: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

Evolucijski razvoj ^{3/4}

- ❖ Prototip v določenih primerih lahko služi kot **specifikacija sistema, ne pa vedno in povsem:**
 - ♦ Nekaterih zahtev v prototip ni mogoče vključiti (varnostno kritične funkcije)
 - ♦ Prototipne izvedbe sistema pravno ni mogoče obravnavati kot pogodbo.
 - ♦ Nefunkcionalnih specifikacij ni mogoče ustrezno preveriti na prototipu (varnost, možnost testiranja, možnost vzdrževanja, razširljivost, nadgradljivost...)

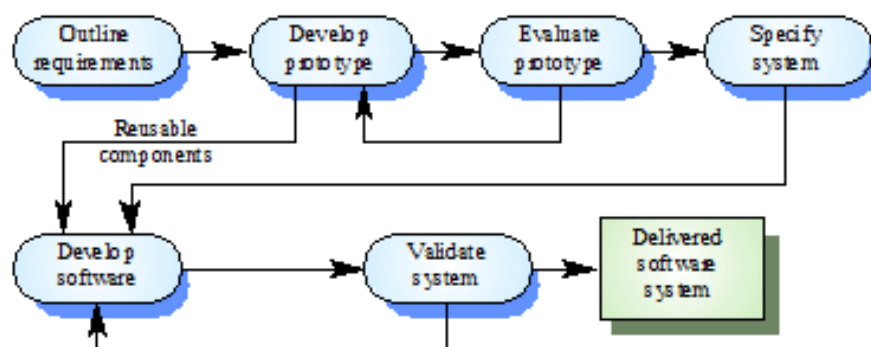
Evolucijski razvoj ^{4/4}

- ❖ Prednosti in slabosti:
 - ♦ + Hitra dobava. Včasih je bolj pomembna od funkcionalnosti ali od možnosti dolgoročnega vzdrževanja.
 - ♦ + Vključenost uporabnika povečuje verjetnost, da sistem ustreza uporabnikovim potrebam, hkrati pa povečuje verjetnost uporabe sistema.
 - ♦ - Obstoječi procesi managementa tipično predpostavljajo linearni model.
 - ♦ - Razvoj prototipa (brez specifikacij) zahteva dodatne sposobnosti razvijalcev.
 - ♦ - Težavnost vzdrževanja. Neprestano spreminjanje negativno vpliva na strukturo sistema, zato je dolgoročno vzdrževanje težavno.
 - ♦ - Pogodbene težave. Uporabniki morajo prevzeti dodatne obveznosti, večkratna predaja produktov.

Razvoj z opustitvijo prototipa 1/3

- ❖ Namen prototipa je zmanjšati tveganje pri določitvi zahtev.
- ❖ Prototip je razvit na osnovi začetnih specifikacij zahtev. Namenjen je eksperimentiranju (ugotavljanju lastnosti različnih izvedb, tehnologij...) in se ga za tem zavrže.
- ❖ Prototip NI namenjen kakršni koli uporabi v končnem sistemu:
 - ♦ Prototip ni narejen z upoštevanjem vseh potrebnih lastnosti sistema.
 - ♦ Prototip ni dokumentiran in bi ga bilo zato težko dolgoročno vzdrževati.
 - ♦ Arhitektura prototipa je napravljena "ad-hoc" zato je praviloma neprimerna za dolgotrajno vzdrževanje.

Razvoj z opustitvijo prototipa 2/3



Vir: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

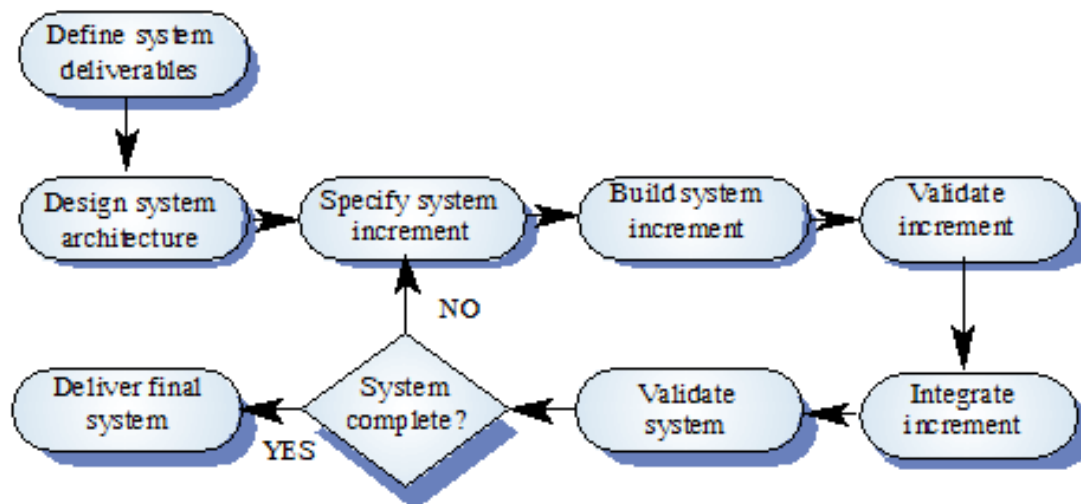
Razvoj z opustitvijo prototipa 3/3

- ❖ Razvijalci so pogosto pod pritiskom, da prototip predajo kot končni sistem.
- ❖ Zakaj to ni primerno?
 - ♦ Prototipi običajno ne ustrezajo nefunkcijskim zahtevam in je le te kasneje težko ali celo nemogoče zagotoviti.
 - ♦ Prototip je nedokumentiran.
 - ♦ Arhitektura/struktura prototipa ni namenjena kasnejšemu dograjevanju in z nadaljnjim razvojem/vzdrževanjem postane neustrezna.
 - ♦ Pri razvoju prototipa se pogosto/večinoma ne upošteva običajnih standardov kakovosti!

Postopni (fazni) razvoj 1/3

- ❖ Sistem je razvit in dostavljen uporabniku v več fazah.
- ❖ Osnova za razvoj (prve faze) je arhitekturni načrt celotnega sistema!
- ❖ Faze, ki so že predane uporabniku, že zagotavljajo neko zaokroženo (čeprav nepopolno) funkcionalnost.
- ❖ Uporabniki lahko preverjajo ali celo uporabljajo rešitve prvih faz, med tem ko so naslednje faze še v razvoju.
- ❖ Na ta način fazni pristop nudi prednosti razvoja s prototipi in hkrati zagotavlja boljšo preglednost projekta, boljše vodenje projekta in v končni fazi boljšo strukturo produkta.

Postopni (fazni) razvoj 2/3



Vir: Ian Sommerville, Software Engineering, 6th edition. Chapter 8, 2000.

Postopni (fazni) razvoj 3/3

- ❖ Osnovni sistem lahko hitro preide v uporabo (produkcijo), s tem se hitro začnejo vračati vložena sredstva.
- ❖ Razvoj naslednjih faz temelji na izkušnjah uporabnikov prejšnjih faz. Posledica tega je boljše razumevanje zahtev v naslednjih fazah.
- ❖ Fazni razvoj je zelo pogost ...
Ali najdeš kakšen primer med programsko opremo za širšo uporabo?

Unified Process

- ❖ Unified Process (UP) (enoten process) je ogrodje, ki ga je mogoče prilagoditi specifičnim organizacijam oziroma projektom.
- ❖ IBM je privzel UP in ga trži pod imenom Rational Unified Process.
- ❖ Ogrodje določa faze razvoja produkta in inženirske discipline potrebne za realizacijo teh faz.
- ❖ Vsaka faza je zgrajena iz korakov, ki so osnovni gradniki procesa.

Unified Process

- ❖ Projekt sestoji iz štirih faz:
 - ♦ **začetek (ang. inception):** definicija problema, študij izvedljivosti, zaključi se z mejnikom ciljev (ang. Lifecycle Objective Milestone)
 - ♦ **dopolnitev (ang. elaboration):** analiza in definiranje zahtev, načrtovanje sistema, načrtovanje programa, zaključi se z arhitekturnim mejnikom (ang. Lifecycle Architecture Milestone)
 - ♦ **gradnja (ang. construction):** načrtovanje programa, izvedba, testiranje, konča se z mejnikom uporabljivosti (ang. Initial Operational Capability Milestone)
 - ♦ **prehod (ang. transition):** validacija, prevzem in izdaja, šolanje uporabnikov, konča se z mejnikom izdaje produkta (ang. Product Release Milestone)

Unified Process

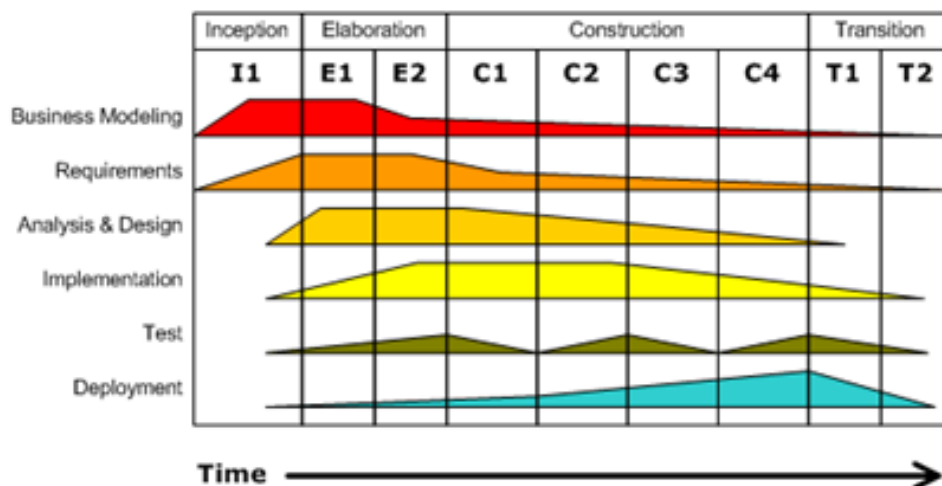
❖ Unified Process definira šet inženirskih disciplin:

- ♦ poslovno modeliranje (ang. bussiness modeling)
- ♦ definiranje zahtev (ang. requirements discipline)
- ♦ analiza in načrtovanje (ang. analysis and design)
- ♦ izvedba (ang. implementation)
- ♦ testiranje (ang. test discipline)
- ♦ namestitev (ang. deployment)

Unified Process

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



Unified Process

- ❖ Proces je zgrajen iz osnovnih elementov, ki določajo kaj naj bo izdelano, kakšna so potrebna znanja in kako doseči cilj:
 - ♦ **Vloge (kdo):** znanja, kompetence, odgovornosti.
 - ♦ **Produkt (kaj):** rezultat naloge - dokumenti, modeli, podatki, porogrami.
 - ♦ **Naloge (kako):** opis postopka za dosego cilja.

Unified Process

- ❖ Dobra praksa, ki jo določa UP:
 - ♦ Razvijaj iterativno
 - ♦ Upravljaljajte zahteve
 - ♦ Uporabljaljajte komponente
 - ♦ Modelirajte vizualno
 - ♦ Preverjaljajte kvaliteto
 - ♦ Obvladuj spremembe

Ogradja programskih procesov

- ❖ Organizacija lahko zgradi svoje ograje programskih procesov ter z njim definira svoje programske procese.
- ❖ Vsak proces sestoji iz korakov, ki
 - ♦ določajo fazo,
 - ♦ zahtevajo določeno inženirsko disciplino,
 - ♦ določajo vloge udeležencev v procesu,
 - ♦ rezultirajo v naprej določenih produktih,
 - ♦ so opremljeni z napotki za izvedbo.
- ❖ Orodja za gradnjo programskih procesov:
 - ♦ Eclipse Process Framework Project (EPF)
<http://www.eclipse.org/epf/>

Zrelostni nivoji programskih procesov

- ❖ Mera efektivnosti programskih procesov v podjetju:
 1. Začetna - uspešnost odvisna od posameznikov.
 2. Ponovljiva - uvedeno osnovno vodenje projektov za sledenje stroškov, rokov in funkcionalnosti. Proces je mogoče ponoviti na podobnih projektih.
 3. Določena – programski procesi so dokumentirani, standardizirani in potrjeni znotraj podjetja. Na teh procesih temeljijo vsi projekti podjetja.
 4. Vodena – pri vseh projektih se meri vrsto indikatorjev, ki govorijo o uspešnosti projekta in s tem programskega procesa.
 5. Optimirana – stalno izboljševanje programskih procesov na podlagi kazalcev uspešnosti, novih idej in tehnologij.

Cena programskega inženirstva

- ❖ Okvirno 60% cene predstavlja razvoj in 40% testiranje. Za programsko opremo po meri je značilno, da stroški prilagajanja po prvi predaji stranki pogosto presegajo stroške razvoja.
- ❖ Stroški so odvisni od tipa razvijanega sistema in zahtev kot sta prepustnost sistema in zanesljivost.
- ❖ Porazdelitev stroškov je odvisna od uporabljenega modela programskega procesa.

Porazdelitev stroškov



Vir: Ian Sommerville, Software Engineering, 8th edition. Chapter 1, 2007.

3 Agilni pristopi

Pogosto slišimo očitke formalnim pristopom češ da zahtevajo prekomerno dokumentiranje, razvijalcem jemljejo motivacijo in otežujejo prilagodljivost programskih procesov. Čeprav so ti očitki večinoma neutemeljeni, je res, da razvoj ob določenih okoliščinah lahko poteka tudi manj formalno, na podlagi t.i. agilnih pristopov, ki so postali zelo priljubljeni in uporabljeni v številnih uspešnih programskih podjetjih. Gre za pristope, ki formalnost nadomeščajo z različnimi pravili skupinskega razvoja, s poudarkom na medsebojni komunikaciji, neprestani integraciji in drugo. Dve izmed najpomembnejših agilnih metodologij sta ekstremno programiranje (XP) in scrum. Obstaja še vrsta drugih, ne glede na izbor pa imajo podobne prednosti in tudi pomanjkljivosti. Načeloma so primerne za manjše razvojne time, kjer medsebojna se komunikacija lahko odvija tekoče in brez obsežnega sestankovanja, za doseganje uspehov pa potrebujejo še motivirane in izkušene razvijalce. Poglavitna prednost glede na formalne pristope je v prilagodljivosti procesov, vendar pa je tudi pri tem potrebno biti previden, saj ima (pre)pogosto spreminjanje funkcionalnosti negativne posledice ne glede na izbrano metodologijo. Omejitve agilnih pristopov se kažejo pri zagotavljanju kakovosti, kjer obsežno testiranje ne more popolnoma nadomestiti skrbnega načrtovanja, na primer v zdravstvu, vojski, nuklearni tehnologiji.

Obvladovanje negotovosti

- ❖ Razvoj programskih produktov je proces učenja, s katerim zmanjšujemo negotovost:
 - ♦ obsega projekta
 - Določitev funkcionalnosti...
 - ♦ potrebnih sredstev
 - Člani razvojne ekipe (čas), potrebna oprema...
 - ♦ doseganja časovnih rokov
 - Čas predaje produktov...

Tradicionalni pristop

- ❖ Tipičen primer je linearni programski proces.
- ❖ Negotovost želimo obvladati kar se da zgodaj v projektu:
 - ♦ Delovanje sistema želimo popolno opisati v fazi analize in definiranja zahtev.
 - ♦ Zgradbo sistema določimo v fazi načrtovanja sistema
 - ♦ Zgradbo modulov (programov, razredov) definiramo v fazi detajlnega načrtovanja.
 - ♦ Šele ko je negotovost glede izvedbe minimalna, se lotimo programiranja (kodiranja).
- ❖ Negotovost zmanjšuje vrsta produktov v času izvajanja projekta (specifikacije zahtev, načrt sistema, načrti programov...)
- ❖ Zmanjšana negotovost pripomore k lažji izvedbi in vzdrževanju.

Tradicionalne težave

- ❖ Razvoj praktično vsakega programskega produkta se sooča z vrsto negotovosti, ki jih v popolnosti težko izničimo v tradicionalno definiranih fazah projekta:
 - ♦ Zahteve se izbistrijo šele v fazi načrtovanja ali celo izvedbe ali testiranja (zaznane nekonsistence, pomanjkljivosti...)
 - ♦ Izvedba sistema se zares izoblikuje šele ob izvedbi (kodiranju) lahko pa celo kasneje v času testiranja.
- ❖ Posledice:
 - ♦ Produkti kot so specifikacije zahtev, načrti izvedbe itd. so ob izgradnji nepopolni in jih je potrebno dopolnjevati v kasnejših fazah.
 - ♦ V času izvajanja projekta se spreminjajo obseg projekta, časovni plan in stroški!

Kako drugače?

- ❖ Čemu obsežno predhodno planiranje in dokumentiranje, če se prava funkcionalnost in izvedba izkažeta šele v času kodiranja in testiranja?
 - ♦ Neformalno izvajanje faz specificiranja zahtev in načrtovanja - v času izvedbe (kodiranja)!
 - ♦ Pogosto in avtomatsko testiranje tekom izvedbe!
 - ♦ Vključitev naročnika/uporabnika v proces razvoja!

Kako drugače?

- ❖ **Dokumentiranje** je namenjeno komunikaciji – zakaj ne bi komunicirali kar se da učinkovito?



- Neposredna komunikacija, brainstormingi, programiranje v parih...

Agilni pristop

- ❖ **Agilni pristopi določene faze procesa izvajajo neformalno z namenom**
 - ♦ povečanja prilagodljivosti procesa in odzivnosti na spremembe
 - ♦ skrajšanja časa do predaje produkta stranki (zgodnejše povračilo investicije, boljše sodelovanje s stranko)
- ❖ **Agilni pristopi formalnost nadomestijo**
 - ♦ z neformalnim odprtim odnosom znotraj razvojnega tima,
 - ♦ s sodelovanjem znotraj tima in navzven,
 - ♦ z aktivnejšim preverjanjem kakovosti.

Vrednote agilnih pristopov

- ❖ Posameznik in odnos med posamezniki je pomembnejši od procesov in orodij.
- ❖ Delujoča programska oprema je pomembnejša od obsežne dokumentacije.
- ❖ Sodelovanje stranke je boljše od pogodbenih pogajanj.
- ❖ Odzivnost na spremembe je pomembnejša kot striktno sledenje planu.

Principi agilnih pristopov

- ❖ Razdelitev funkcionalnosti na majhne neodvisne sklope.
- ❖ Pogoste izdaje novih verzij programov.
- ❖ Spremembe zahtev so dobrodošle, četudi pozno.
- ❖ Dnevno sodelovanje tehničnega in poslovnega osebja.
- ❖ Neposredna vključenost predstavnika stranke v tim.
- ❖ Najboljša je osebna komunikacija (na štiri oči)
- ❖ Projekti naj temeljijo na motiviranih posameznik, ki jim je potrebno zaupati.
- ❖ Enostavnost je vrednota
- ❖ Timi naj se organizirajo sami

Primeri agilnih pristopov

- ❖ **Extreme Programming (XP)**
- ❖ **Scrum**
- ❖ Dynamic Software Development Method (DSDM)
- ❖ Feature Driven Development (FDD)
- ❖ Adaptive Software Development (ASD)
- ❖ Crystal Clear
- ❖ Rapid Application Development (RAD)
- ❖ ...

XP - ekstremno programiranje

- ❖ *Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.*

Vir: <http://www.xprogramming.com/xpmag/whatisxp>

XP: ekstremnosti

- ❖ Ekstremnosti ekstremnega programiranja
 - Če so pregledi kode dobri, naj se izvajajo ves čas (programiranje v parih)
 - Če je testiranje koristno, naj vsakdo ves čas testira ("unit testi")
 - Če je enostavnost koristna, naj bo sistem kolikor se le da enostaven, da le zadosti zahtevam obstoječe funkcionalnosti.
 - Če je načrtovanje koristno, naj vsi dnevno načrtujejo (stalno preurejanje).
 - Če je arhitektura pomembna, naj jo načrtujejo in izboljšujejo vsi.
 - Če je pomembno testiranje ob združevanju ("integration testing"), naj se združevanje in testiranje izvaja večkrat dnevno – zvezna integracija.
 - Če so dobre majhne iteracije, naj bodo le te res majhne (ure in ne tedni)

XP: način izvajanja projektov

- ❖ Določitev zgodb (ang. stories) - lastnosti, ki jih želi stranka/uporabniki.
- ❖ Ocena trajanja in stroškov za vsako zgodbo.
- ❖ Izbira zgodb za naslednjo izdajo/verzijo.
- ❖ Razdelitev nalog (ang. tasks) posamezne izdaje.
- ❖ Priprava testnih primerov za vsako nalogo.
- ❖ Programiranje v parih. Brez specializacije razvojnikov. Prisotnost predstavnika stranke.
- ❖ Zvezna integracija.

XP: principi (1-6)

1. "Planning game" – načrtovanje: določitev ciljev naslednje verzije z upoštevanjem poslovnih potreb in tehničnih ocen.
2. "Small releases" – v uporabo se preda enostaven sistem, ki se ga nadgrajuje z majhnimi in pogostimi nadgradnjami (nove izdaje/verzije).
3. "Metaphor" – enostavna zgodba o delovanju končnega sistema, na osnovi katere se vodi razvoj.
4. "Simple design" – sistem mora biti načrtovan kolikor se le da enostavno (nepotrebno kompleksnost je potrebno odstraniti takoj, ko je ugotovljena)
5. "Testing" – testiranje poteka neprestano, "unit testi" so osnova za programiranje funkcionalnosti.
6. "Refactoring" – stalno preurejanje strukture sistema za zmanjševanje kompleksnosti in v izogib podvajanju kode.

XP: principi (7-12)

7. "Pair programming" – programiranje v parih – vedno dva programerja za enim računalnikom.
8. "Collective ownership" – skupna programska koda celotnega sistema, ki jo lahko kdorkoli kadarkoli spremeni.
9. "Continuous integration" – spremembe se vgrajuje v sistem večkrat dnevno, po vsaki opravljeni nalogi.
10. "40-hour week" - nikoli se ne dela več kot 40 ur tedensko.
11. "On-site customer" – uporabnik je vključen v razvojni tim v smislu stalne razpoložljivosti za odgovore na vprašanja razvijalcev.
12. "Coding standards" – striktno se upošteva pravila programiranja, s poudarkom na načinu komunikacije preko programske kode.

XP ☺



Skram (ang. Scrum)



Scrum: vloge 1/4

- ❖ Scrum definira skelet programskega procesa, v katerem sodelujejo:
 - ♦ **Mojster skrama** (ang. Scrum Master), ki skrbi za nemoteno izvedbo procesa (vloga projektne vodje) . NI vodja tima!
 - ♦ **Lastnik produkta**, (ang. Product Owner), ki zastopa interese stranke oz. uporabnikov in skrbi za pravilno usmerjanje projekta (poslovno, izbira funkcionalnosti...).
 - ♦ **Tim**, multidisciplinarna skupina 7(+/-2) članov, katere naloga je analiza, načrtovanje, izvedba, testiranje...

Scrum: vloge 2/4

- ❖ Zunanji akterji, ki vplivajo na skram (scrum) proces:
 - ♦ **Nosilci interesov** (ang. stakeholders) – stranke, uporabniki – so tisti, ki bodo imeli od projekta koristi in opravičujejo njegovo izvedbo. V procesu so vključeni v času pregledov (ang. Sprint reviews)
 - ♦ **Vodstvo** (organizacije) – ki vzpostavi pogoje za izvedbo projekta.

Scrum: vloge 3/4

- ❖ Vloge “pujsov” (ang. “pig” roles)
 - ◆ mojster skrama
 - ◆ člani tima
- ❖ Vloge “piščancev” (ang. “chicken” roles)
 - ◆ lastnik produkta
 - ◆ vodstvo
 - ◆ nosilci interesov

Scrum: vloge 4/4

Pojasnilo vlog v projektu:

- ❖ *A pig and a chicken are walking down a road. The chicken looks at the pig and says, “Hey, why don’t we open a restaurant?” The pig looks back at the chicken and says, “Good idea, what do you want to call it?” The chicken thinks about it and says, “Why don’t we call it ‘Ham and Eggs’?” “I don’t think so,” says the pig, “I’d be committed, but you’d only be involved.”*

Vir: http://en.wikipedia.org/wiki/Scrum_%28development%29

Scrum: potek projekta

- ❖ Lastnik produkta uredi in skrbi za zalogo produkta (ang. product backlog) – listo projektnih želja oz. značilnosti programskega produkta.
- ❖ Določitev zaloge naslednje verzije (ang. release backlog), kot podmnožice zaloge produkta. Vsaki projektni želji se oceni čas izvedbe.
- ❖ Zaloga verzije se porazdeli med več (tipično 4-12) delov, tako da izvedba vsakega dela traja do 4 tedne. Ti deli se imenujejo "sprinti" in pripadajoči sezname "zaloge sprintov".

Scrum: ilustracija



Scrum: sprint

❖ Sprint:

- ♦ V vsakem sprintu tim izdelava potencialno dobavljiv produkt.
- ♦ Zaloge sprinta se v času izvajanja sprinta ne sme spreminjati (nespremenljive zahteve).
- ♦ Ob koncu sprinta tim demonstrira razvito rešitev.

Scrum: sestanki ^{1/5}

❖ Predpisani sestanki:

- ♦ Načrtovanje sprinta
(ang. Sprint Planning Meeting)
- ♦ Dnevni skram (ang. Daily Scrum)
- ♦ Skram skramov (ang. Scrum of scrums)
- ♦ Pregledni sestanek sprinta
(ang. Sprint Review Meeting)
- ♦ Retrospektivni sestanek sprinta
(ang. Sprint Retrospective)

Scrum: sestanki 2/5

❖ Načrtovanje sprinta (ang. Sprint Planning Meeting)

- ♦ Ob začetku sprinta
- ♦ Izbira obsega dela (kaj naj bo narejeno)
- ♦ Priprava zaloge sprinta (Sprint Backlog) z ocenami potrebnega časa izvedbe (glede na tim).
- ♦ Sestanek omejen na 8 ur.

Scrum: sestanki 3/5

❖ Dnevni skram (ang. Daily Scrum)

- ♦ Vsak dan v obdobju sprinta.
- ♦ Hiter stoječ sestanek o statusu sprinta.
- ♦ Začetek vedno ob istem predpisanem času in na istem mestu.
- ♦ Vsi dobrodošli, besedo imajo le "pujski".
- ♦ Omejitev trajanja sestanka je 15 minut.
- ♦ Vsak član tima odgovori na tri vprašanja:
 - Kaj si naredil do včeraj?
 - Kaj nameravaš narediti danes?
 - Ali imaš kakršne koli probleme, ki te ovirajo pri izvedbi naloge?

Scrum: sestanki 4/5

❖ Skram skramov (ang. Scrum of scrums)

- ♦ Vsak dan po dnevnem skramu
- ♦ Sestanek med predstavniki več timov istega projekta.
- ♦ Vsak tim zastopa po en član.
- ♦ Vsebina enaka kot pri dnevnem skramu, oz.:
 - Kaj je tvoj tim naredil od našega zadnjega srečanja?
 - Kaj bo tvoj tim naredil do našega naslednjega srečanja?
 - Ali vas kar koli ovira pri izvedbi naloge?
 - Ali vidiš potrebo po sodelovanju timov?

Scrum: sestanki 5/5

❖ Pregledni sestanek sprinta (ang. Sprint Review Meeting)

- ♦ Pregled zaključenega in neopravljenega/nedokončanega dela.
- ♦ Predstavitve rezultatov nosilcem interesov (demo).
- ♦ Omejitev štirih ur.
- ♦

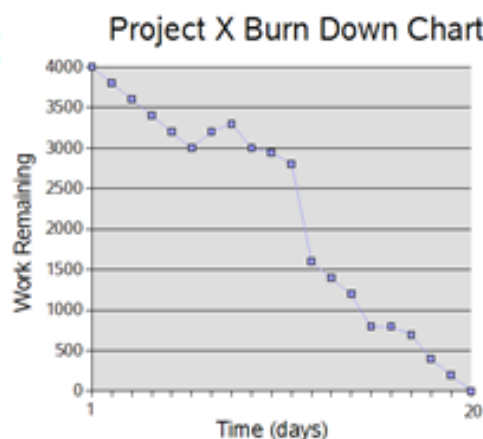
❖ Retrospektivni sestanek sprinta (ang. Sprint Retrospective)

- ♦ Vsi člani tima predstavijo svoje poglede na zaključen sprint.
- ♦ Skrb za stalno izboljševanje programskega procesa.
- ♦ Odgovori na dve vprašanji:
 - Kaj je v sprintu potekalo dobro?
 - Kaj bi se dalo izboljšati v naslednjem sprintu?
- ♦ Omejitev treh ur.

Scrum: graf izgorevanja

Graf izgorevanja (ang burn down chart):

- ❖ Javno objavljen graf, ki prikazuje preostalo delo iz zaloge sprinta.
- ❖ Vsakodnevno posodabljanje.
- ❖ Nudi enostaven vpogled v napredovanje sprinta.



Splošno o agilnih pristopih

- ❖ Agilni pristopi ne pomenijo “ad-hoc” programiranja!
- ❖ Natančno določajo standarde, ki se jih je potrebno držati
 - ♦ stil kodiranja,
 - ♦ način vključevanja novih funkcionalnosti in sprememb,
 - ♦ način testiranja (testiranje enot in integracije)
 - ♦ programski proces na višjem nivoju (zbiranje zahtev in izbiranje funkcionalnosti, izdaje...)

Uspešnost agilnih projektov

- ❖ Nanjo negativno vplivajo:
 - ♦ obsežnost (>20 razvijalcev),
 - ♦ porazdeljenost razvojnega tima,
 - ♦ upravljalška kultura podjetja,
 - ♦ prisilna uporaba agilnih postopkov,
 - ♦ kritične naloge, kjer napake niso dopustne.

Kritike agilnih pristopov

- ❖ Nezdostno načrtovanje.
- ❖ Nezdostno dokumentiranje.
- ❖ Nujni nadpovprečni programerji.
- ❖ Težavno finančno načrtovanje.
- ❖ Težavnejše zagotavljanje kakovosti.
- ❖ ...

Slabosti agilnih pristopov

- ❖ Manjkajoča obsežna dokumentacija.
- ❖ Sprotno testiranje je koristno, vendar vsega ni mogoče stestirati.
- ❖ Težavno sledenje hitro spremenljivih zahtev.
- ❖ Stalna predelava kode lahko vnese napake.

Slabosti agilnih pristopov

- ❖ **Manjkajoča obsežna dokumentacija**
- ❖ Programski produkt je potrebno vzdrževati
 - ♦ Nekateri programski produkti živijo desetletja!
 - ♦ Tudi po zaključku razvoja potrebujemo znanje o zgradbi sistema, ter poznavanje razlogov za sprejete odločitve.
 - ♦ Kako vzdrževati produkte v primeru fluktuacije? Razvoj v parih problematiko le nekoliko omili.

Slabosti agilnih pristopov

- ❖ **Stalna predelava kode lahko vnese napake**
- ❖ Predelave so namenjene izboljšanju delovanja programa in lepšanju kode.
- ❖ “Neoptimalnosti” kode so lahko posledica specifičnega načina delovanja programa zaradi specifičnih zahtev.
- ❖ Spreminjanja delovanja programa (spremembe algoritma) ni mogoče avtomatizirati, zato je podvrženo napakam!

Slabosti agilnih pristopov

- ❖ **Vsega ni mogoče stestirati.**
- ❖ Izgradnja avtomatskih testov je koristna, vendar pa so testi vedno omejeni.
 - ♦ Kdaj je testiranja zadosti?
 - ♦ Testiranje sekvenčnih programov je težavno, kaj pa v primeru paralelizmov?
 - ♦ Kako zagotoviti ustreznost varnostno kritičnih aplikacij?

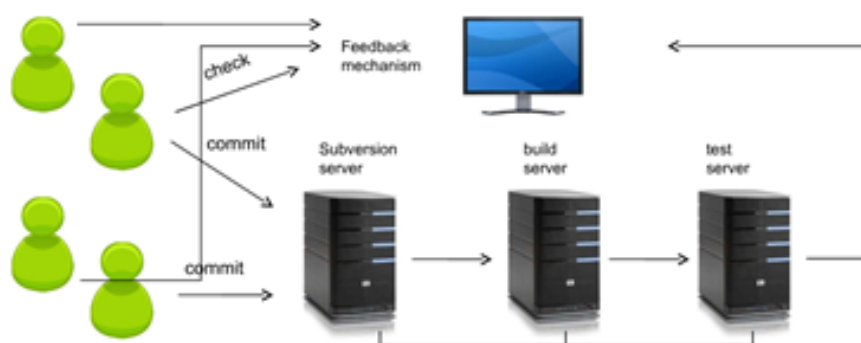
Slabosti agilnih pristopov

❖ Težavno sledenje hitro spremenljivih zahtev

- ♦ Kako upravljati z nekonsistentnimi zahtevami?
- ♦ Zakaj je bila narejena sprememba?
- ♦ Od kod zahteva za spremembo?
- ♦ Je bila sprememba utemeljena?

Neprestana (dez)integracija?

❖ (ang. Continuous Integration – CI)



❖ Uspešnost integracije je pogojena s

- ♦ pogostim osveževanjem izvirne kode,
- ♦ pogosto gradnjo produkta,
- ♦ takojšnjim testiranjem,
- ♦ takojšnjim odpravljanjem napak.

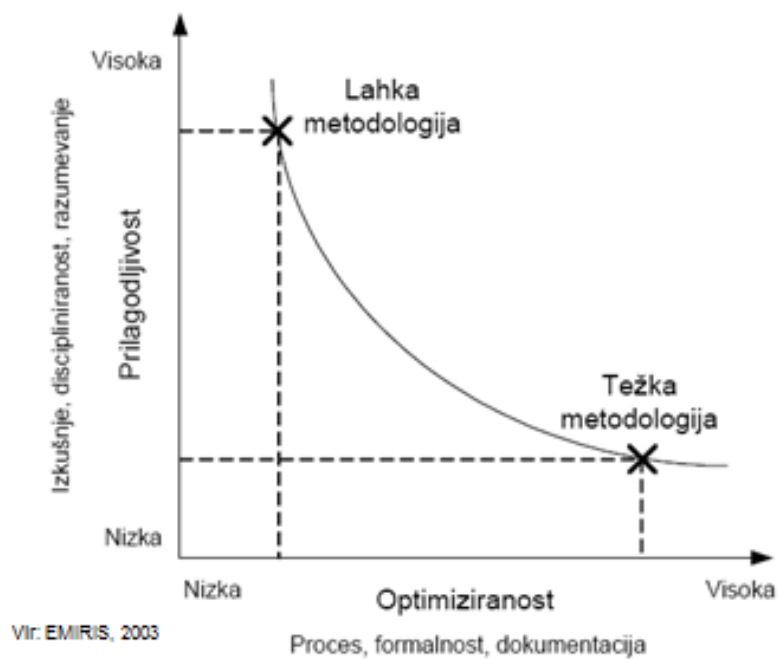
Agilni in formalni pristopi

- ❖ Agilni pristopi so glede na lastnosti projekta lahko optimalnejši od formalnih.
- ❖ Agilni pristopi ne morejo v popolnosti nadomestiti formalnih pristopov!
- ❖ Agilni in formalni pristopi se lahko dopolnjujejo.

Splošno o programskih procesih

- ❖ Procesi so bolj ali manj adaptivni oziroma prediktivni.
 - ♦ Agilni procesi so bolj adaptivni, vendar pa le s težavo predvidijo dogajanje v prihodnosti.
- ❖ Procesi so bolj ali manj optimizirani.
 - ♦ Formalni procesi so bolj optimizirani, vendar pa se težje spopadajo s spremembami.

Teža metodologije



4 Diagramske tehnike

Diagrami so delna grafična predstavitev modela sistema. Delna zato, ker samostojno ne omogočajo podrobnega opisa sistema, kljub temu pa v primerjavi z drugimi predstavitvami omogočajo enostaven in pregleden opis kompleksnih sistemov, s tem pa lažje razumevanje obstoječih in načrtovanih sistemov. Vsak model lahko vključuje več diagramov, poleg njih pa mora vsebovati še opise in razlage. Nekatere pomembne in široko uporabljane diagramske tehnike so obravnavane že pri predmetih drugega letnika, na primer entitetno relacijski diagrami (ERD), diagrami toka podatkov (DFD), diagrami entitetne zgodovine (ELH), pri tem predmetu pa se bomo dodatno spoznali z UML standardom. UML določa diagramske tehnike za opis procesov in objektov sistema, s poudarkom na objektno orientiranem pristopu. UML je celovit v smislu, da naj bi omogočal celovit grafični opis arhitekture programskih sistemov in torej lahko nadomesti tudi druge diagramske tehnike.

Pri uporabi diagramov je pomembno, da se držimo standardnih notacij, saj s tem dosežemo, da so diagrami bolj razumljivi in nedvoumni. Pri uporabi diagramov je pomembno, da imamo jasne cilje, kaj naj diagram sporoča. Torej se moramo omejiti na določen nivo podrobnosti; nivo modeliranja mora biti odvisen od potreb.

Zakaj diagrami?

- ❖ Omogočajo pregleden opis modelov.
- ❖ So glavno orodje pri modeliranju:
 - ♦ Omogočajo opis kompleksnih sistemov.
 - ♦ Nedvoumen opis prispeva k lažjemu in enotnemu razumevanju problema in rešitve.
 - ♦ Olajšajo načrtovanje željene strukture in obnašanja sistemov.

Diagrami in modeli

- ❖ Diagram je **delna** grafična predstavitev modela sistema.
- ❖ Model lahko vključuje več diagramov ter dokumentacijo.
 - ♦ Primer: model primera uporabe (ang. use-case) sestoji iz diagrama (use-case diagram) ter dokumenta, ki opisuje primer uporabe.
- ❖ V tej prezentaciji se omejujemo na diagramске tehnike in ne predstavljamo celotnih modelov.

Vrste diagramov (modelov)

❖ **Kontekstualni (ang. context)**

- ♦ Pogled sistema od zunaj. Opredelitev sistema v okolju.

❖ **Vedenjski (ang. behavioural)**

- ♦ Obnašanje sistema: primeri uporabe, delovanje procesov, tok podatkov, prehodi med stanji...

❖ **Strukturni (ang. structural)**

- ♦ Struktura sistema, komponent/objektov, podatkov...

Domene diagramov (modelov)

❖ **Domena aplikacije**

- ♦ Za analizo zahtev
- ♦ Okolje v katerem deluje sistem

❖ **Domena rešitve**

- ♦ Načrti sistema, načrti objektov
- ♦ Tehnologije za gradnjo sistema

Nivoji modelov

❖ Konceptualni model

- ♦ Predstavitev osnovnih lastnosti sistema, visokonivojski, osnovni/grobi pogled na sistem.

❖ Logični model

- ♦ Konceptualni model dopolnjuje s podrobnostmi, omejuje se na delovanje oziroma uporabo (procesi, podatki...)

❖ Fizični model

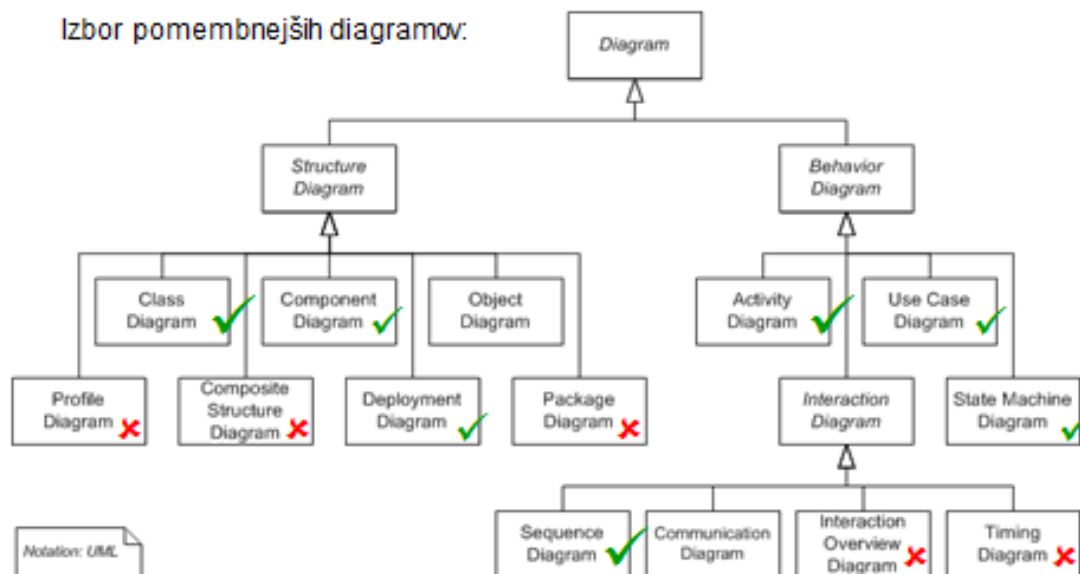
- ♦ Logičnemu modelu doda podrobnosti o dejanski fizični izvedbi (tipi, spremenljivke, funkcije...)

UML diagrami

- ❖ UML (The Unified Modeling Language) se uporablja za specifikacijo, vizualizacijo, modifikacijo, konstruiranje in dokumentiranje procesov in objektov razvijajočega sistema.
- ❖ Poudarek je na objektno orientiranem pristopu.
- ❖ Gre za standardiziran način predstavitev arhitekture sistema.
- ❖ 20% poznavanje UML zadošča za 80% primerov modeliranja sistemov.

UML diagrami

Izbor pomembnejših diagramov:



Vir: http://en.wikipedia.org/wiki/Unified_Modeling_Language

Strukturni diagrami (structure)

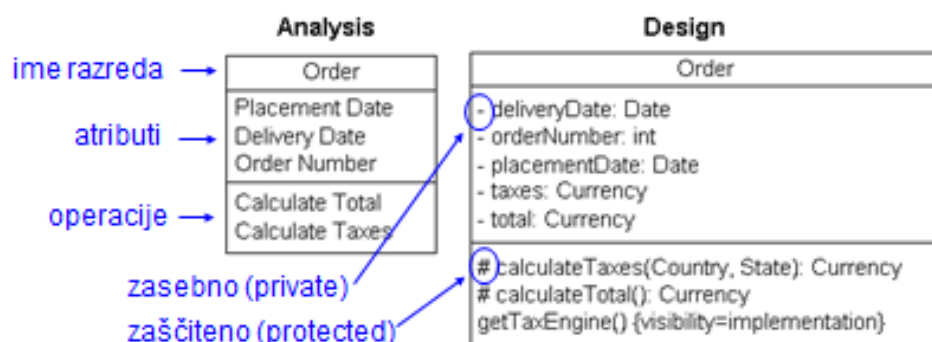
- ❖ **Class diagram** (razredni diagram) opisuje strukturo sistema s prikazom razredov (class), atributov in relacij med razredi.
- ❖ **Component diagram** (komponentni diagram) prikazuje kako je programski sistem razdeljen na komponente in kakšne so njihove medsebojne odvisnosti.
- ❖ **Composite structure diagram** prikazuje notranjo strukturo razreda in sodelovanja, ki jih ta struktura omogoča.
- ❖ **Deployment diagram** (diagram postavitve) prikazuje strojno opremo in okolje, v katerem deluje programska oprema, vključno s povezavami med komponentami.
- ❖ **Object diagram** (objektni diagram) prikazuje celotno ali delno strukturo objektov sistema v specifičnem trenutku.
- ❖ **Package diagram** (paketni diagram) opisuje razdelitev sistema na logične enote in relacije med njimi.

Vedenjski diagrami (behavior)

- ❖ **Activity diagram** (diagram aktivnosti) je diagram poteka izvajanja korakov sistema. Prikaže kontrolni tok pri delovanju sistema.
- ❖ **State machine diagram** (diagram stanj) prikazuje stanja in možno prehanje med stanji kot posledico dogodkov v sistemu.
- ❖ **Use case diagram** (diagram primera uporabe) prikazuje funkcionalnost sistema v smislu akterjev in njihovih ciljev pri uporabi sistema, hkrati pa tudi povezanost primerov uporabe.
- ❖ **Communication diagram** (komunikacijski diagram) prikazuje interakcije med objekti ali deli sistema, ki so prikazani v obliki strukture sistema.
- ❖ **Interaction overview diagram** je tip diagrama aktivnosti, kjer elementi predstavljajo diagrame interakcij.
- ❖ **Sequence diagram** (sekvenčni diagram) prikazuje komunikacijo med objekti v smislu sekvence sporočil.
- ❖ **Timing diagrams** (časovni diagram) je poseben tip diagrama interakcij, kjer so prikazane časovne omejitve pri delovanju sistema.

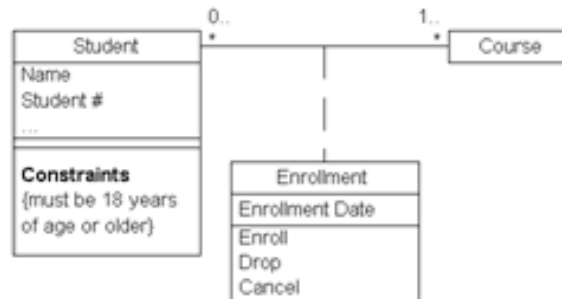
Class diagram (razredni diagram)

- ❖ Primerni za uporabo v različne namene – od razumevanja zahtev, do detajlnega modela izvedbe. Glede na potrebe se uporabljajo zelo različni stili.



Viri: <http://www.agilemodeling.com/style/classDiagram.htm>,
http://en.wikipedia.org/wiki/Class_diagram

Class diagram: association class



- ❖ Združevalni razredi (association classes) so povezani s črtkano črto.
- ❖ Povezav združevalnih razredov ne poimenujemo.
- ❖ Razred in črtkana povezovalna črta naj bosta postavljena centralno glede na združevana razreda.
- ❖ Primeren za načrtovanje podatkovnih modelov (lahko nadomešča ER diagram)

Class diagram: povezave



❖ Množičnost (ang. multiplicity)

- ♦ 0..1 nič ali ena
- ♦ 1 izključno ena
- ♦ 0..* nič ali več
- ♦ 1..* ena ali več
- ♦ n izključno n ($n > 1$)
- ♦ 0..n nič do n ($n > 1$)
- ♦ 1..n ena do n ($n > 1$)

Class diagram: relacije (1)

Združevanje (ang. association)



Agregacija (ang. aggregation), relacija tipa "ima", ang. "has a"

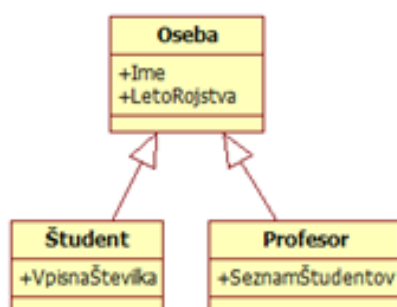


Kompozicija (ang. Composition), relacija tipa "vsebuje", ang. "owns a"

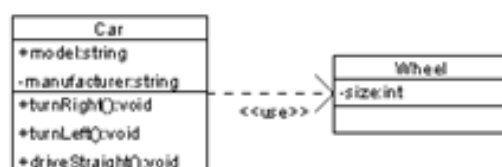


Class diagram: relacije (2)

generalizacija (ang. generalization)



odvisnost (ang. dependency)

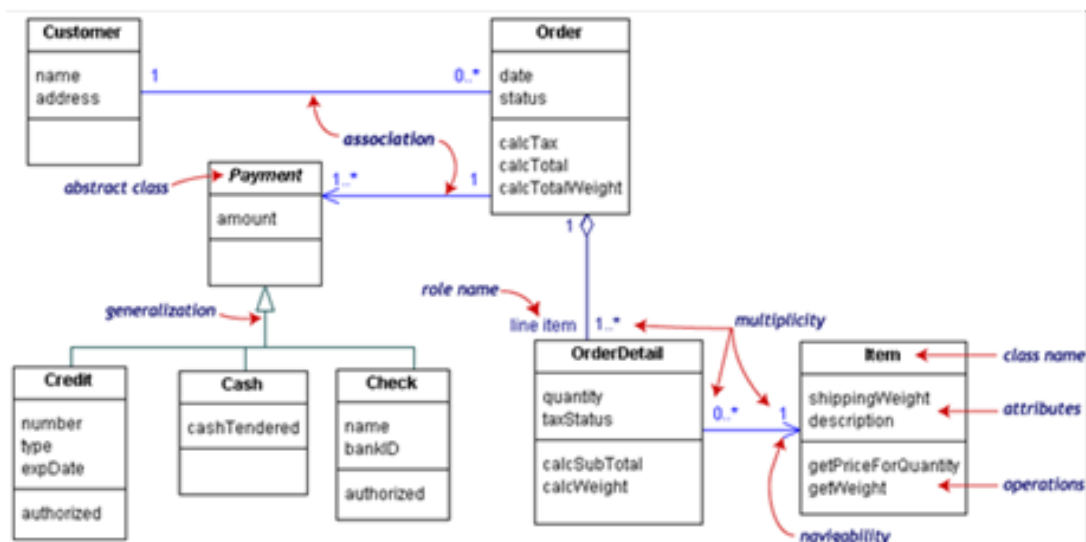


Class diagram: priporočila

- ❖ Imena razredov naj bodo nekrajšani samostalniki, imena operacij naj bodo glagoli, imena atributov samostalniki.
- ❖ Ne modeliraj funkcij ogrodja razreda (ang. scaffolding code), to je atributov in operacij za zagotovitev osnovne funkcionalnosti razreda (koda za relacije z ostalimi razredi, get in set funkcije...).
- ❖ Nikoli ne prikazuj razredov z le dvema sklopoma, saj to ne omogoča ločevanja med atributi in operacijami.
- ❖ Morebitne dodatne sklope razreda ustrezno označi (npr. requirements).
- ❖ Če razred ni prikazan v popolnosti, to ustrezno označi (npr. "...").
- ❖ Statične operacije/atributi naj bodo navedeni pred ostalimi operacijami /atributi.
- ❖ Izjeme je mogoče navesti v opisu operacije, npr.:
 - + findAllInstances(): Vector {exceptions=NetworkFailure, DatabaseError}

Vir: <http://www.agilemodeling.com/style/classDiagram.htm>

Class diagram: primer

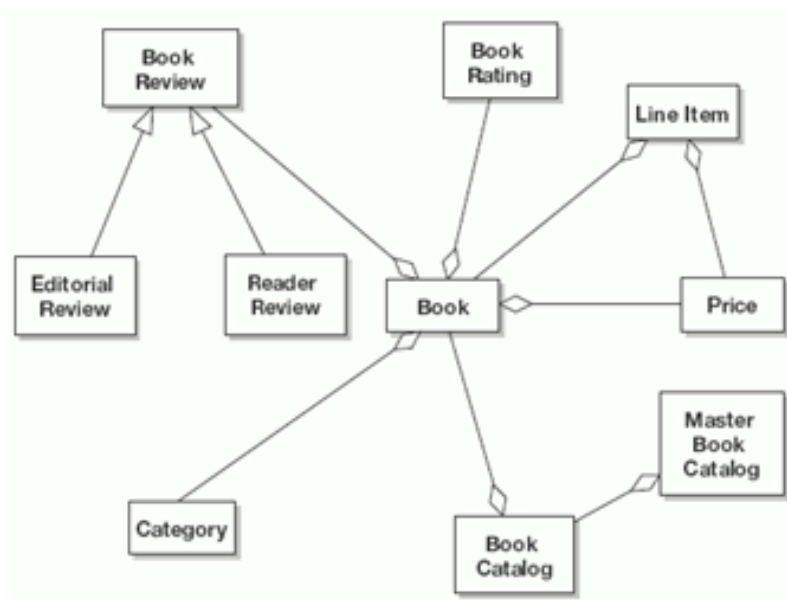


Vir: <http://edn.embarcadero.com/article/31863>

class diagram: domain model

- ❖ Domenski model je enostaven razredni diagram, namenjen hitremu začetnemu opisu sistema v fazi analize zahtev.
- ❖ Vsebuje le povezave tipa generalizacija in agregacija.
- ❖ Običajno ne vsebuje atributov in operacij, če že, pa le najpomembnejše attribute.

class diagram: domain model

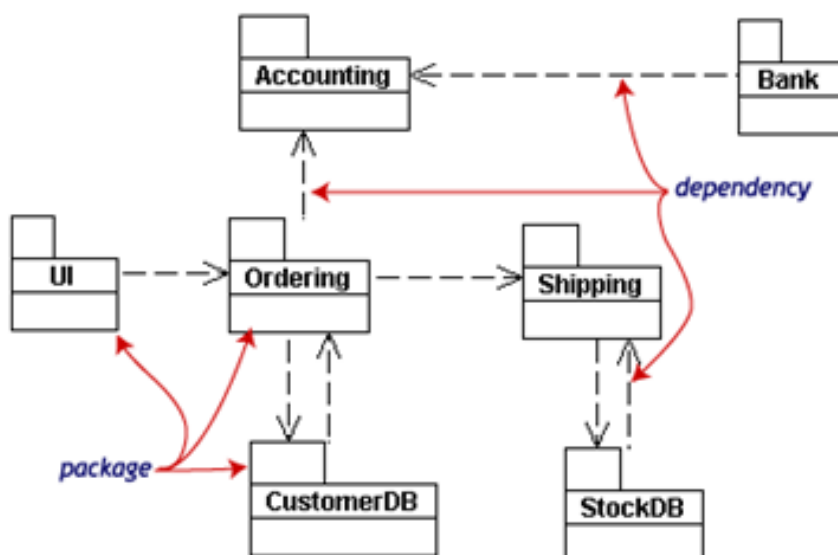


Vir: <http://consultingblogs.emc.com/vanessatong/archive/2008/06/25/where-do-i-start.aspx>

UML package diagram

- ❖ Za poenostavitev kompleksnih razrednih diagramov (class), lahko razrede združimo v **pakete**.
- ❖ Paket je zbirka logično povezanih UML elementov (tipično razredov).
- ❖ Pakete med seboj povezujejo njihove medsebojne odvisnosti. Paket je odvisen od drugega paketa, če sprememba v slednjem lahko zahteva spremembo v prvem paketu.

UML package diagram: primer

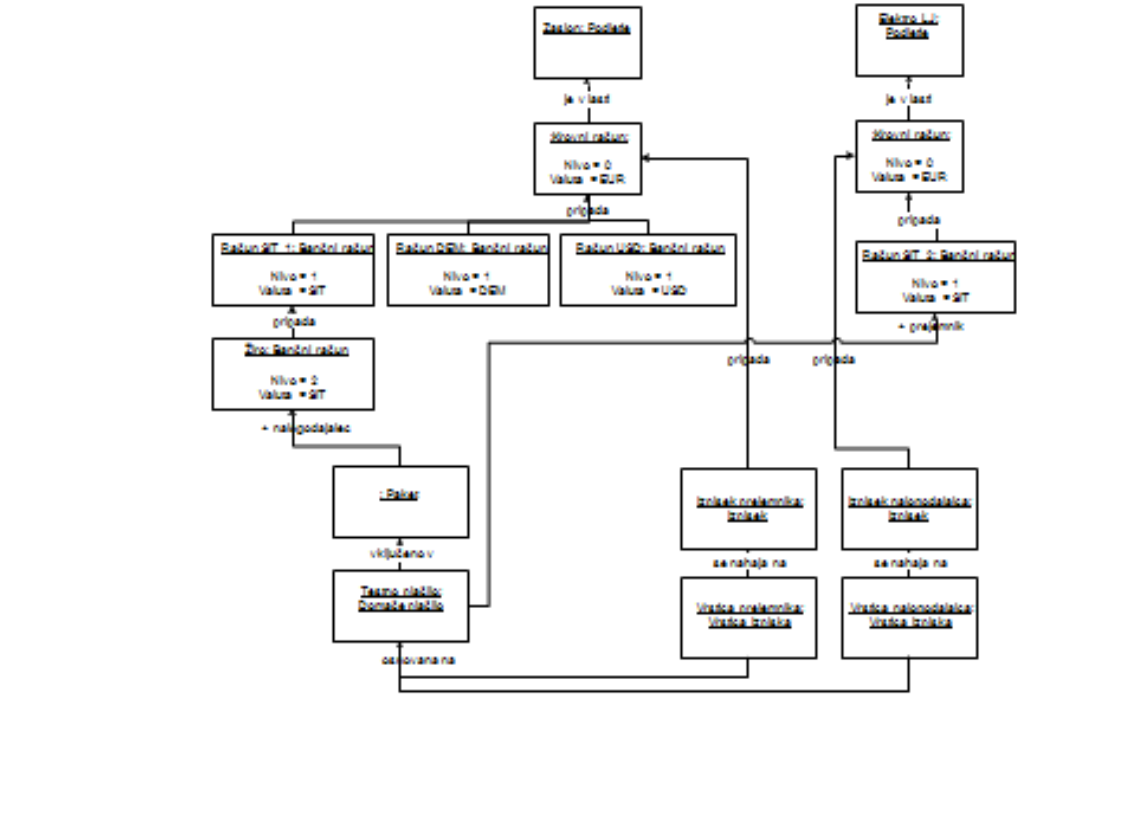


Vir: <http://edn.embarcadero.com/article/31863>

Object diagram (objektni diagram)

- ❖ Objektni diagrami so primerni za prikaz primerov objektov "realnega sveta" in njihovih medsebojnih relacij.
- ❖ Potrebni so v primeru, če je razredni diagram preveč abstrakten. Objektni diagram tako služi kot primer razredov v nekem trenutku izvajanja programa.

Object diagram: primer

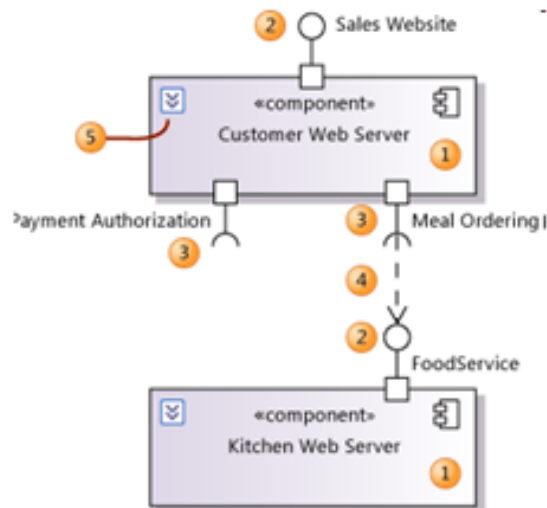


Component diagram

- ❖ Komponentni diagram prikazuje komponente in povezave med njimi.
- ❖ Komponenta je modularna enota z jasnimi vmesniki in pripadajočimi vrati (ang. port).
- ❖ Nekatera vrata komponenta nudi, druge pa od sistema zahteva za svoje delovanje.
- ❖ Interne lastnosti komponente so skrite in nedostopne.

Component diagram: elementi

1. Komponenta
2. Vmesnik (port) ki ga komponenta nudi
3. Vmesnik (port) ki ga komponenta potrebuje za delovanje
4. Povezava med komponentami



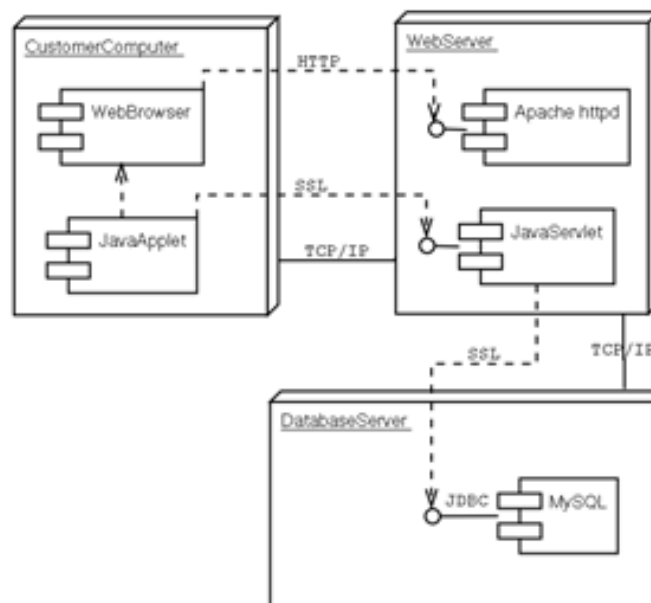
Dodatna literatura:

http://www.disi.unige.it/person/ReggioG/ISII04WWW/COMPONENTDIAGRAM_MOD.ppt

Deployment diagram

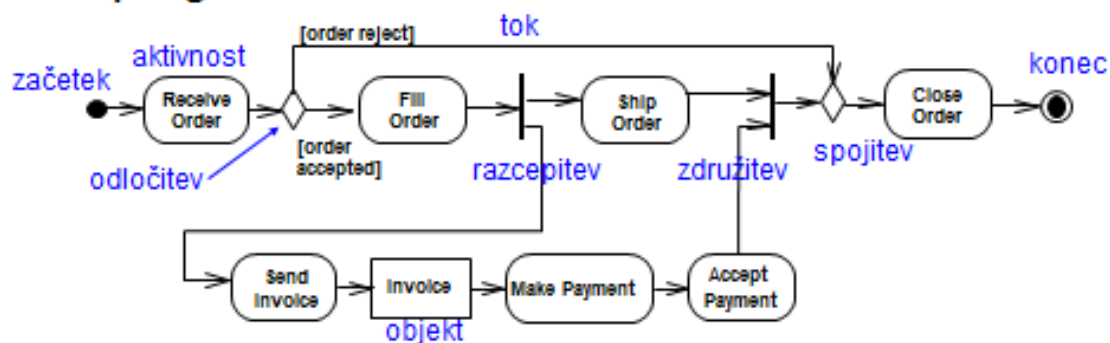
- ❖ Diagram postavitve prikazuje fizične relacije med strojno in programsko opremo sistema.
- ❖ Elementi strojne opreme:
 - ♦ Računalniki (strežniki, klienti)
 - ♦ Vgrajene naprave (ang. embedded)
 - ♦ Naprave (senzorji, periferija)
- ❖ Diagram postavitve prikazuje strojno opremo, kjer deluje programska oprema.

Deployment diagram: primer



Activity diagram

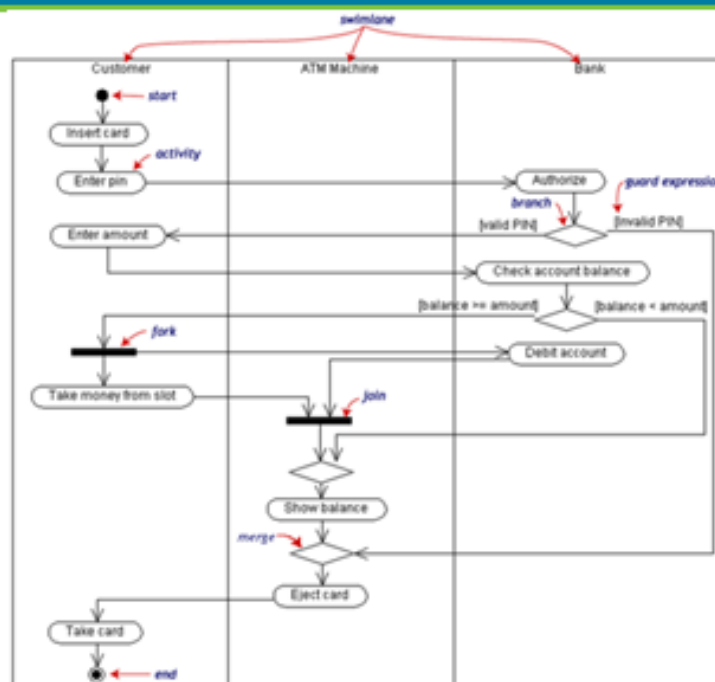
- ❖ Diagram aktivnosti je UML varianta diagrama poteka.
- ❖ Modelira kontrolni tok pri izvajanju programa.



Dodatna literatura:

<http://www.disi.unige.it/person/ReggioG/IS1104WWW/ActivityDiagrams.ppt>

Activity diagram: primer 2

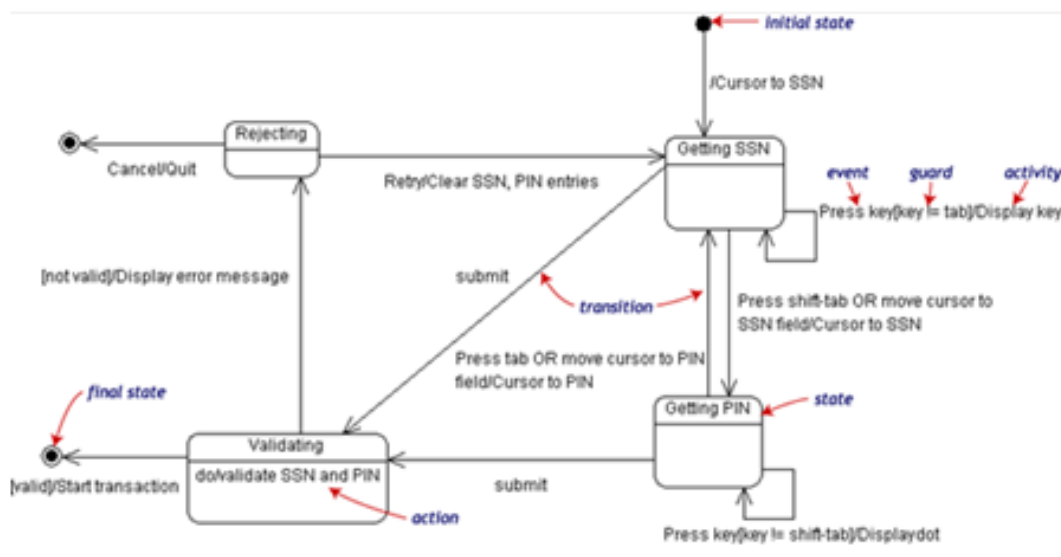


vir: <http://edn.embarcadero.com/article/31863>

State machine diagram

- ❖ Za objekte je značilno določeno obnašanje in stanja.
- ❖ Stanje objekta je odvisno od predhodnega dogajanja in se spreminja na podlagi (zunanjih) dogodkov (ang. events). Za vsako stanje je lahko značilna določena aktivnost, ki se izvede ob prehodu v to stanje.
- ❖ Diagram prehajanja stanj prikazuje možna stanja objekta, možne prehode med stanji, dogodke, ki sprožijo spremembe stanja in aktivnosti, ki se zgodijo v določenih stanjih.

State machine diagram: primer

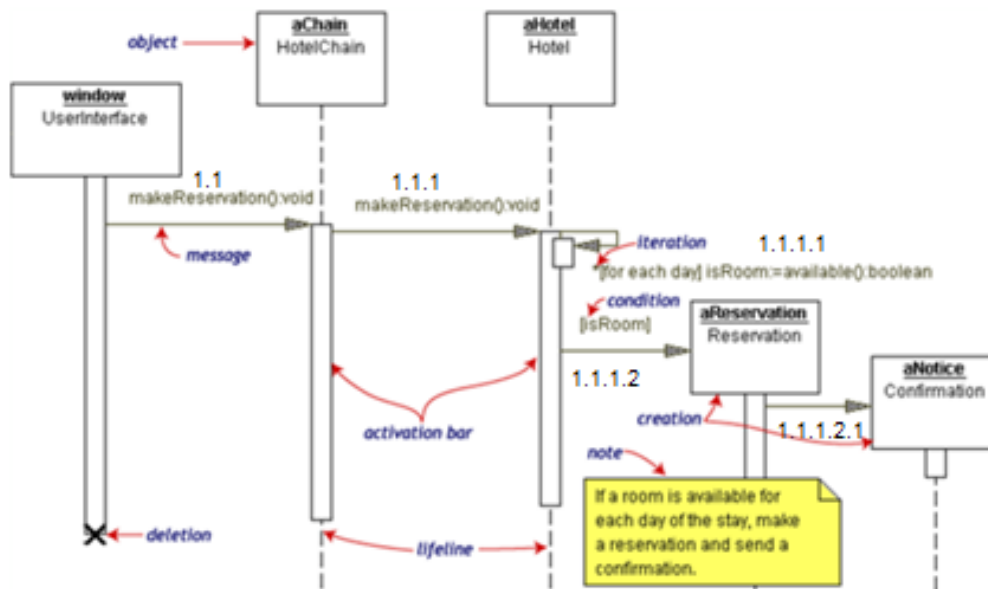


Vir: <http://edn.embarcadero.com/article/31863>

Sequence diagram

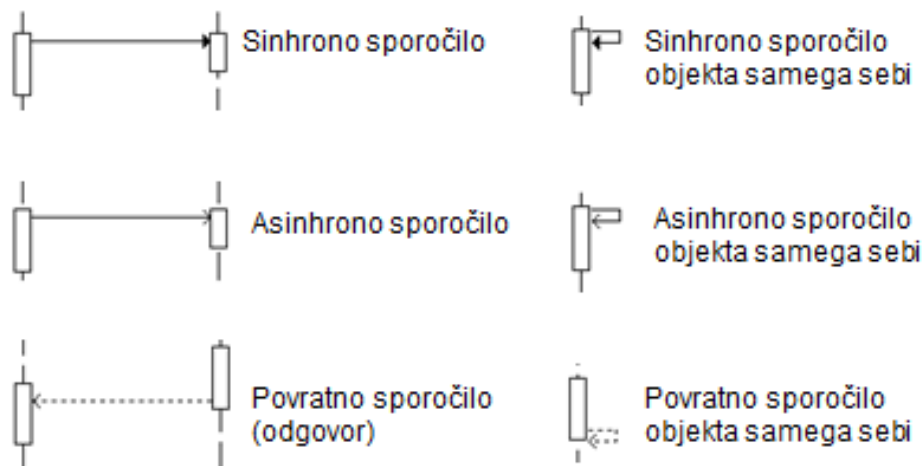
- ❖ Sekvenčni diagram prikazuje komunikacijo med objekti v smislu zaporedja sporočil.
- ❖ Vsakemu objektu pripada svoj pas, ki ustreza časovni osi v času obstoja objekta, na njem pa je prikazan čas aktivnosti objekta.
- ❖ Primerno tako za konceptualne kot tudi logične modele, na različnih nivojih abstrakcije.

Sequence diagram: primer



Vir: <http://edn.embarcadero.com/article/31863>

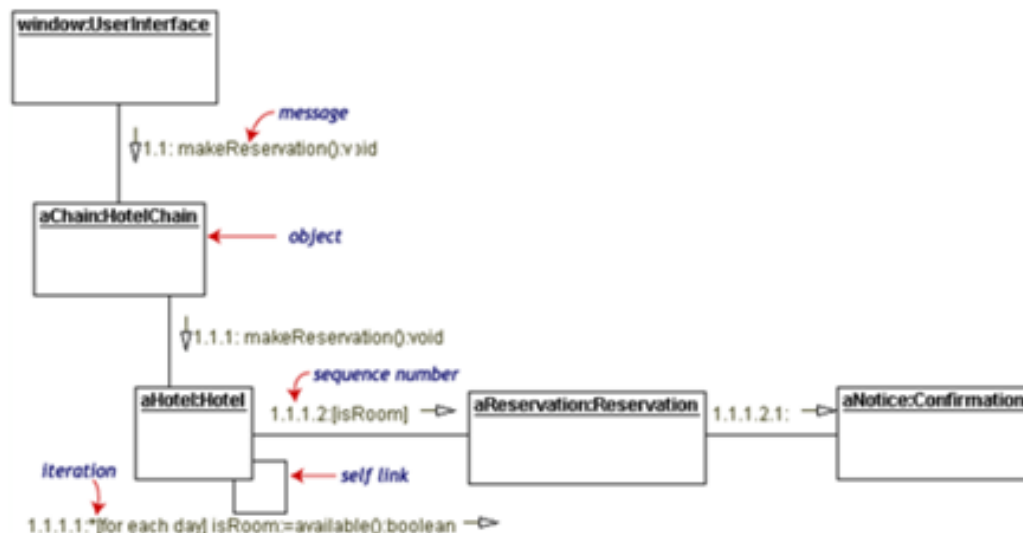
Sequence diagram: sporočila



Communication diagram

- ❖ Komunikacijski diagram - UML 1.x ga imenuje collaboration diagram.
- ❖ Je enakovreden sekvenčnemu diagramu, podaja iste informacije.
- ❖ Bolj poudari povezanost med objekti, nekoliko manj od sekvenčnega diagrama pa zaporedje sporočil.

Communication diagram: primer



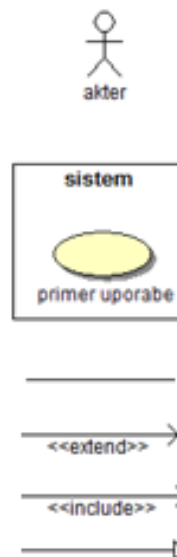
Vir: <http://edn.embarcadero.com/article/31863>

Use case diagram

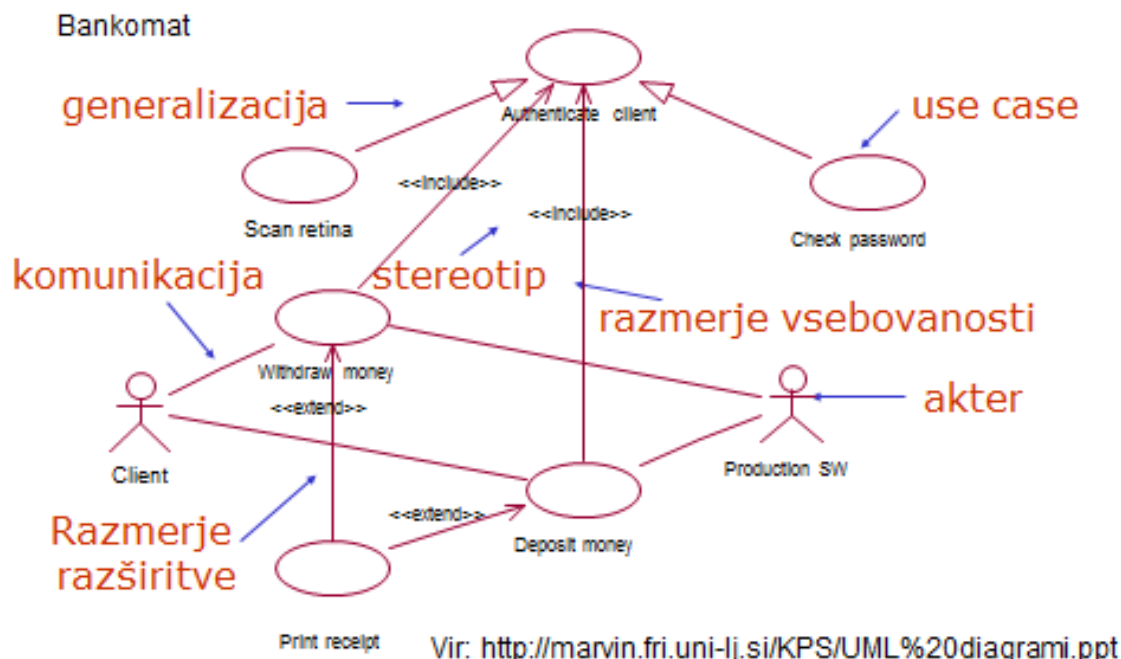
- ❖ Diagram primera uporabe podaja zunanji pogled na sistem, namenjen opisu uporabniških zahtev (kaj naj bi delal sistem).
- ❖ Opisuje interakcijo sistema z uporabniki.
- ❖ Določa meje sistema (kaj je vključeno v sistem in kaj ne)
- ❖ Ni korelacije s strukturo sistema in implementacijo.

Use case diagram: simboli

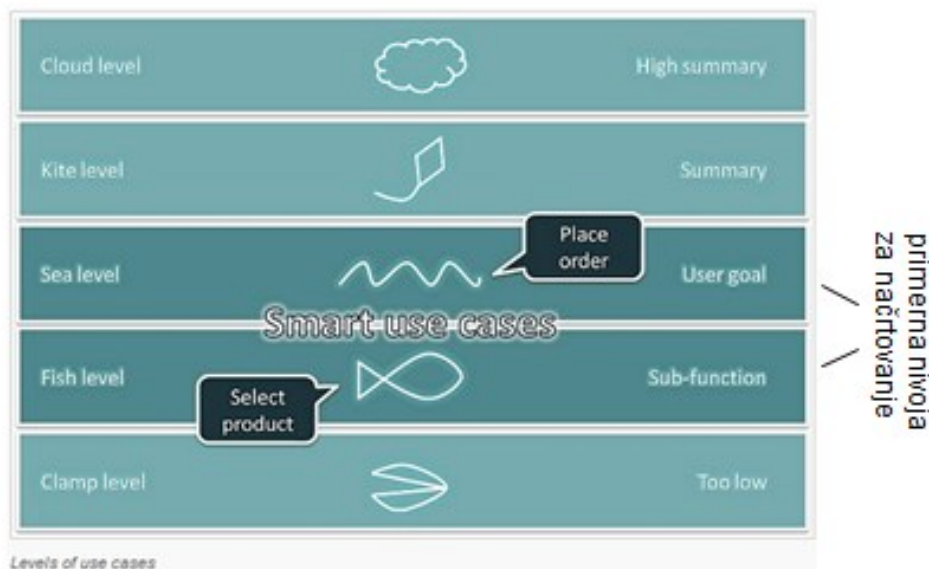
- ❖ Akter je del okolja, izvaja primer uporabe in ima od njega neko korist (posameznik, drug sistem, druga naprava).
- ❖ Meje sistema označimo s pravokotnikom.
- ❖ Primer uporabe (use case) je značilna skupina scenarijev, ki so potrebni, da uporabnik doseže svoj cilj.
- ❖ Scenarij (aktivnost) je zaporedje korakov, ki pomenijo interakcijo uporabnika s sistemom
- ❖ Povezave (komunikacija, razširitev, vključenost, generalizacija)



Use case diagram: primer



Use case diagram: nivoji



Vir: <http://www.accelerateddeliveryplatform.com/SmartUseCaseLevels.ashx>

Use case diagram: priporočila

- ❖ Primeri uporabe naj odražajo aktivnosti za doseganje ciljev uporabnika in ne aktivnosti za delo s sistemom.
- ❖ Glavni akter naj se nahaja zgoraj levo.
- ❖ Akter ne uporablja drugega akterja.
- ❖ Primer uporabe ne uporablja drugega primera uporabe, lahko pa ga vključuje, razširja, ali generalizira).
- ❖ Diagram vsebuje najmanj enega akterja in najmanj en primer uporabe.

UML diagrami

- ❖ Uporabljaljaj diagrame, ki so pomembni pri reševanju problema – diagram naj ne bo sam sebi namen.
- ❖ Ne pretiravaj s količino in podrobnostmi!
- ❖ Standardna notacija olajša uporabo diagramov, nekaj svobode pri sintaksi je dovoljene.

Diagramске tehnike: zaključek

- ❖ Modeliranje je potrebno, obseg modeliranja pa odvisen od potreb!
- ❖ UML je uveljavljen standard, na nekaterih področjih pa so bolj v uporabi druge tehnike:
 - ♦ načrtovanje podatkovnih baz: ER
 - ♦ načrtovanje toka podatkov v sistemu: DFD

5 Analiza

in definiranje zahtev

Analiza in definiranje zahtev je faza programskega procesa v kateri je potrebno natančno opredeliti želeno funkcionalnost programskega produkta. Pri tem je funkcionalnost potrebno definirati z uporabniškega stališča, torej stališča uporabe programskega sistema, z opisom njegove funkcionalnosti in potrebnih navzven merljivih lastnosti. Gre za funkcijske in nefunkcijske zahteve.

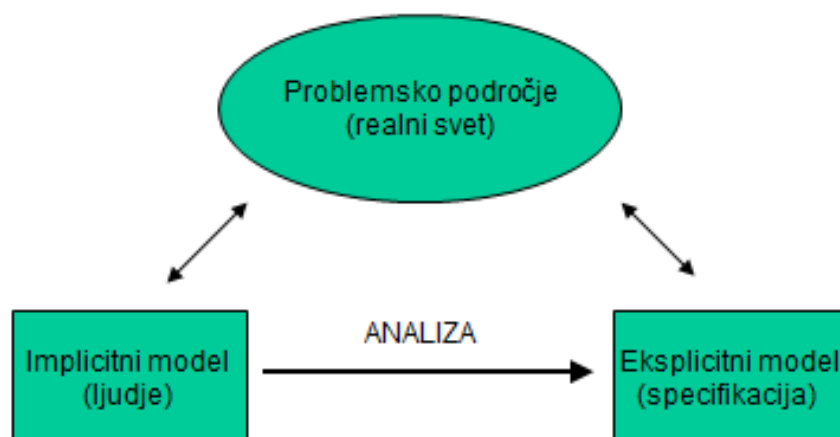
Definiranje zahtev je naloga, ki jo v programskem procesu opravi sistemski analitik, Ta naloga je težavna iz več razlogov med katerimi je eden glavnih težavnost zadostnega razumevanja problemskega področja, s katerim se v vsakdanjem življenju morda ne srečujemo. Ključna je pridobitev ustreznih znanj za razumevanje problema, pri čemer se moramo zavedati razkoraka med implicitnimi modeli uporabnikov oziroma poznavalcev ter eksplicitnimi modeli, torej predstavitev, ki so na voljo sistemskemu analitiku. Sprejeto dejstvo je namreč, da se vsega implicitnega znanja ne da eksplicitno zapisati, prav manjkajoči del pa je lahko ključen za pravilno razumevanje problema in s tem ustrezno opredelitev zahtev programskega produkta.

Definiranje zahtev je eden najpomembnejših korakov razvoja programskega produkta, saj določa ves nadaljnji razvoj programskega produkta, odpravljanje morebitnih napak v kasnejših fazah razvoja pa je težavno in povezano z visokimi stroški. To je tudi razlog, da je specifikacijo zahtev, ki je rezultat omenjene faze, koristno temeljito preveriti/recenzirati.

Analiza in definiranje zahtev

- ❖ Uporabljan je tudi izraz **sistemska analiza** - izvaja jo **sistemski analitik**.
- ❖ Na osnovi informacij o realnem problemu, ki so praviloma slabo definirane, nejasne in včasih nasprotujoče, je potrebno izluščiti jasne in eksplicitne zahteve.
- ❖ Izziv je definirati tak sistem, ki
 - ♦ ustreza uporabnikovim potrebam in
 - ♦ ustreza uporabnikovim pričakovanjem.

Namen analize in definiranja zahtev



Problemi analize

- ❖ Implicitni modeli niso ubesedeni.
- ❖ Implicitni modeli se s časom spreminjajo.
- ❖ Naročnik in sistemski analitik ne govorita istega "jezika".
- ❖ Vsega implicitnega znanja se ne da formalno zapisati.

Analiza in definiranje zahtev

- ❖ Rezultat je specifikacija zahtev - popoln opis obnašanja sistema:
 - ♦ Funkcijske zahteve: primeri uporabe (ang. use cases), za vse interakcije **uporabnika** s sistemom.
 - ♦ Ostale, nefunkcijske zahteve: zahteve, ki omejujejo načrtovanje ali izvedbo sistema, npr. zahtevana prepustnost sistema, standardi kakovosti...
- ❖ Napačna, nepopolna nekonsistentna ali dvoumna specifikacija zahtev je pogosto vzrok za neuspeh projektov in spore.

Nefunkcijske zahteve

❖ Nefunkcijske zahteve:

- ♦ Obratovalne lastnosti
 - varnost, uporabljivost...
- ♦ Evolucijske lastnosti
 - Možnost testiranja, možnost vzdrževanja, razširljivost, nadgradljivost...

❖ Za primere nefunkcijskih zahtev glej:

http://en.wikipedia.org/wiki/Non-functional_requirement

Uporaba specifikacije zahtev

- ## ❖ Namen dokumenta "specifikacije zahtev" (ang. Software Requirements Rpecification - SRS):
- ♦ seznaniti se s potrebami in omejitvami uporabnika,
 - ♦ uskladiti razumevanje problema in rešitve z uporabnikom,
 - ♦ določiti izhodišča za načrtovanje sistema,
 - ♦ določiti izhodišča za validacijo sistema.

Postopek

1. Seznanjanje s problemom in zbiranje zahtev
2. Analiza, modeliranje in usklajevanje
3. Recenzija (validacija zahtev)
4. Upravljanje z zahtevami

Seznanjanje in zbiranje zahtev

❖ Morda se zdi enostavno:

- ♦ Vprašaj stranko, uporabnike in ostale:
 - Kateri so cilji oziroma kaj je namen sistema?
 - Kaj želimo s tem doseči?
 - Kako naj sistem zadovoljuje poslovne potrebe?
 - Kako bo sistem dnevno uporabljan?

❖ Pa vendar je težavno!

- ♦ Problem obsega sistema
- ♦ Problemi razumevanja zahtev
- ♦ Problemi spremenljivosti zahtev

Problem obsega sistema

- ❖ Meje sisteme morda niso jasne?
 - ♦ Kdo bo neposredno uporabljal sistem?
 - ♦ Kateri drugi sistemi so povezani s sistemom?
 - ♦ Katera funkcionalnost naj bo del sistema?
- ❖ Stranka lahko navaja nepotrebne tehnične detajle, ki lahko zmedejo razumevanje obsega sistema (npr. brez razloga navajan operacijski sistem, jezik, strojna oprema)

Problemi razumevanja zahtev

- ❖ Stranke/uporabniki morda niso prepričani kaj točno potrebujejo:
 - ♦ "Naredite tako, da nam bo v pomoč"
 - ♦ "Poskusite izboljšati sistem"
- ❖ Morda slabo poznajo zmogljivost in omejitve svoje računalniške opreme ali procesa.
 - ♦ Nimajo popolnega razumevanja problemskega področja.
 - ♦ Ne znajo opredeliti zahtev sistemskemu analitiku.
 - ♦ Izpustijo informacije, ki naj bi bile "očitne".
 - ♦ Navajajo zahteve, ki nasprotujejo zahtevam drugih.
 - ♦ Navajajo nejasne zahteve in zahteve, ki jih ni mogoče preveriti.
- ❖ Sistemski analitik pogosto ni poznavalec problemskega področja

Problemi spremenljivosti zahtev

- ❖ Zahteve se s časom spreminjajo
- ❖ Spremembe so neizogibne zaradi naslednjih dejavnikov:
 - ♦ Spremembe v strankini organizaciji (npr. novi produkti, oddelki...)
 - ♦ Spremembe obsega delovanja (npr. dnevno število transakcij, število uporabnikov, povečana propustnost povezav...).
 - ♦ Zunanje spremembe
 - Spremembe zakonov (npr. obdavčevanje)
 - Spremembe mednarodnih standardov (npr. MPEG)
 - ♦ Stranke dobijo nove ideje ko spoznajo zmožnosti novega sistema.

Seznanjanje in zbiranje zahtev

- ❖ Študij literature
- ❖ Nasveti strokovnjakov
- ❖ Spraševanje (ankete)
- ❖ Izpeljave iz obstoječega sistema
- ❖ Sinteze iz lastnosti okolice
- ❖ Izdelava prototipov

Študij literature

- ♦ Knjige s problemskega področja
- ♦ Opisi procesov
- ♦ Študije izboljšav procesov
- ♦ Razvojna dokumentacija
- ♦ Evidenca težav in prijavljenih napak
- ♦ Predlogi

Nasveti strokovnjakov

- ❖ Prisluhni stokovnjakom z danega področja.
- ❖ Strokovnjaki naj ne bodo nujno le tisti, ki so povezani z naročnikom.
- ❖ Strokovnjaki
 - ♦ Poznajo zakonitosti problemskega področja.
 - ♦ Pomagajo določiti obseg sistema.
 - ♦ Lahko usmerijo systemskega analitika k študiji alternativnih rešitev.

Spraševanje

- ❖ Sistemski analitik sprašuje stranke in/ali uporabnike
 - ♦ Kaj pričakujejo od novega sistema.
 - ♦ Upamo, da znajo obiti svoje omejitve in predsodke.
 - ♦ Spraševanje lahko poteka v obliki intervjuja, brainstorminga ali vprašalnika.
- ❖ Lahko gre za neposredno komunikacijo s stranko ali za anketiranje uporabnikov.

Izpeljave iz obstoječega sistema

- ❖ Analizo začnemo na osnovi obstoječega sistema.
 - ♦ Analiza lahko poteka na osnovi sistema, ki ga želimo nadomestiti.
 - ♦ Analiza lahko poteka na osnovi sistema v neki drugi organizaciji.
 - ♦ Analiza lahko poteka na osnovi opisa sistema v literaturi.

Sinteze iz lastnosti okolice

- ❖ Za uspešno delovanje programske opreme je potrebno upoštevati zakonitosti okolja.
- ❖ Gre za t.i. procesno analizo, normativno analizo ali decizijsko analizo.

Izdelava prototipov

- ❖ V uporabi predvsem takrat, ko brez prototipa ne znamo jasno določiti zahtev.
- ❖ Prototip je delujoč model dela sistema ali celotnega sistema.
- ❖ Tipično se model od končnega sistema razlikuje zaradi neizpolnjevanja nefunkcijskih zahtev.

Atomske zahteve

❖ Zahteve so atomske, če:

- ♦ natančno določajo potrebo ne da bi jih bilo treba deliti na več zahtev
- ♦ so merljive,
- ♦ so preverljive (testabilne),
- ♦ so sledljive.

Analiza, modeliranje in usklajevanje

❖ Koraki:

- ♦ Analiza
 - Analiza podanih zahtev v smislu konsistentnosti, popolnosti in realnosti.
- ♦ Modeliranje
 - Izgradnja eksplicitnega modela zahtev, v obliki naravnega jezika, diagramov in formalnih notacij (npr. matematično).
- ♦ Usklajevanje
 - Dogovarjanje o obsegu rešitve, konfliktnih zahtevah, prioritetah, rizikih...

❖ Običajno se koraki izmenjujejo iterativno.

Analiza

- ❖ **Kategorizacija zahtev**
 - ♦ Organizacija zahtev v povezane podmnožice
- ❖ **Ugotavljanje povezanosti med zahtevami**
- ❖ **Ugotavljanje**
 - ♦ Konsistentnosti zahtev
 - ♦ Manjkajočih podatkov
 - ♦ Dvoumnosti zahtev
- ❖ **Določanje prioritete glede na potrebe stranke/uporabnikov.**
 - ♦ Lahko služi kot osnova za inkrementalni razvoj

Vprašanja pri analizi zahtev

- ❖ Ali je vsaka od zahtev skladna s cilji novega sistema?
- ❖ Ali so bile vse zahteve določene na ustreznem nivoju abstrakcije?
 - ♦ Preveč tehničnih detajlov omejuje nadaljnji razvoj.
 - ♦ Premalo detajlov pa pušča zahteve nejasne.
- ❖ Ali je vsaka zahteva resnično potrebna za dosego ciljev sistema?
- ❖ Ali je vsaka zahteva jasna in nedvoumna?
- ❖ Ali ima vsaka zahteva jasen izvor (kdo to zahteva)?
- ❖ Ali si katere od zahtev nasprotujejo?
- ❖ Ali je vsaka od zahtev tehnično izvedljiva v okviru danega okolja?
- ❖ Ali bo vsako od zahtev mogoče preveriti, ko bo programski produkt razvit?

Modeliranje

❖ Zahteve je mogoče predstaviti z različnimi vrstami modelov:

- ♦ Naravni jeziki
 - Enostavno razumevanje.
 - Velike možnosti za dvoumnosti in nejasnosti.
- ♦ Pol-formalni zapisi
 - Bolj precizni od naravnih jezikov.
 - Enostavni za razumevanje.
 - Ne zahtevajo uporabe natančno definiranih notacij.
- ♦ Formalni zapisi
 - Npr. matematični zapisi.
 - Zelo ekzakti, nedvoumni.

Modeliranje in modeli

❖ Modeli za specifikacijo zahtev morajo nuditi vpogled v naslednje tri vidike predlaganega sistema:

- ♦ Podatki – podatkovni model
 - Kakšne podatki se bodo uporabljali in kako so medsebojno povezani.
- ♦ Procesi – procesni model
 - Iz katerih procesov sestoji sistem in kakšna je njihova funkcija.
- ♦ Izvajanje – kontrolni model
 - Zaporedje izvajanja procesov, odločitve, spojitve, razcepitve, združitve kontrolnih tokov.

Modeliranje zahtev

❖ Strukturna analiza

- ♦ Seznam dogodkov
- ♦ Kontekstualni diagram
- ♦ Diagrami podatkovnega toka (DFD) + DFD specifikacije
- ♦ Entitetno relacijski diagrami (ERD)
- ♦ Diagram poteka (flowchart)

❖ Uporaba UML (OOA)

- ♦ Primeri uporabe (diagram + opis)
- ♦ Domenski diagram (class diagram)
- ♦ Sekvenčni diagram (sequence), komunikacijski diagram (communication)
- ♦ Diagram aktivnosti (activity), diagram prehajanja stanj (state)

Objektno orientirana analiza

❖ OOA (ang. Object Oriented Analysis)

❖ Izrazitejša ločevanje med analizo in načrtovanjem kot pri postopkih strukturne analize.

- ♦ Meja med analizo in načrtovanjem še veno ni ostra.

❖ Izhodišče je uporaba objektov in predstavitev z objektnimi modeli – tipično UML.

Primeri uporabe (use cases)

- ❖ Primer uporabe opisuje interakcijo med akterji in sistemom za nek scenarij uporabe sistema:
 - ♦ Opisuje funkcijo, ki jo sistem nudi uporabniku.
 - ♦ Opisuje celovito interakcijo uporabnika s sistemom, torej obsežne procese, od začetka do konca uporabe.
 - ♦ Z njim uporabnik doseže nek cilj.
 - ♦ Za definiranje primerov uporabe je potrebno razumevanje zahtev.
 - ♦ Primer uporabe določimo za vsak dogodek sistema.
 - ♦ Podrobno jih opišemo z opisom primera uporabe in ilustriramo z diagramom primera uporabe (use case diagram).

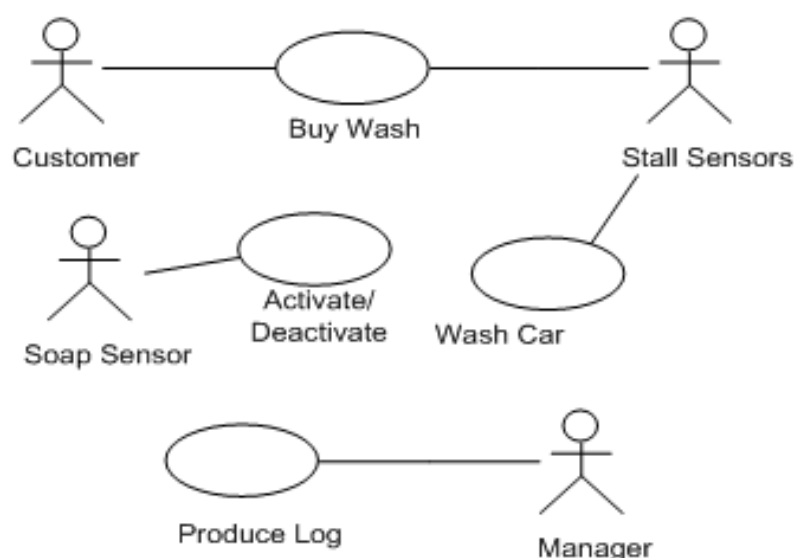
Modeli primera uporabe

- ❖ Model primera uporabe sestoji iz
 - ♦ Diagrama primera uporabe
 - ♦ Opisa primera uporabe (za vsak primer uporabe iz diagrama) .
- ❖ Modeli primera uporabe ne omogočajo popolnega opisa sistema! Dopolnjujejo jih dodatne zahteve:
 - ♦ Podatkovne in nefunkcijske zahteve.
 - ♦ Zahteve fizičnega nivoja.

Primer scenarija (avtopralnica)

Michael drives to the kiosk outside the carwash and attempts to insert five dollars. The kiosk accepts the money and displays the message "You have paid enough for a complete wash. Insert another dollar for a deluxe wash." Michael presses the button for an express wash. The kiosk refunds one dollar. The carwash queries its stall sensors. The stall sensors report that no car is detected, so the kiosk displays the message "Please drive forward into the wash stall."

Diagrama primera uporabe – primer



Primer diagrama uporabe za avtopralnico.

Opis primera uporabe

- ❖ Opis primera uporabe (ang. use case description) je specifikacija interakcije med produktom in akterji za primer uporabe.
- ❖ Podaja informacije o:
 - ♦ aktivnostih akterjev in produkta,
 - ♦ vseh možnih potekih interakcije (kompleksen potek aktivnosti).

Vsebina opisa primera uporabe

1. Naziv primera uporabe
2. Akterji
 - Navedba vseh udeležениh akterjev (oseb, zunanjih sistemov...)
3. Zahteve
 - Zahteve povezane s primerom uporabe (tudi vir zahteve).
4. Sprožilec (trigger)
 - Vzrok za začetek aktivnosti.
5. Osnovni potek
 - Tok aktivnosti za običajno uspešno interakcijo – seznam korakov.
 - Začne se ob trigerju in konča ob uspešnem zaključku.
6. Stanje po zaključku
 - Pričakovan rezultat primera uporabe.

Vsebina opisa primera uporabe

7. Alternativni poteki

- Opisujejo ostale možne uspešne in neuspešne poteke, kot tok aktivnosti podan s seznamom korakov.
- Začejo in končajo se lahko kjer koli, glede na druge poteke.
- Alternativni poteki lahko izhajajo iz drugih alternativnih potekov.

8. Razno

- Komentarji primera uporabe in povezave z drugimi primeri uporabe

Primer iz avtopralnice

- ❖ Naziv: Activate/Deactivate
- ❖ Akterji: Soap sensor
- ❖ Zahteve:
Customers — Need their cars washed with soap, and want a complete wash
Operations — Want the carwash to operate without constant attention
- ❖ Trigger: One minute has passed since the last time the soap sensor was checked.
- ❖ Osnovni potek:
 1. The carwash queries the soap sensor.
 2. The soap sensor indicates that there is soap.
 3. If the carwash is active, it continues its operation and the use case ends.
- ❖ Stanje po zaključku:
- ❖ The carwash is active if and only if the soap sensor indicates that there is soap.
- ❖ No wash currently in progress is interrupted.

Primer - nadaljevanje

- ❖ Alternativni poteki:
- ❖ 1a: The carwash is unable to query the soap sensor:
 - ❖ 1a1. The controller logs the problem and the use case ends.
- ❖ 2a: The soap sensor indicates that there is no more soap:
 - ❖ 2a1. If the carwash is inactive, the use case ends.
 - ❖ 2a2. If the carwash is active, the controller displays an out-of-order message and becomes inactive; the use case ends.
- ❖ 2b: The soap sensor does not respond:
 - ❖ 2b1. The controller logs the problem.
 - ❖ 2b2. If the carwash is inactive, the use case ends.
 - ❖ 2b3. If the carwash is active, the controller displays an out-of-order message and becomes inactive; the use case ends.
- ❖ 2a2, 2b3: A wash is in progress:
 - ❖ 2a21. The carwash completes the current wash, then displays an out-of-order message; the use case ends.
- ❖ 3a: The carwash is inactive:
 - ❖ 3a1. The carwash becomes active and displays a ready message.

Dodatni diagrami in modeli

- ❖ Poteki za posamezen primer uporabe so lahko dokumentirani z uporabo diagramov poteka oziroma aktivnosti.
- ❖ Procesi sistema so lahko predstavljeni z DFD ali komunikacijskim oziroma sekvenčnim diagramom.
- ❖ Podatki v sistemu so lahko predstavljeni z ERD ali razrednim diagramom.
- ❖ Drugo – po potrebi.

Razgradnja sistema na objekte

- ❖ Pri objektnem pristopu se dekompozicija izvaja kot delitev sistema na komponente oziroma podsklope (objekte).
 - ♦ Med tem ko gre pri klasični strukturalni analizi, za funkcijsko dekompozicijo.
- ❖ Objekte opredelimo z razredi (ang. class).
 - ♦ Pri tem se omejimo na domenski model (enostaven model brez atributov in operacij)

Modeliranje razredov

- ❖ Glavno orodje pri določitvi razredov sistema so primeri uporabe.
- ❖ Ena od metod določitve razredov je identifikacija samostalnikov (ang. Noun Identification).
- ❖ Identifikacija samostalnikov je enostavna a lahko tudi zavajajoča (naravni jeziki niso eksaktni).

Primer identifikacije samostalnikov

1. The **carwash** queries the **soap sensor**.
2. The **soap sensor** indicates that there is **soap**.
3. If the **carwash** is active, it continues its **operation** and the use case ends.
 - ❖ 1a: The **carwash** is unable to query the **soap sensor**:
 - ❖ 1a1. The **controller** logs the **problem** and the use case ends.
 - ❖ 2a: The **soap sensor** indicates that there is no more **soap**:
 - ❖ 2a1. If the **carwash** is inactive, the use case ends.
 - ❖ 2a2. If the **carwash** is active, the **controller** displays an out-of-order **message** and becomes inactive; the use case ends.
 - ❖ 2b: The **soap sensor** does not respond:
 - ❖ 2b1. The **controller** logs the **problem**.
 - ❖ 2b2. If the **carwash** is inactive, the use case ends.
 - ❖ 2b3. If the **carwash** is active, the **controller** displays an out-of-order **message** and becomes inactive; the use case ends.
 - ❖ 2a2, 2b3: A **wash** is in **progress**:
 - ❖ 2a21. The **carwash** completes the current **wash**, then displays an out-of-order **message**; the use case ends.
 - ❖ 3a: The **carwash** is inactive:
 - ❖ 3a1. The **carwash** becomes active and displays a ready **message**.

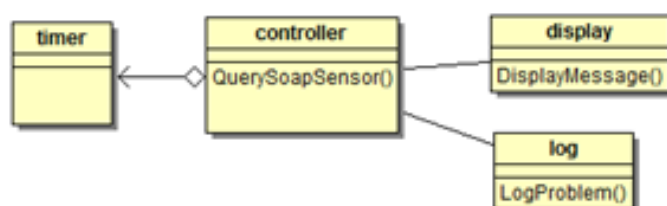
Izločanje kandidatov za razrede

- ❖ Vsi samostalniki niso primerni za razrede:
 - ♦ različni samostalniki imajo lahko isti pomen,
 - ♦ samostalniki poleg razredov identificirajo tudi attribute,
 - ♦ nekateri samostalniki ne predstavljajo objektov zanimanja (jih ne potrebujemo).
- ❖ Iščemo najeostavnejšo skupino razredov, ki zadostijo potrebam in omogočajo zadostno prilagodljivost (vzdrževanje).

Kriteriji izločanja

- ❖ Redundančni razredi (različni samostalniki za isti pomen). Ohranimo bolj informativnega. Primer: stranka/kupec.
- ❖ Nepomembni razredi – izven konteksta problema.
- ❖ Nejasni razredi (nedoločeni). Primer: izpad.
- ❖ Atributi: Primeri: ime, starost, teža, letnik...
- ❖ Funkcije – opisi obnašanja razreda. Primer: pospeševanje.
- ❖ Vloge – imena razredov naj podajajo naravo objekta, ne njegovo vlogo. Primer: stranka ali lastnik avtomobila, voznik avtomobila, najemnik avtomobila...
- ❖ Implementacijski konstrukti. Primer: CPU, array, linked list, subrutine...

Primer – identificirani razredi



Možne so različne skupine razredov.

Vsak dodaten primer uprabe lahko razširi nabor potrebnih razredov.

Dinamični modeli

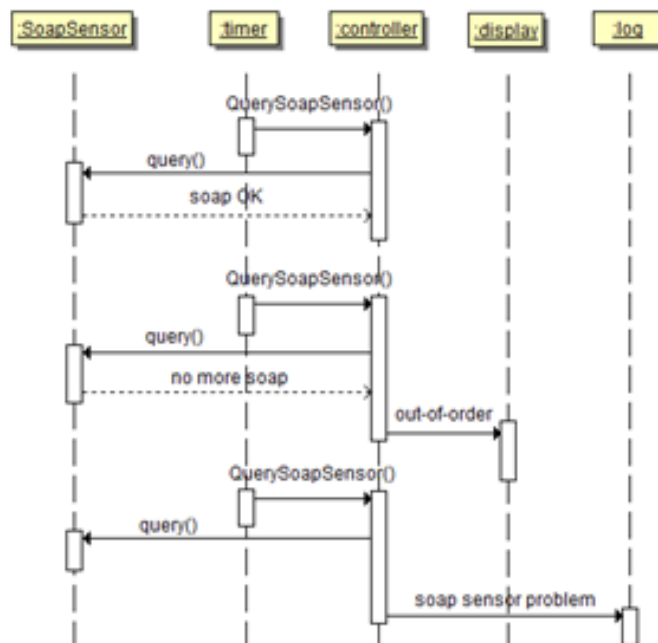
❖ Razredni diagrami podajajo statični pogled na razrede. Njihovo delovanje oziroma dinamično komponento sistema prikažemo z vedenjskimi diagrami.

- ♦ Interakcija med razredi:
 - UML sekvenčni diagrami
 - UML komunikacijski diagrami
- ♦ Aktivnosti znotraj razreda:
 - UML diagram prehajanja stanj
 - UML aktivnostni diagrami

UML diagrami interakcije

- ❖ Opisujejo sodelovanje med objekti sistema.
- ❖ Tipično opisujejo obnašanje za posamezen primer uporabe, tako da prikažejo objekte in tok sporočil med njimi.
- ❖ Za nas sta zanimivi dve vrsti diagramov interakcije (sekvenčni diagram in komunikacijski diagram), ki prikazujeta isto informacijo na dva različna načina.
- ❖ Izbira diagrama je stvar osebnih/organizacijskih naklonjenosti.

Sekvenčni diagram: primer



Komunikacijski diagram: primer

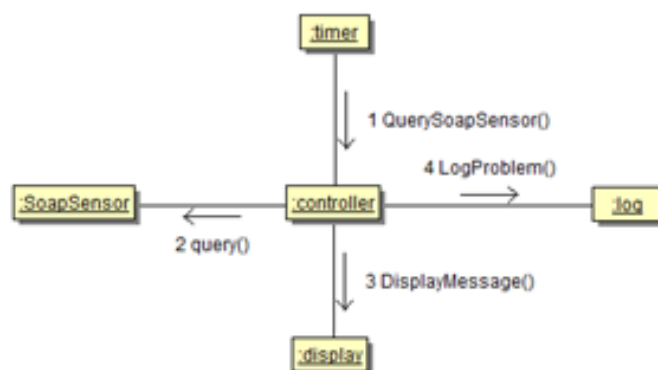


Diagram prehajanja stanj

- ❖ Opisuje življenjski cikel objekta
 - Vsa stanja objekta
 - Prehajanje med stanji kot posledica dogodkov
- ❖ Dobro opisuje obnašanje objekta v celoti – glede na več primerov uporabe.

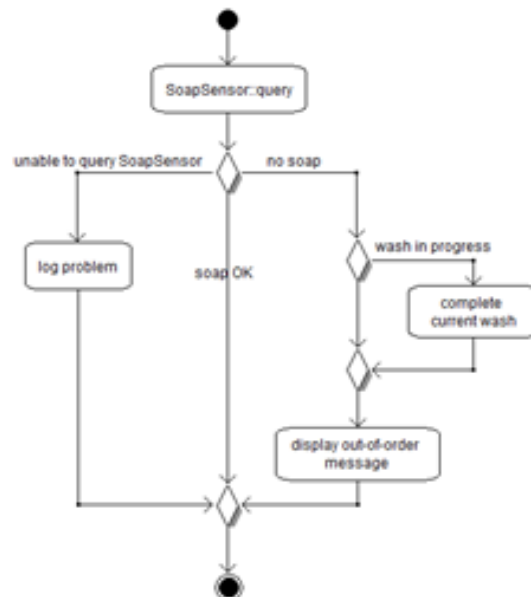


Primer stanj objekta controller

Aktivnostni diagram (UML)

- ❖ UML aktivnostni diagram opisuje aktivnost, ki se tipično izvaja v okviru ene operacije (metode) razreda.
- ❖ Aktivnost lahko pokriva več primerov uporabe.
- ❖ Omogočajo celovit opis obnašanja sistema.

Aktivnostni diagram - primer



Po modeliranju...

- ❖ Ne glede na uporabljeno metodologijo so za različne udeležence v projektu pomembne različne, tipično tudi **drugačne** zahteve!

Usklajevanje zahtev

- ❖ Pogosto stranka/uporabniki želijo več kot je mogoče doseči (običajno gre za vprašanje sredstev).
- ❖ Pogosto podajo različni nosilci interesov nezdružljive zahteve.
 - ♦ Naprimer: management omejuje sredstva ali čas razvoja, uporabniki pa želijo dodatne funkcionalnost ali večjo prepustnost sistema.
 - ♦ Vsak nosilec interesov lahko svoje zahteve označi za ključne.
- ❖ Sistemski analitik mora razrešiti neskladja s pogajanji:
 - ♦ Določiti prioritete zahtev.
 - ♦ Identificirati rizike povezane z zahtevami.
 - ♦ Grobo oceniti (ang. "guesstimate") razvojni vložek za vsako od zahtev.
 - ♦ Zahteve prediskutirati s nosilci interesov.
- ❖ Usklajevanje je iterativen proces spreminjanja, združevanja in izločanja zahtev, tako da rešitev (vsaj deloma) zadovolji vse nosilce interesov.

Specifikacija zahtev

- ❖ Vsebina dokumenta specifikacij zahtev (SRS)
 - ♦ Uvod (namen, obseg, definicije, reference na druge dokumente)
 - ♦ Splošni opis (relacije z drugimi sistemi, pregled funkcij/komponent, značilnosti uporabnikov, splošne omejitve)
 - ♦ Opis zahtev vsake posamezne komponente (opis, vhodi, obdelava, izhodi)
 - ♦ Zahteve zunanjih vmesnikov (formati, strojna oprema, povezave z drugimi programi ali sistemi)
 - ♦ Zahteve po prepustnosti sistema (performance)
 - ♦ Omejitve pri načrtovanju (standardi, omejitve strojne opreme...)
 - ♦ Ostale zahteve
- ❖ Vsebina se mora prilagajati potrebam projekta!
- ❖ Problematiko SRS obravnava standard IEEE 830-1998 (Recommended Practice for Software Requirements Specifications)

Recenzija

- ❖ Recenzija je pregled in vrednotenje izdelka ali procesa s strani kvalificiranega posameznika ali skupine.
- ❖ Recenzijo specifikacij zahtev opravijo predstavniki nosilcev interesov (strank, managementa...).

Recenzija specifikacije zahtev

- ❖ Pregled vseh zahtev:
 - ♦ Veljavnost. Ali sistem nudi funkcije, ki najbolj zadostijo potrebam uporabnika?
 - ♦ Konsistentnost. Ali so kakšne zahteve med seboj v konfliktu?
 - ♦ Popolnost. Ali so upoštevane vse funkcije, ki jih zahteva uporabnik?
 - ♦ Realnost. Ali so zahteve lahko realizirane s predvideno tehnologijo in v okviru omejitve stroškov?
- ❖ Preverjanje posameznih zahtev:
 - ♦ Preverljivost. Ali je zahtevo realno mogoče preveriti?
 - ♦ Razumljivost. Ali je zahteva pravilno razumljena?
 - ♦ Sledljivost. Ali je izvor zahteve jasno naveden?
 - ♦ Prilagodljivost. Ali je zahtevo mogoče spremeniti brez velikega vpliva na druge zahteve?

Tipi recenzij

- ❖ Kontrola s strani posameznika
 - ♦ pogosto s strani avtorja
 - ♦ uporaba **kontrolnih seznamov**
- ❖ Skupinski pregled
 - ♦ S strani skupine pregledovalcev
 - ♦ v skupini ni določenih vlog, postopek ni preddefiniran
 - ♦ običajno vsi pregledovalci predhodno pregledajo sami (kontrola posameznika)
- ❖ Nadzor
 - ♦ formalna recenzija skupine izšolanih pregledovalcev (nadzornikov) z določenimi vlogami
 - ♦ uporaba kontrolnih seznamov
 - ♦ obvezna predpriprava pregledovalcev

Nadzor specifikacije zahtev

- ❖ Namen je najti kar največ napak.
- ❖ Ni namenjeno presojanju avtorja.
- ❖ Nadzor ne popravlja napak.
- ❖ Nadzor je drag in dolgotrajen
- ❖ Nadzor je učinkovit

Primer kontrolnega seznama

- ☐ Every requirement is atomic.
- ☐ Every requirement statement uses "must" or "shall."
- ☐ Every requirement statement is in the active voice.
- ☐ Terms are used with the same meaning throughout.
- ☐ No synonyms are used.
- ☐ Every requirement statement is clear.
- ☐ No requirement is inconsistent with any other requirement.
- ☐ No needed feature, function, or capability is unspecified.
- ☐ No needed characteristic or property is unspecified.
- ☐ All design elements are specified to the physical level of abstraction.
- ☐ Every requirement is verifiable.
- ☐ Similar design elements are treated similarly.
- ☐ Every requirement can be realized in software.
- ☐ Every requirement plays a part in satisfying some stakeholder's needs or desires.
- ☐ Every requirement correctly reflects some stakeholder's needs or desires.
- ☐ Every requirement statement is prioritized.
- ☐ Every requirement priority is correct.

Zaključki nadzora

- ❖ Moderator ugotovi
 - ◆ obstoj napak (ki jih je potrebno odpraviti) ali
 - ◆ da so odpravljene vse napake.
- ❖ Avtor odpravi napake, ki so jih odkrili nadzorniki.
- ❖ V primeru potrebe po večjih spremembah ali če obstaja večja verjetnost obstoja dodatnih napak, se nadzor po odpravljanju napak ponovi.

Upravljanje z zahtevami

- ❖ Spremembe zahtev
 - ♦ so neizogibne
 - ♦ se dogajajo v celem življenjskem ciklu programskih produktov.
- ❖ Upravljanje z zahtevami pomeni
 - ♦ identificiranje,
 - ♦ sledenje in
 - ♦ nadzor nad zahtevami in spremembami

6 Načrtovanje sistema

Sisteme določajo medsebojno povezane komponente. Te so ponovno lahko svoj lasten sistem. Prav delitev programskega produkta na komponente je ključna naloga načrtovanja programskega sistema. Vsaka komponenta mora biti dobro opredeljena v smislu njenih lastnosti, zahtev, ki jih z njo izpolnjujemo, ter vmesnikov do drugih komponent sistema in/ali do zunanjih akterjev.

Prvi korak načrtovanja programskega sistema je določitev načrtovalskih ciljev. To so lastnosti, ki tipično izhajajo iz predhodno opredeljenih nefunkcijskih zahtev. Sledi dekompozicija sistema, s katero celoten kompleksen sistem razdelimo na enostavnejše dele. Razvoj več enostavnejših komponent je namreč enostavnejši od razvoja kompleksnega sistema kot celote, hkrati pa omogoča tudi lažje razumevanje in vzdrževanje sistema. Pri delitvi sistema na komponente so nam lahko v pomoč arhitekturni vzorci; preizkušeni koncepti reševanja tipičnih problemov arhitekture programskih sistemov. Kriterij za optimalno delitev sistema na komponenta sta tipično povezanost komponent, ki naj bo čim manjša, in kohezija posameznih komponent, ki naj bo čim večja.

Načrtovanje sistema ni algoritmično zato sta za doseganje dobrih rezultatov ključna dobro poznavanje tehnologij in ustvarjalnost načrtovalca.

Arhitekturni primer načrtovanja

❖ Načrtovanje prostorov v hiši:

- Ko se s stranko dogovori o številu sob, nadstropij, velikosti in lokaciji hiše, mora arhitekt predlagati načrt razporeditve prostorov ter lokacije vrat in oken.
- Pri tem mora arhitekt upoštevati vrsto standardov (standardne velikosti kuhinjskih elementov, postelj...)
- Arhitekt ne sme omejevati dejanske opreme prostorov!

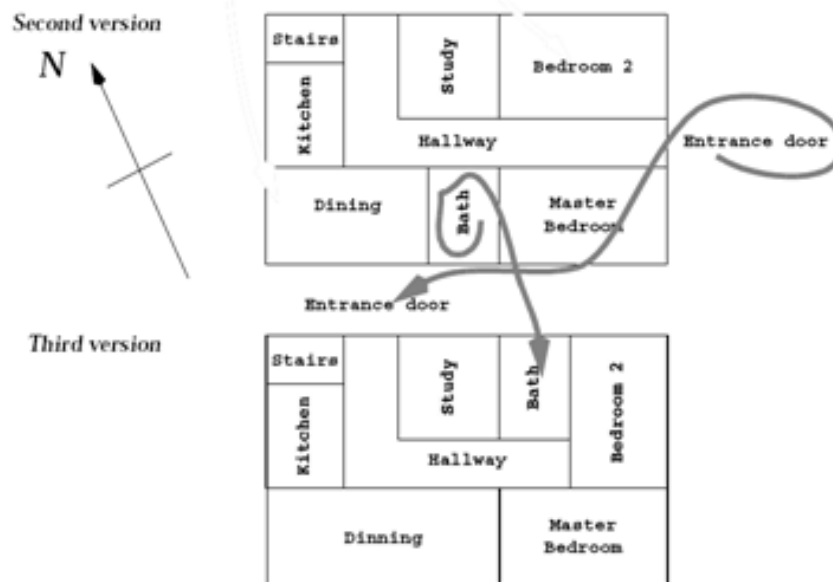
❖ Načrtovalska izhodišča (specifikacije zahtev):

1. Hiša naj ima dve spalnici, študijsko sobo, kuhinjo, in dnevno sobo (funkcijska zahteva).
2. Razdalja, ki jo morajo prebivalci prehoditi tekom dneva naj bo čim manjša (nefunkcijska zahteva).
3. Čim večja izraba dnevne svetlobe (nefunkcijska zahteva).
4. Čim boljša izraba prostora - velikost bivalnih prostorov (nefunkcijska zahteva).

Arhitekturni primer



Arhitekturni primer



Arhitekturni primer

- ❖ Sorodnost z načrtovanjem programskih arhitektur:
 - ♦ Celotni sistem se razdeli na enostavnejše komponente (hiša → sobe, sistem → podsistemi).
 - ♦ Pomembne so povezave med komponentami (vrata, vmesniki).
 - ♦ Prisotnost nefunkcijskih zahtev (velikost prostora, odzivnost).
 - ♦ Prisotnost funkcijskih zahtev (število spalnic, primeri uporabe).
 - ♦ Načrtovanje vpliva na izvedbo (postavitve kuhinje, izvedba podsistemov).
 - ♦ Težavnost (in cena) kasnejšega spreminjanja (premikanje zidov, spremembe vmesnikov).
 - ♦ Načrtovanje posameznih komponent se prepusti kasnejšemu načrtovanju (oprema prostorov, načrt izvedbe podsistemov).

Načrtovanje sistema

- ❖ Načrtovanje sistema je aktivnost s katero preslikamo model specifikacije zahtev v načrtovalski model sistema (ang. system design model).
- ❖ **Načrtovalski model prikazuje sistem kot skupino podsistemov, njihovo medsebojno odvisnost in vmesnike.**
- ❖ Načrtovalski model vključuje vse odločitve, ki vplivajo na strukturo celotnega sistema, ki jo označujemo tudi z izrazom programska arhitektura.
- ❖ Načrtovanje sistema NI algoritmična aktivnost!

Stabilna arhitektura

- ❖ Stabilna arhitektura zagotavlja zanesljivost sistema in enostavno vzdrževanje.
- ❖ Stabilna arhitektura je tista, ki omogoča enostavno dodajanje funkcionalnosti z le majhnimi spremembami arhitekture.

Načrtovanje sistema

❖ Načrtovanje sistema vključuje:

- ♦ definicijo načrtovalskih ciljev,
- ♦ dekompozicijo sistema na podsisteme,
- ♦ izbiro že izdelanih komponent,
- ♦ relacije med programsko in strojno opremo,
- ♦ izbira infrastrukture za upravljanje s perzistentnimi podatki,
- ♦ določitev politike nadzora dostopa,
- ♦ izbira mehanizma globalnega kontrolnega toka,
- ♦ obravnavanje mejnih primerov.

Definicija načrtovalskih ciljev

- ❖ Definiranje načrtovalskih ciljev je prvi korak načrtovanja sistema.
- ❖ Mnogi načrtovalski cilji izhajajo iz nefunkcijskih zahtev ali domene aplikacije.
- ❖ Preostale načrtovalske cilje je potrebno uskladiti s stranko.
- ❖ V splošnem lahko načrtovalske cilje izberemo iz seznama splošno željenih kvalitet programskih produktov, ki jih uredimo v pet skupin:
 - ♦ učinkovitost (ang. performance),
 - ♦ zanesljivost (ang. dependability),
 - ♦ cena (ang. cost),
 - ♦ zmožnost vzdrževanja (ang. maintainance),
 - ♦ uporabniške zahteve (ang. user criteria).

Načrtovalski cilji - učinkovitost

- ❖ Odzivni čas (ang. response time)
 - ♦ Koliko časa preteče od zahteve do odziva?
- ❖ Prepustnost (ang. throughput)
 - ♦ Koliko nalog lahko sistem izvrši v časovni enoti?
- ❖ Pomnilnik (ang. memory)
 - ♦ Koliko pomnilnika (na splošno prostora) je potrebnega za delovanje sistema?

Načrtovalski cilji - zanesljivost

- ❖ Robustnost (ang. robustness)
 - ♦ Zmožnost preživetja ob neveljavnem uporabniškem zahtevku (vhodni tok).
- ❖ Zanesljivost (ang. reliability)
 - ♦ Razlika med specificiranim in opaženim obnašanjem.
- ❖ Razpoložljivost (ang. availability)
 - ♦ Delež časa, ko je sistem lahko uporabljen za opravljanje običajnih nalog.
- ❖ Odpornost (ang. fault tolerance)
 - ♦ Zmožnost delovanja pod neustreznimi pogoji (ang. erroneous conditions).
- ❖ Varnost (ang. security)
 - ♦ Zmožnost zoperstavljanja zlonamernim napadom.
- ❖ Varnost (ang. safety)
 - ♦ Zmožnost ne ogrožanja človeških življenj, tudi v primeru prisotnosti napak in odpovedi.

Načrtovalski cilji - cena

- ❖ **Cena razvoja (ang. development cost)**
 - ♦ Cena razvoja (začetnega) sistema.
- ❖ **Cena namestitve (ang. deployment cost)**
 - ♦ Cena namestitve in izobraževanja uporabnikov.
- ❖ **Cena nadgradnje (ang. upgrade cost)**
 - ♦ Cena prevedbe podatkov predhodnega sistema.
- ❖ **Cena vzdrževanja (ang. maintenance cost)**
 - ♦ Cena odpravljanja napak in razširjanja sistema.
- ❖ **Cena administriranja (ang. administration cost)**
 - ♦ Stroški administriranja sistema.

Načrtovalski cilji – vzdrževanje

- ❖ **Razširljivost (ang. extensibility)**
 - ♦ Kako enostavno je dodajanje novih funkcionalnosti?
- ❖ **Spremenljivost (ang. modifiability)**
 - ♦ Kako enostavno je spremeniti funkcionalnost sistema?
- ❖ **Prilagodljivost (ang. adaptability)**
 - ♦ Kako enostavno je sistem prevesti za uporabo na drugih področjih uporabe?
- ❖ **Prenosljivost (ang. portability)**
 - ♦ Kako enostavno je prenesti sistem na druge platforme?
- ❖ **Čitljivost (ang. readability)**
 - ♦ Kako enostavno je razumeti sistem na osnovi branja izvorne kode?
- ❖ **Sledljivost zahtev (ang. traceability of requirements)**
 - ♦ Kako enostavno je kodo povezati s specifično zahtevo?

Načrtovalski cilji – uporabniške zahteve

- ❖ Koristnost (ang. utility)
 - ♦ Kako dobro sistem podpira delo uporabnika?
- ❖ Uporabnost (ang. usability)
 - ♦ Kako enostavno je uporabniku uporabljati sistem?

Razkoraki načrtovalskih ciljev

- ❖ Hkrati je mogoče zadostiti le manjšemu deležu vseh navedenih kriterijev.
 - ♦ Primer: ni realno pričakovati sistema, ki bo varen in poceni.
- ❖ Tipično razvijalec načrtovalskim ciljem dodeli prioritete, ki jih uporabi pri odločitvah ob razvoju sistema.

Nasprotujoči načrtovalski cilji

- ❖ Pomnilnik proti hitrosti
 - ♦ Prepustnost in odzivnost sistema je mogoče povečati na račun uporabe pomnilnika.
- ❖ Čas dobave proti funkcionalnost
 - ♦ Hitrejši čas dobave je mogoče zagotoviti z omejevanjem funkcionalnosti sistema.
- ❖ Čas dobave proti kakovosti
 - ♦ V priporočanju časa lahko pride do odločitve po predaji produkta z znanimi napakami.
- ❖ Čas dobave proti številu razvijalcev
 - ♦ Skrajševanje časa dobave z večanjem števila razvijalcev je smiselno le v začetnih fazah razvoja, sicer se z večanjem števila razvijalcev čas izvedbe lahko celo podaljša (potreba po izobraževanju, obsežna komunikacija...)
- ❖ ...

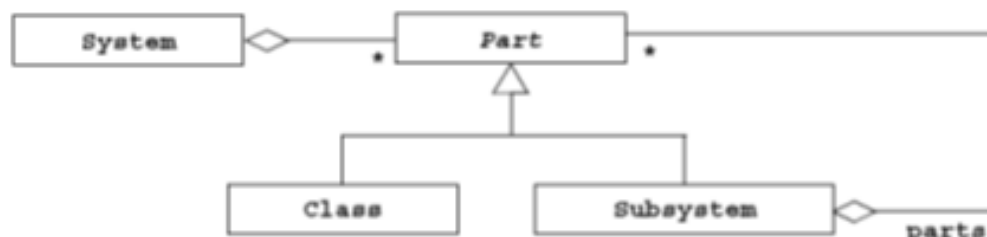
Dekompozicija prog. sistema

- ❖ Obravnavanje velikih sistemov je veliko težje od obravnavanja serije majhnih komponent - podsistemov.
- ❖ Deli in zmagaj („Divide and conquer“)!
 - ♦ Manjše komponente je lažje razumeti kot velike.
 - ♦ Za vsak podsistem lahko skrbijo drugi ljudje.
 - ♦ Možnost specializacije posameznih programskih inženirjev.
 - ♦ Komponente je mogoče zamenjati brez večjih popravkov ostalega sistema.

Identifikacija podsistemov

- ❖ Identifikacija podsistemov zahteva nekaj iznajdljivosti načrtovalca (podobno kot iskanje objektov v fazi specificiranja zatev)
- ❖ Ob vsaki dodatni zahtevi se lahko:
 - več podsistemov združi v en podsistem,
 - kompleksne podsisteme razdeli na več podsistemov,
 - doda podsisteme, ki zagotavljajo dodatno funkcionalnost.
- ❖ Možen hevristični pristop:
 - Objekti iz istega primera uporabe naj bodo del istega podsistema.
 - Ločen podsistem naj bo namenjen prenosu podatkov med podsistemi.
 - Minimiraj število povezav med podsistemi.
 - Vsi objekti v podsistemu naj bodo funkcionalno povezani.

Koncept podsistemov



Koncept podsistemov

- ❖ **Podsistemi** nudijo **storitve** drugim podsistemom.
- ❖ **Storitev** je skupina sorodnih operacij.
- ❖ Ob načrtovanju sistema (arhitektura) je potrebno definirati podsisteme v smislu storitev, ki jih podsistemi nudijo.
- ❖ Operacije podsistema, ki so na voljo drugim sistemom, tvorijo **vmesnik podsistema** (ang. subsystem interface), imenovan tudi programski vmesnik (ang. application programmer interface) (**API**).

Načrtovanje storitev

- ❖ Ob načrtovanju sistema je potrebno definirati storitve vsakega podsistema:
 - ♦ naštetih operacije,
 - ♦ določiti njihove parametre,
 - ♦ določiti njihovo obnašanje na visoki ravni.
- ❖ Šele kasneje, ob načrtovanju objektov, se določi programski vmesnik:
 - ♦ tip parametrov
 - ♦ tipi vrnjenih podatkov vsake operacije

Načini delitve programskih sistemov

- ❖ Porazdeljen sistem se deli na odjemalce (ang. client) in strežnike (ang. server).
- ❖ Sistem se deli na podsisteme (ang. Subsystem).
- ❖ Podsystem lahko sestoji iz enega ali več paketov (ang. package).
- ❖ Paket sestoji iz razredov (ang. class).
- ❖ Razred vsebuje več metod (ang. method).

Sloji in particije

- ❖ **Sloji** (ang. layers) ločujejo podsisteme hierarhično, vertikalno.
- ❖ **Particije** (ang. partitions) ločujejo podsisteme istega sloja, horizontalno.
- ❖ Vsak podsystem (sloj ali particija) je odgovoren za drugačne storitve. Podsistemi so medsebojno šibko povezani in pogosto delujejo samostojno.

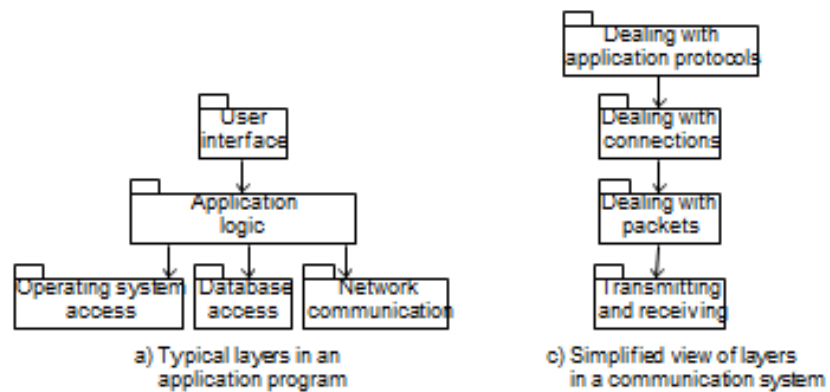
Sloji

- ❖ Sloji tvorijo hierarhično strukturo. Sloj zagotavlja storitve višjim slojem in je odvisen le od nižjih slojev:
 - ♦ Zaprta arhitektura: sloj uporablja le storitve prvega nižjega sloja.
 - ♦ Odprta arhitektura: sloj lahko uporablja storitve vseh nižjih slojev.
- ❖ Sloj je mogoče zamenjati brez vplivov na druge sloje.

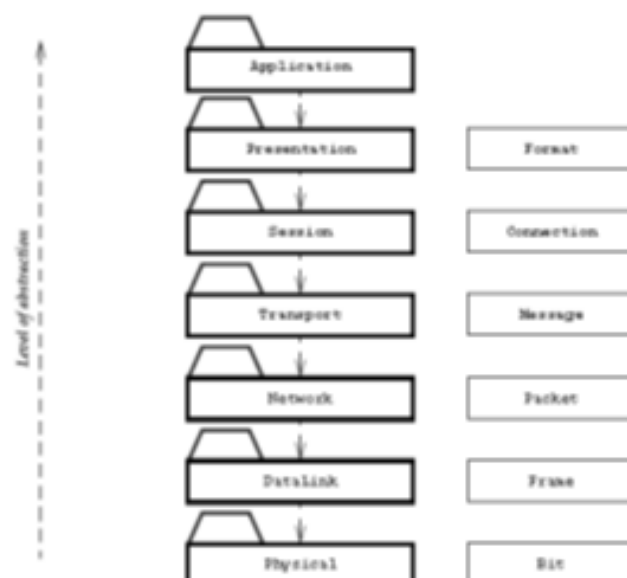
Sloji

- ❖ Sloji v kompleksnih sistemih nudijo vpogled v sistem na različnih nivojih abstrakcije (s tem tudi nivoje abstrakcije pri razvoju sistema):
 - ♦ Pomembno je imeti ločen sloj namenjen uporabniškemu vmesniku (UI).
 - ♦ Sloj pod UI nudi funkcije določene s primeri uporabe (use-case)
 - ♦ Najnižji sloji nudijo splošne storitve, npr. komunikacije po omrežju, dostop do podatkovne baze...

Primeri slojev



Sloji ISO OSI komuikac. modela



Povezanost podsistemov

- ❖ Povezanost (ang. coupling) govori o odvisnosti med dvema podsistemoma.
- ❖ Če sta dva podsistema močno povezana, bodo imele spremembe enega od podsistemov vpliv na delovanje drugega podsistema.
- ❖ Zaželeno je, da so podsistemi povezani kar se da šibko.
 - ♦ S tem je dosežen manjši vpliv napak in sprememb enega podsistema na delovanje drugega podsistema.
- ❖ Ločimo več tipov povezanosti:
 - ♦ Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External

Dodatni vir: http://en.wikipedia.org/wiki/Coupling_%28computer_science%29

Vsebinska povezanost

- ❖ Vsebinska povezanost (ang. content coupling) - ko ena komponenta "tajno" spremeni interni podatek druge komponente.
 - ♦ Za zmanjšanje vsebinske povezanosti je predlagano ovijanje (ang. encapsulate) vseh spremenljivk instance:
 - definiranje spremenljivk kot privatne (`private`)
 - zagotovitev `get` and `set` metod.

Skupna povezanost

- ❖ Skupna povezanost (ang. common coupling) - kadar se uporabljajo globalne spremenljivke
 - ♦ Vse komponente, ki uporabljajo globalno spremenljivko postanejo medsebojno povezane.
 - ♦ Lahko so sprejemljive, če globalne spremenljivke predstavljajo privzete vrednosti sistema.
 - ♦ Za globalni dostop do objekta se lahko uporablja "Singleton" načrtovalski vzorec.

Kontrolna povezanost

- ❖ Kontrolna povezanost (ang. control coupling) - če ena procedura kliče drugo proceduro z navedbo ukaza, ki eksplicitno določa delovanje druge procedure.
 - ♦ V primeru sprememb je potrebno spremeniti obe proceduri.
 - ♦ Kontrolni povezanosti se je pogosto mogoče izogniti z uporabo polimorfnih operacij (ista operacija se lahko različno odraža pri različnih razredih).

Pečatna povezanost

- ❖ **Pečatna povezanost (ang. stamp coupling)** - ko eden od razredov uporabljen kot tip (ang. type) argumenta neke metode (drugega razreda).
 - ♦ Ker eden od razredov uporablja drugega, so spremembe sistema težje:
 - v primeru sprememb so potrebne spremembe obeh razredov.
 - ponovna uporaba razreda zahteva ponovno uporabo obeh razredov.
 - ♦ Dva načina odpravljanja tovrstne povezanosti:
 - uporaba vmesnika (ang. interface) za tp argumenta.
 - uporaba enostavnih spremenljivk (primitivnih/enostavnih tipov).

Podatkovna povezanost

- ❖ **Podatkovna povezanost (ang. data coupling)** - kadar so tipi argumentov metod primitivni ali pa enostavni razredi knjižnic.
 - ♦ Klic metode iz drugega razreda zahteva navedbo (večjega števila) podatkov. Več ko je argumentov, večja je povezanost.
 - ♦ Povezanost se lahko zmanjša z neuporabo nepotrebnih argumentov.
 - ♦ Obstaja razkorak med podatkovno in pečatno povezanostjo
 - Zmanjšanje ene tipično poveča drugo.

Rutinska povezanost

- ❖ Rutinska povezanost (ang. routine call coupling)
 - kadar ena rutina (ozirno metoda v OO sistemu) kliče drugo.
 - ♦ Rutini sta povezani ker je ena rutina odvisna od obnašanja druge rutine.
 - ♦ Rutinska povezanost je vedno prisotna, v vsakem sistemu.
 - ♦ V primeru, če je večkrat uporabljena sekvenca dveh ali več metod (rutin), je rutinsko povezanost mogoče zmanjšati z izgradnjo rutine, ki ovije (ang. encapsulate) sekvenco.

Tipna povezanost

- ❖ Tipna povezanost (ang. type use coupling) - kadar modul uporablja podatkovni tip določen v drugem modulu.
 - ♦ Ob uporabi razreda drugega modula (class) kot podatkovnega tipa za deklaracijo spremenljivk.
 - ♦ V primeru spremembe definicije podatkovnega tipa, lahko to zahteva tudi spremembo procedur, ki ta tip uporabljajo.
 - ♦ V primeru OO sistema naj se vedno uporablja najbolj splošen razred (ali vmesnik), ki vsebuje zahtevane operacije.

Vključevalna povezanost

- ❖ Vključevalna povezanost (ang. inclusion or import coupling) - kadar ena komponenta vključuje paket ("import" v Javi) ali drugo komponento ("include" v C++)
 - ♦ komponenta je podvržena vsemu, kar vsebuje vključena komponenta.
 - ♦ V primeru sprememb ali dopolnitev vključene komponente lahko pride do konfliktov (konflikti v imenih spremenljivk, funkcij, definicij...).

Zunanja povezanost

- ❖ Zunanja povezanost (ang. external coupling) - kadar je modul odvisen od operacijskega sistema, skupnih (shared) knjižnic ali strojne opreme.
 - ♦ Število mest, kjer prihaja do takšnih odvisnosti naj bo čim manjše.
 - ♦ Zunanjo poveznost je mogoče zmanjšati z uporabo "Façade" načrtovalskega vzorca.

Kohezija podsistemov

- ❖ Kohezija (ang. cohesion) govori o notranji povezanosti znotraj podsistema.
- ❖ Kohezija je velika, če podsistem združuje medseboj odvisne oz. povezane komponente ter ne vsebuje ostalih, nepovezanih komponent).
- ❖ Zaželjena je velika kohezija podsistema, kajti takšni podsistemi so lažji za razumevanje in spreminjanje.
- ❖ Ločimo več tipov kohezije
 - ♦ Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Funkcijska kohezija

- ❖ Funkcijska kohezija (ang. functional cohesion) - je dosežena takrat, kadar je združena vsa koda, ki je namenjena pridobivanju določenega rezultata, med tem ko ostala koda ni vključena.
 - ♦ kadar je modul namenjen izvrševanju posamezne operacije.
 - ♦ Prednosti za sistem:
 - enostavnejše razumevanje
 - večja ponovna uporabljivost (ang. reusable)
 - enostavnejša zamenjava

Slojna kohezija

- ❖ Slojna kohezija (ang. layer cohesion) - v sloj (ang. layer) je združeno vse potrebno za zagotavljanje oziroma dostopanje do množice povezanih storitev (in vse ostalo ni vključeno).

Komunikacijska kohezija

- ❖ Komunikacijska kohezija (ang. communicational cohesion) - vsi moduli, ki dostopajo oz. uporabljajo določene podatke, so združeni (npr. v en razred), vse ostalo ni vključeno.
 - ♦ Razred ima dobro komunikacijsko kohezijo:
 - če so v njem vključene vse sistemske operacije povezane s shranjevanjem in upravljanjem njegovih podatkov.
 - če ne opravlja operacij, ki niso povezane z upravljanjem z njegovimi podatki.
 - ♦ Glavna prednost: ko so potrebne spremembe povezane s podatki, je vsa relevantna koda prisotna na enem mestu.

Sekvenčna kohezija

- ❖ Sekvenčna kohezija (ang. sequential cohesion) - združene so procedure v katerih ena procedura priskrbi vhodne podatke za drugo proceduro, vse ostalo ni vključeno.
 - ♦ Pogoji za zagotavljanje sekvenčne kohezije so predhodno zagotovljene vse prej omenjene kohezije.

Proceduralna kohezija

- ❖ Proceduralna kohezija (ang. procedural cohesion) - združene so procedure, ki so uporabljene ena za drugo.
 - ♦ Tudi če ena procedure niso povezane z vhodnimi podatki (ena procedura ne priskrbi vhodnih podatkov za drugo).
 - ♦ Proceduralna kohezija je šibkejša kot sekvenčna.

Časovna kohezija

- ❖ Časovna kohezija (ang. temporal cohesion) - združene so operacije, ki se izvajajo v isti fazi izvajanja programa, vse ostalo ni vključeno.
 - ♦ Primer: skupaj prisotna koda, ki se izvaja med inicializacijo ali zaustavitvijo sistema.
 - ♦ Časovna kohezija je šibkejša od proceduralne.

Kohezija pripomočkov

- ❖ Kohezija pripomočkov (ang. utility cohesion) - združene so operacije, ki ne morejo biti logično vključene v ostale kohezivne enote.
 - ♦ Pripomoček (ang. utility) je procedura ali razred, ki je širše uporaben v več podsistemih in je načrtovan tako, da ga je mogoče večkrat uporabiti.
 - ♦ Primer: razred `java.lang.Math`

Modeli odvisnosti

- ❖ Modeli odvisnosti pomagajo pri analizi programskih arhitektur v smislu odkrivanja težavnih odvisnosti med moduli.
- ❖ Razširjena sta dva načina vpogleda v odvisnost modulov:
 - ♦ Matrika odvisnosti
 - ♦ Graf odvisnosti

Dodatna literatura:

<http://sdq.csail.mit.edu/pubs/2005/oopsla05-dsm.pdf>

<http://www.stsc.hill.af.mil/Crosstalk/2005/11/0511SangalWaldman.html>

Matrika odvisnosti

\$root		←	→	ω	Δ
Module A	1	.		1	2
Module B	2		.	2	
Module C	3	4		.	3
Module D	4				.

Figure 1A

\$root		←	→	ω	Δ
Module D	1	.			
Module A	2	2	.	1	
Module C	3	3	4	.	
Module B	4			2	.

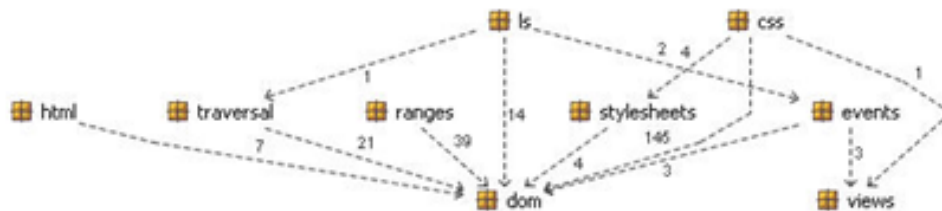
Figure 1B

- ❖ Ang. dependency structure matrix (DSM)
- ❖ Želimo si spodnje trikotne odvisnostne matrike, ki govori o tem, da v sistemu ni cikličnih odvisnosti med moduli.
- ❖ Primer kaže na ciklično odvisnost med moduloma A in C. Če modula združimo, sistem nima cikličnih odvisnosti.

Vir: <http://www.stsc.hill.af.mil/Crosstalk/2005/11/0511SangalWaldman.html>

Graf odvisnosti

- ❖ Diagram prikazuje module in njihove medsebojne relacije.



CASE orodja:

<http://www.headwaysoftware.com/products/structure101/index.php>

<http://www.coverity.com/products/architecture-analysis.html>

...

Relacije med podsistemi (HW, SW)

- ❖ Mnogi sistemi delujejo na več računalnikih, ki med seboj komunicirajo preko intraneta ali interneta.
- ❖ Potrebna je porazdelitev komponent sistema po računalnikih (vozliščih) in načrt komunikacijske infrastrukture.
- ❖ Pogosto so pri delitvi podsistemov po vozliščih potrebni dodatni podsistemi:
 - ♦ Komunikacijski podsistemi, namenjeni prenosu podatkov in zagotavljanju nadzora nad sočasnim izvajanjem transakcij (ang. concurrency),
 - ♦ Kontrolni sistemi, namenjeni zagotavljanju zanesljivosti delovanja (ang. reliability).

Relacije med podsistemi (HW, SW)

- ❖ Z razporeditvijo podsistemov po vozliščih je mogoče funkcionalnost in procesorsko moč porazdeliti tako, da je na voljo tam, kjer jo potrebujemo.
- ❖ Izzivi hranjenja, prenašanja podvajanja in sinhronizacije podatkov.

Upravljanje s perzistentnimi podatki

- ❖ Perzistentni podatki se ne izgubijo zaključkom enega izvajanja programa.
- ❖ Kje in kako so shranjeni podatki vpliva na dekompozicijo sistema.
 - ♦ V nekaterih primerih (primer arhitekture "shared repository" je temu namenjen celoten podsistem.
- ❖ Prvi korak načrtovanja hrambe podatkov je določitev perzistentnih podatkov, sledi odločitev o načinu hrambe:
 - ♦ Kateri podatki morajo biti hranjeni perzistentno?
 - ♦ Kako se dostopa do persiztentnih podatkov?

Upravljanje s podatki

❖ Upravljanje s perzistentnimi podatki pogosto predstavlja ozko grlo pri delovanju sistema:

- ♦ Dostop do podatkov mora biti hiter in zanesljiv.
- ♦ Pogosto zahteva izbiro sistema za upravljanje podatkovnih baz (ang. Database management system) ali dodatne podsisteme namenjene upravljanju perzistentnih podatkov.

Persiztentne shrambe podatkov

❖ Načini:

- ♦ Neposredna uporaba datotečnega sistema (ang. flat files).
 - Dostop do podatkov na relativno nizkem nivoju.
 - Aplikacija mora sama reševati probleme hkratnega dostopa do podatkov, izgube podatkov zaradi izpadov...
- ♦ Relacijske baze podatkov (ang. relational database)
 - Že zagotovljena stopnja abstrakcije pri dostopanju do podatkov.
 - Lahko predstavlja performančno ozko grlo.
- ♦ Objektno orientirane podakovne baze (ang. Object-oriented database)
 - Višja stopnja abstrakcije. Podatke hranijo v obliki objektov in povezav.
 - Tipično počasnejše delovanje kot relacijske baze.

Persiztentne shrambe podatkov

- ❖ Kdaj izbrati datotečni sistem?
 - Velikostno obsežni podatki (npr. slike),
 - Začasni podatki (npr. core datoteke),
 - Majhna gostota informacij (npr. arhivske datoteke, logi).
- ❖ Kdaj izbrati podatkovno bazo?
 - Hkratni dostopi,
 - Dodstop do detajlnih podatkov,
 - Več uporabljenih platform,
 - Več aplikacij uporablja iste podatke.
- ❖ Kdaj izbrati relacijske baze?
 - Kompleksne poizvedbe po atributih.
 - Obsežne podatkovne zbirke.
- ❖ Objektno orientirane podatkovne baze?
 - Obsežna uporaba povezav med objekti za pridobivanje podatkov.

Politika nadzora dostopa

- ❖ V večuporabniških sistemih imajo lahko različni uporabniki dostop do različnih podatkov (vprašanje avtorizacije).
- ❖ Določiti je potrebno kako se uporabniki v sistemu avtenticirajo (dokažejo pristnost).
- ❖ V splošnem je za vsakega akterja potrebno določiti operacije do katerih ima dostop za katere od podatkov (objektov).
- ❖ Nadzor dostopa je sistemska operacija. Biti mora skupen in enak za vse podsisteme.

Politika nadzora dostopa

❖ Možni načini:

- ♦ Globalna tabela dostopa (ang. global access table)
 - Določa pravice za relacijo (akter, objekt, operacija). Če pravice niso eksplicitno odobrene, je dostop zavrnjen.
- ♦ Kontrolna lista dostopa (ang. access control list)
 - Za vsak razred določa pravice za dostop na relaciji (akter, operacija). Primer: lista gostov na zabavi.
- ♦ Zmožnost (ang. capability)
 - Povezuje akterja z relacijo (razred, operacija). Uporabniku z določenimi zmožnostmi dopusti dostop do objekta. Primer: vstop na zabavo s povabilom.

Mehanizem globalnega kontrol. toka

❖ Vprašanja:

- ♦ Kako je v sistemu določen vrstni red izvajanja operacij oziroma aktivnosti?
- ♦ Ali omogoča več kot eno hkratno uporabniško interakcijo?
- ♦ Na kakšen način sistem pričakuje zunanjo (uporabniško) interakcijo.

❖ Izbira kontrolnega toka vpliva na vmesnike podsistemov:

- ♦ Dogodkovno upravljan kontrolni tok zahteva upravljalce dogodkov (ang. event handlers)
- ♦ Izbira niti (ang. threads) zahteva medsebojno izključevanje (ang. mutual exclusion) v kritičnih sekcijah.

Globalni kontrolni tok

- ❖ Uporabljajo se trije mehanizmi upravljanja kontrolnega toka:
 - ♦ Proceduralno upravljan kontrolni tok (ang. procedure-driven control)
 - ♦ Dogodkovno upravljan kontrolni tok (ang. event-driven control)
 - ♦ Niti (ang. threads)

Proceduralno upravljan kontrolni tok

- ❖ Operacije čakajo na vhod kadar potrebujejo podatke akterja.
- ❖ Težavno ob uporabi objektno orientiranih programskih jezikov.
 - ♦ Vrstni red vhodov je težko razbati iz kode, ker so razprešeni med več objektov.
- ❖ Primer:
 - ♦ `In.readLine("Login:");`

Dogodkovno upravljan kontrolni tok

- ❖ Glavna zanka čaka na zunanji dogodek.
- ❖ Ob zunanjem dogodku, posreduje posreduje podatke ustreznemu objektu.
- ❖ Težavnejša realizacija procedur z interakcijo v več zaporednih korakih.
- ❖ Priemer:

```
• while (eventStream.hasMoreElements) {  
    subscribers.processEvent(eventStream.nextEvent);  
}
```

Niti

- ❖ Sistem lahko kreira poljubno število niti, kjer vsaka odgovarja drugemu dogodku.
- ❖ Če nit za delovanje potrebuje dodatne vhodne podatke, jih pridobi (počaka) od specifičnega akterja.
- ❖ Najbolj intuitiven način delovanja.
- ❖ Težavno razhroščevanje zaradi nedeterminističnosti sistema.

Obravnavanje mejnih primerov

- ❖ Kako se sistem zažene?
 - ❖ Kako poteka inicializacija?
 - ❖ Kako poteka zaustavitev?
 - ❖ Kako naj sistem reagira v primeru napak vhodnih podatkov?
 - ❖ Kako naj podsistem reagira v primeru napak drugih podsistemov?
-
- ❖ Obravnavanje mejnih primerov vpliva na vmesnike vseh podsistemov!

Obravnavanje mejnih primerov

- ❖ Izjeme:
 - ♦ Uporabniška napaka:
 - Neustrezni vhodni podatki s strani uporabnika
 - ♦ Strojna napaka:
 - Napake omrežja, datotečnega sistema...
 - ♦ Programska napaka:
 - Sistem ali podsistem vsebuje napako. Posamezen podsistem lahko pričakuje napake drugih podsistemov in jih se pred njimi ustrezno zaščiti.

Arhitekturni vzorci

Kaj je vzorec (ang. pattern)?

- ❖ Christopher Alexander (1977), arhitektura:
 - ♦ *"Each pattern describes a problem which occurs over and over again in our environment, then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*
- ❖ Martin Fowler (1997), programsko inženirstvo:
 - ♦ *"An idea that has been useful in one practical context and will probably be useful in others"*

Kaj je vzorec (ang. pattern)?

- ❖ Dokumentacija rešitve ponavljajočega se problema.
- ❖ Rešitev je večkrat preizkušena in splošno priznana.
- ❖ Rešitev podaja način reševanja, ne pa neposredne rešitve, ki je odvisna od dejanskega reševanega problema.

Arhitekturni vzorci

- ❖ Pojem vzorcev se uporablja tudi v arhitekturah programskih sistemov.
 - ♦ Govorimo o arhitekturnih vzorcih ali arhitekturnih stilih.
 - ♦ Vsak arhitekturni vzorec omogoča načrtovanje prilagodljivih sistemov, ki sestojijo iz kar se da neodvisnih komponent.

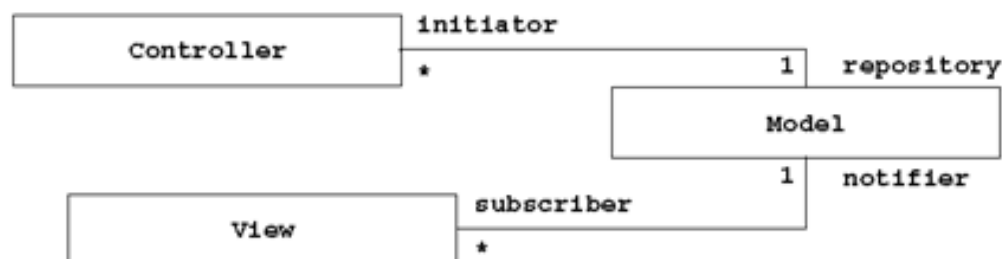
Arhitekturni vzorci

- ❖ Primeri arhitekturnih vzorcev:
 - ♦ "Layers"
 - ♦ "Model/View/Controller"
 - ♦ "Shared repository"
 - ♦ "Microkernel"
 - ♦ "Client/Server"
 - ♦ "Peer-to-peer"
 - ♦ "Pipes and filters"
 - ♦ ...

“Model/View/Controller”

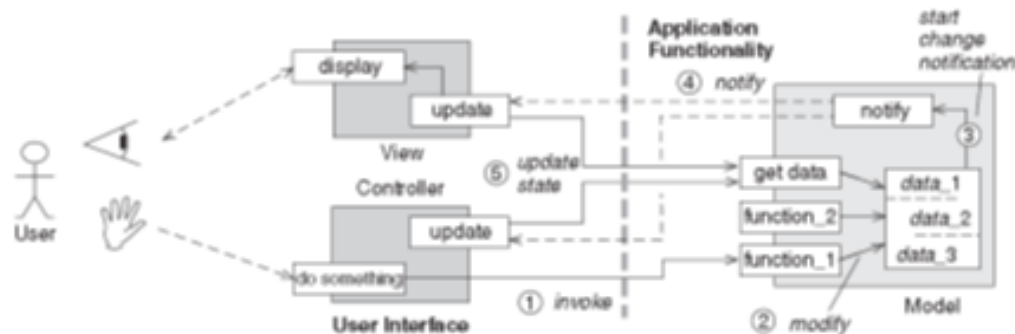
- ❖ Namen je ločiti sloj uporabniškega vmesnika od ostalih delov sistema:
 - ♦ Model (ang. *model*) združuje vse domensko znanje in operacije, med drugim objekte katerih lastnosti želimo prikazati uporabniku ali na njih vplivati z uporabniško interakcijo.
 - ♦ Pogled (angl. *view*) posreduje (prikazuje) podatke uporabniku.
 - ♦ Kontroler (ang. *controller*) nadzira uporabniško interakcijo in posreduje podatke uporabnika modelu in pogledu.

“Model/View/Controller”



- ❖ Kontroler zbira vhodne podatke s strani uporabnika in jih v obliki sporočil posreduje modelu. Model vzdržuje centralno podatkovno strukturo. Pogled prikazuje stanje modela. Obveščen je o vsaki spremembi modela z uporabo “subscribe/notify” protokola.

“Model/View/Controller”



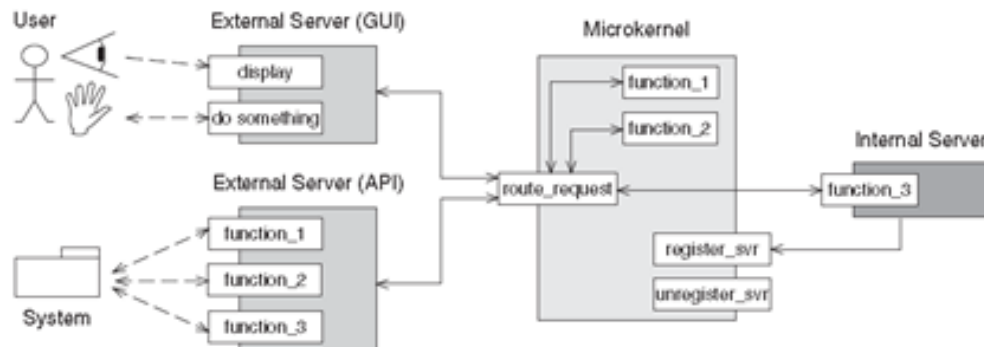
- ❖ Tri komponente so lahko načrtovane relativno neodvisno.
- ❖ Komunikacija med komponentami je minimalna.
- ❖ Možnost testiranja aplikacije (model) brez uporabniškega vmesnika (view+controller).

Vir: Buschmann et.al.: "Pattern-oriented software architecture", Volume 4, 2007

“Microkernel”

- ❖ Omogoča, da aplikacije različnih verzij uporabnikom nudijo različen nabor funkcionalnosti ali pa se razlikujejo od drugih verzij po specifičnih lastnostih, npr. uporabniškem vmesniku.
- ❖ Kljub razlikam med verzijami aplikacije, lahko vse verzije uporabljajo skupno funkcionalno jedro – “Microkernel”.

“Microkernel”



- ❖ Različne verzije aplikacije naj bodo zgrajene z razširitvijo skupnega funkcijskega jedra, npr. z uporabo plug-and-play infrastrukture.

Vir: Buschmann et.al.: "Pattern-oriented software architecture", Volume 4, 2007

Splošno o arhitekturnih vzorcih

- ❖ Nabor arhitekturnih vzorcev je bogat in ni omejen na prikazane vzorce.
- ❖ Med arhitekturnimi vzorci obstajajo povezave (podobnosti, generalizacije)
- ❖ Arhitekturne vzorce je možno kombinirati – več vzorcev se uporabi hkrati za določitev arhitekturne rešitve problema.

Dokumentiranje

- ❖ Dober arhitekturni model predstavi sistem iz več "pogledov" (ang. view):
 - ♦ logična delitev na podsisteme
 - ♦ vmesniki (ang. interfaces) med podsistemi
 - ♦ dinamika interakcije med komponentami v času obratovanja.
 - ♦ skupni podatki podsistemov
 - ♦ porazdelitev komponent/podsistemov po napravah.

Razvoj arhitekturnih modelov

- ❖ Modeliranje se začne s skiciranjem osnutka arhitekture
 - ♦ Glede na osnovne zahteve in primere uporabe.
 - ♦ Določitev osnovnih komponent sistema.
 - ♦ Izbira uporabljenih arhitekturnih vzorcev.
 - ♦ Koristno je, če več neodvisnih skupin ločeno pripravi začetne osnutke, nato pa se jih združi in uporabi najboljše ideje.

Razvoj arhitekturnih modelov

❖ Izboljševanje arhitekture

- ♦ Identifikacija interakcij med komponentami in določitev pripadajočih vmesnikov.
- ♦ Porazdelitev podatkov in funkcionalnosti med komponente sistema.
- ♦ Ugotavljanje možnosti za ponovno uporabo obstoječih ogrodij (ang. framework) oziroma odločitev za gradnjo novega ogrodja.

❖ Upoštevanje vsakega primera uporabe in prilagajanje arhitekture za njegovo izvedbo.

❖ Dokončanje arhitekture.

Opis arhitekture z UML

❖ Vsi UML diagrami so lahko koristni za opis različnih vidikov arhitekturnega modela.

❖ Za arhitekturne modele so še posebej pomembni naslednji UML diagrami:

- ♦ paketni diagrami (ang. package diagrams)
- ♦ komponentni diagrami (ang. component diagrams)
- ♦ diagrami postavitve (ang. deployment diagrams)

Načrt programske arhitekture

❖ Dokumentiranje arhitekture pripomore k boljšim arhitekturnim rešitvam:

- ♦ Dokumentiranje sili v to, da se pomembne odločitve sprejme pred začetkom implementacije.
- ♦ Omogoča preglede arhitekturnega načrta in s tem možnosti za dodatno izboljšavo.
- ♦ Olajša/omogoča/predstavlja komunikacijo
 - s tistimi, ki bodo sistem zgradili,
 - s tistimi, ki bodo v prihodnosti spreminjali arhitekturo sistema,
 - s tistimi, ki bodo gradili druge sisteme, ki se bo načrtovani sistem povezoval.

Struktura arhitekturnega načrta

❖ A. Namen:

- ♦ Na kateri sistem ali podsistem se nanaša arhitekturni načrt.
- ♦ Referenca na zahteve, ki jih upošteva (realizira) načrtan sistem.

❖ B. Upošteevane prioritete

- ♦ Opis načrtovalskih ciljev in prioritet, ki so bile uporabljene pri načrtovanju sistema.

❖ C. Povzetek arhitekturnega načrta:

- ♦ Splošen visoko nivojski opis arhitekturnega načrta, s katerim uporabnik okvirno spozna predlagano rešitev.

❖ D. Opis predlagane arhitekture:

- ♦ Opredelitev glavnih izzivov, ki jih je potrebno razrešiti.
- ♦ Opis alternativnih rešitev in odločitev o izbrani alternativni z obrazložitvijo izbire.

❖ E. Ostali detajli arhitekturnega načrta:

- ♦ Detajli, ki še niso bili omenjeni in so pomembni za bralca.

Napotki

- ❖ Izogibaj se navajanju informacij, ki so za izkušenega načrtovalca ali programerja očitne.
 - ♦ detajlom, ki bi bili bolj primerni za komentiranje programske kode.
 - ♦ detajlov o izvedbi posameznih podsistemov, ki niso pomembni za razumevanje sistema kot celote.
 - ♦ ... detajlom...

Recenzija arhitekturnega načrta

- ❖ Pregled in vrednotenje arhitekturnega načrta s strani kvalificiranega posameznika ali skupine...

Recenzija

- ❖ Za razliko od specifikacije zahtev pri arhitekturnem načrtu ni zunanjih recenzentov (stranke, uporabniki...).
- ❖ Recenzijo naj opravijo razvijalci, ki niso bili vključeni v načrtovanje.
- ❖ Recenzenti morajo imeti zadosten interes, za poglobljeno študijo predlagane rešitve.

Pravilnost arhitekturnega modela

- ❖ Arhitekturni model je pravilen, če omogoča preslikavo modela specifikacije zahtev v model sistema:
 - ♦ Ali je vsak podsistem mogoče povezati s primeri uporabe ali nefunkcionalnimi zahtevami?
 - ♦ Ali je vsak primer uporabe mogoče povezati z skupino podsistemov?
 - ♦ Ali je vsak načrtovalski cilj mogoče povezati z nefunkcijskimi zahtevami?
 - ♦ Ali so upoštevane vse nefunkcijske zahteve?
 - ♦ Ali je za vsakega akterja določen način nadzora dostopa?

Popolnost arhitekturnega modela

- ❖ Ali model zadosti vsem zahtevam?
 - ♦ Ali so bili upoštevani mejni primeri?
 - ♦ Ali model zadosti vsem definiranim primerom uporabe?
 - ♦ Ali so obravnavani vsi vidiki arhitekturnega načrtovanja: povezave HW/SW, shranjevanje podatkov, nadzor dostopa, mejni primeri)
 - ♦ Ali so definirani vsi podsistemi.

Konsistentnost arh. modela

- ❖ Ali so prisotna kakšna nasprotja?
 - ♦ Ali so nasprotujoči načrtovalski cilji urejeni po prioritetah?
 - ♦ Ali kateri načrtovalski cilji nasprotujejo nefunkcijskim zahtevam?
 - ♦ Ali ima morda več podsistemov ali razredov enako ime?

Realnost arhitekturnega modela

- ❖ Ali je sistem mogoče realizirati?
 - ♦ Ali so vključene nove tehnologije ali nove komponente? Ali so primerne in dovolj robustne za načrtovani sistem?
 - ♦ Ali so bile analizirane performančne zahteve in zahteve po zanesljivosti?
 - ♦ Ali so bile upoštevane potrebe po sočasnem izvajanju transakcij (ang. concurrency)?

Berljivost arhitekturnega modela

- ❖ Ali je model mogoče razumeti, tudi za ljudi, ki niso bili vključeni v modeliranje?
 - ♦ Ali so imena podsistemov razumljiva?
 - ♦ Ali objekti (podsistemi, razredi, operacije) s podobnimi imeni opisujejo podobne pojave?
 - ♦ Ali so vsi objekti opisani enako detajlno?

7 Načrtovanje komponent

V fazi načrtovanja komponent se posvetimo posameznim gradnikom programskega sistema. V primeru objektno orientiranega načrtovanja (OOD) gre tipično za načrtovanje objektov. Podrobno je potrebno vse njihove lastnosti s katerimi izpolnijo zahteve oziroma funkcionalnosti, ki so jim določene v fazi načrtovanja sistema, ter podrobno definirati njihove vmesnike. Za objektno orientirano načrtovanje je značilna še uporaba dedovanja, ki ga lahko koristimo za razvrščanje objektov v skupine ali za ponovno uporabo obstoječih objektov, ki jim je potrebno razširiti funkcionalnost ali spremeniti vmesnike. Dedovanje nam omogoča tudi uporabo abstrakcije, s katero dodatno povečamo prilagodljivost programskih produktov v smislu lažjih kasnejših sprememb in nadgradenj v času vzdrževanja.

Načrtovanje komponent nam lahko olajšajo načrtovalski vzorci; učinkovite in preverjene rešitve, ki pogosto niso samoumevne in omogočajo elegantno rešitev pogosto pojavljajočih se problemov. Nabor načrtovalskih vzorcev je velik, omenimo le tri izbrane, ki zadoščajo za demonstracijo široke uporabnosti načrtovalskih vzorcev in lahko bralca pritegnejo k študiju dodatne literature: Abstract factory, Adapter, Observer.

Umestitev načrtovanja

- ❖ **Analiza sistema** (specificiranje zahtev)
 - ♦ vključuje določitev programskih objektov
- ❖ **Načrtovanje sistema**
 - ♦ dekompozicija sistema, določitev globalnega kontrolnega toka, upravljanje perzistenčnih podatkov, HW/SW platforme...
 - ♦ Izbira obstoječih komponent (ponovna uporaba)
- ❖ **Načrtovanje programa (komponente, objektov)**
 - ♦ Načrtovanje uporabe obstoječih komponent (knjižnice, ovijanje, uporaba načrtovalskih vzorcev)
 - ♦ Zapolnitev vrzeli med aplikacijskimi objekti in izbranimi obstoječimi komponentami.
 - Identifikacija dodatnih objektov
 - Dopolnitve obstoječih načrtov objektov (razumljivost, razširljivost)
 - Natančno specificiranje vmesnikov posameznih razredov
 - Optimizacija načrta objektov (odzivnost, pomnilnik...)

Objektni in strukturni pristop

- ❖ **Objektno orientiran pristop (OOAD):**
 - ♦ Načrtovanje sistema
 - Dekompozicija na podsisteme
 - ♦ Načrtovanje objektov
 - Izbira implementacijskega jezika
 - Izbira algoritmov in določitev podatkovnih struktur
- ❖ **Strukturna analiza in načrtovanje (SAD):**
 - ♦ Predhodno načrtovanje (Preliminary Design)
 - Dekompozicija na podsisteme
 - Določitev podatkovnih struktur
 - ♦ Podrobno načrtovanje (Detailed Design)
 - Izbira algoritmov
 - Dopolnitev podatkovnih struktur
 - Izbira implementacijskega jezika
 - Običajno enotna aktivnost s predhodnim načrtovanjem

Načrtovanje programa

- ❖ Osredotočili se bomo na objektno orientiran pristop, načrtovanje objektov.
- ❖ Načrtovanje programa oz. objektov ni algoritmično!
 - ♦ Do izraza pridejo sposobnosti načrtovalca (znanje, izkušnje, ustvarjalnost...)
- ❖ Koraki načrtovanja objektov:
 - ♦ Uporaba obstoječih komponent
 - ♦ Specificiranje vmesnikov
 - ♦ Prestrukturiranje sistema
 - ♦ Optimizacija

Izbira komponent

- ❖ Izbira obstoječih rešitev:
 - ♦ programske knjižnice (off-the-shelf class libraries)
 - ♦ programska ogrodja (ang. frameworks)
 - ♦ komponente (ang. components)
- ❖ Prilagoditev obstoječih knjižnic, ogrodij in komponent
 - ♦ Sprememba API v primeru dostopnosti izvorne kode
 - ♦ "Adapter" ali "bridge" načrtovalski vzorec, če dostopa do izvorne kode ni.
- ❖ Načrtovanje novih komponent glede na podano programsko arhitekturo.

Komunikacija med objekti

- ❖ Konceptualno objekti komunicirajo s posredovanjem sporočil.
- ❖ Sporočila sestojijo iz:
 - ♦ Imena storitve, ki jo zahteva objekt "klicatelj" od klicanega objekta.
 - ♦ Informacij, potrebnih za izvedbo storitve.
- ❖ V praksi so sporočila pogosto realizirana v obliki klicev procedur.
 - ♦ Ime = ime procedure;
 - ♦ Informacije = parametri.

Primeri sporočil

```
// Call a method associated with a buffer  
// object that returns the next value  
// in the buffer
```

```
    v = circularBuffer.Get () ;
```

```
// Call the method associated with a  
// thermostat object that sets the  
// temperature to be maintained
```

```
    thermostat.setTemp (20) ;
```

Usmeritve pri načrtovanju

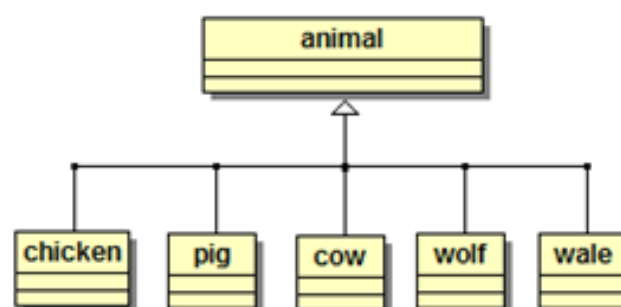
- ❖ Pri objektno orientiranem načrtovanju se poslužujemo različnih pristopov, med njimi so:
 - ♦ uporaba dedovanja,
 - ♦ delegiranje,
 - ♦ abstrakcija / ovijanje,
 - ♦ modularnost.

Uporaba dedovanja

❖ **Dedovanje (ang. inheritance) se uporablja za doseganje dveh različnih ciljev:**

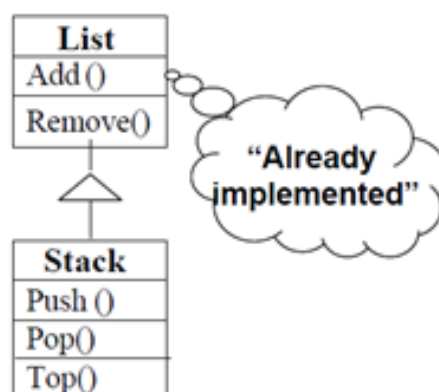
- ♦ **Razvrščanje razredov v skupine (taksonomija)**
 - Uporablja se v fazi analize zahtev
 - Za identifikacijo objektov, ki so medsebojno hierarhično odvisni.
 - Pripomore k razumljivosti modelov.
- ♦ **Ponovna uporaba**
 - Uporablja se v fazi načrtovanja objektov. □
 - Pripomore k možnosti uporabe obstoječih objektov ter možnosti njihovega spreminjanja in razširjanja
 - Razširitev funkcionalnosti (implementacijsko dedovanje),
Specifikacija vmesnikov (specifikacijsko dedovanje)

Primer dedovanja za razvrščanje



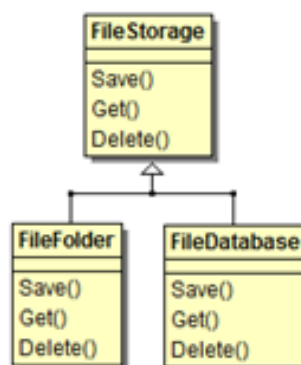
Implementacijsko dedovanje

- ❖ Uporaba obstoječega razreda, ki ima zelo podobno funkcionalnost.
- ❖ Primer:
 - ♦ Obstoječ razred **List** je mogoče uporabiti za izgradnjo razreda **Stack**.
 - ♦ Nove metode **Push()**, **Pop()**, **Top()** bodo zgrajene na osnovi obstoječe funkcionalnosti – metod **Add()** in **Remove()**.
- ❖ Pozor: metode osnovnega razreda so vedno na voljo tudi v razredu dediča...



Specifikacijsko dedovanje

- ❖ Podeduje se le vmesnik.
- ❖ Dedič implementira (na svoj način) metode, dodoločene z osnovnim razredom, ki določa vmesnik.
- ❖ Omogoča različno implementacijo z enakim vmesnikom – zunanjo funkcionalnostjo



Dedovanje za ponovno uporabo

- ❖ Implementacijsko dedovanje (ang. Implementation inheritance)
 - ♦ Namen je razširitev funkcionalnosti ob uporabi obstoječe funkcionalnosti.
- ❖ Specifikacijsko dedovanje (ang. Interface inheritance)
 - ♦ Namen je določiti vmesnik za uporabo različnih implementacij potrebne funkcionalnosti.

Delegiranje

- ❖ Delegiranje je način uporabe kompozicije za večkratno uporabo komponent
- ❖ Pri delegiranju zahtevo klienta sprejme prejemnik (receiver) in jo v ustrezni obliki posreduje delegatu (delegate).
 - ♦ Pri tem prejemnik poskrbi za ustrezno uporabo delegata in onemogoči neprimerno delo z delegatom.



Delegiranje in dedovanje

❖ Delegiranje

- ♦ + Fleksibilnost
- ♦ + Omogočeno za vse programske jezike
- ♦ - Slabša učinkovitost, zaradi ovijanja (encapsulation)

❖ Dedovanje

- ♦ + Enostavnost
- ♦ + Podprto v mnogih programskih jezikih
- ♦ + Enostavno dodajanje novih funkcionalnosti
- ♦ - Izpostavljenost detajlov osnovnega razreda
- ♦ - Vsaka sprememba osnovnega razreda zahteva spremembo dedovanega razreda

Minimiranje odvisosti

- ❖ Odvisnost je prisotna, če sprememba enega elementa programa lahko zahteva spremembo nekega drugega elementa programa.
- ❖ Odvisnost je glavni vir težav pri vzdrževanju programskih produktov.
- ❖ Pri načrtovanju težimo k čim manjši odvisnosti.

Primeri odvisnosti

❖ Dinamična odvisnost (ang. dynamic connascence):

- Odvisnost izvajanja – npr. pred uporabo potrebna inicializacija.
- Časovna odvisnost – npr. ukaz za zaustavitev obsevalne naprave mora slediti ukazu za vklop naprave v manj kot 50 milisekundah.
- Odvisnost vrednosti – npr. aritmetična omejitev – vsota vseh kotov trikotnika je vedno enaka 180 stopinj.
- Odvisnost identitete – npr. če se morata dva objekta (O1 in O2) sklicevati na isti objekt O3 (npr. s kazalci)
- Različnost – npr. če razred C deduje od razredov A in B, potem A in B ne smeta vsebovati metod z istim imenom.

❖ Statična...

- ...

Liskov princip zamenjave

- ❖ (ang. Liskov substitution principle -LSP) po Barbari Liskov in njenemu predavanju "*Data abstraction and hierarchy*" 1987.
- ❖ Razred S "ustreza" razredu T, če je mogoče vsak objekt razreda T nadomestiti z objektom razreda S, ne da bi pri tem ogrozili pravilnost delovanja programa.
- ❖ Če osnovni razred zadosti zahtevam aplikacije, potem mora tudi LSP nadomestni razred zadosti istim zahtevam, tako da s stališča vmesnika deluje povsem enako in da za enake argumente enake rezultate.
- ❖ Če želimo da je nek razred S podtip (ang. subclass) razreda T, mora S "ustrezati" T.

Abstrakcija

- ❖ Označuje bistvene lastnosti objekta, ki ta objekt ločijo od ostalih objektov.
- ❖ Določa konceptualne meje objektov, hkrati pa dopušča različno interpretacijo, odvisno od pogleda opazovalca.



Primer – prižiganje/ugašanje luči

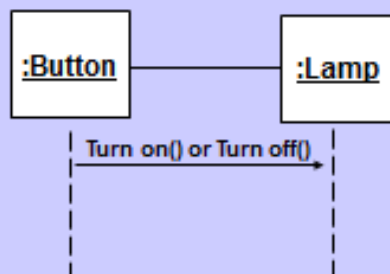
Primer uporabe:

Name: Turn lamp on and off

Actors: ButtonPusher

Description: When the ButtonPusher presses the button, the lamp goes on if it was off, and goes off if it was already on.

Sekvenčni diagram:

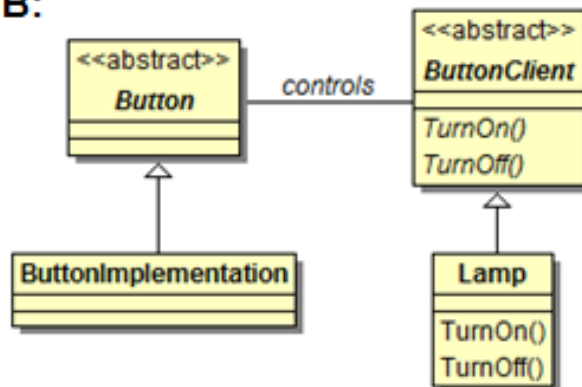


Primer – prižiganje/ugašanje luči

Rešitev A:



Rešitev B:



Primer – prižiganje/ugašanje luči

❖ Abstrakcija:

- ♦ "on/off" relacija med gumbom in ciljnim objektom
- ♦ Implementacija je ločena od abstrakcije

❖ Zakaj je rešitev B boljša?

- ♦ Izvedba gumba (button), npr. način interakcije uporabnika, ni pomembna za interakcijo gumba z lučjo.
- ♦ Gumbu (button) ni potrebno poznavanje luči (lamp).
- ♦ Rešitev omogoča lažje uvajanje sprememb.
- ♦ Rešitev je mogoče ponovno uporabiti za druge namene.
- ♦ Zahteve so še veno enostavno slednjive v izvedbi

Odprto/zaprti princip

- ❖ (ang. the open/closed principle)
- ❖ Razred naj bo odprt (open) za razširitve – dodajanje operacij ali polj podatkovne strukture
- ❖ Razred naj bo zaprt (closed) za spremembe, če je na voljo za uporabo drugim razredom – določati mora stabilen in dobro definiran vmesnik.
- ❖ V objektno orientiranih programskih jeziki je to realizirano z dedovanjem (ang. inheritance) in polimorfnostjo (ang. polymorphism).

Ovijanje (ang. encapsulation)

- ❖ Ovijanje je proces zaokrožanja elementov z določanjem abstrakcije, ki določa strukturo in obnašanje elementov.
- ❖ Ovijanje služi za ločevanje “pogodbenih” vmesnikov (določenih z abstrakcijo) od implementacije.

Kohezija razreda

- ❖ Kohezija razreda (ang. class cohesion) meri medsebično povezanost metod zunanjega vmesnika razreda.
- ❖ Kako zaključeno deluje razred v smislu implementacije abstrakcije.
- ❖ Želimo visoko kohezijo!
- ❖ Kvantitativna mera kohezije razreda ne obstaja.
- ❖ Simptomi nizke kohezije:
 - ♦ Razred vsebuje nekaj komponent (metod), ki niso definirane za vse objekte razreda. Eventuelno izboljšanje z ločitvijo v dva razreda.
 - ♦ Razred vsebuje komponente (metode), ki glede na namen razreda (domeno) niso relevantne.

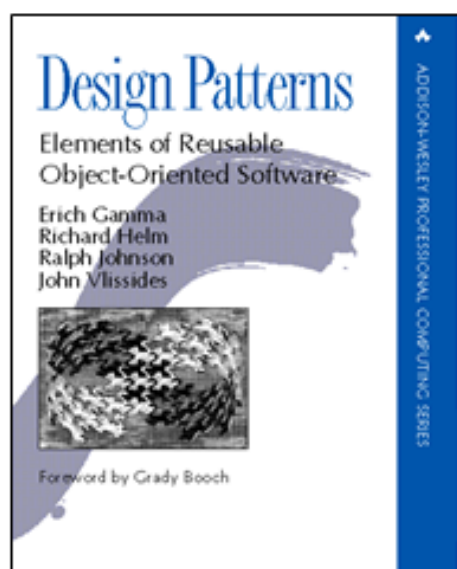
Splošno – modularnost!

- ❖ Modularen program sestoji iz dobro definiranih, konceptualno enostavnih in neodvisnih enot - modulov, ki komunicirajo preko dobro definiranih vmesnikov.
- ❖ Prednosti modularnosti
 - ♦ enostavnejše razumevanje in razlaganje,
 - ♦ enostavnejše dokumentiranje,
 - ♦ enostavnejše spreminjanje,
 - ♦ enostavnejše testiranje in razhroščevanje,
 - ♦ boljša večkratna uporabljivost,
 - ♦ enostavnejše nastavljanje.
- ❖ Modul je vsaka enota programa, ki sestoji iz več delov (program, podprogram, paket, razred, operacija, vrstica kode...)

Principi modularnosti

- ❖ Majhni moduli
 - ♦ Načrtovanje z majhnimi moduli je boljše.
- ❖ Skrivanje informacij (ang. information hiding)
 - ♦ Vsak modul naj drugim modulom prikrije detajle o njegovi interni strukturi in procesiranju.
- ❖ Najmanjši možni privilegiji
 - ♦ Moduli naj nimajo dostopa do nepotrebnih virov.
- ❖ Povezanost, odvisnost
 - ♦ Povezanost med moduli mora biti čim manjša
- ❖ Kohezija
 - ♦ Kohezija (odvisnost znotraj modula) mora biti čim večja.

Načrtovalski vzorci



- ❖ Uporaba objektnih tehnik modeliranja za predstavitev tipičnih rešitev pri načrtovanju programov.
- ❖ <http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612>

Načrtovalski vzorci

❖ Načrtovalski vzorci so:

- ♦ Elegantna rešitev, ki začetnikom ni samoumevna.
- ♦ Neodvisni od sistema, programskega jezika ali domene aplikacije.
- ♦ Dokatano uspešni v realnih OO sistemih.
- ♦ Enostavni, vsebujejo le nekaj razredov.
- ♦ Večkrat uporabni in na tak splošen način tudi dokumentirani.
- ♦ Objektno orientirani, uporabljajo OO mehanizme kot so razredi, objekti, generalizacija in polimorfizem.

Načrtovalski vzorci

❖ Določa jih:

- ♦ Ime vzorca
- ♦ Problem – načrtovalski vzorci vsebujejo opis problema, za katerega je primerno uporabiti vzorec.
- ♦ Rešitev – načrtovalski vzorci podajajo elemente in njihove medsebojne relacije, ki tvorijo rešitev problema.
- ♦ Posledice – rezultati in posledice uporabe načrtovalskega vzorca.

Delitev načrtovalskih vzorcev

❖ Glede na namen:

- ❖ **Kreacijski (ang. creational)**
 - Določajo proces kreiranja objektov
- ❖ **Strukturni (ang. structural)**
 - Določajo kompozicijo razredov in objektov
- ❖ **Vedenjski (ang. behavioural)**
 - Določajo način interakcije med razredi in porazdelitev odgovornosti.

Kreacijski načrtovalski vzorci

- ❖ **Abstract Factory** groups object factories that have a common theme.
- ❖ **Builder** constructs complex objects by separating construction and representation.
- ❖ **Factory Method** creates objects without specifying the exact class to create.
- ❖ **Prototype** creates objects by cloning an existing object.
- ❖ **Singleton** restricts object creation for a class to only one instance.

Strukturni načrtovalski vzorci

- ❖ **Adapter** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- ❖ **Bridge** decouples an abstraction from its implementation so that the two can vary independently.
- ❖ **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- ❖ **Decorator** dynamically adds/overrides behaviour in an existing method of an object.
- ❖ **Facade** provides a simplified interface to a large body of code.
- ❖ **Flyweight** reduces the cost of creating and manipulating a large number of similar objects.
- ❖ **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Vedenjski načrtovalski vzorci

- ❖ **Chain of responsibility** delegates commands to a chain of processing objects.
- ❖ **Command** creates objects which encapsulate actions and parameters.
- ❖ **Interpreter** implements a specialized language.
- ❖ **Iterator** accesses the elements of an object sequentially without exposing its underlying representation.
- ❖ **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- ❖ **Memento** provides the ability to restore an object to its previous state (undo).
- ❖ **Observer** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- ❖ **State** allows an object to alter its behavior when its internal state changes.
- ❖ **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime.
- ❖ **Template method** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- ❖ **Visitor** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Delitev načrtovalskih vzorcev

❖ Glede na področje delovanja:

♦ Razredni (class)

- Vzorec se ukvarja z razredi in njihovimi medsebojnimi relacijami, ki so določene z dedovanjem in so statične.

♦ Objektni (object)

- Vzorec se ukvarja z relacijami med objekti, ki so dinamične in se lahko spreminjajo v času izvajanja.

Abstract factory

- ❖ Abstract factory načrtovalski vzorec določa vmesnik za kreiranje družin med seboj odvisnih objektov, brez določitve njihovega dejanskega razreda.

❖ Umestitev:

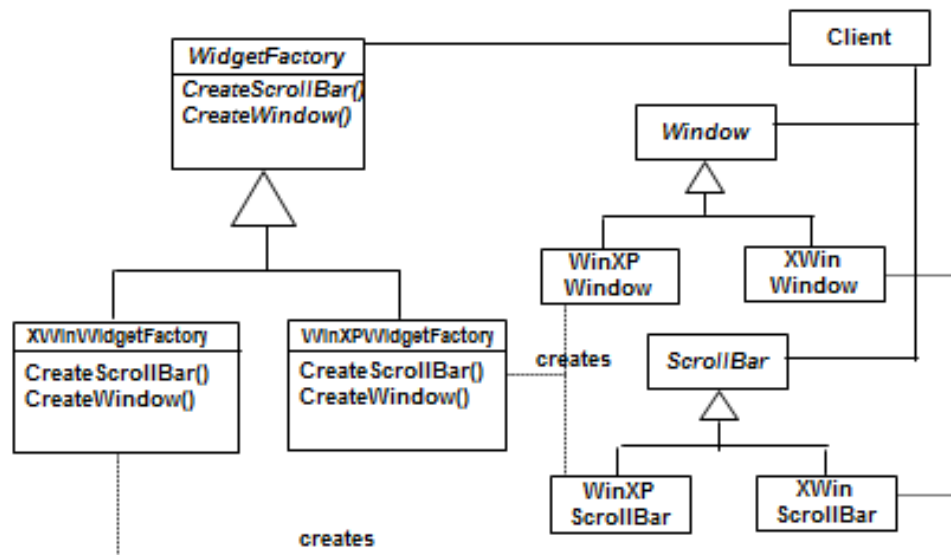
- ♦ Namen: krecijski načrtovalski vzorec
- ♦ Področje: objektni načrtovalski vzorec

❖ Primer:

- ♦ Želimo več različnih izgledov in vedenj (look and feel) za isti produkt. Npr. Windows XP in X Windows.

- ❖ Želimo se izogniti hard-kodiranju gradnikov (ang. widgets) vmesnika.

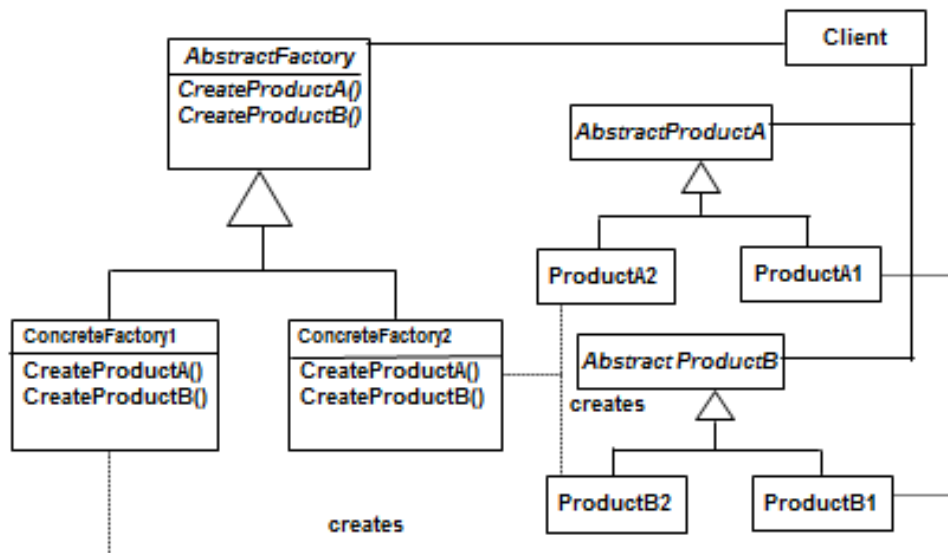
Abstract factory - primer



Abstract factory - uporaba

- ❖ Vzorec uporabimo v naslednjih situacijah:
 - ♦ Sistem mora biti konfigurabilen z različnimi družinami produktov.
 - ♦ Družina odvisnih produktov je načrtovana tako, da je uporabljena skupaj in to omejitve moramo zagotoviti.
 - ♦ Nuditi želimo knjižnico produktov vendar želimo razkriti le vmesnik in ne implementacij.

Abstract factory - struktura



Abstract factory - udeleženci

- ❖ **AbstractFactory**
 - ♦ Deklarira vmesnik za operacije, ki kreirajo objekte abstraktnih produktov.
- ❖ **ConcreteFactory**
 - ♦ Implementacija operacij za kreiranje objektov abstraktnih produktov.
- ❖ **AbstractProduct**
 - ♦ Deklarira vmesnik za tip objekta produkta.
- ❖ **ConcreteProduct**
 - ♦ Določa objekt produkta, ki naj bo kreiran s pripadajočo `ConcreteFactory` in implementira vmesnik za `AbstractProduct`.
- ❖ **Client**
 - ♦ Uporablja le vmesnike deklarirane z `AbstractFactory` in `AbstractProduct` razredi.

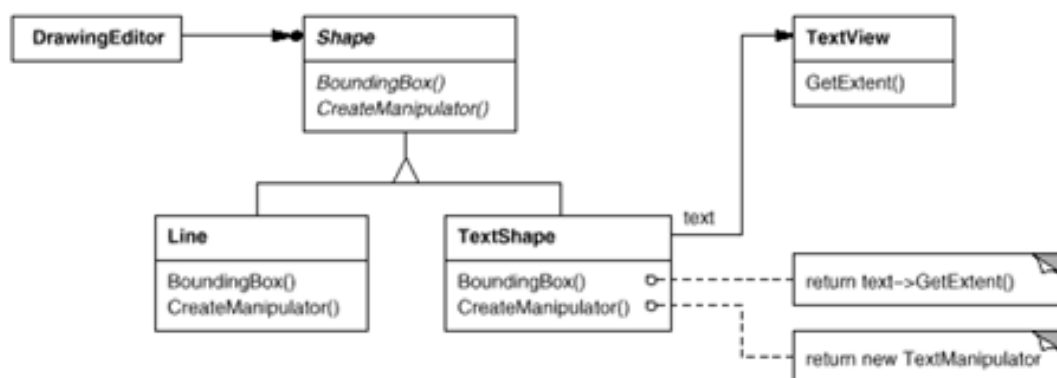
Abstract factory - rezultat

- ❖ Izolacija klientov od implementacij dejanskih razredov.
- ❖ Klienti manipulirajo z instancami preko abstraktnega vmesnika.
- ❖ Imena razredov produktov ne nastopajo v kodi klienta.
- ❖ Omogoča enostavno zamenjavo družin produktov – le z uporabo drugega razreda ConcreteFactory, ki se v kodi klienta pojavi le enkrat.
- ❖ Pripomore k konsistentnosti produktov glede na družine.
- ❖ Razširitev z novimi produkti je težavna, zahteva razširitev razreda AbstractFactory in vseh razredov ConcreteFactory.

Adapter

- ❖ Adapter načrtovalski vzorec pretvori vmesnik razreda v drug vmesnik, kakršnega potrebuje uporabnik in s tem omogoča sodelovanje razredov, ki sicer ni mogoče zaradi nezdružljivega vmesnika.
- ❖ Umestitev:
 - ♦ Namen: strukturni načrtovalski vzorec
 - ♦ Področje: razredni načrtovalski vzorec
- ❖ Primer:
 - ♦ Uporaba obstoječega razreda za izpis teksta v programu za urejanje slik, pri čemer vmesnik razreda ni skladen.
- ❖ Želimo omogočiti uporabo obstoječega razreda, četudi njegov vmesnik ni ustrezen.

Adapter - primer

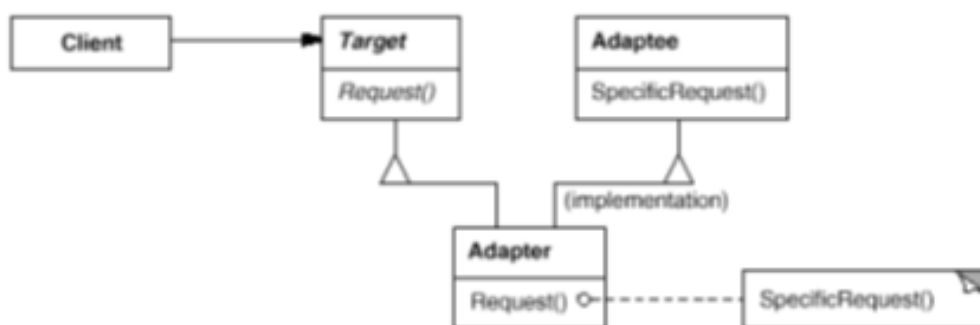


Adapter - uporaba

- ❖ Vzorec uporabimo v naslednjih situacijah:
 - ♦ Želimo uporabiti obstoječ razred, njegov vmesnik pa ne ustreza tistemu, ki ga potrebujemo.
 - ♦ Zgraditi želimo večkrat uporaben razred, ki ga lahko uporablja široka množica drugih razredov (tudi nepoznanih).

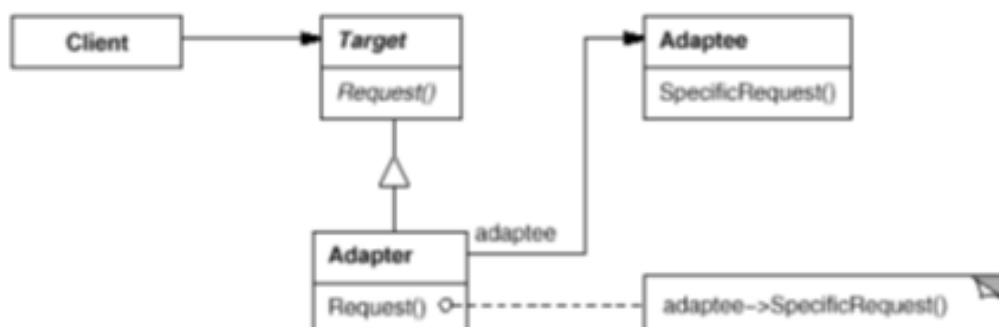
Adapter - struktura

❖ A: z uporabo večkratnega dedovanja



Adapter - struktura

❖ B: z objektno kompozicijo



Adapter - udeleženci

❖ Target

- ♦ Definira domensko specifičen vmesnik, ki ga uporablja objekt Client.

❖ Client

- ♦ Sodeluje z objekti po principu vmesnika Target

❖ Adaptee

- ♦ Določa obstoječi vmesnik, ki ga je potrebno prilagoditi.

❖ Adapter

- ♦ Prilagodi vmesnik objekta Adaptee vmesniku Target

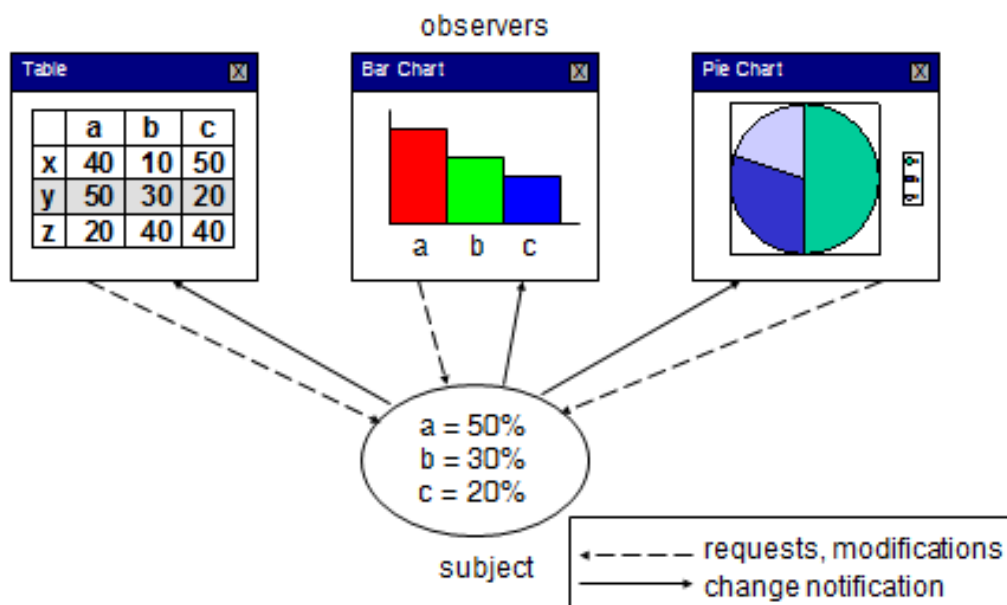
Adapter - rezultat

- ❖ Prilagoditev vmesnika Adaptee vmesniku Target. V primeru dedovanja ni mogoče prilagajati vmesnika z več podrazredi.
- ❖ Adapter lahko povozi (ang. override) lastnosti razreda Adaptee, če od njega deduje.
- ❖ V primeru objektne kompozicije lahko adapter prilagodi več razredov Adaptee hkrati, težje pa povozi njihove lastnosti.
- ❖ Možna je izvedba dvosmernega adapterja.

Observer

- ❖ Observer načrtovalski vzorec določa mehanizem za povezavo več objektov, tako da so vsi objekti obveščeni o spremembi stanja enega od objektov.
- ❖ Umetitev:
 - ♦ Namen: vedenjski načrtovalski vzorec
 - ♦ Področje: objektni načrtovalski vzorec
- ❖ Primer:
 - ♦ Prikaz istih podatkov v preglednici in v grafih.
- ❖ Želimo, da število odvisnih objektov ni omejeno in se lahko spreminja dinamično.

Observer - primer

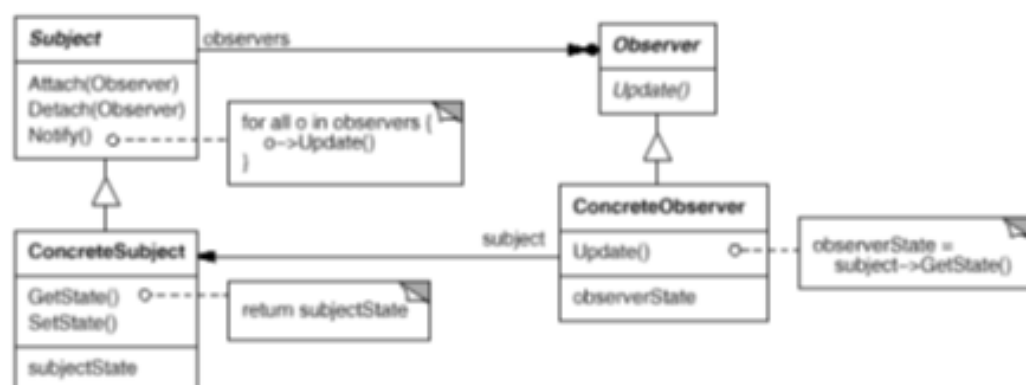


Observer - uporaba

❖ Vzorec uporabimo v naslednjih situacijah:

- ♦ Kadar ima abstrakcija dva vidika, ki sta med seboj odvisna. Ovijanje teh vidikov v ločene objekte omogoča njihovo neodvisno uporabo.
- ♦ Kadar sprememba enega objekta zahteva spremembo drugih objektov, katerih število ni vnaprej znano.
- ♦ Kadar mora biti objekt sposoben obveščati druge objekte brez poznavanja teh objektov in predpostavk o njih – ne želimo, da so objekti tesno povezani.

Observer - struktura



Observer - udeleženci

- ❖ Subject
 - ♦ Pozna objekte Observer, ki jih je lahko poljubno mnogo.
 - ♦ Nudi vmesnik za priklapljanje in odklapljanje objektov Observer.
- ❖ Observer
 - ♦ Določa vmesnik za informiranje objektov o spremembi objekta Subject.
- ❖ ConcreteSubject
 - ♦ Vsebuje stanje, kije pomembno za objekte ConcreteObserver
 - ♦ Pošlje informacijo o spremembi, ko do nje pride.
- ❖ ConcreteObserver
 - ♦ Vsebuje referenco na objekt ConcreteSubject
 - ♦ Hrani stanje, ki mora biti konsistentno s stanjem objekta ConcreteSubject.
 - ♦ Implementira vmesnik za posodabljanje s katerim vzdržuje konsistentno stanje z objektom ConcreteSubject.

Observer - rezultat

- ❖ Abstraktna in minimalna povezanost med objektoma Subject in Observer.
 - ♦ Subject se ne zaveda dejanskega razreda Observerja.
 - ♦ Subject in observer lahko zaradi šibke povezanosti pripadata različnim plastem.
- ❖ Podpora razpršenemu oddajanju.
 - ♦ Brez potrebe po medsebojnem poznavanju.
 - ♦ Sporočilo je posredovano vsem zainteresiranim (naročenim).
 - ♦ Možnost dinamičnega dodajanja ali odvzemanja observerjev.
- ❖ Navidez neškodljiva sprememba Subject-a lahko sproži plaz sprememb observerjev in odvisnih razredov!
- ❖ Slabo definirana odvisnost med razredi lahko vodi do lažnih posodabljanj.

Načrtovalski vzorci - splošno

- ❖ Obstaja bogat nabor načrtovalskih vzorcev.
- ❖ Vsak lahko definira svoje dodatne načrtovalske vzorce.
- ❖ Prikazani vzorci so predstavljeni z namenom boljše predstave o načrtovalskih vzorcih na splošno.
- ❖ Načrtovalski vzorci so uporabni, njihovo poznavanje pa lahko olajša načrtovanje programov in pripomore k kvaliteti.

Dokumentiranje: načrt programa

- ❖ Rezultat načrtovanja programa (načrtovanja objektov) je načrt programa, tudi detajlni načrt.
- ❖ Načrt programa (detajlni načrt) mora določati
 - ♦ notranjo zgradbo vseh delov programa,
 - ♦ njihovo povezanost (vmesniki)
 - ♦ njihovo strukturo (operacije, atributi, dedovanja, kompozicije)
 - ♦ način procesiranja,
 - ♦ algoritme,
 - ♦ podatkovne strukture,
 - ♦ lastnosti (prepustnost, zanesljivost...)
- ❖ Izdelava detajlnega načrta se pogosto deloma prekriva s kodiranjem.

Specifikacija vmesnikov

- ❖ Sintaksa (ang. *Syntax*)
 - ♦ Določa elemente komunikacijskega medija in način njihovega združevanja v sporočila.
- ❖ Semantika (ang. *Semantics*)
 - ♦ Določa pomen sporočil
- ❖ Pragmatika (ang. *Pragmatics*)
 - ♦ Določa kako so sporočila uporabljena za opravljanje nalog.

Recenzija!

- ❖ Podobno kot v primeru načrtovanja sistema:
 - ♦ poudarek na notranji zgradbi programov (modulov) in vmesnikih.

8 Testiranje

Testiranje mora v programskem procesu potekati v vseh fazah razvoja. Tako namreč lahko še pravočasno, brez dodatnih stroškov, odkrijemo kritične napake programskega produkta. Zgrešeno je mnenje, da se testiranje lahko začne šele po izvedbi, nasprotno, testiranje naj bo vključeno v vsak korak programskega procesa, z namenom čim prej odkriti nastale napake. Prav napake, ki se pojavijo zgodaj v programskem procesu, je najtežje odpraviti in so povezane z največjimi dodatnim stroški. Na primer, napaka pri analizi zahtev ima lahko za posledico reševanje »napačnega« problema ali neupoštevanje ključnih potreb uporabnikov. Če takšno napako ugotovimo šele ob validaciji ob koncu razvoja programskega produkta, je razvoj potrebno vrniti nazaj v fazo analize, kar pomeni ne le ponoven razvoj, pač pa tudi časovne zaostanke, nezadovoljstvo naročnikov ali uporabnikov ali celo odpoved uporabe produkta. Če pa napak sploh ne odkrijemo, so težave še toliko večje, med drugim tudi zato, ker odkritje napak s strani uporabnika zmanjšuje zaupanje do razvijalca programskega produkta.

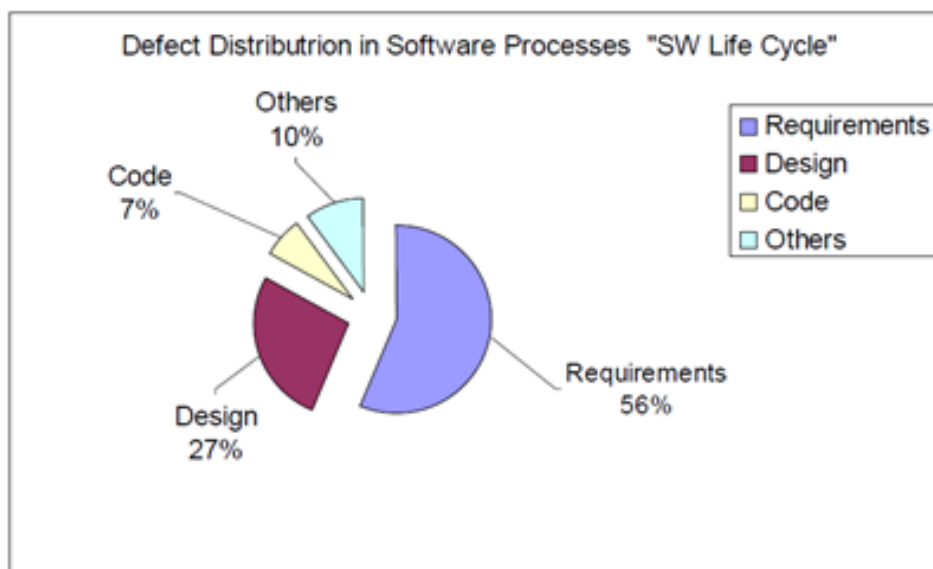
V splošnem testiranje delimo na verifikacijo, s katero preverjamo skladnost rezultatov z načrtanimi cilji, ter validacijo, s katero preverjamo ustreznost programskega produkta za zadovoljevanje uporabniških potreb. Med načini testiranja ločimo med postopki strukturnega testiranja, ki temeljijo na poznavanju izvedbe, ter postopki vedenjskega testiranja, ki temeljijo izključno na izsledkih obnašanja testiranega produkta/sistema. Možne so tudi različne strategije testiranja, glede na to kdaj in na kakšen način želimo preveriti posamezne dele programskega produkta oziroma programski produkt kot celoto. Testiranje moramo načrtovati, tako da vnaprej pripravimo testne scenarije s testnimi primeri. Poleg teh nam standard IEEE 829 priporoča še vrsto drugih dokumentov v postopku testiranja, s katerimi pokrивamo potrebe ob načrtovanju, izvajanju in poročanju o rezultatih testiranja.

Ko ne deluje...

Top 24 replies by programmers when their programs don't work:

- | | |
|--|---|
| 24. "It works fine on MY computer" | 12. "You must have the wrong version." |
| 23. "Who did you login as?" | 11. "It's just some unlucky coincidence." |
| 22. "It's a feature" | 10. "I can't test everything!" |
| 21. "It's WAD (Working As Designed)" | 9. "THIS can't be the source of THAT." |
| 20. "That's weird..." | 8. "It works, but it's not been tested." |
| 19. "It's never done that before." | 7. "Somebody must have changed my code." |
| 18. "It worked yesterday." | 6. "Did you check for a virus on your system?" |
| 17. "How is that possible?" | 5. "Even though it doesn't work, how does it feel?" |
| 16. "It must be a hardware problem." | 4. "You can't use that version on your system." |
| 15. "What did you type in wrong to get it to crash?" | 3. "Why do you want to do it that way?" |
| 14. "There is something funky in your data." | 2. "Where were you when the program blew up?" |
| 13. "I haven't touched that module in weeks!" | 1. "I thought I fixed that." |

Statistika distribucije napak



Zagotavljanje kakovosti

- ❖ Vsak korak programskega procesa se mora zaključiti z recenzijo oziroma testiranjem.
 - ♦ V tej fazi ugotavljamo ali rezultati koraka ustrezajo vhodnim zahtevam (verifikacija) ter ali so skladne s specifikacijo zahtev (validacija).
 - ♦ Zagotavljanje kakovost se začne že zgodaj v programskem procesu – že v začetnih fazah.
 - ♦ Intenzivnost zagotavljanja kakovosti se poveča v koraku izvedbe (kodiranje), z recenzijo izvorne kode in verifikacijo.
- ❖ Zagotavljanje kakovosti doseže vrhunec v koraku testiranja, z validacijo in sistemskim testiranjem.

Verifikacija in validacija

- ❖ Verifikacija
 - ♦ Posamezne komponente so testirane glede na skladnost s specifikacijami programa – načrta programa.
 - ♦ Posamezne komponente so združene v programski sistem in testirane glede na specifikacije sistema – načrta sistema.
- ❖ Validacija
 - ♦ Poteka po vnaprej pripravljenem načrtu, ki izhaja iz **specifikacij zahtev**. Preverja se skladnost sistema z zahtevami (funkcijskimi in nefunkcijskimi, upošteva tudi implicitne oziroma "samoumevne" zahteve).

Recenzije

- ❖ Prikaz strokovnega mnenja (ocena), sodbe o nekem (novem) delu, zlasti glede na kakovost.
- ❖ V programskem procesu kot zaključni del vsakega koraka programskega procesa, namenjen zagotavljanju kakovosti oziroma verifikaciji in validaciji.
- ❖ V koraku izvedbe govorimo o recenziji izvirne kode (ang. code review, code inspection).

Recenzija

- ❖ Gre za aktivnost pri kateri eden ali več ljudi sistematično pregleda izvirno kodo ali dokumentacijo z namenom iskanja morebitnih napak.
- ❖ Običajno recenzija vključuje sestanek, pri katerem sodelujejo tudi avtorji, čeprav recenzenti lahko pregled opravijo tudi samostojno oziroma ločeno.

Konstruktivnost in pozitivnost recenzij

- ❖ V mnogih organizacijah so recenzije nepriljubljeni dogodki:
 - ♦ Avtor ima pogosto občutek, da je namen recenzije spodbijanje njegovega mnenja.
 - ♦ Pogosto se ne razlikuje med pomembnimi in nepomembnimi vprašanji.
 - ♦ Drugi udeleženci lahko recenzijo izrabljajo za dokazovanje svojih "boljših" sposobnosti.
 - ♦ ...
- ❖ Recenzije ne smejo biti uperjene proti avtorju, biti morajo konstruktivne in pozitivne.

Namen recenzij

- ❖ Način zagotavljanja kakovosti, s katerim se zagotavlja kakovost recenziranega izdelka (npr. izvorne kode). Pregledovalci lahko učinkovito najdejo različna neskladja, od odstopanja poslovnih procesov do enostavnih napak v izvorni kodi.
- ❖ Orodje za poučevanje, s katerim razvijalci izmenjujejo znanja o tehnikah izboljševanja kvalitete (npr. izvorne kode), konsistentnosti, enostavnosti vzdrževanja produktov in uporabe različnih pristopov.

Pozitiven pristop recenzentov

- ❖ Sprašuj namesto da grajaš.
 - ♦ Npr. namesto "Tu nisi upošteval standardov!" uporabi "Kakšen je razlog za uporabo uporabljenega pristopa?"
- ❖ Izogibaj se vprašanjem tipa "Zakaj".
 - ♦ Npr. namesto "Zakaj tukaj nisi upošteval standardov?" uporabi "Kakšen je razlog za odstopanje od standardov..."
- ❖ Ne pozabi na pohvalo.
 - ♦ Pomembno je opaziti tudi pozitivno, npr. kreativne rešitve. To avtorju da vedeti, da njegovo delo vidimo širše kot le vrsto napak.
- ❖ Pri stilih kodiranja upoštevaj dogovorjene standarde.
 - ♦ Standardi kodiranja morajo biti sprejeti kot dogovor med razvijalci. Če gre za vprašanje, ki še ni dogovorjeno, ga je primerno dodatno vključiti v standard.
- ❖ Tema recenzije naj bo delo in ne avtor.
 - ♦ Namen recenzije je prispevek k kvaliteti in ne opredeljevanje o lastnostih in sposobnostih avtorja.
- ❖ Zavedaj se, da obstaja več kot ena možna (dobra) rešitev.
 - ♦ Čeprav je avtor nekaj naredil drugače kot bi ti, to ni nujno narobe.

Pozitiven pristop avtorjev

- ❖ Zavedaj se, da recenzirano delo nisi ti.
 - ♦ Razvoj je kreativen proces in normalno je, da si na svoje delo navezan. Vendar pa recenzenti ne želijo povedati, da si nisi dober razvijalec (ali oseba). Njihova naloga je pokazati na pomanjkljivosti in možnosti boljših rešitev. Tudi, če dela ne opravljajo dobro, ne zavzemi obrambnega položaja in prisluhni konstruktivnim pripombam (morebitne napade pa presliži).
- ❖ Naredi seznam stvari na katere se bodo osredotočili recenzenti.
 - ♦ S seznamom ponovno preglej svoje delo in popravi morebitne najdene napake. S tem boš ne le zmanjšal število najdenih napak, pač a tudi skrajšal čas revizije (v veselje vsem udeleženi).
- ❖ Pomagaj vzdrževati standarde.
 - ♦ Ponudi dopolnitev standardov (npr. kodiranja) za tisto, kar je bilo dogovorjeno in ni vključeno v obstoječe standarde. Na ta način dokumentirana rešitev bo v pomoč pri naslednjih delih in recenzijah.

Dokumentiranje recenzij

Pri recenzijah, ki so seznam komentarjev, upoštevaj naslednje napotke:

❖ **Začni s povzetkom in bodi opziten.**

- ♦ Na ta način avtorju pokažeš svoje razporeženje ter omehčaš komentarje, ki sledijo. Komentar mora vsebovati pozitivna sporočila avtorju, saj prav vsako delo vsebuje nekaj dobrega, kar je vredno omeniti (četudi je to le sintaktična pravilnost). Pozitivna naravnost je bistvena.

❖ **Uporabljal elektronske mehanizme za beleženje komentarjev.**

- ♦ Npr. kopiranje v word in uporaba wordovih funkcij za komentiranje. Na ta način nisi omejen z dolžino komentarjev in imaš možnost podrobneje pojasniti svoje stališče ter komentarje lažje formulirati v vprašalni obliki.

❖ **Skleni dogovor, da avtorju ni potrebno odgovoriti na vsa vprašanja.**

- ♦ Dobre recenzije vsebujejo tudi razmišljujoča vprašanja, kot npr. "Ali ne bi bilo bolje uporabiti X načrtovalskega vzorca?". Takšna vprašanja ne potrebujejo odgovora, saj so namenjena izključno spodbujanju razmišljanja. Takšna vprašanja nimajo bistvenega vpliva na obstoječo rešitev, imajo pa pozitiven dolgoročen vpliv na kakovost produktov.

Principi

- ❖ Recenzije opravi za najpomembnejše dokumente (specifikacijo zahtev, načrte sistema in programov, načrte testiranja) in izvorno kodo.
- ❖ Učinkovita skupina recenzentov naj sestoji iz 2 do 5 članov in naj vključuje izkušene razvijalce.
- ❖ Udeleženci se naj na recenzijski sestanek pripravijo.
- ❖ Recenziraj le dokončane dokumente.
- ❖ Izogibaj se diskusijam o načinu izvedbe popravkov.
- ❖ Ne hiti – pri recenziji izvorne kode je dobra hitrost 200 vrstic na uro.
- ❖ Recenzija naj ne traja več kot dve uri skupaj in štiri ure dnevno.
- ❖ Ponovna recenzija naj se izvede, če je spremenjeno več kot 20% dela.
- ❖ Vodstvo naj ne bo vključeno v recenzije, ker to negativno vpliva na odprtost ostalih recenzentov.
- ❖ Nikogar se ne sme grajati.

Primerjava recenzij in testiranj

- ❖ Recenzije in testiranje temeljijo na različnih načinih preverjanja izdelka.
- ❖ Recenzije omogočajo iskanje napak, ki vplivajo na možnosti vzdrževanja in učinkovitost.
- ❖ Testiranje omogoča iskanje napak v kompleksnih rešitvah (napaka globoko v kodi) in imajo jasne posledice.
- ❖ Možnosti napak so manjše kadar se uporabljata tako recenzija kot tudi testiranje.
- ❖ Recenzije naj potekajo pred kakršnim koli testiranjem.

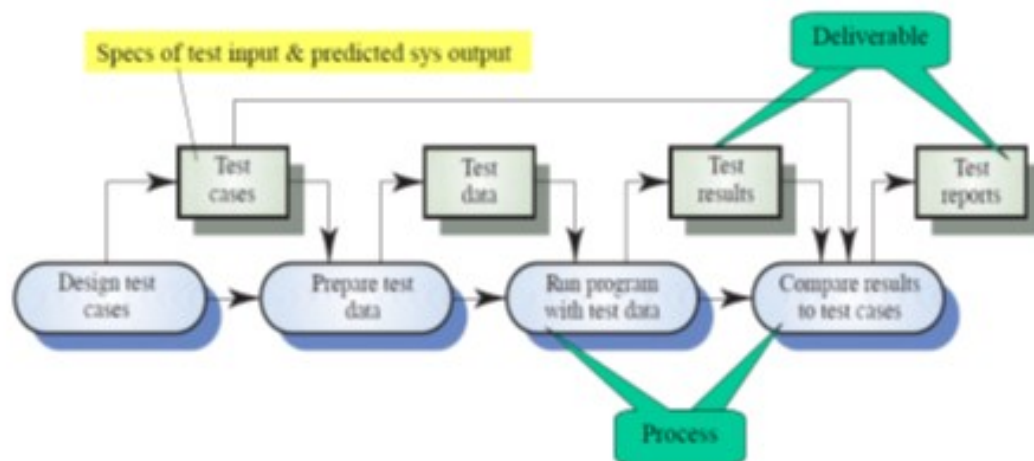
Testiranje

- ❖ Testiranje je proces izvajanja programa z namenom iskanja napak (error) pred predajo končnim uporabnikom.
- ❖ Preizkuševalec (tester) izvede serijo testnih primerov z namenom porušiti pravilno delovanje programa.
- ❖ Dober testni primer je tisti, za katerega obstaja velika verjetnost da bo razkril še ne odkrito napako.
- ❖ Test je uspešen, če je z njim najdena prej nepoznana napaka.

Testiranje ☺

- ❖ *To the optimist, the glass is half full.*
- ❖ *To the pessimist, the glass is half empty.*
- ❖ *To the good tester, the glass is twice as big as it needs to be.*

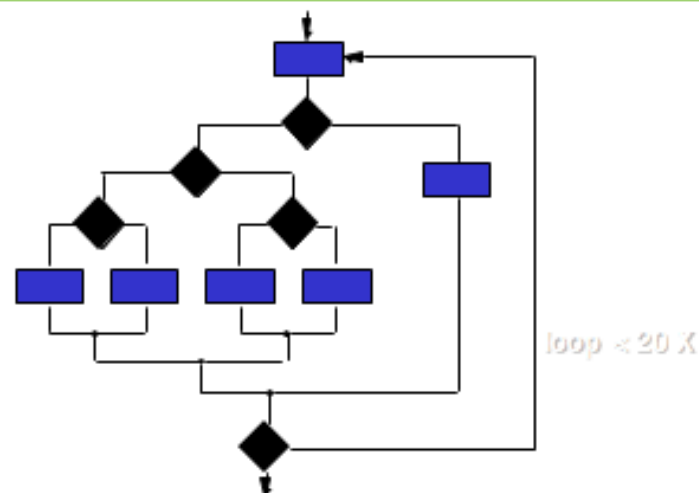
Potek testa



Principi testiranja

- ❖ Vsi testni primeri naj bodo povezani z zahtevami.
 - ♦ Najhujše napake so tiste, ki povzročijo, da program ne izpolnjuje zahtev.
- ❖ Testiranje naj bo načrtovano pred začetkom testiranja.
 - ♦ Načrtovanje testiranja se lahko začne takoj ko so izdelane specifikacije zahtev.
- ❖ Pri testiranju velja zakon 80-20 (princip Pareto).
 - ♦ 80% vseh napak odkritih med testiranjem izvira iz 20% programskih komponent. Te komponente je potrebno testirati toliko bolj temeljito.
- ❖ Izčrpno testiranje ni mogoče.
 - ♦ Število možnih poti izvajanja programa je že pri manjših programih lahko prevelika, da bi lahko preizkusili vse možnosti.
- ❖ Najučinkovitejše testiranje opravijo neodvisni preizkuševalci
 - ♦ Razvijalec, ki je program naredil, ni najprimernejši preizkuševalec zaradi navzkrižja interesov.

Izčrpnost testiranja



- ❖ Prisotnih je 10^{14} možnih poti.
- ❖ Če za izvajanje vsake potrebujemo 1ms bi testiranje programa trajalo 3170 let!
- ❖ Nujno je selektivno testiranje!

Kdo testira?

❖ Razvojni inženir:

- ♦ Dobro razume sistem.
- ♦ Glavni interes je pravočasna predaja produkta.
- ♦ Je konstruktiven in pri testiranju do sistema "nežen".

❖ Neodvisni preizkuševalec:

- ♦ Ne pozna sistema in ga mora še spoznati.
- ♦ Glavni interes je zagotavljanje kakovosti.
- ♦ Je destruktiven in bo iskal načine za zlom sistema.

Preverljivost (ang. testability)

- ❖ Preverljivost ali testabilnost je lastnost, ki pove kako enostavno je testirati program.
- ❖ Dobra testabilnost programa naj bi bila eden od ciljev pri načrtovanju.
- ❖ Testabilnost je odvisna od več drugih lastnosti programa:
 - ♦ **Izvedljivost** (ang. operability) – program je mogoče neovirano izvajati.
 - ♦ **Možnost opazovanja** (ang. observability) – program nudi jasen vpogled v rezultate izvajanja.
 - ♦ **Možnost nadzora** (ang. controllability) – stopnja do katere je testiranje mogoče avtomatizirati in optimirati.
 - ♦ **Razstavljivost** (ang. decomposability) – testiranje je lahko osredotočeno na posamezen modul.
 - ♦ **Enostavnost** (ang. simplicity) – enostavnost programa omogoča temeljitejše testiranje.
 - ♦ **Stabilnost** (ang. stability) – stopnja spreminjanja med testiranjem.
 - ♦ **Razumljivost** (ang. understandability) – razumevanje delovanja programa, tudi na osnovi dokumentacije.

Načini testiranja

❖ Strukturno testiranje (white-box testing)

- ♦ Testiranje temelji na poznavanju notranje strukture programa.

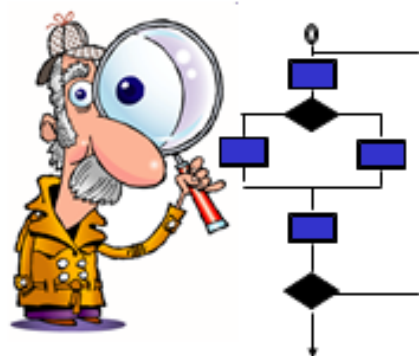


❖ Vedenjsko testiranje (black-box testing)

- ♦ Testiranje temelji na (deklariranih) zunanjih lastnostih sistema.



Strukturno testiranje (white-box)



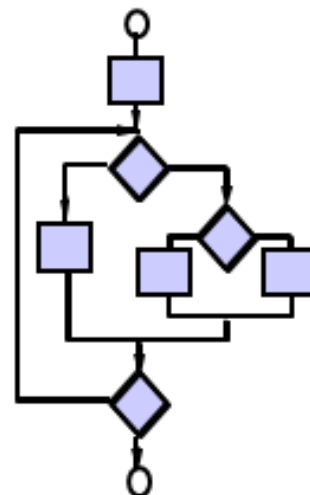
- ❖ Cilj je zagotoviti, da je bil vsak pogoj upoštevan vsaj enkrat in vsak del izvirne kode izveden vsaj enkrat.

Upoštevanje celotne strukture

- ❖ Pri testiranju preveri celotno strukturo programa – vse poti izvajanja programa.
 - ♦ Logične napake in napačne predpostavke so inverzno sorazmerne verjetnosti izvajanja poti.
 - ♦ Pogosto verjamemo, da ni verjetno, da bo določena pot izvedena. Realnost je pogosto v nasprotju z intuicijo.
 - ♦ Tipkarske napake se pojavljajo naključno. Verjetno je, da bodo prisotne tudi na netestiranih poteh.

Testiranje osnovnih poti

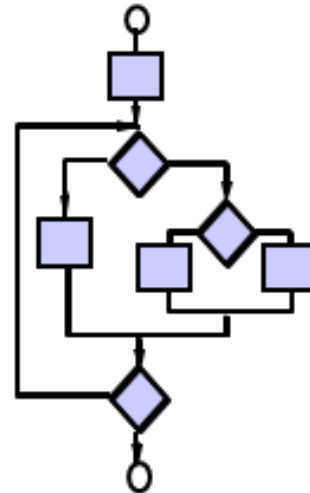
- ❖ **Neodvisna pot** je pot skozi program, ki vsebuje vsaj en nov del programske kode ali nov pogoj.
 - ♦ Vsaj en del poti ni vsebovan v drugih neodvisnih poteh.
- ❖ Število neodvisnih poti je odvisno od **ciklometrične kompleksnosti** - mere logične kompleksnosti programa.



Testiranje osnovnih poti

❖ Izračun ciklometrične kompleksnosti $V(G)$ na osnovi grafa izvajanja G :

- ♦ $V(G) = \text{število enostavnih odločitev} + 1$
- ♦ $V(G) = \text{število vsebovanih področij grafa} + 1$
- ♦ Za dani primer: $V(G) = 3 + 1 = 4$



Testiranje osnovnih poti

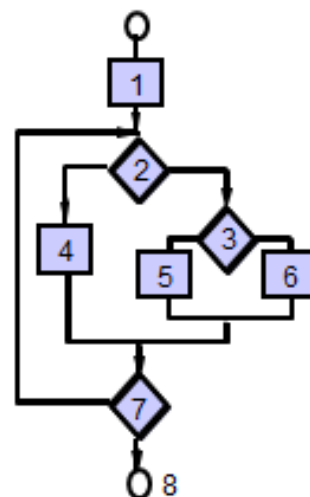
❖ Določitev neodvisnih poti. $V(G) = 4$; določimo 4 osnovne poti:

- ♦ Pot1: 1,2,4,7,8
- ♦ Pot2: 1,2,3,5,7,8
- ♦ Pot3: 1,2,3,6,7,8
- ♦ Pot4: 1,2,4,7,2,4,7,8

❖ Vaka dodatna pot je kombinacija obstoječih poti in torej ni neodvisna.

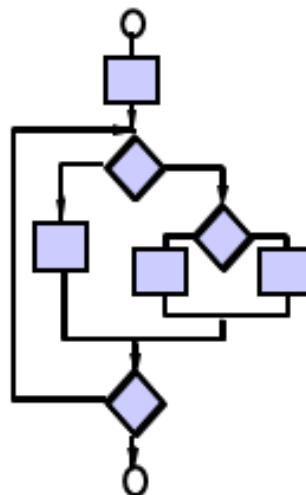
- ♦ 1,2,3,6,7,2,4,7,8 - kombinacija poti 3 in 4.

❖ Osnovne poti določajo testne primere za strukturno testiranje.



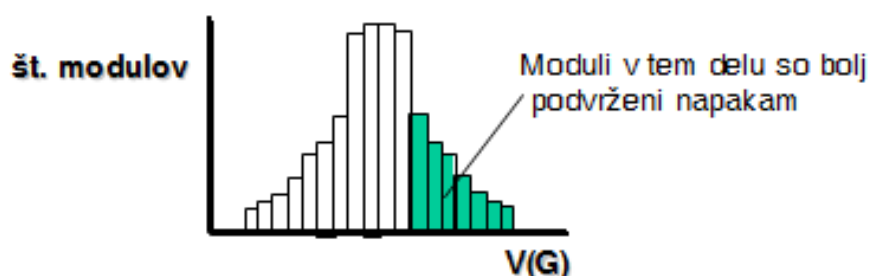
Testiranje osnovnih poti

- ❖ Za načrtovanje osnovnih poti ne potrebuješ grafa, res pa ta olajša sledenje potem.
- ❖ Klikomatično kompleksnost določiš s štejem enostavnih logičnih odločitev. Kompleksne odločitve se šteje za dve ali več enostavnih.
- ❖ Testiranje osnovnih poti je posebno pomembno pri kritičnih modulih.

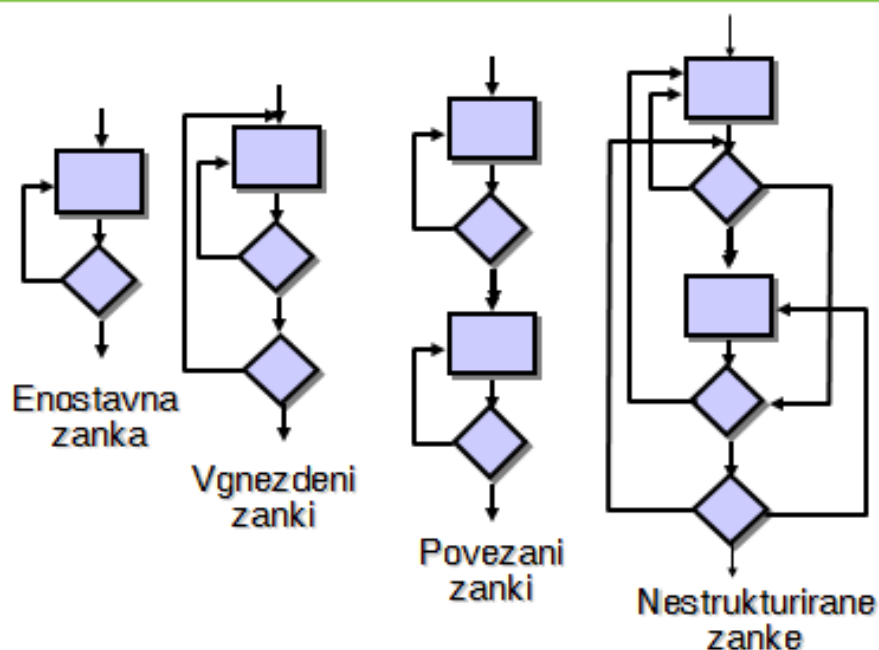


Ciklomatična kompleksnost

- ❖ Raziskave v industriji kažejo, da je verjetnost napak pozitivno povezana s ciklomatično kompleksnostjo $V(G)$.



Testiranje zank



Testiranje enostavnih zank

- ❖ Sklop testnih primerov naj vsebuje:
 - ♦ Izpustitev zanke
 - ♦ En sam prehod skozi zanko
 - ♦ Dva prehoda skozi zanko
 - ♦ m prehodov skozi zanko, kjer je $m < n$
 - ♦ $n-1$, n , $n+1$ prehodov skozi zanko

Testiranje vgnezenih zank

- ❖ Začni z najbolj notranjo zanko. Vse ostale zanke naj se izvedejo z najmanjšim številom iteracij.
- ❖ Zanko testiraj za $\text{min}+1$, $\text{max}-1$ in tipičnim številom iteracij.
- ❖ Premakni se na za en nivo bolj zunanjo zanko in jo testiraj na enak način. Tako nadaljuj dokler ne zaključiš z najbolj zunanjo zanko.

Testiranje povezanih zank

- ❖ Če so zanke neodvisne, potem vsako obravnavaj kot enostavno zanko
- ❖ Če zanke niso neodvisne, jih obravnavaj kot vgnezdene zanke.
 - ♦ Primer odvisnosti: končni števec ene zanke vpliva na inicializacijo naslednje zanke.

Testiranje pogojev

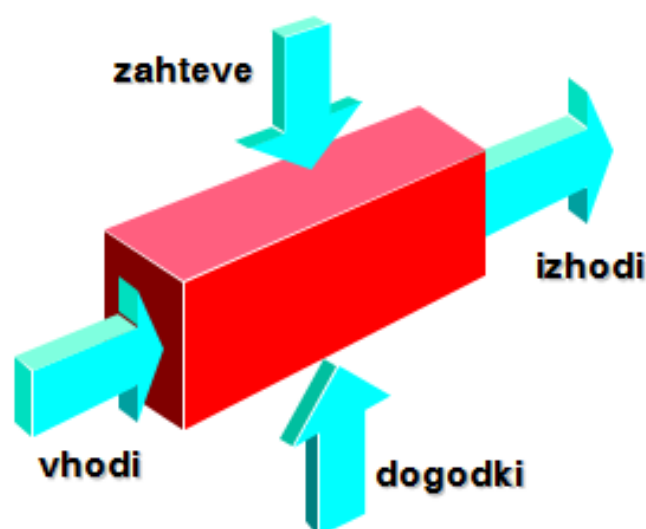
- ❖ Vsaka odločitev se izvede na osnovi pogoja.
- ❖ Testiranje pogojev temelji na izbiri takšnih testnih primerov, da se preveri pravilnost odločitev glede na odločitvene parametre.
- ❖ Možne so naslednje napake:
 - ♦ Napaka logičnega operatorja (neustrezen, manjkajoč, odvečen)
 - ♦ Napaka logične spremenljivke
 - ♦ Napaka v oklepajih logičnega izraza
 - ♦ Napaka v relacijskem operatorju ($>$, $<$, $<=$, $>=$, $==$, $!=$)
 - ♦ Napaka v aritmetičnem operatorju
- ❖ Testni primeri za testiranje pogojev se deloma prekrivajo s testnimi primeri testiranja osnovnih poti.

Vedenjsko testiranje (black-box)



- ❖ Za razliko od strukturnega testiranja (white-box) se vedenjsko testiranje (black-box) uporablja v kasnejših korakih procesa testiranja.
- ❖ Ne osredotoča se na kontrolni tok, pač pa na obdelovane informacije.
- ❖ Načrtovanje testnih primerov temelji na naslednjih vprašanjih:
 - ♦ Kako preveriti funkcionalno pravilnost?
 - ♦ Kako preveriti obnašanje sistema in prepustnost?
 - ♦ Kateri vhodni podatki tvorijo dobre testne primere?
 - ♦ Ali je sistem občutljiv na določene vhodne vrednosti?
 - ♦ Kako so ločene meje podatkovnih razredov?
 - ♦ Kakšen obseg podatkov lahko tolerira sistem?
 - ♦ Kakšen vpliv na delovanje sistema imajo specifične kombinacije podatkov?

Vedenjsko testiranje (black-box)



Ekvivalentne particije

- ❖ Ekvivalentne particije (ang. equivalence partitioning) je postopek za določanje razredov vhodnih podatkov iz katerih izberemo testne podatke.
- ❖ Ekvivalentni razred (ang. equivalence class) je skupina veljavnih ali neveljavnih vhodnih pogojev.
- ❖ Glede na določitev vhodnih pogojev določimo ekvivalentne razrede na naslednji način:
 - Vhodni pogoj je določen s **področjem** – določi en veljaven in dva neveljavna ekvivalentna razreda.
 - Vhodni pogoj je specifična **vrednost** - določi en veljaven in dva neveljavna ekvivalentna razreda.
 - Vhodni pogoj je določen s pripadnostjo **množici** - določi en veljaven in en neveljaven ekvivalentni razred.
 - Vhodni pogoj je **logična vrednost** (ang. boolean) - določi en veljaven in en neveljaven ekvivalentni razred.

Analiza mejnih vrednosti

- ❖ Analiza mejnih vrednosti (ang. boundary value analysis - BVA) vodi do izbire testnih primerov, ki temeljijo na mejnih vhodnih podatkih.
 - ♦ Npr: vhodni pogoj je določen s področjem omejenim z a in b , – testni primeri naj uporabijo vrednosti a in b ter malenkost nižje in višje vrednosti.
 - ♦ ...
- ❖ Izbira z BVA ni vezana le na vhodne podatke, pač pa tudi na izhodne podatke!

Druge tehnike vedenj. testiranja

- ❖ Ugibanje napak
 - ♦ (ang. error guessing methods)
- ❖ Odločitvene tabele
 - ♦ (ang. decision table techniques)
- ❖ Grafi vzroka in posledice
 - ♦ (ang. cause effect graphing)

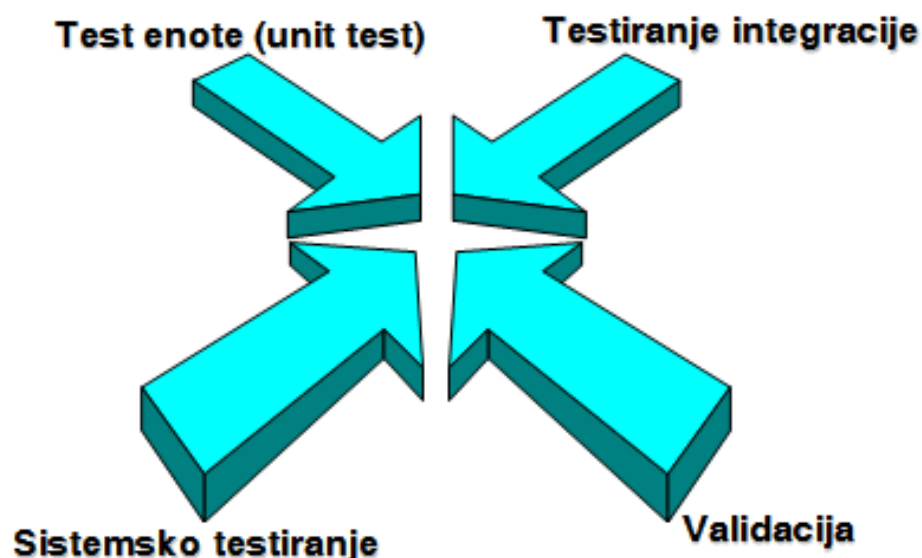
Strategije testiranja

- ❖ V literaturi je bilo predlaganih več strategij testiranja. Skupne so jim naslednje lastnosti:
 - ♦ Testiranje poteka od spodaj navzgor. Začne se na nivoju komponent (npr. razredi, objekti) in se nadaljuje z obravnavanjem vse večjih modulov ter zaključi s testiranjem celotnega sistema.
 - ♦ Različne tehnike so primerne za različne faze testiranja. Tipično strukturno testiranje (white-box) poteka pred vedenjskim (black-box).
 - ♦ Testiranje opravi razvijalec programa in (predvsem za večje projekte) neodvisna testna skupina.
 - ♦ Testiranje in razhroščevanje sta različni in ločeni aktivnosti. Tudi razhroščevanje mora biti vključeno v testno strategijo.

Neodvisna testna skupina

- ❖ Razvijalec tipično želi s testi demonstrirati kakovost svojega produkta (neprisotnost napak, skladnost z zahtevami...).
- ❖ Namen testiranja je iskanje napak in ne demonstracija kakovosti.
- ❖ Razvijalec ni najprimernejša testna oseba
 - ♦ Obstaja konflikt interesov
 - ♦ Detajlno poznavanje sistema pomeni usmerjen pogled na sistem, ki je bistveno drugačen od miselnih modelov uporabnika.
- ❖ Neodvisna testna skupina teh omejitev nima
 - ♦ Vendar pa mora razvijalec pri testiranju sodelovati s pomočjo pri vzpostavljanju okolja in odpravljanju ugotovljenih napak.

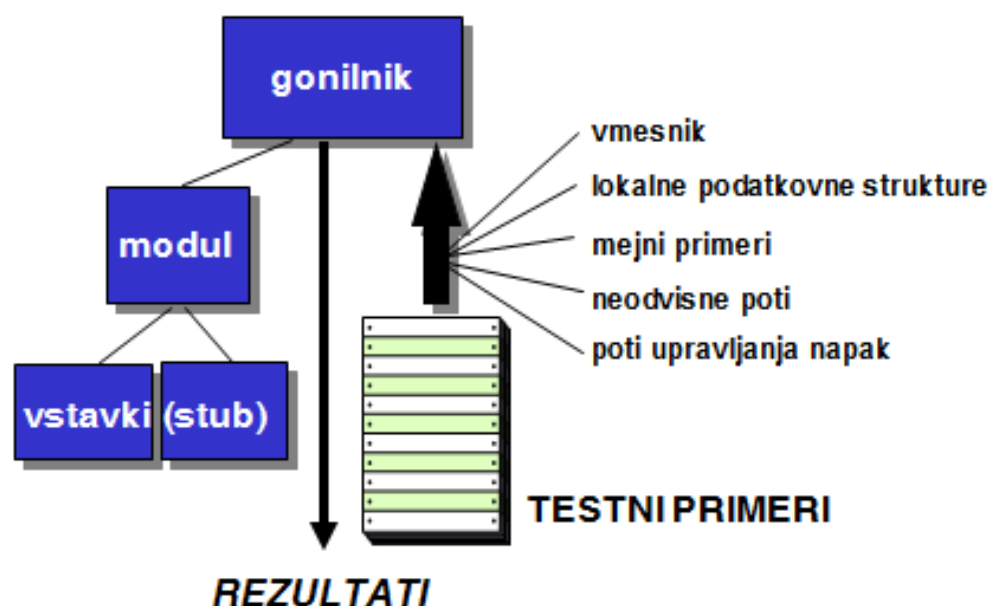
Testna strategija



Test enote (ang. unit test)

- ❖ Obsega testiranje posameznih komponent oziroma enot (ang. unit).
- ❖ Testiranje temelji na izvorni kodi (strukturno testiranje / white-box)
- ❖ Tipično je test enote odgovornost razvijalca.
- ❖ Izvedba testa je odvisna od izkušenj razvijalca.
- ❖ Za testiranje enote pripravimo testni gonilnik (ang. driver).
- ❖ Če enota uporablja druge enote, le te v čim večji meri nadomestimo z nastavki (nadomestna minimalna implementacija enote) – (ang. stub).
- ❖ Za teste enot je na voljo vrsta ogrodij (ang. framework), glej: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Okolje testa enote



Testiranje integracije

- ❖ Testiranje integracije (ang. integration testing) je namenjeno testiranju skupine komponent, ki so združene v sistem ali podsistem.
- ❖ Odgovornost razvijalcev ali (bolje) neodvisne testne skupine.
- ❖ Testiranje temelji na specifikacijah sistema (zahteve, načrt sistema) – vedenjsko testiranje (black-box).
- ❖ Glavna težava je lokalizacija napak, ki jo je mogoče reševati z inkrementalno integracijo.

Strategije integracijskega testiranja

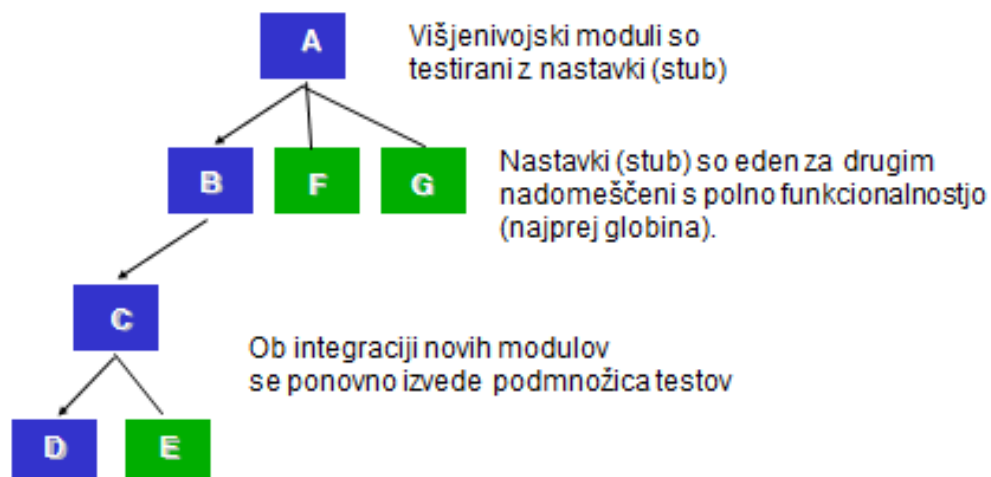
- ❖ Pristop velikega poka (ang. the "big bang" approach)
- ❖ Pristop inkrementalne integracije – postopne gradnje (ang. incremental construction strategy)



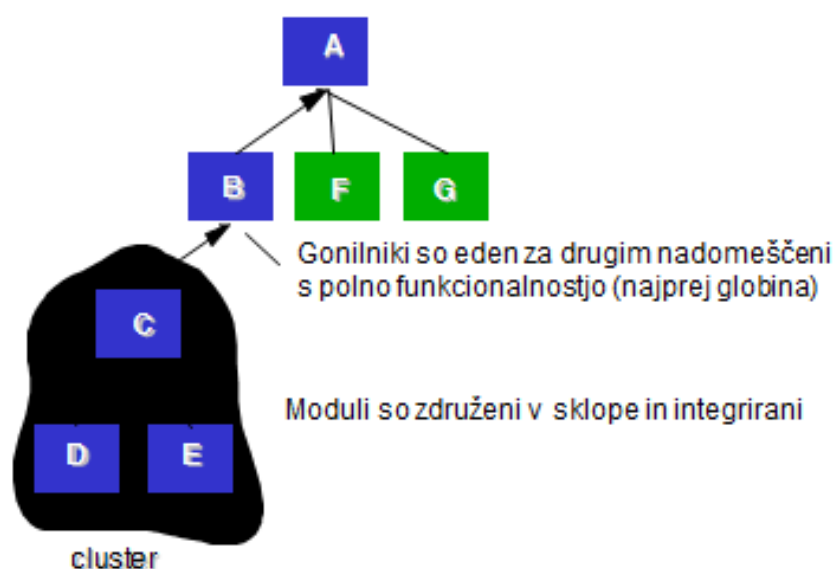
Inkrementalna integracija

- ❖ **Integracija od zgoraj navzdol**
 - ♦ Najprej visokonivojske komponente, ki jih povežemo z nastavki nižjenivojskih komponent (ang. stub).
 - ♦ Nastavki imajo enak vmesnik vendar pa zelo omejeno funkcionalnost.
- ❖ **Integracija od spodaj navzgor (XP)**
 - ♦ Najprej integracija nizkonivojskih komponent s polno funkcionalnostjo, nato razvoj in integracija višjenivojskih komponent dokler ni sistem spopolnjen.
- ❖ V praksi pogosto kombinacija obeh pristopov.

Integracija od zgoraj navzdol



Integracija od spodaj navzgor



Testiranje vmesnikov

- ❖ Je uporabljeno kadar so podsistemi integrirani v večje sisteme.
- ❖ Namen je najti napake vmesnikov, ter napake v predpostavkah povezanih z vmesniki.
- ❖ Uporabno (tudi) pri objektno orientiranem pristopu, kajti objekti so definirani s svojimi vmesniki.

Validacijsko testiranje

- ❖ Validacijsko testiranje je namenjeno ugotavljanju ali programski produkt deluje na način, ki ustreza "razumnim" pričakovanjem uporabnikov.
- ❖ Kaj je "razumno" določajo specifikacije zahtev.
- ❖ Možna sta dva rezultata:
 - ♦ Ugotovljene ustrezne lastnosti
 - ♦ Ugotovljena odstopanja od zahtev, ki so dokumentirana v seznamu odstopanj. O odpravljanju odstopanj se je pogosto potrebno pogoditi s stranko (npr. zaradi časovnih omejitev).

Validacijsko testiranje

- ❖ Pomemben del validacijskega testiranja je pregled konfiguracije.
 - ♦ S tem se zagotovi, da so vsi elementi programskega produkta pravilno razviti, katalogizirani in primerni za prehod v fazo vzdrževanja.

Validacijsko testiranje

- ❖ Alfa in beta testiranje
 - ♦ Razvijalec/preizkuševalec pogosto ne more podrobno poznati načina uporabe programskega produkta s strani uporabnika, zato je v končno validacijo primerno vključiti tudi končne uporabnike.
 - ♦ V primeru posameznih naročnikov se končna validacija izvede v obliki odobritvenih testov (ang. acceptance tests). Ti so lahko neformalni (test-drive) ali formalni z dejanski sistematičnimi testnimi primeri.
 - ♦ Kadar je strank veliko, se končna validacija izvede v obliki alfa in beta testiranja.
 - ♦ Alfa testiranje se izvede na strani razvijalca v nadzorovanem okolju ob prisotnosti uporabnikov.
 - ♦ Beta testiranje izvede eden ali več strank samostojno in brez prisotnosti razvijalca.

Sistemsko testiranje

- ❖ Sistemsko testiranje je namenjeno ugotavljanju obnašanja programskega produkta v prisotnosti drugih sistemskih elementov (strojna oprema, ljudje, informacije)
- ❖ Sistemsko testiranje vključuje več različnih testov, katerih namen je "do popolnosti" preizkusiti računalniški sistem. Različni testi ugotavljajo različne lastnosti sistema.
 - ♦ Testiranje oživljanja (ang. recovery testing) – vzpostavljanje normalnega obratovanja po prisotnosti napak.
 - ♦ Testiranje varnosti (ang. security testing) – preverjanje mehanizmov zaščite pred neželenimi posegi.
 - ♦ Testiranje pritiska (ang. stress testing) – delovanje sistema ob prekomernih obremenitvah.
 - ♦ Testiranje zmogljivosti (ang. performance testing) – preizkušanje prepustnosti sistema.

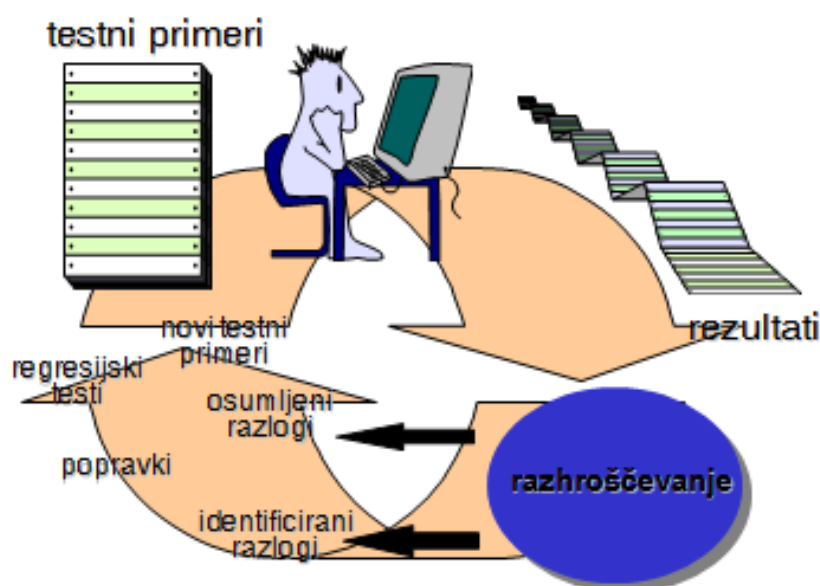
Testiranje pritiska (stress test)

- ❖ Testiranje sistema ob obremenitvi večji od največje načrtovane.
- ❖ Pogosto pokaže neodkrite napake.
- ❖ Ob prekomerni obremenitvi prihaja do napak
 - ♦ Ki pa ne smejo biti katastrofalne.
 - ♦ Ne sme priti do nedovoljenega izpada storitve ali izgube podatkov
- ❖ Posebno pomembno pri porazdeljenih sistemih.

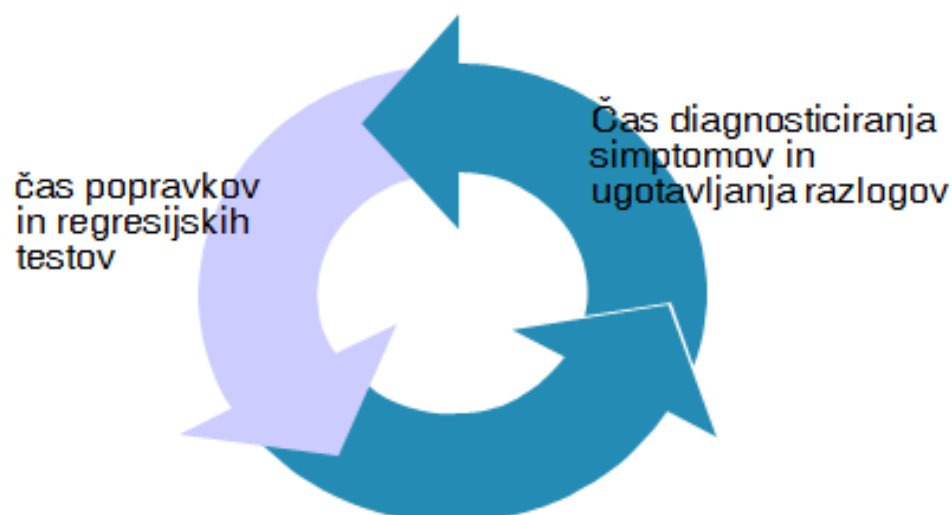
Razhroščevanje

- ❖ Razhroščevanje je diagnostični proces kot posledica uspešnega testiranja.
- ❖ Namen je odprava odkritih napak.
- ❖ Manifestacija napake (simptom) in vzrok zanjo nimata nujno očitne medsebojne povezave.
- ❖ Razhroščevanje je miselni proces, ki simptom poveže z vzrokom napake.

Proces razhroščevanja

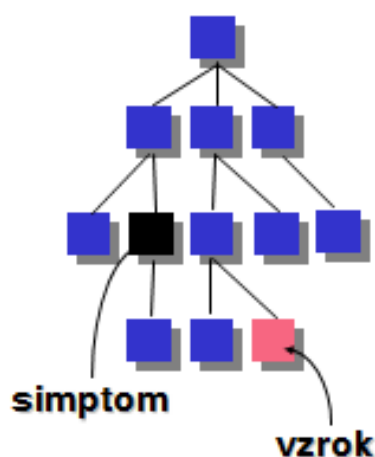


Trud razhroščevanja

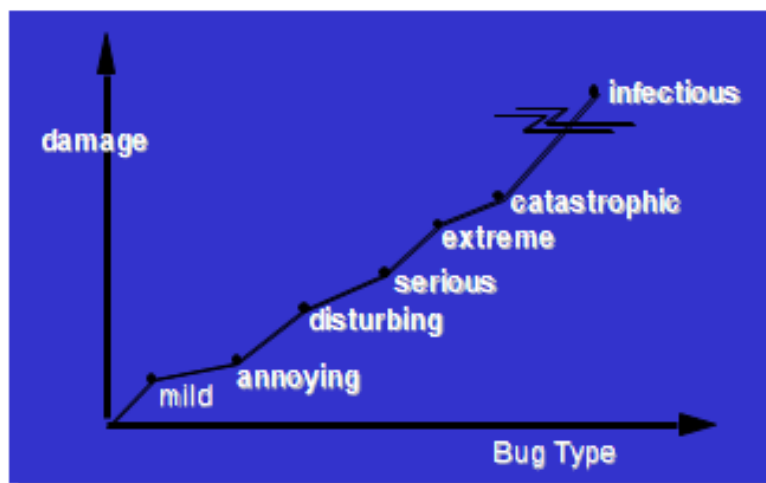


Vzroki in simptomi

- ❖ Simptomi in vzroki napak so lahko geografsko ločeni.
- ❖ Simptom lahko izgine zaradi odprave druge napake.
- ❖ Razlog za simptom je lahko kombinacija ne-napak.
- ❖ Napaka je lahko posledica sistemske napake ali napake pri prevajanju programa.
- ❖ Razlog je lahko v predpostavkah v katere vsi verjamejo.
- ❖ Simptom je lahko nestalen.



Posledice napak



Bug Categories: function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.

Vir: Roger Pressman, Software Engineering. A Practitioner's Approach

Razhroščevanje

❖ Tehnike

- ♦ Obsežno testiranje
- ♦ Sledenje
- ♦ Indukcija
- ♦ Dedukcija

❖ Uporabi orodja, (npr. debugger) za boljši vpogled v dogajanje v programu.

❖ Po "odpravi" napake ne pozabi izvesti regresijskega testiranja!

Dokumentiranje

- ❖ Standard IEEE 829 določa skupino priporočil za dokumentiranje testiranja programskih produktov.
- ❖ IEEE 829 določa osem različnih dokumentov, ki pokrivajo tri korake testiranja:
 - ♦ Priprava na testiranje
 - ♦ Izvajanje testiranja
 - ♦ Zaključek testiranja
- ❖ IEEE 829 je dobro izhodišče za določitev testnih dokumentov. Vendar pa morajo biti dokumenti prirejeni ali spremenjeni, tako da ustrezajo potrebam projekta.
- ❖ Dokumente uporabljaj tako, da zadovoljijo potrebe in ne kot same sebi namen!

IEEE 829 - Priprava na testiranje

- ❖ Načrt testiranja (ang. test plan)
 - ♦ Načrt poteka testiranja. Okvirno določa kako bo potekalo testiranje: kaj bo testirano, kdo bo testiral, kako bo testirano in kdaj bo testiranje potekalo.
- ❖ Specifikacija testa (ang. test design specification)
 - ♦ Generalna določitev testov vključno s pričakovanimi rezultati.
- ❖ Specifikacija testnih primerov (ang. test case specification)
 - ♦ Določitev testnih primerov s testnimi podatki.
- ❖ Testni postopek (ang. test procedure)
 - ♦ Opis praktične izvedbe testiranja.
- ❖ Poročilo predaje v testiranje (ang. test item transmittal report)
 - ♦ Navaja kaj je bilo predano v postopek testiranja.

IEEE 829 - Izvajanje testiranja

- ❖ Dnevnik testiranja (ang. test log)
 - ♦ Seznam vseh izvedenih testov z detajli, podani v časovnem zaporedju.
- ❖ Poročilo testnih incidentov (ang. test incident report)
 - ♦ Seznam dogodkov, ki jih je potrebno dodatno raziskati in njihovim detajlnim opisom (eventuelne odkrite napake).

IEEE 829 - Zaključek testiranja

- ❖ Poročilo testiranja (ang. test summary report)
 - ♦ Povzetek testiranja in sklep o rezultatu testiranja.

Načrt testiranja (po IEEE 829)

- ❖ Identifikator testa
- ❖ Reference: povezave na druge dokumente (projektni plan, specifikacija zahtev, načrti...)
- ❖ Uvod: namen dokumenta
- ❖ Predmet testiranja: seznam predmetov testiranja.
- ❖ Testirana funkcionalnost: kaj bo testirano (s stališča uporabnika).
- ❖ Netestirana funkcionalnost: navedba funkcionalnosti, ki ne bo testirana (s stališča uporabnika)
- ❖ Pristop / strategija: detajlen opis kako bo potekalo testiranje, pod kakšnimi pogoji...
- ❖ Kriterij za uspešnost: določitev kriterijev za uspešnost testa.
- ❖ Pogoj za prekinitev testiranja: npr. število in kritičnost napak
- ❖ Produkt testiranja: poročila, dnevniki, orodja, simulatorji...
- ❖ Dodatne/preostale testne naloge
- ❖ Zahteve vezane na okolje testiranja: posebne zahteve za izvedbo testa.
- ❖ Odgovornosti: seznam odgovornih oseb za izvedbo testiranja
- ❖ Potrebe po osebju in izobraževanju
- ❖ Časovni plan
- ❖ Seznam tveganj in njihovo obravnavanje
- ❖ Odobritve: kdo odobri prehod na naslednjo fazo testiranja.

Vir: <http://gerrardconsulting.com/tkb/guidelines/ieee829/main.html>

Načrt testiranja (splošno)

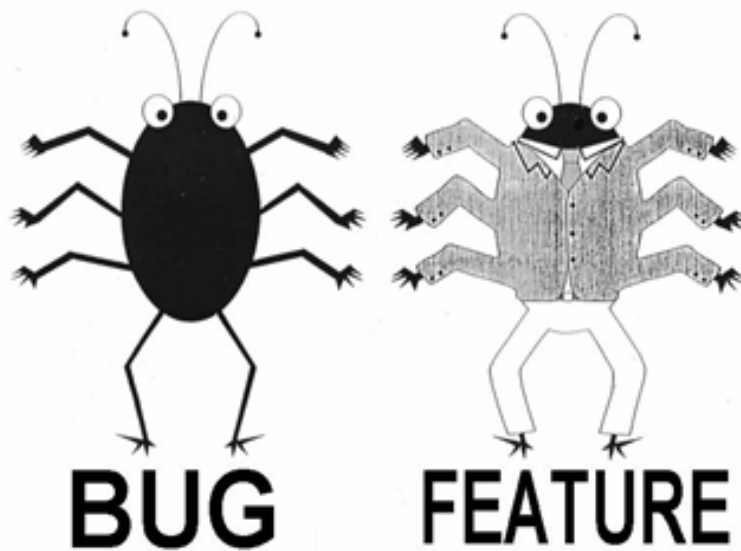
- ❖ Načrt testiranja je dokument, ki določa celotno množico testnih primerov sistema ter dodatne informacije o procesu testiranja.
 - ♦ Neobstoj načrta testiranja pomeni ad-hoc izvedbo testiranja kar pogosto privede do programskih produktov slabe kakovosti.
 - ♦ Načrt testiranja naj bo pripravljen precej pred začetkom testiranja.
 - ♦ S pripravo načrta testiranja se lahko prične takoj po dokončanju specifikacije zahtev.

Testni primeri

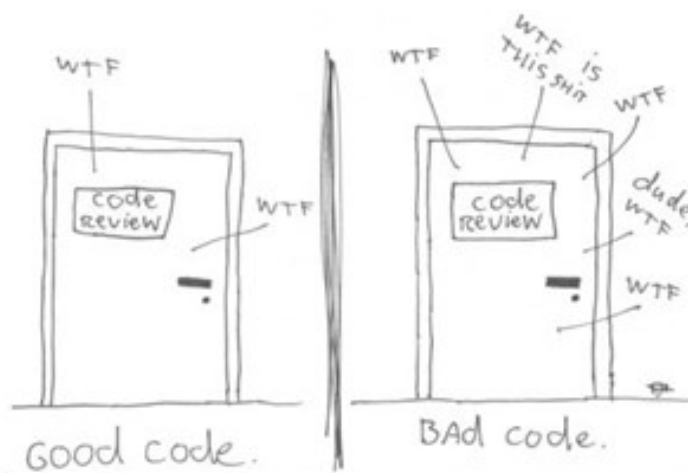
- ❖ Testni primer je eksplicitna množica navodil za detektiranje določenih napak programskega sistema.
 - ♦ Testni primer se uporabi za večje število testov.
 - ♦ Vsak test je izvedba točno določenega testnega primera na točno določeni verziji sistema.
- ❖ Dokumentacija testnega primera vsebuje:
 - ♦ Identifikator in klasifikacija:
 - Številka testnega primera in opisno ime.
 - Navedba sistema, podsistema ali modula na katerega se nanaša.
 - Navedba pomembnosti testa.
 - ♦ Navodila:
 - Točna navodila preizkuševalcu kaj naj naredi. Tipično se ne sklicuje na druge dokumente.
 - ♦ Pričakovani rezultati:
 - Opis kaj naj bi sistem naredil kot posledico izvedbe testnega primera.
 - Preizkuševalec javi napako, če se odziv sistema ne ujema s pričakovanim.
 - ♦ Čiščenje (če je potrebno):
 - Preizkuševalcu pove kako sistem po testu privedi nazaj v 'normalno' stanje.

Poročila

- ❖ Rezultati testiranja so lahko odvisni od okolja, zato naj poročila testiranja vsebujejo navedbo okolja (operacijski sistemi, verzije podatkovnih baz ter povezanih sistemov...)
- ❖ Rezultati so lahko odvisni od drugih programov, ki se izvajajo istočasno s testiranim sistemom!
- ❖ Primer osnutka poročila:
http://sunset.usc.edu/classes/cs577b_2000/EP/05/EP-05.pdf



The ONLY valid measurement
of code quality: WTFs/minute



(c) 2008 Focus Shift/OSNews/Thom Holwerda - <http://www.osnews.com/comics>

9 Vzdrževanje

Vzdrževanje programskega produkta se začne ob njegovi predaji naročniku oziroma uporabnikom. Predaja je lahko kompleksen proces, ki vključuje preverjanje ustreznosti s strani naročnika v obliki SAT (ang. Site Acceptance Test) ali FAT (ang. Factory Acceptance Test). Sledi namestitev v produkcijsko okolje in začetek uporabe. Za uspešen prehod je pomembna kakovostna uporabniška in administratorska dokumentacija, podpora s strani razvijalca ter aktivno upravljanje dogodkov (prijava napak, zahtev in želja uporabnikov).

Po uspešni predaji sledi vzdrževanje, kjer se dogajajo štiri vrste sprememb programskih programskega produkta; korektivne za odpravo napak, adaptivne za prilagajanje produkta spremenjenim pogojem, perfektivne za izboljševanje programskega produkta in preventivne za izboljšanje notranje strukture programskega produkta in s tem lažje nadaljnje vzdrževanje. Zavedati se moramo, da zaradi različnih sprememb produkti s časom izgubljajo na svoji optimalnosti; spremenjene uporabniške potrebe/zahteve in nove tehnološke zmožnosti v največji meri prispevajo k temu, da so novo razviti produkti lahko optimalnejši od starejših. Posledično je tudi vzdrževanje produktov s časom vse težje in dražje, kakovost produktov pa (četudi niso podvrženi spremembam) v splošnem upada.

Vsaka sprememba programskega produkta se izkazuje kot ločen programski proces in bolj ali manj izraženo vsebuje vse faze razvoja od definicije problema do predaje naročniku in nadaljnega vzdrževanja.

Predaja in vzdrževanje

❖ Predaja

- ♦ Ob prevzemu produkta naročnik lahko izvede odobritveno testiranje (ang. acceptance test). Ta se po dogovoru lahko vrši v razvojni organizaciji (factory acceptance test - FAT) ali pri naročniku (site acceptance test - SAT).
- ♦ Celotni sistem (v končni verziji programskih produktov) je dostavljen naročniku (programi, dokumentacija in podatki, potrebni za obratovanje). Sledi namestitev (installation) in prehod v obratovanje.

❖ Obratovanje in vzdrževanje

- ♦ V času obratovanja je potrebno zagotoviti dostopnost uporabniške dokumentacije, izobraževanje uporabnikov ter nuditi podporo uporabnikom.
- ♦ Razvijalec v času obratovanja skrbi tudi za vzdrževanje programskih produktov.
- ♦ Za učinkovito vzdrževanje programskih produktov je pomembno upravljanje konfiguracij.

Dokumentacija in izobraževanje

❖ Dokumentacija je namenjena dvema ciljnim skupinama:

- ♦ Vzdrževalci sistema
- ♦ Uporabniki sistema, ki (dnevno) uporabljajo sistem.

❖ Izobraževanje uporabnikov

- ♦ Prilagojeno specifičnim potrebam uporabnikov
- ♦ Splošno (v primeru produktov za širšo uporabo)

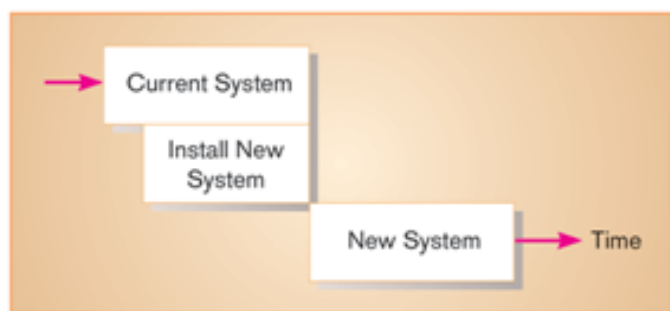
Uporabniška podpora

- ❖ Uporabniška podpora je za uporabnike izjemno pomembna.
- ❖ Nudenje uporabniške podpore je lahko drago (organizacija službe za podporo uporabnikom).

Namestitev

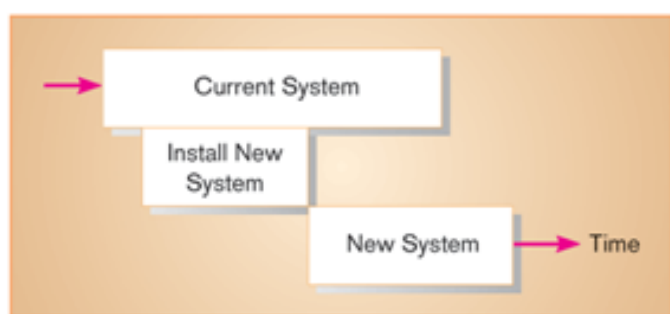
- ❖ Namestitev sistema je organizacijski proces, v katerem obstoječi sistem nadomestimo z novim sistemom.
- ❖ Možne so štiri strategije namestitve:
 - ♦ Neposredna namestitev
 - ♦ Vzporedna namestitev
 - ♦ Namestitev na posamezni lokaciji
 - ♦ Fazna namestitev

Neposredna namestititev



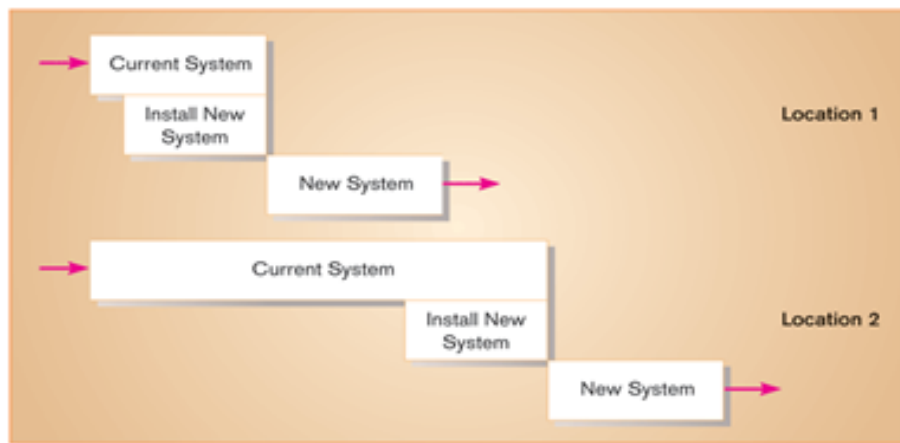
- ❖ Ob zagonu novega sistema se stari sistem zaustavi.

Vzporedna namestititev



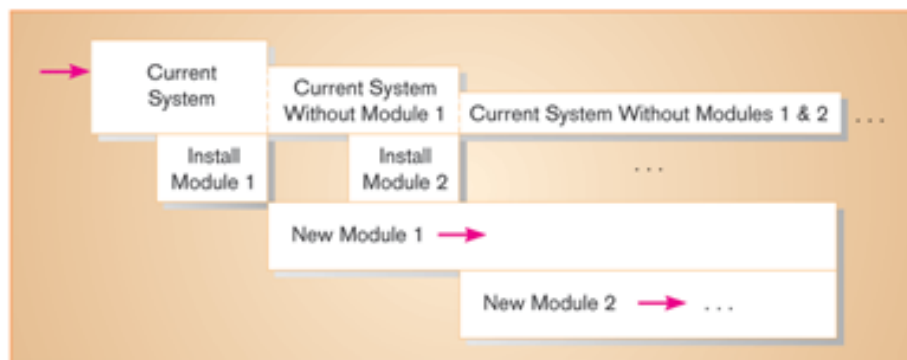
- ❖ Stari in novi sistem vzporedno delujeta toliko časa, dokler se vodstvo ne odloči za izklop starega sistema.

Namestitev na posamezni lokaciji



- ❖ Nov sistem je nameščen na eni sami začetni lokaciji. Na osnovi delovanja na tej lokaciji se odloča o tem, če in kako bo potekala širitev sistema na druge lokacije širom organizacije.

Fazna namestitev



- ❖ Prehod iz starega na nov sistem poteka postopno. Začne se z zamenjavo ene (ali nekaj) funkcijskih komponent, nato pa se komponente postopno zamenjuje dokler ni nov sistem nameščen v celoti.

Načrtovanje namestitve

- ❖ Upoštevati je potrebno
 - ♦ Pretvorbo podatkov obstoječega sistema
 - Odpravljanje napak
 - Prenos podatkov iz obstoječega sistema
 - ♦ Načrtovanje izklopa obstoječega sistema ter postopka preklopa na nov sistem.
 - ♦ Upoštevanje preklopa v poslovnem procesu organizacije.

Faktorji uspešnosti

- ❖ Najpomembnejše merilo uspešnosti projekta razvoja programskega produkta je obseg uporabe sistema.
- ❖ Na uporabo sistema vpliva vrsta dejavnikov:
 - ♦ Osebni interes uporabnikov
 - ♦ Lastnosti sistema
 - ♦ Demografski podatki (spol, starost, rasa, dohodek, izobrazba...)
 - ♦ Organizacijska podpora
 - ♦ Zmogljivost sistema
 - ♦ Zadovoljstvo uporabnikov

Zaključek projekta

- ❖ **Ovrednotenje projektnega tima**
 - ♦ Razporeditev članov na druge projekte
- ❖ **Obveščanje vseh udeležencev o zaključku projekta in prehodu v uporabo in vzdrževanje.**
- ❖ **Izvedba recenzij ob koncu projekta.**
- ❖ **Zaključevanje (vnovčevanje) pogodb s strankami.**
- ❖ **Formalen zaključek**

Vzdrževanje

- ❖ **Splošno:**
 - ♦ Popravila in vse druge aktivnosti, ki ohranjajo produkt v stanju obratovanja.
- ❖ **Programski produkti:**
 - ♦ Vzdrževanje? Programski produkti se ne kvarijo in ne potrebujejo periodičnih vzdrževalnih servisiranja...
 - ♦ Vzdrževanje pomeni spremembe programskega produkta po izdaji z namenom:
 - odpravljanja napak,
 - izboljšanja zmogljivosti (prepustnost sistema...)
 - prilagoditve produkta spremenjenemu okolju.
 - prilagajanje novim zahtevam

Vzdrževanje je ključno za ohranjanje uporabe (programskega) produkta!

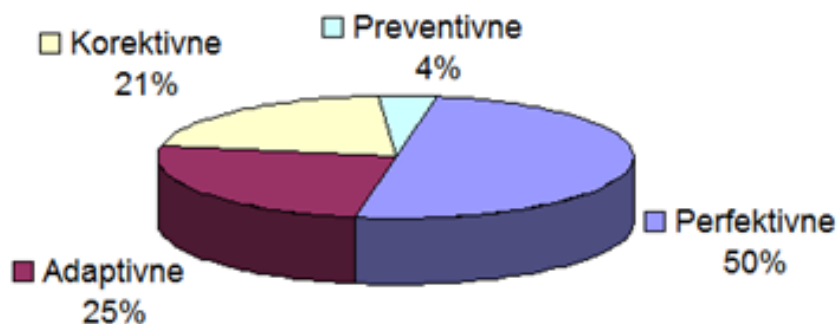
Cena vzdrževanja

- ❖ Vzdrževanje je drago!
Običajno je cena vzdrževanja 2 do 100 krat višja od cene razvoja!
- ❖ Na ceno (težavnost) vzdrževanja vplivajo tako tehnični kot tudi netehnični dejavniki.
 - ♦ tip aplikacije, edinstvenost sistema, fluktuacija in izkušnost kadra, življenska doba sistema, odvisnost od spremenljivega okolja, karakteristike strojne opreme, kakovost načrtovanja (strukturiranost), kakovost izvirne kode, kakovost dokumentacije, kakovost testiranja, število uporabnikov, uporaba orodij...

Spreminjanje programskih produktov

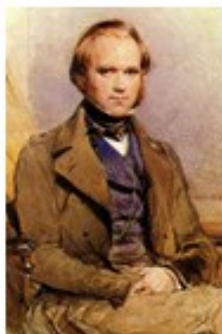
- ❖ V večini primerov je izhodišče za vzdrževanje obstoječi produkt (gre za neskladje z linearnim modelom razvoja).
- ❖ Vzdrževanje vključuje 4 vrste spreminjanja programskega produkta:
 - ♦ Korektivne spremembe
 - odpravljanje napak.
 - ♦ Adaptivne spremembe
 - prilagajanje spremenjenim pogojem.
 - ♦ Perfektivne spremembe
 - izboljšanje programskega produkta.
 - ♦ Preventivne spremembe
 - odpravljanje poslabševanja programskega produkta.

Delež sprememb



Evolucija

- ❖ Vzdrževanje programskih produktov pomeni njihovo evolucijo!
- ❖ **Evolucija:** postopno spreminjanje česa, navadno v popolnejše, bolj dovršene oblike, razvoj (vir: SSKJ).

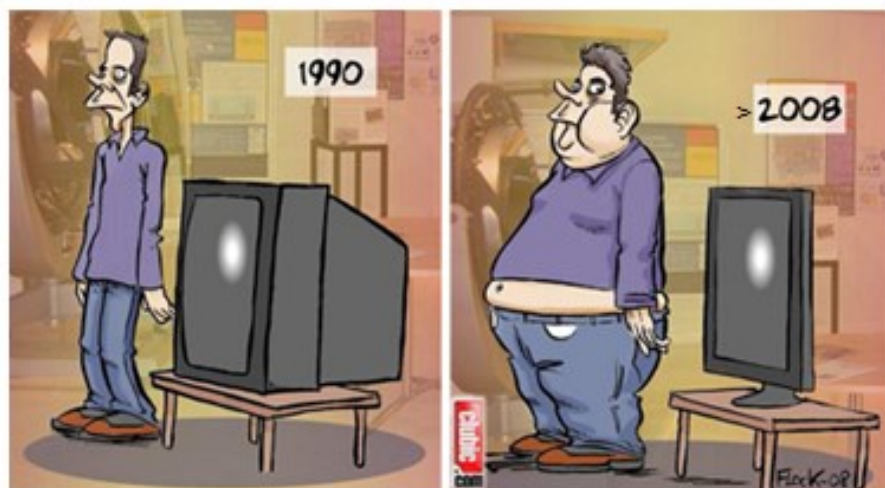


Charles Darwin



Meir Manny Lehman

Evolucija ☺



Kaj se razvija?

- ❖ Specifikacija zahtev
 - ♦ Evolucija zahtev
- ❖ Načrtovanje
 - ♦ Evolucija arhitekture
- ❖ Izvedba
 - ♦ Evolucija podatkov
 - ♦ Evolucija izvorne kode
 - ♦ Evolucija dokumentacije
 - ♦ Evolucija uporabljene tehnologije
- ❖ Testiranje
 - ♦ Evolucija testiranja (testni primeri)

Evolucija mora biti usklajena (ang. co-evolution problem)

Evolucijski tipi prog. produktov

- ❖ Meir Manny Lehman je glede na potrebo po vzdrževanju opredelil tri tipe programskih produktov.
 - ♦ S-system: formalno definirani sistemi, ki v celotni življenjski dobi ne zahtevajo sprememb. V praksi pojavijo zelo redko!
 - ♦ P-system: sistemi katerih zahteve temeljijo na aplikativni rešitvi problema. Potrebujemo inkrementalno prilagajanje dejanskemu svetu.
 - ♦ E-system: sistemi vgrajeni v dejanski svet, zahtevajo stalno prilagajanje dejanskemu svetu. V to skupino sodi večina programskih produktov!

Lehmanovi zakoni evolucije

- ❖ Meir Manny Lehman je zbral 8 zakonov evolucije programskih produktov.
 1. *Continuing Change*
 2. *Increasing Complexity*
 3. *Large Program Evolution*
 4. *Invariant Work-Rate*
 5. *Conservation of Familiarity*
 6. *Continuing Growth*
 7. *Declining Quality*
 8. *Feedback System*
- ❖ Posvetili se bomo štirim izmed omenjenih zakonov.

1. Law of continuing change

❖ Zakon o stalnosti spreminjanja

- ♦ *Sisteme je potrebno stalno prilagajati, sicer postopno vse slabše zadovoljujejo potrebe uporabnikov. Odstopanje med sistemom in njegovim področjem delovanja ustvarja povratno zanko, ki spodbuja spreminjanje sistema.*

6. Law of continuous growth

❖ Zakon o stalni rasti

- ♦ *Če želimo ohranjati zadovoljstvo uporabnikov, se morajo funkcijske zmožnosti programskega produkta ves čas življenjske dobe povečevati. Vsaki izvedbi sistema je zahteve potrebno omejiti. Izpuščene ali prezrte lastnosti bodo sprožile nadaljnje zahteve po spremembah. To je povratna zanka od uporabnikov.*

2. Law of increasing complexity

❖ Zakon o povečevanju kompleksnosti

- ♦ *Z razvojem (spreminjanjem) sistema se kompleksnost sistema povečuje, razen če se kompleksnosti z dodatnim vloženim delom ne zmanjšuje. Če se spreminjanje sistema dogaja brez premisleka o njegovi strukturi, se kompleksnost sistema povečuje in otežuje nadaljnje spremembe. Nasprotno, če se sredstva porabijo tudi za boj proti kompleksnosti, jih je manj na voljo za ostalo spreminjanje sistema. Ne glede na izbrano razmerje se hitrost rasti sistema neizogibno manjša.*

7. Law of declining quality

❖ Zakon o upadanju kakovosti

- ♦ *Če se sistem striktno ne prilagaja spremembam v okolju, v katerem deluje, bo videti, da kakovost sistema upada. Sistem je zgrajen na množici predpostavk in ne glede na njihovo veljavnost v določenem trenutku, bo spreminjajoči svet na njih vplival v smeri njihovega zavračanja. Če se teh predpostavk ne identificira in popravi, bo videti, da kakovost sistema upada, še posebej v primerjavi z alternativnimi produkti, ki so na trg prišli z bolj sveže formuliranimi predpostavkami.*

Veljavnost Lehmanovih zakonov

- ❖ Lehmanovi zakoni se dobro prilegajo komercialnim programskim produktom.
- ❖ V primeru “nestandardne” programske opreme so prisoti odvomi o veljavnosti Lehmanovih zakonov.
 - ♦ Odprtokodna programska oprema...
 - Linux primer...
 - ♦ Drugačnost:
 - razvijalci rešujejo lastne probleme,
 - manjši časovni pritisk...
 - ...

Težavnost vzdrževanja

- ❖ Razkorak med:
 - ♦ Potrebo po spremembah (odpravljanje napak, prilagajanje potrebam)
 - ♦ Potrebo po razpoložljivosti sistema za uporabnike (težnje s strani uporabnikov po minimalni in hitri implementaciji sprememb – predvsem popravkov).
- ❖ Odločitev med:
 - ♦ Hitrim in neelegantnim reševanjem problemov,
 - ♦ Poglobljenim reševanjem in elegantnimi rešitvami.

Konec vzdrževanja?

- ❖ Z odločitvijo o prenehanju vzdrževanja začne programska oprema "propadati".
- ❖ Odločitev o evoluciji oziroma prenehanju vzdrževanja je vezana na naslednja vprašanja:
 - ♦ Je strošek vzdrževanja previsok?
 - ♦ Ali zanesljivost sistema ni zadostna?
 - ♦ Ali se sistem ni več zmožen dovolj hitro prilagajati spremembam?
 - ♦ Ali je zmogljivost sistema nezadostna?
 - ♦ Ali so funkcije sistema le omejeno uporabne?
 - ♦ Ali lahko drugi sistemi isto nalogo opravijo bolje, hitreje, ceneje?
 - ♦ Ali je cena vzdrževanja strojne opreme dovolj velika, da je upravičena zamenjava z novo, cenejšo?

Kdo izvaja vzdrževanje?

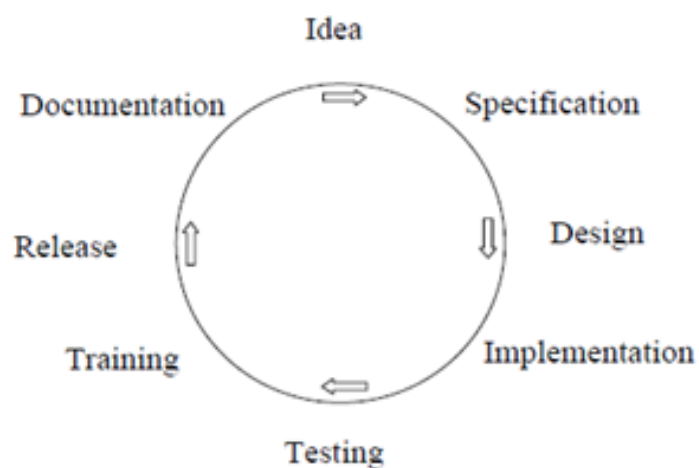
- ❖ **Ločena skupina za vzdrževanje**
 - ♦ Je lahko bolj objektivna
 - ♦ Morda lažje razloči med tem kako naj bi sistem deloval in kako v resnici deluje.
- ❖ **Vzdrževanje opravlja del razvojnega tima**
 - ♦ Sistem bo grajen tako, da bo vzdrževanje lažje.
 - ♦ Člani tima se lahko počutijo preveč samozavestno in zanemarjajo dokumentiranje, ki pomaga pri vzdrževanju.

Proces vzdrževanja

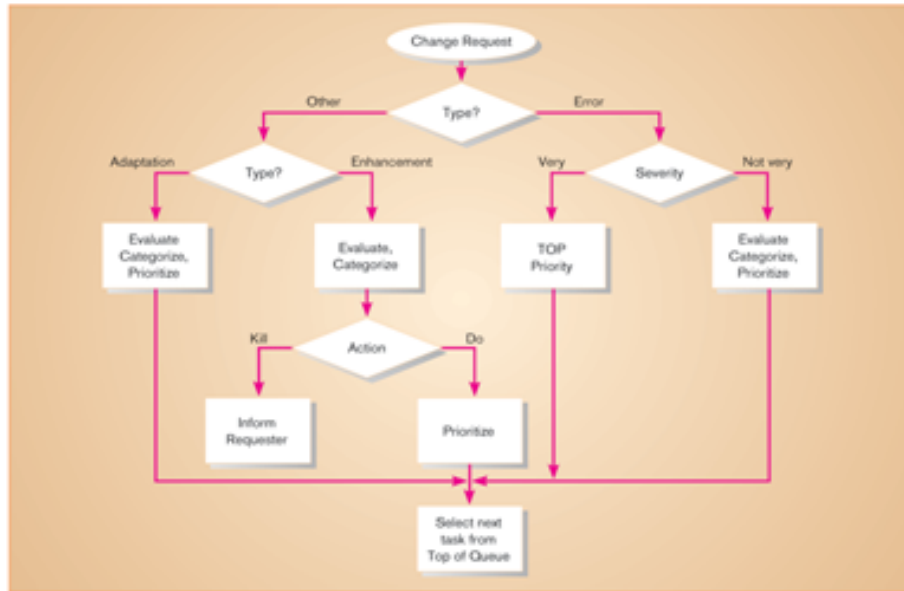
❖ Glavni koraki so:

- ♦ Pridobivanje vlog za vzdrževanje / ideje.
- ♦ Specificiranje zahteve po spremembah.
- ♦ Načrtovanje sprememb
- ♦ Izvedba sprememb
- ♦ Testiranje
- ♦ Izobraževanje
- ♦ Izdaja
- ♦ Dokumentiranje

Proces vzdrževanja



Obravnavanje vlog za vzdrževanje

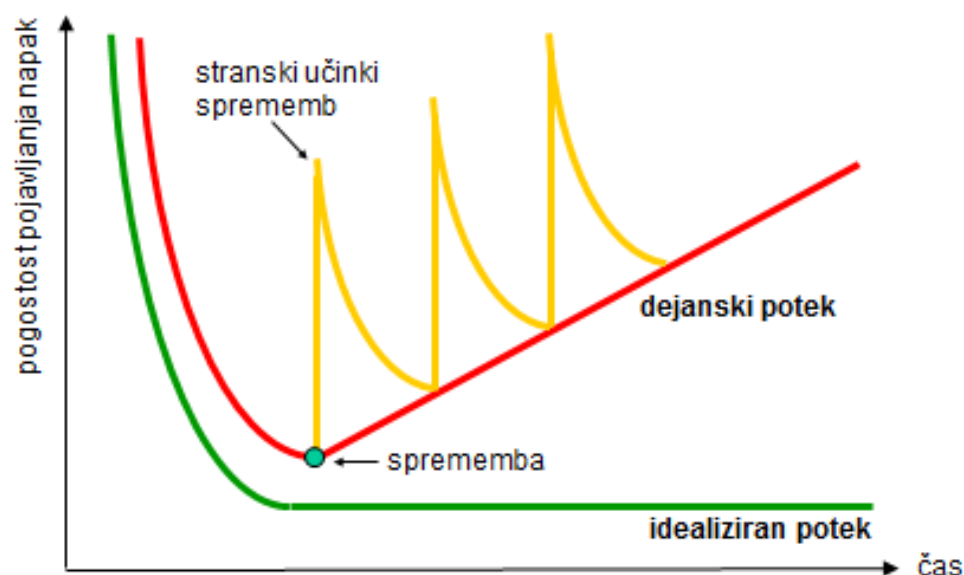


Source: Pressman, R. S. 2001. *Software Engineering: A Practitioner's Approach*, 5th Edition. New York: McGraw-Hill.

Pot vloge za vzdrževanje



Stopnja okvar programske opreme



Tehnike in orodja za vzdrževanje

- ❖ Upravljanje konfiguracij
- ❖ Analiza vplivov (impact analysis)
- ❖ Inženiring, obratni inženiring, ponovni inženiring.

Inženiring

- ❖ **Inženiring** (ang. forward engineering) je tradicionalni proces ki poteka od visokega nivoja abstrakcije in logičnega, izvedbeno neodvisnega načrtovanja, do fizične izvedbe sistema.
- ❖ **Obratni inženiring** (ang. reverse engineering) je proces analize sistema za identifikacijo sistemskih komponent, njihovih medsebojnih relacij in predstavitev sistema v drugačni obliki z višjo stopnjo abstrakcije.
- ❖ **Ponovni inženiring** (ang. Reengineering) je preiskava in spreminjanje sistema z namenom rekonstrukcije in implementacije v novo obliko z enako zunanjo funkcionalnostjo.

Racionalnost?

- ❖ Ang. **rationale**: Explanation of the logical reasons or principles employed in consciously arriving at a decision or estimate. Rationales usually document
 - (1) why a particular choice was made,
 - (2) how the basis of its selection was developed,
 - (3) why and how the particular information or assumptions were relied on, and
 - (4) why the conclusion is deemed credible or realistic.(<http://www.businessdictionary.com/definition/rationale.html>)
- ❖ Racionalnost: Dokumentiran in argumentiran sklep, ki v programskem procesu določa način razvoja produkta.

Racionalnost

❖ Racionalnost vključuje

- ♦ Problematiko, na katero se nanaša
- ♦ Alternative, ki so bile upoštevane
- ♦ Odločitve, ki so bile sprejete za razrešitev problematike
- ♦ Kriterije, ki so vodili do odločitve in
- ♦ Diskusije, ki so bile v razvojni skupini potrebne za doseg odločitve.

Letalski primer

❖ Airbus A320 (1988)

- ♦ Prvo potniško letalo z "fly-by-wire" kontrolnim sistemom upravljanja.
- ♦ 150 sedežev, kratek/srednji dolet.

❖ Airbus A319, A321

- ♦ Oba izpeljana iz A320
- ♦ Enako upravljanje



❖ Racionalnost načrtovanja:

- ♦ Zmanjšani stroški usposabljanja in vzdrževanja
- ♦ Povečana fleksibilnost letalskih družb.

Letalski primer (2)

❖ Airbus A330, A340

- ♦ Dolg/zelo dolg dolet (3xA320)
- ♦ Dvakratno št. sedežev kot A320
- ♦ Podobno upravljanje kot A320



❖ Racionalnost načrtovanja

- ♦ Piloti A320 potrebujejo za letenje z A330, A340 minimalno dodatno usposabljanje.

❖ Posledice

- ♦ Vsaka sprememba v pri navedenih petih letalih mora ohranjati podobnost v upravljanju.

10 Uporabniški vmesniki

Uporabniški vmesnik predstavlja vez med napravo in uporabnikom. Gre za dvosmerno povezavo preko katere uporabnik prejema informacije od naprave in upravlja njeno delovanje. Predvsem s slednjega stališča je razvoj še posebno hiter. Ob številnih in raznolikih vmesniških napravah za upravljanje je posebej pomemben preskok povzročil prihod zaslonov na dotik. Tudi 3D interakcija uporabnika je čedalje enostavnejša in v tej smeri je intenziven razvoj novih brezkontaktnih vmesnikov.

Ne glede na način upravljanja naprav se srečujemo vedno z enakimi izzivi, da je delo uporabnika z napravo čim bolj enostavno, samoumevno in da torej potrebuje čim manj specialnih uporabniških znanj. Težave nam povzročata predvsem dva razloga za razlike med želenim in dejanskim obnašanjem naprave; prepad izvajanja, ko se dejanske aktivnosti naprave razlikujejo od pričakovanih oziroma razpoložljivih, ter prepad vrednotenja, ki od uporabnika zahteva določen napor za pravilno interpretacijo z naprave posredovanih podatkov.

Omenjene težave rešujemo z različnimi principi pri načrtovanju uporabniških vmesnikov, ki jih predstavljamo v tem poglavju. Nenazadnje se moramo še posebej zavedati razlik med uporabniškimi miselnimi modeli in konceptualnim modelom programskega produkta. Zaradi teh razlik, ki jih včasih težko predvidimo, je načrtovanje in vrednotenje uporabniških vmesnikov težavnejše od načrtovanja in vrednotenja vmesnikov med napravami.

Interakcija človeka z računalnikom

- ❖ tipkovnica
- ❖ miška
- ❖ igralna palica
- ❖ wii kontroler...
- ❖ rokavice
- ❖ zasloni na dotik

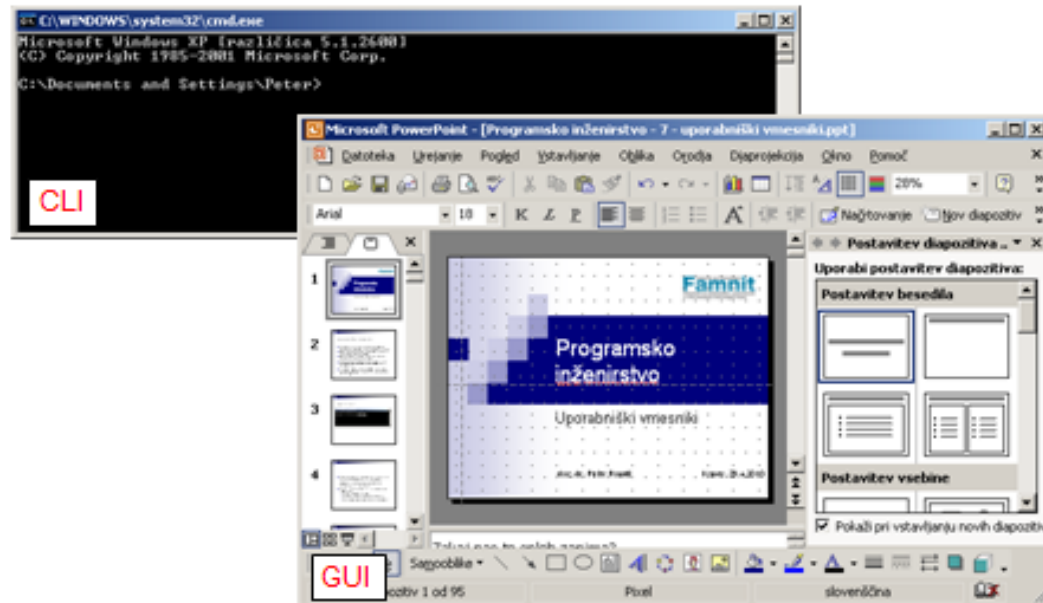


Nove naprave omogočajo evolucijo uporabniških vmesnikov

Interakcija človeka z računalnikom

- ❖ ang. Human-Computer Interaction (HCI)
- ❖ Interakcija med človekom in računalnikom poteka preko uporabniškega vmesnika (ang. user interface).
- ❖ Uporabniški vmesnik predstavlja celoten sistem!
 - ♦ Če je uporabniški vmesnik slab, je slab celoten sistem!
 - ♦ Uporabniki sistem pogosto v večji meri presojujejo po uporabniškem vmesniku kot po funkcionalnosti sistema.
 - ♦ Slabo zasnovan uporabniški vmesnik lahko privede do katastrofalnih napak uporabnikov.
 - ♦ Slab uporabniški vmesnik je razlog za to, da mnogi programski sistemi nikoli niso zaživel.
 - ♦ Uporabniški vmesnik je nujen a ne tudi zadosten del dobrega sistema.

GUI in CLI

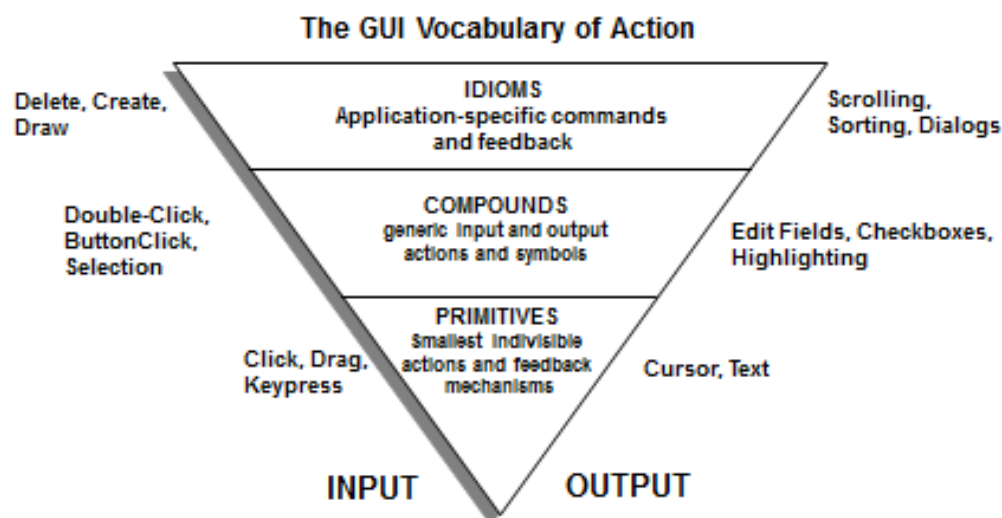


Prednosti GUI

- ❖ So enostavni za uporabo.
 - ♦ Uporabniki brez izkušenj se lahko hitro naučijo dela s sistemom.
- ❖ Uporabnik lahko hitro prehaja iz ene naloge na drugo, med različnimi aplikacijami, ki tečejo hkrati.
 - ♦ Informacije posamezne aplikacije se z delom z drugo aplikacijo ne izgubijo (izgled).
- ❖ Možnost hitre interakcije s sistemom na področju celotnega zaslona.

Omejitve GUI

- ❖ Eden od razlogov za uspeh GUI je omejen (in zato enostaven) nabor akcij, ki jih omogoča.



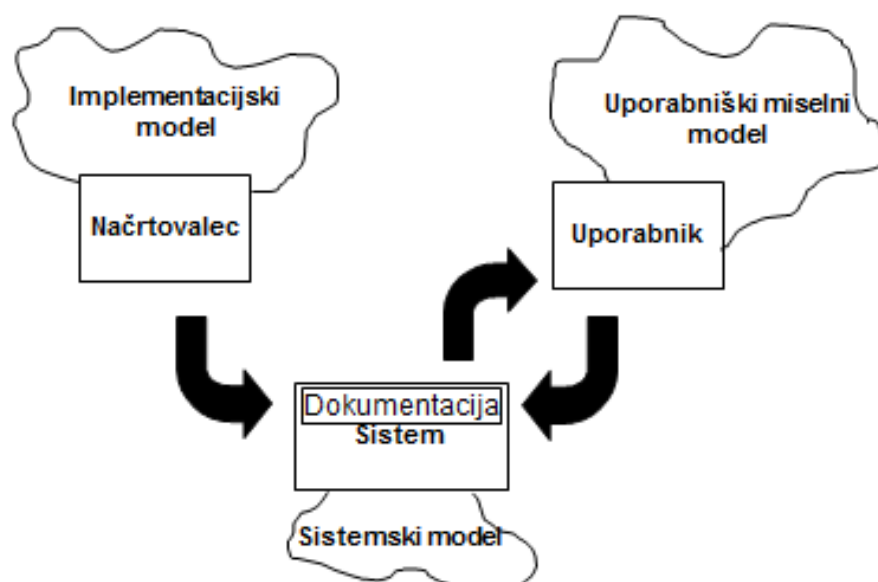
Principi pri načrtovanju GUI

- ❖ Uporaba miselnih modelov
- ❖ Uporaba metafor
- ❖ Evidentnost funkcionalnosti
- ❖ Princip povezanosti
- ❖ Princip povratnih informacij
- ❖ Omejitve

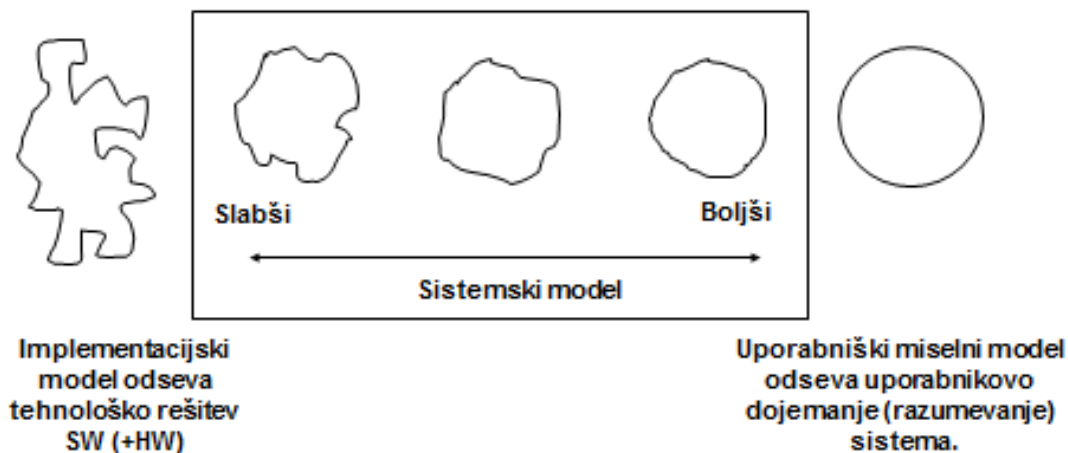
Miselni modeli

- ❖ Ang. mental models.
- ❖ Miselni modeli so človekov lastni in osebni način razumevanja sveta.
- ❖ Ljudje izoblikujemo miselne modele preko izkušenj, vzgoje, šolanja, navodil...
- ❖ Miselni modeli so v principu pogosto napačni, a kljub temu učinkoviti (pravilni v smislu uporabe).
- ❖ Pri razvoju programskih produktov se ukvarjamo z več modeli:
 - ♦ Implementacijski model
 - ♦ Sistemski model
 - ♦ Uporabniški miselni model

Interakcija med modeli



Kakšen naj bo sistemski model?



Delitev mentalnih modelov

❖ Funkcijski

- ♦ Vedenje o tem KAKO, ne pa tudi ZAKAJ.
 - Primer: ugašanje računalnika pred izklopom.

❖ Strukturni

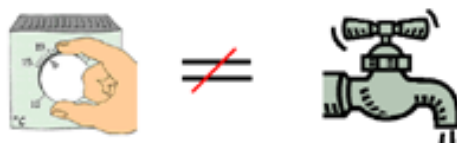
- ♦ Razumevanje komponent in njihove medsebojne odvisnosti (ZAKAJ).
- ♦ Omogočajo reševanje problemov!

Modeli in uporabniški vmesniki

- ❖ Miselni modeli so v splošnem enostavnejši od realnosti.
- ❖ Miselni model novega uporabnika temelji na predhodnih izkušnjah (drugi programi, video igre, druge naprave..)
- ❖ Dokler so miselni modeli učinkoviti, ni potrebe, da so pravilni / točni.
- ❖ Programska oprema, ki ima sistemski model podoben uporabniškemu mentalnemu modelu, potrebuje manj kompleksen uporabniški vmesnik.
- ❖ Večina programske opreme se podreja implementacijskim modelom.

Pravilnost in učinkovitost modelov

- ❖ Računalniški zaslon: mnogi uporabniki vidijo zaslon kot centralni in bistveni del računalnika, kjer se dogaja vse delo...
 - ♦ Nepravilen a učinkovit miselni model.
- ❖ Termostat: ko pridete v hladno sobo in vklopite ogrevanje – kako nastavite termostat? Kar se da na toplo, da bi hitreje segreli prostor?
 - ♦ Ta miselni model ni popolnoma pravilen in je deloma neefektiven.



Konceptualni model

- ❖ Uporabniški miselni modeli niso neposredno pod nadzorom načrtovalca sistema.
- ❖ Uporabniški miselni model, kakršnega bi si želeli, imenujemo (uporabnikov) **konceptualni model**.
- ❖ Kako naj uporabnik pridobi pravi miselni model?
 - ♦ šolanje
 - ♦ dokumentacija
 - ♦ interakcija s sistemom (uporaba)

Konceptualni načrt

- ❖ Da bo sistem enostaven za uporabo mora biti:
 - ♦ načrtovan premišljeno
 - ♦ dovolj enostaven, da je razumljiv na osnovi interakcije.
 - ♦ primeren uporabnikovim nalogam.
- ❖ Načini:
 - ♦ Uporaba znanih konceptov in izrazov
 - ♦ Upoštevanje pričakovanj uporabnikov
 - ♦ Konsistentnost

Primer neustreznega koncepta

- ❖ Datum preteka veljavnosti (ang. expiry date)
 - ♦ S čim povezujemo pretek veljavnosti?
 - ♦ Kakšne aktivnosti so vezane na pretek veljavnosti?
- ❖ Certifikati za elektronsko pošto imajo datum preteka veljavnosti!
 - ♦ Brisanje pretečenega certifikata pomeni nedostopnost s certifikatom šifrirane elektronske pošte.
 - ♦ Primernejši izrez bi bil "Datum obnovitve"!

Prednosti konceptualnih modelov

- ❖ Nudijo možnosti za poenostavitve in inovacije.
- ❖ Definirajo ogrodje za implementacijo:
 - ♦ Dodatni pogledi na jedro problema/rešitve.
 - ♦ Osnova za objektno orientiran razvoj.
 - ♦ Poudarijo način upravljanja s sistemom kot dodatek naboru funkcionalnosti.

Potreba po načrtovanju

- ❖ Z načrtovanjem uporabniških vmesnikov lahko pomembno izboljšamo uporabniško interakcijo s sistemom, ter izboljšamo uporabljivost in zadovoljstvo uporabnikov.
- ❖ Uporabniški vmesniki, ki ne sledijo implementacijskemu modelu zahtevajo več razvojnega vložka. Rezultat pa je bolje prodajana programska oprema.
- ❖ Trendi programskih produktov gredo v nakazano smer.

Metafora

- ❖ **Metaphore so primerjave**, ki pokažejo kako dve stvari, ki sta v večini pogledov različni, po drugi strani v enem pomembnem pogledu **enaki**.
- ❖ **Metafore so analogije** med dvema objektoma ali idejama (wikipedia).

Metafore in uporabniški vmesniki

- ❖ Metafore v primeru uporabniških vmesnikov tipično poudarjajo podobnosti z realnim svetom oziroma obstoječimi konceptualnimi modeli.
- ❖ Primeri metafor:
 - ♦ Namizje (ang. desktop metaphor) : inbox, folder, trash...
 - ♦ Spletne trgovine: nakupovalni voziček, blagajna...
- ❖ Uporabljene so le bistvene lastnosti (npr. pri nakupovalnem vozičku nas ne zanima njegova velikost).

Metafora namizja

- ❖ Macintosh/Windows – zgrajena na metafori namizja
- ❖ Pogosto se metafora namizja navaja kot razlog za uspeh.
- ❖ V več pogledih je metafora namizja nepopolna in omejuje uporabnost uporabniškega vmesnika.
- ❖ Metafore so uporabne, niso pa zadostne!

Meje metafor

- ❖ Pretirano izkrivljanje uporabniškega vmesnika ni priporočljivo.
- ❖ Koristno je uporabiti le tiste lastnosti, ki prispevajo k enostavnosti in razumljivosti.
- ❖ Namesto pretiranega izkrivljanja uporabniškega vmesnika z metaforami je priporočeno idiomatično načrtovanje.
- ❖ **Idiomatično načrtovanje** – uporaba novih simbolov, ki se jih je potrebno naučiti in postanejo značilni za določene objekte, lastnosti...

Primeri metafor



- ❖ Kai's PhotoSoap: Ugani kako povečati sliko...

Idiomatično načrtovanje



Simbol si je enostavno zapomniti, je poseben, značilen.

Evidentna funkcionalnost

- ❖ Evidentna funkcionalnost (ang. *affordance*) je lastnost, da nek objekt kaže možnosti, ki jih nudi uporabniku.
 - ♦ *A visual clue to the function of an object.*
 - ♦ *The perceived and actual properties that determine how something could/should be used.*
- ❖ Evidentna funkcionalnost (*affordance*) je bistveno pomembnejša kot uporabniška navodila – uporabniku postane način dela evidenten že iz predmeta samega (npr. iz izgleda uporabniškega vmesnika).
- ❖ Kako pogosto ste kakšna vrata "potisnili" namesto "potegnili"?

Primer



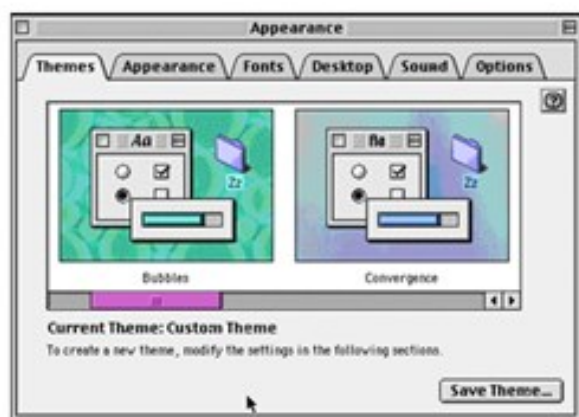
3D gumb:

Samoumevno kaže na možnost, da ga pritismo!

Mac Aqua:

“Čemu služijo pisane lučke na oknu zgoraj levo?”

Primer



❖ Zavihki: evidentno je, kakteri zavihek je izbran, katere so ostale možnosti, kako preklopiti...

Vidnost funkcij

❖ Telefon:

- Običajen telefon ima 15 funkcijskih tipk
- Večina funkcij vključuje uporabo gumbov * in #
- Gumb nima neposredne povezave s funkcijo
- Koliko ljudi zna vzpostaviti konferenčni klic, preusmeritev???

❖ Avto

- V avtomobilu je večje število gumbov, vsak od njih ima svoj namen, ki je na gumbu tudi označen.
- Ljudje imajo bistveno manj težav z uporabo funkcij avtomobila kot telefona...

Funkcije, ki jih nudi sistem, morajo biti vidne!

Princip povezanosti

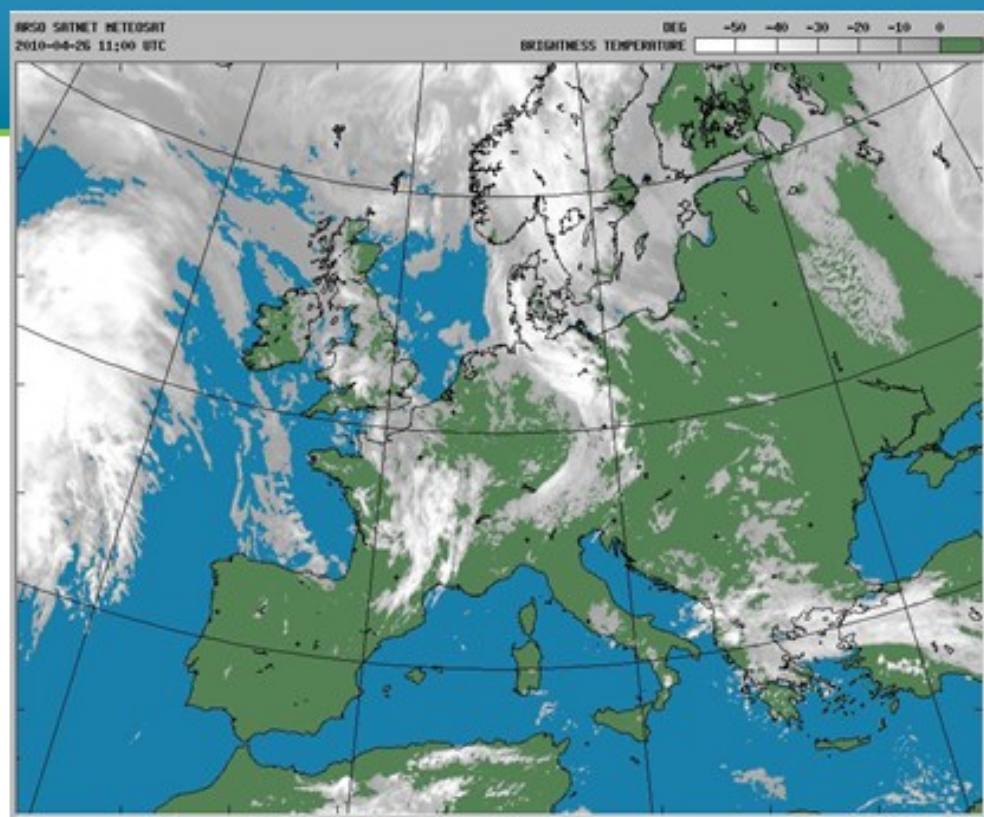
❖ Princip povezanosti odraža povezavo med dvema stvarima.

- Volan – zasuk levo pomeni zavijanje levo...
- Upravljanje računalnika z miško...
- Glasnejši glas pomeni večjo pomembnost...
- Temperaturno kodiranje: barva odraža temperaturo – rdeča=vroče!
- ...

❖ Z uporabo naravnih povezanosti lahko načrtovalec uporabniškega vmesnika doseže, da si uporabnik možne funkcije enostavno nauči in zapomni!

Primeri

VREMENSKA NAPOVED V SLIKI



Princip povratnih informacij

- ❖ Posredovanje povratnih informacij nazaj uporabniku:
 - ♦ Katera aktivnost se je dejansko izvedla / se izvaja.
 - ♦ Kakšni so rezultati predhodne aktivnosti.
- ❖ Povratne informacije morajo biti pravočasne!
- ❖ Primeri:
 - ♦ Indikatorji napredovanja operacije pri časovno daljših operacijah (pridobivanje podatkov iz omrežja, kopiranje, namestitve...)
 - ♦ Tipkovnice bi lahko bile povsem tihe, pa vendar uporabniki ob pritisku tipke pričakujemo "klik".
 - ♦ Premikanje miške po zaslonu. Kadar je sistem prezaposlen in miška zaostaja, je uporabnikom neudobno.

Omejitve

- ❖ Omejitve predstavljajo način za olajšanje interakcije uporabnika s sistemom.
- ❖ Omejitve so lahko fizične, semantične, kulturne, logične.
- ❖ Omejitve lahko omejujejo uporabo določenih aktivnosti:
 - ♦ Določene aktivnosti ne smejo biti izvedene.
 - ♦ Določene aktivnosti morajo biti izvedene.
- ❖ Z omejitvami lahko podrobneje določimo uporabnikove možnosti, npr. pri vnosu podatkov.
 - ♦ Omejitev možnosti izbire (radio box, on/off gumbi, sezname...)
- ❖ Uporabnika je pri interakciji potrebno omejiti na funkcionalnost, ki jo nudi aplikacija.

Primer

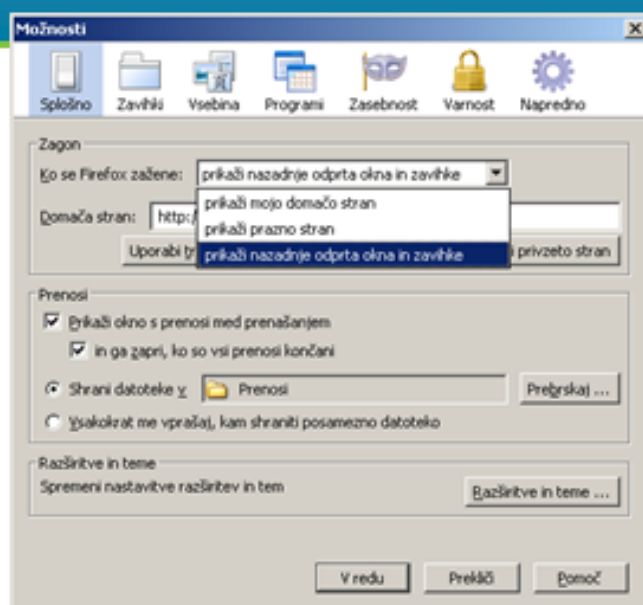
Gender:

✗

Gender: ☐ F ☐ M

✓

Primer



- ❖ Firefox možnosti so urejene tako, da z omejitvami uporabnika usmerijo na možnosti, ki so na voljo.

Predstavitev informacij

- ❖ Informacije so lahko posredovane neposredno ali pa so pretvorjene v nek način njihove predstavitve.
- ❖ Model-View-Controller arhitekturni vzorec podaja način, kako omogočiti več različnih predstavitev istih podatkov (glej načrtovanje sistema).

Izbira načina predstavitev

- ❖ Odločitev o načinu predstavitve informacij je odvisna od lastnosti informacij:
 - ♦ Ali ima uporabnik potrebo po pridobitvi natančnih absolutnih podatkov ali morda po ugotavljanju relativnih povezav med informacijami?
 - ♦ So informacije tekstovne ali številčne?
 - ♦ Kako pogosto se spreminjajo vrednosti?
 - ♦ Ali mora biti sprememba javljena takoj?
 - ♦ Ali mora uporabnik ob spremembi izvesti kakšno aktivnost?
 - ♦ Ali obstaja potreba po neposredni manipulaciji s podatki?

Analogno in digitalno

- ❖ Digitalna predstavitev
 - ♦ Kompakten prikaz zavzame malo prostora.
 - ♦ Podajanje natančnih vrednosti
- ❖ Analogna predstavitev
 - ♦ Hitro podajanje približnih vrednosti.
 - ♦ Možnost podajanja relativnih vrednosti
 - ♦ Boljša indikacija izjemnih vrednosti

Uporaba barv

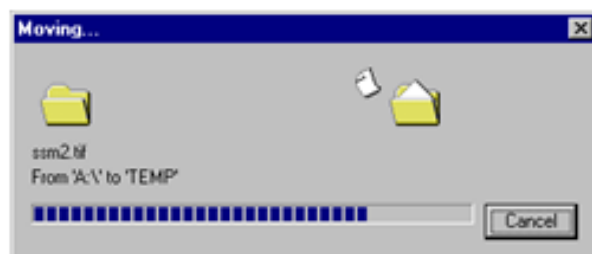
- ❖ Barve predstavljajo dodatno dimenzijo uporabniškega vmesnika.
- ❖ Barve je mogoče uporabiti tako, da uporabnik lažje razume kompleksne strukture informacij.
- ❖ Barve so lahko uporabljene za poudarjanje izjemnih dogodkov.
- ❖ Pogoste napake pri uporabi barv?
 - ♦ Uporaba barv za označevanje pomena.
 - ♦ Prekomerna uporaba barv.

Uporaba pogovornih oken

- ❖ Pogovorna okna (ang. dialog box) se uporabljajo za prikaz dodanih informacij o delovanju sistema.
- ❖ Predstavljajo prekinitev oziroma motnjo normalnega toka uporabe sistema.
- ❖ Pogovornim oknom se je priporočljivo izogibati.
- ❖ Primarna interakcija s sistemom naj se dogaja v primarnem oknu, ne v pogovornih oknih.
- ❖ Ločimo štiri (4) tipe pogovornih oken
 - ♦ Lastnosti (ang. property) – uporabniku predstavi lastnosti (značilnosti) izbranega objekta.
 - ♦ Funkcije (ang. functions) – nadzor nad izvajanjem posamezne operacije, npr. črkovanje, tiskanje.
 - ♦ Objave (ang. bulletin) – podajajo kratek opis problema, npr. obvestila o napakah.
 - ♦ Procesi (ang. process) – uporabnika informirajo, da se program za določen čas ni sposoben normalno odzivati, npr. Windows Explorer v času kopiranja.

Pogovorna okna

- ❖ Zahtevana pogovorna okna so lahko velika in na zaslonu dominantna.
- ❖ Nezahtevana pogovorna okna naj bodo manjša in nevsiljiva.
- ❖ Pogovorna okna imajo prehodni značaj.
- ❖ Nezahtevanim pogovornim oknom se je potrebno izogibati.



Obvestila o napakah

- ❖ Obvestila o napakah so pogosto zlorabljena.
- ❖ Obvestila o napakah poskušajmo odpraviti
 - ♦ Program morda lahko spremenimo tako, da do napak ne more priti.
 - ♦ Morda situacijo lahko obravnavamo kot ne-napako?
- ❖ Obvestila o napakah nikoli niso priljubljena.
- ❖ Obravnavajmo jih kot GOTO:
 - ♦ uporabnika vodi tja, kjer napako lahko odpravi.

Obvestila o napakah

- ❖ Če se jih ne moremo izogniti, naj bodo:
 - ♦ Pozitivna
 - ♦ Pojasnjevalna
 - ♦ Vljudna
 - ♦ Nudijo naj rešitev
 - ♦ Prilagojena naj bodo izkušnjam in kulturi uporabnikov.

Načrtovanje uporabniških vmesnikov

Zmote:

- ❖ Uporabniški vmesnik se mi zdi zadovoljiv – torej bo zadovoljil tudi druge.
- ❖ Uporabniški vmesnik zadovolji povprečno osebo – torej bo zadovoljil kogar koli.
- ❖ Raznolikost ljudi je tako velika, da je ne moremo zadovoljiti z nobenim vmesnikom. Vendar, ker smo ljudje čudovito prilagodljivi, to niti ni pomembno.
- ❖ Dobri uporabniški vmesniki so dragi, vendar pa produkte običajno kupujemo na osnovi izgleda, torej uporabniški vmesnik ni pomemben.
- ❖ Dobri uporabniški vmesniki so čudovita ideja – kadar načrtujem se tega zavedam – delam intuitivno in po zdravi pameti, zato mi jih ni potrebno testirati.

Značaji programov

- ❖ Suveren značaj (1/4)
 - ♦ Programi, ki se polastijo celotnega zaslona za daljše časovno obdobje.
 - ♦ Npr. urejevalniki besedil, preglednice, programska okolja...
 - ♦ Nnačrtovana so za optimalno delo izkušenih uporabnikov.
 - ♦ Čas učenja je relativno kratek glede na celoten čas porabljen za delo z aplikacijo.
 - ♦ Pričakuje se, da bo deloval v maksimiranem oknu.
 - ♦ Uporablja naj neizstopajoče barve in teksture.
 - ♦ Uporabniški vmesnik je lahko zelo bogat, npr. z več orodnimi vrsticami (ang. toolbar).

Značaji programov

❖ Prehoden značaj (2/4)

- ♦ Programi, ki se pokažejo in končajo, ki izvedejo eno samo operacijo.
- ♦ Delujejo kot podpora suverenim aplikacijam.
- ♦ Npr. Windows Explorer, WinZip
- ♦ Uporabniki z njimi porabijo bistveno manj časa kot s suverenimi programi.
- ♦ Zaželeno je, da porabijo čim manjši delež prostora na zaslonu.
- ♦ Uporabniški vmesnik mora biti enostavnejši in bolj poudarjen, saj ima uporabnik manj časa za spoznavanje z aplikacijo.
- ♦ Vse relevantne informacije morajo biti vidne takoj.
- ♦ Navodila naj bodo vgrajena v izgled aplikacije.
- ♦ Ni nujno, da obstaja potreba po povečevanju oken, npr. Windows Calculator

Značaji programov

❖ Demonski značaj (3/4)

- ♦ Programi, ki običajno ne zahtevajo interakcije z uporabnikom in delujejo v ozadju.
- ♦ Npr. Print Manager, Screen Savers
- ♦ Uporabniški vmesnik mora biti enostaven in poudarjen.
- ♦ Uporabniki bodo redko videli uporabniški vmesnik.
- ♦ Do uporabniškega vmesnika se dostopa na vnaprej določen način, npr. preko kontrolne plošče (ang. Control Panel). Ne uporablja se ikon na namizju.

Značaji programov

❖ Parazitski značaj (4/4)

- ♦ Aplikacija, ki je vedno prisotna, vendar je majhna in opravlja podporne vloge.
- ♦ Npr. ura
- ♦ Uporabniški vmesnik mora biti enostaven, zavzemati mora zelo majhen del zaslona.

Način dela

❖ Ko uporabnik želi opraviti določeno nalogo, se srečamo s tremi aktivnostmi.

- ♦ Določanje ciljev (ang. goals) – kaj uporabnik želi, kaj pričakuje od naprave in kakšen je njegov namen uporabe?
- ♦ Izvajanje (ang. execution) – kako uporabnik doseže cilje z uporabo dane naprave?
- ♦ Vrednotenje (ang. evaluation) – kako uporabnik ve ali je dosegel zastavljene cilje?

Način dela

- ❖ Srečamo se z dvema razkorakoma med želenim in dejanskim obnašanjem naprave/programa.
 - ♦ Prepad izvajanja
 - razlike med uporabnikovimi nameni in aktivnostmi, ki so na voljo.
 - na kakšen način naj uporabnik sistemu poda informacije?
 - ♦ Prepad vrednotenja
 - trud, ki je potreben, da uporabnik interpretira stanje naprave.
 - kako uporabniku predstaviti informacije?
 - stopnja zadovoljstva uporabnika glede na namene in pričakovanja.
- ❖ Način dela uporabnika opisuje model sedmih korakov aktivnosti. (ang. The Seven Stages of Action)

The Seven Stages of Action



Uporaba sedmih korakov aktivnosti

- ❖ Za načrtovanje uporabniških vmesnikov.
- ❖ Uporabniku želimo kar se da olajšati prehod med koraki aktivnosti in kar se da zmanjšati število ciklov.
- ❖ Za dosego teh ciljev uporabimo že omenjene principe načrtovanja.

Dokumentiranje

- ❖ Splošno priznane metode dokumentiranja uporabniških vmesnikov NI.
- ❖ Tudi nekega formalnega načina specificiranja uporabniških vmesnikov NI.
- ❖ Možnosti, ki so na voljo:
 - ♦ Slike izgleda
 - Dejanski izgled aplikacije, različna okna (zaslon) v različnih situacijah.
 - ♦ Primeri uporabe
 - Funkcionalnost lahko opišemo s primeri uporabe, ki se sklicujejo na slike izgleda vmesnika.
 - ♦ UML diagrami interakcije
 - Z njimi je mogoče prikazati interakcije med okni.
 - To ni standardni način uporabe diagramov interakcij.

Vrednotenje uporab. vmesnikov

- ❖ Vrednotenje uporabniških vmesnikov je lahko zelo obsežno, težavno in zato tudi drago.
- ❖ Nekaj metod vrednotenja:
 - ♦ Eksperimentiranje – preizkus različnih načinov uporabe elementov uporabniškega vmesnika ločeno in kot celota.
 - ♦ Vprašalniki – uporabniki odgovorijo na vprašanja, ki zadevajo njihovo interakcijo s sistemom.
 - ♦ Neposredno opazovanje – opazovanje interakcije uporabnikov s sistemom.
 - ♦ Posredno opazovanje – analiza video posnetkov interakcije uporabnikov s sistemom.
 - ♦ Avtomatsko pridobivanje podatkov – analiza na podlagi vgrajenih postopkov beleženja dela z uporabniškim vmesnikom.
 - ♦ ...

Kaj vrednotiti?

- ❖ **Čas učenja** – koliko časa potrebuje uporabnik, da je zmožen učinkovito uporabljati sistem.
- ❖ **Hitrost dela** – kako se hitrost odzivanja sistema sklada z uporabnikovimi potrebami oziroma navadami.
- ❖ **Robustnost** – kako toleranten je sistem do uporabniških napak.
- ❖ **Regenerativnost** – kakšna je sposobnost sistema po vzpostavitvi normalnega delovanja po uporabniški napaki.
- ❖ **Prilagodljivost** – kako močno je sistem vezan na en sam način uporabe.
- ❖ **Zanesljivost** – kakšna je pogostost napak v delovanju sistema.
- ❖ **Uporabniško zadovoljstvo** – subjektivno mnenje uporabnikov glede uporabe sistema.
- ❖ **Ponavljajoče operacije** – pogostost uporabe posameznih ukazov uporabniškega vmesnika (pogosto povezano z dostopnostjo teh operacij).

11 Pravne zadeve

Trženje programskih produktov se razlikuje od trženja strojnih komponent. Namreč, z nakupom programskega produkta tipično kupimo licenco za njegovo uporabo in ne produkta kot takega. Pri tem licenca omejuje uporabnika, razvijalca programskega produkta pa odrešuje določenih odgovornosti, ki so tipično povezane s prisotnostjo programskih napak. V zameno lahko uporabnik pridobi pomoč/podporo s strani razvijalca oziroma njegovo zavezo po prioritetenem reševanju odkritih kritičnih napak.

V svetu programskih produktom se srečujemo tudi s pojmom proste (free) in odprtokodne (open source) programske opreme. Oba termina sta si podobna, uvedla pa sta ju organizaciji FSF (Free Software Foundation) in OSI (Open Source Initiative). Najpogostejše zmotno prepričanje je, da je tovrstna programska oprema nujno tudi brezplačna, čeprav je v praksi temu pogosto tako. Pravice povezane z uporabo, nadgradnjo, posredovanjem in podobnimi aktivnostmi povezanimi s programsko opremo so razvidne iz pripadajočih licenc, med katerimi so GPL (GNU General Public License), LGPL (GNU Lesser General Public License), FreeBSD in druge.

Pravne zadeve

- ❖ V programskem inženirstvu se srečamo z vrsto pravnih vprašanj, ki zadevajo odnose s strankami in podizvajalci.
- ❖ Gre za pogodbe, licence in avtorske pravice.
- ❖ Glavni viri predavanja:
 - ♦ <http://www.softwarecontracts.net/>
 - ♦ <http://www.opensource.org/>

Kupovanje programskih produktov

- ❖ Dokumenti povezani s kupovanjem programskih produktov so:
 - ♦ Naročilo nakupa
 - Dokument kupca, tipično v uporabi za nakup standardnih produktov, s katerim se kupec zaveže za nakup produkta.
 - ♦ Povpraševanja
 - Po informacijah (ang. request for information – RFI)
Dokument s katerim kupec povprašuje po rešitvah svojega problema. Zainteresirani ponudniki nanjo odgovorijo s svojo ponudbo izdelkov/storitev.
 - Po ceni (ang. request for quotation – RFQ)
Dokument s katerim kupec neobvezujoče povprašuje po ceni izdelkov ponudnika.
 - Po ponudbi (ang. request for proposal – RFP)
Dokument s katerim kupec povprašuje po rešitvi, s točno navedbo zahtev. Odgovor nanj so ponudbe z navedbo rešitve in cene. Če se kupec za določeno ponudbo odloči, postane ta za ponudnika zavezujoča.

Tipi programskih produktov

- ❖ **Komercialna paketna programska oprema** (ang. commercial off-the-shelf software)
 - ♦ Standardna programska oprema, namenjena široki uporabi, narejena za vnaprej nedoločeno število uporabnikov. Nakup vključuje licenco, garancijo in opsijsko vzdrževanje. Licenčna pogodba običajno stopi v veljavo samodejno z nakupom.
- ❖ **Odprtokodna paketna programska oprema** (ang. open-source off-the-shelf software)
 - ♦ Programska oprema, z licencami odprte kode. To ni nujno brezplačna programska oprema, odprtokodne licence le zagotavljajo vpogled v izvorno kodo in uporabo pod določenimi pogoji.
- ❖ **Programska oprema po meri** (ang. custom developed software)
 - ♦ Programska oprema je razvita za vnaprej znanega kupca. Licenca sledi pogodbi med strankama in lahko vključuje izvorno kodo.

Pogodbe (glede na fazo razvoja)

- ❖ **Definiranje zahtev**
 - ♦ Razvijalec mora priskrbeti za licence orodij, uporabljenih v procesu razvoja (in vzdrževanja) programskega produkta.
- ❖ **Razvoj**
 - ♦ Razvojni tim mora poznati izvor uporabljene kode in knjižnic, ki so lahko:
 - kupljene s komercialno licenco,
 - pridobljene z odprtokodno licenco (in ustreznim izpolnjevanjem pogojev),
 - razvite v lastnem razvojnem timu.
- ❖ **Prodaja/prevzem**
 - ♦ Programski produkt je opremljen z licenčno pogodbo. Ta naj bo skrbno pripravljena, kajti ponudnik običajno želi obdržati lastništvo produkta, da le tega lahko proda še drugim strankam. Ponudnik običajno programskemu produktu določi le omejeno garancijo.

Pogodbe (glede na fazo razvoja)

❖ Namestitev

- ♦ Namestitev lahko opravi ponudnik, če je tako dogovorjeno s pogodbo o namestitvi.

❖ Vzdrževanje

- ♦ Vzdrževanje je podvrženo vzdrževalni pogodbi.

❖ Podpora

- ♦ Pomoč uporabe programskega produkta in prelagoditve okolju se ureja z (ločeno) pogodbo o podpori.

Intelektualne pravice

- ❖ **Lastnik** programskega produkta ima določene ekskluzivne pravice. Osnovne so **avtorske pravice** (ang. copyright), ki lastniku dajejo pravice:

- ♦ lastne uporabe produkta,
- ♦ določanja kako je produkt lahko uporabljen,
- ♦ določanja kdo lahko uporablja produkt,
- ♦ določanja kdo lahko produkt spremeni,
- ♦ določanja kdo lahko kopira in prodaja kopije produkta.

- ❖ Vsaka uporaba (nakup) ne-lastnikov produkta je dovoljena le če ima ta sklenjen dogovor z lastnikom v obliki **licenčne pogodbe**. Nakup licence ne pomeni nakupa programskega produkta, pač pa le nakup omejenega nabora pravic za uporabo produkta.

Patenti

- ❖ Programska oprema lahko vsebuje algoritme, ki so zaščiteni s patenti.
- ❖ Lastnik patenta ima pravico preprečiti uporabo programske opreme, ki vsebuje njihovo inovacijo.
- ❖ Pravico do uporabe programskega produkta s patentom določa licenčna pogodba.

Komercialne licence

- ❖ Večina komercialnih ponudnikov v licenčni pogodbi omeji kaj je kupcu dovoljeno početi s programskim produktom.
 - ♦ prepoved povratnega inženirstva,
 - ♦ prepoved uporabe s strani tretjih oseb
 - ♦ omejitev komercialne uporabe
 - ♦ omejitev števila uporabnikov ali procesorjev
 - ♦ omejitev kopiranja programske opreme
 - ♦ omejitev prenosa licence na tretje osebe.
 - ♦ ...

Odprtokodne licence

- ❖ Odprtokodna programska oprema **ni** brezplačna programska oprema v javni lasti!
- ❖ Avtor obdrži svoje pravice (avtorske pravice, patenti, lastnina) in določene pravice da na voljo drugim z licenco.
- ❖ Tipično avtor kupcu da pravico do
 - ♦ spreminjanja izvorne kode
 - ♦ kopiranja
 - ♦ nadaljnjega razdeljevanja produkta pod isto licenco.
- ❖ Ostale pravice so odvisne od dejanske licence (npr. GNU GPL, GNU LGPL, Apache License...)

Odprtokodne licence

- ❖ **GNU General Public License ("GPL"):**
 - ♦ Najpogosteje uporabljena odprtokodna licenca (>60% odprte kode).
 - ♦ Omogoča prosto distribucijo in spreminjanje
 - ♦ Zahteva, da je originalna in spremenjena koda na voljo svetu pod istimi pogoji kot ga določa originalna licenca.
 - ♦ Mora obdržati navebe avtorskih pravic in garancijskih izjav.
 - ♦ Ne nudi patentnih pravic.
 - ♦ Zelo omejujoča.

Odprtokodne licence

❖ GNU Lesser General Public License ("LGPL"):

- ♦ Podobna licenci GPL
- ♦ Je manj omejujoča in ne omejuje programov, ki se povezujejo z LGPL produkti.
- ♦ Pogosto uporabljena za odprtokodne knjižnice.
- ♦ Zahteva ohranjanje navedbe avtorskih pravic in garancijskih izjav.

Odprtokodne licence

❖ FreeBSD licenca:

- ♦ Neomejujoča licenca:
- ♦ Zahteva le ohranjanje navedbe avtorskih pravic in garancijskih izjav.

Garancije

- ❖ S prodajo produkta (licence) prodajalec kupcu obljubi določeno funkcionalnost produkta. Ta obljuba je lahko neformalna ali formalna v obliki tehničnih specifikacij.
 - ♦ Pri običajnih produktih gre za zavezo, da je produkt brez napak in določitev načina odpravljanja napak. Prodajalec je odgovoren tudi za posredno škodo.
 - ♦ Pri programskih produktih se prodajalec tipično ne nudi neomejene garancije, saj je težko zagotoviti, da programska oprema ne vsebuje napak.

Garancije

- ❖ Nekateri programski produkti se prodajajo brez garancije (ang. "as-is"). Torej ne zagotavljajo, da bo programski produkt deloval kot je dogovorjeno.
- ❖ Poslovna programska oprema in programska oprema po meri tipično vsebujeta omejeno garancijo.
 - ♦ Ta ne garantira delovanja brez napak in skladno s specifikacijami!
 - ♦ Običajno pa vključujeta mehanizme povezane s programskimi napakami.

Omejene garancije in napake

- ❖ Nekateri možni mehanizmi urejanja napak so:
 - ♦ **Kanal sporočanja napak** – je le mehanizem sporočanja napak, ne pa tudi njihovega odpravljanja.
 - ♦ **Vzdrževalne posodobitve** (ang. maintenance update) – nudenje posodobitev za odkrite in odpravljene napake (ni nujno, da so vse odkrite napake tudi odpravljene!)
 - ♦ **Vračilo z nadomestilom** – v času garancije lahko stranka vrne produkt s polno povrnitvijo kupnine, če programski produkt odstopa od specifikacij.
 - ♦ **Upravičenost do odprave napak** – v primeru najdenih napak prodajalec zagotavlja njihovo odpravo v vnaprej določenih časovnih rokih glede na kritičnost napake.
 - ♦ **Upravičenost do zahtev za spremembo** – za programsko opremo po meri – uporabniku se omogoči prilagajanje programskega produkta poslovnim potrebam (spremembe se določajo z ločenimi pogodbami).
 - ♦ **Pomoč uporabnikom** – pomoč uporabnikom pri uporabi produkta.

Pogodbe za vzdrževanje

- ❖ Pogodbe za vzdrževanje (ang. software maintenance contract) so ponavadi način podaljševanja garancije.
- ❖ Vzdrževalne pogodbe v splošnem lahko vsebujejo drugačne pogoje od garancijskih.
- ❖ Tipična razlika je v tem, da te pogodbe ne nudijo pravice za vračilo z nadomestilom.

Pogodbe o podpori

- ❖ Pogodbe o podpori (ang. software support contracts) so orodje za zagotavljanje optimalnega delovalnja produkta.
- ❖ Razširjanje pravice nudene s pogodbo o vzdrževanju. Nudijo:
 - ♦ **Dostop do informacij** o produktu, baze znanja, forumi...
 - ♦ **Inženirsko podporo** pri uporabi programskega produkta (ne le pri odpravljanju napak)
 - ♦ **Pravico do posodobitev** z zadnjimi verzijami produkta (opcijsko).

Pogodbe o upravljanju

- ❖ S pogodbo o upravljanju (ang. managed services contract) ponudnik prevzame obveznost vzdrževanja produkta
 - ♦ vključno z dnevnimi vzdrževalnimi poravili kot so npr. izdelava varnostnih kopij, namestitve programskih popravkov, upravljanje omrežja, urejanje pravic uporabnikov...
- ❖ Stranka se za vse težave lahko obrne na ponudnika, hkrati pa sama (za ta produkt) ne potrebuje vzdrževalnega osebja.

Pogodbe o razvoju

- ❖ Pogodbo o razvoju podjetje sklene takrat, kadar paketna programska oprema ne zadovolji njegovih potreb in se s svojimi potrebami obrne na razvijalca.
- ❖ Tovrstne pogodbe so lahko zelo kompleksne, saj
 - ♦ morata za uspešen produkt stranki sodelovati,
 - ♦ tovrstni projekti so lahko tvegani, kajti v času podpisa pogodbe nobena stran ne ve kakšen bo končni produkt.
- ❖ Pogodba lahko vsebuje
 - ♦ Predpostavke – če te niso izpolnjene lahko strani odstopita od pogodbe.
 - ♦ Obveznosti – obeh strank, kar se lahko razume tudi v smislu garancijskih pogojev
 - ♦ Delo – določitev dela, ki ga opravi razvijalec in določa tudi meje, ki razvijalcu zagotavljajo profitabilnost.
 - ♦ Odškodnine – za primer če ponudnik ne uspe zadovoljiti členov pogodbe.
 - ♦ Pogoji za spremembo pogodbe – na osnovi dela, predpostavk, odgovornosti, brez odstopa od pogodbe.

Odprta koda

Zgodovina odprte kode

- ❖ 1984: Richard Mathew Stallman (RMS) začne projekt GNU (GNU's Not Unix). Uveljavi se izraz "free software".
- ❖ 1991: Linus Torvalds objavi razpoložljivost njegovega Unix jedra – Linux.
- ❖ 1997: Eric Raymond (ER) napiše esej "The cathedral and the Bazaar".
- ❖ 1998: Netscape napove izdajo izvirne kode svojega brskalnika, uveljavi se izraz "open source".

Odprta koda: dve struji



Richard Stallman (FSF)



Eric Raymond (OSI)

FSF - Free Software Foundation



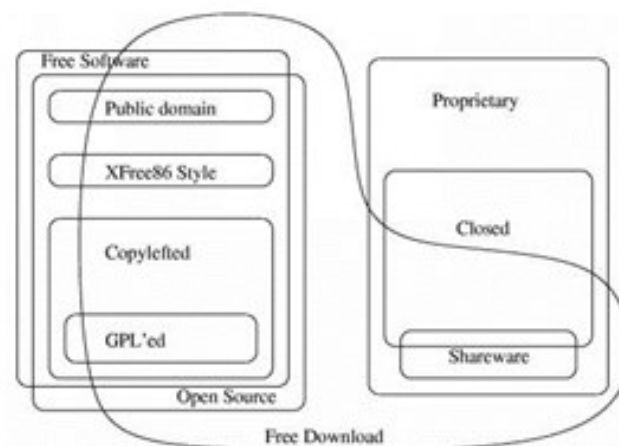
- ❖ *Stallman is a saint in the Church of Emacs--- Saint GNUcius.*
- ❖ *Sainthood in the Church of Emacs requires living a life of purity--but in the Church of Emacs, this does not require celibacy (a sigh of relief is heard). Being holy in our church means installing a wholly free operating system-- GNU/Linux is a good choice--and not putting any non-free software on your computer. Join the Church of Emacs, and you too can be a saint!*

OSI - Open Source Initiative



❖ [...] the real reason for the re-labeling is a *marketing one*. We're trying to pitch our concept to the *corporate world* now. We have a *winning product*, but our positioning, in the past, has been awful. The term "free software" has a load of *fatal baggage*; to a businessperson, it's too redolent of fanaticism and flakiness and strident anti-commercialism. Mainstream corporate CEOs and CTOs *will never buy "free software,"* manifestos and clenched fists and all. *But if we take the very same tradition, the same people, and the same free-software licenses and change the label to "open source" – that, they'll buy.*

Umestitev odprte kode



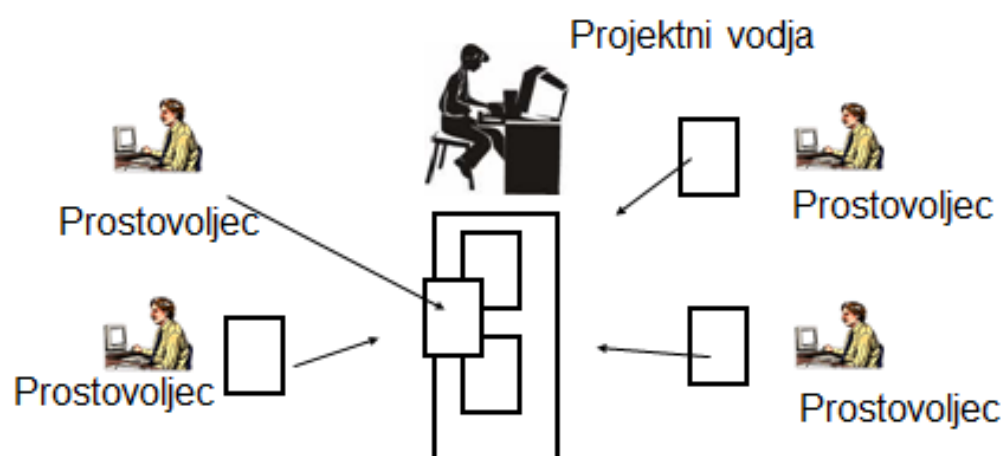
OSI – definicija odprte kode

- ❖ Open source promotes software reliability and quality by supporting **independent peer review** and **rapid evolution** of source code. To be OSI certified, the software must be distributed under a license that guarantees **the right to read, redistribute, modify, and use the software freely**.
- ❖ Odprta koda ne pomeni le dostopnost izvorne kode: <http://www.opensource.org/docs/osd>

Primeri odprtokodnih programov

- ❖ Velik delež v spletni infrastrukturi:
 - ♦ Web strežniki (70%: apache)
 - ♦ Mail strežniki (about 50%: sendmail, exim, postfix)
 - ♦ Skriptni jeziki (perl, php)
 - ♦ DNS (domain name system)
 - ♦ MySQL (database)
- ❖ Uporabniški programi
 - ♦ Spletni brskalniki (ang. web browsers) (Firefox)
 - ♦ Pisarniški programi (OpenOffice)
 - ♦ KDE /GNOME (namizja za Linux)
 - ♦ Darwin (operacijski sistem za Apple)
 - ♦ ...
- ❖ Razvojna okolja
 - ♦ Emacs /GCC/GNU Tools (FSF)
 - ♦ NetBeans (Java IDE)
 - ♦ ...

Model razvoja odprte kode



Pogosto je odprtokodna programska oprema plod prostovoljnega udejstvovanja razvijalcev.

Trženje odprte kode

❖ Strategije:

- ◆ Krepitev izgublajoče pozicije glavnega akterja na trgu (Netscape)
- ◆ Prodor strojne opreme (Apple)
- ◆ Ponudba storitev (Redhat)
- ◆ Trženje drugih, povezanih izdelkov (O'Reilly)
- ◆ Prosti programi, prodajanje blagovne znamke
- ◆ Prosti programi, prodaja vsebine
- ◆ ...

Komercialno ali odprtokodno?

❖ Kakšne prednosti in slabosti pričakuješ od komercialne in odprtokodne programske opreme?

- ♦ Zanesljivost ?
- ♦ Varnost?
- ♦ Podpora ?
- ♦ Cena ?
- ♦ Garancija ?
- ♦ Podpora odprtim standardom ?
- ♦ Kakovost (kode)?
- ♦ Dostopnost kode in možnost spreminjanja ?