Toimintopisteet

- Toimintopisteiden laskemisen pääidea on tuloksen riippumattomuus toteutusteknologiasta: lasketaan *toiminnallisuuden* määrä
  - Ei väliä koodataanko C:llä vai Javalla, toiminnallisuus on sama
  - Tuo esiin *tuottavuuserot* -> kuinka paljon koodia ja työtä tarvitaan tietyn kokoisen toiminnallisuuden toteuttamiseen
- Toimintopisteet on esitelty niinkin aikaisin kuin vuonna 1979 (Albrecht). Vaikka menetelmää on tämän jälkeen kehitetty eteenpäin, on alkuperäinen malli edelleen yllättävän toimiva.
  - Tällä hetkellä (alkuperäisestä) toimintopistemalleista huolehtii kokonainen organisaatio: International Function Point Users Group, IFPUG (www.ifpug.org).
- Toimintopistestandardeja on useita, esimerkiksi suomalainen FiSMA –menetelmä (ISO/IEC 29881:2010, myös suomenkielisenä SFS –standardina)
  - http://www.fisma.fi/toiminta/menetelmat/ ja http://www.fisma.fi/wp-content/uploads/2006/09/fisma_fsm_method_11.pdf

Toimintopisteiden laskenta 1

Perinteiset toimintopisteet lasketaan kolmessa vaiheessa:

1. Lasketaan *raakapisteet*
   - Arvioidaan ohjelmiston ”komponenttien” määrä:
     - syötteiden määrä, kuten lomakkeet
     - tulosteiden määrä, kuten raportit ja virheilmoitukset
     - kyselyiden määrä, kuten generoitavat näytöt
     - loogisten tiedostojen määrä, kuten tietokannan taulut
     - ulkoisten liittymien määrä, kuten käyttöliittymät, liittymät muihin järjestelmiin ja liittymät muihin sovelluksiin
   - Määritellään kullekin komponentille painokerroin sen mukaan, miten työlääksi sen toteuttaminen arvioidaan (helppo/yksinkertainen, keskimääräinen, vaikea/monimutkainen):
     - syötteet: 3 (yksinkertainen), 4 (keskimääräinen), 6 (vaikea)
     - tulosteet: 4, 5, 7
     - kyselyt: 3, 4, 6
     - loogiset tiedostot: 7, 10, 15
     - ulkoiset liittymät: 5, 7, 10
   - Lasketaan komponenttimäärien ja painokertoimien tulojen summa.

Toimintopisteiden laskenta 2

2. Lasketaan ohjelmiston *kompleksisuuskerroin* (complexity factor)
   - Kompleksisuuskerroin saadaan arvioimalla 14 kysymystä asteikolla 0-5, missä
     - 0=ei vaikutusta, 1=satunnainen, 2=kohtalainen, 3=keskimääräinen, 4=merkittävä, 5=olennainen vaikutus.
   - Saadut arvot lasketaan yhteen.

3. Lasketaan toimintopisteet FP kaavalla

   FP = CFP*(0,65 + 0,01* RCAF), missä

   - CFP = raakapisteet
   - RCAF = kompleksisuuskerroin

Kompleksisuuskertoimen kysymykset

Kompleksisuuskertoimen RCAF kysymykset:

- Tarvitaanko luotettavaa tietojen varmistus- ja palautusmenettelyä?
- Tarvitaanko tietoliikenneominaisuuksia?
- Onko hajautettua prosessinhallintaa?
- Onko kyseessä suorituskykykriittinen ohjelmisto?
- Käytetäänkö järjestelmää raskaasti kuormitetussa ympäristössä?
- Tarvitaanko interaktiivista tietojen syöttöä ohjelmiston suoritusaikana?
- Täytyykö interaktiivinen tietojen syöttö synkronoida usealle näytölle tai operaatiolle?
- Päivitetäänkö tiedostoja interaktiivisesti ohjelmiston suoritusaikana?
- Ovatko syötteet, tulosteet, tiedostot tai kyselyt monimutkaisia?
- Onko ohjelmiston suorittama laskenta monimutkaista?
- Onko koodi tarkoitettu uudelleenkäytettäväksi?
- Ovatko ohjelmiston muunnokset ja asennus mukana suunnittelussa?
- Onko ohjelmisto suunniteltu toimivaksi useina versioina eri organisaatioissa?
- Onko ohjelmiston käytettävyys keskeisessä roolissa?

FP-esimerkki

- Olkoon meillä sovellus, jolle on tehty seuraavat kokoarvot:
  - 1 yksinkertainen syöte, 1 monimutkainen
  - 2 keskivertoa tulostetta, 1 monimutkainen
  - 1 helppo kysely, 1 keskiverto, 1 monimutkainen
  - 1 yksinkertainen tiedosto, 1 monimutkainen
  - 2 monimutkaista ulkoista liittymää
- Kompleksisuuskertoimeksi on saatu 41 (14 kysymystä, vajaa 3 pistettä/kysymys).
- Näillä arvoilla saadaan:
  - CFP = (1*3+1*6)+(2*5+1*7)+(1*3+1*4+1*6)+(1*7+1*15)+(2*10) = 9+17+13+22+20 = 81
  - RCAF = 41
  - FP = CFP*(0,65+0,01*RCAF) = 81*(0,65+0,41) = 81*1,06 = 85,86

FP:n edut ja haitat

- Edut:
  - Arviointia voidaan tehdä jo ennen projektin alkua, joten FP:n arvoja voidaan käyttää projektin aloituspäätöksen tukena.
    - ”Pikalaskurit”
  - FP ei riipu käytettävistä ohjelmointikielistä tai työkaluista -> eri teknologioilla toteutettujen projektien vertailtavuus
  - FP on osoittautunut melko luotettavaksi toiminnallisuuden mittariksi.
  - FP:n laskeminen mahdollistaa *yksikkökustannusperustaisen* projektin ohjauksen ja laskutuksen -> laskutus perustuu *toimitetun toiminnallisuuden* määrään, ei tehtyihin tunteihin
    - Vrt. ”Running Tested Features” ketterän kehityksen metriikka[1]

[1] http://xprogramming.com/articles/jatrtsmetric/

FP:n edut ja haitat

- Kritiikkiä:
  - Saadut FP-arvot ovat *osittain subjektiivisia* (menetelmästä riippuen). Ne riippuvat käytetyistä kertoimista, käytetystä FP-variantista ja arvioijasta. Myös *käsityönä tehtävän laskennan kalleutta* on moitittu (laskentaa on pyritty automatisoimaan).
  - FP on tarkoitettu dataintensiivisille sovelluksille. Se antaa liian pieniä arvoja algoritmi-intensiivisille sovelluksille. Tämä tosin voidaan korjata käyttämällä modernimpia FP-variantteja.
- Huomaa, että toimintopisteiden (toiminnallisuuden) määrä on vain *yksi* projektin työmäärään ja kustannuksiin vaikuttava tekijä[1, s. 51]
  - Esimerkiksi FiSMA:n ND21 tilanneanalyysimenetelmä uuskehitykseen ottaa huomioon 21 vaikuttavaa tuottavuus-tekijää, joista tuotteen laatuvaatimukset ovat yksi osuus
  - http://www.fisma.fi/toiminta/menetelmat/

[1] Pekka Forselius: Onnistunut tietojärjestelmän hankinta. Talentum, 2013.

FP:n ja KLOC:n suhde

- Koska sekä KLOC että FP mittaavat ohjelmiston kokoa, luonteva kysymys on, onko mittareilla yhteys.
- Tällainen yhteys on olemassa. Tietyllä ohjelmointikielellä yhden toiminnallisuuspisteen toteutus vie sovelluksesta riippumatta likimain saman verran koodirivejä (”backfired” function point count).
- Seuraavassa muutamia suhdelukuja (LOC/FP) (1 KLOC = 1000 LOC). Luvut ovat suuntaa antavia (täytyy kokeellisesti vahvistaa aika ajoin):
  - Assembler: 320 (esimerkki: 85,86*320=27475 riviä koodia)
  - C: 130-90
  - Cobol: 100
  - C++: 65-55
  - Java: 45 (esimerkki: 85,86*45=3863 riviä koodia)
  - Visual Basic: 30
  - Power Builder (sovelluskehitin): 15
  - SQL: 10
- Mitä ilmaisuvoimaisempi ohjelmointikieli on kyseessä, sitä pienempi on sen LOC/FP-suhdeluku.

PROSESSIN MITTAREITA

Prosessin mittareita

- Kokomittojen määrittelyn jälkeen on helppo määritellä lukuisia prosessin tehokkuuden mittareita, joissa mitattavaa suuretta verrataan ohjelmiston kokoon. Tässä muutama:
  - Vikatiheys DD = NoD / KLOC (tai FP), missä NoD on tunnettujen* vikojen määrä (ohjelman kehityksen ja/tai käytön aikana löydetyt viat).
  - Painotettu vikatiheys: WDD = WNoD / KLOC, missä WNoD on tunnettujen vikojen määrä painotettuna niiden vakavuusasteella.
  - Vikojen poiston tehokkuus: DRE = Kehitysaikana korjatut viat / NoD, missä NoD on nyt kehitysaikana havaittujen ja tietyn käyttöjakson aikana havaittujen vikojen määrä (esim. vuodessa).
  - Kehitystyön tuottavuus: DevP = DevH / KLOC, missä DevH on projektiin käytetty kokonaistyöaika.
  - Kehitystyön tuottavuus toimintopisteillä: FDevP = DevH / FP.
  - Koodin uudelleenkäyttö: CRe = ReKLOC / KLOC, missä ReKLOC on uudelleenkäytettyjen koodirivien määrä / 1000.
- GQM-menetelmällä tai vastaavalla voidaan etsiä yrityksen kannalta hyödyllisimmät mittarit.

(*) tunnettujen vikojen (defect) lisäksi ohjelmistoon jää aina myös tuntemattomia vikoja, joita ei (vielä) ole havaittu

Prosessin mittareita: esimerkki

- Olkoon ohjelmiston koko 40.000 riviä (KLOC=40). Siitä on löydetty projektin aikana 70 ohjelmakoodivikaa, joista 11 on erittäin vakavia, 17 melko vakavia ja 42 vähäpätöisiä.
- Määritellään vikojen vakavuusasteille seuraavat painokertoimet: *vähäpätöinen*: 1, *melko vakava*: 3, *erittäin vakava*: 9.
  - (Kehityksen aikainen) Vikatiheys: DD = NoD / KLOC = 70 / 40 = 1,75.
  - Painotettu vikatiheys: WDD = CC / KLOC = (9∗11+3∗17+1∗42) / 40 = 192 / 40 = 4,8.
- Olkoot mittareilla laatukynnysarvot NoD ≤ 2 ja WDD ≤ 4. Tällöin NoD ei ole viite ohjelmiston huonosta laadusta, mutta WDD on (liikaa erittäin vakavia vikoja).

Vikojen määriin perustuva mittaus

- Kuten jo kurssin alussa todettiin, vikojen ja häiriöiden vähäinen määrä on intuitiivinen ja yleisesti käytetty laadun mittari
- Vikoihin perustuvissa mittareissa on kuitenkin pidettävä mielessä seuraavat asiat:

1. Kaikki viat eivät johda häiriöihin; vika on ohjeman koodin ominaisuus ja häiriö ohjelman suorituksen ominaisuus
   - Niitä havaitaan eri keinoin ja usein eri elinkaaren vaiheissa
   - Pitää olla tarkkana, että kaikki osapuolet ymmärtävät mitä asiaa täsmälleen mitataan

Vikojen määriin perustuva mittaus

2. Tarkoitetaanko vikatiheydellä vikojen määrää suhteutettuna ohjelmiston kokoon vai tiettyyn aikaikkunaan, jolloin vikoja havaitaan? Usein käytetään termiä ”defect rate” epätäsmällisesti edellisessä merkityksessä.
3. On tärkeää ymmärtää, miten ohjelmiston koko mitataan, jos vertaillaan *eri projektien/ohjelmistojen* vikatiheyksiä toisiinsa (KLOC variantit, FP)
4. Vikatiheyden arvoon vaikuttaa paitsi vikojen määrä, myös vikojen löytämiseen käytettyjen *resurssien määrä ja laatu* sekä *raportoinnin tarkkuus* (voi olla vaikea saada vertailukelpoisia tietoja eri yrityksistä)

Vikojen määriin perustuva mittaus

5. Vaikka ”jäännösvikojen” (residual defects) määrästä olisikin hyvä käsitys, on johtopäätöksissä oltava varovainen
   - Vian vakavuutta voi olla vaikea arvioida etukäteen
   - Käyttäjien erilaisista käyttötavoista johtuen voi olla vaikea ennustaa, mitkä viat voivat todella johtaa häiriöihin ja kuinka usein

Muita ”vikamittareita”

- System spoilage = $\frac{\text{Time to fix post-release defects}}{\text{Total system development time}}$
- Kehittäjäorganisaation löytämien vikojen kumulatiivinen määrä
- Asiakkaiden löytämien vikojen kumulatiivinen määrä
- Löytyneiden vakavien vikojen kokonaismäärä
- Vakavan vian korjaamiseen kulunut aika keskimäärin
- Eri kehitysvaiheissa (vian syntyvaihe) ja testaus-tasoilla löydetyt eri vakavuusasteiset viat / KLOC
- Kypsyystason (maturity) päättely vikojen löytymistahdin (defect discovery rate) perusteella

RAKENTEEN ELI SUUNNITTELUN MITTAREITA

Dynaamiset ja staattiset mittarit

- Tuotteen mittarit mittaavat tuotteen laatua joko (staattisesti) ohjelmakoodista tai (dynaamisesti) sen suorituksesta.
- *Suoritusaikana* tehtävät mittaukset kertovat siitä, minkälainen tuote on todellisessa käytössä. Esimerkkejä:
  - Toimintahäiriötiheys: SSFD = NYF / LOC.
  - Painotettu toimintahäiriötiheys: WSSFD = WYF / LOC, missä WYF on käytön aikana havaittujen vikojen määrä (esim. vuodessa) painotettuna niiden vakavuusasteella.
  - Käytöstä poissaolon aste: TUA = NYTFH / NYSerH, missä NYTFH on aika, jonka ohjelmisto on yhteensä vuoden aikana käyttökelvottomassa kunnossa (”kaatuneena”), ja NYSerH on ohjelmiston kokonaiskäyttöaika vuodessa.
- *Staattiset mittarit* kertovat siitä, millainen ohjelmisto on yleisesti ottaen ja kaikissa käyttötilanteissa. Esimerkki: ns. McCaben syklomaattinen kompleksisuus (cyclomatic complexity).

Suunnittelun staattiset mittarit (design, architeture)

- Suunnitteludokumenteista/-malleista ja koodista laskettavia staattisia mittareita on lukuisia. Usein ne eivät suoraan liity mihinkään laatutekijään tai osatekijään, vaan niiden arvoja pitää tulkita esimerkiksi aiempien mittausten valossa.
  - Suhteutus vikamääriin, sovellusalueeseen ja teknologisiin tekijöihin
- Yleisesti ottaen mittarit pyrkivät kvantifioimaan seuraavia ohjelmiston suunnittelun (arkkitehtuurin) ja toteutuksen ominaisuuksia eri arkkitehtuurin tasoilla
  - *Kompleksisuus* (complexity) eli rakenteellinen monimutkaisuus
  - *Kytkentä* (coupling)
  - *Yhtenäisyys* (cohesion)
  - *Koko*

Suunnittelun mittarit (design, architeture)

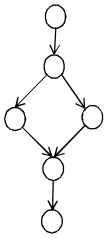
- Taustalla yleisesti hyväksytty ajatus, jonka mukaan ohjelman osan *potentiaalinen vikatiheys* ja sen *kehitys- ja ylläpitokustannukset* ovat suuremmat, jos ohjelman osa on
  - Kontrollin ja datan kulultaan mutkikas (paljon haarautumia ja riippuvuuksia),
  - Sisäisesti epäyhtenäinen,
  - Paljon riippuvuuksia muiden osien kanssa omaava
  - Tai poikkeuksellisen kookas
- Valitettavasti ei ole yhtä yleismittaria, joka pystyisi tuottamaan (mittausteoreettisesti) pätevän mittaluvun kaikille näille ominaisuuksille ja joka olisi suoraan kytkettävissä tuotelaadun piirteisiin
  - Täytyy valita kehitettävän ohjelmiston kannalta oleelliset ominaisuudet, valita niille sopivat mittarit, tehdä huolelliset mittaukset ja määritellä kokemuksen kautta korrelaatio tavoiteltuun laatutasoon
  - ISO 25023 antaa joitakin suosituksia mittareista

Suunnittelun mittarit (design, architeture)

- Tunnetuimmat koodista laskettavat mittarit ovat Chidamberin ja Kemererin oliopohjaiset mittarit (1994). Niitä kutsutaan *CK-mittareiksi* (CK-metrics).
- Osa CK-mittareista (ja vastaavista) voidaan laskea myös suunnittelutason kaavioista (esim. luokkakaavioista).
  - S.R. Chidamber, C.F. Kemerer: A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6, 1994, 476-492.
- Myös McCaben *syklomaattinen kompleksisuus* (cyclomatic complexity) on hyvin tunnettu
- Muitakin mittareita on kehitetty, esimerkiksi R.C. Martinin *pakkaustason kytkentää* ja *koheesiota* koskevat mittarit
  - http://en.wikipedia.org/wiki/Software_package_metrics

Syklomaattinen kompleksisuus

- McCaben määrittelemä syklomaattinen kompleksisuus mittaa ohjelmakoodin mutkikkuutta. Sen avulla voi mm. arvioida tarvittavaa ylläpidon määrää ja suunnitella (riippumattomiin polkuihin perustuvaa) testausta.
- Mittaus perustuu ohjelmakoodia vastaavaan vuokaavioon (flow graph) ja sen haarautumiin. Vuokaaviossa solmut vastaavat ohjelmakoodin lauseita ja kaaret niiden välistä kontrollinkulkua.

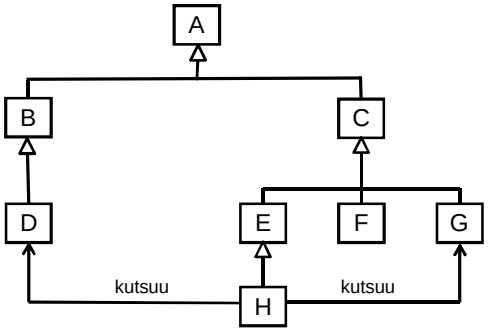


Syklomaattinen kompleksisuus:
riippumattomien polkujen määrä (2) tai
kaarien määrä – solmujen määrä + 2 =
6-6+2 = 2

CK-mittarit

CK-mittarit mittaavat ohjelmakoodista, suunnitelmasta tai ohjelmistoarkkitehtuurista oliopiirteitä. Mittareita on kuusi:

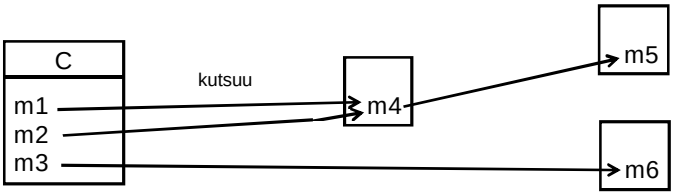
- *Painotettu luokan metodien määrä* (Weighted Methods per Class), WMC
  - Luokan metodien kompleksisuuksien (esim. syklomaattinen McCabe) summa.
  - Usein yksinkertaistettuna: metodien lukumäärä.
- *Perintäpuun syvyys* (Depth of Inheritance Tree), DIT
  - Luokkahierarkiassa pisimmän reitin pituus luokkasolmusta juureen.
- *Luokan lasten lukumäärä* (Number Of Children), NOC
  - Luokkahierarkiassa luokan välittömien aliluokkien lukumäärä.
- *Luokkien välinen kytkentä* (Coupling Between Object classes), CBO
  - Luokkaan sidottujen muiden luokkien määrä. Kahden luokan välinen kytkentä tarkoittaa luokkien välistä metodikutsun tai muuttujaviittauksen kautta syntyvää yhteyttä.
- *Luokasta kutsuttujen metodien lukumäärä* (Response For a Class), RFC
  - Luokan metodien ja niistä suoraan kutsuttujen muiden metodien summa.
- *Metodien yhtenäisyyden puute* (Lack of COhesion in Methods), LCOM
  - Luokan erillisten metodiparien määrä vähennettynä ei-erillisten metodiparien määrällä. (LCOM=0, jos ei-erillisiä pareja on enemmän kuin erillisiä pareja.)
    Kaksi metodia ovat erilliset, jos ne eivät viittaa yhteisiin luokan attribuutteihin.



DIT (A) = 0, DIT (B) = DIT (C) = 1, DIT (H) = 3

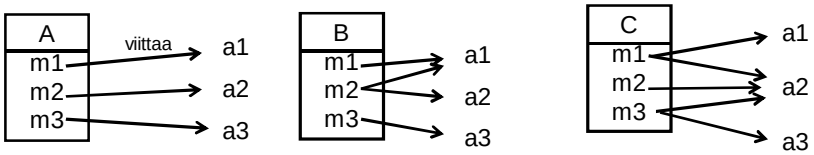
NOC (A) = 2, NOC (C) = 3, NOC (H) = 0

CBO(H) = 2



RFC(C) = 3 + 2 = 5

WMC(C) = kompleksisuus(m1) + kompleksisuus(m2) + kompleksisuus(m3)



LCOM(A) = 3

(m1,m2)=Ø
(m1,m3)=Ø
(m2,m3)=Ø

LCOM(B) = 1

(m1,m2)={a1}
(m1,m3)=Ø
(m2,m3)=Ø

LCOM(C) = 0

(m1,m2)={a2}
(m1,m3)={a2}
(m2,m3)={a2}

CK-mittarien ja laadun yhteys

- CK-mittarit mittaavat olio-ohjelmiston staattisia ominaisuuksia, mutta mittaavatko ne ohjelmiston laatua? Tämä ei ole aivan triviaali kysymys.
- CK-mittareiden laadunennustuskyvystä on tehty paljon tutkimuksia. Tulokset vaikuttavat seuraavilta:
  - CK-mittarit ennustavat melko hyvin, mitkä luokat tulevat aiheuttamaan ongelmia (esim. niiden luotettavuus ja ylläpidettävyys ovat alhaiset).
    - LCOM ei ennusta kovin hyvin ongelmallisia luokkia
    - suuret DIT-, RFC-, NOC- ja erityisesti CBO-arvot ennustavat jossain määrin ongelmaluokkia (Briand et al. 1999, erityisesti ”import coupling”)
  - CK-mittarit ovat toisistaan riippumattomia (Basili et al., 1995) tai osalla CK-mittareista on keskenään vahva riippuvuus (Chidamber et al., 1997).
    - Tulokset ovat ristiriidassa keskenään. Lopullinen vastaus on toistaiseksi auki.
  - CK-mittarit ennustavat jossain määrin tuottavuutta (Kan, 2003).

Mittarien ongelmia

- Periaatteessa mittaus ja mittarit ovat erinomainen laadunvarmistustekniikka, *kunhan mittaukset ovat objektiivisia ja keskenään vertailukelpoisia*. Valitettavasti näin ei aina ole.
- Kun mittaus on subjektiivista tai tilanneriippuvaa, saadut tulokset ovat tulkinnanvaraisia. Näin on esimerkiksi KLOC-mittarin laita:
  - ohjelmointityyli vaikuttaa rivien määrään
  - kommenttien määrä vaikuttaa rivien määrään
  - ongelman vaikeus vaikuttaa koodin laatuun, mutta koodirivien määrä ei suoraan kerro vaikeustasosta
  - jne.

Mittarien tulkinta

- Mittausten subjektiivisuuden lisäksi mittariarvojen tulkinnassa on usein otettava huomioon myös muuta tilannetietoa. Klassinen esimerkki:
  - Olkoon testaustiimin A löytämien vikojen määrä *a* ja testaustiimin B löytämien vikojen määrä *b.* Jos *a* > *b*, niin onko testaustiimi A parempi kuin testaustiimi B?
  - Ei välttämättä. Vaihtoehtoja on useita:
    - A:n testaaman ohjelman vikatiheys oli suurempi kuin B:n
    - A käytti enemmän aikaa testaukseen
    - A:n tekemät testit olivat B:n testejä kevyempiä (testasivat kaikki yhtä ja samaa triviaalia asiaa) jne.
- Mittarien avulla saatujen arvojen objektiivinen ja kontekstin huomioon ottava tulkinta on yhtä tärkeää kuin itse mittaus
  - Syy-seuraussuhteiden löytämisessä pitää yleensä ottaa huomioon *monia tilanne- ja prosessitekijöitä*. Mittarit voivat antaa liian pelkistetyn kuvan!

Laatu ja arkkitehtuuri

- Laatuvaatimuksilla ja ohjelmiston arkkitehtuurilla on ”kohtalonyhteys”
  - Jos ohjelmistolla on vaativia laatuvaatimuksia (esim. suorituskyky, skaalautuvuus, tuoteperheen rakentaminen), arkkitehtuuri tulisi suunnitella vastaamaan näihin vaatimuksiin
  - Laatuvaatimukset on määriteltävä siten, että niille annetaan *mitattavat tavoitetasot* (vasteaika tietyssä tilanteessa, käsiteltyjen tapahtumien määrä), jotta tavoitteiden täyttyminen voidaan todentaa (myös prototyypillä tai simulaatiolla) -> projektikohtainen mittari
- On monia dokumentoituja suunnitteluratkaisuja (patterneja, taktiikoita), jotka tähtäävät suoraan tiettyjen laatupiirteiden tason parantamiseen
  - Kts. esimerkiksi *Ohjelmistoarkkitehtuurit* –kurssin luentomateriaali S2014, 5. luento