KOON MITTARIT

Ohjelmiston kokomittarit

- Ohjelmiston kokoa käytetään monessa johdetussa laatumittarissa *normalisoimaan* eri kohteiden muut mitat keskenään vertailukelpoisiksi. Vaikka erilaisia kokomittareita on monia, kaksi on ylitse muiden:
  - KLOC (Kilos Lines Of Code): Kuinka monta tuhatta *koodiriviä* on ohjelmistossa.
  - FP (Function Points), toimintopisteet: Kuinka paljon *toiminnallisuutta* on ohjelmistossa.
- Muita kokomittareita:
  - Montako lausetta on ohjelmistossa (KLOC-variantti).
  - Montako luokkaa/metodia/funktiota on ohjelmistossa.
  - Montako *riippumatonta polkua* (independent path) on ohjelmistossa tai (yleensä) sen osassa.
    - Riippumaton polku on sellainen suoritusreitti ohjelmistossa, että sitä ei voi saada yhdistämällä muista riippumattomista poluista.

KLOC

- KLOC on yleisin ohjelmiston kokomittari. Sillä mitataan ohjelmiston kokoa koodiriveinä joko kommenttien kanssa tai ilman niitä.
- KLOC on (suhteellisen) selkeä mittari, jota käytetään koon lisäksi monimutkaisuuden sijaismittana (surrogate): mitä isompi ohjelmisto/ohjelma/luokka/metodi tms. on, sitä mutkikkaampi se yleensä on (ja esimerkiksi sitä hankalampi testata ja ylläpitää).
- KLOC:n heikkous on sen ohjelmointikieliriippuvuus. Eri ohjelmointikielillä toteutetuista ohjelmistoista lasketut KLOC-luvut eivät ole keskenään verrannollisia.
- Myös koodaustyylit vaikuttavat jonkin verran koodirivien määrään, mutta isoissa ohjelmistoissa erojen suhde koodirivien kokonaismäärään on melko pieni.
  - KLOC:n sijaan kannattaa laskea esimerkiksi puolipisteitä (so. lauseita), jos rivien määrän vaihtelu osoittautuu ongelmaksi.

KLOC

Start Page | CartBean.java

Source | History

```
27          if (person == null) {
28              throw new BookException("Null person not allowed.");
29          } else {
30              customerName = person;
31          }
32
33          customerId = "0";
34          contents = new ArrayList<>();
35      }
36
37      @Override
38      public void initialize(String person, String id) throws BookException {
39          if (person == null) {
40              throw new BookException("Null person not allowed.");
41          } else {
42              customerName = person;
43          }
44
45          IdVerifier idChecker = new IdVerifier();
46          if (idChecker.validate(id)) {
47              customerId = id;
48          } else {
49              throw new BookException("Invalid id: " + id);
50          }
51
52          contents = new ArrayList<>();
53      }
54
```

Montako
- Koodiriviä?
- Lausetta?
- Puolipistettä?
- Kommenttiriviä?
- Tyhjiä rivejä?
- Jotain muuta?

KLOC-ongelmia

- Koska KLOC riippuu käytetystä ohjelmointikielestä ja koodaustavasta, KLOC-arvot ovat monen, laadun kannalta vähemmän tärkeän muuttujan summa.
- Lisäksi KLOC:lla on ikävä ominaisuus: se on nimensä mukaisesti koodirivien määrä, joten **sitä ei voida laskea, ennen kuin koodi on kirjoitettu.**
- Miten lasketaan kirjastoista/kehyksistä *uudelleenkäytetyt* koodirivit? Uuden koodin määrä voi olla suhteessa mitätön.
  - Lähdekoodia ei edes usein saatavissa, eikä tiedetä mitkä rivit laskea
- Nykyaikaisten online-sovellusten ja palveluiden toteuttamisessa saatetaan käyttää useita ohjelmointi- ja skriptikieliä sekä deklaratiivisia määrityksiä
  - Koodia upotetaan myös esim. templaatteihin, joista taustalla oleva järjestelmä generoi esim. html:ää ja javaa tai javascriptiä
  - Miten nämä lasketaan?

KLOC-ongelmia

- KLOC-arvoja voidaan arvioida, mutta arviot ovat summittaisia. Olisi parempi, jos jo määrittely- tai suunnitteluvaiheessa ohjelman koolle voisi antaa luotettavan arvion.
- Määrittelystä laskettava kokoarvio on mahdollista *toimintopisteillä* (function points). Niiden avulla jo melko yleisistä määrityksistä voidaan arvioida, miten paljon toiminnallisuutta ohjelmistossa on.
  - Arviot tarkentuvat projektin edetessä ja ohjelmiston vaiheittain valmistuessa

Toimintopisteet

- Toimintopisteiden laskemisen pääidea on tuloksen riippumattomuus toteutusteknologiasta: lasketaan suoraan *toiminnallisuuden määrä:* fp(<ohjelmiston spesifikaatio>) → Z
  - 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.
  - Alkuperäinen käyttötarkoitus: auttaa kehittäjiä arvioimaan työmääriä
  - 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 (ISO/IEC 20926) 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?

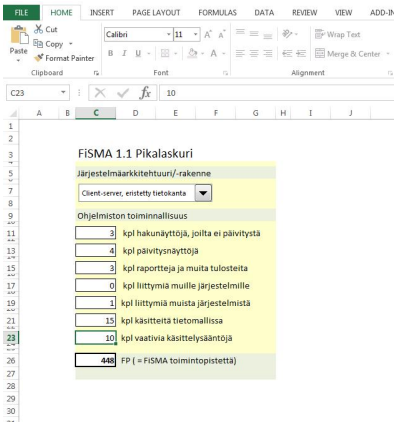
FP-esimerkki

- 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

SNAP

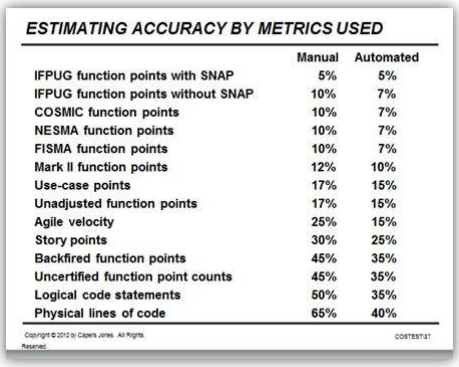
- IFPUG on täydentänyt ohjelmiston koon määrittelyä toimintopisteiden lisäksi *SNAP-pisteillä*
  - Software Non-Functional Assessment Process
  - http://www.ifpug.org/about-ifpug/about-snap/
  - Lasketaan pisteet laatu- eli ei-toiminnallisille vaatimuksille
    - Skaalautuvuus, turvallisuus, tehokkuus, siirrettävyys jne.

FISMA -pikalaskuri



- FISMA FP käyttää järjestelmän käyttäjän ymmärtämää kieltä toimintojen määrittelyssä
- Pikalaskuria käytetään aikaiseen arviointiin projektin valmisteluvaiheessa
  - Luetellaan toiminnot ja käytetään oletuskokoja
- Määritysten tarkentuessa ja implementoinnin edetessä voidaan tehdä uusia tarkentuvia arvioita
- Tarkentuvia arvioita voidaan käyttää mm. muutosten hinnan arviointiin ja jo toimitettujen osuuksien arvon määrittelyyn
- http://www.fisma.fi/toiminta/menetelmat/

FP-estimoinnin tarkkuus työmäärän (effort) arvioinnissa



ESTIMATING ACCURACY BY METRICS USED

	Manual	Automated
IFPUG function points with SNAP	5%	5%
IFPUG function points without SNAP	10%	7%
COSMIC function points	10%	7%
NESMA function points	10%	7%
FISMA function points	10%	7%
Mark II function points	12%	10%
Use-case points	17%	15%
Unadjusted function points	17%	15%
Agile velocity	25%	15%
Story points	30%	25%
Backfired function points	45%	35%
Uncertified function point counts	45%	35%
Logical code statements	50%	35%
Physical lines of code	65%	40%

COSTESTIM

http://www.ifpug.org/about-ifpug/faqs-2/#One

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.

Story points?

- Ketterässä kehityksessä käytetty tehtävän laajuuden/työmäärän/vaikeuden arvioinnin apuväline
- Suhteellinen arvio, jonka merkityksen tiimit itse määrittävät (~ järjestysmitta-asteikko)
- https://www.agilealliance.org/glossary/points-estimates-in/

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.
  - Jotain, mitä voidaan laskea (”perusmitat”)
  - 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)

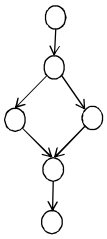
- Taustalla yleisesti hyväksytty ajatus, jonka mukaan ohjelman osan *potentiaalinen vikatiheys* ja sen *kehitys- ja ylläpitokustannukset* ovat suuremmat, jos ohjelman osa on
  - Kontrollin ja datan kulultaan mutkikas (paljon haarautumia ja riippuvuuksia),
  - Sisäisesti epäyhtenäinen,
  - Paljon riippuvuuksia 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.

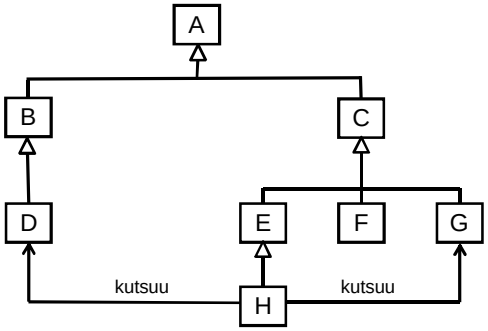


Syklomaattinen kompleksisuus: riippumattomien polkujen määrä (2) tai *kaarien määrä – solmujen määrä* + 2 = 6-6+2 = 2

CK-mittarit

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

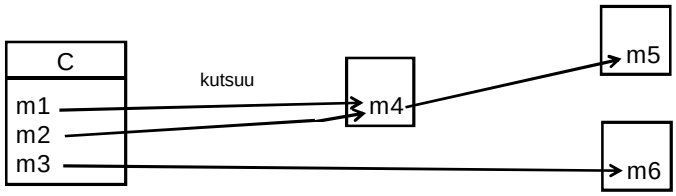
- *Painotettu luokan metodien määrä* (Weighted Methods per Class), WMC
  - Luokan metodien kompleksisuuksien (esim. syklomaattinen McCabe) summa.
  - Usein yksinkertaistettuna: metodien lukumäärä.
- *Perintäpuun syvyys* (Depth of Inheritance Tree), DIT
  - Luokkahierarkiassa pisimmän reitin pituus luokkasolmusta 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

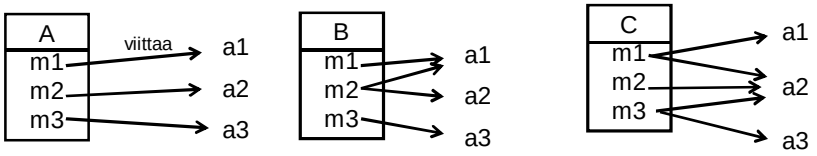
NOC (A) = 2, NOC (C) = 3, NOC (H) = 0

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 jossain määrin, 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).

No mitä sitten kannattaisi laskea?

- Nyrkkisääntö (olio-ohjelmistoille)
  - Coupling Between Objects
  - Size of class (sub-classes, members, methods)
  - McCabe complexity for classes
- Raja-arvot?
  - Muista selvästi ylöspäin poikkeavat luokat/metodit tutkittava tarkemmin
  - Puuttumiskynnykset haettava kokemuksen kautta
- Pelkkä numeroon tuijottaminen ei toimi!

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 / KLOC *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