Ohjelmistoprosessit ja ohjelmistojen laatu

Jukka Paakki
Helsingin yliopisto
Tietojenkäsittelytieteen laitos

1. Prosessi ja laatu

- Ohjelmistojen kehitystyö perustuu *prosessiin* (process):
  - Prosessi on systemaattinen lähestymistapa tuotteen kehitystyöhön tai tietyn tehtävän tekemiseen (Osterweil, 1987).
- Toisin sanoen prosessi on *ohjeisto.* Prosessi kertoo, millä tavoin jokin asia pitää tehdä.
- Prosessin ilmentymä on *projekti* (project).
  - Projekti on tehtävä tai tehtäväjoukko, jossa seurataan yhtä tai useampaa prosessia.

Ohjelmistotuote ja tuotteen laatu

- Projektin tuloksena saadaan *ohjelmistotuote* (software product) eli ohjelmisto.
- *Ohjelmiston laatu* (software quality) tarkoittaa ohjelmiston käyttökelpoisuutta.
  - Määritelmä on yksinkertaistus, koska ”käyttökelpoisuus” on moniselitteinen termi. Luvussa 2 käsitellään yleisesti käytettyjä eksaktimpia määritelmiä.
- Ohjelmiston laatuun vaikuttavat
  - käytetty prosessi (menetelmät, tekniikat, työkalut)
  - infrastruktuuri (organisaatio, työympäristö)
  - sidosryhmät (pääasiassa ohjelmistotuotteen tekijät)

Prosessin vaikutus laatuun

- Prosessi vaikuttaa laatuun kahdella tavalla:
  - Mitä *systemaattisempi* prosessi, sitä *tasalaatuisempia* tuotteita kehitetään.
  - Mitä *sopivampi* prosessi, sitä *edullisemmin* kehitetään laadukkaita tuotteita.
- Prosessin systemaattisuus tarkoittaa selkeyttä ja *mitattavuutta*
  - Mitattavuus tarkoittaa, että prosessista voidaan laskea tunnuslukuja.
- Prosessin sopivuus tarkoittaa prosessin kelpoisuutta tietyn tuotteen tekoon.

Infrastruktuurin vaikutus laatuun

- Infrastruktuuri vaikuttaa laatuun välillisesti. Mitä parempi on työympäristö, sitä motivoituneempia ovat työntekijät.
- Infrastruktuurin vaikutus on merkittävä pääasiassa alaspäin. Erittäin huonossa työympäristössä ei synny laadukasta jälkeä.
- Huippuluokan infrastruktuurilla voidaan nostaa tuotteen laatua, mutta pääasiassa riittää, että infrastruktuuri on riittävän laadukas.
  - Työntekijöillä on työrauha
  - Tarvittavat laitteet ja ohjelmistot ovat saatavilla

Ihmisten vaikutus laatuun

- Ylivoimaisesti tärkein ohjelmiston laatuun vaikuttava tekijä ovat sitä kehittävät ihmiset:

  Process is only a second-order effect. The unique people, their feelings, qualities, and communication are more influential.

  Some problems are just hard, some people are just difficult. These methods are not salvation. (Larman, 2004)

- Vaikka ihmiset ovat tärkein laatuun vaikuttava tekijä, hyvilläkin työntekijöillä epäonnistumisen riski kasvaa suureksi ilman kunnollista prosessia.
  - Oikea prosessi ohjelmistotuotteelle helpottaa ohjelmiston kehitystyötä aivan samoin kuin oikeat ohjeet helpottavat kirjahyllyn kokoamista. Työssä tarvitaan sekä ammattitaitoa (ihmisiä) että oikeita ohjeita (prosessia).

2. Ohjelmistojen laatu

- Edellä ohjelmiston laatu määriteltiin yksinkertaistaen ohjelmiston käyttökelpoisuutena.
- Käyttökelpoisuus on kuitenkin abstrakti suure, jolla ei ole selviä arvoja. Tarvitaan jotain täsmällisempää.
- Käyttökelpoisuutta arvioitaessa täytyy tietää, mitä mahdollisia ongelmia ohjelmistoihin voi liittyä. Myös niille tarvitaan tarkemmat määritelmät.
- Ohjelmistojen ”laatu” on osoittautunut varsin vaikeasti määriteltäväksi asiaksi.

Ohjelmisto

- Ohjelmiston laatuun vaikuttaa ohjelmiston määritelmä. Mitä tarkoitetaan ”ohjelmistolla”?
- Intuitiivisesti ajatellen ohjelmisto on yhtä kuin suoritettava ohjelmakoodi.
  - Laadun kannalta tämä määritelmä ei riitä, sillä tällöin ei huomioida käytettävän datan, dokumentaation ja toimintatapojen laatua.
- IEEE (1991) määrittelee ohjelmiston seuraavasti:
  - Software is: Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Laadun kannalta oleelliset ohjelmistotekijät

- IEEE tunnistaa neljä laadun kannalta oleellista ohjelmistotekijää:
  - ”Computer programs”: Ohjelmakoodi. Ilman suoritettavaa koodia ei voi olla ohjelmistoa.
  - ”Procedures”: Ohjelmistoa käyttävien sidosryhmien toimintatavat. Ohjelmisto ei toimi yksin, vaan osana toimintaympäristöä: sitä käytetään. Käyttäjä voi olla ihminen tai toinen ohjelmisto.
  - ”Documentation”: Ohjelmistoa kuvaavat dokumentit. Dokumentit vaikuttavat välillisesti ohjelmiston ominaisuuksiin, kuten käytettävyyteen ja ylläpidettävyyteen.
  - ”Data pertaining to the operation of a computer system”: Koodin suoritukseen tarvittavat syötteet, tietorakenteet, tiedostot jne.
    - Galin (2004) sijoittaa tänne testitapaukset, koska ne vaikuttavat ohjelmiston suoritukseen. IEEE:n määritelmää seuraten testitapaukset ovat lähinnä dokumentaatiota.

2.1. Ohjelmistolaadun määritelmät

- Ohjelmistojen laadun määrittely ei ole helppo tehtävä, sillä eri ihmisille laatu tarkoittaa eri asioita:
  - ”Parasta mitä rahalla saa”
    - (Mitä ”paras” tarkoittaa?)
  - ”Täyttää kaikki vaatimukset”
    - (Mistä tiedämme, että kaikki vaatimukset on määritelty?)
  - ”Toimii oikein”
    - (Onko ”oikein” sama kuin ”määritysten mukaisesti”? Entä jos määrityksissä on virheitä?)
  - ”Laadukkaan ohjelmiston tunnistaa, kun sellainen tulee kohdalle.”
    - (Laadun havaitsee vasta jälkikäteen? Tällainen määritelmä ei auta kehittämään laadukkaita ohjelmistoja!)

Mitä ”ohjelmistojen laatu” tarkoittaa?

- Intuitiivisesti laadukas ohjelmisto on sellainen, joka ”toimii oikein” ja ”odotusten mukaisesti”
  - Valitettavasti ”toimii oikein” ja ”toimii odotusten mukaisesti” ovat moniselitteisiä termejä.
    - Toimiiko ohjelmisto oikein, jos se toimii käyttöohjeensa mukaisesti mutta ei tee mitään järkevää?
    - Toimiiko ohjelmisto odotusten mukaisesti, jos sen avulla työntekijät voivat toteuttaa vanhat tehottomat rutiinit tarkalleen samalla tavalla kuin ennen, vaikka tehokkaampiakin toimintatapoja voisi käyttää?
- Laadun määrittely formaalisti on yllättävän vaikeaa. Tämän johdosta kirjallisuudessa on esitelty useita hiukan toisistaan eroavia laadun määritelmiä.

IEEE:n laatumääritelmä

- IEEE (1991) määrittelee ohjelmistojen laadun seuraavasti:
  - Ohjelmiston laatu on
    - aste, millä järjestelmä, komponentti tai prosessi täyttää sille asetetut vaatimukset ja
    - aste millä järjestelmä, komponentti tai prosessi täyttää asiakkaan tai käyttäjän tarpeet ja odotukset.
- Ts. IEEE jakaa laadun kahteen komponenttiin:
  - Miten tarkasti ohjelmisto toteuttaa sille kirjatut vaatimukset, eli ohjelmiston spesifikaation (Software Requirements Specification, SRS).
  - Miten hyödyllinen ohjelmisto on loppukäyttäjälle.

Juranin laatumääritelmä

- Juran (1988) määrittelee laadun seuraavasti:
  1. Laatuun sisältyvät ne tuotteen ominaisuudet, jotka täyttävät asiakkaiden tarpeet ja sen kautta pitävät asiakkaat tyytyväisinä.
  2. Laadukkaassa ohjelmistossa ei ole puutteita.
- Juranin määritelmässä laatua lähestytään käyttäjän näkökulmasta. Tyytyväinen käyttäjä on laadun lopullinen tavoite.
- Koska ohjelmisto on enemmän kuin pelkkä koodi, Juranin määritelmän mukaan esimerkiksi laadukkaan ohjelmiston vaatimusmäärittelyssä ei saa olla puutteita.
- Koska Juranin määritelmän mukaan tuotteen laatu mitataan asiakkaan kannalta oleellisten piirteiden laatuna, kehittäjien pitäisi periaatteessa todistaa, miten hyvin ohjelmisto täyttää loppukäyttäjien tarpeet. Tämä voi olla mahdotonta.

Pressmanin laatumääritelmä 1

- Pressman (2000) määrittelee laadun seuraavasti:
  - Ohjelmiston laatu tarkoittaa
    - yhdenmukaisuutta täsmälleen määriteltyjen toiminnallisten vaatimusten ja suorituskykyvaatimusten kanssa
    - tarkasti määriteltyjä kehitystyön standardeja
    - implisiittisiä ominaisuuksia, joiden odotetaan olevan mukana kaikissa ammattimaisesti kehitetyissä ohjelmistoissa.
- Pressmanin määritelmä on kolmesta määritelmästä täsmällisin. Se sisältää kolme vaatimusta:
  - ohjelmiston on toteutettava vaatimusmäärittelyssä spesifioidut toiminnot
  - kehitystyössä on käytettävä standardoitua prosessia
  - kehitystyössä on noudatettava ammattilaisten hyviksi havaitsemia menetelmiä (best practices), vaikka niitä ei olisi erityisesti listattu spesifiointidokumentissa eikä käytettävässä prosessissa

Pressmanin laatumääritelmä 2

- Kaikki Pressmanin vaatimukset voidaan varmentaa prosessista ja ohjelmistotuotteesta.
  - Toiminnalliset ja suorituskykyvaatimukset on kirjattu spesifiointidokumenttiin. Asiakas vastaa dokumentoitujen vaatimusten oikeellisuudesta.
  - Standardoitu prosessi on kirjattu *laatukäsikirjaan* (software quality assurance plan).
  - Hyviksi havaitut menetelmät on kirjattu alan uusimpaan kirjallisuuteen ja mahdollisesti laatukäsikirjaan.
- Toisaalta Pressmanin määritelmä ei ota mitään kantaa asiakkaan tyytyväisyyteen.
  - Tyytyväisyyttä ei määritellä spesifiointidokumentissa.
  - Standardoitu prosessi ei takaa tyytyväistä asiakasta.
  - Tyytyväisyys on yhdistelmä hyviä kirjattuja vaatimuksia ja implisiittisiä vaatimuksia, mutta hyviksi havaitut menetelmät eivät yksin takaa tyytyväisyyttä.
- Kaikesta huolimatta Pressmanin määritelmä on ehkä toimivin laatumääritelmä.

2.2. Laadukkaan ohjelmiston vaatimukset

- Modernit laadun määritelmät huomioivat kirjatut vaatimukset ylitse muiden: laadukas tuote täyttää sille kirjatut vaatimukset.
- ”Kirjatut vaatimukset” ovat
  - toimintoja tai palveluja, siis transformaatioita syötteestä tulosteeksi: *toiminnallisia* vaatimuksia (functional requirements) ja
  - ohjelmiston tai sen osan toimintaan vaikuttavia läpileikkaavia ominaisuuksia: *ei-toiminnallisia* vaatimuksia (non-functional requirements).

Toiminnalliset vaatimukset

- Toiminnalliset vaatimukset ovat laadun kannalta kohtuullisen selkeitä.
  - Jos toiminto tai palvelu toteuttaa asiakkaan tehtävän, se parantaa ohjelmiston laatua.
  - Ohjelmisto ei voi olla laadukas, jos se ei toteuta asiakkaan ohjelmistolta tarvitsemia tehtäviä.
- Valitettavasti pelkät toiminnalliset vaatimukset eivät riitä laadulle.
  - Vaikka ohjelmisto toteuttaisi kaikki asiakkaan vaatimat tehtävät, ohjelmisto ei välttämättä ole laadukas. Toteutuksen taso ratkaisee.

Ei-toiminnalliset vaatimukset 1

- Toiminnallisuudessa ei pääosin oteta kantaa siihen, *miten hyvin* toiminto toteuttaa sille määrätyn tehtävän. Siksi toiminnalliset vaatimukset eivät yksin riitä takaamaan laadukasta tuotetta.
- Ohjelmiston hyvyyttä mitataan myös ei-toiminnallisten vaatimusten avulla. Ne määrittelevät toimintojen ympärille turvaverkon, joka mahdollistaa ohjelmiston järkevän toiminnan eri tilanteissa.
- Ei-toiminnalliset vaatimukset määrittelevät rajoitukset ja reunaehdot, joiden puitteissa ohjelmisto toimii.

Ei-toiminnalliset vaatimukset 2

- Osa toiminnallisten vaatimusten kuvausta on itse asiassa ei-toiminnallisia vaatimuksia.
  - Toiminnalliseen vaatimukseen liittyy hallintavaatimuksia osana toimintoa. Esimerkiksi tehtävän ”Tallenna tiedosto levylle” odotetaan selviävän tilanteesta, jossa levy on täynnä.
  - Hallintavaatimukset eivät ole osa palvelua vaan palveluiden ympärillä olevaa turvaverkkoa. Kyseessä on ei-toiminnallinen vaatimus *luotettavuudesta* (reliability): ohjelmiston on toivuttava poikkeustilanteista ilman että käsiteltyä tietoa katoaa.

Asiakkaan tyytyväisyys ja ohjelmiston ominaisuudet

- Usein asiakkaan tyytymättömyys ohjelmistoon ei johdu siitä, että siitä puuttuisi ominaisuuksia.
  - Toisin sanoen toiminnalliset vaatimukset ovat yleensä kunnossa.
- Itse asiassa useimmissa ohjelmistoissa on ”liikaa” ominaisuuksia. Ominaisuusjoukko on niin laaja, että niiden hallinta on hankalaa.
  - Tällöin kysymyksessä on toteutumaton ei-toiminnallinen käytettävyysvaatimus.

Asiakkaan tyytyväisyys ja laatu

- Sen sijaan tyytymättömyyttä tulee yleisemmistä asioista: ohjelmistoa on vaikea käyttää, ohjelmisto ei toimi oikein, ohjelmisto on hidas, ohjelmiston personointi tai ylläpito on vaikeaa.
  - Toisin sanoen ei-toiminnalliset vaatimukset eivät ole kunnossa.
- Täten ei-toiminnalliset vaatimukset ja ohjelmiston laatu ovat vahvasti yhteydessä toisiinsa. Käyttäjän kokemaan laatuun vaikuttaa voimakkaasti se, miten hyvin sekä kirjatut että *kirjaamattomat* ei-toiminnalliset vaatimukset toteutuvat ohjelmistossa.

2.3. Ohjelmistojen laatutekijät

- Ei-toiminnalliset vaatimukset ovat hankalia, koska asiakas ei osaa antaa niistä kattavaa toivelistaa.
- Onneksi ei-toiminnallisia vaatimuksia voidaan ryhmitellä niiden sisällön perusteella *laatutekijöiksi* (quality factors).
- Jos ohjelmisto on laadukas, se täyttää sille asetetut toiminnalliset vaatimukset ja laatutekijöiden määrittelemät ei-toiminnalliset vaatimukset.
  - Laatutekijöiden määrittämiä ei-toiminnallisia vaatimuksia on paljon ja ne ovat osin ristiriidassa keskenään. Tämän takia vaatimusmäärittelyvaiheessa ei-toiminnalliset vaatimukset on syytä priorisoida.
  - Mitä korkeampi on ei-toiminnallisen vaatimuksen prioriteetti, sitä enemmän projektissa on käytettävä resursseja sen toteuttamiseen.

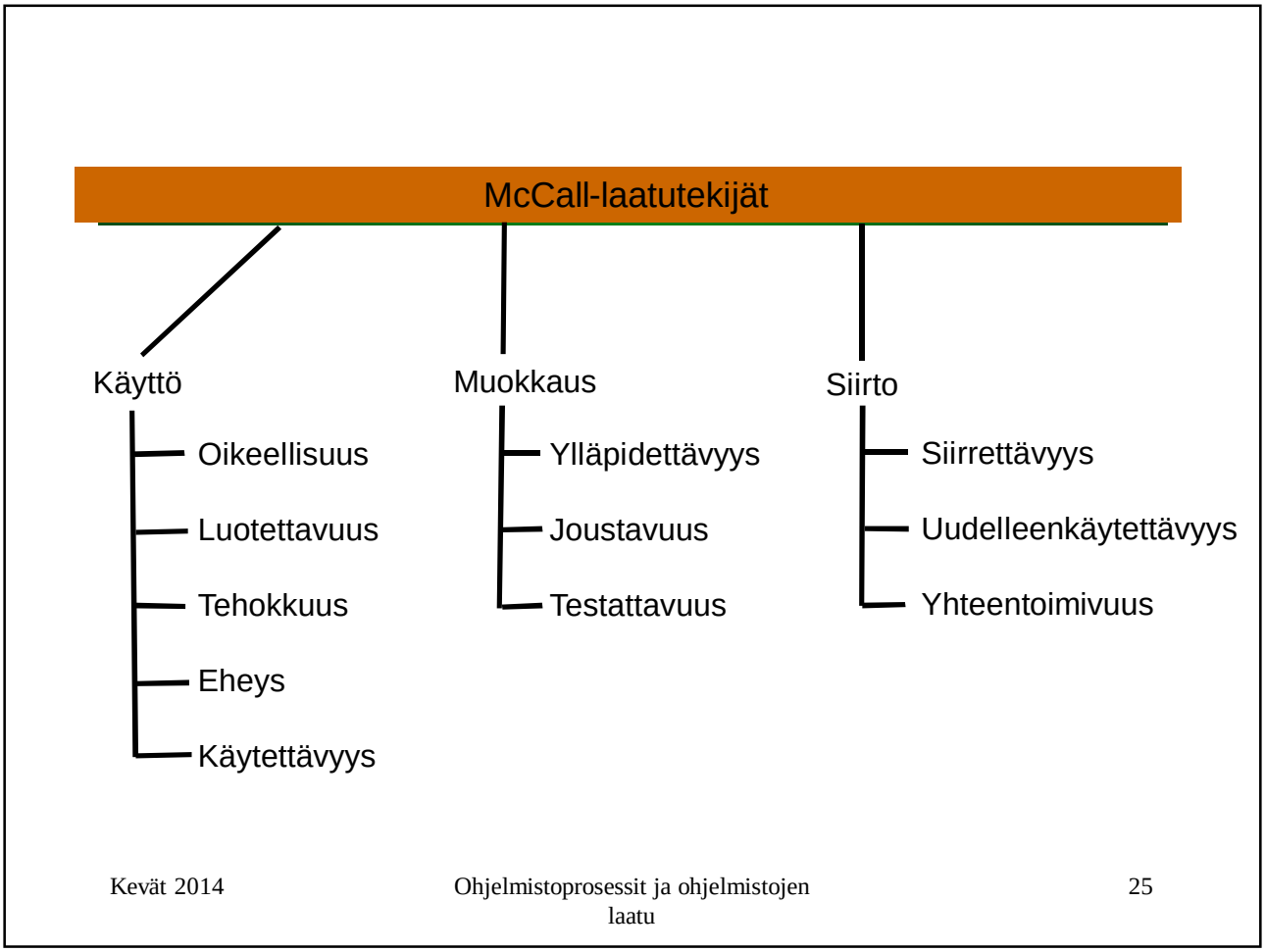
Laatutekijöiden luokittelu

- Koska laatutekijät helpottavat huomattavasti ei-toiminnallisten vaatimusten määrittelyä, niille on tehty erilaisia luokitteluja.
- McCall esitteli klassisen laatutekijöiden luokittelun vuonna 1977. Vaikka luokittelu on yli 30 vuotta vanha, se on edelleen ajan tasalla.
- McCall luokittelee ohjelmiston vaatimukset 11 laatutekijäksi kolmeen luokkaan.

McCallin laatutekijämalli

Laatutekijäluokat:

1. Tuotteen *käytön* laatutekijät (product operation factors), 5 kpl
   - nämä vaikuttavat ohjelmiston käyttöön
2. Tuotteen *muokkauksen* laatutekijät (product revision factors), 3 kpl
   - nämä vaikuttavat ohjelmiston ylläpitoon
3. Tuotteen *siirron* laatutekijät (product transition factors), 3 kpl
   - nämä vaikuttavat ohjelmiston toimivuuteen eri alustoilla


McCall-laatutekijät
Käyttö
Oikeellisuus
Luotettavuus
Tehokkuus
Eheys
Käytettävyys
Muokkaus
Ylläpidettävyys
Joustavuus
Testattavuus
Siirto
Siirrettävyys
Uudelleenkäytettävyys
Yhteentoimivuus
Kevät 2014
Ohjelmistoprosessit ja ohjelmistojen laatu
25

Käytön laatutekijät 1

- *Oikeellisuus* (correctness)
  - Oikeellisuusvaatimus on toiminnallisten vaatimusten laatutekijä: ohjelmisto on oikeellinen, jos se on spesifikaationsa mukainen.
  - Oikeellisuus on absoluuttinen laatutekijä, jota ei voi käytännössä saavuttaa. Ohjelmisto joko on spesifikaation mukainen – siis virheetön – tai ei ole. Ohjelmisto ei voi olla esim. 30% oikeellinen. Koska ei-triviaalien määritysten mukainen ohjelmisto ei ole koskaan virheetön, se ei ole koskaan myöskään oikeellinen.
- *Luotettavuus* (reliability)
  - Luotettavuus on oikeellisuutta vastaava laatutekijä, mutta siinä ei vaadita täydellistä onnistumista. Sen sijaan luotettavuusvaatimus määrittelee, millä todennäköisyydellä (tai miten usein, millä aikavälillä jne.) ohjelmisto tai palvelu epäonnistuu.
  - Ohjelmiston luotettavuus riippuu sen käyttötavasta. Eri käyttäjät saattavat saada erilaisia luotettavuustasoja. Tässä luotettavuus eroaa oikeellisuudesta, joka joko on voimassa tai ei ole voimassa.

Käytön laatutekijät 2

- *Tehokkuus* (efficiency)
  - Tehokkuusvaatimukset määrittelevät, millä vasteajalla, missä tilassa tai millä kuormalla ohjelmisto tarjoaa palveluita.
  - Vasteaika viittaa palveluiden nopeuteen. Tila viittaa käytettävän muistin, levytilan ym. määrään. Kuorma viittaa samanaikaisten käyttäjien määrään.
- *Eheys* (integrity)
  - Eheysvaatimukset määrittelevät, miten hyvin ohjelmisto selviää siihen kohdistuvista vihamielisistä toiminnoista, kuten murtautumisyrityksistä tai tahallisista komentojen väärinkäytöistä.
- *Käytettävyys* (usability)
  - Käytettävyysvaatimukset määrittelevät, miten helppokäyttöinen ohjelmisto on, miten jyrkkä sen oppimiskäyrä on, miten paljon kognitiivista rasitetta sen käyttö aiheuttaa jne.
  - Loppukäyttäjän kannalta ohjelmisto on pitkälti sama kuin sen toiminnot, tehokkuus ja käytettävyys.

Muokkauksen laatutekijät 1

- *Ylläpidettävyys* (maintainability)
  - Ylläpidettävyysvaatimukset määrittelevät, miten helppoa ohjelmistosta on tunnistaa häiriöitä, löytää häiriöiden syitä (vikoja ja virheitä), korjata syyt ja varmentaa korjauksen onnistuminen.
  - McCallin ylläpidettävyys on siis puhtaasti ohjelmistovikojen etsintää ja korjausta. Ylläpito sisältää yleisemmästä näkökulmasta hoitoylläpidon (huolehditaan käyttäjien tarpeista), evoluutioylläpidon (varaudutaan muuttuviin vaatimuksiin) ja korjausylläpidon (korjataan ohjelmistoviat). McCallin ylläpidettävyys vastaa ainoastaan korjausylläpidon vaatimuksiin.

Muokkauksen laatutekijät 2

- *Joustavuus* (flexibility)
  - Joustavuusvaatimukset määrittelevät, miten helppoa ohjelmistoa on muokata uusille tai muuttuneille vaatimuksille sopivaksi, eli miten ohjelmisto suhtautuu evoluutioon.
  - Joustavuus on läheistä sukua ylläpidettävyydelle.
- *Testattavuus* (testability)
  - Testattavuusvaatimukset määrittelevät, miten helppoa ohjelmistolle on kirjoittaa ja suorittaa testitapauksia.
  - Testattavuus on melko rajoittunut laatutekijä. Myöhemmin määriteltävä verifioitavuus sisältää testattavuuden. Verifioitavuusvaatimukset määrittelevät, miten helppoa on varmentaa ohjelmiston toiminta.

Siirron laatutekijät

- *Siirrettävyys* (portability)
  - Siirrettävyysvaatimukset määrittelevät, miten helppoa ohjelmisto on ottaa käyttöön uudessa laitteisto- tai käyttöjärjestelmäympäristössä.
  - Siirrettävyys on läheistä sukua uudelleenkäytettävyyden kanssa.
- *Uudelleenkäytettävyys* (reusability)
  - Uudelleenkäytettävyysvaatimukset määrittelevät, kuinka hyvin tai kuinka paljon ohjelmistolle kirjoitettua koodia voidaan käyttää muissa ohjelmistoissa.
- *Yhteentoimivuus* (interoperability)
  - Yhteentoimivuusvaatimukset määrittelevät, miten hyvin ohjelmisto toimii yhteistyössä laitteiston ja muiden ohjelmistojen kanssa.
  - Yhteentoimivuusvaatimukset koskevat pääasiassa ohjelmiston rajapintoja.

Myöhemmät laatutekijämallit

- 1980-luvun loppupuolella esiteltiin kaksi uutta laatutekijämallia
  - Evans ja Marciniak -laatutekijämalli (1987)
  - Deutsch ja Willis -laatutekijämalli (1988)
- Nämä mallit ovat melko samanlaiset McCallin mallin kanssa:
  - Kymmenen McCallin 11 laatutekijästä on edelleen mukana. Testattavuus on pudotettu pois.
  - Molemmissa malleissa on mukana *verifioitavuus* (verifiability).
  - Molemmissa malleissa on mukana *laajennettavuus* (expandability).
  - Deutsch ja Willis -mallissa mukana ovat lisäksi *käyttöturvallisuus* (safety), *hallittavuus* (manageability) ja *selviytyvyys* (survivability).

Uudemmat laatutekijät 1

- *Verifioitavuus* (muokkauksen laatutekijä)
  - Verifioitavuus tarkoittaa, että kirjatut toiminnalliset ja ei-toiminnalliset vaatimukset voidaan varmentaa ohjelmistosta.
  - Verifioitavuus sisältää testattavuuden, sillä testaus on yksi verifiointimenetelmistä. Tämän johdosta molemmat uudemmat mallit jättivät testattavuuden pois.
- *Laajennettavuus* (muokkauksen laatutekijä)
  - Laajennettavuusvaatimuksilla taataan, että ohjelmisto on muokattavissa aiempaa laajempiin sovellusympäristöihin.
  - Laajennettavuustekijä vastaa pitkälti McCallin joustavuustekijää.

Uudemmat laatutekijät 2

- *Käyttöturvallisuus* (käytön laatutekijä)
  - Käyttöturvallisuusvaatimukset on tarkoitettu eliminoimaan järjestelmän tai käyttäjien kannalta vaaralliset olosuhteet.
- *Hallittavuus* (lähinnä muokkauksen laatutekijä)
  - Hallittavuusvaatimukset ovat pääosin ohjelmiston kehitystyön vaatimuksia. Niillä varmistetaan, että käytössä on työkaluja, joiden avulla ohjelmiston muuttaminen ja ylläpito on turvallista.
- *Selviytyvyys* (käytön laatutekijä)
  - Selviytyvyysvaatimukset varmentavat ohjelmiston tarjoamien palveluiden jatkuvuuden.
  - Selviytyvyys-laatutekijä on lähes sama kuin McCallin luotettavuus-laatutekijä.

Laatutekijämallien yhdistäminen

Mallit yhdistämällä saadaan 13 laatutekijää:

1. Oikeellisuus: toimiiko ohjelmisto spesifiointinsa mukaan
2. Luotettavuus: pysyykö ohjelmisto käyttökuntoisena
3. Tehokkuus: suoriutuuko ohjelmisto riittävän nopeasti sille määrätyistä tehtävistä
4. Eheys: selviääkö ohjelmisto sisäisistä ja ulkoisista hyökkäyksistä
5. Käytettävyys: miten käyttäjäystävällinen ohjelmisto on
6. Ylläpidettävyys: miten työlästä on etsiä ja korjata virheitä ohjelmistosta
7. Joustavuus: miten työlästä on muuttaa ohjelmiston toiminnallisuutta
8. Verifioitavuus: miten helposti ohjelmiston toiminta on varmennettavissa
9. Siirrettävyys: miten helposti ohjelmisto saadaan käyttöön uudessa toimintaympäristössä
10. Uudelleenkäytettävyys: missä määrin koodia voidaan käyttää muissa ohjelmistoissa
11. Yhteentoimivuus: miten hyvin ohjelmisto toimii yhteistyössä järjestelmän muiden osien ja muiden ohjelmistojen kanssa
12. Käyttöturvallisuus: miten turvallista ohjelmiston sisältävää järjestelmää on käyttää
13. Hallittavuus: miten hyvin ohjelmiston kehitys- ja muutosprosessia on tuettu

Tärkeimmät laatutekijät

- Asiakkaan kannalta merkittävimmät laatutekijät ovat käytettävyys, luotettavuus (oikeellisuus) ja tehokkuus.
- Nämä kolme käytön laatutekijää vaikuttavat siihen, miten miellyttävä ja hyödyllinen valmis ohjelmisto on käyttää.
  - Käytettävyys on läsnä lähes kaikissa ohjelmistoissa. Käytettävyysvaatimusten merkitys ymmärretään melko hyvin.
  - Luotettavuus on läsnä kaikissa ohjelmistoissa. Luotettavuusvaatimukset ovat matemaattisesti selkeitä, minkä johdosta niiden merkitys on ymmärretty jo ohjelmistotuotannon alkuajoista lähtien.
  - Tehokkuus on läsnä lähes kaikissa ohjelmistoissa. Aiempina vuosikymmeninä siihen keskityttiin paremmin, mutta laitteistojen ja ohjelmointikielten kehittyessä tehokkuuden merkitys laski. Viime vuosina tehokkuusvaatimukset ovat kokeneet renessanssin ja niitä arvostetaan jälleen enemmän.

Miksi muita laatutekijöitä?

Jos asiakas on pääasiassa kiinnostunut edellisistä kolmesta laatutekijöistä, niin miksi muiden laatutekijöiden pitäisi näkyä vaatimuksissa?

- Käytön laatutekijät vaikuttavat ohjelmiston toiminnallisuuteen, joten ne täytyy määritellä vaatimuksissa.
- Muokkauksen laatutekijät liittyvät siihen, miten helppoa ohjelmistoa on muuttaa ja parantaa hajottamatta sitä. Näiden huolellinen suunnittelu ja huomiointi ohjelmistotuotannossa pienentää asiakkaan kustannuksia. Rahan tulo tai meno on asiakkaalle merkittävin sivuvaikutus, joten nämä vaatimukset kannattaa määritellä vaatimuksissa.
- Siirron laatutekijät liittyvät siihen, miten helposti ohjelmistoa voidaan hyödyntää muissa ympäristöissä tai ohjelmistoissa. Nämä laatutekijät helpottavat tulevaa kehitystyötä, mutta eivät asiakkaan työtä. Niitä ei välttämättä tarvitse määritellä vaatimuksissa.

Asiakasta kiinnostamattomat laatutekijät

- Kannattaa huomata, että vaikka kaikkia laatutekijöitä ei huomioida asiakkaan vaatimuksissa, ne voivat olla tärkeitä kehittäjille.
- Ne laatutekijät, jotka on kirjattu ei-toiminnallisiin vaatimuksiin, näkyvät vaatimusdokumentissa.
- Ne laatutekijät, joita ei ole kirjattu ei-toiminnallisiin vaatimuksiin, näkyvät *jossakin*. Käytännössä on kaksi vaihtoehtoa:
  - Ne näkyvät erillisessä kehitystyön vaatimusdokumentissa.
  - Ne näkyvät laatukäsikirjassa.

Laatutekijöiden toteutuminen

- Pelkkä laatutekijöiden listaus ei riitä, sillä niitä vastaavien vaatimusten täytyy myös toteutua ohjelmistossa.
- Mistä tiedämme, että ohjelmisto täyttää sille määritellyt laatutekijät?
- Paras keino on osittaa ohjelmiston laatutekijät sellaisiksi osatekijöiksi ja vaatimuksiksi, että ne voidaan validoida numeerisesti. Tämä tarkoittaa sopivien *mittareiden* (metrics) käyttöä.

Ylläpidettävyysesimerkki

- Esimerkiksi ohjelmiston ylläpidettävyydelle voidaan määritellä seuraavat osatekijät (Galin, 2004):
  - yksinkertaisuus (simplicity)
  - modulaarisuus (modularity)
  - selittävyys (self-descriptiveness)
  - koodaus- ja dokumentointistandardien mukaisuus (coding and documentation guidlines)
  - yhtenäisyys (compliance / consistency)
  - dokumenttien saatavuus (document accessibility)
- Edellisille osatekijöille on mahdollista määritellä mittarit, jotka korreloivat osatekijän toteutumista.
- Sen sijaan
  - kaikille osatekijöille ei löydy sellaisia mittareita, jotka antaisivat täyden korrelaation (miten mitataan selittävyyttä?) ja
  - osatekijöiden toteutuminen ei anna täyttä korrelaatiota itse ylläpidettävyyden toteutumiselle
- Toistaiseksi ei ole löydetty sellaisia mittareita, jotka kertoisivat suoraan, miten ylläpidettävää koodi on.

Osatekijöiden valinta

- Osatekijöiden valinta on kokemukseen perustuva suunnittelupäätös.
  - Laatutekijälle valittavat osatekijät kannattaa valita sen mukaan, mitä osatekijöitä kirjallisuudessa on ehdotettu ja mitä hyväksi havaittuja osatekijöitä on käytetty aiemmissa projekteissa.
- Mitä enemmän laatua tutkitaan, sitä paremmin voidaan sanoa, mitä osatekijöitä kannattaa valita. Yksiselitteistä ratkaisua ei voida saavuttaa, koska laatu, laadun tekijät ja niiden osatekijät ovat projekti- ja kontekstikohtaisia.

Mahdollisia osatekijöitä

- Seuraavassa on muutamia Galinin listaamia laatutekijöiden osatekijöitä. Täydellinen lista löytyy Galinin kirjasta *Software Quality Assurance*.
  - Oikeellisuus
    - tarkkuus (accuracy), täydellisyys (completeness) , ajantasaisuus (up-to-dateness), saatavuus (availability), koodaus- ja dokumentointistandardien mukaisuus, yhtenäisyys
  - Luotettavuus
    - järjestelmän luotettavuus (system reliability), sovelluksen luotettavuus (application reliability), ohjelmistovirheestä toipuminen (computational failure recovery), järjestelmävirheestä toipuminen (hardware failure recovery)
  - Tehokkuus
    - laskentatehokkuus (efficiency of processing), tilatehokkuus (efficiency of storage), kommunikointitehokkuus (efficiency of communication), energiatehokkuus (efficiency of power usage)
  - Käytettävyys
    - käyttökelpoisuus (operability), koulutus (training)

Huomioita Galinin osatekijälistasta

- Osatekijä voi esiintyä monen laatutekijän osana.
  - Esimerkiksi yhtenäisyys on listattu sekä oikeellisuuden että ylläpidettävyyden osatekijäksi.
- Osatekijöiden valinta on suunnittelupäätös.
  - Esimerkiksi käytettävyyteen lasketaan usein sellaisia osatekijöitä, kuten kognitiivinen rasite (miten paljon muistamista ohjelmisto vaatii), opittavuus (miten jyrkkä oppimiskäyrä ohjelmistolla on) ja mukautuvuus (miten hyvin eri tasoiset käyttäjät voivat käyttää ohjelmistoa).
- Osatekijöistä mittausten kautta saatavat tunnusluvut eivät kerro suoraan, miten hyvin laatutekijä toteutuu.
  - Voidaanko esimerkiksi luotettavuuden osatekijöiden avulla sanoa, miten luotettava järjestelmä on? Ehkä. Onko arvo absoluuttinen (järjestelmämme luotettavuus on 88%)? Ei.
  - Voidaanko käytettävyyden osatekijöiden avulla sanoa, miten käytettävä järjestelmä on? Tämä ei onnistu ainakaan Galinin melko rajoittuneiden osatekijöiden avulla.

2.4. Ohjelmistojen ”virheet”

- Ohjelmistojen laadun kannalta on oleellista erotella toisistaan (kehitys-) *virhe* (error), (ohjelmisto-) *vika* (fault) ja (toiminta-) *häiriö* (failure/defect).
  - Koodin kirjoittajat tekevät virheitä. Virhe voi olla koodin rakenteessa (esim. viitataan väärään muuttujaan) tai koodin logiikassa (esim. vaatimus on tulkittu väärin).
  - Kun virhe saa ohjelmiston toimimaan väärin, kyse on (ohjelmisto)viasta. Kaikki virheet eivät tee näin, sillä virhettä seuraavat koodirivit saattavat ”neutraloida” virheen.
  - Kun viallista koodia suoritetaan, syntyy häiriö. Kaikista ohjelmistovioista ei seuraa toimintahäiriöitä. Vika saattaa olla hautautuneena koodiin sellaiseen paikkaan, että sitä ei suoriteta koskaan (esimerkiksi harvinaiseen poikkeuskäsittelijään tai käytännössä mahdottomaan parametrien kombinaatioon).
  - Puhekielen *bugi* (bug) tarkoittaa yleensä virhettä tai vikaa.

Virheet, viat ja häiriöt

- Vain pieni osa virheistä näkyy vikoina ja pieni osa vioista häiriöinä.
- Kuitenkin mikä tahansa koodiin jäänyt passiivinen virhe saattaa koodia muutettaessa aktivoitua viaksi ja mikä tahansa vika saattaa sopivassa käytössä aiheuttaa häiriön.
- Kehitystiimille laadukas tuote tarkoittaa vähäistä virheiden määrää. Loppukäyttäjän kannalta se tarkoittaa vähäistä häiriöiden määrää.
- Yleensä ohjelmistoista raportoitujen häiriöiden määrä on välillä 0,1 – 5,0 häiriötä tuhatta koodiriviä kohden.

Ohjelmistojen virhetyypit 1

- Koska ohjelmistovirheet vaikuttavat ohjelmiston laatuun, on tärkeää tietää, minkä tyyppisiä virheitä tehdään.
- Galin (2004) listaa yhdeksän virhetyyppiä ja antaa kustakin esimerkkejä:
  1. Väärin määritellyt vaatimukset
     - Vaatimusten määrittelytavassa on virheitä
     - Puuttuvia vaatimuksia
     - Epätäydellisiä vaatimuksia
     - Ylimääräisiä vaatimuksia
  2. Asiakkaan ja kehittäjän väliset kommunikaatiokatkokset
     - Asiakkaan vaatimukset ymmärretään väärin
     - Asiakkaan muutostoiveet ymmärretään väärin
     - Asiakkaan vastaukset kysymyksiin ymmärretään väärin

Ohjelmistojen virhetyypit 2

3. Tahalliset poikkeamat vaatimuksista
   - Ohjelmistossa uudelleenkäytetään koodia ilman kunnollista analyysia vaadittavista muutoksista
   - Osa vaatimuksista jää toteuttamatta budjetin tai aikataulun ylittymisen johdosta
   - Kehitystiimi muuttaa vaatimuksia kysymättä asiasta asiakkaalta
4. Ohjelmiston logiikan suunnitteluvirheet
   - Väärien tai tilanteeseen sopimattomien algoritmien käyttö
   - Tilasiirtymävirheet
   - Raja-arvovirheet
5. Koodausvirheet
   - Painovirheet
   - Kielioppivirheet
   - Vaatimusten tai suunnittelun tulkintavirheet

Ohjelmistojen virhetyypit 3

6. Dokumentointi- ja koodausohjeiden noudattamatta jättämiset
   - Vaikealukuiset dokumentit
   - Vaikeaselkoinen ohjelmakoodi
7. Testausprosessissa oikomiset
   - Vajaa testaussuunnitelma
   - Löydettyjen vikojen ja virheiden kehno dokumentointi
   - Tiukan aikataulun johdosta tapahtuva puutteellinen vikojen ja virheiden korjaus
8. Toimintatapavirheet
   - Väärin tulkittu ohjelmiston tarkoitettu käyttötapa (käyttötapaus)
9. Dokumentointivirheet
   - Dokumentoimattomat toiminnot
   - Virheelliset toimintaohjeet
   - Ylimääräiset ja puuttuvat toiminnot

3. Ohjelmistojen laadunvarmistus

- Kohde (ohjelmisto), ongelmat (virheet, viat, häiriöt) ja ongelmakohdat (laatutekijät) määrittelemällä laadulle saadaan malli, joka ei vielä ole kovinkaan konstruktiivinen.
- Mallia kiinnostavampaa on tietää, mitä laadukkaan ohjelmiston tekemiseen tarvitaan. Tässä auttaa *ohjelmistojen laadunvarmistus* (Software Quality Assurance, SQA) tai lyhyemmin *laadunvarmistus.*
- Laadunvarmistus on kaikissa projekteissa ja projektien osavaiheissa läsnä oleva toiminto, ns. *sateenvarjotoiminto* (umbrella activity).
  - Sateenvarjotoiminnot ovat projektien tukitoimintoja. Esimerkiksi projektikokoukset ovat sateenvarjotoimintoja.
- Laadunvarmistus tarjoaa toimintatavat, joiden avulla projekteissa voidaan tehdä laadukkaita ohjelmistoja.

Laadunvarmistuksen määritelmät 1

- Yksi yleisimmistä laadunvarmistuksen määritelmistä on IEEE:n määritelmä vuodelta 1991:
  - Laadunvarmistus on:
    1. Suunniteltu ja systemaattinen, kaikki tarvittavat toiminnot sisältävä malli, jolla varmennetaan, että tuote tai sen osa toteuttaa sille määritellyt tekniset vaatimukset.
    2. Joukko tuotteen kehitys- tai valmistusprosessia valvovia toimintoja. Eroaa laadunvalvonnasta.
- Galinin mukaan IEEE:n määritelmän ”tekniset vaatimukset” tarkoittavat ei-toiminnallisia vaatimuksia.

Laadunvarmistuksen määritelmät 2

- Galin (2004) käyttää laadunvarmistukselle IEEE:tä laajempaa määritelmää:
  - Ohjelmiston laadunvarmistus on
    - systemaattinen suunniteltu joukko välttämättömiä toimintoja, joiden avulla varmennetaan, että ohjelmiston kehitysprosessi tai ohjelmistojärjestelmän ylläpitoprosessi täyttää sekä määritellyt tekniset vaatimukset että projektin aikataulussa ja budjetissa pysymisen kannalta välttämättömät hallintovaatimukset.
- Toisin sanoen Galinin määritelmässä laadunvarmistuksella varmennetaan
  - toimintojen teknisiä vaatimuksia (siis ei-toiminnallisia vaatimuksia) sekä ohjelmiston kehitys- että ylläpitovaiheessa sekä
  - projektien aikataulussa ja budjetissa pysymistä.
- Budjetissa pysymisen sivuvaikutuksena laadunvarmistuksella pyritään minimoimaan laadukkaiden ohjelmistojen kehitys- ja ylläpitotyön kustannukset.

Laadunvarmistus ja laadunvalvonta

- *Laadunvalvonta* (quality control) tarkoittaa sellaisia toimintoja, joiden avulla estetään huonolaatuisten tuotteiden pääsy markkinoille.
  - Laadunvalvontaa tehdään pääasiassa tuotteen valmistuttua, mutta ennen sen luovuttamista asiakkaalle.
  - Laadunvalvonta on heikon laadun *etsimistä*.
- Laadunvarmistuksen avulla varmennetaan, että tuotteet läpäisevät laadunvalvonnan.
  - Laadunvarmistusta tehdään koko tuotteen ja kehitystyön elinkaaren ajan.
  - Laadunvarmistus on heikon laadun *välttämistä*.
- Laadunvarmistus sisältää laadunvalvonnan eli on sitä laajempi sateenvarjotoiminto.

Laadunvarmistus ja ohjelmistotuotanto

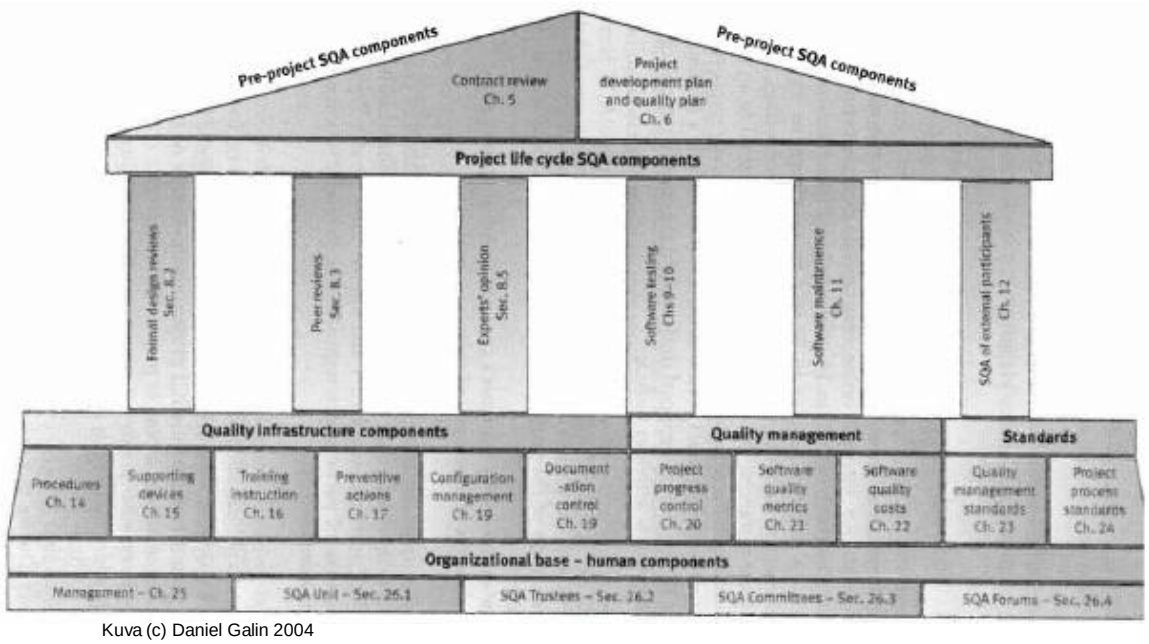
- Edelleen IEEE:n määritelmän (1991) mukaan *ohjelmistotuotanto* (software engineering) määritellään seuraavasti:
  - Ohjelmistotuotanto on systemaattinen, kurinalainen ja mitattava lähestymistapa ohjelmistojen kehitystyöhön, käyttöön ja ylläpitoon.
- Toisin sanoen ohjelmistotuotanto on kehys, jonka sisällä voidaan käyttää laadunvarmistusta.
  - Toisinaan ohjelmistojen kehittäjät (eli ohjelmistotuotannon toteuttajat) pitävät laadunvarmistusta jonkinlaisena välttämättömänä pahana.
  - Näin ei pitäisi olla, sillä oikein käytettynä laadunvarmistus helpottaa ohjelmistojen kehitys- ja ylläpitotyötä.

Laadunvarmistuksen komponentit

Laadunvarmistus voidaan osittaa kuuteen luokkaan:

1. Projektin ennakkovaiheiden laadunvarmistus
   - sopimusten laadunvarmistus
2. Projektin elinkaaren laadunvarmistus
   - kehitystyön ja ylläpidon laadunvarmistus
   - ulkoistuksen laadunvarmistus
3. Laatukehyksen laadunvarmistus
   - yritystason laatupolitiikka
4. Projektinhallinnan laadunvarmistus
   - projektin seuranta ja ohjaus
   - projektin laadunvarmistuksen metriikat
5. Standardointi
   - laadunhallinnan standardit
   - projekti- ja prosessistandardit
6. Laadunvarmistuksen organisointi
   - henkilöstöhallinto

Laadunvarmistuksen komponentit



Kuva (c) Daniel Galin 2004

3.1. Projektin ennakkovaiheiden laadunvarmistus

- (Asiakas)projektin ensimmäinen virallinen vaihe on asiakkaan ja kehitystyön tekevän yrityksen välisen sopimuksen laatiminen.
- Ennen sopimuksen allekirjoittamista sen on oltava riittävän laadukas. Sopimustesti tarkastetaan huolellisesti ennen sen viimeistelyä ja allekirjoittamista.
- Sopimuksen tarkastus on osa laadunvarmistusta. Sillä varmennetaan ainakin seuraavia asioita:
  - Asiakkaan kirjaamat vaatimukset tuotteelle ovat järkeviä.
  - Projektin aikataulu- resurssiarviot ovat järkeviä.
  - Projektityöntekijöiden ammattitaito riittää projektin läpivientiin.
  - Asiakas kykenee täyttämään oman osansa projektista:
    - asiakkaalla on riittävästi aikaa osallistua projektiin ja
    - asiakkaalla on riittävästi taitoa kertoa projektin ongelmakentästä (problem scope).
  - Projektin riskit on tunnistettu ja kirjattu ylös.

Projektin ennakkovaiheiden laadunvarmistus 1

- Sopimuksen allekirjoituksen jälkeen projektia varten tarvitaan kaksi dokumenttia: projektisuunnitelma ja laatusuunnitelma.
- *Projektisuunnitelma* (project plan, Galin: project development plan) sisältää seuraavia tietoja:
  - aikataulu
  - henkilö- ja laiteresurssit
  - projektin riskit
  - projektin jäsenet ja alihankkijat
  - käytettävä prosessimalli
  - käytettävät työkalut
  - uudelleenkäyttösuunnitelma

Projektin ennakkovaiheiden laadunvarmistus 2

- *Laatusuunnitelma* (quality plan) sisältää ainakin seuraavat kohdat:
  - Laatutavoitteet
    - valitut laatutekijät
    - valittujen laatutekijöiden osatekijät
    - osatekijöiden laadun mittarit
  - Projektin osavaiheiden aloitus- ja lopetusehdot
    - kullekin osavaiheelle edellisiltä osavaiheilta vaadittavat tulokset
    - milloin osavaiheen lopputulos on riittävän laadukas
  - Käytettävät laadunvarmistusmenetelmät
    - tarkastusstrategia ja aikataulu
    - testausstrategia ja aikataulu
    - muut verifioinnin ja validoinnin strategiat ja aikataulut

3.2. Projektin elinkaaren laadunvarmistus

- Projektin elinkaari on Galinin mukaan kaksivaiheinen. Se sisältää
  - kehitystyön elinkaaren ja
  - ylläpidon elinkaaren.
- Sekä kehitystyölle että ylläpidolle on lukuisia laadunvarmistustekniikoita. Seuraavassa muutamia yleisimpiä:
  - katselmoinnit ja tarkastukset (reviews, inspections)
  - ulkopuoliset asiantuntijalausunnot (expert opinions)
  - testaus (testing)
  - ylläpidon laadunvarmistus (software maintenance)
  - ulkopuolelta hankittujen ja asiakkaan valmiina tarjoamien komponenttien laadunvarmistus
  - alihankkijoiden laadunvarmistus

Katselmoinnit

- Tutkimusten mukaan katselmoinnit ovat oikein käytettynä tehokkain laadunvarmistustekniikka. Niissä yksi tai useampi henkilö käy dokumentin läpi etsien siitä virheitä.
- Katselmointitekniikat vaihtelevat muodollisista epämuodollisiin. Seuraavassa muutama perustekniikka:
  - *Tarkastus* (review / formal review / formal technical review / inspection) on muodollisin katselmointitekniikka.
    - Tarkastukseen osallistuu yleensä 4-6 ennalta nimettyä henkilöä.
    - Tarkastusprosessissa on tarkka ohjelma ja aikataulu.
    - Tarkastettu dokumentti hyväksytään, hyväksytään muutoksilla tai hylätään. Tulos kirjataan pöytäkirjaan.
  - *Vertaisarviointi* (peer review) on tarkastuksia epämuodollisempi.
    - Vertaisarviointiin osallistuminen on yleensä vapaaehtoista.
    - Vertaisarvioinnin prosessia ei ole määrätty tarkasti.
    - Vertaisarviointitekniikoita on useita. Nykyisin yksi yleisimmistä tekniikoista on *pariohjelmointi (*pair programming). Siinä toinen parista ohjelmoi ja toinen tarkastaa ohjelmoijan kirjoittamaa koodia ja suunnittelee koodiin sopivia testitapauksia.

Ulkopuoliset asiantuntijalausunnot

- Ulkopuoliset asiantuntijalausunnot ovat projektin ulkopuolisia virallisia tai epävirallisia mielipiteitä projektista tai sen tuotoksesta.
- Ulkopuoliset asiantuntijalausunnot ovat käteviä, kun
  - yrityksestä ei löydy omasta takaa riittävää ongelmakentän tuntemusta
  - yrityksestä on vaikeaa löytää riittävästi ihmisiä osallistumaan tarkastuksiin
    - asiantuntijoita voidaan käyttää tarkastuksissa (kallista) tai
    - asiantuntijalausunnolla voidaan korvata tarkastus (edullista)
  - yrityksen omat asiantuntijat eivät ole saatavilla
  - yrityksen omat asiantuntijat ja projektiryhmä eivät löydä yhteistä säveltä ja kaipaavat siten puolueetonta näkökantaa

Testaus

- Testaus on yleisin laadunvarmistustekniikka. Siinä valmista ohjelmistoa tai sen osaa suoritetaan ennalta määrätyillä syötteillä ja tarkastellaan saatua tulosta.
- Testauksella on kaksi tarkoitusta:
  - etsitään ohjelmistosta häiriöitä, vikoja ja virheitä
  - varmennetaan, että ohjelmisto tai sen osa täyttää sille asetetut vaatimukset
- Testaus sopii parhaiten ohjelmiston yksityiskohtien tarkasteluun ja laadun viimeistelyyn. Katselmoinnit ovat parhaimmillaan vaatimusten validoinnissa ja kokonaisuuksien hallinnassa, sillä niitä voidaan pitää ilman suoritettavaa ohjelmakoodia.

Ylläpidon laadunvarmistus

- Ylläpito voi viedä 80% tuotteen elinkaaren resursseista, joten sen laadunvarmistus on erittäin tärkeää.
- Onneksi ylläpitoon sopivat samat laadunvarmistustekniikat kuin kehitystyöhön.
- Modernissa ohjelmistokehityksessä ero ylläpidon ja kehitystyön välillä on häilyvä. Tuotteesta tehdään säännöllisin väliajoin uusia versioita, joilla vastataan muutospaineisiin (ja rahastetaan).

Muualta hankittujen komponenttien laadunvarmistus

- Modernissa ohjelmistotuotannossa kaikkea ei tarvitse tehdä itse. Osa tehtävästä ohjelmistosta saadaan muualta:
  - osa ohjelmistokehityksestä voidaan ulkoistaa muille yrityksille
  - asiakkaalta saadaan mahdollisesti valmiita komponentteja
  - voidaan ostaa valmiita komponentteja tai käyttää avointa lähdekoodia
- Ulkoistettujen komponenttien laadunvarmistus alkaa sopimuksesta. Siitä täytyy selvitä, miten ulkoistetun komponentin tuottava yritys hoitaa laadunvarmistuksen.
- Kaikkien muualta hankittujen komponenttien laadunvarmistuksessa voidaan käyttää samoja laadunvarmistusmenetelmiä kuin kehitystyön laadunvarmistuksessa. Kuitenkin koska komponenttien koodi on jo valmis, kannattaa erityisesti keskittyä seuraaviin tekniikoihin:
  - Rajapintojen testaus: komponentti toimii rajapintojen kautta yhteistyössä muiden komponenttien kanssa. Rajapintojen tulee toimia dokumentaation mukaisesti.
  - Rajapintojen dokumentaation tarkastus: Jotta rajapinnat voidaan testata kunnolla, niiden dokumentaation tulee olla kunnossa. Rajapintadokumentaatio kannattaa tarkastaa huolellisesti.

Alihankkijoiden laadunvarmistus

- Muualta hankittavien komponenttien lisäksi myös niitä tuottavien toimittajien yleislaatu voidaan varmistaa
- Mahdollisia keinoja:
  - luotettavat referenssit
  - sertifioidut ISO-laatuleimat
  - vaaditulla kypsyystasolla toimiminen (esim. CMM)

3.3. Laatukehyksen laadunvarmistus

- *Laatukehys* (quality infrastructure) tarkoittaa laadunvarmistukseen kuuluvia tehtäviä ja ohjeistoja, joiden avulla parannetaan tuottavuutta ja vältetään tai ainakin vähennetään virheitä.
- Laatukehystehtävät ovat sateenvarjotoimintoja. Ne suunnitellaan sellaisiksi, että ne palvelevat laajaa projekti- ja ylläpitopalvelujoukkoa.
- Laatukehyspalveluihin kuuluvat:
  - toimintaohjeet
  - dokumenttipohjat ja tarkistuslistat
  - henkilökunnan koulutus, ammattitaidon ylläpito ja sertifiointi
  - virheitä välttävät ja korjaavat toiminnot
  - versionhallinta
  - dokumenttien hallinta
- Laatukehyksen ohjeet kootaan usein yrityksessä laatukäsikirjaan.

Toimintaohjeet

- Toimintaohjeet ovat yksityiskohtaisia toimintatapakuvauksia. Ne voivat olla
  - käytettävien ohjelmistoprosessien kuvauksia
  - prosessien työvaiheiden (osaprosessien) kuvauksia
  - työmenetelmien toimintatapojen kuvauksia
  - hallinto-ohjeita
  - jne.
- Toimintaohjeisiin kirjataan sellaiset yrityksen toimintatavat, joita noudatetaan sen kaikissa projekteissa.

Dokumenttipohjat ja tarkistuslistat

- Dokumenttipohjat määrittelevät yrityksessä käytettävien dokumenttien korkean tason rakenteen.
  - Yhtenäiset dokumenttipohjat varmistavat, että projektien väliset dokumentit ovat vertailukelpoisia. Tämä helpottaa projektin elinkaaren laadunvarmistusta.
  - Testitapaukset ovat dokumentteja, joille voi olla oma dokumenttipohja.
  - Koodikin on dokumentti, joten sillekin voi olla oma dokumenttipohja. Yleensä tällöin puhutaan koodi- ja/tai kommentointistandardista.
- Tarkistuslistat määrittelevät tarkastuksissa ja joskus vertaisarvioinneissa etsittävät vika- ja virhetyypit.
  - Tarkistuslistat helpottavat vikojen ja virheiden etsintää. Ne listaavat kullekin dokumentille yleisimmät vika- ja virhetyypit.

Henkilökunnan koulutus ym.

- Koska laatu on viime kädessä kiinni ohjelmiston kehittäjistä, laadunvarmistukseen kuuluu myös heidän ammattitaitonsa nosto ja ylläpito:
  - peruskoulutuksella saadaan ammattilaisia
  - täydennyskoulutuksella pidetään ammattilaisten ammattitaitoa yllä
  - sertifioinneilla osoitetaan ammattilaisille ja asiakkaille tekijöiden varmennettu taitotaso
- Työhyvinvointi on osa laadunvarmistusta:
  - henkilö, joka on tyytymätön tai stressaantunut, ei pidemmän päälle tee laadukasta jälkeä
  - henkilö, joka ei voi vaikuttaa työhönsä, ei pidemmän päälle tee laadukasta jälkeä
  - henkilö, jonka ura ei etene, ei pidemmän päälle tee laadukasta jälkeä

Virheitä välttävät ja korjaavat toiminnot 1

- Mitä myöhemmin virhe havaitaan, sitä kalliimmaksi sen korjaaminen tulee. Näin teoriassa virheiden välttäminen on edullisempaa kuin virheiden korjaaminen.
- Käytännössä syy-seuraussuhde ei ole aivan lineaarinen. Silti sopivaan rajaan asti toimintatapoja voidaan muuttaa sellaisiksi, että niiden avulla tehtävien ohjelmistojen virhetiheys laskee.
- Eräs keino välttää virheitä on kerätä ja analysoida historiatietoa aiemmissa projekteissa löydetyistä virheistä, vioista ja häiriöistä.

Virheitä välttävät ja korjaavat toiminnot 2

- Kun tietoa on riittävästi, sen avulla voidaan
  - etsiä ja toteuttaa prosessin muutoksia, jotka estävät samanlaisten virheiden esiintymistä tulevaisuudessa
  - korjata vastaavia virheitä muissa projekteissa
  - käyttää hyväksi havaittuja menetelmiä, joilla parannetaan virheiden välttämisen todennäköisyyttä.
- Käytännössä pelkkä menneiden projektien analyysi ei aina auta välttämään virheitä. Yksinkertaisimmat parannuskeinot on käytetty nopeasti, minkä jälkeen ohjelmistojen virhetiheys ei enää laske.
- Analyysin lisäksi kannattaa seurata ohjelmistoprosessien tutkimusta ja mahdollisesti pienten muutosten sijaan laittaa koko nykyinen toimintatapa remonttiin.

Versionhallinta

- Modernissa kehitystyössä ohjelmistosta saattaa valmistua uusi versio 2-3 viikon välein. Näitä kaikkia versioita pitää hallita ja toisinaan useita niistä pitää kehittää edelleen.
- Käytännössä ohjelmiston eri versioista syntyy *versiopuu* (version tree). Puun hallinta on oleellinen osa laadunvarmistusta.
- Versiopuun hallinta on pääosin automaattista, sillä versionhallintaan on kehitetty hyviä työkaluja.
  - Aika ajoin puuta täytyy kuitenkin siistiä, sillä muuten ylläpidettävien versioiden määrä kasvaa hallitsemattoman suureksi. Tätä ei yleensä voi tehdä automaattisesti.

Dokumenttien hallinta

- Laadunvarmistus vaatii, että projektin tärkeimmät dokumentit ovat saatavilla projektin aikana, ylläpitovaiheessa ja jopa ylläpitovaiheen jälkeen. Tällaisia dokumentteja kutsutaan *hallituiksi dokumenteiksi* (controlled documents).
- Hallitut dokumentit ovat toimivan ohjelmiston lisäksi projektin tärkeimpiä tuotoksia.
- Dokumenttien hallinta on hallittavien dokumenttien laadunvarmistusta. Siihen kuuluvat seuraavat tehtävät:
  - hallittavien dokumenttityyppien määrittely
  - dokumenttien rakenteen (dokumenttipohjan), tunnisteen ym. määrittely
  - dokumenttien tarkastus- ja hyväksymismenettelyjen määrittely
  - dokumenttien tallennusmenetelmien määrittely
- Hallitut dokumentit ovat tärkeä ylläpidon ja virheiden välttämisen tietolähde.

3.4. Projektinhallinnan laadunvarmistus

- Projektinhallinnan laadunvarmistus tukee ohjelmistoprojektien ohjaus- ja ylläpitotehtäviä.
- Projektinhallinnan laadunvarmistustehtäviin kuuluvat seuraavat:
  - projektien etenemisen seuranta ja ohjaus
  - projektien laadun mittaus ja arviointi
  - laadun kustannusten mittaus ja arviointi

Projektien etenemisen seuranta ja ohjaus

- Projektien etenemisen seuranta ja ohjaus on projektinhallinnan laadunvarmistuksen tärkein tehtävä. Sen avulla varmistetaan, että:
  - projektit pysyvät aikataulussa ja budjetissa
  - projektien riskienhallinta toimii
  - projektit reagoivat toteutuneisiin riskeihin ja muuttuneisiin reunaehtoihin oikein.
- On tärkeää, että projekteja seurataan ulkopuolelta. Omalle työlle on helppo tulla sokeaksi. Ulkopuolinen taho näkee työn etenemisen toisella tavalla kuin projektityöntekijä.
- Seuranta ja ohjaus eivät kuitenkaan ole projektista riippumattomia. Projekti raportoi omasta etenemisestään laadunvarmistuksesta vastaaville henkilöille, ja epäselvissä tilanteissa laadunvarmistuksesta kysytään tietoja projektilta.

Projektien laadun mittaus ja arviointi

- Laadunvarmistuksessa projektien laatua arvioidaan sopivilla projektia mittaavilla metriikoilla.
- Metriikat vaihtelevat yrityksittäin. Seuraavassa on muutama yleinen:
  - tuottavuudelle koodirivien määrä / kuukausi
  - testaustehokkuudelle löydettyjen virheiden määrä / 1000 koodiriviä (kloc)
  - työtehokkuudelle tuottava työaika / kokonaistyöaika

Laadun kustannusten mittaus ja arviointi

- Projekteissa arvioidaan myös laadunvarmistuksen hyöty-kustannus-suhdetta: paljonko laadunvarmistus kuluttaa ja paljonko sillä säästetään.
- Jos laadunvarmistus säästää enemmän kuin kuluttaa, se on kannattavaa.
- Optimaalisen laadunvarmistustason löytäminen on optimointitehtävä. Tiettyyn tasoon asti laadunvarmistus vähentää kustannuksia, mutta ei rajattomasti.

3.5. Standardointi

- Ulkoisten standardien käyttö on yksi tapa yhtenäistää ohjelmistoprojekteja ja siten saada niistä ulos tasalaatuisempia tuotteita.
- Ulkoiset standardit ovat yleisiä ja yleensä huolellisesti testattuja. Ne sopivat useimpiin organisaatioihin jossain määrin mutta eivät juuri mihinkään organisaatioon täydellisesti sellaisenaan.
- Käytännössä laatustandardi täytyy sovittaa omaan yritykseen. Tämä voi olla helppoa tai vaikeaa riippuen siitä, minkälainen yritys on kyseessä, minkälaisia prosesseja seurataan ja minkälaisia ohjelmistoja valmistetaan.

Kansainvälisiä laatustandardeja

- Laadunvarmistuksen kannalta mielenkiintoisia ovat laadunhallinnan standardit ja prosessistandardit.
  - Laadunhallinnan standardit määrittelevät, mitä laadukkaaseen ohjelmistokehitystyöhön vaaditaan. Sen sijaan ne eivät määrittele, miten yritys toteuttaa ko. vaatimukset. Tällaisia standardeja ovat:
    - SEI-CMMI: Software Engineering Institute - Capability Maturity Model Integration
    - ISO-9001:2000: International Standards Organization – Quality Management Systems: Requirements
    - ISO 15504 (SPICE): Software Process Improvement and Capability dEtermination
  - Prosessistandardit määrittelevät, miten kehitystyötä tai sen osavaihetta tehdään, ja sen kautta määrittelevät, mitä näin saavutetaan.Tällaisia standardeja ovat
    - IEEE 1012: Institute of Electrical and Electronic Engineers - IEEE Standard for Software Verification and Validation
    - ISO/IEC 12207: ISO/International Electrotechnical Commission – Software Life Cycle Processes

3.6. Laadunhallinnan organisointi

- Koska laadunvarmistus on yhteistä kaikille yrityksen projekteille, se on yrityksen sateenvarjotoiminto. Näin se tarvitsee vastaavaa hallinnointia kuin kehitysprojektit.
- Laadunvarmistuksen hallinnoinnista vastaa *laadunvarmistusryhmä* (quality assurance team, SQA unit). Se valvoo, että laadunvarmistus toteutuu projekti- ja yritystasolla.
- Jos yrityksessä on kokopäiväinen testaustiimi, sen jäsenet ovat täysipäiväisesti laadunvarmistus-ryhmässä. Muut laadunvarmistusryhmän jäsenet osallistuvat ryhmän toimintaan muun työn ohessa.

Laadunvarmistusryhmän tehtävät

- Laadunvarmistusryhmälle kuuluvat usein seuraavat tehtävät:
  - yritystason laadunvarmistussuunnitelman valmistelu ja ylläpito
  - projektien jäsenten ja ulkopuolisten asiantuntijoiden kanssa tehtävä yhteinen laadunvarmistus
  - yrityksen sisäisten laadunvarmistusmenetelmien tarkastaminen
  - laadunvarmistuksen edustaminen yrityksen sisäisissä ja yritysten välisissä kokouksissa
  - laatukehyksestä huolehtiminen
- Laadunvarmistusryhmän lisäksi yrityksessä voi olla suoraan korkeimman johdon alainen ryhmä, joka vastaa
  - yrityksen laatustrategiasta
  - laadunvarmistusryhmän seurannasta
  - yleisistä laatuun liittyvistä resurssipäätöksistä
- Laadunvarmistusryhmä vastaa laatustrategian toteutumisesta.

3.7. Laatukäsikirja

- Kuten edellisistä kalvoista on käynyt ilmi, laadunvarmistus on erittäin monipuolinen sateenvarjotoiminto, johon liittyy paljon standardoitavia elementtejä.
- Tietomäärän laajuuden vuoksi tarvitaan tietokanta, jonne tallennetaan yrityksen laatustrategia ja sen toteutus. Tätä kutsutaan *laatukäsikirjaksi* (Software Quality Assurance Plan).

Laatukäsikirjasta

- Kullakin yrityksellä on oma laatukäsikirja, jonka tiedot ovat liikesalaisuuksia. Niiden rakenne perustuu toisinaan johonkin laatukäsikirjastandardiin, joista yleisin on IEEE-730:2002 (IEEE, 2002).
  - IEEE-720:2002 on laatukäsikirjan rakennemalli, jota yritys voi käyttää oman käsikirjan runkona.
- Kukin alkava projekti joko käyttää laatukäsikirjaa sellaisenaan tai yleisemmin valitsee tehtävään tuotteeseen parhaiten sopivat laatukäsikirjan osat.

Laatukäsikirjan rakenne

- Hyvä laatukäsikirja on yrityksen, laadunvarmistus-ryhmän ja projektien lähdeteos. Se sisältää kolmen tason tietoa:
  - yritystason tietoa
  - projekteille yhteistä laadunvarmistustietoa
  - projekteittain hyödynnettävää tietoa
- Mitä isompi yritys, sitä laajempi laatukäsikirja. Varsin nopeasti laatukäsikirjasta tulee käsikirjasto (tai usea hyllymetri mappeja).

Laatukäsikirjan osat

- Yritystason tiedot, kuten
  - organisaatiokaavio
  - organisaation roolit/työnkuvat
  - roolien vastuut
- Laadunvarmistustiedot, kuten
  - dokumentointiohjeet
  - dokumentti- ja tarkistuslistapohjat
  - hallittavat dokumentit
- Projektikohtaiset tiedot, kuten
  - käytettävät prosessit
  - käytettävät menetelmät
  - käytettävät työkalut
  - tarkastuskäytännöt
  - seuranta- ja raportointitavat

4. Mittarit ja mittaus

- ”You can’t control what you can’t measure” – Tom DeMarco, 1982.
- DeMarcon toteama on kaikkien mittausspesialistien motto: ilman mittausta ei ole ohjausta.
  - Väite on tietenkin liioiteltu. Kaikkia merkittäviä tietoja ei osata, voida tai haluta mitata. Mutta: **mittaamalla ohjaus helpottuu.**
- Koska ohjelmiston laadunvarmistus on ohjausta, tarvitaan ohjelmiston laadun mittaamista ja ohjelmistojen *laatumittareita* (quality metrics).

Mittarit

- *Mittari* tai *metriikka* (metric) on jokin suoraan tai välillisesti kohteesta mitattava ominaisuus.
- *Mittaus* (measurement) tarkoittaa ominaisuuden arvon lukemista tietyllä hetkellä.
- Mittarit voivat olla *suoria* (direct) tai *johdettuja* (indirect).
  - Suorien mittareiden mittaukset kertovat sellaisenaan kohteen tilasta. Esimerkiksi lämpötila on suora mittari.
  - Johdettujen mittareiden mittaukset saadaan yhden tai useamman suoran mittarin mittausten funktiona. Esimerkiksi kuukauden keskilämpötila on johdettu mittari.
- Mittarien avulla voidaan arvioida, ennustaa tai tilastoida mittaajan kannalta mielenkiintoisia kohteen ominaisuuksia.

Ohjelmistomittarit

- Ohjelmistotuotannossa on paljon suoria ja johdettuja mittareita, *ohjelmistomittareita* (software metrics).
- Ohjelmistomittareilla ja laatutekijöillä on yhteys. Useimpien laatutekijöiden sisältä voidaan löytää osatekijöitä, joiden toteutumista voidaan arvioida mittareilla.
- Kun laatutekijän osatekijöiden mittarit näyttävät, että osatekijät toteutuvat ohjelmistossa, laatutekijä toteutuu ohjelmistossa *ainakin osittain*.
  - Miksi vain osittain?
    - laatutekijä on yleensä enemmän kuin osatekijöidensä summa
    - kaikille osatekijöille ei ole varmoja mittareita

Laatumittarit

- Laatumittarit mittaavat ohjelmistotuotteen tai ohjelmistoprosessin laatua.
  - Laatumittari on funktio, jonka syötteenä on ohjelmistoon liittyvä data ja tuloksena dataa kuvaava numeerinen arvo.
  - Laatumittarin numeerinen arvo (mittauksen tulos) kertoo, missä määrin mitattu data täyttää mittarilla kuvattavan laatuattribuutin.
    - Laatuattribuutti tarkoittaa jotakin laadun ominaisuutta. Esimerkiksi laatutekijät ja -osatekijät ovat laatuattribuutteja.

Laatumittarien tarkoitus

- Laatumittarit ovat laadunvarmistuksen hyödyllisimpiä työkaluja. Niiden avulla
  - varmennetaan, että ohjelmistoprojektit etenevät oikein:
    - verrataan toteutunutta laatua ja laatutason vaihtelua suunniteltuun laatuun
    - verrataan toteutunutta aikataulua ja budjettia suunniteltuun aikatauluun ja budjettiin
  - tunnistetaan, milloin kehitys- tai ylläpitoprosessia pitää parantaa:
    - kerätään historiatietoa toteutuneista projekteista ja tuotteista
- Mittarit voivat olla
  - prosessikohtaisia: miten hyvin prosessi ja sen ilmentymä (projekti) vastaavat toisiaan
  - tuotekohtaisia: miten hyvin tuote ja sen laatutekijät vastaavat toisiaan.

Laatumittarien valinta

- Laatumittareita valittaessa määrä ei korvaa laatua. On parempi valita pieni joukko hyvin määriteltyjä mittareita kuin mitata kaikkea mahdollista.
- Valittavat mittarit riippuvat siitä,
  - mikä on kiinnostavaa ja
  - mitä on ylipäänsä mahdollista mitata.
- Jotta mittarien käyttö olisi tehokasta, sen täytyy olla keskitettyä. Mittarit täytyy määritellä ylhäältä alaspäin (ensin tavoite, sitten siihen sopivat mittarit), sillä erilaisia mittareita on valtava määrä.
- Koska kaikkea ei voi eikä kannata mitata, valitut mittarit on perusteltava hyvin. Tähän voidaan käyttää *Goal Question Metric* -menetelmää (GQM, Basili & Rombach, 1988).

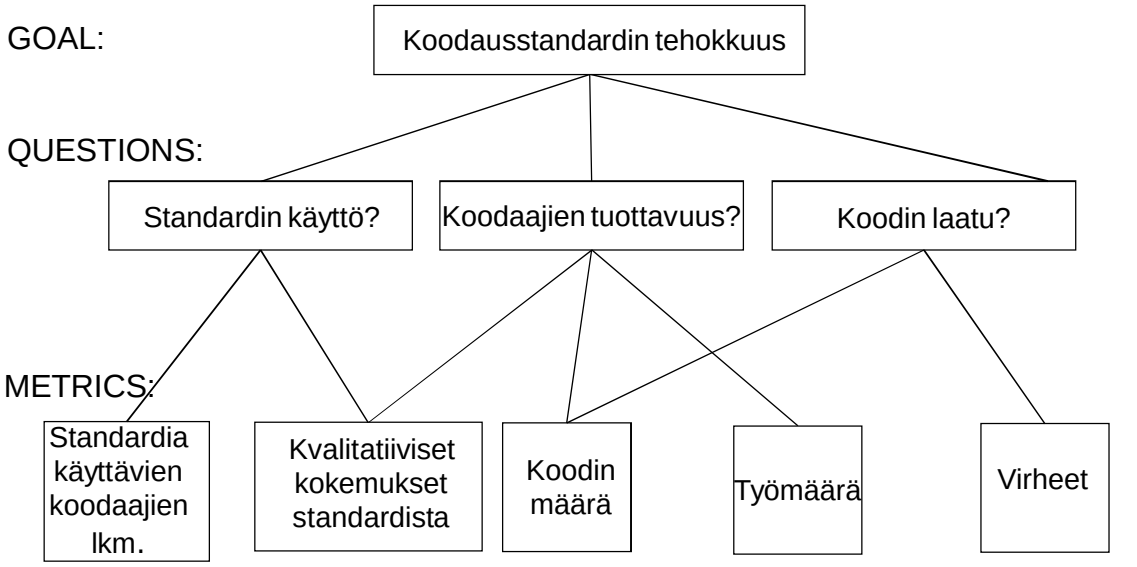
4.1. Goal Question Metric

- GQM perustuu näkemykseen, jonka mukaan järkevään mittaukseen yritys tarvitsee kolme vaihetta:
  1. Yrityksen on määriteltävä yritys- ja projektitason tavoitteet.
  2. Tavoitteiden kautta on löydettävä ne yritys- projekti- ja tuotetason tiedot, jotka kuvaavat tavoitteiden toteutumista.
  3. Yrityksen on suunniteltava kehys, jonka avulla kerätyt tiedot saadaan tulkittua kuvaamaan tavoitteita.
- Eli:
  1. Mitä tietoa yritys tarvitsee?
  2. Miten tietoja voidaan kerätä numeerisesti?
  3. Miten kerättyjä numerotietoja voidaan tulkita?
- GQM tarjoaa keinot määritellä kysymyksiin vastaava mittarijoukko.

GQM-kolmitasomalli

- GQM on kolmitasoinen malli:
  - Käsitetaso (GOAL)
    - Selvitetään halutut tuote-, prosessi- ja resurssitavoitteet.
  - Toimintataso (QUESTION)
    - Tavoitteiden perusteella johdetaan joukko kysymyksiä, jotka kuvaavat kunkin tavoitteen toteutumista.
    - Kukin kysymys kuvaa tavoitetta laatuun liittyvän ongelman tai faktan kannalta.
  - Kvantitatiivinen taso (METRIC)
    - Kuhunkin kysymykseen etsitään sellainen mittari, jonka avulla kysymykseen saadaan numeerisesti kuvattavissa olevia vastauksia.
- GQM:lla ei kannata tehdä liian kunnianhimoisia suunnitelmia. Aluksi on parempi etsiä perusmittarit ydinkysymyksiin ja sen jälkeen vähitellen parantaa mittarikehystä lisäämällä kysymyksiä ja niihin liittyviä mittareita.

GQM-esimerkki



4.2. Ohjelmiston kokomittarit

- Huomattavassa määrässä ohjelmiston laatumittareita mitataan niiden osana ohjelmiston kokoa. Vaikka erilaisia kokomittareita on lukemattomia, kaksi on ylitse muiden:
  - KLOC (Kilos Lines Of Code): Kuinka monta tuhatta koodiriviä on ohjelmistossa.
  - FP (Function Points): 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 selkeä mittari, joka kuvaa koon lisäksi yllättävän hyvin kompleksisuutta: 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 isoilla ohjelmistoilla erojen suhde koodirivien kokonaismäärään on melko pieni.
  - KLOC:n sijaan kannattaa laskea esimerkiksi puolipisteitä (so. lauseita), jos koodirivin pituuden vaihtelu osoittautuu ongelmaksi.

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

Toimintopisteet

- Toimintopisteet on esitelty niinkin aikaisin kuin vuonna 1979 (Albrecht). Vaikka menetelmää on tämän jälkeen kehitetty eteenpäin, on alkuperäinen malli edelleen yllättävän toimiva.
- Tällä hetkellä toimintopistemalleista huolehtii kokonainen organisaatio: International Function Point Users Group, IFPUG (www.ifpug.org).

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.
  - FP ei riipu käytettävistä ohjelmointikielistä tai työkaluista.
  - FP on osoittautunut melko luotettavaksi toiminnallisuuden mittariksi.
- Haitat:
  - Saadut FP-arvot ovat subjektiivisia. Ne riippuvat käytetyistä kertoimista, käytetystä FP-variantista ja arvioijasta.
  - FP on tarkoitettu dataintensiivisille sovelluksille. Se antaa liian pieniä arvoja algoritmi-intensiivisille sovelluksille. (Tämä tosin voidaan osittain korjata käyttämällä modernimpia FP-variantteja).
  - FP on pelkkä luku. Se ei sellaisenaan vielä merkitse mitään.

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.
- Seuraavassa muutamia suhdelukuja (LOC/FP) (1 KLOC = 1000 LOC). Luvut ovat suuntaa antavia:
  - 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.

4.3. Prosessin mittareita

- Kokomittojen määrittelyn jälkeen on helppo määritellä lukuisia prosessin tehokkuuden mittareita, joissa mitattavaa suuretta verrataan ohjelmiston kokoon. Tässä muutama:
  - Virhetiheys: CED = NCE / KLOC, missä NCE on projektin aikana ohjelmakoodista löydettyjen virheiden määrä.
  - Painotettu virhetiheys: WCED = WCE / KLOC, missä WCE on projektin aikana ohjelmakoodista löydettyjen virheiden määrä painotettuna niiden vakavuusasteella.
  - Virheiden poiston tehokkuus: DERE = NCE / (NCE+NYF), missä NYF on käytön 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.
- Näitähän riittää. GQM-menetelmällä tai vastaavalla voidaan etsiä yrityksen kannalta hyödyllisimmät mittarit.

Prosessin mittareita: esimerkki

- Olkoon ohjelmiston koko 40.000 riviä (KLOC=40). Siitä on löydetty projektin aikana 70 ohjelmakoodivirhettä, joista 11 on erittäin vakavia, 17 melko vakavia ja 42 vähäpätöisiä.
- Määritellään virheiden vakavuusasteille seuraavat painokertoimet: *vähäpätöinen*: 1, *melko vakava*: 3, *erittäin vakava*: 9.
  - Virhetiheys: CED = NCE / KLOC = 70 / 40 = 1,75.
  - Painotettu virhetiheys: WCED = WCE / KLOC = (9∗11+3∗17+1∗42) / 40 = 192 / 40 = 4,8.
- Olkoot mittareilla laatukynnysarvot CED ≤ 2 ja WCED ≤ 4. Tällöin CED ei ole viite ohjelmiston huonosta laadusta, mutta WCED on (liikaa erittäin vakavia virheitä).

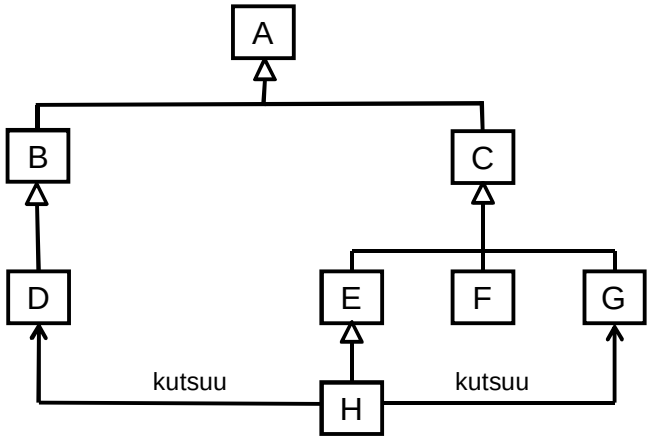
4.4. Suunnittelun mittarit

- Koska suunnittelun tuloksena syntyvä ohjelmakoodi on yksi prosessin tuotoksista, voidaan siitä laskea prosessin mittareita.
- Koodista laskettavia mittareita on lukuisia. Usein ne eivät suoraan liity mihinkään laatutekijään tai osatekijään, vaan niiden arvoja pitää tulkita esimerkiksi aiempien mittausten valossa.
- 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.

CK-mittarit

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

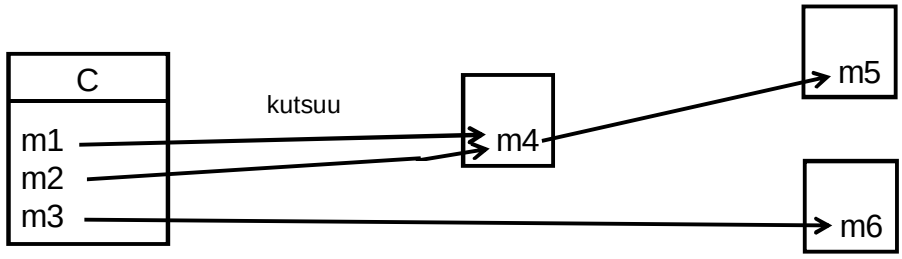
- *Painotettu luokan metodien määrä* (Weighted Methods per Class), WMC
  - Luokan metodien kompleksisuuksien (esim. syklomaattinen McCabe) summa.
  - Usein yksinkertaistettuna: metodien lukumäärä.
- *Perintäpuun syvyys* (Depth of Inheritance Tree), DIT
  - Luokkahierarkiassa pisimmän reitin pituus luokkasolmusta juureen.
- *Luokan lasten lukumäärä* (Number Of Children), NOC
  - Luokkahierarkiassa luokan välittömien aliluokkien lukumäärä.
- *Luokkien välinen sidonta* (Coupling Between Object classes), CBO
  - Luokkaan sidottujen muiden luokkien määrä. Kahden luokan välinen sidonta 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

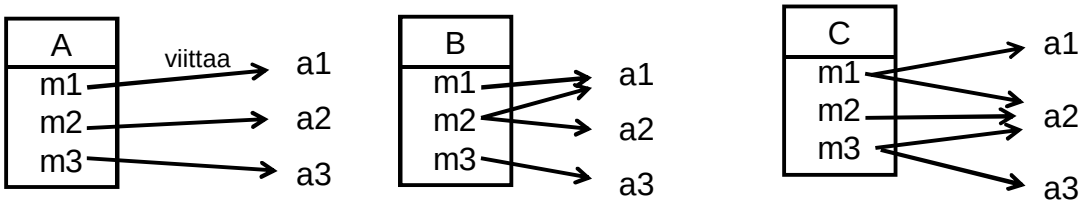
NOC (A) = 2, NOC (C) = 3, NOC (H) = 0

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

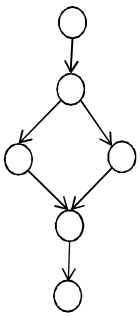
- CK-mittarit mittaavat olio-ohjelmiston staattisia ominaisuuksia, mutta mittaavatko ne ohjelmiston laatua? Tämä ei ole aivan triviaali kysymys.
- CK-mittareiden laadunennustuskyvystä on tehty paljon tutkimuksia. Tulokset vaikuttavat seuraavilta:
  - CK-mittarit ennustavat melko hyvin, mitkä luokat tulevat aiheuttamaan ongelmia (esim. niiden luotettavuus ja ylläpidettävyys ovat alhaiset).
    - LCOM ei ennusta kovin hyvin ongelmallisia luokkia
    - suuret DIT-, RFC-, NOC- ja CBO-arvot ennustavat jossain määrin ongelmaluokkia
  - 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).

4.5. Tuotteen mittareita

- 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).

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

4.6. 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, 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 tulkinta on usein subjektiivista. Klassinen esimerkki:
  - Olkoon testaustiimin A löytämien virheiden määrä *a* ja testaustiimin B löytämien virheiden määrä *b.* Jos *a* > *b*, niin onko testaustiimi A parempi kuin testaustiimi B?
  - Ei välttämättä. Vaihtoehtoja on useita:
    - A:n testaaman ohjelman virhetiheys 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 tulkinta on yhtä tärkeää kuin itse mittaus.

5. Laatustandardit

- *Laatustandardi* (quality standard) tarkoittaa yleensä kansallista tai kansainvälistä laatuun liittyvää ohjeistoa, joka on käynyt läpi usean raakaversion ennen hyväksymistä.
- Laatustandardit helpottavat laadunvarmistuksen suunnittelua ja hallintaa, sillä ne on laadittu huolella sellaisiksi, että ne sopivat sellaisenaan tai pienillä muutoksilla hyvin erilaisten yritysten laadunvarmistuksen rungoksi.
- Laatustandardeja ei ole pakko käyttää. Yritys voi kehittää itselleen sopivimman laadunvarmistusprosessin ja käyttää sitä menestyksekkäästi.
- Toisaalta hyvän prosessin kehittäminen on työlästä ja kallista, joten vähintään prosessiin kannattaa lainata oman yrityksen kannalta hyviä ideoita olemassa olevista standardeista.

Laatustandardien edut

- Yritys saa käyttöönsä kehittyneimmät kehitys- ja ylläpitomenetelmät
  - Laatustandardit on testattu erittäin huolellisesti ennen kuin niistä on tehty viralliset versiot
- Standardit parantavat projektiryhmien keskinäistä ymmärrystä ja koordinaatiota
  - Yhteinen standardi tarkoittaa yhteistä kieltä, mikä parantaa kommunikaatiota
- Yhteistyö ohjelmiston kehittäjien ja ulkoisten sidosryhmien välillä paranee
  - Laatustandardi on julkinen, joten ulkoiset sidosryhmät voivat sen kautta viitata projektin tai tuotteen laatuun
- Yhteistyö asiakkaiden ja alihankkijoiden kanssa paranee
  - Laatustandardi on usein asiakkaalle vakuuttavampi dokumentti kuin itse laadittu ohjeisto
  - Laatustandardi määrittelee (tai sen pitäisi määritellä) myös alihankkijoiden työn laadun

Laatustandardien ongelmat

- Standardi voi olla yrityksen kannalta liian yleisellä tasolla
  - Standardin pitää sopia eri tyyppisiin yrityksiin ja projekteihin melko hyvin, mutta useimpiin yrityksiin ja projekteihin se ei sovi sellaisenaan täydellisesti
- Standardi voi olla yritykselle liian raskas
  - Varsinkin pienten yritysten voi olla vaikeaa toteuttaa haluttua laatustandardia, koska niissä ei ole tarpeeksi resursseja kaikkien laatustandardissa määriteltyjen tehtävien toteuttamiseen
- Standardi ei välttämättä ole yhteensopiva yrityksen nykyisten toimintatapojen kanssa
  - Laatustandardissa määritellään melko tarkasti, mitä kaikkea laadunvarmistukselta vaaditaan, mutta yrityksessä käytetyt toimivat menetelmät voivat olla sellaisia, että ne eivät taivu laatustandardiin ilman perinpohjaista muutosta sekä yritys- että projektitasolla

5.1. ISO 9000 -sarja

- *ISO 9000* (International Organization for Standardization) on laaja laadunhallintastandardien perhe
- Ohjelmistojen laadunvarmistuksen kannalta ISO 9000 -sarjan mielenkiintoisin standardi on ISO 9001, joka määrittelee yleiset laadunhalllinnan vaatimukset ja erityisesti ISO 90003, joka määrittelee, miten ISO 9001 -standardia sovelletaan ohjelmistotekniikassa
- Sekä ISO 9001 että ISO 90003 päivitetään säännöllisin väliajoin. Viimeisimmät versiot ovat ISO 9001:2008 ja ISO 90003:2004.

ISO 90003 -sertifiointi

- ISO 90003:n sertifiointiprosessilla varmistetaan, että sertifioitavan yrityksen ohjelmistojen kehitystyö- ja ylläpitoprosessit täyttävät standardin vaatimukset
- Sertifioinnin voi tehdä ISO:n hyväksymä sertifioija, jonka edustajat tarkastavat sertifiointia haluavan yrityksen prosessit objektiivisesti
- Hyväksytty ISO 90003 -sertifiointi on yritykselle selkeä kilpailuetu, sillä se kertoo asiakkaille, että yritys on standardoinut ja tarkastuttanut prosessinsa
- ISO 90003 -sertifioituun yritykseen tehdään säännöllisesti tarkastuskäynti, jolla varmennetaan, että yritys ei lipsu standardista

5.2. Kypsyysmallit

- *Kypsyysmalli* (capability model) on ohjeisto, jonka avulla mitataan ja parannetaan organisaation prosesseja.
- Hyvä kypsyysmalli tarjoaa hyväksi havaitun ja empiirisesti testatun kehyksen, johon organisaation omat prosessit pyritään sijoittamaan. Mitä paremmin prosessit sopivat kehykseen, sitä lähempänä tätä kypsyysmallia organisaation prosessit ovat.
- Yleensä kypsyysmallit ovat monitasoisia. Mitä korkeammalle tasolle yrityksen prosessit sopivat kypsyysmallissa, sitä kehittyneemmät ne ovat kypsyysmallin näkökulmasta.
- Kypsyysmalleja käytetään parantamaan prosesseja
  - ensin katsotaan, mille tasolle organisaation nykyiset prosessit sijoittuvat kypsyysmallissa
  - tämän jälkeen katsotaan, mitä muutoksia prosesseihin tarvitaan, jotta ne nousisivat kypsyysmallin seuraavalle tasolle

CMM

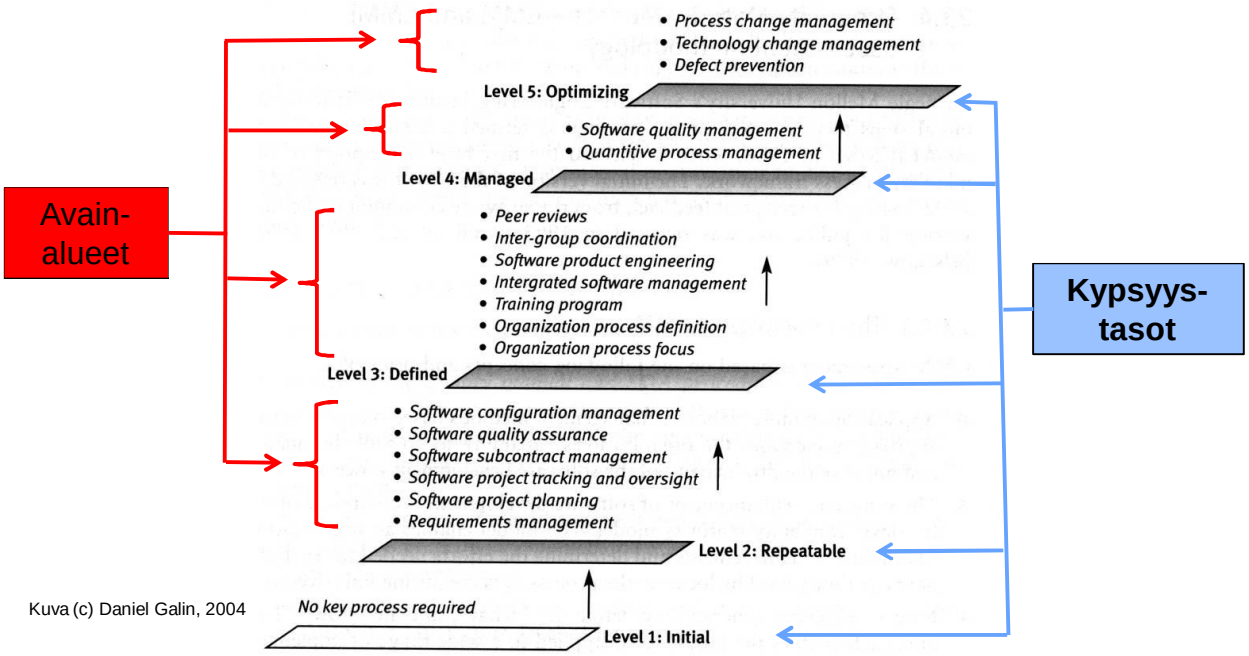
- Kypsyysmallien historia alkaa vuodesta 1986, jolloin Carnegie Mellon -yliopiston Software Engineering Institute (SEI) esitteli kypsyysmallien idean tiedeyhteisölle.
- Ensimmäinen kypsyysmalli *CMM* (Capability Maturity Model) eli SEI-CMM kehitettiin samassa ohjelmistotekniikan instituutissa, josta kypsyysmallien idea on lähtöisin. CMM julkaistiin oppikirjana 1989 ja käytön oppaana 1993.
- Nykyisin on olemassa kymmeniä kypsyysmalleja eri tyyppisille prosesseille ja organisaatioille. Suuri osa näistä on sukua CMM:lle.

- Alkuperäinen oppikirja:
  W.S. Humphrey: *Managing the Software Process*. Addison-Wesley, 1989.

CMM:n periaatteet

- CMM perustuu seuraaviin periaatteisiin (Galin, 2004):
  - Kvantitatiivisiin menetelmiin perustuva prosessien hallinnointi parantaa organisaation kykyä valvoa ohjelmistojensa laatua ja parantaa niiden kehitysprosesseja.
  - Kehitystyön parantamisen työkaluna käytetään viisitasoista kypsyystasomallia. Mallin avulla organisaatio voi arvioida omaa toimintaansa ja arvioida seuraavalle kypsyystasolle vaadittavaa työmäärää. Työmäärää arvioidaan selvittämällä ne kypsyystasoon kuuluvat tehtävät, jotka vaativat parantamista.
  - Tehtävät ovat geneerisiä. Ne määrittelevät, mitä pitää tehdä, mutta eivät, miten se pitää tehdä.
- CMM perustuu siis
  - viisitasoiseen kypsyystasomalliin ja
  - kuhunkin tasoon kuuluviin tehtäviin, joita kutsutaan *avainalueiksi* (key process areas, KPAs)

CMM:n kypsyystasot ja avainalueet



Kuva (c) Daniel Galin, 2004

CMM-arviointi

- Organisaation kypsyys tietyn CMM-tason suhteen arvioidaan tutkimalla, missä määrin organisaation projekteissa noudatetaan ko. tasolle kuuluvia avainalueita
- Periaate 1: ollakseen CMM-kypsyystasolla *n* organisaation *kaikissa* ohjelmistoprojekteissa on noudatettava *kaikkia* tason *n* avainalueita
- Periaate 2: ollakseen CMM-kypsyystasolla *n* organisaation on noudatettava myös tasojen 1…*n-1* kaikkia avainalueita
- CMM-arvioinnissa organisaatiolle tuotetaan profiili, joka kertoo, missä määrin se noudattaa kypsyystasojen 1-5 avainalueita
- Profiilista näkyy, (1) millä CMM-kypsyystasolla organisaatio on tällä hetkellä, ja (2) mihin avainalueisiin sen on panostettava noustakseen seuraavalle kypsyystasolle

CMM:n avainkäytännöt

- CMM:n avainalueille on määritelty joukko *avainkäytäntöjä* (key practices) eli tyypillisiä ko. avainalueen toimintatapoja
- Periaate 1: noudattaakseen tiettyä avainaluetta organisaation on toimittava sille määriteltyjen avainkäytäntöjen mukaisesti, so. toteutettava avainalue avainkäytäntöjen määräämällä tavalla
- Periaate 2: perustelluista syistä organisaatio voi käyttää myös muita (itse määriteltyjä) avainalueen toteutustapoja kuin CMM:ssä määriteltyjä avainkäytäntöjä
- CMM on siis tarkentuva malli kypsyystasoista avainalueisiin ja niistä edelleen avainkäytäntöihin
- Kypsyystasoja on 5, avainalueita 18 ja avainkäytäntöjä satoja
- Kypsyystasolla 1 (*Initial*) ole avainalueita eikä avainkäytäntöjä: mikä tahansa ”prosessi” kelpaa

Esimerkki CMM:n avainkäytännöistä: *Software quality management* (Taso 4: Managed)

- projektissa noudatetaan organisaatiotasolla kirjattua laadunhallintaa
- ohjelmistotuotteiden laadunhallinnalla on riittävät resurssit ja rahat
- laadunhallintaan osallistuvat saavat siihen tarvittavaa koulutusta
- projektiryhmän jäsenet saavat koulutusta laadunvarmistukseen
- laatukäsikirja on perusta projektissa tehtävälle laadunvarmistukselle
- projektin laatukäsikirjaa kehitetään kirjattujen käytäntöjen mukaisesti
- projektille on määritelty kvantitatiiviset laatutavoitteet ja -mittarit, joita kehitetään koko projektin keston ajan
- projektin ohjelmistotuotteiden laatua mitataan ja verrataan kvantitatiivisiin laatutavoitteisiin jatkuvasti
- laatutavoitteet koskevat myös mahdollisia alihankkijoita
- laadunhallinnan tilannetta ja kustannuksia mitataan
- organisaation ylin johto valvoo säännöllisesti laadunhallintaa
- projektipäällikkö seuraa jatkuvasti projektinsa laadunvarmistuksen toimenpiteitä
- laadunvarmistusryhmä seuraa jatkuvasti organisaation laadunhallinnan tilaa

CMM:n elinkaari

- Pian julkistuksen jälkeen SEI laajensi CMM-mallia sopimaan eri tyyppisiin prosesseihin. Tulokseksi saatiin kuusi CMM-varianttia, jotka eivät olleet täysin keskenään yhteensopivia.
- CMM-variantit aiheuttivat ongelmia tilanteissa, joissa saman organisaation eri yksiköt käyttivät eri CMM-varianttia. Tämän ongelman korjaamiseksi SEI kehitti CMM:n seuraajan, *CMMI*:n (Capability Maturity Model Integrated).
- CMM:n ylläpito päättyi vuonna 2007. Nykyisin SEI ylläpitää vain CMMI:ta.

CMMI

- Edeltäjänsä tavoin CMMI perustuu viisitasoiseen kypsyystasomalliin. Mallin tasot ovat:
  1. Lähtötaso (Initial)
  2. Hallittu (Managed)
  3. Määritelty (Defined)
  4. Kvantitatiivinen (Quantitatively managed)
  5. Optimoituva (Optimizing)
- Kuhunkin tasoon (lähtötasoa lukuun ottamatta) kuuluu joukko *prosessialueita* (process areas, PAs). Jos organisaatio noudattaa kaikkia tietyn tason prosessialueita, se kuuluu prosessialueet määrittelevälle kypsyystasolle. Prosessialueet (yhteensä 24) vastaavat CMM:n avainalueita (18).
- Jokaiselle prosessialueelle määritellään CMMI:ssa tavoitteet, menetelmät ja toimintatavat. Ne vastaavat CMM:n avainkäytäntöjä.

CMMI:n prosessialueet 1

- Lähtötaso (0 kpl)
  - ei prosessialueita
- Hallittu taso (7 kpl)
  - vaatimusten hallinta (requirements management)
  - projektin suunnittelu (project planning)
  - projektin seuranta ja ohjaus (project monitoring and control)
  - toimittajasopimusten hallinta (supplier agreement management)
  - mittaus ja analyysi (measurement and analysis)
  - prosessin ja tuotteen laadunvarmistus (process and product quality assurance)
  - versionhallinta (configuration management)

CMMI:n prosessialueet 2

- Määritelty taso (13 kpl)
  - vaatimusten evoluutio (requirements development)
  - tekninen ratkaisu (technical solution)
  - tuotteen integrointi (product integration)
  - verifiointi (verification)
  - validointi (validation)
  - organisaatioprosessien fokusointi (organizational process focus)
  - organisaatioprosessien määrittely (organizational process definition)
  - organisaatiotason koulutus (organizational training)
  - integroitu projektinhallinta (integrated project management)
  - integroitu tiimityö (integrated teaming)
  - riskienhallinta (risk management)
  - päätösanalyysi (decision analysis and resolution)
  - organisaatiotason integroinnin tuki (organizational environment for integration)

CMMI:n prosessialueet 3

- Kvantitatiivinen taso (2 kpl)
  - organisaatioprosessien suorituskyky (organizational process performance)
  - kvantitatiivinen projektinhallinta (quantitative project management)
- Optimoituva taso (2 kpl)
  - organisaatiotason innovointi ja sitoutuminen (organizational innovation and deployment)
  - jatkuva tietojen analysointi ja ongelmien ratkonta (casual analysis and resolution)

CMM-muunnelmat

- 1990-luvun CMM-huumassa luotiin myös joukko muita kuin organisaatioiden ja ohjelmistoprosessien arviointiin ja kehittämiseen tarkoitettuja kypsyysmalleja
- Niissä on sama perusajatus kuin alkuperäisessä CMM:ssä: (1) kehittämisen kohde jaetaan viiteen kypsyystasoon, joille kullekin määritellään joukko tärkeimmiksi katsottuja toimenpiteitä; (2) kehittymistä ohjataan mittaamalla, kuinka hyvin kyseisiä toimenpiteitä noudatetaan
- CMM-huuma kuoli 2000-luvun alussa, kun laajaan suosioon nousseissa ketterissä prosesseissa julistettiin prosessit ja turha ”seremonia” ohjelmistoalan vitsauksiksi, joista pitää päästä eroon (tosin historia tuppaa toistamaan itseään…)

Personal Software Process 1

W.S. Humphrey: *Introduction to the Personal Software Process*. Addison-Wesley, 1997.

- henkilökohtaisen ”ohjelmistoprosessin” (PSP) mittaaminen
- oman ohjelmistokehitystyön (”projektin”) mittaaminen
- projektisuunnittelun parantaminen mittaustulosten perusteella
- työn laadun parantaminen mittaustulosten avulla

Personal Software Process 2

Suunnittelu

- ajankäytön mittaaminen ja kirjanpito
- työajan ja tehtävien jakaminen osiin
- työmäärän arviointi
- ohjelmarivien (LOC) mittaaminen
- tuottavuuden (LOC / h) mittaaminen
- ajankäytön hallinta (mittaaminen ⇒ muutokset)
- tehtävien priorisointi
- aikataulun laatiminen, seuranta ja hallinta (GANTT-kaavio)
- projektisuunnitelman laatiminen, seuranta ja hallinta

Työprosessi

- oman työprosessin määrittely
- (ohjelmointi)virheiden luokittelu ja kirjaaminen
- virheenjäljitys ja -korjaus
- virheenjäljitys- ja korjausajan mittaaminen
- virhetiheyden (virheitä / KLOC) mittaaminen
- koodin katselmointi tarkistuslistan avulla
- laadun valvonta: virhemäärien ja virhetiheyden seuranta vaiheittain
- prosessin valvonta: katselmointiaika vs. testausaika vs. ohjelmointiaika

People CMM 1

B. Curtis, W. Hefley, S. Miller: *The People Capability Maturity Model*. Addison-Wesley, 2002.

- henkilöstöhallinnon kypsyysmalli
- kypsyystasot:

1 Initial (epäjohdonmukainen henkilöstöhallinto)
2 Managed (työntekijöiden hallinta)
3 Defined (osaamisen hallinta)
4 Predictable (osaamisen kehittäminen)
5 Optimizing (muutoksen hallinta)

People CMM 2

Avainalueet:

2. *Managed*: rekrytointi, kommunikointi ja koordinointi, työsuoritusten hallinta, koulutus ja kehittyminen, palkkiojärjestelmä, työympäristö
3. *Defined*: organisaation osaamisanalyysi, organisaation osaamisen kehittäminen, osaamiseen perustuvat käytännöt, urakehitys, työryhmien kehittäminen, henkilöstörakenteen suunnittelu, osallistuva työkulttuuri
4. *Predictable*: osaamisen integrointi, mentorointi, tiimityö, osaamisvaranto, kvantitatiivinen suoritusten hallinta, henkilöstön kyvykkyyden hallinta
5. *Optimizing*: jatkuva henkilöstön kyvykkyyden parantaminen, organisatorinen suoritustason linjakkuus, jatkuva innovointi

SPICE

- Vaikka CMM(I) on ollut yleisesti hyväksytty ja arvostettu, se ei ole kansainvälinen standardi.
- CMM(I):ta vastaavia laatustandardeja on muitakin, sekä organisaatioiden määrittelemiä että kansallisia.
- Koska kansainväliselle laatustandardille on ollut kysyntää, ISO ja IEC aloittivat 1993 tällaisen standardin suunnittelun. Tuloksena saatiin ISO/IEC 15504 eli *SPICE* (Software Process Improvement for Capability dEtermination).
- SPICE määrittelee CMM(I):n tapaan kypsyystasomallin, mutta siinä missä CMM(I) keskittyy organisaation arviointiin, SPICE keskittyy prosessien arviointiin.

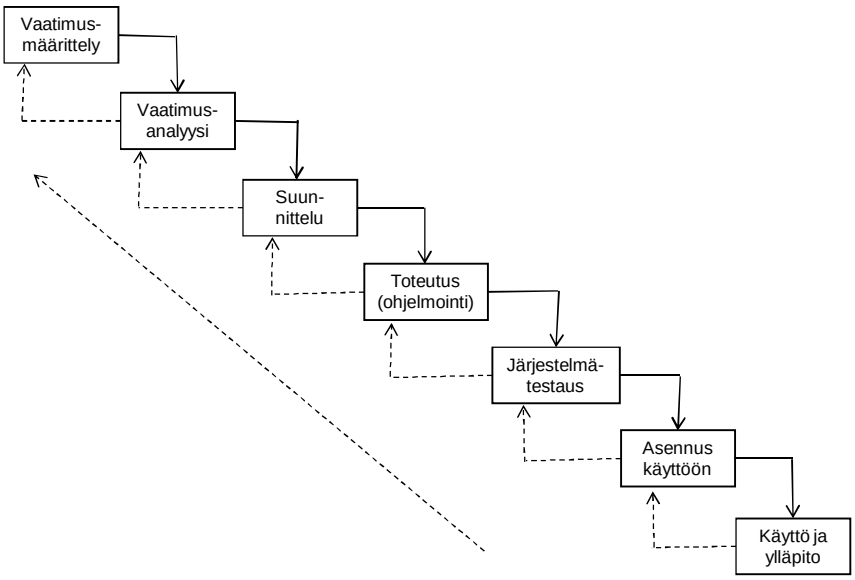
SPICEn tasot ja prosessiattribuutit

- SPICEssa prosesseille on määritelty yhdeksän prosessiattribuuttia, joista kukin sisältää joukon periaatteita
  - SPICElla arvioidaan, miten hyvin organisaation prosessit täyttävät annetut prosessiattribuutit. Luokittelu on asteikolla N,P,L,F (Not achieved, Partially achieved, Largely achieved, Fully achieved).
- Prosessiattribuutit jaetaan kuudelle tasolle, jotka samalla määrittelevät prosessille SPICEn mukaisen kypsyystason.
- Tasot ja prosessiattribuutit ovat
  - keskeneräinen (incomplete): ei prosessiattribuutteja
  - suoritettu (performed process): prosessin suoritus
  - ohjattu (managed process): suorituksen hallinta, lopputuloksen hallinta
  - vakiinnutettu (established process): prosessin määrittely, prosessin resurssien hallinta
  - ennustettava (predictable process): mittaukset, prosessin hallinta
  - optimoituva (optimizing process): prosessin muutoksen hallinta, jatkuva prosessin parannus

6. Iteratiivinen ohjelmistokehitys

- Yleinen (tai ainakin ”virallinen”) käsitys 1990-luvun loppupuolelle asti oli, että laatu saavutetaan huolellisella määrittelyllä ja hallinnalla.
- Käytännössä tämä tarkoitti, että kaikki oleelliset asiat kirjattiin noudatettaviksi säännöiksi. Sääntöjä voitiin toki muuttaa, jos ne osoittautuivat huonoiksi, mutta muutosprosessi oli hidas.
- Ohjelmistoja suunniteltiin ja toteutettiin samoilla periaatteilla kuin mitä tahansa teollisuustuotteita. Tämä on *ennustettavissa olevaa valmistusta* (predicatble manufacturing), siis massatuotantoa.

Klassinen vesiputousmalli



Vesiputousmallin hyvät ja huonot puolet

- Hyvää: selkeä, yksinkertainen, ennalta suunniteltavissa, vakiintunut ohjelmistotuotannon vaihejako
- Huonoa: liian kankea muutoksille, epärealistinen, liian myöhäinen ja kallis ongelmien havaitseminen, liian myöhään testattavissa

Massatuotanto

- Massatuotannon vaatimuksena on toimiva spesifikaatio: sinikopio tehtävästä tuotteesta.
- Spesifikaatio ei tule tyhjästä, vaan sen saamiseksi tehdään paljon suunnittelua ja prototyyppejä yrityksen ja erehdyksen kautta. Tämä on *uuden tuotteen kehitystä* (new product development).
- Perinteisessä ohjelmistokehityksessä uuden tuotteen kehitystä tehtiin osana massatuotantoa. Tekniikka ei sopinut mihinkään ohjelmistoprojekteihin hyvin ja harvoihin edes kohtalaisesti.

Massatuotannon ja kehitystyön vertailu

Perinteistä ohjelmistotuotantoa lukuun ottamatta missään insinöörityössä ei yritetä yhdistää massatuotantoa ja uuden tuotteen kehitystä. Alueet eroavat selvästi toisistaan:

Massatuotanto	Kehitystyö
Lopullinen spesifikaatio saadaan käyttöön ennen tuotantoa.	Valmis muuttumaton spesifikaatio ei yleensä ole saatavilla.
Luotettava kustannus- ja työmääräarvio saadaan ennen tuotantoa.	Kustannus- ja työmääräarvio tarkentuu huomattavasti työn aikana.
Vaadittavat tehtävät voidaan tunnistaa, määritellä, skeduloida ja järjestää etukäteen.	Tehtävät tarkentuvat ja mukautuvat projektin aikana. Toteutus-palaute-sykli tarvitaan.
Tuotantoaikaisten muutosten määrä on suhteellisen pieni. Ennustamattomiin tapahtumiin ei juuri varauduta.	Jatkuva varautuminen ennustamattomiin muutoksiin. Muutosten määrä on suuri.

Havaintoja

- **Ohjelmistotuotanto ei ole ennustettavaa rutiinimaista massatuotantoa vaan uuden tuotteen kehitystyötä**.
- Useimmissa ohjelmistotuotantoprojekteissa massatuotannon menetelmät eivät toimi. Ennalta määritellyt aikataulu, budjetti ja tehtävät eivät toteudu, koska valmista spesifikaatiota ei ole saatavilla ennen projektin alkua.
- Ohjelmistotuotannossa on varauduttava muutokseen. Lähes jokaisen ohjelmiston spesifikaatio muuttuu kehitysprojektin aikana.

- Kuitenkin: ohjelmistotuotantoon pyritään 2010-luvulla jälleen ottamaan mallia massatuotannosta, kun päivän hypetys ”virtaviivainen” (lean) ohjelmistokehitys perustuu Japanissa 1950-luvulla luotuihin autonvalmistuksen periaatteisiin.

Valmiin spesifikaation ongelma

Entä jos spesifikaation saisi valmiina ja muuttumattomana? Tämä onnistuu valitettavan harvoin:

- sidosryhmät eivät tiedä, mitä haluavat ohjelmistolta
- heidän on vaikeaa ilmaista toiveitaan ja taustatietojaan
- suuri osa yksityiskohdista selviää vasta kehitystyössä
- yksityiskohtien määrä ja kompleksisuus ylittävät aivojen hahmotuskyvyn
- sidosryhmät muuttavat mieltään projektin aikana
- ulkoiset tekijät (kuten kilpailijat ja taloustilanne) vaikuttavat spesifikaatioon

Evoluutio ja iteratiivisuus

- Vaikka ohjelmistotuotannon pitää olla uuden tuotteen kehitystyötä, sen ei tarvitse olla hallitsematonta.
- Ohjelmistoprojektilla on aikataulu, budjetti, tehtävät ja spesifikaatio. Niitä ei kuitenkaan kiinnitetä projektin alussa, vaan ne elävät koko projektin ajan. Tämä on *evoluutiolähtöistä kehitystyötä* (evolutionary development).
- Yleensä tällaiset projektit ovat *iteratiivisia:* kehitystyö tehdään askel kerrallaan pieninä miniprojekteina, joista jokainen sisältävää perustehtäviä, kuten vaatimusmäärittelyä, suunnittelua, toteutusta ja validointia.

Ideoista prosessimalleihin

- *Iteratiivinen (ja evoluutio-)ohjelmistokehitys* (iterative and evolutionary software development) on prosessimallien perhe, jossa ohjelmiston elinkaari muodostuu useasta peräkkäisestä iteraatiosta.
  - Yhden iteraation suositeltava pituus on 1-6 viikkoa. Siihen kuuluu aina toteutusta eli uuden koodin kirjoittamista.
  - Useimmissa projekteissa on ainakin kolme iteraatiota.
- Iteratiivinen ohjelmistokehitys on vanha idea. Ensimmäiset iteratiiviset prosessit otettiin käyttöön 1960-luvulla, mutta 1970-luvulla ne menettivät teollisuudessa suosiota lineaarisille prosessimalleille.

Julkaisu

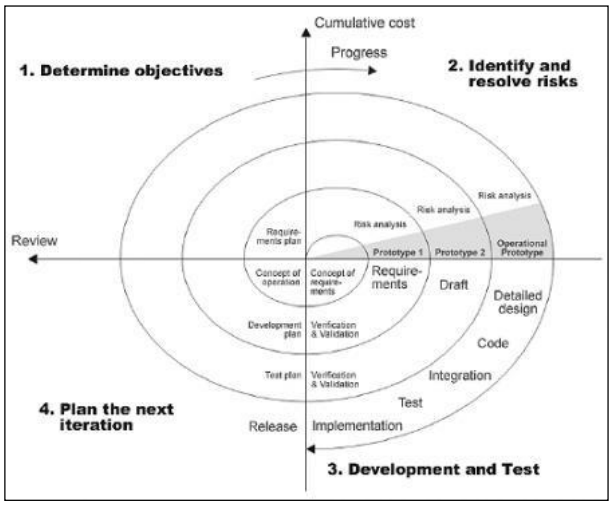
- Jokaisen iteraation tuloksena saadaan *julkaisu* (iteration release).
  - Julkaisu on lopullisen järjestelmän toimiva osa. Prototyyppi ei kelpaa julkaisuksi.
    - Käytännössä projekteissa on myös iteraatioita, joista ei synny uutta julkaisua. Näin käy silloin, kun koodia täytyy refaktoroida (eli siistiä) tai kun ohjelmistoa ei ole saatu testattua tarpeeksi aiemmissa iteraatioissa.
  - Kaikki julkaisut eivät ole julkisia eli loppukäyttäjälle tarkoitettuja.
  - Sisäiset julkaisut ovat kehitystiimin ja asiakkaan käytössä.
  - Julkiset julkaisut ovat loppukäyttäjälle tarkoitettuja versioita tuotteesta.
  - Joskus viimeinen julkaisu on lopullinen tuote, mutta nykyisin kehitystyö jatkuu yleensä tuotteen koko elinkaaren ajan. Tällöin tuotteesta tehdään useita julkisia julkaisuja.
  - Jos projektissa on monta kehitystiimiä, julkaisuun integroidaan kaikkien kehitystiimien tuotokset.

Iteratiivinen ja lisäävä kehitystyö

- Ohjelmistotyötä, jossa tuotteeseen lisätään sykleittäin uusia ominaisuuksia, kutsutaan *lisääväksi ohjelmistokehitykseksi* (incremental development)
- Jos lisäykset liittyvät iteraatioihin, saadaan *iteratiivinen ja lisäävä ohjelmistokehitys* (Iterative and Incremental Development, IID)
- Lähes kaikki modernit ohjelmistoprosessit ovat IID-variantteja
- Erityisesti kaikki nykyiset *ketterät* prosessit ovat IID-variantteja

Spiraalimalli

Iteratiivisten ja lisäävien ohjelmistoprosessimallien klassikko:
spiraalimalli (spiral model, Boehm 1986)



– perustana vesiputousmallin vaihejako
– mukana riskienhallinta
– joka iteraation alussa ko. iteraation suunnittelu: tavoitteet, vaihtoehdot, rajoitteet
– joka iteraation lopussa tulosten arviointi
– viimeisessä iteraatiossa lopputuotteen ohjelmointi ja testaus

Iteraation pituus

- IID ei määrittele iteraation pituutta, mutta moderneissa iteratiivisissa prosessimalleissa iteraatiot ovat lyhyitä.
  - Vanhemmissa prosessimalleissa *syklin* (cycle) pituus on 3-6 kuukautta, joskus jopa enemmän.
    - Sykli on hiukan eri asia kuin iteraatio. Sykli tarkoittaa aikaa, missä käydään läpi kaikki työvaiheet vaatimusmäärittelystä testaukseen. Tuloksena ei välttämättä saada julkaisua vaan esimerkiksi prototyyppi. Iteraation tuloksena saadaan aina julkaisu.
  - Lyhyt sykli tarkoittaa nopeaa reagointia muutokseen. Toisaalta se myös tarkoittaa, että tehtävä ohjelmisto on ositettava hyvin pieniin toteutettaviin osiin ja että tuottamaton työ lisääntyy projektissa (syklin vaihtoon liittyvä työ).

Iteraation työvaiheet

- Iteraatiossa perustehtävien painotukset vaihtelevat sen mukaan, missä vaiheessa tuotteen elinkaarta ollaan:
  - ensimmäisillä iteraatioilla painotetaan vaatimusmäärittelyä ja vaatimusten validointia
  - myöhemmillä sykleillä vaatimusmäärittelyn paino vähenee ja suunnittelun paino lisääntyy
  - vielä myöhemmillä sykleillä vaatimusmäärittelyn ja suunnittelun paino vähenee ja toteutuksen ja järjestelmätestauksen paino lisääntyy
  - ylläpitovaiheessa toteutuksen (plus refaktoroinnin) ja regressiotestauksen paino lisääntyy

Riskilähtöinen ja asiakaslähtöinen iteraatiosuunnittelu

- Iteraatiossa toteutettavat ominaisuudet voidaan valita *riskilähtöisesti* (risk-driven) tai *asiakaslähtöisesti* (client-driven)
  - riskilähtöisessä kehitystyössä iteraatioon valitaan ne tehtävät, jotka ovat vaikeimmat ja riskialtteimmat toteuttaa
  - asiakaslähtöisessä kehitystyössä iteraatioon valitaan ne tehtävät, jotka asiakas asettaa korkeimmalle prioriteetille
- Riskilähtöinen kehitystyö helpottaa projektia, mutta asiakas ei välttämättä pidä lähestymistavasta. Usein syvällä olevat vaikeat ratkaisut eivät suoraan näy parantuneena toiminnallisuutena.
- Asiakaslähtöinen kehitystyö pitää asiakkaan tyytyväisenä, mutta voi johtaa raskaaseen refaktorointiin tai huonoon arkkitehtuuriin.

Aikaikkunat 1

- IID:ssa iteraation kesto kiinnitetään etukäteen. Tätä kutsutaan *timeboxing*-tekniikaksi (aikaikkuna, aikasidonnaisuus).
- Aikaikkuna-tekniikassa iteraation kestoaika ei jousta. Jos allokoituja tehtäviä on jäljellä enemmän kuin iteraatiolle varattuun aikarajaan mahtuu, niitä karsitaan.
- Iteraation tuloksena saadaan aina sovittuna ajankohtana julkaisu eli lopullisen tuotteen toimiva osa.

Aikaikkunat 2

- Aikasidonnaisuus ei tarkoita, että tiimiläisten täytyy tehdä ylitöitä pitääkseen iteraation takaraja
  - joskus ylityöt ovat ok, kunhan sekä tiimi että projektin johto hyväksyvät ne
  - pääsääntöisesti pitäisi kuitenkin seurata normaalia työviikkoa ja ylitöiden sijaan karsia toteutettavia ominaisuuksia
- Aikasidonnaisuus ei tarkoita, että kaikkien iteraatioiden olisi oltava saman mittaisia. Projektin iteraatioiden pituudet voivat vaihdella keskenään, mutta yhden iteraation pituus ei muutu sen aloituksen jälkeen.
- Kaikki yleisimmät ketterät prosessimallit suosittelevat vahvasti tai jopa vaativat aikaikkunoiden käyttöä.

Muutokset iteraatioissa

- IID:ssa varaudutaan muuttuviin vaatimuksiin, mutta ei koska tahansa. Muutoksiin varaudutaan iteraatioiden välillä, ei niiden sisällä.
- Iteraation käynnistymisen jälkeen ulkoiset sidosryhmät eivät vaikuta sen sisältöön:
  - kun tiimi on aloittanut sovitun iteraation toteutuksen, asiakas, projektin vastuuhenkilö tms. ei voi pyytää tai käskeä tiimiä tekemään ylimääräistä
  - tiimi itse voi vähentää iteraatiossa tehtäviä tehtäviä, jos aikaikkuna ei muuten pidä
  - jos iteraatio on menossa todella pahasti metsään, se voidaan keskeyttää ja aloittaa uusi iteraatio etuajassa

IID ja vaatimusmäärittely 1

- IID:n iteraatiossa toteutetaan sovitut tehtävät, mutta mistä ne saadaan?
- Tehtävät saadaan alun perin *vaatimusmäärittelystä* aivan kuten suunnitelmakeskeisissä prosesseissa.
- Vaatimusmäärittelyä tehdään rinnan ohjelmisto-kehityksen kanssa. Vaatimusmäärittely voi olla osa iteraatiota, mutta se voi myös olla iteraatioista erillään.
- IID:n kannalta riittää, että kussakin iteraatiossa tiedetään, mitä tehtäviä siihen kuuluu.

IID ja vaatimusmäärittely 2

- IID sopii projekteihin, joissa vaatimukset muuttuvat. Mutta
  - miten paljon vaatimukset muuttuvat?
  - mitkä vaatimukset saavat muuttua?
- Useimmissa projekteissa *suurin osa* vaatimuksista on melko stabiileja. Nämä vaatimukset tunnistetaan (ja toteutetaan) aikaisissa iteraatioissa.
- Osa vaatimuksista elää. Asiakas ei osaa heti kertoa, mitä tarvitaan, tai ongelmakenttä muuttuu. Nämä vaatimukset tunnistetaan ja toteutetaan projektin edetessä.
- Tärkeimpiä tunnistettavia vaatimuksia ovat laatutekijöihin liittyvät vaatimukset. Ne pitää tunnistaa mahdollisimman aikaisin, jotta ohjelmiston arkkitehtuuri saadaan räätälöityä niille sopivaksi. Niiden pitää myös olla mahdollisimman stabiileja.
  - Yleensä laatutekijöiden stabiilius ei ole ongelma. Lähinnä tehokkuus-laatutekijä voi tuottaa ongelmia.

IID ja aikataulu

- Yksi tavallisimmista IID:n kritiikin kohteista on projektin aikataulutus. IID:n mukautuvan luonteen johdosta varsinkin projektin alussa on vaikea tehdä edes alustavaa aikataulua.
- Keskeiset termit ovat *projektin alussa* ja *alustava aikataulu*:
  - jo muutaman iteraation jälkeen tiimillä on useimmiten jo melko hyvä käsitys siitä, kuinka monta iteraatiota työ kaiken kaikkiaan vaatii
  - mikä tahansa projektin alussa tehty aikataulu on vain alustava, ja myös suunnitelmakeskeisten projektien alussa tehdyt aikataulut ovat summittaisia
- Ongelman ratkaisuna on
  - siirtää yleisen aikataulun tekoa projektin alusta muutaman iteraation päähän ja
  - tehdä yksityiskohtaista aikataulusuunnittelua korkeintaan muutaman iteraation verran eteenpäin

7. Ketterä ohjelmistokehitys

- *Ketterä ohjelmistokehitys* (agile development) ja *ketterät prosessimallit* (agile process models) ovat IID:n osajoukko. Ne sisältävät IID:n periaatteita, kuten aikaikkunat ja viivästetyn projektisuunnittelun, mutta niiden lisäksi ne korostavat *muutosten hyväksymistä* (embrace change).
- Ketterässä ohjelmistokehityksessä oleellista on *ohjautuvuus* (maneuverability). Se tarkoittaa, että ketterät projektit ovat valmiita vaihtamaan suuntaa aina kun tarve vaatii.

Ketterien prosessien muodollisuus

- *Muodollisuus* (ceremony) määrittelee, miten paljon prosessissa on dokumentaatiota, matemaattista formalismia, tarkastuksia yms.
- Ketteriin prosesseihin yhdistetään helposti epämuodollisuus, mutta tämä on myytti. Ketterät prosessit eivät ole sen epämuodollisempia kuin suunnitelmakeskeiset prosessit.
  - Esimerkiksi Scrumissa ei oteta kantaa siihen, miten muodollinen projektin tulee olla. Muodollisuuden aste jätetään projektin päätettäväksi.
  - XP:ssa dokumentaatio pidetään minimissä, mutta vastaavasti XP sisältää paljon muodollisia tekniikoita, kuten pariohjelmoinnin ja testivetoisen ohjelmistokehityksen.
  - Amblerin kehittämä menetelmäjoukko *Agile Modeling* (www.agilemodeling.com) lisää mallinnuksen ketteriin prosesseihin. Menetelmät sopivat käytännössä mihin tahansa ketterään prosessimalliin.

Agile Alliance

- Ketterille prosessimalleille ei ole yhtä ja ainoaa määritelmää. Parhaiten ketteriä prosessimalleja kuvaavat Agile Alliancen vuonna 2001 esittelemät *ketterien prosessimallien manifesti* (Agile Manifesto) ja *ketterät periaatteet* (agile principles).
- Agile Alliance voi edelleen hyvin. Organisaatiossa on kehitelty ketterien prosessimallien yleisiä periaatteita ja tehty ketteriä prosesseja tutuksi.
  - Agile Alliance löytyy osoitteesta http://www.agilealliance.com
- Ketterille prosessimalleille on mahdollisesti tulossa standardi.

Agile Manifesto

Koska ketterien prosessien manifesti menettäisi liikaa käännöksessä, se on tässä alkuperäisessä muodossa (Agile Alliance, 2001):

Agile Manifesto

Individuals and interactions	over processes and tools
Working software	over comprehensive documentation
Customer collaboration	over contract negotiation
Responding to change	over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Ketterät periaatteet 1

Myös Agile Alliancen ketterät periaatteet on paras antaa englanniksi:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Ketterät periaatteet 2

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinetely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Ketterien periaatteiden tulkintaa

Ketteristä periaatteista saadaan hyvä kuva siitä, mitä ketteriin prosesseihin sisältyy:

- projektit ovat joustavia (2, 3)
- toimiva kommunikaatio on erityisen tärkeää (4, 6, 11, 12)
- asiakastyytyväisyys on tärkein laadun mittari (1, 2, 3, 7)
- työntekijöiden tyytyväisyys on tärkein laadun tae (4, 5, 6, 8)
- kehitystyö on systemaattista (1, 3, 9, 10, 11, 12)

Ketterä projektinhallinta

- Ketterät prosessit korostavat tiimityöskentelyä. Tiimi toimii yhtenä tasa-arvoisena yksikkönä.
  - Perinteiset projektipäällikön tehtävät kuuluvat ketterissä prosesseissa koko tiimille. Koko tiimi esimerkiksi vastaa aikataulusta, resurssi- ja kustannusarvioista sekä riskienhallinnasta.
- Tiimeillä on toki johtaja, mutta ei perinteisessä mielessä.
  - Ketterän projektitiimin johtajan tehtäviä:
    - tiimiläisten valmennus
    - työtä hidastavien ongelmien (”töyssyjen”) tasoitus
    - resurssien hankkiminen
    - ketterien periaatteiden ylläpito projektissa
  - Hyvässä ketterässä projektissa johtajan ei juurikaan tarvitse käskeä tiimiläisiä.

Minimalismin periaate

- Ketterä periaate 10 yksinkertaisuudesta (”Simplicity is essential”) on helppo ymmärtää väärin. Sitä tulkitaan usein siten, että ketterät projektit ovat holtittomia ja hallitsemattomia. Tämä ei pidä paikkaansa.
- Periaate 10 on minimalismin periaate. Sen mukaan asiat pitää tehdä niin yksinkertaisesti kuin mahdollista, *mutta ei yhtään yksinkertaisemmin.*
  - Periaate 10 koskee koko projektia, ei vain koodausta. Mikä on yksinkertaisin toimiva tapa kerätä projektin vaatimuksia? Mikä on yksinkertaisin toimiva tapa tehdä jatkuvaa testausta? Jos yksinkertaisin tapa on rakentaa testausta varten työkalu, niin sitten sellainen rakennetaan.

Ketterät prosessit ja systemaattisuus

- Ketterät prosessit eivät ole holtittomia tai hallitsemattomia. Niissä on paljon rakennetta, ohjausta ja hallintaa. Ne ovat systemaattisia.
- Ero suunnitelmakeskeisiin prosesseihin on siinä, että ketterissä prosesseissa lähtökohtana on *muutos*.
- Muutokseen varautuminen pakottaa joustaviin ratkaisuihin projektinhallinnassa, aikataulutuksessa, resurssien arvioinnissa ja dokumentaatiossa.
- Joustavuus saavutetaan sillä, että ketterien menetelmien periaatteet ovat vahvasti julkaisu- ja laatukeskeisiä. Toimiva laadukas tuote on kattavaa dokumentaatiota tärkeämpi ketterän projektin tuotos.

Säännöt vastaan periaatteet

- Ehkä tavallisin virhe siirryttäessä ketterään ohjelmistokehitykseen on määritellä suuri määrä prosessissa seurattavia sääntöjä
  - tuloksena saadaan jäykkä prosessi, joka ei ole joustava eikä työntekijäystävällinen
  - ajan myötä kommunikaatio saattaa jäädä sääntöjen tieltä vähemmälle
  - kommunikoinnin kärsiessä saadut julkaisut eivät vastaa asiakkaan tarpeita, jolloin asiakastyytyväisyys kärsii
  - lopulta toimimattomia sääntöjä noudatetaan epäsäännöllisesti, jolloin kehitystyön systemaattisuus kärsii
- Sääntöjen sijaan ketterissä prosesseissa suositaan ”periaatteita”. Ne määrittelevät raamit, joiden puitteissa projektit voivat toimia. Jos jotkut periaatteista eivät sovi tiimille, niistä pitää luopua.

Ketterien prosessien vaikutus ohjelmistokehitykseen

- Ketterät prosessimallit eivät luoneet iteraatioita, lyhyitä syklejä tai usean julkaisun esittelyä. Kaikki nämä keksittiin jo 1960-luvulla ennen vesiputousmallia.
- Royce ”esitteli” vesiputousmallin 1970, minkä jälkeen lineaarinen suunnitelmakeskeinen ohjelmistokehitys alkoi saada erityisesti yritysten johtotason suosiota.
  - Royce ei itse asiassa esitellyt lineaarista mallia. Artikkelissaan hän ehdotti mallia, jossa kehitystyö tehdään kahdessa iteraatiossa.
  - Ilmeisesti väärinkäsityksen johdosta Roycen artikkelia pidettiin todisteena johtajien suosimasta lineaarisesta mallista.
- Sen sijaan pariohjelmointi, testivetoinen ohjelmistokehitys ja jatkuva integrointi ovat melko tuoreita ja nimenomaan ketterään ohjelmistokehitykseen liittyviä tekniikoita.
- Tekniikoita tärkeämpää ketterissä prosesseissa on filosofia. Vaikka tekniikat kehitettiin aikaisemmin, vasta ketterissä prosessimalleissa ne yhdistettiin yhteisen filosofian alle.

8. Scrum

- *Scrum* on IID-pohjainen prosessimalli, jonka painopiste on projektinhallinnan arvoissa ja tekniikoissa.
  - Nimi ”Scrum” tulee rugbysta ja tarkoittaa pelin aloitusryhmitystä.
- Scrum on tällä hetkellä yleisin ohjelmistoteollisuudessa käytetty ketterä prosessimalli. Scrumin suosio perustuu sen yksinkertaisuuteen ja eleganttiin tapaan suhtautua projektinhallintaan.
- Scrum on joustava prosessimalli, jossa kiinnitetään vain joitain projektinhallintaan liittyviä tekniikoita.
- Scrum-projektissa voidaan käyttää hyväksi muiden ketterien prosessimallien tekniikoita. Esimerkiksi useimmat XP:n tekniikat sopivat Scrum-projektiin.

Scrumin elinkaari

- Scrumin elinkaari rakentuu neljästä vaiheesta:
  - valmistelu (planning)
  - järjestäminen (staging)
  - kehitystyö (development)
  - julkaisu (release)
- Valmistelu ja järjestäminen ovat Scrum-projektin esivaiheita.
- Kehitystyö tehdään iteraatioina
  - alun perin iteraation pituus on ollut tasan 30 päivää, mutta nykyisin Scrum suosii 15-30 päivän mittaisia iteraatioita
- Julkaisussa asiakkaalle toimitetaan ohjelmiston toimiva versio.

Scrum: Valmistelu 1

- Valmisteluvaihe vastaa lineaaristen prosessien esi- tai kelpoisuusselvitystä (feasibility study). Siinä selvitetään tuotettavan ohjelmiston ja projektin taustoja ja varmennetaan, että ohjelmisto kannattaa toteuttaa.
- Tarkoitus:
  - asetetaan tavoite
  - selvitetään odotukset
  - varmistetaan rahoitus

Scrum: Valmistelu 2

- Tehtävät:
  - kirjataan tavoite
  - kirjataan budjetti
  - alustetaan *työlista* (backlog)
    - työlista sisältää tuotteelle tähän mennessä löydetyt *vaatimukset*
    - työlistaa päivitetään sitä mukaa kun uusia vaatimuksia ja niihin liittyviä tehtäviä syntyy ja vanhat vaatimukset muuttuvat
    - työlistan vaatimuksia ja tehtäviä *priorisoidaan* projektin kuluessa
  - tarvittaessa tehdään ongelmakenttää selventävä ohjelmistosuunnitelma ja/tai prototyyppi

Scrum: Järjestäminen

- Järjestäminen vastaa perinteistä vaatimusten kartutusta ja analyysia. Siinä selvitetään alustavat tuotteen vaatimukset ja kuvataan ongelmakenttä.
  - Sekä vaatimukset että ongelmakenttä päivittyvät projektin aikana. Jokaisen iteraation jälkeen tuotteesta on syntynyt aiempaa selkeämpi käsitys.
- Tarkoitus:
  - tunnistetaan tuotteen vaatimuksia ja priorisoidaan niitä riittävästi ensimmäistä iteraatiota varten
- Tehtävät:
  - vaatimusten kartutus (elicitation)
  - alustava ohjelmistosuunnitelma ja/tai prototyyppi

Scrum: Kehitystyö

- Kehitystyössä ohjelmistoa toteutetaan iteratiivisesti IID:n periaatteiden mukaisesti
- Tarkoitus:
  - toteutetaan julkaisukelpoinen ohjelmisto *pyrähdyksinä* (sprint)
    - yksi pyrähdys tarkoittaa yhtä aikaikkuna-iteraatiota
- Tehtävät:
  - pyrähdyksen suunnittelu(kokous)
  - pyrähdyksessä toteutettavien työlistan tehtävien valinta, priorisointi ja kirjaaminen *pyrähdyksen työlistaan* (sprint backlog)
  - jäljellä olevan työmäärän arviointi
  - päivittäiset Scrum-kokoukset
  - *Scrum-katselmointi* (Scrum review) pyrähdyksen lopussa: katsotaan (demotaan) tuotteen omistajalle, mitä tiimi on saanut pyrähdyksessä aikaan
  - mahdollinen tiimin itsearviointi (retrospective), jossa otetaan oppia pyrähdyksestä seuraavaa pyrähdystä varten

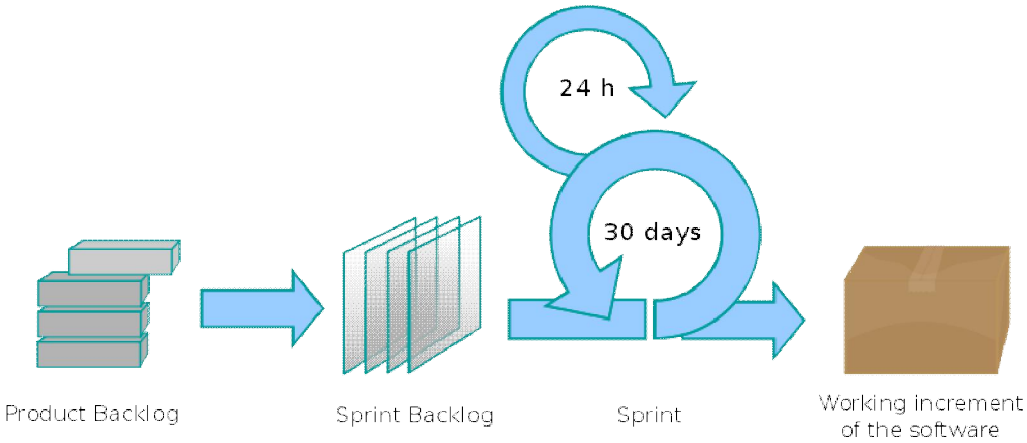
Scrum: Julkaisu

- Julkaisussa loppukäyttäjälle tarkoitettu ohjelmisto annetaan tuotantokäyttöön. Projektin aikana voi olla monta julkaisua.
- Tarkoitus:
  - julkaistaan valmis ohjelmisto
- Tehtävät:
  - dokumentointi
  - koulutus
  - myynti
  - jne.

Scrum-kehitysprosessi lyhyesti

- Kunkin pyrähdyksen alussa työlistalta valitaan pyrähdyksen aikana toteutettavat tehtävät pyrähdyksen työlistalle.
  - Pyrähdyksen työlistalle ei lisätä uusia tehtäviä pyrähdyksen aikana.
  - Suositus: tehtävät on suunniteltava niin pieniksi, että kukin voidaan toteuttaa kahdessa työpäivässä.
- Jokainen työpäivä aloitetaan *Scrum-kokouksella* (Scrum meeting). Kokous pidetään aina samassa paikassa samaan aikaan.
  - Kokouksessa jokainen tiimin jäsen vastaa seuraaviin kysymyksiin:
    - Mitä olet tehnyt edellisen Scrum-kokouksen jälkeen?
    - Mitä aiot tehdä seuraavaan Scrum-kokoukseen mennessä?
    - Mitä esteitä olet kohdannut työssäsi?
  - Lisäksi oppikirjassa (Larman, 2004) suositellaan kahta lisäkysymystä:
    - Puuttuuko työlistasta tehtäviä? (Ei uusia vaatimuksia vaan vanhojen tarkennuksia.)
    - Oletko oppinut jotain sellaista, josta on hyötyä muille tiimiläisille?
- Pyrähdyksen päätteeksi tarkistetaan Scrum-katselmoinnissa, että tuotettu ohjelmistoversio täyttää sille asetetut vaatimukset.

Scrum-prosessimallin kaaviokuva



Kuva on lähteestä wikimedia.org.
Kuva on julkaistu GFDL-lisenssillä
(GNU Free Documentation License)

Scrumin sidosryhmät 1

Scrum-projektilla on neljä sidosryhmää:

1. *Tuotteen omistaja* (product owner)
   - Tuotteen omistaja edustaa projektin asiakasta:
     - vastaa siitä, että projektin työlista on ajan tasalla
     - valitsee seuraavassa pyrähdyksessä toteutettavat tehtävät
     - arvioi yhdessä muiden sidosryhmien kanssa jokaisen pyrähdyksen päättyessä tehdyn tuotoksen
   - Tuotteella voi olla monta yhteistyössä toimivaa omistajaa.
2. *Scrum-tiimi* (Scrum team)
   - Scrum-tiimi vastaa kuhunkin pyrähdykseen valittujen tehtävien toteutuksesta.
   - Tiimiläisille ei ole määritelty tämän tarkempia tehtäviä.
   - Tiimissä ei saisi olla enempää kuin seitsemän jäsentä. Tätä isommat projektit on organisoitava useiksi tiimeiksi.
3. *Kanat* (chickens)
   - Kaikkia muita projektin sidosryhmiä kutsutaan kanoiksi, myös esimerkiksi johtoportaan edustajia. Kanat saavat tarkkailla projektia mutta ne eivät voi vaikuttaa pyrähdysten sisältöön.

Scrumin sidosryhmät 2

4. *Scrum-mestari* (Scrum master)
   - Scrum-mestari vastaa projektin hallinnasta. Hän ei ole varsinainen projektipäällikkö, vaan pikemminkin projektin valmentaja.
   - Scrum-mestari on Scrum-tiimissä myös kehittäjänä, eikä siten projektissa pelkkänä hallintohenkilönä.
   - Scrum-mestarin tehtävät:
     - varmistaa, että projektin tavoitteet säilyvät projektin ajan
     - varmentaa, että projektissa seurataan Scrumin periaatteita
     - toimia Scrum-tiimin ja hallinnon yhteyshenkilönä
     - toimia Scrum-tiimiläisten valmentajana
     - tasoittaa ja poistaa esteitä
     - vastata Scrum-kokouksista
     - vastata Scrum-katselmoinneista (demoista)

Scrumin käytännöt 1

- Tärkeimmät Scrumin käytännöt:
  - Pyrähdykset
    - Alun perin 30 päivää, nykyisin 2-6 viikkoa. Pyrähdyksen aikana Scrum-tiimillä on vapaus tehdä pyrähdyksen työlistalla olevia tehtäviä ilman keskeytyksiä.
  - Itseorganisoituvat tiimit
    - Pyrähdyksen aikana asiakas, Scrum-mestari, managerit ja muut sidosryhmät eivät määrää tiimiä toimimaan kehitystyössä tietyllä tavalla. Kaikki päätökset tehdään pyrähdysten välillä.
  - Scrum-kokoukset
    - Jokainen työpäivä alkaa samaan aikaan ja samassa paikassa Scrum-kokouksella.
  - Rauhoitetut iteraatiot
    - Pyrähdyksen aikana sen työlistalle ei lisätä uusia tehtäviä. Jos jokin tehtävä on aivan pakko lisätä pyrähdyksen työlistalle, on jokin toinen tehtävä samalla poistettava listalta.

Scrumin käytännöt 2

- Päätökset tunnissa
  - Scrum-mestari pyrkii ratkaisemaan päätöstä vaativat kysymykset mieluimmin välittömästi tai viimeistään tunnin sisällä.
- Esteet pois päivässä
  - Scrum-kokouksissa esille tulevat ongelmat pyritään korjaamaan ennen seuraavaa Scrum-kokousta.
- Yhteinen työtila
  - Scrum-tiimi työskentelee yhteisessä projektihuoneessa. Yksityiset työtilat ovat muita tehtäviä varten.
- Päivittäinen integraatio
  - Kaikki varmennettu koodi integroidaan ja regressiotestataan ainakin kerran päivässä.

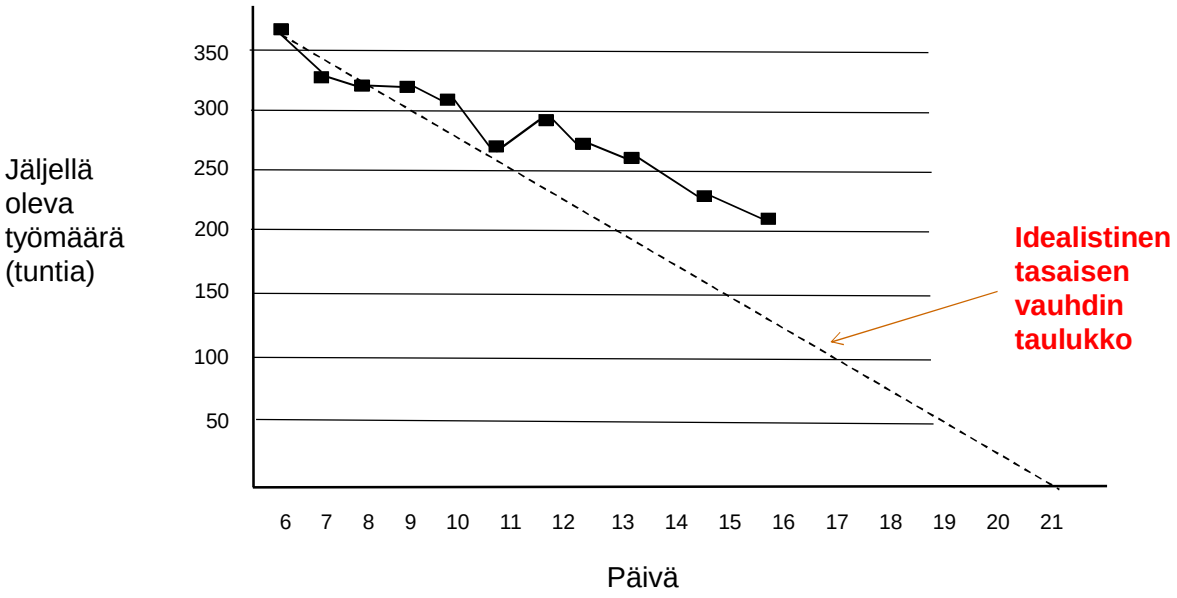
Pyrähdyksen työlista

- Pyrähdyksen työlista sisältää pyrähdyksen kannalta oleellista tietoa siinä tehtävistä töistä. Listalla on ainakin
  - tehtävän kuvaus: mitä tehdään
  - tehtävän esittäjä
  - tehtävästä vastuussa oleva tiimin jäsen
  - tehtävän status: aloittamaton, aloitettu, valmis
  - kesken olevan tehtävän vaatiman työajan arvio
- Jäljellä olevan työajan arvio on mielenkiintoinen. Puhtaassa Scrumissa ei arvioida tehtävien kestoja tai tehtävien välisiä riippuvuuksia. Riippuvuusverkkoa ja kriittistä polkua ei tarvita.
- Kun kaikki pyrähdyksen työlistan jäljellä olevan työajan arviot yhdistetään, saadaan jäljellä olevaa työmäärää kuvaava kaavio (*burndown chart*, *sprint backlog graph*). Se kertoo, paljonko pyrähdyksen kokonaistyömäärästä arvioidaan olevan jäljellä.

Työlistan tehtävät

- Työlistoissa voi olla monentyyppisiä tehtäviä, esimerkiksi:
  - asiakkaan (tuotteen omistajan) esittämiä vaatimuksia
  - tuotteeseen toteutettavien toimintojen kuvauksia
  - korjattavien virheiden kuvauksia (virheraportteja)
  - selvittämättömiä teknisiä ongelmia
- Vaatimukset ja toiminnot voidaan kuvata esimerkiksi käyttötapauksina (use case) tai käyttäjätarinoina (user story).
- Käyttäjätarina: tekstuaalinen, määrämuotoinen kuvaus käyttäjälle hyödyllisestä toiminnallisuudesta
  - ”<tietyssä roolissa> haluan <sitä sun tätä> <saavuttaakseni sen ja sen hyödyn>”
  - esim. ”sihteerinä haluan hakea ohjelmistosta henkilöiden puhelinnumeroita voidakseni soittaa heille ja ilmoittaa kokousten ajankohdat”
- Tehtävään kuluvan työmäärän arviointi: tiimin kokemukseen perustuvana yhteistyönä, esimerkiksi äänestämällä tai pelaamalla ”suunnittelupokeria” (planning poker).

Sprint backlog graph



Scrumin luonne

- Scrum on selkeästi hallinnollinen prosessimalli
  - sen käytännöissä otetaan kantaa siihen, miten projektiryhmä organisoituu ja toimii yhteistyössä
  - sen sijaan siinä otetaan hyvin vähän kantaa siihen, mitä tekniikoita projektiryhmä käyttää kehitystyössä
- Koska Scrum ottaa kantaa erityisesti hallintoon, se sopii hyvin erityyppisten projektien prosessimalliksi. Se sopii myös hyvin yhteen muiden prosessimallien käytäntöjen kanssa.

Scrumin huonoja puolia

- Scrum ei ota kantaa käytettäviin ohjelmistotuotannon menetelmiin, tekniikoihin tai työkaluihin, joten siitä ei saa tukea niiden valinnalle
- Scrum ei ota kantaa projektissa tuotettavaan dokumentaatioon, jolloin hyödyllisiä dokumentteja saatetaan jättää tekemättä
- Scrum edellyttää aktiivista tuotteen omistajaa, mutta asiakkaalta ei sellaista välttämättä saada
- Scrum-projekti nojaa tiimin yhteistyöhön ja ammattitaitoon, joten siihen ei voi ottaa ketä tahansa

9. Extreme Programming

- *Extreme Programming* (*XP*) toi ketterät menetelmät laajempaan tietoisuuteen 1990-luvun loppupuolella. Se on edelleen tunnettu ketterä prosessimalli, vaikka Scrum on vienyt siltä suosiota teollisuudessa.
- XP:n painopiste on kehitystyössä, kun Scrumin painopiste on projektinhallinnassa. Prosessimallit eivät ole ristiriidassa keskenään.
- XP:ssa korostetaan
  - yhteistyötä
  - tiheää ja aikaista julkaisutahtia
  - laadukkaita kehitystyön menetelmiä
- XP perustuu neljälle arvolle:
  - kommunikaatio
  - yksinkertaisuus
  - palaute
  - rohkeus

XP:n elinkaari

- XP:n elinkaari rakentuu viidestä vaiheesta:
  - kartoitus (exploration)
  - suunnittelu (planning)
  - iteraatiot ja ensimmäinen julkaisu (iterations to first release)
  - tuotteistus (productization, productionizing)
  - ylläpito (maintenance)
- Kartoitus ja suunnittelu ovat XP-projektin esivaiheita
- Kehitystyö tehdään IID-periaatteiden mukaan 1-3 viikon iteraatioissa
- Tuotteistus ja ylläpito ovat ohjelmistokehitysprojektin ulkopuolella, mutta kuuluvat XP-prosessiin

XP:n askeleet 1-2

1. XP-projekti voi alkaa kartoitusvaiheella. Siinä ohjelmiston alustavia ominaisuuksia kirjataan *tarinakorteille* (story cards) ja annetaan niille karkeita kokoarvioita.
2. *Julkaisun suunnittelupelissä* (release planning game) asiakkaat ja kehittäjät täydentävät tarinakortteja ja arvioita ja päättävät, mitä seuraavaan julkaisuun tulee.

XP:n askeleet 3-5

3. *Iteraation suunnittelupelissä* (iteration planning game) asiakkaat valitsevat iteraatioon toteutettavat tarinat. Ohjelmistokehittäjät jakavat tarinat pieniksi arvioitavissa oleviksi tehtäviksi. Jos tehtäviä on liikaa yhtä iteraatiota kohti, niiden määrää karsitaan.
4. Kehittäjät toteuttavat valitut tarinat aikaikkuna-periaatteella. Asiakkaat ja kehittäjät käyvät yhteistyössä läpi testejä ja vaatimusten yksityiskohtia.
5. Jos seuraavaa julkaisua varten ei ole saatu kasaan tarpeeksi toteutettuja tarinoita, palataan askeleeseen 3.

XP-prosessimallin kaaviokuva

Kuva on lähteestä wikimedia.org.
Kuva on julkaistu GFDL-lisenssillä
(GNU Free Documentation License)

XP-prosessimallin kaaviokuvan selitystä

- Releaseplan
  - tehdään julkaisun tarinakortit
- Iteration plan
  - valitaan tarinakorteista seuraavassa iteraatiossa toteutettavat tehtävät
- Acceptance Test
  - suunnitellaan iteraation (asiakasvetoisia) hyväksymistestejä
- Stand up meeting
  - pidetään päivittäinen ryhmäpalaveri Scrumin tapaan
- Pair Negotiation
  - muodostetaan seuraavat *pariohjelmoinnin* parit
- Unit Test
  - kirjoitetaan yksikkötestit *kaikelle* (iteraatiossa kirjoitettavalle) koodille
- Pair Programming
  - tehdään *kaikki* koodaus pariohjelmointina (toinen ohjelmoi, toinen katselmoi ja suunnittelee uusia yksikkötestejä)

XP:n sidosryhmät 1

XP-projektilla on kuusi sidosryhmää:

1. *Asiakas* (customer)
   - tarinakorttien kirjoittaminen
   - vastuu hyväksymistesteistä
   - seuraavan julkaisun tai iteraation tarinakorttien valinta
2. *Koodaaja* (programmer)
   - testien kirjoittaminen
   - suunnittelu
   - ohjelmointi
   - refaktorointi (koodin muokkaus arkkitehtonisesti paremmaksi)
   - tarinoiden jako tehtäviksi ja tehtävien koon arviointi
3. *Testaaja* (tester)
   - asiakkaan avustaminen hyväksymistestien suunnittelussa ja kirjoittamisessa

XP:n sidosryhmät 2

4. *Valmentaja* (coach)
   - prosessin valvonta
   - prosessin mukauttaminen
   - väliintulo ongelmatilanteissa
   - koulutus
5. *Seuraaja* (tracker)
   - mittaustietojen keruu
   - edistymisen raportointi
   - palautteen anto huonoista mittausarvioista
6. *Konsultti* (consultant)
   - tekninen konsultaatio
   - XP-valmennus

XP:n käytännöt

Toisin kuin Scrumissa, XP:ssa on melko paljon toisiinsa sidoksissa olevia ydinkäytäntöjä:

- Suunnittelupeli
- Lyhyet jatkuvat julkaisut
- Järjestelmämetaforat
- Yksinkertainen suunnittelu
- Testaus
- Jatkuva refaktorointi
- Pariohjelmointi
- Koodin yhteisomistajuus
- Jatkuva integrointi
- Kestävä kehitystahti
- Tiimi samassa tilassa
- Koodausstandardit

Ydinkäytäntöjen lisäksi on joukko ”avustavia käytäntöjä”

XP:n käytäntöjen luonne

- XP:n käytännöt ovat vahvasti sidoksissa toisiinsa. Siksi on riskialtista muokata prosessia jättämällä siitä pois keskeisiä käytäntöjä.
  - Esimerkiksi lyhyet jatkuvat julkaisut tarkoittavat kevennettyä vaatimusmäärittelyä, mikä taas vaatii asiakkaiden jatkuvaa läsnäoloa ja työskentelyä yhteisessä tilassa.
- Muualla hyväksi havaitut käytännöt on viety äärimmilleen, mistä tulee XP:n termi ”Extreme”.
  - Testaus on hyvä käytäntö, joten tehdään kaikelle koodille yksikkötestit ja kaikille ominaisuuksille hyväksymistestit.
  - Koodikatselmoinnit ovat sitä parempia, mitä lähempänä koodin kirjoittamista ne pidetään, joten katselmoidaan koodia tosiajassa tekemällä pariohjelmointia.
  - Toimiva kommunikaatio on hyvä käytäntö, joten käytetään yhteistä työtilaa, tehdään pariohjelmointia ja otetaan asiakkaat mukaan suunnitteluun, ohjaukseen ja arviointiin.

XP:n vaatimusmäärittelyn käytännöt

- Suunnittelupelit
  - julkaisun ja iteraation suunnittelupelit
- Läsnäoleva asiakas
  - asiakkaan edustajat ovat kokopäiväisesti tiimin kanssa
  - asiakkaan edustajia voi olla useista eri sidosryhmistä
  - asiakas vastaa hyväksymistesteistä
- Hyväksymistestaus
  - jokaiselle ominaisuudelle täytyy olla automatisoidut asiakkaan kirjoittamat hyväksymistestit
  - kaikki testit ovat suoritettavissa binäärisellä hyväksytty/hylätty-tuloksella
  - yksittäisten testiajojen läpimenon tarkistamiseen ei tarvita ihmisiä, vaan ne ovat automatisoituja

XP:n suunnittelun käytännöt

- Järjestelmämetaforat
  - koko järjestelmän tai sen osan kuvaus helposti muistettavana rinnastuksena (eli metaforana)
  - XP:n versio arkkitehtuurisuunnitelmasta
- Yksinkertainen suunnittelu
  - suunnitellaan tätä hetkeä varten
  - ei spekuloida tulevien iteraatioiden vaatimilla piirteillä
  - yksinkertaiset luokat ja metodit
  - ei leikkaa-liimaa-koodausta
- Jatkuva refaktorointi
  - jatkuva prosessi yksinkertaistaa koodia ja suunnitteluratkaisuja, jolloin ylläpidettävyys paranee
  - pieniä muutoksia kerrallaan
  - mahdollisimman paljon valmiiden (suunnittelu)mallien ja refaktorointityökalujen käyttöä

XP:n toteutuksen käytännöt

- Koodausstandardit
  - kaikki tiimiläiset noudattavat samaa koodaustyyliä
  - koodista ei pidä voida päätellä sen kirjoittajaa
- Pariohjelmointi
  - kaikki koodi tehdään parityönä: toinen ohjelmoi ja toinen katselmoi kirjoitettavaa koodia
  - koodausvuoro vaihtuu säännöllisesti
  - parit vaihtuvat säännöllisesti
  - kahta kokematonta ohjelmoijaa ei laiteta pariksi
- Koodin yhteisomistajuus
  - kaikki ovat vastuussa koko koodista
  - kuka tahansa voi muokata mitä tahansa osaa koodista
  - virheen löytäjällä on vastuu sen korjaamisesta

XP:n testauksen ja verifioinnin käytännöt

- Hyväksymistestaus
- Testivetoinen kehitystyö (Test-Driven Development, TDI) (yksikkötestauksessa)
  - yksikkötestit kirjoitetaan ennen niitä vastaavan ominaisuuden ohjelmointia
  - kaikkia yksikkötestejä suoritetaan automaattisesti tauotta
  - yksikkötestit vastaavat ominaisuuden spesifikaatiota
- Läsnäoleva asiakas

XP:n projektinhallinnan käytännöt

- Suunnittelupelit
- Lyhyet julkaisut
  - *evoluutiolähtöinen kehitystyö*: väärät päätökset ovat hyväksyttäviä
  - varautuminen muutokseen
- Kestävä kehitystahti
  - ei jatkuvia ylitöitä
  - tiimin ja asiakkaiden on voitava jaksaa samaa työtahtia periaatteessa rajattomasti
- Päivittäiset ryhmäpalaverit
  - tiimi käy joka aamu yhdessä läpi edellisenä päivänä kohdatut ongelmat ja niihin löydetyt ratkaisut
  - hyvin samanlainen kuin päivittäinen Scrum-kokous

XP:n versionhallinnan käytännöt

- Jatkuva integrointi
  - kaikki valmiiksi katsottu koodi integroidaan noin 15 minuutin välein erillisellä koneella, samalla suorittaen kaikki olemassa olevat yksikkötestit
  - jatkuva integrointi on käynnissä 24 tuntia vuorokaudessa jokaisena viikonpäivänä
- Yhteinen työtila
  - XP-projektit läpiviedään yhteisessä työtilassa, ei erillisissä toimistoissa
  - työtilan keskellä ovat pariohjelmointipöydät
  - seinät on varattu tauluille ja postereille yms.
  - tiimi ja asiakkaan edustajat työskentelevät samassa tilassa
- Suunnittelupelit

XP:n luonne

- XP on selkeästi ohjelmistokehittäjän prosessimalli
  - sen käytännöissä otetaan kantaa siihen, miten ohjelmistokehitys tehdään mahdollisimman tehokkaasti
  - siinä olevat hallinnolliset periaatteet ovat melko yleisiä ja erikoistettavissa eri projekteihin
- Scrumiin verrattuna XP on teknisempi, vaikeammin opittava ja kehitystyötä enemmän määrittelevä prosessimalli. Ehkä sen vuoksi XP on hävinnyt suosiossa Scrumille.

XP:n huonoja puolia

- XP edellyttää aktiivista asiakasta, mutta sellaista ei välttämättä saada projektiin
- XP ei tue ohjelmiston kokonaisarkkitehtuuria
- Useimmat ohjelmistokehittäjät eivät pidä pariohjelmoinnista
- XP ei ota kantaa projektissa tuotettavaan dokumentaatioon, jolloin hyödyllisiä dokumentteja saatetaan jättää tekemättä

XP ja Scrum

- XP:n käytännöt keskittyvät pääasiassa kehitystyöhön. Koska Scrumin käytännöt puolestaan keskittyvät pääasiassa projektinhallintaan, XP:n tekniikat ovat pitkälti yhteensopivat Scrumin kanssa.
- Vaikka Scrum-projektissa ei tarvitse seurata muita kuin Scrumin omia periaatteita, sopivien XP:n periaatteiden lisääminen tehostaa projektityöskentelyä.

Scrumiin sopivia XP:n käytäntöjä 1

Ainakin seuraavat XP:n käytännöt sopivat hyvin Scrum-projektiin:

- Automatisoitu testaus
  - kaikki testit on voitava suorittaa automaattisesti
  - kaikista testeistä on saatava yksiselitteinen läpi/kaatui-tulos (pass/fail)
  - hyväksymistestit kirjoitetaan yhdessä asiakkaan (tuotteen omistajan) kanssa
- Testivetoinen kehitystyö
  - Yksikkötestauksessa testit kirjoitetaan ennen koodia. Yksikkötestit vastaavat sekä testitapauksen kuvausta että testattavan ominaisuuden spesifikaatiota.
- Pariohjelmointi
  - Kaikki koodi kirjoitetaan pariohjelmointina yhden koneen ääressä. Pariohjelmoinnissa toinen kirjoittaa koodia ja toinen tekee koodille jatkuvaa laadunvarmistusta.

Scrumiin sopivia XP:n käytäntöjä 2

- Jatkuva integrointi
  - kaikki hyväksytty koodi integroidaan ja testataan jatkuvasti erillisessä integrointijärjestelmässä
  - integrointijärjestelmä toimii 24/7-silmukassa, jossa käännetään koodi sekä suoritetaan kaikki yksikkötestit ja kaikki/useimmat hyväksymistestit
- Yhteinen koodin omistajuus
  - kuka tahansa tiimin jäsen on velvollinen korjaamaan löytämänsä koodivirheen
  - tiimiläisten on seurattava yhtenäistä koodausstandardia
- Yksinkertainen suunnittelu
  - koodi suunnitellaan ja toteutetaan nykyistä julkaisua varten
- Säännöllinen refaktorointi
  - koodia yksinkertaistetaan ja siivotaan säännöllisesti, tavoitteena minimaalinen ja helposti ymmärrettävä koodikanta

10. Lean

M. Poppendieck, T. Poppendieck: *Implementing Lean Software Development – From Concept to Cash*. Addison-Wesley, 2007.

- Viime vuosina käyttöön otettu seuraaja ketterille menetelmille
- Tavoitteena työn tehostaminen virtaviivaistamalla ja pelkistämällä työprosessia
- Perustuu Toyotan voittokulkuun johtaneisiin periaatteisiin ja käytäntöihin autoteollisuudessa

Lean-historiaa

- 1940- ja 1950-luvut: Toyota kehitti Japanissa nopeaa ”Just-in-Time”-tuotelinjaa kyetäkseen kilpailemaan tehokkuudessa amerikkalaisten autonvalmistajien kanssa
- 1962: *Toyota Production System* (TPS) otettiin käyttöön
- 1980-luku: TPS:n pohjalta kehitettiin ja määriteltiin yleiset *Lean*-tuotantoperiaatteet
- 1990-luku: TPS yleistettiin *Toyota Product Development System* -tuotekehitysmenetelmäksi
- 2000-luku: Lean-periaatteet otettiin käyttöön ohjelmistotuotannossa
- Kantateos: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*. Addison-Wesley, 2003.

Ohjelmistotuotannon Lean-periaatteet 1

1. Jätteen poistaminen (eliminate waste)
   - jäte: kaikki se, mikä ei tuota arvoa asiakkaalle
   - tavoitteena poistaa ohjelmistokehityksestä kaikki turha työ
2. Jatkuva laadunvarmistus (build quality in)
   - virheiden välttäminen koko kehitysprosessin ajan, ei vasta (erillisessä) testausvaiheessa
3. Tietämyksen luominen (create knowledge)
   - ohjelmiston kehittäminen asteittain tarkentamalla
   - vaatimusten, arkkitehtuurin, koodin yms. yksityiskohtia ei kiinnitetä ennen kuin kehitystiimillä on riittävästi siihen tarvittavaa tietämystä

Ohjelmistotuotannon Lean-periaatteet 2

4. Päätöksenteon lykkääminen (defer commitment)
   - (lopullisten) päätösten tekeminen vasta viime hetkellä, tarjolla olevien vaihtoehtojen perusteella
   - tavoitteena välttää jo tehdyn työn peruuttamista
5. Nopeus (deliver fast)
   - tuoteversioiden toimittaminen asiakkaille niin usein ja nopeasti, etteivät he ehdi muuttaa mieltään
6. Tiimityö (respect people)
   - ohjelmistojen kehittäminen autonomisissa tiimeissä, joilla on huipputason johtajat
7. Kokonaisuuden optimointi (optimize the whole)
   - ei yksittäisten vaan tuotantoprosessin kaikkien vaiheiden ja niiden välisten siirtymien tehostaminen

Lean-jäte 1

1. Osittain tehty työ (partially done work)
   - ohjelmistotuotteen osa tai prosessin välituotos, jota pitää jatkuvasti päivittää
   - tyypillisiä esimerkkejä: liian aikaisin jäädytetyt vaatimukset, integroimaton ohjelmakoodi, testaamaton koodi, dokumentoimaton koodi
2. Ylimäärä (extra features)
   - sellaiset ohjelmistoon lisätyt palvelut ja viritykset, joita asiakas ei tarvitse
3. Vanhan toisto (relearning)
   - joskus aiemmin tehdyn ja sittemmin unohdetun työn mieleen palauttamista ja uudelleen tekemistä

Lean-jäte 2

4. Täysi heprea (handoffs)
   - ❖ sellaiset ohjeet tai dokumentit, joita niiden vastaanottaja ei pysty ymmärtämään
5. Poukkoilu (task switching)
   - ❖ jatkuvaa siirtymistä hommasta toiseen tai useamman homman hoitamista yhtä aikaa
6. Viivästykset (delays)
   - ❖ muuta työtä tekevien odottamista, joko antamaan neuvoja tai osallistumaan seuraavaksi vuorossa olevaan työtehtävään
7. Virheet (defects)
   - ❖ vasta prosessin myöhemmissä vaiheissa havaittavat, etsittävät ja korjattavat virheet ohjelmakoodissa

Kanban

- *Kanban*: työtehtävää kuvaava kortti
- Kanban-taulu (Kanban board): tuotelinjaa, tehtäväketjua tai tuotantoprosessia kuvaava taulu
  - sisältää kaikkia tehtäviä kuvaavat Kanban-kortit
  - kortit on sijoitettu sarakkeisiin, jotka vastaavat tuotelinjan, tehtäväketjun tai tuotantoprosessin eri vaiheita
  - kortteja siirretään sarakkeesta toiseen sitä mukaa kun työ etenee
  - korteissa kuvataan tyypillisesti tehtävä, sen tekijät ja sen tekemiseen arvioitu työmäärä tai -aika
  - sarakkeille on määritelty yläraja niihin sijoitettavissa olevien korttien määrälle (so., samassa vaiheessa yhtä aikaa olevien tehtävien määrälle)
  - esimerkiksi korttien väreillä voi kuvata niitä vastaavien tehtävien vaihetta (aloitettu, kesken, valmis)

Kanban-taulu

Lean: käyttöönotto 1

Kokonaisuuden optimointi

1. Ota lean-periaatteet käyttöön koko prosessissa mm. nimeämällä arvoketjusta vastaava tiimi
2. Ota käyttöön arvoketjun mittarit
3. Vähennä vaiheesta toiseen siirtymisestä aiheutuvaa yleisrasitetta arvoketjussa ja prosessissa

Tiimityö

4. Kouluta tiiminjohtajia
5. Siirrä vastuut ja päätösvalta organisaatiossa mahdollisimman alas (tiimeille)
6. Korosta laatua ja ammattiylpeyttä mm. poistamalla rutiinityöt ja liian tiukat takarajat sekä antamalla aikaa testaukselle

Lean: käyttöönotto 2

Nopeus

7. Työskentele pienissä erissä ja lyhyissä iteraatioissa
8. Älä käynnistä uutta työvaihetta ennen kuin edellinen on valmis
9. Älä keskity resurssien tehokkaaseen käyttöön vaan mahdollisimman nopeaan tuotantoon

Päätöksenteon lykkääminen

10. Älä jää odottelemaan täydellistä vaatimusmäärittelyä vaan tuota vaatimuksia rinnakkain kehitystyön kanssa
11. Murra ohjelmiston arkkitehtoniset riippuvuudet
12. Kehitä rinnakkain useita mahdollisia ratkaisuvaihtoehtoja äläkä kiinnitä niistä yhtäkään ennen kuin olet täysin varma

Lean: käyttöönotto 3

Tietämyksen luominen

13. Varmista, että ohjelmistoa voidaan kehittää arkkitehtonisesti itsenäisinä moduuleina, joista kullakin on vastuutiimi
14. Anna tiimeille aikaa kehittää omaa toimintaansa ja järjestä tiimien yhteisiä kehittämistilaisuuksia
15. Järjestä ongelmanratkaisun koulutusta

Jatkuva laadunvarmistus

16. Kehitä ohjelmistoa testivetoisesti ja integroi koodia jatkuvasti
17. Automatisoi rutiinitehtävät, kuten testiajot ja versionhallinta
18. Siivoa ohjelmakoodia, testitapauksia ja dokumentaatiota refaktoroimalla niitä säännöllisesti

Lean: käyttöönotto 4

Jätteen poistaminen

19. Varmista, että tiimit varmasti ymmärtävät, mitä asiakkaat haluavat, ja keskittyvät oikeiden tuotteiden kehittämiseen
20. Älä tuota mitään muuta kuin arvoa asiakkaille
21. Koodaa vähemmän ja julista mutkikkuus pannaan

Agile vs. Lean 1

- Sekä agile että lean:
  - käyttäjätarinat
  - refaktorointi
  - testivetoinen kehitys ja automatisoitu testaus
  - pariohjelmointi
  - jatkuva koodin integrointi
  - versionhallinta
  - katselmoinnit ja tarkastukset
  - lisäävä (inkrementaalinen) kehitys
  - lyhyet iteraatiot
  - sisäisten ja ulkoisten julkaisujen erottaminen toisistaan
  - asiakaslähtöinen vaatimusten priorisointi
  - aikaikkunat
  - vastuulliset ja itsenäiset tiimit
  - yhteiset työtilat

Agile vs. Lean 2

- Agile muttei lean:
  - aina läsnä oleva asiakas
  - koodausstandardit
  - koodin yhteisomistajuus
  - (arkkitehtoninen) suunnittelupeli
  - 40 tunnin työviikko
  - päivittäiset kokoukset
- Lean muttei agile:
  - moduulien arkkitehtoninen riippumattomuus
  - arvoketju ja sen hallinta
  - pääsuunnittelija (chief engineer)
  - Kanban-taulu

11. Tutkittua tietoa

T. Dybå, T. Dingsøyr: Empirical Studies of Agile Software Development : A Systematic Review. *Information and Software Technology* 50, 2008, 833-859.

J.E. Hannay, T. Dybå, E. Arisholm, D.I.K. Sjøberg: The Effectiveness of Pair Programming: A Meta-Analysis. *Information and Software Technology* 51, 2009, 1110-1122.

D.I.K. Sjøberg, A. Johnsen, J. Solberg: Quantifying the Effect of Using Kanban versus Scrum: A Case Study. *IEEE Software* 29, 5, 2012, 47-53.

M. Ikonen: Lean Thinking in Software Development: Impacts of Kanban on Projects. PhD thesis, A-2011-4, University of Helsinki, Department of Computer Science, 2011.

S. Mäkinen: Driving Software Quality and Structuring Work through Test-Driven Development. Pro gradu, HY/TKTL, 2012.

Tutkittua tietoa 1

+ XP-projektit myöhästyvät hieman vähemmän kuin ”perinteiset” projektit
+ XP-projektit ylittävät budjettinsa selvästi vähemmän kuin ”perinteiset” projektit
+ pienten XP-tiimien tuottavuus on useimmiten parempaa kuin vastaavien ”perinteisten” projektiryhmien tuottavuus
+ läsnä oleva asiakas parantaa asiakasyhteistyötä
+ XP-projektien asiakkaat ovat tyytyväisempiä kuin ”perinteisten” projektien asiakkaat
+ XP-projekteissa ohjelmistokehittäjät ovat tyytyväisempiä kuin ”perinteisissä” projekteissa
+ XP toimii parhaiten, kun ohjelmistokehittäjät ovat kokeneita ja hallitsevat sekä toimialan että kehitystyökalut

Tutkittua tietoa 2

+ ketterästi kehitettyjen ohjelmistojen laatu on parempi kuin vastaavien ”perinteisesti” kehitettyjen ohjelmistojen laatu
+ 40 tunnin työviikko vähentää ylitöiden tekemistä

+ testivetoinen kehitys vähentää virheiden määrää
+ testivetoinen kehitys parantaa testauksen koodikattavuutta
+ testivetoinen kehitys pienentää koodin mutkikkuutta (complexity)
+ testivetoinen kehitys lisää ohjelmiston ylläpidettävyyttä
+ testivetoisesti kehitetyn ohjelmiston kokonaiskustannukset ovat pienemmät kuin ”perinteisesti” kehitetyn ohjelmiston, mikäli sen elinkaari on riittävän pitkä

Tutkittua tietoa 3

+ pariohjelmointi parantaa koodin laatua yleisesti ottaen hiukan, kokemattomilla ohjelmoijilla huomattavasti
+ pariohjelmointi nopeuttaa koodin tuottamista
+ pariohjelmointi on parhaimmillaan vaikeissa ja korkeaa laatua vaativissa tehtävissä, joilla ei ole tiukkaa aikataulua
+ ketterät projektit kehittävät opiskelijoiden kommunikointi-, sitoutumis- ja yhteistyötaitoja
+ ohjelmistokehitys on Kanbanilla nopeampaa kuin Scrumilla
+ tehtävät valmistuvat Kanbanilla nopeammin kuin Scrumilla
+ työn tuottavuus on Kanbanilla suurempi kuin Scrumilla
+ Kanban visualisoi hyvin työn etenemistä
+ Kanban tukee tilannekohtaista analyysia ja muutosta

Tutkittua tietoa 4

- työmäärän arviointi menee pahasti metsään XP-projektien alussa (mutta paranee projektien edetessä)
- XP-projekteissa ei kiinnitetä tarpeeksi huomiota ohjelmiston suunnitteluun eikä arkkitehtuuriin
- XP-projektien 40 tunnin työviikko uuvuttaa kehittäjät
- XP ei sovellu suuriin projekteihin
- Scrum-projekteissa pitää kouluttaa myös asiakasta
- Scrumin aikaikkunat ovat keinotekoisia, koska työmäärät yleensä aliarvioidaan ja vastaan tulee ennakoimattomia ongelmia
- Scrumin aikaikkunat johtavat liian nopeasti kehitettyyn ja huonolaatuiseen koodiin
- Scrumin pyrähdyksissä työ jakaantuu epätasaisesti: esimerkiksi testaajilla on pyrähdyksen alussa luppoaikaa ja lopussa liian kiire
- Scrumin järjestämisvaihe (staging) koetaan jätteeksi

Tutkittua tietoa 5

- ketterästi kehitettyjen ohjelmistojen käyttöliittymä on huonompi kuin ”perinteisesti” kehitettyjen ohjelmistojen käyttöliittymä
- ketterissä tiimeissä on vaikeampi vaihtaa ohjelmistokehittäjiä kuin ”perinteisissä” projektiryhmissä
- ketterien projektin onnistuminen ei riipu pelkästään teknisistä vaan myös sosiaalisista taidoista
- aina ketterästi läsnä oleva asiakas stressaantuu ja uupuu

Tutkittua tietoa 6

- testivetoinen kehitys lisää hieman koodin kytkentää (coupling)
- testivetoisesti kehitetyssä ohjelmistossa on lähes yhtä paljon yksikkötestauskoodia kuin varsinaista toiminnallista koodia
- useimmiten testivetoinen kehitys vie enemmän aikaa kuin ”perinteinen” kehitys, lisää kokonaistyömäärää ja kasvattaa kustannuksia
- testivetoinen kehitys vähentää työn tuottavuutta

- pariohjelmointi lisää kokonaistyömäärää
- pariohjelmointi vähentää kokeneiden ohjelmoijien tuottavuutta ja laskee heidän koodinsa laatua
- pariohjelmointi ei sovellu tehtäviin, joilla on tiukka aikataulu, vähäiset resurssit ja korkeat laatuvaatimukset

Tutkittua tietoa 7

– testivetoinen kehitys on vaikeaa monille opiskelijoille

– virheitä korjataan Kanbanilla hitaammin kuin Scrumilla
– Kanban ei tarjoa apua ongelmien varsinaiseen ratkaisemiseen
– Kanban-taulu on liian rajoittunut työkalu suuriin projekteihin

± XP, Scrum, lean ja Kanban vaativat johtamista ja hyviä johtajia onnistuakseen; ilman niitä lopputuloksena on useimmiten kaaos ja sekuli