1. Johdanto

- Termi *Ohjelmistotuotanto* (Software Engineering) esiteltiin ensimmäistä kertaa 1968 pidetyssä NATO:n konferenssissa. Termi määriteltiin näin:

  The establishment and use of *sound* engineering principles in order to obtain *economically* software that is *reliable* and works *efficiently* on *real machines*.

  (P. Naur, R.Randell (eds.): Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee, 1968)

Ohjelmistotuotannon piirteitä

- Määritelmä oli aikaansa edellä, sillä se pätee erittäin hyvin edelleen. Ohjelmistotuotanto
  - on kurinalaista insinöörityötä,
  - pyrkii tarjoamaan keinot valmistaa *laadukkaita* ja *tehokkaita* ohjelmistoja mahdollisimman edullisesti ja
  - tarjoaa tehtäväjaon ja työkalut kaikkiin ohjelmistojen valmistuksen työvaiheisiin.

Ohjelmisto ja järjestelmä

- *Ohjelmisto* (software) on kokoelma yhteistyössä toimivia tietokoneohjelmia, ohjelmien käyttämiä tiedostoja ja niihin liittyviä dokumentteja.
- *(Tietokonepohjainen) järjestelmä* (computer-based system) on joukko toisiinsa liittyviä komponentteja.
  - laitteisto- ja ohjelmistokomponentteja,
  - käyttäjiä ja käyttöympäristöjä.

Osajärjestelmät ja käyttäjät

- Yleensä järjestelmä koostuu useasta osajärjestelmästä.
  - Kukin osajärjestelmä on itsenäinen kokonaisuus, joka toimii yhteistyössä muiden osajärjestelmien kanssa.
  - Osajärjestelmä rakentuu laitteisto- ja ohjelmistokomponenteista, jotka toteuttavat yhteistyössä osajärjestelmän tehtävät.
  - Käyttäjät käyttävät järjestelmää yhden tai useamman osajärjestelmän kautta.

Järjestelmän ja ohjelmiston laadinta

- Järjestelmää ja ohjelmistoa on laadittava yhteistyössä:
  - Ohjelmistotuotantoa ei voi tehdä ilman tietoa ympäristöstä, johon ohjelmistoa ollaan tekemässä.
  - Yhteistyö edellyttää ohjelmiston laatijoilta valmiutta kommunikoida sovellusalueen asiantuntijoiden kanssa.
  - Ohjelmistoa on helppo syyttää, jos jokin menee pieleen, vaikka syy olisi muualla.

Mitä vaaditaan hyvältä ohjelmistolta?

- Hyvällä ohjelmistolla on seuraavat keskeiset ominaisuudet:
  - *Ylläpidettävyys:* tuotetta voidaan muokata vastaamaan muuttuviin vaatimuksiin.
  - *Luotettavuus:* tuote ei aiheuta fyysisiä eikä taloudellisia vahinkoja missään tilanteessa.
  - *Tehokkuus:* tuote ei tuhlaa resursseja.
  - *Käytettävyys:* tuote on looginen ja helppokäyttöinen.

2. Ohjelmistotuotannon historiaa

- Ensimmäinen kaupallinen ohjelmisto tehtiin niinkin aikaisin kuin 1951 Englannissa, tekijänä J. Lyons company.
- Alkuaikoina ohjelmistojen teko oli suoraviivaista ja ohjaamatonta. Silti jo 50-luvulla tehtiin kohtuullisen monipuolisia ohjelmistoja.

1960-luku ja ohjelmistokriisi

- 1960-luvun alussa määriteltiin jo aika moderneja asioita, kuten testattavuus ja rajapinnat. Silti ei vielä voitu puhua varsinaisesta ohjelmistotuotannosta.
- Vuosikymmenen edetessä ohjelmistojen koot kasvoivat ja ongelmat pahenivat. Projektit alkoivat myöhästellä ja ylittää kustannuksensa. Syntyi ns. *60-luvun ohjelmistokriisi.*
- Suoraviivainen koodaus ei enää riittänyt.

Ensimmäiset ohjelmistotuotannon konferenssit

- Lopulta 1968 NATO organisoi jo aiemmin mainitun konferenssin, jossa esiteltiin termi Software Engineering.
- Seuravana vuonna NATO organisoi toisen Software Engineering – konferenssin. Yhdessä näissä konferensseissa määriteltiin ohjelmistotuotannon perusteet.

Ensimmäisten konferenssien tulos

- Konferenssien tulokset eivät ehkä ratkaisseet ohjelmistokriisiä.
  - Jotkut asiantuntijat ovat sitä mieltä, että kriisi ei ole vieläkään kunnolla ratkennut.
- Sen sijaan konferensseissa esitellyt ideat olivat perusta, jolle nykyaikainen ohjelmistotuotanto on rakennettu.

1970-luku ja prosessimallit

- 1970-luvulle tultaessa määriteltiin ensimmäinen ohjelmistotuotannon *prosessimalli*: vesiputousmalli.
  - Prosessimalli on mahdollisimman yleisesti sovellettavissa oleva ohjeisto ohjelmistojen tuottamiseen.
- Vesiputousmallin pohjana käytettiin muiden insinööritieteiden *järjestelmäsuunnittelumallia* (systems engineering model).

Lisää prosessimalleja

- Järjestelmäsuunnittelumallin avulla voidaan suunnitella vaikkapa siltoja. Malli kopioitiin suoraan ohjelmistotuotantoon.
- Varsin pian huomattiin, että vesiputousmalli ei sovi kaikenlaisten ohjelmistojen tekoon. Niinpä 70- ja 80-luvulla esiteltiin lukuisia muita ohjelmistotuotannon prosessimalleja.

Ohjelmointikielten alkuaikoja

- Ensimmäiset ohjelmat kirjoitettiin konekielellä, mutta ensimmäiset lausekielet kehitettiin jo 1950-luvulla.
- 60-luvun alussa kehitetyt *Algol*, *Cobol* ja *PL/1* olivat ensimmäiset kunnon kielet.
- 60-luvun lopussa kehitetty *C-kieli* (tietenkin B-kielestä jatkokehitetty) oli merkittävä askel suorituskykyisten ohjelmien kehitystyössä.

1980-luku ja oliopohjaisuus

- 1980-luvun suurin uusi asia oli *oliopohjaisuus*. Ensimmäinen oliokonferenssi OOPSLA pidettiin 1986.
- Oliomaailman periaatteet oli toki esitelty aiemmin, mutta vasta ensimmäiset oliokielet kokosivat ne yhteen.
- Ohjelmointikielistä oliopohjaisuus levisi vähitellen koko ohjelmistotuotantoon.

1990-luvun alku ja henkilökohtainen tietojenkäsittely

- 90-luvulle tultaessa ohjelmistotuotanto oli aika vakiintunutta. 80-luvulla alkanut *henkilökohtainen tietojenkäsittely* (jokaisella oma tietokone) mullisti ohjelmistojen tarpeen mutta ei niiden valmistuksen tekniikkaa.
- Ohjelmistotuotanto eli *työkalujen aikaa*. CASE-työkalut (Computer-Aided Software Engineering) kehittyivät.

1999 ja ketterät prosessimallit

- Toistaiseksi viimeinen ohjelmistotekniikan murros tapahtui 1999. Tällöin esiteltiin ensimmäinen *ketterä prosessimalli* Extreme Programming (XP).
- Ketterät prosessimallit saavuttivat nopeasti erityisesti ohjelmoijien ja PK-yritysten suosion, sillä mallit suosivat ohjelmointia suunnittelun sijaan.

Nykyhetki (2007)

- Tällä hetkellä (2007) ohjelmistotuotantoa kuvaa parhaiten nopea kehitystyö, hajaantuminen ja erikoistuminen.
- Vaikka perusperiaatteet, joita opetetaan mm. tällä kurssilla, ovat säilyneet ennallaan, niin yritykset käyttävät omia räätälöityjä prosessimalleja, menetelmiä ja työkaluja.
- Asiakkaat odottavat tuloksia usein jo 3-4 kk projektin alkamisen jälkeen.
- Yrityksen käyttämien o*hjelmistorajapintojen* (API) hallinta on iso osa ammattitaitoa.

Erikoistunut nykyhetki (2007)

- Myös tehtävät ohjelmistot eroavat toisistaan enemmän kuin koskaan.
  - Yhtäältä meillä on kännyköihin ja mobiililaitteisiin tehtäviä pieniä ohjelmia.
  - Toisaalta meillä on valtavia ohjelmia, joiden suorituskyky ja käyttöturvallisuus on määritelty erittäin korkealle.
- Näin ohjelmistoyritysten täytyy erikoistua säilyttääkseen markkinansa.

Tulevaisuus

- Tulevaisuudessa ohjelmistotuotannolla on kolme haastetta:
  - Epäyhtenäisyyden haaste (heterogeneity challenge).
  - Toimitusajan haaste (delivery challenge).
  - Luottamuksen haaste (trust challenge).
- Haasteet eivät ole riippumattomia. Osa voi olla jopa keskenään ristiriitaisia.

Epäyhtenäisyyden haaste

- Ohjelmistopohjaisten järjestelmien täytyy toimia hyvin erilaisissa ympäristöissä eri järjestelmien kanssa.
- Ohjelmistojen pitää myös toimia vanhojen järjestelmien kanssa.
- Vanhoihin järjestelmiin täytyy usein kehittää uutta toiminnallisuutta.
- Vanhoja järjestelmiä tekohengitetään.

Toimitusajan haaste

- Ohjelmistojen teko ohjelmistotuotannon menetelmin vaatii paljon aikaa. Aika taas on kallis resurssi.
- Liiketoiminnassa taas täytyy olla dynaaminen ja valmis muutoksiin.
- Näin myös ohjelmistojen teossa täytyy valmistua ja mukautua muutokseen aiempaa nopeammin ilman että laatu kärsii.

Luottamuksen haaste

- Ohjelmistot vaikuttavat aina vain enemmän elämäämme. Mitä enemmän ohjelmistot hallitsevat, sitä enemmän meidän pitää voida luottaa niihin.
- Luottamus ei kuitenkaan synny tyhjästä. Ohjelmistoista pitää voida nähdä, että ne toimivat halutulla tavalla.
- Esimerkiksi autoissa on paljon autojen ohjelmistoista johtuvia pikkuvikoja.

3. Ohjelmistotuotantoprosessi

- *Ohjelmistotuotantoprosessi* (software process) on joukko toimintoja, jotka johtavat ohjelmiston valmistumiseen.
- Prosessi on ohjeisto. Se kertoo työvaiheet ja niiden väliset suhteet.
- Prosessin ilmentymä on *projekti*. Siinä toteutetaan prosessin työvaiheet.
- Käytännössä jokainen ohjelmistoyritys käyttää itselleen räätälöityjä prosesseja.

Prosessimalli

- *Prosessimalli* (process model) on abstrakti kuvaus prosessista.
  - Prosessimallista voidaan johtaa monta erilaista prosessia eri organisaatioihin ja sovellusalueille.
- Prosessimalli kuvaa yleisperiaatteen, ei yksityiskohtia.
- Näin prosessimalli määrittelee kehykset, joihin itse prosessit rakennetaan.

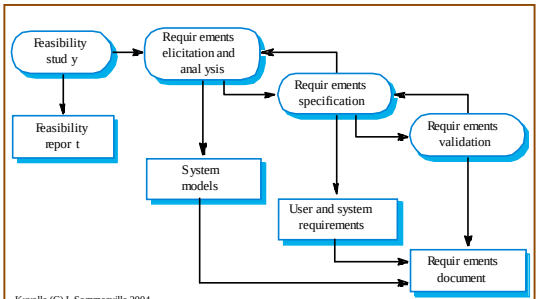
Prosessimallien perustehtävät

- Jokaiseen prosessimalliin sisältyy tavalla tai toisella seuraavat *perustehtävät*:
  - Vaatimusten keruu ja analyysi (vaatimusmäärittely),
  - Ohjelmiston suunnittelu,
  - Toteutus ja yksikkötestaus,
  - Integrointi ja järjestelmätestaus,
  - Käyttö ja ylläpito.

Vaatimusmäärittely

- Mitä vaatimuksia ohjelmistolle asetetaan:
  - Mitä toimintoja sen pitää sisältää.
  - Mitä laadullisia vaatimuksia ohjelmistolle asetetaan (vaikkapa nopeuden tai käyttötavan suhteen).
  - Mitä rajoituksia ohjelmiston toimintaan liittyy (kuten muiden ohjelmistojen yhteistoiminnan synnyttämät ehdot).

Vaatimusmäärittelyprosessi



Ohjelmiston suunnittelu

- Suunnittelu kuvaa ohjelmiston rakenteen, ohjelmiston tarvitsemat ja käsittelemät tiedot, komponenttien väliset rajapinnat ja käytetyt algoritmit.
- Suunnittelu perustuu kerättyihin vaatimuksiin. Sen tuloksena saadaan yksi tai useampi ohjelmiston toteutuksessa käytettävä malli ohjelmiston rakenteesta.

Suunnittelun tasot

- Suunnitteluprosessi vaatii useita työvaiheita, jotka käsittelevät ongelmaa eri tarkkuustasoilla. Tasot voivat olla seuraavat (tai jotain muuta; suunnittelua ei kannata yhtenäistää liikaa):
  - Arkkitehtuurisuunnittelu
    - Tehdään järjestelmästä korkean tason malli, josta selviävät osajärjestelmät ja niissä käytettävät komponentit.

Suunnittelun tasot 2

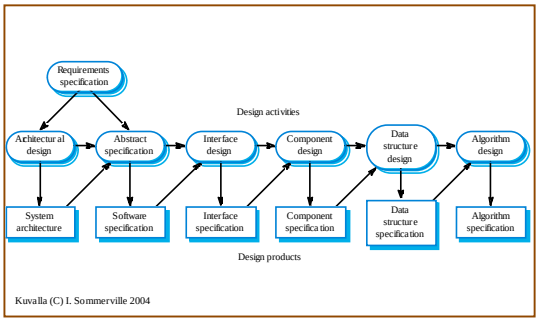
- Abstrakti määrittely
  - Määritellään kunkin osajärjestelmän tarjoamat palvelut ja rajoitteet.
- Rajapintasuunnittelu
  - Määritellään kunkin osajärjestelmän tarjoamat rajapinnat muille osajärjestelmille.
- Komponenttisuunnittelu
  - Määritellään osajärjestelmittäin palvelut toteuttavat komponentit rajapintoineen ja suunnitellaan ne.

Suunnittelun tasot 3

- Tietorakenteiden suunnittelu
  - Suunnitellaan järjestelmän toteutuksessa välttämättömät tietorakenteet.
- Algoritmien suunnittelu
  - Suunnitellaan järjestelmän toteutuksessa käytettävät algoritmit.

- Komponentti on tietyt tehtävät tekevä ohjelmiston osa, joka yhteistyössä muiden komponenttien kanssa toteuttaa ohjelmiston tarjoamat palvelut.

Yleinen suunnitteluprosessi



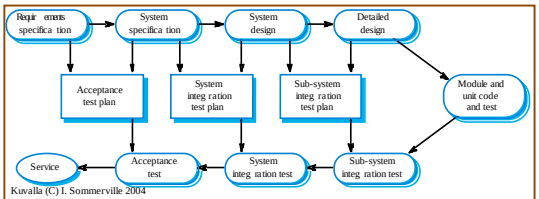
Toteutus ja yksikkötestaus

- Toteutuksessa suunnitelman perusteella tehdään komponentteja.
  - Komponentti tekee yhden asian (eikä muuta) kunnolla. Sillä on selkeä rajapinta ulospäin.
  - Komponentti voi olla kooltaan mitä tahansa yhdestä oliosta kokonaiseen ohjelmaan.
- Toteutuksen osana tehtävässä yksikkötestauksessa varmistetaan, että komponentit toimivat suunnittelussa määritellyllä tavalla.

Integrointi ja järjestelmätestaus

- Ohjelmakomponentit kootaan yhteen.
- Komponenttien välinen yhteistyö testataan integrointitestauksessa.
- Komponenteista kootaan osajärjestelmiä (itsenäisiä osia).
- Osajärjestelmien yhteistyö testataan.
- Koko järjestelmä testataan todellisessa käyttöympäristössä: järjestelmätestaus.

Ohjelmistoprosessin testausvaiheet



- Kaaviossa testaus on alalaidassa ja suunnittelu yläalaidassa.
- Testauksen kaaviokuva kuvaa *V-mallin*. Siitä puhutaan lisää myöhemmin.

Ohjelmiston käyttö ja ylläpito

- Ohjelmistoa on usein tarvetta muuttaa myöhemminkin:
  - Käytössä paljastuu virheitä.
  - Käyttäjien vaatimukset täsmentyvät tai muuttuvat.
  - Käyttöympäristö ja laitteisto muuttuvat.
  - Käyttötavat muuttuvat.
- Ylläpito = ohjelmiston muuttaminen käyttöönoton jälkeen.
- Evoluutio = väistämättömät tekijät, jotka pakottavat ohjelmiston muuttumaan.

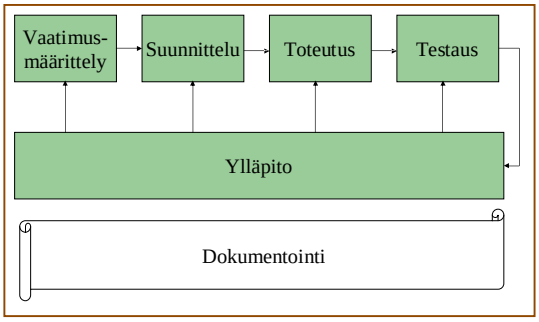
3.1. Yleisimmät prosessimallit

- Yleisimmät prosessimallit ja niiden tyypillisimmät ominaisuudet ovat:
  - Vesiputousmalli: tehtävät peräkkäin.
  - Evoluutiomalli: prototyypit.
  - Komponenttimalli: valmiiden osien käyttö.
  - Lisäävä malli: tehdään vähän kerrassaan.
  - Spiraalimalli: toistetaan samat tehtävät eri mittakaavassa eri vaiheissa.
  - Ketterät mallit: vähemmän hallintoa.

Vesiputousmalli

- Vesiputousmallissa on lineaarinen lähestymistapa:
  - Jo päätettyyn vaiheeseen ei enää palata myöhemmin.
  - Jokainen vaihe saatetaan loppuun, sen tulokset hyväksytään ja jäädytetään ennen seuraavaan vaiheeseen siirtymistä.
- Mallin vaiheet ovat suoraviivaisesti prosessimallin perustehtävät.

Vesiputousmallin tehtäväjako



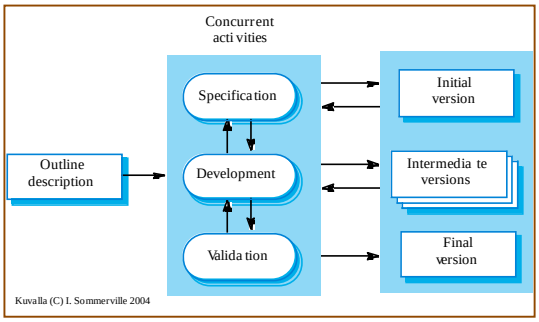
Vesiputousmallin edut ja haitat

+ Malli on selkeä ja helppo omaksua
+ Kustannusarvio ja aikataulu on helpompi tehdä kuin muissa malleissa.
+ Ohjelmiston arkkitehtuuri pysyy yleensä hyvin koossa.
+ Tuloksena saatava hyvä dokumentaatio auttaa ylläpidossa.
- Kaikkien vaatimusten selvittäminen kerralla on hankalaa, ja vaatimukset voivat muuttua.
- Puhtaasti lineaarinen prosessi ei yleensä onnistu, vaan tarvitaan iteratiivisuutta.
- Asiakas ei osallistu kehitystyöhön.
- Virheiden elinkaari voi olla hyvin pitkä.

Evoluutiomalli

- Evoluutio- tai prototyyppimallissa ohjelmistoa kehitetään väliversioittain.
  - Väliversiot ovat prototyyppejä, joita jalostetaan askeleittain tuotantoversioksi.
  - Ensimmäiset prototyypit ovat suppeita ”purkalla koottuja” nopeita versioita.
  - Käyttäjä antaa palautetta väliversioista.
  - Palautteen perusteella jatketaan kehitystä ja parannetaan samalla laatua.

Evoluutiomallin vaiheet



Mitä prototyypille tehdään

- Prototyypin hylkääminen
  - Tarkoituksena on tarkentaa asiakkaan toiveita ja vaatimuksia.
- Prototyypin kehittäminen eteenpäin
  - Tarkoituksena on esitellä asiakkaalle ohjelmiston ydintoiminnot.
- Miksi prototyyppi ei ole valmis ohjelmisto?
  - Prototyyppi on esitoteutus.
  - Prototyyppi ei täytä valmiille ohjelmistolle asetettavia laatuvaatimuksia.

Evoluutiomallin edut ja haitat

+ Vaatimuksia ei tarvitse määritellä kerralla.
+ Asiakas antaa jatkuvasti palautetta.
+ Virheiden elinkaari on yleensä lyhyt.
+ Käyttöliittymä on koko ajan näkyvillä, joten ohjelmisto pitäisi olla helppo oppia.
- Jatkuvat muutokset rikkovat arkkitehtuurin.
- Prototyyppien tekoon tarvitaan sovelluskehitin.
- Asiakkaan voi olla vaikea ymmärtää, että prototyyppi ei kelpaa lopputuotteeksi.
- Viimeinen prototyyppi onkin lopputuote, joten projektilla ei ole selkeää päätepistettä.

Komponenttimalli

- Tavoitteena on käyttää mahdollisimman paljon jo valmiita ohjelman osia eli komponentteja.
- Valmiita komponentteja saadaan
  - omista aiemmista projekteista,
  - ostamalla tai
  - ilmaisista Free Software –projekteista (vaarallisia, koska ylläpidosta ei ole takeita)
- Uudelleenkäytettäviä komponentteja pidetään komponenttikirjastossa.

Komponenttimallin työvaiheet

- Komponenttimallin työvaiheet ovat:
  - Vaatimusmäärittely
    - Etsitään ja analysoidaan vaatimuksia.
  - Komponenttianalyysi
    - Etsitään komponenttikirjastosta vaatimusanalyysin mukaiseen tuotteeseen sopivia komponentteja.
  - Vaatimusten muokkaus
    - Täydennetään vaatimuksia ottaen huomioon löydetyt komponenttiehdokkaat.

Komponenttimallin työvaiheet II

- Työvaiheet jatkuvat:
  - Suunnittelu ja komponenttien eristys
    - Eristetään komponenttikirjastoon omia kelvollisia komponenttiehdokkaiden suunnitelmia.
  - Toteutus ja komponenttien eristys
    - Eristetään komponenttikirjastoon omia kelvollisia komponenttiehdokkaiden toteutuksia.
  - Järjestelmän validointi
    - Varmistetaan, että tehtiin oikea järjestelmä.

Komponenttimallin edut ja haitat

+ Valmiiden komponenttien käyttö voi säästää paljon työtä.
+ Komponenttiajattelu ohjaa uudelleenkäytettäviin ratkaisuihin.
+ Komponenttikirjaston omaa sisältöä voidaan halutessa myydä muille.
+ Pyörää ei tarvitse keksiä joka kerta uudelleen.
- Ohjelmistoa saatetaan tehdä liikaa komponenttien ehdoilla, jolloin asiakkaan vaatimukset yritetään väkisin muokata komponentteihin sopiviksi.
- Komponentin dokumentaatio voi olla vajaa.
- Komponentin liittäminen oman ohjelmiston osaksi voi olla yllättävän työlästä.
- Valmiissa komponentissa olevan virheen löytäminen ja korjaus on usein hyvin hankalaa.

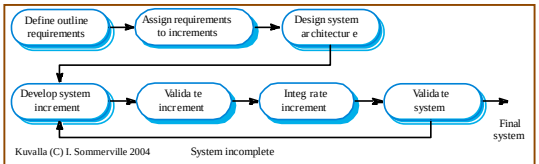
Lisäävä malli

- Vesiputousmallissa ohjelma määritellään ja suunnitellaan yhtenä kokonaisuutena. Tämä ei usein toimi:
  - vaatimuksia ei tunneta kunnolla,
  - vaatimukset ovat epäselviä tai
  - vaatimukset muuttuvat projektin aikana.
- Lisäävässä mallissa ohjelma määritellään ja toteutetaan iteratiivisesti sykleissä: ytimestä erikoisosiin.

Lisäävä malli II

- Ohjelmisto jaetaan osiin, jotka toteutetaan omina sykleinä.
  - Ensin toteutetaan ydin, johon seuraavien syklien tulokset voidaan liittää.
- Vain yhden syklin vaatimukset käsitellään kerrallaan.
- Toteuttamattomien syklien vaatimukset voivat muuttua.

Lisäävän mallin strategia, edut ja haitat



+ Tuote on käyttökelpoinen alusta alkaen.
+ Tärkeimmät piirteet toteutetaan ensin.
+ Virheelliset määritykset huomataan nopeasti.
+ Syklin aikataulu pysyy yleensä hyvin hallussa.
- Koko projektin aikataulun laadinta vaikeutuu.
- Projektin hallinta on hankalaa: tehtävät ovat pieniä ja rinnakkaisia.
- Mitä lyhyempi on sykli, sitä suurempi on kiusaus säästää laadusta.

Spiraalimalli

- Spiraalimallissa kukin spiraalin kierros esittää yhtä prosessin työvaihetta.
- Kussakin spiraalissa on neljä sektoria:
  - tehtävän asetus, tavoite ja aikataulu,
  - riskianalyysi,
  - kehitystyö ja tuloksen verifiointi,
  - tuloksen arviointi ja seuraavan spiraalin suunnittelu.
- Spiraaleissa voidaan käyttää mitä tahansa prosessimallia.

Spiraalimallin edut ja haitat

+ Malli sopii mainiosti erikokoisiin tehtäviin.
+ Kussakin spiraalissa voidaan käyttää omaa prosessia.
+ Riskienhallinta on osa prosessimallia.
+ Mallilla voidaan kuvata tuotteen koko elinkaari mukaan lukien ylläpito.
- Malli voi olla vaikea selittää sitä tuntemattomalle.
- Projektin hallinta voi olla vaikeaa.
- Aikataulun tekeminen vaikeutuu.
- Käytännön kokemuksia on vähän: ketterät prosessimallit ovat syöneet spiraalimallin suosiota.

Ketterät prosessimallit

- 1980- ja 1990-luvun prosessimalleissa korostettiin
  - huolellista projektisuunnittelua,
  - formaalia laadunvalvontaa,
  - yksityiskohtaisia analyysi- ja suunnittelumenetelmiä,
  - CASE-työkalujen käyttöä ja
  - täsmällistä tarkasti ohjattua ohjelmistoprosessia.

Ketterien prosessimallien synty

- Prosessimallit tukivat erityisesti laajojen, pitkäikäisten ohjelmistojen kehitystyötä, mutta pienten ja keskisuurten ohjelmistojen tekoon ne osoittautuivat usein turhan jäykiksi.
- Ristiriidan seurauksena syntyi joukko *ketteriä prosessimalleja* (agile process models), jotka korostivat itse ohjelmistoa yksityiskohtaisen suunnittelun ja dokumentaation sijaan.
  - Näistä tunnetuimmat ovat *eXtreme Programming* (XP) ja *Scrum.*

Ketterien prosessien luonne

- Kaikki ketterät prosessimallit ovat lisäävän mallin tapaan iteratiivisia.
- Yksi sykli on vain muutaman viikon mittainen. Sen aikana käydään läpi vaiheet vaatimuksista luovutukseen.
- Lyhyen syklin johdosta asiakas on jatkuvasti mukana kehitystyössä. Asiakkaan palautetta tarvitaan jatkuvasti

Ketterien prosessien periaatteet

- Asiakkaan osallistuminen
  - Asiakas osallistuu aktiivisesti kehitystyöhön. Asiakas esittää ja priorisoi uusia järjestelmän vaatimuksia ja evaluoi syklin aikana saadut tulokset.
- Lisääntyvät ominaisuudet
  - Kullakin syklillä ohjelmistoon lisätään uusia ominaisuuksia, joiden vaatimukset saadaan asiakkaalta.

Ketterien prosessien periaatteet II

- Ihmiset prosessia tärkeämmät
  - Työntekijöiden taidot ja toimintatavat ovat tärkeämmät kuin vahvasti ohjattu prosessi.
- Muutoksen hyväksyminen
  - Vaatimukset muuttuvat, joten edellisillä sykleillä tehtyä koodia saatetaan joutua uudistamaan.
- Yksinkertaisuuden ylläpito
  - Sekä ohjelmisto että prosessi tulee pitää mahdollisimman yksinkertaisena (mutta ei yhtään yksinkertaisempana). Dokumentaatio minimoidaan

Ketterien prosessimallien edut ja haitat

+ Lyhyessä syklissä vaatimusmäärittely, suunnittelu ja yksikkötestaus on helppo tehdä.
+ Asiakas on jatkuvasti läsnä antamassa palautetta.
+ Mallit ovat joustavia suosien tekijöitä enemmän kuin hallintaa.
+ Vaatimukset saavat muuttua projektin aikana.
+ Pienten ja keskisuurten projektien teko on tehokasta.
- Koko projektin aikataulun laadinta on hyvin vaikeaa.
- Projektin hallinta on vaikeaa.
- Jatkuva koodin uudistaminen on kallista ja voi heikentää tuotteen laatua.
- Pieninkään mahdollinen kokonaisuus ei aina mahdu sykliin.
- Sykliin on kiusaus laittaa enemmän tavaraa kuin sinne oikeastaan mahtuu.
- Isojen projektien tekeminen on vaikeaa.

Extreme programming (XP)

- Ohjelma kehitetään hyvin pieninä sykleinä.
  - Aluksi tehdään pieni ydin.
  - Jatkossa lisätään sykli kerrallaan mahdollisimman pieni järkevä joukko ominaisuuksia.
  - Toteutusjärjestys määräytyy asiakkaiden tarpeista
- Projektia suunnitellaan mahdollisimman lyhyt aika eteenpäin.
- Dokumentaatio karsitaan minimiin. Toimiva tuote on tärkeämpi kuin raskas dokumentti-kirjasto. Koodi on tärkein dokumentti.
- Malliin kuuluu joukko prosessiohjeita.

4. Ohjelmistojen vaatimusmäärittely

- Ehkä kaikkein merkittävin ohjelmistotuotannon ongelma on selvittää asiakkaalta järjestelmälle asetettavat *vaatimukset* (requirements)
  - Kaikki helposti määriteltävät järjestelmät on jo määritelty, joten ongelma vain pahenee.
- Vaatimukset kuvaavat sen, minkälaisen järjestelmän asiakas haluaa auttamaan tai helpottamaan jotain toimintaa.

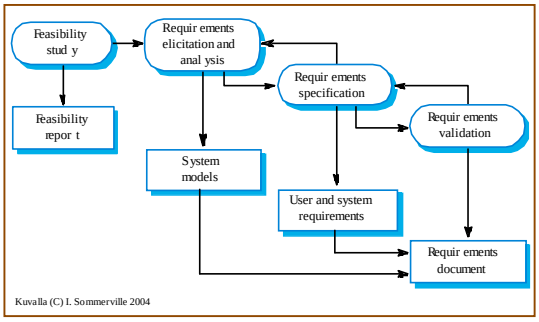
Vaatimusmäärittelyn tehtävät

- *Vaatimusmäärittely* (requirements engineering) tarkoittaa prosessia, jossa selvitetään järjestelmälle asetettavat vaatimukset.
- Vaatimusmäärittelyssä selvitetään
  - mitä järjestelmän pitää tarjota (toiminnot),
  - toivotut ja vaaditut koko järjestelmää koskevat ominaisuudet (laatuvaatimukset),
  - järjestelmän rajoitukset ja reunaehdot.

4.1. Vaatimusmäärittelyprosessi

- Kuten yleisemmissäkin prosesseissa, myös vaatimusmäärittelyprosesseissa on paljon vaihtelua. Käytännössä kaikista prosesseista löytyvät kuitenkin samat perustehtävät:
  - *kelpoisuusselvitys* (feasibility study),
  - *kartoitus ja analyysi* (elictation and analysis),
  - *määrittely* (specification) ja
  - *validointi* (validation).

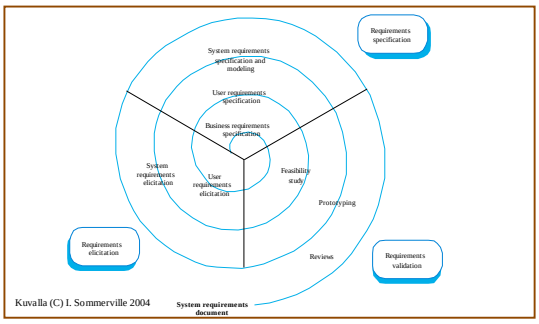
Vaatimusmäärittelyn tehtäväjako (sama kuin kalvo 27)



Vaatimusmäärittelyn spiraalimalli

- Vaatimusmäärittelyyn voidaan ottaa myös spiraalimainen näkökulma.
- Tällöin saadaan mukaan näkökulma, minkä mukaan vaatimuksia ei saada kokoon kerralla vaan iteratiivisesti.
- Malli sallii myös vaatimusmäärittelyn rinnalla tehtävän ohjelmiston suunnittelun/toteutuksen.

Vaatimusmäärittelyn spiraalí



Kelpoisuusselvitys

- Kelpoisuusselvitys on lyhyt esivaihe vaatimusmäärittelylle. Siinä vastataan seuraaviin kysymyksiin:
  1. Tuoko kehitettävä järjestelmä lisäarvoa asiakkaalle?
  2. Voidaanko järjestelmä toteuttaa nykyisellä teknologialla projektille varatulla aikataululla ja budjetilla?
  3. Voidaanko järjestelmä integroida jo käytössä oleviin järjestelmiin?

Kelpoisuusselvitys 2

- Lisäksi, vaikka Sommerville ei sitä listaa:
  4. Kannattaako järjestelmä toteuttaa, vai voidaanko vastaava järjestelmä ostaa valmiina? Onko ostettava järjestelmä sovitettavissa asiakkaan tarpeisiin?
- Tuloksena saadaan päätös siitä, kannattaako järjestelmän kehitystyötä jatkaa.

Vaatimusten kartoitus ja analyysi

- Vaatimusten kartoitus ja analyysi – työvaiheessa etsitään yhteistyössä asiakkaan ja järjestelmän loppukäyttäjien kanssa,
  - mitä palveluja järjestelmältä vaaditaan,
  - mitä järjestelmän suorituskyvyltä vaaditaan,
  - mitä laitteisto- ja ympäristörajoituksia on huomioitava jne.
- Työvaiheessa huomioidaan *sidosryhmät.*

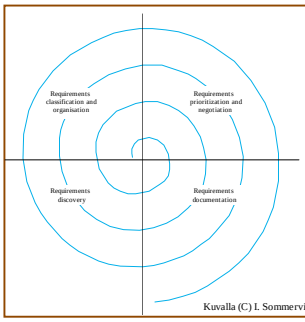
Sidosryhmät

- *Sidosryhmä* (stakeholder) on henkilö tai ryhmä, joka suoraan tai välillisesti on tekemisissä kehitettävän järjestelmän kanssa.
- Eri sidosryhmillä on erilaisia tarpeita kehitettävälle järjestelmälle è ristiriitoja.
- Sidosryhmät eivät välttämättä osaa kertoa, mitä kaipaavat järjestelmältä.

Kartoituksen ja analyysin vaiheet

- Vaatimusten kartoitus ja analyysi voidaan osittaa seuraaviin tehtäviin:
  - *Vaatimusten esiinkaivelu* (req. discovery)
  - *Vaatimusten luokittelu* (req. classification and organization)
  - *Vaatimusten priorisointi* (req. priorization)
  - *Vaatimuksista neuvottelu* (req. negotiation)
  - *Vaatimusten dokumentointi* (req. documentation)

Kartoituksen ja analyysin spiraali



Kuvalla (C) I. Sommerville 2004

- Kartoitus ja analyysi on iteratiivinen prosessi.
- Prosessi aloitetaan isoista vaatimuksista ja edetään kohti pienempiä ja pienempiä yksityiskohtia.

Vaatimusten esiinkaivelu

- Vaatimusten esiinkaivelussa sidosryhmiltä selvitetään, mitä he odottavat järjestelmältä.
- Selvittäminen voi olla suoraviivaista (esim. haastattelut) tai se voi epäsuoraa (dokumentaatioon tutustuminen, loppukäyttäjien työympäristöön tutustuminen).

Esiinkaivelun tekniikoita

- Esiinkaivelussa voidaan käyttää esimerkiksi seuraavia tekniikoita:
  - Näkökulmat (viewpoints): mitä suoria tai epäsuoria yhteyksiä sidosryhmillä on järjestelmään.
  - Haastattelut (interviews): sidosryhmiltä selvitetään sopivin kysymyksin, mitä he odottavat järjestelmältä.

Esiinkaivelun tekniikoita 2

- Lisää tekniikoita:
  - Skenaariot (scenarios) ja käyttötapaukset (use cases): kirjataan suoraviivaisia kuvauksia tulevan järjestelmän käytöstä, skenaarioita. Tarvittaessa toisiinsa sidoksissa olevat skenaariot ryhmitellään yhteen käyttötapauksiksi.
  - Etnografia (ethnography): mennään paikan päälle katsomaan, miten tulevan järjestelmän loppukäyttäjät työskentelevät.

Vaatimusten määrittely

- Vaatimusten määrittelyssä vaatimusten kartoituksessa ja analyysissa löydetyt vaatimukset lajitellaan *käyttäjä-* ja *järjestelmävaatimuksiksi*.
- Tarvittaessa vaatimuksia muokataan ja tarkennetaan. Käyttäjävaatimuksista voidaan johtaa uusia järjestelmävaatimuksia.
- Tuloksena saadaan koko järjestelmää esittävät priorisoidut vaatimukset.

Käyttäjävaatimukset

- Käyttäjävaatimuksissa kuvataan luonnollisella kielellä ja kaavioilla,
  - mitä palveluja järjestelmän odotetaan tarjoavan ja
  - mitä rajoituksia ja reunaehtoja järjestelmällä on.
- Käyttäjävaatimusten lukijat eivät ole ohjelmistotuotannon ammattilaisia, joten kuvausten pitää olla helppolukuisia.

Järjestelmävaatimukset

- Järjestelmävaatimukset määrittelevät yksityiskohtaisesti ja rakenteisella yhtenäisellä kuvaustekniikalla
  - järjestelmän tarjoamat palvelut,
  - järjestelmän toiminnot,
  - järjestelmän toiminnan rajoitteet.
- Järjestelmävaatimukset määrittelevät **täsmälleen**, mitä järjestelmä tekee ja mitä ehtoja sille asetetaan.

Käyttäjä- ja järjestelmävaatimusten käyttö

- Käyttäjävaatimuksia käytetään
  - kilpailutettaessa ohjelmistoyrityksiä ohjelmistosopimuksesta,
  - kuvattaessa järjestelmää loppukäyttäjille,
  - kuvattaessa järjestelmää omalle johdolle.
- Järjestelmävaatimuksia käytetään
  - sopimuksena asiakkaan ja ohjelmistoyrityksen välillä,
  - suunnittelun lähtökohtana.

Vaatimusten luokittelu

- Käyttäjä- ja järjestelmävaatimukset luokitellaan kolmeen ryhmään:
  - *Toiminnalliset vaatimukset* (functional requirements) kuvaavat järjestelmän tarjoamat palvelut, miten järjestelmä reagoi annettuihin syötteisiin ja miten järjestelmä toimii määritellyissä tilanteissa.
    - Joskus toiminnalliset vaatimukset määrittelevät myös, mitä järjestelmä ei missään tapauksessa tee.

Vaatimusten luokittelu 2

- *Ei-toiminnalliset vaatimukset* (non-functional requirements) ovat järjestelmän toiminnoille asetettavia rajoituksia ja reunaehtoja.
- Ei-toiminnalliset vaatimukset koskevat koko järjestelmää. Kaikki järjestelmän toiminnalliset vaatimukset toteuttavat järjestelmän ei-toiminnalliset vaatimukset.
- Jotkut ei-toiminnalliset vaatimukset voivat koskea myös järjestelmän kehitystyötä.

Vaatimusten luokittelu 3

- *Toimintaympäristövaatimukset* (domain requirements) ovat vaatimuksia, jotka järjestelmän täytyy täyttää toimiakseen yhteistyössä muun maailman kanssa.
- Toimintaympäristövaatimukset (tai lyhyemmin ympäristövaatimukset) liittyvät esimerkiksi muihin yhteistyössä toimiviin järjestelmiin tai fysiikan lakeihin.
- Toimintaympäristövaatimukset voivat olla toiminnallisia tai ei-toiminnallisia.

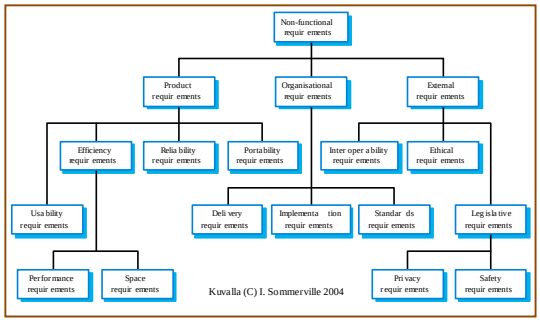
Toiminnalliset vaatimukset

- Toiminnalliset vaatimukset voidaan kuvata
  - palveluina: mitä kaikkea järjestelmä tarjoaa,
  - toimintoina: mitä järjestelmä tekee ja miten,
  - palveluina ja toimintoina: mitkä toiminnot toteuttavat mitkä palvelut (ristiviittaus).
- Toiminnallisten vaatimusten on oltava
  - täydellisiä: kaikki palvelut määritellään,
  - yhdenmukaisia: ei ristiriitoja.

Ei-toiminnalliset vaatimukset

- Ei-toiminnalliset vaatimukset ovat ainakin yhtä tärkeitä kuin toiminnalliset vaatimukset. Ne kertovat sen, miten käyttäjä kokee järjestelmän.
- On mahdollista, että ei-toiminnallisen vaatimuksen pettäessä koko järjestelmä on käyttökelvoton.
- Kehitystyöhön liittyvät ei-toiminnalliset vaatimukset vaikuttavat esim. ylläpitoon.

Ei-toiminnallisia vaatimuksia



Käyttöympäristövaatimukset

- Käyttöympäristövaatimukset ovat tärkeitä, sillä ne määrittelevät ehdot, jotka järjestelmän tulee täyttää, jotta se voi toimia yhteistyössä muiden järjestelmien tai fysiikan lakien kanssa.
- Käyttöympäristövaatimuksia ovat esimerkiksi järjestelmälle määriteltävät rajapinnat muihin järjestelmiin ja luonnonlaeista seuraavat rajoitteet.

Vaatimusten validointi

- Vaatimusten validoinnissa
  - ratkotaan vaatimusten väliset konfliktit ja
  - osoitetaan, että vaatimuksilla kuvattu järjestelmä on sitä, mitä asiakas haluaa.
- Validointi on erittäin tärkeää, sillä virheelliset vaatimukset heijastuvat koko tuotteen elinkaaren ajan.
  - Mitä myöhemmin virheellinen vaatimus keksitään, sitä kalliimpaa sen korjaus on.

Vaatimusten tarkastus

- Vaatimuksista tarkastetaan
  - Yhtenäisyys: onko vaatimukset kuvattu johdonmukaisella tarkkuudella ja tekniikoilla?
  - Täydellisyys: kuvaavatko vaatimukset koko järjestelmän?
  - Verifioitavuus: ovatko vaatimukset testattavissa?
  - Ymmärrettävyys: ymmärtävätkö tekijät ja loppukäyttäjät vaatimukset?
  - Seurattavuus: onko jokaisen vaatimuksen esittäjä ja sidosryhmä tiedossa?
  - Mukautuvuus: Voidaanko vaatimuksia muuttaa ilman laajoja muutoksia muihin vaatimuksiin?

Vaatimusdokumentti

- *Vaatimusdokumentti* (software requirements document tai software requirements specification, SRS) kuvaa yksiselitteisesti ainakin seuraavat asiat:
  - Järjestelmän käyttötarkoituksen.
  - Dokumentaatiossa käytetyn ammattisanaston.
  - Käyttäjä- ja järjestelmävaatimukset.
  - Järjestelmäarkkitehtuurin ja järjestelmämallit.
  - Järjestelmän elinkaaren ja yhteydet muihin järjestelmiin.

Vaatimusdokumentin rakenne

- Vaatimusdokumentin rakenteen pitää olla sellainen, että sitä voidaan käyttää ainakin
  - sopimuksena asiakkaan ja ohjelmistoyrityksen välillä,
  - suunnittelun lähtökohtana,
  - johdolle esiteltävänä yleisenä kuvauksena tehtävästä tuotteesta ja
  - järjestelmätestauksen lähtökohtana.

5. Ohjelmistojen suunnittelu

- Suunnittelussa päätetään ohjelmiston loogisesta rakenteesta.
- Rakenne kuvataan eri *abstraktiotasoilla.*
  - Mitä korkeampi abstraktiotaso, sitä paremmin selviää yleiskuva, mutta sitä vähemmän näkyy yksityiskohtia.
  - Korkein suunnittelun abstraktiotaso on arkkitehtuurisuunnitelma, matalin on pseudokoodi.

Suunnittelun luonne

- Suuunnittelu on luovaa toimintaa, jota ei kannata rajoittaa liikaa prosesseihin.
  - Myös vastakkainen kanta, tarkasti ohjattu suunnitteluprosessi, on saanut kannatusta.
- Ei ole helppoa suoraviivaista tapaa suunnitella ohjelmisto. Jokainen suunnittelu on ainutkertainen.
- Suunnittelun lähtökohtana voi käyttää esimerkiksi prosessia kalvolla 32.

5.1. Arkkitehtuurisuunnittelu

- Suuret järjestelmät ositetaan aina keskenään yhteistyötä tekeviksi osajärjestelmiksi.
- Osajärjestelmien tunnistaminen ja niiden välisen viestinvälityksen määrittely on arkkitehtuurisuunnittelua.
- Arkkitehtuurisuunnittelun lopputuloksena saadaan järjestelmän arkkitehtuurikuvaus.

Osajärjestelmät ja ei-toiminnalliset vaatimukset

- Osajärjestelmiin jako riippuu ei-toiminnallisista vaatimuksista:
  - Jos *suorituskyky* on tärkeää, kannattaa käyttää mahdollisimman pientä lukumäärää laajoja osajärjestelmiä.
  - Jos *turvallisuus* on tärkeää, kannattaa käyttää sisäkkäisiä arkkitehtuureja, joista kaikkein sisimmällä tasolla ovat turvallisuuskriittisimmät toiminnot.
  - Jos *saatavuus* on tärkeää, samat palvelut kannattaa toistaa monessa osajärjestelmässä.
  - Jos *ylläpidettävyys* on tärkeää, kannattaa käyttää suurta määrää pieniä osajärjestelmiä.

Arkkitehtuurisuunnitelma

- Arkkitehtuurisuunnittelun lopputuloksena saadaan järjestelmän arkkitehtuurisuunnitelma. Se sisältää
  - koko järjestelmän kuvauksen,
  - järjestelmästä tehdyt kaaviokuvat selityksineen,
  - järjestelmän jaon osajärjestelmiin ja osajärjestelmittäin komponentteihin,
  - osajärjestelmien välisen yhteydenpidon.

Osajärjestelmin ositus

- Arkkitehtuurisuunnittelun jälkeen kukin osajärjestelmä ositetaan *moduuleiksi* (module).
  - Moduuli on järjestelmän komponentti, joka käyttää muiden moduuleiden palveluja ja tarjoaa muille moduuleille palveluja.
  - Moduuli ei ole itsenäinen,kuten osajärjestelmä, vaan tiiviissä yhteistyössä muiden moduulien kanssa toteuttaa osajärjestelmän toiminnot.

5.2. Rajapintasuunnittelu

- Rajapintasuunnittelu on ehkä suunnittelun tärkein työvaihe. Siinä jokaisen osajärjestelmän tai moduulin rajapinta muihin osajärjestelmiin tai moduuleihin suunnitellaan ja dokumentoidaan yksityiskohtaisesti.
- Rajapintojen pitää olla sellaiset, että tarjottuja palveluja voidaan käyttää tietämättä niiden toteutuksesta mitään.

Rajapintojen määrittely

- Rajapintojen määrittelyt tulee tehdä yksiselitteisiksi ja ristiriidattomiksi.
- Rajapintakuvauksesta tulee näkyä:
  - Rajapinnan palvelut.
  - Kunkin palvelun syötteiden ja tulosteiden tyypit ja arvoalueet.
  - Kunkin palvelun kohdalta ehdot, joiden on oltava voimassa ennen palvelun kutsua ja ehdot, jotka palvelu takaa kutsun jälkeen.
- Rajapinnoissa oikea abstraktiotaso on tärkeä.

5.3. Oliopohjainen suunnittelu

- Aiemmin esitelty suunnitteluprosessi (kalvo 32) on geneerinen, jota voidaan tarkentaa esimerkiksi oliopohjaisessa lähestymistavassa.
- Sommerville listaa viisi työvaihetta, jotka sopivat erityisesti oliopohjaiseen suunnitteluun:
  - ymmärrys, arkkitehtuuri, oliot, mallit, rajapinnat.

Oliopohjaisen suunnittelun perustehtävät

1. Järjestelmän toiminnan ja sen erilaisten käyttötapojen ymmärtäminen.
   - Siis vaatimusten ymmärtäminen.
2. Järjestelmäarkkitehtuurin suunnittelu
   - Arkkitehtuurisuunnittelua, kokonaiskuvan määrittely.
3. Tärkeimpien olioiden tunnistus
   - Oliot voivat olla eritasoisia osajärjestelmästä palvelun tai toiminnon yksityiskohtaan.
4. Järjestelmämallien kehitys
   - Osajärjestelmä- ja komponenttisuunnittelua.
5. Olioiden rajapintojen määrittely
   - Rajapintasuunnittelua.

Oliomalli ja yleinen suunnittelumalli

- Yleiseen suunnittelumalliin verrattuna oliosuunnittelumalli painottaa olioiden roolia.
- Oliot ovat käteviä mm. ulkomaailman mallintamisessa, mutta silti on järkevää pitää lisäksi komponenttitaso kokoamassa yhteistyössä toimivat oliot yhteen.
- Olioita voidaan tunnistaa järjestelmämalleista ja –kuvauksista. Näitä olioita käyttämällä ja uusia lisäämällä saadaan tarkennettua arkkitehtuuria ja kehitettyä järjestelmämalleja.

6. Verifiointi ja validointi

- Verifiointi ja validointi (V&V) on ohjelmistotuotannon työvaihe, jossa varmistetaan, että
  - ohjelmisto täyttää sille asetetut vaatimukset ja
  - ohjelmisto täyttää sen tilanneen asiakkaan ohjelmistolle asettamat tarpeet.
- V&V:ta tehdään koko ohjelmistoprosessin elinkaaren ajan.

Verifioinnin ja validoinnin ero

- Verifioinnissa varmistetaan, että ohjelmisto vastaa määrityksiään – siis kirjattuja vaatimuksia:
  - "Verification: Are we building the product right?"
- Validoinnissa varmistetaan, että ohjelmisto täyttää asiakkaan odotukset ja tarpeet:
  - "Validation: Are building the right product?"

Verifiointi- ja validointitekniikat

- V&V:ssä käytetään enimmäkseen kahta tekniikkaa: tarkastuksia ja testausta.
  - Tarkastukset (software inspections).
    - Tarkastuksessa joukko ihmisiä analysoi ja tarkastaa järjestelmällisesti muodollisessa tarkastustilaisuudessa jonkin projektin tuotoksen tai sen osan etsien siitä puutteita ja virheitä. Tarkastukset ovat staattinen tekniikka. Ne eivät vaadi suorituskelpoista ohjelmaa.
    - Tarkastettavana voi olla esimerkiksi vaatimusdokumentaation, suunnittelun tai ohjelmakoodin osa.

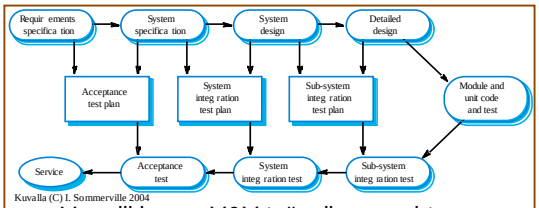
Verifiointi- ja validointitekniikat 2

  - Testaus (software testing).
    - Testauksessa ohjelmistoa tai sen osaa suoritetaan tietyillä testitiedoilla. Tuloksia analysoimalla ja ohjelmiston suoritusta seuraamalla selvitetään, että toiminta on odotettua. Testaus on dynaaminen tekniikka, sillä siihen tarvitaan suorituskelpoinen ohjelma.
- Tarkastuksia voidaan tehdä koko ajan, testausta vasta ohjelmakoodin kanssa.
- Molempia tekniikoita tarvitaan V&V:ssä, sillä ne paljastavat eri tyyppisiä virheitä.

Verifioinnin ja validoinnin tavoite

- V&V:n tavoitteena on varmistaa, että ohjelmisto täyttää sille asetetut tarpeet.
- Ohjelmiston ei tarvitse olla virheetön, eikä se myöskään yleensä ole sitä.
- Ohjelmiston on sen sijaan oltava tarkoitettuun käyttöön "riittävän hyvä".
- Se, miten paljon on "riittävän hyvä", riippuu ohjelmiston käyttöympäristöstä.

V&V:n hallinnan V-malli (V-kirjaimia riittää!)



Kuvalla (C) I. Sommerville 2004

- V-malli kuvaa V&V-työvaiheen suhteen muihin prosessin työvaiheisiin.
- Tarkastuksia voidaan pitää missä tahansa työvaiheessa tai niiden välillä.

6.1. Tarkastukset

- *Tarkastus* (inspection) on muodollinen kokous, jossa tarkastetaan jonkin projektin tuotos, tai osa siitä, ja yritetään löytää tuotoksesta puutteita ja virheitä.
  - Puute = vajaa määritys tai puuttuva toiminta.
  - Virhe = väärin tehty määritys tai ei-toivottu toiminta.

Katselmoinnit

- *Katselmointi* (review) on tekniikka, jossa yksi tai useampi henkilö etsii toisen tuotoksesta puutteita ja virheitä.
  - Tarkastus on muodollisin katselmointitekniikka.
  - Muita katselmointitekniikoita ovat esimerkiksi pariohjelmointi (pair programming), tekninen tarkastus (team review tai technical review) ja koodin läpikäynti (code walkthrough).
- Katselmointeja tehdään kaikissa prosessin työvaiheissa. Mikä tahansa dokumentti voidaan verifioida tai validoida katselmoinnilla.
- Katselmoinnit parantavat tuotteen laatua. Mitä aiemmin puute tai virhe löydetään, sitä helpompi se on korjata.

Tarkastusten luonne

- Tarkastus on muodollinen tilaisuus, johon osallistuu 3-6 henkilöä.
  - Tilaisuudella on tarkka aikataulu.
  - Osallistujat edustavat eri sidosryhmiä asiakkaasta projektiryhmän jäseniin.
  - Tarkastuksessa kootaan ja dokumentoidaan löydetyt puutteet ja virheet.
  - Osallistujat valmistautuvat etukäteen noin kahden tunnin ajan tilaisuuteen tutustumalla tarkastettavaan dokumenttiin.

Tarkastukseen osallistujat

- Osallistujien roolit:
  - Puheenjohtaja (moderator): vastaa tarkastuksen aikataulusta ja ohjelmasta.
  - Sihteeri (scribe): kirjaa ylös löydetyt asiat.
  - Alustaja (reader): kuvaa esitettävän asian.
  - Kirjoittaja (author/owner): edustaa tekijöitä ja selventää tarvittaessa epäselvyyksiä.
  - Tarkastaja (inspector): etsii dokumentista puutteita ja virheitä (kaikkien rooli).

Ennen tarkastusta

- Ennen tarkastustapahtumaa:
  - kaikki tarkastuksessa tarvittavat dokumentit ovat saatavilla,
  - osallistujilla on ollut aikaa tutustua dokumentteihin (valmistautuminen noin 2h),
  - osallistujat ovat saaneet *tarkistuslistat* yleisimmistä puutteista ja virheistä, ja
  - tarkastettava dokumentti on sellaisella tasolla, että siinä ei ole ilmeisiä virheitä.

Tarkastustapahtuma

- Tarkastustapahtuma saa kestää korkeintaan kaksi tuntia. Siinä keskitytään yksinomaan löytämään puutteita ja virheitä.
- Tarkastukseen osallistujat eivät keskustele löydetyistä puutteista. Kun puute on havaittu, sihteeri kirjaa sen ylös ja siirrytään eteenpäin.

Tarkastuksen päätös

- Tarkastuksen lopuksi ryhmä äänestää tuotoksen hyväksymisestä:
  - Hyväksytään sellaisenaan: ei muutoksia.
  - Hyväksytään muutoksin: löydetyt puutteet ja virheet on korjattava, mutta tuotoksesta ei tarvita enää uutta tarkastusta.
  - Hylätään: löydetyt puutteet ja virheet on korjattava. Korjauksen jälkeen tuotoksesta käydään läpi uusi tarkastus.

Tarkastusten kultaiset säännöt

1. Arvioidaan tuotetta, ei tekijää.
2. Suunnitellaan aikataulu ja pidetään siitä kiinni.
3. Ei väittelyä.
4. Ei ratkota löydettyjä ongelmia.
5. Rajoitetaan osallistujien määrä 3-6 henkeen.
6. Valmistaudutaan huolellisesti tarkastukseen.
7. Käytetään tarkistuslistoja sekä valmistautuessa että tarkastuksessa.
8. Varataan riittävästi aikaa ja resursseja.
9. Koulutetaan osallistujat.
10. Pidetään tarkastuksessa kännykät kiinni!

6.2. Testaus

- Testauksella on kaksi tavoitetta:
  - Osoittaa sekä asiakkaille että projektille, että ohjelmisto täyttää sille asetetut vaatimukset. Tämä on *validointitestausta.*
  - Löytää ohjelmistosta puutteita ja virheitä, joiden johdosta ohjelmisto ei toimi, toimii väärin tai ei vastaa sille asetettuja määrittelyjä. Tämä on *määritysten ja syntaksin testausta* (Sommervillella termi on defect testing).

Testauksen piirteitä

- Suunnittelu kannattaa tehdä V-mallin mukaisesti rinnan kehitystyön kanssa.
- Valmistelu on testiympäristön ohjelmointia ja tietojen keruuta, johon pätevät normaalit ohjelmistotekniikan säännöt.
- Suoritus pitää automatisoida, jotta testit voidaan suorittaa helposti uudestaan.
- Analysoinnissa tarvitaan *oraakkeli* (oracle), joka kertoo, oliko testin antama tulos oikea.

Täydellinen testaus mahdotonta

- Täydellinen testaus, missä ohjelma testataan kaikilla mahdollisilla syötteillä, syötekombinaatioilla ja ajoituksilla, ei ole käytännössä mahdollista.
  - Jo hyvin yksinkertaisilla ohjelmilla kaikkien testitapausten suoritus veisi vuosia.
- Tämän johdosta testauksessa valitaan osajoukko kaikista mahdollisista testitapauksista.

Testausvaiheet

- Sommerville jakaa testauksen kahtia:
  - *Komponenttitestauksessa* (Component testing) ohjelmaa testataan osina. Testatut osat kootaan yhteen ja testaan koottuina.
    - Komponenttitestaus kuuluu pääosin ohjelmiston kehittäjille (projektiryhmälle).
  - *Järjestelmätestauksessa* (System testing) testataan järjestelmää kokonaisuutena.
    - Järjestelmätestaus kuuluu pääosin ulkopuoliselle testausryhmälle.

Järjestelmätestaus

- Työvaihe on *liittävä:*
  - Kuhunkin osajärjestelmään liitetään komponentteja ja testataan komponenttien yhteistyö. Tämä on *integrointitestausta.*
  - Komponentit lisätään ja testataan yksi kerrallaan, kunnes kaikki osajärjestelmän komponentit on liitetty ja testattu.
  - Kun kaikki osajärjestelmät on koottu ja testattu, testataan vielä, että osajärjestelmät toimivat yhdessä oikein.

Integrointitestaus

- Integrointitestauksessa testataan valmiiden yksittäin toimivien komponenttien yhteistyö osajärjestelmässä.
- Integrointia voidaan tehdä ylhäältä alas (aloitetaan ohjaavista osista) tai alhaalta ylös (aloitetaan suorittavista osista).
- Voidaan myös integroida molemmista suunnista samaan aikaan.

Rasitustestaus

- Integrointitestauksen jälkeen järjestelmän kriittisiä ominaisuuksia voidaan testata:
  - suorituskykyä,
  - luotettavuutta ja vikasietoisuutta,
  - turvallisuutta.
- Rasitustestauksessa ohjelma viedään äärirajoille ja mielellään vielä niiden yli.

Hyväksymistestaus

- *Hyväksymistestaus* (Acceptance testing, Sommervillella Release testing) tehdään sen jälkeen, kun integrointiestaus on saatu valmiiksi.
- Hyväksymistestauksen tarkoituksena on varmistaa, että tuote on sellaisessa kunnossa, että se voidaan antaa asiakkaalle tuotantokäyttöön.

Yksikkötestaus

- Komponenttitestaus, tai *yksikkötestaus,* (Component testing / Unit testing) tehdään ennen järjestelmätestausta.
  - Komponentin toteuttavat oliot testataan erikseen ja integroidaan sen jälkeen komponentiksi. Tämän jälkeen komponentti testataan vielä jakamattomana kokonaisuutena.
  - Yksikkötestaus ja koodaus ovat sidoksissa: koodatessa tehdään myös yksikkötestejä.

Komponenttien rajapintojen testaus

- Komponenttitestauksen oleellinen osa on rajapintatestaus. Siinä etsitään virheitä, jotka johtuvat rajapintojen virheistä tai vääristä rajapintojen oletuksista.
- Rajapinnat ja rajapintatestaus ovat erittäin tärkeitä, sillä sekä olioiden että komponenttien palvelut määritellään niiden rajapintojen kautta.

Rajapinnat

- Rajapintoja:
  - Parametrien välitysrajapinnat.
  - Jaetun resurssin rajapinnat: muisti ym.
  - Proseduraaliset rajapinnat: palvelukutsut
  - Viestinvälitysrajapinnat: osajärjestelmät.
- Havaittavia virheitä:
  - Rajapintaa käytetään väärin.
  - Rajapinnan toimintaa ei ymmärretä.
  - Rajapinnan käytön ajoitus on väärä.

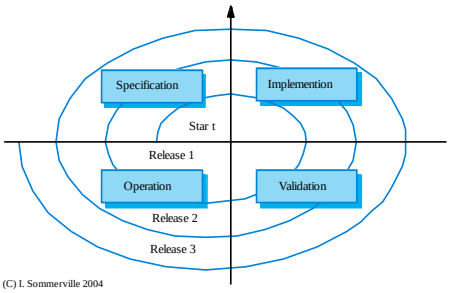
7. Ohjelmiston evoluutio

- Kun kehitettävä ohjelmisto on luovutettu asiakkaalle, version kehitystyö on ohi. Tuote on kuitenkin vasta elinkaarensa alussa.
- Elinkaaren aikana tuotteeseen kohdistuu muutospaineita:
  - Tuotteen vaatimukset muuttuvat.
  - Liiketoiminnan tavoitteet muuttuvat.
  - Uusia korjaamista vaativia virheitä ilmenee.
  - Järjestelmän käyttämä laitteisto muuttuu.
  - Järjestelmän toimintaympäristö muuttuu.
- Muutospaineet pakottavat ohjelmiston muuttumaan: syntyy *ohjelmiston evoluutio* (software evolution).

Muutospaineiden lievitys

- Muutospaineita lievitetään kahdella tavalla:
  - Kun kehitystyötä voidaan jatkaa suoraan, seuraavaan ohjelmistoversioon otetaan huomioon evoluution aiheuttaneet seikat.
  - Kun kehitystyötä ei voida jatkaa suoraan, tarvitaan *ohjelmiston ylläpitoa* (software maintenance). Tällöin vanhaa versiota muokataan vastaamaan uusia tarpeita.
- Molemmissa tapauksissa prosessin tuloksena saadaan uusi *ohjelmiston julkaisu* (software release), jossa muutospaineet on huomioitu.

Kehitystyön ja evoluution spiraali



Kuvalla (C) I. Sommerville 2004

Evoluution kustannus

- Evoluutioon vastaaminen on kallista. Jopa 90% ohjelmiston kehityskustannuksista tulee käyttöönoton jälkeen.
- Mitä dynaamisempaan ympäristöön ohjelmisto on tarkoitettu, sitä suuremmat ovat sen evoluution aiheuttamat kustannukset:
  - Ohjelmisto, joka toteuttaa selkeästi määritellyn algoritmin, ei juurikaan vaadi ylläpitoa (esim. pesukoneen pesuohjelma).
  - Ohjelmisto, joka toteuttaa jatkuvassa muutoksessa olevan ympäristön jonkin tarpeen, vaatii jatkuvan ylläpidon ja tiheän julkaisuvälin (esim. internet-ohjelmistot).

7.1. Evoluution dynamiikka

- Ohjelmiston evoluutioon vastaaminen on muutoksen hallintaa. Työvaiheesta Lehman ja Belady kehittivät joukon lakeja, joita kutsutaan Lehmanin laeiksi.
- Lehmanin lait selittävät ohjelmiston muutospaineita ja niiden vaikutuksia ohjelmistoon ja sidosryhmiin.
- Lakeja ei ole todistettu, joten ne ovat oikeastaan teoreemoja (otaksumia).

Lehmanin 1. laki

- *Jatkuvan muutoksen laki* (Continuing change):
  - Normaalissa käytössä olevan ohjelmiston täytyy muuttua, tai ajan kanssa siitä saatava hyöty vähenee.
- Lain mukaan ohjelmiston evoluutiota ei voida välttää. Ohjelmistot tulevat aina tarvitsemaan muutoksia elinkaarensa aikana.

Lehmanin 2. laki

- *Kasvavan monimutkaisuuden laki* (Increasing complexity):
  - Muutoksen kautta ohjelmiston rakenne heikkenee. Jos rakenteen parantamiseen ei varata resursseja, ohjelmistosta tulee vaikeammin muokattava.
- Laki on eräänlainen ohjelmiston entropian laki. Järjestelmän pitämiseen siistinä tarvitaan resursseja (energiaa).

Lehmanin 3. laki

- *Ison ohjelmiston evoluution laki* (Large program evolution):
  - Ohjelmiston evoluutio ohjaa itse itseään. Kehitystyön attribuutit, kuten ohjelmiston koko, julkaisutiheys ja raportoitujen virheiden määrä, ovat eri julkaisuissa likimain yhtenevät.
- Tämä laki on isojen ohjelmistojen ylläpidon ajankäytön laki. Sen mukaan ylläpito määräytyy kehitystyövaiheessa.

Lehmanin 4. laki

- *Organisaation pysyvyyden laki* (Organisational stability):
  - Ohjelmiston elinkaaren ajan ohjelmiston kehitystyön tehokkuus pysyy lähes vakiona.
- Lehmanin mukaan useimmat isoista ohjelmistoprojekteista saavuttavat *optimitason*, jonka jälkeen resurssien tai henkilökunnan lisääminen ei juurikaan näy evoluutiossa.

Lehmanin 5. laki

- *Tuttuuden laki* (Conservation of familiarity):
  - Peräkkäisten julkaisujen sisältöjen (muutokset lisäykset, poistot) suuruus on likimain vakio.
- Muutoksessa ei voida keskittyä vain lisäämään uusia ominaisuuksia. Lisäävän julkaisun jälkeen tulee välttämättä julkaisu, jossa korjataan edellisen julkaisun esittelemiä virheitä.

Lehmanin loput lait

- Viimeiset Lehmanin lait käsittelevät sitä, miten asiakkaat näkevät ohjelmiston evoluution. Ne voidaan tiivistää seuraavaan sanomaan:
  - mitä vähemmän ohjelmiston evoluutioon varaudutaan, sitä tyytymättömämmiksi ohjelmiston käyttäjät tulevat ajan kanssa.
- Lehmanin lait kuvaavat erityisesti isojen räätälöityjen ohjelmistojen evoluutiota.

7.2. Ohjelmiston ylläpito

- Ylläpito on yleinen termi prosessille, jonka avulla jo luovutettua ohjelmistoa muutetaan sen elinkaaren aikana.
- Ylläpidosta on yleensä vastuussa tuotetta käyttävän yrityksen sisäinen ylläpitoryhmä.
- Ylläpidossa tehdään lähinnä pieniä tai keskisuuria muutoksia. Arkkitehtuuri ei yleensä koe merkittäviä muutoksia.

Ylläpitotyypit

- Ylläpitoa on kolmea eri tyyppiä:
  1. Ohjelmistovirheiden korjaus (korjaava ylläpito).
  2. Ohjelmiston sovittaminen uuteen ympäristöön (sovittava ylläpito).
  3. Ohjelmiston toimintojen muokkaus tai uusien toimintojen lisäys (täydentävä ylläpito).
- Usein ohjelmiston muutos vaatii kaikkia kolmea ylläpitotyyppiä.

Ylläpitotyyppien osuudet

- Tutkimusten mukaan ylläpitotyyppien osuudet ylläpidosta ovat seuraavaa suuruusluokkaa:
  - Ohjelmistovirheiden korjaus: 17%.
  - Ohjelmiston sovittaminen uuteen ympäristöön: 18%.
  - Ohjelmiston toimintojen muokkaus tai uusien toimintojen lisäys: 65%.

7.3. Evoluutioprosessi

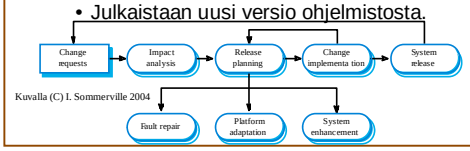
- Ohjelmiston evoluutioprosessi vaihtelee huomattavasti. Prosessiin vaikuttaa
  - ylläpidettävän tuotteen tyyppi,
  - kehitystyössä käytetty prosessi ja
  - ylläpitoon osallistuvien henkilöiden ammattitaito.
- Evoluutioprosessi voi vaihdella täysin epämuodollisesta prosessista hyvin tarkasti ohjattuun prosessiin.

Evoluutioprosessin tehtävät

- Evoluutioprosessiin liittyy
  - *vaikutusanalyysi* (impact analysis) (Sommervillella myös muutosanalyysi, change analysis)
    - Selvitetään, miten paljon muutos vaikuttaa ohjelmiston rakenteeseen ja mitä sen toteutus tulisi maksamaan.
  - *julkaisusuunnittelu* (release planning)
    - Päätetään, mitä muutoksia seuraavaan ohjelmistoversioon tehdään. Muutokset aiheuttavat jonkun kolmesta ylläpitotyypistä.

Evoluutioprosessin tehtävät 2

- Evoluutioprosessin tehtävät jatkuvat:
  - *päivityksen toteutus* (change implementation)
    - Toteutetaan päätetyt muutokset
  - *version julkaisu* (system release)
    - Julkaistaan uusi versio ohjelmistosta.



Kuvalla (C) I. Sommerville 2004

Evoluutio- ja kehitystyöprosessien erot

- Evoluutioprosessi eroaa kehitystyön prosessista seuraavissa kohdissa:
  - Evoluutioprosessi ei ala tyhjästä.
    - Ensimmäisenä vaiheena on ymmärtää jatkokehitettävän ohjelmiston rakenne ja tehdyt ratkaisut.
  - Muutokset tulee tehdä sellaisiksi, että ne eivät riko olemassaolevia ratkaisuja.
- Vaihtoehtona on *ohjelmiston uudistaminen* (software re-engineering).

Ohjelmiston uudistaminen

- Jos näyttää siltä, että ohjelmiston ylläpidon kustannukset alkavat kohota taivaisiin eikä silti voida taata laadukasta lopputulosta, voi olla aika uudistaa ohjelmisto.
- Ohjelmiston uudistamisessa osa ohjelmistosta toteutetaan ja dokumentoidaan uudestaan sellaiseksi, että sen ylläpito helpottuu.

Uudistamisen merkitys

- Ohjelmiston uudistamisessa ohjelmiston toiminnallisuus ei muutu. Ainoastaan toiminnallisuuden toteuttavat ratkaisut muuttuvat. Loppukäyttäjälle ohjelmisto näyttää ja (yleensä) tuntuu samalta.
  - Ohjelmisto saattaa myös tehostua uudistamisessa, jolloin ohjelmiston rajoitukset ja reunaehdot lievenevät. Sen sijaan rajoitukset ja reunaehdot eivät saa koventua.

Uudistamisen ja kehitystyön ero

- Ohjelmiston uudistaminen eroaa kehitystyöstä siinä, että uudistettavasta ohjelmistosta tiedetään jo vaatimukset. Näin vaatimusmäärittely jää pois.
- Lisäksi vanhan järjestelmän suunnitelmasta voidaan käyttää osia, jolloin suunnittelu helpottuu.
  - Esimerkiksi ohjelmiston yleisarkkitehtuuri ei yleensä muutu uudistuksessa.

Uudistusprosessi

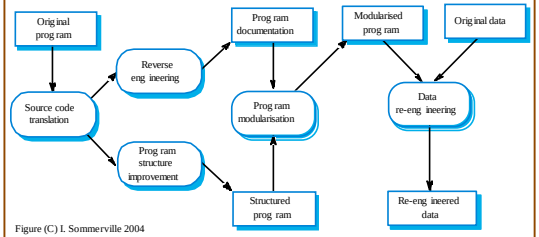


Figure (C) I. Sommerville 2004

7.4. Perinnejärjestelmät

- *Perinnejärjestelmä* (legacy system) on vanhentunut ohjelmisto, jota käytetään edelleen, mutta jonka suunnittelusta ja toteutuksesta ei enää tiedetä juuri mitään.
- Perinnejärjestelmä on tavallaan mennyt yli elinkaarestaan. Järjestelmää käytetään vaihtelevista syistä kuitenkin edelleen.

Perinnejärjestelmän evoluutiostrategiat

- Jossain vaiheessa perinnejärjestelmää käyttävä yritys joutuu päättämään, mitä järjestelmälle tehdään:
  - Järjestelmä voidaan hylätä.
  - Järjestelmälle voidaan antaa tekohengitystä ylläpidon keinoin.
  - Järjestelmä voidaan uudistaa.
  - Järjestelmä voidaan korvata uudella vastaavalla järjestelmällä.

8. Ohjelmistoprojektin hallinta

- Ohjelmistoprojektin hallinta on oleellinen osa ohjelmistotuotantoa. Ilman hallintaa ei synny hyvää tulosta.
- Projektinhallinta on hankalaa, koska
  - tuotoksen kehittyminen ei näy suoraan: työntekijät saattavat näyttää kiireisiltä, vaan tulosta ei synny.
  - erilaisia ohjelmistotuotantoprosesseja on lukematon määrä, ja tietystä prosessista on vaikea sanoa, miten hyvään lopputulokseen sitä käyttämällä päädytään.
  - isot ohjelmistoprojektit ovat ainutkertaisia, joten aiempien projektien kokemukset eivät aina auta.

Projektinhallinnan tehtäviä

- Projektinhallinta on yleensä projektipäällikön tehtävä. Työnkuvaan kuuluu:
  - tarjouksen laadinta asiakkaalle,
  - projektin suunnittelu ja aikataulutus,
  - projektin kustannusten arviointi,
  - projektin seuranta ja katselmointi,
  - henkilökunnan valinta ja arviointi sekä
  - raporttien kirjoittaminen ja projektin esittely.

Projektin seurannan vaiheet

1. Selvitetään projektin lähtötilanne (aika, henkilöt, budjetti, jne.).
2. Asetetaan projektin parametrit (tuotteen rakenne ja koko, iteraatiot, tarkastukset jne.)
3. Määritellään projektin tarkistuspisteet ja tuotokset.
4. Toistetaan vaiheita 5-12, kunnes projekti päättyy.
5. Tehdään aikataulu.
6. Sijoitetaan tehtävät ja henkilöt aikatauluun.
7. Projektiryhmä toimii suunnitelman mukaan.
8. Tarkistetaan eteneminen.
9. Muutetaan tarvittaessa projektin parametreja.
10. Päivitetään aikataulu vastaamaan nykytilaa.
11. Neuvotellaan tarvittaessa päivityksistä projektin rajoihin ja tuotoksiin.
12. Jos ongelmia ilmenee, tarkastetaan prosessi ja tarvittaessa korjataan projektisuunnitelmaa.

Projektisuunnitelma

- Projektin hallinnan keskeinen työväline on *projektisuunnitelma* (project plan).
  - Se laaditaan ennen projektin työvaiheita.
  - Sitä päivitetään ja täydennetään projektin aikana.
- Projektisuunnitelman tarkoitus on:
  - auttaa seuraamaan projektin etenemistä,
  - antaa mahdollisuudet saada projekti valmiiksi aikataulussa ja
  - antaa keinot huomata aikataulusta lipsumiset mahdollisimman pian.
- Tämä on projektin tärkein dokumentti!

Projektisuunnitelman sisältö

- Projektisuunnitelmassa listataan ainakin:
  - projektin jäsenet,
  - projektin työvaiheet,
  - projektin tehtävät ja työnjako,
  - projektin aikataulu ja
  - projektin riskit ja niiden vastatoimet.
- Lisäksi suunnitelmassa voi olla mukana projektiin liittyviä ohjeistoja, kuten seuranta- ja raportointitavat.

8.1. Projektin aikataulutus

- Projektin aikataulun laatiminen on yksi projektinhallinan vaikeimpia tehtäviä. Siinä pitää käytännössä ennustaa tulevaisuutta, sillä aiempien projektien aikataulut eivät sellaisenaan sovi uuteen projektiin.
- Mitä haastavampi projekti on kyseessä, sitä luultavammin aikataulu on liian optimistinen.
- Aikataulutuksessa projekti jaetaan tehtäviin, joiden koko on mahdollista arvioida.

Tarkistuspisteet ja tuotokset

- Projektissa on *tarkistuspisteitä* (milestones)
  - Tarkistuspisteessä selvitetään, ollaanko aikataulussa ja budjetissa.
  - Tarkistuspiste on yleensä jonkin laajemman työvaiheen lopussa.
- ja *tuotoksia* (deliverables).
  - Tuotos on jokin projektista saatava asiakkaan kannalta merkittävä tulos.

Aikataulun laatiminen

1. Jaetaan projekti tehtäviin.
2. Arvioidaan kunkin tehtävän kesto.
3. Selvitetään tehtävien riippuvuudet.
4. Selvitetään, mihin tehtävään kukin tuotos tai tarkistuspiste liittyy.
5. Yhdistetään toiminnot *toimintoverkoksi* (activity network).

Toimintoverkko

- Toimintoverkko kuvaa tehtävien järjestyksen, keskinäiset riippuvuudet ja aikataulun.
- Yleensä projektissa on eräitä tarkistuspisteitä, jossa kaikki käynnissä olevat tehtävät yhtyvät.
- Samalle projektille voidaan tehdä useita erilaisia toimintoverkkoja.
  - Yleensä projektipäällikkö tekee toimintoverkon.
  - Lopputulokseen vaikuttaa mm. varautuminen riskeihin.

Kriittinen polku

- Jokaisella tehtävällä on
  - aikaisin ajankohta, jolloin se voi alkaa,
  - myöhäisin ajankohta, jolloin sen täytyy alkaa,
  - aikaisin mahdollinen lopetusaika,
  - myöhäisin mahdollinen lopetusaika ja
  - joustovara, jonka puitteissa työ ei myöhästy.
- Kaikilla tehtävillä ei ole joustovaraa. Tällaisista tehtävistä muodostuu *kriittinen polku* (critical path).
- Jos kriittisellä polulla oleva tehtävä myöhästyy, koko projekti myöhästyy.

Ajoituskaavio

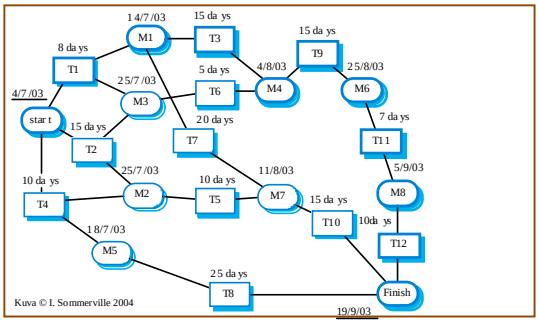
- Yleensä toimintoverkko kuvataan kompaktina *ajoituskaaviona* (activity bar chart).
- Ajoituskaavio sisältää:
  - kaikki tehtävät kestoaikoineen,
  - tarkistuspisteet,
  - perustehtävien rinnakkaisuuden ja
  - perustehtävien ja tarkistuspisteiden päättymisaikojen joustovaran.

Aikatauluesimerkki

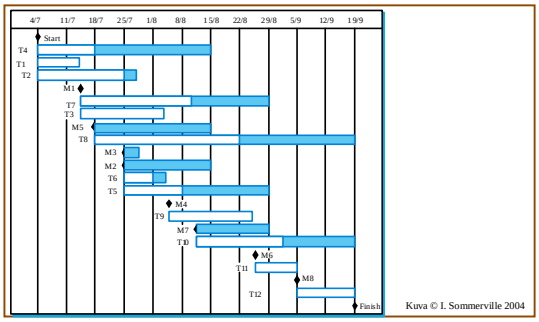
Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

Taulukko © I. Sommerville 2004

Esimerkin toimintoverkko ja kriittinen polku



Esimerkin ajoituskaavio



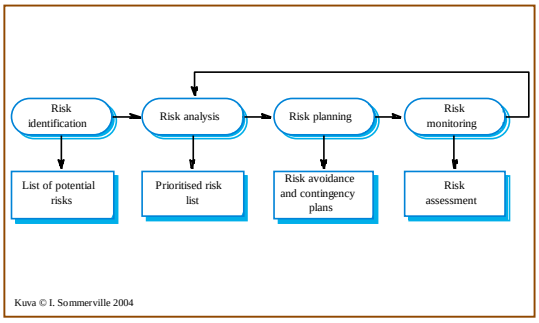
8.2. Riskienhallinta

- Projektin valmistuminen pyritään takaamaan myös tilanteissa, joissa tapahtuu jotakin ei-toivottua: jokin *riski* (risk) toteutuu.
- Riskienhallinnassa
  - tunnistetaan onnistumista uhkaavat riskit,
  - analysoidaan tunnistetut riskit,
  - suunnitellaan vastatoimet ja
  - seurataan ja päivitetään riskejä.

Mikä on riski?

- Riski on tapahtuma, joka
  - on mahdollinen (todennäköisyys >0 mutta <1)
    - Todennäköisyys = 0: mahdoton tapahtuma.
    - Todennäköisyys = 1: projektin rajoite.
  - toteutuessaan vahingoittaa projektia.
- Riski voi olla
  - Projektikohtainen:
    - vaikuttaa aikatauluun tai käytössä oleviin resursseihin.
  - Tuotekohtainen:
    - vaikuttaa kehitettävän tuotteen laatuun.
  - Yrityskohtainen:
    - vaikuttaa tekijä- tai asiakasorganisaatioon.

Riskienhallintaprosessi



Riskien tunnistus

- Pyritään löytämään kaikki riskit, jotka voivat vaikuttaa projektin onnistumiseen.
  - Käytännössä unohdetaan kovin epätodennäköiset ja hyvin merkityksettömät riskit.
- Riskit voivat liittyä ainakin
  - käytettyyn teknologiaan,
  - henkilökuntaan,
  - organisaatioon,
  - käytettyihin työkaluihin,
  - vaatimuksiin tai
  - kustannusten ja aikataulun arviointiin.

Riskien analysointi

- Mietitään kunkin riskin todennäköisyys ja vakavuus:
  - todennäköisyys = miten varmasti riski toteutuu
    - yleensä luokitteluna (esim. viisi luokkaa vähäisestä erittäin todennäköiseen), voi olla myös lukuina.
  - vakavuus = miten merkittävä riski on projektille
    - luokitteluna (esim. tuhoisa, vakava, siedettävä, vähäpätöinen, merkityksetön)
- Päätetään, miten riskeihin varaudutaan
  - Mitkä riskit otetaan huomioon suunnitelmissa
  - Yleensä on syytä ottaa huomioon ainakin kaikki tuhoisat ja kohtalaisen todennäköiset vakavat riskit

Riskien vastatoimet

- Jokaiselle valitulle riskille suunnitellaan vastatoimet.
- Vastatoimet voivat olla
  - riskin välttämistä:
    - pienennetään toteutumisen todennäköisyyttä.
  - vaikutusten minimointia:
    - vähennetään toteutumisen haittavaikutuksia.
  - jatkosuunnitelmia:
    - suunnitellaan riskin toteutuessa seurattavat toimintatavat.

Riskien seuranta

- Riskien toteutumista seurataan koko projektin elinkaaren ajan.
  - Riskin toteutuessa ryhdytään suunnitelmien mukaisiin toimenpiteisiin
- Projektisuunnitelmaa voidaan joutua päivittämään:
  - Projektin kuluessa ilmenee uusia riskejä.
  - Jo tunnistetun riskin todennäköisyys tai vakavuus muuttuu (mutta ei aina pahempaan suuntaan).
  - Jokin sellainen riski toteutuu, jota varten ei ole suunniteltu vastatoimia.

8.3. Projektin kustannusarviot

- Projektin kustannusarvio ja projektin aikataulu tehdään yleensä yhdessä.
- Projektin suurin kustannus tulee tehdystä työstä, joten käytännössä kustannus korreloi vahvasti projektissa käytettyjen työtuntien kanssa.
- Työkustannuksiin lasketaan myös esim. hallinto-, siivous-, tietoliikenne- ja kirjastopalvelut sekä työtilat.

Kustannukset ja tehtävien ulkoistaminen

- Ohjelmistojen teko on kallista. Palkat ja sivukulut yhdessä kaksinkertaistavat helposti työntekijän kuukausipalkan kustannuksen.
- Kustannusten rajoittamiseksi moni ohjelmistoyritys ulkoistaa kokonaisia työvaiheita, kuten vaatimusmäärittelyn tai testauksen.

Tuottavuus

- Teollisuustuotannossa *tuottavuutta* (productivity) mitataan jakamalla tehtyjen tuotteiden määrä niiden tekemiseen vaadituilla työtunneilla.
- Ohjelmistotuotannossa vastaava mittaus ei kertoisi mitään. Siksi tarvitaan muita tuottavuusmittareita.
- Ohjelmistoprojektin tuottavuutta arvioidaan kahdella tavalla:
  1. Tuotteen kokoon perustuvilla mittareilla
  2. Tuotteen toiminnallisuuteen perustuvilla mittareilla.

Kokoon perustuva arviointi

- Tavallisin koon mitta on koodin määrä.
  - Ohjelmatuote koostuu toimivasta koodista.
  - Laaja ohjelmisto sisältää paljon koodia.
  - Koodin määrä on helposti mitattavissa.
- Toisaalta pätee:
  - Koodin laatiminen on vain osa työstä.
  - Koodirivin ilmaisuvoima riippuu käytetystä ohjelmointikielestä.
  - Koodirivejä ei voida laskea alkuvaiheessa.

Toiminnallisuuteen perustuva arviointi

- Kokoon perustuvan arvioinnin sijaan voidaan arvioida ohjelmiston toiminnallisuutta.
- Tällöin lasketaan, mitä kaikkea ohjelmiston halutaan tekevän ja saadaan tuloksena ohjelmointikielestä riippumaton mitta ohjelmalle.
- Toiminnallisuuden laskun huono puoli on, että tulos ei sellaisenaan kerro mitään. Se on vain luku.

Toimintopisteanalyysi

- Tunnetuin toiminnallisuuden mittari on *toimintopisteanalyysi (*function point analysis). Siinä ohjelmistoa arvioidaan laskemalla siitä
  - ulkoisten syötteiden ja tulosteiden lukumäärä,
  - käyttäjän toiminnan määrä (interaktio),
  - ulkoiset liittymät ja
  - tiedostot.

Oliopisteanalyysi

- Toimintopisteanalyysia uudempi tekniikka on *oliopisteanalyysi* (object point analysis). Siinä laskenta perustuu suunnitelmasta löydettyihin toiminnallisuutta kuvaaviin alkioihin.
- Tuloksena saadaan paljas luku, joka voidaan normalisoida kertomaan tuotteesta.
- OP = Σ [(alkioiden lukumäärä) * painokerroin]

Oliopisteiden laskenta

- Oliopisteitä lasketaan kolmesta elementistä:
  - Näytöistä (number of separate screens),
  - Tuotetettavista raporteista (number of reports produced),
  - Toteutettavista moduuleista tai komponenteista (number of modules to be developed).
- Kukin elementti arvioidaan erikseen.

Elementtien arviointi

- Elementin pisteytykseen vaikuttaa se, miten hankalaksi elementin toteutus katsotaan (painokerroin):
  - Yksinkertainen näyttö: 1op
  - Keskivaikea näyttö: 2op
  - Vaikea näyttö: 3op
  - Yksinkertainen raportti: 2op
  - Keskivaikea raportti: 5op
  - Vaikea raportti: 8op
  - Moduuli/komponentti: 10op

Algoritmiset kustannusmallit

- Kustannusmallit ovat muotoa
  - Työmäärä = A * koko^B * M, missä
    - A = tehtävästä ohjelmistosta riippuva vakio,
    - koko = joko rivejä tai funktio-/oliopisteitä,
    - B = vakio väliltä 1 – 1,5
    - M = prosessista, tuotteesta ja kehitystyöstä riippuva kerroin.
- Mallit perustuvat tilastolliseen analyysiin päättyneistä projekteista.

COCOMO II

- *COCOMO* (COnstructive COst MOdel) on ohjelmatuotteen kokoon perustuva kustannusten arviointimalli, joka esiteltiin 1981.
  - Vesiputousmallin mukainen prosessi
  - Proseduraaliset ohjelmointikielet
  - Ei uudelleenkäyttöä
- Nykyisin käytetään COCOMOn seuraajaa 1995 esiteltyä COCOMO II:ta
  - Modernimpi lähestymistapa prosessiin
  - Oliot, uudelleenkäyttö
  - Tarkentuvat osamallit

COCOMO II:n osamallit

- Application composition model
  - käytetään, kun ohjelmisto tehdään suurelta osalta valmiista komponenteista.
- Early design model
  - käytetään, kun vaatimusmäärittely on tehty, mutta suunnittelu ei ole alkanut.
- Reuse model
  - käytetään laskettaessa uudelleenkäytettävien komponenttien vaatimaa työmäärää.
- Post-architecture model
  - käytetään kun järjestelmäarkkitehtuuri on valmis ja järjestelmästä on yksityiskohtaista tietoa.

Application-composition model

- Otetaan esimerkkinä COCOMOn malleista Application-composition model. Sen työmääräarvio perustuu oliopisteisiin:
  - PM = (NAP x (1- %reuse/100)) / PROD.
    - PM = person-months: henkilötyökuukausiarvio
    - NAP = oliopisteiden lukumäärä (COCOMO II puhuu sovelluspisteistä (application points)
    - %reuse = paljonko koodista on valmiina
    - PROD = tuottavuuskerroin (seuraava kalvo)

Tuottavuuskerroin

- Application-composition –mallin arvo riippuu myös seuraavista tekijöistä:
  - miten kokenut tiimi tekee tuotetta ja
  - miten hyviä kehitystyökaluja on käytössä.
- Näille on mallissa annettu kertoimet:

Tiimin kokemus	Hyvin matala	Matala	Keskiverto	Korkea	Hyvin korkea
Kehitystyö-kalujen taso	Hyvin matala	Matala	Keskiverto	Korkea	Hyvin korkea
	4	7	13	25	50

Application-composition model 2

- Lopullinen tuottavuuskerroin, PROD, saadaan laskemalla keskiarvo tiimin kokemuksen ja kehitystyökalujen hyvyyden kertoimista:
  - PROD = (tiimikerroin + työkalukerroin)/2
- Malli on tarkoitettu komponenteista koottavien ohjelmien arviointiin, mutta sitä voidaan kyllä käyttää arvioitaessa karkeasti mitä tahansa ohjelmistoa.

9. Laadunhallinta

- Viimeisen 20 vuoden aikana ohjelmistojen laatu on parantunut huomattavasti. Tähän on monia syitä:
  - modernit prosessimallit opettavat entistä joustavammat ja silti hallitut työtavat,
  - ohjelmistotuotannon tekniikat hallitaan useimmiten aika hyvin, ja vielä
  - ohjelmistojen *laadunhallinta* (quality management) on otettu merkittäväksi osaksi ohjelmistojen kehitystyötä.

Laadunhallinnan tehtävät

- Laadunhallinta
  - varmistaa, että tehtävät ohjelmistot täyttävät vaaditun laatutason,
  - määrittelee tarvittavat laatustandardit ja laadun saavuttamiseksi tarvittavat menetelmät,
  - varmistaa, että määriteltyjä standardeja ja menetelmiä myös seurataan käytännössä,
  - pyrkii kehittämään *laatukulttuurin* (quality culture), jossa laatu on kaikkien vastuulla.

Mitä ohjelmiston laatu tarkoittaa?

- Lyhyesti ohjelmiston laatu tarkoittaa ohjelmiston vastaavuutta määrittelynsä kanssa. Tämä on kuitenkin ongelmallista:
  - Asiakkaan tarkoittama laatu (käytettävyys, luotettavuus, tehokkuus ym.) ei välttämättä ole sama kuin kehitystiimin tarkoittama laatu (ylläpidettävyys, uudelleenkäytettävyys jne.)
  - Kaikkia laadun piirteitä ei osata kuvata yksiselitteisesti (esim. ylläpidettävyys)
  - Vaikka ohjelmisto täyttää määrittelynsä, se ei välttämättä täytä asiakkaan sille asettamia tarpeita. Onko ohjelmisto tällöin laadukas?

Laadun ongelmia

- Edellä luetellut laadun määritelmät voidaan varmentaa vasta valmiista tuotteesta ja asiakkaan palautteesta. Tällöin laatu voidaan *varmentaa* mutta ei *varmistaa.*
- Onnistunut laadunhallinta takaa, että tehtävät tuotteet ovat laadukkaita sekä asiakkaan että kehitystiimin kannalta.

Laatukomponentit

- Parhaiten laatu määritellään *laatukomponenteilla* (quality attributes)
- Laadukas tuote on sellainen, joka täyttää *riittävän hyvin* sille määritellyt laatukomponentit.
- Laatukomponenteilla on läheinen yhteys ei-toiminnallisiin vaatimuksiin.
  - Siksi ei-toiminnallisia vaatimuksia kutsutaan laatuvaatimuksiksi.

Ohjelmistojen laatukomponentteja

- Käyttöturvallisuus (safety)
- Tietoturvallisuus (security)
- Luotettavuus (reliability)
- Tehokkuus (efficiency)
- Käytettävyys (usability)
- Opittavuus (learnability)
- Joustavuus (resilience)
- Vakaus (robustness)
- Muutettavuus (adaptability)
- Virheettömyys (faultlessness)
- Monimutkaisuus (complexity)
- Modulaarisuus (modularity)
- Testattavuus (testability)
- Siirrettävyys (portability)
- Ylläpidettävyys (maintainability)
- Uuskäyttöisyys (reusability)

Laatuvaatimukset

- Ohjelmiston tyyppi vaikuttaa siihen, mitkä laadun komponentit pitää huomioida kehitystyössä ja miten. Tuotteella on *laatuvaatimuksia.*
- Laatuvaatimukset ovat ei-toiminnallisia vaatimuksia, jotka yleensä liittyvät koko järjestelmään.
  - Joskus laatuvaatimus voi liittyä yksittäiseen toimintoon: esimerkiksi tietyn toiminnon vasteaika on laatuvaatimus.

Laatukomponenttien valinta

- Ohjelmiston laatukomponentit ovat usein keskenään ristiriitaisia.
  - Esimerkiksi tietoturvallisuus voi olla ristiriidassa käytettävyyden kanssa.
- Kehitystyössä on päätettävä, mitkä laatukomponenteista ovat tärkeimmät ja mistä voidaan tinkiä.
  - Esimerkiksi useimmiten varmaankin tingittäisiin käytettävyydestä tietoturvallisuuden hyväksi.

Laadunhallinnan tehtävät

- Mitä isompi organisaatio ja mitä isommat järjestelmät, sitä tärkeämpää on tarkasti ohjattu laadunhallinta.
- Laadunhallinnalla on kolme tehtävää:
  1. *Laadunvarmistus* (quality assurance)
     - Vastataan organisaatiotason laatustandardeista ja –menetelmistä
     - Pidetään yllä *laatukäsikirjaa,* joka sisältää yrityksessä käytettävät laatustandardit ja laadunvarmistusmenetelmät.

Laadunhallinnan tehtävät 2

2. *Laadun suunnittelu* (quality planning)
   - Selvitetään ja päätetään, mitkä laatuattribuutit ovat tärkeitä aloitettavassa projektissa.
   - Valitaan tiettyyn projektiin parhaiten sopivat laatukäsikirjan standardit ja menetelmät ja täydennetään niitä tarvittaessa.
3. *Laadunvalvonta* (quality control)
   - Varmistetaan, että projekteissa seurataan sovittuja laatustandardeja ja –menetelmiä
- Pienissä projekteissa ei ehkä tarvita näin muodollista laadunhallintaa.

Laadunhallintaryhmä

- Laadunhallinta kannattaa antaa ohjelmistoprojekteista riippumattomalle *laadunhallintaryhmälle* (quality management team).
  - Laadunhallintaryhmä seuraa projekteja ja raportoi projektien johtoryhmille.
- Laadunhallintaryhmän avulla projekteihin saadaan objektiivinen kehitystyön ulkopuolinen näkökulma.

Standardit

- Standardit ovat sopimuksia, joiden avulla pyritään tasalaatuisuuteen.
- Standardi voi koskea tuotetta tai prosessia:
  - tuotetta koskevat standardit sisältävät tuotteen osien rakenteet ja esitystavat.
  - prosessia koskevat standardit sisältävät prosessin työvaiheet ja niissä käytettävät työtavat.

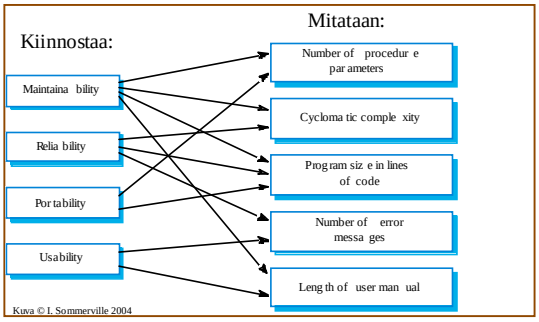
Mittaus

- *Mittaus* (measuring) tarkoittaa, että projektin kestäessä ja päättyessä sekä tuotteesta että prosessista lasketaan sitä kuvaavia arvoja.
  - Projektin aikana tehtävää mittausta käytetään seurattaessa ja ohjattaessa projektin etenemistä ja tuotteen laatua.
  - Projektin lopussa tehtävää mittausta käytetään historiatietona tulevia projekteja varten.

Kiinnostavat ja mitattavat suureet

- Mitattavat suureet eivät yleensä ole sellaisenaan kiinnostavia. Saadut arvot pitää tulkita.
- Tulkinnassa mitattavasta suureesta johdetaan tieto, joka kertoo jostain kiinnostavasta suureesta:
  - esim. mitataan vaatimusmäärittelyssä saatujen näyttöjen määrä è arvioidaan käytettävyyttä.

Kiinnostavat ja mitattavat suureet



Kuva © I. Sommerville 2004

Tuotteen mitat

- Staattiset mitat:
  - Kerätään mittaamalla projektin tuotoksia:
    - Suunnitelmia,
    - Koodia,
    - Dokumentteja.
  - Kerättävissä projektin alusta alkaen.
- Dynaamiset mitat:
  - Kerätään mittaamalla toimivaa ohjelmaa.
  - Mitan arvo riippuu myös siitä, miten ohjelmaa käytetään:
    - Eri toimintojen käyttö,
    - Syötteet.
  - Kerättävissä vasta, kun on jotain, joka toimii.

Mitä mitat kuvaavat?

- Staattiset mitat liittyvät tuotteen rakenteellisiin ominaisuuksiin. Niillä voi olla välillinen yhteys laatuominaisuuksiin
  - Esim. laaja vaatimusmäärittely è vaikeasti ylläpidettävä ohjelmisto.
- Dynaamiset mitat liittyvät tuotteen käyttäytymiseen. Niillä on yleensä suora yhteys laatuominaisuuksiin.
  - Esim. suoritusaika, toipuminen jne.

Prosessin mitat

- Prosessin mittoja käytetään prosessin seurantaan ja parantamiseen:
  - aikamitat, kuten tiettyyn työvaiheeseen kulunut aika.
  - resurssimitat, kuten käytettyjen henkilötyöpäivien määrä, koneaika.
  - tapahtumamitat, kuten testauksessa löytyneiden vikojen lukumäärä, muutospyyntöjen lukumäärä.

Mittatulosten analysointi

- Mitattavaan suureeseen vaikuttaa yleensä monta samanaikaista tekijää.
- Tulosten tulkintaan liittyy epävarmuutta.
  - Esimerkiksi jos testauksessa löytyi vain pieni määrä vikoja, syy voi olla hyvässä koodauksessa, huonossa testauksessa, taitavassa suunnittelussa, huolellisissa tarkastuksissa, runsaassa uudelleenkäytössä jne.

10. Yhteenveto

- Ohjelmistotuotanto jakaantuu selkeästi seuraaviin osa-alueisiin:
  - projektin hallintaan: kuka tekee, milloin tehdään, mitä huomioidaan tehdessä,
  - ongelman määrittelyyn: mitä tehdään,
  - ohjelmistosuunnitteluun: miten tehdään,
  - toteutukseen: tehdään se,
  - verifiointiin ja validointiin: varmistetaan ja
  - ylläpitoon: pidetään huolta ohjelmistosta.

Osa-alueet ja perustehtävät

- Osa-alueet muodostavat myös ohjelmistotuotantoprosessin perustehtävät.
- Tehtäviä ei välttämättä suoriteta tässä järjestyksessä eikä edes peräkkäin. Ohjelmistotuotannossa on tilaa erilaisille prosesseille ja prosessimalleille. Kuitenkin nämä osa-alueet ovat aina läsnä kaikissa malleissa.

Projektin suunnittelu - älä aliarvioi

- Vaikka jokainen osa-alue on tärkeä, niistä kaikkein tärkeimmäksi nostaisin projektin hallinnan.
- Suunnittelu ei ala tuotteesta, vaan prosessista. Ilman tarkkaa suunnitelmaa ei synny laadukasta tuotetta.
  - Tämä osa-alue pettää pahiten varsinkin kokemattomien tekijöiden projekteissa.

Realistinen aikataulu

- Ennen ensimmäistäkään vaatimusmäärittelyn työvaihetta projektillanne on oltava selvä ja realistinen aikataulu, jota seurataan.
- Kun aikataulu on realistinen, työskentely on miellyttävää, tulokset ovat hyviä ja työ valmistuu ajallaan.
- Hyvin tehty aikataulun kanssa yhteensopiva riskienhallinta auttaa yllättävän paljon.

Loppusanat

- On paljon parempi saada aikaan pieni laadukas ohjelmisto, josta voidaan kehittää uusia versioita, kuin tehdä iso huonolaatuinen ohjelmisto.
- Siitä se alkaa ja siihen se päättyy: hyvästä projektin hallinnasta. Loput asiat voi kerrata esimerkiksi tästä luentomonisteesta ja oppikirjasta.