StudyGroup raportti

Miko Korhonen

Studygroup 1:

1. Hookerin 7 periaatetta:

1. The reason it all exist

Tuottaa arvoa käyttäjille

2. KISS (Keep It Simple, Stupid)

Ohjelman pitäisi olla niin yksinkertaista kuin mahdollista, mutta ei liian yksinkertaista.

Eli ei kuitenkaan huonoja ratkaisuja.

3. Maintain the Vision

Pitää yhteinen visio, jottei se projekti lähde karkaamaan eri suuntiin

4. What you produce others will consume

Tee sovellus sellaiseksi että sitä pystyy muut käyttämään, kehittämään ja    ylläpitämään. Käytännössä esimerkiksi kommentoi koodiasi.

5. Be open to the future

Ohjelman pitäisi kestää mahdollisimman pitkälle tulevaisuuteen. Pitää miettiä kaikki mahdolliset asiat mitä voi tapahtua ja varautua niiden varalle ohjelmaa tehtäessä.

6. Plan ahead for reuse

Älä kirjoita ohjelman koodissa asioita uudelleen monta kertaa vaan tee esimerkiksi funktioita joita kutsua.

7. Think

Ajattele ennen kuin teet jotakin

2. Yhtäläisyyksiä ohjelmistojen kehittämisen ja perinteisen teollisen tuotannon välillä. Mikä on mielestäsi paperin tärkein sanoma ohjelmistojen kehittäjille?

Kuten perinteisessä teollisuudessa, myös ohjelmistojen kehityksessä voi kehittyä pullonkauloja (bottlenecks), jossa joko ohjelmiston tai teollisen tuotannon kehitys tyssää, koska vaadittua tahtia ei pystytä pitämään yllä.

1. Hahmottamalla ohjelmistokehitys tuotannon riippuvuussuhde-kaavioita voi helposti huomata, missä pullonkaulat syntyvät ja näin ennaltaehkäistä niiden syntymistä.

2. Cockburn myös huomauttaa, että tuotannon tulisi pyrkiä pienempiin ja jatkuvaan tavaran sisään ja ulos liikkumiseen ennemmin kuin suurempiin ja kasautuviin tilauksiin, eli  ohjelmistokehityksessä vastaavat ovat vaatimukset ja niiden käsittely.

3. Sisäinen viestinnät on tärkeää molemmissa. Yksi tärkeä piirre yritykselle on sisäisen kehityksen tärkeys, kun esimerkiksi ohjelmoijat ymmärtävät ohjelmistotestaajien tehtävän, niin he voivat helpottaa toistensa tehtäviä tekemällä koodista helpommin testattavaa ja näin yrityksen sisäinen tiedonkulku ja ohjelmiston kehitys edistyvät nopeammin.

4. Sulavampi tiedonkulku edesauttaa yrityksen sisällön tuottamista ja ohjelmistojen   julkaisua.

3. Miten ohjelmistotuotanto alana on muuttunut historiansa aikana tähän päivään mennessä? Mitä on pysynyt samankaltaisena?

Eroa:

Ollaan päästy pois vesiputous malleista ja siirrytty käyttämään muita menetelmiä esim. ketteriä malleja.

Pilvipalvelut ova helpottaneet tiedon saamista ja lyhentäneet vasteaikoja ja parantaneet saatavuutta ympäri maailmaa

Nykyään käytetään enemmän valmiita komponentteja koodatessa eikä kirjoiteta enää kaikkea itse

Koodin toimivuus on tullut tärkeämpään rooliin. Ohjelman käytettävyyden tärkeyden kasvaminen nykypäivänä. Koodaus kieli on kehittynyt paljon.

Nykyään tehdään enemmän räätälöityjä ohjelmia, eikä kaikille valmiita ohjelmapaketteja

Samaa:

Koodaaminen on kuitenkin yhä fundamentaalisti samanlaista.

4. Millaisille projekteille protoilumalli ja vesiputousmalli soveltuvat

Parhaiten?

Protoilu sopii monenlaisiin mutta on tärkeää että asiakas on tiukasti mukana toiminnassa koska häneltä tarvitaan paljon palautetta.

Vesiputousmalli sopii parhaiten pieniin projekteihin joissa on vähän henkilöstä ja he tietävät mitä tekevät. Projektin manageri pitää olla pätevä

5. Miksi myös ketterässä ohjelmistokehityksessä tulisi käyttää molempia kehittämistapoja?

Inkrementaalisessa on tärkeää voida pysyä aikatauluissa projekteissa, joten projektin jakaminen pieniin palasiin ja niiden aikatauluttaminen on hyvä tapa tehdä projekti. Pienet palat liitetään projektiin aina kun ne on saatu valmiiksi ja sitten siirrytään seuraavaan palaseen.

Inkrementaalisuus on tärkeää projektin aikataulutuksen kannalta. Laatu saattaa kärsiä

Iteratiivisen kehittämistavan mukaan tulee parantaa niitä projekteja mitä meillä on aikataulua katsomatta. Iteratiivisuus on tärkeää ohjelman laadun kannalta. Sekä on tärkeää ajatella miltä työn ulkoasu tulee näyttää käyttäjälle. Huonona puolena projekti saattaa pitkittyä ja paisua.

On tärkeää käyttää molempia kehittämistapoja, koska ne eivät ole suorassa ristiriidassa ja kun käytetään molempia kehittämistapoja pysyy projektin laatu hyvänä ja aikataulussa.

**Oma pohdinta:**

Hookerin seitsemästä periaatteista omasta mielestäni tärkein on seitsemäs kohta think. Se kattaa parhaiten kaikki kohdat, koska on tärkeää ajatella mitä tekee. Joten kun ajattelee tulee käytyä nuo listan muutkin kohdat mietittyä miten ne on järkevintä hoitaa.

**StudyGroup 2:**

1. Mitä tekijöitä ketterien menetelmien käyttäjien tulisi ottaa huomioon verrattuna suunnitelmavetoisten menetelmien käyttäjiin? Entä päinvastoin?

Ketterissä menetelmissä tarkastellaan ja suunnitellaan projektia päivittäin ja/tai viikoittain, joten niitä käytettäessä on tärkeää panostaa tiimin kommunikointiin, koska mennään nopealla tahdilla eikä projektin visio ei ole yhtä selkeä kuin suunnitelmavetoisissa menetelmissä. Ketterissä muutoksia tapahtuu usein joten niihin pitää pystyä sopeutumaan. Suunnitelmavetoisissa menetelmissä kannattaa kiinnittää huomiota projektin kulkuun ja iteroida, koska monesti ongelmia huomataan vasta liian myöhään. Pitää suunnitella alussa hyvin, jottei tarvitsisi tehdä muutoksia projektin aiheessa.

2. Mikä on olennaista pienten yritysten vaatimusten hallinnalle Aranda et al. Mukaan

Arandan mukaan on olennaista pienelle yritykselle vaatimusten hallinta mikä sopii heidän omalle yritykselleen. Tähän vaikuttaa ketkä ovat pienen yrityksen asiakkaita. Vaatimusmäärittelyn yleensä tekee toimitusjohtaja, koska kaikki kommunikointi ulospäin tapahtuu hänen kautta, toisin kuin isoissa yrityksissä joissa voi olla tehtävään spesifioitu myynti- tai muu henkilö. Kontekstin ymmärtäminen on tärkeää. Vaatimukset pienissä yrityksissä ovat tiukasti sidottuna kulttuuri ja liiketoimintastrategiaan.

3. Aranda et al. esittelevät seitsemän pienen yrityksen vaatimusten hallintaa. Esittele näistä kaksi.

Bespoker:

                      40-45 ihmisen ohjelmistoyritys joka tuottaa räätälöityjä sovelluksia pankeille, vakuutusyhtiöille ja muille isoille firmoille. Bespoker oli näistä seitsemästä yrityksestä eniten perinteistä vaatimusmäärittelyä. Bestpokerin kumppanit vaativat analyysin työstä ennen koodaamisen aloittamista, sekä jatkuvaa dokumentointia projektin aikana. Projektin alussa muutama projektinvetäjä tapaa säännöllisesti asiakkaita kerätäkseen ohjelmistovaatimukset. Bespoker oli ainut näistä seitsemästä firmasta joka käytti UML-diagrammeja dokumentointien tarkentamiseen.

                      Suurissa projekteissa dokumentointiin satoja sivuja erinäisiä dokumentteja kuten kaavioita.  Bestpokerin asiakkaat väittivät tämän toimintatavan mahdollistavan projektin kokonaislaajuisen ymmärtämisen, mikä nosti heidän tehokkuuttaan 10 prosentilla tavanomaisesta. Kun suunnitelmat ovat annettu eteenpäin ja allekirjoitettu, alkaa kehitysvaihe. Kehitysvaiheessa projektinvetäjät tekevät kehittäjänkäsikirjan, jossa määritellään vaatimukset ja joka on kaikille saatavilla helppoa muokkausta varten.

PhoneOffshore:

                      työllistää 20-25 henkilöä, asiakkaina suuret teleoperaattorit. Työntekijät työskentelevät kahdessa eri sijainnissa. Vaatimukset määritellään neuvotteluissa asiakkaan kanssa ”joskus he (asiakkaat) kertovat mitä haluavat, joskus me päätämme. Vaatimusmäärittely on isossa roolissa suojatakseen firmaa oikeusasioissa sekä tuottaakseen asiakkaalle arvoa. Projektinjohtaja keskustelee ohjelmiston vaatimuksista useiden asianomaisten kanssa useita kertoja. Projektinjohtaja on myös vastuussa vaatimusten koordinoinnista tiimille. projektisuunnitelmaa käytetään lähinnä vain  viittauksena koska ne eivät ole kattavia. Kattavaa projektisuunnitelmaa ei tarvita koska kehittäjät ottavat yhteyttä projektin johtoon saadakseen tarvittavat tiedot uuden työn alkaessa.

4. Wiegers esittelee paperissaan 10 ansaa, jotka tulisi välttää vaatimusmäärittely-prosessissa. Esittele näistä neljä.

2. Puutteellinen asiakaspanos  
Vaatimusten tulee olla tarvittavan laajat, tarkasti määritelty ja selkeät. Vaatimusten olisi hyvä tulla kaikilta käyttäjätyypeiltä, eikä vain yhdeltä tietynlaiselta käyttäjätyypiltä, jotta ohjelman kaikki vaatimukset tulisivat ilmi, eikä vain esimerkiksi vaatimukset jota johtoporras tarvitsisi  
Ratkaisu puutteellisen asiakas panokseen voi esimerkiksi olla, että määritellään erilaisia käyttäjärooleja ja määritellä heidän käytettäviä toimintoja, käytön määrää, käyttäjien käyttöoikeuksia.   
  
3.  Hämärät ja tulkinnanvaraiset vaatimukset  
Tulkinnanvaraiset vaatimusmäärittelyt ovat petollisia ohjelmoijille, koska se voi tarkoittaa montaa eri asiaa, ja näin asiakas voi vedota epätoivotun tuloksen tullessa, että toiminto on toteutettu väärin ja näin tulee lisätöitä. Hämärät vaatimukset voi tunnistaa siitä, että asiakkailta joutuu kysymään paljon tarkentavia kysymyksiä tai joutuu arvailemaan mitä asiakas tällä vaatimuksella tarkoittaa ja vaatii. Tämä arvailupeli on vaarallista, koska vaatimuksesta pohjautuva toiminto voi olla väärä asia loppujen lopuksi, jonka korvaaminen vaatii lisätyötä ja tulee kalliiksi näin. Tämän takia tulisi välttää erilaisia tulkinnanvaraisia sanoja, kuten ”helposti, nopea, käyttäjä ystävällinen, kätevä, tehokas.” Monien eri ohjelman tekijöiden osapuolien tulisi tarkastella dokumentteja ja näin antaa omat näkökulmansa asiaan liittyen.  
  
4. Uusien vaatimusten määrittely tuotekehityksen aikana  
Ohjelmistokehityksessä tulisi ainakin olla jonkin verran liikkumavaraa, jos asiakas haluaa lisätä uusia vaatimuksia kesken ohjelmistotuoton. Tyypillisesti projektien deadlinet eivät muutu, eikä budjetitkaan muutu uusien toimintojen tilalle. Näistä uusista vaatimuksista väitellessä tulisi kyseenalaistaa että mahtuvatko ne suunniteltuun liikkumavaraan, eli onnistuuko niiden toteuttaminen.

6. Analyysi paralyysi

Joskus ohjelman suunnitelmaa aletaan suunnittelemaan liikaa ja versioita syntyy aina vain uusia. Tällöin projektia ei voida aloittaa koska suunnitelmaan tapahtuu koko ajan muutoksia. Tätä kutsutaan analyysi paralyysiksi. Se on harvinaisempaa kuin se että jätetään huomioimatta osia suunnitelmasta. Jos vaatimuksissa on vielä jotain epäselvää, voidaan tätä ongelmaa yrittää estää valitsemalla sellainen kehitystapa joka mahdollistaa niiden toteuttamisen kun ne selviävät.

**StudyGroup 3:**

1. Conway:n laki esittää, että ”järjestelmän rakenne heijastaa sitä rakentavan organisaation rakennetta”. Mitä tämä mielestänne tarkoittaa ja löytyykö siihen esimerkkejä?

Conwayn laki mitä pirstaloisempi yhtiön rakenne ja kehitystiimit ovat, niin sitä pirstaloisempi lopputuloskin on. jos kommunikaatio huonoa niin se voi viestiä myös epämieluisasta lopputuloksesta. Joka palanen ohjelmistosta viittaa sen rakentaman organisaation rakenteeseen. Kyllä tämän huomaa tavallaan ohjelmistokehityksessä, kun monen eri ryhmän panos vaatii tietyn määrän työtä, että nämä erilliset komponentit saadaan yhteensopivaksi.

On tutkittu että tiukasti kytköksissä olevat projektityhmät tekevät sovelluksensa (koodinsa) sen huomioiden että projektin osapuolet ymmärtävät toisensa ja open source projektit luodaan open sourcea ajatellen eli ne ovat enemmän modulaarisia ja kommentoitua jne.

2. Esitä kaksi sovellusesimerkkiä seuraaville arkkitehtuurityylille

**Kerrosarkkitehtuurin** perusajatus on, että järjestelmä koostuu joukosta eri käsitetasoja. Nämä käsitetasot ovat itsenäisiä. Samalla käsitetasolla sijaitsevat järjestelmän palvelut muodostavat kerrosarkkitehtuurityylissä kerroksen [Busc96, s. 31].

**liiketoimintajärjestelmät**, esim vakuutusyhtiön tietojärjestelmä

Alin kerros tarjoaa yleistä infrastruktuuritukea, esimerkiksi tietokannan ja sen toiminnot, sen yläpuolella sovellusalueen logiikan sekä käsitteet toteuttava kerros. Tämän jälkeen seuraa sovelluksen logiikan toteuttava kerros ja ylimpänä kerroksena on sovelluksen käyttöliittymän toteuttava kerros. Alin kerros on lähinnä laitteistoa, kun taas ylin lähinnä järjestelmän käyttäjää. Esimerkkijärjestelmässä sovellusaluelogiikkakerroksessa on kaikille vakuutuksille yhteiset ominaisuudet, kuten asiakas tai laskutus. Sovelluslogiikkakerros sisältää esimerkiksi liikennevakuutuslogiikan tai henkivakuutuslogiikan. Tällainen arkkitehtuurityyli mahdollistaa kahden alimman kerroksen uudelleenkäytön, jos ilmenee tarvetta tehdä esimerkiksi uusi vakuutussovellus.

**tietoliikenneprotokollat**: esimerkiksi OSI-malli sekä TCP/IP , Tietoliikenneprotokolliin kerrosarkkitehtuurityyliä sovelletaan järjestämällä yleisen protokollan erikoistuneet aliprotokollat kerrosarkkitehtuurityylin mukaiseen kerrostusjärjestelmään ja kutsuu palveluitaan ainoastaan alemmalta kerrokseltaan.

**Tietovuoarkkitehtuuri** on arkkitehtuurityyli, joka sopii erityisen hyvin sellaisen järjestelmän malliksi, jossa on luonteenomaista tietovirran jäsentäminen ja jalostaminen. Tietovuoarkkitehtuurityyli on eräänlainen sovellus hajota ja hallitse -ongelmanratkaisutekniikasta: monimutkainen tehtävä jaetaan pienempiin kokonaisuuksiin, jotka voidaan sitten ratkaista pienemmällä resurssitappiolla, kuin jos koko tehtävä ratkaistaisiin kokonaisuudessaan kerralla.

**äänenkäsittelyjärjestelmät**

**Kuvankäsittely**

**asiakas/palvelin-arkkitehtuuri**

Asiakas-palvelin-arkkitehtuurityylissä perusajatuksena on kapselointi. Tässä arkkitehtuurityylissä kapseloidaan tietyn arkkitehtuuritason resurssin (palvelin) hallinta siten, että resurssin käyttäjän (asiakas) ei tarvitse huolehtia resurssin käyttöön liittyvistä teknisistä ongelmista. Kapselointi toimii myös toiseen suuntaan, palvelin ei tunne asiakkaan teknistä toteutusta.

**hajautetut liiketoimintajärjestelmät ja niille tarkoitetut ohjelmistoalustat**: Syynä hajautettavuuteen on se, että yleensä asiakkaat ja palvelimet suunnitellaan toisistaan riippumattomiksi jo etukäteen ja ajatellaan että ne tulevat sijaitsemaan eri prosesseissa. Tällä tavalla saadaan myöskin eristettyä palvelimien ja asiakkaiden ongelmat. Esimerkiksi jos asiakas toimii virheellisesti, voi palvelin silti palvella muita asiakkaita

**www-sivut**: Käyttäjä kirjoittaa palvelimen nimen sekä tiedoston nimen, jonka hän haluaa saada internet-selaimen osoite-kenttään. Asiakas-ohjelma, eli selain muodostaa yhteyden palvelimeen ja pyytää siltä käyttäjän pyytämään tiedostoa. Palvelin hakee tiedoston kovalevyltään, lisää siihen tarvittavat otsikkokentät ja lähettää sen asiakkaan koneelle. Asiakkaan kone vastaanottaa lähetetyt tiedot ja näyttää tuloksen selaimen ikkunassa käyttäjälle.

**Tietovarastoarkkitehtuurissa** joukko järjestelmiä tai komponentteja ylläpitää yhteistä tilaa tietovarastossa. Tässä arkkitehtuurissa ydin on jaettu tietovarasto, jota kaikki järjestelmän komponentit voivat tutkia ja muuttaa. Komponentit eivät kommunikoi suoraan vaan tietovaraston kautta. Jotta tiedoille ei tule ristiriitaisuutta, täytyy tietovaraston yleensä tukea transaktiota, eli jos tehdään tapahtumaa, jossa tietoa muutetaan useissa paikoissa, tehdään se aina kokonaan tai sitten ei ollenkaan.

**ohjelmointiympäristöt**

**Pankkijärjestelmät**

3. Garlan et al. esittelee kolme perustekniikkaa arkkitehtonisen yhteensopimattomuuden käsittelemiselle (hoitamiselle). Mitä ne ovat ja mihin periaatteisiin ne nojautuvat?

**Arkkitehtuurinen erikoistuminen**

Yksi tapa estää yhteensopimattomuutta on erikoistua arkkitehtuurisesti tietyntyyppiseen alueeseen. Tällöin rajoitetaan sallittuja komponentteja ja näin vähennetään todennäköisyyksiä yhteensopimattomuuteen.

**Open Source**

Avoimen lähdekoodin projektit parantavat yhteensopivuutta kahdella tavalla. Ensinnäkin standardisoidaan tiettyjä arkkitehtuurisia tyylejä ja viitekehyksiä. Toiseksi luodaan kokemusta ja esimerkkejä joilla selvennetään minkälaisia arkkitehtuurisia tekniikoita kannattaa käyttää mihinkin sovellustyyppeihin.

**Virtualisointi ja yleiset käyttöliittymät**

Korkean tason kommunikointiprotokollat ja tietostandardit sekä yhteiset kieli- ja selainympäristöt helpottavat sovellusten kehittämistä jotka toimivat yhteisissä virtuaaliympäristöissä. Näin estetään osa yhteensopimattomuuksista jotka johtuvat eri alustojen tai käyttöliittymien käytöstä.

4. Kumpi uudelleenkäyttö (komponenttien/luokkien) oli helpompaa? Mitä komponenttien mustalaatikko (black-box) uudelleenkäyttö tarkoittaa?

Tulokset osoittavat että komponentteja on helpompi uudelleen käyttää kuin olioita. Komponenttien käytössä ei tehdä lähdekoodiin muokkauksia, joten ohjelman ei muutu monimutkaisemmaksi. Musta laatikko uudelleenkäytössä pyritään hyödyntämään uudelleen komponentteja, joilla on matala kytkentä. Ohjelmoijat voivat kokonaan käyttää komponentteja uudelleen ilman suurempia muokkauksia koodiin tai komponenttiin, ja näin ohjelman kustannukset vähenee.

5. Mitä tarkoittavat pilvilaskenta-arkkitehtuurin kolme tasoa; SaaS, PaaS ja IaaS (Yau & An mukaan)? Anna esimerkkejä ohjelmistoympäristöistä missä kutakin pilvilaskenta-arkkitehtuurin tasoa on käytetty.

**SaaS (Software as a Service)**

Ohjelma ei sijaitse ollenkaan käyttäjän koneella vaan joku ulkopuolinen palvelu hoitaa sovellusta pilvessä. Tähän luokkaan kuuluvat sekä käyttäjille tarkoitetut kokonaiset ohjelmat että kehittäjille tarkoitetut työkalut tai muut sovellukset. Tehdään yleensä helposti integroitavaksi muiden SaaS sovelluksien kanssa.

Esim. Google Apps jolla nämäkin tehtävät on tehty. Palvelu tarjoaa driven ja työkaluja tehdä asioita samanaikaisesti muiden kanssa.

**PaaS (Platform as a Service)**

Tarkoittaa palvelualustan ulkoistamista pilveen. Tarkoituksena helpottaa esimerkiksi sovellusten suunnittelua, testausta, käyttöönottoa, monitorointia jne. Käyttäjä pystyy keskittymään olennaiseen kun alusta tarjoaa kaiken muun tarvittavan

Esim. Google App Engine joka tarjoaa kehitysalustan sovelluskehittäjille.

**IaaS (Infrastructure as a Service)**

Käytännössä ulkoistetaan palvelimet pilveen. Virtuaalisella palvelinta käytetään sitten hyödyksi samalla tapaa kuin tavallistakin palvelinta esim. verkkoyhteyksissä, tallennustilassa tai halutessa lisää laskentatehona. Asiakas voi skaalata resursseja tarpeen mukaan.

Esim. Google Compute Engine. Tarjoaa google palvelinsaleissa olevia virtuaalikoneita

**StudyGroup 4:**

1. Olet suunnittelemassa nettikauppa.com –nimistä tietotekniikka/viihde- elektroniikkakauppaa webiin. Mieti miten toteutat Olsinan ja Fitzpatrickin laatutekijät sovellukseesi.

Käytettävyys

Sivusto pitää suunnitella järkevästi että asiat eivät hypi silmille mutta asiat ovat silti selkeästi näkyvissä. Sivun käytettävyys tulisi olla hyvä sekä puhelimella että selaimella.

Toiminnallisuus

Sivulla tulee olla hakukone joka osaa avainsanoilla löytää alasivuja sivustolta. Sivuston ylhäällä näkyy leivänmurut josta näkee polku mitä on kulkenut. Vasemmalla reunalla näkyy navigointipalkki josta näkyy tärkeimmät otsikot sivuihin.

Luotettavuus

Linkit sivustolla eivät saa olla rikki.

Tehokkuus

Sivuston vasteaika saa olla max 5 sekuntia. Grafiikka saa olla max 1000x1000 tarkkuudella jotta sen lataamisessa ei mene liian kauan aikaa. Grafiikat kannattaisi silti olla responsiiviset jotta ne skaalautuisivat oikein erilaisille näytöille

Ylläpidettävyys

Sivusto pitää luoda Wordpressillä sillä tavalla että sinne on helppo luoda uusia sivuja ja korjata virheitä vanhoista. Tehdään magentolla

Näkyvyys

Maksaa googlelle jotta se mainostaa sivustoa hakukoneessa ja nostaa sen näkyvyyttä hakusanoilla

Ymmärrettävyys

Teksti sivuilla on selkeästi aliotsikoitu ja kirjoitettu oikeaoppisella äidinkielellä. On myös mahdollista vaihtaa kieltä suomen, ruotsin ja englannin välillä.

Uskottavuus

Ei ole uskottava

Viehättävyys

Navigointipalkki vasemmalla tulee olla selkeä ja mukavan näköinen. Käyttäjälle ei saa tulla tunnetta että joutuu etsimällä etsimään jotakin tietoa.

Erilaistaminen

Meidän nettikauppamme pyrkii olemaan parempi ja erilaisempi kuin muut tarjoamalla laitteistoa, ohjelmistoja ja suoratoistopalveluja toisin kuin kilpailijat.

2.

McCallin laatumalli on kehitetty käyttäjän näkökulmasta kehittäjän tärkeimpiin kehityspiirteisiin.

Se on hyvin tuotepainotteinen, joten se ei ole vanhentunut käytöstä kuluvien ohjelmistokehitystapojen tapaan. McCall tunnisti ja hahmotteli kolme tärkeää jäsenneltävää näkökulmaa ohjelmiston määrittelyyn:

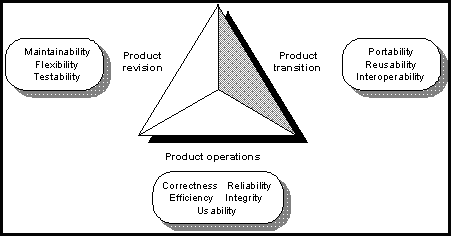
-  1. tuotteen KÄYTÖN laatutekijät, jotka vaikuttavat ohjelmiston käyttöön

-  2. tuotteen MUOKKAUKSEN laatutekijät, jotka vaikuttavat ohjelmiston ylläpitoon

-  3. tuotteen SIIRRON laatutekijät, jotka vaikuttavat ohjelmiston toimivuuteen eri alustoilla

McCallinnin määrittelemät laatutekijät ovat myös säilyneet pitkään hyödyllisinä, koska ne ovat pääosin ei-toiminnallisia ja niitä on vaikeampi määritellä näin asiakkaana toisin, kuin toiminnalliset vaatimukset.

Tämä kaikki tapahtui myös ennen olio-ohjelmoinnin pohjaa



3.

Virheetön koodi ei tee ohjelmasta hyvää, jos esimerkiksi ohjelmiston logiikka ja syntaksi on huono. Tällöin kääntäjä kääntää koodin, mutta ohjelma ei siltikään välttämättä tee mitään järkevää. Koodi voi olla virheetön, mutta se ei siltikään välttämättä täytä vaadittuja vaatimuksia. Koodin huono arkkitehtuuri myös johtaa siihen, että sitä on vaikea laajentaa tai sen kehittäminen muuttuu vaikeaksi, näin sen osia ei voida tulevaisuudessa käyttää enää. Koodissa voi olla myös huonoja algoritmeja tai muita ratkaisuja jolloin koodi toimii mutta epätehokkaasti.

4.

Virheiden määrä on yksi laadun indikaattori. Mitä muita indikaattoreita tunnet?

laadun indikaattoreita:

-          käytettävyys

-          toiminnallisuus

-          luotettavuus

-          tehokkuus

-          ylläpidettävyys

-          näkyvyys

-          ymmärrettävyys

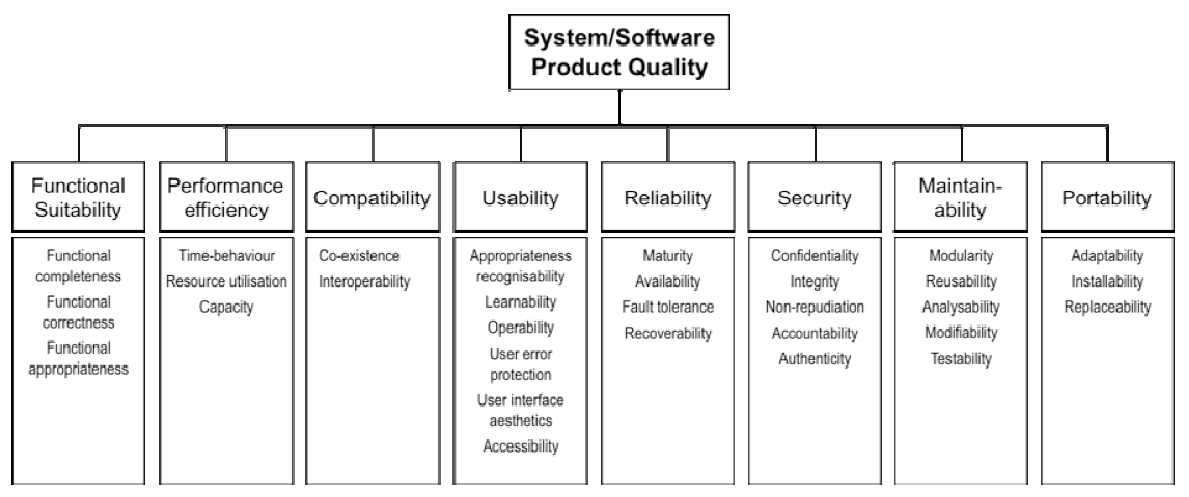
-          uskottavuus

-          viehättävyys

-          erilaistaminen

ISO 25010

-> turvallisuus yms



5. Mitä yhteistä on aktiviteeteilla katselmointi (review), tarkastus (inspection), läpikäynti (walkthrough) ja pariohjelmointi (pair programming). Missä suhteessa ne eroavat toisistaan?

Yhteistä näillä aktiviteeteilla on se, että toinen tai useampi osapuoli tarkastaa toisen tekemän työn virheistä ja antaa palautetta. Tämän ansiosta saadaan useampi mielipide työstä.

Pariohjelmoinnissa toinen ohjelmoija kirjoittaa koodia ja samanaikaisesti toinen ohjelmoija tarkastaa koodia. Rooleja vaihdellaan

Katselmointi eroaa muista siten, että tarkastus tehdään palaverimaisesti tarkastellaan enemmän vaatimusmäärittelyitä

Läpikäynnissä työn tekijä selittää tarkastajille itse mitä/miten hän on työnsä tehnyt

tarkastukset tarkastellaan taas oikeista tuotoksista eli katsotaan onko työ tehty niin kuin on haluttu

Kaikki eroavat myös siinä, että nämä käydään läpi ohjelmistokehityksen eri vaiheissa

**StudyGroup 5:**

1. **Mitkä ominaisuudet tekevät ohjelmistosta helpommin testattavan?**

-Ohjelmiston pitää toimia pääpiirteittäin oikein –  ei saa kaatua kättelyssä, jotta esim ohjelmiston eri osia voidaan testata.

-Ohjelmiston toimintaa voidaan tarkastella ulkoapäin -  Jokaisesta syötteestä tulee erillinen tuloste, eli järjestelmän tilat ja muuttujat on hyvä saada näkyville, jotta suoritusta  voidaan seurata vaiheittain ja virheelliset tulosteet on helppoa havaita

-Ohjelmistoa voidaan ohjata ulkoapäin – Eli mahdolliset tulosteet voidaan tuottaa jollain syötteellä. Ohjelmiston ja laitteiston tilaa voidaan ohjata ulkoapäin ja järjestelmän tilaa voidaan muuttaa. Näin testit voidaan määritellä, automatisoida ja suorittaa uudestaan

-Ohjelmistoa tulee voida testata moduuleittain – Moduulien yhteenkuuluvuus mahdollisimman korkeaksi ja kytkentä mahdollisimman matalaksi

- Ohjelmisto on stabiili – Ohjelmistoon ei tule jatkuvasti uusia ominaisuuksia ja muutokset tehdään versionhallinnan kautta

- Testaajat tuntevat ohjelman

- Hyvä määrittely ja suunnittelu

**2. Black box vs white box testing. Mitkä seikat rajoittavat valkealaatikkotestauksen käyttöä?**

1. black box testaus ei vaadi ohjelmointi – tai implementointitaitoja, kun taas white box testaus vaatii testaajaltaan molempia.
2. black box testaus koettelee ohjelmistoa kokonaisuudessaan ilman tietoa ohjelmiston sisäisestä koodista tai rakenteesta, white box testaaja on tietoinen ohjelman sisäisestä rakenteesta ja saattaa sijoittua rakenteen välisten toimijoiden testaamiseen
3. black box testaaja voi olla ulkopuolinen toimija, jos yritys haluaa pitää tuotteensa sisällön salattuna, white box testaaja pitää tutustuttaa projektiin syvällisesti, ja olla tietoinen ohjelman sisällöstä

black box – kokonaisuuden testaaja

white box – yksittäisten toiminnallisuuksien testaaja

White box testausta rajoittaa se, että testaajan tulee olla tietoinen ohjelmasta ja olla ohjelmoija, jotta hän voi testauksessa käydä läpi kaikki reitit ja mahdolliset tapaukset mitä ohjelmassa voi tehdä. Kaikkia mahdollisia tilanteita on mahdotonta testata, joten jotkut tapaukset jää testaamatta. White box testaus vie myös pidemmän aikaa ja on kallista.

**3. Mitä ohjelmistorajapintoja testaajan täytyy tunnistaa ja simuloida**

**(Whittakerin mukaan)?**

testaajan tulee simuloida ohjelman ja sen ympäristön välistä toimintaa. Testaajan pitää löytää ja tunnistaa kaikki ohjelman käyttämät rajapinnat ja testata kaikki syötteet mitä kullekin rajapinnalle voi tulla. Hyvä testaaja testaa validit kuin invaliditkin ja odottamattomat syötteet. Testaajan tulee myös pohtia mikä voisi kaataa ohjelmiston. esimerkiksi kellonaika liittyy useaan ohjelmaan, mitä jos sitä muuttaa?

**4. Mitkä ominaisuudet tekevät TDD:stä ammattilaisen työkalun**

TDD eli test driven development

Kolme lakia:

* Älä kirjoita koodia ennen kuin olet kirjoittanut testauksen joka epäonnistuu
* Kirjoita vain minimimäärä testausta jota epäonnistuminen tarvitsee
* Kirjoita aina koodia vain sen verran että epäonnistunut testi menee läpi

Martinin mukaan TDD:Tä vastustetaan koska se koetaan liian työlääksi tai ei pidetä siitä että pitäisi testata lyhyin väliajoin eikä saa kirjoittaa koko moduulia kerralla. TDD ei myöskään sovi toteutettavaksi kaikissa ympäristöissä, mutta Martinin mukaan TDD on ammattimaisempi tapa toteuttaa testausta kuin jälkikäteen tehty manuaalinen testaus.

TDD:ssä testejä tehdään useita joka päivä joten loppujen lopuksi kertyy valtavasti testaustiedostoja jotka kattavat lähes koko tuotantokoodin ja ne voidaan sitten helposti ja nopeasti ajaa jälkeenpäinkin.

Monesti kehittäjät eivät halua siivota koodian koska he pelkäävät että jokin menee rikki ja joutuu tekemään lisää töitä. TDD vähentää muutosten tekemisen riskiä, varsinkin pienissä muutoksissa. TDD:n ansiosta ei tarvitse tehdä myöskään paljoa debuggausta koska käyt läpi koodia lyhyissä pätkissä joten tiedät missä virhe on.

Ammattimainen koodi on puhdasta ja joustavaa koodia joka toimii ja tulee ajallaan. TDD vähentää pelkoa tehdä muutoksia joten koodi todennäköisesti on puhtaampaa. Koodi on myös joustavaa koska tiedät että se ei mene rikki vaikka teet siihen muutoksia. Tärkeintä TDD:ssä on kuitenkin että sillä varmistetaan että koodi toimii. Ja vaikka TDD ei sinänsä auta koodin tuottamisessa ajallaan, vähentää se kuitenkin muuttuvia tekijöitä, jolloin on helpompi arvioida kauanko aikaa tulee menemään.

**5. Miten Web-sovellusten toiminnallisuuden testaus eroaa perinteisten**

**ohjelmistojen testauksesta?**

Web-sovelluksia testatessa nähdään asiakaspuolelta ensin virheen oireet eikä itse virhettä. Web-sovelluksia toteutetaan eri konfiguraatioina joten on vaikeampaa toteuttaa sama virhe uudestaan. Web-sovellukset rakennetaan asiakas/palvelin arkkitehtuurille joten virheet voivat löytyä eri tasoilta. Jotkut virheet johtuvat konfiguraatiosta jossa testataan, toiset suoritusaikaisista resursseista. Web sovelluksissa testaus tapahtuu suojatun yhteyden kautta joten täytyy ottaa huomioon erilaisia protokollia.

**StudyGroup 6:**

1. Vaihetaso

Vaihetaso on hallinta-alkion viimeisin versio (komponentti tai konfiguraatio). Uudessa vaihetasossa on tapahtunut aina joku muutos projektissa joka on hyväksytty. Vaihetasosta tiedetään mitä projektissa tarvitsee tehdä, joten ei pitäisi tulla mitään pikaisia ja yllättäviä lisäyksiä. Vaihetaso toimii myös eräänlaisena sopimuksena että mitä asioita olla luvattu asiakkaalle tehdä. Projektissa myös tiedetään mitä tullaan tarvimaan, joten on helpompi ajaa sitä muutenkin eteenpäin.

2. QMOOD

Uudelleenkäytettävyys: (Coupling), Cohesion, Messaging, Design size

Ymmärrettävyys: (Complexity, Design size, Abstraction), Encapsulation, Cohesion

Mitkä vaikuttivat eniten:

Design size, Messaging?

3.

suljettu (closed) paradigma voisi olla tähän tilanteeseen sopiva, kun aikataulu on määritelty ja tiukka. Suljetun paradigman hierarkinen organisaatio järjestely ja selvät roolit helpottavat työn jakoa osiin, suljetussa paradigmassa myös paineensietokyky on parempi kuin muissa. tekstinkäsittelyohjelmassa kehityssuuntaa on varmasti saatu käyttäjien palautteesta, joten työntekijöiden ei tarvi olla innovatiivisemmasta päästä.

muut paradigmat:

satunnainen (random):

avoin (open):

synkroninen(synchronous):

4.

Van Genuchten ja Hatton esittelevät paperissaan uuden mittarin,

ohjelmiston hyöty (software mileage). Mitä sillä tarkoitetaan ja mihin sitä

voidaan käyttää?

Sillä tarkoitetaan sitä, että kuinka monta uutta asiakasta ohjelmisto saa, jokaista uutta kirjoitettua koodiriviä kohden. Eli jos saat paljon uusia asiakkaita kirjoitettua koodia kohden on sinun koodillasi hyvä “ohjelmiston hyöty”.

Tällä hetkellä ohjelman hyötyä voidaan käyttää mittaamaan oman alan yrityksien ohjelmistoja. Näitä tietoja voidaan sittemmin vertailla, josta voidaan selvittää onko omaan ohjelmistoon on hyvä investoida enemmän.

5.

Mitä yhteistä näet ansaitun arvon arviossa ja Scrumin edistymiskäyrä

(burndown) arviossa?

Molemmissa verrataan tuoreita arvoja suunniteltuihin arvoihin, josta arvioidaan projektin edistymistä. Molemmissa tehdään ensin sovittu aikajakso työtä, jonka jälkeen tulokset päivitetään arviointiin.

6 & 7

.

Ohjelmistoprosessin metriikoita tulisi mitata, jotta ohjelmiston edistymistä ja prosessia voitaisiin seurata. Näiden metriikoiden päätavoite on ajaa projektia kohti valmistumista ja samalla tuottaa jotain arvoa. Turhan tiedon kerääminen, joka ei tuota arvoa, voi toisaalta hidastaa projektin valmistumista, joten tälläinen metriikoiden haitallinen mittaaminen tulisi tunnistaa hyvissä ajoin projektia ja lopettaa.

Suoria ja pakollisia metriikoita, joita tulisi käyttää ei ole, vaan tiimin tulisi miettiä omaan projektiin sopivat arvot ja toteuttaa niitä suunnitelman mukaisesti.

Yleisiä tuottavuuden mittareita ovat mm

* koodin määrä (koodirivit, metodien määrä, periytymisen määrä, koodin monimutkaisuus (eli  SYKLOMAATTINEN KOMPLEKSISUUS ), luokkien määrä, aliluokkien määrä, kytkennän määrä ja koheesion määrä)
* prosessin seuranta (työtunnit, sprinttien määrä, sprinttien pituus, tuottavuus/sprint)
* agile prosessi metriikat (vaatimusten hankinnan aika, analysoinnin aika, suunnittelun aika, koodaus aika, testaus aika, toteutus/implementointi aika)

Ohjelmoijilla voi olla perseestä kirjottaa jotai turhaa paskaa joka on tylsää ja voi hidastaa ohjelmistotuotantoprosessin aikataulua huomattavasti, jos ei riitä moti

Gardening on ohjelmiston muokkaamista kauniimmaksi ja hyödyllisemmäksi. Refactoring on vähemmän vaikutuksellisempaa kuin reengineering ja rewriting

Refactoring on ohjelmiston muokkaamista siten, että se ei vaikuta koodin käyttäytymiseen, mutta se kuitenkin parantaa ohjelmiston sisäistä rakennetta

Refactoringilla voidaan parantaa

* Ylläpidettävyyttä ja yhteensopivuutta

Se ei kuitenkaan ole vaihtoehtoinen toteutus muoto bugien korjaamiseen

Reengineeringillä pyritään parantamaan systeemin laatua paremmaksi, kun refactoringilla muokataan koodin ulkonäköä ja rakennetta vain. Reengineeringillä voidaan myös julkaista uutta toiminnallisuutta ja sillä voidaan parantaa jo valmiita hyvin toimivia komponentteja

Re-engineeringiä tulee suorittaa, jos ohjelmiston business case muuttuu tai refactoring on riittämätön ohjelmiston parantamiseen tai uudet operatiiviset tai toiminnalliset funktiot eivät sovellu hyvin ohjelmiston nykyiseen rakenteeseen

Rewritingia kannattaa harkita jos re-engineeringi menee liian kalliiksi. Rewriting on olemassa olevan systeemin tai toimivan funktion uudelleen kirjoittamista. Se on saatanan riskialtista.

Rewriting on myös mahdollisuus silloin kun refactoring on liian hyödytöntä tarvittaviin vaatimuksiin verrattuna

**Oma pohdinta:**

Ohjelmistojen mittaamiseen ei vieläkään ole löydetty parasta mahdollista tapaa, kuten tuossa uudessa mittarissa ”software mileage” huomataan, että koodirivi on hyvin huono mittaus kohde. Koodin voi kirjoittaa niin monella eri tavalla esimerkiksi vaikka kaiken yhdelle riville tai eri määrä tyhjiä rivejä. Tästä syystä tällä uudella mittarilla ei saada kovin järkeviä vertailuja.