VAASAN YLIOPISTO

TEKNIIKAN JA INNOVAATIOJOHTAMISEN YKSIKKÖ

OHJELMISTOTEKNIIKKA

Jesse Koski

TYÖKALUT APUNA KESKUSKONEYMPÄRISTÖN SOVELLUKSIEN LAADUNHALLINNASSA

Vaasassa 2019

Työn valvoja	Prof. Jouni Lampinen
Työn ohjaaja	DI Mika Lomu

ALKULAUSE

Tämä diplomityö on tehty eräälle it-yritykselle toimeksiantona.

Haluan kiittää ohjaajaani Mika Lomua työhön liittyvästä palautteesta ja ohjauksesta työtä kirjoittaessani sekä Johanna Juntusta työn mahdollistamisesta. Yliopiston puolelta haluan kiittää työnvalvojaani Jouni lampista työn tarkastamisesta ja palautteesta koko prosessin aikana.

Vaasassa 04.09.2019

Jesse Koski

SISÄLLYSLUETTELO

SYMBOLI- JA LYHENNELUETTELO

ADP	Automated defect prevention, automatisoitu vikojen ehkäisy
ISO	International Organization for Standardization, kansainvälinen standardointijärjestö
COBOL	Common business-oriented language, kaupallishallinnollisiin sovelluksiin tarkoitettu ohjelmointikieli
COMTRAN	Commercial Translator, kaupallinen kääntäjä (vanha ohjelmointikieli)
MIPS	Millions of instructions per second, suorituskyvyn mittayksikkö, kertoo kuinka monta miljoonaa käskyä tietokone pystyy suorittamaan sekunnissa
MSU	Million service units, suorituskyvyn mittayksikkö, kertoo kuinka monta miljoonaa käskyä tietokone pystyy suorittamaan tunnissa
IEC	International Electrotechnical Commission, kansainvälinen sähköalan standardointiorganisaatio
LOC	Lines of code, lähdekielisen ohjelman rivien lukumäärä
SLOC	Source lines of code, lähdekielisen ohjelman rivien lukumäärä
NLOC	Non-commented lines of Code, lähdekielisen ohjelman rivien lukumäärä, joita ei ole kommentoitu
CLOC	Commented lines of code, lähdekielisen ohjelman kommentoitujen rivien lukumäärä
IMS	Information Management System, tietohallintajärjestelmä
DL/I	Data Language Interface, järjestelmä, jota käytetään IBM:n IMS tietokantoihin ja sen tietoliikennejärjestelmään
TSO	Time Sharing Option, järjestelmä, jonka avulla voidaan käyttää keskustietokoneen tarjoamia palveluita pääteyhteydellä
IDZ	IBM Developer for z systems, integroitu kehitysympäristö IBM:n z-järjestelmälle
IDE	Integrated Development environment, integroitu kehitysympäristö

PL/SQL	Procedural Language for Structured Query Language, proseduraalinen kieli strukturoidulle kyselykielelle
CWE	Common Weakness Enumeration, ohjelmistojen heikkouksien ja haavoittuvuuksien luokkajärjestelmä
MISRA C	Motor Industry Software Reliability Association, joukko ohjelmistokehitysohjeita C-ohjelmointikielelle
SQALE	Software Quality Assessment Lifecycle Expectations, ohjelmiston laadun arvioinnin elinkaaren odotukset

VAASAN YLIOPISTO
Teknillinen tiedekunta
Tekijä: Jesse Koski
Diplomityön nimi: Työkalut apuna keskuskoneympäristön sovelluksien laadunhallinnassa
Valvoja: Professori Jouni Lampinen
Ohjaaja: DI Mika Lomu
Tutkinto: Diplomi-insinööri
Yksikkö: Tekniikan ja innovaatiojohtamisen yksikkö
Koulutusohjelma: Energia- ja informaatiotekniikka
Suunta: Ohjelmistotekniikka
Opintojen aloitusvuosi: 2016
Diplomityön valmistumisvuosi: 2019 **Sivumäärä:** 76

TIIVISTELMÄ

Keskuskoneympäristön sovelluskehityksessä vanhojen ja suurien lähdekielisten ohjelmien laadukkuuden arvioiminen on hankalaa ilman oikeanlaisia työkaluja. Ohjelmistokehittäjien aiemmin tehtyjen ratkaisujen vaikutukset voivat näkyä negatiivisena lopputuloksena vuosienkin päästä erinäisinä ongelmina ohjelmien toiminnoissa. Tähän on ollut aiemmin apuna sovelluskehittäjien kymmenien vuosien vahva kokemus kohteen liiketoiminnasta, ohjelmista ja niiden lähdekielisten ohjelmien kielestä.

Tarkoituksena oli tutkia keskuskoneympäristössä tapahtuvaa sovelluskehitystä sovelluskehittäjien näkökulmasta pyrkimällä parantamaan sovelluskehittäjien tuottamien ratkaisujen laadukkuutta, tehokkuutta sekä helpottaa manuaalisesti tapahtuvaa työtä staattisen testauksen työkalulla, joka soveltuu lähdekielisten ohjelmien katselmointeihin.

Työssä tutkittiin erästä kaupallista ohjelmistoratkaisua ja analysoitiin sen ratkaisujen soveltuvuuksia, erään organisaation käyttöön. Lähtökohtana oli, että pilotoitavia ratkaisuja olisi ollut useampia ja varteenotettavimmat niistä ohjelmistoratkaisuista pilotoitaisiin, mutta kriteerejä täyttäneitä ohjelmistoratkaisuja ei löytynyt kuin yksi. Työssä myös analysoitiin kahta erilaista COBOL-ohjelmaa keskenään, sekä vertailtiin niiden tuloksia keskenään.

Tutkimustuloksista voidaan vetää johtopäätös, että pilotoitavan ohjelmistoratkaisun avulla voidaan paikantaa laadullisia ongelmia suuresta määrästä lähdekielisiä ohjelmia, joka ei ole ihmisvoimin manuaalisesti tehtynä välttämättä kovinkaan kannattavaa. Ohjelmointiympäristöön on mahdollista liittää ohjelmistoratkaisun lisäosa, joka mahdollistaa uuden lähdekielisen ohjelman luonnissa reaaliaikaisen ongelmien tarkastuksen. Ohjelmistoratkaisu tarjoaa myös omanlaisen ratkaisunsa lähdekielisen ohjelman laadunhallintaan, joka perustuu annettuihin kynnysarvoihin.

AVAINSANAT: Lähdekielisten ohjelmien katselmointi, keskuskoneympäristö

UNIVERSITY OF VAASA

Faculty of technology

Author:	Jesse Koski
Topic of the Thesis:	Quality management of applications with tools in the mainframe environment
Supervisor:	Professor Jouni Lampinen
Instructor:	MSc (Tech) Mika Lomu
Degree:	Master of Science in Technology
Department:	School of Technology and Innovations
Degree Programme:	Energy and Information Technology
Major of Subject:	Software Engineering
Year of Entering the University:	2016
Year of Completing the Thesis:	2019 **Pages:** 76

ABSTRACT

Evaluating the quality of old and large program codes in the application development of the mainframe environment is difficult without the right tools. The effects of previous software developer solutions may, over the years, be reflected in several problems with software functions. This has been assisted in the past by decades of application developers strong experience in the subject's business, programs and their code.

The purpose was to study the application development in the mainframe environment from the application developer point of view, aiming to improve the quality, efficiency of the applications produced by the application developers and facilitate manual work with a static testing tool suitable for code review.

This thesis investigated a commercial software solution and analyzed the suitability of its solutions for use by an organization. The starting point was that there would be more and more feasible solutions to be piloted, but there was no one that met the criteria. The work also analyzed two different COBOL programs and compared their results.

It can be concluded from the results of the research that the software solution to be piloted can detect qualitative problems from a large amount of program code, which is not very profitable when manually executed by a human. It is possible to add a software solution plug-in to the programming environment, which allows real-time problem checking when creating new program code. The software solution also offers a unique solution for program code quality management based on the given thresholds.

KEYWORDS: Source code review, Mainframe environment

1. JOHDANTO

Toimivan ohjelmisto it-hankkeen, projektin tai kehityksen ja ylläpidon takana on yhteiset pelisäännöt ohjelmistotuotannon työvaiheista ja metodeista. Vaikka laajojen ohjelmistojen tai ohjelmien tekeminen on aikaa vievää ja haastavaa, sen pilkkominen osiin selkeyttää kehitystyötä. Vaihejakomallit ovat eri tuotantomalleja, jotka ovat osa ohjelmistotuotantoa. Yhtenä suosituimpana vaihejakomallina voidaan pitää vesiputousmallia, joka on helppo omaksua ja jossa edetään vaihe vaiheelta. Vaihtoehtona perinteiselle vesiputousmallille ovat ketterät ohjelmistokehitysmenetelmät, jotka kannustavat jatkuvaan parantamiseen, sekä nopeaan ja joustavaan muutostarpeeseen. Riippumatta siitä, minkälainen ohjelmistojen valmistamisen menetelmä on käytössä, on yhtenä tärkeänä tavoitteena tässä prosessissa tunnistaa ohjelmistosta puutteita tai mitä tahansa sellaisia toiminnallisuuksia, mitkä jollain tavalla poikkeavat odotetusta lopputuloksesta ennen lopputuotteen toimittamista asiakkaalle. Edellä kuvattua prosessia kutsutaan ohjelmistotestaukseksi. Ohjelmistotuotannon alkuvaiheista alkaen on lähdekieliselle ohjelmalle kuitenkin mahdollista tehdä staattista testausta. Staattisen testauksen tavoite on pyrkiä nostamaan ohjelmiston laatua ja havaita vikoja jo aikaisessa vaiheessa, mikä on kustannussyistä kannattavaa. Staattista testausta on mahdollista tehdä joko manuaalisesti tai jollain staattisen testauksen analyysityökalujen avulla, joita tässä työssä tutkitaan.

Ohjelmistokehityksen yhteydessä tehdyillä lähdekielisten ohjelmien katselmoinneilla on tärkeä tehtävä, jotka edesauttavat rakentamaan luotettavamman ja toimivan ohjelman. Lähdekielisen ohjelman katselmoinnit ovat yksi osa staattista testausta ja ovat osoittautuneet poikkeuksellisen kustannustehokkaaksi tavaksi varmistaa ohjelmiston laatu. Nykyisellään eräässä kohdeyrityksessä on käytössään erilaisia dynaamisen testauksen työkaluja. Yksi näistä dynaamisen testauksen tärkeimmistä vaiheista on testauksen lopussa tapahtuva hyväksymistestaus, jossa tärkeimmässä roolissa on tuotteen loppukäyttäjä. Vaikka hyväksymistestauksella varmistetaan toimitettavan tuotteen asetetut vaatimukset ja toimivuus erilaisilla testitapauksilla, se ei kuitenkaan välttämättä kerro ohjelmiston laadukkuudesta, kustannustehokkuudesta, käyttämättömästä lähdekielestä ja kalliista lähdekielisestä ohjelmasta riittävästi.

1.1 Tutkimuksen kohde

Tutkimuksen kohteena on kartoittaa erilaisia valmiita ohjelmistoratkaisuja, jotka pystyvät katselmoimaan ohjelmistokehittäjän COBOL-konekielisen ohjelman laadukkuutta ja näin ennaltaehkäisemään konekielisessä ohjelmassa olevia virheitä ja ohjelmiston laatuongelmia, jotka vaikuttavat negatiivisesti ohjelman ylläpidettävyyteen.

Keskustietokoneet, eli mainframet ovat osa usean suuren yrityksen liiketoimintaa ja ne vastaavat yritysten kriittisistä liiketoiminnoista. Järjestelmä ja ohjelmat ovat yleensä kymmeniä vuosia vanhoja. Tänä aikana ohjelmat ovat paisuneet isoiksi, niitä on pilkottu, yhdistetty isoiksi kokonaisuuksiksi, sekä ne ovat läpikäyneet paljon muutoksia. Ohjelmistokehittäjiä on vuosien varrella ollut erilaisia ja näin ollen myös ohjelmointityylejä. Tämä kaikki on johtanut siihen, että ylläpidosta on tullut entistä hankalampaa ja ylläpitokustannukset ovat nousseet.

On arvioitu, että tänäkin päivänä kahdeksankymmentä prosenttia maailman päivittäisistä liiketoimintatransaktioista toimii yli 60-vuotta vanhan ohjelmointikielen COBOL:n varassa. (Beach 2014) Tyypillisesti mainframella toimivan yrityksen liiketoiminnassa COBOL-ohjelmien lähdekieliset rivimäärät lasketaan miljoonissa ja yhden ohjelman lähdekieliset rivimäärät tuhansissa. Monien tässä kohdeyrityksessä käytössä olevien ohjelmien ensimmäiset kehitysvaiheet on aloitettu noin 30 vuotta sitten ja ne ovat tänäkin päivänä aktiivisessa käytössä. Näiden ohjelmien alkuaikoina ei olla voitu ennakoida tietotekniikan ja liiketoiminnan muuttuvien tarpeiden kehitystä tähän päivään asti, vaikka ylläpitoa on tapahtunut koko ajan tänä aikana. Ei myöskään ole ollut mielekästä, eikä rahallisesti kannattavaa alkaa ohjelmien lähdekieltä käymään läpi manuaalisesti, etsimällä niistä muun muassa virheitä tai huonolaatuisia ohjelmointikäytäntöjä.

Nykyaikana keskustietokoneella toimivat liiketoimintajärjestelmät ovat poikkeuksetta suuria ja kompleksisia kokonaisuuksia, mitkä aiheuttavat haasteita jatkuvalle järjestelmän kehitystyölle. Järjestelmien kompleksisuudesta johtuen ylläpito aiheuttaa huomattavia

kustannuksia. Tästä hyvänä esimerkkinä on eräässä tutkimuksessa, jossa on tutkittu suuren pankin, keskustietokoneella toimivan ohjelmiston kompleksisuutta suhteutettuna kuluihin on arvioitu, että pelkästään COBOL:lla toimivien ohjelmistojen ylläpitoprojektien kulut ovat 60 prosenttia tietotekniikan menoista. (Banker, et al., 1993). Yhtenä merkittävänä kulueränä on mainframen käyttökapasiteetti, joka muodostuu mainframella toimivien ohjelmien ja transaktioiden jatkuvasta ajosta. Käyttökapasiteetti lasketaan MIPSeinä tai MSUna. Samainen laskentatapa on verrattavissa energian kulutukseen (kWh), eli maksetaan siitä mitä käytetään (Mainframes 360 2018). Kuitenkin tyypillisesti on ennalta määritelty tietty kapasiteetti MIPSejä käyttöön ja yli menevästä kapasiteetin käytöstä maksetaan ennalta sovittu korvaus joka MIPSiä kohden mainframe palveluntarjoajalle.

1.2 Diplomityön tavoitteet

Tavoitteena on löytää markkinoilta valmis ohjelmistoratkaisu, joka on soveltuva COBOL-lähdekielisten ohjelmien katselmointiin ja staattiseen testaukseen kohdeyrityksen mainframe-ympäristössä. Ohjelmistoratkaisun avulla olisi tarkoitus pystyä nostamaan yleistä laatua ylläpidettävissä lähdekielisissä ohjelmissa ja parantamaan ohjelmistokehittäjien ymmärrystä ohjelmien ylläpitotehtävissä hyvistä ohjelmointimenetelmistä.

On tärkeää päästä pilotoimaan näitä erilaisia ohjelmistoratkaisuita ja päästä mittaamaan niistä saatavia hyötyjä, koska ne ovat organisaatioille kalliita investointeja ja oletusarvoisesti on vaikea tehdä suoria johtopäätöksiä eri ohjelmistoratkaisusta mainospuheita lukemalla. Keskuskonejärjestelmät ovat yksilöllisiä eikä näin ollen voida olettaa, että jonkin toisen järjestelmän onnistunut pilotointi toimisi juuri toisessa vastaavanlaisessa keskuskonejärjestelmässä. Tällaisella toivotulla ohjelmistoratkaisulla on mahdollista saavuttaa merkittäviä hyötyjä, joita ovat rahan säästäminen, keskustelun rakentaminen hyvistä ohjelmointikäytännöistä ja yrityksen sisäisten ohjelmointistandardien kehittäminen.

1.3 Diplomityön rakenne

Tutkimuksen toisessa luvussa tutustutaan keskuskoneeseen ja sen ympäristöön tutkimuksen kannalta tarvittavalla tasolla. Tässä luvussa tarkastellaan sen keskustietokoneen käyttökohteita ja tuodaan esiin asioita, miksi keskuskoneympäristössä ohjelmointi kehitys on sellaista, kun se on.
Työn kolmannessa luvussa käydään läpi organisaation käyttämät vaihejakomallit keskuskoneympäristössä.
Työn neljännessä luvussa käydään yleisellä tasolla läpi ohjelmistotestausta ja syvennytään tarkemmin lähdekielisten ohjelmien katselmointeihin ja sen vaiheisiin manuaalisesti. Luvussa tarkastellaan myös tarkemmin erilaisia ohjelmistometriikoita ja pohditaan niiden soveltuvuuksia COBOL-ohjelmoinnissa.
Työn viidennessä luvussa käsitellään lähdekielisen ohjelman laadun arvioimiseen vaikuttavia seikkoja ISO/IEC kansainvälisen standardin mukaan.
Työn kuudennessa luvussa tuodaan esille kriteereitä, joita pilotoitavan ohjelmistoratkaisun täytyi täyttää ja jotka vaikuttivat pilotoitavan ohjelmistoratkaisun valintaan. Seuraavassa alaluvussa käydään läpi tarkemmin tutkimuksessa tehdyt käytännön asiat ja esitellään käytettävä aineisto. Tämän luvun viimeisessä vaiheessa esitellään tutkimuksesta saadut tulokset ja havainnot. Tässä tuodaan esille pilotoitavan ohjelmaratkaisun tuottamat analyysit, jotka pohjautuvat käytettyyn aineistoon ja pilotoitavan ohjelman tarjoamat ominaisuudet sen käyttäjälle.
Työn seitsemännessä luvussa analysoidaan ohjelmistoratkaisun käyttöönottoon liittyviä asioita. Työn kahdeksannessa luvussa esitetään johtopäätökset tehdystä tutkimuksesta.

2. KESKUSTIETOKONE JA SEN YMPÄRISTÖ

Tässä kappaleessa tutustutaan IBM:n keskustietokoneeseen ja sen ympäristöön käymällä läpi lyhyesti sen historia, sen nykyiset käyttökohteet ja sen vahvuudet ja haasteet. IBM itse määrittää mainframen tarkoittavan suurta tietokonetta, johon muut tietokoneet voidaan liittää, jotta ne voivat jakaa sen mahdollisuudet kuten ladata tietoja (download), siirtää tietoja (upload), joita itse mainframe tarjoaa. Välillä kuulee myös puhuttavan isosta raudasta (big iron), jotka kaikki tässä asiayhteydessä tarkoittaa suomeksi keskustietokonetta tai suurtietokonetta.

2.1 Historia

IBM:n rakentama ensimmäinen yleiskäyttöinen automaattinen digitaalinen tietokone valmistui vuonna 1944. Se oli sähkömekaaninen kone ja sitä kutsuttiin automaattiseksi sekvenssikontrolloiduksi laskimeksi. Siltä onnistui yhteenlaskut kolmannes sekunnissa ja kertolaskut kuudessa sekunnissa. Tällä laskimella oli pituutta noin 15 metriä. Korean sodan aikaan, mikä itsessään vauhditti laajamittaisten tietokoneiden kehitystä 1950-luvulla, IBM julkisti ensimmäisen täysin sähköisen tietojenkäsittelyjärjestelmänsä. Tämän koneen nimi oli IBM 701, jonka toinen tunnettu nimi on puolustuslaskin (Defense Calculator), joka oli kehitetty lähinnä sotilaskäyttöön (ibm.com 2018). Tämän jälkeen 50-luvulta aina 60-luvun loppuun asti IBM:n 700/7000-sarjan mainframet mahdollistivat kaupallisen ja tietojenkäsittely (data processing) käytön suurissa organisaatioissa. Alun perin IBM myi tietokoneet ilman minkäänlaisia ohjelmistoja, odottaen asiakkaiden kirjoittavan omat tarvittavat ohjelmat. Myöhemmin IBM toimitti kääntäjät (compilers) uusille korkeamman tason ohjelmointikielille kuten fortranille, COMTRANille ja myöhemmin COBOLille (wikipedia.org 2018). Organisaatioissa, joissa keskustietokonetta käytetään, on tänä päivänäkin vielä aktiivisessa käytössä ohjelmointikielet fortran, joka sopii erityisesti numeeriseen laskentaan ja COBOL, jolla kirjoitetaan sovellukset.

Ennen kuin sovelluskehittäjillä oli käytössään päätteet, ohjelmointi tapahtui reikäkorttien avulla, joskin hyvin vaivalloisesti verrattuna nykyaikaiseen ohjelmointiin tietokoneilla.

Reikäkortti vastasi yhtä lähdekielisen ohjelman riviä ja ohjelmoitaessa ohjelmaa reikäkortteja saattoi olla paljon pinossa, jolloin ne oli myös mentävä kääntäjästä läpi. Ohjelman kääntämiseen menevä aika ja kääntämisessä tulleiden virheiden korjailuun saattoi helposti upota useita tunteja aikaa (Kangasniemi 2018). Reikäkortteja käytettiin myös digitaalisen datan tallentamiseen ennen kuin magneettiset tallennusvälineet tulivat ja korvasivat reikäkortit.

Voisi helposti kuvitella aikaa, jolloin ohjelmoitiin reikäkorteilla, että oman ohjelmoinnin oikolukemiseen käytettävä aika oli ollut nykyistä suurempaa, koska nykyaikana ohjelmien kääntäminen ja niiden virheiden korjailu on niin paljon nopeampaa. Toki on otettava huomioon, että ennen ohjelmien koossa oli myös tietty kapasiteetti, joka oli hyvinkin rajallinen verrattuna nykyaikaan. Näin oli myös huomioitava erilaiset ratkaisut ohjelman suunnittelussa. Reikäkorttien avulla ohjelmoidut ohjelmat ovat nykyäänkin aktiivisessa käytössä organisaatioissa, vaikkakin niihin on vuosien varrella tullut muutoksia ja ne ovat kasvaneet suuremmiksi.

2.2 Nykyaikainen keskustietokone ja ohjelmointiympäristö

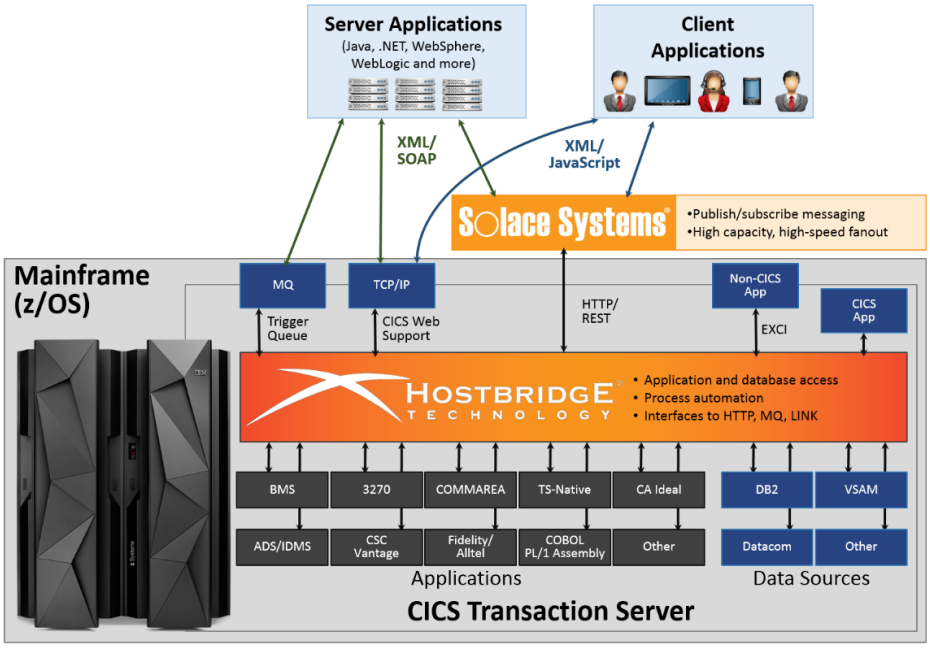
Keskustietokoneen historiaa katsoessa on keskustietokoneen fyysinen koko ajansaatossa jatkuvasti pienentynyt, mutta tehoa on koko ajan tullut lisää.

Nykyaikainen IBM:n z14 keskustietokone (Kuva 1.) kykenee parhaimmillaan 32 teratavulla muistia käsittelemään 12 miljardia tapahtumaa, eli transaktioita yhdessä päivässä. Tällaiset transaktiot voivat esimerkiksi olla luottokorttitapahtumia, jotka käsitellään keskustietokoneilla. Esimerkkinä Citi pankki on ilmoittanut käyttävänsä IBM keskustietokoneita heidän transaktioissaan. Citi pankin keskustietokoneet prosessoivat 150 000 transaktioita sekunnissa (nanalyze.com 2017).

Kuva 1. IBM z14 kuvattuna ulkoa ja sisältä (nanalyze.com)

Vaikka moderneissa keskustietokoneissa on paljon laskentakapasiteettiä sitä ei voida verrata supertietokoneisiin. Luonteeltaan supertietokoneissa lasketaan monimutkaisia matemaattisia laskuja, jotka vaativat paljon tehoa tietokoneen raudalta (hardware). Keskustietokoneessa taas käsitellään transaktioita, joita on todella paljon, mikä sekin vaatii laskentatehoa, mutta transaktiot itsessään ovat yksinkertaisia käsitellä.

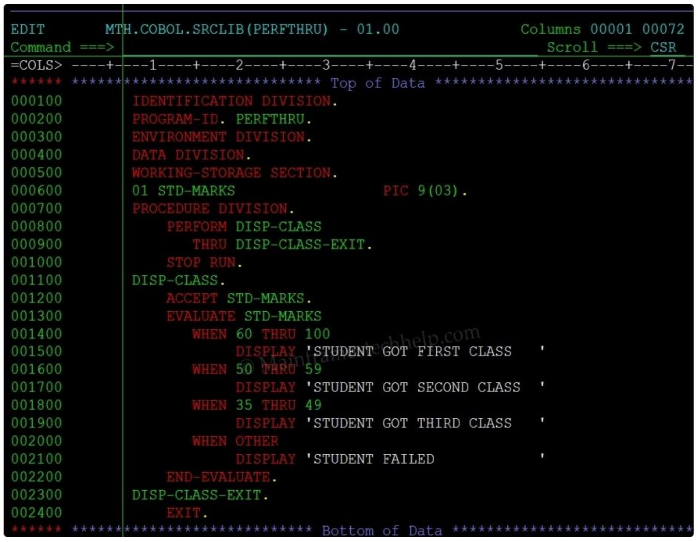
Nykyisin IBM:n keskustietokoneiden käyttöjärjestelmänä on 64-bittinen Z/OS, joka perustuu sen edeltäjäänsä vuonna 1995 julkaistuun OS/390 käyttöjärjestelmään. Uusimmat Z/OS versiot tukevat nykyisin muun muassa moderneja ohjelmointirajapintoja ja suosittuja ohjelmointikieliä kuten Java ja C-kieli.
Tietokannan hallintajärjestelmänä Z/OS käyttöjärjestelmässä on IBM:n kehittämä relaatio tietokanta DB2 (tietokanta 2). Usein tietokantana on myös käytössä vanhempia hierarkkisia tietokantoja kuten IMS (tiedon hallinta järjestelmä) ja DL/I (kuvauskieli hierarkkisille tietokannoille)



Kuva 2. Solacen näkemys modernin keskuskoneympäristön arkkitehtuurista. (solace.com 2018)

Keskustietokoneen pääkäyttäjät käyttävät ohjelmia, ajavat eräajoja tai tekevät tietokanta hakuja TSO:n avulla (kuva 3), joka mahdollistaa interaktiivisen pääteyhteyden Z/OS järjestelmään. Nämä pääkäyttäjät voivat olla esimerkiksi teollisuudessa olevia työntekijöitä tai lentokenttävirkailijoita. Toisena vaihtoehtona on käyttää jotain nykyaikaisempaa graafista käyttöliittymää kuten web-käyttöliittymää. Nykyisin kuluttajat, jotka tietämättään käyttävät sovelluskerrosta, joka kutsuu keskustietokonetta vaikkapa pankkiasioissa, on heillä käytössään jokin moderni käyttöliittymä.

Sovelluskehityksessä TSO:ta käytetään yleisesti ohjelmistotestauksessa ohjelmien ja eräajojen ajamiseen. Tämän lisäksi sen avulla tehdään myös virheenjäljitystä ja lähdekielisen ohjelman kielen editointia.



Kuva 3. TSO-ikkuna, jossa esillä COBOL-ohjelmointikieltä (mainframestechhelp.com 2018).

Ohjelmistokehityksessä TSO on osittain korvattu IBM:n Eclipse-pohjaisella integroidulla kehitys ja ylläpitoympäristöllä nimeltään IDz, joka tarjoaa nykyaikaisemmat työkalut ohjelmistokehitykseen keskuskoneympäristössä.

Kangasniemen mukaan keskuskoneympäristön sovelluskehitys- ja ylläpitotyössä periaatteet ovat samat kuin 30 vuotta sitten, vaikka uusia työkaluja on tullut ja ohjelmat ovat kehittyneet. Uusien ohjelmien luominen alusta asti on yksinkertaista käyttäen hyväksi toimiviksi todettujen vanhempien ohjelmien osia niitä yhdistelemällä ja muokkaamalla. Haasteellisempaa on lähteä tekemään muutoksia vanhempiin kompleksisiin ohjelmiin, jotka ovat tyypillisesti noin 8 000-20 000 ohjelmointiriviä pitkiä ohjelmia. Tähän on ollut apuna se, että ohjelmistokehittäjällä on kokemusta tästä ohjelman kehittämisestä 20-30 vuoden verran. Kangasniemen mukaan COBOLilla ohjelmointi ja sen ymmärtäminen ei ole vaikeaa vähänkään ohjelmistokokemusta omaavalle henkilölle. Tärkeimpänä hän pitää sovellus- ja ylläpitotyössä asiakkaan liiketoiminnan ymmärtämistä ja kokonaisuuden hahmottamista.

Vähemmän kokemuksen omaaville ohjelmistokehittäjille on ollut apua uudemmista työkaluista, jotka vastaavat käytettävyydeltään nykyaikaista ohjelmointiympäristöä kuten IBM:n IDz ohjelman kokonaisuuden hahmottamisessa, asiakkaan liiketoiminnan ymmärtämisessä, ohjelmistojen kehitys- ja ylläpitotehtävissä.

Keskuskone on tyypillisesti huomattava kuluerä yritysten it-kustannuksista, joiden kriittisestä liiketoiminnasta se vastaa. Tällaiset kulut ovat ohjelmistojen lisenssit, erilaiset sopimukset, keskuskoneen operationaaliset kustannukset, sekä järjestelmän jatkuva ylläpito ja kehittäminen. Huolimatta siitä, että keskuskoneen sisältyvät kulut ovat kalliita sitä tarvitseville yrityksille ja sen ylläpito ja kehittäminen on hankalaa, sillä on ollut paikkansa isoissa organisaatioissa huolehtimassa niiden elintärkeistä liiketoiminnoista vuosikymmeniä tauotta.

3. OHJELMISTOTUOTANTO

Tässä luvussa tarkastellaan ohjelmistotuotannon eri tuotantomalleja ja lopuksi tarkastellaan niiden suurimpia eroavaisuuksia toisistaan. Näitä eri ohjelmistotuotannon tuotantomalleja voisi kuvailla myös eri strategioiksi, koska ne ovat pohjimmiltaan suunnitelmia, joilla pyritään saavuttamaan yhteinen päämäärä asiakkaan ja ohjelmiston tuottajan näkökulmasta. Näitä päämääriä yleisesti ovat asiakas ja toimittajat yhtä mieltä siitä, että tuote tekee sen mitä pitää ja toimitus ajallaan.

Ohjelmistojärjestelmien kehittäminen etenee tyypillisesti laajasti määritellyistä asiakkaiden tarpeista, määrittelystä kuinka nämä tarpeet suunnitellaan järjestelmään sekä fyysisen kokonaisuuden rakentaminen, joka on itse järjestelmä. Lähdekielisen ohjelman kieli perustuu myös määrittelyyn siitä, miten asiakkaiden tarpeet on pantava täytäntöön. (Donaldson & Siegel 2000 s.45)

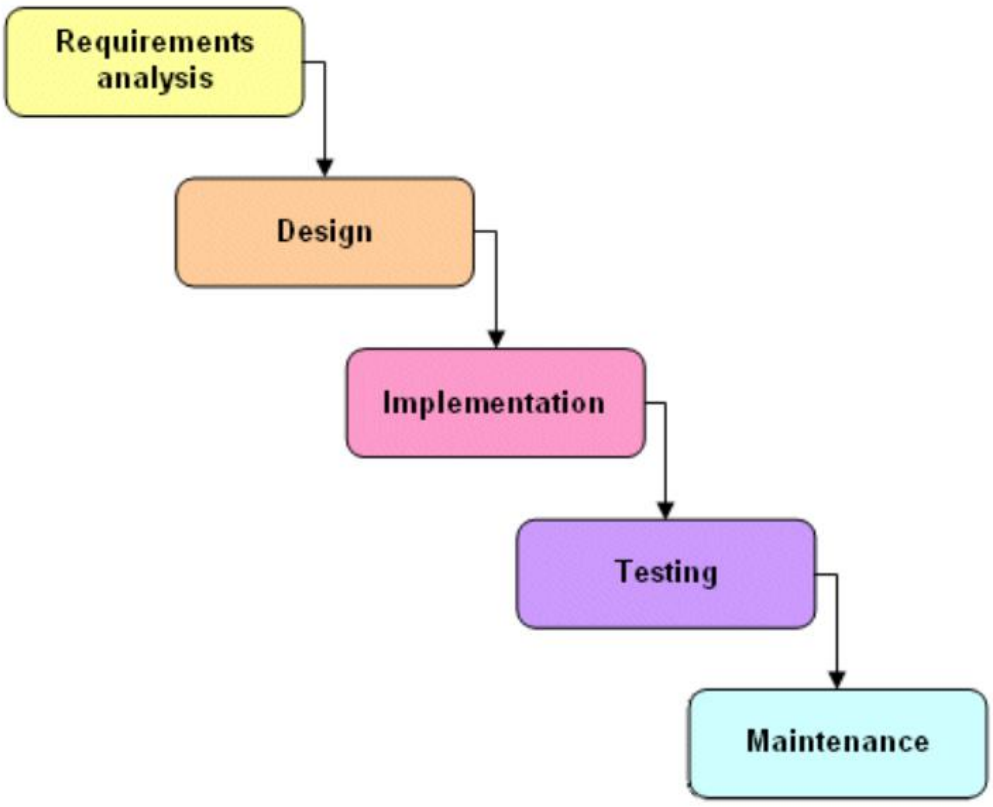
3.1 Vaihejakomallit

Ohjelmistojen kehittämiseen ei ole olemassa yhtä ainoaa oikeaa tapaa, mutta riippuen asiakkaiden tarpeista, ympäristöstä, asiakkaan liiketoiminnasta, käytännöistä, työkaluista, järjestelmistä, joita kehitetään voivat jotkin vaihejakomallit sopia tähän tehtävään paremmin kuin toiset.

3.1.1 Vesiputousmalli

Vesiputousmallia (waterfall model) pidetään ensimmäisenä varsinaisena prosessimallina. Sen isänä pidetään Winston Roycea, joka 1970 kirjoitti artikkelin “Managing the Development of Large Software Systems”. Aikanaan vesiputousmalli saavutti nopeasti suuren suosion, mutta se on tänäkin päivänä yksi suosituimmista vaihejakomalleista huolimatta

siitä, että nykyään on paljon erilaisia vaihejakomalleja tarjolla. Tähän on saattanut vaikuttaa se, että malli on käyttäjilleen selkeä ja helposti omaksuttava, eikä vaadi projektin osallistujilta erityistaitoja. Vesiputousmallin luonne on jakaa koko prosessi lineaarisiin vaiheisiin, jossa edellisen vaiheen tulos on seuraavan vaiheen syöte (cs.helsinki.fi 2009). Hieman erilaisia variaatioita on olemassa vesiputousmalleihin, eikä tässäkään ole olemassa sitä yhtä oikeaa, mutta kuitenkin pääpiirteet pysyvät samankaltaisina kaikissa. Kuvan 4. mukaisesti vesiputousmallin vaiheet ovat vaatimusmäärittely (requirements analysis), suunnittelu (design), jossa tehdään korkean ja matalan tason suunnittelu. Täytäntöönpano (implementation), joka sisältää ohjelmoinnin ja siihen kuuluvat lähdekielisen ohjelman kielen tarkastukset. Testaus (testing), joka sisältää integraatiotestauksen, komponenttitestauksen, järjestelmätestauksen ja lopuksi hyväksymistestauksen. Ylläpito (maintenance).

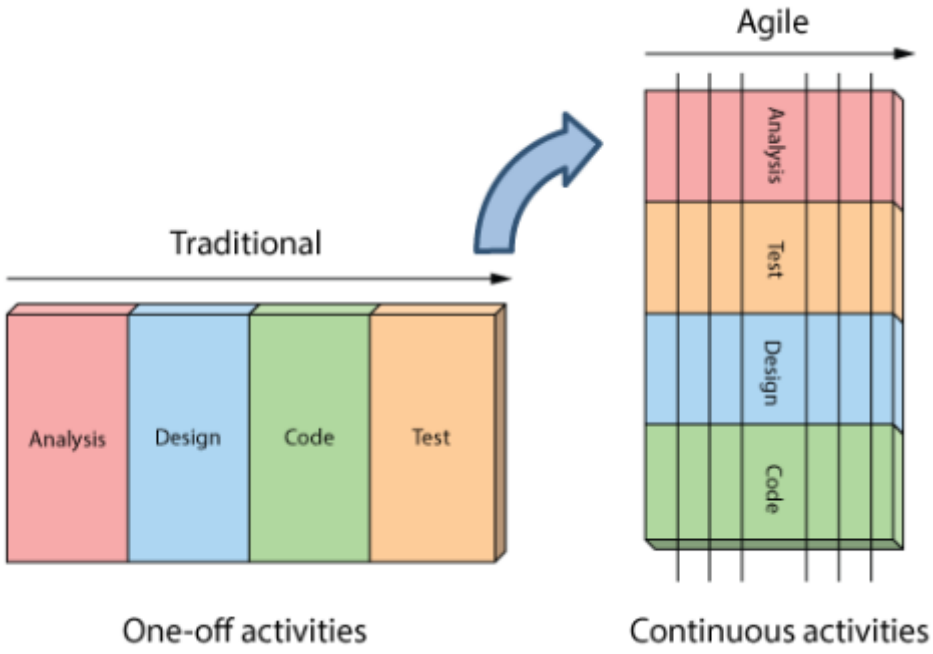


Kuva 4. Vesiputousmalli (technologyuk.net 2018).

3.1.2 Ketterät menetelmät

Ketterien menetelmien painopisteet ovat yksinkertaisuus ja nopeus. Kehitystyössä sen tekijät keskittyvät ensisijaisesti vain tehtävien tarpeeseen, niiden nopeaan toimittamiseen, palautteen keräämiseen ja reagoiminen kerättyyn palautteeseen (Abrahamsson, et al., 2002 s.17).
Tunnusomaisena piirteenä ketterissä menetelmissä on tapana jakaa ohjelmistokehitys lyhyisiin iteraatioihin, jossa iteraation kesto on noin 4 viikkoa. Tästä syystä se tuottaa konkreettisia tuloksia jo muutamassa viikossa. Iteraation aikana käytävät vaiheet ovat iteraation suunnittelu, vaatimusmäärittely, ohjelmistonsuunnittelu, ohjelmointi, testaus ja dokumentointi. Käytännössä iteraatiot ovat hyvin samankaltaisia kuten koko vesiputousmalli, mutta paljon pienemmässä mittakaavassa. Näiden lyhyiden iteraatioiden lopussa on kuitenkin aina päämääränä saada valmiiksi jokin toimiva kokonaisuus, ei siis koko tuotetta kuten vesiputousmallissa.

Agile manifestissa (Manifesto for Agile Software Development 2018) on määritelty neljä arvoa, joita ovat arvostaminen yksilöitä ja vuorovaikutusta enemmän kuin prosesseja ja työkaluja, toimivaa sovellusta enemmän kuin kokonaisvaltaista dokumentaatiota, asiakasyhteistyötä enemmän kuin sopimusneuvotteluita, muutokseen reagoimista enemmän kuin suunnitelman noudattamista.



Kuva 5. Agilen ja vesiputouskehitysmallien erot (agilenutshell.com 2018).

Kuten kuvasta 4 voi nähdä, ohjelmistokehitysprosessi on jaettu vesiputousmallissa (traditional) eri vaiheisiin alusta loppuun, kun taas ketterissä menetelmissä (kuva 5) nämä samat vaiheet on pilkottu pienempiin helpommin hallittaviin osiin tarjoten riskien pienentämistä ja joustavuutta.

On todettu, että ketterät menetelmät on suunniteltu tekemään yksinkertaisia ratkaisuja, jolloin on tehtävät muutokset ovat helpompi toteuttaa jälkeenpäin. Samalla, kun suunnittelun laadun parantaminen on jatkuvaa, on se silloin halvempaa toteuttaa. Samaa pätee myös testaukseen, joka on jatkuvaa. Tämä johtaa siihen, että viat on mahdollista huomata ja korjauttaa ajoissa, mikä on ohjelmistokehityksen kannalta halvempaa (Abrahamsson, et al., 2002 s.15). Agile on ennen kaikkea filosofia, joka ei nojaa yhteen tiettyyn toimintatavan toteutukseen, joita on monia. Keskuskoneympäristöissä on myös paikkansa agile menetelmille, sen periaatteille ja filosofialle pystyäkseen ennakoimaan nopeisiin muutoksiin ja ylläpitämään omaa kilpailukykyä.

4. OHJELMISTOTESTAUS

Tässä luvussa otetaan yleiskatsaus ohjelmistotestauksesta ja siinä käytettävistä käytännöistä. Tämän luvun lopussa tarkastellaan tarkemmin staattisen testauksen menetelmiä.

Nykyään normaalissa elämässä likimain kaikki ihmiset käyttävät ohjelmistoja jollain tasolla joka päivä, vaikka he eivät välttämättä itse tiedä sitä. Ohjelmistojen olemassaolo ei pelkästään rajoitu esimerkiksi perinteiseen pöytätietokoneeseen tai matkapuhelimeen. Tänä päivänä on aivan normaalia, että jonkin tason ohjelmistoja on lähes joka paikassa, jotka liittyvät meidän normaaliin päivittäiseen elämään kuten astianpesukoneissa, rannekelloissa, liikennevaloissa, autoissa ja maksukorttiliikenteessä. Moni henkilö on kuitenkin saattanut kokea tilanteita, jolloin ohjelma ei välttämättä ole toiminut niin kuin se olisi kuulunut. Tällainen tilanne voi olla, kun verkkosivu ei lataudu oikein, jonkin laitteen käyttöliittymä ”jäätyy” paikalleen tai ongelmia maksuliikenteessä. Huomattavaa on, että ohjelmistoista aiheutuvien ongelmien riskit ovat erilaisia riippuen ohjelmistojen käyttökohteista. Esimerkkinä rannekellon ohjelmiston virhetilanne on aivan toisenlainen, kun että liikennevalot näyttäisivät virheellisesti, jolloin ihmishenki on mahdollisesti vaarassa. Ohjelmistoja mietittäessä on selvää, että ihminen tekee virheitä. Tämä johtuu usein siitä, että henkilöllä ei ole tarpeeksi kokemusta, oikeaa informaatioita, on tapahtunut väärinymmärrys, piittaamattomuutta, väsymys tai aikataulupaineet.

Kun mietitään mitä voi mennä vikaan, nousee esiin muutama asia kuten ohjelmiston määrittelyyn, suunnitteluun ja toteutukseen liittyvät virheet. Nämä virheet aiheuttavat itse ohjelmiston kehittäjät. Virheitä syntyy myös järjestelmän käytössä, joka voi olla käyttäjistä johtuvaa. Ympäristöolosuhteet aiheuttavat myös virheitä ja tahalliset vahingot ohjelmistoa kohtaan. (Graham, et al., 2008: s.7)

Ohjelmistotestaus on prosessi, jossa on tarkoitus tunnistaa ohjelmasta, että vastaako tulokset aiemmin määriteltyjä odotuksia, ja saada varmistus siihen, että ohjelmisto täyttää ne vaatimukset, jotka sille on aiemmin määritelty. Testauksessa on tarkoitus löytää ohjelmasta virheitä puutteita tai todeta, että niitä ei ole, mutta ei kuitenkaan voida todistaa, että

virheitä ei ole. Käytännössä dynaamisessa testauksessa testattavaa ohjelmaa tai sen osia, yksin tai erikseen ajetaan läpi erilaisilla ja mahdollisimman monilla testitapauksilla. Toinen tapa on testaus ilman lähdekielisen ohjelman ajamista, tätä kutsutaan staattiseksi taukseksi.

> *"Software testing is easier, too, in some ways, because the array of software and operating systems is much more sophisticated than in the past, providing intrinsic, well-tested routines that can be incorporated into applications without the need for a programmer to develop them from scratch."* (Myers, et al., 2011: s.2)

Perusperiaate ohjelmistotestauksen prosessista kaikilla testaus tasoilla on kehittynyt vuosien aikana. On kyse sitten millä tahansa tasolla olevasta testauksesta, pääpiirteet ovat kuitenkin hyvin samankaltaisia, lukuun ottamatta tiettyjä muodollisia prosesseja kuten esimerkkinä dokumentointi voi olla vähäisempää. Päätös siitä kuinka muodollisella tasolla prosessi käydään läpi, riippuu pitkälti testattavan järjestelmän ja ohjelmiston sisällön liittyvän riskin tasoon. Perusperiaatteeseen jaetut toiminnot kuuluvat seuraavat askeleet: suunnittelu ja valvonta, analyysi ja hahmotelma, täytäntöönpano ja suoritus, arviointi poistumiskriteereistä ja raportointi ja lopuksi testitoimintojen päättäminen. (Graham, et al., 2008: s. 23)

Taulukossa 1 on esiteltynä seitsemän eri ohjelmistotestauksen periaatetta, jotka tarjoavat yleiset ohjeet kaikelle testaukselle.

Taulukko 1. Testauksen periaatteet (Graham, et al., 2008: s. 22).

Periaate 1:	Testauksella voidaan osoittaa, että vikoja on.	Mutta ei voida todistaa, että vikoja ei ole. Testaus toki vähentää todennäköisyyttä tuntemattomien virheiden löytämiseen, mutta vaikka virheitä ei löydy se ei ole todiste ohjelman oikeellisuudesta.
Periaate 2:	Täydellinen testaus on mahdotonta.	Kaikkien mahdollisten yhdistelmien, syötteiden ja ennakkoehtojen testaaminen ei ole mahdollista. Sen sijaan, riskien ja asioiden priorisointiin keskittäminen on mahdollista.
Periaate 3:	Testaaminen aikaisin	Testaus on aloitettava mahdollisimman aikaisin ja sen pitäisi keskittyä määriteltyihin tarkoituksiin.

Jatkuu..

Jatkuu..

Periaate 4:	Vikojen kasaantuminen	Pieni määrä moduuleista sisältää usein eniten virheitä.
Periaate 5:	Hyönteismyrkkyparadoksi	Kun sama testi toistetaan uudelleen ja uudelleen, lopulta uusia vikoja ei enää löydy. Pääsemällä yli tästä paradoksista on testitapaukset päivitettävä.
Periaate 6:	Testaus on tilanneriippuvaista	Testaus tapahtuu eri tavalla eri yhteyksissä. Esimerkiksi turvallisuuden kannalta kriittinen ohjelmisto testataan eri tavoin kuin ei niin kriittinen verkkokauppasivusto.
Periaate 7:	Virheettömyyden harhaluulo	Vikojen etsiminen ja korjaaminen ei auta, jos rakennettu järjestelmä on käyttökelvoton tai ei vastaa käyttäjien tarpeita tai odotuksia.

4.1 Ohjelmistotestauksen historiaa

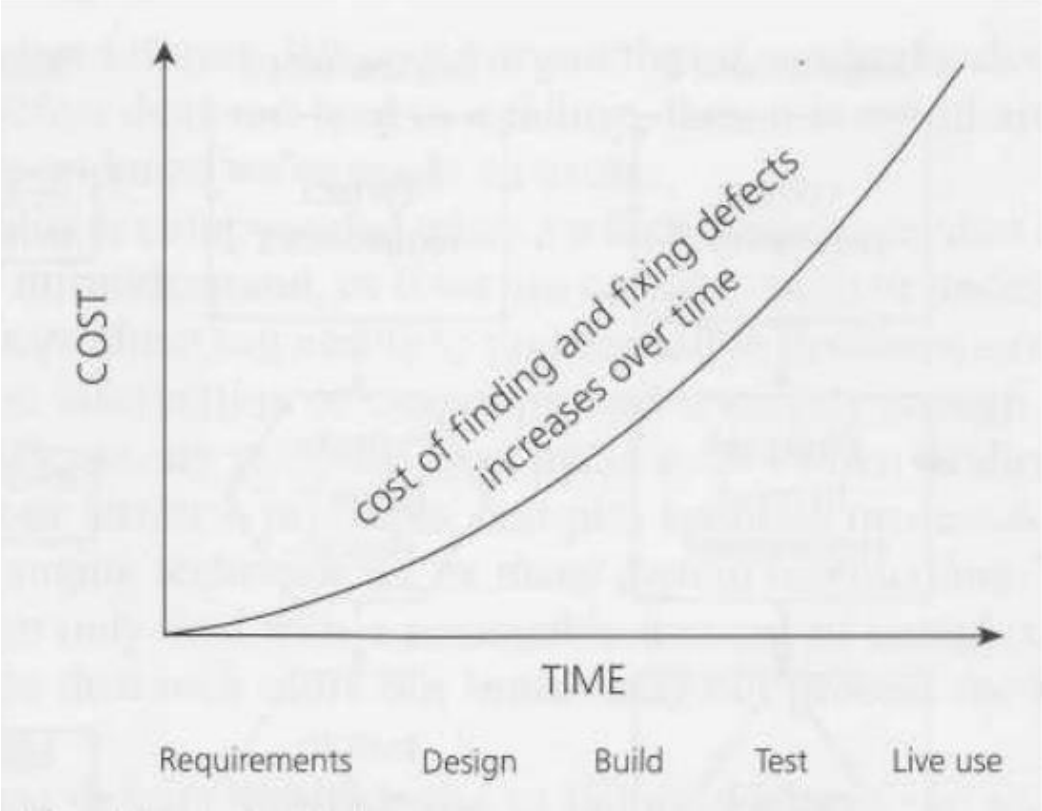
Vuoteen 1956 asti ei ollut selvää eroa ohjelmistotestauksella tai virheenjäljityksellä. Siihen asti keskittyminen oli virheenjäljityksen puolella, eli virheidenkorjauksessa (debug). Vuoden 1957-1978 aikana, jolloin virheenjäljitys ja testaus erotettiin toisistaan – tänä aikana osoitettiin, että ohjelmisto täyttää sille asetetut vaatimukset. Vuoden 1980 aikana oli tavoite löytää virheitä ja näin ollen painopiste oli ohjelmistojen laadussa. Vuosien 1983-1987 välinen aika on luokiteltu arvioinnin suuntaiseksi ajaksi, missä keskityttiin ohjelmiston elinkaaren aikana tuotteen arviointiin ja laadun mittaukseen. Vuoden 1988 lähtien testauksen näkökulman suunta on muuttunut ennaltaehkäiseväksi, missä on virheitä ja vikoja on ollut tarkoitus estää ennen kuin, ne on tapahtunut. (Wikiversity.org 2018)

Ohjelmistotestauksen alkuajoista nykyaikaan on tapahtunut paljon. Ohjelmistojen koko alkuajoista lähtien on kasvanut paljon ja tästä syystä monimutkaistunut. Samaan aikaan kuitenkin ohjelmistotestaajien työkalut ovat kehittyneet tehokkaimmaksi ja mukaan on tullut myös testiautomaatio, mikä helpottaa testaajien työtä.

4.2 Staattinen testaus

Tässä luvussa on tarkoitus tutustua staattiseen testaukseen ja sen tuomiin hyötyihin ohjelmistokehityksessä. Tässä keskitytään tarkemmin lähdekielisten ohjelmien katselmointeihin ja sen avulla ohjelmiston laadun parantamiseen.

Staattisessa testauksessa itse lähdekielistä ohjelmaa ei tarvitse ajaa, kuten dynaamisessa testauksessa. Sen etuna on myös se, että se voidaan aloittaa hyvin aikaisessa vaiheessa ohjelmiston kehityksessä, jolloin virheiden ennaltaehkäisy on mahdollista. Mitä aikaisemmassa vaiheessa havaitaan ja korjataan ohjelmistossa olevat virheet ja ongelmat sitä halvempaa niiden korjaaminen on.



Kuva 6. Virheiden hinta ajan myötä (Graham, et al., 2008: s.9).

Vaikka staattisen testauksen yhteydessä on mahdollista löytää virheitä aikaisessa vaiheessa ja nostaa ohjelmiston laatua, sillä ei kuitenkaan voida korvata dynaamista testausta, koska molemmat tavat ovat taipuvaisia löytämään erilaisia vikoja. Staattisen testauksen aikana on helpointa löytää mm. poikkeamia standardeista, puuttuvia vaatimuksia, suunnitteluvirheitä ja ei ylläpidettävää ohjelman lähdekieltä. Aiemmissa tutkimuksissa on myös pystytty osoittamaan staattisen testauksen hyötyjä, kuten lisääntynyt tuottavuus ja tuotteen laadun nostaminen. Tämä on myös vaikuttanut vähäisempään ajan kulutukseen, jota on joutunut käyttämään testauksessa ja ylläpidossa. (Graham, et al., 2008: s.60).

4.3 Lähdekielisen ohjelman katselmointi

Tässä luvussa tarkastellaan syvemmin lähdekielisen ohjelman tarkastusmenetelmää, joka on yksi ohjelmistokatselmoinnin tyypeistä.

IEEE1028-2008 standardin mukaan (IEEE Std 1028-2008 2009) katselmoinnit (review), läpikäynnit (walkthrough) auditoimiset (audit) tai tarkastamiset (inspection) ovat erilaisia laadunvarmistuksen menetelmiä, joilla on omat luonteenomaiset piirteet ja tavoitteet. Lähdekielisen ohjelman tarkastamisella (code inspection) usein viitataan yleisesti laajasti tunnettuun laadunvarmistusmenetelmään, jolla on tarkoitus etsiä poikkeavuuksia, tarkastaa päätöslauselmia ja pystyä todentamaan tuotteen laatu. Tämä prosessin ensimmäisen tunnetun muodon on kehittänyt 1970-luvulla Michael Fagan. Katselmointitekniikat vaihtelevat muodollisista epämuodollisiin. Fagan (Fagan 1986 s.744) on itse tämän muodollisen prosessin nimennyt tarkastamisella (inspection) ja todennut, että epämuodollisesta tarkastamisesta siirtyminen muodolliseen on aiheuttanut paljon turhautuneisuutta organisaatioissa verrattuna niihin, jotka ovat aloittaneet suoraan muodollisella tarkastuksella. Koulutusnäkökulmaa on myös korostettu ja sille annetaan vahvaa painoarvoa, joka on sisällytetty prosessin alkuun. Raportointi ja analysointi tulokset ovat tärkeitä, jotta virheentunnistuksen tehokkuutta voidaan parantaa jatkuvasti. Lisäksi Fagan korostaa, että tarkastuksen suorittaminen edellyttää asianmukaista koulutusta, jotta tarkastusprosessi voidaan hoitaa oikein (Harjumaa 2005). Faganin mukaan positiivista palautetta on tullut kehittäjiltä, jotka ovat olleet itse tarkastusprosessissa mukana. Katselmoinnit ovat parhaimmillaan vaatimusten validoinnissa ja kokonaisuuksien hallinnassa, koska niitä voidaan tehdä ilman suoritettavaa lähdekielistä ohjelmaa (Paakki 2015).

Perinteinen ohjelmistojen tarkastusprosessi voidaan suorittaa muodollisesti tai epämuodollisesti ja tyypillisesti siihen kuuluu suunnittelu, valmistelu, tapaaminen, uudelleen työstäminen ja seuranta. Vaikka alun perin tämä alkuperäinen tarkastusprosessi esitettiin 1970-luvulla Micheal Faganin toimesta, on siitä lähtien kehitelty erilaisia variaatioita vuosien aikana tutkijoiden toimesta. Alun perin tarkastusprosessia käytettiin vain lähdekielisen ohjelman tarkastuksessa, mutta sitä on sovellettu myös laajasti erilaisiin dokumenttityyppeihin, kuten vaatimusmäärittelyissä ja suunnitteluasiakirjoissa. (Harjumaa 2005).

Muodolliseen ohjelmistojen tarkastusprosesseissa on myös esillä selkeä roolitus, koska se selkeästi auttaa tarkastuksien suorittamisessa ja toivottujen tuloksien saamisessa katsottaessa pitkällä tähtäimellä. Nämä roolit ovat johto, moderaattorit ja tarkastuksiin osallistujat. Tarkastusprosessiin kuuluu kuusi eri vaihetta (Graham, et al., 2008: s.61)., jotka ovat

1. Suunnittelu (planning)
2. Aloittaminen (kick-off)
3. Valmistelu (Preparation)
4. Tarkastelukokous (review meeting)
5. Uudelleen työstäminen (rework)
6. Seuranta (follow-up)

Suunnitteluvaiheessa (planning) tyypillisesti moderaattorin vastuulle on annettu huolehtia päivämääristä, ajasta, paikasta ja kutsuista. Projektitasolla projektin suunnittelu pitää huolen siitä, että kehittäjille on varattu tarpeeksi aikaa osallistua tarkastuksiin. Moderaattorin tehtävä on toteuttaa tässä vaiheessa niin kutsuttu aloitus tarkastus (entry check) ja määrittää muodollinen poistumiskriteeri (exit criteria). Aloitustarkastuksessa on tarkoitus varmistaa, että tarkastuksien suorittajien aika ei kulu turhaan sellaisiin dokumentteihin, jotka eivät selvästikään ole valmiita tarkastuksiin tai jotka sisältävät liikaa selviä virheitä. Aloitustarkastukseen on mahdollista määrittää muitakin oleellisia kriteereitä. Poistumiskriteerit ovat ennalta määriteltyjä kriteereitä tai vaatimuksia, jotka on täytyttävä ennen siirtymistä seuraavan vaiheeseen. Tarkastusprosessin parantamiseksi nimetään osallistujia eri rooleihin. Näillä rooleilla on tarkoitus olla tarkastuksessa erilainen fokus, jolloin pystytään tarkastelemaan kohdetta laaja-alaisemmin ja pienennetään mahdollisuutta löytää samoja virheitä eri tarkastajien toimesta. Nämä roolit ovat tyypillisesti keskittyminen korkeamman tason suunnitteluun, keskittyminen standardeihin, keskittyminen samaan tasoon liittyviin dokumentteihin ja keskittyminen testattavuuteen tai ylläpidettävyyteen.
Aloittamisvaiheen kokouksessa (kick-off) on tarkoitus päästä kaikkien osallistujien kanssa samalle aaltopituudelle ja päästä yhteisymmärrykseen ajankäytössä. Tarkastukseen osallistujat saavat lyhyen esittelyn tarkastuksen tavoitteista ja dokumenteista. Tässä

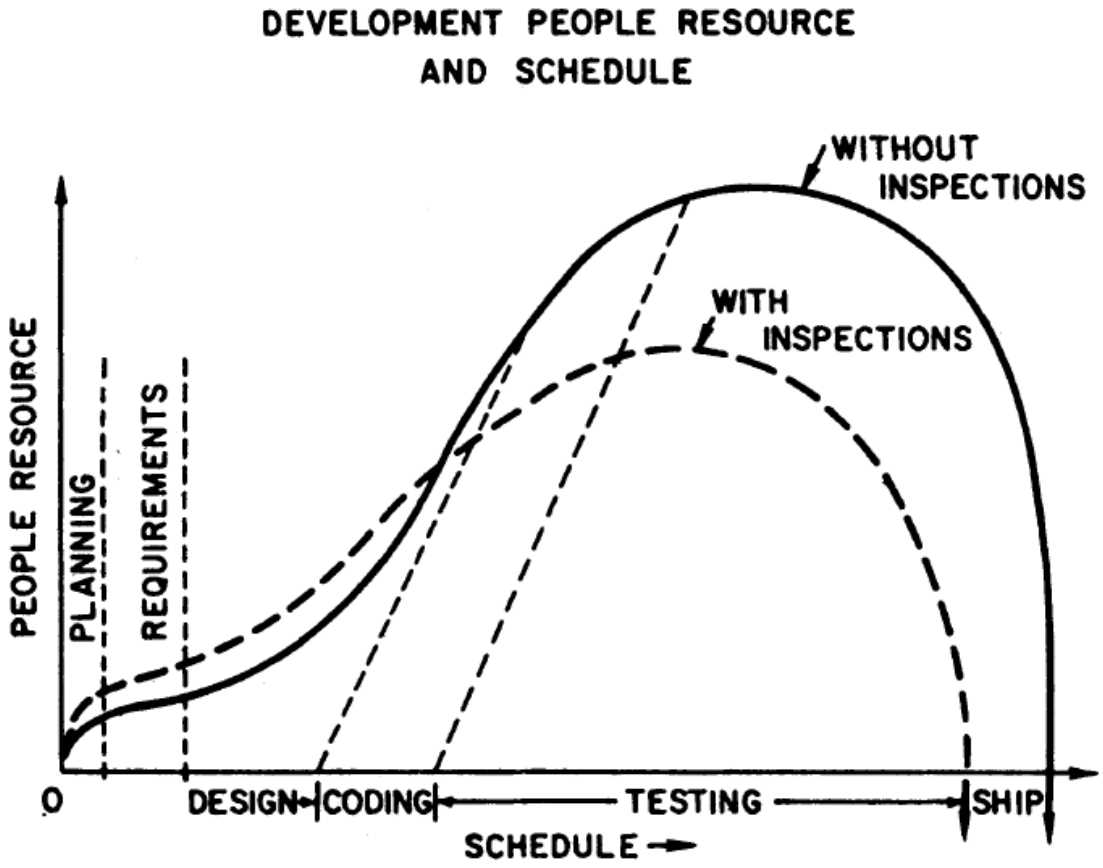
kokouksessa on myös tarkoitus käsitellä aloitustarkastuksen ja (entry check) ja poistumiskriteereiden (exit criteria) asioita. Valmistelussa (preparation) tarkastukseen osallistujat työskentelevät itsenäisesti yrittäen löytää kohteesta virheitä ja muita poikkeamia käyttäen apuna siihen liittyvää dokumentaatiota, menettelyjä, sääntöjä tai tarkastuslistoja. Kaikki ongelmat on tarkoitus kirjata ylös. Tyypillisesti tarkastajien tarkastusnopeus on viidestä kymmeneen sivua tunnissa, mutta se voi olla paljonkin vähemmän epämuodollisessa prosessissa.

Tarkastelukokous (review meeting) tyypillisesti koostuu kirjaamisvaiheesta, keskusteluvaiheesta ja päätöksentekovaiheesta. Kirjaamisvaiheessa on tarkoitus yksinkertaisesti kirjata esille nousseet ongelmat, eikä jäädä tarkemmin pohtimaan ongelmien yksityiskohtaisuuksia. Kuitenkin ongelman mukana kirjataan sen arvioitu vakavuusluokka, joita on kolme: kriittinen (critical), merkittävä (major) ja vähäinen (minor). Keskusteluvaiheessa otetaan esille ne löydetyt ongelmat, jotka on arvioitu sen arvoiseksi. Kokouksen lopussa on tehtävä päätös tarkistettavasta dokumentista, joka joskus perustuu poistumiskriteeriin (exit criteria). Tärkein poistumiskriteeri on kriittisten ja suurimpien virheiden keskimääräinen määrä sivua kohden.

Uudelleen työstäminen (rework) perustuu löydettyjen virheiden korjaamiseen virhe kerrallaan. Jokainen korjattu kohta merkataan selkeästi, jotta on helppo tietää jälkeenpäin mitä kohtia on korjattu. Kaikki virheet eivät kuitenkaan johda korjaamiseen ja on tekijän itse päätettävissä, tuleeko virhe korjata vai ei. Jos jollekin virheelle ei ole mitään tehty, asia on syytä merkata ylös, jotta tiedetään, että asia on käsitelty.

Seurannassa (follow-up) moderaattori vastaa siitä, että kaikkiin kirjattuihin virheisiin on tartuttu ja kaikki tarvittavat toimenpiteet on tehty tyydyttävällä tavalla. Tällä tavalla voidaan prosessin tuomat muutokset osoittaa oikeiksi.

Kokemukset osoittavat, että tarkastukset vaikuttavat henkilöstöresurssien tarpeeseen suunnittelussa ja määrittelyssä, mutta samalla merkittävästi vähentävät tarpeita testausvaiheessa (Kuva 7.). Tuloksena on kokonaisresurssien tarpeen väheneminen kehitystyössä ja usein myös käytetyn ajan pienentyminen kokonaisuudessaan kehitystyössä.



Kuva 7. Resurssien käyttö tarkastuksilla ja ilman tarkastuksia (Fagan 1986 s.745).

Kokemuksien kautta saadut tulokset viimeisten vuosikymmenien aikana osoittavat tarkastuksen avulla saatuja hyviä tuloksia (taulukko 2). On kuitenkin huomatta, että taulukon 2 saatujen tulosten vertailu keskenään ei ole mahdollista. Näihin tuloksiin vaikuttavat ympäristö, joissa ne on tehty ja eri materiaalit.

Taulukko 2. Eri lähteistä saatuja tuloksia tarkastuksien tehokkuudesta. (Laitenberger 2001 s.25).

Reference	Environment	Result
Fagan [40][41]	Aetna Life Casuality	38 defects from 46 detected
	IBM Respond, United Kingdom	93% of all defects were detected by inspections
	Standard Bank of South Africa	Over 50% of all defects detected by inspection
Weller [132]	Bull HN Information Systems	70% of all defect detected by inspection
Grady and van Slack [51]	Hewlett-Packard	60%-70% of all defects detected by inspection
Shirey [122]		60%-70% of all defects detected by inspection
Barnard and Price [4]	AT&T Bell Laboratories	30%-75% of all defects detected by inspection
McGibbon [92]	Cardiac Pacemakers Inc.	70% to 90% of all defects detected by inspection
Collofello and Woodfield [26]	Large real time software project	Defect detection effectiveness is 54% for design inspection, 64% for code inspection, and 38% for testing
Kitchenham et al. [71]	ICL	57.7% of all defects found by code inspection
Franz and Shih [43]	Hewlett Packard	19% of all defects found by inspection
A. Gately [46]	Raytheon Systems Company	The average number of defects found by inspection is 18.2.
Conradi et al. [27]	Ericsson	The average number of defects found by inspection is 3.41.

Vaikka tarkastuksilla saatujen tulosten varjossa voidaan saada hyviä tuloksia ja itse tarkastusprosessi saattaa vaikuttaa yksinkertaiselta ei sen toteuttaminen käytännössä oikealla tavalla ole yksinkertaista. Väärin harjoitetut prosessit voivat johtaa merkittävään tuottavuuden ja motivaation vähenemiseen (Shirey 1992). Tallaisten ongelmien takia Faragan painottaakin muodollista prosessia ja peräänkuuluttaa sen tärkeydestä hyvien tuloksien saamiseksi. (Fagan 1986 s.749).

Muodollisella tavalla suoritetut tarkastukset ovat organisaatioille iso ponnistus ja sen omaksuminen voi viedä vuosiakin aikaa. Siitä saatavat hyödyt eivät kuitenkaan rajoitu kehitysvaiheessa olevien ohjelmien tai ohjelmien lähdekielen laadun parantamiseen, vaan

se myös lisää kehittäjien välistä kommunikaatioita, mahdollistaa hyvien käytäntöjen jakamista ja tiedonsiirtoa kokeneilta aloittelijoille. (Harjumaa 2005 s.47)

4.3.1 Katselmointi staattisten työkalujen avulla

Koska ohjelmistojen perinteiset katselmoinnit ovat osoittautuneet hyväksi tavaksi olemalla kustannustehokas tapa varmistaa ohjelmistojen laatu, näiden työkalujen tarkoitus olisi tarjota ohjelmistokehittäjille avun pystyä suorittamaan katselmointeja itsenäisemmin ja mahdollisuuden löytää poikkeavuuksia, laatuongelmia ja virheitä laajemmista kokonaisuuksista, mikä manuaalisesti on todella vaikeaa tai ajallisesti ei kannattavaa.

Staattisten työkalujen avulla voidaan tunnistaa potentiaaliset ongelmat ennen lähdekielisen ohjelman ajamista ja se myös auttaa tarkastamaan onko ohjelman lähdekieli kirjoitettu standardien mukaisesti. Tämä kaikki perustuu työkalun olemassa oleviin ja mahdollisesti käyttäjän lisäämiin sääntöihin.
Säännöistä johtuen saattaa tulla eteen ongelmia analyysia ensi kertaa ajaessa, koska lähdekielisen ohjelman standardit ja käytännöt vuosien takaa olevassa ohjelman lähdekielessä ei välttämättä vastaa nykyistä standardia ja käytäntöjä. Jos vanha ohjelman lähdekieli on vuosia toiminut ongelmitta, ei välttämättä ole hyvä idea alkaa muuttamaan sitä vain sen takia, että se tyydyttää nykyistä ohjelman lähdekielen standardia tai käytäntöjä. (Graham, et al., 2008: s.189). Hyvässä lähdekielisen ohjelman analysointityökalussa sen kielisäännöt perustuvat tunnettuihin ja toimiviksi todettuihin standardeihin.

4.4 Ohjelmistometriikat

Tässä kappaleessa tutustutaan erilaisiin ohjelmistometriikoihin ja niiden käyttötarkoituksiin. Tarkoituksena on pohtia metriikoiden hyödynnettävyyttä mainframe-ympäristön päivittäisen sovelluskehityksessä ja pohtia sopivien metriikoiden käytettävyyttä apuna mainframen käyttöresurssien pienentämiseen.

When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of science.

- Lord Kelvin

Ohjelmistojen mittaus erilaisilla ohjelmistometriikoilla on olennainen osa hyvää ohjelmistoa. Metriikoiden avulla ohjelmistokehittäjä saa jonkinlaisen käsityksen tuotoksensa ominaisuuksista, tehdyistä ratkaisuista ja voidaanko toteutus julkaista (Fenton & Bieman 2014).

Ratkaisut ohjelmiston laadun tasolla ja teknisillä toteutuksilla ohjelman lähdekielessä on suuri vaikutus ohjelmiston ylläpidettävyyteen ja luotettavuuteen. Esimerkkinä voidaan todeta, että tapoja voi olla monia tehdä tekninen toteutus lähdekielisessä ohjelmassa ja päästä samaan lopputulokseen siinä mielessä, että ohjelman tuloste on sama kuin muissa erilaisissa toteutuksissa, joissa tuloste on sama. Tästä saattaa herätä kysymyksiä kuten olisiko saman toteutuksen voinut tehdä pienemmällä ohjelman lähdekielen rivimäärällä? Tai voisiko tämän toteutuksen tehdä pienemmällä resurssien käytöllä ohjelman lähdekielessä? Erilaiset metriikat voivat auttaa antamalla vastauksia näihin ja mahdollistaa ohjelman lähdekielen optimoinnin.

4.4.1 Ohjelmointirivien määrä

Ohjelman lähdekielen rivimäärä (LOC) tai (SLOC) on yleinen mitta, joka kertoo ohjelman lähdekielen määrän riveissä. Ohjelman lähdekielen rivimäärän laskemisessa on otettava huomioon siihen vaikuttavat tekijät kuten tyhjät rivit, kommenttirivit, tietojen selitykset (data declaration) ja muut rivit, jotka sisältävät erillisiä ohjeita.

Ohjelman lähdekielen rivilukumäärään vaikuttaa paljon tekniikka, kuinka ne lasketaan, eli onko otettu tällaisia tekijöitä huomioon. Ohjelman lähdekielen rivienmäärä, joka ei sisällä kommentointeja ja tyhjiä rivejä (NCLOC) kutsutaan myös tehokkaaksi ohjelman lähdekieleksi (effective lines of code). Tehokkaiden ohjelmien lähdekielien rivilukumäärän laskeminen on hyödyllistä, kun halutaan korkeammalla tasolla vertailla alijärjestelmiä, komponentteja ja ohjelmointikieliä. Koska tehokkaiden ohjelmien lähdekielien laskemisessa ei oteta huomioon, kommentti ja tyhjiä rivejä. Fenton & Bieman ehdottaa siihen sopivaa kompromissia, jossa tehokkaat ohjelman lähdekielirivit ja kommentoidut rivit (CLOC) lasketaan erikseen, jolloin voidaan määrittää (Fenton & Bieman 2014 s.341)

ohjelman lähdekielen kokonaisrivimäärä (LOC) = NCLOC + CLOC.

Ohjelman lähdekielen kokonaisrivimäärän määrittämisen jälkeen voidaan mitata ohjelmasta kommenttien tiheys (%), joka lasketaan seuraavasti

$$\left(\frac{\text{CLOC}}{\text{LOC}}\right) * 100$$

Ohjelman kokoa mitatessaan olisi hyvä mitata ohjelman tehokkaat lähdekieliset rivit. Tämän lisäksi olisi hyvä huomioida ohjelman sisällä oleva kuollut ohjelman lähdekieli (dead code). Prosessi voidaan suorittaa joko eliminoimalla kuollut ohjelman lähdekieli ennen mittausta tai ottaa se huomioon laskennassa. Yleisesti ohjelman koon mittauksessa olisi hyvä vielä ottaa mittaus kommentoiduista ja tyhjistä riveistä, jolloin saadaan kommenttien tiheys myös.
(Fenton & Bieman 2014 s.342) Mukaan käyttökohteita, jossa ohjelman lähdekielisen rivimäärän mittaamista voisi käyttää hyödykseen:

- Saada selville suurin, pienin ja keskimääräinen ohjelman koko
- Ajan myötä kehityssuunta ohjelman koosta
- Tuottavuus

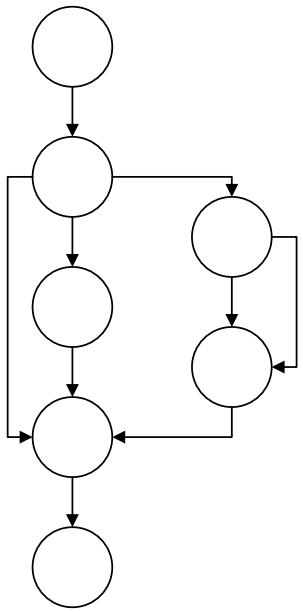
Rivien määrän laskenta mitä tahansa metodia käyttäen on helppoa, mutta tulosten arviointi on vaikeaa jollei mahdotonta tulkita. Ohjelman lähdekielen rivimäärän käyttäminen ohjelman kompleksisuuden arvioinnissa ei tulisi käyttää, koska se ei kerro ohjelman lähdekielessä olevien todellisten ratkaisujen vaativuutta, polkuja tai asioita, joita lähdekielisessä ohjelmassa suoritetaan. Kehittäjien tuottavuuden arviointi ohjelman lähdekielen lukumäärän avulla on myös hieman kyseenalaista, koska rivien lukumäärän avulla ei voida tehdä yksiselitteistä johtopäätöstä työn kompleksisuudesta.

4.4.2 McCaben syklomaattinen kompleksisuus

Thomas McCaben ehdottama kompleksisuuden mittaustapa, joka mittaa lineaarisesti riippumattomien polkujen määrän lähdekielisestä ohjelmasta (Fenton & Bieman 2014 s.391). Syklomaattinen kompleksisuus tunnetaan myös nimellä v(G), missä v viittaa syklomaattiseen numeroon graafiteoriassa ja G osoittaa, että kompleksisuus on funktio graafissa. (Watson & McCabe 1996 s.10) Syklomaattinen kompleksisuus lasketaan ohjelman ohjausvirtakaavion (control flow graph) avulla seuraavanlaisella kaavalla

$$v(G) = E - N + 2P$$

missä E = graafin reunojen lukumäärä, N = graafin solmujen lukumäärä ja P = liitettyjen osien lukumäärä (jos on).



Kuva 8. Esimerkki McCaben syklomaattisen kompleksisuuden laskemisesta. Reunojen lukumäärä on 9 ja solmujen 7, jolloin syklomaattisen kompleksisuuden kaavalla saadaan 9 – 7 + 2 = 4.

Tyypillisesti kompleksisuuden luku lasketaan käyttäen apuna eri työkaluja kuten Eclipse. Riippuen käytettävästä ohjelmasta tulokset ovat hieman erilaisia. Syy tähän on esimerkiksi se, että ehtolause yhdessä tilassa lasketaan erillisiksi haaroiksi ohjausvirtakaaviossa yhdellä työkalulla, kun taas toinen työkalu laskee koko ehtolauseen yhdeksi haaraksi. Tästä syystä huomioon on otettava vertailua tehdessään se, että kompleksisuuden laskenta on tehty samalla työkalulla (Hummel 2014). McGabe on todennut, jotta ohjelma säilyttäisi hyvän testattavuuden ja ylläpidettävyyden ei yksikään ohjelman moduulin tulisi ylittää v(G) = 10 syklomaattista kompleksisuutta.

Vaikka syklomaattisella kompleksisuuden mittaamisen avulla voidaan McCaben mukaan lisätä ohjelman lähdekielen ylläpidettävyyttä, lisätä selkeyttä ja parantaa testattavuutta, on hyvä huomata, että esimerkiksi COBOLissa jokainen WHEN-lause nostaa reunojen

lukumäärää yhdellä, mikä voi nostaa huomattavasti kompleksisuuden lukua, vaikka todellisuudessa tällaisen testattavuus ja luettavuus on käytännössä yksinkertaista. Tästä esimerkkinä seuraava lähdekielinen ohjelmanpätkä.

```
EVALUATE VIIKONPAIVA
            WHEN "1"    MOVE "Maanantai"   TO VIIKONPAIVA-NYT
            WHEN "2"    MOVE "Tiistai"     TO VIIKONPAIVA-NYT
            WHEN "3"    MOVE "Keskiviikko" TO VIIKONPAIVA-NYT
            WHEN "4"    MOVE "Torstai"     TO VIIKONPAIVA-NYT
            WHEN "5"    MOVE "Perjantai"   TO VIIKONPAIVA-NYT
          WHEN OTHER    MOVE "VIRHE"       TO VIIKONPAIVA-NYT.
```

Syklomaattinen kompleksisuus on saanut kritiikkiä siitä, että se ei laske ELSE-haaroja (kuva 9.), jota on pidetty vakavana puutteena metriikassa, jonka tarkoitus on arvioida ohjelman kompleksisuutta.

Se voisi myös antaa vääränlaisia tuloksia mitattaessa moduuleita, koska moduulia mitattaessa se ei kerro kuinka paljon tai vähän kyseinen moduuli voi olla vuorovaikutuksessa muiden moduulien kanssa. Tästä esimerkkinä tulos voi olla kompleksisuudeltaan vähäinen, vaikka mitattava moduuli olisi paljon vuorovaikutuksessa muiden kanssa ja todellisuudessa kompleksisuus olisi moninkertainen. Tämän vuoksi olisi tärkeää myös mitata moduulien kytkentä (coupling) ja yhteenkuuluvuus (cohesion), jotta lähdekielisestä ohjelman kompleksisuudesta saataisiin todenmukainen kuva (Bloom 2015).

```
IF X < 1 THEN
    ...;                                   v(G) = 2

IF X < 1 THEN
    ...
ELSE
    ...;                                   v(G) = 2
```

Kuva 9. Syklomaattinen kompleksisuus, jossa molemmat lauseet arvioidaan saman arvoiseksi (Shepperd 1988).

Pohdittaessa mainframe-ympäristöä, jossa kehittäminen tapahtuu COBOLilla McCaben syklomaattista kompleksisuutta voidaan käyttää hyödyksi kirjoittaessa uusia funktioita, metodeja tai parantaa aiemmin kehitettyjä ratkaisuja kuten keskittää ohjelman lähdekielen tarkastamisen kompleksisiin ohjelman osiin. Tämän lisäksi voidaan kompleksisuuden luvulla arvioida tarvittava testitapausten määrää. Testitapausten määrän arviointi voisi olla ennen kaikkea hyvä tapa kompleksisuus luvulla, kun luodaan uusia ratkaisuja, eikä olla varmoja testauksen määrästä. Kokonaisten ohjelmien mittaaminen ei ole millään määrin kannattavaa, koska tyypillisesti isoissa COBOL-ohjelmissa syklomaattinen kompleksisuuden luku on helposti noin 1000. Tämän vuoksi olisi kannattavampaa pilkkoa mitattavat toiminnallisuudet ja keskittyä mittaamisessa ohjelmien tiettyihin osiin.
Syklomaattisen kompleksisuuden luku voi auttaa kehittäjää ymmärtämään tekemiensä ratkaisujen kompleksisuuden ja se voi auttaa kehittämään metodeita yksinkertaisemmiksi ja paremmin luettaviksi. On siis huomioitava, että yksinomaan syklomaattisen kompleksisuuden luvulla ei voida suoraan tehdä johtopäätöstä jonkin laajemman mitattavan osan ylläpidettävyydestä tai sen laadukkuudesta, vaan se rajoittuu hyvin pitkälti metodien tasolle.

4.4.3 Fan-in and fan-out metriikka

Mahdollisesti yksi yleisin rakenne metriikka on fan-in ja fan-out, joka perustuu Yordonin ja Constantinin vuonna 1979 ja Myersin vuonna 1978 ehdottamiin kytkentäideoihin (Kan 2002). Tätä metriikkaa on tarkoitus voida käyttää tunnistamaan potentiaaliset kompleksiset ja kriittiset osat järjestelmissä.

- Fan-in: moduulien määrä, jotka kutsuvat tiettyä moduulia.
- Fan-out: moduulien määrä, joita kutsutaan tietystä moduulista.

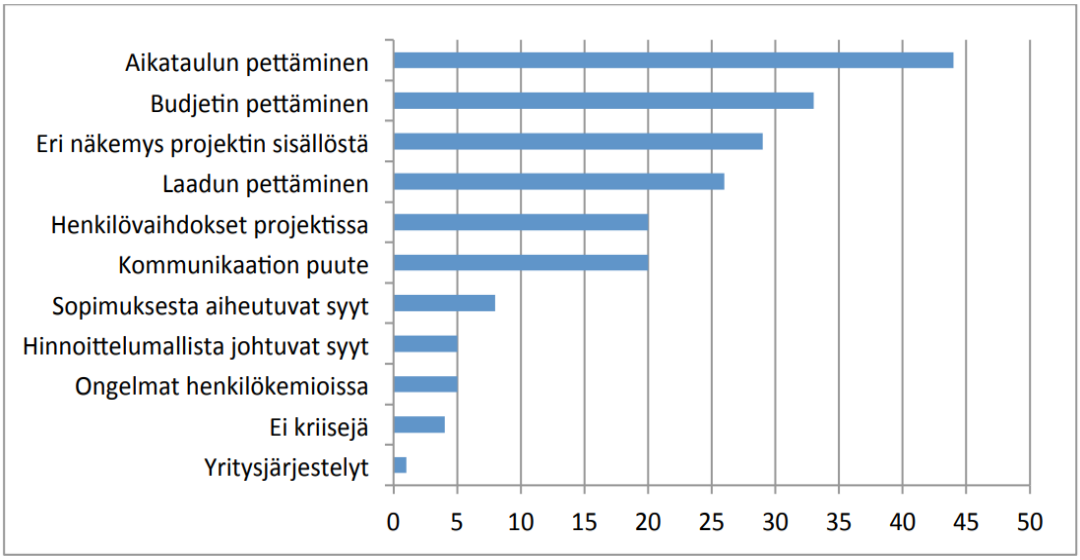
Yleisesti, moduulit, joilla on suuri fan-in ovat suhteellisen pieniä ja yksinkertaisia, ja tyypillisesti ne sijoittuvat alempiin kerroksiin suunnittelurakenteessa (design structure). Sitä

vastoin, moduulit, jotka ovat isoja ja kompleksisia, todennäköisesti omaavat pienen määrän moduuleja, jotka kutsuvat tiettyä moduulia (fan-in). Tämän vuoksi moduulit tai komponentit, joissa on suuri fan-in ja fan-out, voivat antaa viitteitä siitä, että ne ovat huonosti suunniteltuja (Kan 2002). Tämä saattaa johtua siitä, että vuosien aikana kasvanutta kompleksista ja ongelmallista moduulia tai komponenttia ole pilkottu helpommin ymmärrettäviin ja hallittaviin osiin.

5. Ohjelmiston laatu ja sen arviointi

Ohjelmistokehittäjän näkökulmasta voidaan ajatella esimerkiksi, että hyvin kirjoitettu ohjelman lähdekieli on laadukasta, mutta ohjelmointikielen kääntäjä tai tietokone ei tätä osaa erottaa. Hyvin kirjoitettu ohjelman lähdekieli on toki ainakin selkeämmin luettavaa ja ylläpidettävää. Tässä kappaleessa käydään läpi laatua yleisten ISO-standardien kautta, koska laatu käsitteenä on subjektiivinen ja voidaan ymmärtää hyvinkin eri tavoilla, riippuen näkökulmasta.

Valtakunnallinen Tietojärjestelmien hankinta Suomessa -tutkimuksen mukaan syyt kriisiytymisissä tietojärjestelmähankkeissa tilaajannäkökulmasta yksi suurimmista tekijöistä on laadun pettäminen (Kuva 10). Huomattavan pienempänä laadun pettäminen kuitenkin koetaan tietojärjestelmähankkaiden kriisiytymiseen johtuvana syynä toimittajien näkökulmasta, jossa esiin nousivat kommunikaation puute, eri näkemykset projektin sisällöstä ja aikataulun pettäminen.



Kuva 10. Tietojärjestelmähankkeiden kriisiytymisen syyt tilaajanäkökulmasta (tivia.fi 2013)

Ohjelmiston laadun määrittäminen yksiselitteisesti on hankalaa ja näkemyksiä on objektiivinen tai subjektiivinen näkökulma. Lähestymistapoja ohjelmiston laadun arvioimiseen on sisäiset ominaisuudet, ulkoiset ominaisuudet, käyttäjän kokemukset ja valmistusprosessi. (Paakki, et al., 2015) Jos ajatellaan esimerkiksi, eri sidosryhmiä kuten loppukäyttäjät, ohjelmiston maksaja ja ohjelmistokehittäjät. Kaikilla edellä mainituilla sidosryhmillä on erilaisia näkökulmia ohjelmiston laadun mittaamiseen ja jotkin kriteerit ovat täysin subjektiivisia. Esimerkiksi, käyttäjän näkökulmasta helppokäyttöisyyden mittaaminen on subjektiivista, jos käyttäjien tietotaito taso on erilaisia.

Taulukko 3. Taulukossa listattuina yleisimmät ja jotkin vähemmän tunnetut ohjelmiston laatuvaatimukset. Näkökulmina laatuun ulkoiset ja sisäiset ominaisuudet. (Coudert 2011)

	User	Developer	Measurable
External quality			
features	x		yes
speed	x	x	yes
space	x	x	yes
network usage	x	x	yes
stability	x	x	yes
robustness	x	x	somewhat
ease-of-use	x		subjective
determinism	x	x	yes
back-compatibility	x		yes
security	x		difficult
power consumption	x		difficult
Internal quality			
test coverage		x	yes
testability		x	hard
portability		x	somewhat
thread-safeness		x	hard
conciseness		x	somewhat
maintainability		x	hard
documentation		x	subjective
legibility		x	subjective
scalability		x	somewhat

Taulukosta 3 poiketen yleisesti kuitenkin voidaan ajatella, että kaikki sisäiset ja ulkoiset ohjelman laatuvaatimukset koskettavat kehittäjiä jollain tapaa. Loppukäyttäjää koskettavat vain ulkoiset laatuvaatimukset, koska yleensä heillä ei ole pääsyä ohjelman lähdekieleen, eikä heillä ole tarvetta tietää sisäisiä laatuvaatimuksia.

5.1 ISO/IEC 25000 standardit

Tässä kappaleessa otetaan yleiskatsaus ISO/IEC 25000 sarjan standardeista ja tarkastellaan tarkemmin ISO/IEC 25010 standardin laatupiirteitä. Laatumallit ovat käytönaikainen laatumalli ja ohjelmiston / järjestelmän laatumalli, jonka pääasiallinen käyttötarkoitus on ohjelmiston tai järjestelmän kehittämisen aikana. On huomattava, että ISO/IEC -standardit ovat suosituksia ja niitä pitää tulkita tapauskohtaisesti, eikä yhtä ainoaa oikeaa laatumallia ole.

Standardisoinnin ansioista tuotteet, palvelut ja menetelmät sopivat siihen käyttöön ja niihin olosuhteisiin, joihin ne ovat tarkoitettu. Niiden tarkoitus on varmistaa, että tuotteet ja järjestelmät sopivat toisiinsa ja toimivat yhdessä. Kuitenkaan kaikki tietotekniikan standardisointi ei tapahdu virallisten standardisointiorganisaatioiden puitteissa. Muut standardisointiorganisaatiot tekevät standardisointityötä omista lähtökohdistaan, mutta toimivat välillä pohjana virallisille standardeille. (sfsedu.fi 2018)

SFSedu:n mukaan Suomessa standardien käyttö on aiheuttanut laajaa mielenkiintoa ohjelmistoyrityksissä, kun on pitänyt osoittaa ohjelmiston laatu asiakkaalle tai viranomaiselle. Myös tapauksissa, missä tuotelaatu on nähty keskeisenä kilpailutekijänä ja on haluttu mitata sitä suoraan, ei vain asiakastyytyväisyytenä tai kehittämisprosessin kyvykkyytenä.

Ohjelmistojen laadun mittaamisen standardin kehitys alkoi vuonna 1985. Se pohjautui laatuun vaikuttavista tekijöistä. Ensimmäinen standardi julkistettiin vuonna 1991: ISO/IEC 9126: Information technology-Software product evaluation-Quality and the guidelines for their use. Erityisesti ISO/IEC 9126 -standardin johdosta kehittyi vuosien jälkeen nykyiset ISO/IEC 25000 sarjat. (sfsedu.fi 2018)

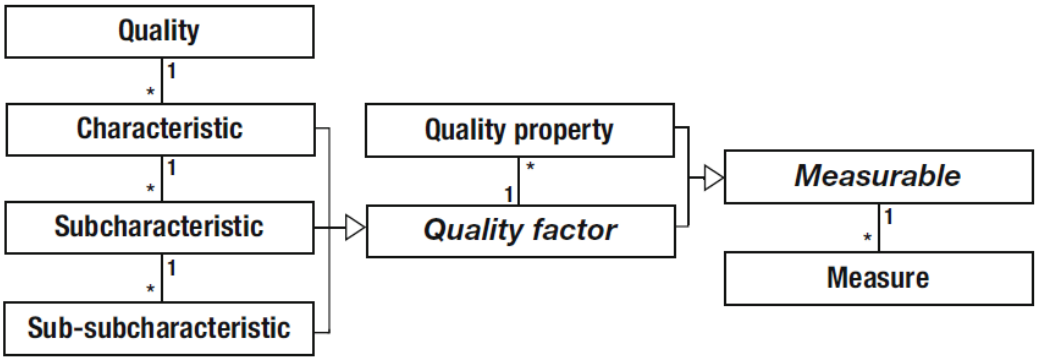
ISO/IEC 25000 -standardeihin kuuluvat sarjat tunnetaan myös nimellä SquaRE (System and Software Quality Requirements and Evaluation). Näiden standardien tavoitteena on

luoda puitteet ohjelmistotuotteiden laadun arvioimiselle. SQuaRE sisältää laatumallin ja joukon laadun mittareita ohjelmistoille ja järjestelmille. (ISO 25000 & sfsedu.fi 2018)

Kuva 11. ISO/IEC 25000 sarjan standardit (ISO 25000 2018) ovat jaettuna viiteen eri osaan: Laatumallien osio 2501n, laadun mittaamisen ja mittojen osio 2502n, ohjelmiston ja järjestelmän laadun hallinnan osio 2500n, laatuvaatimusten osio 2503n ja laadun arvioinnin osio 2504n.

5.1.1 Kehittämisen aikainen laatumalli

Näiden mallien tarkoitus on tarjota luokittelu, joka pilkkoo ohjelmistotuotteiden monimutkaisen käsitteet pienemmiksi paremmin hallittaviksi osiksi.
Ajatuksena on, että pilkkomisen jälkeen se saavuttaa tason, jolla voimme mitata
osia ja käyttöä, jotka arvioivat ohjelmistotuotteiden laatua. (Wagner 2013. s.60) Metamallin mukaisesti eri laatutekijät jakautuvat osiin, näin niitä on mahdollista mitata.



Kuva 12. Metamalli ISO/IEC 25010 (Wagner 2013. s.61)

Kehittämisen aikaisessa laatumallissa ISO/IEC 25010:ssa on määritelty kahdeksan erilaista osa-aluetta, jotka jakautuvat pienempiin tarkemmin määriteltyihin laatuominaisuuksiin. Osa-alueet ovat toiminnallinen sopivuus (functional suitability), tehokkuus (performance effiency), yhteensopivuus (compatibility), käytettävyys (usability), luotettavuus (reliability), turvallisuus (security), ylläpidettävyys (maintainability) ja siirrettävyys (portability).

1. Toiminnallinen sopivuus (functional suitability). Tämä osa-alue kuvaa tuotteen tai järjestelmän tarjoamia toimintoja, jotka täyttävät ilmoitetut ja epäsuorat tarpeet tietyissä olosuhteissa. Tähän kuuluvat laatuominaisuudet ovat toiminnallinen kattavuus (functional completeness), toiminnallinen oikeellisuus (functional correctness) ja toiminnallinen soveltuvuus (functional appropriateness).

2. Luotettavuus (reliability). Tuotteen tai komponentit suorittavat niille määritellyt toiminnot tietyissä olosuhteissa ja tietyssä annetussa ajassa. Siihen kuuluvat ominaisuudet ovat tuotteen kypsyys (maturity), saatavuus (availability), vikasietoisuus (fault tolerance) ja toipumisvalmius (recoverability)

3. Tehokkuus (performance effiency). Tässä alueella on kyse suorituskyvystä suhteessa käytettyjen resurssien määrään. Tähän kuuluvat vasteaika (time behaviour), resurssien käyttösuhde (resource utilization) ja kapasiteetti (capacity)
4. Käytettävyys (usability). Tämä kohta pitää sisällään käytettävyyteen erilaisia näkökulmia. Nämä ominaisuudet ovat soveltuvuuden selkeys (appropriateness recognizability), opittavuus (learnability), helppokäyttöisyys (operability), käyttövirheiden estäminen (user error protection), käyttöliittymän miellyttävyys (user interface aesthetics) ja matala kynnys (accessibility).

5. Ylläpidettävyys (maintainability). Tämä osa-alue kuvaa tehokkuutta ja hyötysuhdetta, jolla tuotetta tai järjestelmää voidaan muuttaa sen parantamiseksi, korjaamiseksi, mukauttaa erilaiseen ympäristöön tai muutoksiin vaatimuksissa. Tämä osa-alue koskeen vain ohjelmistokehittäjiä. Siihen kuuluvat ominaisuudet ovat rakenteellinen selkeys (modularity), uudelleen käytettävyys (reusability), analysoitavuus (analysability), muunneltavuus (modifiability) ja testattavuus (testability).

6. Turvallisuus (security). Missä määrin tuote tai järjestelmä suojaa tietoa tai dataa niin, että henkilöillä tai muilla tuotteilla tai järjestelmillä on pääsy sille määritellyille tasoille. Laatuominaisuuksia ovat luottamuksellisuus (confidentiality), eheys (integrity), kiistämättömyys (non-repudiation), vastuullisuus (accountability) ja aitous (authenticity).

7. Yhteensopivuus (compatibility). Järjestelmä tai komponentti pystyy toimimaan ja liikuttaa informaatioita muiden eri tuotteiden, järjestelmien tai osien kanssa. Se ei myöskään aiheuta haittaa muiden eri tuotteiden, järjestelmien tai niiden osien kanssa. Siihen kuuluvat ominaisuudet ovat rinnakkaiselo (co-existence) ja yhteen toimivuus (interoperability)

8. Siirrettävyys (portability). Kuvaa tehokkuutta ja hyötysuhdetta, jolla järjestelmä, tuote tai tuotteen osa voidaan siirtää laitteistosta, ohjelmistosta tai muusta käyttöympäristöstä toiseen. Siihen kuuluvat ominaisuudet ovat sovitettavuus (adaptability), asennettavuus (installability) ja korvattavuus (replaceability).

Kuva 13. Kehittämisen aikainen laatumalli ISO/IEC 25010 (ISO/IEC 25010 2018)

5.1.2 Käytönaikainen laatumalli

Käytönaikaisessa laatumallissa on määritelty viisi osa-aluetta, jotka jakautuvat pienempiin tarkemmin määriteltyihin laatuominaisuuksiin. Osa-alueet ovat tyytyväisyys (satisfaction), vaikuttavuus (effectiveness), riskivapaus (freedon from risk), effiency (tehokkuus) ja kontekstin kattavuus (context coverage). Tarkemmin määritellyt laatuominaisuudet ovat johdettu yleisistä käyttötilanteista ja ne pitää tulkita tapauskohtaisesti. On hyvä huomata, että yhtä ainoaa oikeaa laatumallia ei ole. (sfsedu.fi 2018).

Tässä käytönaikaisessa laatumallissa tarkastellaan laatua eri sidosryhmien perspektiivistä. Kuitenkin merkittävin näistä sidosryhmistä on ensisijainen käyttäjä. Tämän vuoksi, laatuun käytettäessä liittyy usein pelkästään käytettävyys. (Wagner 2013. s.63)

1. Tyytyväisyys (satisfaction). Tämä osa-alue kertoo sen, kuinka käyttäjä tuntee käyttäessään tuotetta. Siihen kuuluvat tekijät ovat hyödyllisyys (usefulness), luottamus (trust), miellyttävyys (pleasure) ja mukavuus (comfort).

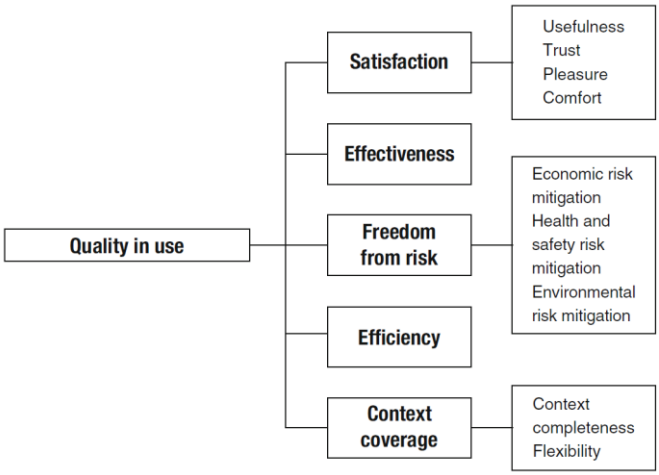
2. Vaikuttavuus (effectiveness). Vaikuttavuus on sitä, kuinka hyvin tuote tai ohjelmisto tukee itse käyttäjää tavoitteiden saavuttamisessa. Vaikuttavuudella ei ole tarkemmin määriteltyjä laatuominaisuuksia kuin se itse.

3. Riskivapaus (Freedom from risk). Yksi tärkeimmistä tässä osa-alueella olevista laatuominaisuuksista on järjestelmien turvallisuus, jotka voivat vahingoittaa ihmistä sekä ympäristö- tai taloudelliset riskit. Kokonaisuutena tähän kuuluvat laatuominaisuudet ovat taloudellisten riskien vähentäminen (economic risk mitigation), terveys- ja turvallisuusriskien vähentäminen (health and safety risk mitigation) ja ympäristöriskien vähentäminen (environmental risk mitigation).

4. Tehokkuus (Effiency). Tehokkuus tarkoittaa näiden tavoitteiden saavuttamiseksi tarvittavien resurssien määrää. Tehokkuudella ei ole tarkemmin määriteltyjä laatuominaisuuksia kuin se itse.

5. Sisällön kattavuus (context coverage). Tämä osa-alue kertoo ohjelmiston tai tuotteen kattavuudesta sekä kyvystä reagoida joustavasti muutoksiin sisällössä. Siihen kuuluvat laatuominaisuudet ovat sisällön täydellisyys (context completeness) ja joustavuus (flexibility)

(Wagner 2013. s.64)



Kuva 14. Käytön aikainen laatumalli ISO/IEC 25010 (Wagner 2013. s.63)

6. Ohjelmistotyökalun valinta

Tässä osassa tarkastellaan asioita, jotka vaikuttivat pilotoitavan työkalun valintaan, kriteereitä, joita sen täytyi täyttää ja rajataan alue, jota ohjelmistotyökalussa tarkastellaan. Seuraavassa osassa laaditaan suunnitelma tutkimukselle ja lopuksi esitellään tutkimuksesta saadut tulokset.

Markkinoita läpikäydessä erilaisia työkaluja COBOL-kielen staattiseen testaukseen löytyi useampia. Koska kyse on kaupallishallinnollisista tai erilaisista liiketoiminnan sovellusten ohjelmointikielen (COBOL) staattisesta testauksesta, mikään näistä ohjelmistotyökaluista ei ole ilmainen ja maksavat merkittäviä summia rahaa.

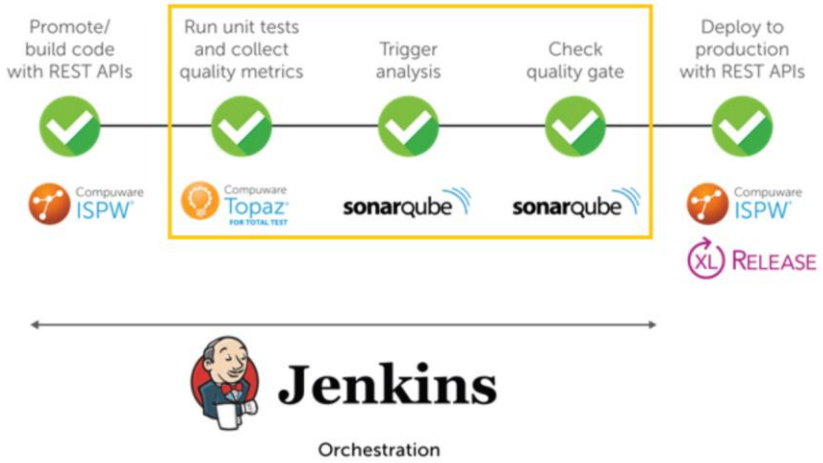
Yhdeksi suureksi haasteeksi tutkimuksen suorittamisen aikana osoittautui kokeiluversion saaminen käyttöön. Yhden työkalun kokeiluversion saamiseen meni aikaa noin kuukauden verran. Ylipäätään kokeiluversion saaminen yksinomaan tutkimuskäyttöön yksityiselle henkilölle ei ollut mahdollista. Kokeiluversion saamiseen vaikuttava asia näytti vahvasti olevan se asia, että tutkimustyö tehdään eräälle organisaatiolle. Ajatuksena oli alun perin päästä kokeilemaan useampaakin tuotetta ja vertailla niiden tarjoamia ominaisuuksia keskenään. Tutkittaessa tarkemmin markkinoilla tarjolla olevia tuotteita kävi nopeasti selväksi, että valinnanvaraa ei juurikaan ollut kriteereiden määrittelemisen jälkeen. Kokeiluversioiden haasteiden vuoksi kokeiluun otettiin vain yksi analysointityökalu, jossa kohteena COBOL-kielen staattinen testaus.

Tärkeitä kriteereitä työkalun valinnassa olivat

- Työkalun taustalla oleva aktiivinen ylläpito ja sen kehitys
- Muiden ohjelmointikielten tuki suotavaa
- Työkalun hyvä ja selkeä käytettävyys
- Integraatio mahdollisuudet käytössä olevaan ohjelmointiympäristöön
- Työkalun tarjoamat ominaisuudet

Oleellisesti tällaisen työkalun valintaan vaikuttavia elementtejä ovat myös sen käytöstä aiheutuvat kustannukset sitä käyttävälle organisaatiolle, työkalun asennus, työkalun ylläpito, työkalun päivitykset ja työkaluun tarvittava koulutus sen mahdollisimman tehokkaaseen käyttöön. Tässä työssä ei kuitenkaan suuremmin oteta kantaa näihin asioihin, vaan keskitytään työkalun ominaisuuksiin ja niistä saataviin hyötyihin.

Edellä mainittujen kriteerien osalta amerikkalaisen yhtiön nimeltään SonarSource ja sen tuotteet SonarQube ja SonarLint vaikuttivat parhaiten sopivimmilta.
SonarSourcen mukaan, heidän tuotteensa on todennäköisesti paras analysointityökalu COBOL-kielen staattiseen analysointiin mitä markkinoilta löytyy. Se tarjoaa kokonaisuudessaan tuen 20 eri ohjelmointikielelle, jotka keskittyvät ohjelmointikielen ylläpidollisiin, luetuttavuudellisiin ja haavoittuvuudellisiin ratkaisuihin. Analysointityökalu käyttää heidän mukaansa kehittyneimpiä tekniikoita (kuvionmääritystä ja tietovirta analyysia) ohjelmointikielen analysoimiseksi. SonarSourcen tuotteet ovat kehitetty seuraavin periaattein: syvyys, tarkkuus ja nopeus (sonarsource.com 2018).
SonarCOBOL perustuu suureen määrään vakiintuneita laatustandardeja ja näitä laatustandardeja ja muita SonarCOBOLin ominaisuuksia voidaan käyttää myös IBM:n IDz:ssa. IDE kuten IDz, voidaan asentaa SonarSourcen lisäosa, jonka nimi on SonarLint. SonarLint mahdollistaa reaaliaikaisen ohjelmointikielen analysoinnin ohjelmistoa kehittäessä samoilla ominaisuuksilla mitä SonarCOBOL tarjoaa.



Kuva 15. Keskuskoneen ohjelmistokehityksessä tapahtuva jatkuva integraatio Jenkins orkestroinnilla (youtube.com 2019).

Compuwaren näkemyksen mukaan (kuva 15) jatkuvassa sovelluskehityksen ideaalisessa tilanteessa joka kerta, kun sovelluskehittäjä rakentaa lähdekielisen ohjelman (build code) laukaistaan automatisoitu yksikkötestaus, kerätään ohjelmistometriikka, ohjelman lähdekielen kattavuus (code coverage) ja viedään se SonarQubelle, jossa voidaan todentaa vastaako uuden ohjelman lähdekielen laatu asetettuja laatuvaatimuksia (youtube.com 2019).

SonarQuben käytön hinta perustuu keskuskonejärjestelmässä olevien ohjelmointirivien määrään mitkä analysoidaan SonarQubella. Tähän hintaan kuuluu myös SonarLint. Saatavilla on tämän lisäksi SonarSourcen oma tuki, johon sisältyy erilaiset asiantuntijapalvelut. Tällä palvelulla on erikseen hintaa 15 000 € + alv / vuosi.

Taulukko 4. SonarQuben hinnoittelu yritysversiolle (sonarsource.com. Enterprise edition. Plans and pricing 2018).

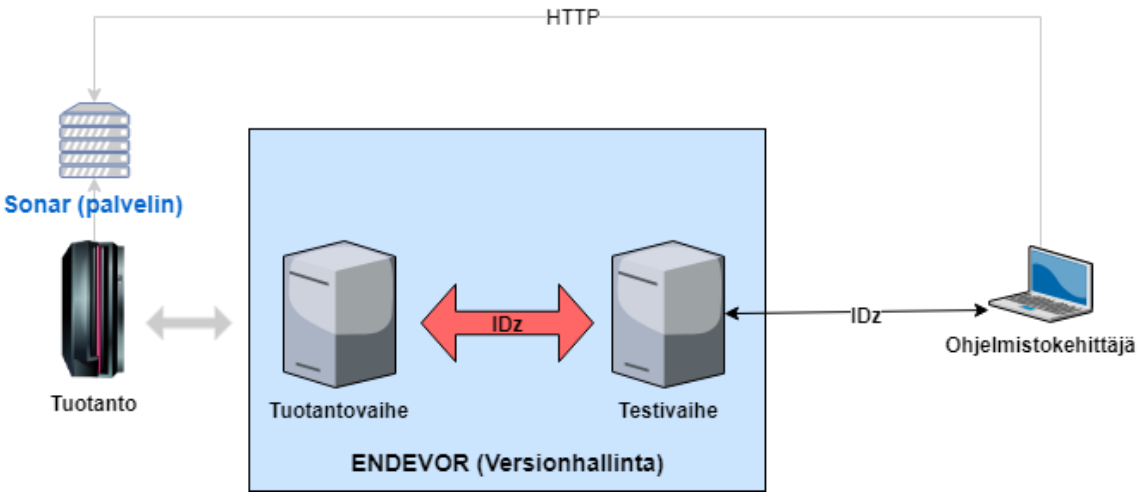
LOC	**Hinta / Vuosi**
1 milj.	15 000 € + alv
5 milj.	25 000 € + alv
10 milj	37 500 € + alv
20 milj.	50 000 € + alv
30 milj.	75 000 € + alv
50 milj.	90 000 € + alv
75 milj.	125 000 € + alv
100 milj.	180 000 € + alv

6.1 Tutkimuksen suunnittelu ja valmistelu

SonarQuben kokeilulisenssin saannin jälkeen voitiin ladata heidän omien sivujen kautta pakattu tiedostopaketti, joka purettiin omalle paikalliselle tietokoneelle. Tämän jälkeen oli määriteltävä ominaisuudet tekstitiedostoon cobol ja copybook (Määrittää COBOL-ohjelman tietorakenteet) -tiedostojen sijainnit, niiden tiedostopäätteet ja oikea merkistöstandardi.
SonarQube palvelimen käynnistys tapahtui tiedostopaketin mukana tulevasta tiedostotyypiltään komentojono -tiedostosta.
Käynnistyksen aikana ohjelma käsitteli kaikki copybookit ja COBOL-tiedostot, jotka olivat aiemmin määriteltynä ominaisuudet tiedostossa. Kaikkien tiedostojen käsittelyyn ja analysointiin aikaa meni noin 20 minuuttia.

Tilanteessa, jossa SonarQubesta on käytössä täydellinen enterprise (yritys) tason lisenssi, sijaitsisi SonarQube omalla palvelimellaan ja sen sisältämät ohjelmaversiot vastaisivat täysin samaa kuin tuotannossa olevat ohjelmat. Tämä mahdollistaa sen, että ohjelmisto kehittäjän tietokoneilla ei tarvitse olla kaikkia järjestelmässä olevia ohjelmia ja joka kerta

SonarQuben avatessaan ei tarvitse synkronoida kaikkia tiedostoja läpi, jotta saadaan tuoreimmat versiot analysoitavaksi.



Kuva 16. Arkkitehtuurikuvaus SonarQubesta keskuskonejärjestelmässä.

Havainnollistava esimerkki kuvan 16 mukaan, niistä työvaiheista, joita SonarQuben hyödyntämisessä käytetään.

1. Ohjelmistokehittäjä ohjelmoi jollain SonarQuben tukemista IDE:stä. Tässä tapauksessa IDE on Eclipse, jossa on myös SonarLint käytössä, mikä tarjoaa reaaliaikaisen ohjelman analysoinnin ohjelmistokehittäjän kehittäessä ohjelmaa.

2. Ohjelmistokehittäjä vie ylläpidetyn ohjelman lähdekielen versiohallintaan.

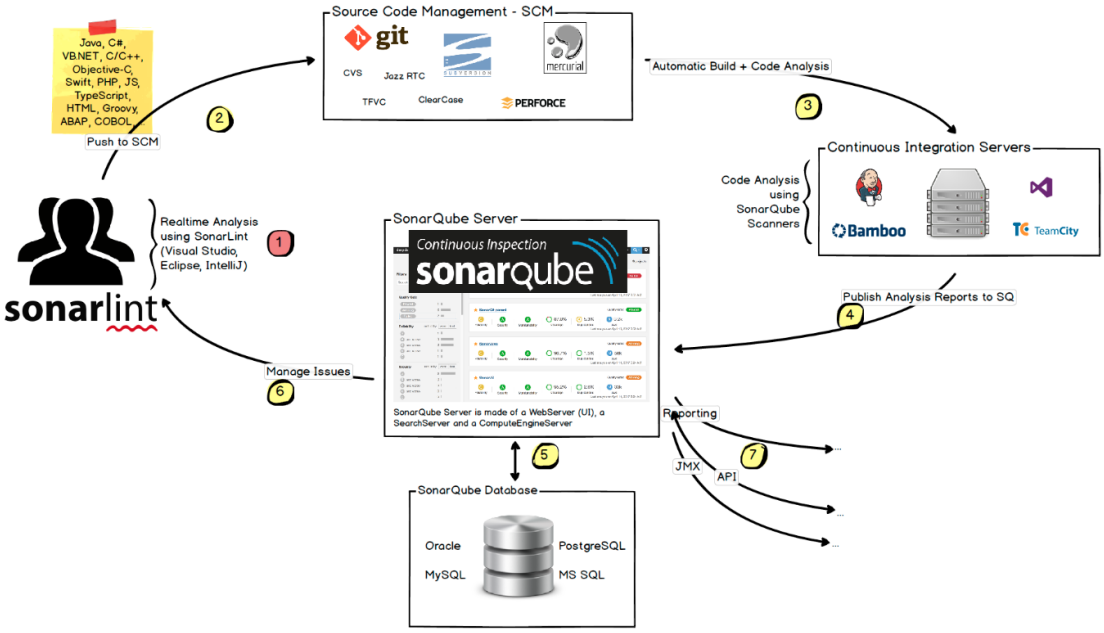
3. Jatkuva integrointipalvelin käynnistää automaattisen luonnin (build) ja käynnistää SonarQube-skannerin, joka on ehto SonarQube-analyysin suorittamiseen.

4. Analyysinraportti lähetetään SonarQube palvelimelle käsittelyä varten

5. SonarQube palvelin käsittelee ja tallentaa analysointikertomuksen tulokset SonarQube-tietokantaan ja näyttää tulokset web-käyttöliittymässä

6. Kehittäjät tarkastelevat ja kommentoivat analysoinnin tuloksia.

7. Ohjaajat (Managers) saavat raportit analyysista.



Kuva 17. Esimerkki SonarQuben integraatio mahdollisuuksista muiden sovelluksen elinkaaren hallintatyökalujen kanssa (sonarqube.org 2018).

6.2 SonarQubessa olevia käsitteitä

Tässä luvussa on avattuna SonarQubessa olevat laadulliset käsitteet ja niiden tarkemmat määritelmät. Ongelmatyypeille on määriteltynä erilaiset vakavuudet, joiden perusteella voidaan tehdä tarvittavat jatko toimenpiteet

Taulukko 5. SonarQubessa olevia laadullisia käsitteitä.

Käsite	**Määritelmä**
Vika (Bug)	Kuvaa kohtaa ohjelman lähdekielessä, jossa on luotettavuuteen liittyvä vika.
Koodihaju (Code Smell)	Kuvaa ohjelman lähdekielessä kohtaa, jossa on ylläpidettävyyteen liittyvä ongelma.
Ohjelmointisääntö (Code Rule)	Kuvaa ohjelman lähdekielessä olevaa kohtaa, jossa ei ole noudatettu hyviä ohjelmointisääntöjä.
Komponentti (Component)	Yksi osa ohjelmistoa eli moduuli, paketti, projekti, tiedosto. Voi myös kuvata näkymää tai kehittäjää.
Vuotoaika (Leak Period)	Aikakehys edellisestä ohjelman lähdekielen julkaisusta uusimpaan versioon, jossa määritetyt kriteerit mitataan juuri lisättyyn ohjelman lähdekieleen.
Mittasuhde (Measure)	Metriikan antama arvo. Esimerkiksi ohjelman lähdekielen määrä 2000 riviä.
Metriikka (Metric)	Mittaustyyppi kuten LOC.
Ei-toiminnallinen vaatimus (Non-functional requirement)	Laatuominaisuuksia, joita on esimerkiksi ohjelmointisäännöissä määriteltynä.
Laatuprofiili (Quality profile)	Joukko ohjelmointisääntöjä. Jokainen tilannekuva perustuu yhteen laatuprofiiliin.
Kunnostamisen kustannus (Remedation Cost)	Haavoittuvuus- ja luotettavuusongelmien korjaamiseen tarvittava arvioitu aika. Yhdelle ohjelmariville on annettu kustannuksen arvo, joka on 0,06 päivää.
Tilannekuva (Snapshot)	Jokaisen analyysin jälkeen luodaan tilannekuva, joka kertoo mittasuhteita ja ongelmia tiettynä ajankohtana.
Tekninen velka (Technical Debt)	Arvioitu aika, joka tarvitaan kaikkien ylläpito-ongelmien, ohjelman lähdekielessä tai koodihajujen ratkaisemiseksi. Perustuu SQALE metodologiaan.
Haavoittuvaisuus (Vulnerability)	Turvallisuuteen liittyvä ongelma. Mahdollistaa hyökkääjille ohjelman lähdekielessä takaportin.
Ylläpidettävyyden luokitus (Maintainability rating)	Luokitukset vaihtelevat hyvästä A:sta (erittäin hyvä) E:hen (erittäin huono). Luokitus perustuu teknisen velan suhde -arvon mukaan, joka vertaa projektin teknistä velkaa kustannuksiin, joita se tarvitsisi kirjoittaa ohjelman lähdekielen uudestaan tyhjästä.

SonarQubessa ongelmien tyypit COBOL-ohjelman lähdekielen osalta ovat jaettuna kolmeen eri osaan, jotka ovat

- Luotettavuuteen liittyvä vika (Bug)
- Haavoittuvuus (Vulnerability)
- Koodihaju (Code Smell). Liittyy ylläpidettävyyteen.

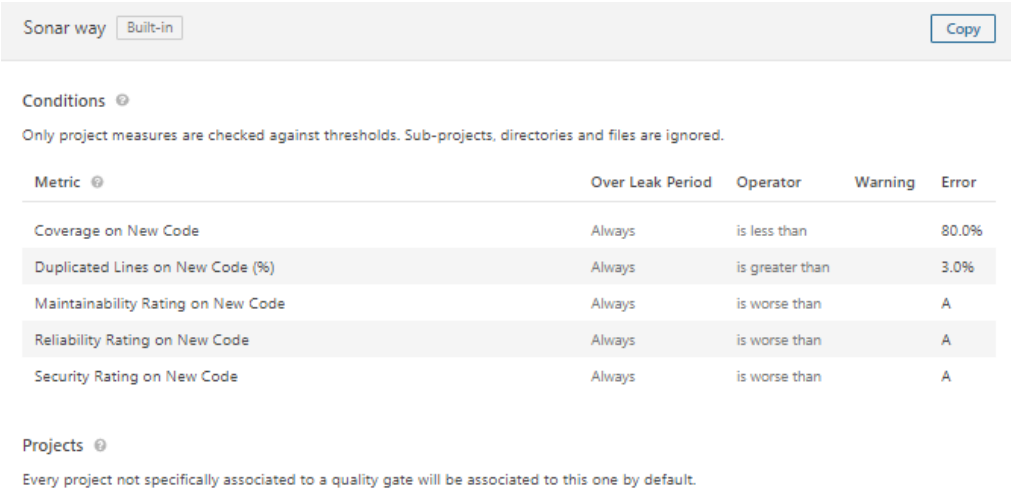
Kaikilla havaituilla ongelmilla on määriteltynä niiden vakavuus, jonka analysoinnista SonarQube vastaa. Näiden ongelmien vaikutus käytössä on arvioitu pienenä tai suurena ja todennäköisyys vaikutusten realisoitumiselle on arvioitu pienenä tai suurena.

Taulukko 6. SonarQubessa ongelmien tyypeille määritellyt vakavuudet (Severity).

Vakavuus	Vaikutus	Todennäköisyys
Esteellinen (Blocker)	Suuri	Suuri
Kriittinen (Critical)	Suuri	Pieni
Suuri (Major)	Pieni	Suuri
Vähäinen (Minor)	Pieni	Pieni

6.3 SonarQuben ominaisuudet

Yhtenä ominaisuutena, jonka SonarQube tuo vahvasti esille on vesivuodon korjaaminen (Fixing the Water Leak), joka lyhyesti tarkoittaa voimavarojen, joka tässä tapauksessa on ohjelmistokehittäjän ajankäytön suuntaaminen uuteen, juuri tuotettuun ohjelman lähdekieleen. Tällä tavalla voidaan lopettaa teknisen velan kerryttäminen. SonarQuben näkemys vesivuodon korjaamiseen on laatuportti (Quality Gate), jonka tehtävänä on valvoa organisaation laatupolitiikka ja joka vastaa kysymykseen voidaanko ohjelma viedä tuotantoon laadullisesta näkökulmasta katsottuna.



Kuva 18. Kuvankaappaus, jossa esiteltynä laatuportti (Quality Gate)

Laatuportissa voidaan määritellä lukuisia erilaisia Sonar-Quben tarjoamia metriikoita. Näille metriikoille annetaan boolean kynnysarvot. Kynnysarvojen realisoituessa annetaan siitä varoitus tai virhe. Esimerkkinä syklomaattiselle kompleksisuudelle määritellään kynnysarvo: suurempi kuin, varoitukselle 10 ja virheelle 20.
Kuvassa 18 on esiteltynä laatuportti, jossa on oletuksena SonarQuben sisäänrakennetut metriikat

- Uuden ohjelman lähdekielen kattavuus (Coverage on New Code). Virheen välttämiseksi laatuportissa tulisi uutta ohjelman lähdekieltä suorittaa vähintään 80%.

- Duplikaatti rivit uudessa ohjelman lähdekielessä (Duplicated Lines on New Code). Virheen välttämiseksi duplikaatti ohjelman lähdekieltä ei saa olla yli 3% uudesta ohjelman lähdekielestä.

- Huollettavuusarviointi (Maintainability Rating). Projektille annettu luokittelu, joka liittyy teknisen velan suhteeseen. Oletuksena huollettavuusluokittelut ovat: A = 0-0.05, B = 0.06-0.1. C = 0.11-0.20, D = 0.21.-0.5, E=0.51-1.

  Vaihtoehtoisesti huollettavuusarviointi voidaan ilmoittaa sanomalla, jos jäljellä oleva korjaus kustannus on: 5% ajasta, joka on jo mennyt sovellukseen, luokitus

on A. Välillä 6 – 10% luokitus on B. Välillä 11 – 20% luokitus on C. Välillä 21 – 50% luokitus on D. Kaikki yli 50% on luokitukseltaan D.

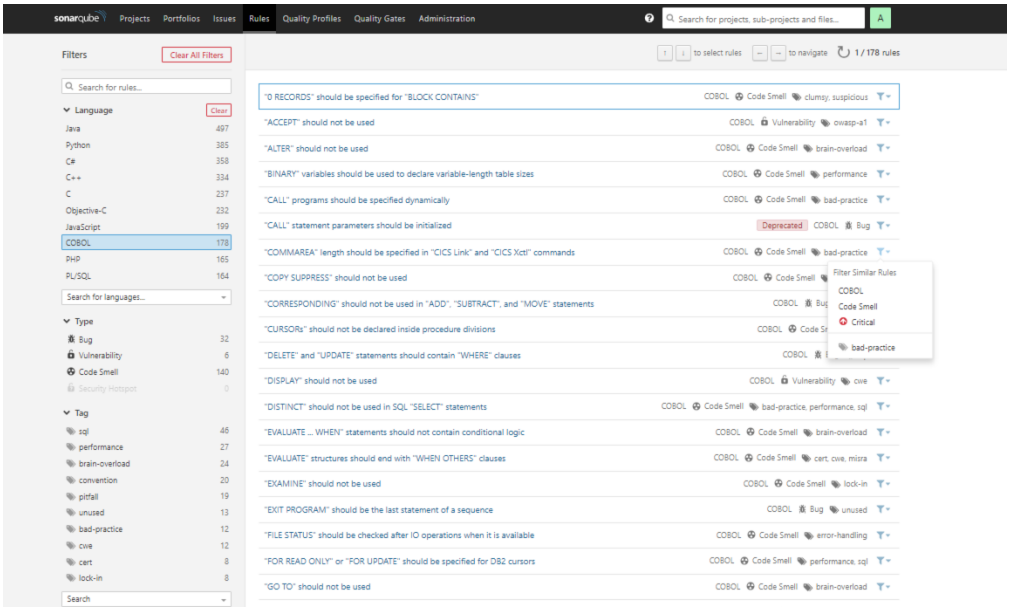
- Luotettavuusluokka uudelle ohjelman lähdekielelle (Reliability Rating on New Code). Luokat ovat A = ei yhtään vikoja, B = vähintään 1 vähäinen vika, C = vähintään 1 suuri vika, D = vähintään 1 kriittinen vika, E = vähintään 1 esteellinen vika.

- Turvallisuusluokitus uudelle ohjelman lähdekielelle (Security Rating on New Code). Luokat ovat A = ei haavoittuvuuksia, B = vähintään 1 vähäinen haavoittuvuus, C = vähintään 1 suuri haavoittuvuus, D = vähintään 1 kriittinen haavoittuvuus, E = vähintään 1 esteellinen haavoittuvuus.

Huomionarvoista laatuportin osalta on se, että oletuksena oleva metriikka huomioi vain uuden ohjelman lähdekielen ja sen tuomat muutokset. Mahdollisuutena on kuitenkin lisätä lukuisten erilaisten metriikoiden lisäksi erilaisia laatuportteja.

Ohjelman lähdekielen analysointi perustuu SonarQubessa sen sisäänrakennettuihin sääntöihin. COBOL-ohjelman lähdekielessä sääntöjä on 178 kpl, jotka keskittyvät ohjelman lähdekielessä oleviin logiikan virtausvirheisiin, huonoihin ohjelmointitapoihin, tiedon typistämisen ongelmiin, tiedon tarkkuuden menettämiseen ja käyttämättömään ohjelman lähdekieleen. Tietokannan kyselykielellä PL/SQL on olemassa sääntöjä 164 kpl. Säännöt ovat saatavissa kehittäjille paikallisella palvelimella suoraan web-selaimen kautta tai SonarLintissä Eclipsessä reaaliaikaisesti ohjelmoitaessa.

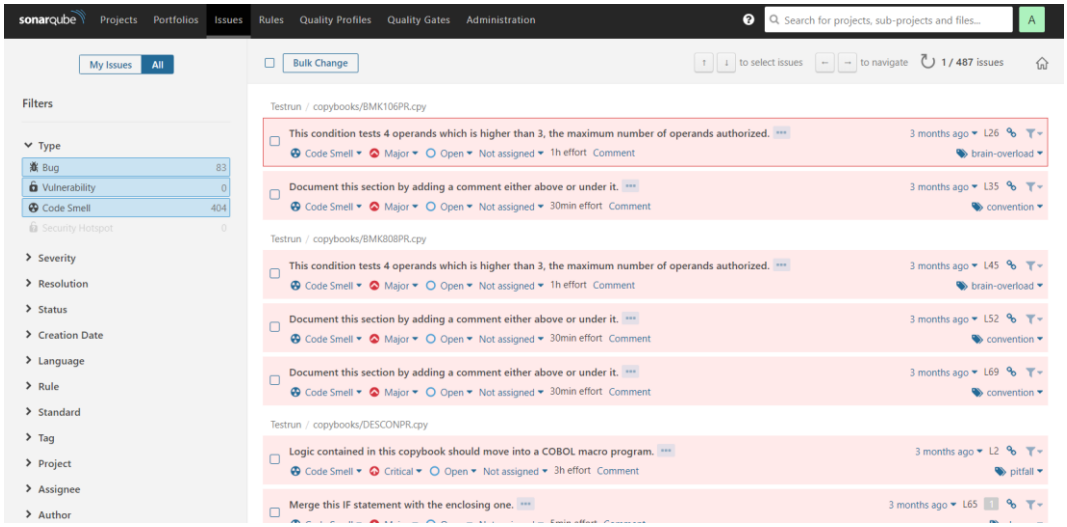
Jokaiselle säännölle on kuvattuna sille kuuluva ohjelmointikieli, ongelman tyyppi ja kuvaavampi sana mitä ongelma koskee. Ongelman vakavuus tulee myös tässä esille valittaessa sääntöä kuvan 19 mukaisesti.



Kuva 19. Kuvankaappaus SonarQubessa olevista säännöistä. Tässä kuvassa on valittuna COBOL-ohjelmointikieli.

Ohjelmien ohjelman lähdekielestä havaitut ongelmat löytyvät ongelmat (issues) -välilehdestä. Oletuksena kaikki ohjelman lähdekielessä olevat ongelmat ovat listattuna tällä sivulla. Sivun vasemmanpuoleisilla suodattimilla voidaan esim. tarkentaa minkälaisesta ongelmien tyypistä on kysymys ja mikä on näiden ongelmatyyppien nykyinen tila, eli onko käyttäjä näihin puuttunut.

Kuvassa 19 näkyvien ongelmientyyppien, vakavuuksien, arvioitu aika ongelmien korjaamiseksi ja tarkempien tietojen listatuista tiedoista vastaa SonarQuben oma algoritmi. Ongelmien listauksessa olevien tyyppien, vaivannäön ja tarkempien tietojen tarkoituksena on tarjota sovelluskehittäjälle mahdollisimman tarkan kuvauksen ongelmasta ja työmäärästä löydetyistä ongelmista lähdekielisissä ohjelmissa, jolloin ongelman korjaamiseen ja sen ajankohtaan sovelluskehittäjän on helpompi tarttua.

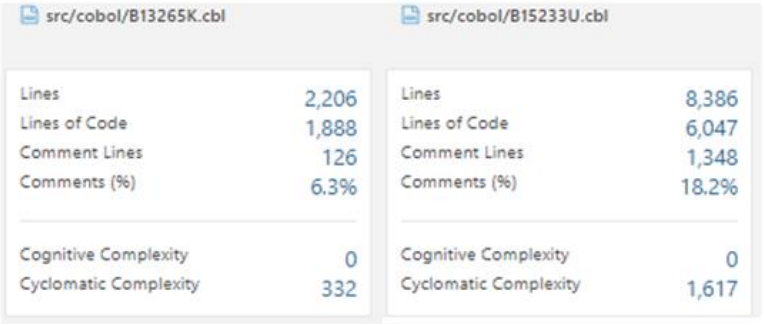


Kuva 20. Kuvankaappaus ongelmat-välilehdestä (Issues)

6.4 Ohjelmien analysoinnin jälkeiset tulokset

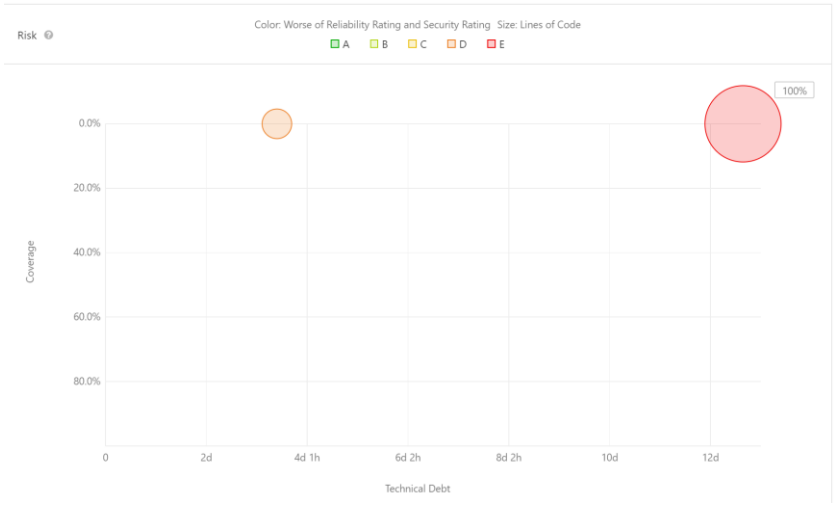
Analysoitaviksi olivat tarkoituksellisesti valikoituneet 2 kappaletta hyvin erilaista COBOL-ohjelmaa. Ensimmäinen näistä COBOL-ohjelmista (B13265K) on kohtuullisen uusi, joka on tässä tapauksessa noin. 2 vuotta vanha. Kokemuksien mukaan tämä ohjelma on kohtuullisen helppo ylläpitää, ymmärtää ja on kohdannut elinkaarensa aikana vähän ongelmia ja ylläpidon tarvetta.

Toinen analysoitavista COBOL-ohjelmista (B15233U) on ylläpito kommenttien perusteella ainakin 32 vuotta vanha ja on kokemuksien mukaan vaikealukuinen sekä haastava ylläpitää ja kehittää.

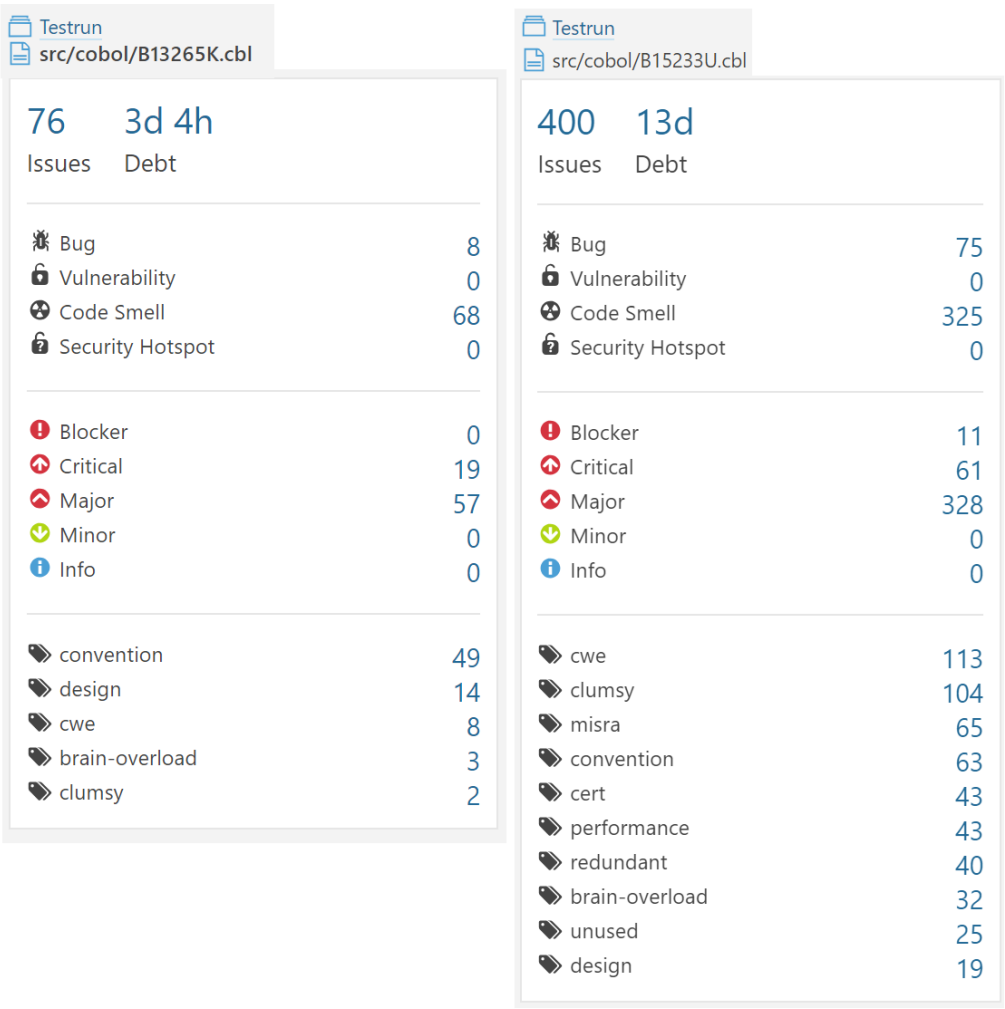


Kuva 21. Kuvankaappaus SonarQuben antamista perustiedoista kahdesta analysoitavasta ohjelmasta.

Kahden ohjelman välisistä perustiedoista on selvästi havaittavissa, että suuremmassa ohjelmassa on kommentoinnin prosentuaalinen määrä paljon suurempi kuin pienemmässä ohjelmassa. Aiempien kokemuksien mukaan ongelmallisemmassa ohjelmassa on syklomaattisen kompleksisuuden luku myös huomattavasti korkeampi.



Kuva 22. Kuvankaappaus SonarQuben antamasta yleiskatsauksesta, jossa vaaka-akselissa tekninen velka (technical debt) ja pystyakselissa kattavuus (coverage).



Kuva 23. Kuvankaappaus SonarQuben analysoinnin tuloksista.

Kehittäjien aiempien kokemuksien perusteella SonarQuben analysoinnin tuloksena ohjelmien B13265K ja B15233U välinen ero teknisessä velassa (Debt) ja ongelmia kokonaismäärää (Issues) voidaan pitää odotetunlaisena. Vikoja (bug) kuvan 23 mukaisesti ohjelmasta B15233U löytyi selvästi enemmän kuin B132265K ohjelmasta.

Kahden ohjelman analysoinnin tuloksien perusteella vaikuttaa siltä, että bug -tyyppiset viat ovat vakavuudeltaan aina korkeimpia. Tämä tarkoittaa ohjelman B13265K osalta 4 kpl kriittistä (Critical) vikaa ja 4 kpl suurta (Major) vikaa. Ohjelman B15233U osalta 11 kpl esteellistä (Blocker) vikaa, 35 kriittistä (Critical) vikaa ja 29 suurta (Major) vikaa. Huomionarvoista on, että ohjelman B15233U jokainen vakavuudeltaan esteellinen (Blocker) vika oli bug -tyyppinen.

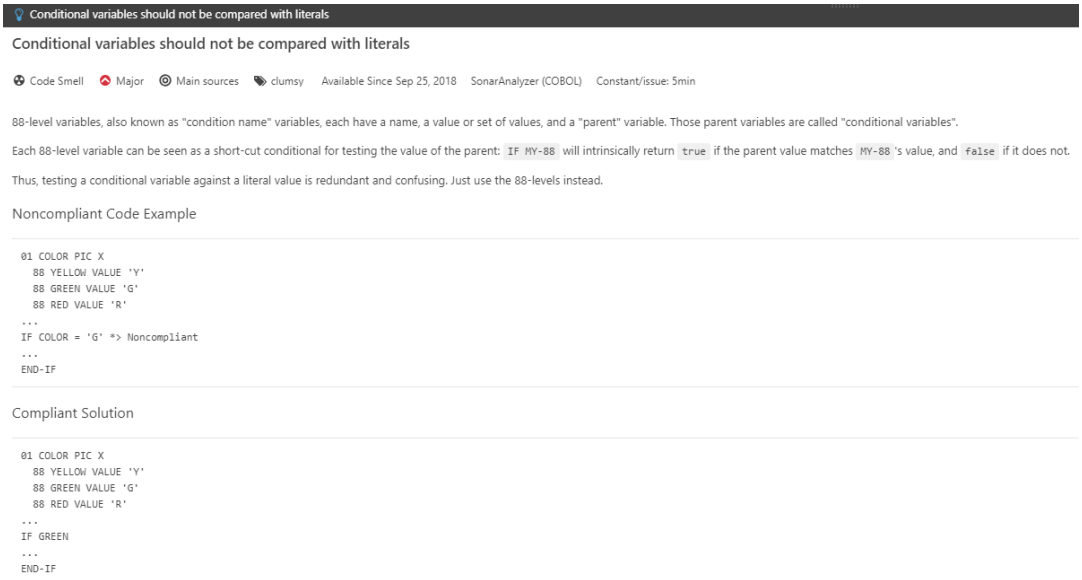
B13265K-ohjelmassa suurin osa analyysissa havaituissa vioista liittyivät hyviin ohjelmointisääntöihin (convention) joita oli 49 kpl. Suurin osa näistä ohjelmointisääntöihin liittyvistä ongelmista olivat kommentointi ja muotoiluvirheet ohjelman lähdekielessä. Esimerkkinä näistä, ehtolauseiden puutteelliset sisennykset ja lohkojen (section) kommentoimatta jättäminen. Kaikki ohjelmointisääntöihin liittyvät ongelmat, jotka havaittiin B13265K-ohjelmassa eivät vaikuta millään tavalla ohjelman toimivuuteen, vaan kyse on ohjelman lähdekielen luettavuuteen ja hyviin käytäntöihin liittyvistä asioista.
Suunnitteluun (design) liittyvät ongelmat (14 kpl) olivat liiallinen duplikaattien käyttö. SonarQuben ehdotus näiden korjaamiseksi on määrittää vakio duplikaattien käytön sijaan. Suunnitteluun liittyvät ongelmat vaikuttavat ohjelman ylläpidettävyyteen, koska jos ylläpitovaiheessa tehdään duplikaatteihin muutoksia, voi paikkoja olla monia, joihin muutos täytyy tehdä yhden vakion määrittämisen sijaan.
Yhteinen heikkouslueteloon (cwe) liittyviä ongelmia löytyi 8 kpl. Nämä ongelmat liittyivät tiedonsiirtoon numeerisesta kentästä aakkosnumeerisiin kenttiin. Ongelman ratkaisuksi ehdotettiin vastaamaan toisiaan. Tämä ongelma voi pahimmassa tapauksessa vaikuttaa ohjelman toimintaan virheellisesti.
Aivojen ylikuormitukseen (brain-overload) liittyviä ongelmia löytyi 3 kpl. Nämä ongelmat johtuivat ehtolauseiden liiallisesta määrästä yhdessä osassa, joka näin ollen on johtanut liialliseen kompleksisuuteen. Tämä vaikuttaa ensisijaisesti ohjelman lähdekielen luettavuuteen negatiivisesti, sekä vaikeuttaa huomattavasti ohjelman ylläpitotehtäviä.
Kömpelömäisyyteen (clumsy) liittyviä ongelmia löytyi 2 kpl, jotka liittyivät ehtolauseen alku ja loppupisteeseen. Nämä ongelmat vaikuttavat ohjelman lähdekielen luettavuuteen.

B15233U-ohjelmassa nousi lukumäärältään suurimpina ongelmina esiin kaksi erilaista ongelmatyyppiä, jotka olivat yhteiseen heikkouslueteloon (cwe) liittyvät ongelmat 113 kpl ja kömpelömäisyyteen (clumsy) liittyvät ongelmat 104 kpl.
Yhteiseen heikkouslueteloon (cwe) olivat tyyliltään tiedonsiirron ongelmia numeerisista kentistä aakkosnumeerisiin kenttiin, liian isojen lukujen siirtäminen liian pieneksi määriteltyihin kenttiin ja boolean lausekkeiden tarpeeton käyttö. Liian isojen lukujen siirtäminen liian pieniin kenttiin aiheuttaa tiedon katkeamisen (data truncation), jolloin pahimmassa tapauksessa siirretty data on käyttökelvotonta tai vääristää liikaa haluttua tulosta.

Kömpelömäisyyteen (clumsy) liittyvissä ongelmissa esiin nousi tarpeeton ja sekava virhetilanteen toteaminen ohjelman lähdekielessä, jossa virhetilannetta verrataan jollain luvulla.

Kuva 24. SonarQuben havaitsema ongelma virhetilanteen käsittelystä ohjelman lähdekielessä ja ratkaisuehdotus.

SonarQube ehdottaa kuvan 24 mukaisesti yksinkertaisempaa ja helppolukuisempaa ratkaisua esimerkkeineen.



Kuva 25. SonarQuben yksityiskohtaisempi selitys esimerkkeineen ratkaisuehdotuksesta.

MISRA-tunnisteella liittyviä ongelmia löytyi 65 kpl. MISRA-standardi on SonarQuben dokumentoinnin (docs.sonarqube.org 2019) mukaan lähinnä tarkoitettu C ja C++-kielille, mutta monet säännöt eivät ole kielikohtaisia. Kaikki tällä tunnisteella luokitellut ongelmilla oli myös yksi tai useampi muu tunniste, näin ei voida yksilöidä kaikkia löytyviä ongelmia vain MISRA-tunnisteella. Monet tällä tunnisteella löydetyistä ongelmista liittyvät hyviin ohjelmointikäytäntöihin, kuten vanhaan ohjelman lähdekieleen, joka on kommentoitu pois ohjelmasta.

Hyviin ohjelmointisääntöihin (convention) liittyviä ongelmia löytyi 63 kpl. Kaikki tällä tunnisteella löydetyt ongelmat viittaavat hyviin ohjelmointikäytäntöihin, kuten puutteelliseen kommentointiin ja ehtolauseiden puutteellisiin sisennyksiin.

CERT-tunnisteella löytyviä ongelmia löytyi 43 kpl. SonarQuben dokumentaation (docs.sonarqube.org 2019) mukaan CERT-standardeja on kolme erilaista, eri ohjelmointikielelle. Standardien säännöt eivät kuitenkaan ole kielikohtaisia, vaan noudattelevat hyviä ohjelmointikäytäntöjä. Kaikki tällä tunnisteella löydetyt ongelmat viittaavatkin tarpeettomiin alaehtojen käyttöihin.

Suorituskykyyn (performance) liittyviä ongelmia löytyi 43 kpl. Suorituskykyyn liittyvissä ongelmissa voidaan nostaa havaintojen perusteella kaksi erilaista ohjelman lähdekielessä havaittua tapausta, jotka toistuivat. SonarQube ehdottaa yleisesti, että lausekkeen tulisi aina olla indeksoitu, jos vain mahdollista COBOL-taulukoita käsiteltäessä. Tapauksissa, jossa indeksointi ei ole mahdollista ehdotetaan välttämään numeronäytön muuttujaa, kun halutaan käyttää taulukon elementtejä. Sen sijaan, ehdotetaan käyttämään binääri tai pakattuja-muuttujia, koska prosessori voi käsitellä ne tehokkaammin. Toisena suorituskykyyn vaikuttavana ongelma oli eri formaateissa olevien numeeristen arvojen vertailu. Esimerkkinä pakattujen kenttien vertailu COMP-3 ja COMP-4 välillä aiheuttaa hitautta, koska tällaisessa tapauksessa joudutaan tekemään konversio ennen vertailua, mikä vaikuttaa suorituskykyyn negatiivisesti.

Tarpeettomuuteen (redundant) liittyviä ongelmia löytyi 40 kpl. Ongelmat liittyvät hyviin ohjelmointikäytäntöihin ja ovat tyyliltään samanlaisia, kun CERT-tunnisteella löydetyt ongelmat, eli tarpeettomien alaehtojen käyttö.

Aivojen ylikuormitukseen (brain-overload) liittyviä ongelmia löytyi 32 kpl. Kaikki näistä ongelmista liittyvät liialliseen kompleksisuuteen ehtolauseissa ohjelman lähdekielessä.

Käyttämättömiä (unused) osiin liittyviä ongelmia löytyi 25 kpl. Suurin osa näistä on ohjelman lähdekielessä vanhaa ohjelman lähdekieltä, joka on kommentoitu pois. Kolmessa eri kohdassa EVALUATE-lauseessa tila on aina tosi. Nämä kolme erilaista ongelmaa ovat luokiteltuna bug -tyyppiseksi.

Suunnitteluun (design) liittyviä ongelmia löytyi 19 kpl. Näiden kaikkien tarkempi ongelma kuvaus oli liiallinen duplikaattien käyttö.

7. TULOKSIEN ANALYSOINTI

Ohjelmistoratkaisulle asetettujen kriteereiden osalta vaihtoehtoja eri ohjelmistoratkaisuiden toimittajien välillä ei ollut muita kuin käytännössä SonarSource. Ohjelmistoratkaisun kokeiluversion saaminen pelkästään tutkimuskäyttöön ei ollut mahdollista. Vasta yrityksen kautta tehdyt useat pyynnöt kokeiluversion saamiseksi käyttöön kahden kuukauden aikana tuottivat tulosta. Mahdottomuus yksityishenkilönä kokeiluversion saamista käyttöön ja ylipäätään kokeiluversion nihkeä saanti saattavat kertoa kuinka vahvasti keskuskonemaailmassa esiintyvä COBOL-ohjelmointikieli on tarkoitettu kaupallishallinnollisiin sovelluksiin, eikä näin ollen siihen liittyviä sovelluksia helposti luovuteta muualle ilman selvää ansaintamahdollisuutta ohjelmistoratkaisua tarjoavalle yritykselle.

Ennen staattisen analyysityökalun hankkimista ensimmäisenä olisi syytä pohtia nykyisiä organisaation käytössä olevia laadunvarmistusprosesseja ja pohtia seuraavia asioita kuten kuinka tehokkaita ovat nykyiset laadunvarmistusprosessit, mikä on niiden kulurakenne suhteutettuna niistä saataviin hyötyihin ja arvioida mitkä ohjelmistometriikan kynnysarvot sopisivat nykyisten ohjelmistojen ja ohjelmistotöiden mittaamiseen.
SonarQuben käyttöönotossa kohde organisaatiossa olisi hyvä idea jakaa sovellusalueet eri projekteille SonarQubessa. Tällä tavalla voidaan saada kokonaiskuva tietyn sovellusalueen mahdollisista jo olevista ongelmakohdista sekä tulevista resurssoinneista pahimpien ongelmien korjaamiseen ennen kuin ne realisoituvat tuotannossa.

Ohjelmistojen katselmoinneissa Faganin (1986) mukaan parhaat tulokset ovat saavutettu prosessilla, joka on ollut muodollinen. Myös hyvä koulutus ennen katselmointien aloittamista mukana oleville henkilöille ja katselmoinneista vastaavalle johdolle on ollut tärkeässä roolissa hyvien tulosten ja myönteisen vastaanoton kannalta.
SonarQuben kaltaisessa ohjelmistoratkaisussa ei tarvitse itse käsin etsiä virheitä, vaan ne etsitään automaattisesti ohjelman analyysin avulla. Siitä huolimatta, voisi olla syytä kiinnittää huomiota siihen, että voitaisiinko jotenkin hyödyntää joitain muodollisen tarkastusprosessin vaiheita, koska ne ovat osoittautuneet tehokkaaksi niiden muodollisuuden, riittävän koulutuksen ja roolituksen takia. Tästä esimerkkinä aiemmin esitellyn Grahamin (2008) tarkastusprosessi kohdat, uudelleen työstäminen (rework) ja seuranta (follow-up).

Näissä vaiheissa nousee esiin se, että löydettyihin ongelmiin on tartuttu virheiden kirjaamisella tai korjaamisella. Tällä tavalla voidaan prosessin tuomat muutokset osoittaa oikeiksi. Faganin (1986) mukaan asenteet ohjelmistojen katselmointeja kohtaan ovat olennainen osa siihen, että ne onnistuvat. Yksi parhaista tavoista hänen mukaansa tähän on vaikuttaa, on johdon tarpeeksi korkea tietämystaso katselmoinneista.
SonarQuben kaltaisella ohjelmistoratkaisulla voisi pohtia myös, että olisi saatava sitoutettua ohjelmistokehittäjät oikeanlaiseen katselmointityöhön alusta alkaen, sekä pystyä motivoimaan siitä saatavat hyödyt. Olennaista olisi myös mukauttaa tällainen laaduntarkastus prosessi päivittäiseen ohjelmistokehitykseen, sekä seurata sen etenemistä. Jokaisella laadunhallintaan osallistuvalla tulisi myös olla selkeä roolitus tässä prosessissa ja mistä osasta kukin vastaa.

Keskuskonejärjestelmän koosta riippuen SonarQuben analyysillä käydään läpi mahdollisesti kaikki järjestelmässä olevat ohjelman lähdekielen rivit, jotka voivat nousta määrältään kymmeniin miljooniin. Analyysin jälkeen ohjelman lähdekielen suuren määrän vuoksi olisi järkevää käsin käydä läpi automaattisesti löydettyjä ongelmia ja pohtia ongelmatyyppien uudelleen määrittämisen mahdollisuutta, koska SonarQuben automaattinen ongelmien määritys ja kategoriointi eivät välttämättä kaikilta osin vastaa kohde organisaation näkemyksiä asiasta. Tämän lisäksi, koska löytyviä ongelmia on todennäköisesti suuri määrä, olisikin tarkkaan pohdittava minkälaisten löydettyjen ongelmien korjaaminen ja mahdollinen uudelleen optimointi olisi vaivannäön arvoisia.
Taulukossa 7 on esiteltynä SonarQubella mahdollisesti saavutettavat asiat, jotka vaikuttavat positiivisella tavalla nykyiseen sovelluskehitystyöhön.

Taulukko 7. SonarQubella mahdollisesti saavutettavia asioita.

Saavutettava asia	Tarkempi kuvaus
Aika-arvio ongelman tai ongelmien korjaamiseen	Voidaan esim. perustaa osa työmääräarvioista SonarQubesta saatuihin aika-arvioihin.
Tilannekuva sovellusalueesta	Saadaan kuvaus eri sovellusalueista, josta suoraan voidaan nähdä niiden ongelmat laadullisesta näkökulmasta, joita SonarQube tarjoaa
Tilannekuva sovellusalueen trendistä tiettynä ajanjaksona	Voidaan saada käsitys laadullisesta kehittymisestä ja vastauksia kysymykseen: ollaanko menossa laadullisesta näkökulmasta oikeaan suuntaan jatkuvassa sovelluskehityksessä?
Visualisointi eri projekteille, jolla saadaan kuva projektista vertailukohtien avulla.	Voidaan verrata useiden eri projektien tai niiden osien nykyistä tilaa visuaalisesti erilaisten vertailukohtien avulla. Vertailukohtina projektien kesken voi olla mm. luotettavuus, huollettavuus ja riskit.

8. JOHTOPÄÄTÖKSET

Tämän tutkimuksen tavoitteena oli löytää markkinoilta valmis ohjelmistoratkaisu, joka soveltuu COBOL-ohjelman lähdekielen katselmointeihin sekä päästä pilotoimaan sitä. Lähdekielisten ohjelmien katselmointien avulla oli pystyttävä laadullisesti parantamaan ohjelmistokehittäjien tuottamaa ohjelman lähdekieltä, parantamaan ymmärrystä hyvistä ohjelmointikäytännöistä, sekä tätä kautta pystyä kehittämään yrityksen sisäisiä ohjelmointi standardeja.

Tehtyjen tutkimustulosten ja pilotoinnin perusteella staattiseen analyysiin keskittyvä SonarSourcen tuotteet, SonarQube ja SonarLint voidaan todeta tarjoavan mahdollisuuden lähdekielisessä ohjelmassa olevien ongelmien löytämiseen isosta määrästä ohjelmia tavalla, joka ei järkevällä tavalla ole ihmisvoimin manuaalisesti toteutettavissa. SonarQubessa analysoitujen ohjelmien jokainen löydetty ongelma on tarkasti määriteltynä erilaisilla tunnisteilla, jotka mahdollisimman tarkasti kuvaavat minkälaisesta ongelmasta on kyse, kuinka vakava se on ja arvioitu aika ongelman korjaamiseksi. Tämän lisäksi ongelmaan tarjotaan esimerkillistä vastausta, kuinka ongelma voidaan korjata. Tämä voi huomattavasti auttaa sovelluskehittäjää työssään ongelman ratkaisua etsiessä. SonarQuben mukana tuleva SonarLint on IDEen liitettävä lisäosa, joka tarjoaa reaaliaikaisen ohjelman lähdekielen analysoinnin, jolloin se mahdollistaa toteutusvaiheessa olevan uuden ohjelman lähdekielen laadukkaan tuoton ennen kuin uusi ohjelman lähdekieli pääsee testausvaiheeseen.

Pilotointivaiheessa analysoitiin kahta erilaista ohjelmaa, jotka olivat ennakkotietojen perusteella yksinkertainen ylläpitää ja hankala ylläpitää. SonarQube:lla ohjelmien analysoinnin jälkeen näiden kahden ohjelman ongelma tyypeissä, määrässä ja vakavuuksissa oli nähtävä selkeä ero toisiinsa nähden ja tulokset vastasivat hyvin odotuksia ennakkotietojen perusteella.

Laadun varmistamiseksi SonarQuben laatuportti (quality gate) on toimintaperiaatteeltaan yksinkertainen, mutta tehokas. Toisaalta sen tehokkuuden määrittelee sille annetut metriikat ja niiden kynnysarvot. Laatuporttia voisi siis kutsua eräänlaiseksi suodattimeksi,

jonka läpi pääsee vain metriikan täyttämät mitat ja se tietyllä tavalla pakottaa sovelluskehittäjän laadukkaisiin tuotoksiin uuden ohjelman lähdekielen osalta. Sisäänrakennetut kynnysarvot ovat laatuportissa hyvä pohja, mutta täydellisen hyödyn saamiseksi olisi laadittava useampia laatuportteja, jotka sopisivat tarkoituksiinsa parhaiten. Uusien laatuporttien laadintaan olisi suotavaa käyttää tarpeeksi resursseja ja aikaa, koska se on yksi tärkeimpiä ominaisuuksia uuden ohjelman lähdekielen laadunhallinnassa SonarQubessa. Yksistään SonarQuben käyttöönottaminen sellaisenaan ei ole ideaali tapa, vaan se vaatii syvällistä pohdintaa ja yhteistyötä ohjelmistokehittäjien kanssa laatuporttien kynnysarvojen asettamisesta sopiviksi ja ongelmatyyppien uudelleen määritteleminen parhailla näkemillään tavoilla. Uudelleen määriteltäessä ongelmien tunnisteita ja vakavuuksia SonarQubessa, jotka vaikuttavat ohjelman lähdekielen laadukkuuteen ja vanhojen ongelmien korjaamiseen, aiheuttanee se hyvää keskustelua myös organisaation sisäisistä laatustandardeista. Lisäksi tämä lisää pohdintaa siitä, mitä voidaan saavuttaa laadukkaamman ohjelman lähdekielen tuottamisella organisaatiossa ja kuinka se tulee näkymään asiakkaan puolella.

LÄHDELUETTELO

Abrahamsson Pekka, Salo Outi, Ronkainen Jussi, Warsta Juhani (2002). *Agile software development methods* [Viitattu 15.08.2018]. VTT Publications. ISBN: 951–38–6010–8

agilenutshell.com (2018). *Agile vs Waterfall* [Verkkodokumentti]. [Viitattu 13.08.2018]. Saatavissa: http://www.agilenutshell.com/agile_vs_waterfall

Banker D. Rajiv, Datar M. Srikant, Kemerer F. Chris, Zweig Dani (1993). *Sofware complexity and maintenance costs* [Verkkodokumentti]. [Viitattu 03.08.2018]. Saatavissa: https://dl-acm-org.proxy.uwasa.fi/citation.cfm?id=163375

Beach Gary (2014). *Cobol Is Dead. Long Live Cobol* [Verkkodokumentti]. [Viitattu 05.07.2018]. Saatavissa: https://blogs.wsj.com/cio/2014/10/02/cobol-is-dead-long-live-cobol/

Bloom Jonathan (2015). *Five Reasons You MUST Measure Software Complexity* [Verkkodokumentti]. [Viitattu 04.09.2018]. Saatavissa: https://www.castsoftware.com/blog/five-reasons-to-measure-software-complexity

businessinsider.com (2017). *Global Mainframes Market 2017-2021*[Verkkodokumentti]. [Viitattu 17.09.2018]. Saatavissa: https://markets.businessinsider.com/news/stocks/global-mainframes-market-2017-2021-with-bmc-software-dell-emc-fujitsu-hitachi-data-system-ibm-dominating-1002746077

Coudert Olivier (2011). *What is software quality?* [Verkkodokumentti]. [Viitattu 15.07.2018]. Saatavissa: http://www.ocoudert.com/blog/2011/04/09/what-is-software-quality/

cs.helsinki.fi (2009). *Ohjelmistoprosessit ja ohjelmistojen laatu* [Verkkodokumentti]. [Viitattu 10.08.2018]. Saatavissa: https://www.cs.helsinki.fi/u/taina/opol/k-2009/pdf/luku-6_2.pdf

docs.sonarqube.org (2019) Built-in Rule Tags [Verkkodokumentti] [Viitattu 02.02.2019]. Saatavissa: https://docs.sonarqube.org/latest/user-guide/built-in-rule-tags/

Donaldson E. Scot, Siegel G. Stanley (2000). *Successful Software Development 2nd Edition*. Prentice Hall. ISBN: 0-13-086826-4

Fagan E. Michael (1986). *Advances in Software Inspections* [Viitattu 20.08.2018]. IEEE. DOI: 10.1109/TSE.1986.6312976

Fenton Norman, Bieman James (2014) *Software Metrics: A Rigorous and Practical Approach*. [Viitattu 28.08.2018] CRC Press. ISBN 9781439838228

Graham Dorothy, Veenendaal van Erik, Evans Isabel, Black Rex (2008). *Foundations of Software Testing*: ISTQB Certification [Viitattu 18.08.2018]. Cengage Learning Emea. ISBN: 978-1844809899

Harjumaa Lasse (2005). *Improving The Software Inspection Process With Patterns* [Viitattu 22.08.2018]. University of Oulu. ISBN 951-42-7894-1

Hummel Benjamin (2014). *McCabe's Cyclomatic Complexity and Why We Don't Use It* [Verkkodokumentti]. [Viitattu 03.09.2018]. Saatavissa: https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/

ibm.com (2018). *Mainframe concepts* [Verkkodokumentti]. [Viitattu 20.09.2018]. Saatavissa:https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zmainframe/toc.htm

IEEE Std 1028-2008 (2009). *IEEE Standard for Software Reviews and Audits* [Viitattu 22.08.2018]

ISO 25000 (2018). *The ISO/IEC 25000 series of standards* [Verkkodokumentti]. [Viitattu 16.07.2018]. Saatavissa: https://iso25000.com/index.php/en/iso-25000-standards

ISO 25010 (2018). *ISO/IEC 25010* [Verkkodokumentti]. [Viitattu 25.07.2018]. Saatavissa: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Kan H. Stephen (2002). *Metrics and Models in Software Quality Engineering (2nd Edition)*. [Viitattu 06.09.2018]. Addison-Wesley Professional ISBN-10: 0133988082

Kangasniemi, P. (2018). Senior Software Developer [Puhelinhaastattelu]. [Viitattu 17.09.2018]. Tieto Oyj

Laitenberger Oliver (2001). *A Survey of Software Inspection Technologies* [Verkkodokumentti]. [Viitattu 27.08.2018]. Saatavissa: https://pdfs.semanticscholar.org/b2f6/037f949a2bddc662ba604e32295fd5d67e17.pdf

Mainframes 360 (2018). *Sizing up your mainframe - MIPS and MSU's* [Verkkodokumentti]. [Viitattu 06.08.2018]. Saatavissa: http://www.mainframes360.com/2014/02/sizing-up-your-mainframe-mips-and-msus.html

mainframestechhelp.com (2018). *COBOL Perform Through* [Verkkodokumentti]. [Viitattu 03.10.2018]. Saatavissa: http://www.mainframestechhelp.com/tutorials/cobol/cobol-perform-through.htm

Manifesto for Agile Software Development (2018). *Manifesto for Agile Software Development* [Verkkodokumentti]. [Viitattu 13.08.2018]. Saatavissa: https://www.agilealliance.org/agile101/the-agile-manifesto/

Myers J. Glenford, Sandler Corey, Badgett Tom (2011). *The Art of Software Testing, 3rd Edition*. John Wiley & Sons. ISBN: 978-1-118-03196-4

Nanalyze.com (2017). *What Do IBM Mainframe Computers Look Like Today?* [Verkkodokumentti]. [Viitattu 15.09.2018]. Saatavissa: https://www.nanalyze.com/2017/10/ibm-mainframe-computers-today/

Paakki Jukka et al. (2015). *Ohjelmistoprosessit ja ohjelmistojen laatu* [Verkkodokumentti]. [Viitattu 12.07.2018]. Saatavissa: https://www.cs.helsinki.fi/u/aptuovin/laatu/k15/Luento1_tulostettava.pdf

sfsedu.fi (2018). *ISO/IEC 25000 SQUARE ohjelmistojen ja järjestelmien laadun mittaaminen* [Verkkodokumentti]. Standardisoinnin oppilaitosportaali SFSedu [Viitattu 16.07.2018]. Saatavissa: https://www.slideshare.net/SFSedu/isoiec-25000-square-ohjelmistojen-ja-jrjestelmien-laadun-mittaaminen

Shepperd Martin (1988). *A critique of cyclomatic complexity as a software metric* [Verkkodokumentti]. Journal Software Engineering Volume 3 Issue 2, March 1988 Pages 30-36 [Viitattu 04.09.2018]. Saatavissa: https://dl.acm.org/citation.cfm?id=48322

Shirey G.C. (1992) *How Inspections Fail. Proceedings of the 9th International Conference on Testing Computer Software*: 151-159. [Viitattu 24.08.2018]

solace.com (2018). *Solace and HostBridge* [Verkkodokumentti]. [Viitattu 03.10.2018]. Saatavissa: https://solace.com/partners/hostbridge

sonarqube.org (2018). *Architecture and Integration* [Verkkodokumentti]. [Viitattu 08.10.2018]. Saatavissa: https://docs.sonarqube.org/display/SONAR/Architecture+and+Integration

sonarsource.com (2018). *SonarCOBOL* [Verkkodokumentti]. [Viitattu 15.10.2018]. Saatavissa: https://www.sonarsource.com/products/codeanalyzers/sonarcobol.html

sonarsource.com (2018). *Enterprise edition. Plans and pricing* [Verkkodokumentti]. [Viitattu 16.10.2018]. Saatavissa: https://www.sonarsource.com/plans-and-pricing/enterprise/

technologyuk.net (2018). The Waterfall Model [Verkkodokumentti]. [Viitattu 10.08.2018]. Saatavissa: http://www.technologyuk.net/software-development/sad/waterfall-model.shtml

tivia.fi (2013). *Tietojärjestelmien hankinta Suomessa* [Verkkodokumentti]. [Viitattu 31.07.2018].http://www.tivia.fi/sites/tivia.fi/files/liitteet/Tietoja%CC%88rjestelmien%20hankinta%20Suomessa%202013_0.pdf

Watson H. Arthur, McCabe J. Thomas (1996). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric* [Verkkodokumentti]. [Viitattu 31.08.2018]. Saatavissa: http://www.mccabe.com/pdf/mccabe-nist235r.pdf

wikipedia.org (2018). *IBM mainframe* [Verkkodokumentti]. [Viitattu 24.09.2018]. Saatavissa: https://en.wikipedia.org/wiki/IBM_mainframe

Wikiversity.org (2018). *Software testing / History of testing* [Verkkodokumentti]. [Viitattu 12.07.2018]. Saatavissa: https://en.wikiversity.org/wiki/Software_testing/History_of_testing

youtube.com (2019) *Modernize Your COBOL Testing Processes with Compuware and SonarSource* [Video]. [Viitattu 04.01.2019]. Saatavissa: https://youtu.be/y9KdemODWAA?t=2065