SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

OHJELMOINTIKÄYTÄNNÖT- JA DOKUMENTOINTI

TEKIJÄ: Joonas Onatsu
ESE14SE

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Sähkötekniikan tutkinto-ohjelma	
Työn tekijä(t) Joonas Onatsu	
Työn nimi Ohjelmointikäytännöt- ja dokumentointi	
Päiväys 21.10.2018	Sivumäärä/Liitteet 50 + 17(29)
Ohjaaja(t) Yliopettaja Väinö Maksimainen, Yliopettaja Arto Toppinen	
Toimeksiantaja/Yhteistyökumppani(t) Delfin Technologies Oy, Microkatu 1, 70211 Kuopio, Juha Pärnänen, Medical Device Engineer	

Tiivistelmä

Tämän opinnäytetyön tavoitteena oli kehittää ohjelmointi- ja dokumentointikäytäntö Delfin Technologies Oy:n tuotekehitysosastolle.

Työn aloitettiin tutustumalla ohjelmiston laatuun vaikuttaviin tekijöihin. Pääasiallisesti tässä osuudessa keskityttiin ohjelmointikäytäntöjen ja dokumentoinnin merkityksiin ohjelmiston laadun parantamisessa. Osiossa esitettiin esimerkkejä yritysten/yhteisöjen käyttämistä käytännöistä, sekä esimerkkejä huonosti kirjoitetun koodin aiheuttamasta koodin luettavuuden heikkenemisestä. Lisäksi osiossa perehdyttiin lääketieteelliseen käyttöön tarkoitettujen ohjelmistojen IEC 62304-standardiin.

Työn lopputuloksena koostettiin yleisistä ohjelmointi- ja dokumentointikäytännöistä parhaat käytännöt eli ns. ”best practices” ohjelmointi- ja dokumentaatiokäytäntödokumenttiin, ja olemassa olevat koodit päivitettiin ohjelmointikäytännön mukaiseksi. Lisäksi koodit dokumentointiin Doxygen-työkalulla.

Avainsanat
Ohjelmistokehitys, ohjelmointikäytännöt, dokumentointi, Doxygen, IEC 62304

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Electrical Engineering			
Author(s) Joonas Onatsu			
Title of Thesis Coding Guidelines and Documentation			
Date	21 October 2018	Pages/Appendices	50 + 17(29)
Supervisor(s) Mr. Väinö Maksimainen, Principal Lecturer, Mr. Arto Toppinen, Principal Lecturer			
Client Organisation /Partners Delfin Technologies Oy, Mr. Juha Pärnänen, Medical Device Engineer			

Abstract

The purpose of this thesis was to develop coding guidelines and documentation practices for the product development department of Delfin Technologies Oy.

The thesis was started by going through the factors affecting software quality. The focus on the software quality section was to investigate what kind of effect coding guidelines and documentation have on software quality. Few examples of publicly available coding guidelines were studied and presented as well as few examples of badly written code to show the degrading effect on code readability. The IEC 62304 medical device software standard was also studied and the effect it has on coding guidelines and documentation.

As a result of this thesis, the best practices from publicly available coding guidelines were collected and incorporated in to the coding guidelines and documentation document. The existing codes were also updated to conform with the coding guidelines, and the codes were documented with Doxygen.

Keywords
Software Development, coding guidelines, documentation, Doxygen, IEC 62304

SISÄLTÖ

1 JOHDANTO

Ohjelmistokehitys ei ole enää vain teknologiayritysten aluetta, vaan se koskettaa kasvavissa määrin kaikkia teollisuuden aloja, sekä pieniä että suuria yrityksiä ympäri maailmaa. Yritykset joutuvat kohtaamaan sen tosiasian, että laadukas ohjelmisto ja ensiluokkainen ohjelmistokehitys ovat yhtä välttämättömiä asioita menestyksen kannalta, kuin erinomainen myynti. Digitalisaation muokatessa kilpailua tuotteet ja palvelut ovat entistä enemmän riippuvaisia ohjelmistoista, jolla ne erottuvat kilpailijoista. Älypuhelimien ja muiden käyttöliittymien taustalla on kuluttajien vuorovaikutusta ohjaavia ohjelmistoja, ja puettavat laitteet mittaavat potilaiden ja urheilijoiden terveyttä ja suorituskykyä ohjelmistojen avulla. (Strålin, Gnanasambandam, Andén, Comella-Dorda ja Burkacky 2016, 5:9).

McKinseyn tekemässä tutkimuksessa kysyttiin 1300 yritykseltä ympäri maailmaa kolme kysymystä: *Mitä ohjelmistoa kehitetään, kuinka ohjelmistoa kehitetään sekä missä ohjelmistoa kehitetään*. Tutkimuksen mukaan yritysten paras neljännes saavutti yli kolme kertaa suuremman tuottavuuden, viisi kertaa suuremman ohjelmistokehityksen tuotannon ja kuusi kertaa vähemmän ohjelmistovirheitä, kuin alin neljännes. (Strålin ym. 2016, 9:17).

Ohjelmistokehityksen laatu on siten merkittävä kilpailutekijä ohjelmistokeskeisissä toimintaympäristöissä. Ohjelmistokehitykseen liittyviä laatutekijöitä- ja määritelmiä on useita. Tässä opinnäytetyössä käsitellään laatua ja sen merkitystä ohjelmistokehityksessä yleisellä tasolla, ja perehdytään ohjelmointikäytäntöjen sekä dokumentoinnin merkitykseen ohjelmistokehityksen laadun tekijöinä.

1.1 Lyhenteet ja määritelmät

- ISO – **I**nternational **S**tandardization **O**rganization
- SEI – **S**oftware **E**ngineering **I**nstitute
- IEC – **I**nternational **E**lectrotechnical **C**ommission
- Koodi – Ohjelmointikielen syntaksin mukaisesti kirjoitettu teksti eli ohjelmakoodi
- API – **A**pplication **P**rogramming **I**nterface eli rajapinta
- SOUP – **S**oftware **O**f **U**nknown **P**rovenance eli tuntemattoman alkuperän ohjelmisto

1.2 Yhteistyökumppanit ja tekijänoikeuksien haltijat tai muut tahot

Liitteen 3 ohjelmointi-ja dokumentaatiokäytäntö on Delfin Technologies Oy:n omaisuutta. Se on pois-tettu julkaistavasta versiosta tekijänoikeuksien- sekä salassapitovelvollisuuden vuoksi

2 LAATU

Chemuturin (2011, 1-2) näkemyksen mukaan sanaa *laatu* käytetään yleensä itsenäisenä terminä ja ihmiset kuvailevat tyytyväisyyttään tuotteisiin tai palveluihin sanomalla niitä *laadukkaiksi tuotteiksi*. Tyytymättömyyttään he puolestaan ilmaisevat sanalla *huonolaatuinen*. Useimmat ihmiset siten liittävät sanaan *laatu* implisiittisesti adjektiivin *hyvä*, vaikka sitä ei erikseen sanota. Laadun käsitteellä kuvataan siis tyytyväisyyttä tuotteeseen tai palveluun. Virallinen laadun määritelmä on esimerkiksi ISO9000-standardin mukaan *taso, jolla ominaiset ominaisuudet täyttävät vaatimukset*.

Kshirasagarin ja Priyadarshin (2008, 5) sekä Chemuturin (2011, 2) näkemysten mukaan laatu merkitsee erilaisia asioita eri tahoille. Chemuturi (2011, 2) listaa muun muassa seuraavanlaisia merkityksiä:

- Tuotteen käyttäjä: virheetön toiminta, luotettavuus, helppokäyttöisyys, vikasietoisuus sekä turvallisuus.
- Tuotteen valmistaja: vaatimuksien täyttyminen, oli vaatimuksien määrittäjä sitten valtiollinen taho, teollinen yhteistyöorganisaatio, standardointiorganisaatio tai jokin tuottajan sisäinen organisaatio.
- Palvelun käyttäjä: luotettavuus, helppo saatavuus sekä asiantunteva ja miellyttävä palvelu.
- Palvelun tuottaja: palvelun toimittaminen aikataulun puitteissa sekä vaatimusten ja standardien täyttyminen.

Kshirasagari ja Priyadarshi (2008, 519) listaavat puolestaan seuraavanlaisia näkemyksiä laadusta:

- Yliluonnollinen näkemys: Tässä näkemyksessä laatu on jotain mikä voidaan tunnistaa kokemuksen kautta. Laatu nähdään ideaalisena asiana, joka on liian monimutkainen määriteltäväksi, mutta esimerkiksi laadukas objekti tunnistetaan helposti.
- Käyttäjänäkemys: Käyttäjien kannalta laatu on käyttäjien tarpeiden ja odotusten täyttymistä. Käyttäjien vaatimukset ja odotukset eriävät suuresti henkilökohtaisten erojen vuoksi, ja tuotetta tai palvelua pidetään hyvälaatuisena, mikäli se täyttää suuren käyttäjäjoukon tarpeet ja odotukset.
- Valmistuksen näkemys: Valmistajien kannalta laatu on vaatimustenmukaisuutta, ja tuotteen tai palvelun poikkeama vaatimuksista nähdään laadun heikkenemisenä.
- Tuotenäkemys: Laatu on sidottu tuotteen ominaisten ominaisuuksien laatuun, jotka määrittävät sen ulkoisen laadun.
- Arvopohjainen näkemys: Tässä tapauksessa erinomaisuus ja arvo ovat nivoutuneet yhteen laadun muodossa. Tässä tapauksessa keskeinen idea on se, paljonko asiakas on valmis maksamaan laadusta. Laatu on merkityksetöntä, mikäli tuote on liian kallis ostettavaksi.

Kshirasagarin ja Priyadarshin (2008, 1) mukaan ihmiset pyrkivät etsimään laatua jokaisesta ihmisen tekemästä asiasta. Internetin kasvu, globaali kilpailu, alihankinta sekä monet muut tekijät ovat ajaneet eteenpäin laadun vallankumousta. Selviytyäkseen globaalissa taloudessa yritysten on valmistettava laadukkaita tuotteita tiukkojen aikataulujen puitteissa. Perinteiset, virheiden havainnointiin ja korjaukseen keskittyvät laadunvalvontamenetelmät ovat vaihtuneet uuteen menetelmään, jossa laatua pyritään valvomaan jokaisessa tuotekehityksen vaiheessa. Kshirasagarin ja Priyadarshin (2008, 1) mukaan tehokas laatuprosessi keskittyy muun muassa seuraaviin asioihin:

- Asiakasvaatimuksien runsas huomiointi.
- Jatkuva pyrkimys laadun parantamiseen.
- Mittaus/arviointiprosessien integrointi tuotekehitykseen.
- Laatukonseptin levittämiseen jokaiselle organisaation tasolle.
- Järjestelmätasoisen perspektiivin kehittäminen, joka painottuu menetelmiin ja prosesseihin.
- Hävikin vähentäminen jatkuvalla kehitystyöllä.

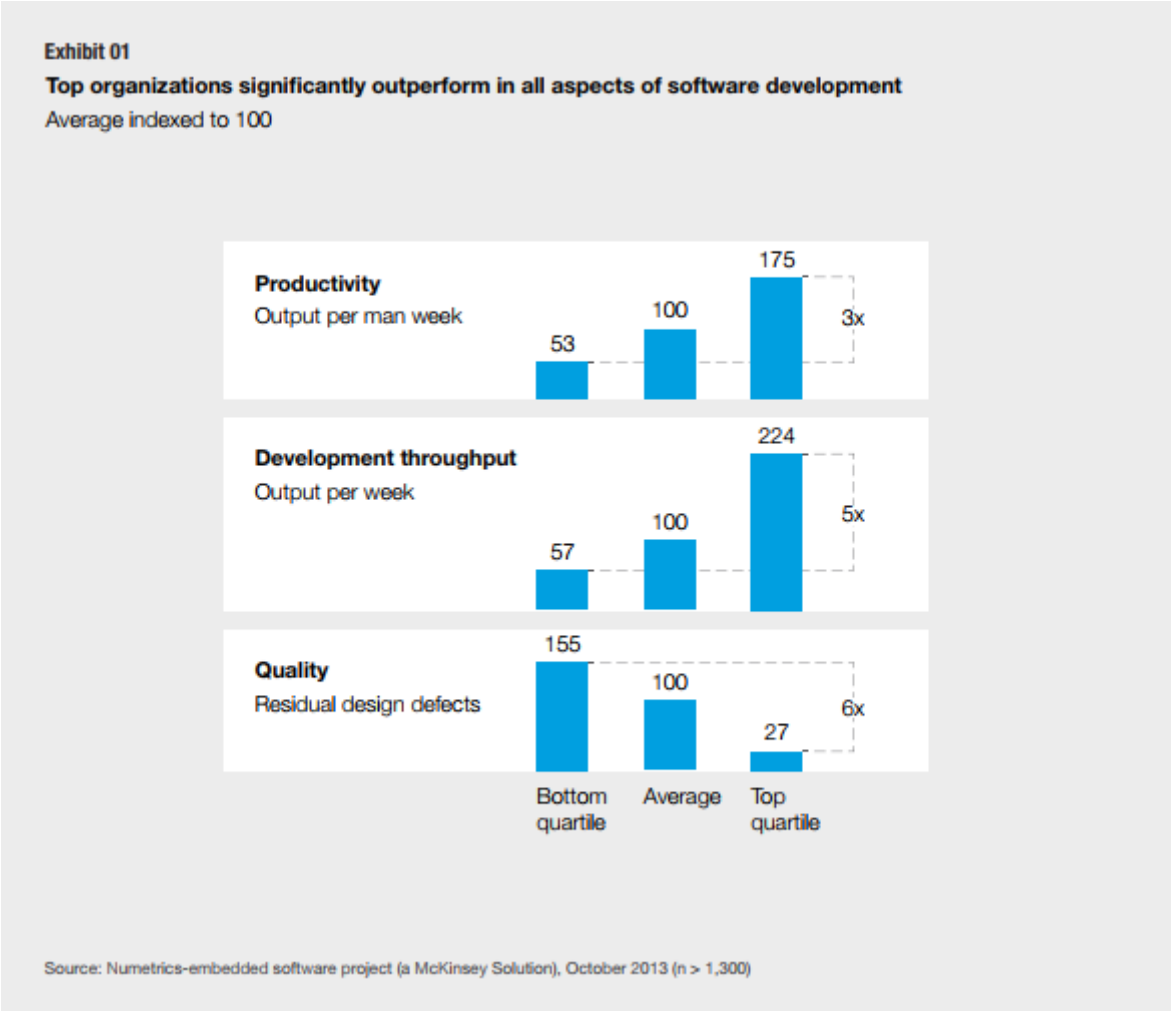
Laadun tarkoituksesta vallitsee siis yhtenäinen käsitys. Käyttäjien kannalta laatu merkitsee eniten tuotteen tai palvelun soveltuvuutta sen aiottuun käyttötarkoitukseen. Käyttäjät ovat tyytyväisiä tuotteeseen, kun sitä on helppo käyttää, siinä esiintyy vähän virheellistä toimintaa ja se on luotettava. Palvelun laatuun ollaan tyytyväisiä, kun palvelu on helposti saatavilla, käyttäjäystävällinen ja asiantunteva. Ohjelmisto on monessa tapauksessa esimerkiksi laitteeseen sulautettu ohjelmisto tai verkkopalvelu, jolloin käyttäjä- ja kehittäjäystävällisyys ovat tärkeitä asioita ohjelmiston laadun kannalta. Tässä opinnäytetyössä keskitytään ohjelmiston laatuun ja sen tekijöihin ennen kaikkea ohjelmistokehittäjän näkökulmasta.

McKinseyn tekemässä tutkimuksessa tutkittiin 1300 erikokoista yritystä ympäri maailmaa kysymällä kolme kysymystä. Alla suora lainaus kysymyksistä tarkennuksineen:

- *What software is being developed?*
    - *Companies need to assess how they prioritize different feature requirements and how they then scope the work and manage requirements (including late requirements). They should also assess how they set up the software architecture and system design to drive efficiency, for example, maximizing code reuse and ensuring a modular software architecture with clear interfaces and fewer interdependencies.*
- *How is the software developed?*
    - *Process is the name of the game here. At the outset, project planning and efficient resource management need to be evaluated. Companies must also assess their current software methodology and process to see if there are new and better processes available to increase productivity, time to market, and quality – for example, by moving away from traditional waterfall methods to agile software teams and development teams with integrated operations expertise (DevOps).*
- *Where software is being developed?*
    - *The actual location of the development is also important. Companies must look specifically at their decisions to outsource versus develop internally and the inner workings of the in-house parts of the organization focused on software development. One additional area to look into is how many sites are currently working with the same code base. McKinsey research shows that every site added to a software project results in a productivity loss of 15 percent.*

      Climbing out of software development mediocrity requires careful analysis. The path toward software development improvement is a highly tailored endeavour, as no two organizations require the exact same approach. McKinsey has developed a five-week diagnostic that helps

organizations understand their current performance. Diagnosis of a company's software development function typically comprises three phases: benchmarking output performance, assessing root causes, and identifying key improvement initiatives.



Kuva 1. McKinseyn tutkimuksen tulokset yritysten suorituskyvystä ohjelmistokehityksessä. (Strålin, Gnanasambandam, Andén, Comella-Dorda ja Burkacky 2016, 16).

Tutkimuksen tulosten mukaan yritykset, jotka panostavat ohjelmiston laatuun eniten, kehittävät ohjelmistoja jopa kolme kertaa tuottavammin ja niiden ohjelmistoissa on 80% vähemmän suunnitteluvirheitä, kuin vähiten laatuun panostavat yritykset. Lisäksi parhailla yrityksillä kuluu jopa 70% vähemmän aikaa tuotteen tai uuden ominaisuuden saattamiseen markkinoille. (Strålin, Gnanasambandam, Andén, Comella-Dorda ja Burkacky 2016, 7-17).

2.1 Laatutekijät, laatukriteerit sekä laatumetriikat

Kshirasagarin ja Priyadarshin (2008, 6,523-527) mukaan ohjelmiston laatu voidaan käsittää *laatukriteereinä* sekä *laatutekijöinä ja/tai laatuominaisuuksina*. Laatutekijä (quality factor) tai laatuominaisuus (Quality Characteristic) kuvastaa järjestelmän ulkoista ominaisuutta, ja laatutekijöitä ovat esimerkiksi oikeellisuus (correctness), luotettavuus (reliability), tehokkuus (efficiency), testattavuus (testability), ylläpidettävyys/hallittavuus (maintainability) sekä uudelleenkäytettävyys (reusability).

Laatukriteeri (Quality Criteria) on ohjelmistokehitykseen liittyvä laatutekijän attribuutti. Esimerkiksi modulaarisuus on yksi järjestelmäarkkitehtuurin ominaisuus eli attribuutti. Modulaarisessa järjestelmässä kehittäjät voivat sijoittaa yhtenäiset komponentit yhteen moduuliin, jolloin järjestelmän hallittavuus, joka on yksi laatutekijä, paranee. Samoin käyttäjävaatimusten jäljitettävyys antaa kehittäjille mahdollisuuden kartoittaa vaatimus tiettyyn ohjelmistomoduuliin, jolloin järjestelmän oikeellisuus paranee. (Kshirasagari ja Priyadarshi 2008, 6:523-527).

Kshirasagarin ja Priyadarshin (2008, 523-529) mukaan yksi vanhimmista laatutekijöiden ja -kriteerien määritelmistä on 1970-luvulla koottu McCallin, Richardsin ja Waltersin 1970-luvulla kokoelma ”McCall’s Quality Factors and Criteria”. Alla suora lainaus McCallin laatutekijöistä:

- ***Correctness:*** *Extent to which a program satisfies its specifications and fulfils the user’s mission objectives.*
- ***Reliability:*** *Extent to which a program can be expected to perform its intended function with required precision.*
- ***Efficiency:*** *Amount of computing resources and code required by a program to perform a function.*
- ***Integrity:*** *Extent to which access to software or data by unauthorized persons can be controlled.*
- ***Usability:*** *Effort required to learn, operate, prepare input, and interpret output of a program.*
- ***Maintainability:*** *Effort required to locate and fix a defect in an operational program.*
- ***Testability:*** *Effort required to test a program to ensure that it performs its intended functions.*
- ***Flexibility:*** *Effort required to modify an operational program.*
- ***Portability:*** *Effort required to transfer a program from one hardware and/or software environment to another.*
- ***Reusability:*** *Extent to which parts of a software can be reused in another applications.*
- ***Interoperability:*** *Effort required to couple one system with another.*

Alla suora lainaus McCallin laatukriteereistä:

- ***Access audit:*** *Ease with which software and data can be checked for compliance with standards or other requirements.*
- ***Access control:*** *Provisions for control and protection of the software and data.*
- ***Accuracy:*** *Precision of computations and output.*
- ***Communication commonality:*** *Degree to which standard protocols and interfaces are used.*
- ***Completeness:*** *Degree to which a full implementation of the required functionalities has been achieved.*
- ***Communicativeness:*** *Ease with which inputs and outputs can be assimilated.*
- ***Conciseness:*** *Compactness of the source code, in terms of lines of code.*
- ***Consistency:*** *Use of uniform design and implementation techniques and notation throughout the project.*
- ***Data commonality:*** *Use of standard data representations.*
- ***Error tolerance:*** *Degree to which continuity of operation is ensured under adverse conditions.*

- ***Execution efficiency:*** *Run time efficiency of the software.*
- ***Expandability:*** *Degree to which storage requirements or software functions can be expanded.*
- ***Generality:*** *Breadth of the potential application of software components.*
- ***Hardware independence:*** *Degree to which software is dependent on the underlying hardware.*
- ***Instrumentation:*** *Degree to which the software provides for measurement of its use or identification of errors.*
- ***Modularity:*** *Provision of highly independent modules.*
- ***Operability:*** *Ease of operation of the software.*
- ***Self-documentation:*** *Provision of in-line documentation that explains implementation of components.*
- ***Simplicity:*** *Ease with which the software can be understood.*
- ***Software system independence:*** *Degree to which the software is independent of its software environment – nonstandard language constructs, operating system, libraries, database management systems etc.*
- ***Software efficiency:*** *Run time storage requirements of the software.*
- ***Traceability:*** *Ability to link software components to requirements.*
- ***Training:*** *Ease with which new users can use the system.*

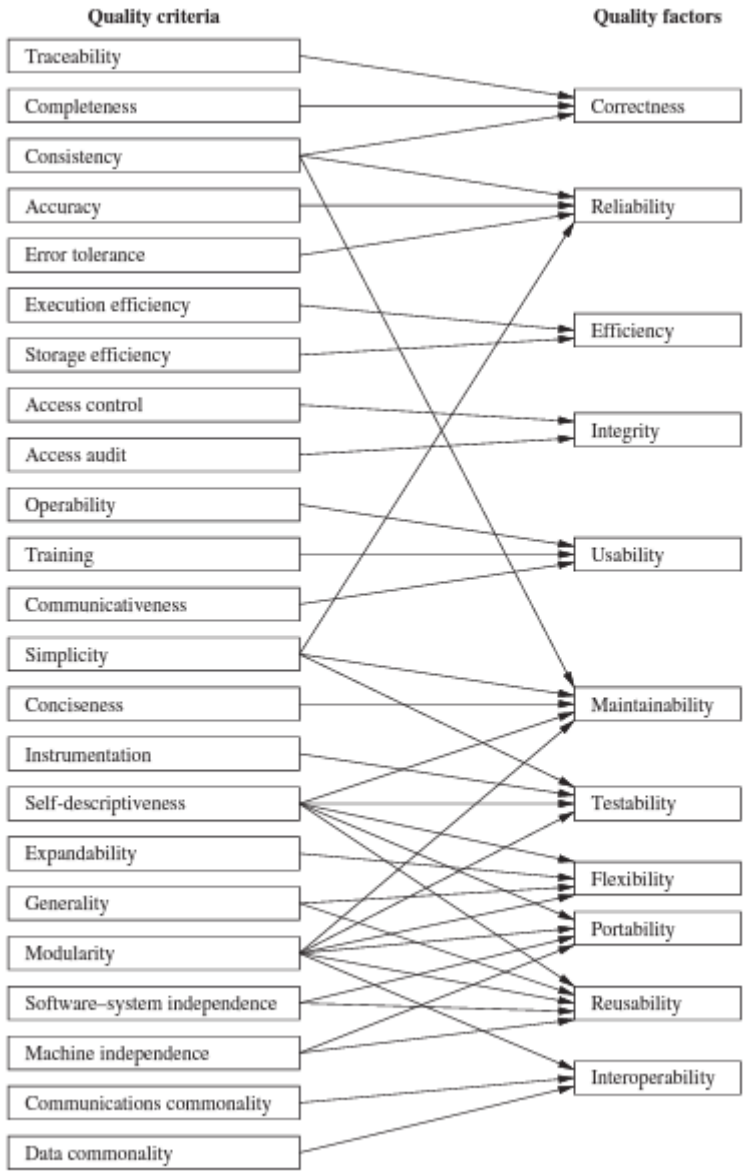


Figure 17.1 Relation between quality factors and quality criteria [6].

Kuva 2. Yhteys McCallin laatukriteerien ja laatutekijöiden välillä. (Kshirasagari ja Priyadarshi 2008, 528)

Erilaiset viiteryhmät ovat kiinnostuneita erilaisista laatutekijöistä. Esimerkiksi asiakkaat voivat haluta tehokkaita ja luotettavia ohjelmistoja, eivätkä he kiinnitä niinkään huomiota esimerkiksi ohjelmiston siirrettävyyteen. Kehittäjät pyrkivät täyttämään asiakkaiden tarpeet tekemällä järjestelmästä tehokkaan ja luotettavan tehden samalla järjestelmän mahdollisimman siirrettäväksi, jolloin ohjelmistokehityksen kustannukset vähenevät. Laadunvarmistuksen parissa työskentelevät kiinnittävät eniten huomiota ohjelmiston testattavuuteen, jolloin muut laatutekijät on helpompi varmentaa testien avulla. (Kshirasagari ja Priyadarshi 2008, 523).

Laatutekijöiden tasoja, kuten esimerkiksi ohjelmiston testattavuutta ei voida suoraan mitata eikä ilmaista yksinkertaisessa muodossa. Testattavuuden taso voidaan määritellä laatumetriikoilla, joita esimerkiksi testattavuuden tapauksessa ovat yksinkertaisuus (simplicity), instrumentointi (instrumentation), itsekuvailevuus (self-descriptiveness) sekä modulaarisuus (modularity). (Kshirasagari ja Priyadarshi 2008, 530).

Kshirasagari ja Priyadarshi (2008, 530) mukaan *laatumetriikka* on mitta, joka mittaa jotain laatukriteerin aspektia. Laatumetriikoita voidaan johtaa seuraavilla tavoilla:

- Koosta laatukriteeriin liittyviä relevantteja kysymyksiä ja etsi ”kyllä/ei” vastaus jokaiseen kysymykseen.
- Jaa ”kyllä”-vastausten määrä kysymysten määrällä, jolloin tulos on arvo 0 ja 1 väliltä. Tulos edustaa kyseistä laatumetriikkaa.

Esimerkiksi itsekuvailevaisuuteen liittyen voidaan kysyä, onko dokumentaatio kirjoitettu selkeästi ja yksinkertaisesti siten, että proseduurit, funktiot ja algoritmit voidaan helposti ymmärtää. Toinen mahdollinen kysymys on, että onko moduulin suunnitteluperustelut selkeästi ymmärretty. Kysymyksillä voi olla erilainen painotus laatumetriikan laskennassa, ja yksittäiset ”kyllä”-vastaukset voidaan painottaa eri tavoin yllämainitussa laatumetriikan laskennassa. Tämä johtaa siihen, että laatumetriikat ovat hyvin subjektiivisia käsitteitä.

2.2 Laatumallit

Kshirasagarin ja Priyadarshin (2008, 6, 530-533) mukaan monenlaisia laatumalleja on ehdotettu ohjelmiston laadun ja siihen liittyvien attribuuttien määrittelyyn. Yksi merkittävä malli on ISO 9126-standardin laatumalli, joka määrittelee kuusi kategoriaa ohjelmiston laatuominaisuuksille (quality characteristic). Nämä kuusi laatumäärettä voidaan jakaa useampiin alakategorioihin. Alla on esitetty ISO 9126-standardin pää- ja alakategoriat suorana lainauksena:

- *Functionality*
  - *Suitability*
  - *Accuracy*
  - *Interoperability*
  - *Security*
- *Reliability*
  - *Maturity*
  - *Fault tolerance*
  - *Recoverability*
- *Usability*
  - *Understandability*
  - *Learnability*
  - *Operability*
- *Maintainability*
  - *Analyzability*
  - *Changeability*
  - *Stability*
  - *Testability*
- *Portability*
  - *Adaptability*
  - *Installability*
  - *Conformance*
  - *Replaceability*

ISO 9126-standardin laatumalli muistuttaa huomattavasti McCallin laatumallia, sillä kumpikin malli käsittelee samaa abstraktia ohjelmiston laadun käsitettä. McCallin mallin laatutekijät (quality factor) ovat laatuominaisuuksia (quality characteristic) IS0 9126-mallissa. (Kshirasagari ja Priyadarshi, 2008, 533).

ISO 9126-standardin sekä McCallin laatumalleihin liittyy muutamia huolenaiheita. Ensimmäinen huolenaihe on, että laatumallien laatutekijöiden (quality factors) ja laatuominaisuuksien (quality characterisctics) tärkeydestä ei ole yhteisymmärrystä. Esimerkiksi McCallin mallissa on 11 laatutekijää, ja ISO 9126-mallissa on 6 laatuominaisuutta. Jotkin McCallin laatumallin tekijät, kuten uudelleenkäytettävyys (reusability) ja yhteen toimivuus (interoperability) ovat tärkeitä ohjelmistokehittäjille, kun taas ISO 9126-standardi keskittyy pelkästään itse tuotteen laatuominaisuuksiin. (Kshirasagari ja Priyadarshi, 2008, 534).

Kshirasagarin ja Priyadarshin (2008, 6, 549) mukaan toinen merkittävä malli on CMM eli *Capability Maturity Model*, jonka on kehittänyt *Software Engineering Institute* eli SEI. CMM-mallissa mitataan organisaatioiden kehitysprosessien kypsyyttä. Esimerkki epäkypsästä kehitysprosessista on, että ongelmanratkaisuun ja muihin kehitystyön aktiviteetteihin ei ole määriteltyjä prosesseja, tai niitä ei noudateta. Kehittäjät ja johtajat reagoivat ongelmiin, kun niitä ilmenee. Kehitystyön kustannusarviot, aikataulut ja ohjelmiston laatu ovat hyvin epätarkkoja. Kypsässä kehitysprosessissa kaikki aktiviteetit tehdään suunnitellusti, prosessien ja tuotteiden ominaisuuksista ja epäkohdista pidetään kirjaa, työntekijöitä koulutetaan sekä uusia teknologioita ja työkaluja sovelletaan ajan säästämiseksi sekä kehityskulujen alentamiseksi.

CMM-mallissa organisaatioiden kehitysprosessit arvioidaan asteikolla 1-5, ja asteikon portaat ovat Jayaramin (2007) mukaan seuraavat:

- Taso 1 eli *Aloitustaso*. Prosessit ovat järjestämättömiä tai jopa kaoottisia. Tällä tasolla menestys riippuu todennäköisesti yksilöiden vaivannäöistä, eikä se ole toistettavissa määritelmien ja dokumentointien puuttuessa.
- Taso 2 eli *Toistettavissa oleva*. Perustavanlaatuiset projektinhallintatekniikat ovat käytössä ja menestykset ovat toistettavissa, koska prosessin osat on määritelty ja dokumentoitu.
- Taso 3 eli *Määritelty*. Tällä tasolla organisaatio on kehittänyt standardoidun ohjelmistoprosessin kiinnittämällä huomiota dokumentointiin, standardointiin sekä integrointiin.
- Taso 4 eli *Hallittu*. Organisaatio valvoo ja hallitsee prosesseja tiedonkeruun ja analyysien avulla.
- Taso 5 eli *Optimointi*. Tällä tasolla prosesseja kehitetään jatkuvasti prosesseista tulevan palautteen perusteella sekä kehittämällä uusia innovatiivisia prosesseja, jotka palvelevat organisaation tarpeita paremmin.

Kypsyysmallien, kuten CMM-mallin käyttö prosessien arviointiin perustuu ideaan, että ilman toistettavissa olevaa prosessia ainoa todennäköisesti toistettavissa oleva asia on virheet. Ohjelmistotuotannossa prosessit koostuvat erilaisista aktiviteeteistä, joilla ohjelmistoja kehitetään, ja nämä aktiviteetit riippuvat vahvasti tiedosta, kuten dokumenteista, standardeista ja menettelytavoista. On siis selvää, että prosessit toimivat huonosti, mikäli tietoa oikeista menettelytavoista ym. ei ole saatavilla. (Kshirasagari ja Priyadarshi, 2008, 546).

2.3 Laadun neljä ulottuvuutta

Chemuturin (2011, 25) näkemyksen mukaan ohjelmistojen laatu voidaan jakaa neljään ulottuvuuteen: spesifikaatioiden laatu (specification quality), suunnittelun laatu (design quality), kehitystyön laatu (development quality) ja vaatimusten mukaisuuden laatu (conformance quality).

Spesifikaatioiden laadulla tarkoitetaan sitä, kuinka kehitettävän ohjelmiston spesifikaatiot on määritelty. Ohjelmiston spesifikaatiot ovat ensimmäinen asia, joka tehdään uutta tuotetta tai palvelua suunniteltaessa, ja jos spesifikaatiot ovat huonot, on myös lopputulos huono. Epämääräiset spesifikaatiot johtavat siihen, että kehitystyö ja laadunvarmistustyö valuvat hukkaan, joten spesifikaatioiden tarkkuus ja ymmärrettävyys ovat kaikista tärkeimmässä asemassa ohjelmiston laadun varmistamisessa. (Chemuturi 2011, 26-27).

Suunnittelun laadulla tarkoitetaan sitä, kuinka hyvin ohjelmisto on suunniteltu. Suunnittelun tavoitteena on täyttää annetut spesifikaatiot, ja mikäli suunnittelu tehdään huonosti, tuote tai palvelu epäonnistuu riippumatta spesifikaatioiden laadusta. Suunnittelu koostuu kahdesta osiosta, abstraktista käsitteellisestä suunnittelusta sekä käytännön suunnittelusta. Ohjelmistokehityksessä käsitteellisessä suunnittelussa määritellään ohjelmiston arkkitehtuuri, navigointi, ja käytännön suunnittelussa suunnitellaan tietokantojen rakenteet, näkymät, raportointi ja muut yksityiskohdat. (Chemuturi 2011, 28).

Kehitystyön laadulla tarkoitetaan sitä, kuinka ohjelmisto on toteutettu. Kehityksen laadun varmistamiseksi käytetään testausta, tarkastuksia, ohjelmointikäytäntöjä sekä noudattamalla suunnitteludokumentin määrityksiä. (Chemuturi 2011, 29-30).

Vaatimusten mukaisuuden laatu tarkoittaa sitä, kuinka hyvin ohjelmiston laatu saavutetaan noudattamalla muita laadun ulottuvuuksia. Vaatimusten mukaisuuden laatu voidaan todeta käyttämällä laatumetriikoita, kuten laadunvarmistusaktiviteettien tehokkuutta virheiden vähentämisessä sekä virheiden esiintymistiheyttä. (Chemuturi 2011, 30-31).

2.4 Laadunvarmistus

Chemuturin (2011, 17) mukaan organisaation laadunvarmistusosaston (Quality Assurance Department) funktio on taloushallinnon tilintarkastustoiminnan kaltainen. Useimmat valtiot ovat asettaneet yritysten tilintarkastukset pakolliseksi toiminnaksi, jolla varmistetaan, että yritysten kirjanpito on kunnossa eikä esimerkiksi yrityksen johto käytä yrityksen varoja väärin. Tilintarkastuksen ollessa pakollista laadunvarmistusosaston ulkopuolinen tarkastaminen ei ole. Muun muassa tämän takia useimmissa yrityksissä ei siten ole kunnollista laadunvarmistusosastoa muutoin kuin nimellisesti, ja useimpien yrityksien tapauksessa ei ole siis edellytyksiä luoda minkäänlaisia laatumetriikoita tai -raportteja.

Chemuturin (2011, 18-19) mukaan organisaatiot saavat monesti laatusertifikaatteja ISO:lta ja ylistyksiä laadusta CMM-prosessissa. Kumpikaan prosessi ei kuitenkaan vaadi, että yrityksessä on olemassa laadunvarmistusosasto ja voidaan miettiä, mikä tarkoitus tilintarkastuksella on, jos yritys voi samaan aikaan tuottaa huo-

nolaatuisia tuotteita tai palveluita ja olla menossa kohti konkurssia laadunvarmistusosaston puuttuessa. Yrityksissä, joissa laadunvarmistusosasto on olemassa, on näiden pääasiallisena tehtävänä monesti toimia sertifiointiorganisaatioiden kanssa, jotta saavutetut laatusertifikaatit säilyvät voimassa tai että laatusertifikaatti voitaisiin saada tulevaisuudessa, ja laadunvalvonta on yleensä toissijainen funktio. Tämä ei kuitenkaan tarkoita, että ohjelmistoyrityksissä ei valvottaisi laatua ollenkaan. Monesti laadunvalvonta on vain jätetty kehittäjien varaan ohjelmistotestauksen muodossa.

Chemuturin (2011, 22) mukaan kokoaikaisen ja ammattitaitoisen laadunvarmistusosaston olemassaololle on olemassa monia perusteita:

- Laadullinen näkökulma asioihin olisi aina saatavilla riippumatta muista asioista.
- Laadunvarmistusaktiviteetit toteutettaisiin jatkuvasti ja ilman poikkeuksia.
- Laadunvarmistusosasto voi jatkuvalla valvonnalla estää laadun heikkenemisen ennen vahingon aiheutumista. Lisäksi se voi ajaa organisaatiota tavoittelemaan korkeampaa laatua ja erinomaisuutta aloitteiden ja korjausten muodossa.
- Laadunvarmistusosasto voi mitata ja analysoida prosessien suorituskykyä, jolla valvotaan organisaation tavoitteiden saavuttamista, kerätään dataa sekä tehdään tarvittavia parannuksia, jotta prosessit toimisivat niin kuin suunniteltu. Tästä seuraa, että organisaation laadullisia saavutuksia voidaan verrata muihin organisaatioihin kerätyn datan ja analyysien avulla.
- Organisaatiolla olisi käytössään asiantuntija-apua kaikissa laatuun liittyvissä asioissa.

3 OHJELMOINTIKÄYTÄNNÖT

3.1 Mitä ovat ohjelmointikäytännöt?

Wikipedian (2018) sekä Chemuturin (2011, 30) mukaan ohjelmointikäytännöllä tarkoitetaan kokoelmaa ohjelmointikieleen liittyviä ohjeistuksia, joissa määritellään yleensä ohjelmointityyli, käytännöt, kooditiedostojen organisointi, sisennykset, kommentit, välilyönnit, nimeämiskäytännöt, ohjelman arkkitehtuurikäytännöt sekä muut mahdolliset kieleen liittyvät ohjelmointisäännöt. Jokaiselle organisaatiossa käytettävälle kielelle on yleensä oma ohjelmointikäytäntö. Nämä ohjeistukset tulisi määritellä ennen kuin kieltä aletaan käyttää ohjelmistotuotannossa.

3.2 Ohjelmointikäytäntöjen merkitys

Microsoftin (2015) C#-ohjelmointikäytännön tavoitteina on luoda koodille yhtenäinen ilme ja ohjeistaa käytäntöjä, joiden avulla kehittäjät voivat lukea, ymmärtää ja ylläpitää koodia helposti.

Googlen (2018) mukaan C++ Style Guiden tarkoituksena on tarjota paras mahdollinen ohjeistus minimaalisilla rajoitteilla. Säännöt perustuvat olemassa oleviin Googlen C++ yhteisön käyttämiin sääntöihin. C++ Style Guide listaa seuraavanlaiset tyylioppaan tavoitteet:

- Tyylisääntöjen tulisi ”kantaa painonsa”.
    - Tyylisääntöjen hyöty täytyy olla riittävän suuri, jotta on oikeutettua pyytää suurta kehittäjäjoukkoa pitäytymään säännöissä.
- Koodi optimoidaan lukijaa, ei kirjoittajaa varten.
    - Koodi täytyy olla ensisijaisesti optimoitu lukijaa varten. Koodipohjan ollessa suuri ja sen elinkaaren ollessa pitkä tullaan koodia lukemaan enemmän kuin kirjoittamaan. Tästä syystä tyylisäännöt on kirjoitettu helpottamaan kehittäjien työtä, kuten koodin lukemista, ylläpitoa sekä virheenkorjausta.
- Säilytetään yhtenäisyys olemassa olevan koodin kanssa.
    - Käyttämällä yhtä tyyliä koodi säilyy yhtenäisenä (consistent), jolloin kehittäjät voivat keskittyä tärkeämpien asioiden hoitamiseen. Lisäksi esimerkiksi automaattiset työkalut eivät toimi oikein, mikäli tyyli vaihtelee.
- Säilytetään yhtenäisyys muiden C++-tyylien kanssa silloin kuin se on soveliasta.
    - Yhtenäisyys muiden organisaatioiden käyttämään tyyliin on arvokasta samoista syistä kuin yhtenäisyys oman koodin kanssa. Jos joku yleisesti hyväksytty ja käytössä oleva tyyli tai C++-standardiominaisuus ratkaisee ongelman, voidaan kyseistä tyyliä tai tapaa käyttää perustellusti.
- Vältetään yllättäviä tai vaarallisia rakenteita.
    - C++-kielessä on ominaisuuksia ja rakenteita, jotka ovat vaaralliseimpia ja yllättävämpiä kuin ensisilmäyksellä vaikuttaa. Tyylioppaassa rajoitetaan kyseisten ominaisuuksien ja rakenteiden käyttöä, jotta mahdollisilta ongelmilta vältytään.

- Vältetään sellaisia C++-kielen rakenteita ja ominaisuuksia, joita keskiverron kehittäjän on hankala ylläpitää.
    - C++-kielessä on mahdollista tehdä monimutkaisia rakenteita, ja se sisältää ominaisuuksia, jotka lisäävät koodin monimutkaisuutta huomattavasti. Koodin ylläpito uusien kehittäjien toimesta hankaloituu, mikäli he eivät ymmärrä käytettyjä rakenteita.
- Huomioidaan koodin laajuus.
    - Googlella on yli 100 miljoonaa riviä C++-koodia ja tuhansia kehittäjiä, joten yksittäisen kehittäjän tekemät virheet voivat kostautua monille muille. On tärkeää, että esimerkiksi globaalia nimiavaruutta (*namespace*) ei ”saastuteta”.
- Sallitaan optimointi, kun tarpeellista.
    - Koodin suorituskykyoptimointi voi olla joskus tarpeellista ja perusteltua, vaikka se olisi ristiriidassa muiden tyylisääntöjen kanssa.

The GNOME Projectin näkemys hyvän koodin kirjoittamisen tarpeesta on pitkälti samanlainen kuin Microsoftilla ja Googlella. The GNOME Projectin ohjeistuksessa sanotaan suoraan lainaten seuraavasti:

” GNOME is a very ambitious free software project, and it is composed of many software packages that are more or less independent of each other. A lot of the work in GNOME is done by volunteers: although there are many people working on GNOME full-time or part-time for here, volunteers still make up a large percentage of our contributors. Programmers may come and go at any time and they will be able to dedicate different amounts of time to the GNOME project. People’s “real world” responsibilities may change, and this will be reflected in the amount of time that they can devote to GNOME.

Software development takes long amounts of time and painstaking effort. This is why most part-time volunteers cannot start big projects by themselves; it is much easier and more rewarding to contribute to existing projects, as this yields results that are immediately visible and usable.

Thus, we conclude that it is very important for existing projects to make it as easy as possible for people to contribute to them. One way of doing this is by making sure that programs are easy to read, understand, modify, and maintain.

Messy code is hard to read, and people may lose interest if they cannot decipher what the code tries to do. Also, it is important that programmers be able to understand the code quickly so that they can start contributing with bug fixes and enhancements in a short amount of time. Source code is a form of communication, and it is more for people than for computers. Just as someone would not like to read a novel with spelling errors, bad grammar, and sloppy punctuation, programmers should strive to write good code that is easy to understand and modify by others.”

The GNOME Projectin näkemys hyvän koodin ominaisuuksista on seuraavanlainen:

- Siisteys. Ihmisten on helppo lukea ja ymmärtää siistiä koodia. Siisteys käsittää koodin tapauksessa ohjelmointityylin (sulkujen asettelu, sisennykset, muuttujien nimet) sekä koodin kontrollivirran.

- Yhtenäisyys. Yhteneväinen koodi helpottaa ohjelman toiminnan ymmärtämistä. Luettaessa yhdenmukaista koodia siitä on helppo tehdä olettamuksia ja odotuksia, kuinka koodi toimii, joten siihen on helpompaa ja turvallisempaa tehdä muokkauksia.
- Laajennettavuus. Yleiskäyttöistä koodia on helpompi muokata ja käyttää uudelleen, kuin koodia, jossa on paljon kiinteästi asetettuja olettamuksia. Alun perin yleiskäyttöiseksi suunniteltuun koodiin on ilmeisten selvästi helpompaa lisätä uusia ominaisuuksia, kuin sellaiseen koodiin, joka on kirjoitettu kiinteästi asetettujen oletuksien perusteella vain yhtä tarkoitusta varten.
- Virheettömyys. Virheettömäksi suunniteltu koodi antaa ihmisille mahdollisuuden keskittyä uusien ominaisuuksien kehittämiseen virheistä huolehtimisen sijaan. Virheettömäksi ja turvalliseksi suunniteltu koodi estää myös triviaaleja ja hölmöjä virheitä. Myös loppukäyttäjät arvostavat virheetöntä koodia, sillä kukaan ei pidä ohjelmistosta, joka ei toimi.

Scott Dormanin mukaan ohjelmointikäytännön omaksumiseen liittyy lisäksi psykologinen puoli, eli niin sanottu koodin omistamisen tunne (code ownership). Koodin omistamisen tunteella tarkoitetaan ylpeyden tunnetta laadukkaan työn tekemisestä, sekä halua nähdä koodin toimivan hyvin. Tunteen kasvaessa kehittäjien itsevarmuus paranee, ja he tuntevat enemmän itsevarmuutta, omista kyvyistään ja taidoistaan, joka edelleen vahvistaa koodin omistamisen tunnetta. Tästä syntyy positiivisesti itseään ruokkiva kehä.

SIGin keräämän aineiston mukaan virheiden korjaaminen ja ohjelmiston parantaminen on kaksi kertaa nopeampaa järjestelmissä, joissa ohjelmiston hallittavuus on keskivertoa parempi, kuin järjestelmissä, joissa se on keskivertoa huonompi. Ohjelmistojen käyttöikä on monesti 10 vuotta tai enemmän, joten hallittavuus on yksi kriittisimmistä tekijöistä ohjelmiston laadun kannalta. (Visser, Rigal, Van Der Leek, Van Eck ja Wijnholds 2016, 3-4).

Ohjelmointikäytäntö on tietenkin vain yksi laatutekijä ohjelmistokehityksessä, mutta kuten yllä todettu, sen merkitys koodin laatuun on äärimmäisen suuri. Laatuosiossa jo käsitellyn McKinseyn tekemän tutkimuksen mukaan yritykset, jotka panostavat ohjelmiston laatuun eniten, kehittävät ohjelmistoja jopa kolme kertaa tuottavammin ja niiden ohjelmistoissa on 80% vähemmän suunnitteluvirheitä, kuin vähiten laatuun panostavat yritykset. Siksi ohjelmointikäytännön luomiseen ja noudattamiseen kannattaa panostaa aikaa, vaivaa ja resursseja.

3.3 Esimerkkejä ohjelmointikäytännöistä

Esimerkkeinä yleisesti käytössä olevista ohjelmointikäytännöistä esitän Microsoftin C# ohjelmointikäytännön sekä Googlen C++ Style Guiden. Koska nämä käytännöt ovat varsin laajoja, en käsittele niitä tässä vaan ne on lisätty liitteinä tämän opinnäytetyön loppuun.

Koin tarpeelliseksi lisätä nämä käytännöt mukaan tähän opinnäytetyöhön, jotta ne olisivat helposti saatavilla, ja opinnäytetyön lukija voi helposti tutkia niitä.

3.4 Esimerkkejä koodin luettavuudesta

Kuvissa 3 ja 4 on esitetty esimerkkinä C-kielinen ohjelma, joka laskee kertoman luvuille 1-10, ja tulostaa jokaisen luvun kertoman näytölle. Kääntäjän näkökulmasta kuvan 1 ja 2 koodit ovat identtisiä, ja se tuottaa samanlaisen ohjelman kummassakin tapauksessa.

Kuvassa 3 on esimerkki huonosti luettavasta C-kielisestä ohjelmasta. Kuvan 1 koodi on C-kielen syntaksin mukainen, ja se kääntyy ja toimii oikein. Kääntäjä ei välitä koodin ulkoasusta tai asettelusta ja sille riittää, että koodi on kielen syntaksin mukainen. Ihmisen on kuitenkin hankala lukea kuvan 1 mukaista koodia, koska sen asettelu on tiivis, sekä muuttujien ja funktioiden nimet eivät ole tarkoitustaan kuvaavia. Lisäksi koodia ei ole kommentoitu mitenkään.

```
#include <stdio.h>

int calc(int i);

int main( int argc, char* argv[] )
{
    int x,y;
    for(x=1;i<= 10;i++)
        y=calc(x);
        printf("Result %d",y);

    return 0;
}

int calc(int i)
{
    if(i<0)
        return 0;
    if(i<=1)
        return 1;
    else
        return i*calc(i-1);
}
```

Kuva 3. Esimerkki huonosti luettavasta koodista C-kielellä.

Kuvassa 4 on esimerkki hyvin luettavasta C-kielisestä ohjelmasta. Koodissa on käytetty rivivälejä, sekä muuttujien ja funktioiden nimet ovat tarkoitustaan kuvaavia. Vaikka kuvan 4 mukainen koodi vie enemmän tilaa tekstimuodossa, on silti suositeltavaa kirjoittaa koodi kuvan mukaiseen selkeään muotoon, koska selkeää koodia on huomattavasti helpompi lukea ja ymmärtää.

```c
#include <stdio.h>


/**
 *  \brief      Calculates factorial of the given number.
 *
 *  \param [in] num     number
 *
 *  \return     factorial of \param number
 *
 *  \details    None.
 */
int factorial( int num );


int main( int argc, char* argv[] )
{
    int i = 0;
    int fact = 0;

    for( i = 1; i <= 10; i++ )
    {
        // Calculate factorial for each number
        fact = factorial( i );

        // Print result
        printf( "\r\nFactorial of number %d is : %d", i, fact );
    }

    return 0;
}


int factorial( int num )
{
    if( num < 0 )
    {
        // Factorial is not defined for negative numbers
        return 0;
    }

    if( num <= 1 )
    {
        // Factorial of 1 or 0 is 1
        return 1;
    }
    else
    {
        // Call factorial function recursively
        return ( num * factorial( num - 1 ) );
    }
}
```

Kuva 4.Esimerkki hyvin luettavasta koodista C-kielellä.

4 DOKUMENTAATIO

4.1 Dokumentoinnin tarkoitus

Dokumentoinnin tarkoituksena on kuvata, mitä koodi tekee. Kunnollinen koodin dokumentaatio on tärkeää, jotta koodin luettavuus ja hallittavuus olisi mahdollisimman hyvä. Koodi ei välttämättä ole itsestään selvää esimerkiksi monimutkaisten algoritmien vuoksi, jolloin tiedon jakaminen muiden kehittäjien käyttöön dokumentoinnin välityksellä on tärkeää. Esimerkiksi tuotannossa esiintyvien ongelmien korjaaminen ja lisäominaisuuksien kehittäminen on huomattavasti helpompaa ja nopeampaa, mikäli ohjelmiston dokumentaatio on kunnollinen ja selkeä. Huonosto tehty dokumentaatio on hyödytön, ja sama kuin dokumentaatiota ei olisi ollenkaan. (Eastern Peak).

Hyvä dokumentaatio on yksinkertaista ja tiivistä, ja se selittää vain tarvittavat asiat, eikä jokaista koodiriviä tarvitse tai kannata dokumentoida. Dokumentaatio tulisi pitää aina ajantasaisena, ja kaikki koodin muutokset tulisi päivittää dokumentaatioon heti. Dokumentaatio kirjoitetaan tyypillisesti Englanniksi, jotta kuka tahansa voisi lukea sitä äidinkielestään riippumatta, ja dokumentaatiota kirjoitettaessa tulisi käyttää imperatiivisia preesensmuotoja ja aktiivista äänensävyä toisessa persoonassa. Dokumentoinnissa kannattaa hyödyntää automaattisia dokumentointityökaluja prosessin nopeuttamiseksi ja helpottamiseksi. (Eastern Peak).

Dokumentaatio kannattaa kirjoittaa yhtaikaa koodin kirjoittamisen kanssa, koska koodin rakenteen avaaminen sanoiksi, jolloin piilevät ajatus- ja logiikkavirheet tulevat helpommin esille. Lisäksi dokumentaation kirjoittaminen kehittää teknisen kirjoittamisen taitoja. (Write The Docs).

Alla on esitetty esimerkkejä dokumentaatiosta. Valitsin nämä kyseiset esimerkit siksi, että mielestäni ne edustavat hyvää ja selkeää dokumentointitapaa. Oikea dokumentointitapa on tietysti jollain lailla mielipidekysymys, eikä nämä esimerkit ole välttämättä jonkun muun mielestä hyviä.

4.2 Esimerkkejä dokumentaatiosta

4.2.1 Dokumentaatio GitHubissa

Kuvassa 5 on esitetty esimerkki projektin README.md-tiedoston sisällöstä. Kyseistä formaattia käytetään hyvin yleisesti esimerkiksi monissa avoimen lähdekoodin projekteissa GitHub-palvelussa. Kuvan mukainen esitysmuoto on selkeä esitys siitä, mikä kyseinen projekti on, mihin tarkoituksiin sitä voidaan käyttää, mitä ominaisuuksia siinä on, kuinka sitä käytetään, kuinka kehitystyöhön voi osallistua ja mistä voi saada tukea ongelmatilanteissa.

```
$project
========

$project will solve your problem of where to start with documentation,
by providing a basic explanation of how to do it easily.

Look how easy it is to use:

    import project
    # Get your stuff done
    project.do_stuff()

Features
--------

- Be awesome
- Make things faster

Installation
------------

Install $project by running:

    install project

Contribute
----------

- Issue Tracker: github.com/$project/$project/issues
- Source Code: github.com/$project/$project

Support
-------

If you are having issues, please let us know.
We have a mailing list located at: [email protected]

License
-------

The project is licensed under the BSD license.
```

Kuva 5. Esimerkki projektin README-tiedostosta (Write The Docs).

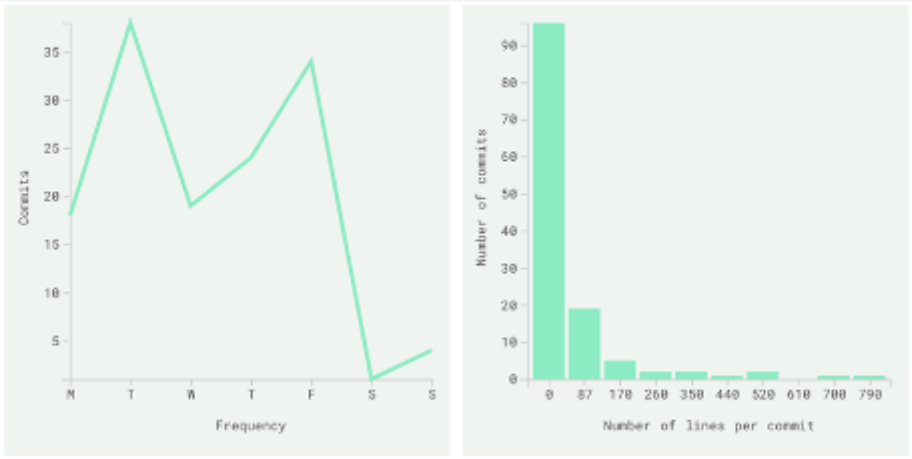
Kuvassa 6 on esitetty esimerkkinä osa Sourcerer-projektin README-tiedostosta. Kyseinen projekti on avoimen lähdekoodin projekti, jolla voidaan tehdä visuaalisia profilointeja esimerkiksi organisaatiossa käytetyistä ohjelmointikielistä. Kuten kuvasta nähdään, esitysmuoto on esimerkillisen selkeä ja jäsennelty, jolloin sen käyttöönotto on hyvin helppoa.

README.md

sourcerer.io

sourcerer start now release v0.3.1 license MIT

A visual profile for software engineers.



Features

- Profile creation with a single click
- Support of 100 languages (even exotic ones like COBOL)
- Detection of more than 1,000 libraries in code with per-line statistics
- Visual presentation your development experience
- *Finally!* Summary of all repositories you've contributed to
- Interesting facts about yourself

Creating your profile is just the first step for us at Sourcerer. Some of the things on our roadmap include:

- Engineers to follow and learn from
- Technology and libraries you should know about
- Projects that could use your help

Get started

The easiest way to get started is with your open source repos. Go to sourcerer.io/start, and select *Build with GitHub* and watch your profile build.

For closed source repos, you will need to use this app. If you already created an account using GitHub, you would have received an email with credentials for the app. If not, You will need a new account, which you can get at sourcerer.io/join.

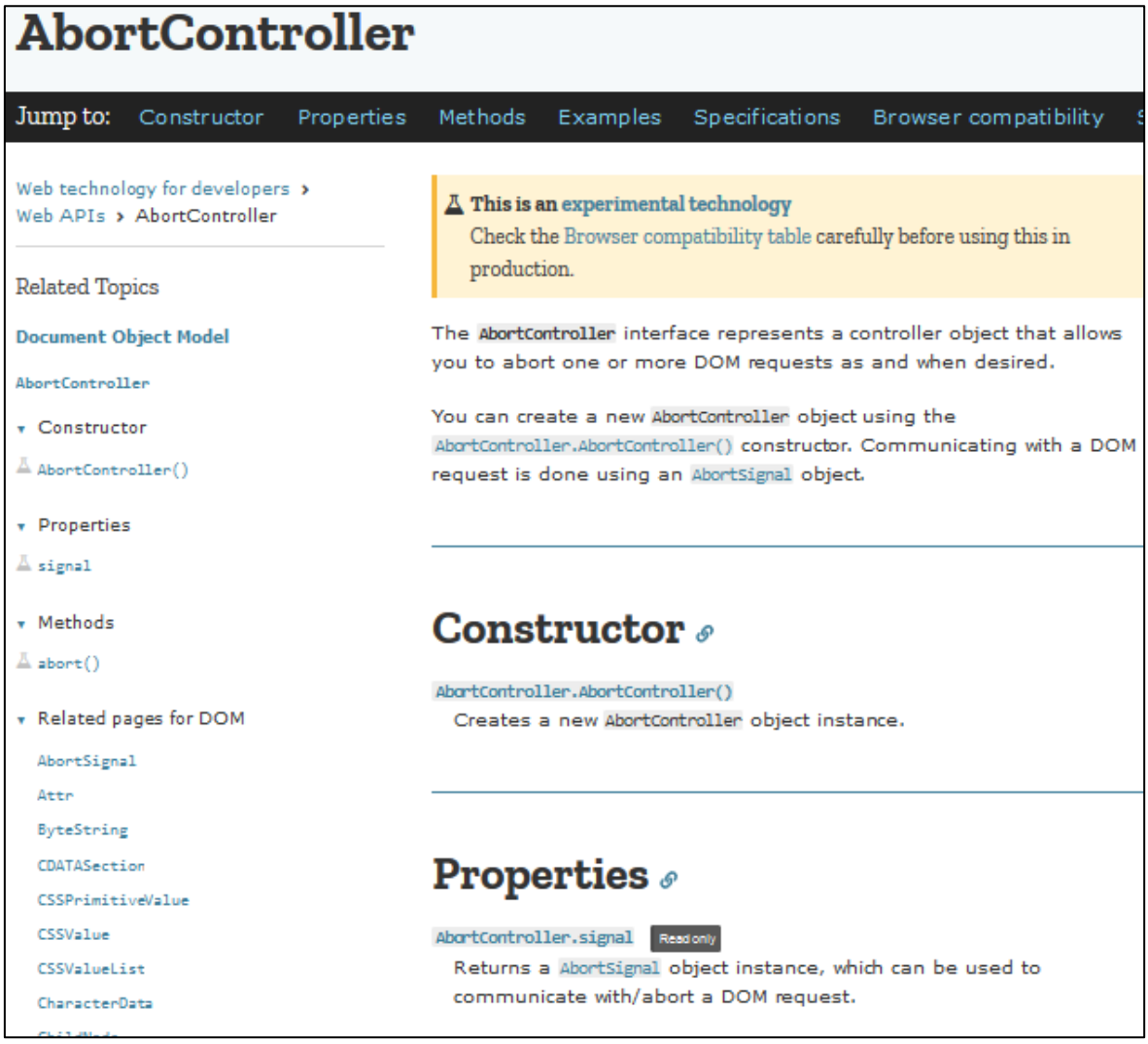
Showcase

Kuva 6. Esimerkkinä Sourcerer-projektin README-tiedostosta GitHubissa (Sourcerer).

4.2.2 Mozilla Firefoxin Web API-dokumentaatio

Kuvissa 7, 8 ja 9 on esimerkkejä Mozilla Firefox-selaimen Web API dokumentaatiosta. Kuvista nähdään, että rajapinnat, ominaisuudet sekä käyttötarkoitukset on selkeästi dokumentoitu ja dokumentaatio on hyvin jäsenneltyä koodiesimerkkeineen. Kyseinen dokumentaatio on HTML-muotoista, ja sitä voidaan selata millä tahansa verkkoselaimella.

Kuva 7. Mozilla Firefox-selaimen Web API dokumentaatio. (MDN Web Docs).



Kuva 8. Esimerkki Mozilla Firefoxin AbortController-rajapinnan dokumentaatiosta. (MDN Web Docs).

Syntax

```
var controller = new AbortController();
```

Parameters

None.

Examples

In the following snippet, we aim to download a video using the Fetch API.

We first create a controller using the `AbortController()` constructor, then grab a reference to its associated `AbortSignal` object using the `AbortController.signal` property.

When the fetch request is initiated, we pass in the `AbortSignal` as an option inside the request's options object (see `{signal}`, below). This associates the signal and controller with the fetch request and allows us to abort it by calling `AbortController.abort()`, as seen below in the second event listener.

```
var controller = new AbortController();
var signal = controller.signal;

var downloadBtn = document.querySelector('.download');
var abortBtn = document.querySelector('.abort');

downloadBtn.addEventListener('click', fetchVideo);

abortBtn.addEventListener('click', function() {
  controller.abort();
  console.log('Download aborted');
});
```

Kuva 9. Mozilla Firefoxin AbortController-rajapinnan konstruktorin käyttö. (MDN Web Docs).

4.3 Dokumentointityökalut

Dokumentointityökalut auttavat dokumentaation kirjoittamisessa automatisoimalla suuren osan dokumentaation kirjoitustyöstä. Koodin dokumentointityökalut ovat usein kielikohtaisia, mutta useaa kieltä tukevia työkaluja on myös saatavilla. (Eastern Peak).

Yleisiä dokumentointityökaluja (Eastern Peak):

- Doxygen (C, C++, C#, Java, Objective-C, PHP, Python)
- GhostDoc (C#, Visual Basic, C++, JavaScript)
- Javadoc (Java)
- Docurium, YARD (Ruby)
- jsdoc (JavaScript)

Käsittelen tässä opinnäytetyössä Doxygen-työkalua, koska toimeksiantajan tuotekehitysosasto käyttää sitä koodin dokumentointiin.

4.3.1 Doxygen

Doxygen on Dimitri van Heeschin kehittämä dokumentointityökalu, jolla voidaan luoda koodista dokumentaatio HTML- tai LaTeX-muodossa. Doxygen lukee erityismerkinnöillä varustettuja lähdekooditiedostoja, joista se luo dokumentaation. Doxygen tukee monia ohjelmointikieliä, kuten C, C++, Objective-C, C#, PHP, Java, Python, IDL, Fortran, VHDL, Tcl ja D. Doxygen on saatavissa Mac OS X-, Linux- ja Windows-käyttöjärjestelmille, ja siitä löytyy käännös myös useimmille Unix-käyttöjärjestelmille. Doxygen on yksi de-facto dokumentointityökaluista. (Van Heesch).

Van Heeschin mukaan Doxygeniä voidaan käyttää kolmella tavalla:

- Doxygenillä voi luoda HTML-muotoisen online-dokumentaation, ja/tai LaTeX-muotoisen offline-dokumentaation. Doxygen tukee myös RTF-, PostScript-, PDF-, kompressoitua HTML- ja Unixin manpage-muotoja. Dokumentaatio luetaan suoraan lähdekooditiedostoista, jolloin dokumentaation ylläpito on huomattavasti helpompaa, koska dokumentaatiomerkinnät voidaan päivittää suoraan samalla kuin koodia päivitetään.
- Doxygenillä voi myös luoda dokumentaation dokumentoimattomista lähdekooditiedostoista. Doxygen osaa myös visualisoida lähdekooditiedostojen välisiä riippuvaisuuksia ja luokkien perintäkaavioita, jolloin suurten kokonaisuuksien hahmottaminen helpottuu.
- Doxygenillä voi myös luoda muitakin dokumentaatioita, kuten esimerkiksi Doxygenin verkkosivut.

4.3.2 Doxygenin käyttö

Doxygenissä on mittava määrä ominaisuuksia, ja tässä osiossa keskitytään Doxygenin perusteisiin, ja kuinka sitä käytetään. Lisäksi koodin kommentointivaatimukset käydään lävitse.

Doxygen käyttää C ja C++-kielille C- ja C++-kielien mukaisia kommentteja, joihin on lisätty erityisiä merkintöjä, jolla Doxygen tunnistaa dokumentaatioon sisällytettävät koodin osat. Jokaiselle koodin entiteetille on kahden- tai kolmen tyyppisiä kuvauselementtejä, jotka muodostavat kyseisen entiteetin dokumentaation. Perusmallisia kuvauksia ovat lyhyt kuvaus, ja yksityiskohtainen kuvaus (brief and detailed descriptions). Funktioille ja luokkien metodeille on lisäksi kolmas kuvaustyyppi, joka on kooste kaikista koodilohkon sisältämistä kommenteista (in-body description). (Van Heesch).

Lyhyitä- ja yksityiskohtaisia kuvauksia voi olla useampia per entiteetti, mutta se ei ole suositeltavaa, sillä Doxygen ei voit taata, että niiden järjestys säilyy samana lopullisessa dokumentaatiossa. Lyhyet kuvaukset ovat sanansa mukaisesti lyhyitä yhden lauseen kommentteja, ja yksityiskohtaisilla kuvauksilla kuvaillaan koodin toimintaa tarkemmin. (Van Heesch).

Yksityiskohtaisten kuvausten merkintään on olemassa useampia tapoja. Ensimmäinen tyyli on JavaDoc-tyyli, jossa kommenttilohko aloitetaan kahdella asteriskimerkillä. Toinen tyyli on Qt-tyyli, jossa kommenttilohko aloitetaan normaalisti C/C++kielen mukaisesti ja sen perään merkitään huutomerkki. Esimerkit näistä tavoista on esitetty kuvassa 10.

```
// Example 1. JavaDoc style

/**
 *  ...text...
 */

// Example 2. Qt style

/*!
 *  ...text....
 */

// Intermediate asterisks are optional,
// so both below examples are also valid

// Example 3. JavaDoc style without intermediate asterisks.

/**
 ...text...
 */

// Example 4. Qt style without intermediate asterisks.

/*!
 ...text....
 */
```

Kuva 10. JavaDoc- ja Qt-tyyliset kommenttilohkot. (Van Heesch).

Kolmas mahdollinen tyyli merkitä yksityiskohtaisia kuvauksia on käyttää vähintään kahta C++-kommenttiriviä, jossa jokainen rivi alkaa ylimääräisellä kenoviivalla tai huutomerkillä. Tässä tyylissä tyhjä rivi kommenttilohkon jälkeen päättää kommentin. Esimerkki tästä tyylistä on esitetty kuvassa 11.

```
// Example 1. C++ multi-line comment block with additional slash.

///
/// ...text...
///

// Example 2. C++ multi-line comment block with exclamation mark.

//!
//! ...text...
//!
```

Kuva 11. Kolmas tyyli yksityiskohtaisten kuvauksien merkintään. (Van Heesch).

Kommenttilohkojen korostamiseen voidaan käyttää kuvan 12 mukaisia kommenttilohkoja. Huomaa, että ensimmäisessä tyylissä kaksi kenoviivaa päättää normaalin kommentin ja aloittaa Doxygen-erikoiskommentin.

```
// Example 1. A more visible comment block.
// Special comment starts with two slashes.

/*******************//**
 *  ...text...
 ***********************/

// Example 2. A more visible comment block.

//////////////////////////
/// ...text...
//////////////////////////
```

Kuva 12. Korostettujen kommenttilohkojen merkitseminen. (Van Heesch).

Yksityiskohtaisia kuvauksia voi myös olla useampia, kuten kuvassa 13. Doxygen ei kuitenkaan voi taata, että niiden järjestys säilyy lopullisessa dokumentaatiossa. Tämä johtuu siitä, että Doxygen ei välttämättä lue lähdekooditiedostoja tietyssä järjestyksessä.

```
// Example 1. Multiple detailed descriptions.

//! Brief description, which is
//! really a detailed description since it spans multiple lines.

/*! Another detailed description!
 */
```

Kuva 13. Useita yksityiskohtaisia kuvauksia. (Van Heesch).

Lyhyiden kuvauksien merkintään on myös useampia tapoja. Ensimmäisessä tapa on käyttää `brief`-komentoa kommenttilohkossa, jossa lyhyt kuvaus päättyy tyhjään riviin kappaleen jälkeen.

Toisessa tavassa JavaDoc-kommenttilohko tai C++-erikoiskommentti aloittaa automaattisesti lyhyen kuvauksen, mikäli JAVADOC_AUTOBRIEF on käytössä. Lyhyt kuvaus päättyy pisteeseen ja välilyöntiin tai tyhjään riviin.

Esimerkit näistä kahdesta ensimmäisestä tavasta on esitetty kuvassa 14.

```
// Example 1. \brief command usage.

/*! \brief Brief description.
 *         Brief description continued.
 *
 *  Detailed description starts here.
 */

// Example 2. JavaDoc style comment with JAVADOC_AUTOBRIEF set to YES.

/** Brief description which ends at this dot. Details follow
 *  here.
 */

// Example 3. Special C++ style comment with JAVADOC_AUTOBRIEF set to YES.

/// Brief description which ends at this dot. Details follow
/// here.
```

Kuva 14. Lyhyiden kuvauksien merkitseminen. (Van Heesch).

Kolmas tapa on käyttää C++-erikoiskommentteja JAVADOC_AUTOBRIEF pois päältä. Tässä tavassa lyhyen kuvauksen maksimipituus on yksi rivi. Huomaa, että jälkimmäisessä esimerkissä tyhjä rivi lyhyen ja yksityiskohtaisen kuvauksen välissä on pakollinen. Esimerkki tästä kolmannesta tavasta on esitetty kuvassa 15.

```
// Example 1
/// Brief description.
/** Detailed description. */

// Example 2
//! Brief description.

//! Detailed description
//! starts here.
```

Kuva 15. Lyhyiden kuvauksien merkitseminen. (Van Heesch).

Muuttujien (variable), struktuurien (struct), unionien (union), luokkien (class) ja enumeraatioiden (enum) kommentit voidaan sijoittaa joko niiden yläpuolelle, kuten kuvassa 16, tai niiden jälkeen lisäämällä komment-tilohkoon <-merkki, kuten kuvassa 17.

```
//! An enum.
/*! More detailed enum description. */
enum TEnum {
             TVal1, /*!< Enum value TVal1. */
             TVal2, /*!< Enum value TVal2. */
             TVal3  /*!< Enum value TVal3. */
           }
//! Enum pointer.
/*! Details. */
*enumPtr,
//! Enum variable.
/*! Details. */
enumVar;

//! A public variable.
/*!
  Details.
*/
int publicVar;
```

Kuva 16. Kommentit yläpuolella (Van Heesch).

```
// Example 1. Detailed description block after a member.
int var; /*!< Detailed description after the member */

// Example 2. Detailed description block after a member.
int var; /**< Detailed description after the member */

// Example 3. Detailed description block after a member.
int var; /**< Detailed description after the member */

// Example 4. Detailed description block after a member.
int var; //!< Detailed description after the member
         //!<

// Example 5. Detailed description block after a member.
int var; ///< Detailed description after the member
         ///<

// Example 6. Brief description block after a member.
int var; //!< Brief description after the member

// Example 7. Brief description block after a member.
int var; ///< Brief description after the member
```

Kuva 17. Muuttujan kuvaustyylit. (Van Heesch).

Funktioiden parametrien kommentointiin voidaan käyttää joko `param`-komentoa ja `[in]`, `[out]` sekä `[in,out]` attribuutteja, tai ne voidaan dokumentoida suoraan funktion parametrilistassa (inline comment). Esimerkki funktioiden parametrien kommentoinnista on esitetty kuvassa 18.

```
// Example 1. Using param command for documenting function parameters.

/**
 *  \brief          Brief description of the function foo.
 *
 *  \param [in]     v   Description for input parameter v.
 *  \param [out]    p   Description for output parameter p.
 *  \return             Description for return value.
 *
 *  \details            Detailed description of the function foo.
 */
void foo(int v, int* p);

// Example 2. Using inline documentation block for
// documenting function parameters.

void bar(int i /**< [in] Description for input parameter i. */ );
```

Kuva 18. Funktioiden parametrien dokumentointi. (Van Heesch).

Kuvassa 19 on esimerkki Qt-tyylisesti kommentoidusta C++-koodista, jossa kommentit on merkitty Doxygenin ymmärtämään muotoon. Kuvassa 20 on esimerkki Doxygenin luomasta HTML-dokumentaatiosta, jossa kuvan 19 C++-koodi on dokumentoitu.

```
//!  A test class.
/*!
  A more elaborate class description.
*/
class QTstyle_Test
{
  public:
    //! An enum.
    /*! More detailed enum description. */
    enum TEnum {
                 TVal1, /*!< Enum value TVal1. */
                 TVal2, /*!< Enum value TVal2. */
                 TVal3  /*!< Enum value TVal3. */
               }
         //! Enum pointer.
         /*! Details. */
         *enumPtr,
         //! Enum variable.
         /*! Details. */
         enumVar;

    //! A constructor.
    /*!
      A more elaborate description of the constructor.
    */
    QTstyle_Test();
    //! A destructor.
    /*!
      A more elaborate description of the destructor.
    */
   ~QTstyle_Test();

    //! A normal member taking two arguments and returning an integer
value.
    /*!
      \param a an integer argument.
      \param s a constant character pointer.
      \return The test results
      \sa QTstyle_Test(), ~QTstyle_Test(), testMeToo() and publicVar()
    */
    int testMe(int a,const char *s);

    //! A pure virtual member.
    /*!
      \sa testMe()
      \param c1 the first argument.
      \param c2 the second argument.
    */
    virtual void testMeToo(char c1,char c2) = 0;

    //! A public variable.
    /*!
      Details.
    */
    int publicVar;

    //! A function variable.
    /*!
      Details.
    */
    int (*handler)(int a,int b);
```

Kuva 19. Qt-tyylisesti kommentoitu C++-koodia Doxygen-merkinnöillä. (Van Heesch).

Qt Style

Public Types | Public Member Functions | Public Attributes |
List of all members

QTstyle_Test Class Reference abstract

A test class. More...

Public Types

enum **TEnum** { **TVal1**, **TVal2**, **TVal3** }
An enum. More...

Public Member Functions

QTstyle_Test ()
A constructor. More...

~QTstyle_Test ()
A destructor. More...

int **testMe** (int a, const char *s)
A normal member taking two arguments and returning an integer value. More...

virtual void **testMeToo** (char c1, char c2)=0
A pure virtual member. More...

Public Attributes

enum **QTstyle_Test::TEnum** * **enumPtr**
Enum pointer. More...

enum **QTstyle_Test::TEnum** **enumVar**
Enum variable. More...

int **publicVar**
A public variable. More...

int(* **handler**)(int a, int b)
A function variable. More...

Kuva 20. Doxygenin luoma HTML dokumentaatio kuvan 19 C++-koodista. (Van Heesch).

5 LÄÄKETIETEELLISEEN KÄYTTÖÖN TARKOITETTUJEN LAITTEIDEN OHJELMISTOT

Lääketieteelliseen käyttöön tarkoitetuissa laitteissa on usein jonkinlainen ohjelmisto. Näiden ohjelmistojen kehitystyötä sääntelevät standardit, kuten myös muita lääketieteelliseen käyttöön tarkoitettujen laitteiden osia. Käsittelen tässä opinnäytetyössä IEC:n 62304-standardia, koska Delfin Technologies Oy:n ohjelmistokehitystyö on lääketieteelliseen käyttöön tarkoitettujen laitteiden ohjelmistojen kehitystä.

5.1 IEC 62304-standardi

Ohjelmisto on usein keskeinen osa lääketieteellisissä laitteissa. Jotta lääketieteelliseen käyttöön tarkoitetun laitteen turvallisuus ja tehokkuus voidaan varmistaa, tarvitaan tietoa ohjelmiston tarkoituksesta laitteessa, sekä todennettavissa olevia näyttöjä, että ohjelmisto täyttää tarkoituksensa aiheuttamatta ei-hyväksyttäviä riskejä. (IEC 62304 2006, 11).

IEC 62304-standardissa määritellään viitekehys lääketieteellisten laitteiden ohjelmistojen elinkaariprosesseille, aktiviteeteille sekä tehtäville, joilla lääketieteelliseen käyttöön tarkoitetun ohjelmiston turvallinen suunnittelu ja huolto voidaan varmistaa. Standardissa oletetaan myös, että lääketieteellisten laitteiden ohjelmistokehitys tapahtuu soveltuvien laadun- ja riskienhallintajärjestelmien puitteissa. IEC 62304-standardin piiriin kuuluviin lääketieteellisiin laitteisiin sovelletaan ISO 14971-standardin mukaista riskienhallintaprosessia. (IEC 62304 2006, 11).

Ohjelmiston epäsuorasti aiheuttamat vaarat tunnistetaan vaarojen tunnistamisaktiviteetin aikana. Päätös ohjelmiston käyttämisestä riskien hallintaan tehdään riskienhallinta-aktiviteetissä. Nämä aktiviteetit täytyy sisällyttää osaksi laitteen riskienhallintaprosessia ISO 14971-standardin mukaisesti. (IEC 62304 2006, 11).

Koska monet lääketieteellisten laitteiden vahingot, kuten esimerkiksi virheelliset ohjelmistopäivitykset liittyvät laitteiden huoltoon, on IEC 62304-standardin mukaan ohjelmiston ylläpitoprosessi aivan yhtä tärkeä kuin itse ohjelmistokehitysprosessi.

Kuvassa 21 on esitetty IEC 62304-standardin piiriin kuuluvat ohjelmistokehityksen aktiviteetit ja prosessit, ja kuvassa 22 puolestaan on esitetty standardin piiriin kuuluvat ohjelmiston ylläpidon aktiviteetit ja prosessit. (IEC 62304 2006, 11).

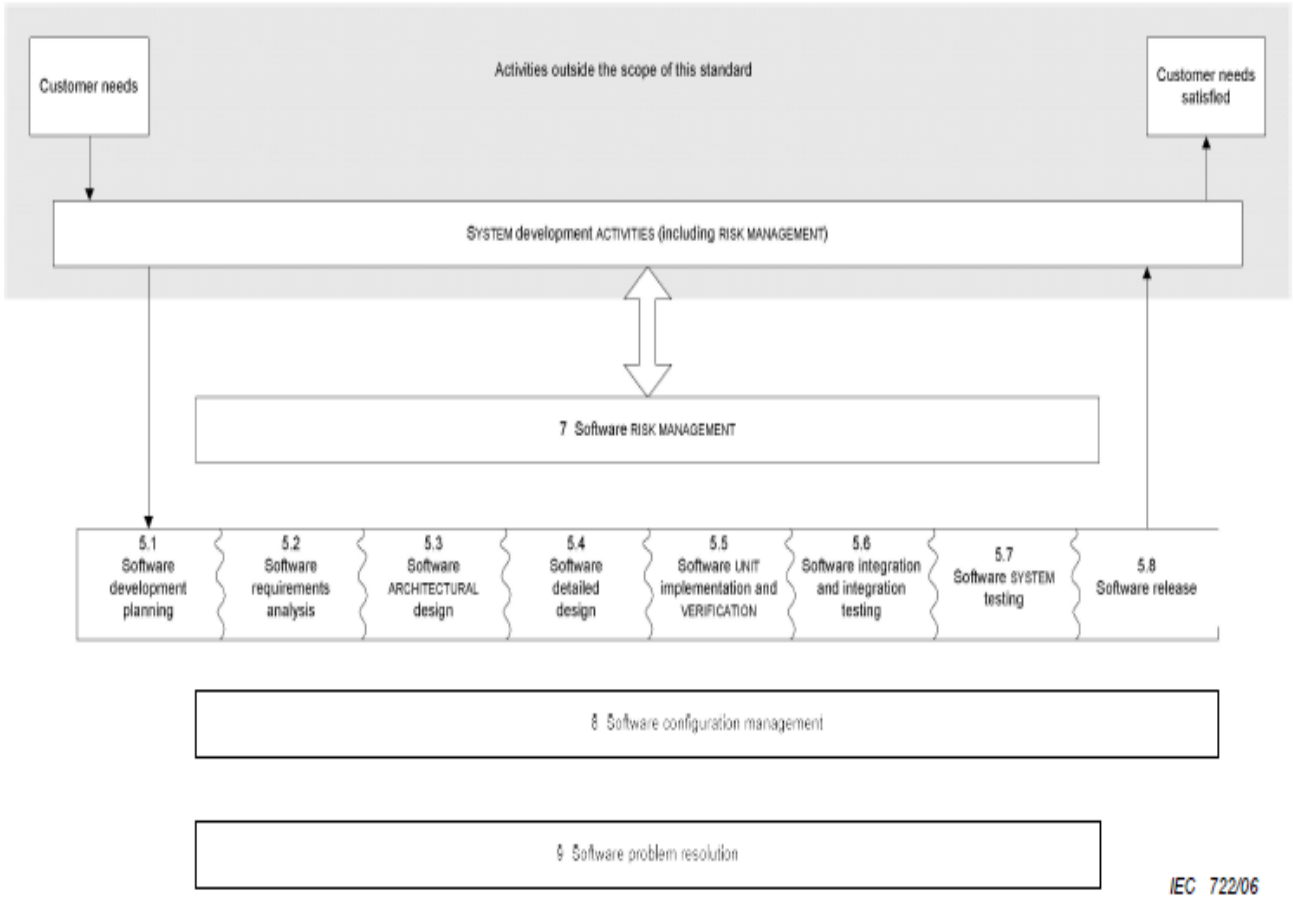


Figure 1 – Overview of software development PROCESSES and ACTIVITIES

Kuva 21. IEC 62304-standardin piiriin kuuluvat ohjelmistokehityksen aktiviteetit ja prosessit. (IEC 62304 2006).

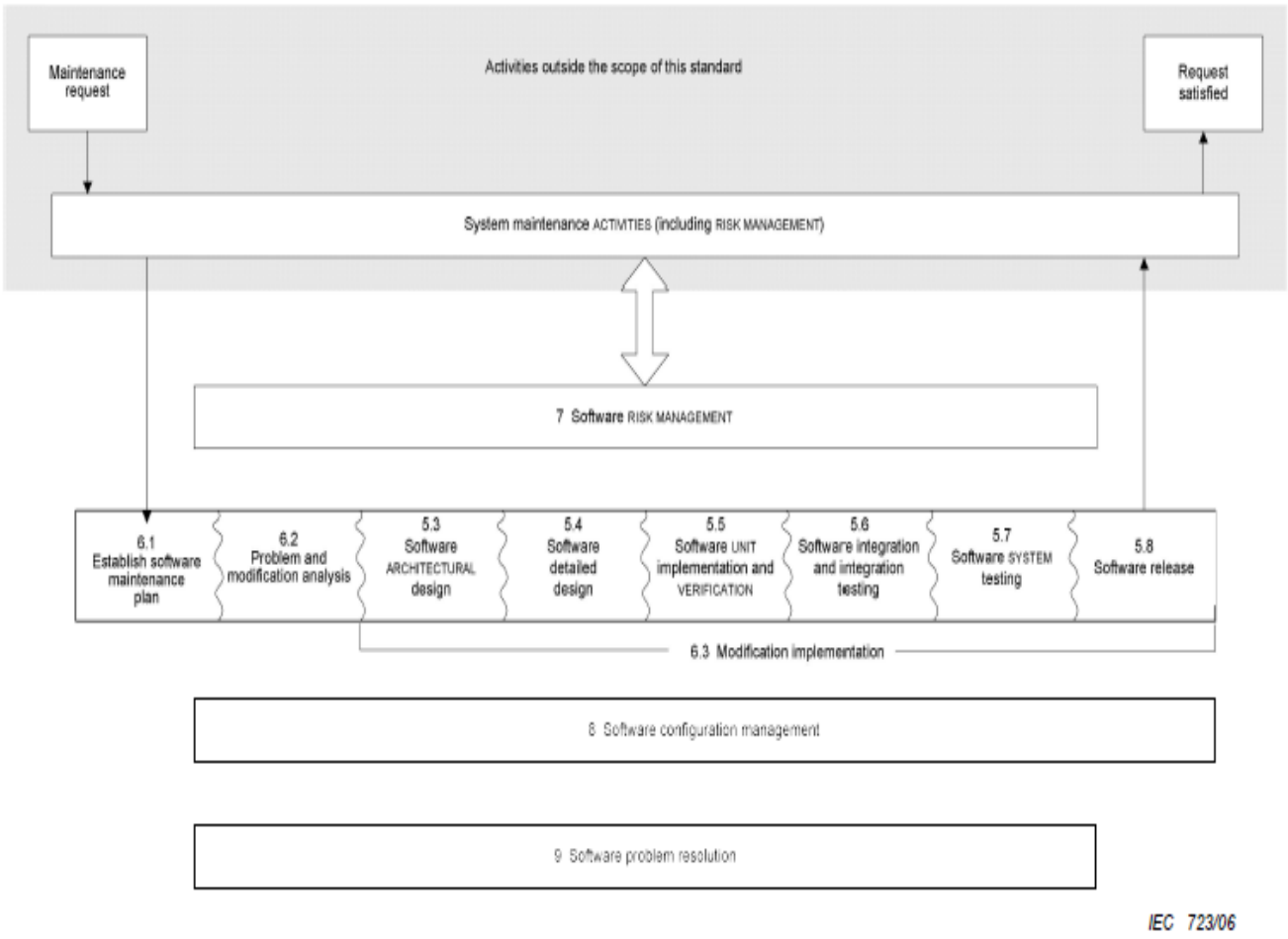


Figure 2 – Overview of software maintenance PROCESSES and ACTIVITIES

Kuva 22. IEC 62304-standardin piiriin kuuluvat ohjelmiston ylläpidon aktiviteetit ja prosessit. (IEC 62304 2006).

IEC 62304-standardi määrittelee elinkaaren vaatimukset lääketieteellisten laitteiden ohjelmistoille, ja standardissa kuvatut prosessit, aktiviteetit ja tehtävät muodostavat yhteisen viitekehyksen näiden lääketieteelliseen käyttöön tarkoitettujen ohjelmistojen elinkaariprosesseille. Standardi koskee sekä itsenäisiä lääketieteelliseen käyttöön tarkoitettuja ohjelmistoja, sekä ohjelmistoja, jotka ovat osana jotain lääketieteelliseen käyttöön tarkoitettua laitetta. (IEC 62304 2006, 17).

Jotta ohjelmisto on yhteensopiva IEC 62304-standardin kanssa, täytyy ohjelmiston toteuttaa turvallisuusluokituksensa mukaiset prosessit, aktiviteetit ja tehtävät standardin vaatimusten mukaisesti. Näiden toteutuminen tarkastetaan asianmukaisista dokumentaatioista. (IEC 62304 2006, 17).

5.2 IEC 62304-standardin ohjelmistojen turvallisuusluokitus

IEC 62304-standardi määrittelee kolme turvallisuusluokitusta, johon ohjelmisto voi kuulua. Ohjelmiston turvallisuusluokitus määräytyy sen mukaan, kuinka vakavaa vaaraa ohjelmisto tai laitteeseen liittyvä ohjelmisto voi virheellisesti toimiessaan aiheuttaa potilaalle, käyttäjälle ja sivullisille. Ohjelmiston turvallisuusluokitus on oletuksena C ennen turvallisuusluokituksen asettamista. Turvallisuusluokitusta määritettäessä oletetaan ohjelmiston toimintavirheen todennäköisyydeksi 100%. (IEC 62304 2006, 29,31)

Standardin mukaiset turvallisuusluokitukset ovat:

- Luokka A. Ei vammautumisen tai terveysvahingon vaaraa.
- Luokka B. Lievän vammautumisen tai terveysvahingon vaara.
- Luokka C. Vakavan vammautumisen tai kuoleman vaara.

Kuvassa 23 on esitetty ohjelmiston turvallisuusluokituksen määrittelyprosessi vuokaaviona.

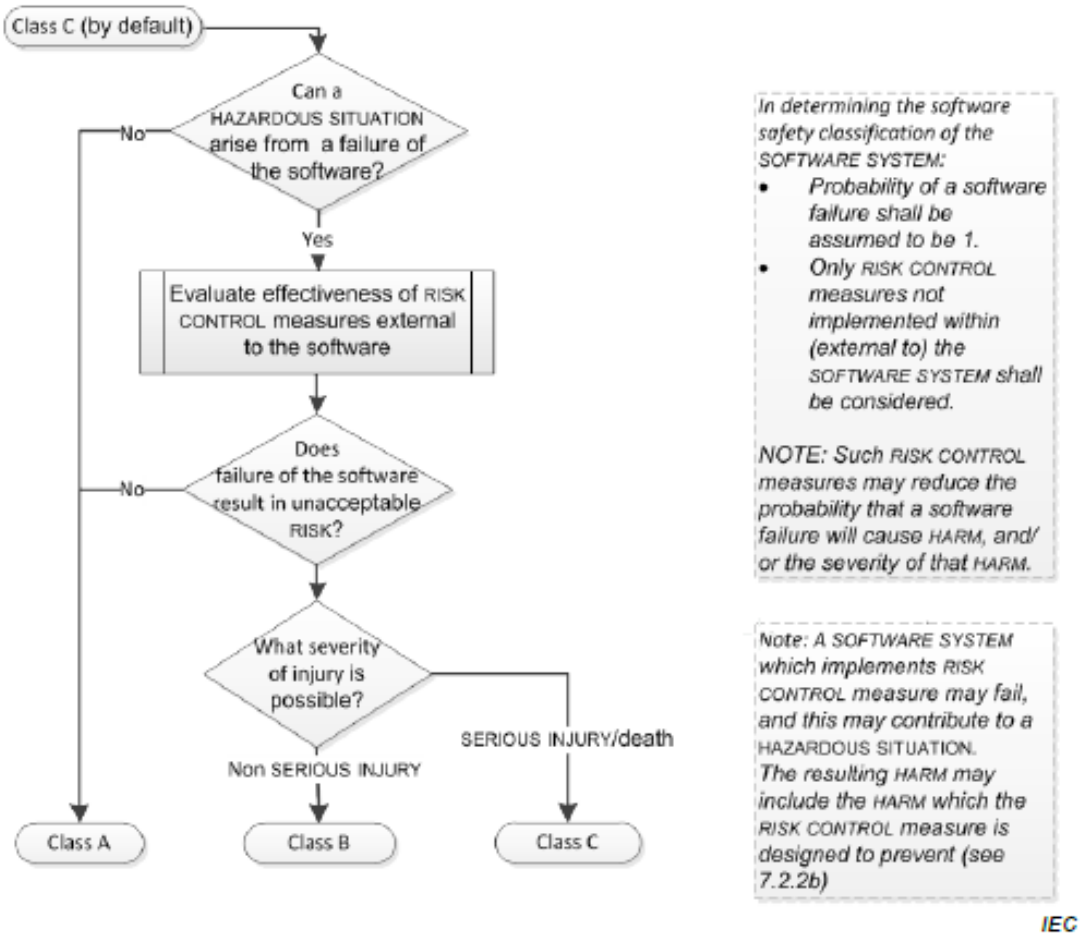


Figure 3 – Assigning software safety classification

Kuva 23. Ohjelmiston turvallisuusluokituksen määrittely. (IEC 62304 2006. Lisäys 1:2015).

Jos laitteistossa oleva mekaaninen rajoitus vähentää ohjelmistovirheen aiheuttamia riskejä, kuten vammojen vakavuutta tai kuoleman todennäköisyyttä hyväksyttävälle tasolle, voidaan ohjelmiston turvallisuusluokitusta alentaa tasolta C tasolle B. Samoin jos ym. rajoitus vähentää ei-vakavien vammojen riskiä hyväksyttävälle tasolle, voidaan luokitusta laskea tasolta B tasolle A. Hyväksyttävien riskien tasot määritellään ISO 14971-standardissa. (IEC 62304 2006, 29).

Ohjelmiston valmistajan täytyy asettaa turvallisuusluokitus jokaiselle ohjelmistojärjestelmälle, joka liittyy riskienhallintaan. Turvallisuusluokitus määritellään pohjautuen sen vaaran mahdollisiin vaikutuksiin, johon riskikontrolli liittyy. Nämä ohjelmistojärjestelmille asetetut luokitukset täytyy kirjata riskienhallintatiedostoon. (IEC 62304 2006, 30).

Jos ohjelmistojärjestelmä jaetaan pienemmiksi osiksi, perivät nämä osat saman turvallisuusluokituksen, kuin kokonaisjärjestelmä. Mikäli osille halutaan poikkeavasti asettaa eri turvallisuusluokitus kuin kokonaisjärjestelmällä tai muilla osilla, täytyy valmistajan perustella poikkeuksen syy. Perustelujen täytyy selventää, kuinka osat eristetään toisistaan, jotta ne voidaan luokitella erikseen. Valmistajan täytyy dokumentoida poikkeavien osien turvallisuusluokitukset. (IEC 62304 2006, 30).

Standardi vaatii, että kehitys- ja ylläpitoprosessit toteutetaan ohjelmistoryhmän korkeimman turvallisuusluokituksen omaavan osan mukaisesti kaikille osille. Tästä vaatimuksesta voidaan poiketa perustelulla, joka kirjataan riskienhallintatiedostoon. (IEC 62304 2006, 30).

5.3 IEC 62304-standardin mukainen kehitysprosessi

Valmistajan täytyy laatia ohjelmistokehityssuunnitelma, jossa määritellään ohjelmiston laajuuden, ulottuvuuden ja turvallisuusluokituksen mukaisten aktiviteettien menettelytavat. Suunnitelmassa täytyy joko määritellä ohjelmistokehityksen elinkaarimalli, tai siihen täytyy viitata, jos malli on määritelty toisessa dokumentissa. (IEC 62304 2006, 31).

Suunnitelmassa täytyy käsitellä seuraavat asiat kaikissa turvallisuusluokissa:

- Ohjelmiston kehityksessä käytettävät prosessit.
- Aktiviteettien ja tehtävien tuottamat tuotteet ja dokumentaatio.
- Järjestelmävaatimusten, ohjelmistovaatimusten, testien sekä ohjelmistossa toteutettujen riskikontrollien keskinäinen jäljitettävyys.
- Ohjelmiston konfiguraatio ja muutoksien hallinta, sisältäen SOUP-ohjelmistot (Software Of Unknown Provenance) sekä kehitystyössä käytettävät ohjelmistot.
- Ohjelmistoissa, tuotteissa sekä aktiviteeteissä havaittujen ongelmien ratkaisumenettelyt.

5.4 Ohjelmointi- ja dokumentointikäytännöt sekä IEC 62304-standardi

Jotta ohjelmisto olisi yhteensopiva standardin IEC 62304 kanssa, täytyy ohjelmiston valmistajan dokumentoida ohjelmisto, sekä kuvissa 30 ja 31 esitetyt kehitys- ja ylläpitotyön prosessit, aktiviteetit ja tehtävät. Standardin vaatimia dokumentteja on turvallisuusluokituksen mukaan useita erilaisia, ja tässä osiossa keskitytään vain tämän opinnäytetyön aiheen, ohjelmointi- ja dokumentointikäytäntöjen kannalta olennaisiin dokumentteihin.

IEC 62304-standardin mukaan tarvittavat dokumentoinnit riippuvat ohjelmiston turvallisuusluokituksesta. Alla on listattuna ohjelmointi- ja dokumentointikäytäntöihin liittyvät vaaditut dokumentit, jotka voivat joko sisältyä ohjelmistokehityssuunnitelmaan (Software Development Plan) tai ne voivat olla viittauksia ulkoisiin dokumentteihin:

- Ohjelmointistandardit [C]
- Metodit [C]
- Työkalut [C]
- Dokumentointisuunnitelma [A,B,C]
    - Otsikko, nimi tai nimeämiskäytäntö
    - Tarkoitus
    - Dokumenttien kohdeyleisö
    - Menetelmät ja vastuut dokumenttien tekemisestä, katselmoinnista, hyväksynnästä ja muokkauksesta.

6 TOTEUTUS

Opinnäytetyön tuloksena loin ohjelmointi- ja dokumentointikäytännön, sekä mallipohjat (template) uusien kooditiedostojen luontiin Delfin Technologies Oy:lle.

Ohjelmointikäytännön suunnittelun tavoitteena oli määritellä selkeä ja yksinkertainen viitekehys, jonka mukaan kaikki jatkossa tehtävä ohjelmiston kehitystyö tulee tehdä. Kun ohjelmointikäytäntö pidetään yksinkertaisena ja selkeänä, kehittäjät muistavat ja viitsivät käyttää sitä ja ohjelmiston laatu pysyy hyvänä. Osana opinnäytetyön toteutusta päivitin jo olemassa olevat koodit ohjelmointikäytännön tasalle. Lisäksi lisäsin olemassa oleviin koodeihin kommentit, jotta koodista voitiin luoda asianmukainen dokumentaatio Doxygenillä.

Dokumentointikäytännön suunnittelun tavoitteena oli määritellä koodissa käytettävä Doxygenin ymmärtämäkommentointityyli, jotta dokumentaatio voitaisiin tehdä helposti käyttäen Doxygeniä apuna.

6.1 Ohjelmointikäytäntö

Laatimani ohjelmointikäytäntö on selkä ja yksinkertainen. Käytäntö antaa kehittäjille liikkumavaraa kirjoittaa koodia omien mieltymyksiensä mukaan, ja käytännön pääasiallinen tavoite on pitää koodi luettavana ja helposti ymmärrettävänä.

Alla tiivistettynä laatimani ohjelmointikäytäntö:

- Käytä uuden koodin pohjana mallipohjia (template), jotta koodi pysyy yhteneväisenä vanhan koodin kanssa.
- Käytä tyhjiä rivejä lausekkeiden (declaration) väleissä jäsennelläksesi koodin osia, jotta koodin luettavuus ja johdonmukaisuus on mahdollisimman hyvä.
- Kommentoi koodia, mikäli sen tarkoitusperän ja toiminnan selventämiseksi on tarpeellista.
- Kommentoi funktion prototyypit header-tiedostoissa (*.h) Doxygen-muotoisilla kommenteilla.
- Kommentoi kooditiedostoissa (*.c) struktuurit yms. Doxygen-muotoisilla kommenteilla.
- Sijoita kommentit lausekkeiden yläpuolelle, tai niiden jälkeen, mikäli kommentit ovat lyhyitä.
- Alusta muuttujat johonkin tiettyyn alkuarvoon.
- Vältä globaalien muuttujien käyttöä mahdollisimman paljon. Esimerkiksi tapahtumalipun tms. tyyppiset globaalit muuttujat ovat hyväksyttäviä.
- Käytä kuvaavia muuttujien ja funktioiden nimiä. Esimerkiksi ”DOSReadMcuADC” ilmaisee selkeästi, että funktio käyttää prosessorin A/D-muunninta.
- Nimeä funktiot siten, että nimen ensimmäinen sana on se koodiyksikkö, johon funktio kuuluu. Esimerkiksi ”DOSReadMcuADC” kuuluu ”DOS”-yksikköön.

6.2 Dokumentointikäytäntö

Koodin kommenttien muoto tulee olla sellainen, että Doxygen osaa etsiä ja tunnistaa kommentit kooditiedostoissa. Kuvassa 24 on esimerkki header-tiedostoissa käytettävästä kommentin muodosta, jolla funktioiden prototyypit tulee kommentoida.

```
/**
 *  \fn             DescriptiveFunctionName ( unsigned char parameterName )
 *
 *  \brief          Do fancy stuff.
 *
 *  \param [in]     parameterName     parameter description
 *
 *  \return         something meaningful
 *
 *  \details        More detailed function description.
 */
unsigned int DescriptiveFunctionName( unsigned char parameterName );
```

Kuva 24. Käytettävä Doxygen-kommentin muoto header-tiedostoissa.

Makrot ja muuttujat tulee kommentoida kuvan 25 mukaisilla kommenteilla.

```
18  /*****************************************************************
19   * Preprocessor macros and definitions                           *
20   *****************************************************************/
21
22  #define FONT_5x7        (1) //!< Font size 5px*7px
23  #define FONT_9x14       (2) //!< Font size 9px*14px
24  #define FONT_16x25      (3) //!< Font size 16px*25px
```

Kuva 25. Makrojen ja muuttujien kommentointi.

Koodin toiminta tulee kommentoida kuvan 26 tyylisesti, mikäli kommentointi on perusteltua.

```
579  void displayDrawRectXY( GLCDCOL_t color, bool fill, GLCDX_t x, GLCDY_t y, unsigned char width, unsigned char height )
580  {
581      GLCDCOL_t prevColor = 0;
582
583      if( fill )
584      {
585          // Save fill color
586          prevColor = glcd_getfillcolor( );
587          glcd_setfillcolor( color );
588
589          // Use bar drawing function to draw a filled rectangle
590          // glcd_bar( x, y, xEnd, yEnd );    <-- Draws too big rectangle
591
592          glcd_fillrectround( x, y, width, height, 0 );
593
594          // Restore previous fill color
595          glcd_setfillcolor( prevColor );
596      }
597      else
598      {
599          // Store previous foreground color and set new color
600          prevColor = glcd_getcolor( );
601          glcd_setcolor( color );
602
603          // Draw rectangle to the desired display area
604          glcd_rectrel( x, y, width, height );
605
606          // Restore foreground color
607          glcd_setcolor( prevColor );
608      }
609  }
```

Kuva 26. Koodin toiminnan kommentointi.

6.3 Dokumentointityökalut

Dokumentoinnin apuna käytetään Doxygen-työkalua. Doxygenillä käydään lävitse kaikki kooditiedostot automatisoidusti, ja Doxygenillä tuotetaan HTML-muotoinen dokumentaatio. Esimerkkinä ja pohjana toimiva Doxygenin konfiguraatiotiedosto on tallennettu Delfin Technologies Oy:n SVN-versionhallintajärjestelmään.

6.4 Esimerkkejä työn tuloksista

Tässä osiossa on muutamia esimerkkejä yllä mainitun ohjelmointi- ja dokumentaatiokäytännön soveltamisesta koodiin.

Kuvassa 27 on esimerkki ohjelmointikäytännön soveltamisesta C-koodissa. Kuvan koodin rakenne on selkeä, funktion nimi sekä muuttujien nimet ovat kuvailevia ja koodia on jäsennelty loogisiksi osiksi välilyönneillä.

```
735  void displayDrawSymbolXY( flash unsigned long *pChar, GLCDX_t x, GLCDY_t y )
736  {
737      if( pChar != NULL ) // Sanity check
738      {
739          // Height is higher int and width is lower int of the first long
740          unsigned int height = ( ( *pChar ) >> 16 );
741          unsigned int width = ( ( *pChar ) & 0xFFFF );
742          unsigned long pixels = 0;
743          unsigned char offset = 0;
744          unsigned char column = 0;
745          unsigned char row = 0;
746          GLCDX_t xEnd = 0;
747          GLCDX_t xPos = 0;
748          GLCDY_t yEnd = 0;
749          GLCDY_t yPos = 0;
750
751          // Limit coordinates
752          xEnd = ( ( x + width ) > glcd_getmaxx( ) ? glcd_getmaxx( ) : ( x + width ) );
753          yEnd = ( ( y + height ) > glcd_getmaxy( ) ? glcd_getmaxy( ) : ( y + height ) );
754
755          // Increment pointer to point to actual character data
756          pChar++;
757
758          // Calculate offset to where the actual "pixels" start
759          offset = ( ( sizeof( unsigned long ) * CHAR_BIT ) - height );
760
761          for( column = 0 ; column < width ; ++column )
762          {
763              // Check if the character was cropped to smaller size.
764              // If so, break out of the loop so that we don't try to print outside of the display edge.
765              if( ( column + x ) > xEnd )
766              {
767                  break;
768              }
769
```

Kuva 27. Ohjelmointikäytännön mukaan kirjoitettua C-koodia.

Kuvassa 28 on kuvan 27 funktion prototyypin määrittely header-tiedostossa, sekä funktion prototyypin Doxygen-muotoinen kommentti.

```
966 /**
967  *  \fn             displayDrawSymbolXY( flash unsigned long *pChar, GLCDX_t x, GLCDY_t y )
968  *
969  *  \brief          Draw a special character image to display on absolute position.
970  *
971  *  \param [in]     pChar       pointer to character data on FLASH
972  *  \param [in]     x           X position
973  *  \param [in]     y           Y position
974  *
975  *  \return         none
976  *
977  *  \details        Character format:
978  *
979  *                  flash unsigned long CHAR_NAME[ size + width ] =
980  *                  {
981  *                      ( ( HEIGHT << 16) | WIDTH ), <-- height and width encoded in first long
982  *                        0b00000000000000000000000000000000,  // column 0
983  *                        0b11000000000000000000000000000000,  // column 1
984  *                        ...
985  *                        0b11000000000000000000000000000000,  // column ( width - 1 )
986  *                  };
987  */
988 void displayDrawSymbolXY( flash unsigned long *pChar, GLCDX_t x, GLCDY_t y );
```

Kuva 28. Kuvan 22 funktion prototyyppi header-tiedostossa sekä kommentointi.

Kuvassa 29 on esimerkki Doxygenillä tehdystä HTML-dokumentaatiosta, joka on luotu kuvan 28 header-tiedostosta.

displayDrawSymbolXY()

```
void displayDrawSymbolXY ( flash unsigned long *  pChar,
                           GLCDX_t                x,
                           GLCDY_t                y
                         )
```

Draw a special character image to display on absolute position.

Parameters

[in] **pChar** pointer to character data on FLASH

[in] **x** X position

[in] **y** Y position

Returns

none

Character format:

```
flash unsigned long CHAR_NAME[ size + width ] =
{
    ( ( HEIGHT << 16) | WIDTH ), <-- height and width encoded in first long
      0b00000000000000000000000000000000,  // column 0
      0b11000000000000000000000000000000,  // column 1
      ...
      0b11000000000000000000000000000000,  // column ( width - 1 )
};
```

Kuva 29. Kuvan 23 header-tiedoston kommentista luotu HTML-dokumentaatio.

7 YHTEENVETO

Tässä opinnäytetyössä tutkittiin ohjelmiston laatua, laadun tekijöitä sekä laadun merkitystä yrityksen ohjelmistokehityksen tehokkuuteen ja tuottavuuteen. Ohjelmiston laatuun liittyviä asioita on paljon, ja tässä opinnäytetyössä keskityttiin pääasiallisesti ohjelmointikäytäntöjen ja dokumentaation merkityksiin ohjelmiston laadussa.

Koska toimeksiantajan ohjelmistokehitys on lääketieteelliseen käyttöön tarkoitetun ohjelmiston kehitystä, tutustuttiin tässä opinnäytetyössä IEC 62304-standardiin, joka määrittelee lääketieteelliseen käyttöön tarkoitettuihin laitteiden ohjelmistokehitysprosessit ja -aktiviteetit.

Kokonaisuutena olen erittäin tyytyväinen opinnäytetyön lopputulokseen. Delfin Technologies Oy:n tuotekehitysosasto sai käyttöönsä ohjelmointi- ja dokumentointikäytännön, olemassa olevat koodit päivitettiin ohjelmointikäytännön mukaisesti, ja niistä luotiin dokumentaatio Doxygenillä. Jatkossa tuotekehitysosaston on huomattavasti helpompaa lisätä koodiin uusia ominaisuuksia sekä korjata ohjelmointivirheitä, kun koodi on selkeää ja johdonmukaista, ja ajantasainen dokumentaatio on jatkuvasti saatavilla.

Itse opin paljon ohjelmointikäytäntöjen ja dokumentaation merkityksestä koodin laatuun, sekä myös lääketieteelliseen käyttöön tarkoitettujen ohjelmistojen IEC 62304-standardista. Tämä opinnäytetyö opetti mm. mihin asioihin ohjelmointikäytännöissä tulisi kiinnittää huomiota, jotta saataisiin aikaiseksi parasta koodia mahdollisimman vähällä vaivalla. Lisäksi opin arvostamaan ajantasaisen dokumentaation olemassaoloa.

LÄHTEET JA TUOTETUT AINEISTOT

STRÅLIN, Tobias, GNANASAMBANDAM, Chandra, ANDÉN, Peter, COMELLA-DORDA, Santiago ja BURKACKY, Ondrej. 2016. Software development handbook – Transforming for the digital age. [Viitattu 2018-07-19] Saatavissa: https://www.mckinsey.com/~/media/McKinsey/Industries/High%20Tech/Our%20Insights/Software%20Development%20Handbook%20Transforming%20for%20the%20digital%20age/Software%20Development%20Handbook%20Transforming%20for%20the%20digital%20age.ashx

CHEMUTURI, Murali. 2011. Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers. Fort Lauderdale, Fla: J.Ross Publishing.

KSHIRASAGAR, Naik, PRIYADARSHI, Tripathy. 2008. Software Testing and Quality Assurance: Theory and Practice. Hoboken, N.J., Wiley-Spektrum

JAYARAM, N., M. 2007. What is Capability Maturity Model? [verkkoaineisto]. [Viitattu 2018-10-08]. Saatavissa: https://searchsoftwarequality.techtarget.com/definition/Capability-Maturity-Model

WIKIPEDIA, 2018. Coding conventions [verkkoaineisto]. [Viitattu 2018-10-09]. Saatavissa: https://en.wikipedia.org/wiki/Coding_conventions

VISSER, Joost, RIGAL, Sylvan, VAN DER LEEK, Rob, VAN ECK, Pascal ja WIJNHOLDS, Gijs. 2016. Building Maintainable Software, C# Edition. Sebastopol: O'Reilly Media Inc.

DORMAN, Scott. Why Coding Standards Are Important [verkkoaineisto]. [Viitattu 2018-06-26]. Saatavissa: https://scottdorman.github.io/2007/06/29/Why-Coding-Standards-Are-Important/

THE GNOME PROJECT. The Importance of Writing Good Code [verkkoaineisto]. [Viitattu 2018-06-18] Saatavissa: https://developer.gnome.org/programming-guidelines/stable/writing-good-code.html.en

MICROSOFT. 2015. C# Coding Conventions [verkkoaineisto]. [Viitattu 2018-06-26] Saatavissa: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

MICROSOFT. 2018. Secure Coding Guidelines [verkkoaineisto]. [Viitattu 2018-10-06] Saatavissa: https://docs.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines

GOOGLE. 2018. Google C++ Style Guide [verkkoaineisto]. [Viitattu 2018-10-06] Saatavissa: https://google.github.io/styleguide/cppguide.html

EASTERN PEAK. Source code documentation best practices [verkkoaineisto]. [Viitattu 2018-07-22] Saatavissa: https://easternpeak.com/blog/source-code-documentation-best-practices/

WRITE THE DOCS. A Beginner's guide to writing documentation [verkkoaineisto]. [Viitattu 2018-07-22] Saatavissa: http://www.writethedocs.org/guide/writing/beginners-guide-to-docs/

SOURCERER. Sourcerer application in GitHub [verkkoaineisto]. [Viitattu 2018-10-14]. Saatavissa: https://github.com/sourcerer-io/sourcerer-app

VAN HEESCH, Dimitri. Doxygen [verkkoaineisto]. [Viitattu 2018-07-28]. Saatavissa: https://www.stack.nl/~dimitri/doxygen/index.html

MDN WEB DOCS. Web API Documentation [verkkoaineisto]. [Viitattu 2018-07-22]. Saatavissa: https://developer.mozilla.org/en-US/docs/Web/API

IEC 62304. 2006. Medical device software – Software life cycle processes.

LIITE 1: MICROSOFT C# CODING CONVENTIONS AND SECURE CODING GUIDELINES

Tässä liitteessä on esitetty esimerkkinä muutamia kohtia Microsoftin C# ohjelmointikäytännöstä sekä turvallisen ohjelmoinnin ohjeista. Joitain koodiesimerkkejä on yhdistetty sekä joitain sanoja on muutettu viittaamaan koodiesimerkkeihin.

Täydellinen ohjelmointikäytäntödokumentti ja turvallisen ohjelmoinnin ohjeet löytyvät lähteistä Microsoft 2015 ja Microsoft 2018.

1 C# Coding Conventions (C# Programming Guide)

Coding conventions serve the following purposes:

- *They create a consistent look to the code, so that readers can focus on content, not layout.*
- *They enable readers to understand the code more quickly by making assumptions based on previous experience.*
- *They facilitate copying, changing, and maintaining the code.*
- *They demonstrate C# best practices.*

The guidelines in this topic are used by Microsoft to develop samples and documentation.

1.1 Naming Conventions

In short examples that do not include using directives, use namespace qualifications. If you know that a namespace is imported by default in a project, you do not have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

You do not have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

1.2 Layout Conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- *Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see Options, Text Editor, C#, Formatting.*
- *Write only one statement per line.*

- *Write only one declaration per line.*
- *If continuation lines are not indented automatically, indent them one-tab stop (four spaces).*
- *Add at least one blank line between method definitions and property definitions.*
- *Use parentheses to make clauses in an expression apparent, as shown in the following code.*

```
if((val1 > val2) && (val1 > val3))
{
    // Take appropriate action
}
```

1.3 Commenting Conventions

- *Place the comment on a separate line, not at the end of a line of code.*
- *Begin comment text with an uppercase letter.*
- *End comment text with a period.*
- *Insert one space between the comment delimiter (///) and the comment text, as shown in the following example.*

```
// The following declaration creates a query. It does not run
// the query.
```

- *Do not create formatted blocks of asterisks around comments.*

1.4 Language Guidelines

The following sections describe practices that the C# team follows to prepare code examples and samples.

1.4.1 String Data Type

- *Use string interpolation to concatenate short strings, as shown in the following code.*

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- *To append strings in loops, especially when you are working with large amounts of text, use a StringBuilder object.*

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

1.4.2 Implicitly Typed Local Variables

- *Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.*

```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- *Do not use var when the type is not apparent from the right side of the assignment.*

```
// When the type of a variable is not clear from the context, use an
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- *Do not rely on the variable name to specify the type of the variable. It might not be correct.*

```
// Naming the following variable inputInt is misleading.
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- *Avoid the use of var in place of dynamic.*
- *Use implicit typing to determine the type of the loop variable in for and foreach loops. The following examples use implicit typing in for and foreach statements.*

```
var syllable = "ha";
var laugh = "";
for (var i = 0; i < 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}

foreach (var ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

1.4.3 Arrays

- *Use the concise syntax when you initialize arrays on the declaration line.*

```
// Preferred syntax. Note that you cannot use var here instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };


// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a
time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

1.5 Secure Coding Guidelines

Evidence-based security and code access security provide very powerful, explicit mechanisms to implement security. Most application code can simply use the infrastructure implemented by .NET. In some cases, additional application-specific security is required, built either by extending the security system or by using new ad hoc methods.

Using .NET enforced permissions and other enforcement in your code, you should erect barriers to prevent malicious code from accessing information that you don't want it to have or performing other undesirable actions. Additionally, you must strike a balance between security and usability in all the expected scenarios using trusted code.

This overview describes the different ways code can be designed to work with the security system.

1.5.1 Securing resource access

When designing and writing your code, you need to protect and limit the access that code has to resources, especially when using or invoking code of unknown origin. So, keep in mind the following techniques to ensure your code is secure:

- *Do not use Code Access Security (CAS).*
- *Do not use partial trusted code.*
- *Do not use the AllowPartiallyTrustedCaller attribute (APTCA).*
- *Do not use .NET Remoting.*
- *Do not use Distributed Component Object Model (DCOM).*
- *Do not use binary formatters.*

Code Access Security and Security-Transparent Code are not supported as a security boundary with partially trusted code. We advise against loading and executing code of unknown origins without putting alternative security measures in place. The alternative security measures are:

- *Virtualization*
- *AppContainers*
- *Operating system (OS) users and permissions*
- *Hyper-V containers*

1.5.2 Security-neutral code

Security-neutral code does nothing explicit with the security system. It runs with whatever permissions it receives. Although applications that fail to catch security exceptions associated with protected operations (such as using files, networking, and so on) can result in an unhandled exception, security-neutral code still takes advantage of the security technologies in .NET.

A security-neutral library has special characteristics that you should understand. Suppose your library provides API elements that use files or call unmanaged code. If your code doesn't have the corresponding permission, it won't run as described. However, even if the code has the permission, any application code that

calls it must have the same permission in order to work. If the calling code doesn't have the right permission, a SecurityException appears as a result of the code access security stack walk.

LIITE 2: GOOGLE C++ STYLE GUIDE

Tässä liitteessä on esitetty esimerkkinä muutamia kohtia Google C++ Style. Joitain koodiesimerkkejä on yhdistetty sekä joitain sanoja on muutettu viittaamaan koodiesimerkkeihin.

Täydellinen C++ Style Guide löytyy lähteestä Google.

1 Google C++ Style Guide

1.1 Background

C++ is one of the main development languages used by many of Google's open-source projects. As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively.

Style, also known as readability, is what we call the conventions that govern our C++ code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.

Most open-source projects developed by Google conform to the requirements in this guide. Note that this guide is not a C++ tutorial: we assume that the reader is familiar with the language.

1.2 Goals of the Style Guide

Why do we have this document?

There are a few core goals that we believe this guide should serve. These are the fundamental whys that underlie all of the individual rules. By bringing these ideas to the fore, we hope to ground discussions and make it clearer to our broader community why the rules are in place and why particular decisions have been made. If you understand what goals each rule is serving, it should be clearer to everyone when a rule may be waived (some can be), and what sort of argument or alternative would be necessary to change a rule in the guide.

The goals of the style guide as we currently see them are as follows:

Style rules should pull their weight.

- *The benefit of a style rule must be large enough to justify asking all of our engineers to remember it. The benefit is measured relative to the codebase we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway. This*

principle mostly explains the rules we don't have, rather than the rules we do: for example, `goto` *contravenes many of the following principles, but is already vanishingly rare, so the Style Guide doesn't discuss it.*

Optimize for the reader, not the writer.

- *Our codebase (and most individual components submitted to it) is expected to continue for quite some time. As a result, more time will be spent reading most of our code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining, and debugging code in our codebase rather than ease when writing said code. "Leave a trace for the reader" is a particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code (for example, transfer of pointer ownership), leaving textual hints for the reader at the point of use is valuable (*`std::unique_ptr`* demonstrates the ownership transfer unambiguously at the call site).*

Be consistent with existing code.

- *Using one style consistently through our codebase lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your #includes only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.*

Be consistent with the broader C++ community when appropriate.

- *Consistency with the way other organizations use C++ has value for the same reasons as consistency within our code base. If a feature in the C++ standard solves a problem, or if some idiom is widely known and accepted, that's an argument for using it. However, sometimes standard features and idioms are flawed, or were just designed without our codebase's needs in mind. In those cases (as described below) it's appropriate to constrain or ban standard features. In some cases we prefer a homegrown or third-party library over a library defined in the C++ Standard, either out of perceived superiority or insufficient value to transition the codebase to the standard interface.*

Avoid surprising or dangerous constructs.

- *C++ has features that are more surprising or dangerous than one might think at a glance. Some style guide restrictions are in place to prevent falling into these pitfalls. There is a high bar for style guide waivers on such restrictions, because waiving such rules often directly risks compromising program correctness.*

Avoid constructs that our average C++ programmer would find tricky or hard to maintain.

- *C++ has features that may not be generally appropriate because of the complexity they introduce to the code. In widely used code, it may be more acceptable to use trickier language constructs, because any benefits of more complex implementation are multiplied widely by usage, and the cost in understanding the complexity does not need to be paid again when working with new portions of*

the codebase. When in doubt, waivers to rules of this type can be sought by asking your project leads. This is specifically important for our codebase because code ownership and team membership changes over time: even if everyone that works with some piece of code currently understands it, such understanding is not guaranteed to hold a few years from now.

Be mindful of our scale.

- *With a codebase of 100+ million lines and thousands of engineers, some mistakes and simplifications for one engineer can become costly for many. For instance, it's particularly important to avoid polluting the global namespace: name collisions across a codebase of hundreds of millions of lines are difficult to work with and hard to avoid if everyone puts things into the global namespace.*

Concede to optimization when necessary.

- *Performance optimizations can sometimes be necessary and appropriate, even when they conflict with the other principles of this document.*

The intent of this document is to provide maximal guidance with reasonable restriction. As always, common sense and good taste should prevail. By this we specifically refer to the established conventions of the entire Google C++ community, not just your personal preferences or those of your team. Be skeptical about and reluctant to use clever or unusual constructs: the absence of a prohibition is not the same as a license to proceed. Use your judgment, and if you are unsure, please don't hesitate to ask your project leads to get additional input.

1.3 C++ Version

Currently, code should target C++11, i.e., should not use C++14 or C++17 features. The C++ version targeted by this guide will advance (aggressively) over time.

Code should avoid features that have been removed from the latest language version (currently C++17), as well as the rare cases where code has a different meaning in that latest version. Use of some C++ features is restricted or disallowed. Do not use non-standard extensions.

1.4 Scoping

1.4.1 Nonmember, Static Member and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Do not use a class simply to group static functions. Static methods of a class should generally be closely related to instances of the class or the class's static data.

Pros:

Nonmember and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

Cons:

Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies

Decision:

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables and should nearly always exist in a namespace. Do not create classes only to group static member functions; this is no different than just giving the function names a common prefix, and such grouping is usually unnecessary anyway.

If you define a nonmember function and it is only needed in its `.cc` file, use internal linkage to limit its scope.

1.4.2 Local Variables

Place a function's variables in the narrowest scope possible and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment, e.g.:

```
int i;
i = f();      // Bad -- initialization separate from declaration.

int j = g();  // Good -- declaration has initialization.

std::vector<int> v;
v.push_back(1);  // Prefer initializing using brace initialization.
v.push_back(2);

std::vector<int> v = {1, 2};  // Good -- v starts initialized.
```

Variables needed for `if`, `while` and `for` statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

```
while (const char* p = strchr(str, '/')) str = p + 1;
```

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope. It may be more efficient to declare such a variable used in a loop outside that loop.

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.DoSomething(i);
}

Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.DoSomething(i);
}
```

1.4.3 Static and Global Variables

Objects with static storage duration are forbidden unless they are trivially destructible. Informally this means that the destructor does not do anything, even taking member and base destructors into account. More formally it means that the type has no user-defined or virtual destructor and that all bases and non-static members are trivially destructible. Static function-local variables may use dynamic initialization. Use of dynamic initialization for static class member variables or variables at namespace scope is discouraged but allowed in limited circumstances; see below for details.

As a rule of thumb: a global variable satisfies these requirements if its declaration, considered in isolation, could be `constexpr`.

Definition:

Every object has a storage duration, which correlates with its lifetime. Objects with static storage duration live from the point of their initialization until the end of the program. Such objects appear as variables at namespace scope ("global variables"), as static data members of classes, or as function-local variables that are declared with the static specifier. Function-local static variables are initialized when control first passes through their declaration; all other objects with static storage duration are initialized as part of program start-up. All objects with static storage duration are destroyed at program exit (which happens before unjoined threads are terminated).

Initialization may be dynamic, which means that something non-trivial happens during initialization. (For example, consider a constructor that allocates memory, or a variable that is initialized with the current process ID.) The other kind of initialization is static initialization. The two aren't quite opposites, though: static initialization always happens to objects with static storage duration (initializing the object either to a given constant or to a representation consisting of all bytes set to zero), whereas dynamic initialization happens after that, if required.

Pros:

Global and static variables are very useful for a large number of applications: named constants, auxiliary data structures internal to some translation unit, command-line flags, logging, registration mechanisms, background infrastructure, etc.

Cons:

Global and static variables that use dynamic initialization or have non-trivial destructors create complexity that can easily lead to hard-to-find bugs. Dynamic initialization is not ordered across translation units, and neither is destruction (except that destruction happens in reverse order of initialization). When one initialization refers to another variable with static storage duration, it is possible that this causes an object to be accessed before its lifetime has begun (or after its lifetime has ended). Moreover, when a program starts threads that are not joined at exit, those threads may attempt to access objects after their lifetime has ended if their destructor has already run.

Decision:

Decision on destruction

When destructors are trivial, their execution is not subject to ordering at all (they are effectively not "run"); otherwise we are exposed to the risk of accessing objects after the end of their lifetime. Therefore, we only allow objects with static storage duration if they are trivially destructible. Fundamental types (like pointers and `int`) are trivially destructible, as are arrays of trivially destructible types. Note that variables marked with `constexpr` are trivially destructible.

```
const int kNum = 10;  // allowed

struct X { int n; };
const X kX[] = {{1}, {2}, {3}};  // allowed

void foo() {
  static const char* const kMessages[] = {"hello", "world"};  // allowed
}

// allowed: constexpr guarantees trivial destructor
constexpr std::array<int, 3> kArray = {{1, 2, 3}};

// bad: non-trivial destructor
const string kFoo = "foo";

// bad for the same reason, even though kBar is a reference (the
// rule also applies to lifetime-extended temporary objects)
const string& kBar = StrCat("a", "b", "c");

void bar() {
  // bad: non-trivial destructor
  static std::map<int, int> kData = {{1, 0}, {2, 0}, {3, 0}};
}
```

Note that references are not objects, and thus they are not subject to the constraints on destructibility. The constraint on dynamic initialization still applies, though. In particular, a function-local static reference of the form `static T& t = *new T;` is allowed.

Decision on initialization

Initialization is a more complex topic. This is because we must not only consider whether class constructors execute, but we must also consider the evaluation of the initializer:

```
int n = 5;     // fine
int m = f();   // ? (depends on f)
Foo x;         // ? (depends on Foo::Foo)
Bar y = g();   // ? (depends on g and on Bar::Bar)
```

All but the first statement exposes us to indeterminate initialization ordering.

The concept we are looking for is called constant initialization in the formal language of the C++ standard. It means that the initializing expression is a constant expression, and if the object is initialized by a constructor call, then the constructor must be specified as `constexpr`*, too:*

```
struct Foo { constexpr Foo(int) {} };

int n = 5;  // fine, 5 is a constant expression
Foo x(2);   // fine, 2 is a constant expression and the chosen constructor
is constexpr
Foo a[] = { Foo(1), Foo(2), Foo(3) };  // fine
```

Constant initialization is always allowed. Constant initialization of static storage duration variables should be marked with `constexpr` *or where possible the* `ABSL_CONST_INIT` *attribute.. Any non-local static storage duration variable that is not so marked should be presumed to have dynamic initialization and reviewed very carefully.*

By contrast, the following initializations are problematic:

```
// Some declarations used below.
time_t time(time_t*);      // not constexpr!
int f();                   // not constexpr!
struct Bar { Bar() {} };

// Problematic initializations.
time_t m = time(nullptr);  // initializing expression not a constant
expression
Foo y(f());                // ditto
Bar b;                     // chosen constructor Bar::Bar() not constexpr
```

Dynamic initialization of nonlocal variables is discouraged, and in general it is forbidden. However, we do permit it if no aspect of the program depends on the sequencing of this initialization with respect to all other initializations. Under those restrictions, the ordering of the initialization does not make an observable difference. For example:

```
int p = getpid();  // allowed, as long as no other static variable
                   // uses p in its own initialization
```

Common patterns

- *Global strings: if you require a global or static string constant, consider using a simple character array, or a char pointer to the first element of a string literal. String literals have static storage duration already and are usually sufficient.*
- *Maps, sets, and other dynamic containers: if you require a static, fixed collection, such as a set to search against or a lookup table, you cannot use the dynamic containers from the standard library*

as a static variable, since they have non-trivial destructors. Instead, consider a simple array of trivial types, e.g. an array of arrays of ints (for a "map from int to int"), or an array of pairs (e.g. pairs of `int` *and* `const char*`*). For small collections, linear search is entirely sufficient (and efficient, due to memory locality). If necessary, keep the collection in sorted order and use a binary search algorithm. If you do really prefer a dynamic container from the standard library, consider using a function-local static pointer, as described below.*

- *Smart pointers (*`unique_ptr, shared_ptr`*): smart pointers execute cleanup during destruction and are therefore forbidden. Consider whether your use case fits into one of the other patterns described in this section. One simple solution is to use a plain pointer to a dynamically allocated object and never delete it (see last item).*
- *Static variables of custom types: if you require static, constant data of a type that you need to define yourself, give the type a trivial destructor and a* `constexpr` *constructor.*
- *If all else fails, you can create an object dynamically and never delete it by binding the pointer to a function-local static pointer variable:* `static const auto* const impl = new T(args...);` *(If the initialization is more complex, it can be moved into a function or lambda expression.)*

1.5 Functions

1.5.1 Output Parameters

Prefer using return values rather than output parameters. If output-only parameters are used, they should appear after input parameters.

The output of a C++ function is naturally provided via a return value and sometimes via output parameters.

Prefer using return values instead of output parameters since they improve readability and oftentimes provide the same or better performance.

Parameters are either input to the function, output from the function, or both. Input parameters are usually values or `const` *references, while output and input/output parameters will be pointers to non-*`const`*.*

When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.
This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

1.5.2 Write Short Functions

Prefer small and focused functions. We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code. You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

1.6 Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

1.6.1 General Naming Rules

Names should be descriptive; avoid abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word. Abbreviations that would be familiar to someone outside your project with relevant domain knowledge are OK. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia.

```
int price_count_reader;    // No abbreviation.
int num_errors;            // "num" is a widespread convention.
int num_dns_connections;   // Most people know what "DNS" stands for.
int lstm_size;             // "LSTM" is a common machine learning abbrevia-
tion.
```

```
int n;                     // Meaningless.
int nerr;                  // Ambiguous abbreviation.
int n_comp_conns;          // Ambiguous abbreviation.
int wgc_connections;       // Only your group knows what this stands for.
int pc_reader;             // Lots of things can be abbreviated "pc".
int cstmr_id;              // Deletes internal letters.
FooBarRequestInfo fbri;    // Not even a word.
```

Note that certain universally-known abbreviations are OK, such as `i` for an iteration variable and `T` for a template parameter.

For some symbols, this style guide recommends names to start with a capital letter and to have a capital letter for each new word (a.k.a. "Camel Case" or "Pascal case"). When abbreviations or acronyms appear in such names, prefer to capitalize the abbreviations or acronyms as single words (i.e `StartRpc()`*, not* `StartRPC()`*).*

Template parameters should follow the naming style for their category: type template parameters should follow the rules for type names, and non-type template parameters should follow the rules for variable names.

1.7 Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

1.7.1 Comment Style

Use either the `//` *or* `/* */` *syntax, as long as you are consistent.*

You can use either the `//` *or the* `/* */` *syntax; however,* `//` *is* much *more common. Be consistent with how you comment and what style you use where.*

1.7.2 Function Comments

Declaration comments describe use of the function (when it is non-obvious); comments at the definition of a function describe operation.

Function Declarations

Almost every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments may be omitted only if the function is simple and obvious (e.g. simple accessors for obvious properties of the class). These comments should be descriptive ("Opens the file") rather than imperative ("Open the file"); the comment describes the function, it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- *What the inputs and outputs are.*

- *For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.*
- *If the function allocates memory that the caller must free.*
- *Whether any of the arguments can be a null pointer.*
- *If there are any performance implications of how a function is used.*
- *If the function is re-entrant. What are its synchronization assumptions?*

Here is an example:

```
// Returns an iterator for this table.  It is the client's
// responsibility to delete the iterator when it is done with it,
// and it must not use the iterator once the GargantuanTable object
// on which the iterator was created has been deleted.
//
// The iterator is initially positioned at the beginning of the table.
//
// This method is equivalent to:
//    Iterator* iter = table->NewIterator();
//    iter->Seek("");
//    return iter;
// If you are going to immediately seek to another place in the
// returned iterator, it will be faster to use NewIterator()
// and avoid the extra seek.
Iterator* GetIterator() const;
```

However, do not be unnecessarily verbose or state the completely obvious.

When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overridden function. In many of these cases, the override needs no additional documentation and thus no comment is required.

When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like "destroys this object" are not useful. Document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

Function Definitions

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should not just repeat the comments given with the function declaration, in the `.h` *file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.*

1.7.3 Variable Comments

In general, the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for. In certain cases, more comments are required.

Class Data Members

The purpose of each class data member (also called an instance variable or member variable) must be clear. If there are any invariants (special values, relationships between members, lifetime requirements) not clearly expressed by the type and name, they must be commented. However, if the type and name suffice (`int num_events_;`), no comment is needed.

In particular, add comments to describe the existence and meaning of sentinel values, such as nullptr or -1, when they are not obvious. For example:

```
private:
 // Used to bounds-check table accesses. -1 means
 // that we don't yet know how many entries the table has.
 int num_total_entries_;
```

Global Variables

All global variables should have a comment describing what they are, what they are used for, and (if unclear) why it needs to be global. For example:

```
// The total number of tests cases that we run through in this regression
test.
const int kNumTestCases = 6;
```