Juhani Haikonen

Ylläpidettävyys avoimen lähdekoodin mukaisen ohjelmistotuotannon näkökulmasta

Tietojärjestelmätieteen
pro gradu -tutkielma
2.6.2006

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Haikonen, Juhani Mikael

Ylläpidettävyys avoimen lähdekoodin mukaisen ohjelmistotuotannon näkökulmasta / Juhani Haikonen

Jyväskylä: Jyväskylän yliopisto, 2006.

89 s.

Pro gradu –tutkielma

Ohjelmiston ylläpidettävyys on eräs tärkeä ohjelmiston laadun mittari. Ylläpidettävyyden arviointiin ei kuitenkaan vielä ole asetettu yhtä tiettyä standardoitua luokittelua. Osittain tämän vuoksi ylläpidettävyys on usein vain implisiittinen mittari, jota ei ole arvioitu eksplisiittisesti. Korkea ylläpidettävyys viittaa ohjelmiston ylläpidon helppouteen ja nopeuteen. Tällöin ohjelmiston lähdekoodi on sekä rakenteellisesti että toteutusteknisesti hyvin ymmärrettävässä muodossa.

Avoin lähdekoodi on perinteistä ohjelmistoalaa vallitsevan suljetun lähdekoodin näkökulmaa vasten syntynyt lähestymistapa – tai jopa ajattelutapa. Avoimen lähdekoodin mukaisen ohjelmistotuotannon pääperiaatteena on tuottaa sellaisia ohjelmistoja, joiden lähdekoodit ovat vapaasti saatavilla, luettavissa, muokattavissa ja edelleenlevitettävissä. Ohjelmistojen kehitykselle avoimen lähdekoodin mukainen ohjelmistotuotanto asettaa varsinkin käytäntöjen osalta useita erityislaatuisia piirteitä.

Tässä pro gradu -tutkielmassa keskitytään arvioimaan avoimen lähdekoodin ohjelmistojen ylläpidettävyysominaisuuksia. Arviointi tehdään käyttäen hyväksi P. Omanin ja J. Hagemeisterin esittämää tieteellistä luokittelua ohjelmistojen ylläpidettävyysaspekteista. Tutkielmassa käsitellään avoimen lähdekoodin ylläpidettävyysominaisuuksia ensimmäistä kertaa tässä laajuudessa.

Tutkimusongelmana on selvittää, mikä on avoimen lähdekoodin ohjelmiston vaikutus ylläpidettävyyteen arvioimalla edellä mainittujen tutkijoiden esittämiä ylläpidettävyysaspekteja. Tutkimusmenetelmänä käytetään käsitteellisteoreettista lähestymistapaa. Tutkielman tuloksena esitetään, että vaikkei avoimen lähdekoodin ohjelmistojen ylläpidettävyys ole poikkeuksellista, siinä ilmenee kuitenkin lähestymistavalle ominaisia piirteitä.

AVAINSANAT: ylläpidettävyys, ylläpito, avoin lähdekoodi, ohjelmistotuotanto

ABSTRACT

Haikonen, Juhani Mikael

Maintainability of open source software / Juhani Haikonen

Jyväskylä: University of Jyväskylä, 2006.

89 p.

Master's thesis

Software maintainability is an important meter of software quality. Assessment of maintainability is however still to be standardised. This partly affects to why maintainability is not frequently explicitly estimated. High maintainability refers to the simplicity and speed of maintenance efforts, due to comprehensible structural form and overall implementation of the source code.

Open source was born as an alternative methodology to resist the dominance and conventions of closed software development. The goal of open source software development is to produce freely available, readable, changeable and re-distributable source code. Open source methodology disposes varying amount of particular features to the software development model, though most of them are especially due to the different kinds of practices.

This thesis focuses on the evaluation of maintainability features of open source software. The evaluation is based on a scientific classification of maintainability aspects for conventional software by P. Oman and J. Hagemeister. The maintainability features of open source software have not earlier been studied in the extent that they are investigated in this thesis.

The research problem of the thesis is to recognize the various effects that open source software disposes to maintainability, by estimating the maintainability aspects of traditional software, as presented by the mentioned researchers. The used research method is conceptual-analytical/theoretical. The results of the thesis show that while maintainability of the open source software is not exceptional, it however includes some specific characteristics.

KEYWORDS: maintainability, maintenance, open source, software development

SISÄLLYSLUETTELO

1 JOHDANTO

Ohjelmiston ylläpitovaihe on perinteisessä mielessä tarkasteltuna ohjelmiston elinkaaren loppupäähän kuuluva vaihe. Sen keston pituus riippuu pääosin ohjelmiston aktiivisen käytön pituudesta. *Ohjelmiston ylläpidon* tehtäviksi käsitetään sekä ohjelmiston toimivuuden takaaminen että ohjelmiston toimintojen paikkansapitävyyden varmistaminen. Ylläpitovaiheen osuus käytössä olevien aktiivisten ohjelmistojen elinkaarista voi monista eri syistä riippuen olla ajallisesti hyvin pitkäkestoinen. Pisimmillään ohjelmistojen ylläpidolliset vaiheet voivat kestää jopa vuosikymmeniä. Läheskään kaikkia ohjelmistojen ilmentymiä ei kuitenkaan ylläpidetä puolta vuosikymmentä pidempään.

Ohjelmiston ylläpidettävyys määritellään ohjelmiston yhdeksi laadulliseksi mittariksi. Sen tehtävänä on puolestaan kertoa, missä suhteessa ja laajuudessa ylläpitoa voidaan suorittaa ylläpidettävälle ohjelmistolle. Ylläpidettävyys siis kuvaa sitä, kuinka yksinkertaista ylläpito on tai kuinka ylläpidettävä ohjelmisto on. Koska ylläpidettävyys on kokonaisuutena liian suuri tarkastelu- ja mittauskohde, se useimmiten jaetaan osakokonaisuuksiin. Näitä osakokonaisuuksia kutsutaan ohjelmiston *ylläpidettävyysaspekteiksi*. Tutkielmassa tehtävän arvioinnin lähtökohtana käytetään Omanin ja Hagemeisterin vuonna 1992 esittelemää tieteellistä luokittelua ohjelmistojen ylläpidettävyysaspekteista (Oman & Hagemeister 1992).

Kuten muiden muassa Raja ja Barry (2005, 43) ovat esittäneet, on ohjelmiston laatu tärkeä ominaispiirre, joka vaikuttaa ohjelmiston elinkaaren kustannuksiin, tehokkuuteen ja käyttöikään. Ohjelmiston laatu alenee väistämättä ikääntymisen myötä ja osittain erinäisten ylläpitotehtävien vaikutuksesta. Vaikka ohjelmistot näyttäisivät edelleen olevan kalliita kehittää ja etenkin ylläpitää, on ohjelmiston laatu yksi selvästi tärkeimmistä todellisista mittareista ohjelmiston ”onnistumiselle”. (Raja & Barry 2005, 43)

Avoin lähdekoodi on saanut osakseen kasvavaa kiinnostusta 1990-luvun loppupuolella ja 2000-luvulla. Sen on nähty murtavan ohjelmistoalaa vallitsevaa käsitystä (voidaan puhua niin kutsutusta *perinteisestä suljetun lähdekoodin näkökulmasta*) sekä ohjelmistojen rakentamisesta että tuotteistamisesta. Tätä aikaa edeltävinä vuosikymmeninä ohjelmisto nähtiin yleisesti kaupallisena tuotteena. A*voimen lähdekoodin ohjelmisto*, tai *avoin ohjelmisto*, ei sisälly tähän samaiseen käsitykseen.

Avoimen lähdekoodin lähestymistavan lähtökohtana on ohjelmiston lähdekoodin vapaa saatavuus, luettavuus, muokattavuus ja edelleenlevitettävyys. Tämän johdosta lähestymistavan mukainen ohjelmistotuotanto nähdään useimmiten eräänä vaihtoehtoisena tapana tuottaa ohjelmistoja. Ohjelmistoja kehitetään etenkin sellaisiin käyttötarkoituksiin, joiden puitteissa saavutettavat edut ylittävät tuotteistamismielessä rakennettujen ohjelmistojen aikaansaannokset.

Ohjelmistotuotteita tarjoavien yritysten joukko käsitetään tässä tutkielmassa ohjelmistoalaksi. Ohjelmistoalaa vallitseva perinteinen näkökulma käsitetään puolestaan tuon joukon edustajien näkemykseksi ohjelmistotuotannosta.

Tässä pro gradu -tutkielmassa käsitellään ohjelmiston ylläpidettävyyttä sellaisten avoimen lähdekoodin ohjelmistojen näkökulmasta, jotka ovat erityisesti avoimen lähdekoodin mukaisen ohjelmistotuotannon synnyttämiä kokonaisuuksia. Tämän lisäksi tutkielmassa käsitellään jonkin verran ohjelmistojen ylläpitoa. Tutkielman pääpaino on kuitenkin ylläpidettävyyden ja ylläpidettävyysaspektien arvioinnissa avoimen lähdekoodin ohjelmiston ominaispiirteisiin nähden. Ylläpidettävyyden arvioinnin lähtökohtana tutkielman piirissä oletetaan, että ohjelmistot ovat fyysisesti olemassaolevia ja toimivia ohjelmistoja. Näin ylläpidettävyyttä arvioidaan ”valmiin” ohjelmiston kannalta, eikä esimerkiksi ohjelmistotuotantoprosessin eri vaiheiden keskinäisten vaikutusten näkökulmasta.

Ensisijainen motiivi aiheen tutkimiseen on syntynyt käytännön tarpeesta. Avoimen lähdekoodin ohjelmiston ylläpito- ja etenkin ylläpidettävyyskysymykset ovat tieteellisissä tutkimuksissa vielä suurelta osin käsittelemättä. Tutkielman tarkoituksena on käsitellä juuri näitä näkökohtia sekä tuoda esille tutkielmaan valittujen kokonaisuuksien yhteisiä ominaisuuksia laajemmassa mittakaavassa.

Tutkielman tutkimusongelma on otsikkoa mukaileva: *"Mikä on avoimen lähdekoodin mukaisen ohjelmistotuotannon synnyttämien ohjelmistojen vaikutus ylläpidettävyyteen arvioituna erityisesti Omanin ja Hagemeisterin (1992) esittämiin ylläpidettävyysaspekteihin nähden?"*. Tutkielmassa keskitytään siis käsittelemään seuraavia alaongelmia:

1) Mitä ylläpidettävyys on avoimen lähdekoodin ohjelmiston kannalta tarkasteltuna?

2) Miten luokittelussa esitellyt suljetun lähdekoodin ohjelmistojen ylläpidettävyysaspektit ilmenevät avoimen lähdekoodin ohjelmistoissa sekä mitä erityispiirteitä voidaan niiden kohdalla havaita?

3) Mitä muita huomionarvoisia seikkoja avoimen lähdekoodin ohjelmiston ylläpidettävyyteen liittyy?

Tutkimusmenetelmänä käytetään *käsitteellisteoreettista* lähestymistapaa. Aluksi keskitytään käsittelemään aikaisempia tutkimuksia ja niiden tuloksia. Niiden pohjalta päädytään lopulta tekemään uusia johtopäätelmiä koskien avoimen lähdekoodin mukaisen ohjelmistotuotannon ylläpidettävyyttä. Tuloksiin sisältyy kaksi näkökohtaa. Sekä se, mitä erityishuomioita avoimen lähdekoodin lähestymistapa aiheuttaa ylläpidettävyydelle että se, kuinka luokittelun ylläpidettävyysaspektit ilmenevät avoimen lähdekoodin ohjelmistoissa.

Tutkielmassa esitellään aluksi aihepiirin keskeiset käsitteet ja niiden ominaisuudet. Luvussa kaksi käsitellään ohjelmiston ylläpitoa ja

ylläpidettävyyttä, jonka jälkeen kolmannessa luvussa esitellään avoimen lähdekoodin mukaista ohjelmistotuotantoa ja sen mukaisten ohjelmistojen ominaispiirteitä. Tämän jälkeen tutkielman neljännessä luvussa arvioidaan avoimen lähdekoodin ohjelmistojen ylläpidettävyyttä luvussa kaksi esiteltyjen ylläpidettävyysaspektien pohjalta. Lopuksi viidennessä luvussa esitetään tutkielman johtopäätökset, sekä tiivistetään yhteenveto. Tämän kiilamaisesti etenevän rakenteen tarkoituksena on ohjata lukijaa mahdollisimman kattavasti kohti tutkielman otsikon esittämää näkökantaa.

2 YLLÄPITO JA YLLÄPIDETTÄVYYS

Tässä tutkielman ensimmäisessä sisältöluvussa esitellään lukijalle käsitteet ohjelmiston ylläpito ja ylläpidettävyys. Ensimmäisessä alaluvussa käsitellään ylläpitoa, koska ylläpidettävyys on ylläpitoa luonnehtiva ominaisuus. Tämä on nähtävissä erityisesti siinä transitiivisessa mielessä, että ylläpito on aina riippuvainen ohjelmiston ylläpidettävyydestä. Tämän jälkeen toisessa alaluvussa keskitytään käsittelemään ylläpidettävyyttä ylläpitoa tarkemmalla tasolla.

Lukiessa on hyvä huomioida se seikka, että ylläpidettävyyttä ei ole tieteellisessä kirjallisuudessa tutkittu läheskään niin laajassa mittakaavassa kuin ylläpitoa. Myös määrittely- ja suunnitteluvaiheet määrittävät lopputuotteen ylläpidettävyyden aspekteja. Tämä toisin sanoen viittaakin siihen, että etenkin monet ohjelmistoprojektien alkupään tekijät vaikuttavat ohjelmiston ylläpidettävyyteen (García & Alvarez 1996, 90).

Tutkielman yhteydessä huomioidaan seuraavat ylläpidolliset käsitteet. V*ika* (*engl. fault*) tarkoittaa virheen ilmentymää ohjelmistossa. V*irhe* (*engl. error*) puolestaan kuvaa ohjelmiston poikkeamista spesifikaatiostaan.

2.1 Ylläpito

IEEE:n määritelmä ylläpidolle on seuraava:

> Ohjelmiston ylläpito käsittää kaikki sellaiset muutostoimenpiteet, jotka toteutetaan tuotteen julkaisun jälkeen vikojen korjauksina, suorituskykyä tai muita ominaisuuksia parantavina toimenpiteinä, tai tuotteeseen toteutettavina, jonkin tietyn käyttöympäristön vaatimina sovittamistoimina (IEEE 1998, 4).

Kyseisen kattavan määritelmän mukaisesti *ohjelmiston ylläpito* (*engl. software maintenance*) on ohjelmiston muuttamista tai korjaamista vaativa toimenpide. Tämä toimenpide kohdistetaan aina valmiiseen ohjelmistoon sitä muuttavina toimina. On huomioitava, että ohjelmiston valmius ei ole suoraan mitattavissa

oleva määre (tai tila) lähinnä siksi, ettei yhtäkään ohjelmistoa voida käytännössä koskaan luonnehtia valmiiksi. Tämän vuoksi ”valmis ohjelmisto” käsitetään sellaiseksi perinteistä suljetun lähdekoodin mukaista näkökulmaa vastaavan ohjelmistotuotannon synnyttämäksi ohjelmistokokonaisuudeksi, joka on sekä *hyväksymistestattu* (*engl. acceptance testing*) että julkaistu (eli otettu) aktiiviseen käyttöön. Huomionarvoinen seikka määritelmässä on se, että se on laadittu erityisesti suljetun lähdekoodin mukaisen ohjelmistotuotannon näkökulmasta, jossa ohjelmistot ovat julkaisuvaiheessa toimivia kokonaisuuksia.

Haworth ym. (1992, 107) kuvaavat, että ylläpidon pienin mahdollinen viitekehys koostuu kolmesta pääkomponentista, jotka ovat: ohjelmoija, lähdekoodi ja ylläpitovaade. Ohjelmoijan ominaisuuksiksi he esittävät kuuluvaksi muun muassa henkilökohtaisen kokemuksen, taidon, koulutuksen ja luonteenpiirteen. Lähdekoodin ominaisuuksiksi he mainitsevat ohjelmointikielen, *modulaarisuuden* (*engl. modularity*), *kytkennän* (*engl. coupling*) ja ymmärrettävyyden. Ylläpitovaateen ominaisuuksiksi tutkijat puolestaan esittävät ylläpidon tyypin sekä vaateen koon ja monimutkaisuuden. Kehyksen ulkopuoli koostuu organisaatioympäristöstä. (Haworth ym. 1992, 107-109)

Erityinen huomio on tehtävä siitä, että ylläpidon ja ylläpidettävyyden vaikutussuhde on aina kaksisuuntainen. Vaikka ylläpidettävyys siis lähtökohtaisesti vaikuttaakin ylläpitoon, myös ylläpito todellisuudessa vaikuttaa ylläpidettävyyteen. Näin on siksi, että ylläpitotehtävien tarkoituksena on muokata juuri ohjelmiston rakennetta ja sisältöä. Sekä *ennaltaehkäisevän ylläpidon* (*engl. preventive maintenance*) että *uudelleenrakenteistamisen* (*engl. restructuring*) tarkoituksina on muiden ohessa parantaa ylläpidettävyyttä (IEEE 1998).

2.1.1 Prosessi ja tehtävät

Yleisesti hyväksytyn periaatteen mukaisesti ohjelmisto säilyy käytössä niin kauan kuin se pystyy vastaamaan eri käyttäjien ja käyttäjäryhmien tarpeisiin riittävällä tasolla. Ohjelmiston käytön on tämän lisäksi oltava kaikkien resurssien käytön suhteen riittävän kustannustehokasta. On selvää, ettei mikään pitkäikäinen ohjelmisto välty muutosehdotuksilta ja -tarpeilta ajan myötä. Ohjelmiston kehittyminen ja kasvaminen onkin ylläpidollisten toimien peruspilari, sillä ilman ylläpitoa ei ohjelmistoa voida karakterisoida sen eri käyttötarpeita soveltuvampiin suuntiin.

Ylläpito on usein reaktiivista (ei-suunniteltua) ja voimakkaasti ohjelmiston vaihtelevista subjektiivisista ominaisuuksista riippuvaa toimintaa. Tämän vuoksi ylläpitoa ei kaikissa tilanteissa voidakaan tarkasti rajata sellaiseksi kokonaisuudeksi, joka miltei poikkeuksetta käsitettäisiin toteutettavaksi vain ohjelmiston julkaisun jälkeisten elinkaaren vaiheiden tiettyinä vakaasti toistettavina toimenpiteinä.

Loppukäyttäjille ohjelmiston elinkaari näyttäytyy evolutiivisena ohjelmiston versiointien muodossa. V*ersionhallinta* (*engl. version control*) ja *konfiguraationhallinta* (*engl. configuration management*) ovat olennainen osa ylläpitoa. Ohjelmiston versioiden ja konfiguraatioiden hallinta kasvaa monimutkaisemmaksi ohjelmistojen eri instanssien määrän ja ohjelmiston koon kasvamisen myötä. Varsinkin useiden eri ohjelmistoversioiden rinnakkainen ylläpito vaatii runsaasti ponnisteluja ja resursseja.

Kuten muun muassa Swanson (1999, 67) määrittelee, voidaan ylläpito käsittää tehtäväkokonaisuudeksi. Ylläpitosuoritteen syötteet koostuvat ylläpitotoimille varatuista ja sen aikana käytetyistä resursseista. Ylläpitosuoritteen tulosteet puolestaan koostuvat korjaavien, sopeuttavien ja täydentävien osatehtävien joukosta. (Swanson 1999, 67)

Ylläpidolla voidaan selkeästi nähdä olevan kaksi ulottuvuutta, sillä käsitteelliseksi ongelmaksi muodostuu rajanveto uuden kehityksen ja ”ylläpidon” suhteen. Swanson (1999, 68) onkin huomauttanut, että ylläpito ja uusi kehitys ovat toisilleen läheisiä toimenpiteitä, sillä molemmat ovat riippuvaisia käyttäjien innovaatioista ja tarpeista. Ylläpidon hän katsoo jatkuvaksi kehitykseksi, mutta uuden kehityksen hän näkee radikaalien muutosten tai lisäysten toimiksi (Swanson 1999, 68). Tässä tutkielmassa ohjelmiston uusi kehitys käsitetään kuitenkin ylläpitoon kuuluvaksi toimeksi. Näkökohtaa tukee se, että sillä ylläpidettävyys vaikuttaa aina ylläpidon lisäksi myös uuteen kehitykseen, niin näin on mahdollista päästä lähemmäs avoimen lähdekoodin näkökulmaa.

Ylläpito jaetaan usein eri osa-alueisiin, mutta kuten Chapin ym. (2001, 5-7) ovat huomauttaneet, on eri tahojen kesken vielä tiettyjen osa-alueiden määritelmien kesken epäselvyyksiä, etenkin eri tehtävien kohdistamisten osalta. Tutkielmassa käytetään hyväksi IEEE:n vuonna 1998 esittämää kokoelmaa ohjelmiston ylläpidon määritelmästandardiksi. Ylläpidon osa-alueista huomattavimmiksi voidaan lukea seuraavat: *korjaava ylläpito* (*engl. corrective maintenance*), *täydentävä ylläpito* (*engl. perfective maintenance*), *sopeuttava ylläpito* (*engl. adaptive maintenance*) ja *ennaltaehkäisevä ylläpito* (IEEE 1998).

Jokaisella näistä ylläpidon tärkeimmistä tehtäväalueista on oma tarkoituksensa, joita termit kuvaavat. Korjaavan ylläpidon, kaikkein tunnetuimman ja yleisimmän ylläpidon muodon, tehtäviin kuuluu reaktiivinen muutosten toteuttaminen. Se toisin sanoen kattaa ohjelmiston kaikkien vikojen vaatimat korjaustoimenpiteet. Täydentävän ylläpidon piiriin puolestaan kuuluu ohjelmiston sekä suorituskyvyn että ylläpidettävyyden parantaminen. Sopeuttavan ylläpidon tehtäviin toisaalta kuuluu pitää ohjelmisto toimivana ja käytettävänä muuttuneessa tai muuttuvassa järjestelmäympäristössä. Näistä useimmiten hankalimman ylläpidollisen toimen, ennaltaehkäisevä ylläpidon, tehtävänä on ennaltaehkäistä mahdollisesti myöhemmin syntyviä ongelmia esimerkiksi uudelleenrakenteistamalla ohjelmistoa. (IEEE 1998)

Kyseisten neljän käsitteen voidaan todeta kattavan suurimman osan ylläpitotehtävistä, joten siksi ne tässä yhteydessä katsotaan riittäviksi esiteltäviksi näkökulmiksi. On kuitenkin hyvä huomioida, että muun muassa juuri Chapin ym. (2001, 10) ovat määritelleet huomattavasti tarkemmankin ylläpitotehtäväjaon, joka koostuu kahdestatoista ohjelmiston evoluution ja ylläpidon eri tehtäväalueesta.

Käytännön ylläpitotyössä on järkevää suunnitella etukäteen ylläpitotehtäviä, jotta sekä toisaalta ylläpidettävyyttä voidaan pitää yllä (aktiivisesti arvioida) että toisaalta on olemassa etukäteistietoa myöhemmin mahdollisesti tarpeellisista ylläpitotehtävistä. Ylläpitosuunnitelmia on siis syytä kehittää ja toki suunnitelmien ajan tasalla pitäminen on koko ohjelmiston elinkaaren ajan tärkeää. Ylläpitosuunnitelmat eivät tosin aina ole tarkkoja ja virallisia suunnitelmia vaan enemmänkin ”tiekarttamaisia”. (Koponen & Hotti 2005a, 33)

Yksittäisen ylläpitäjän taidot riippuvat ylläpitäjän ohjelmointitaidoista, ohjelmiston ymmärryksestä ja myös henkilön analyyttisista ongelmanratkaisutaidoista. Ylläpitäjän taidot kehittyvät yleensä vuosien saatossa aktiivisen osallistumisen ja tekemisen kautta. Ylläpitotehtäviä pidetään etenkin suurten ohjelmistojen kannalta hyvin haastavina, joten aiempi kokemus ja tuntemus sekä toisaalta ohjelmistoista että toisaalta ylläpidosta on usein välttämätöntä.

2.1.2 Tarve

Ylläpito on miltei poikkeuksetta pakollinen osa kaikkien käyttöön otettujen ohjelmistojen elinkaaria. Ajankohtaiseksi ylläpidolliset toimet tulevat käytännössä silloin, kun ohjelmiston toimivuus on rajoitettu tai ohjelmiston käyttäytymistä on tarpeellista muokata. Ylläpito voi olla hyvin merkityksellistä esimerkiksi silloin, kun ohjelmistossa esiintyy lukuisia vikoja, eikä resurssien ja varojen puute mahdollista uuden ja paremman ohjelmiston rakentamista (tai toimeksiantona rakennuttamista).

Käytännön työssä ylläpito on suuri kustannustekijä ohjelmistotuotannossa (Raja & Barry 2005, 44), mutta se nähdään kuitenkin pakollisena toimena. Ylläpito käsitetään tavanomaisesti myös kaikkein resursseja kuluttavimmaksi työkokonaisuudeksi ohjelmistojen kehityksessä (García & Alvarez 1996, 89). Swanson (1999, 66) on muun muassa esittänyt, että kokonainen *tietojärjestelmä* (*engl. information system*) (johon lukeutuu ohjelmiston tai ohjelmistojen ympärille rakennettu järjestelmäkokonaisuus) on ylläpidettävä aina siihen pisteeseen saakka, jossa järjestelmän ylläpito, toiminta ja käyttö eivät enää ole taloudellisesti järkeviä vaihtoehtoja. Swanson (1999, 66) toisin sanoen esittääkin, että myös ohjelmiston toiminta ja käyttö voidaan käsittää ylläpidon toimialueeseen kuuluvaksi.

Korjaavaa ja sopeuttavaa ylläpitoa suoritetaan pakon sanelemana, mutta täydentävää ylläpitoa tehdään yleensä taloudellisten oikeutusten perusteella. Erityisesti parannuksia ohjelmistoihin tehdään silloin, kun odotetut hyödyt ylittävät kustannukset. Lisäksi pienet ja halvat parannukset toteutetaan säännöllisesti aikataulutettujen ylläpitosaavutusten (esimerkiksi julkaisuetappien) yhteydessä. (Swanson 1999, 67) Mitään yleiskäyttöistä mallia ylläpidon vaatiman työmäärän arviointiin järjestelmän ominaisuuksien avulla ei ole olemassa (Leitch & Stroulia 2003, 310).

Lopullisesti ylläpitotoimet päättyvät silloin kun ohjelmiston käytöstä luovutaan. Toki ylläpitotoimien säännöllisyys voi päättyä myös silloin, kun ohjelmisto kypsyy vakaasti toimivaksi ja käytettäväksi kokonaisuudeksi. Ohjelmiston käytön lopettaminen voi johtua esimerkiksi siitä, että ylläpitotoimien tai -vaateiden määrä kasvaa räjähdysmäisesti. Ylläpitotoimet voivat toisaalta tulla myös liian hankaliksi ja kalliiksi toteuttaa juuri esimerkiksi rappeutuneen ylläpidettävyyden johdosta. On toki aina olemassa myös se mahdollisuus, että parempi ohjelmisto vain yksinkertaisesti kehitetään ja otetaan aktiiviseen käyttöön.

2.2 Ylläpidettävyys

IEEE:n mukaan ylläpidettävyyden määritelmä on seuraavanlainen:

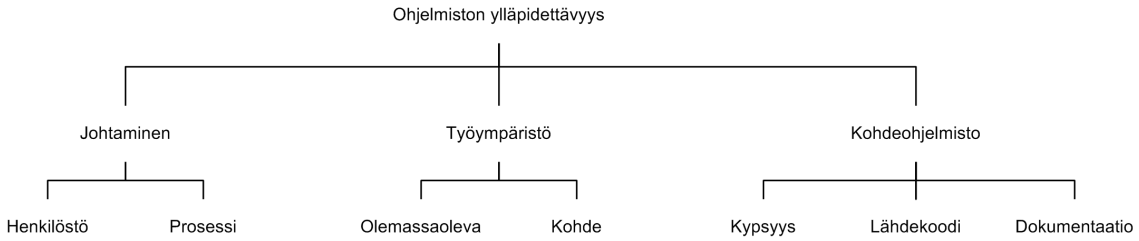
> Ohjelmiston ylläpidettävyys kuvaa vikojen korjauksien, suorituskykyä tai muita ominaisuuksia parantavien toimenpiteiden tai tuotteeseen toteutettavien, jonkin tietyn käyttöympäristön vaatimien, sovittamistoimien helppoutta (IEEE 1990, 127).

Ylläpidettävyys (*engl. maintainability*) on ohjelmiston laadullinen mittari. Se kattaa koko ohjelmiston elinkaaren, mutta realisoituu usein käytännössä vasta ohjelmiston ylläpitovaiheen alkamisen myötä. Ylläpidettävyyttä joudutaan arvioimaan osakokonaisuuksina, sillä käsite kattaa muuten liian laajan arvioitavan kokonaisuuden. Ylläpidettävyys on ylipäätään selvästi tärkeä ohjelmiston laadun tae. Edellisten lisäksi ylläpidettävyyden katsotaan yleisesti heikkenevän ohjelmistojen ikääntyessä, koska toistuvat ylläpitotoimet, vieläpä kulloinkin eri henkilöiden toimesta tehtyinä, heikentävät lähdekoodin rakennetta, luettavuutta, todennettavuutta ja korjattavuutta. Ylläpitotoimet saattavat myös luoda uusia ja jopa vaikeammin tunnistettavissa olevia virheitä.

On tärkeää huomata, kuten muun muassa Schach, Jin ym. (2002, 19) esittävät, ettei vielä ole olemassa mitään tarkkaa yleisesti hyväksyttyä ylläpidettävyyden luokittelua. Näin ollen myöskään yleisiä arviointimetriikoita ylläpidettävyyden arviointiin ei ole olemassa. Tämä ei onneksi kuitenkaan tarkoita sitä, etteikö erilaisia luokitteluja olisi esitelty.

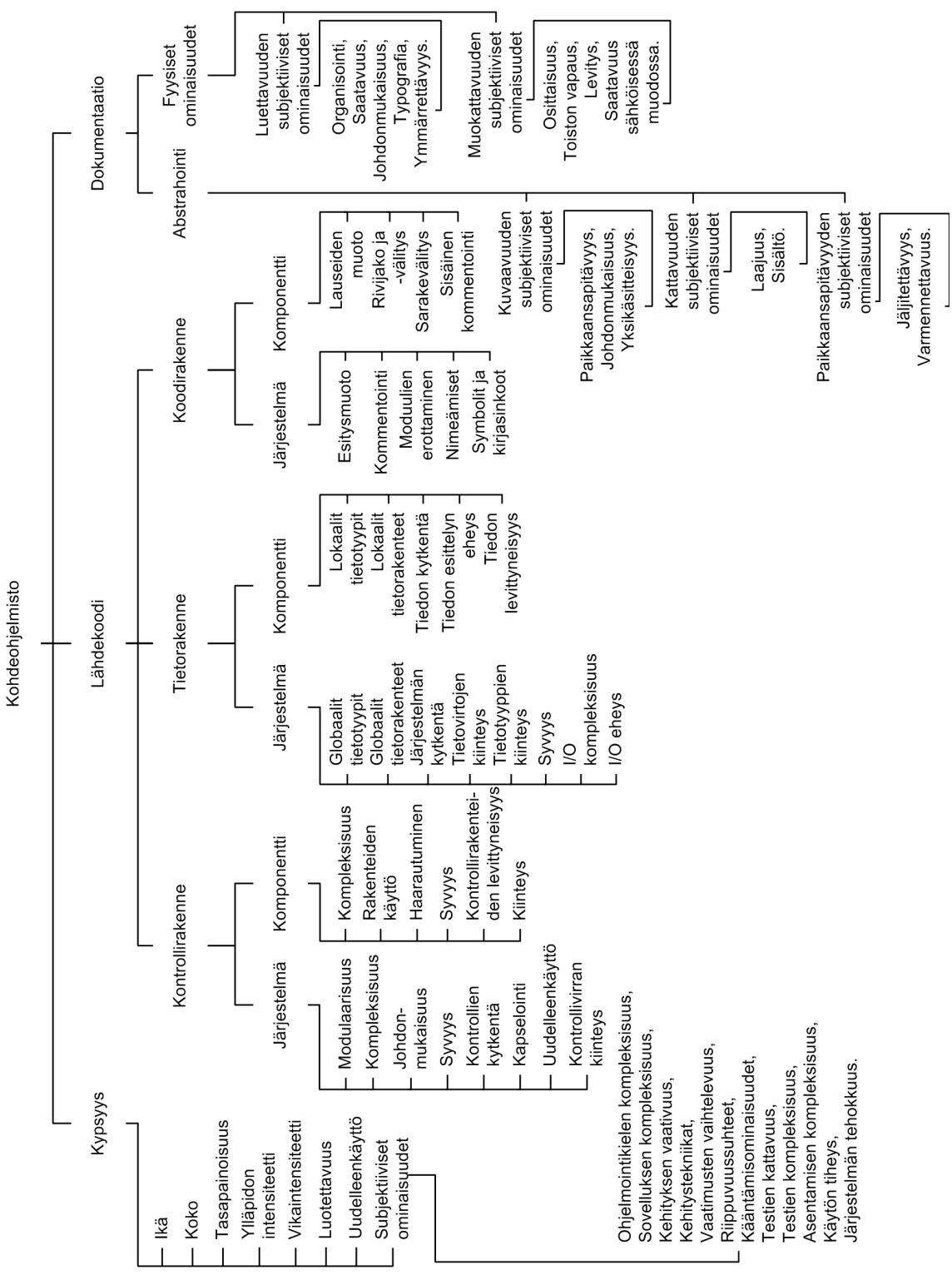
Omanin ja Hagemeisterin (1992, 337) mukaan kaikki ne ohjelmistojen osatekijät, jotka joko johtavat tai vaikuttavat ylläpidettävyyteen, voidaan järjestää mitattavissa olevien ominaisuuksien hierarkkiseksi rakenteeksi. Kuviossa yksi on esitetty ohjelmiston ylläpidettävyysaspektit, jonka lisäksi kuviossa kaksi on kohdeohjelmiston ylläpidettävyysaspektit tarkennettuna omaksi alahierarkiakseen. Tutkielmassa keskitytään tämän Omanin ja Hagemeisterin tieteellisen luokittelun arviointiin avoimen lähdekoodin ohjelmistojen näkökulmasta. Kyseinen luokittelu on valittu tutkielmaan siksi, että se toisaalta

on hyvin modularisoitu ja toisaalta tuntuu myös kaikista kattavimmalta rakenteelta systemaattiseen ylläpidettävyyden arviointiin. Tämä on myös Di Luccan ym. (2004, 280) näkemys.



KUVIO 1. Ohjelmiston ylläpidettävyysaspektit (Oman & Hagemeister 1992, 338).

Oman ja Hagemeister (1992) ovat jakaneet ohjelmiston ylläpidettävyyden siis kolmeen pääkategoriaan, joita ovat johtaminen, työympäristö ja kohdeohjelmisto. Tutkielman neljännessä luvussa syvennytään kohdeohjelmiston ylläpidettävyyden arviointiin, mutta toisaalta myös johtamista ja työympäristöä käsitellään tutkielman kolmannessa luvussa pintapuolisesti. Lähtökohtana arvioinnissa on se, että kaikki olemassaolevat ohjelmistot olisivat mitattavissa esitellyillä ylläpidettävyysaspekteilla. Vaikka valittu luokittelu kattaa myös ylläpidettävyysaspektien arviointimetriikat, joiden pohjalta eri piirteiden vaikutuksia ylläpidettävyyteen voidaan laskea, ei tässä tutkielmassa tulla itse piirteiden metriikoita tarkastelemaan ja käsittelemään. Ylläpidettävyyden arviointia voidaan ilmentää niin sanotun *ylläpidettävyysindeksin* (*engl. maintainability index, MI*) avulla. Ylläpidettävyyden mittarit mittaavat muun muassa ohjelmiston kompleksisuuden aspekteja (Koskinen ym. 2001, 132).



KUVIO 2. Kohdeohjelmiston ylläpidettävyysaspektit (Oman & Hagemeister 1992, 339).

Vaikka ylläpidettävyys voidaan nähdä yhdeksi erittäin hankalasti arvioitavissa olevaksi ohjelmiston laatu-attribuutiksi (Dagpinar & Jahnke 2003, 155), on olennaista tunnistaa ylläpidettävyyden eri aspekteja. Kun ylläpidettävyyden tekijät ovat tiedossa, voidaan muun muassa ohjelmistokehitystä parantaa ottamaan kaikki ylläpidettävyyden osatekijät tarkemmin huomioon. Ylläpidettävyyden arviointi ja mittaaminen on hyödyllistä etenkin silloin, kun ohjelmiston ylläpidettävyydestä saadaan hyväksikäytettäviä arvoja ennen ja jälkeen erinäisten ylläpidollisten toimien (Coleman ym. 1995, 9). Näin pystytään vertaamaan, onko ylläpidettävyys esimerkiksi huonontunut jonkin tietyn ylläpidollisen toimen johdosta. Tällöin ongelmatilanteisiin reagointi on välittömämpää perinteikkäiden vikaraporttien tai muiden ongelmien ilmaantumisten odottamisen sijaan.

Laatu ja *tuottavuus* (*engl. productivity*) ovat sidoksissa toisiinsa siten, että laadun kohoamisen nähdään usein parantavan myös tuottavuutta. Tämän vuoksi myös ylläpito voidaan nähdä tuottavampana toimintana juuri silloin, kun ohjelmiston ylläpidettävyys on korkealla tasolla. Toisaalta ylläpidettävyyden ei aina myöskään tarvitse olla ”tasaista”, sillä vain niiden ohjelmiston osa-alueiden, joiden ylläpito on runsasta, täytyy periaatteessa olla paremmin ylläpidettäviä kuin vähemmän ylläpitoa vaativat osa-alueet. (García & Alvarez 1996, 89)

Tutkielmaan valitun tieteellisen luokittelun iästä (noin neljätoista vuotta) johtuen nousee esiin kysymys ylläpidettävyysaspektien todennettavuudesta rakenteista lähestymistapaa modernimpaan oliolähestymistapaan nähden. Soveltuuko luokittelu ylipäätään oliolähestymistavan keskeisiin ominaispiirteisiin? Koska oliolähestymistavan piirteet ovat kehittyneet erityisesti hyvien suunnitteluperiaatteiden joukoksi, on itse lähdekoodista lievästi hankalaa tunnistaa kaikkia vastaavia piirteitä. Näin myös muun muassa oliolähestymistavan yhteydessä on suositeltavaa jo toteutusta edeltävissä ohjelmiston elinkaaren vaiheissa ottaa ylläpidettävyyden arviointi huomioon. Edellä todetusta huolimatta voidaan luokittelusta löytää oliopiirteitä.

Käytännössä tiedon ja toiminnan yhdistäminen olioksi johtaa siihen, että kontrolli- ja tietorakenteen yhteinen arviointi on oliolähdekoodissa lisäksi tarpeellista. Varsinaisia soveltuvuusesteitä ei luokittelusta kuitenkaan voida löytää.

Hordijk ja Wieringa (2005, 385) esittävät aiheellisesti, ettei ylläpidettävyyttä voida nähdä ainoastaan ohjelmiston ominaisuutena, sillä ylläpidettävyys on myös suuresti riippuvainen ylläpitäjistä ja heidän ominaisuuksistaan. Esimerkiksi ohjelmiston rakenteellinen kompleksisuus ja yksittäisten ylläpitäjien taidot vaikuttavat siihen, kuinka toteutettavat ylläpitotoimet lopulta vaikuttavat ylläpidettävyyteen. Tämän lisäksi vaikka lähdekoodi onkin välitön muutosten indikaattori, myös dokumentaatio on välillisesti ylläpidon kohteena. (Hordijk & Wieringa 2005, 385) Ylläpito käynnistää dokumentaation muokkauksen lisäksi aina myös testaus- ja raportointitoimenpiteitä ynnä muita senkaltaisia toimenpiteitä. Myös ohjelmointikielen valinta käsitetään yhdeksi vaikuttavaksi ylläpidettävyyden tekijäksi, eli eri ohjelmointikielillä tuotetun identtisen toiminnan omaavan lähdekoodin on eräissä tutkimuksissa havaittu sisältävän eroja ylläpidettävyydessä.

Ohjelmiston ylläpidettävyyden merkitys kasvanee enenevässä määrin, etenkin suurten ohjelmistojen osalta. Tämä perustuu siihen tosiasiaan, että sekä ikääntyvät vanhat ohjelmistot että uudet kehitettävät ohjelmistot vaativat molemmat aktiivisia ylläpitotoimia. Muun muassa vanhoja ohjelmistoja ei toisaalta välttämättä haluta uusia, ja toisaalta uusien ohjelmistojen koot ja kompleksisuusasteet kasvanevat vuosi vuodelta. (Coleman ym. 1995, 3)

Käytäntöihin liittyen Swanson (1999, 66-67) on esittänyt, että parempi ylläpidettävyys voi vähentää ylläpidon määrää, mutta toisaalta ylläpidon vähentäminen ei aina osoittaudu parhaaksi vaihtoehdoksi kokonaisuuden kannalta. Hän on lisäksi määritellyt seitsenkohtaisen joukon seuraavia ylläpidettävyyden edellytyksiä, jotka enemmän tai vähemmän auttavat ylläpitosaavutteen toteuttamista ”riittävällä tasolla”. *Yhteensoveltuvuus* (*engl.*

compatibility) viittaa järjestelmässä käytettyihin yleisiin tietoihin ja teknologioihin. *Eheys* viittaa järjestelmän luotettavaan ja virheettömään suorittamiseen. *Yksinkertaisuus* (*engl. simplicity*) viittaa järjestelmän kykyyn käyttää hyväksi muutamia yksinkertaisia toimintoja. *Käytettävyys* (*engl. usability*) viittaa organisaation toimille soveltuvien järjestelmän toiminnallisuuksien ja niiden sopivuuksien käyttöön. *Laajennettavuus* (*engl. extensibility*) viittaa järjestelmän laajennettavuuteen uusien tarpeiden ja vaatimusten täyttämiseksi. *Tasapainoisuus* (*engl. stability*) viittaa järjestelmän kykyyn sietää ympäristömuutoksia ja muita mukauttavia toimia. *Perehtyneisyys* (*engl. familiarity*) viittaa henkilöiden tuntemukseen järjestelmästä. (Swanson 1999, 66-67)

Yksi tärkeä kysymys liittyy siihen, kuinka saavutettu ylläpidettävyys voidaan säilyttää ainakin nykyisellä tasollaan ilman ylläpidettävyyden huononemista. Ylläpidettävyyttä onkin hyvin vaikea pitää yllä. Ylläpidettävyys jää hyvin usein pelkästään ylläpidon tehtävien ”jalkoihin” ja sen asemaa ei enää pitkän päälle osata tai muisteta arvostaa. (Swanson 1999, 73) Vaikka ohjelmisto olisi suunniteltu hyviä suunnitteluperiaatteita käyttäen, on usein mahdotonta pitää suunnitelmat nopeasti muuttuvan toteutuksen tasolla. Ongelmiin joudutaan helposti silloin, kun ylläpidon suunnittelu ei enää yhtäkkiä olekaan yhtenäinen alkuperäisten periaatteiden kanssa. Tämän vuoksi ylläpidettävyyttä on aika ajoin olennaista arvioida. Ylläpidettävyyden muutoksiin voidaan reagoida esimerkiksi tunnistamalla erilaiset ylläpidettävyysongelmat jo mahdollisesti ennen muutosten tekemistä.

Ylläpidettävyyden ongelmia käytännönläheisesti tarkasteltaessa voidaan ongelmat usein rajata ohjelmiston tiettyihin osa-alueisiin. Samoladas ym. (2004, 86) ovat erityisesti huomioineet, että Pareton periaatteen mukaan mistä tahansa tarkastelukohteesta riippumatta 80 prosenttia seurauksista johtuu 20 prosentista syistä. Periaatetta jatkamalla vain siis pieni osa ohjelmistoa on vastuussa suurimmista ohjelmiston virheistä. On siis tapana, että ylläpidettävyysongelmat kasautuvat ohjelmiston jonkin tietyn osan kohdalle.

(Samoladas ym. 2004, 86) Periaate ei luonnollisestikaan ota kantaa siihen, mistä osista ongelmat löytyvät. Muun muassa rakenteelliset ongelmat voivat kuulua ongelma-alueeseen.

Seuraavissa alaluvuissa esitellään kohdeohjelmiston ylläpidettävyyden aspekteja kuviossa kaksi esitetyn luokittelun rakenteen mukaisesti. Täysin kattavasti ei kaikkia eri aspekteja tässä yhteydessä käsitellä, vaan pyritään ensisijaisesti antamaan yleiskuva jokaisesta osakokonaisuudesta sekä niiden tarkoitusperistä ja aspekteista.

2.2.1 Kypsyys

Omanin ja Hagemeisterin (1992, 337-338) mukaan kohdeohjelmiston kypsyysaspektit kuvaavat kaikkia sellaisia ohjelmiston fyysisiä ominaisuuksia, jotka ovat yhteydessä kohdeohjelmiston ikään, käyttöön, luotettavuuteen sekä ylläpidon ja vikojen intensiteettiin. Kypsyys toisin sanoen mittaa sitä, kuinka pitkään ja kattavasti ohjelmistoa on käytetty, testattu ja muokattu. Se kuvaa ohjelmiston toiminnan yleistä stabiiliutta ja kehittyneisyyttä. Tähän vaikuttavat implisiittisesti muun muassa vikojen ja häiriöiden tunnistamiset ja korjaukset.

2.2.2 Lähdekoodi

Oman ja Hagemeister (1992, 338-340) kuvaavat kohdeohjelmiston lähdekoodin ominaisuuksien jakautuvan sekä kontrolli- ja tietorakenteisiin että lähdekoodin moniin ulkoasuteknisiin piirteisiin, tai toisin sanoen koodirakenteeseen. Edelleen kukin näistä kolmesta rakenteesta jakautuu järjestelmän ja komponentin näkökulmiin.

Jokaisen edellä mainitun rakenteen kohdalla järjestelmän näkökulma kuvaa valitun rakenteen yleisiä ja koko järjestelmän kattavia piirteitä, ja komponentin näkökulma kuvaa puolestaan valitun rakenteen tiettyjä kohdealueita, eli kaikkia tietyn komponentin ominaisuuksia. Komponentit koostuvat

moduuleista, ja moduuliksi käsitetään esimerkiksi jokin seuraavista kokonaisuuksista: ohjelma, funktio, proseduuri, alirutiini. (Oman & Hagemeister 1992, 340) Moduuliksi voidaan tässä yhteydessä käsittää myös olio. Komponentti puolestaan tarkoittaa esimerkiksi lähdekoodikirjastoa tai ohjelmistokomponenttia.

2.2.2.1 Kontrollirakenne

Järjestelmän kontrollirakenteen ylläpidettävyysaspektit ovat sellaisia ominaisuuksia, jotka liittyvät *moduulien välisiin* (*engl. intermodular*) (tai komponenttien välisiin) kontrolliominaisuuksiin, etenkin valittujen kontrollivirtojen ja algoritmien rakenteisiin. Järjestelmän kontrollirakenteen ylläpidettävyysaspekteja ovat muun muassa *modulaarisuus*, joka kertoo sekä moduulien määrän että keskimääräisen koon, *johdonmukaisuus*, joka kuvaa moduulien kokojen vaihtelut, *kontrollien kytkentä*, joka esittää kuinka kytkettyjä järjestelmän komponentit ovat, ja *kapselointi* (*engl. encapsulation*), joka kertoo tiedon kytkentöjen ja moduulien kutsujen väliset suhteet. (Oman & Hagemeister 1992, 340)

Komponentin kontrollirakenteen ylläpidettävyysaspektit ovat puolestaan sellaisia ominaisuuksia, jotka liittyvät *moduulien sisäisiin* (*engl. intramodular*) (tai komponenttien sisäisiin) kontrollivirtoihin, muun muassa kontrollivirtojen rakenteiden toteutustapoihin. Komponentin kontrollirakenteen ylläpidettävyysaspekteja kuvaavat muun muassa *rakenteiden käyttö*, joka esittää kuinka paljon yksipolkuisia rakenteita käytetään, *haarautuminen* (*engl. branching*), joka kertoo kuinka paljon ehdottomia rakenteita käytetään, ja *syvyys* (*engl. nesting*), joka kuvaa rakenteiden syvyyttä. (Oman & Hagemeister 1992, 340)

Tahvildari ym. (2001, 73) kuvaavat kontrollirakenteen korkean laadun muodostuvan muun muassa korkeasta modulaarisuudesta, korkeasta *kiinteydestä* (*engl. cohesion*), löyhästä kytkennästä, vähäisestä ehdottomien

hyppyjen käytöstä, tuntuvasta moduulien uudelleenkäytöstä, korkeasta kapseloinnista ja kontrollirakenteiden keskittämisestä.

2.2.2.2 Tietorakenne

Järjestelmän tietorakenteen ylläpidettävyysaspektit ovat sellaisia ominaisuuksia, jotka liittyvät moduulien (tai komponenttien) välisiin tietojen liikehtimisiin ja tallentamisiin, mukaan lukien globaalien tietorakenteiden ja -tyyppien käyttö. Järjestelmän tietorakenteen ylläpidettävyysaspekteja ovat muun muassa *globaalit tietotyypit* ja *-rakenteet*, jotka kuvaavat kuinka paljon globaaleja tietotyyppejä ja -rakenteita käytetään, *tietotyyppien kiinteys*, joka esittää kuinka paljon tietotyyppejä joudutaan konvertoimaan edestakaisin, ja *I/O kompleksisuus*, joka kertoo kuinka paljon I/O käsittelyjä käytetään. (Oman & Hagemeister 1992, 340-341)

Komponentin tietorakenteen ylläpidettävyysaspektit ovat puolestaan sellaisia ominaisuuksia, jotka ovat yhteydessä moduulien (tai komponenttien) sisäisiin tiedon tallentamistapoihin kattaen moduulien sisäiset tietovirrat. Komponentin tietorakenteen ylläpidettävyysaspekteja kuvaavat muun muassa *lokaalit tietotyypit* ja *-rakenteet*, jotka esittävät kuinka paljon lokaaleja tietotyyppejä ja -rakenteita käytetään ja *tiedon kytkentä*, joka kertoo globaalien tietorakenteiden ja proseduurien parametrien käytön suhdetta kaikkiin moduulissa käytettyihin tietorakenteisiin nähden. (Oman & Hagemeister 1992, 341)

Tietorakenteen korkean laadun Tahvildari ym. (2001, 73) esittävät muun muassa koostuvan tiedon korkeasta konsistenssista ja löyhästä kytkennästä sekä matalasta I/O kompleksisuudesta.

2.2.2.3 Koodirakenne

Järjestelmän koodirakenteen ylläpidettävyysaspektit liittyvät ohjelmiston tai järjestelmän yleiseen ulkoasuun ja kommentointiin. Tämän lisäksi komponentin

koodirakenteen ylläpidettävyysaspektit ovat toisaalta moduulien (tai komponenttien) sisäisiin typografioihin ja kommentointiin liittyviä ominaisuuksia. (Oman & Hagemeister 1992, 341) Tahvildari ym. (2001, 73) kertovat koodirakenteen korkean laadun johtuvan hyvästä lähdekoodin yleisestä muotoilusta sekä yhtenäisistä ja loogisista nimeämis- ja kommentointikäytänteistä.

2.2.3 Dokumentaatio

Dokumentaation ylläpidettävyysaspektit ovat kohdeohjelmiston dokumentaation abstraktioon ja fyysisiin ominaisuuksiin liittyviä ominaisuuksia. Dokumentaation abstraktion ylläpidettävyysaspektit ottavat kantaa siihen, kuinka itse lähdekoodi on abstrahoitu dokumentaatioksi. Toisaalta dokumentaation fyysiset ylläpidettävyysaspektit ovat ominaisuuksia, jotka kuvaavat, millainen fyysinen ulkoasu dokumentaatiolla on – esimerkiksi kuinka luettava dokumentaatio ylipäätään on. (Oman & Hagemeister 1992, 342)

2.3 Ylläpidettävyyden vaikutus

Ylläpidettävyyttä kuvaavan funktion mukaisesti korkean ylläpidettävyyden omaavien ohjelmistojen ylläpitotehtävät ovat helpommin ja nopeammin toteutettavissa kuin vastaavissa alhaisen ylläpidettävyyden omaavissa ohjelmistoissa. Tämä kertoo sen, että korkean ylläpidettävyyden aseman omaava ohjelmisto on rakenteellisesti ja toteutusteknisesti ymmärrettävämmässä muodossa kuin alhaisen ylläpidettävyyden tapauksessa.

Ylläpidettävyyden vaikutus ohjelmiston toimintaan, käyttöön ja ylläpitoon on sekä välitön että välillinen. Välitön yhteys ilmenee siitä, että ylläpidettävämpi ohjelmisto on yleensä käytettävämpi sekä ohjelmistossa esiintyy vähemmän virheitä, joka tosin ei ole ehdoton vaatimus hyvälle ylläpidettävyydelle. Välillinen yhteys puolestaan ilmenee siitä, että ylläpidettävämpi ohjelmisto on

helpommin laajennettavissa uusilla toiminnallisuuksilla ja se ei ole yliherkkä muun muassa muutoksille ympäristössään.

Ylläpidettävyys on käytännössä implisiittinen mittari, joka vaatii aina aluksi identifiointia tullakseen huomioonotetuksi. Ylläpidettävyyden arviointi kannattaa lähes poikkeuksetta aina, sillä vaikka sen aktiivinen arviointi vaatii lisäponnisteluja, osoittautuu ylläpidettävyyden huomioiminen aina panostamiseksi sekä kehitettävää ohjelmistoa että itse ohjelmistotuotantoprosessia kohtaan. Tämän voikin huomioida viimeistään silloin, kun ohjelmiston varsinaiset ylläpitotoimet käynnistetään.

Erityisesti yritysmaailmassa ylläpidettävyys vaikuttaa voimakkaasti ohjelmiston ylläpidon kustannuksiin, sillä ylläpito on perinteisesti yksi ohjelmiston suurimmista kustannuseristä. Pelkästään tämä näkökulma vaikuttaa siihen, että ylläpidettävyyden aktiivisen arvioinnin mahdollisimman aikaisin aloittaminen voi minimoida tulevia kustannuksia. *Ylläpidettävyyden suunnittelu* (*engl. design for maintainability*) voisi kuulua esimerkiksi yhdeksi määrittely- ja suunnitteluvaiheiden alitehtäväksi.

3 AVOIMEN LÄHDEKOODIN OHJELMISTO

Tässä tutkielman kolmannessa luvussa keskitytään käsittelemään sellaisia käsitteitä kuten avoin lähdekoodi, avoimen lähdekoodin ohjelmisto, avoimen lähdekoodin mukainen ohjelmistotuotanto ja avoimen lähdekoodin projekti. Kaikki neljä käsitettä liittyvät olennaisesti toisiinsa, mutta eivät kuitenkaan tarkoita täysin samaa asiaa. Luku jakautuu neljään alalukuun, joista ensimmäisessä tarkastellaan avoimen lähdekoodin lähestymistavan ominaispiirteitä. Toisessa alaluvussa käsitellään lähestymistavan projekteille läheisiä käytänteitä. Kolmannessa sekä neljännessä alaluvussa käsitellään ja kevyesti myös kerrataan lähestymistavan mukaisen ohjelmistotuotannon kautta saavutettavia etuja ja toisaalta sen synnyttämiä haittatekijöitä.

Avoin lähdekoodi (*engl. open source*) viittaa lähdekoodin yleiseen saatavuuteen. Käytännössä tämä tarkoittaa sitä, että kehitetyt ohjelmistot usein määritellään yleisesti saataviksi, muokattaviksi ja edelleenlevitettäviksi käytettyjen lisenssien määräämien ehtojen mukaisesti. Ohjelmistoalalla vallitseva näkökulma, s*uljettu lähdekoodi* (*engl. closed source*), puolestaan viittaa lähdekoodin saatavuuden rajoittamiseen. Näiden kahden toisistaan täysin eroavan filosofisen näkökulman vuoksi on syntynyt osittainen vastakkainasettelun kuva, joka on luonut (muutos-) vastarintaa ohjelmistoalalla vallitsevan lähtökohdan kannattajien joukossa. Näkökulmat voidaan tosin nähdä myös ”kokonaisuutta” täydentävinä tekijöinä.

Vaikka avoin lähdekoodi (alunperin nimeltään vapaa ohjelmisto) on vasta 1990-luvulla ja sen jälkeen noussut yleiseen tietoisuuteen varsinkin internetin suosion kasvaessa, ovat sen lähtökohdat pitkälle jopa 1950-luvulla (Raymond 2001). Avoimen lähdekoodin mukainen ohjelmistotuotanto on kuitenkin vielä suhteellisen uusi alue, joten sen merkitys ohjelmistojen kehitykseen on elävä ja muotoutuva.

Kuten muun muassa Mockus ym. (2002, 310) esittävät, avoimen lähdekoodin lähestymistapaa kuvataan usein lähtökohdaltaan periaatteellisemmaksi lähestymistavaksi kuin suljetun lähdekoodin lähestymistapaa. Lähtökohdiltaan avoimen lähdekoodin lähestymistavan sanotaan pohjautuvan kehittäjien motivaatioihin ja filosofisiin periaatteisiin sekä jopa toisaalta uudenlaiseen markkinatalousvoimaan. Eräät ovat myös viitanneet avoimen lähdekoodin olevan ”demokraattisempi” lähestymistapa. Tämän lisäksi etenkin oppiminen on korostetussa asemassa kyseisessä lähestymistavassa.

Eräs tunnettu organisaatio, joka vuodesta 1998 lähtien on edistänyt avoimen lähdekoodin aatetta, on nimeltään *Open Source Initiative (OSI)* (*suom. avoimen ohjelmiston aloite*). Käytännön työssään se sertifioi kaikki avoimen lähdekoodin lisenssit, jotka täyttävät sen asettamat kriteerit. OSI:n määrittelemä kymmenkohtainen *vaatimuskriteeristö* (*Open Source Definition, OSD*) avoimen lähdekoodin ohjelmistolle käsittää seuraavat piirteet (versio 1.9) (Open Source Initiative, 2005):

1. Vapaa edelleenlevitysoikeus (*engl. Free Redistribution*)
2. Ohjelman ja sen lähdekoodin vapaa levitettävyys (*engl. Source Code*)
3. Vapaa muokattavuus (*engl. Derived Works*)
4. Tekijän tunnustaminen lähdekoodin luojana (*engl. Integrity of the Author's Source Code*)
5. Yhtäläiset oikeudet käyttäjästä tai käyttäjäryhmästä riippumatta (*engl. No Discrimination Against Persons or Groups*)
6. Yhtäläiset oikeudet toimialasta riippumatta (*engl. No Discrimination Against Fields of Endeavor*)
7. Yhtäläisen lisenssin levitys (*engl. Distribution of License*)

8. Lisenssi ei saa olla tuotekohtainen (*engl. License Must Not Be Specific to a Product*)

9. Lisenssi ei saa rajoittaa muita ohjelmistoja (*engl. License Must Not Restrict Other Software*)

10. Lisenssi ei saa rajoittua vain tietyn teknologian piiriin (*engl. License Must Be Technology-Neutral*)

Yksinkertaistettuna voidaan todeta, että ohjelmisto on avoimen lähdekoodin ohjelmisto silloin, kun se täyttää yhden avoimen lähdekoodin lisenssin asettamat ehdot.

Avoimen lähdekoodin mukainen ohjelmistotuotanto (*engl. open source software development*) on ohjelmistokehitystä, jonka tuotoksena on tarkoitus tuottaa *avoimen lähdekoodin ohjelmisto* (*engl. open source software*) (*lyh. OSS*). Erityisesti a*voimen lähdekoodin projektit* (*engl. open source projects*) ovat avoimen lähdekoodin mukaisen ohjelmistotuotannon ilmentymiä.

Tämä ohjelmistokehityksen muoto voidaan periaatteessa toteuttaa joko minkä tahansa olemassaolevan tai ennaltatuntemattoman ohjelmistotuotantoprosessiviitekehyksen kautta. Yleisen käsityksen mukaan avoimen lähdekoodin projektit käsitetään niin sanottuihin *ketteriin menetelmiin* (*engl. agile methodologies*) perustuviksi verrattuna ohjelmistoalalla vallitsevassa roolissa oleviin niin kutsuttuihin *raskaisiin menetelmiin* (*engl. heavy methodologies*). Ketteriä menetelmiä voidaan karakterisoida sellaisiksi menetelmiksi, joissa ”turha byrokratia” on karsittu pois ja työn tekemistä on virtaviivaistettu siten, että vain menetelmien määrittelemiin olennaisiin tehtäviin pyritään keskittymään. Mielenkiintoista on se, että avoimen lähdekoodin mukainen ohjelmistotuotanto voidaan jopa itsessään nähdä, ainakin hyvin pitkälti, ketteränä menetelmänä (Warsta & Abrahamsson 2003, 147). Teoriassa on toki täysin mahdollista toteuttaa avoimen lähdekoodin

mukaista ohjelmistotuotantoa myös raskain menetelmin, mutta usein erityispiirteiset käytänteet estävät tämän.

Tässä tutkielmassa tehdään rajanveto sille, että avoimen lähdekoodin ohjelmisto käsitetään vain avoimen lähdekoodin mukaisen ohjelmistotuotannon aikaansaattamaksi ohjelmistoksi. Tämä merkitsee sitä, että esimerkiksi aikaisemmin suljetun lähdekoodin ohjelmiston siirtämistä lisenssimuutoksella avoimen lähdekoodin ohjelmistoksi ei käsitetä tutkielmassa aidoksi avoimen lähdekoodin ohjelmistoksi.

Avoin lähdekoodi ei ole yksikäsitteinen termi, vaan vaikka kaikissa avoimen lähdekoodin projekteissa on yhteisiä piirteitä, ne myös aina sisältävät tiettyjä yksilöllisiä piirteitä. Kaikkia avoimen lähdekoodin projekteja yhdistää vain kaksi yhteistä tuntomerkkiä. Ensimmäinen on se, että kaikki avoimen lähdekoodin projektit täyttävät edellä esitetyn OSI:n kymmenkohtaisen vaatimuskriteeristön avoimelle lähdekoodille. Toinen tuntomerkki on puolestaan se, että ohjelmistojen kehittäjät ovat aina myös käyttäjiä, vaikka käyttäjät eivät läheskään aina ole kehittäjiä. Avoimen lähdekoodin projektit poikkeavat toisistaan yleensä hyvin voimakkaasti muun muassa seuraavissa tuntomerkeissä: projektien lähtökohdissa, kehittäjien motivaatiotasoissa, kehittäjäyhteisöissä, kehityksen ylläpidoissa, ohjelmistojen lisensoinneissa ja koissa. (Gacek & Arief 2004, 34-36)

Avoimella lähdekoodilla on joukko vastustajia. Esimerkiksi Glass (2004) on kritisoinut avoimen lähdekoodin taloudellisten perusteiden viittaavan toimimattomaan kommunismin aatteeseen sekä laajaan ohjelmistoalan taantumaan (toisin sanoen paluuseen 1950-luvun periaatteisiin). Hän on myös väittänyt, että avoin lähdekoodi voi hyvin osoittautua vain väliaikaiseksi utopiaksi. Lisäksi muun muassa ohjelmistoalan suuryritys Microsoft on ottanut näkyvästi aktiivisen aseman avoimen lähdekoodin vastustajana. Tämä on toisaalta luonnollista, sillä avoin lähdekoodi on saavuttanut riittävästi huomiota useilla eri tahoilla muodostaakseen tietynlaisen uhkan perinteisten suljetun

lähdekoodin toimijoille, eli ohjelmistoalaa vielä vain ja ainoastaan perinteisellä tavalla ymmärtäville ja johtaville yrityksille.

Kaikkein tunnetuimpia avoimen lähdekoodin ohjelmistoja ovat muun muassa: *Apache* (http-palvelinjärjestelmä), *CVS* (lähdekoodien versionhallintajärjestelmä), *Linux* (käyttöjärjestelmä), *Mozilla* (http-selain) ja *MySQL* (tietokannan hallintajärjestelmä).

3.1 Ominaispiirteet

Avoimen lähdekoodin ohjelmiston voidaan sanoa koostuvan monenlaisista ominaispiirteistä, mutta tässä tutkielmassa on piirteet jaettu seuraaviin neljään näkökulmaan: saatavuus, lisensointi, kehityksen luonne ja uudelleenkäyttö. Seuraavissa alaluvuissa käsitellään jokaisen piirteen merkitystä avoimen lähdekoodin lähestymistavalle.

3.1.1 Saatavuus

Saatavuus (*engl. availability*) viittaa avoimen lähdekoodin ohjelmistoon ja erityisesti sen lähdekoodiin. Tämä lähtökohtaisesti kattaa ohjelmiston ilmaisen käytön, sen lähdekoodin maksuttoman lukemisen, käytön ja muokkaamisen sekä edelleen muokatun lähdekoodin käytön ja edelleenlevitysoikeuden. Päävaatimuksena on kuitenkin se, että lähdekoodi on yleisesti saatavilla ja vapaasti jaettavissa, koska ilman lähdekoodin yleistä saatavuutta voidaan ohjelmisto käytännössä lähes suoraan käsittää suljetun lähdekoodin piiriin kuuluvaksi.

Erityisesti internetin rooli yleisesti käytettävissä olevana, koko maapallon kattavana tietoverkkona, on monilta osiltaan mahdollistanut avoimen lähdekoodin lähestymistavan olemassaolon nykyisessä muodossaan. Siksi internet onkin nykyään erittäin tärkeässä roolissa avoimen lähdekoodin periaatteiden ylläpitämisessä ja levittämisessä. Avoimen lähdekoodin internet-portaalit kuten *FreshMeat.net* ja *SourceForge.net* ovat käytännössä kasvattaneet

avoimen lähdekoodin suosiota moninkertaiseksi mahdollistamalla keskitetyn levityksen. Kyseiset portaalit toimivat toisin sanoen avoimen lähdekoodin projektien hallinnointi- ja levityskeskuksina. Esimerkiksi sekä Capiluppi ym. (2003) että Spinellis ja Szyperski (2004) ovat suorittamissaan tutkimuksissa käyttäneet hyväkseen kyseisten portaalien tarjoamien avointen lähdekoodien projektien yksityiskohtaisia tietoja.

Yleisen saatavuuden ja vapaan levitettävyyden periaate on voimakkaasti markkinatalouden perusperiaatteiden vastainen ja ei siksi ensisilmäyksellä näyttäisi soveltuvan yhdenkään perinteikkään ohjelmistotalon liiketoimintalogiikaksi. Tämä onkin suurin periaatteellinen ero suljetun ja avoimen lähdekoodin lähestymistapojen välillä, ja ei selvästikään lievennä kahdenvälistä konfliktitilannetta. Onkin kysyttävä, mitä saatavuuden periaatteella voidaan saavuttaa ja miksi sen kannatus kaikesta huolimatta näyttää kasvavan enemmissä määrin. Erityisesti yrityksille voimakkain syy näyttäisi olevan raha, sillä erityispiirteistään riippumatta avoin lähdekoodi on lähtökohdiltaan (tai perusperiaatteiltaan) ilmaista. Toinen syy on yksittäisten henkilöiden motivaatio ja halu tiedon yleiseen jakamiseen tai muihin omien henkilökohtaisten tarpeiden täyttämisiin. Kolmantena syynä on selvä tarve halutunlaisille ohjelmistoille.

Avoimen lähdekoodin yleinen saatavuus saattaa aiheuttaa erään huomattavan käytännön ongelman. Ongelma konkretisoituu tietoturvaan liittyvien yksityiskohtaisten algoritmien toteutusmenetelmien ja käytettyjen toteutustapojen vapaaseen saatavuuteen. Avoin lähdekoodi voi tietyissä arkaluontoisissa toteutuksissa olla rikollisten tahojen saatavilla, jotka sitten helposti pystyvät tunnistamaan lähdekoodista ohjelmiston haavoittuvuuspiirteet, ja myöhemmin myös hyväksikäyttämään tietojaan vääriin tarkoituksiin. (Spinellis & Szyperski 2004, 31)

3.1.2 Lisensointi

Lisensointi (*engl. licensing*) on läheisesti saatavuuteen liittyvä ominaisuus, jonka voidaan sanoa käsittävän kaikki lakispesifiset ulottuvuudet avoimen lähdekoodin ohjelmiston osalta. Esimerkiksi suljetun lähdekoodin ohjelmistot ovat huomattavasti rajoitetummin lisensoidut kuin avoimen lähdekoodin ohjelmistot. Voidaankin todeta, että avoimen lähdekoodin ohjelmistojen kohdalla ohjelmistojen lisenssien tarkoitus on tukea saatavuuden periaatetta lain kannalta. Jotta ohjelmistoa voidaan ylipäätään kutsua avoimen lähdekoodin ohjelmistoksi, on sen käytännössä mukauduttava jonkin avoimen lähdekoodin ohjelmiston identifioivan lisenssin periaatteisiin.

Ohjelmistolisenssit ovat sopimusvelvoitteisia asiakirjoja, jotka määräävät millaisia tekijänoikeusoikeuksia käytetään ja millaisia patenttioikeuksia annetaan edelleenkäytettäväksi (Välimäki 2005, 4). Lähdekoodi ei perinteisesti yleensä kuulu jaettavaksi resurssiksi suljetun lähdekoodin lisensseissä. Ohjelmistot toimitetaankin perinteisesti vain binäärimuodossa olevina, dokumentaation ollessa ainoa todiste ohjelmiston oikeasta toiminnasta. Tämän lisäksi lisensseillä rajoitetaan hyvin usein käänteistekniikoiden hyväksikäyttöä. (Välimäki 2005, 29) Lähdekoodin saatavuus tietyille valikoiduille suljetun lähdekoodin asiakkaille on toki usein mahdollista, tietyin erityisehdoin.

Avoimen lähdekoodin ohjelmistolisenssin tehtävänä on vapauttaa ohjelmiston lähdekoodi lakien halutuista suojarajoitteista tietyin valituin periaattein. Avoimen lähdekoodin tunnetuimpia lisenssejä ovat *GPL* (*GNU General Public License*), *LGPL* (*GNU Lesser General Public License*), *BSD* (*BSD License*) ja *MIT* (*MIT License*). On kuitenkin tärkeää huomioida, etteivät kaikki lisenssit mahdollista suoraan esimerkiksi lähdekoodin vapaata siirtämistä tai muokattavuutta edelleentuotteistettaviksi suljetun lähdekoodin ohjelmistoiksi. Tämän vuoksi on erittäin tärkeää tunnistaa käytetyn lisenssin suomat oikeudet ja toisaalta sen ohjelmistolle asettamat rajoitukset riittävän ajoissa mahdollisten ongelmatilanteiden välttämiseksi. Eräänä yksityiskohtana Spinellis ja Szyperski

(2004, 30) ovat huomauttaneet, että tietyt avoimen lähdekoodin lisenssit sanelevat muun muassa sen, minkä lisenssin puitteissa edelleenmuokattua ohjelmistoa täytyy jatkossa levittää.

Lisensointi on osaltaan ongelmallinen avoimen lähdekoodin kannalta etenkin, kun avoimen lähdekoodin projektin omistajuus ei useinkaan ole yrityksen tasoisten tahojen hallinnassa. Näin projektilla ei ole rahallisia varoja käytössä, eikä projektin hallinto voi toisaalta puolustaa tuotostaan lain koko mittaan. Muita ongelmia tuottavat muun muassa ohjelmistopatentit. Kuten Välimäki (2004, 523) viittaa, on ohjelmistopatenttien nähty uhkaavan innovaatioita ja näin ollen niiden on nähty myös uhkaavan avoimen lähdekoodin lähestymistavan toiminnallisuutta. Muutenkin avoimen ohjelmiston etiikka ja filosofia ovat voimakkaasti kaikenlaisia patentteja vastaan, muistuttaa Välimäki (2004).

GPL- ja LGPL -lisenssit eivät tunnusta sellaista ohjelmiston kehitystä, joka vaatisi minkäänlaisten lisenssimaksujen hyväksymistä olemassaolevista patenteista. Mikäli siis patentti jollekin innovaatiolle on olemassa, ei ohjelmistonkehitystä pystytä enää kyseisten lisenssien ehtojen mukaan jatkamaan. Avoimen lähdekoodin mukainen ohjelmistotuotanto tuleekin ongelmalliseksi, mikäli patenttien määrä kasvaa kasvamistaan (Välimäki 2004, 525). Ehkä kuitenkin kaikista haluttavin toiminnallisuus juuri GPL -lisenssissä on se, että se kontrolloi ohjelmiston edelleenlevittämistä ja johdettuja töitä. Lisenssi nimenomaan vaatii sen, ettei lisensoitujen ohjelmistojen alkuperäisiä lisenssiehtoja voida jatkossa muuttaa. Muulloin ohjelmiston edelleenlevittäminen ja muokkaaminen on kiellettyä (Välimäki 2005, 130). Ehkä juuri siksi, kuten Capiluppi ym. (2003, 321) ovat tutkimuksessaan huomauttaneet, näyttäisi GPL olevan kaikista lisensseistä ylivoimaisesti suosituimmassa asemassa.

3.1.3 Kehityksen luonne

Avoimen lähdekoodin ohjelmiston kehityksen luonteen voidaan sanoa olevan ainakin osittain erilainen suljetun lähdekoodin ohjelmistojen kehitykseen verrattuna. Tämä näkyykin osittain DiBonan ym. (1999, 7) esittämästä mielenkiintoisesta väitteestä, jonka mukaan avoimen lähdekoodin mukainen ohjelmistotuotanto on jatke tieteellisestä toiminnasta. Erityisesti he ovat nähneet tiedon todentamisen näiden kahden yhteiseksi yhtymäkohdaksi. Eli kun jokin tieto, kuten hypoteesi tai tutkimustulos, on tutkijoiden joukossa käsiteltävänä, voivat muut tutkijat sen joko todentaa tai hylätä. (DiBona ym. 1999, 7)

Perinteisesti avoimen lähdekoodin ohjelmistot on tuotettu hajautetusti, lähinnä yhteisöllisinä projekteina. Ohjelmistojen kehittämisessä käytetyt menetelmät ovat syntyneet suurilta osin käytäntöjen tuloksina. Vaikka jotkin ohjelmistotalot ovatkin jo ottaneet osaa avoimen lähdekoodin ohjelmistojen kehittämiseen, on yleinen käsitys vielä kuitenkin se, että avointa lähdekoodia kehittävät ja tuottavat vain tiettyihin projekteihin sitoutuneet kehittäjät. Lisäksi kehittäjien ajatellaan elävän hajautetusti eri puolilla maapalloa ja käyttävän internetiä keskitettynä hallinnointijärjestelmänään (yhteydenpito- ja jakelukanava). Yleisten uskomusten mukaan heidän ei perinteisesti ole käsitetty saavan rahallista korvausta tekemästään työstä, minkä vuoksi kehittäjillä onkin useimmiten varsinainen (tai ”pakollinen”) päivätyö. Kehittäjillä on lisäksi käsitetty olevan keskimääräistä ohjelmiston kehittäjää paremmat taidot.

Vaikka yritysten kiinnostus avointa lähdekoodia kohtaan on siis pitkän aikaa ollut pintapuolista, niin voi avoin lähdekoodi kuitenkin olla helppo valttikortti tietyille yrityksille. Kuten Mockus ym. (2002, 310) ovat todenneet, kaikki avoimen lähdekoodin projektien toimijoiden joukot eivät enää nykyään koostu pelkistä ”ilmaisista” vapaaehtoisista, vaan voivat sisältää myös yritysten työntekijöitä. Avoin lähdekoodi on saanut ilmaista, ja toisaalta myös erittäin positiivista huomiota uutismediassa, joka luonnollisesti on lisännyt avoimen lähdekoodin yleistä haluttavuutta. Yleinen kiinnostuneisuus pohjautunee

kuitenkin vain näennäiseen ohjelmistojen ja kehittäjien ilmaisuuteen, ei niinkään itse kehitysprosesseihin tai niiden eri vaiheisiin.

3.1.3.1 Yhteisö ja kehittäjät

Avoimen lähdekoodin projekteja identifioi parhaiten yhteisöllisyys. Yhteisöt perustuvat useimmiten tiivistetysti aina jonkin tietyn projektin piiriin, vaikka internet-portaalit toimivat tavallaan myös laajemman yhteisön roolissa. Muun muassa Gacek ja Arief (2004, 36) ovat huomioineet, että avoimen lähdekoodin yhteisöt ovat yleensä tarkasti rajattuja yhteisöjä, joita yhdistävät etenkin yhtäläiset mielenkiinnon kohteet. Heidän mukaansa yhteisön rakenne ei kuitenkaan yleensä ole kovin selkeä. Jotkin projektit voivat toisaalta koostua vain yhdestä kehittäjästä, jolloin varsinaista projektiyhteisöä ei luonnollisestikaan muodostu. Toisaalta joissain yhteisöissä on hyvin tarkka hierarkia. Varsinkin näissä tarkan hierarkian yhteisöissä on valta projektin tärkeistä päätöksistä keskittynyt niin sanottujen *ydinkehittäjien* (*engl. core developers*) vastuulle *avustavien kehittäjien* (*engl. codevelopers*) kustannuksella. (Gacek & Arief 2004, 36)

Toisaalta sellaisissa yhteisöissä, joissa hierarkia on löyhempi, on päätöksenteko myös hajautetumpaa, joka joissain tapauksissa ilmenee jopa yhteisymmärryksenä. On ymmärrettävä, ettei yksittäisen kehittäjän rooli projektin elinaikana ole muuttumaton, vaan toimijoiden roolit vaihtuvat projektin aikana. (Gacek & Arief 2004, 36) Eräänä käytännön esimerkkinä voidaankin mainita avustavan kehittäjän vaikutusvallan kasvu saavutustensa myötä ja lopulta pääsy projektin ydinkehittäjäksi.

Avoimen lähdekoodin projektin ydin koostuu yleensä vain muutamasta avainhenkilöstä, joita avoimen lähdekoodin liikkeen eräs tunnettu puolestapuhuja E. Raymond (2001) luonnehtii *koordinoijiksi* (*engl. coordinator*). Koordinoijien rooli on useimmiten sama kuin ydinkehittäjien rooli. Heidän ensisijaisena tehtävänä on pitää projekti ”kasassa”. Eli toisin sanoen sekä

koordinoida projektin rakennetta ja yhteisöä että pitää huolta myös kaikista projektin tuotosten osa-alueista. On lisäksi mahdollista, että projektilla on vain yksi päävastuullinen henkilö, *johtaja* (*engl. leader*), joka jakaa ja delegoi projektin tuotosten eri tehtäväkokonaisuudet muille vähempivaltaisille pääkehittäjille. Tästä tunnetuimpana esimerkkinä voidaan mainita Linux, jossa L. Torvalds on toiminut projektin päävastuullisen roolissa. Koordinoijan rooli on projektissa erittäin huomattava, sillä hänen työssään onnistuminen voi yksistään johtaa projektin kaatumiseen tai onnistumiseen.

Olennaista on se, että koordinoija pystyy tunnistamaan hyvät suunnitteluideat toisistaan, vaikkei koordinoijan itse välttämättä tarvitsekaan olla erinomainen suunnittelija. Tämän lisäksi yksinkertaisuuden periaatteen noudattaminen on tärkeää, ja etenkin siinä suhteessa, että kaikkien valittujen toteutusten on oltava toimivia eikä ”yltiönerokkaita”. Erityinen vaatimus syntyy myös sille, että koordinoijan täytyy olla ihmissuhdetaitoinen, mutta erityisesti kuitenkin kommunikointitaitoinen. Näin on etenkin siksi, että iso osa koordinoijan työstä on juuri ihmisten ohjaamista ja heidän kanssaan vuorovaikuttamista, muttei kuitenkaan motivoimista! (Raymond 2001, 47-49)

Motivaatio syntyy aina havaitusta tarpeesta riippumatta siitä, missä roolissa kehittäjä toimii. Siksi on periaatteessa sama, onko kehittäjä yksilö vai yritys. Toisilla kehittäjillä on syviä filosofisia periaatteita ohjelmistojen avoimuuteen liittyen, kun taas toiset eivät ole kiinnostuneita sellaisista asioista. Yritysten motiivit liittyvät lähes poikkeuksetta rahaan – esimerkiksi haluun ja tarpeeseen saavuttaa nopeita voittoja. (Gacek & Arief 2004, 36) Kehittäjien korkea motivaatio ja henkilökohtaiset taidot ovat avoimen lähdekoodin lähestymistavan mukaisessa ohjelmistotuotannossa huomattavan tärkeässä asemassa, kuten muun muassa Samoladas ym. (2004, 86) kuvaavat. Kehittäjien taidot ovatkin paras keino saavuttaa kollektiivista suosiota ja hyväksyntää toisten kehittäjien ja jopa käyttäjien silmissä.

3.1.3.2 Kehityksen piirteet

Kehittäjien tehtäviä pyritään usein jakamaan siten, että tietyt kehittäjät keskittyvät tiettyjen ohjelmiston osien toteuttamiseen tai korjaamiseen, jotta mahdollisilta päällekkäisyyksiltä vältyttäisiin. On tosin täysin mahdollista, että projektit saavat kehityksensä aikana useita lähdekoodisyötteitä samoihin tehtäviin liittyen, eli saadaan siis toisistaan poikkeavia toteutuksia käytännössä ohjelmiston saman paikan täyttävistä osista. Tällöin ydinkehittäjien on valittava, mikä toteutus on kuhunkin paikkaan sopivin. Toisaalta ei ole myöskään epätavanomaista, että avoimen lähdekoodin ohjelmistoja kehitetään täysin (tai osittain) pelkästään intuition pohjalta. Koordinoijat lopulta päättävät sen, mitä kaikkia ominaisuuksia ohjelmiston piiriin hyväksytään, jolloin kaikkea projektin keston aikana tuotettua toiminnallisuutta ei luonnollisestikaan löydy lopullisesta ohjelmistosta (Samoladas ym. 2004, 84). Konfiguraationhallinnan osa-alueena tätä hyväksyntäprosessia voidaan avoimen lähdekoodin osalta kutsua *muutostenhallinnaksi* (*engl. change management*). Asklund ja Bendix (2002, 42) ovat esittäneet, että avoimen lähdekoodin mukaisen ohjelmistotuotannon muutostenhallinta poikkeaa merkittävästi suljetun lähdekoodin muutostenhallinnasta.

Asklund ja Bendix (2002, 44) esittävät, että avoimen lähdekoodin mukainen ohjelmistotuotanto ohjaa sellaiseen läpinäkyvään kehitykseen, jossa ”heikoimpien lenkkien” identifiointi on helppoa. Tämä synnyttää heidän mukaansa jokaiselle yksittäiselle kehittäjälle sosiaalisia paineita muun muassa ennaltamääriteltyjen prosessien sekä projektin suuntaviivojen noudattamiseen. Tutkijat väittävätkin, että juuri kyseinen seikka vähentää *tietovaraston* (*engl. repository*) ränsistymisen riskejä. Tämä on kriittistä, sillä tietovarasto on lähdekoodien varasto, jonka täyttyminen esimerkiksi huonolaatuisista toteutuksista johtaa hyvin helposti kaaokseen. Onneksi lähdekoodivaraston jopa vaistomainen suojeleminen on siis yksi tärkeä avoimen lähdekoodin mukaisen ohjelmistotuotannon erityispiirre. Erääksi vertailukohdaksi voidaankin esittää suljetun lähdekoodin projektin tietovaraston kontrollointia,

joka on vähemmän valvottua muun muassa vertaisarviointien vähyyden vuoksi. Toki lähdekoodien katselmoinnit ovat suljetun lähdekoodin kehitykselle ominaisia, mutta tällöinkään ei vertaisarviointi ole riittävän kattavaa. (Asklund & Bendix 2002, 45)

Raymond (2001, 116) on esittänyt mielenkiintoisen väitteen, jonka mukaan avoimen lähdekoodin mukainen ohjelmistotuotanto perustuu niin sanottuun *lahjakulttuuriin* (*engl. gift-culture*), jossa osanottajat kilpailevat suosiosta ja arvoasemasta antamalla asioita pois. Hän on jopa sitä mieltä, että kehittäjien keskinäinen kilpailu on hiljainen voimavara (Raymond 2001, 116). Kehittäjillä on siis selvästi erinäisiä motiiveja esittää henkilökohtaista osaamistaan, ja osallistuminen avoimen lähdekoodin projekteihin on yksi sellainen tapa. Toisaalta avoimen lähdekoodin projektin yhteisöön kuuluvien yksittäisten jäsenten taitojen, roolien ja aikaansaannosten tunnustukset motivoivat myös muita osallisia parantamaan työnsä laatua (Gacek & Arief 2004, 36). Näin ansioituneiden kehittäjien, tai jopa osallistuvien ohjelmistotalojen, asema voi kasvaa sekä projektien että mahdollisesti myös projekteja suurempien yhteisöjen sisällä. Onhan luonnollista se, että kun kehittäjä pääsee toteuttamaan itseään, hän myös kokee työn tekemisen iloa! Vapaus tehdä omaehtoisia päätöksiä ja motivaatio työtään kohtaan on kriittisessä roolissa missä tahansa työssä, jolloin ihminen varmasti pystyy parhaaseen mahdolliseen työhön omien taitojensa puitteissa. Tämä vapaus on ehdottomasti yksi huomattavimmista avoimen lähdekoodin mukaisen ohjelmistotuotannon tuomista eduista.

Ohjelmistojen kehittäjät ja käyttäjät toimivat eri rooleissa. Kehittäjät katsovat ohjelmistoa sisältä-ulospäin, kun käyttäjät puolestaan katsovat ohjelmistoa ulkoa-sisällepäin. Tämä piirre korostuu suljetun lähdekoodin ohjelmistoissa, joissa kehittäjien ja testaajien roolit ovat hyvin usein muuttumattomat. Avoimen lähdekoodin lähestymistavassa kehittäjien ja käyttäjien vuorovaikutus sekä roolien epämääräisyys parantavat virheiden korjausten helppoutta ja nopeutta. (Raymond 2001, 33-34) Vaikka avoimen lähdekoodin projekteissa kehittäjien interaktio on vähäistä ja kehittäjien työ on jaettu

itsenäisiksi toisistaan riippumattomiksi osakokonaisuuksiksi, ei kehittäjien suuri määrä (tai etenkin suuri lisääntyminen) johda kaaokseen kuten kuuluisa Brooksin laki esittää. Vika voi esiintyä myös useampana kuin yhtenä virheenä, mutta usean testaajan joukko pystyy tällaiset hankalat tilanteet nopeahkosti selvittämään. (Raymond 2001, 35)

3.1.3.3 Avoimen lähdekoodin mukainen ohjelmistotuotanto

P. Vixie (DiBona ym. 1999, 97-99) on tarkahkosti kuvannut avoimen lähdekoodin mukaista ohjelmistotuotantoa seuraavanlaisesti. Järjestelmätason suunnittelu puuttuu hyvin monista avoimen lähdekoodin projekteista, mutta mikäli sellainen on olemassa, on suunnitelma useimmiten vain implisiittisenä tietona. Tämän lisäksi yksityiskohtainen suunnittelu on puutteellista, ja syntyy vain toteutusten sivutuotteina. Toisaalta toteutus on hyvin kriittisessä ja korostuneessa asemassa. Testaus – kuten esimerkiksi yksikkö-, regressio-, järjestelmätestaus ja ynnä muut testaukseen kuuluvat osa-alueet – on erittäin vähäistä ennen tuotoksen julkaisua. Kentällä testaus julkaisun jälkeen on kuitenkin erittäin kattavaa. Ylläpito ei ole järjestäytynyttä täydellistä toimintaa vaan tarpeeseen perustuvaa. (Vixie & DiBona ym. 1999, 97-99)

Avoimen lähdekoodin mukainen ohjelmistotuotanto vaatii tukitoimia siinä missä muukin ohjelmistotuotanto. Erityiset huomiokohdat (muun muassa kehittäjien hajautuneisuuden vuoksi) liittyvät ohjelmiston modulaarisuuteen, arkkitehtuurin näkyvyyteen, dokumentaatioon ja testaukseen, hyväksymiskäytäntöihin ja toisaalta myös työvälineisiin (Gacek & Arief 2004, 37). Modularisuuden näkökohtaan yhtyy myös muun muassa L. Torvalds (DiBona ym. 1999, 108-109) esittämällä seuraavan aiheellisen huomion: ”Avoimen lähdekoodin mukainen ohjelmistotuotanto jopa suoranaisesti vaatii modulaarisuutta, jotta samanaikainen hajautettu työskentely olisi ylipäätään mitenkään mahdollista”.

Ohjelmiston *modularisointi* (*engl. modularization*) auttaa muun muassa suunnittelun paremmassa ymmärtämisessä, ja toteutuksen jakamisessa osiin kullekin kehittäjälle. Hyvin kuvatut *rajapinnat* (*engl. interface*) ja lähdekoodin modularisointi ovat esivaatimus tehokkaalle hajautetulle yhteistoiminnalle. Ohjelmistoarkkitehtuuri kuvaa ohjelmiston rakenteen, kuten komponentit, niiden ominaisuudet ja niiden väliset suhteet. Hyvä dokumentaatio auttaa kehittäjiä ymmärtämään ja muokkaamaan ohjelmistoa. Huolellinen testaus puolestaan antaa käyttäjille luottamuksen ohjelmiston oikeasta toimivuudesta. Dokumentaation ja testauksen merkitystä usein vähätellään avoimen lähdekoodin projekteissa, sillä kehittäjät mieltävät lähdekoodin parhaaksi mahdolliseksi dokumentaatioksi. Lisäksi usean kehittäjän osallistuminen esimerkiksi saman moduulin tai komponentin kehittämiseen todentaa lähdekoodin usean ”silmäparin” ansiosta. (Gacek & Arief 2004, 37)

Avoimen lähdekoodin projektit saavat vikaraportteja ja lähdekoodisyötteitä eri lähteistä. Hyväksymiskäytännöt koostuvat kolmesta osa-alueesta. Ensimmäisenä osa-alueena on työskentelyalueen valinta. On mahdollista ottaa vastaan joko vain haluttuja alueita koskevia toteutuksia tai toisaalta spontaaneja ja innovatiivisia toteutuksia. Toisena osa-alueena on päätöksenteko, joka perustuu neljään eri ulottuvuuteen: toteutuksen laatuun, hyväksymiskriteereihin, päättävien henkilöiden kognitiivisiin taitoihin ja projektin sosiaaliseen rakenteeseen. Kaikki nämä dimensiot vaikuttavat siihen, mihin suuntaan projektia halutaan kehittää, eli mitä projektin tarkoituksiin sopivia syötteitä valitaan kulloinkin käytettäväksi. Kolmantena osa-alueena on syötteiden tietojen oikeaoppinen pilkkominen ja sisäistäminen. (Gacek & Arief 2004, 38)

Mockus ym. (2002, 310) ovat huomioineet, että muun muassa projektisuunnitelmat, aikataulut, toteutustehtävät ja -vaiheet eivät avoimen lähdekoodin projekteissa ole alussa yleensä määriteltyinä, vaan ne muotoutuvat havaitusta käytännön tarpeesta. Tämän lisäksi työtehtäviä ei myöskään jaeta

vain tietyille valikoiduille kehittäjille, vaan jokaisen kehittäjän on mahdollista valita omaan osaamisalueeseensa kuuluvat tehtävät (Mockus ym. 2002, 310).

Myös Spinellis ja Szyperski (2004, 32) ovat tehneet samansuuntaisen huomion: ”Kun organisaatioissa otetaan käyttöön avoimen lähdekoodin mukaisen ohjelmistotuotannon periaatteita, niin muutos helposti tapahtuu suunnitelmien kustannuksella”. Esimerkkeinä he esittävät strategisen suunnittelun, yksityiskohtaisten vaatimusten määrittelyn, testauksen ja tuen organisoinnin (Spinellis & Szyperski 2004, 32). Huomattavaa piirteissä on se, että järkevien suunnitelmien merkitys vain korostuu projektin koon kasvaessa, mutta tällainen impromptutyylinen läpivienti taas toisaalta mahdollistaa voimakkaasti reaktiivisen käyttäytymisen projektin edetessä.

Raymond (2001, 146) kuvaa avoimen lähdekoodin tuottavan parhaiten odotusten suhteen silloin, kun ohjelmistolta vaaditaan luotettavuutta, stabiiliutta, skaalautuvuutta sekä suunnittelun että toteutuksen paikkansapitävyyttä. Lisäksi hän kannustaa kaikkia sellaisia yrityksiä, joille jonkin tietyn ohjelmiston käyttö on liiketoiminnalle keskeisessä asemassa, mikäli kyseinen ohjelmisto perustuu yleisiin atk-periaatteisiin ja tiedonvälitysrakenteisiin, siirtymään avoimen lähdekoodin ohjelmiston käyttäjäksi. Hän perustelee kantansa sillä, että niin ei joudu yhden ohjelmistotalon ”armoille”, joten ohjelmiston käyttäytymisen hallinta on turvatumpaa. (Raymond 2001, 146)

Raymond (2001) on lisäksi esittänyt, että yhä kasvavassa määrin eri tahot huomaavat ohjelmistoalan painopisteen siirtyvän teknologiapainotteisesta informaatiopainotteiseen. Hänen näkemyksensä mukaan yritysten on siis tietyillä toimialoilla järkevämpää panostaa yhteen yhteiseen toimivaan järjestelmään, eikä niinkään useisiin kilpaileviin järjestelmiin. Näin tästä keskeisestä järjestelmästä saadaan huomattavasti parempilaatuinen ja toiminnallisempi kokonaisuus, jolloin yksittäisten yritysten liiketoiminnot voidaan piilottaa informaatioksi. (Raymond 2001) Tähän ajatukseen on yhtynyt

myös T. O'Reilly (DiBona ym. 1999, 195), jonka mukaan "*tieto-ohjelmistot*" (*engl. infoware*) kasvattavat merkitystään. Tämä tosin hänen mukaansa merkitsisi sitä, etteivät kehitetyt tekniset ratkaisut tulisi enää jatkossa olemaan informaatiota tärkeämmässä asemassa (O'Reilly & DiBona ym. 1999, 195).

3.1.4 Uudelleenkäyttö

Eräs huomattava etu avoimen lähdekoodin puolestapuhujaksi on mahdollisuus perustaa suunnittelu ja toteutus olemassa oleviin ohjelmistokomponentteihin. *Uudelleenkäyttö* (*engl. reuse*) käsittää ohjelmiston eri osa-alueiden, kuten sen komponenttien edelleenkäytön joko täysin uusissa tai jo uudelleenkäytetyissä ohjelmistoissa. Näin kaikkia välttämättömiä ohjelmiston osia ei aina ole tarve kehittää itse. Spinelliksen ja Szyperskin (2004, 29) huomioiden mukaan uudelleenkäytetty avoin lähdekoodi on yleensä laadukkaampaa kuin esimerkiksi toimeksiantona tehdyn koodin ensimmäinen ilmentymä. Lisäksi uudelleenkäytetyn komponentin tarjoama toiminnallisuus on heidän mukaansa usein paljon kattavampi kuin mihin toimeksiantona tehdyn komponentin tapauksessa useimmiten on rahallisesti varaa (Spinellis & Szyperski 2004, 29).

Avoimen lähdekoodin uudelleenkäyttö on hienovaraista, sillä uudelleenkäyttöä ei rajoita mitkään binäärimuodossa olevien komponenttien keinotekoiset rajoitteet. On siis täysin mahdollista käyttää uudelleen vain osia lähdekoodista esimerkiksi metodeja, luokkia, komponentteja tai muita sellaisia osa-alueita. Myös lähdekoodin muokkaaminen, parantaminen ja korjaaminen syventää uudelleenkäytettävän lähdekoodin integroitavuutta. Tämän lisäksi myös *kohdealustan* (*engl. target platform*) siirto on helpompaa. (Spinellis & Szyperski 2004, 29)

On kuitenkin täysin selvää, ettei läheskään kaikkea lähdekoodia pidä ja edes voi käyttää uudelleen. Yleinen käsitys on se, että uudelleenkäytettävän lähdekoodin tulisi aina olla mahdollisimman yleistettävää ja erityisesti suunniteltua uusiokäyttöön. Spinellis ja Szyperski (2004, 29) ovatkin

muistuttaneet perinteisestä ohjelmistokomponenttien uudelleenkäytön mallista, joka koostuu seuraavista kolmesta koordinaattiakselista:

1. Mitä käytetään uudelleen?
2. Miten käytetään uudelleen?
3. Missä käytetään uudelleen?

Uudelleenkäyttöön liittyy joukko mielenkiintoisia ongelmia, joita tulisi aina mahdollisista kiireistäkin riippumatta pohtia. Alkuperäisen projektin lähdekoodiin tehdyt tietyt räätälöidyt muutokset voivat synnyttää projektista eriäviä kehityspolkuja (Spinellis & Szyperski 2004, 29). Tämä johtaa helposti ongelmiin, etenkin mikäli esimerkiksi alkuperäiseen lähdekoodiin tehty teknologiamuutos täytyy siirtää myös projektista eriäviin kehityshaaroihin. Tällöin samankaltaisten mutta välttämättömien muutosten siirto aikaisemmin integroituihin tuotoksiin voi olla huomattavan vaikeaa. Schach, Jin ym. (2002) ovat huomioineet samoin: ”On siis mahdollista, että avoimen lähdekoodin ohjelmistot kehittyvät useamman projektin piirissä ja rinnakkainen kehitys on mahdollista”. Päähaarasta jakautuvat kehityshaarat voivat heidän näkemyksensä mukaan myöhemmin yhdistyä takaisin päähaaraan, tuhoutua tai jatkaa omaa ”rinnakkaiselämäänsä” (Schach, Jin ym. 2002). Tämä tosin helposti johtaa myös muunkaltaisiin yhteensopivuusongelmiin. Eräissä avoimen lähdekoodin projekteissa, kuten esimerkiksi Linuxissa, keskitytään jopa kahteen yhtäaikaisesti kehitettäviin niin sanottuihin *kokeellisiin* (*engl. development path*) ja *vakaisiin* (*engl. stable path*) kehityspolkuihin (Godfrey & Qiang 2000, 132).

Uudelleenkäytettävän lähdekoodin käyttöönotossa on mahdollista, että ohjelmistoon syntyy turhia kytkentöjä. Tämän lisäksi, kun uudelleenkäytettävien elementtien määrää ei osata karsia ja minimoida, jää ohjelmistoon helposti niin sanottua *kuollutta koodia* (*engl. dead code*). Kuollut koodi kuvaa kaikkia sellaisia lähdekoodin osia, joita ei koskaan suoriteta. Tämä

aiheuttaa turhaa rasitusta ohjelmiston ylläpidolle. Toisaalta tarkkaan määriteltyjen rajapintojen käyttö voi hankaloittaa ohjelmiston ylläpitoa synnyttämällä liian syviä riippuvuuksia. Näin käy esimerkiksi silloin, kun uudelleenkäytettävien komponenttien rajapinnat muuttuvat radikaalisti seuraavissa kehitysversioissa. Vaikka avoimen lähdekoodin mukaisen ohjelmistotuotannon nopean innovatiivisen luonteenpiirteen mukaisesti tämä ominaisuus korostuu, on hyvä huomata, ettei tämä ongelma liity pelkästään avoimen lähdekoodin mukaiseen ohjelmistotuotantoon, vaan sama haitta on havaittavissa myös binäärimuodossa olevien komponenttien kohdalla. (Spinellis & Szyperski 2004, 29-30)

Spinellis ja Szyperski (2004, 30) ovat lisäksi tunnistaneet avoimen lähdekoodin laadun vaihtelevan voimakkaasti: ”Toisten toteutusten ollessa täysin kehnoja, voivat toiset kuitenkin olla huippulaadukkaita”. He lisäksi muistuttavat, että avoimen lähdekoodin laadun verifiointi voi olla hankalahko prosessi, joka kattaa vähintään muun muassa lähdekoodin, vikaraporttien ja toteutuneen ylläpidon tutkimisen. Vaikka ylläpidettävyyttä ei siis osatakaan arvioida avoimen lähdekoodin projekteissa, niin näin ei myöskään suljetun lähdekoodin projekteissa useinkaan tapahdu. Perinteisesti laadunvarmistuksessa käytetään hyväksi lähinnä tuotteen testausta, suunnitelmien katselmointeja ja muita sellaisia menetelmiä, mutta ei esimerkiksi ylläpidettävyyden arviointia. (Spinellis & Szyperski 2004, 30)

3.2 Käytänteet

Oikeat työvälineet ovat missä tahansa ohjelmistokehityksessä tärkeässä roolissa, ja tämä pitää paikkansa myös avoimen lähdekoodin mukaisessa ohjelmistokehityksessä. Varsinaisten työvälineiden tehtävänä on mahdollistaa ohjelmistokehitys, kun avustavien työvälineiden tehtävänä on puolestaan helpottaa ohjelmistoprosessin hallintaa ja läpivientiä. Hankinta- ja käyttöhinnoiltaan kalliit kaupalliset tuotteet eivät käytännössä ole olleet avoimen lähdekoodin mukaiselle ohjelmistotuotannolle mahdollisia ja

käyttökelpoisia vaihtoehtoja. Tämän vuoksi avoimen lähdekoodin mukaisessa ohjelmistokehityksessä työvälineinä useimmiten käytetäänkin vain ilmaisia avoimen lähdekoodin ohjelmistoja.

Ohjelmointiympäristön tarjoamien työvälineiden lisäksi tärkeimpiä ovat ohjelmatiedostojen hallintajärjestelmä ja vikojen raportointijärjestelmä. Mockus ym. (2002, 313) ovat arvioineet kahden suuren avoimen lähdekoodin ohjelmiston kehitystä, Apachen ja Mozillan, ja ovat tunnistaneet niissä käytetyn seuraavia työvälineitä. *CVS* (*Concurrent Version Control*) on lähdekoodien versionhallintajärjestelmä, joka toimii myös konfiguraationhallinnan työvälineenä. *BugDB* (Apache) ja *Bugzilla* (Mozilla) ovat vikojen raportointijärjestelmiä. Avoimen lähdekoodin projektien piireissä käytetään erittäin usein hyväksi myös sähköpostilistoja (Mockus ym. 2002, 313). Sähköpostilistoilla ylläpidetään sekä vanhaa että uutta tietoa projektin eri kehityksen vaiheista ja ne ovat tarkoitettuja etenkin projektien kehittäjille, vaikka kaikki projektin tilasta kiinnostuneet voivat myös liittyä niille.

Työvälineistä erityisesti CVS on saavuttanut suurta suosiota. Se onkin käytössä lähes kaikissa avoimen lähdekoodin projekteissa. Suuri käyttöaste pohjautuu ohjelmiston monipuolisuuteen, taipuisuuteen ja toisaalta yksinkertaisuuteen. CVS:n suurin etu on juuri siinä, että se on pystynyt vastaamaan tehokkaasti hajautetun kehityksen malliin monipuolisilla toiminnoilla. (Bar & Fogel 2003, 11)

Capiluppin ym. (2003, 320) mukaan kaikenlaisia avoimen lähdekoodin ohjelmistoja kehitetään, suosituimpina kohdealueina ovat etenkin internet, tiedonvälitys ja multimedia. Gacek ja Arief (2004, 36) ovat tunnistaneet avoimen lähdekoodin projektien lähtökohtien vaihtelevan kolmessa pääpiirteessä. Projektit joko alkavat tyhjästä, perustuvat jo olemassaolevien kaupallisten ohjelmistojen edelleenkehittämistoimiin tai käynnistyvät suljetun lähdekoodin ohjelmistojen tutkimisella. Edelleen he ovat tunnistaneet kolmenlaista

liiketoimintamallia käytettävän avoimen lähdekoodin mukaisen ohjelmistotuotannon piirissä (Gacek & Arief 2004, 36):

1. Ohjelmiston omaehtoinen käyttö
2. Ohjelmiston myynti
3. Alusta kaupallisille tai kokeileville ohjelmistoille

Siinä missä avoimen lähdekoodin mukaisessa ohjelmistotuotannossa käytetyt menettelytavat vaihtelevat voimakkaasti projektikohtaisesti, on näin myös version- ja konfiguraationhallinnassa. Lisäksi muun muassa Linux, mielenkiintoista kyllä, poikkeaa muun muassa myös näissä osin valtaosasta muiden avoimen lähdekoodin projektien käytänteistä. (Asklund & Bendix 2002, 40)

Asklund ja Bendix (2002, 41-43) kuvaavat avoimen lähdekoodin projekteista tekemiään huomioita seuraavasti. Versionhallinnassa kaikkien tiedostojen versiot pidetään tietovarastossa ja erinäisillä mekanismeilla minimoidaan tilankäyttöä. Vaikka kirjoitusoikeuksia jaetaan usein avokätisesti, eivät lähdekooditiedostot siirry aikaisempiin versioihinsa kovin usein. Näiden lisäksi toiminnallinen lähdekoodi ja käyttöjärjestelmäkohtainen lähdekoodi erotellaan usein siten, että ohjelmistojen kehittäminen eri käyttöjärjestelmille on kohtuullisen vaivatonta. *Rakentamisen hallinta* (*engl. build management*) on keskitettyä, sillä kaikki käännöksissä tarvittavat lähdekooditiedostot sijaitsevat *paikallisissa työtiloissa* (*engl. local workspace*). Rakentamisessa käytetään lisäksi usein esikäännettyjä tiedostoja, jotta varsinaiset käännösajat pysyisivät järkevinä. *Konfiguraation valinta* (*engl. configuration selection*) perustuu siihen, että automaattisesti vain jokaisen tiedoston viimeisintä julkaisua ylläpidetään. Lisäksi kehityshaaroja harvoin käytetään samanaikaisiin töihin, vaan työt mieluummin jaetaan eri projekteiksi. Sekä *työtilojen hallinta* (*engl. workspace management*) että s*amanaikaisuuden hallinta* (*engl. concurrency control*) toteutetaan käytännössä lähes aina CVS ohjelmiston avulla. *Julkaisun hallintaa* (*engl. release*

management) ei varsinaisessa mielessä ole, vaan tiedostojen versiot jäädytetään aika ajoin niin sanotuiksi ”sisäisiksi julkaisuiksi”. (Asklund & Bendix 2002, 41-43)

Capiluppin ym. (2003, 321) tutkimuksessa esiintyy kiinnostava seikka: C-ohjelmointikieliperhe (sekä C- että C++-ohjelmointikielet) näyttäisi olevan suosituin toteutuskieliperhe avoimen lähdekoodin projekteissa. Esimerkiksi täysin alustariippumattomaksi suunniteltu Java -ohjelmointikieli seuraa C-kieliperhettä. Tämä käytännössä tarkoittaa sitä, että kohtalaisen suuri osa avoimen lähdekoodin ohjelmistoista näyttäisi olevan toteutettu vielä rakenteisella ohjelmointiparadigmalla, eikä niinkään olioparadigmalla!

Ohjelmiston koon suhteen ei itse asiassa käytännössä ole suurta merkitystä sillä, kuinka monta kehittäjää projektiin on sitoutunut. Kehittäjät ovat suuressa osassa projekteja *aktiivisia kehittäjiä* (*engl. stable developer*), ja vain muutamissa projekteissa esiintyy *liikkuvia kehittäjiä* (*engl. transient developer*). Kehittäjät ovatkin yleensä erittäin projektiuskollisia, vaikka yksittäisen kehittäjän rooli ja aktiviteetti vaihteleekin aika ajoin. (Capiluppi ym. 2003, 321)

Lisäksi Capiluppin ym. (2003, 320-321) tutkimuksessa avoimen lähdekoodin projekteista valtaosa (noin 80 prosenttia) oli pienikokoisia projekteja (alle 50 000 koodiriviä, noin 20 merkkiä per rivi), joita yksi tai kaksi kehittäjää kehittivät joko jatkuvasti tai aika-ajoin. Lisäksi heidän tutkimuksessaan avoimen lähdekoodin projekteissa oli yleensä kiinnitettynä vain yksi pysyvä aktiivinen kehittäjä. Tämän lisäksi projektit olivat useimmiten nuoria, vain alle kolmen vuoden ikäisiä. Tosin kuten tutkijat ovat itsekin tiedostaneet, heidän tutkimuksessaan käytetyn internet-portaalin tarkoituksena on erityisesti antaa uusille avoimen lähdekoodin projekteille näkyvyyttä. (Capiluppi ym. 2003, 320-321) On tietenkin selvää, ettei läheskään kaikkien projektien ympärille synny suurta suosiota, vähättelemättä yhtään projektin kohdealuetta tai rakennetta.

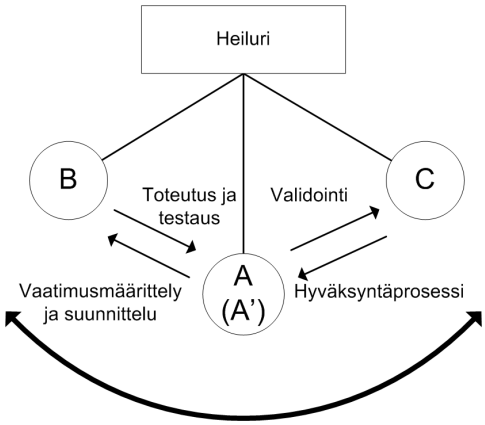
Mockuksen ym. (2002, 324) tutkimus esittää, että avoimen lähdekoodin projektit ovat ainakin yhtä tehokkaita ja tuottavia kuin suljetun lähdekoodin projektit, erityisesti kun vielä ottaa huomioon avoimen lähdekoodin kehitystyön osa-aikaisuuden ominaispiirteen. On vielä tosin epäselvää, missä laajuudessa itse kussakin projektissa tuotoksen laatua voidaan todentaa riittävän tiheällä aikajanalla.

Verrattuna avoimen lähdekoodin mukaista ohjelmistotuotantoa suljetun lähdekoodin mukaiseen ohjelmistotuotantoon, ovat avoimen lähdekoodin mukaisen ohjelmistotuotannon ”kilometripylväät” epämääräisiä (Potdar & Chang 2004, 107). Tämä merkitsee muun muassa sitä, ettei etukäteen suunniteltujen tehtävien ja saavutusetappien perusteella voi, ainakaan ulkopuolinen tarkkailija, tehdä pitkälle vieviä johtopäätöksiä ohjelmiston edistyksestä ja suunnasta.

Avoimen lähdekoodin mukainen ohjelmistotuotanto ei siis ole selkeästi vaiheistettua. Ohjelmointiparadigmojen perinteiset elinkaaren vaiheet esiintyvät avoimen lähdekoodin lähestymistavassa voimakkaasti iteratiivisina. Perinteiset määrittely- ja suunnitteluvaiheet sekä myös toteutus- ja testausvaiheet kuvautuvat avoimen lähdekoodin mukaisessa ohjelmistotuotannossa toisiaan seuraavina mutta myös päällekkäisinä toimenpiteinä, jotka jatkuvasti iteroivat aktiivisen kehityksen edetessä.

Potdar ja Chang (2004, 108-109) esittävät avoimen lähdekoodin mukaisen ohjelmistotuotannon kehityksen arviointiin erittäin mielenkiintoista mallia, jota kutsutaan kontrolloidun heilurin malliksi (kuvio kolme). Aluksi heiluri on lepotilassa liikkumattomana. Mutta kun jokin tietyn ohjelmiston osan toteutus tai korjaus aloitetaan, heiluri ”päästetään” heilumaan lepotilastaan (tila A), joka käynnistää vaatimusten ja suunnitelmien määrittelyn (tila B). Heiluri heilahtaa tästä toisesta ääripäästä toiseen ääripäähän (tila C), jossa itse toteutus-, testaus- ja todentamisosat instantioidaan. (Potdar & Chang 2004, 108-109) Heilurista tekee kontrolloidun se tosiseikka, että heilurin liike voidaan aloittaa ja

pysäyttää silloin, kun niin halutaan. Kyseinen malli toimii käsitteellisesti juuri kuten avoimen lähdekoodin mukainen ohjelmistotuotanto käytännössäkin näyttäisi toimivan.



KUVIO 3. Avoimen lähdekoodin mukaisen ohjelmistotuotannon heilurimalli (Potdar & Chang 2004, 108).

Samansuuntaisesti toteavat myös Samoladas ym. (2004, 84). Heidän mukaansa avoimen lähdekoodin projektien kehitys on iteratiivista, eli sama lähdekoodisyötteiden ja vikakorjausten käsittelyprosessi jatkuu pitkin avoimen ohjelmiston kehitystä. Rajaus näemmä toteutuu sovelluksen teknisen ylläpidon, dokumentaation ja kiireellisyyden pohjalta. Heidän käsityksensä mukaan jotkin tietyt sovellusalueet soveltuvat paremmin avoimen lähdekoodin mukaisen ohjelmistotuotannon piiriin – esimerkiksi yleiset alustat, kuten käyttöjärjestelmät tai internet-palvelimet. (Samoladas ym. 2004, 84)

Godfrey ja Qiang (2000, 133) huomauttavat aiheellisesti, että suunniteltu kehitys, testaus ja ennaltaehkäisevä ylläpito saattavat kärsiä, sillä avoin lähdekoodi rohkaisee mieluummin aktiiviseen osallistumiseen kuin välttämättä huolelliseen harkitsemiseen ja uudelleenorganisointiin. He jatkavat todeten, että lähdekoodin laatua ylläpidetään lähinnä voimakkaaseen rinnakkaiseen *virheenjäljitykseen* (*engl. debug*) kuin mihinkään systemaattiseen testaukseen tai

muunlaisiin ennalta suunniteltuihin ja määriteltyihin etenemistapoihin luottaen (Godfrey & Qiang 2000, 133).

Yhtenä poikkeuksellisena esimerkkitapauksena toimii jo aiemminkin mainittu Linux. Erityisesti kokeellisen kehityspolun mukaisten julkaisujen koot kasvavat voimakkaasti. Lisäksi Linux ei näytä noudattavan ohjelmiston kehityksen Lehmanin kolmatta lakia, jonka mukaan järjestelmän elinkaaren aikana tehtävät saavutukset vakiintuvat tietylle tasolle. Saavutusten määrä onkin tasaisesti kasvanut koko Linuxin elinkaaren ajan. (Godfrey & Qiang 2000, 134) Lehmanin ym. (1998, 217) luokittelun mukaisesti kyseinen laki on Lehmanin viides laki.

3.3 Saavutettavat edut

Mitä etuja avoimen lähdekoodin mukaisen ohjelmistotuotannon kautta voidaan saavuttaa? Selvästikin nopea innovatiivinen kehitys on juuri omiaan tälle kehitystavalle. Lisäksi lähdekoodin avoimuus, vapaus ja se seikka, että lähdekoodi on periaatteessa ilmaista, ovat varsin houkuttelevia näkökulmia jopa perinteiseen ohjelmistojen kaupallistamisen näkökulmaan verrattuna. Vapaus muokata ja edelleenkehittää lähdekoodia omien tarpeidensa mukaan on erittäin asiakasystävällinen näkökulma tuottajapainotteisessa nyky-yhteiskunnassa.

Tämän lisäksi avoimen lähdekoodin kautta on mahdollisuus saavuttaa suurta positiivishenkistä suosiota, sekä toisaalta houkutella projektille suuri joukko motivoituneita ja kehityshaluisia kehittäjiä. Näin projektin resursseista saadaan helposti paljon dynaamisempi kokonaisuus kuin esimerkiksi suljetun lähdekoodin resurssien kohdalla. Yhteisön tuomiin etuihin kuuluu esimerkiksi tehokas arviointi- ja aktiivinen kenttätestaushalu. Lisäksi on mahdollista saada muun muassa todella laadukkaita lähdekoodisyötteitä. Parhaassa tapauksessa on aina mahdollista valita tiettyyn tilanteeseen sopivin toteutus, eikä vain kuten useimmiten on tehty: ensimmäinen keksitty käytännön ratkaisu.

Itse projektiin osallistuville kehittäjille avoimen lähdekoodin lähestymistapa tarkoittaa vapautta valita tekemisensä omien taitojensa ja motivaatiotasojensa mukaan. Kehitystä ei siis ole rajoitettu keinotekoisesti. Koska lähestymistapa lisäksi kannustaa oppimiseen, niin osaavammat kehittäjät useimmiten avustavat vähemmän kokeneita henkilöitä. Lisäksi kehittäjien taitojen on käsitetty olevan keskimääräisiä ohjelmistonkehittäjiä paremmat. Myös kehitysprosessin läpinäkyvyys tukee oppimista.

Teknologian hallinta ja toisaalta sen läpinäkyvyys ovat avoimen lähdekoodin mukaiselle ohjelmistotuotannolle tyypillistä. Käytännössä näkökulmat poistavat tarpeen vaihtoehtoisille ohjelmistoille, joten yksi tietty ohjelmisto voi toimia alustana myöhemmälle yhteiselle kehitykselle. Tällöin jo pelkän tilannekohtaisen informaatiovuon ohjelmoinnin varaan on mahdollista siirtää osa yrityksen liiketoiminnasta.

Ehkä parhaimman kuvan avoimen lähdekoodin mukaisen ohjelmistotuotannon voimasta antaa kuitenkin L. Torvalds (DiBona ym. 1999, 108-109), joka on esittänyt väitteen: ”Linuxin voima tulee yhtäältä sekä sen yhteisön yhteistyöstä että myös sen lähdekoodin laadusta”.

3.4 Syntyvät haitat

Mitä mahdollisia haittoja avoin lähdekoodi voi puolestaan aiheuttaa? Ongelmia selvästi tuottaa se, että avoimen lähdekoodin lähestymistapaa on vaikea ymmärtää liiketoimintastrategisesti. Ensinäkymältä onkin vaikeaa ymmärtää lähestymistavan tuomien etujen ja avoimuuden näkökulman yhteensovittamista. Tämän lisäksi ongelmia aiheuttavat monet eri lisenssivaihtoehdot ja mahdolliset suoranaiset patenttiongelmat, joita monet olemassa olevat lisenssit vielä hankaloittavat.

Toisaalta projektien hallinnointi on usein vain välttävällä tasolla ja tarkkojen suunnitelmien taikka aikarajojen puuttuminen aiheuttanee päänvaivaa monelle eri taholle, varsinkin projektin ollessa isokokoinen. Koska avoimen lähdekoodin

mukainen ohjelmistotuotanto panostaa lisäksi erityisesti toteutukseen ja testaukseen, jäävät monen kehittäjän mielestä toissijaiset tehtävät, kuten dokumentointi, puutteellisiksi. Dokumentoinnin puute johtuu yleensä siitä, että kehittäjät helposti kuvittelevat, että dokumentaation tarvetta ei ole, koska lähdekoodi on itsensä selittävää. Kuten esimerkiksi Capiluppi ym. (2003, 324) ovat tunnistaneet seuraavasti: ainakin projektien pienuus korreloi hyvin usein huonon ja puutteellisen dokumentaation kanssa. Toisaalta tämänkaltaiset ongelmat ovat kuitenkin helposti kierrettävissä esimerkiksi palkkaamalla projektille työntekijöitä tekemään juuri näitä kyseisiä ”ongelmallisia” töitä.

On toki mahdollista ettei projekti saavuta tarpeellista määrää suosiota, jolloin myös yhteisön koko jää helposti pieneksi. Yhteisöön liittyy myös muitakin ongelmia. Etenkin työntekijöiden hajautuneisuus ei perinteistä suljetun lähdekoodin näkökulmaa kannattavien joukossa ole suuressa suosiossa, sillä tällöin työntekijöiden suhteen ei enää ole suoraa kontrollia. Lisäksi muuna ongelmakohtana voi olla kehittäjien keskinäinen kilpailu. Toki kilpailu voi parhaassa tapauksessa motivoida kehittäjiä parantamaan työn laatuaan, mutta toisaalta se voi helposti aiheuttaa konfliktitilanteita eri henkilöiden välillä.

Fitzgerald (2004, 92) on huomauttanut aiheellisesti, että avoimen lähdekoodin aatteen levitessä yhä laajemmalle, myös kehittäjien keskimääräiset taidot välttämättä vähenevät. Tämä tuottaa hänen mukaansa poikkeuksetta ongelmia tulevaisuudessa tuotettaviin ohjelmistoihin, sillä lähdekoodin laatu varmasti kärsii asiasta. Asia on jo osaltaan johtanut muun muassa kehittäjien hajaantumisiin moniin sellaisiin pienikokoisiin projekteihin, joiden tulevaisuudennäkymät eivät näytä toiveikkailta, Fitzgerald (2004, 92) toteaa.

Ongelmia voivat aiheuttaa myös sellaiset asiat kuin ylläpito, käyttövarmuus, pitkäikäisyys ja tietoturva. Ylläpito ei useinkaan ole järjestelmällistä ja koordinoitua toimintaa, vaan pelkästään tarpeeseen perustuvaa toimintaa. Koska käyttövarmuudesta ja pitkäikäisyydestä ei ole järjestelmällisiä tutkimustuloksia, on niiden arviointi hankalaa. Lisäksi tietoturvan kohdealueen

ohjelmistoja ei avoimen lähdekoodin projekteina pidä aktiivisesti lähteä tuottamaan, koska muun muassa tiettyjen salausalgoritmien yksityiskohtaisten kuvausten pääsy ”vääriin käsiin” aiheuttaisi huomattavia ongelmia.

Käytettävyyden suhteen avoimen lähdekoodin ohjelmistot voivat useimmiten olla teknis- ja ammattilaispainotteisia. Näin on lähinnä siksi, että kehittäjät ovat itse myös pääasiallisia käyttäjiä. Nichols ja Twidale (2002, 1-3) kuvaavat, että rajoitetun levinneisyyden käsitetään useimmiten johtuvan juuri käytettävyydestä, sillä avoimen lähdekoodin ohjelmistojen käyttäjät ovat teknisesti osaavia käyttäjiä. Avoimen lähdekoodin kehittäjät eivät siis ehkä osaa ottaa huomioon erinäisiä käytettävyyden piirteitä niin hyvin kuin esimerkiksi suljetun lähdekoodin kehittäjät.

Tämä johtuukin Nicholsin ja Twidalen (2002, 3) mukaan siitä seikasta, että suljetun lähdekoodin kehittäjät eivät itse asiassa ole ohjelmiston ensisijaisia käyttäjiä. Muuna syynä on osaltaan myös se, että avoimen lähdekoodin kulttuuri on vielä osittain sisäänpäin kääntynyttä. Toisaalta avoimen lähdekoodin virikkeet johtavat helpommin eri toiminnallisuuksien lisääntymiseen kuin esimerkiksi käytettävyyden parantumiseen. Avoimen lähdekoodin eräänä periaatteenahan on se, että toiminnallisuus on esimerkiksi yksinkertaisuutta tärkeämpää. Ensisijaisena syynä on kuitenkin se tosiasia, että käytettävyyden ongelmat ovat toiminnallisia ongelmia huomattavasti vaikeampia käsitellä, ja niiden ratkaiseminen vaatii aikaisempaa enemmän resursseja. (Nichols & Twidale 2002, 4-6)

Avoimen lähdekoodin mukainen ohjelmistotuotanto näyttäisi olevan kohtalaisen voimakkaasti riippuvainen uuden toiminnallisuuden lisääntymisestä. Tämä on Lehmanin ohjelmiston evoluution kuudennen lain mukainen näkökohta. Lain mukaan käyttäjien (tai tässä tapauksessa kehittäjien) tarve uusien toiminnallisuuksien jatkuvalle lisääntymiselle kasvaa ohjelmiston elinkaaren edetessä. (Lehman ym. 1998, 217)

Ongelmia voi lisäksi tuottaa myös erittäin kattava eri käyttöjärjestelmien suosiminen. Toisaalta monipuolisten käytänteiden vaihtelu voi aiheuttaa ongelmia ymmärryksen suhteen. Toki on myös mahdollista, että kun avoimen lähdekoodin ohjelmistoihin integroidaan toisia avoimen lähdekoodin ohjelmistoja, rakennettavan ohjelmiston rakenne voi muodostua liian monimutkaiseksi juuri täydellisen avoimuuden vuoksi.

On vielä olemassa monia pelkoja ja huolenaiheita avoimen lähdekoodin laatuun ja luotettavuuteen liittyen, vaikkakin muutamien tunnettujen avoimen lähdekoodin ohjelmistojen käyttö on jo merkittävällä tasolla. Huolenaiheina ovat muun muassa, kuinka toisaalta lähdekoodin laadunvarmistusta voidaan toteuttaa sekä toisaalta, kuinka myös muita projektinsisäisiä tuotoksia pystytään varmentamaan laadun kannalta. Pelot pohjautuvat osin siihen, ettei juuri avoimen lähdekoodin piirteitä huomioonottavaa varttunutta laadunarviointimetriikkaa ole esitetty. (Zhou & Davis 2005, 67) Tämä johtaa helposti siihen, ettei aina osata arvioida sitä, milloin ohjelmisto on tarpeeksi kypsä käyttöönotettavaksi (Zhou & Davis 2005, 68). On siis hyvin mahdollista, että projektit kestävät huomattavan kauan kypsyä ”valmiiksi”.

Yhteenvetona on huomautettava, ettei mikään edes avoimen lähdekoodin mukaisessa ohjelmistotuotannossa ole lopulta ilmaista. Projektien puolesta on tehtävä paljon aktiivista työtä, jotta lopputuloksista saataisiin toivottuja ja toimivia, sekä toisaalta yleishyödyllisiä, kuten pääperiaatteena on. Huomioitujen ongelmakohtien monimuotoisuus sekä useiden ongelmakohtien hankala korjaaminen näyttävät muodostavan avoimelle lähdekoodille ison tulevaisuuden haasteen.

4 AVOIMEN LÄHDEKOODIN OHJELMISTON YLLÄPITO JA YLLÄPIDETTÄVYYS

Tässä tutkielman neljännessä luvussa arvioidaan erityisesti avoimen lähdekoodin ohjelmiston ylläpidettävyyspiirteitä luvussa kaksi esitetyn tieteellisen luokittelun perusteella ylläpidettävyyden aspekteista. Näiltä osin ylläpidettävyyden arviointi kohdistuu kohdeohjelmiston ylläpidettävyyteen. Tämän lisäksi luvussa käsitellään myös jonkin verran avoimen lähdekoodin mukaisen ohjelmistotuotannon ylläpitoa. Huomionarvoista on se, että ohjelmistot käsitetään luvun yhteydessä ainoastaan avoimen lähdekoodin mukaisen ohjelmistotuotannon (tai vapaan ohjelmistokehityksen) tuotoksiksi. Ohjelmistojen on toisin sanoen oltava kehitetty avoimen lähdekoodin projekteina, lähestulkoon alusta alkaen. Näin esimerkiksi avoimen lähdekoodin ohjelmistoksi siirrettyä täysin muuttumatonta suljetun lähdekoodin ohjelmistoa ei käsitetä aidoksi avoimeksi ohjelmistoksi.

Lukijan on syytä huomioida, että ylläpidon määritelmä avoimen lähdekoodin mukaiselle ohjelmistotuotannolle on ”häilyvä”, sillä sekä kehityksen luonne että kehityksessä hyväksikäytettävät apuvälineet mahdollistavat iteratiivisen kehittämisen. Tämä selvinnee siitä seikasta, että ohjelmistoa on samanaikaisesti mahdollista sekä kehittää että ylläpitää. Näkökulma on mielenkiintoinen, koska kuten jo toisessa luvussa esitettiin, on ohjelmiston ylläpito perinteisesti nähty vain ohjelmiston julkaisuajankohdan jälkeen tapahtuvaksi välttämättömäksi haitaksi. Avoimen lähdekoodin mukaisessa ohjelmistotuotannossa voidaan ongelmakohdat jo hyvissä ajoin kehityksen aikana tunnistaa, sekä näin ollen integroida tarvittavat korjaukset kehittyvään lähdekoodikokoelmaan – ohjelmaan tai ohjelmistoon. Tällöin ohjelmiston kehitys epäilemättä kuormittuu enemmän, mutta varsinaisessa perinteisessä mielessä nähty ylläpito puolestaan kevenee.

Kuten aiemmin on jo mainittu, tässä tutkielmassa ylläpito ja uuden toiminnallisuuden kehitys käsitetään yhteisesti ylläpidoksi, sillä ylläpidettävyys

toimii molempien toimenpiteiden ”mittarina”. Kannattaa tosin muistaa, että valittu näkökulma vääristää hieman todellisuutta, koska avoimen lähdekoodin mukaisen ohjelmistotuotannon yhtenä periaatteena on julkaista ohjelma tai ohjelmisto hyvin aikaisessa vaiheessa testattavaksi (ja jopa käytettäväksi). Todellisuudessa tämä näkökulma merkitsisikin sitä, että lähes koko avoimen lähdekoodin mukainen ohjelmistotuotanto olisi pelkkää ylläpitoa, mikä ei kuitenkaan pidä paikkaansa.

4.1 Ylläpito

Avoin lähdekoodi vaatii ylläpitoa siinä missä esimerkiksi suljettu lähdekoodikin. Toisaalta avoimen lähdekoodin mukaisen ohjelmistotuotannon ylläpidolliset toimet poikkeavat jonkin verran muiden lähestymistapojen ylläpitotoimista. Kuten aiemmin on todettu, ylläpito käsiteltävässä lähestymistavassa on reaktiivista, tarpeeseen perustuvaa toimintaa, jota toteutetaan iteratiivisesti aina tarpeen niin vaatiessa. Tämän lisäksi korjausten nopea saapuminen ja leviäminen on hyvin tavanomaista.

Avoimen lähdekoodin mukaiseen ohjelmistotuotantoon sisältyy joukko erityisesti toistuvia ylläpitotehtäviä, jotka koostuvat sekä olemassaolevien toimintojen testauksesta että uusien toimintojen lisäyksistä. Kyseiset tehtävät ilmenevät lähinnä korjaavan ja täydentävän ylläpidon muodossa. (Samoladas ym. 2004, 84)

Toisin sanoen, sekä vanhan korjaaminen ja parantelu että toisaalta uuden toiminnallisuuden lisääminen on lähestymistavan mukaiselle ohjelmistotuotannolle luonteenomaista, ja siksi esimerkiksi ennaltaehkäisevää ylläpitoa ei suosita kovinkaan aktiivisesti. Tämä voi tosin pitkällä tähtäimellä johtaa rakenteellisiin ongelmiin, jolloin muun muassa uudelleenrakenteistamista luultavasti tarvitaan korjaamaan syvälle lähdekoodin rakenteisiin ulottuvat ongelmat.

4.1.1 Prosessi ja tehtävät

Varsinaista yleistä ylläpitoprosessikehystä ei avoimen lähdekoodin mukaisen ohjelmistotuotannon yhteydessä ole mahdollista esittää. Lisäksi avoimen lähdekoodin mukaisen ohjelmistotuotannon ylläpito ei useinkaan koostu systemaattisesti toistuvista prosessimaisista tehtävistä. Toisaalta Koponen ja Hotti (2005a, 30) ovat kuitenkin huomioineet Apachen ja Mozillan ylläpitoprosessien perusteella yleisesti, että avoimen lähdekoodin ylläpitoprosessi on muodollinen (eli kontrolloitu). Käytännössä siis kuka tahansa voi syöttää korjauksia ja vikaraportteja avoimen lähdekoodin projekteille. Ehkä jopa hieman yllättäen vertailukohdaksi valitsemastaan kuusikohtaisesta ISO/IEC-ylläpitoprosessikehyksestä he löysivät jopa neljä vastaavaa aktiviteettia kahden edellä mainitun avoimen lähdekoodin ohjelmiston ylläpitoprosesseista. Nämä neljä osa-aluetta olivat pääkohdiltaan samansuuntaiset, mutta itse tehtäväsisällöiltään jokseenkin poikkeavat. Varsinaisten erojen voi osatehtävien osilta selkeästi identifioida kohdistuvan ISO/IEC-prosessikehyksen virtaviivaistamiseen (eli ripeästi ja kevyesti läpivietäviin tehtäväkokonaisuuksiin). (Koponen & Hotti 2005a, 30)

Ylläpitoprosessin toteutuksessa sekä vianhallinnan että version- ja konfiguraationhallintojen osalta avoimen lähdekoodin ylläpitoprosessin tehtävät ovat samankaltaisia ISO/IEC-kehykseen verrattuna. Avoimen lähdekoodin ylläpidon yhteydessä sekä vianhallinnassa että version- ja konfiguraationhallinnassa luotetaan usein luotettaviin työvälineisiin, kuten muun muassa Bugzilla- ja CVS -ohjelmistoihin. Tämän kokonaisuuden lisäksi vika- ja muutosanalyysissä, muutoksen toteutuksessa, sekä muutoksen arvioinnissa ja hyväksynnässä on kummassakin prosessikehyksessä yhteneviä piirteitä. Varsinaiset erot tulevat esille viimeistään, tosin kuitenkin luonnollisesti, julkaisun hallinnassa ja ylläpitoprosessin päättymisessä. (Koponen & Hotti 2005a, 33-34)

Eräs ylläpitoon liittyvistä käytännön ongelmista liittyy lähdekoodiriveihin tehtävien muutosten tunnistamisiin. Ylläpitäjän on aina tiedettävä mihin varsinaisiin kohtiin muutoksia pitää tehdä ja millaisia muutoksia on ylipäätään tehtävä. Tämä on ongelma sekä suurissa että pirstoutuneissa, juuri vaikeasti ylläpidettävissä, ohjelmistoissa. Ying ym. (2004, 574) ovat esittäneet soveltuvan menettelytavan, jonka avulla muutosta tarvitsevat lähdekoodirivit sisältävät lähdekooditiedostot on mahdollista tunnistaa. Heidän esittelemä menetelmä perustuu lähdekooditiedostojen muutoshistorioiden *louhintaan* (*engl. mining*), jossa muun muassa edellisten samankaltaisten vikakorjausten aikana muutetut lähdekooditiedostot identifioidaan. Kyseisen menetelmän käyttö on järkevää jo muun muassa siksi, että sen avulla voidaan muutosten tekemistä vääriin paikkoihin minimoida, jolloin myös itse ylläpidettävyyttä varjellaan.

4.1.2 Tarve

Ylläpidon tarve on projektikohtaisesti luonnollisesti hyvin vaihtelevaa. Avoimen lähdekoodin mukaisessa ohjelmistotuotannossa ylläpidolliset tehtävät jaetaan useimmiten prioriteetteihin (esimerkiksi kriittinen, normaali, ei-kriittinen -luokittelu), vähintäänkin implisiittisesti, jonka seurauksena vähemmän tärkeät tai erittäin hankalat tehtävät saattavat jäädä vain tiettyjen ansioituneiden kehittäjien harteille.

Ylläpito on avoimen lähdekoodin ohjelmistolle merkittävää erityisesti projektin alkuvaiheissa muun muassa lähdekoodien yhteensovittamisen muodossa, sillä toisaalta kehittäjien yhteistyö ei ehkä ole vakiintunut eikä toisaalta ohjelmiston rakenne ole useinkaan vielä selvillä. Tarpeellista ylläpito on myös silloin, kun uusia ja tärkeitä, mutta viallisia toiminnallisuuksia (tai rakenteita) lisätään. Tarpeen määrää kuvaa osaltaan myös se, kuinka monen käyttäjän työskentelyä jokin tietty ongelma haittaa. Tarve voi toki syntyä myös silloin, kun jokin projektin ydinkehittäjistä tai koordinoijista päättää niin.

4.2 Kohdeohjelmiston ylläpidettävyys

Ylläpidettävyys viittaa ylläpitokykyyn. Kaikki ohjelmiston laadulliset vaatimukset, kuten myös ylläpidettävyys, ovat ei-toiminnallisia ja subjektiivisia luonteenpiirteiltään. Laadullisia vaatimuksia ei useimmiten koskaan voida absoluuttisessa mielessä saavuttaa. Onkin vain mahdollista päästä sellaiselle tasolle, jossa pakolliset tarpeet saadaan täytettyä ja lopputulos on tyydyttävää. (Tahvildari ym. 2001, 72) Erityisesti ylläpidettävyyteen johtavat tietyt lisävaikutteet, kuten muun muassa lähdekoodin ja ylläpitovaateen keskinäinen subjektiivinen vuorovaikutus (Haworth ym. 1992, 108).

Avoimen lähdekoodin tulee olla korkeasti ylläpidettävä juuri lähestymistavan mukaiselle ohjelmistotuotannolle ominaisten piirteiden vuoksi (Raja & Barry 2005, 44). Myös Samoladas ym. (2004, 84) ovat esittäneet, että avoimen lähdekoodin mukainen ohjelmistotuotanto korostaisi ohjelmiston ylläpidettävyyttä. Samoladaksen ym. (2004, 86-87) vertailututkimuksessa avoimen ja suljetun lähdekoodin ylläpidettävyydestä ilmaantui mielenkiintoinen näkökulma. Vaikka avoin lähdekoodi näyttäisi olevan joko yhtä laadukasta tai jopa laadukkaampaa kuin vastaavan toiminnallisuuden sisältävä suljettu lähdekoodi, kärsii avoin lähdekoodi kuitenkin samankaltaisista ongelmista kuin suljettu lähdekoodikin. Kuten suljetun lähdekoodin ohjelmistojen ylläpidettävyys rappeutuu ajan kuluessa, niin näin käy myös poikkeuksetta avoimen lähdekoodin ohjelmistojen ylläpidettävyydelle. Tämä rappeutuminen tuottaa niin kutsuttuja *perinnejärjestelmiä* (*engl. legacy systems*). Myös avoimen lähdekoodin ohjelmistojen ikääntyessä on hyvin todennäköistä, että käänteistekniikoita tarvitaan myöhemmin avointen ohjelmistojen rakenteiden ja toimintojen selvittämiseksi. (Samoladas ym. 2004, 86-87) Tämä laadun rappeutumisen näkökulma noudattaa suoraan Lehmanin ohjelmiston evoluution seitsemättä lakia (Lehman ym. 1998, 217).

Zhou ja Davis (2005, 71) ovat varovaisesti esittäneet, että avoimen lähdekoodin kehityssykliin liittyy samankaltaista luotettavuuden kasvua kuin suljetun lähdekoodin kehityssykliin. Tämä ilmenee siten, että vikaraporttien tulo vakiintuu ajan mittaan erittäin alhaiselle tasolle. Toisaalta on hyvin todennäköistä, ettei avoimen lähdekoodin ohjelmiston arviointiin ole olemassa yhtä yleispätevää menetelmää projektikohtaisten käytänteiden merkittävän vaihtelun johdosta. Esimerkiksi erilaiset suosiomittarit eivät riitä laadullisiksi takeiksi. (Zhou & Davis 2005, 71)

Kuinka ylläpidettävyyttä voidaan parantaa? Ohjelmiston kompleksisuus on eräs ohjelmiston laadun aspekti (Stewart ym. 2005, 62), jonka johdosta kompleksisuus vaikuttaa olennaisesti myös muun muassa ylläpidettävyyteen. Kompleksisuuden hallinta onkin eräs huomattava näkökulma ylläpidon kannalta. Ohjelmiston koko, kytkentä ja kiinteys on perinteisesti käsitetty eräiksi tärkeiksi kompleksisuuden ilmentymiksi.

Kytkentä ja kiinteys ovat erityisesti lähdekoodin rakenteen kompleksisuuden ominaisuuksia, jotka muodostuvat luonnollisesti ohjelmiston modularisoinnista eli ongelmien osiin pilkkomisesta (Stewart ym. 2005, 63). Kytkentä viittaa siihen, että ohjelmiston eri kokonaisuuksien väliset suhteet ovat mahdollisimman vähäiset, jolloin kokonaisuudet eivät ole liian riippuvaisia toisistaan. Kiinteys puolestaan viittaa siihen, että kyseiset kokonaisuudet on jaettu loogisiin kokonaisuuksiinsa, joilla on vain yksi tietty tarkoitus. Korkea kiinteys ja matala kytkentä ovat ohjelmiston yleisesti suosittuja periaatteita (Dagpinar & Jahnke 2003, 155).

Briand ym. (2001, 526) ovat tutkineet olioperustaisen lähestymistavan vaikutuksia ylläpidettävyyteen ja tulleet siihen tulokseen, että juuri yleisten olioperustaisten suunnitteluperiaatteiden noudattaminen johtaa helpommin ylläpidettäviin ohjelmistoihin. Ylläpidettävyys on kuitenkin arkaluonteinen huonoille suunnitteluperiaatteille. Mielenkiintoista on myös se, että

suunnittelustandardien parantaminen vähentäisi ylläpidettävyyden kustannuksia huomattavasti. (Briand ym. 2001, 526)

Lisäksi erityisesti Kiran ym. (1997, 120) ovat tehneet subjektiivisen huomion siitä, että olioparadigma näyttäisi hieman vähentävän ylläpidon vaatimien muutosten määrää lähdekoodiriveissä verrattuna esimerkiksi rakenteiseen paradigmaan. Näin on siksi, että muutokset keskittyvät oliototeutuksissa useimmiten pienempiin yksiköihin (Kiran ym. 1997, 120).

Eräänä varmana ja hyödyllisenä keinona parantaa tulevien ohjelmistojen ylläpidettävyyttä on luonnollisesti tutkia vanhoja aiemmin toteutettuja (ja tietysti ylläpidettyjä) ohjelmistoja, sekä toisaalta niiden kehityksen vaiheita. Etenkin osajärjestelmien kasvukuvioita on hyödyllistä tutkia, jotta voidaan saada hyvä kokonaiskuva siitä, miksi ja kuinka koko järjestelmä on kehittynyt sellaiseksi kuin se on (Godfrey & Qiang 2000, 141).

Omanin ja Hagemeisterin (1992) ylläpidettävyyden aspektien luokittelu saa kritiikin aihetta siitä, ettei oliolähdekoodin ylläpidettävyysaspekteja ole riittävän kattavasti integroitu mukaan, vaikkakin luokittelu kattaa myös joitakin yleisiä oliopiirteitä. On kuitenkin todettava se, että molempien sekä rakenteisen paradigman että olioparadigman ylläpidettävyyden aspektien yhdistäminen yhdeksi luokitteluksi on erittäin haastava, tai jopa mahdoton, tehtävä! Toinen kritiikin aihe luokittelussa on se, että muutamista ylläpidettävyysaspekteista jää osittain pinnallinen kuva. Tämä johtunee varsinkin siitä seikasta, ettei kaikkien aspektien ole suoraan osoitettu (tai todistettu) vaikuttavan ylläpidettävyyteen. Toinen seikka lienee se, että osa ylläpidettävyysaspekteista on enemmän mittarinomaisia ominaisuuksia kuin pelkästään lähdekoodin ominaisuuksia ilmentäviä.

Dagpinar ja Jahnke (2003) ovat esittäneet olioiden ylläpidettävyyspiirteistä erittäin mielenkiintoisia huomioita. Sekä koko että *sisältävä suora kytkentä* (*engl. import direct coupling*), joka käsittelee toisten olioiden toiminnallisuuksia

hyväksikäyttäviä olioita, on oliolähdekoodissa nähty olioiden huomattaviksi ylläpidettävyyden mittareiksi. Tämän lisäksi kuitenkaan *perintä* (*engl. inheritance*), kiinteys ja *epäsuora kytkentä* (*engl. indirect coupling*) tai *vievä kytkentä* (*engl. export coupling*) eivät näyttäisi selkeästi ilmentävän olioiden ylläpidettävyyttä! Vievä kytkentä käsittelee sellaisia olioita, joiden toiminnallisuuksia toiset oliot hyväksikäyttävät. (Dagpinar & Jahnke 2003, 157-163) Kuitenkin muun muassa *perintäpohjaisen kytkennän* (*engl. class inheritance-related coupling*) voidaan osaltaan käsittää kuvaavan sisältävää suoraa kytkentää.

On kuitenkin olennaista huomioida, että oliolähestymistavan perinteisten lähtökohtien (kuten perinnän, informaation piilottamisen, polymorfismin, dynaamisen sidonnan ynnä muiden) mukaan lukeminen oliolähdekoodin ylläpidettävyyttä osaltaan kuvaaviksi, edes mittarinomaisiksi, aspekteiksi olisi luultavasti hyvinkin luonnollinen ja järkevä lähtökohta. Esimerkiksi Wilde ym. (1993, 78) mainitsevat etenkin *dynaamisen sidonnan* (*engl. dynamic binding*) ja *polymorfismin* (*engl. polymorphism*) pystyvän aiheuttamaan vakavia ongelmia toteutusten ymmärtämiselle.

Seuraavissa alaluvuissa käsitellään ylläpidettävyyden aspekteja avoimen lähdekoodin mukaisen ohjelmistotuotannon tuottamien kohdeohjelmistojen näkökulmista.

4.2.1 Kypsyys

Valitettavasti vanhimmatkin nykyisin käytössä olevista, ja etenkin kolmannessa luvussa kuvattujen menettelytapojen tuottamista, avoimen lähdekoodin ohjelmistoista ovat vain jonkin verran yli kymmenen vuoden ikäisiä. Suurin osa kyseisellä lähestymistavalla toteutetuista ohjelmistoista on vielä nuoria ja monin osin myös pienikokoisia. Tämän vuoksi ei siis ole mahdollista kattavasti arvioida pitkäikäisten avoimen lähdekoodin ohjelmistojen kypsyysaspekteja.

Valitettavaa on myös se, ettei elinkaarensa loppupäässä olevien ohjelmistojen rappeutumisaspekteja voida vastaavasti laajasti arvioida.

Kaikista tunnetuimpana pitkäikäisistä ohjelmistoista on etupäässä Linux, joka on jo useissa eri foorumeissa yleisesti käsitetty olevan kypsä ohjelmisto. Muun muassa Koponen ja Hotti (2005b, 691-692) tutkivat Mozillan ja Apachen kasvukuvioita ja päätyivät tulokseen, että molemmat ohjelmistot ovat vakiintuneita ja kypsiä ohjelmistoja. Tämän he päättelivät siitä, että toisaalta parannusten prosentuaalinen määrä oli alhainen sekä siitä, että toisaalta korjauksiin johtaneiden vikaraporttien määrä oli myös alhaisella tasolla. Lisäksi vakavia virheitä ei ylipäätään ollut prosentuaalisesti paljoa ja suurin osa vioista voitiin kategorisoida (noin 50 prosenttia) normaaleiksi ei-kriittisiksi vioiksi. Tuloksen yllättävä piirre oli havainto, että vain vähäinen osa virheistä korjattiin. Tämä johtunee siitä, etteivät läheskään kaikki projektien vikaraportit olleet aiheellisia ja monet viat raportoitiin useampaan kertaan. (Koponen & Hotti 2005b, 691-692) Vaikka vikaintensiteetti vaihtelee huomattavasti projektikohtaisesti, niin vähänkin vanhemmat ja kypsemmät ohjelmistot ovat selvästi vähävikaisia avoimen lähdekoodin lähestymistavassa.

Luotettavuuden on monissa yhteyksissä erityisesti korostettu olevan kaikista kypsyyden luokittelun aspekteista luonteenomaisinta lähestymistavalle. Luotettavuuden voidaan aiheellisesti nähdä johtuvan juuri voimakkaasta rinnakkaisesta virheenjäljityskulttuurista. Lisäksi toisaalta myös uudelleenkäyttöä on avoimen lähdekoodin lähestymistavassa korostettu voimakkaasti, sillä käytännönläheinen periaate ”pyörän uudelleenkeksimisestä” ohjaa kehittäjiä käyttämään uudelleen jo kirjoitettua ja toimivaksi osoitettua lähdekoodia niin paljon kuin vain ylipäätään on järkevää ja mahdollista.

Muuttuneiden rivien määrän on luokittelussa arvioitu osaltaan ilmentävän tasapainoisuutta (Oman & Hagemeister 1992, 338). Tämän johdosta monia avoimen lähdekoodin ohjelmistoja ei voidakaan käsittää kovin tasapainoisiksi,

sillä muutoksia ohjelmistoihin tehdään projekteissa yleensä paljon ja nopealla tahdilla. Lisäksi aktiivisen kehittämisen tahti ei vielä pitkänkään ajan kuluttua näyttäisi monissa projekteissa ainakaan huomattavasti vähentyvän. Tämä aiheuttaa ongelmia ylläpidettävyydelle etenkin silloin, kun eri muutoksilla ei ole aikaa stabiloitua osaksi ohjelmistoa.

Ohjelmointikielinä avoimen lähdekoodin projekteissa käytetään monipuolisesti eri kieliä, joita ovat muun muassa C, C++, Java ja Python. Etenkin hyvin suosittu C-ohjelmointikieli on kuitenkin monissa yhteyksissä osoitettu kompleksiseksi ohjelmointikieleksi, joka altistaa helposti vaikeasti havaittaville virheille. On toki huomautettava, että C-ohjelmointikielen on yleisesti käsitetty olevan ammattilaisten työväline rakenteiseen ohjelmointiin kielen vanhuudesta riippumatta. Rakenteisen C-ohjelmointikielen verraten suuri suosio perustuu ehkä juuri kielen näennäiseen yksinkertaisuuteen ja virtaviivaisuuteen. Kieli tosin tuottaa myös helposti hallinnoitavia projektirakenteita. Sekä ohjelmointikielen valinta että kehittäjien taidot siis vaikuttavat suoraan ylläpidettävyyteen.

Sovellusalueiden vaihtelevuuksien vuoksi on vaikea todeta monia yleispäteviä asioita sovellusten kompleksisuustasoista. Sovellusalue ei yksistään ole vaikuttava tekijä sovelluksen kompleksisuuteen. Se tosin on selvää, että avoimen lähdekoodin ohjelmistot voivat olla kompleksisia ohjelmistoja. Ongelmia syntyy helposti varsinkin silloin, kun kompleksisuutta ei aktiivisesti pyritä hallitsemaan. Tietyissä, varsinkin suurikokoisissa, projekteissa on usein osaava kehittäjäkanta, joka osaltaan pienentää ohjelmiston kompleksisuuden merkitystä ylläpidettävyyden kannalta. Lähestymistavalle osittain luonteenomaiseksi kuvatun yksinkertaisuuden periaatteen noudattamisen voidaan osaltaan toki nähdä myös ylläpidettävyyttä parantavaksi tekijäksi.

Avoimen lähdekoodin projektit ovat orgaanisia ja muuttuvia ympäristöjä, jonka vuoksi kehityksen vaativuus vaihtelee usein huomattavasti kehittäjäkohtaisesti. Vaikeimpia tehtäviä pyritäänkin jakamaan osaavimpien kehittäjien vastuille.

Tämä johtaa osin siihen, että asetetut vaatimukset ovat vaihtelevia, jolloin ohjelmiston tasaisen laadun ilmentäminen voi osoittautua paikoin hankalaksi. Epäselvyyttä onneksi tosin minimoi osaltaan se, että testauksen kattavuus kenttätestauksen muodossa on avoimen lähdekoodin projekteissa usein aivan riittävällä tasolla.

Liiallinen uusiokäyttö voi avoimen lähdekoodin ohjelmistoissa johtaa hankaliin ja syviin riippuvuussuhteisiin. Tiettyjen avointen ohjelmistojen jopa toimimaan saattaminen voi vaatia useiden eri komponenttien hakua eri lähteistä. Ongelmaa hankaloittaa lisäksi se, että jo uudelleenkäytetyissä komponenteissa myöhempien uusien versioiden muutokset voivat aiheuttaa muun muassa ikäviä ylläpidettävyyttä huonontavia rakenteellisia ongelmia.

Avoimen lähdekoodin projekti pohjautuu yhteiseen tietovarastoon, jonne on *integroitu* (*engl. integrated*) ja *synkronisoitu* (*engl. synchronize*d) kaikki projektissa käytetyt ja käytössä olevat tiedostot. Esimerkiksi kääntämistä helpottavia toimenpiteitä tehdään useissa projekteissa, jo pelkästään siksi, että ohjelmistoja julkaistaan etenkin testattavaksi tiheään tahtiin. On tosin olemassa viitteitä siitä, että etenkin aloitteleville kehittäjille jo pelkästään ohjelmiston kääntäminen voi olla vaikea tehtävä (ehkä osin juuri käytettävyyden johdosta).

Vaikka esimerkiksi Fitzgerald (2004, 92) on viitannut esitettyihin väitteisiin ohjelmistojen nopeasta kehityksestä, on vielä olemassa epäilyjä sen suhteen, että ohjelmistojen kypsyneisyyden saavuttaminen kestää avoimen lähdekoodin lähestymistavassa jopa huolestuttavan kauan. Tämän osoittavat esimerkiksi Linux ja Mozilla tarkasteltaessa niiden kehityksen kestoa. Kypsyminen vaatii tosiasiassa muun muassa rakenteen stabiiliutta ja toiminnallisuuksien harmoniaa (tai tasapainoisuutta), eli uusien toiminnallisuuksien lisääntymisten huomattavaa vähenemistä.

4.2.2 Lähdekoodi

Avoimen lähdekoodin ohjelmiston ylläpidettävyyteen vaikuttavat eniten juuri lähdekoodin eri aspektit, sillä lähdekoodilla on lähestymistavassa erityisen korostettu rooli. Seuraavissa alaluvuissa käsitellään sekä järjestelmän että komponenttien kohdalta kontrolli-, tieto- ja koodirakenteiden ylläpidettävyysaspekteja.

4.2.2.1 Kontrollirakenne

Avoimen lähdekoodin ohjelmistot tukevat etenkin järjestelmätason kontrollirakenteen modulaarisuuden periaatetta voimakkaasti. Eri toiminnallisuudet on avoimen lähdekoodin projekteissa useimmiten jaettu toisiaan lähellä oleviin kokonaisuuksiin, vieläpä mahdollisesti siten, että komponentit (tai moduulit) on jaettu kehittäjäkohtaisiksi. Tämä parantaa osaltaan ylläpidettävyyttä siksi, että kehittäjä on näin ollen aina tietoinen omasta työstään ja työalueestaan.

Modularisoinnissa on kuitenkin myös ongelmansa. Esimerkiksi liiallinen modularisointi jakaa ohjelmistoa liian pieniin kokonaisuuksiin, jolloin pelkästään niiden yhteensovittaminenkin voi osoittautua ongelmalliseksi. Myös vääränlaisten periaatteiden käyttö lisää helposti liiallisia kytkentöjä ohjelmistossa. Modularisoinnin periaate on omiaan helpottamaan muun muassa avoimen lähdekoodin ohjelmiston rakenteen muotoutumista, sillä ohjelmiston arkkitehtuuri koostuu vuorovaikutuksellisista osista eli elementeistä.

Järjestelmän kompleksisuus perustuu osaltaan komponenttien käyttöön kontrollirakenteen kannalta. Järjestelmän kompleksisuutta voidaan minimoida panostamalla erityisesti komponenttien oikeanlaiseen toteutukseen, käyttöön ja sijoitteluun. Tämä on avoimen lähdekoodin ohjelmistoille luonteenomaista. Ohjelmiston ylemmän tason kontrollirakenteen tehtävänä on kuitenkin koota

komponenttien toiminnot yhtenäiseksi toimivaksi kokonaisuudeksi, joka joko vaikeissa algoritmisissa suoritteissa tai voimakkaasti modularisoiduissa järjestelmissä voi kuvautua järjestelmän kontrollirakenteen kompleksisuutena.

Kompleksisuuden hallinta on tietyissä avoimen lähdekoodin projekteissa aktiivisesti tarkkailtava näkökulma. Lisäksi joissain projekteissa pystytään selvästi parantamaan ohjelmiston laatua sen koon kasvaessa. Tämä tulos perustuu kompleksisuuden mittarien, kytkennän ja kiinteyden, tarkkailuun oliokielillä toteutetuissa ohjelmissa ja ohjelmistoissa. On siis olemassa empiiristä näyttöä siitä, että sekä kytkentöjen vähentymisen että kiinteyden parantumisen, tai vain pelkästään toisen kompleksisuuden mittarin paranemisen, myötä on ohjelmiston laatu parantunut. Parhaassa tapauksessa avoimen lähdekoodin mukaista ohjelmistotuotantoa pystytäänkin käyttämään hyväksi juuri laadun parantamisessa. (Stewart ym. 2005, 65) Kompleksisuuden kasvamisen näkökulma on suoraan verrannollinen Lehmanin ohjelmiston evoluution toiseen lakiin nähden (Lehman ym. 1998, 217).

Johdonmukaisuus kuvaa sitä, miten yhtenäisiä moduulit ovat (Oman & Hagemeister 1992, 340). Tämä on luonnollisesti hyvin voimakkaasti subjektiivinen mittarimainen ominaisuus, jonka yleinen arviointi on käytännössä mahdotonta.

Avoimen lähdekoodin ohjelmistojen järjestelmän kontrollirakenteet voivat olla syviä, sillä useissa tapauksissa kontrollin rakenne on puumainen, jolloin etenkin isoissa ohjelmistoissa voi tasoja olla useita. Esimerkiksi kolme tasoa syvä puurakenne ei varmaankaan ole mikään mahdottomuus avoimille ohjelmistoille. Pienissä projekteissa syvyys ei kuitenkaan luonnollisesti muodostu ongelmaksi. Ylläpidettävyyttä parantavat toisaalta se, että puurakenne on mahdollisimman tasapainoinen ja toisaalta se, että puurakenne on sopiva suhteessa ohjelmiston kokoon.

Kontrollien kytkentä on eräs huomattava ylläpidettävyyden tekijä. Dagpinarin ja Jahnken (2003, 163) mukaan se on myös olioiden ylläpidettävyyteen vaikuttava tekijä. Valitettavasti avoimen lähdekoodin ohjelmistojen osiin pilkkominen ilman selvää kokonaiskuvaa osien välisistä suhteista johtaa helposti siihen, että eri ohjelmistokomponentit tai -moduulit käyttävät hyväkseen toisissa komponenteissa tai moduuleissa olevia toiminnallisuuksia. On erittäin viisasta arvioida aika ajoin etenkin avoimen lähdekoodin ohjelmistojen kehityksessä kontrollien kytkennän etenemistä, sillä se osaltaan kuvaa sitä, onko ohjelmiston osiin jako onnistunut vai ei. Kytkentöjen madaltamisen huomioiminen ylläpidettävyyden parantamiseksi on muun muassa komponenttien uudelleenjaon eräs huomiokohde.

Järjestelmän tasolta eri komponenttien kontrolleihin syntyy luonnollisesti kytkentöjä, sillä järjestelmä muodostaa eri toiminnallisuuksista loogisen kontrollivuon. Komponenttien välinen kontrollien kytkentä on kuitenkin huomattavasti suurempi vaikuttava tekijä ylläpidettävyyteen, kuin mitä järjestelmän ja komponenttien välinen kontrollien kytkentä on. Toki modularisoinnin mahdolliset ongelmat kuvautuvat etenkin kytkentöjen suhteen.

Avoimen lähdekoodin ohjelmistojen komponenttien välille syntyy välttämättä kytkentöjä muun muassa siksi, että eri komponenttien toiminnallisuuksiin viitataan toisista komponenteista yleisesti. Joissain avoimen lähdekoodin ohjelmistoissa voi kontrollikytkentöjen liian suuri määrä olla huomattava ongelma. Schach, Jin ym. (2002, 19) ovat kuvanneet, että kytkentä on pakollinen seuraus modulaarisuudesta. Schach ja Offutt (2002, 1) ovat lisäksi huomauttaneet, että kytkentöjen vuoksi jo yhdenkin komponentin taipuneisuus vikoihin vaikuttaa vahingollisesti muiden moduulien ylläpidettävyyteen.

Komponenttien ja moduulien uudelleenkäyttö on avoimen lähdekoodin ohjelmistoille luonteenomaista. Uudelleenkäyttö ei lisäksi ole rajoittunut vain projektikohtaiseksi, vaan lähdekoodien eri osa-alueita käytetään kattavasti

uudelleen koko avoimen lähdekoodin alalla. Paras huomio avoimen lähdekoodin uudelleenkäytöstä on se, että uudelleenkäyttö toimii lähdekoodin tasolla. Voimakkaan (ja oikeaoppisen) uudelleenkäyttöstrategian noudattaminen parantaa monien avoimen lähdekoodin ohjelmistojen ylläpidettävyyttä.

Yksittäisten komponenttien kompleksisuus on jopa todennäköistä useimmissa avoimissa ohjelmistoissa. Käyttöjärjestelmätason kutsuja voidaan tarvita tietyissä alemman tason komponenteissa, kuten esimerkiksi graafisissa tai tietoliikenteen komponenteissa. Komponentin kompleksisuus on avoimen lähdekoodin mukaisessa ohjelmistotuotannossa usein lievästi näennäistä, sillä kehittäjien osaamistaso on perinteisesti ollut korkea. Tämän vuoksi ainakin yksi tai useampi kehittäjä pystyy näkemään komponentin mahdollisen kompleksisuuden ”läpi”.

Rakenteiden käyttö kuvaa muun muassa sitä kuinka yksi- tai monisuuntaisia komponenttien sisäiset kontrollivuot ovat (Oman & Hagemeister 1992, 340). Paras vaihtoehto on se, etteivät kontrollivuot ole monisuuntaisia. Tämä parantaa kontrollivuon suunnan ymmärtämistä ja ylläpidettävyyden selkeytymistä. Avoimen lähdekoodin kannalta kyseinen aspekti voi olla epäselvä muun muassa siksi, että kehityksen katkonaisuus ja systemaattisen suunnittelun puute rikkoo kontrollivoiden rakennetta.

Haarautuminen kuvautuu osaltaan siihen kuinka monen eri moduulin välille kontrollivuon toteutus on asetettu (Oman & Hagemeister 1992, 340). Kontrollin haarautuminen on ohjelmistoissa selvästi pakollista etenkin monimutkaisten algoritmien läpiviemisessä. Moduulin (tai komponentin) monihaarautuminen on usein ongelmallista ja siksi sen ratkaisemiseksi on perinteisesti suosittu joko lisämodularisoinnin periaatetta tai kontrollivuon yksinkertaistamista.

Kontrollirakenteen kiinteys viittaa osin siihen, kuinka muuttujia käytetään moduuleissa (Oman & Hagemeister 1992, 340). Mahdollisimman vähäinen

väliaikaisten muuttujien käyttö selventää kontrollin rakennetta huomattavasti. Muuttujien käytön on lisäksi oltava systematisoitua. Avoimen lähdekoodin ohjelmistoissa on monien muutosten jälkeen mahdollista se, että muuttujien käyttö ei ole virtaviivaista eikä järkevää, jolloin ylläpidettävyys kärsii. Kiinteyden pysymistä korkeana voidaan kuitenkin osittain varmistaa sillä, että komponentit tai moduulit jaetaan kehittäjien omiksi työskentelyalueiksi.

4.2.2.2 Tietorakenne

Yleisen periaatteen mukaisesti globaalien tietomuuttujien käyttö on erittäin huono käytäntö, joka muun muassa sekä vähentää tiedon muuttumisen kontrollia että sekoittaa tietorakennetta. Lokaalien tietomuuttujien käyttö lieventää tätä puolta dramaattisesti. Globaalit tietotyypit ja -rakenteet parantavat tiedon eksaktiutta (vähentävät siis tiedon eheyden rikkomista eli tarpeettomia rakenteen- ja muodonmuutoksia) ja ehkäisevät tiedon vääristymiä. Lokaalit tietotyypit ja -rakenteet puolestaan ovat hyödyllisiä spesifisten tietojen käsittelyssä, mutta ongelmia ne tuottavat silloin, kun tietoa on siirrettävä muodosta toiseen – esimerkiksi eri komponenttien tai moduulien välillä.

Paras keino tiedon käsittelemisessä korkean ylläpidettävyyden saavuttamiseksi on tiedon kompaktointi. On siis hyödyllistä sekä välttää turhaa redundanssia että myös pitää huolta siitä, että tietoa kontrolloidaan siten, että sitä käsittelevät vain sellaiset instanssit, joille on annettu oikeus tehdä niin. Oliolähestymistavan mukainen tiedon käsittely ohjaa tällaisiin oikeaoppisiin käytäntöihin. Capiluppin ym. (2003, 321) tutkimuksessa ilmeni subjektiivinen huomio siitä, että vähän alle kolmannes avoimen lähdekoodin ohjelmistoista oli rakennettu oliokielillä.

Yleinen kytkentä (*engl. common coupling*) viittaa kytkentään, jossa esimerkiksi kaksi moduulia jakavat saman globaalin datan. Schach, Jin ym. (2002, 23) esittävät, että Linuxin funktionaalisuuksien jatkuva kasvaminen näyttää johtavan uusien yleisten kytkentöjen syntymiseen. Heidän näkemyksen

mukaan Linuxista on siis tulevaisuudessa tulossa erittäin vaikea ylläpitää. Tästä tutkijat ovat tehneet loogisen johtopäätöksen: ”Mikäli avoimen lähdekoodin ohjelmistot kärsivät laajasti yleisistä kytkennöistä, voisi tämä jopa vaikeuttaa avoimen lähdekoodin ohjelmistojen omaksumista tulevaisuudessa”. Yleisten kytkentöjen vähäisyys toisin sanoen viittaa osaltaan korkeaan ylläpidettävyyteen. (Schach, Jin ym. 2002, 23)

Edellä huomioitu viittaa osaltaan siihen, ettei avoimen lähdekoodin ohjelmistojen tietorakenteita ole aina etukäteen suunniteltu. Niinpä tiedon käyttö ja tietojen rakenteet, liikkumiset ja tallentamiset määräytyvät (ja muuttuvat) usein käytännön tarpeiden sanelemina. Paljon muuttuvissa komponenteissa tai moduuleissa tämänkaltaisia ongelmia esiintynee valitettavan herkästi.

Tiedon levittyneisyys on hyvin komponenttikohtainen ominaisuus, joka osaltaan kuvaa sitä, kuinka tiiviinä ja keskitettynä tieto moduulissa (tai komponentissa) sijaitsee. Toisaalta tiedon esittelyn eheys on puolestaan voimakkaasti kehittäjästä riippuva ominaisuus, joka kuvaa etenkin tiedon eheyden rikkoutumista. On kuitenkin todettava, että osaavat kehittäjät pystyvät esimerkiksi edellä esiteltyjä tietorakenteen ongelmakohtia minimoimaan.

Erityisesti on todettava se, että useissa oliolähestymistavalla rakennetuissa avoimen lähdekoodin ohjelmistoissa on tietorakenteen muotoutuminen todennäköisesti oikeaoppista. Oliolähestymistavan käyttö onkin erittäin suositeltava näkökulma avoimen lähdekoodin ohjelmistojen kehitykselle. Vaikka ohjelmiston eri osat vaihtelisivatkin voimakkaasti, niin ongelmakohdat voidaan aina kohdistaa tiettyihin loogisiin kokonaisuuksiin (eli olioihin).

4.2.2.3 Koodirakenne

Koodirakenteen aspektit eivät vaikuta mitenkään itse ohjelmiston suoritukseen, vaan ne kuvaavat täysin lähdekoodin esitysmuotoon liittyviä ominaisuuksia

(Oman & Hagemeister 1992, 341). Koodirakenne on luonnollisesti voimakkaasti riippuvainen käytetystä ohjelmointikielestä. Toisaalta se on riippuvainen myös ohjelmoijan ohjelmointi- ja kommentointityyleistä. Toiset ohjelmointikielet pyrkivät esittämään lauseet kompaktissa muodossa, kun taas toiset pyrkivät enemmän lausemaisiin ja kuvaaviin esitysmuotoihin. Lisäksi ohjelmoijan taidot ja henkilökohtaiset ominaisuudet vaikuttavat siihen, kuinka laveasti tai kuvaavasti lähdekoodia tuotetaan, ylläpidetään ja kommentoidaan.

Ongelmaksi koodirakenne voi muodostua heti vähänkään suuremmissa projekteissa, joissa useat kehittäjät rakentavat tai ylläpitävät ohjelmistoja. Perinteisen ylläpidon kannalta tämä on yleensä vähäisempi ongelma, sillä kaikki muotoilu ja nimeämiskäytänteet ovat jo hyvissä ajoin ennalta sovittuja. Näin ei valitettavasti ole kuitenkaan avoimen lähdekoodin ylläpidettävyyden suhteen, sillä uusi kehitys ja perinteinen ylläpito ovat projekteissa usein samanaikaisesti suoritettavia toimenpiteitä.

Onneksi suurissa avoimen lähdekoodin ohjelmistoissa kontrollin ja tiedon käsittelykäytänteet, sekä esitysmuoto-, kommentointi- ja nimeämiskäytänteet vaihtelevat vähänlaisesti. Kehittäjät useimmiten siis noudattavat projekteissa määrättyjä käytänteitä tunnollisesti, kun sellaiset on jo hyvissä ajoin jouduttu kiinnittämään. Kiinnostavaa on se, että periaatteiden noudattamatta jättäminen johtaa helposti joko projektin koordinoijan tai muiden vertaiskehittäjien nopeisiin palautteisiin huonoista käytänteistä. Yleisesti sovittujen periaatteiden yhtenäisyyttä suojaa myös se, että lähdekoodisyötteitä pystytään kontrolloimaan helposti, etenkin monipuolisen työvälineistön avustuksella.

Näyttäisi erityisesti siltä, että ongelmallisempia ovat keskisuuret ja alkuvaiheessa olevat avoimen lähdekoodin projektit. Käytänteiden vakiinnuttaminen eli niiden ylöskirjaaminen ja aktiivisesti tiettäväksi teettäminen mahdollisimman aikaisessa vaiheessa kaikille kehittäjille on aina sekä ohjelmiston että kehittäjien etu. Useamman kehittäjän projekteissa lähdekoodivarastoon luonnollisesti muodostuu toisistaan erilaisia käytänteitä.

Nämä saattavat suuresti toisistaan poiketessaan vaikuttaa voimakkaastikin lähdekoodin ylläpidettävyyteen. Avoimen lähdekoodin lähestymistavassa poikkeavuudet on kuitenkin mahdollista harmonisoida ja korjata hyvän ylläpidettävyyden takaamiseksi.

Koodirakenteen ongelmat voivat pahimmassa tapauksessa lisätä ohjelmiston kompleksisuutta huomattavasti. Esimerkiksi nimeämiskäytänteiden vaihtelu aiheuttaa ongelmia ohjelmiston komponenttien integroinnissa järjestelmätasoon. Toisaalta esimerkiksi virheellisten kommenttien jääminen ohjelmistoon erinäisten muokkaustoimenpiteiden jälkeen voi altistaa varsinkin kompleksisten rakenteiden väärinymmärtämiselle.

Ongelmia avoimessa lähdekoodissa voivat lisäksi aiheuttaa eri käyttöjärjestelmät ja useat erilaiset konekieliset kääntäjät. Esimerkiksi projektin lähdekoodissa käytetty merkistö voi toisissa käyttöjärjestelmissä tai kääntäjissä olla erilainen, jolloin muun muassa käännösten tuottaminen voi vaatia lähdekoodeihin jonkinlaisia etukäteismuutoksia.

4.2.3 Dokumentaatio

Kohdealueen ja ohjelmiston tuntemuksella on huomattava vaikutus ylläpidettävyyteen. Muun muassa Hordijk ja Wieringa (2005, 388) ovat esittäneet, että korkealaatuinen dokumentaatio vaikuttanee suoraan ylläpidettävyyden kohoamisena. Avoimen lähdekoodin mukaisessa ohjelmistotuotannossa kohdealue on useimmiten erittäin hyvin tiedossa, mutta vain implisiittisesti.

Dokumentaatio on useissa eri yhteyksissä ollut kaikista vähiten arvostettu ohjelmistotuotannon eri osa-alueista, sillä kehittäjät ovat nähneet sen johtavan asioiden ”kahteen kertaan” tekemiseen. Perinteisesti onkin ollut osittain väärä lähtökohta asettaa lähdekoodin kehittäjät ensisijaisiksi dokumentaation

tuottajiksi, sillä he eivät yleensä näe ohjelmiston prosesseja epäselvinä tai erikseen kuvaamista vaativina asioina.

Dokumentaation on lähdettävä osaltaan käyttäjä-näkökulmasta, jotta dokumentaation käytännön hyöty saataisiin maksimoitua. Tämä seikka painanee eniten avoimen lähdekoodin ohjelmistojen kehityksessä, sillä lähestymistavassa näkökulma dokumentaatioon ei suoraan ole vain pelkästään käyttäjä-näkökulma vaan kehittäjä-käyttäjä-näkökulma. Tämän vuoksi on helppo ymmärtää, miksi kattavan ja luotettavan dokumentaation tuottaminen on avoimen lähdekoodin lähestymistavassa hankalaa. Dokumentaation olemassaolo on luonnollisesti tärkeintä silloin, kun ”hiljaista tietoa” ei voida siirtää kehittäjiltä toisille ja järjestelmän pätevä tuntemus riippuu siitä.

Dokumentaation merkitystä ylläpidettävyydelle voi avoimen lähdekoodin tapauksessa tosin minimoida se, että kehittäjien näkökulmasta katsottuna lähdekoodi nähdään usein selkeänä ja hyvin ylläpidettävänä lähdekoodina. Toisaalta myös yhteisön tuoma kommunikointiympäristö mahdollistaa ihmiselle luontaisen oppimisympäristön, jonka avulla oikea informaatio siirtyy kehittäjien kesken nopeaan tahtiin. Näin kehittäjille suunnatulle dokumentaatiolle ei näyttäisikään olevan varsinaista käytännön tarvetta.

Avoimen lähdekoodin ohjelmistojen dokumentaatio ja sen laatu on luonnollisesti hyvin subjektiivinen asia. Se on riippuvainen sekä olemassa olevasta projektista ja sen piirteistä, mutta varsinkin dokumentoijan näkökulmasta, kirjoitustaidoista ja luonteenpiirteestä. Perinteisesti dokumentoijan näkökulma on avoimen lähdekoodin mukaisessa ohjelmistotuotannossa ollut sekä toisaalta hyvin tekninen että toisaalta niukasti asioita kuvaava.

Avoimen lähdekoodin lähestymistavassa päädytään helposti dokumentoimaan vain ohjelmistojen yleisiä piirteitä, koska muun muassa monien toiminnallisuuksien pysyvyydestä ei useinkaan ole tarkkaa selvyyttä. Toisaalta

dokumentaation tekemättä jättäminen johtaa nopeasti siihen, ettei dokumentaatioprosessia lopulta jakseta viedä enää ollenkaan päätökseen.

Systemaattisen dokumentoinnin puute johtaa monissa projekteissa toisaalta dokumentaation sisällön ja kattavuuden että toisaalta dokumentaation kuvaavuuden epämääräisyyteen. Dokumentaatio ei lisäksi kaikissa tilanteissa ole paikkansa pitävää muun muassa siksi, että nopeasti muuttuvat toteutukset tekevät dokumentaatiosta helposti vanhentunutta. Avointen lähdekoodien projekteissa dokumentaatio on kuitenkin lähes poikkeuksetta helposti muokattavaa ja saatavissa olevaa.

Panostaminen dokumentaation tuottamiseen ja toisaalta tuotoksen laadukkuuteen takaavat ainakin sen, että avoimen lähdekoodin ohjelmistojen toiminta ja käyttäytyminen selvenee helpommin myös muille kuin vain projektin sisäisille tahoille. Dokumentaatioon on tulevaisuudessa panostettava enemmän, sillä avointen ohjelmistojen leviäminen yhä suurempiin konteksteihin kasvattaa validin dokumentaation tarpeellisuutta huomattavasti. Tämä tuleekin luultavasti olemaan yksi suurten avointen ohjelmistojen kaikkein hankalimmista ylläpidettävyyden parantamisen näkökulmista tulevaisuudessa.

5 JOHTOPÄÄTÖKSET JA YHTEENVETO

Tässä pro gradu -tutkielmassa on käsitelty ohjelmiston ylläpitoa, ylläpidettävyyttä ja avoimen lähdekoodin lähestymistavalle ominaisia piirteitä. Avoimen lähdekoodin mukaisen ohjelmistotuotannon ylläpidettävyyttä on arvioitu Omanin ja Hagemeisterin (1992) esittämien ylläpidettävyyden aspektien mukaisen tieteellisen luokittelun perusteella. Edeltävissä luvuissa käsiteltyjen erityispiirteiden perusteella esitetään tässä luvussa tutkielman yhteenveto ja seuraavia huomioita sekä johtopäätöksiä.

Ylläpidettävyys on voimakkaasti subjektiivinen mittausarvo, eikä vähiten siksi, että myös ohjelmisto on itsessään subjektiivinen kokonaisuus. Pelkästään tämän johdosta on ylläpidettävyyden objektiivinen arviointi vaikeaa. Koska toisaalta myös arviointimenetelmät ja -kriteerit vaihtelevat suuresti, voidaan todeta, ettei absoluuttista luokittelua mittausarvoineen voida koskaan ulottaa koskemaan jokaista mahdollista ohjelmistoa. Siksi olennaista onkin pystyä tekemään suuntaa antavia arvioita ylläpidettävyydestä asetettavien mittarien ja kriteerien avulla.

5.1 Ylläpidettävyysaspektit

Ohjelmiston kypsyyden ylläpidettävyysaspektit ovat avoimen lähdekoodin ohjelmistojen kannalta tärkeitä siinä suhteessa, että ne kertovat millaisia ohjelmistot ovat käyttää ja onko niissä epäjohdonmukaisuuksia tai paljon vikoja. Ne siis kertovat sen, ovatko ohjelmistot kypsiä ohjelmistoja, joiden turvallinen ja ongelmaton jokapäiväinen käyttö on mahdollista. Avoimen lähdekoodin mukainen ohjelmistotuotanto on nopealuonteista ohjelmistojen tuottamista, mutta sillä ei aina ole välttämättä mitään tarkkaa ja spesifiä suuntaa. Tämän vuoksi avoimen lähdekoodin ohjelmistot eivät välttämättä kypsy nopeassa tahdissa, vaikka ne niin kehittyisivätkin. Avoimen lähdekoodin ohjelmistot ovat luotettavia varsinkin silloin, kun projektin toimintaympäristö ja ohjelmiston kehitys on vakiintunut sekä projektin yhteisö koostuu useista

osaavista kehittäjistä. Tämän takaa osittain se tosiasia, että voimakkaalla rinnakkaisella virheenjäljityksellä pystytään minimoimaan ainakin ”pintapuoliset” luotettavuusongelmat.

Ohjelmiston järjestelmätason kontrolli- ja tietorakenteiden osalta ovat avoimen lähdekoodin ohjelmistot hyvin modulaarisia, jolloin sekä ongelmien rajaus että tehtävien jakaminen on helpompaa. Tämä toisaalta tarkoittaa sitä, että vaikka jakaminen avustaa ohjelmiston rakenteen ja arkkitehtuurin muotoutumista, ne eivät ole virtaviivaisia ohjelmiston kehittyessä. Liiallinen modulaarisuus voi osoittautua ongelmalliseksi joissain projekteissa. Lisäksi liian vapaamielisen modulaarisuuden aiheuttamat liialliset kytkennät voivat tulla haittatekijöiksi tai jopa huomattaviksi ongelmiksi, kuten on jo esimerkiksi Linuxin tapauksessa todettu tietorakenteiden osalta tapahtuneen.

Komponenttitason kontrollirakenteen monimutkaisuutta kuvaavien aspektien mukaisia piirteitä esiintynee useimmissa avoimen lähdekoodin ohjelmistoissa, mutta tällaisia ongelmallisuuden mittareita on kyseisessä lähestymistavassa mahdollista lieventää. Nämä ongelmat nimittäin hälvenevät useimmiten monien kyvykkäiden kehittäjien ansiosta. Esimerkiksi suljetun lähdekoodin ohjelmistonkehityksen on vaikea vastata yhtäpitävästi kehittäjien taitojen korkeaan tasoon ja kehittäjien nopeaan ja dynaamiseen vuorovaikutukseen. On tosin erittäin vaikea arvioida sitä, millaisia vaikutuksia kulloinkin tehdyt valinnat lopulta aiheuttavat kokonaisuudelle.

Komponenttitason tietorakenteen kohdalla ei helposti saavuteta täyttä varmuutta siitä, että jokainen ohjelmiston komponentti käyttäisi juuri identtisiä arvoja tietojen käsittelyissä. Varsinaisesti poikkeuksellisia tietojen käsittelytapoja ei avoimen lähdekoodin lähestymistavassa ole esimerkiksi suljetun lähdekoodin lähestymistapaan nähden. On kuitenkin mielenkiintoista se, ettei läheskään kaikkia avoimen lähdekoodin ohjelmistoja ole kehitetty oliokielillä, vaan muun muassa rakenteisilla kielillä. Tällaisten virheille ennalta

altistavien käytäntöjen ja menettelytapojen käyttö on vanhanaikaista, ja jopa tietyissä tapauksissa edesvastuutonta.

Ohjelmiston koodirakenteessa on varteenotettavissa avoimen lähdekoodin projekteissa harvoin ongelmia, sillä projektien lähdekoodin rakenteessa käytetyt käytänteet on määritelty hyvin ja kehittäjät myös noudattavat niitä. Ongelmakohdiksi voivat tältä osin osoittautua esimerkiksi monien eri järjestelmien tuottamien merkistöjen yhteensovittaminen.

Dokumentaation ylläpidettävyysaspektit ovat avoimen lähdekoodin ohjelmistojen suhteen hieman ongelmallisia, sillä dokumentaatioiden kehittyminen hyödyllisiksi näyttää vievän erittäin kauan. Toisaalta panostaminen dokumentaatioon on projekteissa vielä kovin matalalla tasolla. Dokumentaation, mutta myös käytettävyyden, parantaminen vaatii selvästi aiempaa enemmän käyttäjien (eikä vain pelkästään kehittäjä-käyttäjä) näkökulmien allokointia avoimen lähdekoodin projekteihin.

5.2 Ylläpidettävyys ja avoin lähdekoodi

Ylläpidettävyys avoimen lähdekoodin mukaisen ohjelmistotuotannon näkökulmasta on voimakkaasti subjektiivinen ja projektikohtainen mittari. Tutkielmassa esiteltyjen huomiokohtien perusteella avoimen lähdekoodin ohjelmistojen ei voida todeta olevan perustavanlaatuisesti erilaisia kuin esimerkiksi suljetun lähdekoodin ohjelmistot ylläpidettävyyden kannalta tarkasteltuna. Tämän määrittelee osaltaan se, että Lehmanin ohjelmiston evoluution lait näyttäisivät toteutuvan voimakkaasti myös useimpien avoimen lähdekoodin ohjelmistojen tapauksessa. Mielenkiintoista on kuitenkin se, että avoimen lähdekoodin lähestymistapa pystyy näitä lakeja ainakin joiltain osin rikkomaan. Lisäksi kehittäjien voimakas lisääntyminen ei lähestymistavassa näyttäisi johtavan kaaokseen kuuluisan Brooksin lain mukaisesti (Raymond 2001, 35).

Vaikkei avoimen lähdekoodin ohjelmistojen ylläpidettävyyden voidakaan tutkielman piirissä todeta olevan poikkeuksellista muihin ohjelmistoihin nähden, niin on kuitenkin muistettava, etteivät esitellyt tulokset ole aina yleistettävissä. Lisäksi suurempi tutkimuskantapooli olisi huomattavasti parantanut sekä käsiteltävien kokonaisuuksien että esitettyjen johtopäätösten validiteettia. Koska toisaalta tutkielman näkökanta on verrattavan uusi, on kaikki resurssit pyritty tutkimusmenetelmän puitteissa käyttämään mahdollisimman kattavasti hyväksi.

Avoimen lähdekoodin koordinoijan (tai koordinoijien) roolia ei voi korostaa liikaa, sillä hänen valinnoillaan on valtava vaikutus avoimen ohjelmiston ”onnistumiseen” nähden. Koordinoija on siis keskeisessä asemassa projektin ylimpänä päättävänä elimenä siihen, millaiseksi lopputuote kehittyy. Hänen on sekä toisaalta osattava jakaa projektin vastuut juuri oikeille ihmisille, jotka toimivat pääkehittäjien rooleissa, että toisaalta pystyttävä myös erottamaan vaihtoehtoisten toteutusten hyvät ja huonot piirteet.

Avoimen lähdekoodin mukainen ohjelmistotuotanto kilpailee paikastaan yhtenä ohjelmistojen kehitysmuodoista erityisesti sen suomilla vapauksilla johtamistavoissa. Lähestymistavan johtamisstrategia kannustaakin useimmiten vapauteen ja omaehtoisuuteen. Kontrollien puute toisaalta haastaa kehittäjiä ja toisaalta syventää heidän omistautumistaan projektiin. Lisäksi kehityksen löyhät rakenteet vähentävät kehittäjien rutinoitumista, jolloin ajalliset ja laadulliset tavoitteet ovat kehittäjien omien toimintojen ja päätösten mukaisia. Motivaatio syntyy tarpeesta ja halusta, joten aikaansaannosten palkitseminen syventää motivaatiota. Koska kehittäjien motivaationa ei kuitenkaan selvästi ole raha, täytynee kehittäjien perustarpeiden olla muilta osin täytetyt.

Keino, jolla avoimen lähdekoodin mukaisen ohjelmistotuotannon vaihtelevia käytänteitä ja rakenteita voisi pyrkiä yhtenäistämään, olisi esimerkiksi prosessien standardointi. Tämä tuskin avoimen lähdekoodin kehittäjille näyttäisi luonnolliselta toiminnalta, sillä standardit käytännössä rajoittaisivat

kehittäjien vapautta tehdä omaehtoisia päätöksiä. Näyttääkin edelleen hyvin todennäköiseltä, että avoimen lähdekoodin lähestymistavan luonteenomaisuus suuresti vaihtelevista käytänteistä jatkunee vielä kauan.

Avoimen lähdekoodin lähestymistavan informaalisuus ja vapaamuotoinen, dynaaminen, elävä kehitys on ehdottomasti yhteisöjen ja kehittäjien vapauden ohella yksi määrittävistä tekijöistä, jotka luovat mainiot edellytykset innovatiiviselle ja kehittävälle yhteistoiminnalle. Vaikka avoimen lähdekoodin mukainen ohjelmistotuotanto onkin vielä suhteellisen nuori innovaatio, on olemassa selviä merkkejä siitä, ettei se ole lähiaikoina poistumassa ohjelmistoalalta. Erityistä avoimen lähdekoodin mukaisessa ohjelmistotuotannossa on se, että sille näyttää olevan selkeä tarve. Vielä on tosin liian aikaista arvioida, missä laajuudessa ja mittakaavassa sekä miten ja millaiseksi entiteetiksi avoimen lähdekoodin lähestymistapa tulevaisuudessa vakiintuu.

Avoimen lähdekoodin mukaisen ohjelmistotuotannon tuottamat ohjelmistot soveltuvat etenkin yleisiksi ja avoimiksi alustoiksi, jolloin eri ratkaisuteknologioiden kilpailu on hyödytöntä, ja siis teknologioiden vakiintuminen on olennaista. Toisaalta on täysin mahdollista kehittää avoimen lähdekoodin ohjelmistoja tutkielmassa esitetystä ”perusmallista” (eli vapaasta ja avoimesta ohjelmistokehityksestä) poikkeavalla tavalla. Tällöin on kuitenkin huomioitava, että tästä perusmallista poikkeavan ohjelmistokehityksen ja evoluution kautta tuotetut ohjelmistot eivät välttämättä ole vertailukelpoisia tutkielmassa aidoiksi avoimen lähdekoodin ohjelmistoiksi käsitettyihin ohjelmistoihin nähden.

Ylläpidettävyyden arviointi on toisaalta vielä usein toissijaisessa asemassa, koska se jää usein yksittäisten kehittäjien implisiittisesti tekemäksi. Avoimen lähdekoodin lähestymistapa on tosin onneksi antamassa selvästi enemmän painoarvoa ohjelmistojen ylläpidettävyydelle, sekä implisiittisesti että eksplisiittisesti, etenkin lähestymistavalle ominaisen avoimuuden piirteen

vuoksi. Jatkossa on vielä käsiteltävä esimerkiksi sitä, millä menetelmillä avoimen lähdekoodin ohjelmistoja voidaan mitata juuri ohjelmiston kypsyyden kannalta.

Ylläpidettävyyden näkökulmasta avoin lähdekoodi on siis mielenkiintoisessa asemassa. Kuten tutkielmassa on aiemmin esitetty, vaikka avoimen lähdekoodin laatu voi vaihdella suuresti, niin ylläpidettävyys voi parhaillaan olla myös erinomaista. Tulevien kehittäjien on kuitenkin erittäin suositeltavaa tiedostaa kaikki tässä tutkielmassa tehdyt ja esitetyt huomiot, jolloin juuri ongelmallisten kohtien käsittelyä voidaan parantaa käytännössä. Esimerkiksi aiemmin esitettyjen suurimpien ongelmien systemaattinen käsittely ja ratkaiseminen on omiaan parantamaan huomattavasti yleistä avoimen lähdekoodin mukaisen ohjelmistotuotannon, ja näin lopulta myös itse tuotteiden, laatua.

Jatkotutkimusaiheena olisi mielenkiintoista tutkia Omanin ja Hagemeisterin esittämien ylläpidettävyyden metriikoiden soveltuvuutta avoimen lähdekoodien ohjelmistoille. Näin ylläpidettävyyden arviointia voitaisiin toteuttaa paremmin lähestymistavalle soveltuvin mittauskriteerein. Tärkeää olisi tutkia syvemmin myös juuri avoimelle lähdekoodille spesifisiä ylläpidettävyysaspekteja. Eri ohjelmointiparadigmojen ylläpidettävyyden aspektien tarkempi yhteensovittaminen näyttäisi olevan eräs tärkeä näkökohta edellä mainitulle tutkimusaiheelle. Jatkossa olisi lisäksi mielenkiintoista tutkia lähemmin avoimen lähdekoodin mukaisen ohjelmistotuotannon työvälineitä. Etenkin CVS -versionhallintaohjelmiston merkitystä avoimen lähdekoodin lähestymistavalle olisi mielekästä tutkia, sillä työväline näyttää olevan useimmissa projekteissa erittäin keskeisessä asemassa.

Avoimen lähdekoodin ohjelmiston ylläpidettävyys on erittäin mielenkiintoinen tutkimusalue, sillä kyseistä näkökohtaa arvioimalla on mahdollista ainakin osittain todentaa avoimen lähdekoodin mukaisen ohjelmistotuotannon toimivuutta yhtenä tehokkaana keinona tuottaa tietynlaisia ohjelmistoja.

Aihealueen tutkimus todentaa vähintään sen, ettei liiallinen yleinen tyytyväisyys käytettyihin nykymenetelmiin ainakaan vähennä todellisia sekä toisaalta asenteisiin että toisaalta käytäntöihin liittyviä muutostarpeita. On kuitenkin erittäin todennäköistä, ettei kaikille avoimen lähdekoodin ohjelmistoille soveltuvaa yleistä ylläpidettävyyden arviointiin perustuvaa menetelmää tai luokittelua ole mahdollista löytää. Tämä ei kuitenkaan vähennä sitä tarvetta, etteikö tutkielman näkökohtaa tarvittaisi. Käytäntöjen muutostarpeiden tunnistaminen hyödyntää ylläpidettävyyttä ja lopulta myös ohjelmistotuotantoprosessia.

LÄHDELUETTELO

Asklund U., Bendix L. 2002. A study of configuration management in open source software projects. IEE Proceedings On Software. Volume 149, No. 1, February, 40-46.

Bar M., Fogel K. 2003. Open source development with CVS, 3rd edition [online]. Paraglyph Press, Inc [viitattu 2.3.2006]. Saatavilla www-muodossa <http://cvsbook.red-bean.com/OSDevWithCVS_3E.pdf>.

García M. J. B., Alvarez J. C. G. 1996. Productive maintainability. ACM SIGSOFT Software Engineering Notes. Volume 21, Issue 2, March, 89-91.

Briand L. C., Bunse C., Daly J. W. 2001. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Transactions on Software Engineering. Volume 27, No. 6, June, 513-530.

Capiluppi A., Lago P., Morisio M. 2003. Characteristics of open source projects. Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03), Benevento, Italy. March, 317-330. IEEE.

Chapin N., Hale J. E., Khan K., Ramil J. F., Tan W.G. 2001. Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice. Volume 13, Issue 1, January, 3-30.

Coleman D., Lowther B., Oman P. 1995. The application of software maintainability models in industrial software systems. Journal of Systems and Software. Volume 29, No. 1, April, 3-16.

Dagpinar M., Jahnke J. H. 2003. Predicting maintainability with object-oriented metrics – An empirical comparison. Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), Victoria, Canada. November, 155-164. IEEE.

Di Lucca, G. A., Fasolino A. R., Tramontana P., Visaggio C. A. 2004. Towards the definition of a maintainability model for web applications. Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04), Tampere, Finland. March, 279-287. IEEE.

DiBona C., Ockman S., Stone M. 1999. Open sources: Voices from the open source revolution. O'Reilly Media.

Fitzgerald B. 2004. A critical look at open source. IEEE Computer. Volume 37, Issue 7, July, 92-94.

Gacek C., Arief B. 2004. The many meanings of open source. IEEE Software. Volume 21, No. 1, January/February, 34-39.

Glass R. L. 2004. A look at the economics of open source. Communications of the ACM. Volume 47, Issue 2, February, 25-27.

Godfrey M. W., Qiang T. 2000. Evolution in open source software: A case study. Proceedings of the 2000 International Conference on Software Maintenance (ICSM'00), San Jose, California. October, 131-142. IEEE.

Haworth D. A., Sharpe S., Hale D. P. 1992. A framework for software maintenance: A foundation for scientific inquiry. Journal of Software Maintenance: Research and Practice. Volume 4, Issue 2, June, 105-117.

Hordijk W., Wieringa R. 2005. Surveying the factors that influence maintainability: research design. Foundations of Software Engineering: Proceedings of the 10th European Software Engineering Conference, Lisbon, Portugal. September, 385-388. ACM Press.

IEEE Computer Society. 1990. IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries. Standards Coordinating Committee of the IEEE Computer Society.

IEEE Computer Society. 1998. IEEE standard for software maintenance (IEEE std 1219-1998). Software Engineering Standards Committee of the IEEE Computer Society.

Kiran G. A., Haripriya S., Jalote P. 1997. Effect of object orientation on maintainability of software. Proceedings of the International Conference on Software Maintenance (ICSM'97), Bari, Italy. October, 114-121. IEEE.

Koponen T., Hotti V. 2005a. Open source software maintenance process framework. International Conference on Software Engineering: Proceedings of the 5th Workshop on Open Source Software Engineering (5-WOSSE), St. Louis, Missouri. May, 30-34. ACM Press.

Koponen T., Hotti V. 2005b. Defects in open source software maintenance – two case studies: Apache and Mozilla. Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP'05), Las Vegas, Nevada. June, 688-693. CSREA Press.

Koskinen J., Sakkinen M., Paakki J. 2001. Ohjelmistotekniikka (opetusmonisteita OM-10), 2. täydennetty painos. Jyväskylän yliopiston yliopistopaino.

Lehman M. M., Perry D. E., Ramil J. F. 1998. Implications of evolution metrics on software maintenance. Proceedings of the International Conference on Software Maintenance (ICSM'98), Bethesda, Maryland. November, 208-217. IEEE.

Leitch R., Stroulia E. 2003. Assessing the maintainability benefits of design restructuring using dependency analysis. Proceedings of the 9th International Software Metrics Symposium (METRICS'03), Sydney, Australia. September, 309-322. IEEE.

Mockus A., Fielding R. T., Herbsleb J. D. 2002. Two case studies of open source software development: apache and mozilla. ACM Transactions on Software Engineering and Methodology. Volume 11, No. 3. July, 309-346.

Nichols D. M., Twidale M. B. 2002. Usability and open source software [online]. MIT Working Paper Series [viitattu 20.3.2006]. Saatavilla www-muodossa <http://opensource.mit.edu/papers/nicholstwidale1.pdf>.

Oman P., Hagemeister J. 1992. Metrics for assessing a software system's maintainability. Proceedings of the Conference on Software Maintenance, Orlando, Florida. November, 337-344. IEEE.

Open Source Initiative (OSI). 2005. The open source definition [online], [viitattu 5.12.2005]. Saatavilla www-muodossa <http://www.opensource.org/docs/definition.php>.

Potdar V., Chang E. 2004. Open source and closed source software development methodologies. The 26th International Conference on Software Engineering (ICSE 2004): Proceedings of the 4th Workshop on Open Source Software Engineering, Edinburgh, Ireland. May, 105-109. IEEE.

Raja U., Barry E. 2005. Investigating quality in large-scale open source software. International Conference on Software Engineering: Proceedings of the 5th Workshop on Open Source Software Engineering (5-WOSSE), St. Louis, Missouri. May, 43-46. ACM Press.

Raymond E. S. 2001. The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary, revised edition. O'Reilly Media, Inc.

Samoladas I., Stamelos I., Angelis L., Oikonomou A. 2004. Open source software development should strive for even greater code maintainability. Communications of the ACM. Volume 47, No. 10, October, 83-87.

Schach S. R., Jin B., Wright D. R., Heller G. Z., Offutt A. J. 2002. Maintainability of the Linux kernel. IEE Proceedings Software. Volume 149, No. 1, February, 18-23.

Schach S. R., Offutt A. J. 2002. On the nonmaintainability of open-source software. Proceedings of International Conference on Software Engineering: 2nd Workshop on Open Source Software Engineering, Orlando, Florida. May, 1-3. ACM Press.

Spinellis D., Szyperski C. 2004. How is open source affecting software development? IEEE Software. Volume 21, Issue 1, January/February, 28-33.

Stewart K. J., Darcy D. P., Daniel S. L. 2005. Observations on patterns of development in open source software projects. International Conference on Software Engineering: Proceedings of the 5th Workshop on Open Source Software Engineering (5-WOSSE), St. Louis, Missouri. May, 62-66. ACM Press.

Swanson E. B. 1999. IS "maintainability": Should it reduce the maintenance effort? The Database for Advances in Information Systems. Volume 30, No. 1, 65-76.

Tahvildari L., Kontogiannis K., Mylopoulos J. 2001. Requirements-driven software re-engineering framework. Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), Stuttgart, Germany. October, 71-80. IEEE.

Välimäki M. 2004. A practical approach to the problem of open source and software patents. European Intellectual Property Review. Volume 26, Issue 12, December, 523-527.

Välimäki M. 2005. The rise of open source licensing: A challenge to the use of intellectual property in the software industry [online]. Turre Publishing [viitattu 14.2.2006]. Saatavilla www-muodossa <http://pub.turre.com/openbook_valimaki.pdf>.

Warsta J., Abrahamsson P. 2003. Is open source software development essentially an agile method? International Conference on Software Engineering (ICSE'03): 3rd Workshop on Open Source Software Engineering, Portland, Oregon. May, 143-147. ACM Press.

Wilde N., Matthews P., Huitt R. 1993. Maintaining object-oriented software. IEEE Software. Volume 10, No. 1, January/February, 75-80.

Ying A. T. T., Murphy G. C., Ng R., Chu-Carroll M. C. 2004. Predicting source code changes by mining change history. IEEE Transactions on Software Engineering. Volume 30, No. 9, September, 574-586.

Zhou Y., Davis J. 2005. Open source software reliability model: An empirical approach. International Conference on Software Engineering: Proceedings of the 5th Workshop on Open Source Software Engineering (5-WOSSE), St. Louis, Missouri. May, 67-72. ACM Press.