Tero Lehmonen

Ohjelmistojen ylläpitäjien tietotarpeet ja niiden arviointitavat

Tietojärjestelmätieteen
kandidaatin tutkielma
4.12.2006

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

Tiivistelmä

Lehmonen, Tero

Ohjelmistojen ylläpitäjien tietotarpeet ja niiden arviointitavat

Jyväskylä: Jyväskylän yliopisto, 2006.

29 s.

Kandidaatin tutkielma

Ylläpidon kustannukset ovat huomattavasti muita ohjelmiston elinkaaren vaiheita korkeammat. Ylläpidon osuus kustannuksista on ollut tiedossa jo useamman vuosikymmenen ajan, mutta lukuisista tutkimuksista huolimatta ylläpidon kustannukset ovat edelleen korkeat. 1990-luvun loppupuolella vuosituhannen vaihtumisesta aiheutuneet ylläpito-ongelmat tuottivat huomattavan määrä tutkimuksia ohjelmistojen ylläpidosta. Tutkimusten tuloksena saatiin parempi käsitys ylläpitoon liittyvistä tekijöistä, kuten ohjelman ymmärtämisestä ja ylläpitäjien tietotarpeista.

Ohjelmistojen ylläpidossa ohjelman ymmärtäminen on eniten aikaa vievä vaihe. Ohjelman ymmärtäminen on myös välttämätöntä, jotta ylläpito voidaan tehdä turvallisesti aiheuttamatta ohjelmaan vikoja. Ohjelman ymmärtäminen vaatii ylläpitäjien tietotarpeiden täyttämistä. Tässä tutkielmassa tehdään kirjallisuuskatsaus ohjelmistojen ylläpitoon, ohjelmien ymmärtämiseen ja ylläpidon työvälineiden käyttämiin tekniikoihin. Lisäksi käydään lävitse tutkimuksia, joissa on arvioitu ylläpidon työvälineitä. Tavoitteena on löytää viitekehys, jonka avulla on mahdollista arvioida ylläpidon työvälineiden tukea ylläpitäjien tietotarpeille.

AVAINSANAT: ylläpito, ohjelman ymmärtäminen, ylläpitäjien tietotarpeet

Sisällysluettelo

1 Johdanto

Ylläpito on eniten resursseja vievä vaihe ohjelmiston elinkaaresta (Pressman 2000, Sommerville 2001). Tämä seikka on huomattu jo 1970-luvulla ja aihetta on tutkittu siitä lähtien, mutta ylläpidon osuus on edelleen pahimmillaan suurempi kuin muiden vaiheiden kustannukset yhteensä (Zelkowitz ym. 1979, Erlikh 2000). Ylläpidon kustannuksiin vaikuttaa moni tekijä, mutta eniten aikaa vievä ylläpidon vaihe on ohjelman ymmärtäminen (Fjeldstad & Hamlen 1983, Standish 1984, Bennett & Vaclav 2000).

Ohjelman toiminnan ymmärtäminen on ohjelman ylläpidolle välttämätöntä, jotta ylläpito voidaan tehdä turvallisesti (Choi & von Mayrhauser 1994). Ilman täyttä ymmärrystä ohjelman toiminnasta ja ylläpidossa tehtävien muutosten vaikutuksista voidaan aiheuttaa vikoja, joiden korjaaminen vaatii lisää työtä. Erityinen ongelma-alue ohjelmistojen ylläpidossa on suuret perinneohjelmistot, joiden dokumentaatio voi olla vaillinaista ja ohjelmistoarkkitehtuuri rappeutunut. Tämänlaisten ohjelmien toiminnan ymmärtäminen voi olla erityisen vaikeaa ja aikaa vievää. Ohjelman ymmärtäminen vaatii ylläpitäjän tietotarpeiden täyttämistä (Koskinen ym. 2004). Ohjelmien ylläpitoa helpottamaan on kehitetty työvälineitä, jotka pyrkivät tukemaan ylläpitäjän eri tietotarpeita ja siten auttamaan ohjelman ymmärtämisessä.

Tässä tutkimuksessa on tarkoitus selvittää mitä on ohjelmistojen ylläpito ja siihen liittyviin ohjelman ymmärtäminen sekä ohjelmistojen ylläpitotyövälineiden käyttämät tekniikat. Lisäksi tarkoituksena on käydä lävitse tutkimuksia, joissa on arvioitu ohjelmistojen ylläpidon työvälineitä. Tavoitteena on löytää menetelmä, jonka avulla on mahdollista arvioida ylläpidon työvälineiden tukea ylläpitäjien tietotarpeille.

Tämän tutkimuksen rakenne jakautuu kahteen osaan. Ensimmäisessä osassa selvitetään mitä on ohjelmistojen ylläpito ja siihen liittyvät ohjelman ymmärtäminen sekä ylläpitäjän tietotarpeet. Lisäksi ensimmäisessä osassa käydään lävitse ohjelmistojen ylläpidon työvälineiden käyttämiä tekniikoita. Toisessa osassa tehdään kirjallisuuskatsaus tutkimuksista, joissa on arvioitu ohjelmistojen ylläpidon työvälineitä.

2 Ohjelmistojen ylläpito

Tässä luvussa käydään lävitse ohjelmiston ylläpidon eri osa-alueita. Tutkielmassa selvitetään mitä on ohjelmistojen ylläpito ja miten ylläpidolle välttämätön ohjelman ymmärtäminen tapahtuu sekä mitä ohjelmistojen ylläpidon tietotarpeet ovat ja lopuksi annetaan yleinen kuvaus ohjelmistojen ylläpitotyövälineistä sekä niiden käyttämistä tekniikoista.

2.1 Ohjelmistojen ylläpito ja evoluutio

Haikonen (2006) on suomentanut IEEE:n (1998, 4) määritelmän ohjelmiston ylläpidosta seuraavasti:

> 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.

Eli käytännössä ohjelmistojen ylläpito tarkoittaa muutosten tekemistä valmiiseen ohjelmistoon sen julkaisun jälkeen. Ylläpito on tämän määritelmän lisäksi jaettu edelleen eri ylläpidon tyyppeihin riippuen sen tarkoituksesta. Swanson (1976) määrittelee ohjelmistojen ylläpidolle neljä tyyppiä:

1. *Korjaava ylläpito* (corrective maintenance) on ylläpitoa, jossa korjataan ohjelmistosta löytyneitä vikoja. Korjauksen kustannus riippuu monesti siitä missä vaiheessa virhe on tehty. Ohjelmiston kehityksen alkuvaiheessa vaatimuksiin ja suunnitteluun tehdyt virheet voivat olla erittäin kalliita korjata, kun taas loppuvaiheen ohjelmointi virheet halvempia korjata.

2. *Mukauttava ylläpito* (adaptive maintenance) on ylläpitoa, jossa ohjelmistoa mukautetaan toimimaan eri ympäristöissä, kuten käyttöjärjestelmissä tai laitealustoilla. Käyttäjälle näkyvä ohjelman toiminta ei muutu mukauttavassa ylläpidossa.

3. *Parantava ylläpito* (perfective maintenance) on ylläpitoa, jossa ohjelmistoon toteutetaan uusia toiminnallisia ja ei-toiminnallisia vaatimuksia. Uudet vaatimukset tulevat asiakasyritysten tarpeiden muutoksista.

4. *Ennalta ehkäisevän ylläpito* (preventive maintenance) on ylläpitoa, jonka tarkoitus on muuttaa ohjelmaa helpommaksi ylläpitää tulevaisuudessa.

Swansonin vuonna 1975 määrittelemät ylläpidon tyypit ovat edelleen käytössä nykypäivän kirjallisuudessa (Pressman 2000, Sommerville 2001). *Ohjelman evoluutiolla* tarkoitetaan ohjelman muuttumista, mutta sen määritelmästä ei olla täysin yksimielisiä. 1) Suppeamman näkemyksen mukaan ohjelman evoluutiolla tarkoitetaan ohjelman muuttumista ylläpidon seurauksena (Lehman 1980). 2) Toinen laajempi näkemys on, että ohjelman evoluutio käsittää ohjelman koko elinkaaren aikana siihen tehdyt muutokset (Harsu 2003, 20).

Ohjelmistojen ylläpito koostuu useasta eri vaiheesta. Yau ym. (1978) ovat määritelleet ylläpidolle seuraavat vaiheet:

- Muutosvaatimus
- Suunnitteluvaihe
  - Ohjelman ymmärtäminen
  - Muutosten vaikutusten analysointi
- Muutosten tekeminen
  - Ohjelman uudelleen strukturointi muutoksia varten
  - Muutosten tekeminen
- Muutosten verifiointi ja validointi
- Uudelleen dokumentointi

Ohjelman ymmärtäminen on ohjelmistojen ylläpidossa eniten aikaa vievä vaihe (Fjeldstad & Hamlen 1983, Standish 1984, Bennett & Vaclav 2000). Ohjelman ymmärtämistä on tutkittu ymmärtämismallien ja niistä johdettujen ylläpidon tietotarpeiden avulla. Näiden tutkimusten pohjalta on kehitetty ohjelmistojen ylläpitotyövälineitä, joiden tarkoitus on auttaa ohjelman sekä siihen tehtävien muutosten vaikutusten ymmärtämistä.

2.2 Ohjelman ymmärtäminen ja ymmärtämismallit

Ohjelmistojen ylläpidossa ohjelman ymmärtäminen on välttämätöntä, jotta muutokset ohjelmaan voidaan tehdä turvallisesti (Choi & von Mayrhauser 1994). Muutosten tekeminen ilman täyttä ymmärrystä niiden vaikutuksista voi aiheuttaa vikoja ohjelmassa ja johtaa suureen määrään lisätyötä. Ohjelman ymmärtäminen vie yli 50% ohjelmistojen ylläpidon resursseista, joten se vastaa suurimmasta osasta ylläpidon kustannuksista (Fjeldstad & Hamlen 1983, Standish 1984, Bennett & Vaclav 2000). Helpottamalla ja nopeuttamalla ohjelman ymmärtämistä voidaan siten merkittävästi vaikuttaa ylläpidon kustannuksiin.

Ohjelman ymmärtämisen tavoitteena on, että ylläpitäjän mielikuva ohjelmasta vastaa ohjelman todellista rakennetta (Harsu 2003, s. 105). Ohjelman ymmärtämistä on tutkittu *ohjelman ymmärtämismallien* avulla, joiden tarkoitus on kuvata ohjelman

ymmärtämisprosessia ja siihen liittyviä tekijöitä. Ohjelman ymmärtämismallit koostuvat ymmärtämisstrategioista ja mentaalimalleista. *Mentaalimalli* kuvaa ohjelmoijan ymmärrystä kohdeohjelmasta ja y*mmärtämisstrategia* on ohjelmoijan käyttämä lähestymistapa ohjelman ymmärtämiseen. Tämän hetken ohjelman ymmärtämismallit voidaan jakaa kolmeen eri luokkaan niiden ymmärtämisstrategioiden mukaan: kokoaviin, osittaviin ja tilanteen mukaisiin ymmärtämismalleihin.

2.2.1 Osittavat ohjelman ymmärtämismallit

Osittavaa ymmärtämismallia käyttävä ohjelmoija hyödyntää omaa tietämystä sovellusalueesta ja muista vastaavanlaisista ohjelmista yrittäessään ymmärtää ohjelman toimintaa. Ohjelmoijalla on jonkinlainen alustava käsitys ohjelman toiminnasta, jota hän täydentää lukiessaan lähdekoodia. Ohjelmoija etenee lähdekooditasolla ensin tarkastelemalla kutsuhierarkian ylätasoa ja siirtyy vain tarvittaessa kutsuhierarkiassa alaspäin. Osittavia ohjelman ymmärtämismalleja pidetään opportunistisena strategiana, koska lähdekoodia luetaan sitä mukaan kun tarvitaan, jolloin suuri osa siitä jää lukematta. Osittavaa strategiaa käytetään yleensä silloin kun ohjelmoijalla on hyvä tietämys ohjelman aihealueesta tai muista vastaavanlaisista ohjelmista. Kaksi tunnettua ymmärtämismallia, jotka käyttävät osittavaa ymmärtämisstrategiaa ovat Brooksin malli, sekä Solowayn, Adelsonin ja Erhlichin malli. (von Mayrhauser & Vans 1995B)

Brooksin ymmärtämismallissa ohjelman ymmärtämistä pidetään ohjelman tekijän sovellusaluetietämyksen uudelleen luomisena. Ohjelmasta luodaan mentaalinen malli tekemällä hypoteeseja ohjelman toiminnasta sovellusaluetietämyksen avulla. Ohjelmoija tarkentaa, luo uusia ja hylkää hypoteesejaan sitä mukaan kun hän lukee dokumentaatiota ja lähdekoodia. (Brooks 1983)

Solowayn, Adelsonin ja Ehrlichin kehittämässä ymmärtämismallissa mentaalinen malli koostuu suunnitelmien ja päämäärien hierarkiasta. Ymmärtämisprosessi alkaa korkean tason päämäärästä, joka jakautuu alemman tason päämääriin joiden avulla pystytään saavuttamaan korkeamman tason päämäärä. Suunnitelmat ovat jaettu kolmeen eri ryhmään: strategisiin, taktisiin ja toteutussuunnitelmiin. Nämä eri suunnitelmat kuvaavat mentaalisen mallin eri tasoja ohjelmakoodista. (von Mayrhauser & Vans 1995B)

2.2.2 Kokoavat ohjelman ymmärtämismallit

Kokoavissa ohjelman ymmärtämismalleissa ohjelmoija muodostaa ylemmän tason abstraktioita ohjelmasta lukemalla lähdekoodia. Ohjelmoijat käyttävät kokoavaa ymmärtämismallia yleensä silloin kun ohjelman sovellusalue on vieras. Osittavia ja kokoavia ymmärtämismalleja voidaan pitää toistensa vastakohtina. Kaksi tunnettua kokoavaa ohjelman ymmärtämismallia ovat Penningtonin, sekä Shneidermanin ja Mayerin mallit. (von Mayrhauser & Vans 1995B)

Shneidermanin ja Mayerin (1979) kehittämässä ymmärtämismallissa mentaalimalli luodaan lohkomisprosessin avulla. *Lohkomisprosessissa (chunking)* lähdekoodia tallentuu lyhytkestoiseen muistiin sitä mukaan kun sitä luetaan. Lyhytkestoisesta muistista se muutetaan erilaisiin korkeamman tason abstraktioihin pitkäkestoiseen muistiin.

Toinen tunnettu ymmärtämismalli, jossa käytetään kokoavaa strategiaa, on Penningtonin kehittämä malli. Penningtonin mallin mukaan ymmärtämisprosessissa luodaan kaksi eri mentaalimallia: ohjelma- ja tilannemalli. Ohjelmamalli luodaan ennen tilannemallia *suunnannäyttäjien (beacons)* avulla, jotka ovat helposti havaittavia lähdekoodin osia. Ohjelmamalli on kontrollirakenneabstraktio, joka muodostetaan lohkomalla ja ristiinviittaamisella. Tilannemalli on tietovuo- ja funktionaalinen abstraktio, joka luodaan ristiinviittauksen avulla. (von Mayrhauser & Vans 1995B)

2.2.3 Tilanteen mukainen ohjelman ymmärtämismalli

Tilanteen mukaisten ohjelman ymmärtämismallien mukaan ohjelman ymmärtämiseen tarvitaan sekä kokoavia että osittavia ymmärtämisstrategioita. Ohjelman ymmärtämismalleja, jotka sisältävät sekä kokoavia sekä osittavia ymmärtämisstrategioita ovat Letovskyn malli ja integroitu ymmärtämismalli (von Mayrhauser and Vans , 1995B).

Letovskyn malli koostuu kolmesta eri komponentista: tietämyskannasta, mentaalimallista ja omaksumisprosessista. Mentaalimalli koostuu kolmesta osasta: spesifikaatiosta, implementaatiosta ja annotaatiosta. Spesifikaatiotaso on mentaalimallin korkein taso, joka kuvaa ohjelman päämäärää. Implementaatio on alin taso ja se kuvaa lähdekoodin osia. Annotaatio yhdistää spesifikaation päämäärät ja implementaation lähdekoodin osat toisiinsa. Omaksumisprosessi voi tapahtua kokoavasti, osittavasti tai sitten niiden yhdistelmänä. (von Mayrhauser & Vans 1995B)

Integroitu ymmärtämismalli kostuu neljästä osasta: ohjelma-, tilanne ja osittavasta-mallista, sekä tietämyskannasta. Osittava malli pohjautuu Solowayn, Adelsonin ja Ehrlichin ymmärtämismalliin ja ohjelma- ja tilannemalli perustuvat taas puolestaan Penningtonin ymmärtämismalliin. Ohjelmoija käyttää näitä kolmea mallia yrittäessään ymmärtää lähdekoodia ja vaihtaa ymmärtämisstrategiasta toiseen aina tarvittaessa. Ymmärtämisprosessin aikana tietämyskantaan tallentuu ohjelmoijan ymmärrys ohjelman toiminnasta. Ohjelmoija käyttää myös tietämyskannasta aikaisempaa tietämystä hyväksi yrittäessään ymmärtää ohjelmaa. (von Mayrhauser & Vans 1995B)

2.3 Ylläpitäjien tietotarpeet

Ymmärtämismallien avulla on selvitetty mitä tietoa ylläpitäjät tarvitsevat ohjelman toiminnan ymmärtämiseen. Edellisessä luvussa esitetyt ohjelman ymmärtämismallit käyttävät syötteinä tietoa eri tietolähteistä, kuten esimerkiksi lähdekoodista ja dokumentaatiosta. Selvittämällä ylläpitäjien tietotarpeet voidaan kehittää työkaluja, jotka helpottavat ohjelman ymmärtämistä tarjoamalla ylläpitäjille heidän tarvitsemaa tietoa. (von Mayrhauser & Vans, 1995B)

von Mayrhauser ja Vans (1995A, 1995B, 1997A, 1997B & 1998) ovat tutkineet ylläpitäjien tietotarpeita ylläpidon eri tyypeissä. Tutkimukset kattavat korjaavan, mukauttavan ja parantavan ylläpidon. von Mayrhauserin ja Vansin tutkimuksia voidaan pitää tämän hetken kattavimpina ohjelmistojen ylläpidon tietotarpeista, koska niissä on huomioitu ylläpidon eri tyypit, sekä ylläpidon kohteena tutkimuksissa on ollut suuri ohjelmisto. Näiden tutkimusten mukaan ohjelmiston ylläpidon tietotarpeet eroavat ylläpidon tyypin ja ylläpitäjien välillä. Yleisimmin esiintyneet tietotarpeet ovat kuitenkin samat kaikissa tutkimuksissa vaikka niiden esiintymismäärissä on eroja. Taulukossa 2-1 on Koskisen ym. (2004) tekemä yhteenveto von Mayrhauserin ja Vansin tutkimusten tuloksista. Taulukko sisältää 24 yleisintä ylläpitäjien tietotarvetta järjestettynä niiden frekvenssin mukaan ylläpidossa. Taulukkoon on myös merkitty jokaisen tietotarpeen kohdalle X-kirjaimella ne tietolähteet, joiden avulla tietotarve voidaan täyttää. Taulukossa lyhenne S tarkoittaa staattista lähdekoodin analysointia, A lähdekoodin ajonaikaista analysointia, D dokumentaatiota ja H istunto historiaa.

	Tietotarve	**Frekvenssi**	**Tietolähde**			
			S	**A**	**D**	**H**
1	Sovellusalueen käsitteiden määrittelyt	68	X		X	
2	Tunnuksien (funktion, muuttujan tai tietotyypin) sijainti- ja käyttökohdat	54	X			
3	Yhdistetty sovellusalue- ohjelma-tilannemallitieto	41	X		X	X
4	Lista selatuista kohteista	33				X
5	Lista rutiineista, jotka kutsuvat tiettyä rutiinia	29	X			
6	Yleinen rutiinien ja funktioiden luokittelu siten, että ymmärrettäessä ryhmästä yksi, niin myös muut ryhmästä ymmärretään	19	X		X	
7	Tietorakenteet ja kuvaus niiden käytöstä ohjelma- ja sovellusalueessa sekä odotetut arvot ja määritykset	18	X	X	X	
8	Muutoshistoria	18	X		X	X
9	Vian vaikutusten eristäminen	18	X	X		
10	Lista suoritetuista lauseista, proseduurikutsuista ja muuttujien arvoista	17	X	X		
11	Kutsukaavion esittäminen	16	X			
12	Muuttujien määritykset, miksi muuttujat ovat välttämättömiä ja miten niitä käytetään, oletusarvot ja odotetut arvot.	14	X	X	X	
13	Halutun lähdekoodin osan sijainti	12	X	X		
14	Hakemistorakenne: otsikkotiedostot, päätiedosto, tukitiedostot, kirjastotiedostot, tiedostorakenne.	12	X			
15	Korostetut lauselohkojen aloitus- ja lopetuskohdat	11	X			
16	Lähdekoodikohta, johon tehdä muutokset	10	X		X	X
17	Tietyn lähdekoodin haaran edellyttämät ehdot joiden seurauksena se suoritetaan tai ei suoriteta, sisältäen muuttuja-arvot	9	X	X		
18	Tarkka sijainti johon asettaa keskeytyskohta	7	X	X		X
19	Sijainti ja kuvaus kirjastorutiineista ja järjestelmäkutsuista	7	X		X	

	Tietotarve	Frekvenssi	Tietolähde			
			S	A	D	H
20	Kirjastorutiinien ja järjestelmäkutsujen sijainti ja kuvaus	7	X	X		
21	(Laatua heikentävä) sivuvaikutus	7	X		X	
22	Odotettu ohjelman tila, esim. odotetut arvot muuttujille proseduuria kutsuttaessa	7	X	X	X	
23	Seuraavien tarkasteltavien lähdekoodin osien osoittaminen jo tiedetyn perusteella	7	X	X	X	
24	Proseduurin sisäkkäisyystaso	7	X			

Taulukko 2-1: Ylläpitäjän tietotarpeet (Koskinen, Salminen & Paakki, 2004) S = Staattinen lähdekoodin analysointi, A = lähdekoodin ajonaikainen analysointi, D = Dokumentaatio, H = istunto historia.

Ohjelmistojen ylläpidossa ylläpitäjän tietolähteet voidaan jakaa kolmeen luokkaan: muut ihmiset, ohjelmistojen ylläpidon työvälineiden tarjoama digitaalinen tieto ja työvälineiden ulkopuolella oleva tieto. Työvälineiden tarjoama tieto jakautuu edelleen neljään ryhmään: lähdekoodin staattinen analysointi, lähdekoodin ajonaikainen analysointi, ohjelmiston dokumentaatio ja istuntohistoria. Lisäksi näiden neljän työvälineiden käyttämien tietolähteiden tieto voidaan myös yhdistää. (Koskinen ym. 2004)

Staattinen lähdekoodin analysointi tarjoaa tietoa ohjelman osien riippuvuuksista, kuten esimerkiksi viittauksista tiettyyn muuttujaan tai funktioon. Staattisen analysoinnin avulla voidaan tukea eniten erityyppisiä tietotarpeita verrattuna muihin tietolähteisiin, mutta se ei kuitenkaan täytä kaikkia ylläpitäjän tietotarpeita. (Koskinen ym. 2004)

Lähdekoodin ajonaikainen eli dynaaminen analysointi tarjoaa ohjelman ajon aikaista tietoa, kuten tietoa muuttujien arvoista ja kutsutuista funktioista. Dynaaminen analysointi on tarpeellista erityisesti olio-ohjelmointikielissä monimuotoisuuden (polymorphism) takia. (Koskinen ym. 2004)

Dokumentaatio koostuu lähdekoodiin tehdyistä kommenteista sekä muista dokumenteista. Taulukon 2-1 mukaan useimmin esiintyvä tietotarve on sovellusaluetieto, joka voidaan täyttää dokumentaation ja lähdekoodin staattisen analysoinnin avulla. (Koskinen ym. 2004)

Istuntohistoria tarjoaa ylläpidon aikana tallennettua tietoa, kuten selatuista kohteista ja tehdyistä muutoksista. Istuntohistorian avulla on mahdollista nopeuttaa ohjelman ymmärtämistä, koska ylläpitäjä palaa usein aikaisemmin tarkastelemaansa kohtaan. (Koskinen ym. 2004)

Eri tietolähteet eivät itsestään täytä kaikkia ylläpitäjän tietotarpeita, vaan tarvitaan lisäksi yhdistettyä sovellusalue-ohjelma-tilannemallitietoa, joka syntyy yhdistämällä tietoa eri tietolähteistä. Taulukossa 2-1 olevat ylläpitäjien tietotarpeet on mahdollista täyttää olemassa olevien tekniikoiden avulla. (Koskinen ym. 2004).

2.4 Ohjelmistojen ylläpidon työvälineet ja käänteistekniikat

Ohjelmistojen ylläpidon työvälineet eroavat ominaisuuksiltaan toisistaan paljon. Työvälineet voivat olla tarkoitettuja vain dokumentointiin tai olla kattavia kehitysympäristöjä. Työvälineiden tarkoitus on kuitenkin helpottaa ohjelmistojen ylläpitoa ja ohjelman ymmärtämistä. Ohjelmistojen ylläpidon työvälineiden tutkimukset ovat erityisesti keskittyneet ohjelmien takaisinmallinnukseen lähdekoodista, koska esimerkiksi perinneohjelmistojen ylläpidossa dokumentaatio ei ole ajan tasalla ja ohjelmistoarkkitehtuuri voi olla rappeutunut, jolloin ainoa luotettava käytössä oleva tietolähde on lähdekoodi. Takaisinmallinnuksen avulla voidaan tuottaa ajan tasalla olevaa tietoa ohjelmasta.

Ohjelmien takaisinmallinnus on lähtöisin laitteistokäänteistekniikoista, joiden tarkoituksena on yrittää selvittää miten jokin laite toimii (Waters & Chikofsky 1994). Chikovsky & Cross (1990) määrittelevät ohjelmien takaisinmallinnuksen yrityksenä tunnistaa ohjelman eri komponentit ja niiden riippuvuudet toisiinsa, sekä luoda niistä korkeamman tason kuvaus. Harsu (2003, 119-120) määrittelee takaisinmallinnukselle seuraavanlaisia tavoitteita: monimutkaisuuden hallinta, vaihtoehtoisten näkymien tarjoaminen, kadonneiden tietojen jäljittäminen, sivuvaikutusten paljastaminen, abstraktien esitysten tuottaminen, uudelleenkäytön mahdollistaminen ja suunnitelmaratkaisujen jäljittäminen. Eli takaisinmallinnus tarjoaa kohdeohjelmasta tietoa eri muodoissa helpottamaan sen toiminnan ymmärtämistä. Ohjelmistojen ylläpidon työvälineet eivät siten välttämättä tarjoa ominaisuuksia ylläpidossa tehtävien muutosten tekoon, vaan voivat olla vain aputyövälineitä ymmärtämisprosessissa. Ohjelmien takaisinmallinnus tapahtuu *ohjelmistokäänteistekniikoiden* avulla. Ohjelmien takaisinmallinnuksella tässä tutkimuksessa tarkoitetaan vain ohjelmien takaisinmallinnusta lähdekoodista.

Takaisinmallinnus jakautuu staattiseen ja dynaamiseen takaisinmallinnukseen, jotka vastaavat edellisessä luvussa esitettyjä staattista ja dynaamista lähdekoodin analysointia. Staattinen takaisinmallinnus perustuu *jäsentimen* jäsentämään lähdekoodiin. Staattisen takaisinmallinnuksen avulla lähdekoodista on mahdollista luoda tieto- ja kontrollivuoanalyysejä sekä määrittää erilaisia metriikoita. (Harsu 2003, 44-46)

Dynaamisessa takaisinmallinnuksessa käytetään lähdekoodin ajonaikaista tietoa, mikä vastaa luvussa 2.3 esitettyä lähdekoodin dynaamista analysointia. Dynaaminen takaisinmallinnus on erityisen hyödyllinen olio-ohjelmointikielillä toteutettujen ohjelmien sekä samanaikaisten ja reaaliaikaisten järjestelmien ymmärtämisessä. (Harsu 2003, 44-46)

Viipalointi (slicing) on ohjelman analysointimenetelmä, jossa tarkastelun kohteena on vain osa lähdekoodista. Viipaloinnissa käytetään apuna tietovirtoja ja kutsukaavioita. On olemassa kahden tyyppistä viipalointia, etenevää (forward slicing) ja takautuvaa (backward slicing). Takautuvassa viipaloinnissa saadaan selville se mitkä lauseet vaikuttavat tiettyyn muuttujaan tietyssä kohdassa lähdekoodia. Etenevässä viipaloinnissa puolestaan saadaan selville kaikki lauseet, joihin tietty muuttuja vaikuttaa. Viipaloinnissa voidaan käyttää hyväksi staattista ja dynaamista lähdekoodin analysointia. (Harsu 2003, 47-48)

3 Aikaisemmat tutkimukset ja tutkimusviitekehys

Tässä luvussa on tarkoitus käydä läpi tutkimuksia, joissa on arvioitu ylläpidon työvälineitä. Tarkoituksena on löytää viitekehys, jonka avulla on mahdollista arvioida ylläpidon työvälineiden tukea ylläpitäjien tietotarpeille. Ensin käydään läpi mitä tutkimuksia aiheesta on tehty, jonka jälkeen arvioidaan niiden soveltuvuutta.

3.1 Aikaisemmat tutkimukset

Ohjelmistojen ylläpidon työvälineiden arvioinnista on olemassa useita tutkimuksia. Bellayn & Gallin (1998) ovat tehneet tutkimuksen, jossa on arvioitu neljää käänteistekniikkatyövälinettä. Koskinen ym. (2004) ovat tutkimuksessaan arvioineet käänteistekniikkatyövälinettä nimeltä HyperSoft ja tehneet yhteenvedon usean eri ohjelmistojen ylläpidon työvälineen ominaisuuksista. Luvussa 2 esitetyt von Mayrhauserin ja Vansin (1995A, 1995B, 1997A, 1997B & 1998) tutkimukset toimivat pohjana Koskisen ym. (2004) tutkimukselle. Gannod & Cheng (1999) ovat tehneet viitekehyksen käänteistekniikkatyövälineiden sekä suunnitteluratkaisujen käänteistekniikoiden vertaamiseen. Lisäksi Choi & von Mayrhauser (1990) ovat tehneet tutkimuksen ohjelman ymmärtämisprosessia tukevien työvälineiden vaatimuksista sekä von Mayrhauser & Lang (1998) ovat tehneet tutkimuksen ohjelmistojen ylläpitotyökalujen tuesta ohjelman ymmärtämisessä. Seuraavissa kappaleissa käydään tarkemmin lävitse näiden tutkimusten sisältöä.

Bellayn & Gallin (1998) tutkimus on tehty Brownin ym. (1996) kehittämän yleisen ohjelmien ja teknologioiden arviointiviitekehyksen pohjalta. Viitekehyksen mukaan ensimmäinen askel arvioitaessa ohjelmia on kartoittaa saatavilla olevat ohjelmat ja teknologiat. Saatavilla olevista teknologioista ja ohjelmista luodaan kaavio, joka kuvaa teknologioiden ja ohjelmien suhteet toisiinsa. Kaavio kertoo mitä teknologioita on

mihinkin tuotteisiin toteutettu, sekä mitkä teknologiat ovat kilpailevat keskenään. Seuraavaksi luodaan toinen kaavio, joka kuvaa mihin sovellusalueen ongelmiin kyseiset ohjelmat ja teknologia tarjoavat ratkaisun. Näiden kaavioiden avulla on tarkoitus luokitella ohjelmat ja teknologiat niiden tarjoamien ominaisuuksien perusteella. Vertailun avulla saadaan selville kilpailevat sekä toisiaan täydentävät ohjelmat ja teknologiat. Bellayn & Gallin (1998) tutkimuksessa arvioidaan neljää käänteistekniikkatyövälinettä: Imagix 4D, Refine/C, Rigi V ja SNiFF+. Arvioinnissa käytetty ohjelma oli laajuudeltaan 150 LOC ja se oli tehty C- ja Assembler-ohjelmointikielillä. Työvälineitä arvioitiin neljässä eri luokassa:

- lähdekoodin analysointiominaisuudet
- tiedon esitysominaisuudet
- lähdekoodin muokkaus- ja selausominaisuudet
- yleiset ominaisuudet

Jokainen luokka sisältää ominaisuuksia, joita käänteistekniikkatyökalu tulisi tukea. Ominaisuuksia arvioitiin neliportaisella asteikolla: ei toteutettu - toteutettu - hyvä - erittäin hyvä. Tutkimuksessa ei kerrottu minkä pohjalta nämä eri ominaisuudet joita käänteistekniikkatyökalun tulisi tukea oli laadittu.

Koskinen ym. (2004) tutkimus jakautuu kahteen osaan. Ensimmäisessä osassa tutkimuksessa on arvioitu kokeellista käänteistekniikkatyövälinettä nimeltä HyperSoft. Toisessa osassa on arvioitu ohjelmistojen ylläpidon työvälineistä ja niiden ominaisuuksista. Tutkimuksessa oli mukana takaisinmallinnus- sekä uudelleendokumentointityövälineitä. HyperSoft-ohjelman arviointi on tehty arviointiviitekehyksen avulla, joka pohjautuu von Mayrhauserin & Vansin(1995B, 1997A, 1997B & 1998) ohjelmistojen ylläpidon tietotarpeita käsitteleviin tutkimuksiin. HyperSoftin ominaisuuksia verrattiin taulukossa 2-1 esitettyihin ylläpitäjien tietotarpeisiin. Tutkimuksen toisessa osassa ohjelmistojen ylläpidon työkalujen ominaisuudet on jaettu viiteen luokkaan:

- navigointiominaisuudet
- tiedon kyselyominaisuudet
- visualisointiominaisuudet
- automatisoidut korkean tason tietorakenteet
- automatisoidut perustietorakenteet

Työkalujen arvioinnissa kuitenkin käytetään vain kolmea ensimmäistä näistä luokista. Tietorakennetasoja ei käsitellä, koska ne ovat välttämättömiä kaikille ohjelmistojen ylläpidon työvälineille. Tutkimuksen erityisenä kiinnostuksen kohteena on väliaikaisten hypertekstirakenteiden tarjoamat mahdollisuudet ohjelmistojen ylläpidon työvälineissä.

von Mayrhauser & Lang (1998) ovat verranneet empiirisessä tutkimuksessaan Lemma-nimistä ohjelmistojen ylläpidon työvälinekokoelmaa tavalliseen ohjelmointiympäristöön. Tutkimuksessa koehenkilöt suorittivat yleisiä ohjelmistojen ylläpitotehtäviä käyttäen ääneen ajattelu -menetelmällä. Tehtävien suorittamisesta mitattiin niihin kulunut aika sekä oikeellisuus. Tutkimuksen tuloksissa painotetaan korkean tason tiedon ja lähdekoodin dynaamisen analysoinnin tärkeyttä ylläpidossa.

Gannod & Cheng (1999) ovat kehittäneet viitekehyksen käänteistekniikoiden ja suunnitteluratkaisujen takaisinmallinnustekniikoiden vertaamiseen. Viitekehyksen avulla arvioidaan ohjelmistojen ylläpidon työvälineiden sivutuotteita, kuten esimerkiksi työvälineen luomaa kutsukaaviota. Tarkoituksena on arvioida ohjelman ymmärtämistä helpottavan ominaisuuden kykyä ilmaista haluttu asia.

3.2 Arviointiviitekehys

Edellisessä luvussa esiteltiin tutkimuksia, joissa on arvioitu ohjelmistojen ylläpidon työvälineitä. Tässä luvussa vertailemme näiden tutkimusten lähestymistapoja ja valitsemme sopivan viitekehyksen ohjelmistojen ylläpidon työvälineiden arviointia varten.

Bellay & Gall (1998) arvioivat tutkimuksessaan neljää käänteistekniikkatyövälinettä. Tutkimus perustui Brownin ym. (1996) kehittämään yleiseen ohjelmien vertailuun tarkoitettuun viitekehykseen. Sen tarkoituksena on mahdollistaa vertailu saatavilla olevien ohjelmien ja niissä toteutettujen teknologioiden välillä. Arvioidut ominaisuudet perustuvat siten saatavilla olevien ohjelmien ominaisuuksiin. Tämän tutkimuksen tarkoitus on vastata kysymykseen tukevatko ylläpidon työvälineet ylläpitäjien todellisia tietotarpeita. Bellayn & Gallin (1998) viitekehys ei sovellu tähän tarkoitukseen.

von Mayrhauserin & Langin (1998) tekemä tutkimus sisälsi empiirisen kokeen jossa verrattiin tavallista ohjelmointiympäristöä käänteistekniikkatyövälinekokoelmaan teettämällä koehenkilöillä ylläpitotehtäviä. Tutkimuksen tarkoituksena oli todistaa empiirisen kokeen avulla käänteistekniikkatyövälineiden hyödyt verrattuna tavalliseen ohjelmointiympäristöön, joten se ei sovellu tähän tutkimukseen.

Gannod & Cheng (1999) arvioivat tutkimuksessaan käänteistekniikkatyövälineiden kykyä viestiä tietty toiminnallisuus. Tutkimuksen viitekehyksen avulla ei selviä tukevatko käänteistekniikkatyövälineet ylläpitäjien tarpeita vaan esimerkiksi miten hyvin tietty esitys kuvaa riippuvuuksia lähdekoodissa.

Koskinen ym. (2004) tutkimus pohjautuu von Mayrhauserin & Vansin (1995B, 1997A, 1997B, 1997C) tutkimuksissa empiiristen kokeiden avulla määriteltyihin ylläpitäjien tietotarpeisiin. Tutkimuksessa tehty ylläpidon työvälineiden arviointi perustuu siten

ylläpitäjien todellisiin tietotarpeisiin. Viitekehys soveltuu siten parhaiten myös tämän tutkimuksen tarkoitukseen. Viitekehyksessä työvälineitä arvioidaan kolmessa eri luokassa navigointi-, visualisointi- ja tiedonkyselyominaisuudet. Seuraavissa luvuissa käydään tarkemmin lävitse mitä ominaisuuksia kussakin luokassa arvioidaan.

3.2.1 Navigointiominaisuudet

Navigointiominaisuuksien tarkoitus on helpottaa tiedon selaamista eri tietolähteistä. Ohjelmoijan tarvitsema tieto on usein pirstaloituneena useaan eri lähdekooditiedostoon ja dokumentteihin, jolloin tiedoston etsiminen ja oikean kohdan löytäminen tiedostosta voi viedä paljon aikaa. Navigointiominaisuudet jakautuvat kahteen ryhmään: lähdekoodin ja dokumentaation navigointiominaisuuksiin. (Koskinen ym. 2004)

Lähdekoodin navigointiominaisuudet helpottavat ja nopeuttavat lähdekoodin selaamista. Navigointiominaisuudet ovat tavallisesti hyvin läheisesti sidottuja visualisointiominaisuuksiin, koska osa lähdekoodin navigoinnista tapahtuu erilaisten graafisten esitysten, kuten kutsukaavioiden kautta. Lähdekoodin navigointiominaisuudet käyttävät yleensä hyväksi väliaikaisia hypertekstirakenteita, jotka luodaan lähdekoodin staattisella analysoinnilla. Yleisiä lähdekoodin navigointiominaisuuksia ohjelmistojen ylläpidon työvälineissä ovat muuttujien, luokkien ja funktioiden ristiinviittaukset, joiden avulla on mahdollista esimerkiksi siirtyä funktion käyttökohdasta suoraan sen määrittelyyn. Lähdekoodin navigointi on mahdollista myös istuntohistorian avulla, jolloin ylläpitäjä pystyy siirtymään nopeasti aikaisemmin tarkastelemaansa kohtaan lähdekoodissa. (Koskinen ym. 2004)

Dokumentaation navigointi perustuu yleensä ohjelmoijien tekemään ja ylläpitämään linkitykseen lähdekoodin ja dokumentaation välillä. Dokumentaatio voi koostua lähdekoodin ulkopuolisista dokumenteista ja lähdekoodin kommentteihin tehdystä dokumentoinnista. Dokumentaation navigoinnissa voidaan myös käyttää hyväksi istuntohistoriaa. (Koskinen ym. 2004)

3.2.2 Visualisointiominaisuudet

Visualisointiominaisuuksilla tarkoitetaan korkean tason kuvauksia lähdekoodista, jotka on luotu käänteistekniikoiden avulla. Lisäksi visualisointiominaisuuksiin luetaan erilaiset lähdekoodin merkkaukset, jotka helpottavat sen luettavuutta. Yleisiä ohjelmistojen ylläpitotyökalujen visualisointiominaisuuksia ovat esimerkiksi lähdekoodista automaattisesti luodut kutsu- ja luokkakaaviot sekä vastakkaisten lausesulkujen merkkaus lähdekoodiin. Visualisointiominaisuudet voivat käyttää hyväksi staattista ja dynaamista lähdekoodin analysointia. (Koskinen ym. 2004)

3.2.3 Tiedonkyselyominaisuudet

Tiedonkyselyominaisuudet luokassa tarkastellaan työvälineen tarjoamia ominaisuuksia, joiden avulla ohjelmoija pystyy tekemään kyselyjä eri tietolähteistä. Tiedonkysely voidaan tehdä graafisen käyttöliittymän avulla tai formaaleilla kielillä. Tiedonkyselyominaisuuksiin kuluu myös viipalointi. (Koskinen ym. 2004)

Tavallisesti ohjelmistojen ylläpitotyökalulla kyselyt tehdään jonkinlaisen graafisen käyttöliittymän avulla. Tällaisia kyselyjä ovat esimerkiksi eteen- ja taaksepäin viipaloinnit. Ongelmana graafisella käyttöliittymällä toteutetuissa kyselyissä voi olla kyselyn ehtojen määrittelyn ilmaisuvoiman rajoittuneisuus. (Koskinen ym. 2004)

Formaalien kielten käyttö tarjoaa paljon joustavuutta ja ilmaisuvoimaa kyselyihin, mutta suoraan itse ylläpidon yhteydessä käytettynä ne voivat olla liian monimutkaisia ja hitaita käyttää. Formaalit kyselykielet mahdollistavat laajennoksien kehittämisen ohjelmiston ylläpidon työkaluun, minkä avulla työvälinettä on mahdollista muokata eri käyttötarkoituksiin. (Koskinen ym. 2004)

4 Yhteenveto

Tässä tutkielmassa oli tarkoitus selvittää mitä ovat ohjelmistojen ylläpito sekä siihen liittyvät ohjelman ymmärtäminen ja käänteistekniikat. Lisäksi tarkoitus oli käydä lävitse tutkimuksia, joissa on arvioitu ohjelmistojen ylläpidon työvälineitä ja löytää näistä viitekehys, jonka avulla on mahdollista arvioida ohjelmistojen ylläpidon työvälineiden tukea ohjelman ymmärtämisessä.

Vuosituhannen vaihtumisen aiheuttamien ylläpito-ongelmien innoittamana 1990-luvun lopulla tehtiin huomattava määrä tutkimuksia ohjelmistojen ylläpidosta ja ohjelmien ymmärtämisestä. Tuloksena näistä tutkimuksista on saatu syvällisempi näkemys ohjelmien ymmärtämisprosessista, määriteltyä ylläpitäjien tietotarpeet ja vaatimuksia ylläpidon työvälineille. Empiiristen kokeiden avulla on myös todistettu, että tukemalla ylläpitäjien tietotarpeita ylläpidon työvälineiden avulla voidaan nopeuttaa ylläpitoa sekä vähentää ylläpidosta aiheutuneita vikoja. Ohjelman ymmärtämisen helpottamiseen ja nopeuttamiseen on siten löydetty keinoja, mutta ylläpidon kustannukset ovat edelleen muita ohjelmiston elinkaaren vaiheita huomattavasti korkeammat. Koskisen ym. (2004) tutkimuksen viitekehyksen avulla on mahdollista selvittää tukevatko tämän hetken ylläpidon työvälineet ylläpitäjien todellisia tietotarpeita.

5 Lähteet

Bellay B & Gall H. 1998. An Evaluation of Reverse Engineering Tool Capabilities. Journal of Software Maintenance: Research and Practice 10, 305–331.

Bennett K., Vaclav R. 2000. Software Maintenance and Evolution: A Roadmap. Teoksessa International Conference on Software Engineering, Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 73-87.

Brooks R. 1983. Towards a Theory of the Comprehension of Computer Programs, International Journal Man-Machine Studies, Vol. 18, 543-554.

Brown A.W. & Wallnau K.C., 1996. A framework for evaluating software technology. Software, IEEE, Volume: 13, Issue: 5, 39-49.

Chikofsky E.J. Cross J.H., 1990. Reverse engineering and design recovery: a taxonomy. Software, IEEE. Volume: 7, Issue: 1, 13-17.

Choi E., von Mayrhauser A. 1994. Automated Assistance to Program Understanding. Teoksessa TENCON 94. IEEE Region 10's Ninth Annual International Conference. Theme: 'Frontiers of Computer Technology, 1002-1007 Vol. 2, 22-26.

Gannod G., Cheng B., 1999. A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. Sixth Working Conference on Reverse Engineering, 77.

Erlikh, L. (2000). Leveraging Legacy System Dollars for E-business. (IEEE) IT Pro, May/June 2000, 17-23.

Fjeldstad, R. & Hamlen, W. (1983). Application Program Maintenance-Report to our Respondents. Tutorial on Software Maintenance, 13-27. Parikh, G. & Zvegintzov, N. (Eds.). IEEE Computer Soc. Press.

Haikonen J., 2006. Ylläpidettävyys avoimen lähdekoodin mukaisen ohjelmistotuotannon näkökulmasta. Pro gradu -työ, Jyväskylän yliopisto: tietojenkäsittelytieteiden laitos. Saatavilla sähköisessä muodossa <http://thesis.jyu.fi/06/URN_NBN_fi_jyu-2006408.pdf>.

Harsu, 2003. Ohjelmien ylläpito ja uudistaminen. Talentum.

IEEE Computer Society. 1998. IEEE Standard for Software Maintenance (IEEE std 1219-1998). Software Engineering Standards Committee of the IEEE Computer Society.

Koskinen J., Salminen A., Paakki J. 2004. Research Hypertext support for the information needs of software maintainers. Journal of Software Maintenance and Evolution: Research and Practice 16(3), 187-215.

Lehman M.M., 1980. Programs, life cycles, and laws of software evolution, Proceedings of the IEEE, Publication Date: Sept. 1980, Volume: 68, Issue: 9 1060- 1076.

Pressman R. 2000. Software Engineering - A Practitioner's Approach 4. painos. McGraw Hill.

Shneiderman B. & Mayer R., 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental. International Journal of Parallel Programming, Issue Volume 8, Number 3, 219-238.

Sommerville I. 2001. Software Engineering 5. Painos. Addison-Wesley.

Standish T. (1984). An Essay on Software Reuse. IEEE Transactions on Software Engineering SE-10 (5), 494-497.

Swanson EB., 1976. International Conference on Software Engineering, Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, United States, 492-497.

von Mayrhauser A., Vans A.M. 1995A. Industrial Experience with an Integrated Code Comprehension Model. Software Engineering Journal, 10(5), pp. 171-182.

von Mayrhauser A., Vans A.M. 1995B. Program Comprehension During Software Maintenance and Evolution. Computer, 28(2), 44-45.

von Mayrhauser A., Vans A.M. 1997A. Program Understanding needs During Maintenance of Large Scale Software. Teoksessa Proc. 21st Annual Computer Software & Application Conference (COMPSAC'97). IEEE Computer Soc., 630-637.

von Mayrhauser, A, Vans, A.M. 1997B. Program Understanding Behaviour During Enhancement of Large-scale Software. Journal of Software Maintenance: Research and Practice 9(5), 229-327.

von Mayrhauser, A., Vans, A.M. 1998. Program Understanding During Software Adaptation Tasks. Teoksessa Proc. Int. Conf. Software Maintenance (ICSM'98). IEEE Computer Soc., 316-325.

von Mayrhauser A., Lang S., 1999. A Coding Scheme to Support Systematic Analysis of Software Comprehension, Software Engineering, IEEE Transactions on, Volume: 25, Issue: 4, 526-540.

Waters R. & Chikofsky E., 1994. Reverse Engineering: Progress Along Many Dimensions, Communications of the ACM Archive, Volume 37 , Issue 5 (May 1994), 22-25.

Yau S.S., Collofello J.S., MacGregor T. 1978. Ripple Effect Analysis of Software Maintenance, Proceedings of Compsac, IEEE Computer Society Press, Los Alamitos, CA, 60-65.

Zelkowitz M., Shaw A., & Gannon J., 1979. Principles of Software Engineering and Design. Prentice-Hall.