Kehysriippuvaisen PHP-ohjelmiston päivitettävyyden arviointi

Jukka Roihuvaara

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Timo Poranen
Huhtikuu 2013

Tampereen yliopisto
Informaatiotieteiden yksikkö
Jukka Roihuvaara: Kehysriippuvaisen PHP-ohjelmiston päivitettävyyden arviointi
Pro gradu -tutkielma, 56 sivua
Huhtikuu 2013

Ohjelmistokehykset nopeuttavat ohjelmistokehitystä, mutta niistä riippuvaiset ohjelmat vaativat ajoittain päivitystä ohjelmistokehysten muutosten vuoksi. Tässä tutkielmassa esitellään kevyt menetelmä ohjelmistokehystä käyttävän PHP-ohjelmiston päivitettävyyden arvioimiseksi. Arvio tehdään COCOMO II -kustannusarviomenetelmän Application Composition ja Post Architecture -mallien avulla. Työmäärää mittaavia menetelmiä sovelletaan Joomla-ohjelmistokehyksellä toteutetun PHP-ohjelmiston kehityksen työmäärän arvioimiseksi ja eri arvioiden perusteella päätellään edullisin tapa ohjelmiston päivittämiseksi.

Avainsanat ja -sanonnat: työmääräarviointi, COCOMO II, ohjelmistokehykset, päivitettävyys, WWW, PHP, Joomla.

Sisällys

1 Johdanto

Ohjelmistokehityksessä pyritään jatkuvasti kasvattamaan ohjelmistojen uudelleenkäytön määrää, sillä siten voidaan vähentää kustannuksia ja nopeuttaa tuotekehitystä [Frakes & Terry, 1996]. Ohjelmistokehyksen käyttäminen vähentää ohjelmiston toteuttamiseen vaadittua työmäärää murto-osaan [Kim *et al.*, 2012] verrattuna alusta asti itse toteutettuun ohjelmistoon, joten niiden käyttö on luonnollinen osa ohjelmistokehitystä. Myös WWW-ympäristössä [Coda *et al.*, 1998] käytetään ohjelmistokehyksiä, mutta ympäristön lyhyen historian vuoksi tuotteilta vaaditut ominaisuudet eivät ole vakiintuneet. WWW-ohjelmistokehysten tarjoamat ominaisuudet eivät ole kiinteät, vaan niitä kehitetään ja muutetaan jatkuvasti. Kun ohjelmistokehyksiin lisätään uusia ominaisuuksia, tai niiden suorituskykyä tai turvallisuutta parannetaan, niiden käyttötavat saattavat muuttua.

Ohjelmistokehysten kehittyessä niistä riippuvaiset ohjelmistot vaativat ylläpitoa ja päivittämistä, mikäli niitä halutaan käyttää ohjelmistokehyksen uusimpien versioiden kanssa. Kun ohjelmistokehys muuttuu merkittävästi, voi sitä käyttävän ohjelmiston päivittäminen olla kalliimpaa kuin ohjelmiston uudelleentoteuttaminen. Ohjelmistokehysten ja niiden avulla ohjelmistojen toteuttamiseen käytettyä työmäärää [Mattsson, 2000] sekä ohjelmistojen muuntamisen hintaa ja vaikutusta laatuun [Leitch & Stroulia, 2003] on tutkittu ohjelmistokehitystä käsittelevässä kirjallisuudessa. Työmäärien arviointi keskittyy yleensä tulevien toteuttamattomien projektien koon ja aikataulun määrittämiseen, eikä ole olemassa yksinkertaista menetelmää selvittää, onko kustannustehokkaampaa päivittää ohjelmisto käyttämällä nykyisiä lähdekoodeja, vai onko halvempaa aloittaa alusta uudella ohjelmistokehyksellä.

Esittelen tässä tutkielmassa tavan, jolla voidaan arvioida ohjelmiston päivittämisen kannattavuutta. Ajatuksena on soveltaa hyvin dokumentoitua kustannusarviomenetelmää [Boehm *et al.*, 2000] ohjelmistosta kerätyillä tiedoilla, ja saada kaksi arviota eri vaihtoehtojen työmääristä. Ensin arvioidaan ohjelmiston toteuttamiseen vaadittavaa työmäärää ohjelmiston lähdekoodin määrän perusteella, minkä jälkeen sama arvio tehdään ohjelmiston käyttöliittymistä lasketun ohjelmiston ominaisuuksien määrän perusteella. Käyttöliittymien pohjalta tehdyn arvioinnin perusteella päätellään ohjelmiston lähdekoodin soveltuvuutta tarkoitukseensa. Tavoitteenani on osoittaa, että soveltamalla kustannusarviomenetelmää jo toteutetun ohjelmiston arvioimiseen jälkikäteen, voidaan tehdä johtopäätöksiä ohjelmiston

lähdekoodin laadusta ja tulevan päivittämisen työläydestä myös käytännössä. Kuvaamani menetelmän etuna on sen nojautuminen hyvin määriteltyyn kustannusarviomenetelmään, jonka tarkkuudesta ja toimivuudesta on olemassa tutkittua tietoa [Dillibabu & Krishnaiah, 2005]. Tavoitteenani on myös soveltaa ehdottamaani päivitettävyyden arviointimenetelmää päivitystarpeessa olevaan ohjelmistoon ja saada konkreettista hyötyä tutkimuksen tuloksista.

Tutkielman luvussa 2 esittelen ohjelmistojen päivittämiseen liittyviä käsitteitä, ohjelmistojen laadun ja uudelleenkäytön mittareita sekä ohjelmistokehysten kehityskulkua. Luku 3 käsittelee kustannusarviomenetelmä COCOMOa, jonka tuoreinta versiota käytän luvussa 4 esittelemässäni menetelmässä. Luvussa 4 ehdotan ohjelmiston päivitettävyyden arviointiin uutta menetelmää, ja sovellan sitä luvussa 5 PHP-ohjelmointikielellä toteutettuun WWW-yhteisösovellukseen. Menetelmän soveltamiseen sisältyy arviointi sen hyödyistä.

2 Ohjelmistotuotanto

Ohjelmistotuotannolla tarkoitetaan tietokoneohjelmien, erityisesti suurten ohjelmistokokonaisuuksien valmistamista. Se sisältää menetelmät paitsi ohjelmien suunnitteluun ja toteutukseen, myös suunnittelu- ja ohjelmointityön ohjaukseen, prosessihallintaan sekä laadun mittaamiseen ja valvontaan.

2.1 Ohjelmistokehityksen historia

Tietojenkäsittelyn alkuvuosina 1950-luvulla ohjelmistot olivat pieniä, yleensä yksittäisten henkilöiden, alan asiantuntijoiden kehittämiä ja käyttämiä. 60-luvulle mennessä kasvavat ohjelmistokoot, puutteelliset prosessit sekä ohjelmointimenetelmät johtivat ongelmiin tuottavuuden ja ohjelmien luotettavuuden kanssa. Vuonna 1968 NATOn ensimmäisessä ohjelmistokehityksen konferenssissa otettiin käyttöön termi "software engineering", joka oli tarkoituksella provokatiivinen [Naur *et al.*, 1969; Randell, 1997] ja viittasi perinteisiin tuotanto- ja suunnittelualoihin kuten rakentamiseen ja arkkitehtuuriin, ja tarpeeseen käyttää ohjelmistotuotannossa samankaltaisia teoreettisia ja käytännön menetelmiä.

Ohjelmistotuotannon luotettavuutta ja ennustettavuutta pyrittiin ensin kehittämään prosesseilla kuten vesiputousmalli [Royce, 1970] ja prosesseja parantavilla tarkistustavoilla, laadunvalvontamenetelmillä ja koodinkatselmoinnilla [Randell, 1997]. Roycen [1970] esittämään malliin sisältyi myös iteratiivinen vaihe, jossa ohjelmistosta valmistetaan ensin koeversio ennen varsinaisen ohjelmiston tuottamista [Larman & Basili, 2003]. Valitettavasti mallin iteratiivinen vaihe unohdettiin ja mallista muodostui dokumenttivetoinen vesiputousmalli. Tämän jälkeen keskityttiin parantamaan ohjelmien laatua muodollisilla menetelmillä, jotka pyrkivät tarkkaan vaatimusten määrittelyyn ja muodollisten määrittelyjen hyödyntämiseen ohjelmiston vaatimusten validoinnissa [Randell, 1997].

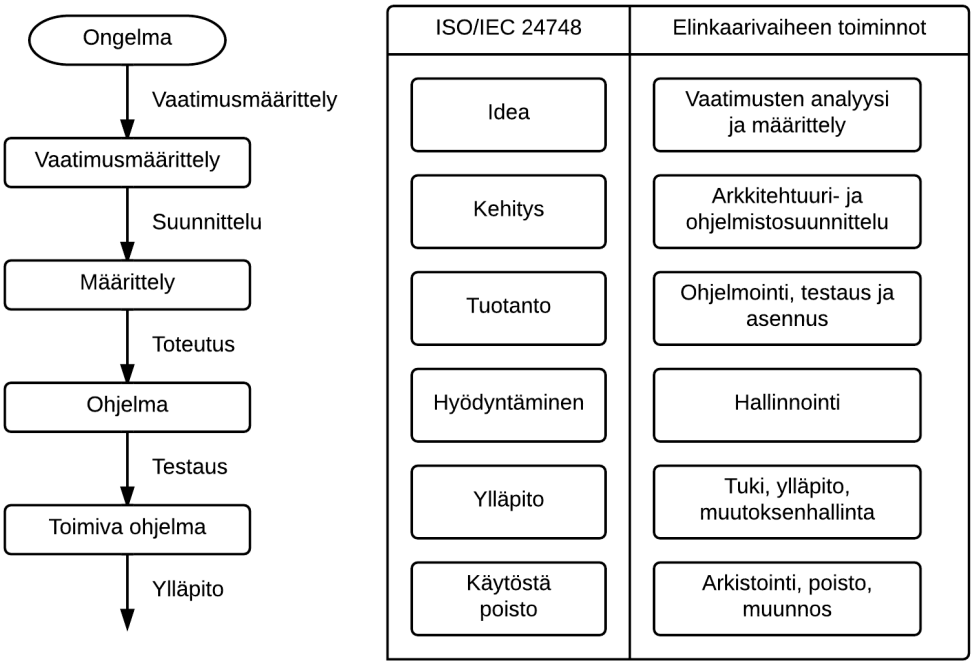
Mikrotietokoneiden tulo 80-luvun alkupuolella muutti ohjelmistojen tärkeyttä ja ohjelmistotuotannon prioriteetteja [Randell, 1997]. Jo ennen 80-lukua ohjelmoinnissa alettiin keskittyä rakenteellisiin metodeihin, ohjelmakoodin selkeyteen ja luettavuuteen. Tässä ajatuksena oli funktiokutsujen, lohkojen, struktuurien, silmukoiden ja muiden ohjausrakenteiden käyttö hyppylauseiden ja suoraviivaisten ehtolauseiden käyttämisen sijaan. Rakenteellisesta ohjelmakoodista on jatkettu

olio-ohjelmoinnin luokkien ja nimiavaruuksien käyttöön. Idea olio-ohjelmoinnista ja ohjelmistojen uudelleenkäytettävyydestä esiteltiin jo 1968 [McIlroy *et al.*, 1968], mutta vasta 80-luvun puolivälissä sitä alettiin käyttää laajasti ohjelmistotuotannossa, uudelleenkäyttämällä ohjelmakoodia luokkakirjastoissa ja erityisesti käyttöliittymäohjelmoinnissa ohjelmistokehyksissä. Olio-ohjelmoinnin puutteita on pyritty vähentämään esimerkiksi aspektien [Kiczales *et al.*, 1997] käyttämisellä tai komponenttiperustaisella ohjelmistotuotannolla [Capretz *et al.*, 2001], jotka myös pohjautuvat oliokeskeiseen ohjelmointiin.

2.2 Ohjelmiston elinkaari

Van Vliet [2008, s. 11] esittää ohjelmistokehityksen yksinkertaistetusti viisivaiheisena: vaatimusmäärittely, suunnittelu, toteutus, testaus ja ylläpito. Aluksi on olemassa ongelma, josta vaatimusmäärittelyn avulla saadaan tulokseksi vaatimusmäärittelydokumentti. Vaatimusten perusteella suunnitellaan ohjelmisto ja saadaan ohjelmiston määrittely, joka toteutetaan. Valmis ohjelma pitää testata, jonka jälkeen se voidaan todeta toimivaksi ja sitä voidaan ylläpitää. ISO/IEC 24748 -standardin [2010] mukaan järjestelmän elinkaari muodostuu kuudesta hieman erilaisesta vaiheesta, joihin kuuluu ylläpitovaiheen jälkeen vielä käytöstä poisto. Standardin määrittelemät vaiheet ovat idea, kehitys, tuotanto, hyödyntäminen, ylläpito sekä käytöstä poisto. Kuvassa 2.1 on esitetty näiden määritelmien mukaiset elinkaarivaiheet rinnakkain.

Dutil ja muut [2010] selittävät standardin määrittelemät työvaiheet siten, että vaatimusten määrittely kuuluu ideavaiheeseen, arkkitehtuuri- ja ohjelmistosuunnittelu ovat kehitystä, ohjelmointi, testaus ja asennus muodostavat tuotantovaiheen ja hyödyntämisvaiheessa ohjelmistoa hallinnoidaan. Heidän mukaansa ylläpitovaihe sisältää paitsi käyttötuen ja ohjelmiston ylläpidon, myös muutoksenhallinnan. Käytöstä poistoon kuuluvat mahdollisesti arkistointi tai esimerkiksi ohjelmiston tuottaman datan muunnos uusien ohjelmistojen ymmärtämään muotoon. Kummassakaan versiossa ohjelmistokehityksen vaiheista ei erikseen mainita ohjelmiston päivittämistä, mutta Dutilin ja muiden [2010] mainitsema muutoksenhallinta voi käsittää myös tämän. Ohjelmiston päivitystarve sijoittuu elinkaarella ylläpidon ja käytöstäpoiston välille, ja päivitys voidaan suorittaa uudella syklillä vaatimusmäärittelyineen ja toteutuksineen.

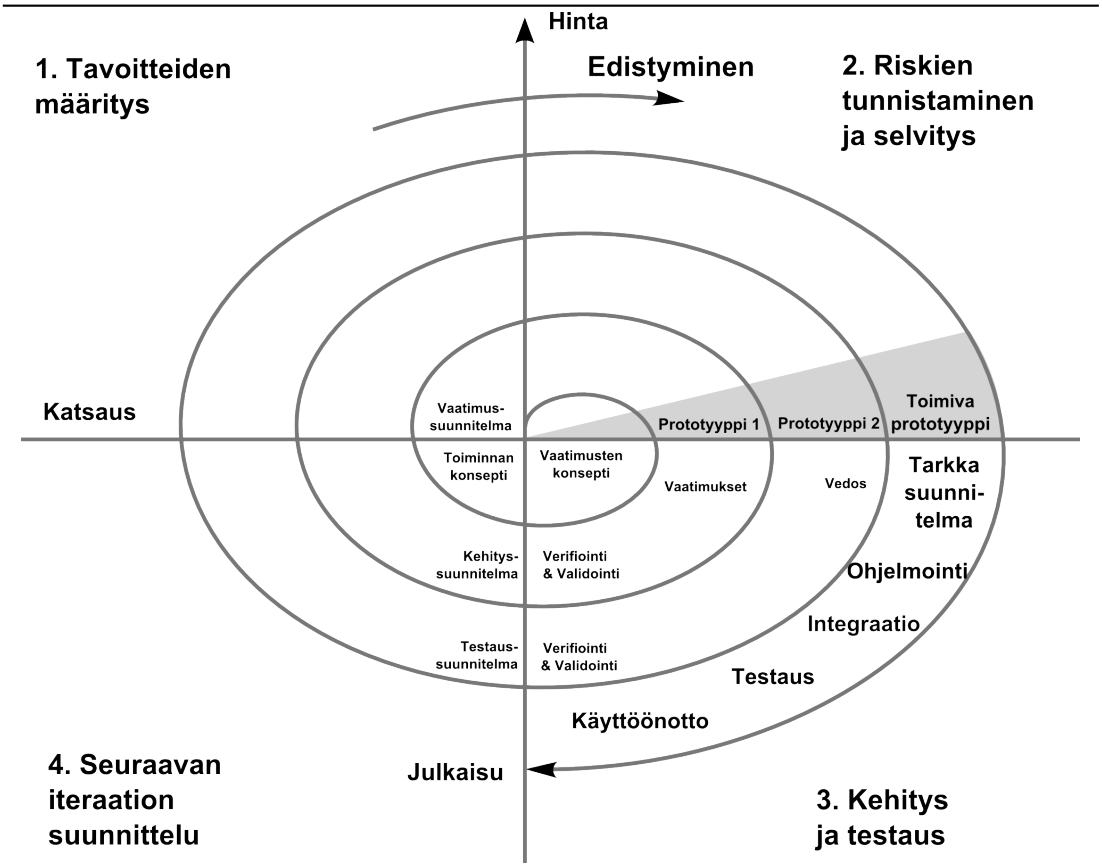


Kuva 2.1 Van Vlietin [2008] (vasemmalla) ja Dutilin ja muiden [2010] (oikealla) käsitykset ohjelmistokehityksen vaiheista.

2.3 Ketterä ohjelmistokehitys

Iteratiivinen ohjelmistokehitys poikkeaa vesiputousmallisesta ohjelmistokehityksestä siten, että ohjelmistoa ei yritetä tehdä valmiiksi yhdessä suunnittelu-toteutus -vaiheessa, vaan näitä vaiheita toistetaan useaan kertaan. Kevyttä ja ketterää ohjelmistokehitystä pidetään usein nykyaikaisena vaihtoehtona vesiputousmalliselle kehitykselle, mutta ketterien menetelmien periaatteet voidaan jäljittää jo 1930-luvulle Walter Shewartin laadun parantamista ehdottamaan suunnittelu-toteutus-tutkimus-toiminta -malliin (Plan-Do-Study-Act, PDSA) [Larman & Basili, 2003]. NASA:n Project Mercury 1960-luvun alusta ja aiempi X-15 -lentokoneen laiteprojekti käyttivät lisäyksittäistä eli inkrementaalista kehitystä, ja Project Mercuryssä hyödynnettiin muitakin nykyään ketterään ohjelmistokehitykseen yhdistettäviä periaatteita. Näitä olivat ominaisuuksien katselmoinnit (reviews), aikarajoitetut lisäykset (time-boxed increments) ja testivetoinen kehitys (test driven development, TDD), jossa suunnitellaan ja toteutetaan kehitettävälle ominaisuudelle testaus ennen ominaisuuden toteutusta [Larman & Basili, 2003]. Larman ja Basili [2003] kertovat iteratiivisen ohjelmistokehityksen menestystarinoista myös 70- ja 80-luvuilta. 80-luvulla Boehm [1986] kuvaili TRW:llä useita vuosia käytössä olleen

spiraalimallin, joka esitetään kuvassa 2.2. Siinä kehitysvaiheita toistetaan useaan kertaan ja jokaisen kehityssyklin jälkeen analysoidaan tulokset.



Kuva 2.2 Boehmin [1986] spiraalimalli.

90-luvulla iteratiivista ohjelmistokehitystä alettiin suosia yhä enemmän sekä Yhdysvaltain hallituksen projekteissa että kaupallisella puolella. Kaupallisen puolen Rational Unified Process [Kruchten, 2000] ja Scrum-metodi [Takeuchi & Nonaka, 1986] ovat iteratiivisia menetelmiä, jotka ovat olleet yleisessä käytössä jo yli 20 vuotta. Ketterän kehityksen käsite muodostettiin kuitenkin vasta vuonna 2001, kun 17 erilaisten iteratiivisten prosessien kannattajaa kokoontuivat keskustelemaan prosessien yhteisistä arvoista. Tuloksena oli ketterän kehityksen manifesti [Beck *et al.*, 2001] jonka myötä moni ketterän kehityksen menetelmä sai myös enemmän huomiota ja julkisuutta. Kaikesta iteratiivisen ohjelmistokehityksen historiasta ja tutkituista eduista huolimatta vesiputousmallista ohjelmistokehitystä yhä käytetään, sillä prosessi on helppo ymmärtää ja sen yksinkertaisuus viehättää muunmuassa tuotteen tilaajia, sillä se antaa kuvan järjestelmällisestä ja mitattavasta prosessista, jolla on selkeä alku ja lopputila [Larman & Basili, 2003].

2.4 Ohjelmiston laatu

Ohjelmiston laatu on vaikeasti määriteltävä asia, sillä laadukkuus tarkoittaa eri sidosryhmille erilaisia asioita, ja eri sidosryhmät mittaavat laatua erilaisin kriteerein [van Vliet, 2008]. Laadun mittaamiseen on pyritty kehittämään erilaisia mittareita eli metriikoita ja näiden avulla on luotu sopimuksia hyvän laadun määritelmästä. Esimerkiksi ISO 9001 [2008], 9126 [2003] ja 25010 [2011] -standardeja voidaan noudattaa, kun valvotaan ohjelmistoprosessin, ohjelmistojen kehityksen ja ohjelmistojen laadukkuutta.

2.4.1 Laatuattribuutit

Laatuun liittyviä ohjelmiston ominaisuuksia kutsutaan yleisesti laatuattribuuteiksi. ISO 9126 -standardin määrittelemät laatuattribuuttien pääkategoriat [Heitlager *et al.*, 2007; ISO, 2003] ovat

- toiminnallisuus (functionality)
- luotettavuus (reliability)
- käytettävyys (usability)
- tehokkuus (efficiency)
- ylläpidettävyys (maintainability) sekä
- siirrettävyys (portability).

Jokainen pääkategoria jaetaan 3-5 alakategoriaan, joita on yhteensä 27 ja joihin suoraan mitattavissa olevat laatuattribuutit lajitellaan.

Ylläpidettävyys

Ohjelmiston päivitettävyyttä käsiteltäessä kiinnostavin attribuuttikategoria on ylläpidettävyys, sillä nimenomaan vaikeasti ylläpidettävä ohjelmisto halutaan ensimmäisenä korvata uudella [Broy *et al.*, 2006]. Heitlager ja muut [2007] ovat kritisoineet ISO 9126 -standardissa käytettyä ylläpidettävyysindeksiä (Maintainability

Index) ylläpidettävyyden mittarina ja muodostaneet käytännöllisemmän laatumallin "SIG Maintainability Model" nimenomaan ylläpidettävyyden määrittämiseksi. He käyttävät ISO 9126:n määrittämiä termejä alakategorioiden nimeämiseen, mutta kytkevät ne selkeästi määritettyihin lähdekoodin ominaisuuksiin kaksiulotteisina taulukkoina. Heitlagerin ja muiden [2007] määrittämät ominaisuudet ovat:

- lähdekoodin määrä
- ohjelmayksiköiden monimutkaisuus
- lähdekoodin toisto eli kloonaus
- yksikkökoko ja
- yksikkötestaus.

Yksiköllä tarkoitetaan pienintä määrää koodia, joka voidaan suorittaa ja testata irrallaan muusta ohjelmasta. Ohjelmayksiköt tarkoittavat eri asioita eri ohjelmointikielillä: Javassa ja C#:ssä yksikkö on metodi, C:ssä proseduuri tai funktio ja joissain kielissä, kuten COBOL, pienin yksikkö on kokonainen ohjelma. Ohjelmistojen koot Heitlager ja muut [2007] ovat määrittäneet lähdekoodin fyysisen rivimäärän (josta on poistettu kommentit ja tyhjät rivit) sekä Jonesin ja SPR:n [2005] luoman taulukon avulla. Taulukko kertoo ohjelmointikielikohtaisesti, kuinka paljon lähdekoodia on keskimäärin funktiopisteellä ja kuinka monta funktiopistettä keskimäärin ohjelmoija tuottaa kyseisellä ohjelmointikielellä kuukaudessa. Funktiopisteet ovat erään ohjelmiston kokoa arvioivan mittaustavan yksikkö. Niiden avulla voidaan esimerkiksi arvioida ohjelmiston lähdekoodin määrää tai mitata ohjelmiston kokoa tietämättä lähdekoodin määrää tarkasti. Funktiopisteiden avulla Heitlager ja muut [2007] muuntavat ohjelmistojen lähdekoodien määrät keskenään vertailukelpoisiksi miestyökuukausiksi.

Monimutkaisuus mitataan SIG Maintainability Model -mallissa luokittelemalla yksiköt syklomaattisen kompleksisuuden mukaan riskiluokkiin [Heitlager *et al.*, 2007] ja painottamalla riskien arvot niiden suhteellisella osuudella koko ohjelmiston lähdekoodin määrästä. Tällä tavalla saadaan määritettyä, miten suuri osa lähdekoodista on monimutkaista eli riskialtista, toisin kuin ISO 9126 -standardissa, jossa

monimutkaisuus lasketaan keskiarvona, joka tasoittaa ääriarvojen erot. Mainittujen metriikoiden lisäksi mallissa annetaan taulukkoarvot toistettujen lähdekoodin rivien määrälle prosentteina, yksiköiden koolle ja yksikkötestien peitolle (Unit test coverage), joka kertoo kuinka suurelle osalle ohjelmiston yksiköistä on olemassa automaattinen testiohjelma, joka varmistaa kyseisen yksikön toiminnan.

SIG Maintainability Model -mallin mukainen ohjelmiston ylläpidettävyyden tutkiminen voidaan suorittaa ohjelmallisesti, koska se käyttää automaattisesti mitattavissa olevia metriikoita arvioinnin perustana. Esimerkiksi ohjelmistojen arviointityökaluun *Sonar* [Sonar, 2013] on saatavilla SIG Maintainability Model -laajennus [Sonar, 2011].

Uudelleenkäytettävyys

Ohjelmistojen uudelleenkäyttö tehostaa tuottavuutta ja laatua merkittävästi [Frakes & Terry, 1996]. Uudelleenkäytettävyys on ohjelmiston ominaisuus, joka määrittää, miten suuri osa ja kuinka isolla työmäärällä ohjelmistoa voidaan käyttää uudelleen uusissa ohjelmistoprojekteissa. Frakes ja Terry [1996] lajittelevat ohjelmiston uudelleenkäyttöä mittaavat metriikat ja mallit kuuteen luokkaan: hinta-hyöty -analyysit, valmiuden määritys (maturity assessment), uudelleenkäytön määrä (amount of reuse), virhetila-analyysit, uudelleenkäytettävyyden määritys (reusability analysis) sekä uudelleenkäyttökirjastojen metriikat. Päivitettävyyden näkökulmasta näistä kiinnostavimpia ovat uudelleenkäytettävyyden määritys ja uudelleenkäytön määrä. Uudelleenkäytön määrällä kuvataan ohjelmiston lähdekoodin tai komponenttien sisäisiä suhteita ja sitä, miten suurta osaa ohjelmistosta käytetään useaan kertaan. Uudelleenkäytettävyyden määrityksellä pyritään esimerkiksi löytämään ohjelmiston osia, joita voidaan hyödyntää uusissa ohjelmistoprojekteissa [Sandhu & Singh, 2006].

Myös ohjelmien toteutusideoita voidaan uudelleenkäyttää. Ajatus siitä, millä tavalla tietty ongelmatilanne ratkaistaan, voidaan dokumentoida, ja samaa ratkaisumallia käyttää uudelleen, kun samankaltainen ongelmatilanne kohdataan. Tätä ratkaisumallia kutsutaan suunnittelumalliksi. Gamma ja muut [1994] kirjoittivat suunnittelumalleista yhden ohjelmistosuunnittelun perusteoksista, jossa he kuvaavat olio-ohjelmoinnin ongelmia ja mahdollisuuksia, ja dokumentoivat 23 erilaista uudelleenkäytettävää suunnittelumallia.

Ohjelmistokehityksessä on pyritty kasvattamaan uudelleenkäytön määrää ensin uudelleenkäyttämällä kokonaisia ohjelmia, sitten järjestämällä ohjelmien sisältämät lähdekoodit funktioiksi, joita voidaan uudelleenkäyttää, ja edelleen yhdistämällä samaan asiaan liittyvät funktiot luokiksi ja niiden sisältämiksi metodeiksi. Luokkia on kerätty luokkakirjastoiksi. Luokkakirjastoista on kehitetty ohjelmistokehyksiä, joiden avulla voidaan toteuttaa sama ohjelmisto pienemmällä työmäärällä kuin aikaisemmin.

2.4.2 Mittaaminen ja metriikat

Mittaamisessa pyritään formaaliin ja verrattavissa olevaan esitysmuotoon. On hyödyllisempää kertoa reseptiin kuuluvan maitoa 1 litra tai 1 desilitra, sen sijaan että sanottaisiin “paljon” tai “vähän”. Samaa vertailtavuutta tarvitaan ohjelmistojen mittaamisessa [Fenton, 1994]. Matematiikassa termi metriikka tarkoittaa etäisyysfunktiota, joka määrittää etäisyyden joukon kahden elementin välillä. Ohjelmistotuotannossa termin merkitys on väljempi, ja sillä saatetaan tarkoittaa mittaustulosta tai mittaustuloksen yksikköä. Van Vliet [2008] määrittää sen tarkoittamaan seuraavien asioiden yhdistelmää:

- asian ominaisuus eli attribuutti
- funktio, joka määrittelee arvon em. ominaisuudelle
- yksikkö, jolla em. arvoa ilmaistaan sekä yksikön
- mitta-asteikko (luokittelu, järjestys, välimatka tai suhdeasteikko).

Metriikoiden määrittäminen ei ole yksinkertaista, sillä monet ominaisuudet vaikuttavat toisiin ominaisuuksiin. Mikäli ominaisuus määritellään toisten ominaisuuksien avulla, se mitataan *epäsuorasti*. Jos ominaisuuden mittaamiseen eivät vaikuta toiset ominaisuudet, sitä voidaan mitata *suoraan* [Fenton, 1994]. Jos ohjelmiston ylläpidettävyys on yksi ominaisuus, niin siihen vaikuttavia ominaisuuksia on muunmuassa ohjelmiston koko. Edes ohjelmiston koosta ei ole yksiselitteistä metriikkaa, sillä esimerkiksi ohjelmiston lähdekoodin rivien määrä ja ohjelmiston sisältämien luokkien määrä ovat erilaiset mittarit ohjelmiston koon määrittämiseksi. Ohjelmiston lähdekoodin rivien määrällekin on erilaisia määritelmiä, joista toiset laskevat fyysisiä tiedostojen rivejä, toiset loogisia lausekkeita. Loogisten

lausekkeiden määrittäminen on edelleen hankalaa — sisältääkö *if {} else {}* yhden vai kaksi loogista lauseketta?

Ohjelmiston mittaamiseen käytettävät metriikat tulee valita käytetyn ohjelmointikielen ja ohjelmointiympäristön ominaisuuksien mukaan. Lähdekoodin rivien määrä on ollut parempi mittari COBOL-ohjelmointikielellä toteutetun sovelluksen koosta kuin mitä se on esimerkiksi Visual Studio -sovelluksella [Microsoft, 2012] toteutetun käyttöliittymän koosta. Chidamber ja Kemerer [1994] keräsivät erityisesti olio-ohjelmointiin soveltuvia metriikoita malliksi, jota voidaan hyödyntää oliokeskeisessä ohjelmistokehityksessä. He analysoivat seuraavat metriikat:

1. Painotettu luokan metodien määrä (Weighted Methods per Class, WMC)
2. Perimäpuun syvyys (Depth of Inheritance Tree, DIT)
3. Välittömien aliluokkien määrä (Number of Children, NOC)
4. Luokkien välinen riippuvuus (Coupling between object classes, CBO)
5. Luokan vaste (Response For a Class, RFC)
6. Metodien yhdenmukaisuuden puute (Lack of Cohesion in Methods, LCOM).

Chidamber ja Kemerer [1994] eivät anna metriikoille ohjearvoja, joihin tulisi pyrkiä, mutta joitain viitearvoja voidaan esittää empiiristen tutkimusten perusteella. Esimerkiksi Laing ja Coleman [2001] tutkivat kolmea järjestelmää, joissa luokan metodien määrän keskiarvo oli hyvälaatuisessa ohjelmakoodissa 11 ja huonolaatuisessa 45. Maksimiarvot olivat 381 ja 596.

2.4.3 Refaktorointi

Koodin refaktorinnilla tarkoitetaan yleensä ohjelmakoodin muuttamista ymmärrettävämpään muotoon tai ohjelmiston rakenteen muuntamista paremmin tarkoitukseen sopivaksi. Ohjelmiston ylläpidettävyys ja siirrettävyys vaikuttavat siihen, miten yksinkertaista ohjelmistoa on refaktoroida. Refaktoroinnille on olemassa akateeminen määritelmä, jonka mukaan refaktoroinnin tuloksena ohjelmiston toiminta ei muutu, mutta käytännössä refaktoroinnin yhteydessä yleensä korjataan virheitä tai lisätään uusia ominaisuuksia [Kim *et al.*, 2012].

Dall'Agnol ja muut [2003] kertovat yhden ketterän kehityksen menetelmän, XP:n (Extreme Programming), tarjoavan ohjelmistokehittäjille konkreettisia ohjeita ja työtapoja ketterän ohjelmistokehityksen toteuttamiseksi. Yksi näistä työtavoista on refaktorointi. Myös muissa iteratiivisissa menetelmissä käytetään refaktorointia, kun uutta iteraatiota toteutetaan edellisen ohjelmistoversion päälle, mutta siihen ei välttämättä erikseen kehoteta.

Refaktoroinnin hyödyistä on ristiriitaista tutkimustietoa. Ainakin joillain ohjelmistokehityksen osa-alueilla on näyttöä siitä, että refaktorointi parantaa ohjelmiston laatua, vähentää virheitä ja helpottaa ylläpidettävyyttä [Kim *et al.*, 2012]. Ohjelmistokehyksiin liittyvissä tutkimuksissa refaktoroinnista johtuvat rajapintamuutokset ovat lisänneet virheiden määrää ohjelmistokehystä käyttävissä ohjelmissa [Kim *et al.*, 2011]. Samanaikaisesti kuitenkin myös virheiden korjausten määrä on kasvanut näissä ohjelmissa, eli asiakasohjelmiin on pitänyt tehdä ylläpitotöitä ohjelmistokehyksen rajapintojen muuttuessa. Myös puutteelliset ja keskeneräiset refaktoroinnit toisistaan riippuvissa ohjelmistojen osissa aiheuttavat virheitä.

Kim ja muut [2012] ovat todenneet refaktoroinnin hyödyt systemaattisessa refaktorointiprojektissa, jossa Windowsin lähdekoodia oli muokkaamassa tähän tehtävään varattu tiimi. Tiimin refaktoroimien ohjelmien osien riippuvuudet muista komponenteista vähenivät, ja niiden ylläpidettävyys helpottui. Näissä komponenteissa oli myös vähemmän julkaisun jälkeisiä ohjelmistovirheiden korjauksia kuin niissä Windowsin osissa, joihin lähdekoodia oli muutettu muista syistä kuin refaktoroinnin seurauksena. Tämä tutkimus ei ottanut kantaa siihen, mikä oli refaktorointiin käytetty työmäärä suhteessa uuden ohjelmakoodin kirjoittamiseen, eli olisiko uuden ohjelmakoodin laatu ollut yhtä hyvä, jos se olisi käynyt läpi samankaltaisen koodin katselmoinnin. Leitch ja Stroulia [2003] pyrkivät mittaamaan refaktoroinnin aiheuttamaa työmäärää vertaamalla ohjelmiston refaktoroitua ja refaktoroimatonta versiota COCOMO II -kustannusarviomallin avulla, mutta he eivät tutkineet refaktoroinnin vaikutusta ohjelmiston laatuun.

2.5 Ohjelmistokehykset

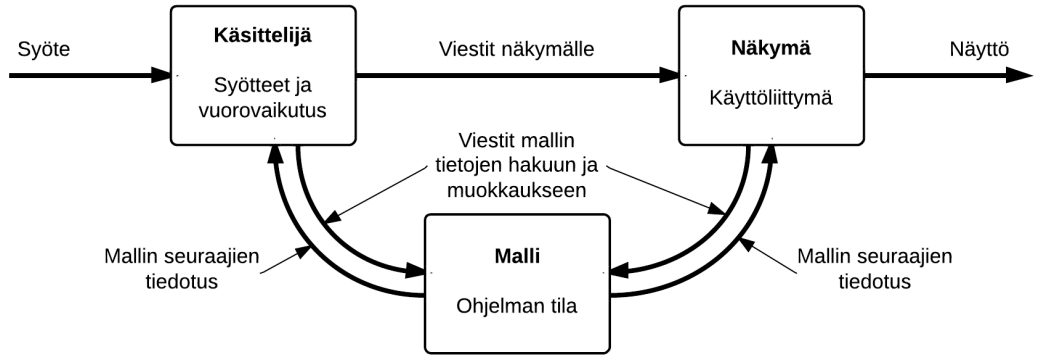
Luokkakirjastoja ja ohjelmistokehyksiä käytetään nopeuttamaan ja helpottamaan sovelluskehitystä. Ohjelmistokehyksen tarjoamat valmiit toiminnot paitsi vähentävät vaadittavaa kehitystyötä, myös helpottavat kommunikointia dokumentoitujen

rajapintojen avulla. Johnsonin ja Footen [1988] mukaan ohjelmistokehys on "luokkakokoelma, joka ilmentää ratkaisuluonnoksen toisiinsa liittyvien ongelmaryhmän ongelmille, ja tukee uudelleenkäyttöä suuremmassa mittakaavassa kuin yksittäiset luokat". Mattsson [2000] valitsi edellämainitun Johnsonin ja Footen [1988] määritelmän omasta mielestään parhaaksi useista erilaisista vaihtoehdoista, mutta jätti pois vertauksen yksittäisiin luokkiin. Myöhemmin Johnson [1997] on myös määritellyt ohjelmistokehyksen ohjelmiston rungoksi, jonka ohjelmistokehittäjä voi muokata tarkoitukseen sopivaksi.

2.5.1 Ohjelmistokehysten historia

Ohjelmistokehykset saivat alkunsa 1980-luvun alkupuolella olio-ohjelmoinnin yleistymisen myötä. Smalltalk [Kay, 1993] on Simulan [Wexelblat, 1981] inspiroima ohjelmointikieli, johon kuului MVC-arkkitehtuurin [Krasner & Pope, 1988] ohjelmistokehys käyttöliittymien luomista varten. Kuvassa 2.3 visualisoitu MVC-arkkitehtuuri (Model-View-Controller eli malli-näkymä-käsittelijä) on ohjelmiston suunnittelumalli, joka pyrkii erottamaan käyttöliittymän sovelluksen käsittelemästä datasta ja ohjelmalogiikasta. Mallin tehtäviin kuuluu ohjelmiston käsittelemän tiedon mallintaminen ja tallentaminen sekä ohjelmiston tilatiedon ylläpito. Näkymä määrittää ohjelman käyttöliittymän ja mallin sisältämien tietojen näyttämisen, ja käsittelijä ohjaa näkymää ja mallia käyttäjältä tulevan syötteen mukaisesti [Krasner & Pope, 1988].

Samaan aikaan, kun Smalltalkilla käytettiin MVC-mallin ohjelmistokehystä, Applen Lisa Toolkit -kehystä [Simonoff, 1983] käytettiin ohjelmien tuottamiseen Lisa-tietokoneelle. Apple jatkoi Lisan työkalujen kehittämistä ja julkaisi 1985 oliokeskeisen ohjelmistokehyksen nimeltä MacApp [Wilson *et al.*, 1990]. MacAppin kehitys oli vaihtelevaa, mutta samoihin periaatteisiin nojautuvia Microsoftin MFC- (Microsoft Foundation Classes) ja Borlandin OWL-kirjastoa (Object Windows Library) päivitettiin usean vuoden ajan säännöllisesti. MFC-kirjasto oli mukana Mattssonin [2000] ohjelmistokehysten vakautta tutkivassa työssä. Borland korvasi OWL-kirjaston Visual Component Library -kirjastolla, ja Microsoft on keskittynyt .NET-ohjelmistokehykseen, vaikkakin MFC on edelleen käytössä. Applen MacApp jäi lopulta NeXTSTEP-järjestelmästä [Isaacson, 2011] periytyvän Cocoa-kirjaston [Apple, 2013] jalkoihin.



Kuva 2.3 MVC-malli Krasnerin ja Popen [1988] mukaan.

2.5.2 Ohjelmistokehysten hyödyt ja haitat

Ohjelmistokehysten käytön etuja on tutkittu muunmuassa ohjelmistokoodin uudelleenkäytettävyyden ja ylläpidettävyyden kannalta, ja esimerkiksi Mattsson [2000] on kvalitatiivisella analyysillä todennut ohjelmistokehysten hyödyllisyyden työmäärän vähentämiseksi. Hän tutki kehyksen päivittämisen aiheuttamaa työmäärää käyttäen historiallista dataa useasta eri ohjelmistokirjastosta, ja toteaa ohjelmistokehyksen vakauden vaikuttavan työmäärään merkittävästi. Ohjelmistokehyksen käytöllä voi olla siis myös negatiivisia vaikutuksia. Mikäli ohjelmistokehys ei täytä sovellusalueen vaatimuksia riittävän kattavasti tai ei sovellu arkkitehtuuriltaan tietyn ongelman ratkaisuun, voi olla tehokkaampaa toteuttaa ohjelmisto jollain muulla ohjelmistokehyksellä tai olla kokonaan käyttämättä valmista ohjelmistokehystä.

Kun ohjelmistokehys sopii sovellusalueen tehtäviin ja tarjoaa tarvittavat työkalut ongelmien ratkaisuun, sitä voidaan käyttää oikeaoppisesti käyttämällä kehyksen julkisia rajapintoja koskematta ohjelmistokehyksen sisäiseen logiikkaan. Mikäli avoimen lähdekoodin ohjelmistokehyksessä on puutteita joko arkkitehtuurin tai sovellusalueen vaatimien ominaisuuksien suhteen, sitä voidaan yleensä muokata tarkoitukseen sopivaksi. Tämä saattaa kuitenkin rikkoa yhteensopivuuden ohjelmistokehyksen uudempien versioiden kanssa, eikä muokkaus yleensä ole tarkoituksenmukaista tai kannattavaa ylläpidettävyyden näkökulmasta. Ohjelmistokehys, kuten mikä tahansa ohjelmisto, muuttuu ja kehittyy muuttuvien vaatimusten mukana. Tämä aiheuttaa haasteita kehystä käyttävien ohjelmistojen ylläpidettävyydelle, kun muuttuvan kehyksen päivittäminen aiheuttaa muutoksia kehystä käyttävässä ohjelmistossa [Kim *et al.*, 2012].

2.5.3 WWW-ohjelmistokehykset

1990-luvun aikana WWW kehittyi nopeasti maailmanlaajuiseksi tietoverkoksi. WWW-sivut olivat aluksi staattisia, mutta 90-luvun loppupuolella ne muodostuivat jo paljon enemmän aktiivisista elementeistä ja palvelinpuolen ohjelmistoista, jotka myös sisälsivät tietokantaan tallennettua dataa [Ginige & Murugesan, 2001]. Nämä ohjelmistot oli pääsääntöisesti toteutettu ilman ohjelmistotuotannossa käytettyjä prosesseja samaan tapaan kuin ohjelmistotuotannon alkuaikoina koostettiin ohjelmistoja ilman kunnollisia työkaluja, vaatimusmäärittelyjä, laadunvalvontaa tai testausta [Coda *et al.*, 1998; Ginige & Murugesan, 2001]. Coda ja muut [1998] esittivät oliomallin WOOM (Web Object Oriented Model), jolla voidaan formalisoida WWW-sivuston käsitteet kuten sisältö, navigaatio ja palveluarkkitehtuuri. He myös toteuttivat Java-ohjelmointikielellä tämän mallin toteuttavan työkalun. WOOM-työkalu sisälsi WOOM-luokkakirjaston, joka on yksi ensimmäisiä WWW-sivustojen kehittämiseen tarkoitettuja dokumentoituja kirjastoja. Javalle kehitettiin vuoteen 2002 mennessä myös Struts-ohjelmistokehys [Apache Software Foundation, 2013], joka käyttää MVC-suunnittelumallia [Krasner & Pope, 1988].

2.5.4 PHP ja Joomla

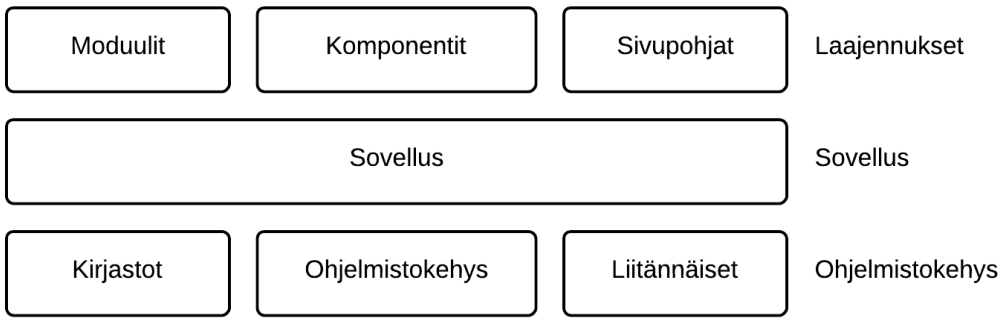
PHP-ohjelmointikielen historia juontaa juurensa WWW-sivustojen muuttumisesta staattisista aktiivisiksi. PHP sai alkunsa pienten kotisivujen aktiivisten elementtien luomiseen tarkoitettuna työkaluna, mutta 90-luvun aikana se kehittyi omaksi erityisesti WWW-ympäristöön tarkoitetuksi ohjelmointikielekseen [Lerdorf & Tatroe, 2002]. PHP:n etuina oli matala ohjelmoinnin aloittamiskynnys, kun tulkki oli helppo asentaa palvelimille myös yksittäisille käyttäjille ja se oli esiasennettuna suurella osalla internetpalvelimia. Näin se saavutti suuren käyttäjäkunnan nopeasti. Vuonna 2000 julkaistu PHP:n versio 4 sisälsi rajallisen tuen olio-ohjelmoinnille, jolloin sillä voitiin toteuttaa helpommin myös formaalimpaa ohjelmistokehitystä. Täydellisemmällä olio-ohjelmoinnin tuella varustettu PHP 5 julkaistiin vuonna 2004 [Lerdorf & Tatroe, 2002; The PHP Group, 2013a].

PHP:lla on toteutettu useita erityyppisiä ohjelmistoja, joiden käyttämistä luokkakirjastoista on muodostettu jossain ohjelmiston elinkaaren vaiheessa ohjelmistokirjastoja. Vuonna 2000 aloitettu Mambo-projekti [Wikipedia, 2013] on yksi tällaisista ohjelmistoista. Se on sisällönhallintajärjestelmä, joka toteuttaa samoja asioita

kuin WOOM-työkalu. Sisällönhallintajärjestelmän ominaisuuksiin kuuluu esimerkiksi sisältöartikkelien kirjoittaminen WYSIWYG-editorissa, artikkelien julkaisu, poisto ja muokkaaminen tai käyttäjien ja käyttöoikeuksien hallinnointi. Mambo-sisällönhallintajärjestelmä haarautui vuonna 2005 Joomla-nimiseksi projektiksi [Joomla, 2013c] hallinnollisten ristiriitojen vuoksi. Puhtaasti ohjelmistokehyksiksi suunniteltuja PHP-ohjelmistokehyksiä ovat esimerkiksi CakePHP [CakePHP, 2013], Symfony [Symfony, 2013] sekä PHP:n kehittäjien luoma Zend Framework [Zend, 2013]. Kaikkien edellämainittujen ohjelmistokehysten suunnittelussa on käytetty MVC-mallia.

Joomla-järjestelmä muodostuu ohjelmistokehyksestä ja sisällönhallintajärjestelmästä. Sisällönhallintajärjestelmä sisältää ohjelmiston ydintoimintojen lisäksi laajennuksia jotka jaetaan komponentteihin, moduuleihin ja sivupohjiin kuvan 2.4 mukaisesti [Joomla, 2013a]. Komponentit on yleensä toteutettu MVC-mallia hyödyntäen, mutta ne voidaan toteuttaa muillakin tavoilla, sillä Joomlan arkkitehtuuri ei ota kantaa kuin komponentin nimeämiseen ja sijaintiin hakemistopuussa. Joomlan komponentit ovat itsenäisiä sovelluksia, jotka käyttävät sisällönhallintajärjestelmän tarjoamia palveluita ohjelmistokehyksen tapaan. Esimerkiksi VirtueMart [VirtueMart, 2013] on WWW-kauppaohjelmisto ja Kunena [The Kunena Project, 2013] on keskustelufoorumi. Ne voidaan asentaa samaan Joomla-instanssiin ja ne tarjoavat merkittävästi lisäominaisuuksia sisällönhallintajärjestelmään. Moduulit ovat pieniä lisäosia, jotka saattavat esimerkiksi näyttää säätietoja sivustolla. VirtueMart-komponentin ostoskoritoiminto on toteutettu Joomlan moduulina. Sivupohjat määrittävät WWW-sivuston ulkoasun. Lisäksi Joomlan laajennuksiin kuuluu liitännäisiä, jotka toimivat taustalla reagoiden tapahtumiin. Esimerkiksi sivustolle kirjautuminen voidaan tehdä kolmannen osapuolen liitännäisen avulla vaikkapa Facebook-tunnuksilla.

Joomlan 1.0-versio oli käytännössä sama ohjelmisto kuin Mambo 4.5.2, koska tavoitteena oli säilyttää yhteensopivuus ja varmistaa loppukäyttäjille helppo siirtyminen uudelle alustalle. Versio 1.5 oli ensimmäinen nykyisellä organisaatiolla tuotettu varsinainen Joomla-versio [Hutchinson, 2010], ja se julkaistiin vuonna 2008. Versiolle 1.5 tarjottiin ohjelmistopäivityksiä vuoteen 2012 asti, ja se nimettiin ensimmäiseksi “LTS” eli pitkällä aikavälillä tuetuksi Joomlan versioksi. Tässä vaiheessa julkaisujen aikaväliä haluttiin lyhentää kehitystahdin nopeuttamiseksi, ja versiot 1.6, 1.7 ja 2.5 julkaistiin puolen vuoden välein. Versio 2.5 on myös LTS-versio, ja syksyllä 2012 julkaistusta 3.0-versiosta kehitetään LTS-versiota



Kuva 2.4 Joomla-arkkitehtuuri [Joomla, 2013a].

3.5, joka julkaistaan keväällä 2014. Joomla-järjestelmän päivityssykli on siis vakautunut vasta hiljattain, ja pitkällä aikavälillä tuettujen versioiden aikataulu on varmistunut vasta viimeisen vuoden aikana.

Samaan aikaan Joomla 1.6-version julkaisun kanssa Joomla-sisällönhallintajärjestelmä erotettiin Joomla! Platform -ohjelmistokirjastosta [Joomla, 2013b], ja tätä jakoa on tehty selkeämmäksi myöhemmissä julkaisuissa. Nykyään Joomlan sisällönhallintaohjelmisto on erotettavissa sen käyttämästä ohjelmistokehyksestä, mutta versiossa 1.5 niillä oli vielä riippuvuuksia toisiinsa.

3 Ohjelmistokehityksen työmäärän arviointi

Ohjelmistojen kehityksen työmäärää voidaan arvioida heuristisilla tai parametrisilla tavoilla. Heuristiset tavat sisältävät muun muassa samankaltaisten järjestelmien vertailun ja asiantuntija-arvioinnin. Parametriset tavat tarkoittavat arviointitapoja, joissa arviointi perustuu ohjelmiston tai siihen liittyvien ominaisuuksien matemaattiseen lasketaan [PMHut, 2008]. Tällaisia menetelmiä on mahdollista soveltaa ilman syvällistä tietämystä kehitettävästä ohjelmistosta ja jotkut parametrit voidaan laskea automaattisilla työkaluilla. Työmäärän, kustannusten ja aikataulun arvioiminen liittyvät toisiinsa läheisesti, ja esimerkiksi Putnamin malli arvioi tarvittavaa miestyövoimaa tietyllä projektin hetkellä [van Vliet, 2008]. Keskitymme seuraavassa menetelmiin, jotka pyrkivät arvioimaan työmäärää, emme kustannuksia tai aikataulua.

3.1 Constructive Cost Model COCOMO

Vuonna 1981 julkaistu COCOMO 81 -malli on yksi parhaiten dokumentoiduista kustannusarviomenetelmistä. Perusmuodossaan se tiivistyy kaavaan, joka arvioi työmäärää E ohjelmiston koodirivien määrän KLOC perusteella. Lähdekoodin rivien määrän laskemiseen on erilaisia kriteereitä: COCOMO 81:n KLOC tarkoittaa tuhatta loogista lähdekoodiriviä [van Vliet, 2008]. Kaava on seuraava:

$$E = b * KLOC^c. \tag{3.1}$$

COCOMO 81:n perusmuoto luokittelee ohjelmistot kolmeen eri luokkaan: orgaanisiin (organic), sulautettuihin (embedded) ja puoli-irrallisiin (semi-detached). Orgaaniset projektit ovat suhteellisen pieniä, pienten tiimien toteuttamia ja joustavia. Sulautetut ovat toisesta ääripäästä ja puoli-irralliset ovat jotain näiden väliltä. Luokittelu vaikuttaa kaavan muuttujiin b ja c, ja sitä kautta menetelmän antamaan arvioon.

COCOMO 81:een kuuluu myös tarkemmat menetelmät (Intermediate COCOMO ja Detailed COCOMO, keskimääräinen ja yksityiskohtainen), jotka ottavat huomioon 15 eri kustannuksiin vaikuttavaa muuttujaa, ja tarkentavat perusmuodon antamaa arviota riippuen esimerkiksi arvioitavan ohjelmiston monimutkaisuudesta. Näihin ei ole syytä syventyä, sillä COCOMO 81 on kehitetty vesiputousmallin ohjelmistokehitykselle ympäristöön, jossa ohjelmistoja ajetaan keskustietokoneella

eikä kehitettävällä ohjelmistolla ole modernia käyttöliittymää, joten sen soveltaminen nykyaikaisiin ohjelmistoihin on ongelmallista. Menetelmästä on olemassa päivitetty versio COCOMO II, joka käsitellään tarkemmin myöhemmin.

3.2 Funktiopisteanalyysi

Funktiopisteanalyysi (Function point analysis) on arviointitapa, jolla pyritään arvioimaan ohjelmiston koodirivien määrää sen sisältämien rajapintojen, käytettyjen tietotyyppien ja niihin kohdistettujen toimintojen määrän perusteella. Näiden avulla voidaan arvioida lopullista lähdekoodin rivien määrää käytetystä ohjelmointikielestä riippuen.

Funktiopisteanalyysissä lasketaan seuraavat asiat [Matson *et al.*, 1994]:

- *Syötteet (I).* Käyttäjän syöttämien erilaisten tietotyyppien lukumäärä.
- *Tulosteet (O).* Ohjelmiston tuottamien tietotyyppien määrä.
- *Kyselyt (E).* Niiden ohjelmiston toimintaa ohjaavien syötteiden lukumäärä, jotka eivät muokkaa tietueita.
- *Sisäiset tietueet (L).* Ohjelmiston sisäisesti luomien tietotyyppien, kuten indeksitiedostojen lukumäärä.
- *Rajapinnat (F).* Muiden ohjelmistojen kanssa toimimiseen tarkoitettujen rajapintojen lukumäärä.

Muuntamattomien funktiopisteiden määrä (Unadjusted Function Points) UFP saadaan painottamalla määrät tietotyyppien ja toimintojen painokertoimilla, joita voidaan säätää esimerkiksi tietueiden monimutkaisuuden mukaan. Keskimääräisillä kertoimilla kaava on $UFP = 4I + 5O + 4E + 10L + 7F$. Funktiopisteet muunnetaan tarkemmiksi ohjelmiston ominaisuuksien perusteella. Van Vlietin [2008, s. 161] mukaan arvioitavia ominaisuuksia on esimerkiksi hajautetut ominaisuudet, tehokkuus, uudelleenkäytettävyys, asennuksen helppous ja niin edelleen, yhteensä arvioitavia asioita on neljätoista. Ohjelmiston ominaisuudet vaikuttavat asteikolla nollasta (ei vaikutusta) viiteen (suuri vaikutus), ja yhteen laskettuna luvut muodostavat *vaikuttavuuden* (degree of influence) DI. Vaikuttavuuden avulla lasketaan

tekninen monimutkaisuus (technical complexity factor) $TCF = 0.65 + 0.01 * DI$ ja edelleen muunnetut funktiopisteet $FP = UFP * TCF$.

Muunnetut funktiopisteet voidaan muuntaa suoraan tarvittavan lähdekoodin määräksi [Jones & SPR, 2005] ja käyttää tätä koodin määrää työmäärän arvioimiseen muiden arviointimenetelmien tarjoamien kaavojen avulla.

3.3 Oliopisteiden arviointi

Oliopisteiden arvioinnin (Object point extraction) tavoitteena on Bankerin ja muiden [1992] mukaan aiempaa paremmin arvioida neljännen sukupolven ohjelmointikielten ja ICASE-työkalujen avulla toteutettujen ohjelmistojen kokoa. Siinä lasketaan ohjelmiston näyttöjen ja raporttien lukumäärä ja arvioidaan näiden monimutkaisuutta. Lisäksi otetaan huomioon kolmannen sukupolven ohjelmointikielillä (3GL) luodut oliot. Boehm ja muut [1995] ovat koostaneet yhden COCOMO II -mallin arviointimenetelmistä perustuen oliopistearviointiin.

3.4 COCOMO II

COCOMO II on päivitetty versio COCOMO 81 -mallista. Se kehitettiin vastaamaan 1990- ja 2000-lukujen muuttuneiden ohjelmistokehitysmenetelmien tarpeita. COCOMO II koostuu kolmesta erilaisesta arviointimenetelmästä, joita on tarkoitus soveltaa ohjelmistoprojektin eri vaiheissa. Aivan aluksi, kun ohjelma on prototyyppiasteella, voidaan käyttää Application Composition -mallia, jossa keskitytään käyttäjälle näkyvien näkymien ja raporttien määrään. Menetelmässä käytetään oliopisteiden laskemista lisättynä erilaisilla taulukkoarvoilla työn vaativuudesta [Boehm *et al.*, 1995]. Kun ohjelmistoa kehitetään eteenpäin, COCOMO II tarjoaa työkaluksi Early Design -mallin, jossa mitataan ohjelmiston käsittelemiä ja sisältämiä tietovarastoja funktiopisteanalyysillä. Lähdekoodin rivien määrä arvioidaan tietovarastojen määrän ja käytetyn ohjelmointikielen perusteella, ja tätä rivien määrää käytetään työmäärän arvioimisen lähtökohtana. Lisäksi käytetään kuutta erilaista työmääräkerrointa, jotka vaikuttavat arvioon. Valmiimpia ohjelmistoja varten on olemassa Post Architecture -malli, joka keskittyy koodirivien määrän arviointiin, ja käyttää perustana samaa kaavaa kuin Early Design, mutta arviota tarkentavien muuttujien määrä kasvaa kuuteentoista [Boehm *et al.*, 2000]. Early Design -mallin kuusi työmääräkerrointa muodostetaan yhdistämällä

Post Architecture -mallin hienojakoisempia työmääräkertoimia, joita käydään läpi tarkemmin alakohdassa 3.4.2.

3.4.1 Application Composition

COCOMO II:n oliopistearviointi on yhdistelmä Kauffmanin ja Kumarin [1993] menetelmästä sekä tuottavuustiedoista Bankerin ja muiden [1992] tutkimista 19 projektista. Seuraavaksi esitettävissä taulukoissa 3.1 ja 3.2 *palvelin* tarkoittaa palvelinpään tietovarastoja ja *asiakas* henkilökohtaisen työaseman tietovarastoja. 3GL-komponentti tarkoittaa kolmannen sukupolven ohjelmointikielellä [Minerich, 2009] toteutettua ohjelmiston osaa. Boehm ja muut [1995] määrittävät COCOMO II:n oliopistearvioinnin vaiheet seuraavasti:

- Laske oliot. Arvioi ohjelmistoon tulevien näyttöjen, raporttien ja 3GL-komponenttien määrä. Näitä elementtejä kutsutaan olioiksi, eikä termiä pidä sekoittaa olio-ohjelmoinnissa käytettävään olion käsitteeseen.

- Luokittele jokainen olio yksinkertaiseksi, keskitasolle tai monimutkaiseksi sen ominaisuuksien mukaan. Taulukon 3.1 mukaisesti näytöille annetaan monimutkaisuus niiden sisältämien näkymien ja niiden käyttämien datataulujen perusteella, raporteille vastaavasti taulukon 3.2 mukaisesti osioiden ja datataulujen perusteella.

- Painota jokaisen olion pisteet niille annetun monimutkaisuuden perusteella taulukon 3.3 mukaisesti. Esimerkiksi yksinkertainen näyttö saa yhden oliopisteen, keskitason raportti saa viisi ja monimutkainen 3GL-komponentti kymmenen pistettä.

Näytöt			
	Datataulujen lähde ja lukumäärä		
Sisältyvien näkymien määrä	**Yht. < 4 (<2 palvelin <3 asiakas)**	**Yht. < 8 (2/3 palvelin 3-5 asiakas)**	**Yht. 8+ (>3 palvelin >5 asiakas)**
<3	yksinkertainen	yksinkertainen	keskitaso
3-7	yksinkertainen	keskitaso	monimutkainen
> 8	keskitaso	monimutkainen	monimutkainen

Taulukko 3.1: Näyttöjen lajittelu.

Raportit			
	Datataulujen lähde ja lukumäärä		
Sisältyvien osioiden määrä	**Yht. < 4 (<2 palvelin <3 asiakas)**	**Yht. < 8 (2/3 palvelin 3-5 asiakas)**	**Yht. 8+ (>3 palvelin >5 asiakas)**
0 tai 1	yksinkertainen	yksinkertainen	keskitaso
2 tai 3	yksinkertainen	keskitaso	monimutkainen
4+	keskitaso	monimutkainen	monimutkainen

Taulukko 3.2: Raporttien lajittelu.

	Monimutkaisuus / Paino		
Oliotyyppi	**Yksinkertainen**	**Keskitaso**	**Monimutkainen**
Näyttö	1	2	3
Raportti	2	5	8
3GL-komponentti			10

Taulukko 3.3: Oliopisteiden painotus olioiden monimutkaisuuden perusteella

- Määritä oliopisteet: laske kaikki painotetut olioiden pisteet OP yhteen.

- Arvioi uudelleenkäytön $REUSE\%$ määrä prosentteina ja laske uusien oliopisteiden määrä $NOP = (OP)(100 - REUSE\%)/100$.

- Määritä tuottavuus, $PROD = NOP/PM$ taulukon 3.4 avulla. Taulukkoarvot muuttujalle PROD on analysoitu Bankerin ja muiden [1992] tekemästä tutkimuksesta ja ne perustuvat CASE-työkalulla toteutettujen projektien tuottavuuteen [Boehm *et al.*, 1995].

- Laske arvioitu henkilötyökuukausien määrä $PM = NOP/PROD$.

Sovelluskehittäjän kokemus ja taidot	Erit. pieni	Pieni	Normaali	Suuri	Erit. suuri
ICASE valmius	Erit. pieni	Pieni	Normaali	Suuri	Erit. suuri
PROD	4	7	13	25	50

Taulukko 3.4: Ohjelmointitiimin tuottavuus

3.4.2 Early Design ja Post Architecture

COCOMO II Early Design ja Post Architecture -malleilla on hyvin paljon yhteistä. Niillä voidaan arvioida sekä työmäärää, hintaa että aikataulua, mutta keskitymme mallien kuvauksissa vain työmäärää koskeviin osiin. Mallit käyttävät samoja kaavoja työmäärän ja aikataulun arvioimiseen, mutta Post Architecture -mallissa otetaan huomioon enemmän ohjelmiston ja prosessin attribuutteja, ja tämä antaa tulokseksi tarkemman arvion. Ohjelmistojen työmäärää arvioidaan molemmissa malleissa seuraavalla kaavalla [Boehm *et al.*, 2000]:

$$\begin{aligned} PM_{NS} &= A \times Size^{E} \times \prod_{i=1}^{n} EM_i, \\ \text{kun } E &= B + 0.01 \times \sum_{j=1}^{5} SF_j. \end{aligned} \tag{3.2}$$

Kaavan EM tarkoittaa työmääräkertoimia (Effort Multiplier), joita on Post Architecture -mallissa 16 ja Early Design -mallissa 6. Muuttujien A, B, EM_1, ..., EM_{16}, SF_1, ..., ja SF_5 arvot saadaan COCOMO II:n tietokannasta, johon on hyödynnetty oikeita työmäärätietoja 161 eri projektista [Boehm *et al.*, 2000]. Taulukkoarvot A:lle ja B:lle ovat $A = 2.94$ ja $B = 0.91$. Muuttujien arvot ovat muuten samat molemmissa malleissa, mutta Early Design -mallin työmääräkertoimet muodostetaan yhdistämällä niitä vastaavat ominaisuudet Post Architecture -mallin useammasta eri työmääräkertoimesta COCOMO II -ohjekirjassa [Boehm *et al.*, 2000, s. 36] kuvatulla tavalla.

Ohjelmiston koko määritetään COCOMO II -mallissa joko tuhansina loogisina lähdekoodin riveinä tai muuntamattomien funktiopisteiden määränä. Vain uuden tai muunnetun lähdekoodin määrä huomioidaan työmääräarvioissa, ja funktiopisteitä arvioitaessa on huomioitava, ettei muuntamattomia funktiopisteitä muunneta 14 arviointikriteerin perusteella kuten tavallisesti funktiopistearvioinnissa.

Muista projekteista uudelleenkäytetyn tai muunnetun lähdekoodin määrä muunnetaan vastaamaan uuden lähdekoodin määrää, jotta uudelleenkäytöstä johtuva työmäärä voidaan sisällyttää ohjelmiston koon arvioon. Ohjelmiston uudelleenkäyttö huomioidaan epälineaarisella kaavalla, johon tarvitaan muunnettavan lähdekoodin määrä ja kolme muunnoksen astetta kuvaavaa muuttujaa:

- suunnittelun muutososuus (design modified, DM)
- muunnetun lähdekoodin osuus (code modified, CM)
- integroinnin vaativuuden osuus (integration effor required, IM).

Mainitut muuttujat, ohjelmiston ymmärrettävyys ja se, miten hyvin ohjelmistokehittäjät tuntevat uudelleenkäytettävän lähdekoodin, huomioidaan käyttämällä taulukkoarvoja COCOMO II -mallista [Boehm *et al.*, 2000]. Ohjelmiston ymmärrettävyys (software understanding increment, SU) arvioidaan kolmen eri kategorian keskiarvona. Arvioitavat asiat ovat ohjelman rakenne (structure), selkeys (application clarity) ja kuvaavuus (self-descriptiveness). Ymmärrettävyyden ollessa pieni SU saa arvoksi 50 ja suurella ymmärrettävyydellä 10 [Boehm *et al.*, 2000, s. 10]. Se, miten hyvin ohjelmistokehittäjät tuntevat ohjelmakoodin, arvioidaan välille 0-1, jossa 1 on täysin tuntematon ja 0 on täysin tunnettu. Myös vaatimusten muuttuvuus, automaattiset lähdekoodin muunnostyökalut ja ohjelmiston ylläpidon työmäärän vaikutus voidaan huomioida.

Mittakaavamuuttujat

Ohjelmiston koko *Size* skaalataan mittakaavamuuttujilla SF (Scale Factors), jotka vaikuttavat kaavan (3.2) muuttujaan E. Jos $E = 1.0$, projektin koon negatiiviset ja positiiviset vaikutukset ovat tasapainossa. Tätä arvoa käytetään usein suhteellisen pienille projekteille. Jos $E > 1.0$, projektin mittakaava kasvattaa kokonaistyömäärää. Suuremmissa projekteissa työmäärää lisää esimerkiksi eri tiimien välinen kommunikaatio tai eri osa-alueiden integraatio [Banker *et al.*, 1992].

Mittakaava-muuttujat	Erit. pieni	Pieni	Normaali	Suuri	Erit. suuri	Suurin
PREC	6.20	4.96	3.72	2.48	1.24	0
FLEX	5.07	4.05	3.04	2.03	1.01	0
RESL	7.07	5.65	4.24	2.83	1.41	0
TEAM	5.48	4.38	3.29	2.19	1.10	0
PMAT	7.80	6.24	4.68	3.12	1.56	0

Taulukko 3.5: Mittakaavamuuttujien arvot.

Mittakaavamuuttujat ovat nimeltään ennakoitavuus (Precedentedness, PREC), kehityksen joustavuus (Development Flexibility, FLEX), arkkitehtuuri / riskien ratkaisu (Architecture / Risk Resolution, RESL), työryhmän yhtenäisyys (Team Cohesion, TEAM) sekä prosessin kypsyys (Process Maturity, PMAT). Mitä pienempi muuttujan arvo on, sitä vähemmän se vaikuttaa työmääräarvioon. Esimerkiksi huonosti ennakoitavissa oleva projekti saa suuren mittakaavamuuttujan PREC, mutta mikäli aikaisemmin on toteutettu hyvin samankaltainen projekti, sen ennakoitavuus on suuri ja muuttujan PREC arvo lähestyy nollaa. Kuten taulukosta 3.5 voidaan lukea, muuttujien arvot vaihtelevat välillä [0..8]. Oikean suuruusluokan valinta kullekin muuttujalle käydään kattavasti läpi COCOMO II:n ohjekirjassa [Boehm *et al.*, 2000].

Post Architecture -mallin työmääräkertoimet

Työmääräkerrointen tarkoituksena on muuntaa työmääräarviota PM vastaamaan paremmin toteutettavaa ohjelmistoa sen ominaisuuksien perusteella. Jokaisen kertoimen normaaliarvona on 1.0, joka ei muuta työmääräarviota suuntaan tai toiseen. Normaalista poikkeavat määritykset muuttavat työmääräarviota. Esimerkiksi korkea arvo ohjelmiston vaadittavalle luotettavuudelle (RELY) nostaa kokonaistyömääräarviota 10%, erittäin korkea arvo 26%. Muuttujien arvot voidaan valita myös taulukkoarvojen välistä lineaarisesti tai joissain tapauksissa epälineaarisesti [Boehm *et al.*, 2000, s. 25]. Kertoimet jaetaan tuotteen ominaisuuksiin (taulukko 3.6), tuotealustaan (taulukko 3.7), henkilöstöön (taulukko 3.8) ja projektiin (taulukko 3.9) liittyviin kertoimiin.

Taulukossa 3.6 muuttujat RELY, RUSE ja DOCU liittyvät ohjelmistolle asetettuihin vaatimuksiin. RELY on vaadittu luotettavuus, RUSE mittaa sitä, aiotaanko ohjelmistoa uudelleenkäyttää, ja DOCU mittaa vaaditun dokumentaation määrää. DATA ja CPLX määrittävät suoraan ohjelmiston ominaisuuksia, nimittäin tarvittavan testidatan määrää ja ohjelmiston monimutkaisuutta. Kertoimien määrityksessä ei käytetä metriikoita kuten Chidamberin ja Kemererin [1994] analysoimat luokan metodien määrä tai perimäpuun syvyys, koska arviointimenetelmällä on tarkoitus arvioida vielä toteuttamattoman ohjelmiston monimutkaisuutta. Tuotteen monimutkaisuus arvioidaan jakamalla monimutkaisuuden osa-alueet viiteen eri kategoriaan ja arvioimalla näitä erikseen. Esimerkiksi yksi kategoria on laskennallisten operaatioiden monimutkaisuus, jonka skaalan toisessa ääripäässä on yksinkertaisten yhteenlaskujen tai kertolaskujen ratkaisu, toisessa ääripääs-

sä satunnaisdatan analyysi rinnakkaislaskennalla. Eri kategorioiden arvioiden keskiarvosta voidaan päätellä yksi yhdistetty luku monimutkaisuudelle.

Kerroin	**Nimi**	**Kuvaus**
RELY	Required software reliability	Vaadittu ohjelmiston luotettavuus
DATA	Data Base Size	Testidatan määrä
CPLX	Product Complexity	Tuotteen monimutkaisuus
RUSE	Developed for Reusability	Uudelleenkäytettävä toteutus
DOCU	Documentation Match to Life-Cycle Needs	Dokumentaation ja ohjelmiston elinkaaren vastaavuus

Taulukko 3.6: Tuotteen ominaisuuksiin liittyvät työmääräkertoimet.

Tuotealustan työmääräkertoimissa taulukossa 3.7 suoritusaika- ja tallennusmäärärajoitukset kuvaavat suoritusympäristölle asetettavia rajoituksia, ja alustan pysyvyys tarkoittaa sen alustan muuttuvuutta, jota ohjelmisto suoraan kutsuu. Käyttöjärjestelmän alustana voi olla laitteisto, verkkosovelluksen alustana voi olla esimerkiksi käyttöjärjestelmä, verkko, tietokanta ja ohjelmistokirjasto. Taulukon 3.8 henkilöstöön liittyvissä työmääräkertoimissa merkillepantavaa on, että suunnittelijoiden kyvyt (ACAP) ja ohjelmoijien kyvyt (PCAP) mittaavat suunnittelijoiden ja ohjelmoijien kommunikaatiotaitoja, eivätkä niinkään teknisiä taitoja. Toteuttajien tieto ja kokemus mitataan erikseen muuttujilla APEX, PLEX ja LTEX. Lisäksi otetaan huomioon henkilöstön jatkuvuus PCON.

Projektin yleisiä asioita taulukossa 3.9 ovat automaattisten työkalujen käytön määrä (TOOL) ja se, onko tuotantotiimejä useammassa eri sijainnissa (SITE). Useammassa eri sijainnissa työskentely vaikeuttaa kommunikointia, joten se myös hidastaa kehitystyötä. Projektin aikataulua (SCED) verrataan normaaliin vastaavaan projektiin ja päätellään, onko aikataulu tiukka vai väljä. Tämä vaikuttaa työtehoon projektin alku- ja loppuvaiheissa.

Kerroin	**Nimi**	**Kuvaus**
TIME	Execution Time Constraint	Suoritusaikarajoitus
STOR	Main Storage Constraint	Tallennusmäärärajoitus
PVOL	Platform Volatility	Alustan pysyvyys

Taulukko 3.7: Tuotealustaan liittyvät työmääräkertoimet.

Kerroin	**Nimi**	**Kuvaus**
ACAP	Analyst Capability	Suunnittelijoiden työn laatu ja sen kommunikointi
PCAP	Programmer Capability	Ohjelmoijien työn laatu ja kommunikaatiotaidot
PCON	Personnel Continuity	Henkilöstön jatkuvuus
APEX	Applications Experience	Kokemus vastaavista ohjelmistoista
PLEX	Platform Experience	Kokemus laitteisto-, ohjelmisto- ja ajoympäristöstä
LTEX	Language and Tool Experience	Kokemus ohjelmointikielestä ja työkaluista

Taulukko 3.8: Henkilöstöön liittyvät työmääräkertoimet.

Kerroin	**Nimi**	**Kuvaus**
TOOL	Use of Software Tools	Automaattisten työkalujen käyttöaste
SITE	Multisite Development	Hajautettu ohjelmistokehitys
SCED	Required Development Schedule	Vaadittu aikataulu

Taulukko 3.9: Projektiin liittyvät työmääräkertoimet.

Mallin kalibrointi

COCOMO II -mallin antaman arvion tarkkuus paranee huomattavasti, mikäli malli kalibroidaan ympäristön mukaiseksi. Kalibrointi muuttaa COCOMO II:n regressiokaavan (3.2) muuttujien A ja B arvoja. Kalibrointi suoritetaan aiempien organisaatiossa toteutettujen projektien tietojen perusteella. Siihen tarvitaan todellinen työmäärä, joka johonkin organisaation projektiin on käytetty, kyseisen projektin tuottaman tuotteen koko, mittakaavamuuttujat sekä työmääräkertoimet, ja siihen suositellaan käytettäväksi tietoja ainakin viidestä eri projektista. Kalibroinnissa lasketaan ensin muuntamaton työmääräarvio kaavan (3.2) avulla ilman kerrointa A, minkä jälkeen lasketaan todellisen työmäärän ja työmääräarvion luonnolliset logaritmit ja näiden erotus. Luonnollisten logaritmien erotusten keskiarvosta X lasketaan uusi A kaavalla $A = e^X$ [Boehm *et al.*, 2000, s. 68]. Näin kalibroidulla kaavalla voidaan arvioida tulevien projektien työmääriä siinä organisaatiossa ja ympäristössä, johon kaava on kalibroitu.

4 Menetelmä päivitettävyyden arviointiin

Menetelmän tavoitteena on selvittää, saadaanko vanhentunutta ohjelmistokehystä käyttävä ohjelmisto päivitettyä edullisemmin refaktoroimalla nykyinen ohjelmisto käyttämään uutta ohjelmistokehystä, vai saadaanko sama toiminnallisuus toteutettua pienemmällä työmäärällä toteuttamalla samat ominaisuudet uudella ohjelmistokehyksellä hyödyntämättä nykyistä lähdekoodia. Ehdottamani menetelmän ajatuksena on arvioida ohjelmiston toteutuksen työmäärää kahdella eri tavalla ja päätellä näiden arvioiden perusteella, kumpi tapa on taloudellisempi, ottamatta kantaa siihen kumpi tapa tuottaa laadukkaamman tai ylläpidettävämmän ohjelmiston. Valittu tapa saattaa vaikuttaa päivitetyn ohjelmiston laatuun [Kim *et al.*, 2012], mutta ohjelmiston laatu huomioidaan vain COCOMO II:n määrittämillä ohjelmiston ymmärrettävyyden parametreilla, jotta saadaan vertailukelpoiset arviot eri ratkaisujen työmääristä.

4.1 Menetelmän kuvaus

Ennen varsinaisen arvioinnin aloittamista määritetään arvioinnin tarve vertailemalla uuden ja vanhan ohjelmistokehyksen julkisten rajapintojen eroja. Mikäli ohjelmistokehyksen käyttötapa on muuttunut merkittävästi, jatketaan menetelmän suorittamista taulukon 4.1 mukaisesti. Jos kehys ei ole muuttunut, ohjelmiston päivittäminen käyttämään uutta ohjelmistokehystä on triviaalia, eikä arvioinnin jatkamiselle ole tarvetta.

Ensin arvioidaan nykyisen ohjelmiston toteutuksen työmäärä COCOMO II Post Architecture -mallin avulla, jossa käytetään lähdekoodin määränä todellista lähdekoodin määrää. Lisäksi samalla arvioidaan ohjelmiston refaktoroinnin työmäärä. Sen jälkeen päätellään ohjelmiston uudelleentoteuttamisen työmäärä arvioimalla COCOMO II Application Composition -mallin perusteella nykyisen ohjelmiston käyttöliittymien toteutuksen työmäärää.

Post Architecture -mallissa tarvittava ohjelmiston uudelleenkäytön määrä arvioidaan mittaamalla lähdekoodista osuus, joka käyttää ohjelmistokehyksen muuttuneita osia. Mikäli käyttöliittymien perusteella tehty arviointi tuottaa merkittävästi pienemmän työmääräarvion, toteamme, että uudelleen toteuttamalla voidaan toteuttaa sama ohjelmisto pienemmällä työmäärällä.

Vaihe 1	Vertaile rajapintoja luokittelemalla niiden tarjoamat luokat.
Vaihe 2	Laske lähdekoodin osuus, joka käyttää vanhan rajapinnan muutettuja luokkia
Vaihe 3	Arvioi nykyiseen ohjelmistoon käytetty työmäärä COCOMO II Post Architecture -mallin avulla
Vaihe 4	Arvioi päivittämisen työmäärä COCOMO II Post Architecture -mallin avulla
Vaihe 5	Arvioi uudelleentoteuttamisen työmäärä COCOMO II Application Composition -mallilla
Vaihe 6	Vertaa työmääriä ja päättele työmääräarvioiden perusteella paras lähestymistapa.

Taulukko 4.1: Päivitettävyyden arvioinnin menetelmä.

4.2 Arvioinnin kannattavuuden määritys

Sovelluksen ohjelmistokehystä päivitettäessä tulee ensin selvittää uuden kehyksen erot vanhaan nähden. Jos ohjelmistokehys on vakaa [Mattsson, 2000], niin voi olla, ettei ohjelmiston päivittämiseksi tarvitse tehdä kovinkaan paljon töitä, ja päivitettävyyden määrittäminen on triviaalia. Mattsson [2000] ehdottaa ohjelmistokehyksen vakauden mittaamiseen menetelmää, joka soveltuu tähän tarkoitukseen heikosti, sillä se mittaa ohjelmistokehyksen vakautta pidemmällä aikavälillä. Tässä tapauksessa olemme kiinnostuneita vain ohjelmistokehyksen kahden version niistä eroista, jotka suoraan vaikuttavat päivitettävään ohjelmistoon. Mikäli ohjelmistokehyksistä on olemassa hyvät ja ajantasaiset rajapintadokumentaatiot, niitä on syytä hyödyntää vertailussa. Lisäksi, erityisesti mikäli dokumentaatio on puutteellista, on syytä vertailla ohjelmistokehysten toteutuksia suoraan.

Seuraavaksi kuvaan tavan, jolla PHP-ohjelmointikielellä toteutetun ohjelmistokehyksen versioiden eroja voidaan arvioida. Mattsson [2000] käyttää ohjelmistokehyksen vakauden määrittämisen yhtenä mittarina ohjelmistokehyksen muuttuneiden luokkien suhdelukua, ja seuraavalla menetelmällä voidaan laskea suhdeluku joukko-opillisten periaatteiden mukaan ohjelmistokehysten luokkien ja niiden metodien erotuksena. Kuvauksen selvyyden vuoksi nimetään ohjelmistokehyksen vanhempi, päivitettävä versio A:ksi ja uudempi, se johon ollaan päivittämässä, B:ksi.

Arviointi alkaa seuraavasti:

- Nimetään luokat, jotka löytyvät kehyksestä A, mutta eivät löydy kehyksestä B, *poistetuiksi luokiksi*.

- Nimetään luokat, jotka löytyvät kehyksestä B, mutta eivät löydy kehyksestä A, *lisätyiksi luokiksi*.

- Nimetään luokat, jotka löytyvät molemmista kehyksistä, *yhteisiksi luokiksi*.

Tehdään oletus, että *lisättyjä luokkia* ei tarvitse huomioida kehysten eroja vertaillessa. Näitä ei varmuudella ole käytetty nykyisessä sovelluksessa ainakaan niillä nimillä, joilla ne uudessa kehyksessä ovat, koska niitä ei päivitettävässä kehyksessä A ole olemassa.

Tarkennetaan arviota vertaamalla A:n ja B:n *yhteisiä luokkia* metoditasolla. Oletetaan, että mikäli kahdella samannimisellä metodilla on sama julkisuusmääre ja sama määrä parametreja, ne ovat toiminnaltaan samat. Tätä oletusta voi ohjelmointikielestä riippuen tarkentaa koskemaan myös metodin paluuarvon tyyppiä, mutta PHP:n heikon tyypityksen [Lerdorf & Tatroe, 2002] vuoksi sitä ei voi tässä hyödyntää. Lajitellaan luokat seuraavasti:

- Jos A:n luokasta on poistettu metodeita, se nimetään *muutetuksi luokaksi*.

- Jos A:n luokassa on metodeita, joiden parametreja on muutettu, se nimetään *muutetuksi luokaksi*.

- Jos A:n luokasta ei ole poistettu eikä muutettu metodeita, se nimetään *samaksi luokaksi*. Lisätyistä metodeista ei välitetä samasta syystä kuin lisätyistä luokistakaan.

Tällä tavalla saadaan lista kehyksen A:n luokista, jotka ovat joko *poistettuja*, niitä on *muutettu* tai ne *ovat samoja* uudessa kehyksessä B. Mikäli *poistettujen* ja *muutettujen* luokkien lukumäärä on suuri suhteessa kaikkien kehyksen A luokkien lukumäärään, päätellään, että ohjelmistokehysten rajapinnoilla on merkittäviä eroja, ja sovelluksen päivitettävyyttä on syytä arvioida tarkemmin. Tämän alustavan arvioinnin puutteena on se, ettei ohjelmistokehyksen julkisten rajapintojen muutoksista saada tietoa, vaan se kertoo kehyksen versioiden sisäisistä muutoksista. Tapa on kuitenkin nopea toteuttaa, ja kerättyä tietoa voidaan hyödyntää jatkoanalyysissä, jolla saadaan kohdistettua vertailua myös ohjelmistokehyksen julkisiin rajapintoihin.

4.3 Sovelluksen päivitettävän lähdekoodin määrän laskeminen

COCOMO II:n Post Architecture -mallissa tarvitaan arvio siitä, miten suuri osa lähdekoodista voidaan käyttää uudelleen [Boehm *et al.*, 1995]. Koko nykyisen sovelluksen lähdekoodin määrä on maksimimäärä uutta koodia, joka refaktoroituun toteutukseen arvioidaan tarvittavan. Tällöin uudelleenkäytettyä lähdekoodia ei olisi lainkaan. Tätä arviota pyritään tarkentamaan etsimällä minimimäärä uutta lähdekoodia, joka joudutaan toteuttamaan, ja todellinen uudelleenkäytetty lähdekoodin määrä arvioidaan sijoittuvaksi johonkin tälle välille. Lähtökohtana käytetään ohjelmistokehyksen vertailussa tulokseksi saatua luokkien erotusta.

Ensin sovelluksesta etsitään ne luokat, joissa kutsutaan ohjelmistokehyksen muutettuja tai poistettuja luokkia. Nämä tunnistettiin kehyksen alustavassa arvioinnissa. Tämä kohdistaa myös ohjelmistokehyksen versioiden vertailun implisiittisesti kehyksen julkiseen rajapintaan, kun voidaan olettaa, että kutsuttavat muuttuneet metodit ovat kuuluneet tähän rajapintaan. Absoluuttinen minimi muutetuille riveille on näiden ohjelmistokehyksen metodikutsujen määrä, mutta realistisesti voidaan olettaa, että suurin osa niistä luokista, joissa metodikutsuja on, on muutettava. Näiden luokkien rivimäärä on siis minimimäärä uudelle lähdekoodille.

Arviota voidaan laajentaa lisäämällä edellä löydettyihin luokkiin ne ohjelmiston luokat, jotka liittyvät edellä löydettyihin luokkiin jollain riippuvuussuhteella. Mukaan voidaan ottaa esimerkiksi ne luokat, jotka ovat löydettyjen luokkien aliluokkia.

4.4 Käyttöliittymien laskeminen ja työmäärien arviointi

COCOMO II:n Application Composition -malli on tarkoitettu ohjelmistojen prototyypitysvaiheeseen [Boehm *et al.*, 1995], ja sitä käytetäänkin tavallaan siihen tarkoitukseen: uuden ohjelmiston toteutuksen työmäärää arvioidaan oliopisteiden perusteella. Oliopisteiden laskemiseen käytetään kuitenkin nykyistä ohjelmistoa, eikä arvioida niiden määrää tyhjästä. Arviointimenetelmän painotukset näyttöjen ja raporttien työmääristä ja monimutkaisuudesta saattavat olla riittämättömät tai väärät WWW-pohjaisille ohjelmistoille, joten Application Composition -mallin työmääräarvio kalibroidaan tiimin aiempien suoritusten perusteella.

COCOMO II:n mallissa olioiden määrää arvioidaan, mutta ehdottamassani menetelmässä voidaan laskea ja lajitella nykyisen ohjelmiston kaikki oliot. Ne lajitellaan

kolmeen kategoriaan (helppo, keskitaso ja vaikea) sen perusteella, montako paikallista ja palvelinpään tietokantataulua niihin liittyy, ja annetaan niille painoarvo olion tyypin mukaisesti. COCOMO II:ssa painotetaan raportit työläämmiksi kuin näytöt, mutta WWW-ohjelmistoa arvioidessa voidaan jättää koko raporttityypitys pois ja lajitella oliot näytöiksi ja 3GL-komponenteiksi. Tämä sen vuoksi, että internet-sovelluksessa raporttien luominen on hyvin samankaltainen työ kuin näyttöjen ja näkymien luominen, ja käytännön kokemus on osoittanut sen olevan usein pienempitöistä kuin näyttöjen luominen. Raportit lasketaan siis yhdentyyppisiksi näkymiksi, ja kaikki oliot painotetaan vain joko näytön tai 3GL-komponentin painoarvon mukaan. Mikäli arvioitavassa sovelluksessa on asiantuntija-arvioinnin perusteella löydettävissä COCOMO II:n tarkoittamia raportteja, voidaan silti luonnollisesti käyttää raporttien painokertoimia näille olioille.

Kun näkymät ja muut oliot on laskettu ja painotettu, saadaan tulokseksi oliopisteiden lukumäärä, jota voidaan pienentää arvioimalla näyttöjen samankaltaisuuden perusteella niiden uudelleenkäytettävyyttä. Uudelleentoteuttamisen lopullinen työmääräarvio saadaan jakamalla oliopisteiden määrä tuottavuudella, joka luetaan valmiista taulukosta COCOMO II -mallissa, mutta jota kalibroidaan tiimin historiallisen suorituksen perusteella, mikäli tällaista tietoa on saatavilla.

Ohjelmiston päivittämisen työmäärä lasketaan suoraviivaisesti COCOMO II Post Architecture -mallin mukaisesti. Ohjelmiston kokona käytetään nykyisen ohjelmiston todellista lähdekoodin määrää, josta arvioidaan myös lähdekoodin uudelleenkäytön määrä.

4.5 Yhteenveto

Ohjelmiston arviointi aloitetaan määrittämällä ohjelmistokehyksen versioiden eroavaisuudet. Mikäli erot ovat merkittävät, verrataan eri ratkaisuihin käytettäviä työmääriä. Ensin lasketaan ohjelmiston päivitettävän ohjelmakoodin määrä lukemalla päivitettäväksi koodiksi kaikki luokat, jotka käyttävät ohjelmistokehyksen muuttuneita luokkia. Nykyiseen ohjelmistoon käytetty työmäärä ja päivittämisen työmäärä arvioidaan COCOMO II Post Architecture -mallin mukaisesti käyttäen päivitettävän koodin määrää lähtökohtana. Vertailun toinen puoli saadaan lajittelemalla kaikki ohjelmiston näkymät ja arvottamalla ne COCOMO II Application Composition -mallia varten. Tämän jälkeen sovelletaan kyseistä mallia ja

saadaan työmääräarvio ohjelmiston uudelleentoteuttamiselle. Vertaamalla näitä työmääräarvioita voidaan päätellä, kumpi lähestymistapa on taloudellisempi.

5 Menetelmän soveltaminen

Tässä luvussa suoritan edellisessä luvussa esittelemäni menetelmän ja arvioin PHP:lla ohjelmoidun WWW-sisällönhallintajärjestelmän päivitettävyyttä taloudellisesta näkökulmasta. Arvioitava ohjelmisto on Datapolis Solutions Oy:n toteuttama yhteisösovellus, joka on toteutettu Joomla! 1.5 -sisällönhallintajärjestelmän avulla, ja se pitäisi päivittää käyttämään uusimman Joomla-version käyttämää ohjelmistokirjastoa. Yritys on toteuttanut sivustoja Joomla-järjestelmän avulla sen ensimmäisestä versiosta lähtien, joten päivitettävään ohjelmistoon on kertynyt ominaisuuksia usean vuoden ja projektin ajalta. Esittelen ensin Joomla-sisällönhallintajärjestelmän ominaisuuksia ja käyn läpi mitä ominaisuuksia analysoimani sovellus lisää Joomla-järjestelmään. Tämän jälkeen sovellan edellisessä luvussa ehdottamaani menetelmää selvittääkseni onko kuvailemani ohjelmiston päivittäminen vai uudelleenohjelmointi taloudellisesti kannattavampaa.

5.1 Päivitettävän ohjelmiston esittely

Joomla-sisällönhallintajärjestelmä on jaettu Frontend- ja Backend-osioihin, joista Frontend on tarkoitettu julkisivuksi, joka näkyy verkkoon kaikille käyttäjille, ja Backendiin pääsevät vain ylläpitokäyttäjät. Sisällön hallinnointi on tarkoitus suorittaa Backend-puolella. Joomlan Backend-käyttöliittymät ovat kuitenkin olleet monelle asiakkaalle liian vaikeita käyttää, joten Datapolis Solutions Oy on toteuttanut Frontend-puolelle helppokäyttöisempiä hallintatyökaluja. Nämä julkisivupuolen työkalut muodostavat suuren osan päivitettävästä ohjelmistosta.

Joomla 1.5 -ohjelmistossa oli rajallinen tuki pääsynvalvontalistoille eri käyttäjäryhmien toimintojen rajaamiseksi, mutta pääsynvalvonnan puutteiden vuoksi yrityksessä kehitettiin laajennus, joka lisäsi Joomlaan tiimiperustaisen pääsynvalvonnan [Thomas, 1997]. Tiimiperustaisen pääsynvalvonnan toteutuksen myötä ylläpidolle lisättiin mahdollisuus luoda työryhmiä, joihin pääsy on rajattu määritellyille käyttäjille, ja näihin työryhmiin työkaluja, jotka hyödyntävät tätä pääsynvalvontamekanismia. Joomla 1.6:ssa julkaistiin kunnollinen tuki pääsynvalvontalistoille, joita käyttämällä suurin osa näiden työryhmien tarvitsemista pääsynvalvontaominaisuuksista voitaisiin toteuttaa. Ohjelmiston muuntaminen käyttämään pääsynvalvontalistoja tiimiperustaisen pääsynvalvonnan sijaan on yksi suurimpia syitä, miksi ohjelmiston ominaisuudet voi olla helpompaa toteuttaa uudelleen päivittämisen sijaan.

Päivitettävä ohjelmisto koostuu useista Joomla-komponenteista, moduuleista, liitännäisistä sekä kirjastokoodista, joka on toteutettu täydentämään Joomla-ohjelmistokehyksen tarjoamia ominaisuuksia. Myös osa tämän Datapolis Solutions Oy:n toteuttaman kirjastokoodin tarjoamista ominaisuuksista on toteutettu uuden Joomla-version ohjelmistokehyksessä eri tavalla.

5.2 Arvioinnin kannattavuuden määritys

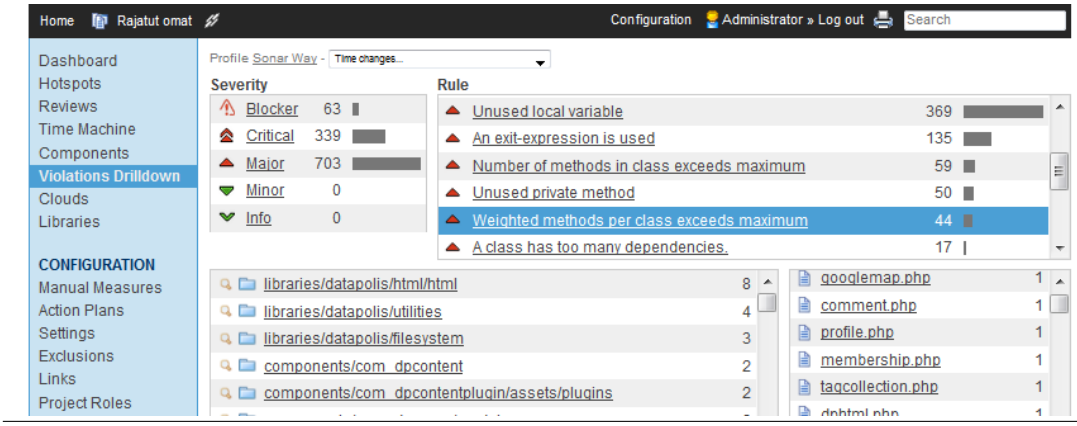
5.2.1 Ohjelmistokehysten koot

Saadakseni käsityksen ohjelmistokehysten ja päivitettävän ohjelmiston koosta suoritin Sonar-työkalulla [Sonar, 2013] ohjelmistokehysten kirjastojen lähdekoodeille analyysin, joka tuotti lähdekoodeista vertailukelpoiset metriikat. Taulukossa 6.1 luetellaan ohjelmistojen koodirivien, tiedostojen, luokkien ja metodien lukumäärät, ja niistä nähdään, että Joomla 3.0:ssa ohjelmistokehys on suuruudeltaan noin kaksi kertaa Joomla 1.5:n ohjelmistokehyksen kokoinen.

	Joomla 1.5	Joomla 3.0	Päivitettävä ohjelmisto
Koodirivejä	20894	42486	89274
Tiedostoja	176	298	1854
Luokkia	182	310	1159
Metodeja	1348	2141	6485

Taulukko 5.1: Ohjelmistokehysten koot.

Sonar-ohjelmistolla voidaan analysoida PHP-lähdekoodista muun muassa painotettu luokkien metodien määrä WMC [Chidamber & Kemerer, 1994]. Työkalu varoittaa monista koodin laatuun liittyvistä virheistä, ja esimerkiksi WMC-arvosta se kertoo, miten moni luokka ylittää raja-arvon. Lukumäärää ei kuitenkaan voi suhteuttaa mihinkään järkevästi, sillä esimerkiksi päivitettävästä ohjelmistosta löytyi 44 luokkaa, jotka ylittivät raja-arvon, kun Joomla 1.5:n kirjastokoodista niitä löytyi 31 ja Joomla 3.0:n kirjastosta 58. Uudempi ohjelmistokehys siis sisälsi vanhaan verrattuna suhteellisesti saman verran luokkia, joissa WMC-raja ylittyy: 18.7% luokista uudemmassa, 17% vanhemmassa. Kuten kuvasta 5.1 voidaan nähdä, Sonar-työkalulla saadaan mitattua monia metriikoita saamatta hyvää kuvaa siitä, mitä luvut tarkoittavat.



Kuva 5.1 Sonar-työkalun näkymä WMC-luvun raja-arvon ylittäneistä luokista.

5.2.2 PHP-lähdekoodin vertailun ongelmat

Joomlan versio 1.5 on ohjelmoitu PHP 4 -yhteensopivaksi. Tämä tarkoittaa muunmuassa erittäin rajoittunutta kapseloinnin tukea, ja se aiheuttaa luokkien vertailun suhteen muutamia ongelmia. Yritin ensin verrata ohjelmistokehyksen lähdekoodeja tekstitasolla käyttäen työkaluja DiffMerge [DiffMerge, 2013], Beyond Compare [Scooter Software, 2013] ja Atomiq [Atomiq, 2013]. Tekstitason vertailussa ongelmaksi muodostui PHP 4 ja 5 -versioiden erot, jotka näkyvät listojen 5.1 ja 5.2 lähdekoodiesimerkeissä.

```
function &getApplication($id = null, $config = array(),
    $prefix='J') {

  static $instance;

  if (!is_object($instance)) {
    jimport( 'joomla.application.application' );

    if (!$id) {
      JError::raiseError(500, 'Application Instantiation Error');
    }

    $instance = JApplication::getInstance($id, $config, $prefix);
  }

  return $instance;
}
```

Lista 5.1: PHP 4: Joomla 1.5 JFactory::getApplication() -metodi.

```
public static function getApplication($id = null, array $config =
    array(), $prefix = 'J') {
  if (!self::$application) {

    if (!$id) {
      throw new Exception('Application Instantiation Error', 500);
    }

    self::$application = JApplication::getInstance($id, $config,
        $prefix);
  }
  return self::$application;
}
```

Lista 5.2: PHP 5: Joomla 3.0 JFactory::getApplication() -metodi.

Jo Joomla 1.5 ja 3.0 -versioiden yhden metodin tarkastelulla oli selvää, ettei tekstitason vertailusta ole hyötyä erojen selvittämisessä. JFactory::getApplication() -metodin PHP 5 -toteutus listassa 5.2 on yhteensopiva vanhemman version kanssa, mutta sen lähdekoodi on muuttunut sisäiseltä logiikaltaan, eikä metodeilla ole kuin yksi rivi yhteistä lähdekoodia. Vaikka verrattaisiin pelkkää metodin nimiriviä, puuttuu PHP 4 -versiosta metodikutsun julkisuusmääreet *public static* sekä parametrin tyypitys *array*. PHP 5 -versiossa on myös poistettu *&*-merkki, joka PHP 4:ssä tarkoitti olioviitteen palauttamista. Vertailun pitäisi kuitenkin todeta nämä metodit identtisiksi rajapinnan käytön näkökulmasta.

5.2.3 PHP-lähdekoodin vertailumenetelmä

PHP-lähdekoodin vertailuun ei löytynyt ilmaista työkalua, joka ymmärtäisi yllä olevat lähdekoodit semanttisesti samoiksi. Tämän vuoksi ohjelmoin työkalun itse. Ensin yritin hyödyntää PHP 5:n Reflection-rajapintaa [The PHP Group, 2013b], joka tarjoaa rajapinnan metatiedon lukemiseksi PHP:n luokista. PHP:n sisäänrakennettu Reflection-rajapinta kuitenkin vaatii sen tutkimien luokkien lataamisen ajoympäristöön, jolloin samannimisiä luokkia ei voida vertailla. Yritin ensin toteuttaa lähdekoodin varattujen sanojen tulkitsemisen ja jäsentämisen lähdekoodista itse, kunnes löysin ongelmaan ratkaisuksi TokenReflection-luokkakirjaston [Nešpor & Hanslík, 2013], joka pyrkii tarjoamaan PHP:n sisäisen Reflection-rajapinnan ominaisuudet lukemalla lähdekoodia varattu sana kerrallaan, lataamatta läh-

dekoodeja ajoympäristöön. Tällä kirjastolla voidaan siis verrata samannimisiä PHP-luokkia toisiinsa.

Toteuttamallani työkalulla ladataan määritellyistä hakemistopuista kaikki PHP-tiedostot ja niiden sisältämien luokkien tiedot muistiin. Hakemistopuiden sisältämiä tiedostoja ja niissä määriteltyjä luokkia vertaillaan ensin pelkän luokan nimen perusteella. Vanhan rajapinnan luokannimien listasta vähennetään uuden rajapinnan luokannimien lista, jolloin saadaan rajapinnasta kokonaan poistetut luokat. Lisäksi listoista otetaan leikkaus, jolloin saadaan molemmissa rajapinnoissa olevat luokannimet. Näitä luokkia vertaillaan yksitellen toisiinsa. Listassa 5.3 kuvattu kahden PHP-luokan vertailufunktio *compareClass* ottaa parametrikseen luokat A ja B, joiden metodien nimet se kerää listoihin LA ja LB. Listojen erotus $LA \setminus LB$ on yhtä kuin A:sta poistetut metodit. Luokkien yhteisistä metodeista käydään läpi metodien kutsuparametrit samalla tavalla: luokka A:n metodin kutsuparametreista (KPA) vähennetään luokka B:n samannimisen metodin parametrit, ja mikäli erotus on tyhjä, metodi ei ole muuttunut.

```
function compareClass( $A, $B ) {
  global $muutetutTaiPoistetutLuokat;
  $LA = $LB = array();
  $classNameA = $A->getName();
  $methodsA = $A->getMethods(ReflectionMethod::IS_PUBLIC);
  $methodsB = $B->getMethods(ReflectionMethod::IS_PUBLIC);

  foreach ( $methodsA as $method ) $LA[] = $method->getName();
  foreach ( $methodsB as $method ) $LB[] = $method->getName();

  $poistetutMetodit = array_udiff($LA, $LB, 'strcasecmp');
  foreach ( $poistetutMetodit as $metodi )
     $muutetutTaiPoistetutLuokat[ $classNameA ][] = $metodi;

  $yhteisetMetodit = array_uintersect($LA, $LB, 'strcasecmp');
  foreach ( $yhteisetMetodit as $metodi ) {
    $metodiA = $A->getMethod($metodi);
    $metodiB = $B->getMethod($metodi);

    $KPA = $KPB = array();
    foreach ( $metodiA->getParameters() as $param )
      $KPA[] = $param->getName();
    foreach ( $metodiB->getParameters() as $param )
      $KPB[] = $param->getName();
    $KP_erotus = array_udiff($KPA, $KPB, 'strcasecmp');
```

```
    if ( !empty( $KP_erotus ))
      $muutetutTaiPoistetutLuokat[ $classNameA ][] = $metodi;
  }
}
```

Lista 5.3: Kahden PHP-luokan vertailufunktio.

5.2.4 Vertailun tulokset

Ohjelmistokehysten vertailun tulokset ovat esitettynä taulukossa 5.2. Joomla 1.5:n ja 3.0:n ohjelmistokehyksillä on hyvin vähän samoja luokkia. Vain neljätoista luokkaa Joomla 1.5:n 182:sta on säilynyt muuttumattomana, ja jopa 85 on kokonaan poistettu tai nimetty uudelleen. Metodikutsuja on muutettu tai poistettu 83 luokasta, ja Joomla 3.0:n kirjastossa on 213 luokkaa, joita 1.5-versiossa ei ole.

Poistettuja luokkia	85
Muutettuja luokkia	83
Samoja luokkia	182 - (85 + 83) = 14
Lisättyjä luokkia	213

Taulukko 5.2: Ohjelmistokehysten erot.

Joomla 1.5:n ohjelmistokehyksen luokista on siis muuttunut 92% 3.0-version käyttämään ohjelmistokehykseen verrattuna ja on syytä tehdä tarkempi analyysi ohjelmiston päivitettävyydestä.

5.3 COCOMO II Post Architecture -mallin soveltaminen

Aloitin Post Architecture -mallin soveltamisen antamalla arvot COCOMO II:n regressiokaavan (3.2) työmääräkertoimille ja mittakaavamuuttujille. Määrittelin prosessin kypsyyden (PMAT) SEI Capability Maturity Model -mallin prosessin avainalueiden (Key Process Areas, KPAs) kyselyarvioinnilla [Paulk *et al.*, 1995], muut muuttujat poimin COCOMO II -mallin ohjekirjan taulukkoarvoista [Boehm *et al.*, 2000]. Taulukosta 5.3 nähdään valitut mittakaavamuuttujien arvot, joissa normaalitasolta poikkeavat erityisesti tiimin yhtenäisyys (TEAM) tiimin pienen koon vuoksi sekä riskienhallinnan (RESL) suuri arvo riskisuunnitelman puuttumisen vuoksi.

Muuttuja	Arvo
PREC	2.48
FLEX	3.04
RESL	5.65
TEAM	1.10
PMAT	4.68

Taulukko 5.3: Mittakaavamuuttujien arvot.

Työmääräkertoimissa normaalitasolle jäävät puolet muuttujista. Nämä näkyvät taulukossa 5.4 arvolla 1.00, eli ne eivät muuta työmääräarviota pienemmäksi eivätkä suuremmaksi. Vaadittu luotettavuus (RELY), uudelleenkäytettävyys (RUSE), kokemus alustasta (PLEX) sekä käytettävät työkalut (TOOL) nostavat työmääräarviota, mutta vaadittava dokumentaatio (DOCU), ohjelmoijien vähäinen vaihtuvuus (PCON), kokemus vastaavasta ohjelmistosta (APEX) ja vain yhdessä paikassa työskentely (SITE) laskevat sitä.

Muuttuja	Arvo		Muuttuja	Arvo
RELY	1.10		ACAP	1.00
DATA	1.00		PCAP	1.00
CPLX	1.00		PCON	0.81
RUSE	1.15		APEX	0.88
DOCU	0.81		PLEX	1.09
TIME	1.00		LTEX	1.00
STOR	1.00		TOOL	1.17
PVOL	1.00		SITE	0.86
			SCED	1.00

Taulukko 5.4: Työmääräkertoimien arvot.

5.3.1 Nykyiseen ohjelmistoon käytetyn työmäärän arviointi

COCOMO II:n kaavassa (3.2) mittakaavan määrittävän E:n arvoksi sain $E = 0.91 + 0.01 \times (2.48 + 3.04 + 5.65 + 1.10 + 4.68) = 1.0795$ ja työmääräkerrointen tuloksi $\prod_{i=1}^{n} EM_i \approx 0.801$. Ohjelman koon määrittämisessä ensimmäinen arvio oli Sonar-ohjelmiston antama 89 KSLOC, jolla COCOMO II:n kaava antaa työmääräarvioksi $PM = 2.94 \times 89^{1.0795} \times \prod_{i=1}^{n} EM_i = 299$. Tämä arvio on kuitenkin niin suuri (24 miestyövuotta), että historiallisen datan perusteella sen on pakko

olla virheellinen. Ohjelmistoa on kyllä kehitetty noin kuusi vuotta keskimäärin neljän ohjelmistokehittäjän voimin, mutta henkilöt eivät ole olleet tekemässä vain kyseistä ohjelmistoa, joten arvion on pakko olla liian suuri.

Kalibroin mallin käyttämällä lähtökohtana ohjelmiston yhtä komponenttia, sillä pystyin erottamaan sen sisältämän lähdekoodin sekä arvioimaan sen kehitykseen käytetyn todellisen työmäärän melko tarkasti. Komponentin nimi on "Tori", ja se sisältää ohjelmakoodia $0.9KSLOC$. Arvioin sen kehitykseen kuluneen yhden miestyökuukauden ja sovelsin COCOMO II -mallin kalibrointia [Boehm *et al.*, 2000, s. 68] näillä arvoilla. Sain uudeksi kaavan muuttujan A arvoksi 1.399. Lisäksi kävin ohjelmiston lähdekoodin uudelleen läpi ja löysin ohjelmiston osia, jotka sisälsivät integroitua kolmannen osapuolen ohjelmakoodia sekä kopioituja osia, jotka eivät olleet enää lainkaan käytössä. Vähensin nämä ohjelmiston lähdekoodin määrästä, jotta saisin paremmin verrattua käytössä olevia käyttöliittymiä lähdekoodin määrään. Tämän tiivistämisen jälkeen sain Sonar-työkalulla lähdekoodin rivimääräksi 71909.

Uusilla luvuilla kaavaksi tulee $PM = 1.399 \times 71.9^{1.0795} \times 0.801 \approx 113$. Tämä on huomattavasti realistisempi arvio ohjelmistoon käytetystä kokonaistyömäärästä, sillä 9,5 miestyövuotta neljän hengen tiimillä jaettuna kuudelle kehitysvuodelle tarkoittaa, että tähän ohjelmistoon olisi käytetty noin 40% kokonaistyöajasta. COCOMO II -ohjelmistolla laskettuna ohjelma antaa 113 miestyökuukautta optimistisimpana vaihtoehtona ja tarjoaa näillä parametreilla todennäköisimmän vaihtoehdon työmääräksi 141 miestyökuukautta, mikä sekin on mahdollinen kokonaistyömäärä.

5.3.2 Refaktoroinnin työmäärän arviointi

Uudelleenkäytön määrän arvioimiseksi tein pienen ohjelman, joka etsi jokaisesta PHP-tiedostosta tiedossa olevia rajapinnan muuttuneiden luokkien nimiä. Se löysi 1033 tiedostoa, joissa on käytetty vanhan rajapinnan ominaisuuksia, ja näissä tiedostoissa oli yhteensä 36526 riviä lähdekoodia. Laskin vielä koko ohjelmiston lähdekoodin rivien määrän samalla ohjelmalla, ja sain tulokseksi 55437 riviä. Erot Sonar-työkalun antamaan tarkennettuun rivimäärään (71.9 KLOC) selittyvät rivimäärien laskentatapojen eroilla. Voin kuitenkin todeta, että noin 60% ohjelmiston lähdekoodista käyttää vanhan ohjelmistokehyksen rajapinnan luokkia.

Taulukossa 5.5 on listattuna COCOMO II:n uudelleenkäyttömallin mukaisessa arvioinnissa käytetyt arvot. Muunnettavan lähdekoodin määrän (CM) sain aiemmin

kuvaamallani lähdekoodin tutkimisella. Suoritin COCOMO II Post Architecture -mallin mukaisen ohjelmakoodin muunnostyön laskemisen ja arvioin ohjelmistokehittäjien tietämyksen nykyisestä lähdekoodista (UNFM, SU) melko matalaksi. Toisaalta arvioin, etteivät vaatimukset muutu juuri lainkaan, sillä oletuksella että täsmälleen nykyisen ohjelmiston ominaisuudet toteutettaisiin päivitettäessä. Sain muunnetun lähdekoodin määräksi 30920, jolla päivityksen työmääräksi tulee parhaimmassa tapauksessa $PM = 1.399 \times 30.9^{1.0795} \times 0.801 \approx 45$. COCOMO II -ohjelmiston tarjoama todennäköisin työmäärä on 56,7 miestyökuukautta.

Muuttuja	**Arvo**
Design Modified (DM)	10%
Code Modified (CM)	60%
Integration Modified (IM)	10%
Software Understandability (SU)	40
Assessment & Assimilation (AA)	2
Unfamiliarity with Software (UNFM)	0.8

Taulukko 5.5: Uudelleenkäytön arviointiin käytetyt arvot.

5.4 COCOMO II Application Composition -mallin soveltaminen

5.4.1 Oliopisteiden laskeminen

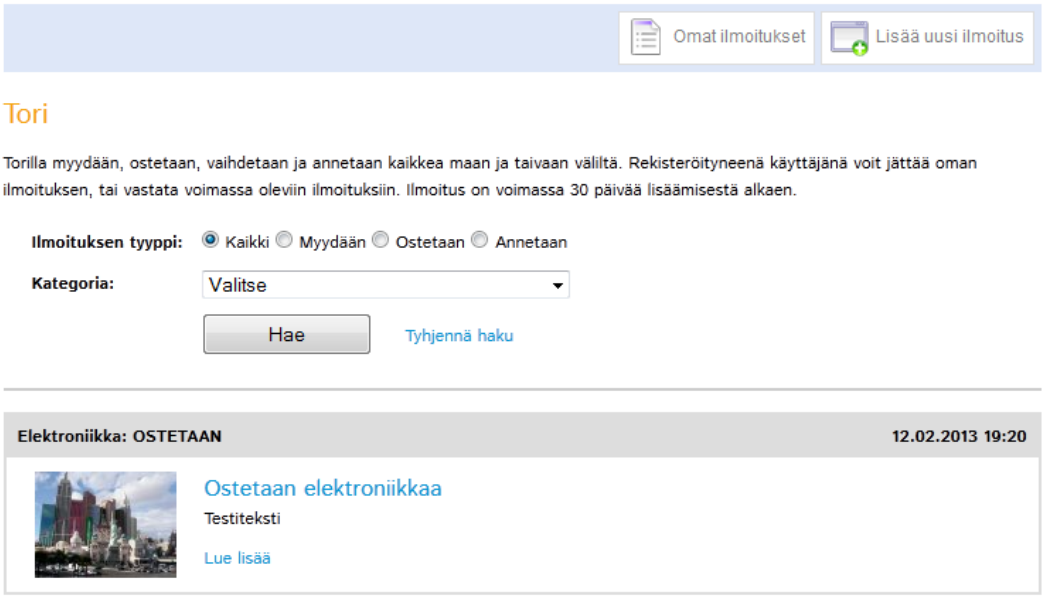
Laskemalla ohjelmiston näkymät ja lajittelemalla ne monimutkaisuuden mukaan sain taulukon 5.6 mukaiset näkymien lukumäärät. Painotin pisteet COCOMO II:n mukaisesti $56 + 36 * 2 + 11 * 3 + 10 * 3$ ja sain yhteenlasketuksi oliopisteiden määräksi 191. Laskemissani näkymissä oli muutamia samankaltaisia näkymiä, joiden perusteella arvioin uudelleenkäytettävien elementtien määrää. Arvioin työn puolittuvan näiden näkymien osalta, joten laskin uusien oliopisteiden määräksi 186.

Oliotyyppi	**Yksinkertainen**	**Keskitaso**	**Monimutkainen**
Näyttö	56	36	11
3GL-komponentti			3

Taulukko 5.6: Oliot lajiteltuna monimutkaisuuden mukaan.

5.4.2 Työmäärän arviointi

Käyttämällä COCOMO II:n taulukkoarvoja tuottavuudelle ohjelmistokehittäjän taitojen perusteella saataisiin näillä luvuilla normaalitaidoilla työmääräksi $PM = 186/13 \approx 14$. COCOMO II:n tuottavuuden taulukkoarvot on kuitenkin kehitetty WWW-ohjelmistoon verrattuna erityyppisten projektien perusteella, joten pyrin kalibroimaan tuottavuusluvun historiallisen datan perusteella. Suoritin kalibroinnin samalla komponentilla ja datalla kuin Post Architecture -mallin kalibroinnin, jotta arviot olisivat vertailukelpoisia. Arvioin ohjelmiston Tori-komponentin luomiseen vaadittavan työmäärän Joomla 1.5 -ohjelmistokehyksellä ja lisäsin työmääräarvioon 40% uuden Joomla-kehyksen tuntemattomuuden vuoksi, vastaavasti kuin COCOMO II:ssa sovelluskehittäjän kokemus ja taidot vaikuttavat arvioon. Kuvassa 5.2 nähdään esimerkki yhdestä Tori-komponentin näkymästä, jonka luokittelin monimutkaiseksi.



Kuva 5.2 Esimerkkinäkymä ohjelmiston Tori-komponentista.

Arvioin komponentin luomiseen kuluvan *1,4 henkilötyökuukautta*. Sen jälkeen laskin Tori-komponentin oliopisteet kuten aiemmin koko ohjelmiston oliopisteet, ja sain tälle komponentille luvuksi *8 oliopistettä*. Tämän kalibroinnin perusteella tuottavuus *PROD* on matalampi kuin COCOMO II:n taulukkolukema, eli $PROD = 8/1.4 = 5.7$. Korjatulla tuottavuusluvulla laskin kokonaistyömääräksi $PM = 186/5.7 \approx 32.5$.

5.5 Johtopäätökset

Riippumatta siitä, miten laskin lähdekoodin määrän tai uudelleenkäytettävän lähdekoodin määrän, arvioin käyttöliittymien perusteella ohjelmiston olevan paljon pienempi kuin lähdekoodin määrän perusteella. Koko ohjelmiston näkyvät ominaisuudet pitäisi arvion perusteella olla mahdollista toteuttaa neljäsosassa siitä työmäärästä, joka nykyiseen ohjelmistoon on todennäköisesti käytetty. Kalibroinnin tarkkuus vaikuttaa arvioihin merkittävästi, mutta saamani tulokset eroavat toisistaan niin suurella marginaalilla, että totean sen olevan merkityksellistä.

Nykyisen ohjelmiston päivittämisen työmäärä vaikuttaa myös olevan suurempi kuin uudelleentoteuttamisen. Optimistisin lähdekoodin perusteella tehty arvio ylittää käyttöliittymien perusteella tehdyn työmääräarvion puolella. En kuitenkaan pelkästään tämän perusteella totea, että lähdekoodin päivittäminen olisi ehdottomasti huono vaihtoehto, sillä muitakin syitä arvioiden eroavaisuuteen on mahdollisesti olemassa. Käyttöliittymien oliopisteitä laskettaessa joitain toimintoja on saattanut jäädä huomiotta, ja vaikka kävin lähdekoodin läpi useaan kertaan, on mahdollista, että se sisältää edelleen ohjelmakoodia, joka ei ole aktiivisessa käytössä. Mahdollisesti käyttöliittymien luokittelussa tekemäni päätelmät niiden monimutkaisuudesta ovat puutteellisia, tai näkymien osittaminen ei PHP-ohjelmoinnissa WWW-ympäristöön vastaa COCOMO II Application Composition -mallin käyttöliittymäkehityksen arvioita. Application Composition -mallin tarkkuudesta on myös aiemmin esitetty kritiikkiä [Stutzke, 2000], joka perustui arvioinnissa käytettyjen näyttöjen määrään, eli menetelmän ongelmana on juuri näyttöjen oikea laskeminen.

Mikäli käyttöliittymäarvion oletetaan olevan millään tavalla oikeaa kokoluokkaa, voidaan kuitenkin todeta, että työmääräarvioiden perusteella ohjelmisto saattaa sisältää suuria määriä lähdekoodia, joka ei ole käytössä, tai lähdekoodi on tehottomasti toteutettu verrattuna Tori-komponentin toteutukseen. Joka tapauksessa suuri osuus (60%) nykyisestä lähdekoodista käyttää ohjelmistokehyksen vanhentuneita osia. Refaktoroinnin työmäärä voi näistä syistä olla suurempi kuin uudelleen toteuttamisen.

Kustannustehokkain tapa päivittämiseen on todennäköisesti nykyisen ohjelmiston käyttäminen vain määrittelynä ja uuden sovelluksen mallina. Uusi sovellus kannattanee toteuttaa pala kerrallaan hyödyntäen vanhan ohjelmiston osia yksitellen

mahdollisuuksien mukaan, esimerkiksi kopioimalla monimutkaiset algoritmit tai selkeät kokonaisuudet sellaisenaan uuteen ympäristöön. Näiden uudelleenkäytettävien kokonaisuuksien tunnistaminen on kuitenkin tämän tutkielman aihepiirin ulkopuolella, mutta esimerkiksi Jatainin ja Gaurin [2012] arviointi komponenttien uudelleenkäytettävyydestä voi olla avuksi ohjelmiston tarkemmassa analyysissä.

Käyttämäni kalibrointimenetelmä nojautuu vahvasti asiantuntija-arvioon tuotantotiimin tuottavuudesta, ja vain yhteen ohjelmiston komponenttiin. Se on vastoin COCOMO II -mallin suosituksia, ja on mahdollista, että se aiheuttaa vääristymää tuloksissa. Käytin kuitenkin samaa työmääräarviota (Tori-komponentin työmäärä) molempien kustannusarviomenetelmien kalibrointiin, joten oletan arvioiden olevan oikeat ainakin suhteessa toisiinsa, vaikka ne eivät olisi absoluuttisesti oikeat. Näin teen johtopäätöksen, että ehdottamani menetelmä paljastaa todellisia heikkouksia lähdekoodin toteutuksesta.

6 Yhteenveto

Käsittelin tutkielmassa ohjelmistokehysten käytön etuja ja ongelmia. Halusin ratkaista yhden konkreettisen ongelman, ohjelmistokehyksen päivittämisen aiheuttaman tilanteen, jossa paras ratkaisu ohjelmiston päivittämiseksi ei ollut selvä. Ratkaisun lähtökohtana oli työmäärän arviointi, johon on olemassa ymmärrettävä, dokumentoitu ja työmäärältään kevyt arviointimalli. Tavoitteenani oli muodostaa menetelmä, jolla voidaan määrittää ohjelmistokehystä käyttävän ohjelmiston päivitettävyys käyttämällä mittarina työmäärää.

Menetelmässä vertaillaan ensin käytettävien ohjelmistokehyksen versioiden eroja, ja tätä analyysiä hyödynnetään päivitettävän lähdekoodin osuuden etsimisessä, mikäli vertailun mukaan menetelmää kannattaa jatkaa. Nykyisen ohjelmiston toteuttamiseen käytetty työmäärä arvioidaan COCOMO II Post Architecture -mallin avulla käyttämällä todellista lähdekoodin määrää laskujen perustana. Tätä verrataan ohjelmiston ominaisuuksien uudelleentoteuttamiseen kuluvaan työmäärään, joka arvioidaan COCOMO II Application Composition -mallin avulla nykyisen ohjelmiston näyttöjen perusteella. Varsinainen päivittämisen työmäärä arvioidaan myös Post Architecture -mallin avulla käyttäen päivitettävän lähdekoodin määränä sitä ohjelmiston osaa, joka käyttää ohjelmistokehyksen muuttuneita osia.

Sovelsin menetelmää PHP-ohjelmointikielellä toteutetun sisällönhallintajärjestelmän avulla toteutettuun yhteisösovellukseen, joka on tarkoitus päivittää käyttämään uudempaa ohjelmistokehystä. Toteutin oman työkalun PHP-ohjelmointikielellä tehdyn rajapinnan vertailuun, sillä tarkoitukseen sopivia ilmaisia tai kohtuuhintaisia ohjelmistoja ei ollut tarjolla, ja sain vastauksen kysymykseen, kannattaako käsittelemääni PHP-ohjelmistoa päivittää. Arviointini mukaan lähdekoodin laadussa todennäköisesti on puutteita, ja suuri osa ohjelmistosta käyttää muuttuneita rajapinnan ominaisuuksia, joten päättelen uudelleentoteuttamisen olevan pienempitöistä kuin ohjelman muuntamisen. Ehdottamani menetelmä paljasti arvioitavasta ohjelmistosta piirteitä, joita en löytänyt pelkästään käyttämällä yleisiä oliokeskeisen ohjelmistokehityksen metriikoita, joten siitä voi olla hyötyä päivitettävyyttä arvioidessa.

Menetelmän perusajatus, eli työmääräarvioinnin käyttäminen ohjelmiston laadun arvioimiseksi, vaikuttaa toimivan ainakin silloin, kun ohjelmiston laadussa

on selkeitä ongelmia. Menetelmä näyttää kuitenkin mittaavan enemmän nykyisen ohjelmiston toteutukseen käytettyä työmäärää suhteutettuna siihen, millä tavalla ohjelmiston olisi voinut toteuttaa. Tällä tavalla se mittaa epäsuorasti ohjelmiston laatua, ja ohjelmiston laatu antaa viitteitä päivittämisen työläydestä. Kaupallisessa ympäristössä yleensä kustannukset ratkaisevat ja liian kalliit menetelmät jäävät käyttämättä. Päivittämisen työmäärän tarkaksi määrittämiseksi tarvittaisiin monimutkaisempia analyysejä, joihin käytetty aika ei välttämättä ole perusteltua, mutta nopeasti suoritettava COCOMO II -arvioiden vertailu on mahdollista toteuttaa pienilläkin resursseilla. Kovin tarkkaa tulosta COCOMO II -mallin työmääräarvioilla ei saada, sillä jo optimistisimman ja todennäköisimmän työmäärän erot tekemässäni arviossa erosivat toisistaan 20%. Tekemäni analyysin perusteella työmääräarvionnin avulla voidaan kuitenkin saada viitteitä ohjelmiston laadusta, mikäli ohjelmistossa on ongelmia. Suurin hyöty menetelmästä saadaan, kun tutkittavasta ohjelmistosta ei ole olemassa kunnollista dokumentaatiota eikä sitä tunneta hyvin. Menetelmässä ei tarvita syvällistä tietämystä ohjelmistosta, sillä analyysi perustuu helposti mitattavaan lähdekoodin määrään sekä laskettavissa oleviin ohjelmiston näkyviin osiin.

On syytä miettiä, onko kuvaamaani menetelmää parempi soveltaa päivittämisen määrittämiseen, vai olisiko siitä muodostettavissa yksiselitteinen metriikka nykyisen ohjelmiston lähdekoodin laadun mittaamiseksi. Mikäli työmääräarvioinnista saadaan riittävän tarkkoja tuloksia, voidaan niiden suhdelukua mahdollisesti käyttää kuvaamaan lähdekoodin tehtäväänsä sopivuutta tai sitä, miten paljon turhaa lähdekoodia ohjelmisto sisältää. Menetelmän toimivuus on vahvasti sidottu työmääräarvion tarkkuuteen, joten kaikki työmääräarviointiin liittyvä tutkimus hyödyttää menetelmän soveltamista. Ei ole mitään syytä rajata työmääräarviointia vain COCOMO II -mallilla tehtävään arviointiin, vaan on aiheellista selvittää, antaisiko jokin muu arviointimenetelmä tarkemman arvion esimerkiksi päivitettävyyden työmäärästä. Itse menetelmää voisi kehittää laskemalla työmääräarvioille luottamusvälit ja mahdollisesti tekemällä taulukot, joista voisi tarkastaa, ovatko arvioiden erot merkityksellisiä. Myös tutkimalla arviointimenetelmiä eri kokoisilla ja eri ohjelmointikielillä toteutetuilla ohjelmistoilla saataisiin lisätietoa menetelmän tarkkuudesta.

Ohjelmiston laadun mittaamiseksi on olemassa paljon metriikoita, jotka antavat numeerisia arvoja esimerkiksi luokkien lukumäärästä, metodien koosta tai luokkien sisältämistä metodien lukumääristä. Näillä metriikoilla on helppo saada selkeitä

lukuja ohjelmistosta, mutta lukujen tulkitseminen voi olla vaikeaa. Menetelmäni heikkoutena on kustannusarviomenetelmien epätarkkuus, mutta sen vahvuutena on ymmärrettävyys. On intuitiivisesti selvää, että mikäli lähdekoodin perusteella arvioitu työmäärä ei lainkaan vastaa ohjelmiston ominaisuuksien perusteella arvioitua työmäärää, jokin on vikana joko arviointimenetelmässä tai arvioitavassa sovelluksessa.

Viiteluettelo

[Apache Software Foundation, 2013] Apache Software Foundation. Struts User Guide - A Brief history of Struts, 2013. `http://struts.apache.org/development/1.x/userGuide/introduction.html#history` [Accessed March 29, 2013].

[Apple, 2013] Apple. Cocoa - Max OS X Technology Overview, 2013. `https://developer.apple.com/technologies/mac/cocoa.html` [Accessed March 30, 2013].

[Atomiq, 2013] Atomiq. Atomiq - Code Similarity Finder, March 2013. `http://www.getatomiq.com/about` [Accessed March 26, 2013].

[Banker *et al.*, 1992] Rajiv D. Banker, Robert J. Kauffman, & Rachna Kumar. An empirical test of object-based output measurement metrics in a computer aided software engineering (CASE) environment. *Journal of Management Information Systems*, 8(3):127–150, 1992.

[Beck *et al.*, 2001] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, & Dave Thomas. The Agile Manifesto, February 2001. `http://www.agilealliance.org/the-alliance/the-agile-manifesto/` [Accessed March 30, 2013].

[Boehm *et al.*, 1995] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, & Richard Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1(1):57–94, 1995.

[Boehm *et al.*, 2000] Barry Boehm, Chris Abts, A. Winsor Brown, Brad Clark, Sunita Chulani, Ellis Horowitz, Ray Madachy, Don Reifer, & Bert Steece. COCOMO II Model Definition Manual. *The University of Southern California*, 2000. Version 2.1.

[Boehm, 1986] Barry Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, August 1986.

[Broy *et al.*, 2006] Manfred Broy, Florian Deissenboeck, & Markus Pizka. Demystifying maintainability. In *Proceedings of the 2006 International Workshop on Software Quality*, WoSQ '06, pages 21–26, New York, NY, USA, 2006. ACM.

[CakePHP, 2013] CakePHP. CakePHP: the rapid development PHP framework, 2013. `http://cakephp.org/` [Accessed April 17 2013].

[Capretz *et al.*, 2001] Luiz F Capretz, Miriam AM Capretz, & Dahai Li. Component-based software development. In *Industrial Electronics Society, 2001. IECON'01. The 27th Annual Conference of the IEEE*, volume 3, pages 1834–1837. IEEE, 2001.

[Chidamber & Kemerer, 1994] Shyam R. Chidamber & Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[Coda *et al.*, 1998] Francesco Coda, Carlo Ghezzi, Giovanni Vigna, & Franca Garzotto. Towards a software engineering approach to web site development. In *Proceedings of Ninth International Workshop on Software Specification and Design*, pages 8–17, 1998.

[Dall'Agnol *et al.*, 2003] Michela Dall'Agnol, Andrea Janes, Giancarlo Succi, & Enrico Zaninotto. Lean management — a metaphor for extreme programming? In Michele Marchesi & Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 26–32. Springer Berlin Heidelberg, 2003.

[DiffMerge, 2013] DiffMerge. DiffMerge.net, March 2013. `http://www.diffmerge.net/` [Accessed March 26, 2013].

[Dillibabu & Krishnaiah, 2005] R. Dillibabu & K. Krishnaiah. Cost estimation of a software product using COCOMO II. 2000 model–a case study. *International Journal of Project Management*, 23(4):297–307, 2005.

[Dutil *et al.*, 2010] Daniel Dutil, José Rose, Witold Suryn, & Bettina Thimot. Software quality engineering in the new ISO standard: ISO/IEC 24748 - systems and software engineering — guide for life cycle management. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, C3S2E'10, pages 89–96, New York, NY, USA, 2010. ACM.

[Fenton, 1994] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[Frakes & Terry, 1996] William Frakes & Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys*, 28(2):415–435, June 1996.

[Gamma *et al.*, 1994] Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Ginige & Murugesan, 2001] Athula Ginige & San Murugesan. Web engineering: an introduction. *MultiMedia, IEEE*, 8(1):14–18, 2001.

[Heitlager *et al.*, 2007] Iilja Heitlager, Tobias Kuipers, & Joost Visser. A practical model for measuring maintainability. In *Proceedings of 6th International Conference on Quality of Information and Communications Technology*, QUATIC 2007, pages 30–39, 2007.

[Hutchinson, 2010] James Hutchinson. The history of Joomla!: An in-depth chat with CMS core developer, Andrew Eddie, July 2010. `http://www.computerworld.com.au/article/354247/history_joomla_an_in-depth_chat_cms_core_developer_andrew_eddie/` [Accessed March 26, 2013].

[Isaacson, 2011] Walter Isaacson. *Steve Jobs*. Simon & Schuster, 2011.

[ISO, 2003] ISO. *ISO/IEC 9126:2003 Software engineering – Product quality*. International Organization for Standardization, 2003.

[ISO, 2008] ISO. *ISO 9001:2008 Quality management systems - Requirements*. International Organization for Standardization, 2008.

[ISO, 2010] ISO. *ISO/IEC 24748:2010 Systems and software engineering – Life cycle management – Guide for life cycle management*. International Organization for Standardization, 2010.

[ISO, 2011] ISO. *ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System*

and software quality models. International Organization for Standardization, 2011.

[Jatain & Gaur, 2012] Aman Jatain & Deepti Gaur. Estimation of component reusability by identifying quality attributes of component: a fuzzy approach. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, pages 738–742, New York, NY, USA, 2012. ACM.

[Johnson & Foote, 1988] Ralph E. Johnson & Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):23–35, June 1988.

[Johnson, 1997] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[Jones & SPR, 2005] Capers Jones & SPR. Programming Languages Table, 2005. `http://www.spr.com/programming-languages-table.html` [Accessed March 27, 2013].

[Joomla, 2013a] Joomla. Joomla! Framework vs. Joomla! Platform, March 2013. `http://docs.joomla.org/Joomla!_Framework_vs_Joomla!_Platform` [Accessed March 29, 2013].

[Joomla, 2013b] Joomla. Joomla Platform Manual, March 2013. `http://joomla.github.com/joomla-platform/` [Accessed March 26, 2013].

[Joomla, 2013c] Joomla. What is Joomla?, March 2013. `http://www.joomla.org/about-joomla.html` [Accessed March 26, 2013].

[Kauffman & Kumar, 1993] Robert J. Kauffman & Rachna Kumar. Modeling Estimation Expertise in Object Based ICASE Environments. In *Stern School of Business Report*. New York University, January 1993.

[Kay, 1993] Alan C. Kay. The early history of Smalltalk. *SIGPLAN Notices*, 28(3):69–95, March 1993.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, & John Irwin. *Aspect-Oriented Programming*. Springer, 1997.

[Kim *et al.*, 2011] Miryung Kim, Dongxiang Cai, & Sunghun Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE 2011, pages 151 –160, May 2011.

[Kim *et al.*, 2012] Miryung Kim, Thomas Zimmermann, & Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, Article 50, 11 pages, New York, NY, USA, 2012. ACM.

[Krasner & Pope, 1988] Glenn E. Krasner & Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August 1988.

[Kruchten, 2000] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley, 2000.

[Laing & Coleman, 2001] Victor Laing & Charles Coleman. Principal Components of Orthogonal Object-Oriented Metrics. Software Assurance Technology Center, White Paper SATC-323-08-14, NASA Goddard Space Flight Center, Greenbelt, Maryland 20771, 2001.

[Larman & Basili, 2003] Craig Larman & Victor Robert Basili. Iterative and incremental developments. A brief history. *Computer*, 36(6):47–56, 2003.

[Leitch & Stroulia, 2003] Robert Leitch & Eleni Stroulia. Understanding the economics of refactoring. In *Proceedings of the EDSER-5 5th International Workshop on Economic-Driven Software Engineering Research*, page 44, 2003.

[Lerdorf & Tatroe, 2002] Rasmus Lerdorf & Kevin Tatroe. *Programming PHP*. O'Reilly, 2002.

[Matson *et al.*, 1994] Jack E. Matson, Bruce E. Barrett, & Joseph M. Mellichamp. Software development cost estimation using function points. *IEEE Transactions on Software Engineering*, 20(4):275–287, 1994.

[Mattsson, 2000] Michael Mattsson. *Evolution and composition of object-oriented frameworks*. PhD thesis, University of Karlskrona/Ronneby. Sweden, 2000.

[McIlroy *et al.*, 1968] M. Douglas McIlroy, John M. Buxton, Peter Naur, & Brian Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch-Partenkirchen, Germany*, pages 88–98. sn, 1968.

[Microsoft, 2012] Microsoft. Visual Studio 2012, 2012. `http://www.microsoft.com/visualstudio/eng/products/visual-studio-overview` [Accessed March 28, 2013].

[Minerich, 2009] Rick Minerich. A Short history of Programming Languages, June 2009. `http://www.atalasoft.com/cs/blogs/rickm/archive/2009/06/11/a-short-history-of-programming-languages-generations.aspx` [Accessed March 26, 2013].

[Naur *et al.*, 1969] Peter Naur, Brian Randell, Friedrich L. Bauer, & NATO Science Committee. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.

[Nešpor & Hanslík, 2013] Ondřej Nešpor & Jaroslav Hanslík. PHP Token Reflection, March 2013. `https://github.com/Andrewsville/PHP-Token-Reflection` [Accessed March 26, 2013].

[Paulk *et al.*, 1995] Mark C. Paulk, Charles V. Weber, Bill Curtis, & Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, Reading, MA, 1995.

[PMHut, 2008] PMHut. Agile Estimating - Estimation Approaches, 2008. `http://www.pmhut.com/agile-estimating-%E2%80%93-estimation-approaches` [Accessed March 26, 2013].

[Randell, 1997] Brian Randell. The 1968/69 NATO Software Engineering Reports. In William Aspray, Reinhard Keil-Slawik, & David L. Parnas, editors, *The History of Software Engineering*. Citeseer, 1997.

[Royce, 1970] Winston W Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, volume 26. Los Angeles, 1970.

[Sandhu & Singh, 2006] Parvinder Singh Sandhu & Hardeep Singh. A reusability evaluation model for OO-based software components. *International Journal of Computer Science*, 1(4):259–264, 2006.

[Scooter Software, 2013] Scooter Software. Beyond Compare, March 2013. `http://www.scootersoftware.com` [Accessed March 26, 2013].

[Simonoff, 1983] Jonathan D. Simonoff. *The Lisa Applications ToolKit Reference Manual*. Apple Computer, Inc., 1983.

[Sonar, 2011] Sonar. SIG Maintainability Model Plugin, March 2011. `http://docs.codehaus.org/display/SONAR/SIG+Maintainability+Model+Plugin` [Accessed March 26, 2013].

[Sonar, 2013] Sonar. SonarSource.org, March 2013. `http://www.sonarsource.org/` [Accessed March 26, 2013].

[Stutzke, 2000] Richard D. Stutzke. Experience with the COCOMO II Application Point Model. 2000. Presented at the 15th International Forum on COCOMO and Software Cost Modeling, Los Angeles, 24-27 October 2000.

[Symfony, 2013] Symfony. Symfony: High Performance PHP Framework for Web Development, 2013. `http://www.symfony.com` [Accessed April 17 2013].

[Takeuchi & Nonaka, 1986] Hirotaka Takeuchi & Ikujiro Nonaka. The new new product development game. *Harvard Business Review*, 64(1):137–146, 1986.

[The Kunena Project, 2013] The Kunena Project. To Speak! Next Generation Forum Component for Joomla, March 2013. `http://www.kunena.org/` [Accessed March 29, 2013].

[The PHP Group, 2013a] The PHP Group. PHP Manual: History of PHP, March 2013. `http://php.net/manual/en/history.php.php` [Accessed March 26, 2013].

[The PHP Group, 2013b] The PHP Group. PHP Manual: Reflection, March 2013. `http://www.php.net/manual/en/intro.reflection.php` [Accessed March 26, 2013].

[Thomas, 1997] Roshan K. Thomas. Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, RBAC '97, pages 13–19, New York, NY, USA, 1997. ACM.

[van Vliet, 2008] Hans van Vliet. *Software Engineering: Principles and Practice*. IT Pro. John Wiley & Sons, 2008.

[VirtueMart, 2013] VirtueMart. VirtueMart, March 2013. `http://virtuemart.net/` [Accessed March 29, 2013].

[Wexelblat, 1981] Richard L. Wexelblat, editor. *History of Programming Languages I*. ACM, New York, NY, USA, 1981.

[Wikipedia, 2013] Wikipedia. Mambo software, 2013. `http://en.wikipedia.org/w/index.php?title=Mambo_(software)&oldid=546330137` [Accessed March 29, 2013].

[Wilson *et al.*, 1990] David A Wilson, Larry S Rosenstein, & Dan Shafer. *Programming with MacApp*. Addison-Wesley Reading, Massachusetts, 1990.

[Zend, 2013] Zend. Zend Framework, 2013. `http://www.zend.com/en/community/framework` [Accessed April 17 2013].