Rauno Räihä

Abstrakti luokka vai rajapinta uudelleenkäytettävyyden näkökulmasta

Tietotekniikan
kandidaatintutkielma
20. lokakuuta 2013

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Rauno Räihä
Yhteystiedot: rauno.raiha@jyu.fi
Työn nimi: Abstrakti luokka vai rajapinta uudelleenkäytettävyyden näkökulmasta
Title in English: Abstract class or interface from reusability perspective
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 24
Tiivistelmä: Tämä kandidaatintutkielma on kirjallisuuskatsaus. Tutkielman tutkimuskysymys on, tarjoaako abstrakti luokka paremman uudelleenkäytettävyyden kuin rajapinta. Tutkielmassa noudatettiin EBSE-menetelmää. Tutkielmassa havaitaan, että kuluja, joita syntyy ohjelmistokehityksessä voidaan vähentää uudelleenkäyttämällä ohjelmakoodia. Tutkielmassa saatiin tulokseksi, että uudelleenkäytettävyyttä edistävät kapselointi, matala kytkösaste, korkea koheesio ja perintä. Uudelleenkäytettävyyttä haittaa heikko ymmärrettävyys. Perintä ja koostaminen ovat keinoja, joilla koodia voidaan uudelleenkäyttää. Tutkielmassa havaittiin, että abstraktin luokan ja rajapinnan keskeisin ero on se, että abstraktilla luokalla voidaan määritellä myös toteutusta. Niinpä jos käyttökonteksti tunnetaan huonosti ja on hyvin muutosaltis on rajapinta turvallisempi vaihtoehto koska se ei rajaa uudelleenkäytettävyyttä samaan tapaan kuin abstrakti luokka. Abstrakti luokka on hyvä, jos halutaan määritellä attribuutteja tai toiminnallisuutta, jolla on toteutus.
Abstract: This bachelor's thesis is a review of the literature. The research question for this study is whether an abstract class offers better reusability than interface. The EBSE method was followed in this thesis. Expenses of software development can be reduced by reusing software code. Thesis results were that factors that promote reusability are encapsulation, loose coupling, high cohesion and inheritance. Poor undestandability weakens the possibility for reuse. Inheritance and composition are ways of reusing software code. Observation made in the thesis is that the most significant difference between abstract class and interface is that abstract class can also define implementation. Therefore if the context of use is not know very well and is likely to change then interface is more safer alternative than abstract class, because it does not delimit reusability in the same way as abstract class does. Abstract class is good when there is a need to define attributes or functionality that has implementation.
Avainsanat: rajapinnat, abstraktit luokat, uudelleenkäytettävyys, olio-ohjelmointi
Keywords: interfaces, abstract classes, reusability, object-oriented programming

Sisältö

1 Johdanto

Tässä kandidaatintutkielmassa on tutkittu abstraktien luokkien ja rajapintojen vaikutusta ohjelmiston uudelleenkäytettävyyteen. Tämän tutkielman varsinainen tutkimuskysymys on, tarjoaako abstrakti luokka paremman uudelleenkäytettävyyden kuin rajapinta. Tämän kandidaatintutkielman pääasiallinen tyyppi on kirjallisuuskatsaus, jossa keskeisiä asioita on havainnollistettu esimerkein. Eräänä keskeisenä teemana ovat koheesio ja kytköaste sekä niiden mahdolliset vaikutukset uudelleenkäytettävyyteen. Ohjelmistokehityksessä syntyviä kuluja voidaan merkittävästi vähentää uudelleenkäyttämällä ohjelmakoodia. Ohjelmakoodin uudelleenkäyttämisellä voi olla esimerkiksi seuraavia vaikutuksia. Ohjelmiston kehitykseen käytettävä aika voi lyhentyä, koska samaa asiaa ei tarvitse kehittää uudelleen. Lisäksi ohjelmakoodin luettavuus voi helpottua, koska luettavaa ohjelmakoodia on vähemmän. [9] Ohjelmistokehityksessä rajapinnat ja abstraktit luokat tarjoavat menetelmän, jolla voidaan erottaa ohjelmistoon liittyvät huolenaiheet. [5]

2 Tausta

2.1 Rajapinta

Rajapinta on ohjelmistotekniikassa hyvin monikäsitteinen, mutta tässä kandidaatintutkielmassa rajapinnalla tarkoitetaan ohjelmointikielissä määriteltyä rajapintaa. Rajapinta on kokoelma siirrettyjä toimintoja. Siirretty toiminto tarkoittaa toimintoa, jota ei ole toteutettu kyseisessä määritelmässä vaan se toteutetaan konkreetissa luokassa. Rajapinnasta ei voi luoda ilmentymiä, mutta luokka voidaan määritellä toteuttamaan rajapinta ja, jos luokka on konkreetti, siitä voidaan luoda ilmentymiä.

2.2 Abstrakti luokka

Abstrakti luokka on luokka, joka sisältää siirretyn toiminnon. Tämä tarkoittaa, että luokka sisältää ainakin yhden rajapintamäärittelyn, jota ei ole toteutettu kyseisessä abstraktissa luokassa. Abstraktista luokasta ei voida luoda ilmentymiä. Abstraktin luokan avulla voidaan määrittää yleinen käyttäytyminen joukolle luokkia, mutta jättäen kuitenkin abstraktin luokan perillisille mahdollisuus muunnella käyttäytymistä. [15, s.484–504] Eräs abstraktien luokkien käyttömahdollisuus on tehdasmetodit. Tehdasmetodi on metodi, joka palauttaa instanssin konkreetista aliluokas-

ta. Etuna tässä on se, että palautettava luokka instanssi voidaan päätellä ajon aikana vaikkapa metodille välitetyistä parametreista. [18, s.109-111] Toinen tyypillinen käyttökohde abstrakteille luokille on sovelluskehykset. [13] ”Abstraktilla luokalla voi olla myös etukäteen toteutettuja metodeja aliluokille. Abstraktit luokat muodostavat perustan laajennettaville ja uudelleenkäytettäville ohjelmistoille.” [16, suom. minun]

2.3 Perintä ja koostaminen

Yleisesti ”periä” tarkoittaa saada ominaisuuksia tai piirteitä toiselta. Perinnässä tarkoituksena on hyödyntää perivässä luokassa jo olemassa olevaa luokan määrittelyä siten, että uuteen luokkaan tarvitsee määritellä vain ne ominaisuudet, jotka eroavat entisestä luokasta. Formaalisti perintä voidaan määritellä seuraavasti:

$$R = P \oplus \Delta R$$

Tässä R on perivä luokka ja P merkitsee ominaisuuksia, jotka peritään olemassa olevasta luokasta. ΔR merkitsee inkrementaalisesti lisättyjä ominaisuuksia, jotka erottavat R:n P:stä. $\oplus$ on operaatio, joka yhdistää ΔR:n ja P:n ominaisuudet. ΔR voi myös sisältää ominaisuuksia siten, että ne kumoavat tai uudelleenmäärittelevät P:stä saatuja ominaisuuksia. [19, s.439]

Perinnässä perivä luokka erikoistaa perittyä luokkaa saaden ominaisuudet perityltä luokalta. Perintä on erikoistavan ja inkrementaalisen ohjelmistokehittämisen työkalu. Esimerkki perinnästä voidaan ajatella, että meillä olisi luokka Ajoneuvo, josta perittäisiin luokka Auto ja Moottoripyörä. Tässä siis yliluokkana Ajoneuvo ja erikoistavina luokkina Auto ja Moottoripyörä.

Koostaminen (aggregation) merkitsee periaatetta, jossa käsitellään asioita siten, että tietty kokonaisuus kootaan osista. Olio-ohjelmointikielissä tämä periaate toteutuu siten, että oliolla voi olla viite toiseen olioon tai olioihin. [19, s.442] Luonnollinen esimerkki koostamisesta voisi olla vaikkapa polkupyörä, joka ajatellaan koostuvan renkaista, polkimista ja rungosta.

Sekä perintä että koostaminen ovat uudelleenkäytön menetelmiä. Perinnän ero koostamiseen on, että aliluokan rajapinta peritään suoraan yliluokalta. Mikäli näin halutaan, tarjoaa perintä tässä etua suhteessa koostamiseen. Koostamisessa taas, jos halutaan koostetun olion rajapinta koostavalle oliolle, joudutaan kirjoittamaan delegaatiometodeja, joissa viitataan koostetun olion metodeihin. Koostaminen toisaalta tarjoaa mahdollisuuden piilottaa ne ominaisuudet, joita koostetusta oliosta ei haluta

esille toisin kuin perintä. [6] Perintä mahdollistaa perivän luokan käyttämisen vastaavissa konteksteissa, joissa perittävää luokkaa käytetään. Tämä johtuu siitä, että perivällä luokalla on vastaavanlainen rajapinta. [7]

2.4 Koheesio ja kytkösaste

Koheesio tarkoittaa luokan metodien ja attribuuttien välistä kytkeytyneisyyttä. Vahva päällekkäisyys metodien parametrien ja attribuuttien välillä indikoi voimakasta koheesiota. [4] Kytkösaste mittaa niiden olioiden lukumäärää, joita olio tarvitsee toimiakseen oikein. [4]

2.5 Ohjelmistometriikat ja QMOOD

Ohjelmistometriikoilla voidaan laskea erilaisia arvoja ohjelmiston rakenteesta. Eräs ohjelmistometriikka on koodirivien lukumäärä.

Bansiya ja Davis [4] ovat esittäneet hierarkisen mallin QMOOD oliosuuntautuneen ohjelmistosuunnitelman laadun arviointiin. Mallin tavoite on käyttää kokoelmaa ohjelmistometriikoita korkeamman tason ohjelmiston laadullisten tekijöiden arvioimiseen. Tämä tehdään siten, että ensin ohjelmistometriikat yhdistetään sopiviin ohjelmistosuunnittelun ominaisuuksiin ja näistä ominaisuuksista johdetaan vaikutukset ohjelmiston laadullisiin ominaisuuksiin. Bansiya ja Davisin QMOOD laajentaa Dromeyn [12] kehittämää laatumallia. Bansiya ja Davisin artikkelissa [4, s.7] määritellään yksitoista ohjelmistosuunnittelun ominaisuutta: Suunnitelman koko, hierarkiat, abstraktio, kapselointi, kytkösaste, koheesio, kompositio, perintä, polymorfismi, viestintä ja kompleksisuus.

Bansiya ja Davis [4] määrittelevät kuusi laadullista tekijää ohjelmistolle: uudelleenkäytettävyys, joustavuus, ymmärrettävyys, funktionaalisuus, laajennettavuus ja toimivuus.

Uudelleenkäytettävyys määritellään olevan joukko ohjelmistosuunnitelman ominaisuuksia, jotka mahdollistavat ohjelmakoodin uudelleenkäytön uuteen ongelmaan ilman suurta vaivaa.

Laajennettavuus määritellään siten, että se viittaa sellaisiin suunnittelukeskisiin ominaisuuksiin, jotka mahdollistavat uusien vaatimusten sisällyttämisen olemassa olevaan ohjelmistosuunnitelmaan. [4, s.7]

2.6 Uudelleenkäytettävyys

Uudelleenkäytettävän ohjelmiston kehittäminen tarkoittaa sellaisten komponenttien luomista, joita voidaan käyttää kaikissa oleellisissa konteksteissa mahdollisimman vähällä työpanoksella. Avainsanoja tässä ovat oleellinen konteksti sekä mahdollisimman vähäinen työpanos. Mahdollisimman vähään työpanokseen kuuluu myös työ, joka vaaditaan, kun tutkitaan käytettävissä olevat komponentit, työ joka vaaditaan tutkimiseen, että komponentti on sopiva aiottuun käyttöön, työ, joka vaaditaan komponentin varsinaiseen käyttämiseen. Vaikkakin ensimmäiset kaksi askelta ovat tärkeitä on viimeinen se kaikkein tärkein. Kolme merkittävää uudelleenkäytettävyyden ominaisuutta ovat: geneerisyys, joustavuus ja turvallisuus. Geneerisyys tarkoittaa komponentin rakennetta, joka mahdollistaa komponentin käyttäytymisen muokkaamisen. Joustavuus tarkoittaa komponentin vaatimuksien, jotka kohdistuvat kontekstiin minimointia. Turvallisuus tarkoittaa, että geneerisyys ja joustavuus eivät aiheuta, että väärää komponenttia käytetään. [8][s.2-3] Uudelleenkäytössä voidaan eritellä kaksi erilaista tapausta: komponentin uudelleenkäyttö eri konteksteissa ja eri komponenttien käyttäminen tietyssä kontekstissa. [9] Seuraavassa esimerkki saman komponentin uudelleenkäytöstä eri konteksteissa. Ajatellaan, että on olemassa komponentti, joka kuvaa autoa ja sen ominaisuuksia: nopeus, paino, väri jne. Tällaistä auto komponenttia voitaisiin käyttää esimerkiksi kuvaamaan autoa autopelissä tai autorekisterikeskuksessa, jossa pidetään yllä tietoja autoista. Seuraavaksi esimerkki eri komponenttien käytöstä samassa kontekstissa. Autopelissä voitaisiin käyttää eri komponentteja auto, kuorma-auto ja pakettiauto samassa kontekstissa autopeli.

2.7 Polymorfismi ja dynaaminen sidonta

Meyerin kirjan [15, s. 467-469] mukaan polymorfismi tarkoittaa terminä mahdollisuutta ottaa monta eri muotoa. Tämä tarkoittaa olio-ohjelmoinnissa sitä, että muuttuja, entiteetti tai tietorakenne elementti voi ottaa eri muotoja ajon aikana, joita staattinen määritys kontrolloi. Dynaaminen sidonta tarkoittaa, että dynaaminen olion muoto määrittää mikä operaatio suoritetaan. [15, s. 480] Meyer kertoo kirjassaan [15, s. 482], että polymorfismi ja dynaaminen sidonta mahdollistavat abstraktioiden käyttämisen ohjelmistojen suunnittelussa ja että voimme luottaa, että ajon aikana valitaan oikea toteutus. Lisäksi Meyer kertoo, että aina kaiken ei tarvitse olla toteutettu loppuun saakka vaan, että toteuttamattomat elementit helpottavat ongelman

analysointia ja arkkitehtuurin suunnittelua. Meyer toteaakin kirjassan, että siirretyt toiminnot ja abstraktit luokat tarjoavat tähän tarvittavan abstraktiomekanismin. Bansiya ja Davis toteavat artikkelissaan [4], että polymorfismi parantaa laajennettavuutta, joustavuutta ja toiminnallisuutta, mutta toisaalta heikentää koodin ymmärrettävyyttä. [4, s. 9-10] Seuraavassa esimerkki viitepolymorfismista ja dynaamisesta sidonnasta Java ohjelmointikielellä.

```
public class A {
        public void operoi(){
                System.out.println("A:n␣operoi");
        }
}


public class B extends A{
        public void operoi(){
                System.out.println("B:n␣operoi");
        }
}
public class MainClass {
        public static void main(String[] args) {
                A a = new A();
                B b = new B();
                //Alla olevat viitausten sijoitukset polymorfismia.
                //operoi metodin avulla demonstroidaan dynaaminen sidonta.
                a=b; //A:n viittaus viittaa B:n instanssiin.
                a.operoi(); //Nyt tulostaa konsoliin "B:n operoi"
        }

}
```

Yllä olevassa esimerkissä nähdään, että koska luokka B perii luokan A voidaan se sijoittaa viitteeseen, joka on tyypitetty luokaksi A ja kutsua operoi metodia. Koska luokassa B on määritelty metodi operoi, kutsutaan A:n operoi-toteutuksen sijaan B:n operoi toteutusta.

3 Menetelmät

Tämä kandidaatintutkielma on kirjallisuuskatsaus. Tavoite on kerätä löydetty tieto yhteen ja johtaa niistä tulokset. Tutkielmassa noudatettiin EBSE-menetelmää. [17]

Tutkielman tekeminen aloitettiin aluksi tekemällä esihakuja, joiden tuloksena saadut artikkelit luettiin. Esihaun artikkelit pohjustivat varsinaisia kirjallisuushakuja ja niitä ei taulukoitu kuten varsinaisia hakuja, jotka tehtiin myöhemmin. Esihauista valittiin seuraavat artikkelit mukaan varsinaiseen tutkielmaan: [4], [10] ja [12]. Ohjaaja Kaijanaho vinkkasi artikkelista [17], jossa kuvattua menetelmää käytettiin tutkielman teossa sekä artikkelista [19], jota käytettiin kappaleessa tausta. Esihaun jälkeen muodostettiin hakusanat, joita varsinaisessa kirjallisuus hauissa käytettiin. Kyseiset haut tehtiin ja tulokset taulukoitiin.

3.1 Hakuprosessi ja tulosten tulkitseminen

Ensimmäinen haku tehtiin 16.12.2011 ja viimeinen 5.1.2012. Ensiksi pyrittiin suorittamaan hakuja, joissa löytyisi uudelleenkäyttö ja abstraktit luokat samasta artikkelista, jotta mahdollisesti löytyisi yhteys näiden välille. Taulukot 1–4 kuvaavat hakujen tuloksia. Sarakkeessa ”otsikko” on kyseisen artikkelin otsikko ja lähdeviittauksen numero hakasulkeissa, jos on hyväksytty tutkielmaan, tai numeroviite hylättyjen artikkeleiden lähdetietoihin suluissa. Viittausten lukumäärä poimittiin Google Scholar -palvelusta, ja se kertoo, kuinka monta eri artikkelia on viitannut tähän artikkeliin. Viittausten suuri määrä viittaa siihen, että kyseinen artikkeli on tunnettu artikkeli. Sarakkeessa ”hyväksytty tutkielmaan” ilmaistaan, täyttääkö artikkeli kriteerit ja voidaanko se hyväksyä tutkielmassa käsiteltäväksi. Sarakkeessa merkintä ”kyllä” tarkoittaa, että artikkeli on täyttänyt kriteerit ja sitä voidaan käyttää tutkielmassa. Merkintä ”ei” tarkoittaa, että artikkeli ei täytä kriteereitä siten, että se hyväksytään tutkielmassa käsiteltäväksi. Perustelu kohdassa ilmaistaan syy, miksi kyseinen artikkeli on jätetty pois tai hyväksytty. Jos artikkeli täyttää sulkukriteerin, jätetään se pois vaikka se täyttäisi hyväksymiskriteerin. Sulkukriteerit (Jos kyllä suljetaan artikkeli tarkastelusta):

1. Artikkeli käsittelee muuta kuin olio-ohjelmointia
2. Artikkeli ei ole saatavissa
3. Artikkelia ei ymmärretä

Hyväksymiskriteerit (Jos kyllä hyväksytään artikkeli):

4. Artikkelissa ilmenee malli, teoria, johtopäätös tai määritelmä joka liittyy tutkimuskysymykseen

3.2 Haut

Haku 1, hakukone: scholar.google.fi, julkaisuajankohtaa ei rajattu sekä haku myös lainauksiin. Hakusana `intitle:"abstract class" AND reusability AND "object oriented"` oli ilmeinen, koska tutkimuskysymyksessä on tarkoitus tutkia abstraktin luokan uudelleenkäytettävyyttä. Haku kohdistettiin intitle hakuun, koska siten hakutulokseksi muodostui joukko artikkeleita, jotka varsinaisesti käsittelevät abstrakteja luokkia. Haun tarkoitus oli löytää tietoa uudelleenkäytöstä ja abstrakteista luokista. Haun tulokset ovat eriteltynä taulukossa 1.

Taulukko 1: haku 1 tulokset

Hakusana:intitle:"abstract class" AND reusability AND "object oriented"			
Otsikko	Viittausten lukumäärä	Hyväksytty tutkielmaan	Perustelu
The abstract class pattern [20]	18	Kyllä	Käsittelee olio-ohjelmointia. Artikkeli on saatavissa ja ymmärrettävissä. Esittää mallin abstraktien luokkien käytöstä.
Abstract class hierarchies, factories, and stable designs [18]	9	Kyllä	Käsittelee olio-ohjelmointia. Teksti on selkeää englantia. Sisältää johtopäätöksia ja mallin abstraktien luokkien käytöstä.
Intra-class testing of abstract class features [11]	2	Kyllä	Sulkemiskriteerit ei sulje pois ja sisältää johtopäätöksiä sekä määritelmiä.
SYSTEM AND METHOD FOR INSTANTIATING AN INTERFACE OR ABSTRACT CLASS IN APPLICATION CODE [5]	0	Kyllä	Sulkemiskriteerit ei sulje pois ja sisältää määritelmiä liittyen tutkimuskysymykseen.

Haussa 2 käytettiin hakukoneena scholar.google.fi. Julkaisuajankohtaa ei rajattu sekä haku myös lainauksiin. Hakusana oli `intitle:"software reusability" AND "abstract class"`. Haussa pyrittiin etsimään yhteyttä uudelleenkäytettävyyden ja abstraktien luokkien välillä. Tässä käännettiin toisin päin intitle-vaatimus hakuun 1 nähden. Haussa käytettiin intitle-kohdistusta, jotta löydettäisiin artikkeleita, jotka varsinaisesti käsittelevät ohjelmiston uudelleenkäytettävyyttä. Haun tulokset ovat eriteltynä taulukossa 2.

Taulukko 2: haku 2 tulokset

Hakusana: intitle:”software reusability” AND ”abstract class”			
Otsikko	Viittausten lukumäärä	Hyväksytty	Perustelu
Software reusability using object-oriented programming [2]	3	Kyllä	Sulkemiskriteeri ei sulje pois ja esittää määritelmiä liittyen tutkimuskysymykseen.
Software Reusability (1)	3	Ei	Varsinaista artikkelia ei saatavilla.
Reusability Assessment of Open Source Components for Software Product Lines [1]	0	Kyllä	Sulkemiskriteerit ei sulje pois. Hyväksymiskriteerit täyttyvät, koska sisältää teoriaa liittyen tutkimuskysymykseen.

Haussa 3 hakukoneena scholar.google.fi, julkaisuajankohtaa ei rajattu sekä haku myös lainauksiin. Hakusana `intitle:”reusability” AND ”interface” AND ”abstract class”`. Hakua kevennettiin hakuun 2 verrattuna otsikon osalta ja otsikossa tarvitsee esiintyä vain uudelleenkäytettävyys. Hakuun 2 erona myös, että interface sana tulee esiintyä artikkelissa. Haun tulokset ovat eriteltynä taulukossa 3.

Taulukko 3: haku 3 tulokset

Hakusana:intitle:”reusability” AND ”interface” AND ”abstract class”			
Otsikko	Viittausten lukumäärä	Hyväksytty tutkielmaan	Perustelu
Understanding the impact of language features on reusability (2)	38	Ei	Ei käsittele olio-ohjelmointi sinällään vaan ohjelmointi kielien piirteitä.
Explaining inheritance: A code reusability perspective [9]	24	Kyllä	Käsittelee uudelleenkäytettävyyttä ja perintää.
Object-Oriented Programming and Reusability [8]	7	Kyllä	Käsittelee olio-ohjelmointia ja uudelleenkäytettävyyttä.
Understanding OOP language support for reusability [7]	4	Kyllä	Käsittelee olio-ohjelmointia ja uudelleenkäytettävyyttä.
Reusability problems of object-oriented software building blocks [16]	3	Kyllä	Käsittelee sovelluskehyksien osalta uudelleenkäytettävyyttä.

Taulukko 3: haku 3 tulokset

Hakusana:intitle:”reusability” AND ”interface” AND ”abstract class”			
Otsikko	Viittausten lukumäärä	Hyväksytty tutkielmaan	Perustelu
Two-level variability analysis for business process with reusability and extensibility (3)	8	Ei	Käsittelee liiketoimintamallintamisen osalta asiaa, etäinen yhteys luokkiin ja varsinaseen olio-ohjelmointiin.
Increasing Reusability in Information Systems Development by Applying Generic Methods (4)	5	Ei	Artikkelissa ei ilmene mallia, teoriaa, määritelmää tai johtopäätöstä, joka liittyisi tutkimuskysymykseen.
Teaching programming by teaching reusability (5)	6	Ei	Käsittelee yleisesti ohjelmointia ei rajaudu olio-ohjelmointiin.
Inheritance and reusability [6]	3	Kyllä	Liittyy tutkimuskysymykseen esittämällä, että polymorfismilla on merkitystä uudelleenkäytettävyyteen.
Software Reusability	Käsitelty haussa 2. kts haku2		
Enabling Interoperability, Accessibility and Reusability of Virtual Patients across Europe–Design and Implementation (6)	3	Ei	Ei liity tutkimuskysymykseen.
Software reusability using object-oriented programming	Käsitelty haussa 2.		
Reusability Analysis of Four Standard Object-Oriented Class Libraries [3]	1	Kyllä	Käsittelee metriikoita joilla uudelleenkäytettävyyttä voidaan analysoida ja lisäksi esittää näistä sitten arvion vaikutuksista uudelleenkäytettävyyteen.
NextPDM: Improving Productivity and Enhancing the Reusability with a Customizing Framework Toolkit (7)	0	Ei	Käsittelee artikkelin kirjoittajien kehittämää sovelluskehystä sen rakennetta. Ei liity tutkimuskysymykseen.
Reusability in the Smalltalk-80 Programming System (8)	0	Ei	Ei mitään määritelmiä, teorioita tai johtopäätöstä.

Taulukko 3: haku 3 tulokset

Hakusana:intitle:”reusability” AND ”interface” AND ”abstract class”			
Otsikko	Viittausten lukumäärä	Hyväksytty tutkielmaan	Perustelu
New perspective to improve reusability in object-oriented languages [14]	19	Kyllä	Esittää mallia/teoriaa liittyen tutkimuskysymykseen.

Haussa 4 käytettiin hakukoneena scholar.google.fi. Julkaisuajankohtaa ei rajattu ja haku kohdistui myös lainauksiin. Hakusana oli ”`reusability” AND ”interface versus abstract class''` Haun tulokset ovat eriteltynä taulukossa 4.

Taulukko 4: haku 4 tulokset

Hakusana:”reusability” AND ”interface” AND ”abstract class”			
Otsikko	Viittausten lukumäärä	Hyväksytty tutkielmaan	Perustelu
Requirements analysis and system design (9)	224	Ei	Varsinaiseen kirjaan ei päästä käsiksi. Vain kalvot saatavilla.

Lisäksi haettiin Googlen Scholar -palvelulla myös käyttäen seuraavia hakusanoja ilman rajoituksia, mutta haku ei löytänyt yhtään artikkelia. Näitä hakusanoja olivat: reusability AND ”interface vs abstract class”, intitle:”interface versus abstract class” ja intitle:”interface vs abstract class”. Lisäksi kirjoitusprosessin aikana artikkeleista löydettiin lisäksi taulukossa 5 esitellyt artikkelit.

Taulukko 5: Artikkeleista löydetyt artikkelit

artikkeleista löydetyt artikkelit			
Löydetty artikkelista	Löydetty artikkeli	Hyväksytty tutkielmaan	Perustelu
[11]	[15]	Kyllä	Käsittelee olio-ohjelmointia.
[6]	[13]	Kyllä	Käsittelee olio-ohjelmointia.

Hakujen tulokset käytiin läpi tarkistaen, täyttävätkö ne hyväksymis- ja sulkemiskriteerit. Tämän jälkeen aloitettiin syntetisoimaan artikkeleiden sisältöä määritelmiin ja tuloksiin. Havaittiin, että artikkeleista kahdessa käsiteltiin uudelleenkäyt-

töä ohjelmistometriikoiden kautta. Nämä artikkelit ovat Bansiyan ja Davisin artikkeli [4] ja Amin ym. artikkeli [1] Kaksi artikkelia, [2] ja [11], hyväksyttiin tutkielmaan, mutta niihin ei viitattu, koska niissä ei ollut mitään, mihin viitata, jota ei olisi ollut toisessa artikkelissa. Eräitä artikkeleita käytettiin vain muualla kuin tuloslu-vussa. Nämä artikkelit ovat: [6], [7], [8], [9], [12], [13], [15], [18], [17] ja [19].

4 Tulokset

4.1 Metriikat, mallit ja niiden validointi

Amin ym. arvioivat avoimen lähdekoodin komponentteja artikkelissaan [1]. Amin ym. käyttävät artikkelissaan [1, s. 525-526] kytkösasteen mittaamiseen CBO-metriikkaa. CBO-metriikka lasketaan siten, että lasketaan kuinka moneen luokkaan jokin luokka on kytköksissä. Koheesion mittaamiseen Amin ym. [1, s. 525] käyttivät LCOM-metriikkaa. LCOM-metriikka laskee koheesion puutetta. Amin ym vertaavat artikkelissaan [1] esittämäänsä mallia 49 ihmisarvioijan tuloksiin. Nämä tulokset ovat taulukossa 6, joka on peräisin samasta artikkelista. [1][s.529] Ensimmäisellä sarakkeella on komponentti, johon arviointi kohdistuu. Toisella sarakkeella arvioijien komponentille annettu keskiarvo. Kolmannella sarakkeella on kyseisen komponentin luokka ja viimeisellä sarakkeella metriikoiden avulla lasketut arvot erikseen jokaiselle komponentin luokalle. Arvioitavat komponentit ovat peräisin Merobasesta(www.merobase.com).

Taulukko 6: Amin ym. mallin arviointi

Amin ym. mallin tulosten vertailu arvioijien keskiarvoihin.			
Komponentti	Arvioijien keskiarvo	Luokka	Mallin antama arvo
A	3.31	Class 1	2.89
		Class 2	2.88
		Class 3	2.68
B	3.15	Class 1	3.03
		Class 2	3.22
		Class 3	3.31
C	3.22	Class 1	3.09
		Class 2	3.24
		Class 3	2.98
		Class 4	3.07
		Class 5	3.08

Taulukko 6: Amin ym. mallin arviointi

Amin ym. mallin tulosten vertailu arvioijien keskiarvoihin.			
Komponentti	Arvioijien keskiarvo	Luokka	Mallin antama arvo
		Class 6	3.1
		Class 7	2.93
		Class 8	3.03
		Class 9	3.26

Ihmisarvioijina toimivat viimeisen vuoden tietotekniikan opiskelijat. Amin ym. mallissa [1] uudelleenkäytettävyyden osalta malli yhteni ihmisarvioijien kanssa. Toisaalta kuitenkin arvioijien joukko on melko suppea neljäkymmentäyhdeksän arvioijaa.

Bansiya ja Davis ovat kehittäneet QMOOD-mallin ohjelmakoodin laadun arviointiin. Kyseinen malli esitellään artikkelissa [4]. Bansiya ja Davis kävivät läpi artikkelissaan [4] useita ohjelmistosuunnittelukirjoja (6 kpl) ja julkaisuja (7 kpl) yhdistääkseen ohjelmiston ominaisuudet ohjelmiston laadullisiin tekijöihin mm. uudelleenkäytettävyyteen. Bansiya ja Davis käyttävät artikkelissaan [4] kytkösasteen mittarina DCC-metriikkaa (Direct Class Coupling), joka lasketaan siten, että yksittäiselle luokalle lasketaan lukumäärä eri luokista, jotka ovat siihen kytköksissä. Koheesion mittarina Bansia ja Davis käyttävät artikkelissaan [4] seuraavia kolmea metriikkaa: CAM (Cohesion among method of class), MOA (Measure of Aggregation) ja Luokan rajapinnan koko (Class Interface Size). CAM metriikka lasketaan laskemalla leikkaus luokan jokaisen metodin parametrien ja luokan attribuuttien tyyppien välillä ja summaamalla nämä arvot. MOA lasketaan laskemalla luokassa ne attribuutit, jotka ovat tyypiltään käyttäjän määrittämiä. Luokan rajapinnan koko lasketaan siten, että lasketaan luokan julkisten metodien lukumäärä.

Bansiyan ja Davisin artikkelissa [4, s. 16] mallin toimivuutta testattiin laskemalla QMOOD-mallin mukaisesti arvot ja vertaamalla niitä ihmisarvioijien antamiin arvoihin käyttäen Spearmanin rank correlation coefficient testiä. Ihmisarvioijia oli kolmetoista. Arvioijilla oli kahdesta seitsemään vuoteen kokemusta kaupallisesta ohjelmistokehityksestä. Lisäksi ymmärrys olio-paradigmasta ja olivat kehittäneet ohjelmistoja käyttäen C++ ohjelmointikieltä. Taulukon alimassa rivissä $r_s > 0.55$ merkintä $\surd$ tarkoittaa, että korrelaation on yli 0.55 ja X merkintä tarkoittaa, että korrelaatio on alle sen. Vertailu tehtiin siten, että arvioijan asettivat projektit laadun osalta järjestykseen 1. paras 14. huonoin jne. Vastaava tehtiin käyttäen QMOOD:n laskentaa ja tuloksia verrattiin ja ne näkyvät alla olevassa taulukossa.

Taulukko 7: Bansiya ja Davisin mallin vertailu [4][s. 28 Taulukko:14]

Bansiya ja Davisin mallin vertailu. QMOOD ja COOL-projektit													
	1	2	3	4	5	6	7	8	9	10	11	12	13
Σd^2	180	146	91	68	198	196	42	154	142	136	168	214	260
Σr_s	0.60	0.68	0.80	0.85	0.56	0.57	0.91	0.66	0.69	0.70	0.63	0.53	0.43
$r_s >$ 0.55	√	√	√	√	√	√	√	√	√	√	√	X	X

Taulukosta voidaan havaita, että korrelaatiota on olemassa arvioijien ja QMOOD:n välillä kun projektit on asetettu järjestykseen. Bansiya ja Davis toteavatkin, että korrelaatio on korkea ja tilastollisesti merkittävä tälle suppealle joukolle projekteja. Huomattavaa on kuitenkin, että arvioijia on vain kolmetoista.

4.2 Uudelleenkäytettävyyttä edistävät tekijät

Bansiya ja Davis mainitsevat artikkelissaan [4] uudelleenkäytettävyyttä edistäviksi tekijöiksi kapseloinnin, matalan kytkösasteen, korkean koheesion, viestinnän ja perinnän. Amin ym. [1, s. 526-529] artikkelissa kytkösaste nähdään koodin ymmärrettävyyteen vaikuttavana tekijänä. Mikäli luokalla on paljon kytköksiä on se vaikea ymmärtää ja joustavuus kärsii. Lisäksi koheesion vähyys nähdään ymmärrettävyyttä ja joustavuutta vähentävänä tekijänä. Lisäksi koodin rivien suuri lukumäärä vaikuttaa koodin ymmärrettävyyteen negatiivisesti. Joustavuus taas nähdään uudelleenkäytettävyyteen liittyvänä tekijänä kahdella eri tavalla. Ensimmäiseksi mahdollisuus käyttää komponenttia useissa eri konfiguraatioissa. Toisekseen attribuuttina, joka koskee tulevia vaatimuksia ja parannuksia.

Araban ja Sajeev toteavat artikkelissa [3, s.175], että kytkösaste ei perinnän osalta ole sovelias mittari mittamaan uudelleenkäytettävyyttä, koska ohjelmoijan ei tarvitse tietää asiakas luokan kytkösluokkien toimintaa vaan ainostaan ymmärtää asiakasluokka. Toisaalta tämän artikkelin sisältöä ei ole validoitu mitenkään vaan kyseessä on artikkelissa esitetty päätelmä, jota ei ole testattu esim. kyselemällä ihmisarvioijilta. Tällaiset päätelmät tulisi varmistaa ihmisarvioijien kautta, koska ohjelmakoodia uudelleenkäyttävät ihmiset.

Biemann ja Kang tutkivat artikkelissaan [10][s.261-262] koheesion merkitystä uudelleenkäytettävyyteen. He olettivat, että kaikista uudelleenkäytetyimmät luokat ovat korkean koheesion omaavia, mutta tutkimuksessaan saivat päinvastaisen tuloksen. Eli matalan koheesion luokat olivat eniten uudelleenkäytettyjä. Bieman ja

Kang lisäksi havaitsivat, että koheesion ja koostamisen avulla suoritetun uudelleenkäytettävyyden välillä ei ollut relaatiota, mutta koheesion ja perinnän avulla tehdyn uudelleenkäytön välillä oli. Artikkelissa tutkittiin InterViews C++ järjestelmää, jota Bieman ja Kang luonehtivat suhteellisen suureksi C++ järjestelmäksi. Tämä järjestelmä oli kehitetty Stanfordin yliopistossa ja järjestelmän tarkoitus oli määrittää luokkia käyttöliittymää varten. Järjestelmä koostui 25,000 rivistä koodia pois lukien kommentit.

Uudelleenkäytettävyyden eräs kulmakivi on ohjelmistovaatimusten eriyttäminen siten, että ne ovat mahdollisimman itsenäisiä toisiinsa nähden sekä vähäisesti riippuvia tulevista käytön konteksteista. Jos ohjelmistovaatimus on levittynyt ohjelmiston useille eri alueille tai vaatimusten määrä kehittyy jatkuvasti saattaa, olio-ohjelmoinnin uudelleenkäytön menetelmät olla epäsopivia. [14][s.1-2]

4.3 Uudelleenkäytettävyyden ongelmat

Wolfgang Pree tunnistaa artikkelissaan [16] uudelleenkäytön ongelmiksi seuraavat. Tehoton uudelleenkäyttö, joka johtuu siitä, että on epäselvää kuinka tietty toiminnallisuus lisätään olemassa oleviin komponentteihin. Epäselvyys johtuu taas komponentin kompleksisuudesta. Jotta voidaan uudelleenkäyttää täytyy tuntea olemassa olevien luokkien tarjoamat ominaisuudet ja joskus jopa miten yksittäisten luokkien käyttäytyminen on toteutettu. Vastaavasti Bansiya ja Davis toteavat myös artikkelissaan [4, s. 9-10], että heikko ymmärrettävyys heikentää uudelleenkäytettävyyttä. Barcian ym. artikkelin [5][s.1] mukaan teknisesti konkreetti luokka voi toteuttaa useita rajapintoja, toteuttamalla niiden abstrakteille metodeille toteutuksen, mutta periä vain yhden luokan kun kyseessä on yksinperintäinen olio-ohjelmointikieli. Vastaavasti myös Bobby Woolf kertoo artikkelissaan [20, s. 2-5], että abstraktin luokan käyttäminen pakottaa kaikki sen aliluokat olemaan samassa luokkahierarkiassa. Lisäksi abstraktin luokan siirretyt toiminnot on toteutettava konkreeteissa aliluokissa, joten jos on määritelty toimintoja joita ei voida toteuttaa aliluokassa tulee ongelmia.

Siten siis jos jokin konkreetti luokka perii abstraktin luokan toisen eriävän luokkahierarkian luokan periytyminen estyy. Esimerkiksi jos meillä on luokat: A1, A2 ja B1 ja B2. Luokka A2 perii luokan A1. Luokka B2 perii luokan B1. Nyt luokka A2 ei voi periä luokkaa B1 tai B2, koska se perii jo luokan A1. Siten eriävän luokkahierarkian periminen estyy.

5 Pohdinta

5.1 Esimerkki perinnän estymisestä

Alla esimerkki abstraktista luokasta ja abstraktin luokan perivästä luokasta. Perivä luokka toteuttaa abstraktin metodin.

```
public abstract class Abstract_A {

        public abstract void methodA();

}


public class Inheriting_B extends Abstract_A {

        @Override
        public void methodA() {
                // TODO Auto-generated method stub

        }

}
```

Alla java esimerkki rajapinnasta ja rajapinnan toteuttavasta luokasta.

```
public interface Interface_A {
        public void methodA();

}


public class Implementing_B implements Interface_A {

        @Override
        public void methodA() {
                // TODO Auto-generated method stub

        }

}
```

Voidaan havaita, että rajapintaa käyttämällä voidaan toteuttaa samantapainen konstruktio kuin abstraktilla luokalla. Molemmissa esimerkeissä metodi methodA() edel-

lytetään toteutettavaksi konkreetissa aliluokassa tai rajapinnan toteuttavassa luokassa. Kielissä joissa ei tueta moniperintää voisi rajapintatoteutuksessa vielä periä jonkun toisen luokan. Molemmissa tapauksissa Implementing_B ja Inheriting_B luokkia voitaisiin uudelleenkäyttää koostamisen avulla. Abstraktin luokan tapauksessa ei voi periä muita luokkia jos kieli ei ole moniperintää tukeva.

5.2 Esimerkki abstraktin luokan määrittelykyvystä

Abstrakti luokka voi sisältää metodeja joilla on toteutus sekä attribuutteja toisin kuin rajapinta. Siksi abstraktit luokat mahdollistavat konstruktioita joita ei voida luoda rajapinnoilla kuten esimerkiksi alla oleva javalla toteutettu konstruktio:

```
public abstract class Abstract_Class {
        private String doSomeOperationText = "doSomeOperation␣finished";

        public abstract void methodA();

        public void doSomeOperation(){
                methodA(); //Osittainen toteutus konkreetissa aliluokassa.
                System.out.println(doSomeOperationText); //Toinen osa toteutuksesta
        }

}
```

Esimerkissä siis abstrakti yliluokka voi kutsua konkreetin aliluokkansa methodA:ta ja suorittaa sillä osan doSomeOperation-metodin toteutuksesta ja osan toteutuksesta määritellä itse. Abstraktin luokan määrittelykyky on suurempi kuin rajapinnan sillä rajapinta on osajoukko abstraktista luokasta. Rajapinta on parempi vaihtoehto jos ei tarvita attribuutteja tai toteutettuja metodeja, koska rajapinta ei estä perintää kielissä, jotka eivät tue moniperintää.

5.3 Abstrakti luokka ja uudelleenkäytettävyys

Koheesion osalta kuten koheesion määritelmästä ja metriikoista, joilla sitä lasketaan voidaan havaita, että abstrakti luokka kykenee vaikuttamaan koheesioon kasvavasti, koska se voi määritellä attribuutteja ja toteutusta. Kytkösasteen osalta abstrakti luokka voi määritellä kytköksiä toisiin luokkiin ja siten aiheuttaa kytkösasteen kasvua. Toisaalta voi vähentää suoria kytköksiä toisiin luokkiin määrittelemällä siirrettyjä toimintoja. Perintä on yksi osa uudelleenkäytettävyyttä, jos peritään abstrak-

ti luokka toisen luokkahierarkian periminen estyy sellaisissa kielissä, jotka eivät tue moniperintää Koska abstrakti luokka voi määritellä enemmän kuin rajapinta se myös rajaa enemmän kuin rajapinta. Siten abstrakti luokka sopii tilanteisiin, joissa käyttökonteksti tunnetaan hyvin ja halutaan määritellä rajapinnan lisäksi myös attribuutteja tai toteutettua toiminnallisuutta. Yleensä sovelluskehyksissä käyttökonteksti tunnetaan hyvin ja niissä käytetäänkin usein kantaluokkina abstrakteja luokkia. Esimerkiksi ASP.NET-sovelluskehyksessä kontrolleri luokkien kantaluokkana on ControllerBase luokka.

5.4 Rajapinta ja uudelleenkäytettävyys

Koheesion osalta kuten koheesion määritelmästä ja metriikoista, joilla sitä lasketaan voidaan havaita, että rajapinta ei voi suoraan vaikuttaa koheesioon, paitsi luokan rajapinnan koon kautta. Kytkösasteen osalta rajapinta ei voi kasvattaa kytkösastetta, koska se ei voi määritellä suoria kytköksiä toisiin luokkiin. Rajapinta määrittelee siirrettyjä toimintoja ja siten se vähentää suoria kytköksiä. Perinnän osalta rajapinta ei estä perimästä toisia luokkia, vaan luokka voi määritellä useita rajapintoja ja lisäksi periä jonkun luokan. Rajapinta ei juurikaan aiheuta milloinkaan ongelmia uudelleenkäytettävyyden osalta.

6 Yhteenveto

Tässä kandidaatintutkielmassa tehtiin kirjallisuuskatsaus, jossa etsittiin vastausta kysymykseen tarjoaako rajapinta paremman uudelleenkäytettävyyden kuin abstrakti luokka vai onko asia juurikin päinvastoin. Menetelmänä käytettiin EBSE-menetelmää.

Kirjallisuus tunnistaa, että matala kytkösaste ja korkea koheesio olisivat suotuisia tekijöitä uudelleenkäytettävyyden osalta. Kytkösasteen osalta rajapinta ja abstrakti luokka eroavat siten, että abstraktilla luokalla on mahdollista luoda suoria kytköksiä toisin kuin rajapinnalla ja siten kasvattaa kytkösastetta. Rajapinta ei suoranaisesti vaikuta koheesion, mutta abstrakti luokka voi kasvattaa koheesiota. Rajapinta ei aiheuta ongelmia uudelleenkäytettävyyden osalta toisin kuin abstrakti luokka, joka rajaa perintähierarkiaa. Siten siis mikäli tunnetaan käyttökonteksti huonosti on rajapinta parempi vaihtoehto, mutta jos käyttökonteksti tunnetaan hyvin voi abstrakti luokka olla hyödyllisempi kun halutaan määritellä attribuutteja ja/tai toteutettuja toimintoja.

Tulokset joita tässä tutkielmassa on esitetty ovat hyvin suuntaa antavia ja niihin tulee suhtautua varauksella. Varsinaiseen tutkimuskysymykseen ei yksikäsitteistä vastausta löydetty, koska uudelleekäytettävyys on melko monitahoinen käsite ja on riippuvainen ohjelmistokehittäjistä ja siitä miten hyvin tunnetaan alue, johon ohjelmistokehitys kohdistuu.

Lähteet

[1] Fazal e. Amin, Ahmad Kamil. Mahmood, ja Alan. Oxley. Reusability assessment of open source components for software product lines. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, 1(3):519 –533, 2011.

[2] D.B. Anderson ja S. Gossain. Software reusability using object-oriented programming. Kirjassa *UK IT 1990 Conference*, ss. 299 –305, mar 1990.

[3] Saeed Araban ja A. Sajeev. Reusability analysis of four standard object-oriented class libraries. Kirjassa Walter Dosch, Roger Lee, ja Chisu Wu, toim., *Software Engineering Research and Applications*, sarjan *Lecture Notes in Computer Science* osa 3647, ss. 171–186. Springer Berlin / Heidelberg.

[4] J. Bansiya ja C.G. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4 –17, jan 2002.

[5] R. Barcia, K.S. Bhogal, G.M. Hambrick, ja R.R. Peterson. System and method for instantiating an interface or abstract class in application code, lokakuu 23 2006. US Patent App. 20,080/127,070.

[6] R.L. Biddle ja E.D. Tempero. Inheritance and reusability. Kirjassa *Software Engineering Conference, 1998. Proceedings. 1998 Australian*, ss. 184 –191, nov 1998.

[7] Robert Biddle ja Evan Tempero. *Understanding OOP language support for reusability*. Department of Computer Science, Victoria University of Wellington, 1995.

[8] Robert Biddle, Evan Tempero, ja Peter Andreae. Object-oriented programming and reusability. Technical report, Victoria University of Wellington, July 1995.

[9] Robert Biddle ja Ewan Tempero. Explaining inheritance: a code reusability perspective. Kirjassa *Proceedings of the twenty-seventh SIGCSE technical sympo-*

sium on Computer science education, sarjassa *SIGCSE '96*, SIGCSE '96, ss. 217–221, New York, NY, USA, 1996. ACM.

[10] James M. Bieman ja Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. Kirjassa *Proceedings of the 1995 Symposium on Software reusability*, sarjassa *SSR '95*, SSR '95, ss. 259–262, New York, NY, USA, 1995. ACM.

[11] P.J. Clarke, D. Babich, T.M. King, ja J.F. Power. Intra-class testing of abstract class features. Kirjassa *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, ss. 191–200. IEEE, 2007.

[12] R.G. Dromey. A model for software product quality. *Software Engineering, IEEE Transactions on*, 21(2):146 –162, feb 1995.

[13] Ralph E. Johnson ja Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, June/July 1988.

[14] Philippe Lahire ja Laurent Quintian. New perspective to improve reusability in object-oriented languages. *Journal of Object Technology*, 5(1):117–138, tammikuu 2006.

[15] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall PTR, One Lake Street, Upper Saddle River, New Jersey, 07458, 1997.

[16] Wolfgang Pree. Reusability problems of object-oriented software building blocks. *EastEurOOPe*, 1991.

[17] Austen Rainer ja Sarah Beecham. Supplementary guidelines, assessment scheme and evidence-based evaluations of the use of evidence based software engineering. Technical report, University of Hertfordshire, February 2008.

[18] Friedrich Steimann. Abstract class hierarchies, factories, and stable designs. *Commun. ACM*, 43:109–111, April 2000.

[19] Antero Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28:438–479, September 1996.

[20] B. Woolf. The abstract class pattern. *Pattern Languages of Program Design*, 4, 1997.

A Hylättyjen artikkeleiden lähdetiedot

1. Software Reusability, Kim, Soo Dong "Software Reusability," Wiley Encyclopedia of Computer Science and Engineering, vol. 4, Jan. 2009

2. Understanding the impact of language features on reusability, Biddle, R.L. ja Tempero, E.D. Software Reuse, Proceedings Fourth International Conference on, april, 1996, 10.1109/ICSR.1996.496113

3. Two-Level Variability Analysis for Business Process with Reusability and Extensibility, Moo, Mikyeong ja Hong, Minwoo ja Yeom Keunhyuk, Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International, 28 august 2008

4. Increasing Reusability in Information Systems Development by Applying Generic Methods, Eckstein, Silke ja Ahlbrecht, Peter ja Neumann, Karl, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 978-3-540-42215-0

5. Teaching programming by teaching reusability, Biddle, Robert ja Tempero,Evan, Technical Report CS-TR-96/2, Jan. 1996

6. Enabling Interoperability, Accessibility and Reusability of Virtual Patients across Europe – Design and Implementation, Zary, Nabil ja Hege, Inga ja Heid, Jörn ja Woodham, Luke ja Donkers, Jeroen ja Kononowicz Andrzej A., Medical Informatics in a United and Healthy Europe, IOS Press 2009, doi:10.3233/978-1-60750-044-5-826

7. NextPDM: Improving Productivity and Enhancing the Reusability with a Customizing Framework Toolkit, Hwang, Ha ja Kim, Soung Computational Science and Its Applications – ICCSA 2004, Springer Berlin / Heidelberg, 978-3-540-22054-1

8. Reusability in the Smalltalk-80 Programming System, Deutsch, Peter L., Xerox PARC, Software Concepts Group.

9. Requirements Analysis and System Design 3rd ed., Addison Wesley, Harlow England, ISBN 978-0-321-44036-5