Opinnäytetyö (AMK)

Tietojenkäsittely

Tietojärjestelmät

2011

Antti Suutarinen

OHJELMAKIRJASTOT JA PUURAKENTEITA KÄYTTÄVIIN ALGORITMEIHIN PERUSTUVA ESIMERKKIKIRJASTO

TURUN AMMATTIKORKEAKOULU

TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

Turun ammattikorkeakoulu

Tietojenkäsittely | Tietojärjestelmät

Helmikuu 2011 | 63

Tuomo Helo

Antti Suutarinen

OHJELMAKIRJASTOT JA PUURAKENTEITA KÄYTTÄVIIN ALGORITMEIHIN PERUSTUVA ESIMERKKIKIRJASTO

Työssä pyritään kuvaamaan ohjelmakirjaston luomisen perusteita, ja tavoitteena on luoda pienimuotoinen tietopankki niiden koostamisesta. Työssä kuvataan myös kaksi esimerkkiä luokkakirjastoista eli C++ Standard Template Library sekä .NET Framework Class Library. Työhön on sisällytetty myös puurakenteita, joihin esimerkkikirjaston algoritmit perustuvat. Tavoitteena on toteuttaa esimerkkikirjasto Javalla, sisältäen binäärisiä puurakenteita käyttäviä algoritmeja. Ohjelmakirjaston luomista kuvataan ns. vesiputousmallin avulla eli määrittely-, suunnittelu-, toteutus- ja integrointi ja testaus –vaiheilla.

Esimerkkialgoritmien toteutukseen valittiin ohjelmointikieleksi Java, koska se on olio-ohjelmointikieli ja koska se on laajassa käytössä esimerkiksi opetuksessa. Esimerkkikirjasto voitiin rajata Javan package-määreellä. Algoritmeihin otettiin mallia kirjallisuudessa olevista pseudokooditoteutuksista, ja niiden perusteella luotiin Java-versiot.

Tuloksena saatiin tietoa ohjelmakirjastojen uudelleenkäytettävyydestä, niiden hyvistä ominaisuuksista, modulaarisuudesta, komponenteista, puurakenteista, ohjelmakirjaston koostamisesta ja algoritmeista. Esimerkkikirjaston algoritmit on kommentoitu mahdollisimman hyvin, ja niitä voivat työn lukijat muokata halutessaan omaan käyttöönsä.

Ohjelmakirjastot ovat olleet olemassa kauan ja niitä tarvitaan edelleen. Tämän työn perusteella voisi sanoa, että niiden luominen ei ole vaikeaa, riippuen tietenkin rakennettavan kirjaston koosta. Puurakenteita käsittelevien algoritmien ei tarvitse olla kovin monimutkaisia, silti niiden avulla voidaan analysoida ja käsitellä puurakenteisia tietorakenteita melko tehokkaasti. Olio-ohjelmointiominaisuudet auttavat tässäkin esimerkkialgoritmikokoelmassa. Niiden tiedonkapseloimisominaisuudet, periyttämismekanismit sekä luokkien käyttö yleensä tuovat luotettavuutta algoritmien toimintaan.

ASIASANAT:

Ohjelmakirjasto, luokkakirjasto, komponentti, moduuli, modulaarisuus, puurakenne, algoritmi, vesiputousmalli

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology | Information Systems

February 2011 | 63

Tuomo Helo

Antti Suutarinen

PROGRAM LIBRARIES AND A SAMPLE LIBRARY BASED ON ALGORITHMS USING TREE-STRUCTURES

In this study, the aim is to describe the basis of creating a software library in order to create a small information database on how to compile libraries. The study describes two examples of class libraries, that is, C++ Standard Template Library and .NET Framework Class Library. The study also includes tree structures, on which the algorithms of the example library are based. The objective is to implement the example library with Java, including algorithms using binary tree-structures. The construction of a software library is described with the so-called waterfall model, in other words, using the following phases in sequential order: defining, designing, implementation, integration and testing.

The implementation language of the example library was chosen to be Java, based on its nature as an object-oriented language and because it is widely used in education, for example. The example library was constrained with the package-definition. The algorithms were modeled on pseudo-code implementations in literature, based on which Java-versions were created.

This study generated knowledge on the reusability of software libraries, their good qualities, modularity, components, tree-structures, construction of a software library and algorithms. The algorithms of the example library were commented on as well as possible, and the readers of this study can use the algorithms for their own purposes if they wish.

Software libraries have existed for a long time and are still needed. Based on this study, it is possible to say that creating them is not difficult, depending on the size of the library being constructed, of course. Algorithms using tree-structures need not be very complex, yet it is possible to analyze and handle tree-structured information structures quite effectively. Object-oriented technologies help also in this collection of example algorithms. Their information encapsulation and inheritance mechanisms, and using classes in general, bring reliability to the functioning of algorithms.

KEYWORDS:

Program library, class library, component, module, modularity, tree structure, algorithm, waterfall model

SISÄLTÖ

KÄYTETYT LYHENTEET **6**

1 JOHDANTO **6**

2 OHJELMAKIRJASTOT **7**

2.1 Ohjelmakirjaston alkuperä ja määritelmä 7

2.2 Luokkakirjastoja 8

2.2.1C++ Standard Template Library 8

2.2.2.NET Framework Class Library 9

3 OHJELMAKIRJASTON OMINAISUUKSIA **11**

3.1 Skaalautuvuus 11

3.2 Siirrettävyys 11

3.3 Tehokkuus 12

3.4 Uudelleenkäytettävyys 12

3.4.1 Perusteita 12

3.4.2Mustan laatikon uudelleenkäytettävyys 13

3.4.3Valkoisen laatikon uudelleenkäytettävyys 13

3.4.4Horisontaalinen uudelleenkäyttö 14

3.4.5Vertikaalinen uudelleenkäyttö 14

3.4.6Uudelleenkäytettävyyden esteitä 15

3.5 Staattiset ja jaetut kirjastot 17

3.5.1Staattisten kirjastojen määritelmä 17

3.5.2Jaettujen kirjastojen määritelmä 18

3.6 Modulaarisuus 20

3.6.1Modulaarisuuden etuja 20

3.6.2Komponentit yleisesti 21

3.6.3Komponenttien käytön etuja 21

3.6.4Komponenttien uudelleenkäytön ongelmia 24

4 PUURAKENTEITA **28**

4.1 Binääripuu 29

4.2 Keko 29

4.3 Järjestetty binääripuu 30

5 OHJELMAKIRJASTON KOOSTAMINEN **32**

5.1 Määrittely 32

5.2 Suunnittelu 32

5.3 Toteutus 33
5.4 Integrointi ja testaus 34
6 ESIMERKKIKIRJASTO 36
6.1 Algoritmien ominaisuuksista 36
6.2 Puun syvyyden määrittäminen 38
6.3 Puun läpikäynti 38
6.4 Puun solmun edeltäjän etsiminen 40
6.5 Järjestetyn binääripuun minimi- ja maksimiarvot 41
6.6 Heapsort eli kekolajittelu 42
6.7 Muut algoritmit 44
7 JOHTOPÄÄTÖKSET 45
LÄHTEET 46

LIITTEET

Liite 1. Ohjelmalistaukset

KUVAT

Kuva 1 Yksinkertainen binääripuu 28
Kuva 2 Keko 30
Kuva 3 Binäärihakupuita 30
Kuva 4 Binääripuu, jolla syvyys 3 38
Kuva 5 Läpikäytävä puu 39
Kuva 6 Demonstraatiossa käytetty puu 39
Kuva 7 Solmun edeltäjän etsiminen 40
Kuva 8 Javadoc-dokumentaatiota 41
Kuva 9 Järjestetyn binääripuun ääriarvojen etsiminen 41
Kuva 10 Heapify-metodin toiminta 42
Kuva 11 Esimerkki BuildHeap-metodin toiminnasta 43

KÄYTETYT LYHENTEET

COM	Component Object Model, Microsoftin avoin olio- ja komponenttimalli (FOLDOC 1999)
COTS	Commercial Off The Shelf, suoraan hyllystä eli valmiina käyttöönotettu kaupallinen sovellus (FOLDOC 2007)
DLL	Dynamically linked library, kirjasto joka linkitetään sovellusohjelmaan latauksen aikana (FOLDOC 2000)
MVC	Model-View-Controller eli Malli-Näkymä-Ohjain, tapa osittaa vuorovaikutteinen ohjelmisto (FOLDOC 2007)

1 JOHDANTO

Olen ollut kiinnostunut ohjelmoinnista. Pidän Javasta ohjelmointikielenä, joten ohjelmointiaihe tuntui luontevalta. Työ tehtiin tutkimustyöluonteisena. Tietoni puurakenteista perustuvat paljolti oppimäärän mukaisesti suoritettuun kurssiin tietorakenteista ja algoritmeista. Vaikka algoritmien toteuttaminen ohjelmallisesti onkin haastavaa, koska toteutuksen on toimittava täydellisen virheettömästi, eikä vain sinnepäin, pidin työn vaikeusastetta sopivana.

Työn tavoitteena on luoda tietopaketti ohjelmakirjastoista, niiden hyvistä ominaisuuksista kuten uudelleenkäytettävyydestä, linkittämisestä, modulaarisuudesta sekä staattisista ja jaetuista kirjastoista. Työ kuvaa ohjelmakirjaston luomista vesiputousmallin avulla. Työssä esitellään joitakin myöhemmin käytettäviä puurakenteita. Empiirisessä osassa toteutetaan esimerkkikirjasto, jonka sisältämistä algoritmeista osa kuvataan lähemmin ja osa sisällytetään vain liitteeksi.

Algoritmien rajaaminen puurakenteisiin ja vielä niiden lisäksi binäärisiin puihin tuntui toimivan rajauksena hyvin, aihetta oli mahdollista tutkia ja siitä löytyi kirjallisuutta. Toteutin algoritmit Javalla siten, että etsin lähteistä pseudokoodina toteutettuja algoritmeja ja käänsin ne siitä Javaan. Luvun ohjelmakirjastojen luomisesta vesiputousmallin avulla kirjoitin itsenäisesti. Ohjaajani kanssa käymien keskustelujen perusteella päätin ottaa sen mukaan eikä ko. aiheesta löytynyt paljoakaan lähdemateriaalia.

Ohjelmakirjastojen kehitystä olisi ollut mielenkiintoista tutkia myös tietokonehistoriallisesta näkökulmasta, mutta siihen ei ollut varsinaista mahdollisuutta tämän työn puitteissa. Toivottavasti työ kuitenkin tarjoaa perustietoa ohjelmakirjastoista tietoa hakevalle, ja tarjoaa esimerkin, miten kyseisen tyyppisiä algoritmeja voi toteuttaa.

2 OHJELMAKIRJASTOT

2.1 Ohjelmakirjaston alkuperä ja määritelmä

Ohjelmakirjasto on kokoelma ohjelmia ja paketteja jotka ovat tehtyjä saataville yleiseen käyttöön jossakin ympäristössä; yksittäisten kirjaston osien ei tarvitse olla riippuvaisia toisistaan. Tyypillinen kirjasto saattaa sisältää muun muassa apuohjelmia ja matemaattisia toimintoja sisältäviä paketteja. Yleensä on tarpeellista vain viitata kirjastoon, jotta se tulee automaattisesti sisällytettyä käyttäjän ohjelmaan. (Encyclopedia.com 2004.)

Plaugerin ym. mukaan *kirjasto* on kokoelma ohjelmakomponentteja, joita voidaan käyttää monissa ohjelmissa (Plauger ym. 2001, 1).

FOLDOCin mukaan ohjelmakirjasto on kokoelma aliohjelmia ja funktioita, joita säilytetään yhdessä tai useammassa tiedostossa. Tavallisesti niitä säilytetään käännetyssä muodossa, jotta niitä voidaan linkittää muiden ohjelmien kanssa. Ohjelmakirjastot ovat yksi ensimmäisistä organisoidun koodin uudelleenkäytön muodoista. Ne tulevat usein käyttöjärjestelmän tai ohjelmankehitysympäristön mukana käytettäviksi monien eri ohjelmien kanssa. Kirjaston rutiinit voivat olla yleiskäyttöisiä tai suunniteltuja jotakin tiettyä toiminnallisuutta, kuten esimerkiksi kolmiulotteista animoitua grafiikkaa varten. (FOLDOC 1998.)

Levine (1999, 169) kertoo ohjelmakirjastojen alkuperäisestä tarkoituksesta, että 1940-luvulla sekä 1950-luvun alussa, ohjelmakaupoissa oli todellisia koodikirjastoja sisältäen keloittain kasetteja tai myöhemmin korttien pinoja joista ohjelmoija tarvitessaan valitsi rutiineja ladatakseen omaan ohjelmaansa. Kun lataajat ja linkittäjät alkoivat ratkaista symbolisia viittauksia, tuli mahdolliseksi automatisoida prosessi valitsemalla rutiineja kirjastosta joka ratkaisi muutoin määrittelemättömät symbolit.

Levine jatkaa, että kirjastotiedosto ei ole pohjimmiltaan muuta kuin kokoelma objektitiedostoja tavallisesti niihin lisätyn hakemistotiedon hakujen nopeuttamisen kanssa (Levine 1999, 169).

Olio-ohjelmoinnissa ohjelmakirjastoa käsitteellisesti läheinen luokkakirjasto on kokoelma valmiita luokkia, joita ohjelmoija voi käyttää kehittäessään sovellusohjelmaa. Proseduraalisessa ohjelmoinnissa termiä luokkakirjasto käytetään tarkoittamaan aliohjelmakirjastoa. (Whatis.com 2011.)

Ohjelmakomponentti on se mikä todellisuudessa julkaistaan – järjestelmän eristettävissä oleva osa – komponenttilähtöisessä lähestymistavassa. Vaikka komponentti voi olla yksittäinen luokka, se on todennäköisemmin kokoelma luokkia, jota joskus kutsutaan moduuliksi. Komponenteilla on edelleen ominaisuuksia, jotka erottavat ne moduuleista. (Szyperski ym., 2002, 10-11.) Moduuleja voidaan käyttää, ja on aina käytetty, paketoimaan useampia entiteettejä, kuten luokkia, yhdeksi yksiköksi. Komponentti voi sisältää myös komponentteja tai tavallisia proseduureja. (Szyperski ym., 2002, 38-39.)

2.2 Luokkakirjastoja

2.2.1 C++ Standard Template Library

C++:n Standard Template Library (STL) on suuri komponentti ANSI/ISO-standardin C++:n kirjasto-osalle. Se kehitettiin Hewlett-Packard Labsilla Alexander Stepanovin ja Meng Leen toimesta. He perustivat työn vahvasti Stepanovin ja David R. Musserin työhön. (Plauger ym. 2001, ix.)

C++-ohjelmointikieli on perinteisesti tarjonnut peruskirjaston joka tarjoaa *funktioita* syöttöä ja tulostusta, matemaattisia funktioita sekä muita varten. Jotkin näistä funktioista kuuluvat Standard C –kirjastoon, kuten printf, toiset vain STL:ään, kuten set_new_handler. Joka tapauksessa kirjastofunktio vastaanottaa joitakin ennaltamääriteltyjä argumentteja ja palauttaa jokin ennaltamääritellyn tyyppisen arvon. Muutoin se vaikuttaa toisiin tallennettuihin arvoihin. Osa laajasti hyödyllisen C- tai C++-kirjaston suunnittelusta on kaikkein suosituimpien argumenttityyppien määrittämisessä eri funktioiden tukemisessa. (Plauger ym. 2001, 1.)

C++ on C:tä kehittyneempi tärkeällä tavalla. Se laajentaa C:n tietorakennetta sisältämään jäsenfunktioita ja jäsenobjekteja. C++:n luokat voivat paremmin kapseloida dataa ja operaatioita joita voidaan suorittaa ko. datalle. Osa laajasti hyödyllisen C++-luokkakirjaston suunnittelusta on suosituimpien jäsenobjektityyppien yhdistelmien määrittämisessä niiden tukemiseksi näissä eri luokissa. (Plauger ym. 2001, 1.)

2.2.2 .NET Framework Class Library

.NET Frameworkin luokkakirjasto on nimeltään .NET Framework Class Library (FCL). Siihen kuuluu useita tuhansia tyyppimäärittelyjä ja jokainen tyyppi tarjoaa toimintoja. Kaiken kaikkiaan Common Language Runtime ja FCL mahdollistavat seuraavanlaisten sovellusten rakentamisen:

- XML- pohjaiset verkkopalvelut
- Web forms –sovellukset
- Windows Forms-sovellukset
- Windows-konsolisovellukset
- Windows-palvelut
- Itsenäisistä komponenteista muodostuvat kirjastot (Richter 2002, 21-22).

FCL sisältää kirjaimellisesti tuhansia tyyppejä. Samaan aiheeseen liittyvät tyypit sijaitsevat samassa nimiavaruudessa. Esimerkiksi system-nimiavaruus sisältää Object-perustyypin, josta kaikki muut tyypit viime kädessä periytyvät. Lisäksi System-nimiavaruus sisältää tyypit kokonaisluvuille, merkeille, merkkijonoille, poikkeusten käsittelylle ja konsolin I/O-toiminnoille sekä nipun työvälinetyyppejä, jotka turvallisesti muuntavat tyyppejä toisiksi, luovat satunnaislukuja ja suorittavat laskutoimituksia. Kaikissa sovelluksissa tarvitaan System-nimiavaruuden tyyppejä. (Richter 2002, 22.)

Ajoympäristön toimintojen käyttäminen edellyttää tietoa, missä nimiavaruudessa on tarkoitukseen sopiva tyyppi. Luokkakirjaston tyypeistä voi periyttää omia tyyppejä ja näin lisätä niihin omia räätälöityjä ominaisuuksia. .NET Framework perustuu olioajatteluun ja tarjoaa yhtenäisen ympäristön ohjelmistokehittäjille. Jokainen voi helposti luoda myös omia nimiavaruuksia ja niihin tyyppejä. Tämä

tapa helpottaa merkittävästi ohjelmistokehitystä verrattuna Win32-ohjelmointiympäristöön. (Richter 2002, 22.)

Useimmat FCL-kirjaston nimiavaruudet sisältävät tyyppejä, jotka ovat käytettävissä kaiken tyyppisissä sovelluksissa (Richter 2002, 22).

3 OHJELMAKIRJASTON OMINAISUUKSIA

Ominaisuuksiltaan korkealuokkainen ohjelmakirjasto on skaalautuva, tehokas, siirrettävä sekä uudelleenkäytettävä. Se on myös rakenteeltaan modulaarinen.

3.1 Skaalautuvuus

Skaalautuvuus (scalability) tarkoittaa sitä, miten hyvin ratkaisu johonkin ongelmaan toimii, kun ongelman koko suurenee (FOLDOC 1998).

Skaalautuvuus vaatii, että ohjelman tulee olla jotakuinkin tehokas laajan prosessorilukumäärän kanssa. Rinnakkaisalgoritmit voivat käyttää laajoja arkkitehtuurisuunnitelmia ja prosessorien lukumäärää. Niiden skaalautuvuus todennäköisesti vaatii, että pohjimmainen laskennan hienojakoisuus tulee olla säädettävissä sopimaan tiettyihin olosuhteisiin. Näissä olosuhteissa ohjelmisto voi tulla ajetuksi. (Demmel ym. 2004, 4.)

3.2 Siirrettävyys

Siirrettävyys eli portattavuus (portability) on helppous, jolla ohjelman osa, kuten komponentti, voidaan ”portata”, eli se voidaan tehdä ajettavaksi uudella alustalla ja/tai käännettäväksi uudella kääntäjällä (FOLDOC 1998).

Ohjelmistojen portattavuus on aina ollut tärkeä huomion kohde. Portattavuus oli helppo saavuttaa kun oli olemassa vain yksi arkkitehtuurinen paradigma (von Neumannin kone) ja yksi ohjelmointikieli tieteelliseen ohjelmointiin (Fortran) käsittäen yleisen tietojenkäsittelymallin. Arkkitehtuurinen ja kielellinen monimuotoisuus ovat tehneet portattavuuden paljon vaikeammaksi, mutta ei vähemmän tärkeäksi, saavuttaa. Käyttäjät eivät yksinkertaisesti tahdo sijoittaa suuria määriä aikaansa luodakseen suuriluokkaisia ohjelmakoodeja jokaiselle uudelle koneelle. Demmelin ym. vastaus tähän on kehittää portattavia ohjelmakirjastoja, jotka peittävät kone-spesifiset yksityiskohdat. (Demmel ym. 2004, 2.)

3.3 Tehokkuus

Kronsjön mukaan on suhteellisen helppoa keksiä algoritmeja, mutta tavoitteena onkin luoda hyviä algoritmeja ja todistaa ne hyviksi. Yksi tärkeimmistä kriteereistä on aika, joka algoritmin toteutuksen suoritukseen kuluu. Voidaan mitata ohjelman suoritusaikaa tietokoneella, mutta hyödyllinen vaihtoehto sille on analysoida matemaattisesti suoritusnopeutta. Algoritmien suhteen voidaan mitata sekä suoritusaikaa että muistinkulutusta. (Kronsjö 1987, 2.)

Naiivisti käsitettynä ‘tehokkuus’ on ennalta tuttu asia: me pyrimme minimoimaan ajan jonkin ongelman ratkaisuun. Kuitenkin, termi ’algoritmi’ on tässä avoin tulkinnoille. (Demmel ym. 2004, 3.)

3.4 Uudelleenkäytettävyys

3.4.1 Perusteita

Ohjelmakoodin uudelleenkäytöstä puhuttaessa voidaan sanoa, että uudelleenkäyttää voi algoritmeja, suunnittelutyylejä, vaatimusmäärittelyjä, proseduureja, moduuleja, sovelluksia, ideoita, suunnittelumalleja sekä arkkitehtuureja (Sametinger 1997, 1).

Hyvä ohjelmiston uudelleenkäyttöprosessi johtaa tuottavuuden, laadun ja luotettavuuden lisääntymiseen, ja rahallisten kulujen sekä toteutusajan vähentymiseen. Alkuinvestointi vaaditaan ohjelmiston uudelleenkäyttöprosessin aloittamiseen, mutta investointi maksaa itsensä takaisin muutaman uudelleenkäytön myötä. Lyhyesti sanottuna uudelleenkäyttöprosessin sekä tietovaraston kehittäminen tuottavat tietopohjan, joka kehittyy joka uudelleenkäyttökerralla. Se minimoi tulevaisuudessa suoritettavien projektien kehitystyön määrää ja lopulta vähentää uusien tietovaraston tietämykseen perustuvien projektien riskejä. (Jalender ym. 2010, 87.)

Käyttämällä hyviä ohjelmakirjastoja, ohjelmankehittäjät voivat keskittyä yhteisten alueiden toimintoihin samalla luottaen kirjastojen käsittelevän yleisempiä, aiemmin kohdattuja ongelmia. Vaikka uudelleenkäytöstä seuraa

selviä etuja, korkealuokkaisten ohjelmakirjastojen tunnistaminen voi olla vaikeaa. (Vaucher & Sahraoui 2010, 1.)

3.4.2 Mustan laatikon uudelleenkäytettävyys

Ohjelmistokomponentit tarvitsevat aina liittymän. Yksinkertaisesti ohjelmakoodin kopioiminen ja sen liittäminen uuteen paikkaan ei vastaa käsitystämme ohjelmistokomponentista. Tarvitaan jonkintyyppistä abstraktiota, ja uudelleenkäytön pitäisi olla mahdollista tietämättä komponentin sisäistä toteutusta. Tätä kutsutaan mustan laatikon uudelleenkäytöksi. (Sametinger 1997, 3.)

Toisaalta mustan laatikon uudelleenkäytettävyys tarkoittaa muuttamattoman lähdekoodin uudelleenkäyttöä. Kirjaston funktiota käytetään normaalisti ilman muutoksia ja on siten esimerkki mustan laatikon uudelleenkäytettävyydestä. (Kernebeck 1997, 50.)

Jotta mustan laatikon periaatteella toteutetun komponentin käyttö ja uudelleenkäyttö olisi mahdollista, on komponentin toiminta ja liittymä dokumentoitava riittävällä tasolla.

3.4.3 Valkoisen laatikon uudelleenkäytettävyys

Valkoisen laatikon uudelleenkäytettävyys sallii muutosten teon ohjelmiston osassa. Esimerkkinä valkoisen laatikon uudelleenkäytettävyydestä voidaan pitää mallipohjan uudelleenkäytön vaatimaa olemassa olevan mallin muuttamista tai laajentamista. (Kernebeck 1997, 50.)

Valkoisen laatikon uudelleenkäyttö on tyypillinen tapaus suunnittelemattomassa ns. ad-hoc-uudelleenkäytössä. Se tarkoittaa komponenttien, joiden sisäinen toteutus on muutettu uudelleenkäyttötarkoituksessa, uudelleenkäyttöä. Valkoisia laatikoita ei tyypillisesti käytetä sellaisenaan, vaan sopeuttamalla. Ne luovat enemmän mahdollisuuksia uudelleenkäyttäjille mielivaltaisten muutosten tekemisen helppouden vuoksi. Negatiivinen puoli valkoisen laatikon uudelleenkäytössä on se, että se edellyttää lisää testausta ja kalliimpaa

ylläpitoa. Toisin kuin mustien laatikoiden kanssa, uutta komponenttia, joka on johdettu modifioimalla aiemmin olemassa olleesta, täytyy kohdella uutena komponenttina ja testata se läpikotaisin. Lisäksi uusi komponentti vaatii erillisen ylläpidon. Jos komponentista on olemassa monia, vain vähän toisistaan eroavia kopioita, tulee työlääksi korjata virheitä, jotka vaikuttavat niihin kaikkiin. Jos komponenttiin tehdyt muutokset ovat vähäisiä, esimerkiksi muutamia muuttujan uudelleennimeämisiä tai muutoksia proseduurien kutsuissa, voidaan käyttää termiä *harmaa laatikko.* (Sametinger 1997, 29.)

3.4.4 Horisontaalinen uudelleenkäyttö

Horisontaalinen uudelleenkäyttö viittaa ohjelmistokomponentteihin, joita käytetään laajasti eri ohjelmistoissa. Koodin ominaisuuksien kannalta tämä sisältää tyypillisesti suunnitellut komponenttien kirjastot, kuten linkitettyjen listojen luokan, merkkijonojen käsittelyrutiinit tai graafisen käyttöliittymän toiminnot. Horisontaalinen uudelleenkäyttö voi myös viitata kaupallisen suoraan hyllystä otetun sovelluksen (Commercial Off The Shelf, COTS) tai kolmannen osapuolen ohjelman käyttöön. On olemassa suuri määrä ohjelmakirjastoja sekä tietovarastoja, jotka sisältävät tämäntyyppistä koodia ja dokumentaatiota eri paikoissa Internetiä. (Jalender ym. 2010, 88.)

3.4.5 Vertikaalinen uudelleenkäyttö

Vertikaalinen uudelleenkäyttö on uudelleenkäyttöä saman domainin tai sovellusalueen alueella. Sen tavoite on johtaa yleiset mallit järjestelmäperheille, joita voidaan käyttää pohjina uusien järjestelmien kokoamiseen. Mitä kapeampi domain eli alue sitä suurempi etu. (Prieto-Díaz 1993, 63.)

Vertikaalisella uudelleenkäytöllä, ohjelmistoteollisuuden laajasti huomiotta jättämänä, mutta potentiaalisesti erittäin hyödyllisenä, on kauaskantoisia vaikutuksia nykyisille sekä tulevaisuuden ohjelmiston kehitystoimille. Jalender ym. viittaavat tekstiin (Sametinger ym. 1997), jossa sanotaan, että perusidea on järjestelmien toiminnallisten alueiden ja domainien uudelleenkäyttö. Silloin niitä

voivat käyttää saman toiminnallisuuden sisältävät järjestelmäperheet. (Jalender ym. 2010, 88.)

3.4.6 Uudelleenkäytettävyyden esteitä

Sametinger kertoo, että huolimatta ohjelmistojen uudelleenkäytön eduista, sitä ei harjoiteta niin usein kuin voisi luulla. On monia tekijöitä, jotka suoraan tai epäsuorasti vaikuttavat uudelleenkäytön onnistumiseen tai epäonnistumiseen. Nämä tekijät voivat olla käsitteellisiä, teknisiä, johtamistekijöitä, organisaatiosta johtuvia, psykologisia, taloudellisia tai luonteeltaan lainopillisia. (Sametinger 1997, 15.)

Yleisiä uudelleenkäytön esteitä ovat seuraavat:

- Johdon tuen puute: koska ohjelmiston uudelleenkäyttö aiheuttaa etukäteisiä kuluja, sitä ei voida hyväksyä laajasti organisaatiossa ilman ylimmän johdon tukea. Johdon täytyy olla informoituja aloituskuluista ja heidän täytyy olla vakuutettuja odotetuista säästöistä. (Sametinger 1997, 15.)

- Projektin johtaminen: perinteisen projektin johtaminen ei ole helppo tehtävä. Meillä on jopa vielä vähemmän kokemusta projekteista jotka hyödyntävät uudelleenkäyttöä. Askeleen nousu laaja-alaiseen ohjelmistojen uudelleenkäyttöön vaikuttaa koko ohjelmistojen elämänkaareen. (Sametinger 1997, 15.)

- Selvien toimintatapojen puute: Ohjelmistonkehitys on monimutkainen prosessi, joka sisältää monenlaisia aktiviteetteja. Onnistunut ohjelmiston uudelleenkäyttö vaikuttaa koko ohjelmiston elinkaareen, suunnittelumenetelmiin, projektisuunnitteluun sekä arviointiin. Malleja sellaisille prosesseille käytetään määrittämään erilaisia askeleita, jotka tulevat suoritetuksi tietyssä järjestyksessä. Jos nämä mallit eivät eksplisiittisesti edellytä ohjelmiston uudelleenkäyttöä, on todennäköistä, että se ei tule tapahtumaan käytännössä. (Sametinger 1997, 15.)

- Riittämättömät organisaation rakenteet: organisatoristen rakenteiden suhteen täytyy ottaa huomioon erilaisia tarpeita kun eksplisiittistä, laaja-alaista uudelleenkäyttöä otetaan käyttöön. Esimerkiksi erillinen tiimi voidaan asettaa uudelleenkäytettävien komponenttien keräämiseen, ylläpitämiseen ja tarjoamiseen. (Sametinger 1997, 16.)

- Ei-keksitty-täällä (not invented here): ihmiset voivat tuntea itsensä rajoitetuksi luovuudessaan ja itsenäisyydessään uudelleenkäyttäessään jonkin toisen ohjelmistoa. He haluavat kehittää oman uuden ohjelmistonsa ennemmin kuin pitää yllä jonkun toisen. He voivat myös olla ennakkoluuloisia jonkun toisen tekemää ohjelmistoa kohtaan luottamuksen puutteen kautta. Tätä kutsutaan ei-keksitty-täällä –syndroomaksi. (Sametinger 1997, 16.)

- Lainopilliset kysymykset: Mitä laajemmaksi ohjelmistojen uudelleenkäyttö tulee, sitä useampia lainopillisia sekä liiketoiminnallisia kysymyksiä täytyy kohdata, esimerkiksi vastuukysymyksiä sekä tietosuojakysymyksiä. Ulkopuolisten ohjelmatalojen käyttö lisää näiden kysymysten painoarvoa. (Sametinger 1997, 16.)

- Johdon aloitteiden puute: Kannustusten puute estää johtoa antamasta kehittäjien käyttää aikaa järjestelmän komponenttien tekemiseksi uudelleenkäytettäväksi. Heidän onnistumisensa määritellään usein sen ajan mukaan, jonka projektin valmiiksi saattaminen vei. Sen jälkeisen työn tekeminen, vaikka se olisi hyödyllistä yritykselle kokonaisuudessa, vähentää heidän onnistumistaan. Jopa silloin, kun komponentteja uudelleenkäytetään ohjelmistovarastojen kautta, saavutetut edut ovat vain murto-osa mitä voitaisiin saavuttaa eksplisiittisellä, suunnitellulla ja järjestelmällisellä uudelleenkäytöllä. Ei ole riittävää yksinkertaisesti käyttää tietovarastossa olevia komponentteja. Hyvin suunniteltujen komponenttien täytyy tulla kehitetyksi systemaattisesti ja käytetyiksi

> seuraten huolellista uudelleenkäyttöön perustuvaa prosessia. (Griss 1993, Sametinger 1997, 16 mukaan.)

Uudelleenkäytettäessä eri lähteistä peräisin olevia ohjelmamoduuleja voi esiintyä nimitörmäyksiä, jos moduuleja ei ole paketoitu omiin nimiavaruuksiinsa. Nimitörmäys syntyy, jos kaksi tai useampi samaan nimiavaruuteen sisältyvää moduulia käyttää esimerkiksi samaa muuttujan nimeä.

DLL Helliksi kutsutaan sitä, kun eri ohjelmat käyttävät samaa ohjelmakirjastoa, ja päivitettäessä ohjelmakirjastoa uudempaan versioon jotkin sitä käyttävistä ohjelmista lakkaavat toimimasta. Tutkimisen arvoinen asia voisi olla, olisiko mahdollista kehittää jonkinlainen hallintatyökalu käyttöjärjestelmäpalveluksi, joka luovuttaisi kunkin ohjelman käyttöön sellaisen version kustakin ohjelmakirjastosta, jolla se parhaiten toimii.

3.5 Staattiset ja jaetut kirjastot

3.5.1 Staattisten kirjastojen määritelmä

Wheelerin mukaan staattiset kirjastot ovat yksinkertaisesti kokoelma tavallisia objektitiedostoja. Staattisia kirjastoja ei käytetä niin usein kuin ennen, koska jaetut kirjastot ovat niitä kehittyneempiä. Staattiset kirjastot sallivat käyttäjien linkittävän ohjelmiin ilman, että koodi täytyisi kääntää uudelleen, mikä säästää uudelleenkääntämisen viemältä ajalta. Tosin uudelleenkääntämisen viemä aika on vähemmän tärkeä nykyajan nopeampien kääntäjien kanssa, joten tämä perustelu ei ole yhtä tärkeä kuin ennen. Staattiset kirjastot ovat usein hyödyllisiä kehittäjille jos he haluavat sallia ohjelmoijien linkittävän heidän kirjastoonsa, mutta eivät halua antaa kirjaston lähdekoodia. Tämä on etu kirjaston toimittajalle, muttei selvästikään ohjelmaa käyttävälle ohjelmoijalle. (Wheeler 2003, 2.)

Staattisesti jaetuissa kirjastoissa kirjasto rajataan tiettyihin osoitteisiin kirjaston luontihetkellä, ja linkittäjä sitoo ohjelman viittaukset kirjaston rutiineihin linkityksen aikana spesifisiin kirjastorutiineihin. Staattiset kirjastot tapaavat

osoittautua epämukavan joustamattomiksi, koska ohjelmat mahdollisesti täytyy linkittää uudelleen joka kerta, kun joku kirjaston osa muuttuu. Staattisten jaettujen kirjastojen luomisen yksityiskohdat voivat osoittautua erittäin hankaliksi. (Levine 1999, 1-10.)

Staattisen linkityksen aikana kaikki koodimoduulit kopioidaan yhteen ajettavaan tiedostoon. Eri moduulien sijainti kyseisessä tiedostossa viittaa niiden sijaintiin muisti-kuvassa ja siten osoiteavaruudessa ajon aikana. Tällöin linkittäjä voi määrittää kohdeosoitteet kaikille moduulien välisille viittauksille sekä lisätä ne ajettavaan tiedostoon. (Deller & Heiser 1999, 3.)

3.5.2 Jaettujen kirjastojen määritelmä

Jaetut kirjastot ovat kirjastoja, jotka ladataan ohjelmien toimesta niiden käynnistyessä. Kun jaettu kirjasto on asennettu asianmukaisesti, kaikki ohjelmat jotka käynnistyvät jälkeenpäin käyttävät automaattisesti uutta jaettua kirjastoa. (Wheeler 2003, 3.)

Staattisten kirjastojen ylläpito- ja resurssiongelmien käsittelemiseksi useimmissa nykyaikaisissa järjestelmissä käytetään jaettuja kirjastoja tai dynaamisesti linkitettyjä kirjastoja eli DLL:iä. Pääasiallinen ero staattisten ja jaettujen kirjastojen välillä on se, että jaettuja kirjastoja käytettäessä varsinainen linkittäminen viivästyy ajon aikaan, jolloin sen suorittaa erityinen linkittäjä-lataaja (linker-loader). Ohjelma ja sen kirjastot ovat erillään toisistaan ennen kuin ohjelma todellisuudessa ajetaan. (Dubois 2001, 91.)

Smaragdakisin mukaan dynaamisten kirjastojen käytön komponenttien esittämiseksi etuja ovat:

- Ohjelmointikielten käytön vapaus. Komponentit voidaan luoda keskenään eri ohjelmointikielillä.
- Tekijänoikeussuoja. Dynaamiset kirjastot ovat binäärimuotoisia komponentteja eli niiden ohjelmakoodi ei ole saatavilla.

- Latauksen aikainen konfigurointi. Eri dynaamisten kirjastojen yhdistäminen tehdään sovelluksen latauksen aikana, ei kääntämisen aikana. (Smaragdakis 2010, 1.)

Dynaaminen linkittäjä, erotuksena staattisesta linkittäjästä, lisää symbolisia viittauksia kirjastojen moduuleihin ajettavassa tiedostossa, ja jättää ne ratkaistaviksi ajon aikana (Deller & Heiser 1999, 3).

Smaragdakisin mukaan dynaaminen kirjasto ladataan suorittamisen aikana ohjelman osoiteavaruuteen. Dynaamisen kirjaston rutiinit tunnistetaan symboleilla ja on dynaamisen linkittäjän vastuulla yhdistää symbolit, joihin viitataan ajettavassa tiedostossa niihin symboleihin, jotka on tuotu dynaamisesta kirjastosta. Dynaamisten kirjastojen pääetu on se, että ajettavaa tiedostoa ei tarvitse rasittaa yleisen kirjastokoodin sisällyttämisellä. Tämä johtaa pienempiin ajettaviin tiedostoihin ja säästää siten levytilaa. Vielä tärkeämpää on se, että se mahdollistaa kirjastokoodin yksittäisen kopion pitämisen muistissa ajon aikana, vaikka koodia voi käyttää monet ajettavat tiedostot tai jopa käyttöjärjestelmän ydin itse. (Smaragdakis 2010, 3.)

Toinen dynaamisten linkittäjien etu on että niiden avulla voidaan välttää riippuvuuksien pysyvä kiinnittäminen kirjaston koodiin. Sen sijaan että staattisesti linkitetään johonkin koodin versioon, joka estää tulevaisuuden parannusten hyödyntämisen, tapahtuu dynaaminen linkitys ohjelman käynnistymisen yhteydessä. Siten eri kirjastoja, mahdollisesti uudempia ja parannettuja tai vain erilaisia toteutuksia, voidaan käyttää. Tämä mahdollistaa modulaarisuuden ja on ollut yksi pääsyistä miksi dynaamiset kirjastot ovat hyvä teknologia binäärisille objektijärjestelmille, kuten Microsoftin COM:lle. On syytä mainita, että tämä dynaamisten kirjastojen joustavuus on ollut myös ongelmien lähde kun kirjastojen versiointia ei ole toteutettu huolellisesti – termistä ”DLL hell” on tullut yleistä terminologiaa. (Smaragdakis 2010, 3.)

Dynaaminen linkitys siirtää suuren osan linkitysprosessista siihen kunnes ohjelma käynnistetään. Se tarjoaa monia etuja, joita on muuten vaikea saavuttaa:

- Dynaamisesti linkitetyt jaetut kirjastot ovat helpompia luoda kuin staattisesti linkitetyt jaetut kirjastot.
- Dynaamisesti linkitetyt jaetut kirjastot ovat helpompia päivittää kuin staattisesti linkitetyt jaetut kirjastot.
- Dynaamisesti linkitettyjen jaettujen kirjastojen semantiikka eli käyttäytyminen voi olla huomattavasti lähempänä kuin jakamattomien kirjastojen.
- Dynaaminen linkitys sallii ohjelman ladata ja vapauttaa rutiineja ajon aikana, toiminnallisuus, jota on muuten todella vaikea toteuttaa. (Levine 1999, 205.)

Dynaamisessa linkityksessä on myös joitakin haittapuolia. Ajon aikainen dynaamisen linkityksen aiheuttama suoritusajan kulutus on huomattava verrattuna staattisen linkityksen käyttöön, koska suuri osa linkitysprosessista täytyy tehdä uudelleen joka kerta kun ohjelma ajetaan. Jokainen ohjelmassa käytetty dynaamisesti linkitetty symboli täytyy tulla etsityksi symbolitaulukosta ja ratkaistuksi. Dynaamiset kirjastot ovat myös suurempia kuin staattiset kirjastot, koska dynaamisten kirjastojen täytyy sisältää symbolitaulukot. (Levine 1999, 247.)

3.6 Modulaarisuus

3.6.1 Modulaarisuuden etuja

Lähtökohtana on, että ongelman jakaminen osiin helpottaa hallitsemaan sen monimutkaisuutta (Blume 1997, 38).

Kun ohjelmakirjasto koostetaan itsenäisistä moduuleista, ne ovat toisistaan riippumattomia.

Blumen mukaan abstraktio ja modulaarinen ohjelmistosuunnittelu lisäävät selvyyttä ja tarjoavat selvät linjat sen suhteen, miten suuret projektit voidaan jakaa osiin. Usein abstraktion käytöstä kuitenkin maksetaan suuri hinta tehokkuuden suhteen. (Blume 1997, 89.)

3.6.2 Komponentit yleisesti

Ohjelmistokomponentti voidaan määritellä itsenäiseksi ohjelmistoyksiköksi, joka tarjoaa palvelujaan hyvin määriteltyjen rajapintojen kautta (Koskimies & Mikkonen 2005, 53).

Ohjelmakomponentti voidaan myös määritellä tyypiksi, luokaksi tai miksi tahansa työn tuotteeksi joka on erityisesti suunniteltu uudelleenkäytettäväksi (Jacobson ym. 1997, 85).

Modernit ohjelmistoympäristöt käyttävät hyödykseen komponentteja monista eri ohjelmakirjastoista. Ohjelmoijien käytössä olevat ohjelmakirjastot kehitetään tyypillisesti eri tahojen toimesta, ilman keskitettyä valvontaa. Täten eri kirjastojen liittymät ovat harvoin suoraan yhteensopivia. Kirjastojen yhdistämiseen tarvittavan koodin hinta ja monimutkaisuus on merkittävä. Se voi olla kohtuuttoman kallista: voi olla helpompaa uudelleenkirjoittaa tarvittavat komponentit komponenttien uudelleenkäytön sijaan, tai kirjaston koostamismekanismin tehokkuuskustannus voi olla hyväksymättömissä. (Smith 2009, 19.)

Komponentin uudelleenkäytön saaminen käytännölliseksi ja tehokkaaksi monien komponenttijärjestelmistä tuotujen komponenttien täytyy olla jossakin määrin muunneltavia ollakseen hyödyllisiä. Yksittäisten ohjelmistojärjestelmien erojen helpottamiseksi voidaan joko tarvita suuri määrä samanlaisia komponentteja tai voidaan rakentaa pienempi määrä joustavampia, yleisempiä komponentteja. Nämä yleisemmät komponentit täytyy erikoistaa ennen uudelleenkäyttöä. Monenlaiset erilaistamismekanismit tarjoavat erilaisia tapoja erikoistaa uudelleenkäytettäviä komponentteja. (Jacobson ym. 1997, 95-96.)

3.6.3 Komponenttien käytön etuja

Ohjelmakirjaston komponenttien yhteistoiminnan voidaan sanoa tuottavan synergia-etua eli sitä, että komponenttien yhteentoimiva kokonaisuus luo jotain sellaista toiminnallisuutta, jota ne eivät yksittäisinä saa aikaiseksi.

Gunasekaran (2003, 16-18) mukaan komponenttien käytön etuja ovat uudelleenkäytettävyys, modulaarisuus, ominaisuuksien kapselointi, joustavuus, laajennettavuus, portattavuus, huomionaiheiden jako sekä virheiden käsittely ja turvallisuus.

- Uudelleenkäytettävyys: Uudelleenkäytettävyys on yksi päämotivaatioista komponenttien käyttöön. Se voi merkittävästi vähentää ohjelmistojärjestelmän kehitysaikaa. Käytettynä asianmukaisesti, kokoelma aiemmin rakennettuja, standardoituja ohjelmistokomponentteja voidaan käyttää olemassa olevista kirjastoista sopivaan ohjelmistoarkkitehtuuriin perustuvan ohjelmistojärjestelmän rakentamiseksi. (Gunasekaran 2003, 16.)

- Modulaarisuus: Komponenttien käyttö ohjelmistojen rakentamiseksi saa aikaiseksi modulaarisen järjestelmän. Kukin komponentti on erillinen tai itsenäinen moduuli, joka suorittaa joukkoa itsenäisiä funktioita. Sellaisten modulaaristen komponenttien kokoaminen on kaikki mitä tarvitaan suurempien järjestelmien rakentamiseksi. (Gunasekaran 2003, 16.)

- Ominaisuuksien kapselointi: Komponentit kapseloivat päätietorakenteensa, algoritminsa sekä tietonsa. Niiltä edellytetään joustavia, standardoituja liittymiä, joita voivat toiset komponentit käyttää vuorovaikutukseen niiden kanssa. Tässä mielessä useimpia komponentteja voi ajatella mustan laatikon objekteina, jotka ottavat sisäänsä joukon syötteitä ja tuottavat yhdenmukaiset tulosteet. (Gunasekaran 2003, 16.)

- Joustavuus: Yksi pääasiallisista eduista komponenttien käytön suhteen on niiden joustavuus. Komponenttia, jonka tiedetään suorittavan tietyn joukon tehtäviä, voidaan käyttää missä tahansa sovelluksessa, joka tarvitsee kyseistä toiminnallisuutta. Joustavuus on yleisemmin nähty laitteiston toimintona. Komponenttia, kuten korvakuulokkeita, joilla on tietynlainen liitin, voidaan käyttää minkä hyvänsä äänentoistolaitteiston,

jossa on yhteensopiva portti, kanssa. Ohjelmistokomponenteilla ei vielä ole samaa standardisaation ja joustavuuden tasoa. Tämä johtuu useista käytössä olevista ohjelmointikielistä sekä nykypäivän teollisten standardien puutteesta. Kuitenkin, standardien ja ennalta-arvattavien ohjelmistoarkkitehtuurikäytänteiden ja infrastruktuurin käytöllä sekä valmistajista vapaiden teknologioiden käytöllä voidaan auttaa rakentamaan ohjelmistokomponentteja, jotka ovat erittäin joustavia. (Gunasekaran 2003, 16.)

- Laajennettavuus: Ohjelmistot, jotka ovat rakennettu käyttäen komponentteja, ovat helpommin laajennettavia kuin luonteeltaan monoliittiset järjestelmät. Tärkeä vaatimus monille tämän hetken järjestelmille on, että ne ovat laajennettavia ja yhteensopivia tulevaisuuden teknologioiden kanssa. Käyttämällä itsenäisiä komponentteja järjestelmien rakentamiseen, tämä kysymys voidaan käsitellä tehokkaasti. Niin kauan kuin liittymiä, joita komponentti esittää, ylläpidetään yhtenäisesti, uusia ominaisuuksia voidaan lisätä säännöllisesti tukemaan uudempia vaatimuksia. (Gunasekaran 2003, 17.)

- Portattavuus: Portattavuus on ”työ joka vaaditaan ohjelman siirtämiseksi yhdeltä laitealustalta ja/tai ohjelmistojärjestelmältä toiseen”. Ideaalitapauksessa, tämän työn pitäisi olla minimaalinen, jos sovellusohjelmisto on tarkoitettu toimimaan laaja-alaisilla alustoilla ja ympäristöissä. Käyttämällä alustariippumattomiin teknologioihin perustuvia komponentteja kuten JavaBeans tai ohjelmointikieliriippumattomia teknologioita kuten COM/DCOM, portattavuutta voidaan lisätä merkittävästi. (Charbonneau 2001, Gunasekaran 2003, 17 mukaan.)

- Toiminnallisuuden jakaminen: Yksi modulaarisen, komponentteihin perustuvan lähestymistavan eduista on ohjelmistojärjestelmien, joiden

toiminnallisuus voidaan jakaa erillisille alueille, rakentaminen. Esimerkkejä sellaisista järjestelmistä ovat ne, jotka on rakennettu ohjelmistoarkkitehtuurien kuten Model-View-Controller (MVC) perustalle. (Gunasekaran 2003, 17.)

- Virheiden käsittely ja turvallisuus: On paljon helpompaa käsitellä virheitä, jotka liittyvät joihinkin modulaarisen järjestelmän koostaviin spesifisiin komponentteihin kuin yhteen, suureen monoliittiseen järjestelmään. Gunasekaran viittaa (Charbonneau 2001) sen parantavan myös testattavuutta tai vähentävän vaadittua työtä sen testaamiseksi, suorittaako ohjelma halutun toiminnon. Tietoturvallisuusnäkökohtia voidaan myös käsitellä jokaiselle yksittäiselle komponentille tai järjestelmälle kokonaisuutena. (Gunasekaran 2003, 18.)

3.6.4 Komponenttien uudelleenkäytön ongelmia

Kimin (2005, 46-48) mukaan komponenttien käyttöön perustuvassa uudelleenkäytössä on ongelmia. Hän listaa uudelleenkäytön esteitä:

- Toiminnalliset erot: Tämä on mahdollisesti kaikkein vakavin syy miksi olemassa olevaa komponenttia ei voi käyttää uudelleen ilman muutoksia. Harvoin kyseessä on tapaus, että olemassa oleva komponentti ja uusi, kehitettävänä oleva komponentti kohtaavat täsmällisesti toiminnallisesti. Uusi, kehitettävä komponentti voi vaatia joitakin muutoksia vastaaviin toimintoihin olemassa olevassa komponentissa tai vaatia lisätoimintoja. Olemassaoleva komponentti voi olla liian monimutkainen toiminnallisesti ja esimerkiksi tehokkuussyistä useimmat ylimääräiset toiminnot voidaan joutua poistamaan. (Kim 2005, 46.)

- Ohjelmointikielten erot: Jos on päätetty että uusi ohjelmisto täytyy toteuttaa esimerkiksi Javalla, olemassa olevia komponentteja, jotka ovat kirjoitettuja esimerkiksi C:llä, ei voida käyttää ilman muutosten tekoa. Ei vaikka ne vastaisivatkin kaikkia muita vaatimuksia, kuten

toiminnallisuutta, käyttöympäristöä ja järjestelmäarkkitehtuuria. Vaikka ristiin kääntäminen voi joskus sallia komponenttien uudelleenkäytön, yleensä se ei ole mahdollista. (Kim 2005, 47.)

- Kohdeympäristön erot: Kohdeympäristöt sisältävät alustat, tietojärjestelmäarkkitehtuurit, tietojenkäsittely-ympäristöt ja käyttörajoitukset. Yhdelle tietylle kohdeympäristölle kehitettyjä ohjelmistokomponentteja ei usein voi uudelleenkäyttää ilman muutoksia toiselle kohdeympäristölle. (Kim 2005, 47.)

- Käyttöympäristön erot: Käyttöympäristöt sisältävät yhtäaikaisen käyttäjien lukumäärän, vain luku-tilan tai luku-kirjoitus-tilan käytön, hallittavan tiedon määrän ja tiedon syöttö- ja tulostusnopeuden. Yhdelle käyttäjälle kehitettyä ohjelmistoa ei voi yleensä muuntaa tukemaan monien yhtaikaisten käyttäjien ympäristöä. Vain lukemaan suunniteltua ohjelmistoa ei yleensä voida muuntaa tukemaan luku-kirjoitus-ympäristöä. Hitaalle tiedon syötölle tai kirjoitukselle suunniteltua ohjelmistoa ei voida yleensä muuntaa tukemaan ympäristöä, jossa syöttö- ja tulostusnopeus ovat hyvin korkeat. (Kim 2005, 47.)

- Teollisuusstandardien erot: On olemassa monia teollisuusstandardeja laaja-alaisilla tietojenkäsittelyn ja viestinnän alueilla. Näitä on olemassa esimerkiksi:
    - web-dokumenttistandardit (HTML, XML)
    - viestintästandardit (CDMA, GSM, 3G, Bluetooth, WiFi)
    - kaapelitelevisiostandardit (OCAP, ACAP)
    - yhdistettävyysstandardit (DLNA, Marlin, OSGi, UPnP)
    - multimedia-tiedonpakkausstandardit (MPEG, JPEG, BluRay)
    - kansalliset kielen merkistöstandardit (Unicode)
    - metadatan hallintastandardit (MOF, XMI, CWM, Dublin Core)
    - ulkoiset tietokannan käyttöstandardit (ODBC, JDBC) (Kim 2005, 47.)

- Tietotyyppien erot: Suuria tietomääriä hallitaan joko tiedostojärjestelmällä tai tietokannanhallintajärjestelmällä. Ohjelmistokomponentit, jotka ovat vuorovaikutuksessa tiedostojärjestelmien kanssa, ovat selvästi erilaisia kuin tietokantajärjestelmien kanssa olevat. Eri maat käyttävät erilaisia formaatteja sellaisen informaation kuin päiväys, aika tai valuutta tallettamiseen ja käyttävät eri mittayksiköitä. Jotkin ohjelmistot koodaavat tietoa ASCII-järjestelmällä, kun taas toiset EBCDIC:llä. Ohjelmistokomponentteja, jotka eksplisiittisesti käsittelevät sellaisia tietotyyppejä ja koodaustapoja, ei voida uudelleenkäyttää ilman muutoksia. (Kim 2005, 48.)

- Algoritmien ja tietorakenteiden erot: Laaja valikoima algoritmeja toteuttaa avainfunktiot ja –kyvykkyydet ohjelmistojärjestelmissä. Esimerkit sisältävät lajittelun, etsimisen, tiedon kahdentamisen, tietokannan lukitsemisen, tietokannan lokittamisen tiedon palautusta, turvallisuutta ja salausta varten sekä viestien ohjaamisen tietoverkossa. Jokaista sellaista toiminnallisuutta tai kykyä kohden on tyypillisesti enemmän kuin yksi algoritmi tai tekniikka ja mukanatulevat tietorakenteet niiden toteuttamiseen, jokainen erilaisine etuineen. Näitä tietorakenteita voivat olla esimerkiksi linkitetyt listat, hash-funktiot, binääripuut, B+-puut, R*-puut ja keot. Jokin tekniikka voi olla yksinkertainen ja nopea toteuttaa, mutta se voi johtaa alhaiseen suorituskykyyn, luotettavuuteen tai tietoturvatasoon. Tekniikka voi olla hyvä tukemaan pienen datamäärän hallintaa, kun se voi olla täysin sopimaton suurelle tietomäärälle tai nopealle tiedon syötölle. Tekniikka voi olla hyvä suuren määrän pieniä viestejä ohjaamiseen, kun se voi olla sopimaton pientä määrää suuria viestejä ja suurta määrää pieniä viestejä ohjaamiseen. (Kim 2005, 48.)

Jacobson ym. (1977, 299) kertovat vaatimusten määrittelystä, että komponenttijärjestelmän määrittelyjä on erityisen vaikea luoda seuraavista syistä:

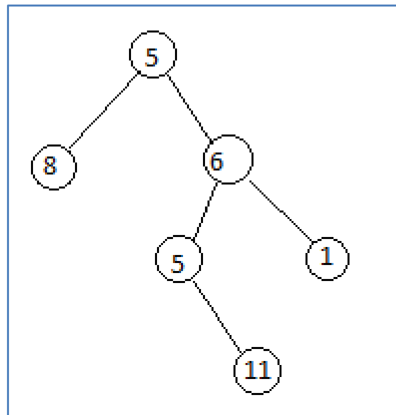
- Komponenttijärjestelmän täytyy täyttää monien ohjelmistojärjestelmien tarpeet.
- Komponenttijärjestelmää ei kehitetä, jotta se asennettaisiin ja otettaisiin käyttöön välittömästi, vaan enemmänkin toimimaan uudelleenkäytettävänä komponenttina monien vuosien käyttöajalle.
- Yhä vaihtelevampi joukko ihmisiä tuottaa määritykset komponenttijärjestelmälle nykyisten vaatimusten ymmärtämiseksi ja uusien ennustamiseksi.
- Oikean muunneltavuuden järjestäminen vaatii huolellista hyödyn määrittämistä ja huomattavaa kokemusta.

(Jacobson ym. 1997, 299.)

4 PUURAKENTEITA

Kari Peisan (2006, 1) mukaan puu on yhtenäinen, syklitön ja suuntaamaton verkko. Puut voidaan toteuttaa myös suunnattuna verkkona. (Peisa 2006, 1.)

Kokkarinen ja Ala-Mutka (2002, 91) kertovat puurakenteista, että ne ovat hierarkkisiksi rajoitettuja verkkorakenteita, joissa kullakin solmulla on korkeintaan yksi isä ja jokin määrä lapsia. Solmu on toisen solmun isä, jos ja vain jos toinen solmu on ensimmäisen lapsi. Puussa on yksi juurisolmu, jolla ei ole isää ja josta on lapsia seuraamalla tarkalleen yksi reitti jokaiseen muuhun solmuun. Puun ei tarvitse olla rakenteeltaan mitenkään tasapainoinen tai –laatuinen. Tämän vuoksi rakennetta ei voi tehokkaasti esittää yksinkertaisilla isä- ja lapsisolmujen osoitteet antavilla funktioilla, vaan solmuissa on oltava osoittimet, jotka sisältävät tämän informaation. (Kokkarinen & Ala-Mutka 2002, 91.)



Kuva 1 Yksinkertainen binääripuu

Kuvassa 1 on kuvattu yksinkertainen binääripuu, jossa on kuusi solmua ja jonka syvyys on kolme. Puu ei ole järjestetty.

Puun jokaisen solmun voi ajatella muodostavan alipuun, jonka juuri kyseinen solmu on. Solmu, jolla ei ole lapsia, on lehti, joka on itsessään yhden solmun muodostama alipuu. Puu voi olla tyhjä, jolloin osoitin sen juureen on arvoltaan NIL. (Kokkarinen & Ala-Mutka 2002, 91.)

4.1 Binääripuu

Binääripuu on joko tyhjä tai se on sellainen suunnattu puu, jonka jokaisella solmulla on korkeintaan kaksi lasta (Peisa 2006, 2).

Tasapainotettu binääripuu on puu, jonka juuresta on jokaiseen lehteen yhtä pitkä matka. Muutoin sanotaan puun olevan vino. (Kokkarinen & Ala-Mutka 2002, 95.)

Binääripuun rekursiivinen määritelmä on se, että binääripuu on äärellinen solmujen joukko, joka on joko tyhjä tai koostuu juurisolmusta sekä kahdesta erillisestä binääripuusta, joita kutsutaan juurisolmun vasemmaksi ja oikeaksi **alipuuksi**. Binääripuiden algoritmit ovat yleensä rekursiivisia. (Peisa 2006, 2.)

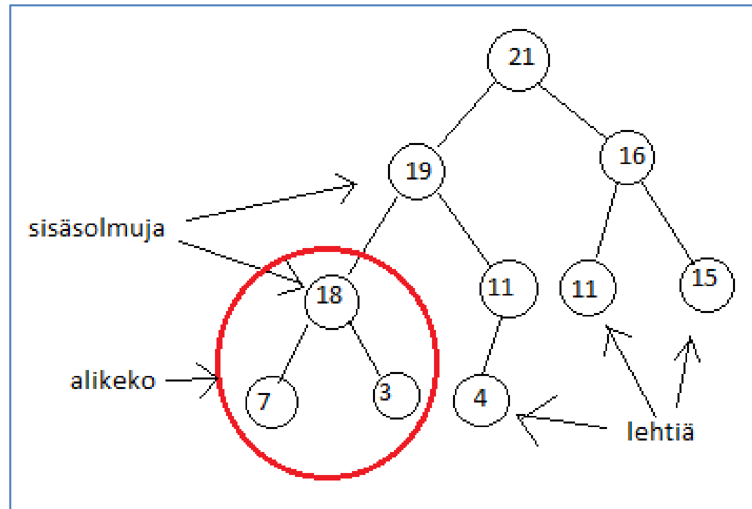
4.2 Keko

Keko voidaan esittää isä- ja lapsisolmujen muodostamana puurakenteena. Puurakenne on aina tasapainossa, koska sen ylemmät kerrokset ovat täysiä, ja ainoastaan alin kerros voi olla oikeasta reunastaan vajaa. Maksimikeon ollessa kyseessä keon suurin alkio on aina taulukon alussa eli puun juuressa. (Kokkarinen & Ala-Mutka 2002, 56.)

Kekoehto tarkoittaa sitä, että maksimikeossa isäsolmu on suurempi kuin lapsisolmunsa, ja minimikeossa isäsolmu on pienempi kuin lapsisolmunsa (Teknillinen korkeakoulu 2011).

Binäärinen keko tietorakenteena on taulukkotyyppinen objekti, joka voidaan esittää täydellisenä binääripuuna. Jokaisella puun solmulla on vastineensa taulukossa, jossa säilytetään solmun arvoa. (Cormen ym. 1998, 140.)

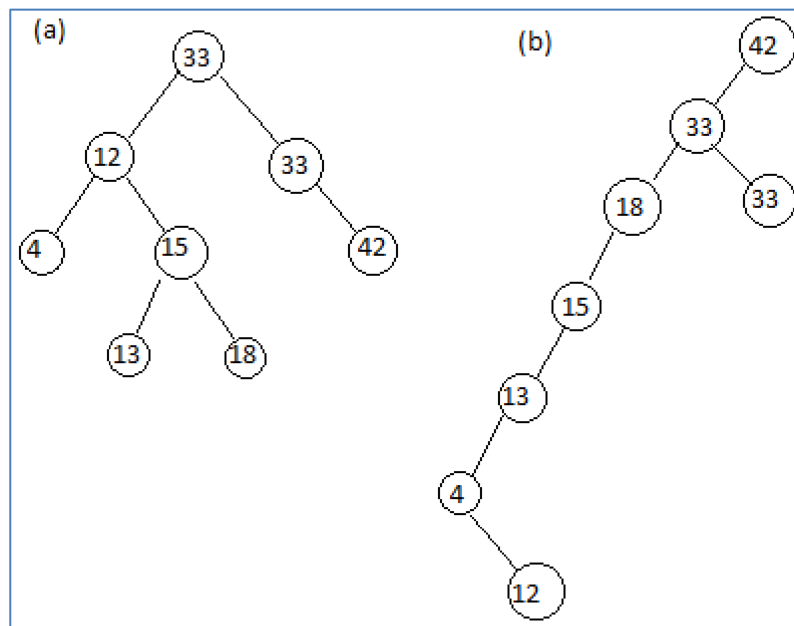
Kuvassa 2 kuvataan keko isä- ja lapsisolmujen muodostamana puurakenteena. Taulukkona puu voitaisiin ilmaista arvojensa mukaan {21, 19, 16, 18, 11, 11, 15, 7, 3, 4}. (Kokkarinen & Ala-Mutka 2002, 56.)



Kuva 2. Keko (Kokkarinen & Ala-Mutka 2002, 56)

4.3 Järjestetty binääripuu

Järjestetyssä binääripuussa jokaisen solun vasemmassa alipuussa on solua pienempiä arvoja ja solun oikeassa alipuussa suurempia.



Kuva 3. Binäärihakupuita (Kokkarinen & Ala-Mutka 2002, 96)

Kuvassa 3 kuvataan kaksi binäärihakupuuta, joilla on samat alkiot. Alkioiden lisäysjärjestys vaikuttaa syntyvän binääripuun muotoon. Mitä enemmän lisättävät alkiot ovat keskenään järjestyksessä, sitä pahemmin syntyvä puu on

epätasapainossa. (a) Alkiot lisätty järjestyksessä 33, 12, 4, 33, 15, 42, 18, 13. (b) Alkiot lisätty järjestyksessä 42, 33, 18, 33, 15, 13, 4, 12. (Kokkarinen & Ala-Mutka 2002, 96.)

5 OHJELMAKIRJASTON KOOSTAMINEN

5.1 Määrittely

Ohjelmakirjaston perusratkaisu on siihen mukaan otettavien algoritmien valinta. Jokainen niistä suorittaa oman osaongelmanratkaisunsa, ja yhdessä ne voivat koostaa jonkin ongelma-alueen kokonaisratkaisun. Ohjelmakirjaston sisältämien algoritmien rajapintojen tulisi olla tyypiltään konsistentteja, jotta niiden käyttö olisi yhtenäistä ja loogista. Ohjelmakirjaston komponentteina olevat algoritmit voivat joko riippua toisistaan tietyillä tavoilla tai olla itsenäisiä. Toisistaan riippuvat algoritmit ovat testauksensa osalta vaikeammin toteutettavissa.

Rajapintojen määrittely seuraa vaatimusta pystyä ohjaamaan komponentin algoritmia ulkoapäin. Rajapinta voi olla toteutettu läpinäkyvästi (valkoisen laatikon periaate) tai se voi piilottaa sisältönsä täydellisesti (mustan laatikon periaate). Rajapinnan tulisi olla periaatteessa muuttumaton kirjaston eri versioiden välillä, jotta sitä hyödyntävä koodi ei rikkoonnu kun tuotetaan uusia ohjelmaversioita, silti komponentin sisäistä toteutusta voidaan muuttaa, jos se ei näy ulospäin.

5.2 Suunnittelu

Ohjelmakirjaston komponentin tulevaisuuden muutokset tulisi ottaa huomioon ohjelmakehityksen suunnitteluvaiheessa. Jos ne jätetään täysin huomiotta suunnitteluvaiheessa, ne tulevat vastaan rajoituksina toiminnallisuudessa toteutusvaiheessa sekä tulevaisuudessa komponenttien toiminnallisuutta muutettaessa. Jos komponentti tarvitsee jonkin uuden toiminnallisuuden, jota ei suunnitteluvaiheessa ole otettu mitenkään huomioon, on mahdollista, että se aiheuttaa suuria muutoksia myös muualla ohjelmakirjaston sisäisessä toteutuksessa.

Ohjelmakirjaston suunnittelussa on otettava huomioon haluttava skaalautuminen, eli jos ohjelmakirjaston sisältämän ohjelmakoodin on

tarkoitettu tulevan suoritetuksi säikeistetyksi, tämä suunnittelupäätös tulee tehdä jo aloitettaessa kirjaston algoritmien suunnittelua.

Tärkeä vaatimus hyvälle ohjelmistokomponentin suunnittelulle on se, että sen rajapinnan tulisi pysyä tulevaisuuden muutoksiin nähden mahdollisimman muuttumattomana. Komponentin ”kypsyysvaatimus” olisi siis se, että sen rajapinta mahdollistaa riittävän sisäisen toiminnallisuuden suhteessa siihen tulevaisuudessa välttämättä tuleviin muutoksiin.

Viittaukset ulospäin komponenttimoduulin sisältä voivat olla järjestelmäriippuvaisia, kuten käyttöjärjestelmäspesifisiä tai käyttöjärjestelmäriippumattomia, kuten Javaa käytettäessä. Java on riippumaton käytettävästä alustatoteutuksesta. Java myös toteuttaa nimiavaruuksia package-määreellä.

Aina kun se on mahdollista, tulisi käyttää jo valmiina olemassa olevia kirjastoja, kuten C/C++:n, Javan tai .NET-arkkitehtuurin mukana tulevia peruskirjastoja. Onhan turhaa toteuttaa uudelleen jo ennen toteutettuja ohjelman osia, ja luokkakirjastojen koodi on yleisyytensä myötä testattua ja rajapintojensa osalta standardoitua.

Komponentin sisällyttäminen johonkin kirjastoon tulisi tehdä siten, että se sopii komponenttien muodostamaan kokonaisuuteen mahdollisimman hyvin, jotta ei jälkeenpäin nouse tarvetta siirtää sitä johonkin toiseen ohjelmakirjastoon.

Komponentin uudelleenkäytön tekemiseksi käytännölliseksi ja tehokkaaksi, monien komponenttijärjestelmän tuomien komponenttien täytyy tarjota jonkinasteista muunneltavuutta ollakseen käytännöllisiä (Jacobson ym. 1997, 95).

5.3 Toteutus

Ohjelmakirjaston algoritmit voidaan ensin ohjelmoida ja testata erikseen, ja sitten yhdistää ne yhdeksi ohjelmakirjastoksi. Toinen tapa olisi kehittää montaa, yhteentoimivaksi tarkoitettua, algoritmia yhtä aikaa. Sellainen ohjelmointityyli ei

tosin tue välttämättä modulaarisuusperiaatetta kovin hyvin, jos ohjelmamoduulien toteutuksista tulee riippuvaisia toisistaan.

Ohjelmakirjastoon voidaan sisällyttää täysin uutta koodia, joka testataan ja joka otetaan sellaisenaan tuotantokäyttöön, tai osa kirjaston komponenteista voidaan ottaa aiemmin luoduista komponenteista, jos soveliaita on saatavilla.

Osa ohjelmakirjaston komponenteista voi olla vain kirjaston sisäisessä käytössä, ja vain osa tarjoaa palveluitaan liittymänsä kautta koko kirjaston käyttäjälle eli kirjastosta ulospäin.

Ohjelmakirjasto voi muodostua yksinkertaisesta kokoelmasta algoritmeja, jotka liittyvät yhteen ohjelmointiprojektiin, tai toisessa ääripäässä voidaan puhua luokkakirjastoista, joita tulee esim. ohjelmointikielten kuten C++ tai Java mukana. Luokkakirjastoihin liittyy piirre, että ne löytyvät jokaiselta ohjelmankehittäjältä, silloin kun ne tulevat ohjelmointikielen kehitysympäristön mukana.

Ohjelmistoprojektissa, jossa on useita ohjelmoijia, ohjelmakirjasto valmistuu vasta, kun kaikki moduulit saatu valmiiksi ja testattua sekä yksin että yhdessä.

5.4 Integrointi ja testaus

Integrointi on erityisen tärkeä vaihe ohjelmakirjastoa luotaessa, sillä on mahdollista, että kirjaston sisältämät ohjelmakomponentit on tarkoitettu käytettäväksi vain ja ainoastaan yhdessä.

Niin kauan kuin ohjelmakirjaston sisältämät algoritmit ovat toisistaan riippumattomia, ne voidaan testata erikseen. Muutoin testiolosuhteet on järjestettävä siten, että myös algoritmien yhteentoimivuus testataan jollakin tavoin. Algoritmin testaus on järjestettävä sen mukaan, onko se joka suorituskerrallaan alustuva vai eri suorituskertojen välillä tilansa säilyttävä. Testaustilanteiden järjestäminen monimutkaistuu, jos osa algoritmeista on tilansa säilyttäviä. On myös mahdollista, että osa algoritmeista on joka suorituskerralla itsensä alustavia ja osa tilansa säilyttäviä.

Peräkkäisillä suorituskerroilla tilansa säilyttäviä algoritmeja voidaan tutkia jättö- ja tuloarvojensa osalta siten, että tarkoitus on tutkia peräkkäin syötettyjen arvojen sarjaa. Itsensä joka suorituskerralla alustavat testataan vertaamalla kerran syöttö- ja tulosarvoja.

Yhdessäkin moduulissa oleva virhe yhdistetään koko ohjelmakirjaston toimimattomuudeksi. Komponenttien yhteentoimimattomuudesta kumpuavat ongelmat ovat erityisen vaikeita, jos esimerkiksi uuden komponentin lisäys saa aiempien käytössä olleiden komponenttien koodin rikkoontumaan.

6 ESIMERKKIKIRJASTO

6.1 Algoritmien ominaisuuksista

Koska algoritmit on toteutettu Javalla, ne ovat suoritettavissa kaikissa laiteympäristöissä, johon Javan ajoympäristö on asennettu. Ohjelmakoodia ei tarvitse muokata eikä kääntää, jotta sitä voi ajaa eri laiteympäristöissä tai käyttöjärjestelmissä. Täten myös liitteenä olevista ohjelmalistauksista generoidut javadoc-dokumentaatiot pätevät samalla lailla kaikkiin ajoympäristöihin. Tämän työn algoritmit oletettavasti toimivat myös vanhempien Javan versioiden kanssa, koska algoritmit käyttävät vain Javan perusominaisuuksia, tätä ei tosin ole testattu tätä työtä tehtäessä.

Työn algoritmit on kirjoitettu mahdollisuuksien mukaan olio-ohjelmointiominaisuuksia käyttäviksi, eli tiedot on pyritty kapseloimaan (private-määreillä) luokkien sisään ja puurakenteiden solmuista on tehty oma luokkansa (Solmu-luokka) aina kuin mahdollista. Täten siirtäminen esimerkiksi C++-kielelle ei tulisi olla vaikeaa, kun taas proseduraaliset kielet edellyttäisivät perustavanlaatuisempia muutoksia ohjelman toimintalogiikkaan ja tietorakenteisiin.

Työn algoritmeja eli niiden toteutuksia ei ole valittu tehokkuussyiden perusteella, esimerkiksi ratkaisua käyttää nimenomaan rekursiivisia algoritmeja ei ole perusteltu millään tehokkuuteen liittyvällä perusteella.

Algoritmien toimintaa on esitetty ja demonstroitu vain hyvin pienillä esimerkkipuilla, jotka eivät sinällään kykene kertomaan toteutusten tehokkuudesta mitään.

HeapSort-toteutus vie aikaa matemaattisesti katsottuna O(n lg n) verran, koska kutsu BuildHeapiin vie O(n) ja jokainen n-1 kutsu Heapifyyn vie O(lg n) (Cormen ym. 1998, 147).

Puun solmun edeltäjän etsimisalgoritmin matemaattinen ajoaika puulle, jonka korkeus on h, on O(h), koska algoritmissa joko seurataan polkua ylös puuta tai seurataan polkua alas puuta (Cormen ym. 1998, 249).

Puun läpikäymisalgoritmin suoritus vie O(n) aikaa n-solmuiselle binäärihakupuulle, koska ensimmäisen kutsukerran jälkeen toimenpidettä kutsutaan rekursiivisesti täsmälleen kahdesti joka solmulle puussa – yhden kerran sen vasemmalle ja yhden kerran sen oikealle puulle (Cormen ym. 1998, 245-246).

Puun ääriarvot tutkiva algoritmi ajetaan O(h)-ajassa puulle, jonka korkeus on h, koska silloin seurataan polkua alaspäin puussa (Cormen ym. 1998, 248).

Solmun edeltäjän etsimisalgoritmi puulle, jonka korkeus on h, vie O(h) aikaa, koska algoritmissa joko seurataan polkua ylös puuta tai seurataan polkua alas puuta (Cormen ym. 1998, 249).

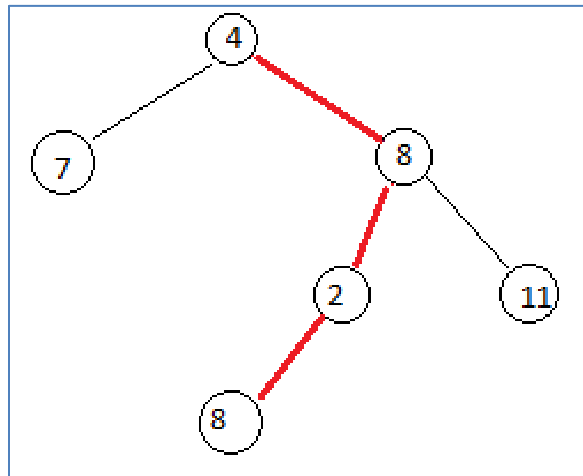
Esimerkkikoodit eivät tutki syötteiden kokoa mitenkään, eli jos Javan tietotyyppien koot tulevat vastaan on odotettavissa ylivuotoja, eivätkä algoritmitoteutukset tutki ja varoita sellaisesta laisinkaan.

Työn algoritmien uudelleenkäytettävyys on helppoa, koska algoritmit eivät ole kovin monimutkaisia ja ne on pyritty kommentoimaan mahdollisimman ymmärrettävästi. Puun prosessoinnin toteuttava algoritmi mahdollistaa oman käsittelijän käytön puun solmujen läpikäyntiin, sallien siten muita algoritmeja hieman paremman uudelleenkäytön.

Toteutetut algoritmit ovat uudelleenkäytettäviä valkoisen laatikon uudelleenkäyttöperiaatteella, sillä niiden koodia voi tutkia ja muuttaa vapaasti. Näiden toteutettujen algoritmien päälle voidaan siis haluttaessa rakentaa toiminnallisuudeltaan runsaampia versioita kehittämällä olemassa olevaa koodia eteenpäin.

Työn algoritmit eivät hyödynnä säikeitä eivätkä täten tue useampien prosessoriytimien hyödyntämistä.

6.2 Puun syvyyden määrittäminen



Kuva 4 Binääripuu, jolla syvyys 3

Kuvassa 4 on kuvattu yksinkertainen binääripuu, jonka syvyys on 3. Sillä on siis maksimissaan kolme askelta juuresta alimpaan lehteen. Polku juurisolmusta alimpaan solmuun on merkitty punaisella. Puun ei tarvitse olla järjestetty.

Liitteessä 1 EtsiSyvyys.java, jossa lähdekoodina toteutus puun syvyyden määrittävästä algoritmista.

Algoritmi toimii rekursiivisesti, eli se kutsuu itseään siten, että se aina ensisijaisesti lähtee käsittelemään vasenta alipuutaan, ja kun alin puu on löytynyt, se palaa ylemmälle rekursiotasolle ja siirtyy käsittelemään oikeanpuoleista alipuutansa.

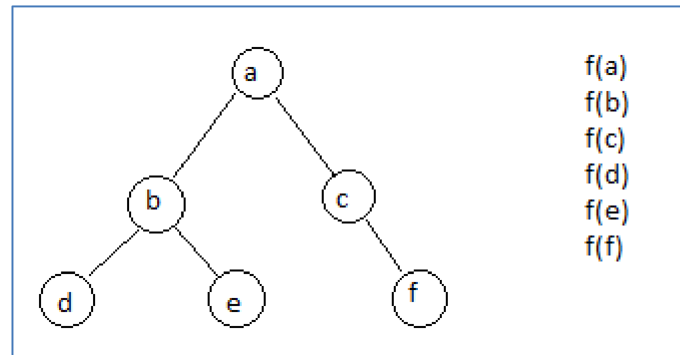
Luokka Solmu kuvaa puun solmua ja sisältää asetusmetodit, joilla solmulle voi asettaa vasemmanpuoleisen lapsisolmun tai oikeanpuoleisen. Metodi syvyys() laskee puun syvyyden, kun sille annetaan parametrinä puun juurisolmu.

Algoritmin koostamisessa käytetty mallia sivustolta (Penton 2001).

6.3 Puun läpikäynti

Binääripuun suhteen voidaan toteuttaa toiminnallisuus, jolla voidaan käsitellä halutulla tavalla kaikkien puun solmujen sisältämä yksilöllinen tietoaines. Liitteessä 1 ProsessoiPuu.java on em. toiminnallisuus toteutettu siten, että

jokaista luotavaa solua kohden asetetaan käsittelijä, jonka saa vapaasti määrätä. Käsittelijä määrittää sen, miten kunkin solun tieto käsitellään.



Kuva 5 Läpikäytävä puu

Kuvassa 5 on kuvattu binääripuu, jolla on arvot a-f. Funktio f kuvaa funktiota, jolla käsitellään jokaisen binääripuun solmun arvo.

Käytännössä luokka Käsittelijä voidaan siis periyttää. Uuteen luokkaan muodostetaan metodi:

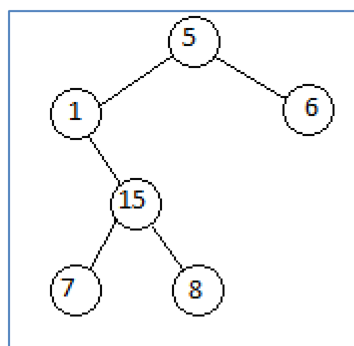
```
protected void käsittele(Solmu S)
```

Metodi korvaa siis yläluokkansa samannimisen metodin. Luokan periyttäminen tapahtuu seuraavan syntaksin mukaisesti:

```
class OmaKäsittelijä extends Käsittelijä
```

Ja solmu asetetaan silloin:

```
Käsittelijä k=new OmaKäsittelijä();
Solmu n1=new Solmu(k);
```

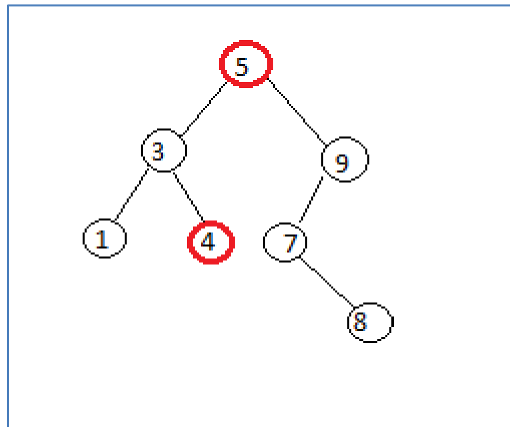


Kuva 6 Demonstraatiossa käytetty puu

Kuvassa 6 on esimerkkialgoritmin demonstraatiossa käytetty puu.

Algoritmin koostamisessa on käytetty mallia kirjallisuudesta (Cormen ym. 1998, 245).

6.4 Puun solmun edeltäjän etsiminen



Kuva 7 Solmun edeltäjän etsiminen

Järjestetyn binääripuun tietyn solmun edeltäjä määritellään siksi solmuksi, jonka arvo on seuraavaksi suurin itse solmulle, jonka edeltäjää haetaan. Algoritmi on liitteessä nimellä Successor.java. Algoritmin koostamisessa on käytetty mallia kirjallisuudesta (Cormen ym. 1998, 249).

Kuvassa 7 etsitään solmulle, joka sisältää arvon 4 edeltäjää. Edeltäjä on siis solmu, jossa on arvo 5, koska se on seuraavaksi suurin puun arvoista nähden arvoa 4.

Jos toteutetulla algoritmilla yritetään hakea puun suurimman arvon sisältävälle solmulle edeltäjää, algoritmi palauttaa null-arvon.

Kuvassa 6 on esimerkki demonstraatio-puun käytöstä.

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS   NEXT CLASS   FRAMES   NO FRAMES   All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

Class Successor

```
java.lang.Object
  └ Successor
```

```
public class Successor
extends java.lang.Object
```

Testaa Solmu-luokan toteutusta puun solmun edeltäjän etsimisestä

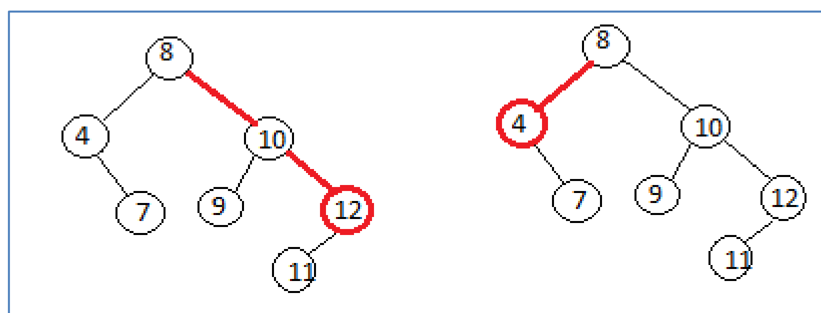
Author:

Antti Suutarinen

Kuva 8 Javadoc-dokumentaatiota

Kuvassa 8 on kuva esimerkkikirjasto-paketin Successor-luokan javadoc-dokumentaatiosta. Javadoc-dokumentaatiot generoidaan ohjelmalistaukseen kirjoitettavien ns. tagien perusteella HTML-tiedostoiksi.

6.5 Järjestetyn binääripuun minimi- ja maksimiarvot



Kuva 9 Järjestetyn binääripuun ääriarvojen etsiminen

Järjestetyn puun ääriarvojen etsiminen on suhteellisen yksinkertaista. Maksimiarvo löytyy käymällä puun juurisolmusta alkaen jokaisen läpikäydyn solmun oikeaan lapseen eli siirrytään aina, kun oikea solmu on olemassa siihen ja jatketaan prosessia, kunnes saavutaan solmuun, jolla ei ole oikeaa lasta. Tämä solmu sisältää silloin suurimman arvon. Minimiarvon sisältävä solmu löytyy symmetrisesti eli vastaavasti, mutta etsimisprosessissa käydään läpi

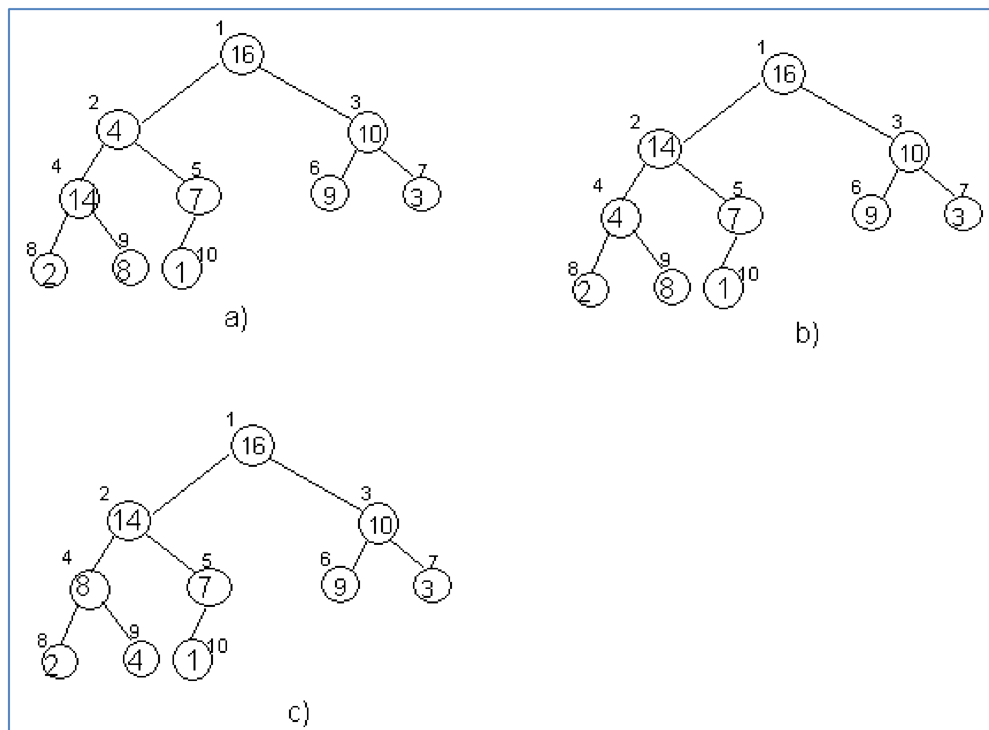
solmujen vasemmat lapset, kunnes saavutaan solmuun, jolla ei ole vasenta lasta.

Kuvassa 9 vasemmassa puussa on etsitty maksimiarvoa, ja löydetty arvo 12, joka on siis koko puun suurin arvo. Oikeanpuoleisessa puussa on haettu pienintä arvoa ja löydetty arvo 4.

Algoritmit löytyvät liitteenä PuunÄäriarvot.java:sta. Algoritmin koostamisessa on käytetty mallia kirjallisuudesta (Cormen ym. 1998, 248). Kuvassa 6 on esimerkki demonstraatio-puun käytöstä.

6.6 Heapsort eli kekolajittelu

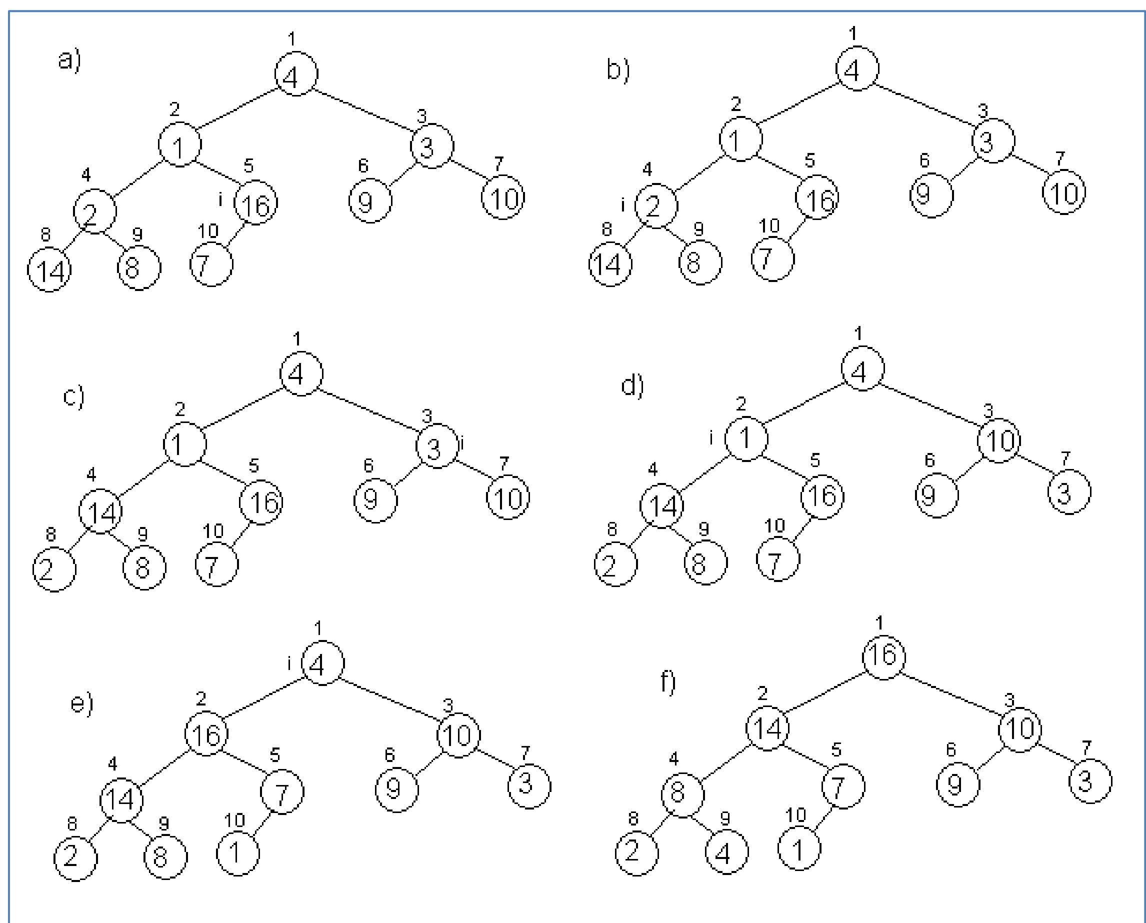
Kekolajittelu toimii muodostaen lajiteltavan taulukon perusteella keon sisäisesti, ja tuloksena on paikallaan järjestetty taulukko. Liitteessä HeapSort.java on toteutettu algoritmi. Algoritmin koostamisessa on käytetty mallia kirjallisuudesta (Cormen ym. 1998, 143, 145, 147).



Kuva 10 Heapify-metodin toiminta (Cormen ym. 1998, 143)

Kirjassa (Cormen ym., 1998, 143) selostetaan myös algoritmin Heapify-metodin toimintaa. Kuvassa 10 näytetään kolme vaihetta, joista metodin toiminta selviää. Kohdassa a) on alkuperäinen keon järjestys, jossa solmu i=2 rikkoo keko-ominaisuutta, koska se ei ole suurempi kuin molemmat lapsisolmunsa. Keko-ominaisuus palautetaan solmulle i=2 kohdassa b).
b) Solmut i=2 ja i=4 vaihdetaan keskenään, joka tuhoaa keko-ominaisuuden solmu i=4:lle. Rekursiivinen kutsu Heapify(A, 4) asettaa i=4. Solmu i=4 ja i=9 vaihdetaan kohdassa c)

c) Nyt solmu i=4 korjaantuu, ja rekursiivinen kutsu Heapify(A, 9) ei aiheuta enää lisämuutoksia puuhun.



Kuva 11 Esimerkki BuildHeap-metodin toiminnasta (Cormen ym. 1998, 146)

BuildHeap-metodin toimintaa Cormen ym. (Cormen ym., 1998, 146) selostavat kuvan 11 mukaisesti. Kohdassa a) on 10-solmuinen taulukko A ja binääripuu joka sitä esittää. Kuvassa näkyy silmukan indeksi i osoittaa solmuun 5 ennen Heapify(A, i) –kutsua. Kohdassa b) seurauksena oleva tietorakenne. Silmukan indeksi i osoittaa solmuun 4. Kohdissa c) – e) BuildHeap-metodin for-silmukan iteraatiot. Kohta f) lopputuloksena keko.

6.7 Muut algoritmit

Binääripuun rakentaminen ja sen pienimmän alkion etsiminen (liitteenä BinaariPuu.java), mallia otettu Todd Ebertin sivulta (Ebert 2010). Kuvassa 6 on esimerkki demonstraatio-puun käytöstä.

7 JOHTOPÄÄTÖKSET

Ohjelmakirjastot ovat toisaalta sekä tietokoneiden historiasta kumpuava että ajankohtainen aihe, ja niitä tarvitaan väistämättä nykyisissäkin ohjelmistoprojekteissa. Aiheen rajaus puurakenteisiin valittavien algoritmien suhteen oli toimiva, mutta tarkentui työn kuluessa binääripuurakenteisiin. Kaiken kaikkiaan tuntui ohjelmointityö mielekkäältä.

Tietoa ohjelmakirjastoista löytyy melko helposti Internetistä, mutta paperille painettua materiaalia oli vaikeampi löytää. Joitakin perusteoksia kuitenkin saattoi hyvin hyödyntää. Osa lähdemateriaalista oli myös hyvin vanhaa, mikä rajasi niiden luotettavuutta lähteenä.

Java ohjelmointikielenä soveltui mielestäni hyvin tällaiseen projektiin, vaikkei sen kaikkein vahvimpia mekanismeja, kuten templateja tai keskeytyksiä, työssä käytettykään. Algoritmien pituuden suhteen osoittautui, että työn puitteissa ei ollut mahdollista paneutua kovin monimutkaisiin algoritmeihin, lähinnä niiden testaamisen vaatima aika olisi kasvanut liian suureksi. Java on tällaiseen projektiin hyvä myös sen vuoksi, että se on alustariippumaton, ja toisaalta sitä käytetään melko yleisesti nimenomaan opiskelutyössä. Javassa on myös mahdollisuus rakentaa ns. paketteja (package-määre), jolla voi luoda ohjelmakirjaston. Toisaalta pidin hyvänä ominaisuutena mahdollisuutta Javaa käytettäessä käyttää javadoc-kommentointia, joka mahdollistaa järjestelmällisen kommentoinnin.

Luku ohjelmakirjastojen koostamisesta vesiputousmallin avulla itse kehitettynä lisäsi työn pohdinnallisuutta ja koin sen kirjoittamisen kiinnostavaksi.

Kaiken kaikkiaan työtä oli mukava tehdä, koska se ei juuttunut missään vaiheessa paikalleen, ja jos siltä tuntui, niin saattoi konsultoida ohjaajaansa. Aloitin työn työstämisen riittävän aikaisessa vaiheessa, joten työtä oli mukava tehdä, kun ei ollut kova kiire aikataulun suhteen. Työ tuli tehtyä ikään kuin oikeassa järjestyksessä siinä mielessä, että kirjoitin ensin teorian ja ryhdyin vasta sitten empirian työstämiseen.

LÄHTEET

Blume M. 1997. Hierarchical Modularity and Intermodule Optimization. Viitattu 13.7.2010. http://grosskurth.ca/bib/1997/blume.pdf.

Charbonneau L. 2001. Quality Management Initiative Overview. Viitattu 18.9.2010. gnu.cs.pu.edu.tw/software/gnue/project/docs/quality.ps.

Cormen H. C.; Leiserson C. E. & Rivest R. L. 1998. Introduction to Algorithms. Twentieth printing.

Deller L. & Heiser G. 1999. Linking Programs in a Single Address Space. Viitattu 13.10.2010. http://www.usenix.org/event/usenix99/full_papers/deller/deller.pdf.

Demmel J.; Dongarra J.; Eijkhout V.; Fuentes E.; Petitet A.; Vuduc R.; Whaley R. C. & Yelick K. 2004. Self Adapting Linear Algebra Algorithms and Software. Viitattu 14.7.2010. http://bebop.cs.berkeley.edu/pubs/ieee_sans.pdf.

Dubois P. F. 2001. The Inside Story on Shared Libraries and Dynamic Loading. Viitattu 13.9.2010. cseweb.ucsd.edu/~ricko/CSE131/the%20inside%20story%20on%20shared%20libraries%20and%20dynamic%20loading.pdf.

Ebert T. 2010. Todd Ebert's Homepage. Viitattu 1.11.2010. http://www.cecs.csulb.edu/~ebert/.

Encyclopedia.com 2004. Viitattu 22.1.2011. http://www.encyclopedia.com/doc/1O11-programlibrary.html.

Griss L. 1993. Software reuse: from library to factory. IBM systems journal, 32(4): 548-566.

Gunasekaran M. K. 2003. Component-based application development using a Mixed-Language Programming (MLP) approach. Viitattu 29.7.2010. http://www.aoe.vt.edu/~brown/VTShipDesign/DesignforAffordability/3_MuraliKrishnanThesis.pdf

FOLDOC 1998. Free On-Line Dictionary Of Computing. Viitattu 13.7.2010. http://foldoc.org/.

Jacobson I.; Griss M. & Jonsson P. 1997. Software reuse. Architechture, Process and Organization for Business Success. Addison-Wesley Professional.

Jalender B.; Govardhan A. & Premchand P. 2010. A Pragmatic Approach to Software Reuse. Viitattu 14.10.2010. http://www.jatit.org/volumes/research-papers/Vol14No2/3Vol14No2.pdf.

Kernebeck U. 1997. Component libraries for software re-use. Microprocessors and Microsystems Volume 21. Number 1. July 1997.

Kim W. 2005. On Issues with Component-Based Software Reuse. Viitattu 11.9.2010. http://www.jot.fm/issues/issue_2005_09/column5/column5.pdf.

Kokkarinen I. & Ala-Mutka K. 2002. Tietorakenteet ja algoritmit. 2. uudistettu painos. Talentum Media Oy.

Koskimies K. & Mikkonen T. 2005. Ohjelmistoarkkitehtuurit. Talentum Media Oy.

Kronsjö L. 1987. Algorithms, Their Complexity and Efficiency. Second Edition. John Wiley & Sons.

Levine J. R. 1999. Linkers & Loaders. Academic press. Saatavana http://download.ourdev.cn/bbs_upload647900/files_32/ourdev_574025.pdf.

Peisa K. 2006. Tietorakenteet ja algoritmit. Viitattu 13.7.2010. ta.ramk.fi/~kari.peisa/Opetus/TR_Alg/Osa4_Puut.pdf.

Penton R. 2001. Trees Part 2: Binary Trees. Viitattu 25.11.2010. http://www.gamedev.net/reference/articles/article1433.asp.

Plauger P. J.; Stepanov A. A.; Lee M. & Musser R. D. 2001. The C++ Standard Template Library. Prentice Hall.

Prieto-Díaz R. 1993. Status report: software reusability. Viitattu 11.9.2010. https://users.cs.jmu.edu/prietorx/Public/publications/StatusRepSoftReus.pdf.

Richter J. 2002. Inside .NET-ohjelmointi. IT-Press.

Sametinger J. 1997. Software Engineering with Reusable Components. Viitattu 13.7.2010. http://www.swe.uni-linz.ac.at/publications/pdf/TR-SE-97.04.pdf.

Smaragdakis Y. Layered Development with (Unix) Dynamic Libraries. Viitattu 16.7.2010. http://www.cs.umass.edu/~yannis/dynamic.pdf.

Smith J. N. 2009. Techniques in Active and Generic Software Libraries. Viitattu 11.9.2010. http://students.cs.tamu.edu/thechao/jns-thesis.pdf.

Szyperski C.; Gruntz S. & Murer S., 2002, Component software: beyond object-oriented programming. Addison-Wesley. Second Edition.

Teknillinen korkeakoulu. Prioriteettijonot ja keko. Viitattu 29.1.2011. http://www.cs.hut.fi/Opinnot/T-106.1220/kekotutoriaali/.

Vaucher S. & Sahraoui H. 2010. Do Software Libraries Evolve Differently than Applications? An Empirical Investigation. Viitattu 14.10.2010. http://www-etud.iro.umontreal.ca/~vauchers/articles/LCSD2007.pdf.

Whatis.com. 2005. Viitattu 22.1.2011. http://whatis.techtarget.com/definition/0,,sid9_gci860265,00.html.

Wheeler D. A. 2003. Program Library HOWTO. Viitattu 13.9.2010. tldp.org/HOWTO/pdf/Program-Library-HOWTO.pdf.

Ohjelmalistaukset

HeapSort.java

```
package esimerkkikirjasto;

/**
 * Lajittelee taulukon nousevaan järjestykseen käyttäen
 * keko-tietorakennetta
 * @author Antti Suutarinen
 */
class HeapSort
{
         private int heapsize;

         /**
          * Järjestää parametrina annetun taulukon
          * nousevaan järjestykseen
          * @param A lajiteltava taulukko
          */
         void Heapsort(int[] A)
         {
                  BuildHeap(A);

                  for(int i=length(A); i>=2; i--)
                  {
                           exchange(A, 1, i);
                           heapsize--;
                           Heapify(A, 1);
                  }
         }

         /**
          * Rakentaa keon
          * @param A taulukko, jossa käsiteltävät arvot
          */
         void BuildHeap(int[] A)
         {
                  heapsize=A.length-1;

                  //seuraavassa floor-toiminto
                  for(int i=heapsize/2;i>=1;i--)
                           Heapify(A, i);
         }

         /**
          * Järjestää osan puusta kohti keko-ominaisuutta
          * @param A taulukko, jossa järjestettävä puu
          * @param i kohta taulukossa eli puussa jota
```

```
         * operoidaan
         */
        void Heapify(int[] A, int i)
        {
                 int le=2*i;
                 int ri=2*i+1;
                 int largest;

                 if(le<=heapsize && A[le]>A[i])
                          largest=le;
                 else
                          largest=i;
                 if(ri<=heapsize && A[ri]>A[largest])
                          largest=ri;
                 if(largest!=i)
                 {
                          exchange(A, i, largest);
                          Heapify(A, largest);
                 }
        }

        /**
         * Vaihtaa taulukon kaksi arvoa keskenään
         * @param A taulukko, jossa vaihdettavat arvot
         * @param i osoittaa ensimmäisen vaihdettavan
         * kohdan taulukosta
         * @param j osoittaa toisen vaihdettavan kohdan
         * taulukosta
         */
        private static void exchange(int[] A, int i,
                 int j)
        {
                 int temp=A[i];
                 A[i]=A[j];
                 A[j]=temp;
        }

        /**
         * Demonstroi HeapSort-algortmin toimintaa
         */
        public static void main(String[] args)
        {
                 int t[]={0,4,65,1,2,4,52,4,5,3,12};
                 HeapSort h=new HeapSort();
                 h.Heapsort(t);
                 for(int i=0;i<t.length;i++)
                          System.out.print(t[i] + ", ");
        }
}
```

PuunÄäriarvot.java

```
package esimerkkikirjasto;

/**
 * Luokka, jota käytetään yhden solmun sisältämän
 * tiedon käsittelemiseen.
 * @author Antti Suutarinen
 */
class Solmu
{
	private Solmu vasen;
	private Solmu oikea;
	private int arvo;

	/**
	 * Konstruktori
	 * @param arvo solmun arvoksi asetettava arvo
	 */
	Solmu(int arvo)
	{
		this.arvo=arvo;
	}

	/**
	 * Tällä voi asettaa solmun vasemmanpuoleiseksi
	 * lapseksi solmun
	 * @param vasen asetettava solmu
	 */
	void asetaVasen(Solmu vasen)
	{
		this.vasen=vasen;
	}

	/**
	 * Tällä voi asettaa solmun oikeanpuoleiseksi
	 * lapseksi solmun
	 * @param oikea asetettava solmu
	 */
	void asetaOikea(Solmu oikea)
	{
		this.oikea=oikea;
	}

	/**
	 * Etsii puun sisältämän minimiarvon
```

```
         * @return puun minimiarvo
         */
        int minimi()
        {
                int x=-1;
                Solmu s=vasen;

                while(s!=null)
                {
                        x=s.arvo;
                        s=s.vasen;
                }
                return x;
        }

        /**
         * Etsii puun maksimiarvon
         * @return puun maksimiarvo
         */
        int maksimi()
        {
                int x=-1;
                Solmu s=oikea;

                while(s!=null)
                {
                        x=s.arvo;
                        s=s.oikea;
                }
                return x;
        }
}

/**
 * Demonstroi Solmu-luokan käyttöä
 */
class PuunÄäriarvot
{
        public static void main(String[] args)
        {
                Solmu n1=new Solmu(8);
                Solmu n2=new Solmu(10);
                Solmu n3=new Solmu(9);
                Solmu n4=new Solmu(12);
                Solmu n5=new Solmu(11);
                Solmu n6=new Solmu(4);
                Solmu n7=new Solmu(7);
                n4.asetaVasen(n5);
                n2.asetaOikea(n4);
```

```
                n2.asetaVasen(n3);
                n6.asetaOikea(n7);
                n1.asetaVasen(n6);
                n1.asetaOikea(n2);
                int s=n1.minimi();
                System.out.println("Minimi: " + s);
                s=n1.maksimi();
                System.out.println("Maksimi: " + s);
        }
}
```

ProsessoiPuu.java

```
package esimerkkikirjasto;

/**
 * Luokka, jota käytetään yhden solmun sisältämän
 * tiedon käsittelemiseen.
 * @author Antti Suutarinen
 */
class Käsittelijä //voidaan periyttää
{
        /**
         * Metodi joka käsittelee Solmun s
         * omalla tavallaan.
         * @param s käsiteltävä solmu
         */
        protected void käsittele(Solmu s)
        {
                System.out.print(s.kerroArvo() + ", ");
        }
}

/**
 * Ise määritetty versio solmun arvon käsittelijästä
 */
class OmaKäsittelijä extends Käsittelijä
{
        /**
         * Oma käsittelijä
         * @param s käsiteltävä solmu
         */
        protected void käsittele(Solmu s)
        {
                System.out.print((s.kerroArvo() + 1)
                        + ", ");
        }
```

```
}

/**
 * Esittää yhtä binääripuun solmua
 * @author Antti Suutarinen
 */
class Solmu
{
	private Solmu vasen;
	private Solmu oikea;
	private int arvo;
	private Käsittelijä k;

	/**
	 * Konstruktori
	 * @param k käsittelijä solmun
	 * sisällön käsittelemistä varten
	 */
	Solmu(Käsittelijä k)
	{
		this.k=k;
	}

	/**
	 * Asetetaan solmun vasen lapsi
	 * @param n asetettava solmu
	 */
	void asetaVasen(Solmu n)
	{
		vasen=n;
	}

	/**
	 * Asetetaan solmun oikea lapsi
	 * @param n asetettava solmu
	 */
	void asetaOikea(Solmu n)
	{
		oikea=n;
	}

	/**
	 * Palauttaa solmun oikean lapsen
	 * @return palautettava solmu
	 */
	Solmu kerroOikea()
	{
		return oikea;
	}
```

```
        /**
         * Palauttaa solmun vasemman lapsen
         * @return palautettava solmu
         */
        Solmu kerroVasen()
        {
                 return vasen;
        }

        /**
         * Asettaa solmun sisältämän arvon
         * @param arvo asetettava arvo
         */
        void asetaArvo(int arvo)
        {
                 this.arvo=arvo;
        }

        /**
         * Palauttaa solmun sisältämän arvon
         * @return palautettava arvo
         */
        int kerroArvo()
        {
                 return arvo;
        }

        /**
         * Käsittelee solmun sille asetetun
         * käsittelijän perusteella
         */
        private void käsittele()
        {
                 k.käsittele(this);
        }

        /**
         * Käy läpi puun ja käsittelee jokaisen
         * solmun sille asetetun metodin mukaisesti
         * @param p läpikäytävän puun juurisolmu
         */
        static void läpikäy(Solmu p)
        {
                 if(p==null)
                          return;
                 p.k.käsittele(p);
                 läpikäy(p.kerroOikea());
                 läpikäy(p.kerroVasen());
```

```
        }
}

/**
 * Demonstroi Solmu-luokan käyttöä
 */
class ProsessoiPuu
{
        public static void main(String[] args)
        {
                Solmu n1=new Solmu(k); n1.asetaArvo(5);
                Solmu n2=new Solmu(k); n2.asetaArvo(6);
                Solmu n3=new Solmu(k); n3.asetaArvo(1);
                Solmu n4=new Solmu(k); n4.asetaArvo(15);
                Solmu n5=new Solmu(k); n5.asetaArvo(7);
                Solmu n6=new Solmu(k); n6.asetaArvo(8);
                n4.asetaVasen(n5);
                n4.asetaOikea(n6);
                n3.asetaOikea(n4);
                n1.asetaVasen(n3);
                n1.asetaOikea(n2);
                System.out.println(
        "Puu prosessoidaan:");
                Solmu.läpikäy(n1);
        }
}
```

BinaariPuu.java

```
package esimerkkikirjasto;

/**
 * Sisältää binääripuun yhden solmun tiedot sekä
 * toiminnallisuuden solmuista muodostetun puun
 * pienimmän arvon etsimiseen sekä metodin yhden solmun
 * lisäämiseen puuhun
 * @author Antti Suutarinen
 */
class Solmu
{
        private int arvo;
        private Solmu vasen;
        private Solmu oikea;
        private boolean tyhjä;

        /**
         * Konstruktori, jolla luodaan tyhjä solmu
         * @param arvo Arvo jonka solmu tulee sisältämään
```

```
         */
        public Solmu(int arvo)
        {
                 this.arvo=arvo;
                 vasen=new Solmu();
                 oikea=new Solmu();
                 tyhjä=false;
        }

        /**
         * Konstruktori, jolla luodaan uusi solmu
         * ja asetetaan samalla solmun sisältävä arvo
         */
        public Solmu()
        {
                 arvo=0;
                 vasen=null;
                 oikea=null;
                 tyhjä=true;
        }

        /**
         * Alustusmetodi, jolla asetetaan myös solun arvo
         * @param arvo Arvo jonka solmu tulee sisältämään
         */
        void alusta(int arvo)
        {
                 this.arvo=arvo;
                 vasen=new Solmu();
                 oikea=new Solmu();
                 tyhjä=false;
        }

        /**
         * Palauttaa onko solmu tyhjä
         * @return totuusarvo
         */
        public boolean tyhjä()
        {
                 return this.tyhjä;
        }

        /**
         * Kertoo solmun sisältämän arvon
         * @return Solmun sisältämä arvo
         */
        public int kerroArvo()
        {
                 return arvo;
```

```
        }

        /**
         * Palauttaa solmun vasemman lapsen
         * @return Vasen solmu
         */
        public Solmu vasenLapsi()
        {
                return vasen;
        }

        /**
         * Palauttaa solmun oikean lapsen
         * @return Oikea solmu
         */
        public Solmu oikeaLapsi()
        {
                return oikea;
        }

        /**
         * Palauttaa puun pienimmän arvon sisältämän
         * solmun
         * @param puun juurisolmu
         */
        public static Solmu etsiPienin(Solmu n)
        {
                if(n.tyhjä()==true)
                        return new Solmu();
                if(n.vasenLapsi().tyhjä()==true)
                        return n;
                return(etsiPienin(n.vasenLapsi()));
        }

        /**
         * Solmun lisääminen puuhun oikeaan paikkaan
         * @param x lisättävän solmun sisältämä arvo
         * @param n puun juurisolmu
         */
        public static void insert(int x, Solmu n)
        {
                if(n.tyhjä()==true)
                {
                        //System.out.print(x + ", ");
                        n.alusta(x);

                        return;
                }
                if(x==n.kerroArvo())
```

```
                        return;
                if(x<n.kerroArvo())
                        insert(x, n.vasenLapsi());
                //x>n.kerroArvo()
                        else insert(x, n.oikeaLapsi());
        }
}

/**
 * Tämä demonstroi Solmu-luokan käyttöä
 */
public class BinaariPuu
{
        public static void main(String[] args)
        {
                Solmu puu=new Solmu();
                Solmu.insert(5, puu);
                Solmu.insert(1, puu);
                Solmu.insert(15, puu);
                Solmu.insert(7, puu);
                Solmu.insert(8, puu);
                Solmu.insert(6, puu);
                int p=
        (Solmu.etsiPienin(puu)).kerroArvo();
                System.out.println("Pienin: " + p);
        }

}
```

EtsiSyvyys.java

```
package esimerkkikirjasto;

/**
 * Esittää binääripuun solmua, jolla on kaksi lapsisolmua.
 * @author Antti Suutarinen
 */
class Solmu
{
        private Solmu vasen;
        private Solmu oikea;

        /**
         * Laskee ja palauttaa alipuunansa, tai, jos
         * solmu on juurisolmu, koko puun syvyyden.
         * @return alipuunsa syvyyden
         */
```

```
         int syvyys()
         {
                  int vasenAlipuu=-1;
                  int oikeaAlipuu=-1;

                  if(vasen!=null)
                           vasenAlipuu=vasen.syvyys();
                  if(oikea!=null)
                           oikeaAlipuu=oikea.syvyys();
                  if(vasenAlipuu>oikeaAlipuu)
                           return vasenAlipuu+1;
                  return oikeaAlipuu+1;
         }

         /**
          * Tällä voi asettaa solmun vasemmanpuoleiseksi
          * lapseksi solmun
          * @param vasen asetettava solmu
          */
         void asetaVasen(Solmu vasen)
         {
                  this.vasen=vasen;
         }

         /**
          * Tällä voi asettaa solmun oikeanpuoleiseksi
          * lapseksi solmun
          * @param oikea asetettava solmu
          */      void asetaOikea(Solmu oikea)
         {
                  this.oikea=oikea;
         }
}

/**
 * Demonstroi Solmu-luokan käyttöä
 */
class EtsiSyvyys
{
         public static void main(String[] args)
         {
                  Solmu n1=new Solmu();
                  Solmu n2=new Solmu();
                  Solmu n3=new Solmu();
                  Solmu n4=new Solmu();
                  Solmu n5=new Solmu();
                  Solmu n6=new Solmu();
                  n4.asetaVasen(n5);
                  n4.asetaOikea(n6);
```

```
                    n3.asetaOikea(n4);
                    n1.asetaVasen(n3);
                    n1.asetaOikea(n2);
                    int s=n1.syvyys();
                    System.out.println("Syvyys: " + s);
          }
}
```

```
Successor.java

package esimerkkikirjasto;

class Solmu
{
          private Solmu vasen;
          private Solmu oikea;
          private Solmu isä;
          private int arvo;

          /**
           * Konstruktori
           * @param arvo arvo jonka solmu tulee sisältämään
           */
          Solmu(int arvo)
          {
                    this.arvo=arvo;
                    vasen=null;
                    oikea=null;
                    isä=null;
          }

          /**
           * Tällä voi asettaa solmun vasemmanpuoleiseksi
           * lapseksi solmun
           * @param vasen asetettava solmu
           */
          void asetaVasen(Solmu vasen)
          {
                    this.vasen=vasen;
                    vasen.isä=this;
          }

          /**
           * Tällä voi asettaa solmun oikeanpuoleiseksi
           * lapseksi solmun
           * @param oikea asetettava solmu
           */
          void asetaOikea(Solmu oikea)
```

```
        {
                this.oikea=oikea;
                oikea.isä=this;
        }

        /**
         * Etsii puun pienimmän solmun
         * @param x solmu joka on etsittävän puun juuri
         */
        static Solmu minimi(Solmu x)
        {
                while(x.vasen!=null)
                        x=x.vasen;
                return x;
        }

        /**
         * Etsii solmusta seuraavaksi suurimman puussa
         * @param x solmu, jonka seuraaja etsitään
         * @return palauttaa solmun, jossa seuraavaksi
         * suurin,
         * jos tehtäväksi annettiin etsiä suurimman arvon
         * seuraaja palauttaa null:in
         */
        static Solmu successor(Solmu x)
        {
                if(x.oikea!=null)
                        return Solmu.minimi(x.oikea);
                Solmu y=x.isä;
                while(y!=null && x==y.oikea)
                {
                        x=y;
                        y=y.isä;
                }

                return y;
        }

        /**
         * Palauttaa solmun arvon
         */
        int arvo()
        {
                return arvo;
        }
}

/* Testaa Solmu-luokan toteutusta puun solmun
 * edeltäjän etsimisestä
```

```
 * @author Antti Suutarinen
 */
class Successor
{
         public static void main(String[] args)
         {
                  Solmu n1=new Solmu(5);
                  Solmu n2=new Solmu(3);
                  Solmu n3=new Solmu(4);
                  Solmu n4=new Solmu(9);
                  Solmu n5=new Solmu(1);
                  Solmu n6=new Solmu(7);
                  Solmu n7=new Solmu(8);
                  n1.asetaVasen(n2);
                  n1.asetaOikea(n4);
                  n2.asetaVasen(n5);
                  n2.asetaOikea(n3);
                  n4.asetaVasen(n6);
                  n6.asetaVasen(n7);

                  Solmu successor=Solmu.successor(n3);
                  if(successor==null)
                           System.out.println(
                           "Suurin arvo!");
                  else
                           System.out.println(
                           "Successor: "
                           + successor.arvo());
         }
}
```