UML-luokkakaavioiden automaattinen generointi

Timo Helander

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Erkki Mäkinen
Syyskuu 2007

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos

Tietojenkäsittelyoppi
Timo Helander: UML-luokkakaavioiden automaattinen generointi
Pro gradu -tutkielma, 50 sivua

Syyskuu 2007

Tässä tutkielmassa käsitellään UML-luokkakaavioiden automaattiseen generointiin liittyviä ongelmia. Tutkielma esittelee lyhyesti UML-notaation luokkakaavioille, jonka jälkeen siirtyy tarkastelemaan kaavion piirtoa ensiksi heuristiikkojen ja estetiikkojen osalta, ja sen jälkeen esittelee tämän ongelman ratkaisevia algoritmeja.

Avainsanat ja -sanonnat: graafinpiirto, ohjelmistonkehitys, UML

Sisällys

1. Johdanto

Nykyaikaisessa ohjelmistosuunnittelussa käytetään UML-tyyppisiä kaavioita kuvaamaan komponentteja ja niiden välisiä relaatioita. UML tulee sanoista Unified Modeling Language ja tämä kieli standardoitiin Object Management Groupin toimesta vuonna 1997. UML ei niinkään ota kantaa, kuinka ohjelmisto toimii vaan kuinka se on rakennettu. UML-kieltä ei tulisi käyttää mallintamaan itse toteutusta.

Graafinpiirtoa pidettiin ennen melkein täysin teoreettisena matematiikan haarana kunnes sitä alettiin käyttää esimerkiksi piirilevyjen ja ohjelmistojen suunnittelussa. Graafinpiirto ja UML muodostavat yhdessä erittäin hyvät työkalut ohjelmistojen visuaaliseen mallintamiseen. Tietenkin on olemassa muitakin mahdollisia mallinnustapoja, mutta tällä hetkellä tämä on kaikkein suosituin [Eiglsperger et al., 2003]. Viime aikoina on graafinpiirtoalgoritmit nousseet merkittävään rooliin ohjelmistonkehityksessä.

Graafien näkökulmasta UML-tyyppisessä ohjelmistonkehityksessä tarvitaan suunnattua ortogonaalista graafia, jolla voidaan osoittaa komponenttien välisiä relaatioita. Nämä ovat yleensä myös hierarkkisia ja koostuvat useasti eri osagraafista. Esimerkiksi yksi käytetty yhdistelmä on hierarkiapuu ja sen alusgraafi [Noack and Lewerentz].

Graafinpiirron tavoitteena on graafin osien esteettinen sijoittelu tasolle. Ei pelkästään riitä se, että eri komponentit voidaan erotella selkeästi, vaan myös niiden väliset relaatioviivat, joita käytetään esimerkiksi UML:ssä, täytyy piirtää niin, että pyritään minimoimaan entiteettien väliset leikkaukset ja relaatiota kuvaavien särmien taivutukset. Lopuksi pitää eri ohjelmistokomponentit ryhmitellä järkeviksi kokonaisuuksiksi, eli ei ole visuaalisen informaation kannalta järkevää erotella esimerkiksi samaan luokkaan kuuluvia metodikutsuja kauaksi toisistaan graafissa. Yksi graafinpiirron kannalta haastavia ongelmia onkin graafin klusterointi.

Tulevaisuuden ohjelmistonkehityksessä tullaan tarvitsemaan enenevimmässä määrin ohjelmiston takaisinmallinnusta. Myös tässä graafinpiirto nousee merkitsevään rooliin. Tätä aihetta käsitellään hieman tutkielman loppupuolella.

Tutkielman alkuosassa esitellään UML yleisesti ja erityisesti luokkakaavion ominaisuuksia. Seuraavaksi paneudutaan graafinpiirron alkeisiin heuristiikkojen avulla, josta sujuvasti siirrytään hierarkkisiin graafeihin. Hierarkkisten graafien jälkeen tutustumme Noackin ja Lewerentzin [2005] artikkeliin, joka kertoo meille tavan jakaa graafinpiirto kolmeen eri dimensioon takaamaan onnistuneen ohjelmistonotaation. Tästä innostuneina perehdymme Purchasen ja muiden [2001] empiiriseen tutkimukseen ja estetiikkakriteereihin, kunnes pääsemme tositoimiin aluksi katsastamalla Eiglspergerin ja muiden [2003] artikkelissa esitettyä algoritmia graafin piirtämiseen. Heti perään Gudenbergin [2006] ja muiden geneettinen algoritmi herättää

mielenkiintoa, josta päästäänkin sitten ohjelmistojen takaisinmallinnukseen ennen yhteenvetoa.

2. Ohjelmistojen mallinnus ja UML

UML-mallinnus kehitettiin helpottamaan ohjelmiston hallintaa. Kun ohjelmisto on hyvin mallinnettu, voidaan helposti löytää ohjelmointivirheet ja korjata ohjelmistoa, vaikka ohjelmiston tekijä ei enää talossa toimisikaan. Tällainen mallinnus erottelee ohjelmiston moduuleihin, joiden käsitteleminen on helppoa, kun ne on mallinnettu hyvin. Yksi merkittävimpiä etuja suurissa ohjelmistoissa on koodin uudelleenkäyttö, koska ohjelmisto voidaan jakaa kirjastoihin, joita voidaan lisätä aina tarvittaessa.

Ohjelmistojen mallinnuksella tarkoitetaan työtä, joka tehdään ennen kuin aletaan ohjelmiston ohjelmointityö. Ohjelmistosta luodaan malli, joka vastaa esimerkiksi pohjapiirustuksia rakennusalalla. Ohjelmistoprojekteilla on suuri epäonnistumistodennäköisyys, joten on selvää, että pyritään lisäämään onnistumistodennäköisyyttä jollain tavalla. Kun ohjelmistosta luodaan hyvin määritelty malli ennen ohjelmointityön aloittamista, pienennetään riskiä koko projektin epäonnistumiseen.

UML on kieli, joka on tehty ohjelmistojen mallinnusta varten, mutta sillä voidaan mallintaa myös muitakin rakenteita kuin vain ohjelmistoja. Mallinnuksella voidaan esittää eri sovelluksia, kuten laitteistoja, käyttöjärjestelmiä, ohjelmointikieliä ja verkkoja. Kuitenkin UML perustuu olio-ohjelmointikonseptiin, joissa käytetään luokka-operaatio-rakennetta. Tällaisia kieliä ovat esimerkiksi C++, C# ja Java.

UML 2.0 määrittelee kolmetoista eri kaaviotyyppiä, jotka voidaan jakaa kolmeen eri kategoriaan [OMG, 2007]:

- Rakennekaaviot (Structure Diagram): komponenttikaavio (Component Diagram), koostekaavio (Composite Structure Diagram), luokkakaavio (Class Diagram), oliokaavio (Object Diagram), pakkauskaavio (Package diagram) ja sijoittelukaavio (Deployment Diagram).
- Käyttäytymiskaaviot (Behavior Diagram): aktiviteettikaavio (Activity diagram), käyttötapauskaavio (Use case diagram) ja tilakaavio (State Machine diagram).
- Vuorovaikutuskaaviot (Interaction Diagram): ajoituskaavio (Timing Diagram), kokoava vuorovaikutuskaavio (Interaction Over Diagram), kommunikointikaavio (Communication Diagram) ja sekvenssikaavio (Sequence Diagram).

2.1. Luokkakaaviot

Luokkakaaviot ovat kaikkein suosituin mallinnustapa olioperustaisessa ohjelmistonkehityksessä niiden yksinkertaisen notaation ja monipuolisuuden ansiosta. Luokkakaaviot vastaavat yksi yhteen olio-ohjelmoitua sovellusta. Tämän takia on myös ohjelmistojen takaisinmallinnus sekä ohjelmistonkehitystyökaluista ohjelmistojen muodostaminen mahdollista.

Object Management Group [2007] kuvaa luokkakaavioita seuraavalla tavalla. Luokka kuvaa joukon objekteja, joilla on yhtenäiset piirteet, rajoitukset ja merkitykset. Luokan osat jakautuvat attribuutteihin ja operaatioihin. Luokan attribuutit esitetään ominaisuuksina, jotka luokka omistaa. Jotkut näistä attribuuteista voivat esittää binäärisen assosiaation ohjautumista.

Luokan tarkoituksena on muodostaa erottelu objektien välille ja määrittää ominaisuudet, jotka kuvaavat objektien rakennetta ja käyttäytymistä. Luokan omistamien objektien eli attribuuttien täytyy sisältää arvo. Tämä voi esimerkiksi olla tyyppi tai määrä. Kun objekti toteutetaan luokassa, niin jokaiselle attribuutille täytyy määrittää oletusarvo. Kun attribuuttia ei alusteta, niin annettu oletusarvo annetaan attribuutille, joka määrittelee objektin.

Luokan operaatiot voidaan kutsua antamalla erityinen joukko sijoituksia parametreina kyseiselle operaatiolle. Operaation kutsuminen voi vaihtaa luokan attribuuttien arvoja tai se voi myös palauttaa attribuutin arvon, joka operaatiolle on erityisesti määritelty. Operaation kutsuminen voi myös muuttaa jonkin toisen objektin attribuuttien arvoja suorasti tai epäsuorasti, lähtien objektista, josta operaatiota oli kutsuttu siten, että objektille on määritelty parametrina toinen objekti. Operaation kutsuminen voi myös aiheuttaa uuden objektin luomiseen tai tuhoamiseen.

Luokka ei kuitenkaan voi päästä käsiksi toisen luokan suljettuihin ominaisuuksiin, mikäli tämä ei ole luokan yliluokka. Kun luokkaa luodaan tai tuhotaan, niin ainakin toisella osapuolella täytyy olla pääsy luokan sisältöön.

2.2. Luokan notaatio

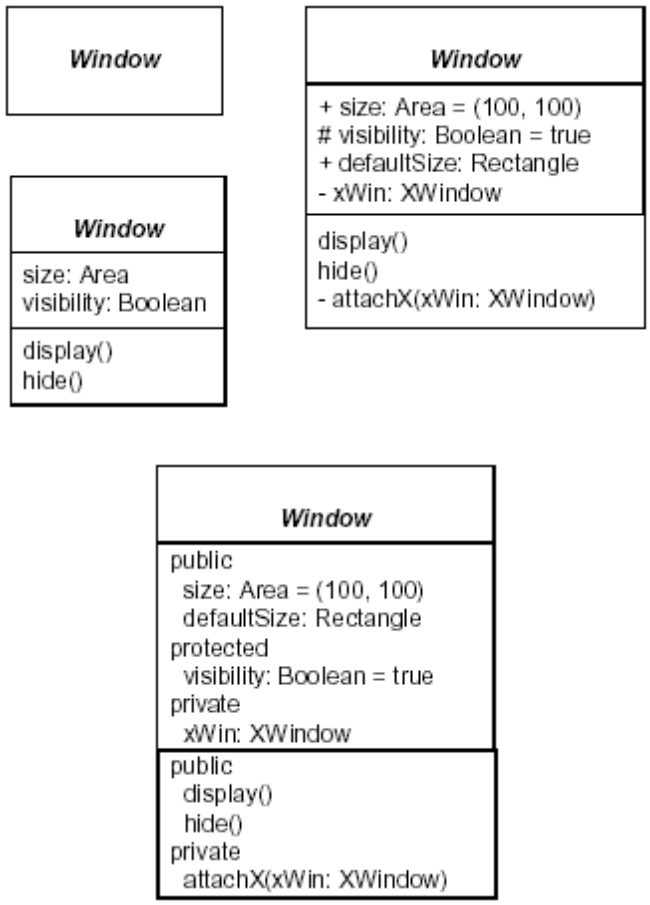
Luokka esitetään kaavioissa laatikkona, joka on nimetty samoin kuin luokka, jota se esittää. Laatikon sisältö on jaettu kolmeen eri lokeroon. Lokerossa luokan nimen alapuolella voidaan vaihtoehtoisesti esittää luokan muuttujat, joista näytetään ainakin muuttujan nimi, mutta voidaan esittää myös sen tyyppi. Luokan operaatiot esitetään alimmassa lokerossa. Myös operaatioista näytetään ainakin operaation nimi, mutta voidaan esittää myös operaation attribuutit ja palautusarvo. Myös voidaan lisätä lokeroita, mikäli luokasta täytyy esittää rajoitteita tai jaettuja ominaisuuksia. Luokan operaatiot ja attribuutit voidaan jakaa näkyvyytensä mukaan erillisiin osiin.

Object Management Groupin antamia tyyliohjeita luokkakaavion piirtämiseen [OMG, 2007]:

- Luokan nimi keskitetään ja vahvennetaan.
- Luokan nimet kirjoitetaan isolla.
- Attribuuttien ja operaatioiden nimet kirjoitetaan ilman vahvennusta ja ne tasataan vasemmalle.
- Attribuuttien ja operaatioiden nimet kirjoitetaan pienellä.

- Kursivoidaan abstraktin luokan nimi.
- Esitetään kaikki attribuutit ja operaatiot, kun sille on tarvetta ja jätetään niitä esittämättä yhteyksissä, joissa vain viittataan luokkaan.

Tyyliohjeiden mukaisia piirroksia on esitetty kuvassa 2.1.



Kuva 2.1. Esimerkkejä luokkapiirroksista. Alimmassa kuvassa luokan objektit on jaettu näkyvyytensä mukaan.

2.3. Assosiaatiot luokkien välillä

Assosiaatio määrittelee joukon pareja, joiden arvot viittaavat tyypitettyihin objekteihin. Assosiaatio-objektia kutsutaan linkiksi. Assosiaatio määrittää suhteen merkityksen tyypitettyjen objektien välille. Assosiaatiolla on vähintään kaksi päätä, ja jokaiselle päätepisteelle määritellään assosiaation ominaisuus ja merkitys. Yhdellä tai useammalla assosiaation päätepisteellä voi olla sama tyyppi.

2.4. Assosiaationotaatio

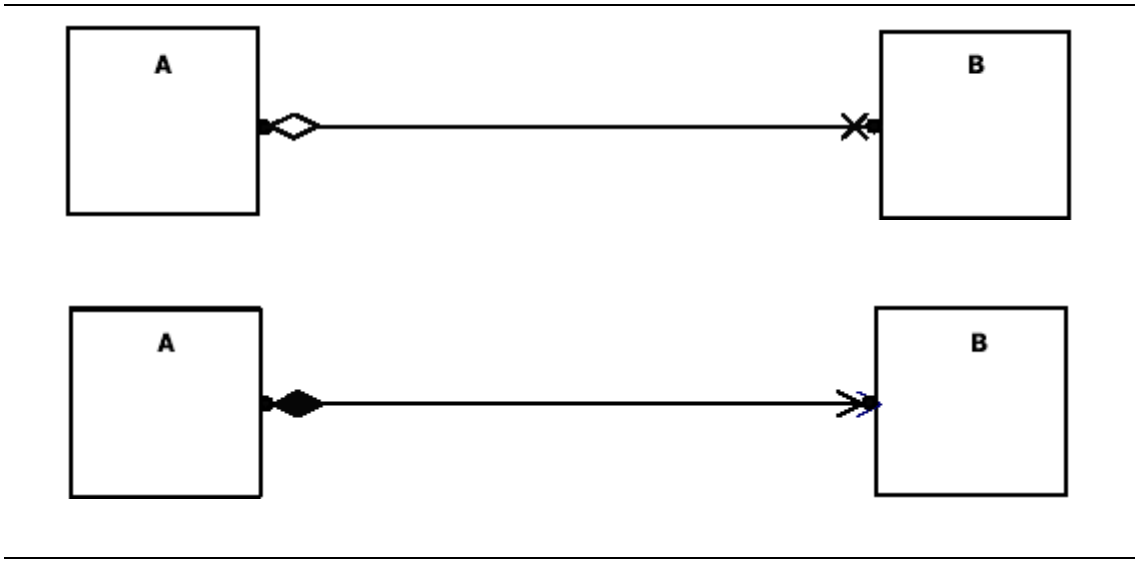
Tavallinen avoin nuolenpää assosiaation päässä tarkoittaa, että tämä pää on tavoitettavissa lähtöluokasta. Pieni x assosiaation päädyssä tarkoittaa taas sitä, että tämä pää ei ole tavoitettavissa lähtöluokasta.

Jos luokkia on useampia kuin kaksi samassa assosiaatiossa, niin luokista lähtevät assosiaatioviivat yhtyvät timantinmuotoisessa kuviossa.

Yhdistelmäassosiaation merkitseminen eroaa binääriassosiaatiosta niin, että assosiaation päähän piirretään avoin timantinmuotoinen merkki. Tämä on kuitenkin huomattavasti pienempi kuin timantti, joka kuvaa useamman luokan assosiaatiota. Yhdistelmäassosiaation ja binääriassosiaation erona on, että yhdistelmäassosiaatiossa olevat luokat ovat riippuvaisia toisistaan. Siinä assosiaation päässä, jossa timantti on, oleva luokka luo tai omistaa assosiaation vastakkaisessa päässä olevan luokan. Mikäli timantti ei ole avoin, tarkoittaa se, että omistajaluokka on vastuussa omistettavan luokan olemassaolosta ja käytetyistä resursseista.

Assosiaatiopäätyjen omistussuhteet voidaan esittää pienellä väritetyllä ympyrällä, jota kutsutaan pisteeksi. Piste piirretään assosiaatioviivan päähän siten, että sen tangenttina toimii omistussuhteessa olevan luokan särmä. Toisin sanoen, se piirretään kiinni luokkaa kuvaavaan suorakulmioon (ks. kuva 2.2). Pisteen halkaisijan tulee olla suurempi kuin viivan leveys, mutta se ei saa olla olla suurempi kuin puolet yhdistelmäassosiaation timantin koosta.

Kuva 2.2. Binääriassosiaatio ja sen omistusta kuvaava piste.



Kuva 2.3. Kaksi yhdistelmäassosiaatiota.

Kuvassa 2.3 on kaksi yhdistelmäassosiaatiota, joista ylemmässä A omistaa B:n, mutta A:lla ei ole pääsyä B:n assosiaatiopäädyn arvoihin. Alemmassa A:lla on pääsy B:n assosiaatioarvoihin ja täytetty neliö tarkoittaa, että koska A on luonut B:n, on sillä myös vastuu B:n olemassaolosta ja muistinkäytöstä.

3. Graafinpiirrosta

Ohjelmistollisessa graafinpiirrossa täytyy ottaa huomioon se, ettei ohjelman lukemalla graafilla ole vielä mitään tietoa siitä, miten se tulisi piirtää. Ohjelma lukee graafin siten, että sillä on tieto ainoastaan solmuista, särmistä ja niiden relaatiosuhteista. Tästä käytetään usein nimitystä abstrakti graafi.

Graafin esteettisyys on informaation lukemisen kannalta olennaista ja usein graafia piirrettäessä käy niin, että esteettisyysheuristiikat joutuvat keskenään ristiriitaan, joten graafinpiirto on algoritmisesti haastavaa. UML-tyyppisessä graafinpiirrossa käytetään ortogonaalista ja hierarkkista lähestymistapaa. Tämä luku perustuu Di Battistan, Eadesin, Tamassian ja Tollisin kirjaan *Graph Drawing* [1999].

3.1. Topologia, muoto ja metriikka

Tässä esitellään graafin piirtämistä niin, että särmät kulkevat kohtisuoraan tai poikittain toisiinsa nähden. Tämä ortogonaalisen piirroksen lähestymistapa karakterisoidaan kolmeen perusperiaatteeseen, jotka ovat ekvivalentteja keskenään:

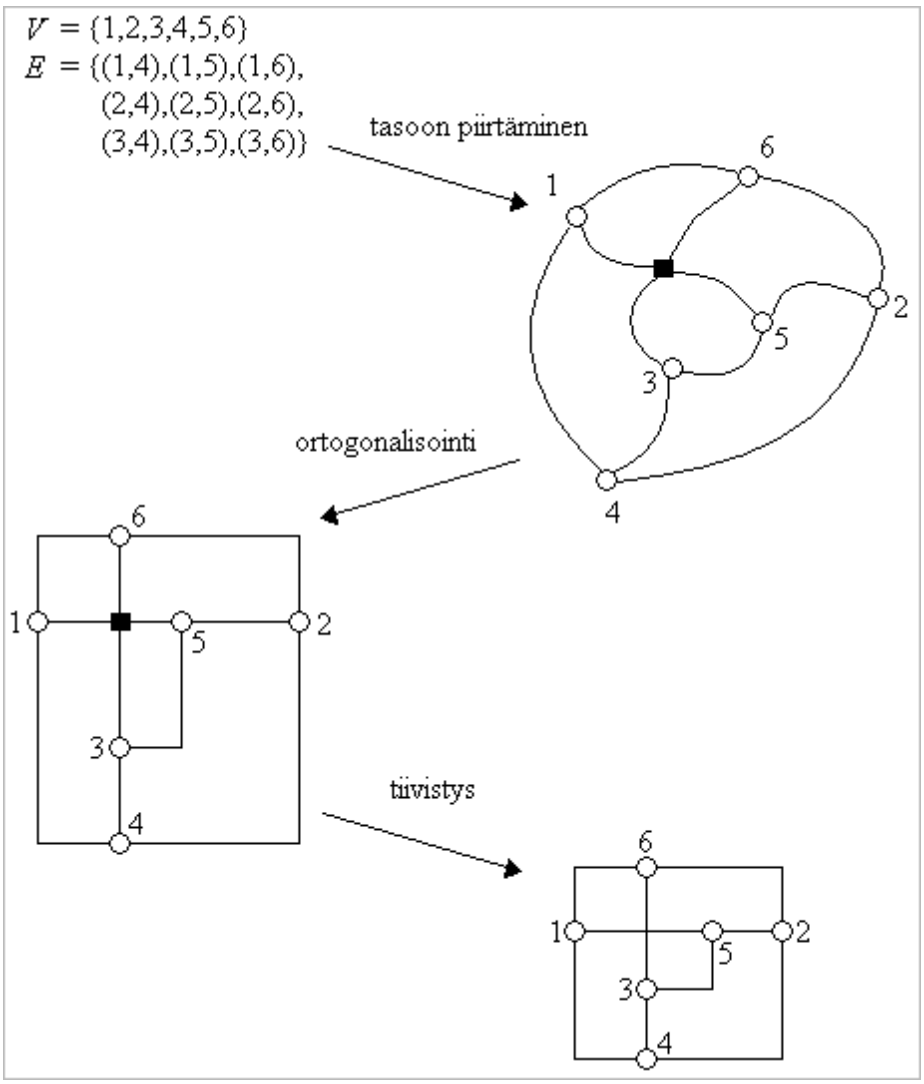
- *Topologisuus*: Kahdella ortogonaalisella piirroksella on sama topologia, jos toinen voidaan muodostaa toisesta jatkuvalla uudelleenmuodostamisella siten, että särmien järjestys ei muutu piirroksen pinnalla.
- *Muoto*: Kaksi ortogonaalista piirrosta ovat keskenään samanmuotoiset, jos niillä on sama topologisuus ja toinen voidaan muodostaa toisesta muuttamalla särmien pituutta muuttamatta niiden kulmaa toisiinsa nähden. Esimerkiksi voidaan piirrosta venyttää, niin pystysuuntaan kuin sivusuuntaankin.
- *Metriikka*: Kahdella ortogonaalisella piirroksella on sama metriikka, jos ne ovat kongruentteja keskenään kääntämiseen ja/tai kiertoon asti.

Huomautettakoon, että yllämainitut ominaisuudet eivät päde pelkästään ortogonaalisiin piirroksiin, vaan myös moniviivapiirroksiin (polyline drawings). Topologisuudesta, muodosta ja metriikasta saadaan askellettu piirrosheuristiikka ortogonaaliselle graafille:

- Ensimmäinen askel on graafin tasoon piirtäminen. Tässä askeleessa pyritään välttämään särmien leikkaus. Tasoon piirtämistä on tutkittu paljon kirjallisuudessa. Tämä askel tapahtuu seuraavasti: ensiksi erotetaan graafista sen osajoukko, joka voidaan suoraan piirtää tasoon ja sitten lisätään särmiä siten, että pyritään estämään särmien väliset leikkaukset. Jos tulee särmien välisiä leikkauksia, niin graafiin lisätään solmu, joka estää leikkaukset ja samalla esittää leikkauskohtaa.

- Kun graafi on saatettu topologiansa suhteen tasoon, niin seuraa askel, jossa saatetaan graafi ortogonaaliseksi. Tässä askeleessa graafin solmuilla ei ole koordinaatteja, mutta graafin särmille annetaan lista kulmia, jotka kertovat särmän suunnan lopullisessa piirroksessa.
- Kun graafi on saatettu ortogonaaliseksi, niin tiivistysaskel päättää solmujen lopulliset koordinaatit ja särmien tekemät taitot. Yleensä tässä askeleessa pyritään saattamaan graafin piirros sellaiseksi, että se vie mahdollisimman vähän tilaa.

Askeleet on esitetty kuvassa 3.1.



Kuva 3.1.Ortogonaalisen graafin piirtäminen. Musta neliö esittää graafiin lisättyä solmua.

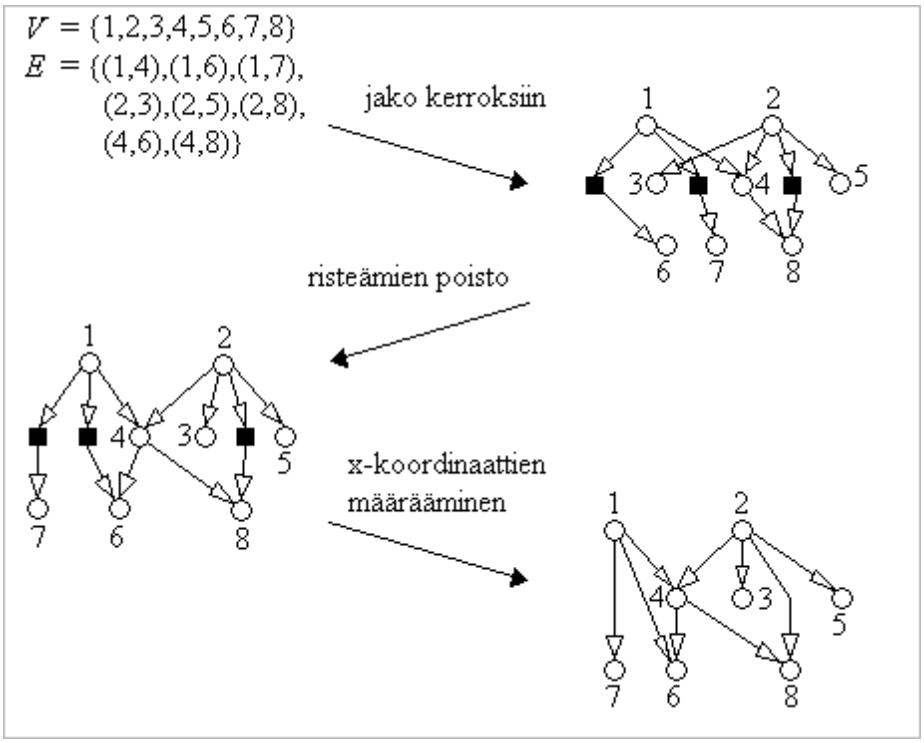
Yllämainittu heuristiikka pyrkii minimoimaan särmien leikkauksen piirroksessa ja tämän takia särmien taivutusten määrää ei yritetä minimoida. Tässä on juuri esimerkki

algoritmien suunnittelun vaikeudesta, kun pitää valita särmien leikkausten minimointi tärkeämmäksi kuin särmien taivutusten määrä. Jos haluttaisiin minimoida taivutusten määrä pitäisi ortogonaaliaskel tehdä ennen tasoon piirtämistä. Tiivistystä ei pidetä tärkeänä ortogonaalisen graafin piirtämisessä.

3.2. Hierarkkinen lähestymistapa

Edellisessä kohdassa käsiteltiin suuntaamatonta graafia. Koska ohjelmistokehityksessä on tärkeää tietää relaation suunta, tarvitaan suunnattua graafia, jossa särmillä on suunta. Tällaisia särmiä voidaan kutsua kaariksi. Tässä kohdassa käsitellään tapauksia, joissa kaaret piirretään alaspäin tai ylöspäin suuntautuvilla moniviivoilla. Suunnatun syklittömän graafin piirto koostuu kuten edellisenkin kohdan piirrosheuristiikka kolmesta eri vaiheesta:

- Ensimmäisessä askeleessa graafi jaetaan kerroksiin. Sen jälkeen graafin solmut asetetaan tasoille L_1, L_2,..., L_h siten, että jos (u,v) on kaari, jossa $u \in L_i$ ja $v \in L_j$, niin $i > j$. Lopullisessa piirroksessa jokaisella solmulla tulee olemaan y-koordinaatti, jota vastaa arvo i. Seuraavaksi muodostetaan kerroksiin hyvin järjestetty suunnattu graafi. Tämä tarkoittaa sitä, että jos (u,v) on kaari, jossa $u \in L_i$ ja $v \in L_j$, niin $i = j + 1$. Tämä saadaan aikaiseksi siten, että piirrokseen lisätään solmuja niihin kerroksiin, jotka kaari ylittäisi muuten kokonaan. Tämä muodostus on nähtävissä kuvassa 3.2.
- Kaarien risteämien vähentäminen on seuraava askel. Tässä vaiheessa määrätään kaikille solmuille järjestys tasoilla ja tämän järjestyksen tulee minimoida kaarien keskinäiset leikkaukset. Tämän vaiheen määräävä topologisuus jää myös piirroksen lopulliseksi.
- Viimeisessä vaiheessa jaetaan solmuille x-koordinaatit. Nämä koordinaatit määräytyvät solmujärjestyksen mukaan, joka tehtiin edellisessä vaiheessa. Koordinaattien jakaminen ei ole yksikäsitteistä, sillä ensimmäisessä vaiheessa lisätyt solmut joudutaan poistamaan ja vaihtoehtona on piirtää joko kohtisuora viiva tai sitten taivuttaa kaarta. Vaihtoehtojen keskinäisen järjestyksen paremmuus riippuu täysin tapauksesta.

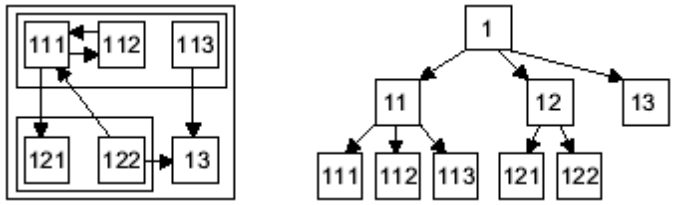


Kuva 3.2. Hierarkkisen suunnatun graafin piirtokulun esitys

Suunnatussa graafissa voi myös olla sykli, joka ei kumminkaan muodosta mitään ongelmaa vastaavanlaisen alaspäin suunnatun graafin piirtämiseen. Tällaisessa tapauksessa tehdään graafiin sellainen muunnos, että yksi sykliin kuuluvista kaarista käännetään toisin päin ja piirros muodostetaan samaan tapaan. Sen jälkeen kun *x*-koordinaatit on määrätty, käännetään sama kaari takaisin niin päin kuin se aluksi oli.

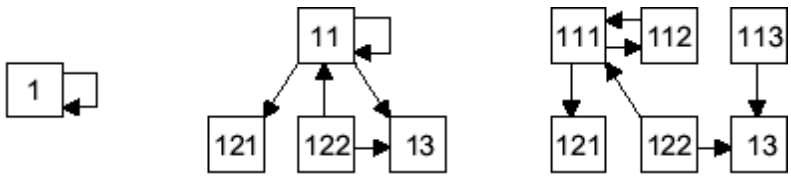
3.3. Hierarkkinen graafi

Hierarkkinen graafi $H = (G, T)$ koostuu suunnatusta graafista G ja puusta T, jonka lehdet ovat täsmälleen samat kuin graafin G solmut. Puuta T kutsutaan graafin H hierarkkiseksi puuksi ja graafia G hierarkkisen graafin H alusgraafiksi. Kuvassa 3.3 on hierarkkinen graafi ja sen hierarkkinen puu. Tämä erottelu antaa hyvän pohjan ohjelmistojen mallinnukseen, sillä hierarkkinen puu mallintaa luokkien perimäjärjestystä ja alusgraafista käy ilmi operaatiot eli metodikutsut. Mikäli alusgraafissa on särmä solmujen välillä, tarkoittaa se kahden eri ohjelmistoentiteetin välistä vuorovaikutusta.



Kuva 3.3. Alusgraafi ja hierarkkinen puu.

Kahdelle hierarkkisen puun solmulle *u* ja *v* määritellään joukko leaves(*u*,*v*), joka on niiden särmien joukko graafissa G, jotka kulkevat solmun *u* lehdestä solmun *v* lehteen. Jos leaves(*u*,*v*) on ei-tyhjä, niin sanomme, että on olemassa indusoitu särmä solmusta *u* solmuun *v*. Puun solmujen osajoukolle *V* määritellään induced(*V*) kaikkien indusoitujen särmien joukkona joukon *V* solmujen välillä. Kuvan 3.3 esimerkissä solmun 11 lehdet ovat 111, 112 ja 113, ja solmun 13 ainoa lehti on solmu 13 itse. Alusgraafin ainoa särmä solmun 11 lehdistä solmun 13 lehtiin on {(113,13)} ja täten leaves(11,13) = {(113,13)}. Joukko induced({11,13}) sisältää ei ainoastaan särmän (11,13), mutta myös särmän (11,11), joka on indusoitu alusgraafin särmien (111,112) ja (112,111) johdosta.



Kuva 3.4: Kolme näkymää Kuvan 6 hierarkkisesta graafista.

Hierarkkisen graafin *H* näkymä on sellainen suunnattu graafi (*V*,induced(*V*)), että joukko *V* sisältää täsmälleen yhden esivanhemman jokaista alusgraafin *G* solmua kohden. Kuvassa 3.4 on kolme eri näkymää kuvan 3.3 hierarkkisesta graafista. Intuitiivisesti *näkymä* on alusgraafin selvitys tai abstraktio. Tällaiset abstraktiot ovat välttämättömiä laajojen ohjelmistojen kattavaan visualisointiin. Ohjelmistoa voidaan sanoa laajaksi, kun se käsittää yli 100000 metodia ja attribuuttia. Esimerkki kuvan 3.3 hierarkkisen graafin aligraafista, joka ei ole näkymä, on graafi, joka sisältää solmut 11 ja 112. Nämä solmut ovat molemmat solmun 112 esivanhempia. Solmun 112 kahden jäsenen esiintyminen tekee vastaavuuden alusgraafiin vähemmän selkeämmäksi kuin näkymille, koska alusgraafi sisältää vain yhden kopion solmusta 112.

4. Hierarkkisen graafin rakenne ohjelmistonkehityksessä

Graafien rakenne on yksi tärkeimpiä ominaisuuksia, kun pyritään mallintamaan ohjelmistoa luokkakaaviolla. Tietokoneelle on yhdentekevää, kuinka siihen tallennettu informaatio tuodaan esille, mutta ihmissilmälle ei. Ihmiselle on huomattavasti helpompaa hahmottaa hyvin järjestelty näkymä järkeväksi kokonaisuudeksi kuin sotkuinen piirros, jossa on esimerkiksi useita luokkien ja assosiaatioiden päällekkäisyyksiä. Myös toisiensa kanssa tekemisissä olevat luokat on järkevää sijoittaa lähelle toisiaan. Sellaista menetelmää, jossa pyritään jaottelemaan substansseja eri ryhmiksi, kutsutaan klusteroinniksi.

Hierarkkisia graafeja käytetään laajalti ohjelmistoja mallinnettaessa. Keskeisin ongelma graafin piirtämisessä on sen rakenteen eli ulkoasun muodostaminen. Solmut voidaan kuvata kaksi- tai kolmiulotteisessa avaruudessa, joka tekee graafin rakenteen laskemisesta hankalaa. Andreas Noack ja Claus Lewerentz [2005] esittävät, miten ohjelmistoja mallintavat graafit voidaan piirtää kolmiulotteisessa avaruudessa. He tarkentavat ongelmat seuraavasti: rakenteet, joissa otetaan huomioon yksittäisten solmujen etäisyys toisistaan verrattuna solmuklusterien välisiin etäisyyksiin, naapurusto verrattuna hierarkkisuuteen ja rakenteet, joissa kuvataan graafin osajoukon kokoa verrattuna tilanteeseen, että osajoukko suurennettaisiin.

4.1. Energia- ja voimamallit

Energiamallit on laajalti käytetty lähestymistapa mallin luontiin, kun pitää yhdistää geometrisia primitiivejä, kuten sylintereitä ja suorakulmioita, moninaisiin operaatioihin, kuten siirtämiseen, kiertoon, Boolen-arvoihin ja muodonmuuttamiseen. Tällaisen mallin rakenteen luomiseen tarvitaan kahta asiaa: hierarkian rakentaminen sekä kappaleiden sisäisten parametrien ja operaatioiden määritteleminen. Energia- ja voimamallit on jo kauan tunnettu automaattisen asettelun heuristiikka tietokonegrafiikassa [Witkin et al., 1987].

Kun malli muodostetaan, käytetään hierarkian kuvaamisessa rajoitteita, jotka määrittelevät mallin rakenteen. Rajoitteet voidaan muotoilla energiafunktioksi mallin parametriavaruudessa. Funktion arvot ovat positiivisia ja vaaditut rajoitteet vaativat funktioilta arvon nolla. Funktioiden arvot summataan yhteen skalaariseen funktioon, joka kuvaa koko mallia. Skalaarifunktion energiaa voidaan minimoida käymällä läpi parametriavaruutta.

Rajoitefunktioita ei kutsuta energiafunktioiksi siksi, että ne mallintaisivat todellista fysikaalista järjestelmää, vaan koska ne toimivat samalla tavalla rajoitteiden ratkaisemisessa kuin fysikaalinen energia. Esimerkiksi energiarajoite, joka liittää yhteen kaksi pistettä tasossa, toimii kuten jousi, joka vetää kahta kappaletta yhteen. Kappaleiden sisäiset parametrit ovat vapaasti määriteltävissä, kuten esimerkiksi sylinteri voi muuttaa pituuttaan ja sädettään sopivaksi rajoitteeseen.

Määritellään kaikkien kappaleen parametrien unioni Ψ, johon kuuluvat esimerkiksi kappaleiden välimatkat ja koot. Rajoitteiden algebrallisen laskemisen sijasta määritellään rajoitteet energiafunktioina ja liikutaan parametriavaruuden läpi energian gradientin mukaan. Määritellään sellainen vain positiivisia arvoja saava sileä funktio $E(\Psi)$, että rajoitteet täytetään, kun $E(\Psi)=0$. n-rajoitteisen joukon ratkaisut ovat muuttujan ψ sellaisia arvoja, että

$$E(\Psi)=\sum_{i}^{n} E_i(\Psi)=0,$$

joten rajoitteiden yhdistäminen tapahtuu yksinkertaisesti vain energioiden summana. Energiaa E voidaan vapaasti ilmaista paikkafunktiona, normaalifunktiona tai implisiittisenä funktiona, tai millä tahansa muulla suureella.

Intuitiivisesti energiarajoitteita voidaan tarkastella voimina, jotka vetävät mallin osat haluttuun muotoon ja pitävät ne siinä. Vaikka voimia ei ole tarkoitettu realistisiksi fysikaalisiksi voimiksi, yksinkertainen liitosrajoite voitaisiin toteuttaa jousella, joka yhdistää pisteitä. Saadaan

$$E_{spring}=k\left|P_1(u_1,v_1)-P_2(u_2,v_2)\right|^2,$$

jossa *k* on jousivakio ja voimavektori parametriavaruudessa on ΔE_{spring}.

Energiamallin minimoiminen on yleisesti käytetty menetelmä graafin muodostamisessa [Di Battista et al., 1999]. Suunnatun graafin (*V*,*E*) *n-dimensioinen* rakenne p on solmujen paikkavektori $(p_v)_{v\in V}: p_v\in\mathbb{R}^n$. Rakenteessa p solmujen $u, v\in V$ euklidisesta etäisyydestä käytetään merkitään $\text{dist}_p(u,v)$.

4.2. Ohjelmistorakenne ja sen kolme dimensiota

Noack ja Lewerentz [2005], joiden artikkeliin tämä kohta pohjautuu, toteavat, että suurien ohjelmistojen ymmärtäminen, arvioiminen ja parantaminen vaatii eri abstraktitasojen näkymiä. Näkymät voidaan jakaa kolmeen eri ryhmään: luokkapakettien väliset yhteydet, vuorovaikutus operaatioiden ja attribuuttien välillä, sekä operaatioiden ja attribuuttien tarkastelu globaalisesti. Koska ei ole olemassa yhtä ainoaa rakennetta, joka tyydyttäisi kaikki vaatimukset, tarvitaan jaottelua kolmeen eri dimensioon:

- Klusteroinnin aste: Rakenteet, jotka perustuvat yksittäisten solmujen etäisyyksiin (analysoidaan ohjelmistoentiteettien paikallista naapurustoa) verrattuna rakenteisiin, joissa otetaan huomioon solmuryhmien väliset etäisyydet (analysoidaan yleistä ohjelmistorakennetta).

- Hierarkkisuuden aste: Rakenteet, joissa otetaan huomioon särmien esiintyminen solmujen välillä (analysoidaan ohjelmistoentiteettien välistä suhdetta) verrattuna rakenteisiin, joissa huomioidaan yhteisiä vanhempia hierarkkisessa puussa (analysoidaan hierarkian hallintaa). Näiden kahden ääritapauksen suhde on mielenkiintoinen, koska tulee näyttää, kuinka suuri määrä läheisessä suhteessa olevia ohjelmistoentiteettejä soveltuvat yhteen hierarkkisuuden hallitsemisen kanssa.
- Hajonnan aste: Rakenteet, jotka kuvaavat tarkasti solmujen ja särmien kokoa (analysoidaan entiteettien ja suhteiden yhtäläistä rakeisuutta) verrattuna rakenteisiin, joissa tietyt solmut ja särmät ovat suurennettuina (analysoidaan yksityiskohtia globaalisti).

Klusteroinnin, hierarkkisuuden ja hajonnan asteet ovat energiamallin parametreja. Kun graafi piirretään kahteen eri dimensioon, on kyseessä jako alusgraafiin ja hierarkiapuuhun. Alusgraafi esittää metodien väliset kutsut ja hierarkiapuu kuvaavat luokkarakennetta.

4.2.1. Klusteroinnin aste

Klusteroinnin aste on riippuvainen kahden eri rakenteellisuuden tasapainosta. Lokaalisesti tulkittavat etäisyydet eli vierekkäiset solmut ovat lähempänä toisiaan kuin ei-vierekkäiset solmut, ja globaalisesti tulkittavat etäisyydet eli vahvasti toisiinsa liittyvät joukot ovat lähempänä toisiaan kuin ei niin vahvasti toisiinsa liittyvät joukot. Ensimmäinen näistä käsittelee ohjelmistoanalyysin kannalta yksittäisten entiteettien välisiä suhteita ja jälkimmäinen kohdistuu globaalin rakenteen muodostamiseen tai aliohjelmistojen mallintamiseen.

Lokaalisesti tulkittavat etäisyydet esiintyvät usein esimerkiksi silloin, kun tarkastellaan, mikä operaatio eli metodi kutsuu tiettyä metodia m. Jos metodin m allekirjoitusta muutetaan, joudutaan kaikki sitä kutsuvat metodit sovittamaan uudelleen. Sama lokaalitulkinta tulee esiin, kun tarkastellaan, mitkä metodit muuttavat attribuutin a arvoa. Jos a:ta ei ole dokumentoitu, niin näiden metodien ymmärtäminen auttaa attribuutin tarkoituksen löytämisessä. Lausekielen luonne esittää keskeistä roolia ohjelmistoarkkitehtuurien ymmärtämisessä, arvioinnissa ja parantamisessa. Esimerkiksi aiotun arkkitehtuurin mukaan pakettia p1 voivat käyttää vain paketit p2 ja p3, joten mitkä muut paketit käyttävät pakettia p1? Vastauksena on ristiriitojen lista, joka löytyy aiotusta arkkitehtuurista. Käyttääkö paketti p4 suoraan vai epäsuorasti pakettia p5? Muuten pakettia p4 voidaan käyttää itsenäisesti riippumatta paketista p5.

Globaalisesti tulkittavat etäisyydet käsittelevät ohjelmiston hajottamista osiin. Kuinka voidaan erotella ohjelmistoentiteetit eri ryhmiin sitten, että ainoastaan vahvasti toisiinsa liittyvät entiteetit ovat samassa ryhmässä, mutta kahden eri ryhmän sisäisillä

entiteeteillä ei ole vahvaa vuorovaikutusta keskenään Tällaista menetelmää kutsutaan myös ohjelmiston klusteroinniksi. Yksi tapa erottaa eri ryhmiä on rajapinnat. Tällöin selvitetään, onko olemassa koko ryhmälle yhtenäinen rajapinta.

Näkymän (V,induced(V)) rakenteelle p ja klusteroinnin asteelle c ($0 < c \leq 1$) määritellään energia kaavalla

$$U(p,c) = U_{attr}(p,c) + U_{repu}(p),$$

jossa

$$U_{attr}(p,c) = \sum_{(u,v)\in \text{induced}(V)} \left|\text{leaves}(u,v)\right| \ \text{dist}_p(u,v)^{1/c}$$

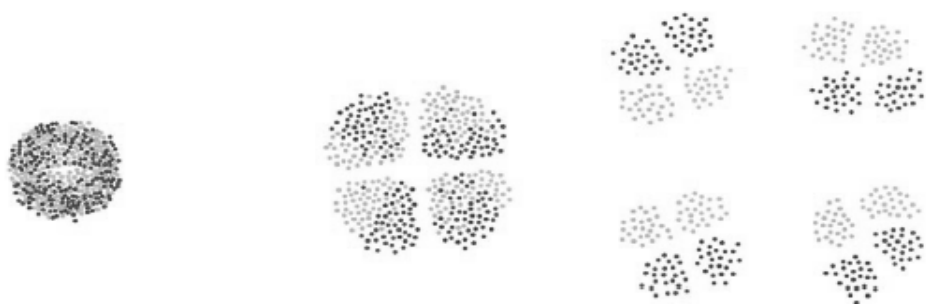
$$U_{repu}(p) = \sum_{(u,v)\in \text{induced}(V)} -w(u)\,w(v)\ln(\text{dist}_p(u,v))$$

$$w(v) = \sum_{(u,v)\in \text{induced}(v)} \left|\text{leaves}(u,v)\right| + \sum_{(v,u)\in induced(V)} \left|\text{leaves}(v,u)\right|.$$

Klusteroinnin asteen vaikutusta piirrokseen esitellään kuvassa 4.1.

Kokonaisenergia rakenteelle saadaan kahdesta termistä, jotka voidaan tulkita vierekkäisten solmujen väliseksi vetovoimaksi (engl. attraction) ja kaikkien solmuparien väliseksi työntövoimaksi (engl. repulsion). Kahden vierekkäisen solmun välistä vetovoimaa painotetaan lehtien välisten särmien lukumäärällä. Jokaiselle solmulle v kokonaistyöntövoima toisiin solmuihin nähden lasketaan solmuun tulevien ja solmusta lähtevien lehtien välisten särmien lukumäärällä $w(v)$. Tämä painotus voidaan tulkita alusgraafin särmien välisenä vaikutuksena eli tarkemmin graafin päätesolmujen välisenä esiintymisenä. Samalla tämä on myös solmun vetovoimavaikutusten summa, joten jokaisella solmulla v on johdonmukaisesti vaikutuksensa rakenteeseen riippuen solmun suhteellisesta lehtien särmämäärästä, jolla on vaikutuksensa veto- ja työntövoimaan kokonaisenergiassa.

Tämä ominaisuus näyttelee keskeistä roolia Noackin ja Lewerentzin [2005] energiamallissa.



Kuva 4.1. Klusterointi eri asteilla: $c = 0.2$, $c = 0.5$ ja $c = 1.0$.

4.2.2. Hierarkkisuuden aste

Hierarkkisuuden aste on vaihtokauppaa kahden eri rakenteen välillä. Toinen rakentaa alusgraafin eli pyrkii asettamaan vierekkäiset solmut lähelle toisiaan ja toinen rakentaa hierarkkisen puun eli pyrkii asettamaan solmut, joilla on yhteisiä esivanhempia, lähelle toisiaan. Ensimmäisessä käsitellään sellaisia ohjelmistoentiteettien välisiä relaatioita kuin metodikutsuja ja perintää ja jälkimmäisessä keskitytään entiteettijoukkojen hierarkkisuuden hallintaan. Tässä tarkoitetaan sellaista hierarkkisuutta, jossa jälkeläinen kuuluu aina esivanhempijoukkojensa osajoukkoon. Hyvä esimerkki löytyy vaikka biologiasta: eläin → lintu → petolintu → kotka → maakotka. Käytämme jatkossa sanaa luokitus kuvaamaan hierarkkisuuden hallintaa.

Ohjelmistojen rakenteessa tulee kolme eri asiaa kyseeseen: ohjelmistoentiteettien väliset relaatiot, luokitus ja molemmat yhdessä.

Luokituksessa täytyy ottaa seuraavanlaisia asioita huomioon: Onko luokkajako tasapainoinen eli onko samalla tasolla olevat luokkajoukot luokituksessa samankokoisia? Mitkä ovat suurimmat ja pienimmät luokkajoukot samalla tasolla? Mihin sijoitetaan yksittäinen luokka? Jos luokka L1 kuuluu useampaan kuin yhteen luokkajoukkoon, niin se on helppo sijoittaa, mutta jos luokka L1 kuuluu useampaan luokkajoukkoon, niin mikä on sen paikka luokituksessa. Mihin luokkajoukkoon kuuluvat L1 ja L2?

Tällaisiin kysymyksiin on helpompi löytää vastaus graafinpiirron ulkopuolelta, joten graafinpiirrossa keskitytään ohjelmistoentiteettien välisten relaatioiden ja luokituksen yhtäaikaiseen määrittämiseen. Ensimmäinen näistä vaatii, että tarkastellaan vahvoja solmuyhteyksiä alusgraafissa ja sellaisia solmuja, joilla on yhteinen vanhempi hierarkiapuussa. Luokitus tapahtuu, kun graafille määritellään hierarkkisuuden aste.

Saavuttaaksemme näkymän rakenteelle (V,induced(V)) laajennamme edellisen kohdan kaavaa gravitaatioenergian summalla. Energialla saadaan nyt kaava

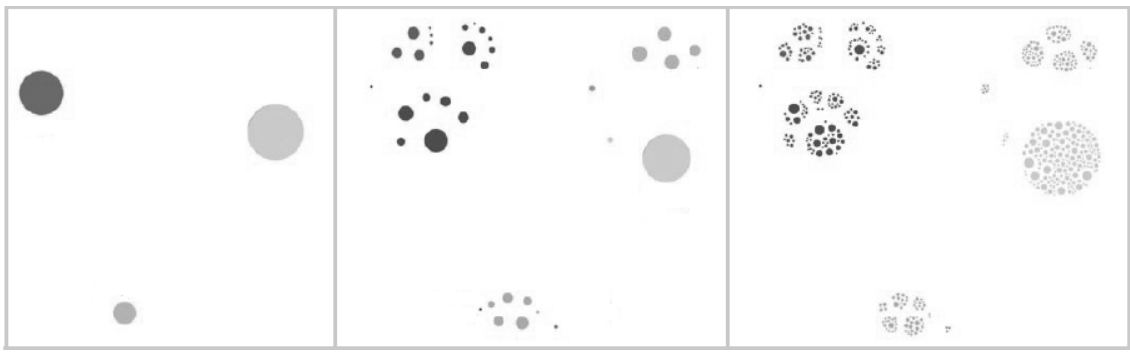
$$U(p,c,h) \quad = \quad (1-h)U_{attr}(p,c)+U_{repu}(p)+hU_{grav}(p,c),$$

jossa

$$U_{grav}(p,c) \quad = \quad \sum_{v \in V} \mathrm{w}(v)\mathrm{dist}_p(v,\mathrm{parent}(v))^{1/c}.$$

Gravitaatioenergiaa eli painovoimaa painotetaan hierarkkisuuden asteella h ($0 \le h \le 1$) ja vetovoimaa painotetaan arvolla $(1 - h)$. Edellisen alakohdan energiamalli on sama kuin hierarkkisuuden aste 0. Gravitaatio on samanlainen kuin vetovoima, mutta se ei vaikuta vierekkäisen solmujen välillä, vaan kaikkien solmujen v vanhempien parent(v) välillä hierarkiapuussa. Suoraviivaisesti yleistettynä gravitaatio voi vetää jokaista solmua lähemmäksi esivanhempiaan kuten kuvassa 4.2. Solmun vanhemmat eivät kuulu näkymään, eivätkä vaikuta vetovoimaan tai työntövoimaan, joten niitä ei näytetä

piirroksessa. Painovoiman perustarkoituksena on järjestää yhteisen vanhemman omistavat solmut samaan ryhmään. Kuten veto- ja työntövoima, niin myös painovoima painotetaan yksittäisellä painolla w(*v*) jokaiselle solmulle *v*.



Kuva 4.2. Kolme eri näkymää. Ensimmäinen on ylin luokkapakettinäkymä, toinen on kolmas luokkapakettinäkymä ja viimeinen on ylin luokkanäkymä.

4.2.3. Hajonnan aste

Hajonnan tai säröisyyden aste voidaan jakaa kahteen eri rakenteellisuuden vastakkainasetteluun. Rakenteisiin, joissa indusoidun särmän tärkeys on verrannollinen vastaaviin särmiin alusgraafissa, ja rakenteisiin, joissa kaikki indusoidut särmät ovat yhtä tärkeitä. Jos esimerkiksi paketilla p1 on 10 luokkatason perimärelaatio pakettiin p2 ja paketilla p3 on kahden luokkatason perimärelaatio pakettiin p4, niin tällöin voidaan sanoa, että indusoitu perimärelaatio paketista p1 pakettiin p2 on viisi kertaa tärkeämpi kuin indusoitu perimärelaatio pakettien p3 ja p4 välillä, tai sitten se on yhtä tärkeä. Ensimmäinen vaihtoehto esittää objektiiviset koot tarkasti, kun taas jälkimmäisellä voidaan esittää jonkin jäsennyksen tärkeyttä.

Korkeamman tason näkymä, kuten luokkapakettinäkymä, voidaan käsittää tiivistelmänä alemman tason näkymästä, kuten luokkanäkymästä. Vääristelemättömässä tiivistelmässä ylemmän tason entiteetin tai relaation paino on sen edustamien alemman tason entiteettien tai relaatioiden painojen summa.

Joissakin tapauksissa tietyt entiteetit tai relaatiot ovat tärkeämpiä kuin niiden koko antaa ymmärtää. Esimerkiksi kuinka luokan L metodit ja attribuutit vaikuttavat muuhun ohjelmistoon? Luokka L voi olla pieni osa ohjelmistoa, mutta tärkeä analysointimielessä. Furnasin [1986] terminologialla tämä voitaisiin esittää seuraavasti: entiteetin tai relaation kiinnostavuuden aste ei ainoastaan riipu sen *a priori* tärkeydestä, vaan myös fokuspisteiden valinnasta.

Monet ongelmat tarvitsevat rakenteita, jotka tarkasti tiivistävät alusgraafin, koska graafin hajottamisessa osiin piilee väärinymmärryksen vaara, mutta ongelmat, joissa

tarkastellaan paikallisia ominaisuuksia, voivat hyötyä hajottamisesta pienempiin suurennettuihin osiin.

Saattaaksemme näkymän rakenteelle (V, induced(V)), jossa kaikki indusoidut särmät ovat yhtä tärkeitä, tulee meidän laajentaa vetovoimaa ja painoa funktiossamme parametrilla d ($0 \le d \le 1$), jota kutsutaan hajonnan asteeksi:

$$U_{attr}(p, c, d) = \sum_{(u, v) \in \text{induced}(V)} \left|\text{leaves}(u, v)\right|^{1-d} \text{dist}_p(u, v)^{(1/c)}$$

$$\text{w}(v, d) = \sum_{(u, v) \in \text{induced}(V)} \left|\text{leaves}(u, v)\right|^{1-d} + \sum_{(v, u) \in \text{induced}(V)} \left|\text{leaves}(v, u)\right|^{1-d}.$$

Tämä kaava vastaa erikoistapausta, jossa hajonta on 0 ($d = 0$). Maksimienergialla jokaisella särmällä on sama paino 1.

5. Graafin estetiikka ja empiirinen tutkimus

Helen C. Purchase on uraauurtava tutkija graafien estetiikalle ja tässä luvussa keskitytään hänen tutkimuksiinsa, mutta myös tuodaan esille automaattisten graafien rakenteen muodostavien algoritmien suunnittelijoiden näkökulmia graafien estetiikalle.

On olemassa useita automaattisia algoritmeja graafin rakenteen muodostamiseksi [Di Battista et al., 1999; Eiglsperger et al., 2003; Von Gudenberg et al., 2006], jotka saavat syötteenä graafirakenteen objekteina ja niiden välisinä relaatioina. Algoritmit muodostavat syöteinformaatioista kaavion, jonka rakennetta algoritmien suunnittelijat optimoivat tietyillä estetiikkakriteereillä, ja näin tehdessään he väittävät helpottavansa graafin lukijoita ymmärtämään sen ilmaisemaa informaatiota. Esteettiset kriteerit ovat määritelty ja sittemmin käytetty tutkijoiden toimesta graafin automaattisessa asettelussa, joten ne eivät välttämättä ole esteettisesti miellyttäviä visuaalisen havainnoinnin mielessä. Nämä algoritmit on suunniteltu ja määritelty abstraktien graafirakenteiden pohjalta eikä niissä ole otettu huomioon ihmisen ja tietokoneen välistä vuorovaikutusta kaavioiden ymmärtämiseen.

Purchase ja muut [2001] ovat järjestäneet kokeita, joissa on testattu ihmisten hahmotuskykyä kaavioista, joiden rakenteet on tuotettu automaattisesti algoritmeilla. Näissä kokeissa on ilmennyt, että kun minimoidaan särmien leikkauksia ja taivutuksia sekä maksimoidaan symmetriaa, niin koehenkilöt ratkaisevat tehokkaammin graafeihin liittyviä päättelytehtäviä. Testitapaukset olivat sovellusriippumattomia eli käytetyt graafit edustivat merkityksettömiä objekteja ja relaatioita, joten näillä sovellusriippumattomilla testeillä ei välttämättä ole suoraa yhteyttä UML-luokkakaavioiden ymmärtämisen tehokkuuden kanssa.

Koejärjestelyitä tehtiin kaksi erilaista, joissa molemmissa pyrittiin päättelemään, mikä graafinpiirtoestetiikka on kaikkein tärkein esitettäessä UML-luokkakaavioita. Kokeissa ei tarkasteltu laskennallista tehokkuutta, suunnittelijoiden mieltymyksiä tai kohteidenkaan mieltymyksiä, vaan estetiikkaa, joka on helpottaa testihenkilöiden päättelyä. Molemmissa koejärjestelyissä käytettiin identtistä metodologiaa, mutta erilaista tapaa muodostaa koekaaviot. Kokeessa A estetiikkaa mitattiin laskennallisesti ja kokeessa B sitä mitattiin hahmottamiskyvyllä.

5.1. Estetiikkakriteerit

Kokeessa A käytettiin viittä eri yleistä graafinpiirto-estetiikkakriteeriä [Di Battista et al., 1999; Eiglsperger et al., 2003]:

- (b) Minimoidaan taivutuksien määrä (särmientaivutusten kokonaismäärä moniviivassa tulee minimoida).
- (n) Solmujako (solmut tulisi jakaa tasan rajoittavan laatikon sisällä).
- (ev) Särmien pituuksien vaihtelu (särmien pituudet tulisi olla yhdenmukaisia).
- (f) Virransuunta (suunnatuilla särmillä yhdenmukainen suunta).

- (o) Ortogonaalisuus (asetetaan solmut ja särmät ortogonaalisesti).

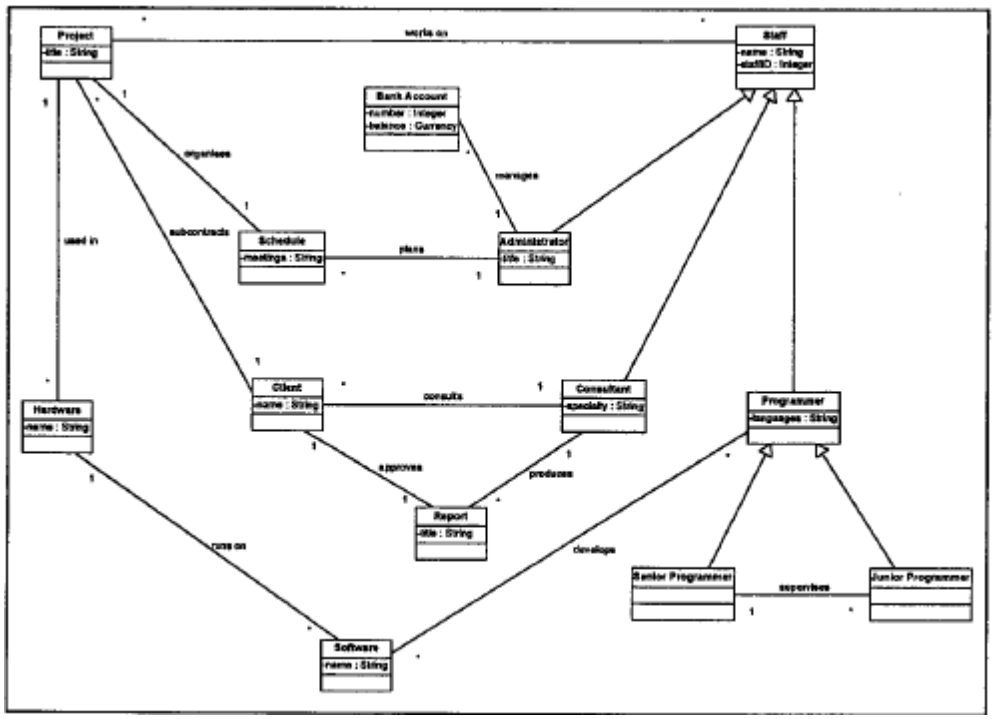
Kaksi muuta estetiikkakriteeriä lisättiin kokeeseen B:

- (el) Särmien pituus (särmien pituus tulisi minimoida, mutta särmät eivät saisi olla liian lyhyitä).
- (s) Symmetrisyys (kun vain mahdollista, graafista tulisi esittää symmetrinen näkymä).

5.2. Koe A

Luokkakaaviot, joita tässä kokeessa käytettiin, pohjautuivat yksinkertaiseen joukkoon, joka mallintaa pientä tietotekniikka-alan yritystä, joka työllistää hallinnon henkilökuntaa, konsultteja ja ohjelmoijia toimenkuvanaan asiakasprojektit. Esimerkkikaavio käsittää 13 objektia, 12 assosiaatiota ja 5 perimisrelaatiota (ks. kuva 5.1).

Opaslehtinen selitti UML-luokkakaavioiden tarkoituksen ja selitti niiden semantiikka yksinkertaisilla esimerkeillä. Koehenkilöiden ei odotettu omaavan kokemusta UML-notaatiosta ja tällä opaslehtisellä tarjottiin kaikki tieto, jota he tarvitsivat koetehtävään. Varta vasten tehty esimerkki havainnollisti koehenkilöille tehtävän, joka heidän tulisi suorittaa. Esimerkki koostui neljästä eri kaaviosta ja määritellystä. Jokaisen esimerkkikaavion kohdalla kerrottiin, kuuluiko annettu kaavio annettuun määrittelyyn vai ei. Opastusjärjestelyissä pidettiin huolta siitä, ettei koehenkilöt kohdistaneet mieltymyksiään tiettyyn graafirakenteeseen.



Kuva 5.1. UML-luokkakaavio, jota käytettiin sekä kokeessa A, että kokeessa B.

5.2.1. Koekaaviot ja koejärjestely

Koekaaviot tuotettiin laskennallisten metriikoiden keinoin, jotka mittasivat jokaista estetiikkavaatimusta kaaviossa. Tälle metriikalle asetettiin rationaaliluku väliltä 0 ja 1, jossa 1 tarkoitti hyvää estetiikkaa kriteerien mukaan.

Jokaiselle estetiikalle tehtiin kaavioversio, joissa oli kriteerille joko vähäinen vaikutus (-) tai suuri vaikutus (+). Tässä vähäinen vaikutus oletettiin metriikalle kriteerien määrittelystä siten, että esimerkiksi kaaviossa esiintyy paljon särmien taivutuksia ja eri pituuksia, ja suuri vaikutus esimerkiksi, että enemmistö suunnatuista särmistä osoittaa ylöspäin ja solmujako on tasainen. Kaavioiden välisiä eroja kontrolloitiin mahdollisimman paljon, ettei syntyisi estetiikkojen välisiä sekoittavia tekijöitä. Esimerkiksi, kun vähennettiin kahden kaavion välillä mahdollisesti sekoittavaa estetiikkakriteeriä, niin kaikki muut kriteerit pidettiin keskivälillä vaikutusaluettaan. Tämä takasi sen, että mikä tahansa merkittävä ero vähäisen vaikutuksen ja suuren vaikutuksen välillä voitiin määrittää asiaan kuuluvaan estetiikkaan.

Aikaisemmat Helen C. Purchasen [Purchase et al., 2001a] tutkimukset ovat vakuuttavasti osoittaneet, että särmien risteymät ovat este ihmisten kyvyille ymmärtää kaaviopiirroksia, joten yhdessäkään koekaavioista ei ole särmien risteymiä.

Diagram	Aesthetic				
	bends (b)	orthogonality(o)	edge variation (ev)	node distribution(n)	direction of flow (f)
b+	**1**	0.43	0.66	0.59	0.6
b-	**0.71**	0.46	0.64	0.56	0.6
o+	0.85	**0.70**	0.66	0.56	0.4
o-	0.85	**0.32**	0.64	0.56	0.6
ev+	0.85	0.44	**0.74**	0.59	0.6
ev-	0.85	0.41	**0.55**	0.59	0.6
n+	0.85	0.41	0.66	**0.73**	0.4
n-	0.85	0.48	0.64	**0.45**	0.6
f+	0.85	0.44	0.65	0.59	**1**
f-	0.85	0.46	0.66	0.59	**0**
control	0.85	0.45	0.66	0.57	0.6
example	0.85	0.44	0.66	0.56	0.6

Taulukko 5.2. Kokeessa A käytetyt kaaviot ja estetiikkojen painotukset.

Taulukossa 5.2 esitetään kokeessa käytettyjen kaavioiden estetiikkapainotukset. Ensimmäinen reaalilukusarake kuvaa sellaisen kaavion, jossa pyritään selvittämään taivutuksien määrän merkitystä graafin ymmärtämisessä. Siitä seuraavissa sarakkeissa estetiikka-arviot eri kaavioille menee järjestyksessä: ortogonaalisuus, särmien pituuksien vaihtelu, solmujako ja virransuunta. Taulukosta huomataan, että tarkasteltavan estetiikan kohdalla muut estetiikkakriteerit asetetaan yhtä merkitseviksi, jotteivat ne vaikuta tulkittuun tulokseen.

Koehenkilöt käyttivät verkon kautta yhteydessä olevaa ohjelmaa kokeen suorittamiseen. Kopio esimerkkikaaviosta ja esimerkkikaavion määritelmästä oli tulostettuna tietokoneen viereen, mikä takasi koehenkilöille helpon tavan kerrata graafinrakenteen tulkitsemista. Kokeen UML-kaaviot esitettiin satunnaisessa järjestyksessä jokaiselle koehenkilölle erikseen ja heidän tuli antaa kyllä- tai ei-vastaus jokaisen kaavion kohdalla, että oliko kaavio heidän mielestään yhtäläinen annetun määrittelyn kanssa. Vastaaminen tapahtui painamalla toista kahdesta eri napista.

Koehenkilöille näytettiin ensiksi 21 kaavion joukosta 16 satunnaisesti valittua kaaviota harjoituksena, mutta koehenkilöt eivät olleet tietoisia, että näistä 16 kaaviosta ei kerätty tuloksia ollenkaan. Näin koehenkilöt saivat harjoitella tehtävää ennen kuin varsinaiset tulokset kerättiin. Kokeessa esitettiin 32 kaavion joukosta 11 sellaista kaaviota, jotka vastasivat määrittelyä ja 10 sellaista, jotka eivät vastanneet määrittelyä. Jokaista kaaviota esitettiin niin kauan kunnes henkilö antoi vastauksensa tai 50 sekunnin aika kului loppuun. Ajan loppuun kuluminen tulkittiin virheeksi. Harjoituskaaviot auttoivat koehenkilöitä tottumaan käytettävissä olevaan aikaan.

5.2.2. Tulokset ja päätelmät

Koehenkilöiden suorituksista mitattiin sekä nopeutta että tarkkuutta, jotta saatiin kaksi eri muuttujaa tuloksien analysointiin. Tarkkuusarvossa ei esiintynyt tuloksissa mitään merkittävää, joka tarkoitti käytettävissä olevan ajan riittävän koehenkilöille kaavion oikeelliseen tulkintaan. Eli vain yhtä muuttujaa käytettiin tulosten tulkitsemiseen – koehenkilöltä kulunut aika vastataukseen. Käyttämällä kaksisuuntaista t-testiä tilastollisesti merkitsevät vastausnopeudet 5% riskitasolla olivat:

- Taivutukset
  - Kontrollikaavio on parempi kuin b+.
- Särmien pituuksien vaihtelu
  - Kontrollikaavio on parempi kuin ev+.
  - Kontrollikaavio on parempi kuin ev-.
- Virransuunta
  - Kontrollikaavio on parempi kuin f+ (tässä $p = 0.058$ lähestyy merkitsevää tulosta).
  - f- on parempi kuin f+.

Tuloksista ilmenee, että kaavio, jossa koetettiin minimoida taivutuksien määrä (b+) tuotti huonompia suorituksia koehenkilöillä kuin kontrollikaavio, jossa oli keskinkertaisesti taivutuksia. Tämä oli yllättävä tulos, sillä aikaisemmat tutkimukset ovat osoittaneet sovellusriippumattomien kaavioiden tulkitsemisen helpommaksi vähemmillä särmien taivutuksilla [Purchase, 1997]. Myös tutkimus käyttäjien mieltymyksistä UML-estetiikkoihin osoitti, että koehenkilöt eivät pitäneet taivutuksista [Purchase et al., 2000]. Yksi mahdollinen selitys löytyy kaavion ortogonaalisuudesta, sillä taivutusten minimoiminen vähentää graafin ortogonaalista rakennetta ja niiden lisääminen taas parantaa rakennetta tältä kannalta.

Kontrollikaavio, jossa oli keskinkertainen määrä eri mittaisia särmiä tuotti paremman tuloksen kuin kaavio, jossa kaikki särmät olivat saman mittaisia (ev+) ja paremman tuloksen kuin kaavio, jossa osa särmistä oli todella lyhyitä ja osa todella pitkiä (ev-). Tämäkin oli yllättävä tulos, sillä odotuksena oli, että saman mittaisia särmiä käyttävä kaavio tuottaisi parempia tuloksia kuin kontrollikaavio tai suuria särmän pituuksien vaihteluita sisältävä kaavio. Purchase ja muut oli sitä mieltä, että tasaisia särmänpituuksia sisältävä kaavio on vaikeampi ymmärtää, koska särmän pituudella on oma kognitiivinen tarkoituksensa.

Myös virransuunta tuotti ei-odotettuja tuloksia, sillä kaaviot, joissa enemmistö suunnatuista särmistä osoitti ylöspäin (f+) aiheutti koehenkilöille enemmän tulkitsemisvaikeuksia kuin kaavio, jossa särmiä ei koetettu ohjata samaan suuntaan (f-). Aikaisempi tutkimus UML-luokkakaavioiden syntaksista [Purchase et al., 2001] osoitti, että luokkakaavioiden tulkintaa helpotti, jos yliluokka asetetaan aliluokan päälle. Tälle oudolle tulokselle ei pystytty osoittamaan mitään järkevää selitystä.

Purchase ja muut [2001b] tämän kokeen jälkeen siihen tulokseen, että laskennallisten metriikoiden tuottamat estetiikat eivät ole verrattavissa ihmisen ymmärtämään esteettisyyteen kaavioista. Esimerkiksi ortogonaaliset metriikat mittaavat paikkoja ruudukolla, joihin solmut ja särmät asetetaan ja ihmisen havainnointi ortogonaalisuudesta ei välttämättä vastaa metriikan tuottamia numeerisia arvoja.

5.3. Koe B

Kaavion aihealue, UML-ohjekirja, valmiit esimerkit, valmistautumisaika, vastausohjelma ja tulostenkeräystapa, oli sama kuin kokeessa A. Ainoa ero koejärjestelyissä oli aikaraja, joka edellisessä kokeessa oli 50 sekuntia ja kokeessa B 40 sekuntia. Tämä muutos tehtiin, koska kaaviot, joita kokeessa B käytettiin, muodostettiin ihmisten haastattelujen tuloksena toisin kuin laskennallisten metriikoiden keinoin ja olivat täten koehenkilöille helpompilukuisia.

Kokeessa B tarkasteltiin samat estetiikat kuin kokeessa A ja lisäksi kaksi uutta estetiikkaa, joiden ajateltiin vaikuttavan suoritusnopeuteen. Nämä kaksi uutta estetiikkakriteeriä olivat:

- Särmien pituus, jota kokeessa A tarkasteltiin vain pituuden vaihtelujen muodossa. Koska kokeen A tulokset osoittivat, että parhaita tuloksia saadaan keskinkertaisella särmien pituuksien vaihteluilla, niin tässä kokeessa pyrittiin tarkastelemaan tätä tilannetta lähemmin.
- Symmetria graafin rakenteen osalta on parempi ajatella havainnoinnin kuin laskennallisuuden avulla. Symmetrian laskennallinen määritelmä, joka ainoastaan huomioi solmujen välisen geometrisuuden vertikaaliseen ja horisontaaliseen akseliin nähden, ei ota huomioon paikallisia symmetrioita. Ihmiselle ei myöskään ole tärkeää, onko tarkasteltava solmua esittävä laatikko pikselin tai kaksi alusruudukon reunan toisella puolella. Laskennallinen algoritmi, joka ottaa huomioon kaikki paikalliset symmetriat ja havainnon toleranssit olisi laskennallisesti monimutkainen ja pystyisi tuottamaan vain karkean mallin siitä, jota ihminen pitää havainnollisesti symmetrisenä.

5.3.1. Koekaaviot

Kokeessa A käytettiin yhtä kontrollikaaviota tulosten vertailemiseen. Kokeeseen B muodostettiin erillinen kontrollikaavio jokaista estetiikka kohden. Tulosten tulkitseminen tehtiin jokaiselle estetiikkakriteerille erikseen, koska se oli mahdollista toteuttaa kokeessa B toisin kuin kokeessa A. Täten luotiin käsin jokaiselle estetiikalle kolme kaaviota: vähäinen vaikutus (-), keskinkertainen vaikutus (0) ja suuri vaikutus (+).

Estetiikkojen väliset vaihtelut mitattiin koehenkilöryhmällä, joka oli erillinen varsinaisesta koehenkilöryhmästä. Henkilöiden tuli arvottaa kolme kaaviota estetiikan havainnoinnin mukaan. Esimerkiksi koehenkilölle näytettiin n+, n0 ja n- -kaaviot ja häntä pyydettiin arvottamaan ne solmujaon mukaan. Toivottiin vielä, että samojen kaavioiden jakaminen symmetrian mukaan tuottaisi ongelmia koehenkilölle, jotta voitaisiin erottaa eri estetiikkojen väliset mittaustulokset paremmin. Tämä vastaa sitä kokeessa A käytettyä metriikkatapaa, että pyrittiin asettamaan muut estetiikat samaan tasaiseen painotukseen.

Särmien taivutukset ja virransuunta jätettiin mittaamatta, koska ne ovat enemmänkin laskennallisen estetiikan arvoja, sillä on laskettavissa, montako taivutusta ja montako särmää osoittaa ylöspäin kaaviossa. Näitä estetiikkoja huomioitiin kumminkin symmetrisyyden ja ortogonaalisuuden seassa.

Koehenkilöille esitettiin 21 oikeaa ja 10 väärää kaaviota kokeen yhteydessä. Väärät kaaviot muodostettiin vaihtamalla satunnaisesti yhden relaation alkua ja loppua. Tämä ei vaikuttanut koetulokseen, sillä väärien kaavioiden rakenteella ei ollut merkitystä.

5.3.2. Tulokset ja päätelmät

Toisin kuin kokeessa A, jossa mitattiin vain nopeutta, otettiin kokeen B tuloksissa huomioon myös tarkkuus. Tämä ehkä juuri alennetun suoritusajan takia, joka johti useampiin virheisiin kuin kokeessa A. Kuten kokeessa A käytettiin tässäkin kaksisuuntaista t-testiä 5% riskitasolla, josta saatiin seuraavat tulokset:

- Taivutukset
  - b0 on nopeampi kuin b-.
  - b+ on nopeampi kuin b-
  - b+ on tarkempi kuin b0 (tässä $p = 0.057$ lähestyy merkitsevää tulosta).
- Särmien pituuksien vaihtelu
  - ev+ on nopeampi kuin ev0.
  - ev- on nopeampi kuin ev0.
  - ev+ on tarkempi kuin ev0.

Testeissä huomattiin, että taivutusten määrän vähentäminen tuotti nopeimmat tulokset. Myös kaavio, jossa oli pyritty vähentämään taivutusten määrää, tuotti tarkempia tuloksia 5% riskitasolla kuin keskimääräisesti taivutuksia sisältävä kaavio. Nämä tulokset ovat yhtäpitäviä edellisten tutkimusten kanssa [Purchase, 1997; Purchase et al., 2000].

Särmien pituuksien vaihteluissa huomattiin, että keskimääräisesti eri vaihteluita sisältävä kaavio tuotti kaikkein hitaimmat tulokset. Tämä on täysin vastakkainen tulos verrattuna kokeessa A saatuun tulokseen, jossa keskinkertainen kaavio oli kaikkein nopein. Nämä konfliktit särmien pituuden vaihtelussa johdattavat mieleen, että on otettava huomioon muita tekijöitä. Kokeessahan ei huomioitu särmien pituuksien merkitystä tai solmujaon estetiikkaa.

Kaavioissa ei pyritty tekemään mitään erityistä jakoa semanttisiin ryhmiin, vaan solmut aseteltiin mielivaltaisesti kaavioon. Näyttää siltä, että solmujaon tai särmien pituudella ei ole väliä tällaisessa asetelmassa. On täysin mahdollista, että koehenkilöiden suoritusnopeuteen vaikuttaisi, jos solmuja ei olisi aseteltu mielivaltaisesti, vaan ne olisi aseteltu semanttisesti siten, että toisiinsa läheisesti liittyvät solmut olisivat lähellä toisiaan, vaikka niitä ei edes yhdistäisi eksplisiittinen särmä.

5.4. Johtopäätelmät

Kahden erilaisen koeversion ja muutaman konkreettisen tuloksen jälkeen Purchase ja muut päätyivät siihen yllättävään tulokseen, että millään testeissä käytetyllä heuristiikalla ei ole oikeastaan merkitystä ja sen takia ei ole ihmisen kaavionluvun kannalta eroja, mitä estetiikkakriteeriä UML-luokkakaavioita piirtävä algoritmi käyttää. Tosiasioita huomioon jättämättä Purchase ja muut [2001b] uskovat, että on olemassa

ylimääräisiä semantiikkaa koskevia asioita, joita pitää ottaa huomioon graafin rakenteenluontialgoritmeissa aihekohtaisissa työkaluissa.

Tuloksista käy ilmi, että kaavion luettavuuteen ei liity pelkästään tasainen solmujako kuvitellun laatikon sisällä, särmien pituus tai särmien pituuksien vaihtelun minimoiminen, vaan se vaatii myös jotain muuta. Purchase ja muut esittävät, että ylimääräinen asia, jota tulee ottaa huomioon, on toisiinsa liittyvien objektien semanttinen jäsennys. Jopa särmien taivuksiin liittyvät yllättävät tulokset voidaan selittää ryhmien semanttisen rakenteen hajottamisella, joka voi johtua taivutusten äärimmäisestä minimoinnista. Esimerkiksi voi olla hyödyllistä lisätä joitain taivutuksia kaavioon, jos se tuo aliluokat lähelle toisiaan perimyshierarkiassa.

Purchase ja muut kritisoivat omaa empiiristä tutkimustaan, koska heidän mielestään olisi näiden tulosten pohjalta järkevämpää tehdä kaavioiden semanttisuudelle samantyyppinen koe. Heidän mielestään myös koehenkilöt eivät olleet aivan oikeanlaisia UML-kaavioiden lukijoiksi, koska he olivat yliopisto-opiskelijoita eivätkä ohjelmistoinsinöörejä. Myös kaavioiden olisi heidän mielestään pitänyt esittää jotain realistisempaa mallia, jota olisi esimerkiksi ohjelmistosuunnittelijoiden helpompi tulkita.

Graafinpiirtoalgoritmia valittaessa pitää sen tarkoituksenmukaisuutta harkita. Erilaiset geneeriset algoritmit voivat tuottaa visuaalisesti kauniita rakenteita, mutta eivät välttämättä intuitiiviseen käyttöön riittäviä. Algoritmit, jotka on tehty pelkästään UML-kaavioiden luomiseen ja joissa on otettu semantiikka huomioon, ovat varmasti paljon tehokkaampia ihmisten käsityskyvyn näkökulmasta.

6. Topologia, muoto ja metriikka -algoritmi

Topologia, muoto ja metriikka -nimitys on peräisin Di Battistalta ja muilta [1999]. Tähän lähestymistapaan törmäsimme luvussa 3 ja siellä kävimme läpi vaiheet, jotka tehdään saattaaksemme graafin ortogonaaliseen muotoon. Ensimmäinen askel oli graafin tasoon piirtäminen, seuraavaksi tuli ortogonalisointiaskel ja viimeiseksi tiivistettiin. Seuraavaksi esittelemme algoritmin, joka käyttää kyseistä tekniikkaa ja pyrkii ottamaan huomioon estetiikkakriteerit, jotka olivat esillä luvussa 5. Luvussa 5 kävi ilmi, että estetiikkakriteereissä täytyy ottaa huomioon myös inhimillinen ajattelu siten, että solmut, jotka ovat vahvasti tekemisissä keskenään, olisivat myös piirroksessa lähellä toisiaan. Nämä kaikki asiat on otettu huomioon algoritmissa, jonka kehittelivät Eiglsperger ja muut [2003].

Automaattisen rakenteen muodostavan algoritmin tulee ottaa huomioon paljon enemmän kaaviosta kuin vain sen ulkoasun, kuten näimme luvussa 5. Ensimmäiset topologiaan, muotoon ja metriikkaan perustuvat algoritmit eivät pystyneet täyttämään kaikkia näitä vaatimuksia, mutta viime aikoina on pystytty täyttämään nämä aukot. Eiglspergerin ja muiden [2003] algoritmi tuottaa automaattisen rakenteen topologia, muoto ja metriikka -periaatteella. Seuraavaksi listataan olennaisia tekniikoita, jotta Eiglsperger ja muut ovat päässeet päämääräänsä:

- **Solmun koko:** Jotta pystytään pitämään solmujen koko määrätyissä rajoissa toisessa ja kolmannessa askeleessa, täytyy ottaa huomioon erityisiä vaatimuksia. Annettu solmun koko voimaan taata *Kandinsky*-mallilla, jota kutsutaan joskus myös nimellä podevsnef-malli. Tässä mallissa muodon täytyy olla yhdenmukainen tiettyjen sääntöjen kanssa. Kandinsky-algoritmi laskee minimitaivutusten määrän Kandinsky-mallissa [Fößmeier and Kaufmann, 1996]. Kolmannessa askeleessa solmun kokoon liittyvä ehto hoidetaan Di Battistan ja muiden [1999] algoritmilla.
- **Suunta:** Jos ensimmäisessä askeleessa ei käsitellä edellisestä luvusta tuttua virransuuntaa, sitä seuraavat askeleet eivät välttämättä pysty muodostamaan piirrosta, joka ottaa huomioon tämän estetiikkakriteerin ja siten eivät kaikki särmät välttämättä osoita ylöspäin. Tällöin ei ole muuta tehtävissä kuin yrittää minimoida tällaisten särmien määrä. Tällainen lähestyminen on otettu näennäisnousevista piirroksista [Bertolazzi et al., 2002]. Näennäisnousevissa piirroksissa kaikki särmät eivät osoita ylöspäin, mutta ylöspäin suunnatut särmät lähtevät solmun ylemmästä puolikkaasta ja päätyvät solmun alempaan puolikkaaseen. Jos haluamme välttää ei-ylöspäin suunnatut särmät, niin ensimmäisen askeleen tarvitsee täyttää erityisiä vaatimuksia. Eiglsperger ja muut [2003] esittävät ensimmäiseen askeleeseen heuristiikan, joka tuottaa sekoitetun nousevan sulautuksen syötegraafista. Heuristiikka muodostaa sellaisen piirroksen, joka noudattaa virransuuntaestetiikkakriteeriä ja

ortogonaaliaskeleessa voidaan laskea tälle piirrokselle särmien kulmat ja taivutukset siten, että kaikki särmät osoittavat ylöspäin.

- **Nimiöt:** Yleensä graafinpiirtoalgoritmit lisäävät nimiöt kaavioon erillisesti vasta laskennallisten askelien jälkeen käyttämällä karttanimiöintitekniikkaa [Wagner and Wolff, 1998], jota käytetään esimerkiksi kaupunkien nimien automaattiseen asetteluun maantieteellisellä kartalla. Nimiöinti voidaan sisällyttää tiivistysaskeleeseen, jolla yleensä saadaan parempia tuloksia kuin karttatekniikalla. Nimiöiden asettelulle löytyy lineaarisen ohjelmointiin perustuva algoritmi [Klau and Mutzel, 1999; Binucci et al., 2002] ja särmien nimiöintiin löytyy myös heuristiikka [Binucci et al., 2001]. Kartan nimiöinti on NP-kova ongelma [Wagner and Wolff, 1998].
- **Klusterointi:** Kaikissa kolmessa askeleessa täytyy ottaa huomioon klusterointi. Fenging ja muiden [1995] esittelemä c-tasoon piirtäminen laajentaa tasoon piirtämisen klusteriuiduille graafeille. Di Battista ja muut [2001] esittävät c-tasoon piirtämisalgoritmin yhdessä ortogonaalisointialgoritmin hahmotelman kanssa. Cortese ja Di Battista [2005] toteavat, että yhdistetty klusteriuitu graafi $C(G,T)$ on c-tasoon piirrettävissä, jos ja vain jos G on tasoon piirrettävissä, eikä ole olemassa piirrosta graafista G siten, että jokaiselle puun T solmulle v, kaikki graafin $G - G(v)$ solmut ja särmät ovat piirroksen $G(v)$ ulkopuolella..

6.1. Algoritmin taustaa

Edellä mainittujen teknikoiden käyttö yhdessä ainoassa algoritmissa on ongelmallista. Kuten luvussa 3 on todettu, niin myös tässä eri tekniikat ovat ristiriidassa keskenään. Toinen ongelma, jonka edellä mainitut algoritmit tuottavat, on suorituskyvyn riittämättömyys, koska automaattisen rakenteen tuottavia algoritmeja käytetään pääsääntöisesti graafisella käyttöliittymällä, niin täytyy algoritmin toimia niin nopeasti, että se ei vaikututa tietokoneen ja käyttäjän väliseen vuorovaikutukseen.

Klusteroinnin ja suunnan toteutukset ovat ristiriidassa keskenään, koska molemmat pyrkivät laajentamaan tasoon piirtämistä eri suuntiin: c-tasoon piirtäminen klusteroinnissa ja näennäisnouseminen suunnassa. Tämän takia voidaan määritellä näennäisnouseva c-tasoon piirtäminen, jossa yhdistetään molemmat samojen parametrien alle toimimaan vierekkäin, mutta tällä hetkellä ei ole olemassa vielä algoritmia tähän. Eiglsperger ja muut [2003] kuitenkin uskovat, että sellaisen kehittäminen ei ole liian monimutkaista. Tällä hetkellä tulee heidän mielestään kuitenkin keskittyä pelkästään joko hierarkkisuuteen tai klusterointiin. Tässä kuitenkin voimme huomioida luvussa 4 annettu heuristiikka, joka ottaa huomioon molemmat samaan aikaan hierarkkisella graafilla ja energiamallilla. Ehkäpä energiamallit ovat joustavampia graafinpiirtämiseen kuin metriikoita käyttävät algoritmit. Seuraavissa luvuissa tarkastelemmekin lähemmin evolutiivisia ja energiamalleihin perustuvia

algoritmeja, mutta tässä kuitenkin käymme läpi vielä metriikoita käyttävän menetelmän, että saamme yleisen käsityksen graafinpiirtoalgoritmeista eri tekniikoilla.

Eiglspergerin ja muiden [2003] algoritmi olettaa, että syötteenä annetaan luokkakaaviograafi G = (V, E) ja joukko nimiöitä L sekä solmujen koon kuvaus $S : V \cup L \to \mathbb{N}^2$, joka ottaa huomioon nimiöiden koon. Jokaiselle solmulle, särmälle ja nimiölle asetetaan tyyppi kuvaamaan sen ominaisuuksia:

- **Solmut:** luokka, rajapinta tai muu.
- **Särmät:** riippuvuus, yleistys, assosiaatio tai muu.
- **Nimiöt:** monimuotoisuus, rooli, nimi tai muu.

Tyyppiä muu käytetään yleisenä tyyppinä objekteille, joihin ei mikään muu tyyppi päde. Transformaatioaskeleessa luokkakaavion graafisyöte muutetaan sopivaksi algoritmille, jolloin saadaan:

- Graafi $G = (V, E)$.
- Osajoukko $D \subseteq E$ esittämään suunnattuja särmiä.
- Osajoukko $H \subseteq E$ esittämään niitä särmiä, jotka ovat potentiaalisia hypersärmän osia.
- Nimiöiden joukko L yhdessä kuvauksen $T : L \to \{source, center, target\}$ kanssa esittämään haluttuja paikkoja särmien vieressä nimiöille.
- Kuvaus $S : V \cup L \to \mathbb{N}^2$ esittämään solmujen kokoa nimiöiden suhteen.

Transformaatiolle asetetaan parametrit, joilla voidaan käyttää tehtäessä valintaa eri tyylien välillä. Esimerkiksi yksi mahdollinen tyyli on, että kaikki yleistysrelaatiot suunnataan ja piirretään ne hypersärminä. Toinen mahdollisuus on suunnata kaikki assosiaatiot ja jättää yleistykset koskemattomiksi. Algoritmi tekee erottelun luokkien ja rajapintojen välille, joten niille voidaan antaa erilaiset tyylit. Tämä on erityisesti hyvin käyttökelpoista Java-ohjelmointikielellä tehtyjen luokkien piirtämisessä. Transformaatioaskeleessa asetetaan jokaiselle nimiölle suositeltu sijainti. Nimiöt, joiden tyyppi on monimuotoisuus tai rooli, asetellaan aina assosiaation loppuun. Ne saavat siten suositeltuna sijaintina joko lähteen (engl. source) tai maalin (engl. target). Kaikki muut nimiöt saavat suositeltuna sijaintina keskustan (engl. center).

Mainitut tyyppi- ja transformaatiosäännöt tulee nähdä enemmänkin esimerkkeinä kuin määritelminä. Käyttämällä välillistä transformaatioaskelta helpotetaan mallin elementtien lisäintegraatiota korostamalla tyyppitaulua ja transformaatiota. Itse transformaatioaskel vaikuttaa pääestetiikkakriteeriin kaaviossa. Jos ei ole yhtään suunnattua särmää tai hypersärmää määriteltävänä, niin pyritään minimoimaan särmien leikkauksia, muuten leikkauksien minimointi jätetään sekundääriseksi virransuunnan ja/tai hypersärmien määrittelyjen taakse.

6.2. Algoritmi

Eiglspergerin ja muiden [2003] algoritmin ydin piilee uudessa ortogonaalisoinnissa, joka käyttää estetiikkakriteereitä virransuunta ja n:n assosiaation keskittäminen. Kun UML-luokkakaaviossa kolme tai useampia luokkia ovat assosiaatiossa keskenään, niin niistä lähtevä assosiaatioviivat yhtyvät timantinmuotoisessa objektissa, ja tätä kutsutaan n:n assosiaatioksi, sekä myöhemmin pelkästään keskitysestetiikkakriteeriksi, koska tätä termiä Eiglsperger ja muut [2003] käyttävät.

Oletetaan, että predikaatilla H indusoitu suunnattu graafi on syklitön. Tämä on perusteltua, sillä yleistysrelaatio luokkien välillä on syklitön määritelmän perusteella ja hypersärmänotaatio on rajoitettu yleistysrelaatioon UML-luokkakaavioissa [OMG, 2007]. Jos suunnattu graafi sisältää syklejä ja se on indusoitu osajoukoilla H ja D, käytetään samanlaista strategiaa kuin hierarkkisen graafin piirtämisessä: piirretään ainoastaan särmien osajoukko, joka indusoi syklittömän aligraafin ylöspäin. Tällöin täytyy poistaa ainoastaan särmät osajoukosta D, koska osajoukolla H indusoidu suunnattu graafi on syklitön.

Algoritmi voidaan kirjoittaa seuraavassa muodossa:

1. **Esikäsittely**
   a) Ensimmäisessä esikäsittelyaskeleessa graafi jaetaan yhtenäisiin komponentteihin. Jokainen yhtenäinen komponentti käsitellään erikseen algoritmissa, joten voimme olettaa seuraavissa askelissa graafin olevan yhtenäinen.
   b) Poistetaan särmiä joukosta D, kunnes $D \cup H$ indusoi syklittömän aligraafin graafista G.
   c) Seuraavaksi korvataan sellaiset särmät joukossa H, että ei jätetä hypersärmiä, vaan ne korvataan ns. paikanpitimellä kuten luvussa 3 esitettiin graafin tasoon piirtämisen yhteydessä.
   d) Jolleivat joukon D särmät indusoi yhtenäistä aligraafia, niin joitakin särmiä lisätään väliaikaisesti joukkoon D käyttämällä pienimmän virittävän puun muodostavaa algoritmia.

2. **Tasoon piirtäminen**
   Algoritmia, jonka Eiglsperger ja Kaufmann [2001] ovat jo kuvanneet aikaisemmin, käytetään laskemaan sekoitetun nousevan tasoon piirron syötegraafista. Kaikkia särmiä joukossa $D \cup H$ pidetään suunnattuina. Tämä algoritmi käsittelee myös särmien painoja, joten täysin energiamallien tekniikoita käyttämättä ei ole tämäkään algoritmi. Joukon H särmille annetaan suurimmat painot, jonka jälkeen painotetaan joukon D särmät.

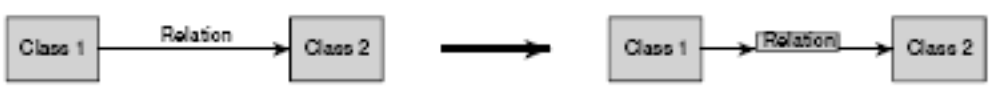
3. **Ortogonaalisointi**
   Ortogonaalisointi käyttää algoritmia, joka on kuvattu kohdassa 6.4.

4. **Tiivistäminen**
   a) Särmien nimiöt laitetaan piirroksessa haluttuun paikkaan keskitysestetiikkakriteerillä. Tästä kuvaus kohdassa 6.2.1.
   b) Eiglspergerin ja Kaufmannin kuvaamaa algoritmia käytetään laskemaan alustava piirros.
   c) Näkyvyysjälkikäsittelyä, joka tunnetaan VLSI-piirien rakenteen suunnittelusta, käytetään vähentämään piirroksen käyttämää aluetta ja särmien pituutta.

5. **Jälkikäsittely**
   a) Kaikki ylimääräiset solmut poistetaan mukaan lukien leikkaukset, nimiösolmut, hypersärmäsolmut ja keinotekoiset taivutukset.
   b) Särmien nimiöt asetetaan paikalleen, joko lähteen tai maalin mukaan kartta-algoritmilla.
   c) Yhtenäisten komponenttien rakenne graafissa järjestetään lattiasuunnittelualgoritmilla [Freivalds et al., 2001].

6.3. Nimiöiden asettelu

Särmien nimiöiden halutut kohteet lähde ja maali jätetään huomioimatta ja ne asetellaan käyttämällä karttanimiöintialgoritmia rakenteen muodostamisen jälkeen. Tämä menetelmä toimii hyvin luokkakaavioille, koska nimiöt ovat suhteellisen pieniä. Moninaisuusrelaatiossa on esimerkiksi ”1..n” tai ”*”. Särmät, jotka asetellaan keskitysestetiikalla käsitellään, eri tavalla. Ennen tiivistysaskelta jaetaan särmä, johon nimiö kuuluu, lohkoihin. Sen jälkeen valitaan lohko keskeltä särmää, jaetaan se kahteen osaan ja liitetään ne yhteen paikanpitosolmulla. Tämä solmu on yhtä suuri kuin itse nimiö. Tiivistysaskeleen jälkeen nimiö asetetaan solmun kohtaan ja solmu poistetaan graafista. Tästä on esitetty esimerkki kuvassa 6.1.



Kuva 6.1. Nimiöt joihin toteutetaan keskitysestetiikkaa käsitellään solmujen tavoin.

6.4. Ortogonalisointi

Ortogonalisointi on mielenkiintoisin aihe Eiglspergerin ja muiden algoritmissa, mutta ilman edellä mainittuja asioita koko algoritmin monimutkaisuuden hahmottaminen olisi jäänyt suppeaksi. Luvussa 3 käsiteltiin yleisesti topologia, muoto ja metriikka -heuristiikkaa, mutta ortogonaalisointi jätettiin vain kuvailun varaan, joten tässä on tarkasti kuvattu yksi mahdollinen, mutta sääntöjä noudattava tapa.

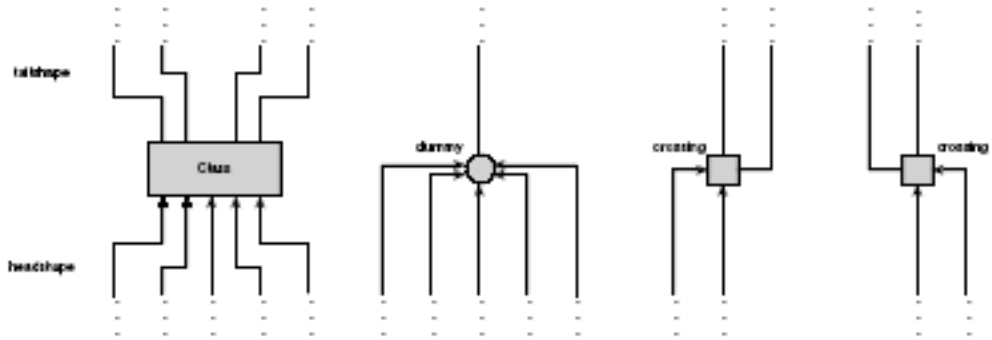
Kuten aikaisemminkin on mainittu, ortogonaalisointi laskee kulmat ja taivutukset piirrokselle. Graafisyöte tässä tapauksessa on sekoitettu nouseva tasoon piirretty luokkakaaviograafi. Oletetaan, että paikanpitosolmut on asetettu leikkausten tilalle sen mukaisesti. Oletetaan myös, että aligraafi, joka on indusoitu suunnatuilla särmillä, on yhtenäinen, sillä tämä tehdään kohdassa 1d.

Perinteisesti ortogonaalisointialgoritmit pyrkivät minimoimaan taivutusten määrää. Tässä esitetty versio pyrkii aluksi optimoimaan virransuunta- ja keskitysestetiikkaa, taivutusten määrä on sekundäärinen. Tämä saavutetaan toteuttamalla ortogonaalisointi kahdessa vaiheessa: ensiksi lasketaan ortogonaalisointi suunnatuille särmille ja hypersärmille ja seuraavaksi jäljelle jääneet särmät ortogonaalisoitaan.

Kuten jo mainittiin, ensimmäinen vaihe työstää suunnattujen särmien ja hypersärmien indusoimaa aligraafia. Tulosteena tuotetaan merkkijono aakkosista {←,↑, →} jokaiselle aligraafin särmälle. Tämä merkkijono kuvaa särmän lohkot ja niiden orientaatiot. Eiglsperger ja muut [2003] kutsuvat tätä merkkijonoa särmän *muodoksi*. Muodot on määrätty siten, että kaikki solmuun tulevat särmät ovat samalla puolella ja kaikki solmusta lähtevät särmät ovat vastakkaisella puolella kuin tulevat.

Algoritmi toimii seuraavasti: jokaiselle särmälle lasketaan häntä- ja päämuoto. Nämä muodot yhdistetään mukailemalla koko särmän muotoa. Algoritmi valitsee yhden solmun kerrallaan ja asettaa solmuun tuleville särmille päämuodon ja solmusta lähteville särmille häntämuodon. *Kandinsky*-mallin mukaan on olemassa vain yksi lähtevä ja yksi tuleva särmä, joka ei taivu. Valitaan tämä taipumaton särmä ja noudatetaan seuraavia sääntöjä: jos on olemassa särmiä, jotka ovat osa hypersärmää, valitaan mediaanisärmä näiden särmien asettelun mukaan. Jos ei löydy särmiä, jotka ovat osa hypersärmää, valitaan mediaanisärmä asettelun mukaan vain, jos särmien lukumäärä on parillinen, muuten parittomassa tapauksessa ei valita taipumatonta särmää. Tämä tehtävä on havainnollistettuna kuvassa 6.2. Häntä- ja päämuodot voidaan aina yhdistää, koska muotojen antaminen tapahtuu särmille niin, että häntämuodot loppuvat aina merkkiin ↑ ja päämuodot alkavat samalla merkillä ↑. Muodon asettamiselle särmienleikkauksessa voidaan antaa kaksi eri tapaa, jotka ovat symmetrisiä. Molemmissa tavoissa toinen särmistä taipuu ja toinen pysyy suorana. Särmille annetut painot ratkaisee kumpi särmistä taipuu ja kumpi ei. Kevyemmät särmät taipuvat särmienleikkauksessa. Joten aina joukon *D* särmät taipuvat joukon *H* särmien painotuksessa ja ainoastaan hypersärmät taipuvat painavampien hypersärmien vuoksi.

Algoritmissa 6.3 merkintä *l* tarkoittaa listaa särmiä, jotka ovat vierekkäisiä solmulle *v*. Särmät on lueteltu listassa *l* vasemmalta oikealle sekoitetun nousevan asettelun mukaan.



Kuva 6.2. Muotojen asettelu suunnatuille särmille ja hypersärmille.

Input: Upward planarization $G=(V,E)$
Output: A function $shape: E \to \{\leftarrow, \uparrow, \rightarrow\}^*$

// Tailshape
for $v \in V$ **do**
  $l = \{e \in E \mid e=(v,w), e \in D \cup H\}$
  $m = median(l)$
  $shape(l_m) = \rightarrow \uparrow \rightarrow$
  if *v is crossing* ***then***
    if $m=0$ ***then*** $shape(l_1) = \rightarrow \rightarrow \uparrow \rightarrow$
    else $shape(l_0) = \rightarrow \leftarrow \uparrow \rightarrow$
  else
    for $0 \le i < m$ ***do*** $shape(l_i) = \rightarrow \uparrow \leftarrow \uparrow \rightarrow$
    for $m < i < |l|$ ***do*** $shape(l_i) = \rightarrow \uparrow \rightarrow \uparrow \rightarrow$
  end
end
// Headshape
for $v \in V$ ***do***
  $l = \{e \in E \mid e=(w,v), e \in D \cup H\}$
  $m = median(l)$
  if *v is crossing* ***then***
    if $m=0$ ***then***
      $shape(l_1) += \rightarrow \leftarrow \rightarrow$
    end
    else $shape(l_0) += \rightarrow \rightarrow \rightarrow$

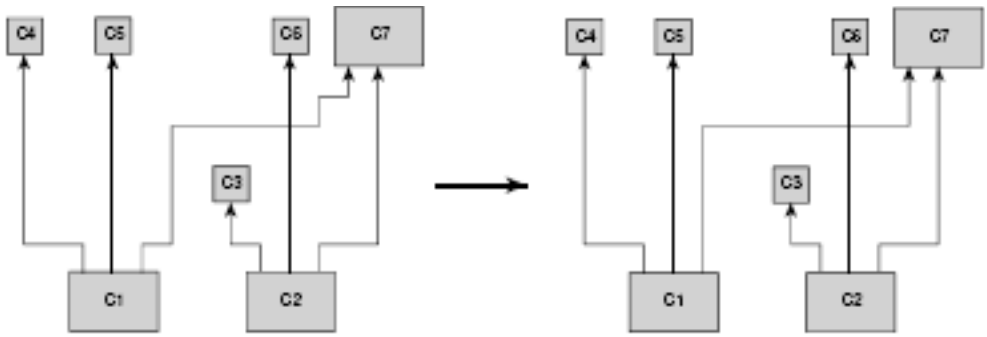
else
if *v is hyperedge node* ***then***
for $0 \le i < m$ ***do*** $shape(l_i) += \rightarrow\rightarrow\rightarrow$
for $m < l < |l|$ ***do*** $shape(l_i) += \rightarrow\leftarrow\rightarrow$
else
for $0 \le i < m$ ***do*** $shape(l_i) += \rightarrow\rightarrow\uparrow\rightarrow$
for $m < i < |edges|$ ***do*** $shape(l_i) += \rightarrow\leftarrow\uparrow\rightarrow$
end
end
end

Algoritmi 6.3. Muodon laskeminen särmälle [Eiglsperger et al., 2003].

Tämän tehtävän jälkeen suoritetaan taivutuksien suoristaminen, jotta saadaan taivutuksien määrää vähennettyä. Taivutuksiensuoristusoperaatio poistaa turhat taivutukset särmästä uudella muotoilulla poistamatta alku- tai loppusuuntaa särmästä. Taulukko 6.4 esittää suoristamisoperaatioita ja kuvassa 6.5 näytetään kuinka suoristamisoperaatiot 2 ja 3 vähentävät taivutuksia särmässä (C1,C7) neljällä.

shape	*v* is crossing	*w* is crossing	new shape
↑→↑→↑			↑→↑
↑→↑→		X	↑→
→↑→↑	X		→↑
→↑→	X	X	→
↑←↑←↑			↑←↑
↑←↑←		X	↑←
←↑←↑	X		←↑
←↑←	X	X	←

Taulukko 6.4. Taivutuksiensuoristusoperaatio särmälle $e = (v,w)$.



Kuva 6.5. Esimerkki taivutuksiensuoristamisoperaatiosta.

6.5. Pohdintaa

Eiglsperger ja muut [2003] toteavat, että vaikka heidän algoritmistaan puuttuu tällä hetkellä klusterointi, niin tämä olisi mahdollista toteuttaa raskaammilla särmäpainoilla.

Suoritusaikaa kuluttaa eniten sekoitettu nouseva tasoon piirtäminen ja virransuunta ortogonaaliaskeleessa. Jos n on solmujen lukumäärä, m särmien lukumäärä ja c särmien leikkausten lukumäärä laskettuna tasoon piirto askeleessa, niin tasoon piirtämisen aikakompleksisuus on $O(nm^2+(n+c)^2 m)$ ja virransuunnan laskemiseen kuluu $O((n+c)^2 \log(n+c))$.

Eilgsperger ja muut [2003] vertaavat algoritmiaan hierarkkiseen lähestymistapaan ja toteavat, että topologia, muoto ja metriikka tarjoavat enemmän joustavuutta, kun huomioidaan estetiikkakriteereitä, koska hierarkkinen tapa on noudattaa pelkästään virransuuntaa. He toteavat, että hierarkkisella tavalla on etuja, kun piirroksessa on paljon särmienleikkauksia tai jos särmien hierarkkisuus noudattaa jo valmiiksi virransuuntaa vahvasti. Yksi hierarkkisen tavan vahvuuksia on klusterointi.

7. Geneettinen algoritmi

Geneettistä etsintäalgoritmia käytetään yleensä optimointiin ja oppimiseen [Goodman, 2007]. Algoritmit ovat sattumanvaraisia, mutta eivät umpimähkäisiä. Voidaan ajatella, että algoritmi jäljittelee evoluutiossa tapahtuvaa sopivimman selviämistä. UML-luokkakaavioita voidaan luoda myös käyttämällä geneettistä algoritmia [Gudenberg et al., 2006].

Klassinen esitys geneettiselle algoritmille on merkkijonoista koostuva populaatio. Jokaista merkkijonoa kutsutaan kromosomiksi ja ne muodostuvat biteistä. Esimerkiksi kymmenen bitin merkkijono 1011101010 voisi esittää mahdollista ratkaisua ongelmaan. Yksi bitti tai osa biteistä voi esittää jotain tiettyä ominaisuutta, kuten ruskeasilmäisyyttä, jos kysymys on henkilöstä. Jokaista koodattua ominaisuutta kutsutaan geenikohdaksi ja kaikkia mahdollisia arvoja geenikohdassa kutsutaan alleeliksi.

Merkkijonon hyvyys (sopivuus) tai sen optimaalisuus määrittelee, kuinka paljon juuri tämä merkkijono tulee vaikuttamaan etsintäprosessiin tulevaisuudessa. Evoluutiossa puhutaan sopivimman selviämisestä. Hyviä ratkaisuja käytetään uusien ratkaisujen generoimiseen, jotka voivat mahdollisesti olla yhtä hyviä tai jopa parempia. Populaatio pitää muistissa koko ajan kaiken mitä olemme oppineet ratkaisusta kaikissa kohdissa. Jokaiselle kromosomille on pystyttävä laskemaan hyvyys (sopivuus), jolla voidaan mitata kuinka lähellä tavoitetta olemme. Geneettisen algoritmin klassiset perusoperaatiot ovat mutaatio, valinta ja risteytys [Goodman, 2007]:

Mutaatio

- Toteutetaan yhdelle ”vanhempi”kromosomille.
- Tuottaa muuttuneen jälkeläisen.
- Klassisessa esitystavassa käännetään yksi bitti merkkijonossa.
- Merkkijono 1101000110 voisi muuttua esimerkiksi merkkijonoksi 1111000110.
- Jokaisella bitillä on yhtä suuri todennäköisyys mutaatioon.

Valinta

- Perinteisesti vanhemmat valitaan risteytymään todennäköisyydellä, joka on suhteutettuna sopeutuvuuteen: *suhteellinen valinta*.
- Vanhempien tuottamat jälkeläiset korvaavat vanhemmat.
- Yleinen toimintaperiaate on sopivimman selviäminen.

Risteytys

- Toteutetaan kahden vanhemman välillä.
- Tuottaa yhden tai kaksi jälkeläistä.
- Klassinen risteytys tapahtuu yhdessä tai kahdessa kohdassa.

- Yhdessä kohdassa: risteytetään merkkijonot 1111111111 ja 0000000000 kolmannesta merkistä eteenpäin merkkijonoiksi 1110000000 ja 0001111111.
- Kahdessa kohdassa: risteytetään merkkijonot 1111111111 ja 0000000000 kolmannesta ja kahdeksannesta merkistä eteenpäin merkkijonoiksi 1110000011 ja 0001111100.

Nämä kolme edellä mainittua perusoperaatiota ovat yksittäin tehtynä tuloksettomia, mutta kun ne kaikki toimivat yhteistyössä, niin joskus tulokset ovat erittäin hyviä.

Geneettistä algoritmia kannattaa käyttää, kun kysymyksessä on esimerkiksi multimodaaliset funktiot, diskreetit tai ei-jatkuvat funktiot, epälineaariset parametrien riippuvuudet tai NP-täydellisen kombinatorisen ongelman approksimointi. Geneettistä algoritmia ei kannata käyttää, jos löytyy jo olemassa olevia parempia algoritmeja ongelman ratkaisulle.

Gudenberg ja muut [2006] ovat tehneet uuden kevyen ja epädeterministisen geneettisen algoritmin UML-luokkakaavion rakenteen muodostamiseen.

Kaavion solmujen koko on vakio ja sitä ei muuteta algoritmin aikana, joten ei ole myöskään tarvetta tallettaa tietoa solmun koosta jokaiseen kaavioon. Solmu esitetään x- ja y-koordinaateilla sen vasemmasta yläkulmasta katsoen. Särmä liittää kaksi solmua yhteen ja särmän koordinaatit tallennetaan suhteessa solmu-nelikulmion vasempaan yläkulmaan. Hierarkkiset särmät piirretään suoraan pisteestä pisteeseen, kun taas ei-hierarkkiset särmät piirretään ortogonaalisesti ja niitä voidaan taivuttaa, ja taivutus määritellään yhdellä pisteellä. Koska ei-hierarkkiset särmät liitetään vain joko oikealle tai vasemmalle puolen solmua, on taivutusten määrä aina parillinen. Gudenbergin ja muiden [2006] toteutus tyytyy vain kahteen taivutukseen, sillä nämä kaksi taivutusta voidaan tallentaa yhdellä x-koordinaatilla.

Mutaatio	Todennäköisyys
Liikuta solmua	0.54
Liikuta särmän porttia	0.1
Vaihda särmän porttia	0.21
Liikuta taivutusta	0.1
Poista taivutus	0.05

Kaavio 7.1. Operaatioiden todennäköisyydet evoluutioaskeleessa.

Jokaisella mutaatiolla on omat askelkokonsa eli oppimisnopeutensa ja todennäköisyytensä. Mahdolliset mutaatiot on esitetty kaaviossa 7.1. Särmän portti tarkoittaa solmun reunan kohtaa, jonka särmä leikkaa. Tavallisesti tällaista kohtaa merkitään nuolella ja samaan nuoleen voi tulla monta eri särmää, sekä samalla solmun

sivulla voi olla monta eri nuolta. Operaatio valitaan annetulla todennäköisyydellä evoluutioaskeleessa eli liikuta solmua operaatio valitaan todennäköisyydellä 0,54. Tämä operaatio toimii seuraavasti:

1. Valitse satunnaisesti solmu.
2. Liikuta solmua $(x', y') = (x, y) + (dx, dy)$, jossa dx ja dy ovat satunnaislukuja, jotka noudattavat normaalijakaumaa keskihajonnoilla $\sigma_x = 100$ ja $\sigma_y = 50$.
3. Tarkista, leikkaako solmuun tulevat tai solmusta lähtevät särmät toisiaan ja tarvittaessa vaihda porttia.
4. Iteroinnin todennäköisyyden mukaan palaa kohtaan 1.

Kaikki operaatiot toimivat samaan tapaan ja kaikilla operaatioille alustetaan iteraatiotodennäköisyys arvolla 0,7. Iteraatiotodennäköisyys on mahdollisen yhden operaation toiston todennäköisyys heti sen suorituksen jälkeen.

Gudenberg ja muut [2006] laskevat sopivuuden tai virhemäärän yhdeksän metriikan lineaarikombinaationa väliltä [0, 1] seuraavasti:

1. **NN** Solmujen päällekkäisyydet lasketaan ja niitä vähennetään asteittain. Päällekkäisyyden alaa ei kannata laskea, jolloin

$$\rho_{NN} := \frac{|\text{overlaps}|}{|\text{overlaps}| + 1}.$$

2. **EN** Särmien ja solmujen päällekkäisyydet hoidetaan analogisesti, jolloin

$$\rho_{EN} := \frac{|\text{overcuts}|}{|\text{overcuts}| + 1}.$$

3. **EE** Särmien päällekkäisyydet lasketaan ja jaetaan mahdollisten leikkausten ylärajalla. Kaava on samantapainen kuin Purchasen [2002] ehdottama. Yläraja riippuu särmien ja taivutusten määrästä, jolloin

$$\text{eb} := |\text{edged}| + 2 \cdot |\text{bends}| \text{ ja}$$

$$\rho_{EE} := \frac{2 \cdot |\text{crossings}|}{\text{eb} \cdot (\text{eb} - 1)}.$$

4. **H** Hierarkkisten särmien tulee noudattaa yhteistä suuntaa alhaalta ylöspäin. Gudenbergin ja muiden UML-luokkakaavio-näkymässä hierarkkiset särmät

kuvaavat luokkien perimistä. Yhteisen suunnan löytämiseen riittää lineaarinen skaalaus, jolloin

$$\rho_H := \frac{|\text{direcion misses}|}{|\text{generalizations}|}.$$

5. **GL** Hierarkkisten särmien pituudet voidaan määrittää uudelleen. Tämä metriikka mittaa eroa haluttuun pituuteen. Lyhentämistä rangaistaan enemmän kuin pidentämistä, jolloin

$$\rho_{GL} := 1 - \frac{\sum_{g \in generalizations} \text{mix}(\bar{g}, l_{pref})}{|\text{generalizations}|},$$

jossa $\text{mix} : \mathbb{R}_0^+ \times \mathbb{R}^+ \to [1,0]$, $\text{mix}(x, y) := \frac{\min(x, y)}{\max(x, y)}$ ja $\bar{g}$ on g:n pituus.

6. **AL** Assosiaatioita eli ei-hierarkkisia särmiä käsitellään samalla tavalla. Määritellään haluttu pituus mahdollisesti kahdelle horisontaaliselle osalle. Koko särmän pituutta ja kahden osan pituutta vertaillaan erikseen. Tällöin

$$\text{piece(a)} := \text{mix}(\bar{a}_{h1}, l_{pref}) + \text{mix}(\bar{a}_{h1} + \bar{a}_{h2}, 2 \cdot l_{pref}), \text{ jolloin}$$

$$\rho_{AL} := 1 - \frac{\sum_{a \in associations} \text{piece(a)}}{3 \cdot |\text{associations}|}.$$

7. **A** Hierarkkisen särmän ja solmun reunan, johon särmä on liitetty, välistä kulmaa arvioidaan horisontaalisen etäisyyden ja etäisyyksien summan suhteena. Olkoon Δx horisontaalinen ja Δy vertikaalinen etäisyys alku- ja loppupisteen välillä. Tällöin

$$(x, y) \to (x + \Delta x, y + \Delta y), \text{ jolloin}$$

$$\rho_A := \frac{\sum_{generalizations} \left(\frac{\Delta x}{\Delta x + \Delta y}\right)^2}{|\text{generalizations}|}.$$

8. **MP** Jos useampia särmiä on kiinni samalla puolella solmua, pyritään tasapainottamaan särmät porttien välille, jolloin porttien väliin jäävä alue lasketaan. Saman matkan päähän asetettujen porttien minimiväli

$$\text{equi}(k) := n_k \cdot \left(\frac{k}{n_k}\right)^2$$

ja maksimi on

$\bar{k}^2$, kun $n_k = 0$, missä

k on solmun jaettu sivu n_k portilla $n_k + 1$ osaan $k_{0,} k_{1,} \ldots, k_{n_k}$ pituuksilla $\bar{k}_i, i = 0, \ldots, n_k$.

Skaalattu virhe per sivu kumuloituu: side(k) : =

$$\frac{\sum_{i=0}^{n_k} (\bar{k}_i)^2 - \text{equi}(k)}{(\bar{k})^2 - \text{equi}(k)}, \text{ jolloin } \rho_{MP} := \frac{\sum_{k \in nodesides} \text{side}(k)}{|\text{nodesides}|}.$$

9. **B** Taivutusten määrä tulisi minimoida, kuten olemme jo aikaisemminkin todenneet luvussa 5. Tulokseksi saadaan

$$\rho_B := \frac{|\text{bends}|}{2 \cdot |\text{associations}|}.$$

Olkoon x luokkakaavio ja $\rho_m(x)$ luokkakaavion arvot metriikalle m. Sopivuusfunktio tai hyvyysfunktio lasketaan lineaarikombinaationa näistä arvoista painokertoimilla kaaviosta 7.2 siten, että

$$F(x) := \sum_{m=1}^{9} \omega_m \cdot \rho_m.$$

m	NN	EN	EE	H	GL	AL	A	MP	B
ω	1	1	5	10	2	1	1	0.5	0.5

Kaavio 7.2. Metriikoiden painokertoimet.

Gudenberg ja muut [2006] huomauttavat tarvinneensa paljon kokeita, että nämä kertoimet saatiin. Kävi ilmi, että hierarkiaa täytyi painottaa, sillä on paljon helpompaa hienosäätää kaaviosta päällekkäisyyksiä kuin yhtä huonosti sopivaa hierarkiaa, joten siksi hierarkkisuudelle on annettu suurin kerroin.

Päällekkäisyyksien regressiivinen skaalaus, kuten esimerkiksi NN-metriikka, tuottaa suhteellisen suuria arvoja (yli 0,5), joten nämä metriikat muutetaan hyvin usein nollaksi.

Suurien kaavioiden särmien päällekkäisyyksien painokerrointa tulisi nostaa.

Gudenberg ja muut [2006] ovat huomanneet hienon hienojen metriikoiden huomioiminen, kuten metriikoiden A ja MP, parantaa lopullisen kaavion rakennetta huomattavasti.

8. Takaisinmallinnus

UML-luokkakaavio toimii hyvin suunniteltaessa ohjelmistoa, mutta niitä harvemmin käytetään ohjelmistojen ylläpitoon tai jatkokehitykseen. Tämä johtuu siitä, että tällaisten kaavioiden palauttaminen ja ylläpitäminen manuaalisesti vie paljon aikaa. Kun UML-kaaviota ei päivitetä manuaalisesti, niin kaavio pysyy ennallaan sillä välin kuin lähdekoodia kehitetään eteenpäin ja kaavio ei vastaa enää ohjelmistoa halutulla tavalla. Koska dokumentaatio ei vastaa ohjelmistoa, niin ei siitä ole silloin enää hyötyäkään. UML-luokkakaavio voidaan päivittää generoimalla se uudestaan lähdekoodista. Tätä prosessia kutsutaan ohjelmiston takaisinmallinnukseksi.

Takaisinmallinnus tapahtuu jäsentämällä lähdekoodia ja luomalla jäsennyksien mukaan kartoituksen syntaksista ja semantiikasta mukaan lukien ohjelmointikäytännöt, kieli ja uudelleen käytetyt kirjastot. Sutton ja Maletic [2007] käyttävät näitä kartoituksia takaisinmallinnustyökalussa nimeltä *pifler*.

Seuraavaksi luetellaan Suttonin ja Maleticin [2007] käyttämät kartoitukset, joita he pitävät parempana kuin tällä hetkellä käytössä olevissa takaisinmallinnustyökaluissa. Nämä kartoitukset ovat heuristiikkoja, jotka perustuvat enemmän syntaktiseen ja semanttiseen kuin deterministiseen analyysiin. Heidän takaisinmallinnustyökalunsa käyttää lähdekoodimateriaalina C++-kirjastoja.

8.1. Luokkatyyppien tunnistaminen

Luokkien, rajapintojen ja datatyyppien välinen erottelu on semanttisesti tärkeää UML:ssä. Kokonaisuutena nämä elementit kuvataan luokittelijoina tai nimetään mallinnuselementeiksi, joilla on (a) ominaisuuksia ja käyttäytymismalleja ja (b) ne voivat osallistua yleistysrelaatioon. Kuitenkin tällaisten elementtien kohtelu mallissa voi vaihdella paljon. Esimerkiksi UML ei salli assosiaatioita luokkien ja datatyyppien välille ja vain rajapinnat voidaan toteuttaa. Valitettavasti C++:ssa ei ole tarpeeksi tarkkaa sanakirjastoa, jotta nämä luokittelijat olisi helppo jäsentää yksinkertaisesti. Myöskään ei ole selvää onko luokka UML-tietotyyppi vai rajapinta. Seuraavaksi keskitytään erottelemaan nämä.

8.1.1. Rajapinnat

Metodi, jota käytetään erottelemaan rajapinnat, on hyvin suoraviivainen. Rajapinnan määritelmä on lainattu muista olio-ohjelmointikielistä, kuten Java ja C#. Määritellään C++-rajapinta luokaksi, joka määrittelee ainoastaan julkisia, aitoja virtuaalisia metodeja, eikä se omista yhtään muuttujia, eikä myöskään toteuta rakenninta tai hajotinta ja jos se periytyy jostain muusta luokasta, niin myös yläluokan pitää olla rajapinta.

Suurin osa näistä rajoitteista tulevat näkemyksestä, että rajapinta muodostuu sopimuksesta eikä niinkään ohjelmiston käyttäytymisestä. Esimerkiksi luokka, joka toteuttaa metodin assosioi käyttäytymistä sillä luokalla. Myös, jos luokka määrittelee muuttujan, assosioi muuttujan tila luokan kanssa. Jos luokalla ei ole muuttujia, niin ei se tarvitse erillistä rakenninta tai hajotinta. Lopuksi rajapinnat rajoitetaan periytymään ainoastaan rajapinnoista, joka on tavallista olio-ohjelmoinnissa.

Täysin virtuaalisien metodien erottelulla on tehokasta tunnistaa aito C++-rajapinta muista abstrakteista luokista, mutta silloin joudutaan luopumaan muutamasta asiasta. Ensiksi, C++-rajapinnan määrittely on hyvin rajoittavaa ja voi olla, että se ei sovellu tavalliseen tapaan kohtelemaan abstrakteja luokkia rajapintoina. Tämä kartoitus voi siis tuottaa odottamattomia tuloksia. Toiseksi, aikaisemmat UML-versiot vahvasti kieltävät käyttämästä rajapintoja assosiaatioelementteinä. Nämä rajoitukset on korjattu uudempiin versioihin.

8.1.2. Tietotyypit

UML:ssä tietotyyppi määritellään vain sen arvon mukaan. Jos kahdella ilmentymällä on sama arvo, niin silloin ne ovat samoja ilmentymiä. Luokkien ilmentymillä on erilliset identiteetit ja kaksi objektia samassa tilassa ei välttämättä ole sama objekti. Tavallisia esimerkkejä tietotyypeistä ovat ohjelmointiprimitiivit kuten kokonaisluvut, Boolen arvot ja enumeraatioarvot. Merkkijonoluokka on myös esimerkki tietotyypistä. Jotta voidaan identifioida C++-tietotyypit, täytyy luottaa täysin siihen, kuinka luokkia käytetään ohjelmoinnissa. Erityisesti käytetään luokkien rakennin, kopiointi ja tehtävä semantiikkoja tunnistettaessa tietotyyppejä.

C++-tietotyypit käsittää kaksi eri variaatiota. Luokka, joka toteuttaa julkisen rakentimen, kopiointirakentimen ja tehtäväoperaattorin on tietotyyppi. Myös luokka, joka ei toteuta rakenninta tai tehtäväoperaattoria on myös tietotyyppi. Tuhoajia ei käsitellä luokituksessa, sillä ne lisäävät vain vähän luokan suunnittelutason semantiikkaa, koska kaikilla luokilla on hajotin, joko implisiittinen tai eksplisiittinen. Hajottimien käyttö luokituksessa voi johtaa moniselitteisyyteen.

8.2. Attribuuttien tunnistaminen

Tyypillinen takaisinmallinnustyökalu korreloi UML-attribuutit luokan jäsenmuuttujiksi. Kuitenkin UML:ssä attribuutteja käytetään jokseenkin eri tavalla. Tyypillisesti attribuutit heijastaa luokan rajapintaa, jota voidaan lukea tai kirjoittaa, eikä niinkään jäsenmuuttujan yksityiskohtia. Voidaan sanoa, että UML-attribuutit enemmänkin vastaavat ominaisuuksien ilmentymiä kuin luokan jäsenmuuttujia. Tämä on paljon soveliaampaa takaisinmallinnustyökaluille, koska se esittää paljon abstraktimpaa näkymää luokasta, eikä vain sen toteutusyksityiskohtia.

Luokan attribuuttien tunnistaminen pohjautuu kokoelmaan palauttajia ja muuttajia. Palauttajat määritellään vakiometodeiksi, jotka palauttavat jäsenmuuttujan ja muuttajat ovat metodeja, jotka kirjoittavat jäsenmuuttujan arvon. Jäsenmuuttuja ryhmittelee palauttajat ja muuttajat, joissa se toimii. Vain luettavissa oleva ominaisuus voidaan identifioida palauttajan olemassa ololla ja luettavissa ja kirjoitettavissa oleva ominaisuus voidaan identifioida palauttajan ja joukolla muuttajia. Kirjoitettavissa olevalla ominaisuudella on joukko muuttajia, koska ominaisuuden rajapinta voi tukea eri semantiikkoja, kuten lisää, poista ja tyhjennä.

Kuitenkaan tällainen tapa löytää C++-luokan attribuutit ei ole täydellinen. Yritykset automatisoida löytäminen, ilman hienostuneempia analysointitekniikoita, johtaa hyvin todennäköisesti vääriin tunnistuksiin.

8.2.1. Attribuutin tyyppi

Vaikka on verraten helppoa määrittää jäsenmuuttujan tyyppi C++:ssa, niin tyyppi ei välttämättä aina suoraviivaisesti sovi UML-tyypiksi. Esimerkiksi UML ei tarjoa syntaksi mallintamaan osoitinta tai viittausta ja monet C++-kielen tyyppimääritteet, kuten *const*, *mutable* ja *volatile*, eivät välttämättä sisällä kovinkaan paljon informaatiota, koska UML ei mallinna ylimääräistä informaatiota attribuutin tyyppiin.

Määritellään yksinkertainen kartoitus tyypin määritykselle, jossa mallin attribuutin tyyppi on viittaus C++tyyppi-ilmaukselle. Tyypin viittaus saadaan, kun poistetaan kaikki, osoitin, viittaus, taulukkosymbolit ja kaikki muutkin määritteet muuttujasta.

Yksinkertaisen kartoituksen ohella käsitellään monimutkaisempia kaavaintyyppejä. Kaavaimia käytetään usein C++-ohjelmissa, mutta ne eivät vaikuta UML-mallin tyyppi-informaatioon. Sen sijaan ne sisältävät semantiikkaa luokkien välisistä assosiaatioista, mutta eivät luokkien tyypeistä assosiaatiossa. Tällaiset luokat esittävät transitiivistä ominaisuutta luokkien välillä. Luokkien sanotaan olevan transitiivisiä, jos luokka *A* sisältää luokan *B*, joka sisältää luokan *C*, niin silloin luokka *A* sisältää luokan *C*. Lopputoteamus on soveliaampi UML-tyypin määritykselle. Merkityksellinen informaatio voidaan erotella kaavaimeen määrittelystä ottamalla sen kaikkein sisimmät argumentit. Eroteltuun tyyppi-informaatioon voidaan käyttää yksinkertaista kartoitusta. Tämän automatisointi vaatii paljon pohjatietoa. Ohjelman, joka suorittaa tämän, tarvitsee tietää etukäteen, mitkä luokat toteuttavat transitiivisuusehdon ja mitkä kaavainparametrit vastaavat tyyppi-informaatiota.

8.2.2. Monilukuisuus

Monilukuisuus määrittelee sallitun määrän attribuutin ilmentymiä ja se esitetään sallitun alueen ala- ja ylärajana, kuten esimerkiksi 0..*. Attribuuttien monilukuisuus voi olla vaikeaa tunnistaa. Ei ole olemassa joukkoa yksinkertaisia sääntöjä, jotka helposti

määrittävät kartoituksen monilukuisuudelle. Onneksi C++:ssa on useita merkkejä, jotka auttavat arvioimaan attribuutin monilukuisuuden. Erityisesti osoittimet, taulukkosulut ja transitiiviset luokat.

Vain monilukuisuudet, jotka voidaan yksiselitteisesti tunnistaa ovat sellaisia, jotka määritellään yhdellä ilmentymällä, viittauksella yhteen ilmentymään tai rajatulla taulukolla. Kaikissa muissa tapauksissa ei voida tarkasti sanoa ylä- tai alarajaa. Tämä johtuu C++-muuttujien moniselitteisyydestä.

8.2.3. Järjestys

Koska monet attribuutit esittävät usean ilmentymän hallintaa, tarjoaa UML-metamalli mahdollisuuden kuvailla järjestyksen semantiikkaa. UML määrittää kaksi järjestyksen tyyppiä attribuuteille: järjestetty ja järjestämätön. Nämä yksinkertaisesti määrittävät, sisältääkö attribuutti järjestetyn ilmentymän, kuten listan tai vektorin, vai järjestämättömän, kuten joukon. Tähän päivään mennessä ei ole hyvää analyysimetodia, joka pystyisi tarkasti määrittämään järjestyksen semantiikan. Onneksi voidaan käyttää taulukoiden määrittelysyntaksia ja sisältöluokkia auttamaan tätä kartoitusta.

Kartoitukset järjestyksen semantiikalle voidaan helposti johtaa jäsenmuuttujan tyypistä. C-taulukolle ja C-vektorille varataan aina peräkkäiset muistipaikat.

8.3. Parametrin suunnan tunnistaminen

UML-parametreilla on joitakin samoja ominaisuuksia UML-attribuuttien kanssa, jolloin voidaan käyttää samanlaista kartoitusta. UML-parametrit sisältävät myös tiedon siitä, kuinka niiden arvot siirretään ja palautetaan operaatiosta. Tätä informaatiota kutsutaan parametrin suunnaksi. UML määrittelee neljä suuntaa parametreille: sisään, ulos, sisään-ulos ja paluuarvo. Suunta määrittelee käytetäänkö parametria syötteenä, tulosteena, molempina vai puhtaana palautusarvona. Koska C++:n syntaksi ei tarjoa tarpeeksi informaatiota tälle kartoitukselle, määritellään lista kartoituksia, jotka perustuvat parametrin tyyppiin ja sen määrittelyyn.

C++:ssa argumentit annetaan metodeille, joko viittauksina tai arvoina. Koska arvona annetut parametrit muodostavat lokaalin kopion, ne voidaan tunnistaa sisään-arvoina. Jos parametri annetaan viittauksena, täytyy tarkastella parametrin *const*-määrettä. Jos parametri sisältää *const*-määreen, sitä voidaan pitää sisään-arvona, muuten se mallinnetaan ulos-arvona.

8.4. Assosiaatioiden tunnistaminen

Vaikka UML-assosiaatioita käytetään yleensä kuvaamaan omistusrelaatiota, voidaan niillä mallintaa myös kahden eri luokan välistä relaatiosemantiikkaa. Tällaisille semanttisille linkeille ei löydy C++-syntaksista merkintää. Kuitenkin on melko helppoa erottaa omistusrelaatio luokan jäsenmuuttujan avulla. Määritellään C++-assosiaatio jäsenmuuttujaksi, jolla on tyyppiviittaus UML-malliluokkaan, mutta joka ei ole tietotyyppiä. Estetään luokkien ja rajapintojen assosiaatiot tietotyyppien kanssa, koska

tämän nimenomaisen relaation semantiikka on täysin sen varassa, että jäsenmuuttuja mallinnetaan attribuuttina.

8.5. Aggregaatti

Aggregaattiassosiaatio määrittelee elinajan semantiikan ilmentymille, jotka kuuluvat relaatioon. UML määritelle kolme eri semantiikkaa assosiaatioille: ei mitään, aggregaatti ja yhdistetty. Ainoat aggregaattityypit, jotka voidaan erotella C++-kieliopista ovat aggregaatti ja yhdistetty. Aggregaattityyppi *ei mitään* esittää puhtaasti semanttisia linkkejä luokkien välillä ja sillä on vähän mielenkiintoa tässä kontekstissa.

Aggregaatin tunnistaminen ominaisuudesta on vaikeaa, koska C++ tarjoaa hyvin vähän kielellistä apua assosiaatiosemantiikkojen tunnistamiseen. Esimerkiksi jäsenmuuttuja, joka osoittaa toiseen objektiin voi olla muodostettu luokassa, joka sen omistaa tai se voidaan tallentaa ja jakaa muiden objektien välillä. Fiksujen osoittimien, jotka osaavat itse hoitaa roskienkeruun ja tuhoutumisensa, käyttäminen antaa ohjelmoijille mahdollisuuden tietyntyyppisen elinajan käyttämiseen ohjelmassa. Näiden merkkien tunnistaminen voi radikaalisti vähentää työn määrää päätettäessä oikeaa aggregaattia näille elementeille.

Kuten monilukuisuuden tunnistaminen myös aggregaattityypin tunnistamiseen ei ole yksinkertaista joukkoa sääntöjä. C++ tarjoaa merkinnällistä informaatiota, jota voidaan käyttää kartoituksessa, kuten osoittimet, viittaukset ja taulukkosymbolit. Myös transitiivisiä luokkia voidaan käyttää tässä kartoituksessa.

8.6. Toteutuksen tunnistaminen

UML:ssä periytyminen esitetään relaationa, joka voidaan helposti päätellä C++-perimismääritelmästä. Kuitenkin rajapinnan toteutus aiheuttaa hieman erilaisen ongelman. Syntaktinen mekanismi perimälle C++:ssa on sama. UML tarjoaa ylimääräisen syntaksin, jota voidaan käyttää rajapintojen esittämiseen. UML-periytyminen on riippuvuus luokan ja rajapinnan välillä, jossa luokka toteuttaa sopimuksen, jonka rajapinta määrittelee.

Metodi, jolla rajapintarelaatio tunnistetaan, pohjautuu luokituksen tyypin oikeaan päättelyyn. Jos luokka toteuttaa kaikki rajapinnan abstraktit metodit, niin voidaan luoda relaatio luokan ja rajapinnan välille. Jos luokka ei toteuta kaikkia metodeja, niin kyseessä on abstrakti luokka, joka periytyy rajapinnasta.

9. Yhteenveto

UML-notaatio käy yhtä hyvin niin suorien kuin pienempienkin ohjelmistojen mallintamiseen, mutta suurissa ohjelmistoissa UML tarjoaa helposti luettavan kokonaisuuden koko ohjelmistosta, eikä vain sen toteutusyksityiskohtia. Suurien ohjelmistojen UML-kaavioiden luomiseen kuluva työmäärä riippuu hyvin paljon käytetystä työkalusta. Jos käytössä oleva työkalu pystyy auttamaan mallintajaa mahdollisimman paljon, niin on selvää, että tämä vähentää työhön kuluvaa aikaa. UML-mallinnustyökalujen tulee käyttää tässä tutkielmassa mainittuja estetiikkakriteereitä, joiden toteuttaminen on haasteellista.

UML-kaavioiden automaattinen generointi ei ole yksiselitteistä. Parhaimpaan tulokseen päädytään, kun käytetään useita tarjolla olevia heuristiikkoja. Tosin vaikeuksia syntyy, koska heuristiikoilla on tapana kumota toistensa vaikutus. Edellä nähtiin esimerkki deterministisestä ja epädeterministisestä toteutuksesta. Molemmat olivat erittäin monimutkaisia, mutta deterministisellä tavalla päästiin parempiin tuloksiin, sillä se pystyi noudattamaan paremmin estetiikkakriteereitä. Deterministisyydestä voidaan vielä todeta, että se ei tule täysin kattamaan koko generointiongelmaa, sillä kartan nimiöiden asettelu on NP-kova ongelma [Wagner and Wolff, 1998].

Di Battistan ja muiden [1999] topologia, muoto ja metriikka -menetelmä on yksinkertaisuudessaan täysin käyttökelpoinen vielä nykyisiinkin algoritmeihin [Eiglsperger et al., 2003] verrattuna.

Noack ja Lewerentz [2005] esittävät hyvin tarkasti miten klusterointiongelma voidaan ratkaista. Tätä tosin ei ole käytetty vielä hyväksi algoritmeissa.

UML-luokkakaaviot ovat tulleet jäädäkseen ja tutkimus graafinpiirrosta tulee varmasti jatkumaan tällä alalla vielä tulevaisuudessakin. Varsinkin ohjelmistojen takaisinmallinnuksen suosion kasvaessa UML-kaavioiden generointiin käytettävät työkalut kehittyvät. Tämä tutkielma teki katsauksen tällä hetkellä olevaan tutkimukseen alalta ja toi esiin sieltä merkittävimmät askeleet UML-luokkakaavioiden piirtämiseen, estetiikkaan ja niitä generoiviin algoritmeihin

Viiteluettelo

[Bertolazzi et al., 2002] Paola Bertolazzi, Giuseppe Di Battista and Walter Didimo, Quasi-upward planarity. *Algorithmica,* **32**, 3, 474-506.

[Binucci et al., 2001] Carla Binucci, Walter Didimo, Giuseppe Liotta and Maddalena Nonato, Labeling heuristics for orthogonal drawings. In: *Proc. of the 9th International Symposium on Graph Drawing, 139-153.*

[Binucci et al., 2002] Carla Binucci, Walter Didimo, Giuseppe Liotta and Maddalena Nonato, Computing labeled orthogonal drawings. In: *Proc. of the 10th International Symposium on Graph Drawing,* 66-73.

[Di Battista et al., 1999] Giuseppe Di Battista, Peter Eades, Roberto Tamassia and Ioannis G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[Di Battista et al., 2001] Giuseppe Di Battista, Walter Didimo and A. Marcandalli, Planarization of clustered graphs. In: *Proc. of the 9th International Symposium on Graph Drawing,* 60-74.

[Cortese and Di Battista, 2001] Pier Francesco Cortese and Giuseppe Di Battista, Clustered Planarity. In: *Proc. of the 21th Annual Symposium on Computational Geometry*, 32-34.

[Eiglsperger and Kaufmann, 2001] Markus Eiglsperger and Michael Kaufmann, An approach for mixed upward planarization. In: *Proc. of the 7th International Workshop on Algorithms and Data Structures*, 352-364.

[Eiglsperger and Kaufmann, 2001] Markus Eiglsperger and Michael Kaufmann, Fast compaction for orthogonal drawings with vertices of prescribed size. Revised Papers from the 9th *International Symposium on Graph Drawing*, 124-138.

[Eiglsperger et al., 2003] Markus Eiglsperger, Michael Kaufmann and Martin Siebenhaller, A topology-shape-metrics approach for the automatic layout of UML class diagrams. In: *Proc. of the 2003 ACM Symposium on Software Visualization*, 189-198.

[Feng et al., 1995] Qing-Wen Feng, Robert F. Cohen and Peter Eades, Planarity for clustered graphs. In: *Proc. of the Third Annual European Symposium on Algorithms*, 213-226.

[Furnas, 1986] G. W. Furnas, Generalized fisheye views. In: *Proc. of the SIGCHI Conference on Human Factors in Computer Systems*, 16-23.

[Fößmeier and Kaufmann, 1996] Ulrich Fößmeier and Michael Kaufmann. Drawing high degree graphs with low bend number. In: *Proc. of the 3rd International Symposium on Graph Drawing*, 254-266.

[Freivalds et al., 2001] Karlis Freivalds, Ugur Dogrusöz and Paulis Kikusts, Disconnected graph layout and the polymino packing approach. Revised Papers from the 9th *International Symposium on Graph Drawing*, 378-391.

[Goodman, 2007] Erik D. Goodman, Introduction to genetic algorithms. In: *Proc. of the 2007 GECCO Conference Companion on Genetic and Evolutionary Computation*, 3205-3224.

[Gudenberg et al., 2006] J. Wolff v. Gudenberg, A. Niederle, M. Ebner and H. Eichelberger, Evolutionary layout of UML class diagrams. In: *Proc. of the 2006 ACM Symposium on Software Visualization*, 163-164.

[Klau and Mutzel, 1999] Gunnar W. Klau and Petra Mutzel. Combining graph labeling and compaction. In: *Proc. of the 7rd International Symposium on Graph Drawing*, 27-37.

[Noack and Lewerentz, 2005] Andreas Noack and Claus Lewerentz, A space of layout styles for hierarchical graph models of software systems. In: *Proc. of the 2005 ACM Symposium on Software Visualization*, 155-164.

[OMG, 2007] Introduction to OMG's Unified Modeling Language. Available as http://www.omg.org.

[OMG, 2007] Unified Modeling Language: Superstructure. Available as http://www.omg.org.

[Purchase, 1997] Helen C. Purchase, Which aesthetic has the greatest effect on human understanding? In: *Proc. of the 5th International Symposium on Graph Drawing*, 248-261.

[Purchase, 2002] Helen C. Purchase, Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing.* **13**, 5, 501-516.

[Purchase et al., 2000] Helen C. Purchase, Jo-Anne Alider and David A. Carrington, User preference of graph layout aesthetics: a UML study. In: *Proc. of the 8th International Symposium on Graph Drawing*, 5-18.

[Purchase et al., 2001] Helen C. Purchase, Matthew McGill, Linda Colpoys, David Carrington and Carol Britton, UML class diagram syntax: an empirical study. In: *Proc. of the 2001 Asia-Pacific Symposium on Information Visualization*, 113-120.

[Purchase et al., 2001] Helen C. Purchase, Matthew McGill, Linda Colpoys and David Carrington, Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In: *Proc. of the 2001 Asia-Pacific Symposium on Information Visualization*, 129-137.

[Sutton and Maletic, 2007] Andrew Sutton and Jonathan I. Maletic, Recovering UML class models from C++: a detailed explanation. *Information and Software Technology.* **49**, 3, 212-229.

[Von Gudenberg et al., 2006] J. Wolff Von Gudenberg, A. Niederle, M. Ebner and H. Eichelberger, Evolutionary layout of UML class diagrams. In: *Proc. of the 2006 ACM Symposium on Software Visualization*, 163-164.

[Wagner and Wolff, 1998] Frank Wagner and Alexander Wolff, A combinatorial framework for map labeling. In: *Proc. of the 6th International Symposium on Graph Drawing*, 316-331.

[Witkin et al., 1987] Andrew Witkin, Kurt Fleischer and Alan Barr, Energy constraints on parameterized models. In: *Proc. of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 225-232.