Suunnitteludokumentti

Ohjelmistotuotantoprojekti Todistustoimitin

Teppo Kankaanpää, Jaakko Nenonen, Aki Nyrhinen, Visa Röyskö, Juha-Matti Tapio

27. maaliskuuta, 2003
Helsingin yliopisto
Tietojenkäsittelytieteen laitos

Sisältö

Kuvat

Liitteet

Historia

Päiväys	Versio	Kuvaus	Tekijät
13.3.2003	0.1	Ensimmäinen luonnos	Kaikki
27.3.2003	1.0	Jäädytetty versio	Kaikki

Osa I

Yleistä

1 Yleistä

Syksyllä 2002 luennoitiin laitoksella kurssi Proof Theory and Proof Search, jossa käsiteltiin luonnollisen päättelyn tapaista formalismia, jonka etuina tähän nähden on se, että todistamisprosessi vaatii vähemmän intuitiivista päättelyä. Lauseen todistaminen tapahtuu "käänteisessä järjestyksessä", lähtien lopputuloksesta ja jakaen sitä yhä pienempiin osiin sääntöjä käyttäen kunnes päädytään aksioomiin tai erityisiin kohtiin, joissa ollaan päädytty "identtisesti tosiin" lauseisiin, jolloin todistus on valmis.

Aiemmin kurssin käytössä on ollut vain tekstipohjaisia ohjelmia. Ohjelmistotuotantoprojektin toteutettavana oleva todistustoimitin on graafinen helppokäyttöinen ohjelma, jolla todistuksia voidaan laatia mukaillen tapaa, jolla todistukset kirjoitettaisiin paperille.

1.1 Dokumentin rakenne

Tämä dokumentti jakautuu neljään osaan. Ensimmäisessä osassa kuvataan ohjelmaa yleisellä tasolla, esitellään käytetty termistö ja ohjelman päävaatimukset.

Toisessa osassa kuvataan se, miten ohjelma toteuttaa kalkyylin tallentamisen muistiin ja sen käsittelyn muistissa. Luvussa 4 kuvataan staattinen luokkarakenne ja luvussa 5 olioiden yhteistoiminta sekvenssikaavioilla.

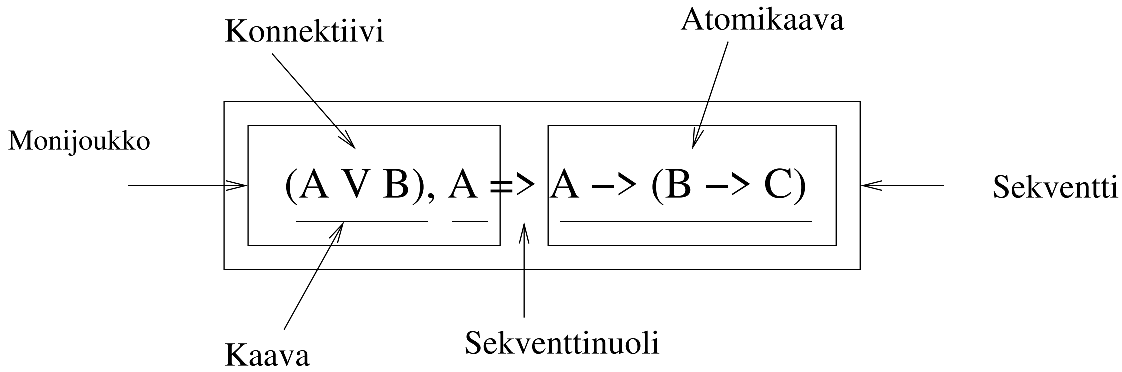
Kolmannessa osassa kuvataan ensin ohjelman käyttöliittymä yleisellä tasolla luvussa 6. Tämän jälkeen kuvataan käyttöliittymän staattinen luokkarakenne luvussa 7, jonka jälkeen käyttöliittymän sekvenssikaaviot kappaleessa 8.

Neljännessä osassa kerrotaan muita suunniteltuja asioita, testaus kappaleessa 9 ja yhteenveto kappaleessa 10.

1.2 Suunnitteluperiaatteet

Suunnittelussa on pyritty välttämään tarpeetonta pikkutarkkutta luokkien ja toimintojen kuvaamisessa. Aksessoreita ei tulla ohjelmoimaan ohjelmoinnin itsensä vuoksi, vaan pyritään siihen, että pääosa kirjoitetusta koodista on hyödyllistä ja ymmärrettävää.

Myöskään olioajattelua ei yritetä venyttää äärimmilleen. Proseduraalista tai funktionaalista ajattelutapaa siis käytetään häpeilemättä silloin kun se tuntuu tuottavan yksinkertaisemman ja toimivamman ratkaisun kuin oliomalli.



1.3 Termien selityksiä

Kerrataan aluksi eräs täsmällinen määritelmä käytetyille termeille (Väänänen: Matemaattinen logiikka, 1987), ja sen jälkeen selvitetään dokumentissa käytetyt muunnelmat näistä.

Aakkosto L on kokoelma kolmenlaisia symboleita: funktio-, relaatio- ja vakiosymboleita. Jokaiseen symboliin liittyy *paikkaluku*, joka on vakiosymboleilla 0.

L*-termien* joukko on pienin joukko, joka toteuttaa seuraavat ehdot: (i) $v_n \in A$, kun $n \in \mathbb{N}$ (muuttujasymbolit), (ii) $c \in A$, kun $c \in L$, (iii) Jos $f \in L$ on n-paikkainen muuttujasymboli ja $t_1, \ldots, t_n \in A$, niin $f(t_1, \ldots, tn) \in A$.

L*-atomikaavoja* ovat kaikki merkkijonot $R(t_1, \ldots, t_n)$, missä $R \in L$ on n-paikkainen relaatiosymboli ja $t_i, 1 \leq i \leq n$ ovat L-termejä, sekä kaikki merkkijonot $t = t'$, missä t ja t' ovat L-termejä.

L*-kaavojen* joukko Γ on pienin joukko merkkijonoja, joka sisältää atomikaavat ja on suljettu negaation, konjunktion, disjunktion, eksistentiaali- ja universaalikvanttorien suhteen, eli jos esimerkiksi valitaan kaavat ψ ja $\phi \in \Gamma$, niin merkkijono $\psi \vee \phi \in \Gamma$. Implikaatio ja ekvivalenssi voidaan esittää edellisten avulla.

Ohjelma ei käytä aivan samoja määritelmiä asioille. Ensinnäkin aakkosto L sisältää vain numeroituvan joukon 0-paikkaisia relaatiosymboleita (joita merkitään yleensä isoilla kirjaimilla $A, B, \ldots$). Symbolit kuvaavat siis vain jotain totuusarvoa. Jos ohjelmaan toteutetaan kvanttorit, myös monipaikkaiset relaatiosymbolit pitää ottaa huomioon. Näin pitkälle ei kuitenkaan ajateltu ohjelmaa tehdessä. Lisäksi atomikaavoina ei voi olla yhtäsuuruustermejä.

Toisekseen kaavat ovat syntaktisia olioita, joten täsmällinen sulutus on tärkeää. Atomikaavojen ympärille ei tule sulkuja, mutta konnektiivien yhdistäessä kaavoja tulos tulee ympäröidä suluilla.

Kolmanneksi vain disjunktio, konjunktio ja implikaatiosymbolien käyttö sallitaan kaavoissa. Muut kaavat voidaan esittää näiden avulla. Koska kvanttoreita ei toteutettu, myöskään näitä ei sallita kaavoissa.

Lisäksi ohjelman käyttämässä sekventtikalkyylissä on jotain omia erikoisuuksiaan. Seuraavassa on lista käsitteistä:

Sekventti: Sekventti on todistuspuun solmu. Se jakautuu kahteen monijoukkoon, eli sekventtinuolen oikeaan ja vasempaan puoleen. $(A \vee B), A \Rightarrow A \rightarrow (B \rightarrow C)$

Monijoukko: Monijoukko on joukko kaavoja, jossa sama kaava voi esiintyä useammin kuin kerran. Esimerkiksi $(A \vee B), A, A$ on monijoukko, jossa atomikaava A esiintyy kahdesti.

Sekventtinuoli: Sekventtinuolella erotetaan sekventin monijoukot toisistaan.

Kaava: Kaavat koostuvat atomikaavoista ja konnektiiveista. Esimerkiksi kaavassa $(A \vee B)$ A ja B ovat atomikaavoja, ja $\vee$ on konnektiivi.

Atomikaava: Atomikaavat (joskus myös lyhennettynä atomi) ovat 0-paikkaisia relaatiosymboleja.

Konnektiivi: Konnektiivit ovat loogisia operaatioita.

Kalkyyli: Kalkyyli on sääntöjen joukko.

Ratkeavuus: Mikäli todistus voidaan saattaa sellaiseen tilaan että Calculus havaitsisi sen oikeaksi, todistus on ratkeava.

Todistus: Todistus on tiukasti ottaen puumaisesti järjestetty joukko sekventtejä, joka osoittaa puun juuressa olevan sekventin loogisesti identtisesti todeksi (eli todeksi relaatiosymbolien tulkinnasta riippumatta). Dokumentissa todistus-sanaa käytetään myös mistä tahansa sääntöjen mukaan luoduista puista, jotka eivät välttämättä osoita juurisekventtiä todeksi.

Kesken oleva todistus: Mikäli todistus ei ole sellainen, että se osoittaisi juurisekventin todeksi, todistuksen sanotaan olevan kesken.

2 Tärkeimmät vaatimukset

Ohjelman on tietenkin toimittava loogisesti oikein ja todistamisen on tapahduttava kalkyylien määrittelemien sääntöjen kuvaamalla tavalla, ja toteutettava kaikki allamääriteltyjen kalkyylien säännöt.

Toinen ohjelman päävaatimus on se, että sen on oltava helppokäyttöinen, niin että suurin osa ohjelman toiminnallisuudesta on selvää ilman erillisen käyttöohjeen lukemista. Tähän ideaalitilanteeseen ei olla aivan päästy muutamissa kohdissa (esimerkiksi vientidialogin toiminta), mutta ongelmat on huomioitu ja niihin on yritetty varautua.

2.1 Muutoksia määrittelydokumenttiin

Suurin osa vaatimusmäärittelydokumentissa kuvatuista vaatimuksista on suunniteltu ohjelmaan sellaisenaan. Muutamat dokumentissa avoimeksi jätetyt kysymykset on pääosin ratkaistu. Nämä on listattu alla. Joidenkin, esimerkiksi puun värityksen valinnan, ratkaisu on perustellusta syystä jätetty toteutusvaiheeseen.

Tallennusmuodon määrittelyä muutettiin asiakkaan toivomuksesta siten, että todistuspuun ns. undo-haaroja ei tallenneta levylle, vaan tallennus-lataus sykli jätetään käyttäjän viimeiseksi mahdollisuudeksi päästä lopullisesti eroon tekemistään virhevalinnoista.

Lisäksi tallennettuun tiedostoon täytyy tallentaa tieto siitä, mitä kalkyyliä käyttäen kyseinen todistus on tehty (tätä ei sanottu määrittelydokumentissa täsmälliseesti). Tiedostoon ei tallenneta ikkunoiden koordinaatteja.

Undo-haaroja ei tarvitse poistaa puusta manuaalisesti, koska muistin loppuminen ei ole todennäköisesti ongelma.

Automaattinen tulostimelle tulostaminen päätettiin jättää toteuttamatta, koska tulostuksessa apuna käytetyt järjestelmät (LaTeX, *proof.sty*) voivat olla jo yksittäisessä käyttöjärjestelmässä asennettuna numeroituvasti äärettömän monella tavalla. Automatisoimista ei myöskään helpota Javassa äärimmilleen viety järjestelmäriippumattomuuden tavoittelu.

Todistuksen syöttödialogin tarkka syntaksi päätetään vasta myöhemmin, mutta se muistuttaa hyvin läheisesti jäsentäjän käyttämää syntaksia. Negaation aukipurkamista kaavoista ei suunniteltu toteutettavaksi, joten negaatio pitää kirjoittaa implikaation avulla.

Kaavojen sulutukseen ei suunniteltu lyhennysmerkintää, joten kaava $(A \vee B \vee C)$ ei ole hyväksyttävä kaava, se on kirjoitettava esimerkiksi muodossa $(A \vee (B \vee C))$.

Käyttöliittymä suunniteltiin englanninkieliseksi, mutta kielen vaihtaminen on helppoa.

Navigointipuuhun lisättiin myös solmutyyppi, jota vastaavasta sekventistä ei voi enää johtaa uusia sekventtejä, mutta joka ei myöskään ole tosi. Tällainen solmu on ”umpikuja”: jos sellainen on puussa, juurikaava ei mitenkään voi olla tosi, joten todistuspuuta pitää muokata niin että näistä solmuista päästään eroon.

2.2 Toteutettavat kalkyylit

Tämän suunnittelun yhteydessä todistustoimittimeen toteutetaan ainoastaan kaksi kalkyyliä, G3IP ja G3CP (Gentzen, 3 konnektiivia, klassinen ja intuitionistinen, propositiologiikka).

Intuitionistisen ja klassisen kalkyylin suurin ero on siinä, että intuitionistisessa kalkyylissä sekventtinuolen oikealla puolella sallitaan korkeintaan yksi kaavaa. Klassisessa sensijaan sekä oikea että vasen puoli ovat monijoukkoja.

Luetellaan ensin G3CP:n säännöt,

$$\frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \wedge B \Rightarrow \Delta} L\wedge, \qquad \frac{\Gamma, A \Rightarrow \Delta \qquad \Gamma, B \Rightarrow \Delta}{\Gamma, A \vee B \Rightarrow \Delta} L\vee, \qquad \frac{\Gamma \Rightarrow \Delta, A \qquad \Gamma, B \Rightarrow \Delta}{\Gamma, A \to B \Rightarrow \Delta} L\to,$$

$$\frac{\Gamma \Rightarrow \Delta, A \qquad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \wedge B} R\wedge, \qquad \frac{\Gamma \Rightarrow \Delta, A, B}{\Gamma \Rightarrow \Delta, A \vee B} R\vee, \qquad \frac{\Gamma, A \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \to B} R\to$$

ja sitten G3IP:n.

$$\frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \wedge B \Rightarrow \Delta} L\wedge, \qquad \frac{\Gamma, A \Rightarrow \Delta \qquad \Gamma, B \Rightarrow \Delta}{\Gamma, A \vee B \Rightarrow \Delta} L\vee, \qquad \frac{\Gamma \Rightarrow A \qquad \Gamma, B \Rightarrow \Delta}{\Gamma, A \to B \Rightarrow \Delta} L\to,$$

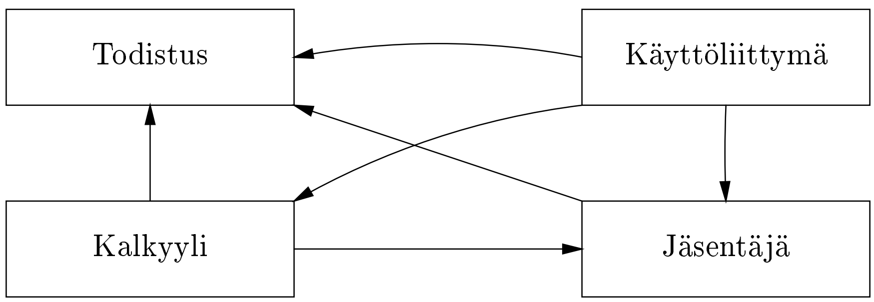
$$\frac{\Gamma \Rightarrow A \qquad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B} R\wedge, \qquad \frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B} R\vee^1, \qquad \frac{\Gamma \Rightarrow B}{\Gamma \Rightarrow A \vee B} R\vee^2, \qquad \frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \to B} R\to$$

3 Arkkitehtuuri

Todistustoimitin jakautuu toiminnallisesti neljään osajärjestelmään: kalkyyliin, jäsentäjään, käyttöliittymään ja todistukseen. Näistä muita paitsi käyttöliittymää ja jäsentäjää kutsutaan myöhemmin dokumentissa yhteisnimellä “tietosisältö” tai “tietorakenne” (vaikka ne sisältävät myös ohjelman toiminnallisuuden).

Osajärjestelmät eivät ole erityisen itsenäisiä. Erityisen vahva sidonta on kalkyylin, todistuksen ja jäsentäjän välillä. Voisi sanoa että todistus ja jäsentäjä ovat kalkyylin osajärjestelmiä, mutta myös käyttöliittymällä on tehokkuussyistä pääsy suoraan todistukseen kulkematta kalkyylin kautta. Oheisesta kuvassa 1 olevasta osajärjestelmäkaaviosta nähdään, miten huonosti järjestelmä jakautuu itsenäisiin komponentteihin.

Kaaviossa olevat nuolet tarkoittavat osajärjestelmien välisiä riippuvuuksia. Nuolen lähtöpäässä oleva osajärjestelmä ei muodosta mielekästä kokonaisuutta ilman osajärjestelmää johon nuolen kärki osoittaa.



Kuva 1: Osajärjestelmäkaavio

3.1 Kalkyyli

Kalkyyli-osajärjestelmän ensisijainen tehtävä on todistuspuun muokkaaminen siten, että todistuspuu säilyy oikeellisena. Kalkyyli-osajärjestelmä tarjoaa kutsujalleen riittävän vahvan mutta tarpeeksi rajoittavan rajapinnan muokata todistusta niissä puitteissa, joissa sen tulee olla kyseisellä kalkyylillä mahdollista. Teknisenä yksityiskohtana kalkyyli toteuttaa myös yksitasoisen undo-operaation, jolla voidaan perua edellinen tehty operaatio (myös kyseinen yksitasoinen undo kuuluu näiden operaatioiden joukkoon).

Kalkyyli-osajärjestelmä tarvitsee toimiakseen todistus-osajärjestelmän.

3.2 Todistus

Todistus-osajärjestelmän tehtävä on esittää todistuksen tila kullakin hetkellä ja tallentaa muiden osajärjestelmien tarvitsemaa, todistukseen tiukasti liittyvää tietoa (koordinaatteja, kaavojen muoto, jne). Erityispiirteenä todistus-osajärjestelmä ei koskaan unohda todistuspuun haaroja, joita siihen on kalkyyli-osajärjestelmän avustuksella rakennettu. Kyseinen erityispiirre on tärkeä järjestelmän undo-vastineen kannalta.

Todistus-osajärjestelmä on todistustoimittimen ainoa osajärjestelmä, joka ei ole suoraan riippuvainen muista osajärjestelmistä.

3.3 Käyttöliittymä

Käyttöliittymän tehtävä on esittää todistus 3.2 käyttäjälle ymmärrettävässä muodossa sekä tarjota sitä käyttäen käyttäjälle helppokäyttöinen ja ymmärrettävä käyttörajapinta kalkyyliin. Käyttöliittymä luo jäsentäjää käyttäen todistuksen, jonka kalkyyli tarvitsee toimiakseen.

3.4 Jäsentäjä

Jäsentäjän tehtävä on todistuspuun tuottaminen merkkijonoesityksestä tai merkkijonoesityksen tuottaminen todistuspuusta. Jos sekä vienti että jäsennys on toteutettu, tulee jäsennyksen olla viennin käänteiskuvaus: $\forall x, jasenna(vie(x)) = x$. Vienti ei kuitenkaan ole jäsennyksen käänteiskuvaus, sillä $\forall x \exists y \neq x, jasenna(y) = jasenna(x)$.

Osa II

Jäsentäjä ja tietosisältö

4 Luokkarakenne

Ohjelman rakenteen kuvaus jakautuu kahteen osaan, tietosisällön ja toiminnallisuuden toteuttavan luokkarakenteen kuvaukseen ja käyttöliittymän kuvaukseen. Tässä osassa kuvataan ensin edellisen luokkarakenne (luku 4), jonka jälkeen esitellään muutamia sekvenssikaavioita olioiden yhteistoiminnasta (luku 5).

Seuraavassa osassa kuvataan ensin käyttöliittymä yleisellä tasolla (luku 6), tämän jälkeen esitellään tämän käyttöliittymän toteuttava luokkarakenne (luku 7), ja lopuksi kuvataan sekvenssikaavioilla muutama olioiden yhteistoimintaa vaativa tapaus (luku 8).

Liitteessä 1 on kuvattu ohjelman luokkarakenne UML-kaavioina. Tässä kappaleessa rakenne ja sen suunnittelun motiivit kuvataan tarkemmin.

Ohjelma jakautuu neljään enemmän tai vähemmän itsenäiseen komponenttiin: jäsentäjään, todistusrakenteen tallentajiin, kalkyylin toteuttajaan ja käyttöliittymään.

4.1 Jäsentäjä

Jäsentäjän tehtävänä on tuottaa annetusta merkkijonosta kokoelma jäsennyspuita. Sitä käytetään ohjelmassa kolmessa tarkoituksessa. 1) Kun käyttäjä luo uuden todistuksen ja kirjoittaa kaavan ohjelmaan, jäsentäjä kääntää sen ohjelman käyttämään muotoon. 2) Kun kaava ladataan levyltä, jäsentäjä jäsentää sen. Tallennustiedoston muoto on merkkijono, joka on tarvittaessa editoitavissa tavallisella tekstieditorilla. on merkkijonona, joka pitää kääntää ohjelman käyttämään muotoon. 3) Kun halutaan tallentaa todistus levylle jossain muodossa. Joillekin tallennusmuodoille on olemassa vastaava jäsentäjä, jolloin sekä tallennus- että jäsennysoperaatio tulisi sijoittaa samaan luokkaan.

4.1.1 Merkkijonon muoto

Jäsentäjälle annettava merkkijono noudattaa seuraavaa pseudo EBNF-muotoista[1] määritystä:

```
proof := calculus LF selection? '{' SP* sequent SP* proof* '}'
SP := x in ASCII, 0 < ASCII_VAL(x) <= 32
LF := '\n'
calculus := 'G3IP' | 'G3CP'
sequent := '[' SP* multiset+ SP* ']'
multiset := '[' SP* formula* SP* ']'
formula := '(' SP* (formula SP+)? connective SP+ formula SP* ')' | atom
atom := [A-Z]([a-z][A-Z])*
connective := [a-z]([a-z][A-Z])*
selection := prmset ',' prfmla (',' prrule)?
prmset := [0-9]+
prfmla := [0-9]+
prrule := [0-9]+
```

Kaikki kieliopissa käytetyt merkit tarkoittavat niitä vastaavia ASCII-koodeja. Tulevaisuudessa *atom*:in ja *connective*:n kuvaus saattaa monimutkaistua, ja merkistövaatimus vaihtua, jos siirrytään tukemaan esimerkiksi UNICODE:ssa määritteltyjä loogisia konnektiiveja UTF-8 koodauksella. UTF-8 on ASCII-yhteensopiva koodeilla 0-127.

Tallennuksen ja latauksen tapauksessa lähtösymboli on *proof*, ja käyttäjän syöttämän kaavan tapauksessa tapauksesta riippuen joko *formula* tai *sequent*.

Selection eli *prmset*, *prfmla* ja *prrule* määräävät *proof*-yksilön prinsipaalisen kaavan ja monijoukon (vasen=1, oikea=2) jossa *prfmla*-numerointi pätee. *Prrule* kertoo G3IP-kalkyylin oikeanpuoleista disjunktiota vastaavissa tapauksissa mitä säännöistä on käytetty. Jos kuvaus on yksikäsitteinen ilman *prrule*-määrettä, se voidaan jättää kokonaan pois kieliopissa kuvatulla tavalla.

Tapausta jossa prinsipaalista kaavaa ei olla vielä valittu merkitään jättämällä *selection* kokonaan pois. Tällöin *proof*:n täytyy sisältää *sequent*-välike, eikä yhtään alitodistusta.

Numerointi toimii siten, että numerolla yksi (1) tarkoitetaan joukon ensimmäistä eli vasemmanpuoleisinta alkiota (monijoukon, todistuspuun). Nollalle (0) varataan jokin tulevaisuudessa tarvittu erikoismerkitys, kenties indikaatio katkaisusäännön (cut-rule) soveltamisesta.

[1] Extended Backus-Naur Form

Kuten kieliopista on nähtävissä, konnektiivit ja propositiot voivat olla lähes mielivaltaisia merkkijonoja. Jäsentäjä ei ota kantaa kaavan semantiikkaan muuten kuin erottelemalla propositiot ja konnektiivit toisistaan siten, että propositiot alkavat aina isolla kirjaimella. Yksikäsittelisen ja helposti verrattavan olion kiinnittäminen kuhunkin konnektiiviin jätetään *Calculus*-luokan tehtäväksi.

4.1.2 Jäsentäjäluokat

Jäsentäjä koostuu kolmesta luokasta: *ParsedFormula*, *SequentParser* ja *ProofParser*. ParsedFormula on oksa- ja lehtisolmu binääripuussa, joka on jäsennetyn kaavan esitys. *ParsedFormula*:n konstruktori osaa luoda jäsennyspuun merkkijonoista jotka vastaavat luvun 4.1.1 välikettä *formula*.

SequentParser sisältää metodin joka tuottaa kaksiulotteisen taulukon ParsedFormula-olioita merkkijonosta joka vastaa luvussa 4.1.1 esitettyä välikettä *sequent*.

ProofParser on kuten *SequentParser*, mutta lähtösymboli on *proof*. *ProofParser*:n toteutus käyttää sopivilta osin hyväkseen *SequentParser*:ia.

SequentParser ja *ProofParser*-luokilla on vähän merkitystä jäsennyksen ulkopuolella. Sen sijaan ParsedFormula osallistuu myös tietosisällön esittämiseen.

4.2 Tietosisältö

Tietosisällöltään todistustoimitin jakautuu kalkyyleihin (*Calculus*-aliluokat), tietorakenteisiin (*Proof*, *Formula*, *ParsedFormula*), erilaisiin tietorakenteiden rakentajiin ja tallentajiin (*ProofParser*, *Sequentparser* aliluokat) jotka osaavat joko luoda tietorakenteen tietovirrasta tai tallentaa tietorakenteen tietovirraksi, sekä apuluokista (*Connective*, *Elimination*-aliluokat).

4.2.1 Elimination

Koska Javassa metodit eivät ole olioita, on eliminointimetodeille määriteltävä luokkahierarkkia. *Elimination*-luokat ovat luokkia joista ei tarvitsisi koskaan luoda instansseja, mutta vaikuttaa siltä että Javassa ei ole tapaa välittää luokkaa parametrina, joten instantiointi on välttämätöntä. *Elimination*-instanssin ainoan metodin tehtävä on suorittaa konnektiivin poisto-operaatio halutussa todistuksen solmussa halutun kaavan perusteella.

Elimination-luokan rajapinta on seuraavanlainen:

```
apply(Proof p, int m, int f)
```

Apply-metodi parametrisoidaan sillä todistuspuun solmulla johon sääntöä halutaan soveltaa, tarkennettuna kokonaisluvuilla jotka määrittävät halutun monijoukon ja kaavan sen sisällä.

Selvennyksen vuoksi pseudokoodikuvaus eräästä *Elimination*-luokasta seuraa.

```
class Elimination {
    /*
        library-like methods for use in subclasses.
        makes a sequent from arraylists
    */
    static Formula[][] mkseq(ArrayList a[]) {
        ...
    }
}

class G3IPRightImplicationElimination extends Elimination {
    apply(Proof p, int m, int f) {
        Formula[][] seq = p.children;
        ParsedFormula pf = seq[m][f].parsetree;

        ArrayList left = new ArrayList(seq[LEFT]);
        ArrayList right = new ArrayList(seq[RIGHT]);
        left.addLast(new Formula(pf.left));
        right.insert(f, new Formula(pf.right));
        p.addChildren(m, f, 0, {mkseq({left, right})});
   }
}
```

4.2.2 Calculus

Koska kalkyylejä on useita, on luontevaa määritellä kalkyylille rajapinta, jonka varsinaiset kalkyylin toteuttavat luokat perivät, ja esiintyvät varsinaisessa toimintatilanteessa rajapinnan nimellä.

Varsinaisten *Calculus*-olioiden luominen tapahtuu *Calculus*-luokan staattisella metodilla getCalculus, joka parametrisoidaan kalkyylin määrittelevällä merkkijonolla, joka on määritelty kieliopin 4.1.1 välikkeenä *calculus*.

Calculus-luokan kentät ovat seuraavat.

```
public Proof root;
private abstract static Connective connectives[];
```

Calculus on abstrakti luokka, jossa ainoana abstraktina sisältönä on *Connective*-instansseista koostuva taulukko, jota käyttäen *Calculus*-luokan konstruktori käy muokkaamassa *ParsedFormula*-olioista konnektiiveja esittävät merkkijonot viittauksiksi kyseisen taulukon alkioihin.

Root on käsiteltävän todistuspuun (*Proof*) juurisolmu. *Root* ei saa olla *null* millään *Calculus*-instanssilla.

Jokainen *Calculus* toteuttaa seuraavat operaatiot:

```
Calculus(Proof p) throws ParseError
```

Calculus:ksella tarkoitetaan tässä tapauksessa luokan konstruktoria. Parametrina saatu *Proof* on luotu *ProofParser*-luokan avulla joko tietovirrasta tai käyttäjän syöttämästä merkkijonosta.

Konstruktori käy läpi parametrina saamansa *Proof*-olion kaikki *ParsedFormula*-instanssit ja korvaa kaikista ei-atomisista kaavoista konnektiivia kuvaavat merkkijonot yksikäsitteisesti kyseistä konnektiivia vastaavilla *Connective*-instansseilla. Jos tässä korvaamisessa tapahtuu jokin virhetilanne, heitetään poikkeus *ParseError*.

Konstruktoria ei tyypillisesti kutsuta muualta kuin *Calculus*-luokan getCalculus-metodista.

```
static Calculus getCalculus(Proof p, String name) throws ParseError
```

getCalculus palauttaa parametrina saamaansa merkkijonoa vastaavan *Calculus*-aliluokan instanssin. Tyypillisesti kalkyyli-instanssien luominen tapahtuu käyttäen *getCalculus* metodia *Calculus*-luokasta, jolloin kutsujan ei ole tarpeen tietää mitä kalkyylejä on olemassa. Jos kutsujan tarvitsee jostain syystä tietää mitä eri nimiä *getCalculus* sallii parametreiksi, lista hyväksytyistä nimistä saadaan seuraavalla metodilla.

```
static String[] getValidCalculusNames()
```

Jos tarpeen, kalkyylien toteutukset voivat suorittaa ParsedFormuloille manipulaatiota ennen todistuspuun alkioiden luontia. Tyypillisesti tämä manipulointi on ainoastaan konnektiivinimien korvaamista vastaavilla yksikäsitteisillä olioviitteillä vertailuoperaatioiden tehostamiseksi, minkä vuoksi kyseinen operaatio on nostettu *Calculus*-luokkaan asti. Jatkokehittelyn vuoksi mahdollisuus tehdä suurempia muutoksia *ParsedFormula*-instansseihin on tarpeen, koska katkaisusäännön (cut-rule) toteuttaminen on mahdollista lisäämällä uusien atomien esittelyn lisäksi atomin korvaaminen kaavalla. Tällöin on miellyttävää, jos kaavan atomi-instanssit ovat yksikäsitteisiä, ja tämä yksikäsitteistämisvaihe on siis jätettävä mahdolliseksi kalkyylille.

Nykyisellään tämä on mahdollista määrittelemällä oma konstruktori kyseiselle kalkyylille, ja suorittamalla tarvittavat lisämanipulaatiot heti yliluokan (*Calculus*) konstruktorikutsun jälkeen.

```
apply(Proof proof, int formula, int multiset)
```

Apply soveltaa kalkyyliä valitun todistuksen (proof) valittuun kaavaan (formula, multiset). On selvää, että kutsujalla täytyy olla pääsy kalkyylin käyttämään todistuspuuhun, jotta tätä metodia voitaisiin ylipäätään käyttää.

```
undo(void)
saveundo(Proof p)
```

Undo peruuttaa edellisen tehdyn *apply*:n. Undo on toteutettu siten, että kalkyylissä on undoa varten kentät joiden sisältönä on viimeksi käsitelty proof-instanssi, ja sitä koskevat kokonaisluvut (taulukoiden indeksit) multiset ja formula, ja formulaa koskevasti vastaava rule (sovelletun säännön kalkyylikohtainen järjestysnumero).

Saveundo pitää huolen siitä, että seuraava Undo-kutsu palauttaa tilanteen parametrina annetun Proof-olion suhteen sellaiseksi kuin se oli ennen *saveundo*-kutsua.

Calculus-luokan toteutuksen ylimalkainen kuvaus Javalla seuraa.

```
abstract class Calculus {

    private Proof root;
    private abstract static Connective connectives[];

    private Proof uproof;
    private int umset, ufmla, urulenum;

    private setConnectives(ParsedFormula pf) throws ParseError {
        if (pf == null)
            return;
        if (pf.data == null)
            throw new ParseError("null data in ParsedFormula?!");
        for (int i = 0; i < connectives.size; ++i) {
            for (int j = 0; j < connectives[i].names.size; ++j) {
                if (!pf.data.instanceOf("String"))
                    throw new ParseError("Connectives already set?");
                if (connectives[i].names[j].equals((String)pf.data)
                    pf.data = connectives[i];
                if (pf.data.instanceOf("String"))
                    throw new ParseError("no such connective:" +
                        (String)pf.data);
            }
        }
        setConnectives(pf.left);
        setConnectives(pf.right);
    }

    private setConnectives(Proof p) {
        if (p == null)
            return;
        for (int m = 0; m < p.formulas.size; ++m)
            for (int f = 0; f < p.formulas[m].size; ++f) {
                Formula fmla = p.formulas[m][f];
                setConnectives(fmla.parsetree);
                for (int r = 0; r < fmla.children.size; ++r)
                    for (int c = 0; c < fmla.children[r].size; ++c)
                        setConnectives(fmla.children[r][c]);
            }
        }
```

```
    }

    Calculus(Proof p) {
        setConnectives(p);
    }

    String[] getValidCalculusNames() {
        return { "G3IP", "G3CP" };
    }

    Calculus getCalculus(Proof p, String name) {
        Calculus c = null;
        if (name.equals("G3IP"))
            c = new G3IPCalculus(p);
        else if (name.equals("G3CP"))
            c = new G3CPCalculus(p);
    }

    apply(Proof p, int m, int f) {
        saveundo(p);
        ((Connective)(p.children[m][f].parsetree.data)).
            elimination[m].apply(p, m, f);
    }

    saveundo(Proof p) {
        uproof = p;
        umset = p.mset; ufmla = p.fmla; urulenum = p.fmla.rulenum;
    }
}
```

Calculus-luokan *apply*-toteutus delegoi varsinaisen operaation sille eliminointiluokalle, joka *apply*:n kohteena olevaan konnektiiviin on liitetty.

Calculus noudattaa ns. Tehdasmetodi-suunnittelumallia jossa luokka sisältää metodin jonka avulla voi luoda instanssin haluamastaan luokkahierarkkian jäsenestä. Yleiset toiminnot on sijoitettu *Calculus*-luokkaan, jolloin tiettyihin kalkyyleihin sidotut luokat voivat olla niinkin yksinkertaisia kuin seuraava koodilistaus antaa ymmärtää.

```
class G3CPCalculus extends Calculus {
    private static Connective connectives[] = {
        new Connective({"->", "impl", "implication"},
            {new G3CPLeftImplicationElimination(),
             new G3CPRightImplicationElimination()}),
        new Connective({"|", "or", "disjunction"},
            {new G3CPLeftDisjunctionElimination(),
             new G3CPRightDisjunctionElimination()}),
        new Connective({"&", "and", "conjunction"},
```

```
            {new G3CPLeftConjunctionElimination(),
             new G3CPRightConjunctionElimination()})
    };
}
```

Kalkyylikohtaisessa luokassa riittää toteuttaa ainoa *Calculus*-luokassa abstraktiksi määritelty osa, eli *connectives*-taulukko, jota *Calculus*-luokassa määritelty konstruktori käyttää hyväkseen käydessään läpi parametrina saamaansa todistuspuuta korvatakseen sen *ParsedFormula*-olioista konnektiiveja vastaavat merkkijonot *Connective*-olioilla.

4.2.3 Proof

Todistus (*Proof*) ja kaava (*Formula*) muodostavat yhdessä todistuspuun rakenteen. Syy siihen että rakenteen rekursio kiertää kaavan kautta on undo-operaation toteutus. Kalkyyleissä voi nimittäin olla useita sääntöjä joita voidaan soveltaa samaan kaavaan, jolloin sovelletun säännön numeron tallentaminen todistukseen olisi tarpeettoman monimutkaista.

Konseptuaalisesti rekursio tapahtuu indeksoimalla neliulotteisen taulukon kolmea ensimmäistä ulottuvuutta. Ensimmäisellä indeksillä valitaan monijoukko (multiset), joka on tyypilisesti (ainakin kaikissa kirjoittajan tuntemissa sekventtikalkyyleissä) vain kaksiarvoinen (oikea, vasen). Toinen indeksi valitsee prinsipaalisen kaavan valitun sekventin sisältä (kalkyyli pystyy jatkamaan todistusta näiden indeksien varassa). Kolmannella indeksillä valitaan sovellettu sääntö siitä sääntöjen joukosta jota kyseiseen kaavaan kyseisessä monijoukossa voidaan soveltaa. Viimeinen ulottuvuus numeroi todistuspuun haarat, joita tunnetuissa kalkyyleissä on korkeintaan kaksi.

Proof:n kentät ovat seuraavat.

```
public Formula sequent[][]
public int mset, formula
```

Sequent-taulukon ensimmäinen ulottuvuus määrittää monijoukkoparin (0=vasen, 1=oikea), ja toinen kaavajoukon sen sisällä. Kyseessä ei siis ole varsinaisesti kaksiulotteinen taulukko, vaan vektoripari.

Proof sisältää seuraavanlaisia metodeja.

```
Proof(Formula sequent[][])
Proof(ParsedFormula sequent[][])
```

Peruskontruktori luo uuden todistuksen, joka sisältää parametrina annetun sekventin, eli 2-ulotteisen taulukon kaavoja. Variaatio luo itse *Formula*-oliot, ja ottaa parametrina jäsennyspuut.

```
AddChildren(int multiset, int formula, int rule, Proof children[])
removeChildren(int multiset, int formula, int rule)
```

AddChildren lisää parametrina saadut alipuut (*children*) indeksien määrittämään paikkaan taulukossa. *RemoveChildren* on vastaava poisto-operaatio.

```
getVisibleChildren(void)
```

GetVisibleChildren palauttaa tällä hetkellä näkyvissä olevat todistuksen haarat, jotka ovat jatkoa kyseiselle solmulle.

```
int setVisibleFormula(int)
int setVisibleMultiset(int)
```

SetVisibleMultiset ja *setVisibleFormula* asettavat näkyvän monijoukon ja kaavanumeron kyseisen monijoukon sisällä. Paluuarvo on kyseisen attribuutin vanha arvo, jolla saman metodin kutsuminen palauttaa tilanteen sellaiseksi kuin se oli ennen asettamista, toisin sanoen $setVisible(setVisible(k)) == k$.

4.2.4 Formula

Formula sisältää sen valinnasta seuraavien todistusalipuiden lisäksi kyseisen kaavan jäsennetyn muodon (*ParsedFormula*). Formula toteuttaa Javan kloonirajapinnan *Cloneable*.

Formula sisältää seuraavanlaiset kentät

```
Proof children[][]
int rulenumber
```

Children sisältää kyseisen kaavan prinsipaaliseksi valitsemista seuraavat alitodistukset. Ensimmäinen ulottuvuus on sovellettavan säännön numero tapauksissa kuten G3IP-kalkyylin oikeanpuoleinen implikaatio, jossa pelkän kaavan valitseminen ei määrää yksikäsitteisesti todistuksen jatkoa. Toinen ulottuvuus numeroi alitodistukset (0=vasen, 1=oikea haara).

Rulenumber kertoo G3IP:n oikean implikaation kaltaisissa tapauksissa mikä mahdollisista vaihtoehdoista on kullakin hetkellä valittuna.

```
Formula(ParsedFormula f)
```

Konstruktori luo uuden kaavan ja kiinnittää sille jäsennyspuun. Jäsennyspuutonta kaavaa ei ole olemassa, joten sellaisen luomiseen ei tule antaa konstruktoria.

```
addChildren(int rulenumber, Proof children[]);
```

AddChildren lisää todistukseen jatkon sääntönumerolla *rulenumber*.

```
clone(void)
```

Clone palauttaa ohuen kopion kaavasta, toisin sanoen kopion jossa Formulan osaluokkien instansseja ei ole kopioitu, vaan ne ovat viitteitä vanhoihin. Lisäksi clone poistaa mahdolliset olemassaolevat alitodistukset luodusta kopiosta. Jos tämä toimintatapa huomataan kloonausrajapinnan hengen vastaiseksi, *clone*:n nimi vaihdetaan, toiseksi ja muutetaan *Formula* luokaksi joka ei toteuta kloonirajapintaa, jolloin jokaisesta kloonausyrityksestä saadaan miellyttävästi käännösaikainen virhe.

4.2.5 ParsedFormula

ParsedFormula esittää kaavan jäsennyspuuta ja on siten rekursiivisesti määritelty tietotyyppi (binääripuu). *ParsedFormula* toteuttaa Javan kloonirajapinnan *Cloneable* ja sisältää seuraavat metodit.

`Parse(Stream s)`

Parse jäsentää tietovirrasta kaavan kieliopin 4.1.1 välikkeen *formula* kuvaamassa muodossa. Jäsentäjä ei kiinnitä operaattoreiden nimiä vaan sen sijaan erottelee atomit ja konnektiivit pienen ja suuren alkukirjaimen (tai jos myöhemmin nähdään tarpeelliseksi, jonkin tarkennetun erityispiirteen) perusteella. *Parse* tallentaa jokaiseen solmuun sillä paikalla olleen merkkijonon (*String*). Lisäksi se jättää kalkyylille (*Calculus*) niiden mahdollisen korvaamisen tietokoneelle helpommin käsiteltävissä olevilla olioilla (*Object*).

`writeToStream(Stream s) throws IOException`

WriteToStream kirjoittaa todistuspuun tietovirtaan muodossa josta *Parse* osaa jäsentää nykyisen todistuksen kanssa isomorfisen jäsennyspuun. Katso kielioppi 4.1.1.

`clone(void)`

Clone palauttaa syvän (rekursiivisen) kopion kaavasta. Kloonausta käytetään lähinnä luokan itsensä sisällä, kun toteutetaan vasemmalle assosioiva jäsennys konnektiiveille.

4.2.6 ProofParser

ProofParser luokkien tehtävä on luoda kokonaisia todistuspuita parametrina saadun tietovirran perusteella. Rajapintaan kuuluu, että *SequentParser* on osa *ProofParser*:ia, mutta toteutuksen ei ole pakko sitä hyödyntää. Luontioperaatiolle *Parse* on rajapinnassa vastaava käänteisoperaatio *writeToStream*, ja luokkien jotka toteuttavat sekä luonnin että tallennuksen tulisi varmistaa että tallennettu tieto on luettavissa siten että luetun perusteella muodostettu todistuspuu on samanlainen kuin se, jonka perusteella tallennusoperaatio suoritettiin. Toinen näistä operaatioista voidaan jättää toteuttamatta, jolloin toteuttamatta jätetyn operaation tulisi heittää poikkeus *ParserOperationNotImplementedError*.

ProofParser on abstrakti luokka, jonka varsinaiset toteutukset perivät. Kaikki muut *ProofParser*:n metodit ovat abstrakteja paitsi tehdasmetodi *getProofParser*, joka parametrisoidaan halutun *ProofParser*-aliluokan nimellä. Tarvittaessa *ProofParser*:lta voidaan pyytää lista tuetuista nimistä metodilla *getProofParserNames*.

ProofParser-luokkia ei ole tarkoitettu instantioitaviksi, mutta koska Javassa ei ole mahdollisuutta välittää instantioimatonta luokkaa parametrina, on instansseja kuitenkin luotava. Koska instanssiin ei tallenneta mitään tilatietoa, rajoitetaan luotujen instanssien määrä ajoympäristössä yhteen käyttämällä Singleton-suunnittelumallia, josta seuraa että *ProofParser*:lla on yksi kenttä.

```
static ProofParser instance
```

Instance-kenttä asetetaan ensimmäisellä *getInstance*-kutsulla luoduksi olioksi. Kaikki *getInstance*-kutsut palauttavat *instance*-kentän viittaaman *ProofParser*-instanssin.

ProofParser sisältää seuraavat metodit.

```
static getInstance()
static ProofParser getProofParser(String name)
static String[] getProofParserNames()
```

getProofParser palauttaa halutun *ProofParser*-aliluokan instanssin käyttäen kyseisen luokan *getInstance*-metodia.

```
static abstract Proof parseProof(Stream s)
static abstract writeToStream(Stream s, Proof p) throws IOException
```

ParseProof on yllä kuvattu jäsennusoperaatio, ja *writeToStream* sitä vastaava tallennusoperaatio. Metodit ovat abstrakteja itsessään *ProofParser*:ssa, ja ne on implementoitava varsinaisissa toteutuksissa.

Lyhyt pseudokoodilistaus seuraa selvennyksen vuoksi.

```
class ProofParser {
    static SequentParser seqparser;
    static ProofParser instance = null;
    static getInstance() {
        if (instance == null)
            instance = new ProofParser();
        return instance;
    }
    static getProofParser(String name) {
        if (name.equals("proof.sty"))
            return ProofStyProofParser.getInstance();
        if (name.equals("pimpelipom"))
            return PimpeliPomProofParser.getInstance();
    }
    static String[] getProofParserNames() {
        return { "proof.sty", "pimpelipom" };
    }
    static abstract Proof parseProof(Stream s);
    static abstract writeToStream(Stream s, Proof p) throws IOException;
}

class ProofStyProofParser {
    ProofStyProofParser {
        seqparser = new ProofStySequentParser();
```

```
    }
    static Proof parseProof(Stream s) {
        ...
    }
    static writeToStream(Stream s, Proof p) throws IOException {
        ...
    }
}
```

4.2.7 SequentParser

SequentParser luokkien tehtävä on luoda tietovirran perusteella kaksiulotteisia taulukoita jotka sisältävät kaavoja (*Formula*), ja haluttaessa myös tuottaa tietovirtaan merkkijonoja jotka saman luokan jäsentäjämetodi pystyy jäsentämään tallennetun sekventin kanssa isomorfiseksi. Toinen näistä operaatioista voidaan jättää toteuttamatta, jolloin toteuttamatta jätetyn operaation tulee heittää poikkeus *ParserOperationNotImplementedError*.

SequentParser on rajapinta (*interface*), jonka varsinaiset toteutukset perivät.

SequentParser sisältää seuraavat metodit.

```
static ParsedFormula[][] parseSequent(Stream s)
static writeToStream(Stream s, ParsedFormula[][] seq)
```

Kyseisten metodien toiminta on analoginen vastaavien *ProofParser*:n metodien kanssa.

5 Tietorakenteiden toiminta

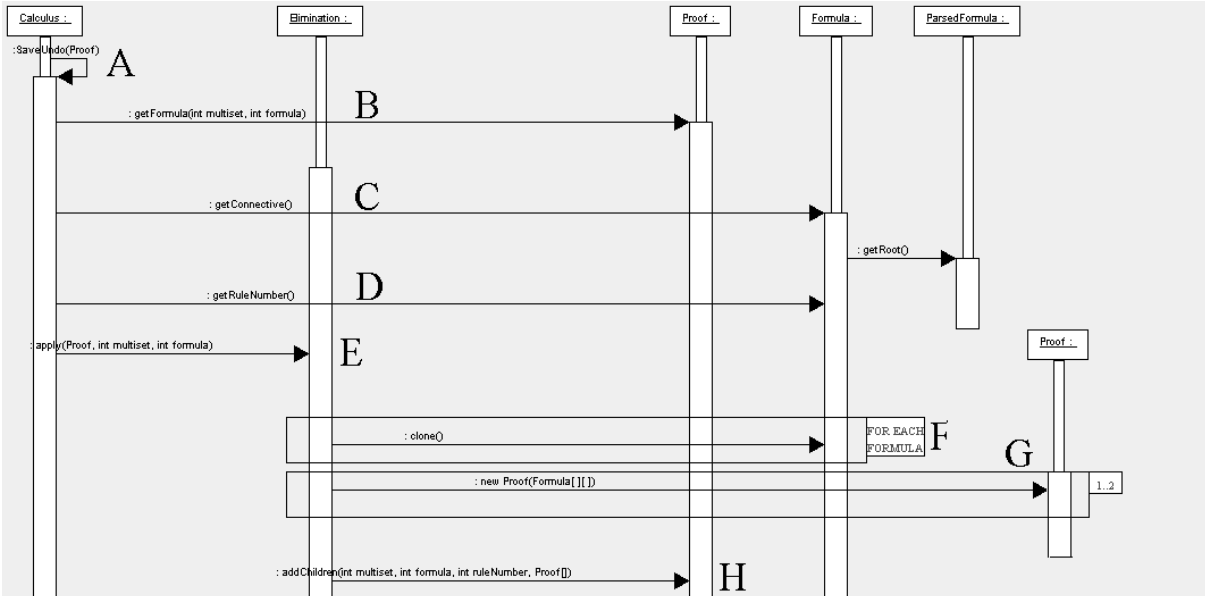
Seuraavaksi tutkitaan ohjelman tietorakenteiden toimintaa. Toiminta tulee parhaiten esille niissä käyttötapauksissa, joissa tapahtuu kalkyylin soveltamista (apply), tietovirran lukemista (read), tietovirtaan kirjoittamista (write), tai joissa tehdään viimeisimmän toiminnon peruuttaminen (undo). Tutkitaan kutakin tapausta erikseen sekvenssikaavioilla.

Erityistä huomioitavaa kaavioissa on se, että niissä käytetään selkeyden vuoksi arvoja asettavia ja palauttavia metodeja (aksessorit), vaikka niitä ei oikeasti toteutetakaan. Osaan metodeista on kirjoitettu myös paluuarvo, jos se ei tule ilmi asiayhteydestä.

5.1 Apply

Kun käyttäjä klikkaa todistuksen tiettyä kaavaa, käyttöliittymärakenne kutsuu *Calculus*-luokan metodia apply, joka soveltaa kalkyyliä kaavaan. Tätä ennen käyttöliittymä on selvitettävä koordinaattien avulla mistä todistuksen osasta, sekventistä ja kaavasta on kyse.

Kuvan 5 kaaviossa nähdään vaiheissa A-H kuinka apply-metodi suoritetaan.



Kuva 2: Sekvenssikaavio: apply

Ensimmäiseksi talletetaan setUndo()-metodilla tiedot, joilla Apply voidaan peruuttaa (vaihe A). Seuraavaksi kalkyyli saa *Proof*-oliolta viitauksen klikattuun formulaan (vaihe B). Formulalta kysellään sen konnektiivia ja sovellettua kaavanumeroa (vaiheet C ja D).

Seuraavaksi poistetaan klikattu konnektiivi kutsumalla sitä vastaavaa eliminointimetodia (vaihe E). Eliminointi aloitetaan kloonaamalla kaikki *Proof*-olion kaikki formulat (vaihe F), koska nämä kopioidaan lapsisolmuihin. Myös klikattu *Formula* kloonataan, mutta tältä otetaankin vasen ja oikea alipuu lapsisolmuihin. Tätä operaatiota ei näy kuvassa 5.

Lopuksi luodaan vasen ja oikea lapsisolmu saadulla formulataulukolla (vaihe G) ja sitten metodilla createChildren(..) luodaan todistuspuun isäsolmulle viittaukset uusiin lapsisolmuihin (vaihe H).

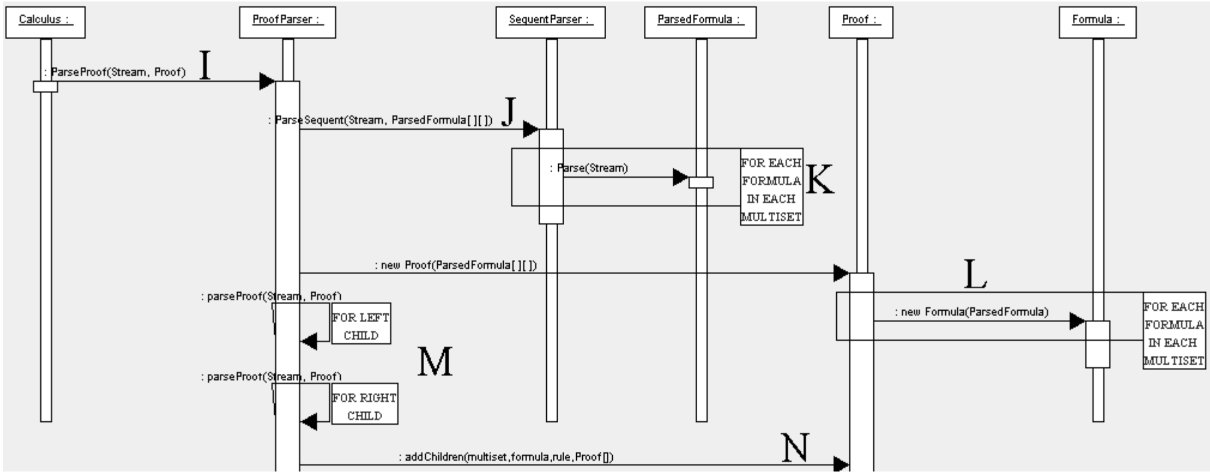
5.2 Read

Ohjelma lukee tietovirtaa silloin, kun käyttäjä syöttää uuden kaavan tai lataa tiedostosta vanhan todistuksen. Kummassakin tapauksessa toiminta on samanlaista. Kuvan 5 kaavion vaiheissa A-E nähdään, kuinka tietovirta jäsennetään todistuspuuksi. Tietovirran tarkempi esitysmuoto löytyy kappaleesta 2.1.1.

Mihin tämä 2.1.1 viittaa?

Aluksi Kalkyyli pyytää *ProofParseria* jäsentämään tietovirran (vaihe A). ParseProof-metodin tarkempi toiminta nähdään ao. pseudo-Javakoodista.

```
public static void parseProof(Stream stream, Proof p) {
    int multiset = read_multiset_from(stream);
    int formula  = read_formula_from(stream);
```



Kuva 3: Sekvenssikaavio: read

```
    ParsedFormula[][] pf;
    SequentParser.ParseSequent(stream, pf); //J
    p = new Proof(pf); //L

    Proof children  = new Proof[2];
    children[LEFT].parseProof(stream);  //M
    children[RIGHT].parseProof(stream);

    p.addChildren(multiset,formula,ruleNumber,children); //N
}
```

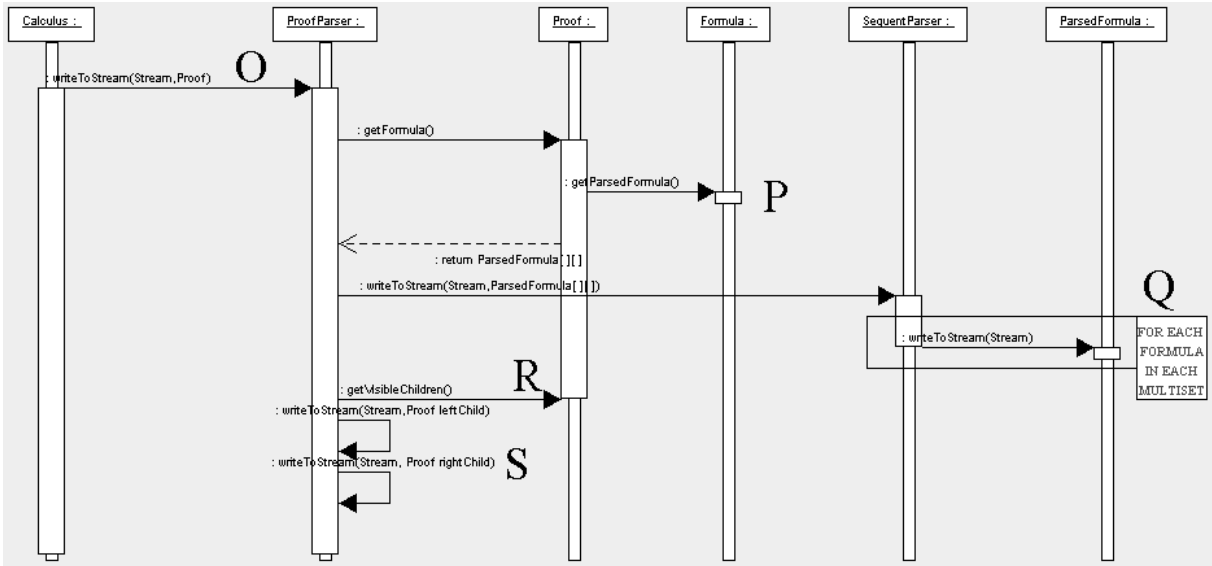
Koodissa näkyvät samat vaiheet J, L, M ja N kuin kuvassa 5. Ensiksi luetaan tietovirrasta, mistä sekventistä ja kaavasta todistus jatkuu eteenpäin (vaihe I). Seuraavaksi tietovirta sysätään sekventtiparsijalle (vaihe J), joka puolestaan pyytää *ParsedFormulaa* jäsentämään sen (vaihe K). Jäsennetty tieto talletetaan *ParsedFormula*-taulukkoon, jossa ensimmäinen indeksi kertoo sekventin numeron ja toinen indeksi formulan numeron (Vaihe K).

Uusi todistus ja sen kaikki formulat luodaan saadun taulukon perusteella (vaihe L). Tämän jälkeen kutsutaan parseProofia rekursiivisesti kahteen kertaan. Ensimmäinen rekursiivinen kutsu luo vasemmanpuoleisen lapsen ja toinen oikeanpuoleisen lapsen. (vaihe M). Lopuksi isäsolmuun p luodaan viittaukset lapsiin (vaihe N).

5.3 Write

Tietovirtaan kirjoitetaan silloin, kun käyttäjä haluaa tallettaa todistuksen tai tulostaa sen. Kuvassa 5 nähdään vaiheissa O-S kuinka todistuspuu kirjoitetaan tietovirtaan.

Käyttötapaus alkaa siitä, kun kalkyyli pyytää *ProofParseria* kirjoittamaan virtaan todistuspuun (vaihe O). *ProofParser* pyytää *Proofilta* todistuksen kaavat taulukkona (Vaihe



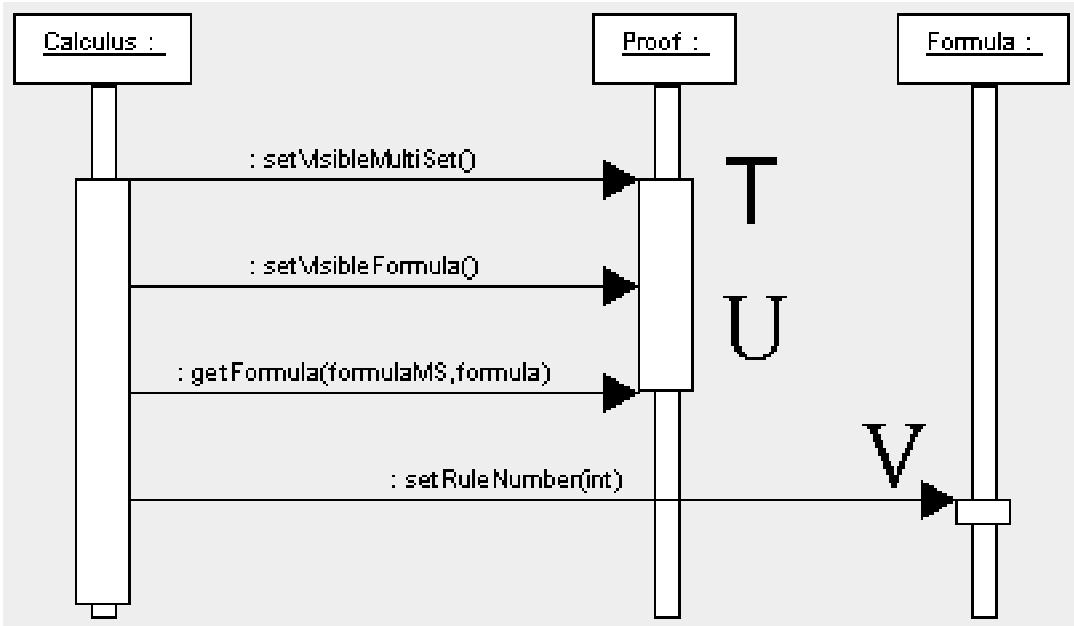
Kuva 4: Sekvenssikaavio: write

P). *ProofParser* kutsuu *SequentParseria*, joka osaa tulkita taulukkoa *ParsedFormulan* avulla ja kirjoittaa kaavat tietovirtaan (vaihe Q). Lopuksi *Proofparser* selvittää saamansa todistuksen lapsisolmut (Vaihe R) ja suorittaa vaiheet B-D rekursiivisesti kullekin lapsisolmulle (Vaihe S).

5.4 Undo

Undo peruuttaa viimeisimmän applyn. Käytännössä todistuspuun oksien näkyvyyksiä muutetaan siten, että uusi oksa katoaa näkyvistä ja vanha palautetaan näkyviin. Kuvassa 5 nähdään, kuinka se vaihe vaiheelta tapahtuu.

Kalkyyliin on talletettu tiedot todistuksen vanhasta näkyvyydestä. Kun Undo käynnistyy, kalkyyli asettaa *Proofille* vanhat talletetut näkyvyysarvot ja saa paluuarvona uudet näkyvyydet jotka se tallettaa kenttiinsä (Vaihe T). Tästä syystä undon undo toimii kuten redo. Seuraavaksi Kalkyyli pyytää todistukselta sitä kaavaa, johon kalkyyliä on sovellettu (vaihe U) ja kasvattaa kaavan sääntönumeroa (vaihe V), jos sovellettavia sääntöjä oli kaavalle useita.



Kuva 5: Sekvenssikaavio: undo

Osa III

Käyttöliittymä

6 Käyttöliittymä

Ohjelman käyttöliittymän perusyksikkö on ikkunapari, joista toinen, pääikkuna (luokka *MainWindow*), sisältää käsillä olevan todistuksen, valikot sekä pikapainikkeet. Toinen ikkuna sisältää pelkästään navigointipuun. Navigointipuuikkuna voidaan myös piilottaa ja sen saa takaisin esille valikosta.

Ikkunan otsikkopalkissa lukee avoinna olevan tiedoston nimi ja ohjelman nimi. Siinä on myös merkintä siitä, onko tiedosto tallennettu. Jos todistus on luotu Uusi todistus -ikkunalla ja sitä ei olla tallennettu vielä kertaakaan, otsikkopalkissa tiedoston nimenä on "Untitled 1.ped", seuraavan luodun todistuksen nimenä on "Untitled 2.ped" jne. Otsikkopalkissa voisi lukea esim. "myproof.ped* - ProEdit" jossa tähti merkitsee sitä, ettei tiedostoa ole tallennettu. Navigointi-ikkunassa vastaavasti esim. "myproof.ped* - Navigation".

Avoinna voi olla kerrallaan useita ikkunapareja. Yhden ikkunaparin tunnistaa toisiinsa liittyväksi ikkunan otsikoista, joissa näkyy sama tiedostonimi. Jos avoinna on useampia todistuksia, joilla on samanniminen tiedosto, otsikkopalkissa on myös juokseva numero osoittamassa näiden kahden ikkunan riippuvuuden toisistaan. (Tämä numero on eri kuin yllämainittu Untitled-numerointi, ohjelmassa voi siis periaatteessa olla avoinna todistukset "Untitled 1.ped - 1" ja "Untitled 1.ped - 2".) Avoinna on aina vähintään yksi ikkunapari. Molempien ikkunoiden kokoa voi muuttaa. Tällöin kyseisen todistusalueen tai navigointialueen koko muuttuu, nappuloiden pysyessä ennallaan.

Jokainen ohjelman käyttämä dialogi-ikkuna on modaalinen, eli se estää kaiken toiminnan muilta ikkunoilta päällä ollessaan. Tästä voi aiheutua pientä haittaa, jos ohjelmassa on auki useita todistuksia. Silloin esimerkiksi yhden todistuksen vienti-ikkunan avaaminen estää jonkin kokonaan toisen todistuksen editoinnin. Tämä on huomioitu ja todettu ettei siitä ole liian suurta haittaa.

Jokaisen dialog-ikkunan voi sulkea ikkunointijärjestelmän sulkemistoiminnolla ("ruksista"). Jonkinlaisen peruutusnappulan (Cancel, Close) tarjoavissa dialogeissa tämä vastaa kyseisen nappulan painamista. Jos dialogissa on vain Ok-nappula, sulkeminen vastaa sen painamista.

6.1 Pääikkunan sisältö

Pääikkunassa on kolme valikkoa, File, Edit ja Help. File-valikko sisältää seuraavat toiminnot:

- New... - Avaa ikkunan, johon kirjoitetaan uusi todistettava kaava.
- Load... - Avaa standardin tiedostonlatausdialogin, jolla valittu tiedosto ladataan näkyviin.
- Save - Jos tiedostolle ei ole asetettu nimeä, toimii kuten Save as..., muuten tallennetaan todistus tiedostoon.
- Save as... - pyydetään standardidialogilla tiedostonimi, johon todistus tallennetaan.
- Export... - Avaa vientidialogin, jolla todistus voidaan viedä tulostuskelpoiseen tiedostoon.
- Close - Sulkee avoinna olevan ikkunan, tai jos ikkuna oli viimeinen, koko ohjelman. Varmistusdialogi ennen sulkemista.

This sentence no verb.

- Exit - Sulkee kaikki ikkunat ja ohjelman. Varmistusdialogi ennen tätä.

Seuraavana on pieni Edit-valikko, jossa on seuraavat:

- Undo - Toteuttaa undo-toiminnon. Toimii kuten allakuvattu Undo-nappula.
- Redo - Toteuttaa redo-toiminnon. Toimii kuten allakuvattu Redo-nappula. Oletuksena disabloitu.
- Hide navigation window - Piilottaa navigointi-ikkunan, valikkoon tulee tällöin checked-valinta. Navigointi-ikkuna voidaan piilottaa myös sulkemalla se ikkunan "ruksista", jolloin checked-valinta syttyy myös. Tällöin valitseminen palauttaa ikkunan ruudulle ja poistaa valinnan.
- Large font - Valitsee suuremman fontin todistusalueen kaavoille, jolloin valikkoon tulee checked-valinta. Uudelleen valitseminen palauttaa pienen fontin ja poistaa checked-valinnan.

Help-valikko sisältää vain About...-valinnan joka avaa ohjelmasta kertovan dialogi-ikkunan.

Pääikkunan vasen puolisko on varattu todistusalueelle (luokka *ProofArea*), oikealle puolelle on sijoiteltu pikapainikkeet: kaikki file-valikon painikkeet paitsi ehkä Close ja Exit, ja lisäksi Undo- ja Redo-nappulat. Näistä toinen on aina poiskytkettynä, normaalitilassa Redo. Kun Undo-nappulaa painetaan, Redo vaihtuu aktiivisesti ja Undo kytketään pois. Kun todistus ruudulla muuttuu, nappuloiden kytkentä palaa normaalitilaansa, eli Undo aktiiviseksi ja Redo poiskytketyksi. Todistus muuttuu, jos käyttäjä klikkaa joko jotain konnektiivia todistuksessa tai Redo-nappulaa. Sama päälle- ja poiskytkentä tapahtuu myös valikossa oleville Undo- ja Redo-valinnoille.

Jossain (ehkä painikkeiden yhteydessä tai otsikkorivillä) lukee myös käytetyn kalkyylin nimi.

6.2 Todistusalue

Pääikkunan vasen puoli koostuu "todistusalueesta", johon itse todistuspuu piirretään. Alue on valkeataustainen ja puu on piirretty sille samaan tapaan kuin se olisi kirjoitettu paperille.

Todistuspuun yksi solmu koostuu sekventistä ja siihen mahdollisesti sovelletusta säännöstä. Solmun lapset ovat tästä solmusta valitulla säännöllä johdetut uudet sekventit (joita voi olla käyttöliittymän nykytoteutuksessa, ja tunnetuissa kalkyyleissä, enintään kaksi).

6.2.1 Solmun ulkoasu

Solmu piirretään seuraavasti: sekventin vasemman monijoukon kaavat piirretään pilkuilla erotettuna, jonka jälkeen tulee niinsanottu sekventtinuoli erottamaan monijoukot toisistaan, ja oikean monijoukon kaavat pilkuilla erotettuna. Kaavojen atomikaavoihin ja sulkuihin tarvitaan tavallisia merkkejä. Nämä voidaan tehdä kahdella eri tavalla: joko järjestelmästä löytyvällä sopivalla fontilla (esim. Arial) tai piirtää kukin erillisistä bittikartoista. Jälkimmäistä vaihtoehtoa puoltaa se, että eri järjestelmien tarjoamat fonttivaihtoehdot saattavat vaihdella huomattavasti ja ohjelman ulkoasua ei näitä käyttämällä pystyttäisi takaamaan. Bittikartat voidaan tuottaa vaikkapa LaTeX:illa. Toteutusvalinta voidaan jättää toteutusvaiheeseen, käyttää toteutuksessa aluksi vaikkapa järjestelmän fontteja ja vaihtaa tilalle bittikartat, kun ne saadaan valmiiksi.

Ainakin sekventtinuolen ja muiden fontista löytymättömien merkkien piirtämiseen käytetään bittikarttoja, jotka voivat myös olla sopivan levyisiä, mutta korkeus on sama kuin fontilla.

Piirrettäessä käytetään eri värejä, jotka on kuvattu kappaleessa 6.2.6.

Jos solmuun on sovellettu sääntöä, solmun yläpuolelle piirretään viiva. Viivan yläpuolelle siis piirretään lapsisolmu tai jos lapsia oli useampia, solmut vierekkäin sopivan vaakasuuntaisen välimatkan päähän toisistaan. Lapsilla voi olla myös omia lapsia, joten tätä välimatkaa pitää suurentaa, jos vasen ja oikea alipuu menisivät päällekkäin. Solmujen välissä olevan viivan leveys riippuu siitä, kuinka leveitä solmun lapset ovat. Viiva ylettyy vähintään tämän sekventin vasemmasta reunasta oikeaan ja tämän sekventin vasemman lapsisekventin vasemmasta reunasta oikean lapsisekventin oikeaan reunaan. Solmun korkeus on aina vakio. Se riippuu käytetystä fontista ja päätetään toteutusvaiheessa.

Sovellettu sääntö näkyy viivan oikealla puolella sopivan matkan päässä. Sääntö voi olla vaikkapa "R $\rightarrow$". Tämän tekstin pystysuuntainen keskikohta on viivan korkeudella. Säännön näyttämiseen käytetään samaa merkistöä ja bittikarttoja kuin kaavoille.

Jos solmu on käytetyn kalkyylin mukaan tosi, sillä ei ole teknisessä (tietorakenteiden osalta) eikä graafisessa (kuvaan piirtämisen osalta) mielessä lapsia. Sen yläpuolelle piirretään myös tällöin viiva. Se sääntö, jonka perusteella solmu on tosi, kirjoitetaan viivan oikealle puolelle.

Fontti voi olla kahta kokoa, joko normaalikokoista tai suurta, jos ohjelmaa halutaan käyttää esimerkiksi videoprojektorilla. Molempia fonttikokoja varten on eri bittikartat erikoismerkeille.

6.2.2 Konnektiivin klikkaaminen

Käyttäjä voi klikata mitä tahansa ruudulla näkyvää konnektiivia. Perustapauksessa konnektiivi on lehtisolmussa, yleisessä tapauksessa se voi olla missä kohdassa tahansa (ks. kappale undo-toiminnosta).

Tällöin ohjelma laskee ne säännöt, joita pitää soveltaa siihen kaavaan, jossa olevaa konnektiivia klikattiin, niin että klikattu konnektiivi häviää. Käytettävät säännöt selviävät yksiselitteisesti, tai jos jossain myöhemmin toteutettavassa kalkyylissä näin ei olisi, käytettäisiin ensimmäistä sopivaa sääntöä. Jos ruudulla on esimerkiksi kaava "(A $\vee$ (B $\wedge$ C))", konjunktion klikkaaminen tuottaa ensin disjunktiosäännön käyttämisen, josta tuotettun sekventin kaavaan "(B $\wedge$ C)" sovelletaan konjunktiosääntöä.

Kun sääntöä sovelletaan, solmulle luodaan kalkyylin perusteella yksi tai kaksi lapsisolmua. Kalkyyli kertoo millaiset monijoukot lapsisolmuille tulee. Säännön soveltaminen poistaa kaavasta konnektiivin ja jakaa konnektiivin vasemmalla ja oikealla puolella olevan kaavan lapsisolmujen monijoukkoihin.

Säännön soveltaminen muuttaa myös ruudun keskitystä (ks. kappale 6.2.4).

Jos klikattiin sellaista konnektiivia, johon kalkyylissä on vain yksi sääntö, ei puulle tapahdu mitään (alue kuitenkin keskitetään kappaleen 6.2.4 mukaisesti). Jos konnektiiville on olemassa useita sääntöjä, klikkaaminen purkaa ruudulle järjestyksessä (kalkyylin määrittelemä) seuraavan säännön.

6.2.3 Geneerinen undo ja eksplisiittinen undo

Ohjelman todistuspuun tallettava tietorakenne on toteutettu niin, että mitään käyttäjän joskus todistamisen aikana soveltaman säännön tulosta ei unohdeta ohjelman suorituksen aikana. Käyttöliittymässä tämän hyväksikäyttäminen tehdään seuraavasti:

Käyttäjä voi klikkata konnektiivia, joka ei ole lehtisolmussa. Tällöin valitun säännön soveltaminen tähän konnektiiviin aiheuttaisi sen, että tässä puun solmussa jossa aiemmin sovellettiin sääntöä S pitäisi soveltaa nyt eri sääntöä T, vanha alipuu joka oli tehty sääntöä S soveltamalla pidetään muistissa. Jos käyttäjä nyt klikkaa sellaista konnektiivia, joka aiheuttaisi uudelleen säännön S soveltamisen tässä solmussa, muistista haetaan näytölle koko tallessa oleva vanha alipuu.

Undo ei ole kuitenkaan aivan näin yksinkertainen, koska käyttäjä voi myös klikata kaavassa syvemmällä olevaa konnektiivia, jolloin kaavaa puretaan auki niin että tämä konnektiivi häviää. Undo voidaan ilmaista toisella tavalla niin, että jos käyttäjä on joskus todistamisen aikana luonut solmulle x alipuun, tämä alipuu näytetään aina silloin jos solmu x olisi jonkin myöhemmin todistamisen aikana tuotettavan alipuun lapsisolmu. Solmut x ja y ovat tässä samoja jos niihin on päästy juuresta soveltamalla täsmälleen samoja sääntöjä samoihin kaavoihin samassa järjestyksessä.

Kalkyylissä ja käyttöliittymässä on myös yhden tason eksplisiittinen undo. Tämä palauttaa puun sellaiseksi kun se oli ennen edellistä konnektiivin klikkausta. Undo ei ole aivan ideaalinen, sillä se muuttaa vain ruudun näkymää, mutta jättää muistiin kaiken minkä edeltävä toiminto tuotti. Tästä ei kuitenkaan ole oikeastaan muuta haittaa kuin käyttäjän pieni hämmennys jos hän ei ole lukenut käyttöohjetta, jossa undon toiminta kuvataan.

Eksplisiittiselle undolle on vastatoiminto redo, joka palauttaa tilanteen sellaiseksi kun se oli ennen undoa. Redo on käytettävissä vain heti undon käyttämisen jälkeen.

6.2.4 Keskitys

Puusta saattaa tulla suurikokoinen ja se ei mahdu kokonaisuudessaan ikkunaan. Kun todistuspuu on jossain vaiheessa, sekventtien sijoittelu alueelle on kiinteä. Käyttäjä voi alueen reunalla olevista vierityspalkeista muuttaa ruudulla näkyvää osaa koko todistuspuun alueesta. Vierityspalkkien käyttö on turvallista, koska näytettävä alue on oleellisesti bittikartta.

Alue voidaan keskittää johonkin sekventtiin. Tämä tarkoittaa aina sitä, että alueen kohdistus asetetaan niin, että sekventin nuoli on todistusalueen keskellä. Alue voidaan keskittää johonkin sekventtiin klikkaamalla sen sekventtinuolta, tai mitä tahansa muuta kohtaa piirretyn sekventin alueella (esim. propositiota tai pilkkua), kunhan se ei ole mikään konnektiivi.

Alueen keskipiste toimii kiintopisteenä myös silloin, kun ruudun kokoa muutetaan (jolloin myös todistusalueen koko muuttuu). Suurennetun tai pienennetyn alueen keskipiste on sama kuin mikä se oli ennen koon muuttamista.

Puun asettelu ruudulla muuttuu vain silloin kun puu itse muuttuu, tai fontin kokoa muutetaan. Kun puuta muutetaan, asetetaan todistusalueen keskitys siihen sekventtiin, johon muutos tapahtui. Jos klikattiin jonkin sekventin konnektiivia, keskitetään alue tähän sekventtiin. Jos painettiin Undo- tai Redo-nappia, keskitetään alue siihen sekventtiin, johon muutos kohdistui. Jos fontin kokoa muutetaan, alue keskitetään juureen.

Alueen keskitys on juuressa silloin kun luodaan uusi todistus tai avataan vanha tiedostosta.

Todistusalueen keskitystä voidaan muuttaa myös navigointi-ikkunasta: ikkunassa klikattu sekventtisolmu asetetaan silloin alueen keskelle.

6.2.5 Todistuksen valmistuminen

Todistuksen valmistuminen kokonaan voidaan ilmaista esimerkiksi värittämällä koko puu jollain toisella värillä tai avaamalla pieni asiasta kertova dialogi. Valmistuminen näkyy kuitenkin myös navigointi-ikkunasta, joten tällainen lisätoiminta ei ole välttämättä tarpeellista. Idean käytettävyyttä voidaan testata vielä toteutusvaiheessa, koska toiminnon toteuttaminen on helppoa.

6.2.6 Väritys

Sekventistä väritetään eri värillä, vaikkapa sinisellä, (tai ympyröidään), se konnektiivi johon sääntöä on sovellettu. Jos käyttäjä on klikannut suluissa sisempänä olevaa konnektiivia, tätä valintaa ei näytetä siinä kaavassa jossa klikkaus on tehty, vaan tässäkin kaavassa uloin konnektiivi on merkitty. Koko kaava johon sääntöä sovellettiin voidaan värittää jollain toisella värillä (eri sävyinen sininen esimerkiksi).

Ne sekventit jotka ilmestyivät ruudulle edellisen klikkauksen jälkeen väritetään jollain erityisellä värillä (esim. punaisella), jolloin käyttäjä näkee heti, mitä klikkaus puussa muutti.

Jos toteutusvaiheessa löydetään helppo tapa (toteutukseltaan ja käytännöllisyydeltään) avata jokin ruudulla näkyvä sekventti omaan ikkunaansa alimmaksi sekventiksi, sellainen toiminto voidaan toteuttaa.

6.3 Navigointipuu

Navigointipuu on omassa ikkunassaan. Jos puun tarvitsema alue on suurempi kuin käytettävissä oleva piirtoalue, ikkunassa on vierityspalkit.

Navigointipuun solmu vastaa yhtä sekventtiä todistusalueessa ja sen nimenä on joko sovelletun säännön nimi, kysymysmerkki, jos sekventtiin ei ole vielä sovellettu mitään sääntöä, tai suuri X-kirjain, jos sekventti on "umpikujaan" johtanut lapsisolmu, eli se ei ole tosi eikä siihen voi soveltaa mitään sääntöä. Solmujen välillä on pieni viiva. Jos solmu on lehti ja se on kalkyylin mukaan tosi, solmun nimenä on sääntö, jonka mukaan se on tosi. Solmu on silloin "suljettu".

Solmut on väritetty. Musta solmu vastaa sekventtiä joka on kesken ja keskellä puuta. Punainen solmu vastaa sekventtiä, joka on kesken ja puun lehti (eli kysymysmerkki), joten käyttäjän pitää vielä jatkaa niitä. Punaisella värillä on merkitty myös umpikujat (eli ruksit), näistä käyttäjän pitää päästä eroon soveltamalla aikaisemmin jotain muuta sääntöä. Vihreällä värillä on merkitty kaikki ne solmut, jotka ovat valmiita, eli kaikki niiden lehdet on suljettu. Siis kun koko puu on vihreä, todistus on valmis.

Solmun klikkaaminen keskittää todistusalueen tähän sekventtiin. Muualla puun alueella klikkaaminen ei tee mitään. Aina kun todistusalue keskitetään johonkin sekventtiin, merkitään esimerkiksi laatikolla navigointipuun se solmu, johon keskitys on tehty. Jos todistusalueen kohdistusta muutetaan vierityspalkeilla, tämä merkki ei enää ole täsmällinen joten joko poistetaan merkki näkyvistä kokonaan tai dynaamisesti lasketaan ruudun keskustaa lähinnä oleva sekventti ja merkitään tämä navigointipuussa, tai keksitään jokin muu tapa merkitä navigointipuusta se kohta jota todistusikkunassa ollaan tarkastelemassa.

Myös navigointipuussa on vierityspalkit, joilla ruudun keskikohtaa voidaan muuttaa. Myös navigointiruutu luultavasti keskitetään aina kaavaa purettaessa samaan solmuun kuin todistusruutu. Syy on sama kuin todistusruudussa: navigointiruudun bittikartta on kiinteä vain tarkasteltaessa kiinteää puuta ja saattaa muuttua villisti, kun puu muuttuu. Periaatteessa keskityksen suhde puun juureen pysyy kiinteänä, joten voi olla että keskitys purettaessa ei sittenkään ole selkeämpi vaihtoehto ja että käyttäjällä säilyy riittävän hyvä kuva siitä, missä kohtaa puuta kulloinkin ollaan. Parempi näistä voidaan valita vasta toteutusvaiheessa, kun käyttöliittymää testataan.

6.4 Uusi todistus -ikkuna

Ylimpänä uusi todistus -ikkunassa on kalkyylin valitseva radiobutton, tässä versiossa siis vain valinnat G3IP ja G3CP.

Sen lisäksi siinä on kentät sekventin syöttöä varten. Tämä voidaan toteuttaa joko yhdellä kentällä (jolloin käyttäjä kirjoittaa kenttään itse sekventtinuolen ja erottelee eri kaavat toisistaan pilkuilla), kahdella kentällä, joiden välissä on sekventtinuoli, tai usealla kentällä, joihin kuhunkin kirjoitetaan yksi kaava ja joista puolet on nuolen toisella ja puolet toisella puolella. Toteutuksessa valitaan parhaiten toimiva ratkaisu.

Lisäksi ikkunassa on alue, jossa näkyvät eri kirjoitusmuodot eri matemaattisille symboleille. Alueen sisältö voisi muuttua kun radiobuttonia vaihdetaan, jos käytetyt merkinnät riippuisivat kalkyylistä, mutta koska nyt toteutettavat kalkyylit käyttävät samaa merkintätapaa, ei muuttumista tehdä.

Käyttäjä voi klikkailla alueella näkyviä symboleja, ja symbolia vastaava sopiva merkkijono (onko se sitten “impl”, “implication” tai jokin muu, päätetään myöhemmin) lisätään silloin siihen kaavaan, jossa tekstikohdistin vilkkuu. Jos tekstikohdistin ei ole missään kaavakentässä, klikkaaminen siirtää kohdistimen kaikista ensimmäiseen kaavaan ja lisää merkkijonon siihen. Alueella on myös muutamia kirjainmerkkejä propositioita varten, joten käyttäjä voi kirjoittaa yksinkertaiset kaavat pelkästään hiirtä käyttäen.

Radiobuttonin vaihtaminen ei myöskään tee mitään tekstikentille, vaan vasta kun Ok-nappulaa on painettu, kenttien sisällöt tarkistetaan ja virheen sattuessa huomautetaan käyttäjää.

Ikkunassa on Help-nappi joka avaa (modaalisen) apudialogin, joka kertoo miten uusi todistus -dialogia käytetään.

Lisäksi ikkunan alalaidassa ovat Ok- ja Cancel-nappulat. Cancel kumoaa todistuksen luonnin, Ok tarkistaa kaavan syntaktisen oikeellisuuden ja avaa uuden todistusikkunan sille. Jos kaava oli virheellinen, huomautetaan käyttäjää (muttei eritellä virheen sijaintia, jos se on teknisesti liian vaikeaa) dialogilla, josta palataan Uusi todistus -ikkunaan.

6.5 Vienti-ikkuna

Vienti-ikkunassa on radiobutton-valinta sille, missä muodossa vienti halutaan tehdä. Tässä versiossa valittavana ovat ASCII- ja LaTeX`proof.sty`-vaihtoehdot.

Seuraavana on tiedoston valinta. Tämä voidaan valita esimerkiksi kirjoittamalla tekstikenttään tiedoston nimi. Vaihtoehtoisesti voidaan valita tiedosto "Browse. . . "-napilla joka avaa standardin tiedostontallennusdialogin, jolloin valitun tiedoston nimi ilmestyy tekstikenttään.

Ikkunassa on kaksi pääosaa, jotka muistuttavat pääikkunan ja navigointi-ikkunan piirtoalueita. Pääikkunaa muistuttava osa toimii esikatseluna, joka näyttää dynaamisesti käyttäjälle sen alueen, jonka vientikomento sillä hetkellä tuottaisi. Kuten pääikkunassa ja navigointi-ikkunassa, koko todistuspuun juuri on alueiden kaikista alin solmu.

Navigointi-ikkunaa muistuttavasta alueesta valitaan vietävä alue. Molempien alueiden vierellä on vierityspalkit. Alueen lähellä, esim. alapuolella, on ohjepainike, joka avaa pienen ohjeruudun siitä, kuinka vientialue valitaan.

Ikkunassa on lisäksi "Export"- ja "Close"-nappulat, joista edellinen vie valitun alueen tiedostoon sekä pyyhkii navigointi-ikkunan valinnat, ja jälkimmäinen sulkee ikkunan.

Vientialue valitaan seuraavasti: Käyttäjä käyttää kahdenlaisia hiiren klikkauksia, esimerkiksi tavallinen klikkaus ja muutettu klikkaus ("mod-klikkaus", voidaan toteuttaa esim. shift-klikkauksena). Tavallinen klikkaus valitsee vientialueen alimman solmun. Shift-klikkaus rajaa aluetta ylhäältä päin siten, että klikatun solmun kaikki lapsisolmut tästä puun haarasta ylöspäin jätetään pois valitusta alueesta. Klikkaus pyyhkii aina kaikki aiemmin valitut mod-klikkaukset.

Shift-klikkaukselle on alueen vieressä yksitasoinen undo-nappi, jolla edellinen mod-klikkaus voidaan perua. Undo-nappi on aktiivinen vain jos edellinen klikkaus oli mod-klikkaus. Kun undo-nappia on kerran painettu ja undo tehty, se disabloituu. Undon tarkoituksena on auttaa käyttäjää silloin, jos se on klikannut monta alisekventtiä mod-klikeillä pois mutta tekee pienen virheen. Ilman undoa koko prosessi pitäisi tehdä alusta, vietävän alueen juuren valinnasta, lähtien uudelleen.

Alueen valintametodia voidaan muuttaa, jos toteutuksen aikana havaitaan vielä parempi ratkaisu. Tähän mennessä valintametodi on muuttunut niin monta kertaa, että ei ole mitään takeita, että edellä esitetty metodi olisi absoluuttisesti paras. Metodin muuttaminen kesken kaiken ei tuottane suuria toteutusongelmia.

Esikatseluruudussa voitaisiin näyttää vientialue mustana tekstinä valkoisella pohjalla, ja viemättä jäävät sekventit harmaalla tekstillä. Alueenvalintaruudussa näkyy valittu vientialue jollain näkyvällä värillä, esimerkiksi vihreällä.

Kun jokin alue on viety onnistuneesti ikkunan export-nappulalla, alueenvalintaruutu palaa tilaan, jossa kaikki solmut puusta on valittu. Esikatseluruutu näyttää tämän myös. Alueenvalintaruudussa ne solmut jotka vietiin muuttuvat kuitenkin "varjostetuiksi", mistä käyttäjä näkee heti, mitä sekventtejä ei tarvitse enää viedä. Varjostus näkyy niin, että normaalisti mustatekstinen solmun nimi näkyy harmaana, tai jos solmu on valittu vietäväksi ja näkyy vihreänä, se näkyy tällöin vaaleamman vihreänä. Kun export-napilla viedään joitain muita alueita, se lisää varjostettujen solmujen määrää. Varjostukset katoavat vain kun sulkee vienti-ikkunan ja avaa sen uudelleen.

6.6 Virhetilanteiden hallinta

Ohjelman toiminnan aikana ei voi tapahtua kovin monenlaisia virheitä. Tässä vaiheessa näyttää siltä että vain kolmenlaisia virheitä voi tulla. Mikäli toteutusvaiheessa löytyy uusia virheitä[2], käsitellään nämä vastaavalla tavalla.

Tiedostojärjestelmän virheet: lataus/tallennus/export

Mikäli tiedostojärjestelmässä tapahtuu jokin virhe ja joko lataus, tallennus tai export epäonnistuu tämän takia, virheestä ilmoitataan käyttäjälle popup- ikkunalla. Muuten pysytään siinä ikkunassa missä oltiin.

Syötetty kaava on vääränmuotoinen

Mikäli kaava minkä käyttäjä ohjelmalle syöttää on vääränmuotoinen, ProofParser huomaa tämän ja heittää poikkeuksen. Ohjelma ilmoittaa käyttäjälle mikä kaavan osa on virheellinen esim. alleviivaamalla virheellisen osan punaisella värillä. Pysytään uusi todistusikkunassa.

Ladattava tiedosto on vääränlainen

Jos ladattava tiedosto on vääränlainen sen lataus ei onnistu. Ohjelma havaitsee virheen ja heittää poikkeuksen. Käyttäjälle tämä näkyy popup- ikkunana joka ilmoittaa tiedoston olevan vääränlainen. Pysytään lataus-ikkunassa.

Kriittiset virheet kuten pinon tai kasan loppuminen

Mikäli tulee jokin kriittinen virhe, käsittelyssä ollut todistus pyritään tallentamaan jos vain mitenkään mahdollista. Tämä ei mitenkään välttämättä enää onnistu, joten tallentamattomat osat saattavat kadota.

7 Käyttöliittymän luokkarakenne

Ohjelma käyttää käyttöliittymäkomponentteja Javan AWT-kirjastosta. Käyttöliittymä on jaettu osiin periyttämällä luokista uusia aliluokkia, johon sisällytetään tämän luokan toiminnot toteuttavat rutiinit. Samat asiat olisi voitu tehdä myös ilman periytystä, mutta käyttöliittymäosa jakautuu loogisesti eri tavoin toimiviin osiin, jotka voidaan toteuttaa itsenäisesti ja jotka riippuvat periytetystä luokasta.

Vaikka tietorakennepuoli sallii monimutkaisten kalkyylien luomisen, käyttöliittymään on sisällytetty rajoituksia. Rajoitukset voi tiivistää siihen, että käyttöliittymä toimii G3IP-

[2] Esimerkiksi jokin käyttöliittymäkomponentti heittää poikkeuksen, jonka olemassaoloa ei tiedetty suunnitteluvaiheessa.

ja G3CP-kalkyyleillä. Tämä tarkoittaa oikeastaan sitä, että sekventillä on korkeintaan kaksi lasta.

Ohjelmassa on kaksi *Frame*-tyyppistä ikkunaa, pääikkuna ja navigointi-ikkuna, sekä useampia *Dialog*-tyyppisiä ikkunoita: vienti-ikkuna, lataus-, tallennus- ja uusi-ikkunat. Lataus- ja tallennusikkunoiden toiminta käsitellään pääikkunassa, muut on periytetty omiksi luokikseen. Navigointi-ikkunan (harvat) muut toiminnot käsitellään pääikkunassa, paitsi navigointi-ikkunan puunpiirtoalueen, jolla on oma käsittelijänsä.

Kaksi luokkaa perii *Canvas*-piirtoalueluokan: pääikkunan todistuksenpiirtoalue (todistusalue) sekä navigointi-ikkunan puunpiirtoalue (navigointialue).

7.1 Käynnistysluokka ProEdit

Ohjelma käynnistetään ajamalla tämän luokan main-metodi. Luokka sisältää useamman todistusikkunan hallintakomponentit.

```
Vector mainWindows // (of MainWindow), != null
```

Lista ohjelman käyttämistä todistusikkunoista, sisältää aina vähintään yhden ikkunan tai muuten ohjelma lopetetaan.

```
int getWindowId(MainWindow)
void checkTitles()
```

Edellinen metodi käy läpi kaikki Vector-olion sisältämät MainWindowit ja selvittää, onko siellä ikkunaa, jolla on sama tiedostonimi kuin kutsuvalla ikkunalla. Jos näin on, palautetaan järjestysnumero, kuinka mones tämä ikkuna oli (Vector-järjestyksessä) niistä, joilla sama tiedostonimi oli.

Jälkimmäinen metodi käy läpi kaikki ohjelman ikkunat ja kutsuu jokaisen setTitle-metodia, käyttäen ikkunan id-numeroa. Ikkunan id-numero on 0, jos numerointia ei tarvita, järjestysluku alkaen luvusta 1, jos se tarvitaan. MainWindowin setTitle-metodi muuttaa tämän perusteella otsikkopalkkinsa.

Metodia tarvitaan sen takia, että jos ohjelmassa on auki kaksi ikkunaa, joilla on sama tiedostonimi, ikkunan otsikkotietoihin tulee myös juokseva numerointi, jolla todistusikkuna ja navigointi-ikkuna voidaan tunnistaa samaan ikkunapariin kuuluvaksi. Kun MainWindow-olio nimeää itseään, se asettaa ensin tiedostonimen itselleen ja kutsuu checkTitles-metodia. Metodi kutsuu setTitle-metodia, joka lopulta asettaa ikkunan otsikon oikeaksi. Kaikkien muidenkin ohjelman ikkunoiden otsikot muutetaan sen varalta, että jokin niistä tarvitsee numerointia uuden ikkunan luonnin jälkeen.

```
public static void main(String args)
```

Käynnistysmetodi, ajaa newWindow-metodin.

```
void newWindow()
```

Avaa tyhjän ikkunan, jonka tiedostonimi on null. Tiedostonimeksi tulee "Untitled" vasta kun luodaan uusi todistus.

```
void newWindow(Calculus, String name)
```

Avaa ikkunan, ja siihen Calculus-olion sisältämän todistuksen. Tiedostonimeksi name. Kun ikkuna on luotu, se kutsuu checkTitles-metodia.

```
void closeWindow(MainWindow)
```

Tätä metodia kutsutaan MainWindow-olion closeWindow-metodista. Poistaa viitteen ikkunaan taulukosta. Tässä tapauksessa itse ProEdit-olio kutsuu lopuksi checkTitles-metodia, jotta ikkunoiden numerointi muuttuu ajantasaiseksi, kun edellisessä metodia kutsui ikkuna. Lopuksi jos ikkunataulukko on tyhjä, ohjelma lopetetaan.

```
void exit()
```

Sulkee kaikki ikkunat ja ohjelman. Tekee sen kutsumalla kunkin MainWindow-olion okToClose()-metodia. Tämä avaa jokaiseen ohjelman pääikkunaan varmistusdialogin, jos tiedostoa ei ollut tallennettu, ja sulkee kaikki ikkunat, joissa tallennus oli tehty. Varmistusdialogeihin myönteisesti vastaaminen aiheuttaa lopulta sen että kutsutaan closeWindow-metodia, ja kun viimeinenkin ikkuna on näin suljettu, ohjelma loppuu. Jos mihin tahansa dialogiin vastattiin kielteisesti, tämä ikkuna jää vielä päälle, mutta muut suljetaan.

7.2 Pääikkuna MainWindow

Pääikkunaluokka toteuttaa pääikkunan ja navigointi-ikkunan tapahtumankäsittelyn (poislukien Canvas-osat) ja rutiinit muiden ikkunoiden näyttämiseen, jolloin toiminta ohjataan näille ikkunoille. Pääikkunassa on tallennus- ja latausrutiinit ja navigointi-ikkunan käsittely.

```
ProEd program       // != null
Dialog saveDialog
Dialog loadDialog
PrintDialog printDialog
NewDialog newDialog
Frame naviWindow    // != null
NaviArea naviArea   // != null, saadaan proofArea-oliolta.
ProofArea proofArea // != null
Calculus calculus
boolean naviState
```

Luokan sisältämistä kentistä viisi on epätyhjiä. Näistä ProEd ja Calculus saadaan konstruktorikutsussa, naviWindow- ja proofArea-oliot ikkuna luo itse, ja naviArea-olion se hakee proofArea-oliolta sen luonnin jälkeen. Dialogioliot luodaan tarvittaessa ja poistetaan käytön jälkeen.

`boolean okToClose()`

Jos ikkunassa oleva todistus on tallentamatta, okToClose()-metodi avaa varmistusdialogin joka kysyy, voidaanko ikkuna sulkea. “Tiedosto Xyz on tallentamatta. Voiko todistuksen sulkea?”, Ok ja Cancel-napit tarjotaan. Ok-napin painaminen ja tilanne, että todistus oli tallennettu aiheuttaa MainWindow-olion closeWindow-metodin kutsumisen. Cancel-napin painaminen ei tee mitään.

`void closeWindow()`

Sulkee ikkunan mitään kyselemättä. Piilottaa sen näkyvistä ja kutsuu ProEdit-olion closeWindow-metodia.

`MainWindow(ProEd, Calculus)`

Konstruktori, ProEd-kentän arvo on erisuuri kuin null. Calculus voi olla null, tällöin ikkunassa ei ole yhtään todistusta auki.

```
void openNewDialog()
void new(Calculus)
```

Edellinen metodi luo ja avaa uusi todistus -ikkunan. Jälkimmäinen metodi avaa uuden todistuksen. Jos tässä ikkunassa calculus-olio oli null, uusi calculus laitetaan tähän ikkunaan. Muuten kutsutaan ProEdit-luokan newWindow-metodia ja käytetään tiedostonimenä “Untitled”.

```
void openSaveDialog()
void save(String filename)
```

Edellinen metodi luo ja avaa tallennusikkunan. Jälkimmäinen metodi tallentaa tiedoston annetulla tiedostonimellä. Save-metodi luo tiedostovirran ja FileProofParser-olion ja antaa ne Calculus-olion save-metodille. Jos tallennus epäonnistui, näytetään virhedialogi ja pidetään tallennusikkuna avoinna. Muuten ikkuna suljetaan ja poistetaan muistista.

```
void openLoadDialog()
void load(String filename)
```

Edellinen luo ja avaa latausikkunan. Jälkimmäinen avaa tiedoston: Avaa tietovirran ja luo FileProofParser-olion, jotka se antaa Calculus-luokan konstruktorille. Jos lataus epäonnistui, näytetään käyttäjälle virhedialogi ja pidetään tiedostonavausikkuna auki. Muuten ikkuna suljetaan ja poistetaan muistista. Jos tämän ikkunan Calculus-olio oli null, siihen asetetaan tämä olio. Muuten kutsutaan ProEdit-olion newWindow-luokkaa, jolle annetaan luoto Calculus ja tiedostonimi.

`void openExportDialog()`

Luodaan ja avataan vientidialogi.

`void naviState(boolean toggle, boolean close)`

Muuttaa navi-ikkunan tilaa. Jos toggle on tosi, vaihdetaan tila toiseksi kuin se oli. Close-arvoa ei silloin käytetä. Jos toggle on false ja close on tosi, ikkuna suljetaan ja jos close on epätosi, se avataan.

`public void actionPerformed(Event)`

Ikkunna käsittelee seuraavat tapahtumat:

- Menu-New: kutsu openNewDialog().
- Menu-Load: kutsu openLoadDialog().
- Menu-Save: Jos tiedostolla oli jo nimi muu kuin "Untitled", kutsu save(). Muuten kutsutaan openSaveDialog().
- Menu-Save as: Kutsu openSaveDialog().
- Menu-Export: kutsu openExportDialog().
- Menu-Close, WindowClosed: kutsu ProEdit-olion closeWindow()-metodia.
- Menu-Exit: kutsu ProEdit-olion exit().
- Menu-Undo: kutsu undo().
- Menu-Redo: kutsu redo().
- Menu-Navi Toggle: kutsu naviState(true, false).
- Menu-Large Font: kutsu fontToggle().
- Vastaavat toiminnot ikkunassa oleville nappuloille
- Navi-ikkunan WindowClosed: kutsu naviState(false, true)
- Load- ja Save-dialogin tapahtumankäsittely.
- Virhedialogien tapahtumankäsittely
- Erityisesti okToClose-dialogin Ok-nappulan painaminen aiheuttaa closeWindow()-metodikutsun.

7.3 NewDialog

Luokka sisältää Uusi todistus -dialogin käyttämät toiminnot, ja se luo Ok-nappulaa painettaessa uuden kalkyyli-olion käyttäjän syöttämien tietojen perusteella. Tämä olio annetaan pääikkunalle.

Luokan vastuulla on pelkän käyttöliittymällisen olemassaolonsa ja toiminnallisuutensa lisäksi formuloida käyttäjän syöttämä kaava sellaiseen muotoon (ehkä esim. katenoimalla merkkijonoja ja luomalla näistä merkkijonovirran), jonka ProofParser-olio hyväksyy, ja lisäksi luoda Calculus-olio, jolle ProofParserilta saatu Proof-olio annetaan juureksi.

Ikkunassa on radiobuttoneina kaikki ohjelman tukemat kalkyylit. Tämä lista saadaan Calculus-luokan staattiselta metodilta.

`MainWindow mainWindow // != null`

Viittaus ikkunaa kutsuneeseen pääikkunaan, jolle lopetettaessa annetaan luotu kalkyyli.

`NewDialog(MainWindow)`

Konstruktori, MainWindow != null, tai muuten heitetään poikkeus.

`void parse()`

Lukee dialogin kenttien tiedot ja tekee niiden perusteella uuden Calculus-olion. Jos poikkeusta ei heitetty, kaikki sujui hienosti ja kutsutaan MainWindow-olion new-metodia. Sen jälkeen suljetaan ikkuna. Jos jokin virhe tapahtui, avataan dialogi, jossa kerrotaan tapahtuneesta.

`public void actionPerformed(Event)`

Käsiteltävät tapahtumat:

- Ok-nappulaa painettu. Kutsutaan parse-metodia.
- Cancel-nappulaa painettu, WindowClosed. Ikkuna suljetaan.
- Virhedialogien tapahtumankäsittely.

7.4 PrintDialog

Vienti-ikkunassa on viennin toteuttavat rutiinit. Ikkuna sisältää myös molemmat piirtoa-lueluokat, navigointialue toimii tässä tapauksessa vientialueen valintaikkunana ja todistusalue esikatseluruutuna.

Ikkunassa on lisäksi muita kenttiä ja nappuloita, jotka on kuvattu käyttöliittymäkappaleessa.

```
MainWindow mainWindow  // != null
Calculus proof         // != null
ProofArea previewArea  // != null
NaviArea selectionArea // != null
```

MainWindow-viittausta ei välttämättä tulla tarvitsemaan. Calculus viittaa käytettävään kalkyyliin ja saadaan parametrina. Tämän perusteella luodaan konstruktorissa ProofArea ja NaviArea.

```
PrintDialog(MainWindow, Calculus)
```

Konstruktori, Calculus != null.

```
void export()
```

Lukee tiedostokentästä tiedostonimen. Avaa virran tiedostoon ja kutsuu Calculus-olion writeToStream-metodia. Jos jokin virhe tapahtui, avataan asiasta kertova virhedialogi. Jos kaikki oli kunnossa, pyyhitään tiedostonimi kentästä ja kutsutaan Proof-olion clearExportRange-metodia.

```
public void actionPerformed(Event)
```

Käsiteltävät tapahtuat:

- Export-nappulaa painettu. Kutsu export-metodia
- Close-nappulaa painettu, WindowClosed. Sulkee ikkunan.
- Virhedialogien tapahtumankäsittely.

7.5 Käyttöliittymäelementit tietorakenteessa

Tietorakenneolioissa Proof, Formula, ParsedFormula ja Connective on käyttöliittymäspesifisiä osia. Tällä vältetään ylimääräinen kerros käyttöliittymän ja tietorakenteen välillä, mutta ongelmana on komponenttien jakokohdan hämärtyminen. Tätä yritetään välttää sillä, että käyttöliittymäspesifiset osat määritellään komponenteissa omassa osassaan koodia tai jopa omassa oliossaan tietorakenneolion sisällä.

Pitää siis huomata, että tietorakenneosa ja kalkyyli ei tarvitse mitään käyttöliittymäspesifisiä asioita toimiakseen.

Seuraavien metodien toiminta on esitelty myös käyttöliittymän sekvenssikaaviossa, joiden avulla olioiden yhteistoiminta valottuu ehkä paremmin.

Käyttöliittymä tarvitsee kahta eri osatoiminnallisuutta tietorakenneluokilta. Ensinnäkin niiden on pystyttävä määrittelemään omat koordinaattinsa ruudulla ja piirtämään itsensä ruudulle. Toisekseen, kun ne tietävät koordinaattinsa, niiden on pystyttävä löytämään se konnektiivi, jota käyttäjä on ruudulla klikannut.

Metodien lisäksi luokissa on muutamia kenttiä, joita tarvitaan vienti-ikkunassa ja erilaisia värityksiä varten.

7.5.1 Paint

Connective-luokalle määritellään metodit

```
Connective:
void paint(Graphics g, int x, int y, int size)
int getWidth(int size)
```

Edellinen metodi piirtää kyseisen konnektiivin Graphics-oliota käyttäen kohtaan, jonka vasemman yläreunan koordinaatit määrittelee (x, y) (ruutukoordinaatteina). Size-parametri määrittelee sen, minkä kokoisella fontilla konnektiivi piirretään (nykyään vain joko 0 tai 1 pienelle tai isolle fontille). Paint-metodin pitää pystyä piirtämään kummankin kokoinen konnektiivi. Jälkimmäinen metodi palauttaa pelkästään sen symbolin koon jonka paint-metodi piirtäisi. Kuvassa itse symbolin vasemmalla ja oikealla puolella on sopivasti tyhjää tilaa, muutamia pikseleitä, että se sopii kauniisti vaihtuvanlevyisen fontin sekaan.

Piirretyn konnektiivin korkeus on aina vakio yhtä fonttikokoa kohden, ja se määritellään toteutusvaiheessa kun valitaan todistusalueen fontin pistekoko. Piirtorutiini määritellään konnektiivin yläluokassa, mutta kussakin konnektiivissa on erikseen alustuskomennot jotka lataavat oikean bittikartan konnektiivia varten.

ParsedFormula-luokka saa samannäköiset metodit:

```
ParsedFormula:
void paint(Graphics g, int x, int y, int size, boolean isPrincipal)
int getWidth(int size)
```

Metodit piirtävät ja laskevat leveyden, jonka ParsedFormula tuottaisi siitä solmusta puuta alaspäin, jota kutsutaan. Rekursiivinen piirtorutiini on kuvattu sekvenssikaavioissa. isPrincipal-arvo on tosi prinsipaalisen kaavan prinsipaalille konnektiiville, kaikille muille se on false. Tämän perusteella konnektiivi piirretään eri tavalla: se piirretään sinisellä ja ympärille piirretään laatikko.

Formula-luokan metodit ovat samannäköiset ja periaatteessa vain ohjaavat kutsun ParsedFormula-oliolleen.

```
Formula:
void paint(Graphics g, int x, int y, int size, boolean isPrincipal)
int getWidth(int size)
```

Paint maalaa kaavan kohtaan (x, y) (ruutukoortinaatteina). GetWidth-metodi palauttaa sen leveyden, jolle paint-metodin piirtämä alue ulottuisi jos se piirrettäisiin.

Proof-luokka sisältää koko sekventin (eli vasemman ja oikean monijoukon) jotka sen on myös piirrettävä. Myös tämä luokka toteuttaa metodit

```
Proof:
int x,y
void paintProof(Graphics g, int centerX, int centerY, int width, int height, int size)
int getWidth(int size)
void paint(Graphics g, int centerX, int centerY, int width, int height, int size)
```

Metodi paintProof ei saa parametrinaan piirtokohtaa x ja y, vaan ne on aiemmin sijoitettu muuttujiin x ja y. Nämä koordinaatit on tallennettu virtuaalitason mukaan (ks. ProofArea-kappale) asettelun yhteydessä. Sen sijaan metodi saa ne koordinaatit, jonka kohta pitää piirtää alueen keskikohtaan. Lisäksi se saa tietää alueen koon (josta se voi keskikohdan laskea).

PaintProof-metodia ei ole mainittu sekvenssikaaviossa, mutta se on oikeastaan vain ylimääräinen väliaskel paint-metodia suoritettaessa. Näin piirtorutiinit saadaan kuitenkin jaettua selkeämmiksi kokonaisuuksiksi.

PaintProof laskee virtuaalikoordinaattien, keskipisteen koordinaattien ja alueen koon perusteella oikeat piirtokoordinaatit. Jos huomataan, että oikeastaan mikään metodin piirtämä asia ei tule näkymään ruudulla näkyvällä alueella, piirto voidaan jättää tekemättä.

Sen jälkeen metodi piirtää proof-olion sisältämän sekventin tähän kohtaan, Graphics-oliota käyttäen, fonttikoolla size. Jokainen vasemman monijoukon kaava käydään yksitellen läpi ja piirretään paint-metodillaan, siirretään piirtokohtaa oikealle kaavan leveyden verran, piirretään normaalilla fontilla pilkku, siirrytään sen leveyden verran oikealle ja piirretään seuraava kaava. Vasemman monijoukon läpikäynnin jälkeen piirretään sekventtinuoli, jonka jälkeen piirto toistetaan oikealle monijoukolle. Muut kaavat saavat isPrincipal-booleanarvokseen false, paitsi se joka vastaa Proof-olion visible- ja visibleMultiset-arvoja, jolle annetaan arvoksi true.

PaintProof-metodi ei piirrä sekventin yläpuolelle viivaa, eikä käytettyä sääntöä, vaan piirtää pelkästään yhden sekventin. Itse metodi paint tekee sen. Se käy rekursiivisesti läpi koko puun. Sääntö piirretään käyttäen samoja merkkejä kuin atomikaavojen piirtämisessä (aakkosista ”L” ja ”R” sekä konnektiivimerkit), mutta lisäksi käytetään pieniä numeromerkkejä, joilla piirretään säännön numero konnektiivin alaindeksinä, jos konnektiiviin oli useampia sääntöjä.

GetWidth-metodi palauttaa sen leveyden, jolle paint-metodi piirtäisi sekventin.

Proof-olion x ja y-koordinaatit pitää asetella ennen piirtoa. Asettelu pitää tehdä vain silloin, kun Proof-puun sisältö on muuttunut. Tässä x-koordinaatti on sekventin keskipisteen x-koordinaatti, y on sen yläreunan koordinaatti. Asettelua varten on kentät ja metodit

```
Proof:
int x, y, totalWidth
int calculateTotalWidth()
int arrange(int center, int direction, int height)
```

Asettelu tapahtuu kahdessa vaiheessa. Ensimmäisessä vaiheessa lasketaan rekursiivisesti koko alipuun viemä leveys. Seuraavassa vaiheessa asetetaan näiden arvojen perusteella kaavoille koordinaatit.

Ensimmäinen vaihe (calculateTotalWidth) toimii seuraavasti: Kun ollaan jossain puun solmussa, kysytään rekursiivisesti sen alipuilta (0-2 kappaletta) alipuiden leveys. Jos alipuita oli 0, asetetaan kokonaisleveydeksi tämän sekventin getWidth-leveys plus kiinteä tila, johon tullaan piirtämään käytetty sääntö.

Jos alipuita oli 1, asetetaan leveydeksi suurempi näistä plus säännölle varattu kiinteä tila. Jos alipuita oli kaksi, leveydeksi asetetaan joko tämän sekventin leveys, jos se oli suurempi, tai alipuiden leveyksien summa plus jokin sopiva pikseliarvo, jolla puut erotetaan toisistaan (5-20 tms). Tähän lisätään lisäksi säännölle varattu tila.

Tämä laskettu leveys palautetaan metodikutsun arvona rekursion edelliselle askelmalle.

Kun leveys on selvitetty, voidaan itse asettelu tehdä seuraavasti: Kun asetellaan jotain sekventtiä, se on joko alapuolella olevan sekventtinsä ainut, vasemmanpuoleinen, tai oikeanpuoleinen lapsisekventti. Tämä selviää arvosta direction, joko on em. järjestyksessä joko 0, -1 tai 1. Parametrina saadaan myös alapuolella olleen sekventin keskikohta. Tämän sekventin asettelu on siis seuraava: jos direction on 0, sekventin x-koordinaatti on sama kuin alapuolella olevan, eli center. Jos direction on -1, x-koordinaatiksi tulee center miinus puolet sekventin totalWidth-arvosta miinus puolet siitä pikselimatkasta, joka on määritelty kahden alipuun väliseksi eroksi. Jos direction on 1, koordinaatti lasketaan samoin kuin edellä, paitsi vähennyksen sijasta ynnätään.

Tämän jälkeen kutsutaan rekursiivisesti lapsisolmuja käyttäen tätä koordinaattia center-arvona, ja asettamalla direction vastaamaan lapsisolmujen sijoittumista.

Myös NaviArea säilyttää jotain tietoja puunsa tilasta ja esilasketuista asetteluarvoista suoraan Proof-olioissa. Kentät ja metodit ovat lähes vastaavat kuin ProofArealle, eli

```
Proof:
int naviX, naviY, naviTotalWidth
void naviPaint(Graphics g, int centerX, int centerY, int width, int height, int size)
```

```
int naviGetWidth(int size)
int naviCalculateTotalWidth()
int naviArrange(int center, int direction, int height)
```

Niiden toiminta on myös melkein samanlaista. Itse asiassa nämä metodit voitaisiin toteuttaa samoissa metodeissa kuin ProofArea-toimintakin, ja erottaa toiminta boolean-arvolla. Kumpi tehdään päätetään toteutuksen aikana.

7.5.2 MouseClicked

Kun käyttäjä on klikannut hiirtä ruudulla, pitää voida löytää se konnektiivi, jota klikattiin, ja lisäksi laskemaan tarvittava apply-jono, jonka avulla konnektiivi saadaan purettua auki. Vastuu on jaettu eri luokille.

Metodien käyttämät koordinaatit lasketaan paint-toiminnan yhteydessä ja ovat ruutukoordinaatteina.

Proof-luokassa on konnektiivin etsimistä varten metodi

```
Proof:
Proof findProof(int x, int y)
```

Proof-luokan metodi palauttaa itsensä, jos x ja y osuivat sen itsensä kohdalle. Muuten se jatkaa etsintää rekursiivisesti vasempaan ja oikeaan lapseensa ja palauttaa rekursion antaman vastauksen. X- ja y-koordinaateista voidaan päätellä rekursiohaun suunta, joten molempia lapsia ei tarvitse testata.

Käyttöliittymä kutsuu todistuspuun juuren tätä metodia, jolloin se saa vastauksena halutun sekventin. Sekventistä pitää löytää lisäksi oikea kaava, jota varten on Proof-luokassa myös metodi

```
Proof:
Formula findFormula(int x, int y)
```

Myös tämä metodi on Proof-luokassa ja hakee luokan sisältämistä monijoukoista sen kaavan, johon x-koordinaatti osuu (y-koordinaattia ei oikeastaan käytetä). Haku voidaan tehdä pitämällä muistissa paintin aikana kahta valmiiksilaskettua taulukkoa kaavojen leveyksistä, eli taulukon ensimmäiseen paikkaan on tallennettu vasemman monijoukon ensimmäisen kaavan leveys, seuraavaan paikkaan toisen ja ensimmäisen kaavan leveyksien summa jne. Molemmille monijoukoille on oma taulukkonsa.

Metodi voi palauttaa joko suoraan Formula-olion, tai parin (monijoukko, kaavanumero) joiden perusteella kaava haetaan Proof-oliolta (kuten sekvenssikaaviossa on kuvattu).

```
Formula:
void findConnective(int x, int y, ArrayList)
ParsedFormula:
void findConnective(int x, int y, ArrayList)
```

Formula-luokalla ja ParsedFormula-luokalla on samannimiset metodit findConnective. Formula-olion metodi ohjaa metodikutsun ParsedFormula-oliolleen. Tämän metodi kasaa ArrayList-olioon jonon kokonaislukuja, jotka kertovat apply-metodille, kummalla puolen jonkin tason konnektiivia klikattu konnektiivi oli, jos se oli syvemmällä puussa. Pseudokoodi on esitetty sekvenssikaavion yhteydessä.

ArrayList-olion perusteella voidaan kutsua apply-metodia riittävän monta kertaa ja oikealla tavalla.

7.5.3 Muut kentät

Sen lisäksi Proof-oliolla on ainakin seuraavat kentät:

```
int naviState
int exportState
int newState
```

Näistä edellinen on navi-ikkunaan liittyvää tilainformaatiota, jonka perusteella valitaan väritys. Kun puun kokoja lasketaan (calculateTotalWidth, selvitetään myös, onko kaava valmis lehti, umpikujalehti vai keskeneräinen lehti. Valmiille lehdelle annetaan arvo 1, keskeneräiselle 2 ja umpikujalle 3. Normaali solmu on 0. Jos solmun kaikki lapset ovat 1, myös tämä solmu saa arvokseen 1. Muuten arvoksi tulee 0.

ExportState-arvoja muutetaan PrintDialog-toiminnassa. Arvo on välillä 0-3. Arvoja 0-1 ei tulosteta, arvot 2-3 tulostetaan. Arvot 1 ja 3 ovat lisäksi "valaistuja" merkiksi siitä, että ne on jo tulostettu. Kun käyttäjä klikkaa puuta PrintDialogissa, näitä arvoja muutetaan.

Viimeinen kenttä liittyy ProofAreaan ja sitä käytetään ilmoittamaan, että solmu oli uusi, verrattuna ruudun edelliseen staattiseen tilanteeseen. Tämä muuttaa piirrettävän solmun väritystä. Uusi-merkinnät pyyhitään ennen applyn kutsumista, joten vain ne solmut, jotka viimeisin apply-jono loi, saavat merkinnän.

7.6 ProofArea

Asettelee ja piirtää todistukset ruudulle.

ProofArea ja NaviArea ovat tiiviisti toistensa kanssa yhteistyössä (niiden välillä on 1-1 -suhde), joten ProofAreaa luotaessa luodaan sille myös NaviArea. NaviAreaa ei luoda missään muualla (luomista ei pystytä kuitenkaan pakolla estämään). NaviArea-olio pyydetään tältä oliolta ja sijoitellaan sitten sopivaan paikkaan käyttöliittymässä.

ProofArea toimii kahdessa eri tilassa riippuen siitä, onko se pääikkunassa vai PrintDialogissa. Edellinen tila on normaalitila, jälkimmäinen pelkkä esikatselutila, jossa toiminta on muuten samanlaista, mutta seuraavat toivat eri tavalla: Hiiren klikkaus vain keskittää, ei käytä applyä. Solmuilla ei ole tavanomaista väritystä, vaan jos solmun printState sanoo, että solmu on vietävänä (tulostettavana), se maalataan paremmin näkyvällä värillä ja jos ei, haaleammalla värillä. Voidaan ehkä tehdä myös niin, että ei maalata ollenkaan solmuja, joita ei viedä. Tällöin pitää etsiä myös puusta juurisolmu, joka on ylin vietävä solmu, jolloin voidaan puu keskittää tähän, jos aiempi keskitys on valittu pois vientialueesta.

Sekventtiolioille ProofArea-alueella käytetään virtuaalisia koordinaatteja. Kun sekventtejä asetellaan, käytetään näitä. Puun alin sekventti on sijoiteltu niin että sen yläreuna ja vaakasuuntainen keskipiste on koordinaateissa $(0,0)$, ja muut puun solmut suhteessa tähän. Kasvusuunta on samoin kuin normaalilla piirtoalueella eli koordinaatit kasvavat oikealle ja alas. Jokainen sekventti on siis sijoiteltu "negatiiviselle" puolelle, koska puu kasvaa juuresta ylöspäin.

Jokaiselle Proof-oliolle tallennetaan asetteluvaiheessa sen sijoittelukoordinaatit. Koordinaatit kertovat sekventistä vaakasuunnassa keskipisteen ja pystysuunnassa yläreunan paikan virtuaalitasossa. Proof-oliolle lasketaan ja tallennetaan asetteluvaiheessa sen vaakasuuntainen leveys. (ks. seuraava kappale)

Kun sekventit piirretään, koordinaatteja siirretään sen perusteella, mihin virtuaalitason kohtaan centerX ja centerY. Nämä kertovat sen, mikä piste on alueen keskellä.

`Calculus calculus`

Käytetty calculus. Jos se on null, mikään metodi ei tee mitään (tapahtuu silloin kun ohjelma on juuri avattu).

`Proof root`

Viittaus puun juureen. ProofArea käyttää Proof-puuta suoraan. Sama kuin calculus.root.

`int centerX, centerY`

Piirtoalueen keskipisteessä olevan pisteen koordinaatit puun virtuaalisen sijoittelun suhteen, jossa puun juuri on koordinaateissa 0,0.

`Proof centerProof`

Piirtoalueen keskipisteessä oleva kaava. Jos != null, centerXY-tiedot lasketaan tämän avulla.

`NaviArea naviArea`

Vastaava naviArea, kutsutaan applyn jälkeen tämän paint-metodia jotta sen ruutu päivittyy. Luodaan konstruktorissa, joten sen jälkeen aina != null.

`public void mouseMotionListener(MouseEvent)`

Kutsutaan aina hiirtä liikutettaessa. Voidaan muuttaa hiiren alla olevan tekstin väriä, jos siltä tuntuu. Luultavasti tätä tapahtumankäsittelijää ei tarvita.

`public void mouseListener(MouseEvent)`

Kutsutaan hiirtä klikatessa. Selvitetään paikka, siihen piirretty Proof-olio, olion kaava jota klikattiin, ja viimeisenä kaavassa oleva konnektiivi. Näiden perusteella kutsutaan calculus-olion apply-metodia niin monta kertaa kuin on tarpeen. Kuvattu myös sekvenssikaaviossa.

Jos hiirtä klikattiin minkä tahansa kaavan kohdalla, kutsutaan myös center-metodia, jolla kaava keskitetään ruudun keskelle.

`public void paint(Graphics g)`

Asettelee Proof-oliot alueelle puun muotoon ja piirtää sen. Kuvattu seuraavassa kappaleessa.

`void center(int x, int y)`

Keskittää alueen annettuihin koordinaatteihin.

`void center(Proof)`

Keskittää alueen sekventtiin, eli kutsuu sekventin x- ja y-koordinaateilla center(int,int)-metodia.

`void setPreviewMode(boolean)`

Asettaa preview-moodin päälle. Tällöin näytetään vain osa puusta, eikä sallita muutoksia puuhun. Käytetään vientidialogissa.

7.7 NaviArea

Asettelee ja piirtää navigointipuun ruudulle. Suurelta osin toimii kuten ProofArea-luokka. Paint-sekvenssi toimii muuten samankaltaisesti, mutta nyt ei lasketa kaavojen leveyksiä, vaan varataan vain kiinteänkokoinen tila säännön näyttämistä varten. Maalattaessa maalataan lisäksi viivat solmujen välille.

Myös NaviArea käyttää asettelussa ja keskityksen määrittelemiseen virtuaalikoordinaatteja, jolloin puun juuri on koordinaateissa $(0, 0)$. NaviArean keskitys toimii täsmälleen kuten ProofAreassa. Vasta toteutusvaiheessa päätetään, keskitetäänkö NaviArea aina kaavaa purettaessa siihen solmuun, jota purettiin (eli samaan kuin ProofAreakin), vai pidetäänkö keskityksen virtuaalikoordinaatit tällöin ennallaan. Jompi kumpi näistä saattaa aiheuttaa käyttäjälle hämmennystä, oikealla todistuksella testatessa nähdään, kumpi. Toteutuksen vaihtaminen on kuitenkin helppoa.

NaviArea, kuten ProofAreakin, toimii kahdessa eri tilassa riippuen siitä onko se pääikkunassa vai PrintDialogissa. Tila valitaan setExportMode-metodilla.

Keskitykseen liittyviä kenttiä ja metodeja sekä setExportMode-metodia ei ole luokkakaaviossa.

```
ProofArea proofArea // != null
Proof root // == proofArea.root
```

Viittaus käytettyyn proofAreaan ja puun juureen.

`int centerX, centerY`

NaviArean keskityksen koordinaatit virtuaalikoordinaatteina.

`NaviArea(ProofArea)`

Konstruktori, ProofArea != null

`public void mouseMotionListener(MouseEvent)`

Kutsutaan aina hiirtä liikutettaessa. Voidaan muuttaa hiiren alla olevan symbolin väriä. Todennäköisesti näin ei tehdä.

`public void mouseListener(MouseEvent)`

Kutsutaan hiirtä klikatessa. Navi-moodissa selvitetään hiiren paikka, siihen piirretty Proof-olio ja kutsutaan proofArean center-metodia. Export-moodissa merkitään vientialue.

`public void paint(Graphics g)`

Asettelee Proof-oliot puuksi, selvittää värityksen (riippuen moodista) ja piirtää oliot.

`void center(int x, int y)`

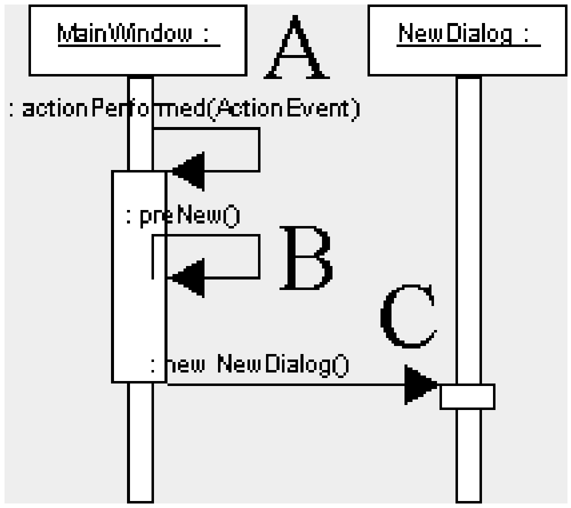
Keskittää alueen annettuihin koordinaatteihin.

`void center(Proof)`

Keskittää alueen sekventtisolmuun, eli kutsuu solmun x- ja y-koordinaateilla center(int,int)-metodia.

`void setExportMode(boolean)`

Asettaa PrintDialogissa käytetyn export-moodin päälle. Tällöin väritys ja puun toiminta muuttuu.



Kuva 6: Sekvenssikaavio: newDialog

8 Käyttöliittymän toiminta

Käyttöliittymäluokkien toimintaa käydään läpi sekvenssikaavioiden avulla seuraavassa.

Kaavioissa käytetään selkeyden vuoksi arvoja asettavia ja palauttavia metodeja (aksessorit) vaikka niitä ei oikeasti toteutetakaan. Osaan metodeista on kirjoitettu myös paluuarvo, jos se ei tule ilmi asiayhteydestä.

8.1 newDialog

Käyttäjä valitsee pääikkunan File-valikosta kohdan new halutessaan avata uuden todistuksen. Tämän jälkeen ohjelma avaa näkyville uusi todistus -ikkunan, johon käyttäjä voi kirjoittaa kaavan sekä valita haluamansa kalkyylin.

Kuvan **??** kaaviossa nähdään vaiheissa A-C kuinka uusi todistusikkunan avataan.

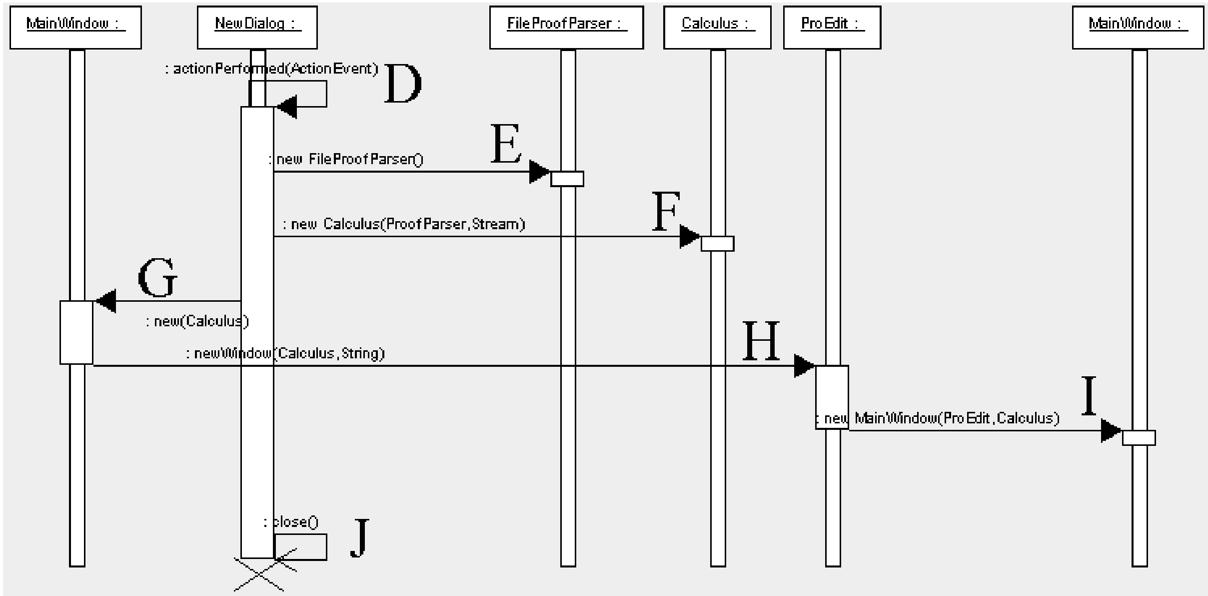
Aluksi huomataan käyttäjän valinneen new-valinnan (vaihe A). Sitten luodaan ja avataa uusi todistus -ikkuna (vaihe B) minkä jälkeen uusi todistus -dialogi luodaan (vaihe C).

8.2 newProof

Kun käyttäjä on syöttänyt uuden kaavan, luodaan uusi todistus. Se luo uuden kalkyyliolion käyttäjän syöttämien tietojen perusteella. Tämän jälkeen luodaan uusi pääikkuna jossa todistus on.

Kuvan **??** kaaviossa nähdään vaiheissa D-J kuinka uuden todistuksen avaaminen tapahtuu.

Kun ok-nappia on painettu uusi todistus -dialogissa (vaihe A), luodaan ensin uusi *FileProofParser*-ilmentymä (vaihe E) ja tämän jälkeen uusi kalkyyli-olio (vaihe F). Sitten avataan uusi todistus (vaihe G). Pääikkuna avaa uuden ikkunan ja siihen kalkyyli- olion



Kuva 7: Sekvenssikaavio: newProof

sisältämän todistuksen (vaihe H) minkä jälkeen *ProEdit* luo uuden pääikkunan tällä todistuksella.

8.3 newMainWindow

Uutta pääikkunaa luotaessa luodaan samalla myös navigointi-ikkuna.

Kuvan **??** kaaviossa nähdään vaiheissa K-O mitä tapahtuu uutta pääikkunaa avattaessa.

Aluksi pääikkuna luo todistusalueen (vaihe K) joka taas luo navigointialueen (vaihe L). Pääikkuna luo navigointiframen (vaihe N). Lopuksi pääikkuna

Lopuksi lisätään pääikkunaan luotu todistusalue (vaihe O).

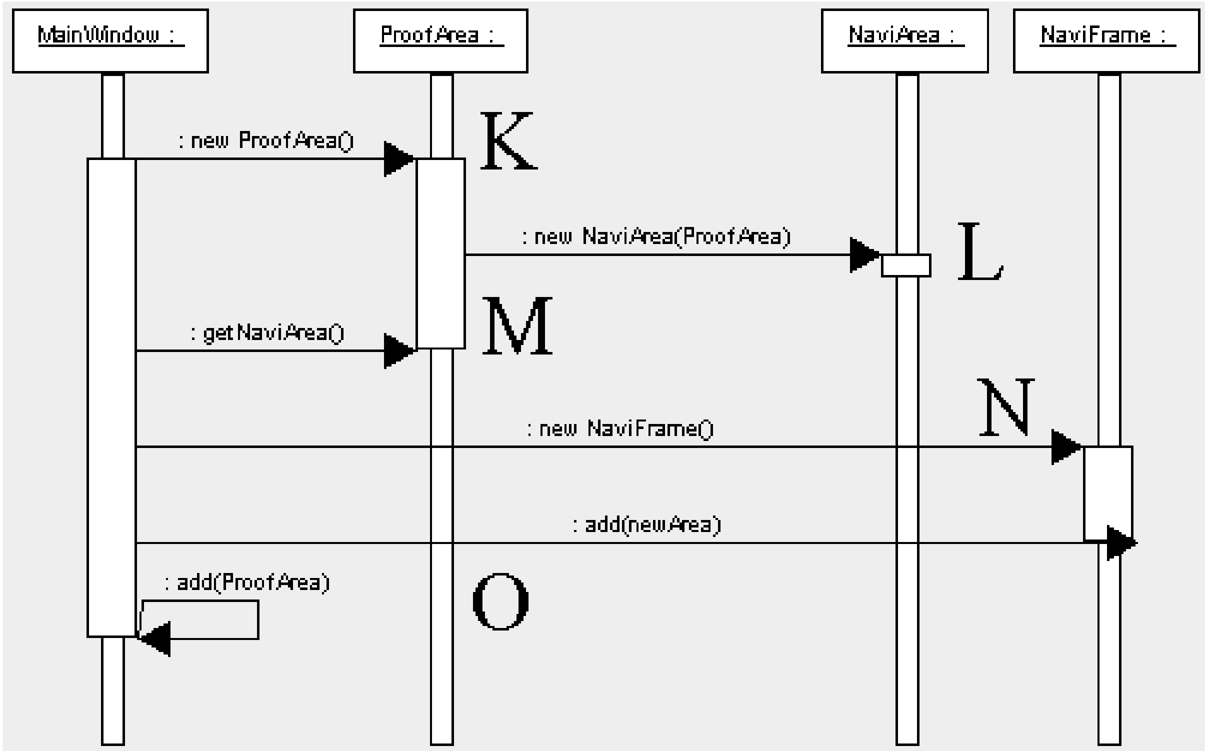
8.4 paint

Kuvan **??** kaaviossa nähdään vaiheissa T-Y kuinka todistuspuun piirtäminen suoritetaan. Aluksi tutkitaan rekursiivisesti oikea paikka ja tämän jälkeen suoritetaan todistuksen piirtäminen.

Asettelua varten ensin selvitetään rekursiivisesti koko alipuun viemä leveys (vaiheet T, U). Sitten asetetaan sekventti paikalle. Kun oikea paikka on löytynyt (vaihe W), seuraavaksi piirretään todistus paikalleen (vaihe X). Tämä tapahtuu piirtämällä rekursiivisesti jokainen konnektiivi, kaava ja monijoukko (vaihe Y).

Lopuksi tehdään samat operaatiot navigointipuulle (vaihe Z).

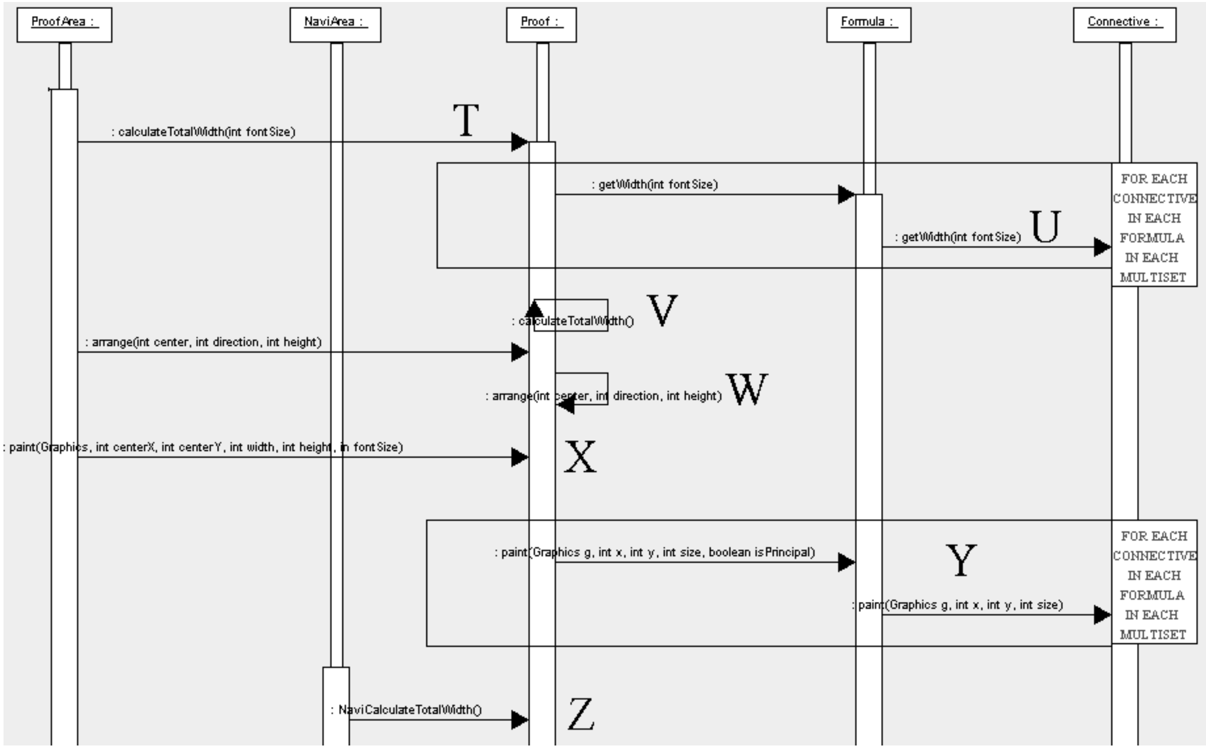
Seuraavassa on *ParsedFormulan* piirto-metodi joka palauttaa piirretyn formulan leveyden (parametrin fonttikokoa ja koordinaatteja ei ole otettu huomioon piirtämisessä):



Kuva 8: Sekvenssikaavio: newMainWindow

```
public int paint(Graphics g, int x, int y, int fontSize) {
   int width = 0;
   paint("(");
   width +="(".width();  //lasketaan sulun leveys ja lisätään se
                         //x-koordinaattiin
   if (leftChild instanceof ParsedFormula)
      width += leftChild.paint(g,x+width,y,fontSize); //rekursio
   else {
      paint(leftChild);  // jos vasenta alipuuta ei ole, piirretään lehti
      width += leftChild.width();
   }
   paint(Connective);
   width += Connective.width();

   if (rightChild instanceof ParsedFormula)
      width += rightChild.paint(g,x+width,y,fontSize);
   else {
      paint(rightChild);
      width += leftChild.width();
   }
   paint(")");
   width += ")".width();
   return width;
```



Kuva 9: Sekvenssikaavio: paint

```
}
```

Paintissa voidaan ottaa myös huomioon, että ruudun ulkopuolelle jääviä alueita ei turhaan piirretä.

8.5 mouseClicked

Kun konnektiivia on klikattu hiirellä, etsitään ensin mitä konnektiivia on klikattu ja tämän jälkeen tehdään tarvittavat operaatiot.

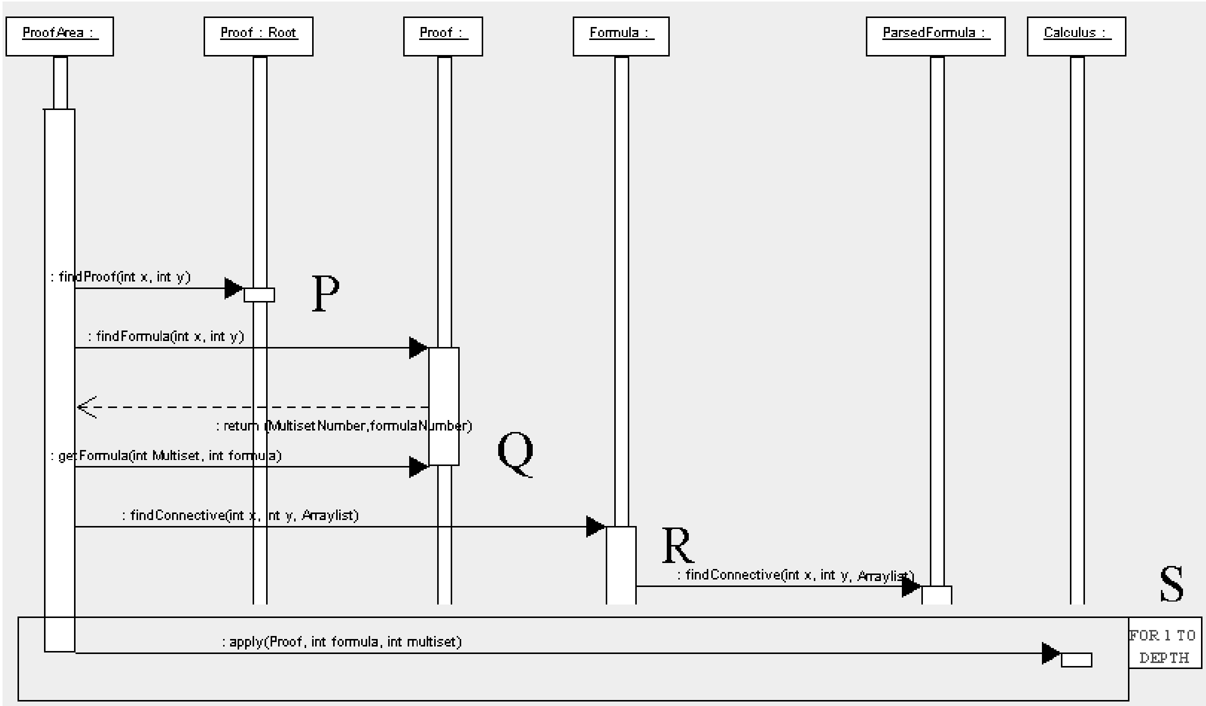
Kuvan **??** kaaviossa nähdään vaiheissa K-O miten toimitaan.

Ensin selvitetään koordinaattien avulla mistä kaavasta konnektiivia on klikattu (vaihe P). Sitten Kutsutaan *Formula*-luokan findConnective- metodia (vaihe R), joka välittää kutsun edelleen *ParsedFormulalle* (vaihe S).

Seuraavassa on *ParsedFormulan* findConnective-metodi, joka palauttaa listan reitistä (route), jolla klikattu konnektiivi löydetään jäsennyspuusta.

```
public ArrayList findConnective(int x) {
   ArrayList route = new ArrayList();

   if (x<this.leftX) {
      route.add(GO_LEFT);
```



Kuva 10: Sekvenssikaavio: mouseClicked

```
      if (leftChild instanceof String) //jos päädyttiin lehteen
         return null;                  //palautetaan null
      else
         return leftChild.findConnective(x); //rekursio

   else if (x>this.rightX) {
      route.add(GO_RIGHT)
      if (rightChild instanceof String) //jos päädyttiin lehteen
         return null;                   //palautetaan null
      else
         return rightChild.findConnective(x); //rekursio
   }
   else return route;   //konnektiivi on löytynyt kun
                        // leftX <= x <= rightX
}
```

Lopuksi suoritetaan apply- metodia rekursiivisesti (vaihe S).

Osa IV

Muuta huomioitavaa

9 Testaussuunnitelma

Testaussuunnitelmassa kerrotaan testauksen yleispiirteistä ja käytettävistä testausmenetelmistä. Testauksessa on tarkoitus löytää suurin osa ohjelmiston virheistä. Testauksesta laaditaan testausdokumentti jossa raportoidaan kaikki testit ja tiedot niiden onnistumisesta.

9.1 Yleistä testauksesta

Ohjelman testaus jakaantuu kolmeen osaan, Java-luokkien yksikkötestaukseen, testattujen luokkien yhteistoimintaa testaavaan integrointitestaukseen sekä käyttötapauksiin perustuvaan järjestelmätestaukseen. Yksikkötestauksessa etsitään lähinnä erillisten moduuleiden ohjelmointivirheitä ja niiden sisäisessä toiminnallisuudessa olevia vikoja, kun taas integrointitestauksessa testataan useammasta moduulista koostuvia kokonaisuuksia. Validointitestauksessa ohjelmiston toiminnallisuus testataan suorittamalla järjestelmällisesti mahdollisimman kattavasti kaikki käyttötapaukset.

Projektissa noudatetaan XP-mallin mukaisesti testausta jo ohjelmoinnin yhteydessä, joten testaus alkaa samaan aikaan kuin toteutuskin. Toteutusvaiheen jälkeen on varattu kolme viikkoa aikaa pelkkään testaukseen. Testauksen deadline on 6.5, jolloin myös testausdokumentti on valmis.

Testaukseen osallistuvat kaikki ryhmän jäsenet.

9.2 Testausmenetelmät

Testitapauksia valittaessa on käytettävissä kaksi päästrategiaa, mustalaatikkotestaus (black box testing, functional testing) tai lasilaatikkotestaus (glass/white box testing, structural testing).

White box- testauksessa tunnetaan kohteen sisäinen rakenne ja toteutus. White box-testauksessa testidatan valinnalla pyritään siihen, että testattava ohjelma tulee testattua mahdollisimman kattavasti.

Black box- testauksessa syötteet jaetaan ekvivalenssiluokkiin, joista samaan kuuluvat ne jotka käyttäytyvät samalla lailla testauksen kannalta. Näin ollen mahdollinen virhe löytyy jokaisella tämän luokan arvolla.

9.3 Yksikkötestaus

Toteutuksen jälkeen on jätetty kolme viikkoa pelkkään testaukseen, mutta jo toteutusvaiheessa ohjelmoija testailee tekemäänsä koodia. Main-metodi testausta varten kirjoitetaan ennen varsinaisen luokan kirjoittamista, jolla luokan valmistuttua sitä voidaan testata. Testit tehdään white box- testauksena ja niissä pyritään saavuttamaan haarakattavuus, eli jokaisessa ohjelman lauseessa pyritään käymään vähintään kerran.

Jotta voidaan olla varmoja että testausluokka on tehty ennen varsinaista luokkaa, tämä laitetaan CVS:ään heti testauksen jälkeen.

9.4 Integrointitestaus

Integrointitestauksessa pyritään testaamaan järjestelmän eri osien toimivuus yhdessä. Se tehdään siinä vaiheessa kun yksikkötestaus liitettävien luokkien osalta on valmis. Testaus tehdään lasilaatikkotestauksena. Erityisesti luokkien väliset rajapinnat ovat huomion kohteena. Integrointitestauksessa käytetään bottom up- menetelmää, luokkien toimintaa yhdessä testaillaan kun luokat on valmistuneet. Jos jotain integrointitestauksessa mukana ollutta luokkaa muokataan, joudutaan testit tekemään uudestaan.

Integrointitestaukseen osallistuvat luokkien tekijät.

9.5 Validointitestaus

Validointitestauksessa pyritään todentamaan että ohjelmisto toimii määritellyllä tavalla. Siinä pyritään selvittämään ulkoisesti havaittava toiminta, suorituskyky ja virheensietokyky. Määrittelydokumentissa esitettyjen käyttötapausten avulla kehitetään sopivat testit. Validointitestaukseen osallistuvat kaikki ryhmän jäsenet.

Testitapauksia esitellään seuraavassa.

Todistuksen lataaminen tiedostosta:

Testitapaus	Odotettu tulos
Ladataan kelvollinen todistus Ladataan kelpaamaton tiedosto	Onnistuu, todistus latautuu. Lataus ei onnistu. Ohjelma ilmoittaa virheestä ja pysytään latausikkunassa.

Todistuksen tallentaminen:

Testitapaus	Odotettu tulos
Tallennetaan todistus levylle.	Todistus tallentuu.

Todistuksen tulostaminen:

Testitapaus	Odotettu tulos
Tulostetaan koko puu. Tulostetaan puusta vain osa. Tulostetaan puu ASCII- muodossa. Tulostetaan puu latex- muodossa.	Toimii. Haluttu osa todistuspuuta tulostuu. Tulostuu Tulostuu

Todistuksen rakentaminen:

Testitapaus	Odotettu tulos
Peruutetaan toiminto Klikataan navigointipuun solmua Muutetaan fonttia.	Toiminto peruuntuu. Todistusikkuna kohdistuu solmun kohdalle. Ok

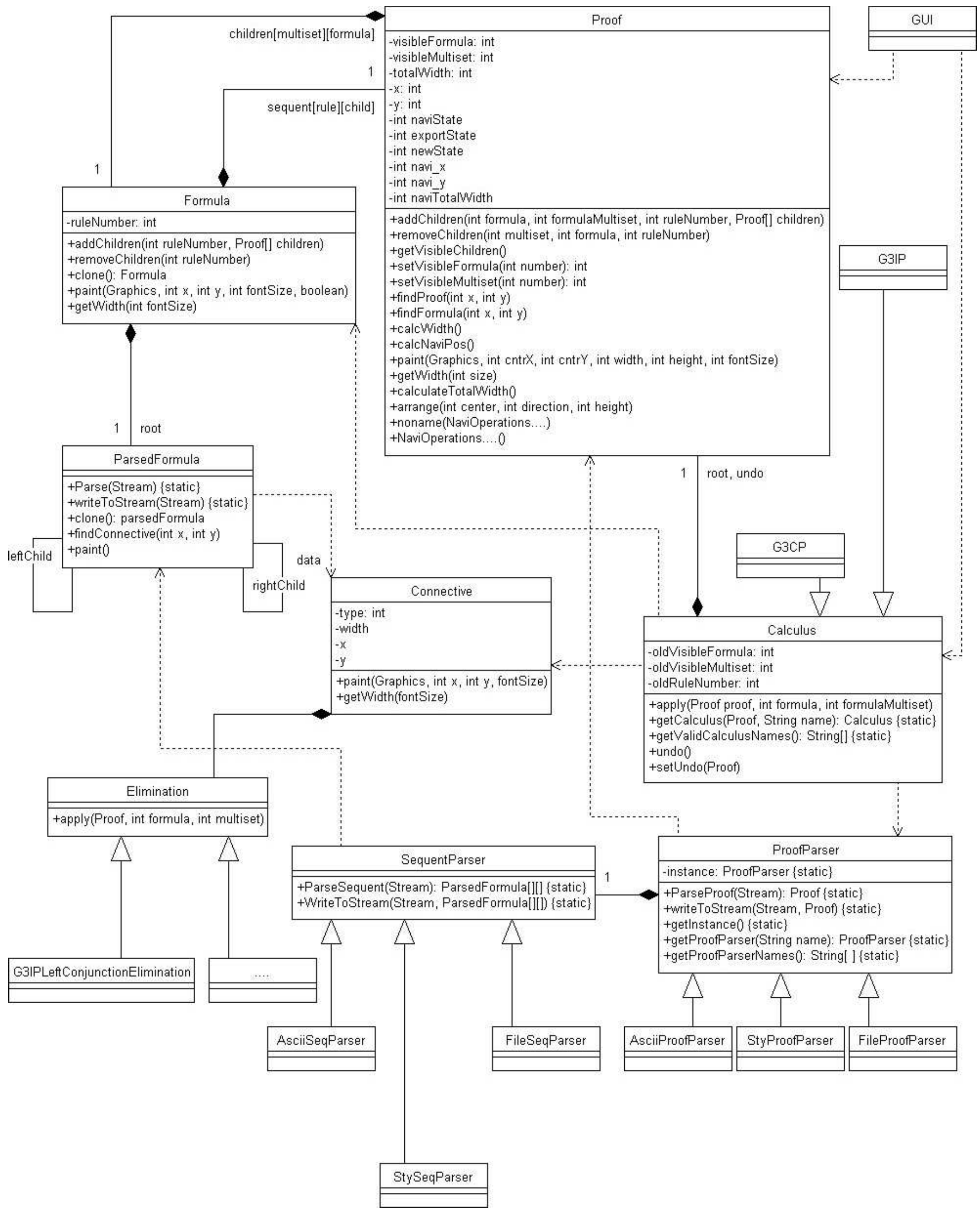
10 Yhteenveto

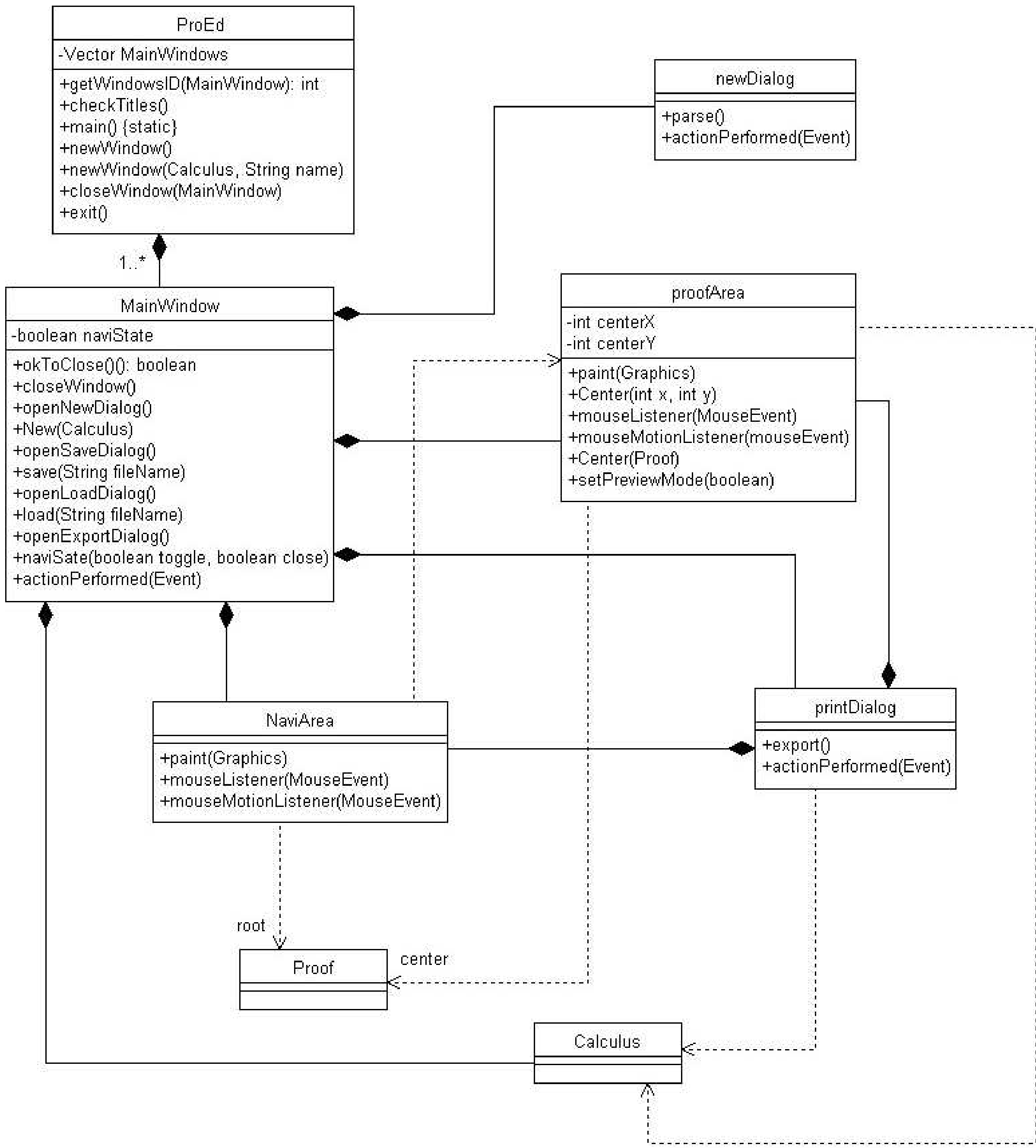
Todistustoimittimen suunnitteleminen oliomallilla tyylikkäästi on osoittautunut varsin vaikeaksi, eikä saatu ratkaisu ole oliovinkkelistä katsottuna erityisen elegantti. Paremmin oliomainen ratkaisu olisi varmasti mahdollinen, mutta ei ole luultavaa että se olisi yhtään helpommin ymmärrettävä, paremmin todellisuutta vastaava tai laajennettava. Näyttää myös siltä että järjestelmän suunnitteleminen merkittävästi itsenäisemmillä komponenteilla pakottaa järjestämään useampia kirjanpitoja samasta rakenteesta, mikä ei voi olla laajennettavuuden, ymmärrettävyyden tai ylläpidettävyyden kannalta hyvä ratkaisu.

Suunnitteludokumenttia kirjoitettaessa huomasimme että lukujen määrän säilyttäminen parittomana saattaa olla todella vaikeaa. Juuri kyseisen ensisijaisen tärkeän suunnitteludokumentin piirteen säilyttäminen on syy tämän luvun olemassaoloon.

Hakemisto

Liite 1 - luokkakaaviot




ProEd
-Vector MainWindows
+getWindowsID(MainWindow): int
+checkTitles()
+main() {static}
+newWindow()
+newWindow(Calculus, String name)
+closeWindow(MainWindow)
+exit()
newDialog
+parse()
+actionPerformed(Event)
1..*
MainWindow
-boolean naviState
+okToClose()(): boolean
+closeWindow()
+openNewDialog()
+New(Calculus)
+openSaveDialog()
+save(String fileName)
+openLoadDialog()
+load(String fileName)
+openExportDialog()
+naviSate(boolean toggle, boolean close)
+actionPerformed(Event)
proofArea
-int centerX
-int centerY
+paint(Graphics)
+Center(int x, int y)
+mouseListener(MouseEvent)
+mouseMotionListener(mouseEvent)
+Center(Proof)
+setPreviewMode(boolean)
NaviArea
+paint(Graphics)
+mouseListener(MouseEvent)
+mouseMotionListener(MouseEvent)
printDialog
+export()
+actionPerformed(Event)
root
Proof
center
Calculus