Tero Jäntti

Rakenteiset editorit

Tietotekniikan
(Ohjelmistotekniikka)
pro gradu -tutkielma
26. joulukuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Tero Jäntti
Yhteystiedot: tero.k.jantti@student.jyu.fi
Työn nimi: Rakenteiset editorit
Title in English: Structured editors
Työ: Tietotekniikan (Ohjelmistotekniikka) pro gradu -tutkielma
Sivumäärä: 70
Tiivistelmä: Työssä tarkastellaan rakenteisia editoreita, erityisesti niiden hyötyjä ohjelmoinnissa verrattuna perinteisiin tekstipohjaisiin editoreihin, sekä vertaillaan erilaisia olemassaolevia rakenteisia editoreita.
English abstract: The thesis examines structured editors, especially their advantages in programming over traditional text–based editors, and compares different existing structured editors.
Avainsanat: rakenteinen, editori
Keywords: structured, editor

Sisältö

1 Johdanto

Nykyään lähes kaikki suositut ohjelmointikielet sekä niiden kirjoittamiseen käytetyt editorit perustuvat tekstimuotoiseen dataan, jolla on tietty kielestä riippuva syntaksi. Tämä ei ole välttämättä huono asia: puhdas teksti on yksinkertainen ja universaali muoto tiedon säilyttämiseen ja tekstieditorien käytöllä on pitkät perinteet. Kuitenkin siinä on myös ongelmansa: syntaksin opettelu nostaa oppimiskynnystä ohjelmoinnin aloittelijoille. Koodin näkymä editorissa on sidottu tallennusmuotoon. Syntaksi tekee ohjelmointikielen laajentamisesta hankalaa.

Rakenteinen editori antaa käyttäjän käsitellä suoraan editoitavan ohjelman rakennetta siten, että ohjelmointikielen rakenteet ovat editorin atomisia osia, toisin kuin merkit ja tekstirivit perinteisessä tekstieditorissa [55]. Rakenteinen editori voi muistuttaa tekstieditoria tai se voi olla visuaalinen, esimerkiksi näyttäen ohjelman kontrollivuon nuolilla. Tässä työssä tutustutaan olemassaoleviin ohjelmoinnissa käytettäviin rakenteisiin editoreihin.

Rakenteisia ympäristöjä on kehitetty jo useiden vuosikymmenien ajan. Viime vuosina niiden kehitys on ollut aktiivista kielisuuntautuneen ohjelmoinnin (language oriented programming) kehityksessä, sillä rakenteinen lähestymistapa helpottaa uusien sovellussuuntautuneiden kielten kehitystä. Toisaalta, rakenteisia ympäristöjä ja kieliä voisi ihan yhtä hyvin käyttää perinteisten yleiskäyttöisten kielten sijasta. Työssä käydään läpi molempia vaihtoehtoja, kuten myös ohjelmoinnin oppimisympäristöjä, joissa ohjelmointi tapahtuu tekstin muokkaamisen sijaan graafisia elementtejä yhdistelemällä.

Editoreita tarkastellaan seuraavista näkökulmista:

- millaisia näkymiä koodiin on tarjolla
- ohjelmointikielten tuki
- käytön tehokkuus

Näiden lisäksi olisi voinut ottaa huomioon myös sellaisia seikkoja kuin opittavuus tai ylläpito (kuinka helppoa editorilla tuotettua ohjelmaa on ylläpitää). Kuitenkin, tämän tutkielman puitteissa nämä asiat jäävät käsittelemättä.

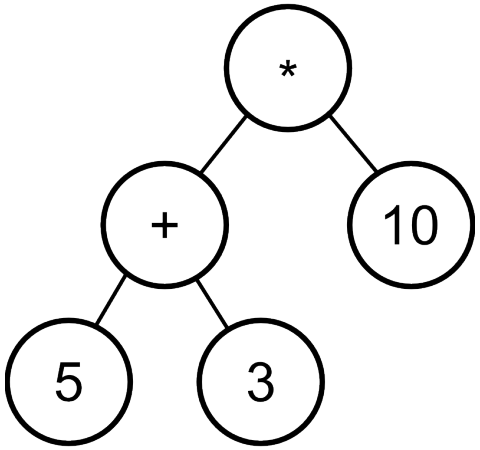
2 Tekstin editoinnista

Perinteisesti ohjelmontikielet ovat perustuneet tekstimuotoiseen dataan. Tosin sanoen ohjelma on jono merkkejä. Ohjelmointikielelle on määritelty *syntaksi* eli säännöt sille, miten merkeistä muodostuu ohjelmointikielen rakenteita, kuten esimerkiksi luokka tai funktio.

Kun ohjelma käännetään, kääntäjän ensimmäinen osa (niin sanottu *front-end*) muodostaa jäsennyksen lopuksi ohjelman rakenteesta jäsennyspuun (*abstract syntax tree*, lyh. *AST*). Tämä on puumuotoinen tietorakenne ohjelman rakenteesta. Esimerkiksi laskutoimituksesta

```
(5 + 3) * 10
```

voisi muodostua kuvan 2.1 mukainen jäsennyspuu.



Kuva 2.1: Yksinkertaisen laskutoimituksen jäsennyspuu

Jäsennyspuutä läpikäymällä kääntäjä voi tehdä ohjelmalle mm. tyyppitarkistuksen ja lopulta välikielen koodin generoinnin (josta edelleen voidaan muodostaa lopullinen konekielinen ohjelma).

2.1 Tekstieditoreiden historiaa

Tekstieditoreiden alkuajat ajoittuvat 60–luvulle. Deutschin ja Lampsonin vuoden 1967 paperissa [12] todetaan: ”Tavallisin tapa tekstin syöttämiseen muutaman viime vuoden aikana on ollut reikäkortit. Myös paperi– sekä magneettinauhoja on käytetty syötteenä, mutta se, että yksittäisiä kortteja voidaan lisätä ja poistaa reikäkorttipakan keskelle, on tehnyt reikäkorteista kätevimmän ja suosituimman tavan tekstin muokkaamiseen.”. Lisäksi todetaan, että etäkäyttöjärjestelmien myötä, joissa dataan pääsee käsiksi vain konsolin kautta, reikäkorteille on noussut vakava haastaja. Tarvitaan ohjelma, jota kutsutaan *editoriksi*, joka mahdollistaa käyttäjän luoda ja muokata tekstiä.

Samaisessa paperissa [12] esitellään tekstieditori nimeltä *QED*, jota on käytetty Berkeleyn SDS–930 –koneen moniajojärjestelmässä ja joka on suunniteltu käytettäväksi kaukokirjoittimien kanssa. Editorin ovat kehittäneet Butler Lampson, L. Peter Deutsch sekä Dana Angluin vuosina 1965–1966 [6].

QED–Editoria käytetään kolmessa eri *moodissa*: komentomoodi, tekstimoodi sekä rivin editointimoodi. Komentomoodissa editoria ohjataan tekstimuotoisilla komennoilla, joissa muokattava rivi (tai useampi rivi) määritellään rivinumerolla. Esimerkiksi rivien 12, 13 ja 14 tulostaminen tapahtuu komennolla

```
*12,14PRINT.
```

Komento

```
*12INSERT.
```

siirtää editorin tekstimoodiin, jolloin riville 12 voi kirjoittaa vapaata tekstiä.

Editorin komentokieli on yksinkertainen [12]. Editorin ominaisuuksia ovat mm. tekstin haku- ja korvaustoiminnot, puskurit joihin voi tallentaa usein käytettyjä tekstijaksoja tai komentoja sekä rivin editointimoodi, joka mahdollistaa rivin muokkaamisen merkki kerrallaan.

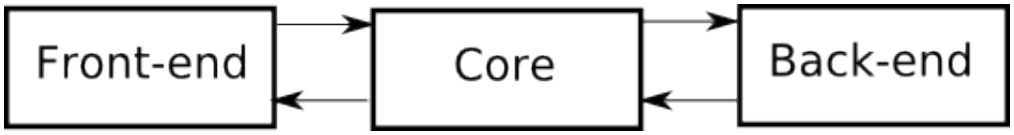
QED:sta on tehnyt oman versionsa mm. Ken Thompson, joka lisäsi editoriin tekstin hakemisen sekä korvaamisen säännöllisillä lausekkeilla. QED on merkittävä siinä mielessä, että siihen perustuu tunnettu editori *ed*, jonka ensimmäinen versio on vuodelta 1969. Edelleen ed:iin pohjautuu *vi* (1976), joka tehtiin näyttöpäätteillä käytettäväksi ja siinä on yksittäisen rivin muokkaamisen sijaan ajantasainen näkymä tekstiin, mikä on tietysti nykyään tavallista. [17] [60] [23]

2.2 IDE

Kun reikäkorteilla ohjelmoinnista siirryttiin aikoinaan terminaalien käyttöön, mahdollisti se myös interaktiiviset ohjelmointiympäristöt, joilla ohjelman kehitys onnistui kokonaan yhdestä koodieditorista käsin. Ensimmäinen tällainen ympäristö on Darthmouth BASIC -kielen terminaalista käsin käytettävä ympäristö, joka ilmestyi vuonna 1964. Ympäristö mahdollisti ohjelman ajamisen suoraan editorista käsin. Ympäristöä käytettiin kirjoittamalla tekstimuotoisia komentoja. Esimerkiksi komento *RUN* analysoi auki olevan ohjelman ja sen jälkeen ajaa sen tai osoittaa siinä olevat syntaksivirheet. Koodirivejä kirjoitetaan antamalla komentorivin alussa rivinumero ja sen jälkeen kyseisellä rivillä oleva ohjelmakoodi. [11]

Sittemmin Dartmouth BASIC:in kaltaisista interaktiivisista ohjelmointiympäristöistä on kehittynyt yhä monipuolisempia ja niistä on alettu käyttää nimitystä *IDE*, joka tulee sanoista *integrated development environment* [18].

Kirill Osenkov kirjoittaa rakenteisia editoreita käsittelevässä lopputyössään [55] tyypillisen tekstipohjaisen IDE:n arkkitehtuurista. Se koostuu kolmesta osasta: *front–end*, *core* sekä *back–end*, jotka näkyvät kuvassa 2.2.



Kuva 2.2: Tekstipohjaisen IDE:n arkkitehtuuri

Front–end on käyttöliittymä, eli tekstieditori ja siihen kuuluvat kontrollit. Back–endillä tarkoitetaan kääntäjää sekä muita ohjelmointityökaluja kuten debuggeria. Edellämainittujen välissä oleva osa, core, sitoo edellä mainitut yhteen ja sisältää *language service* -palvelun, joka tekee IDE:stä ”älykkään”. Language service muodostaa ohjelman jäsennyspuun ja tämän avulla tarjoaa kielen rakennetta ymmärtäviä palveluita, kuten vaikkapa muuttujan uudelleennimeämisen.

Osenkov jatkaa, että tyypillisessä IDE:ssä em. kolmen osan väliset rajapinnat perustuvat tekstiin. Vaikka core muodostaakin tekstistä jäsennyspuun, viedään koodi silti tekstimuotoisena kääntäjälle. Kääntäjä muodostaa oman sisäisen jäsennyspuunsa ja toimii mustana laatikkona coreen nähden. Tässä menetetään Osenkovin mukaan monia mahdollisuuksia muokata käännösvaihetta: kielen laajennokset, koodin injektiot, aspect–oriented programming (AOP) ja niin edelleen.

Samoin front–endin ja coren välinen rajapinta perustuu tekstiin. Osenkov kirjoittaa, että käyttäjän muokkausten merkitys menetetään heti muokkausta tehdessä

ja että core joutuu tekemään paljon työtä ja käyttämään monimutkaisia algoritmeja merkityksien hakemiseksi tekstistä.

3 Rakenteisen editorin toimintaperiaatteet

Puhtaasti rakenteisessa editorissa ohjelmoijan ei tarvitse ajatella syntaksia, vaikka toki ohjelman tallennusmuoto voi määritellä jonkunlaisen syntaksin. Editointikomennot kohdistuvat suoraan ohjelman rakenteisiin. Esimerkiksi ohjelmoija voisi luoda uuden listan valitsemalla editorin valikosta lista–rakenteen, kun taas tekstieditorissa täytyisi muistaa oikea syntaksi saman asian saavuttamiseksi. Editori voisi helposti myös olla näyttämättä lista–rakennetta, kun listan luominen ei ole järkevää, esimerkiksi jos ollaan luokan ulkopuolella.

Rakenteisessa editorissa näkymä on eroteltu sisällöstä, samaan tapaan kuin CSS–kieli [36] määrittää ulkoasun HTML–kuvauskielellä [3] luodulle dokumentille. Niinpä kielen syntaksi ei määritä sitä, miten editori näyttää koodin käyttäjälle. Rakenteinen editori mahdollistaa näkymän kustomoimisen halutunlaiseksi tai jopa useita erilaisia näkymiä samaan koodiin.

Rakenteisen editorin arkkitehtuuri perustuu Osenkovin mukaan [55] *malli–näkymä–ohjain–suunnittelumalliin* (*model–view–controller* eli *MVC*) [58], sillä siinä on periaatteena näkymän erottaminen sisällöstä niin, että näkymä on helposti vaidettavissa ja että niitä voi olla useita. Mallin paikalla on koodin jäsennyspuu, näkymänä editointinäkymä ja ohjaimena käyttöliittymäelementit jotka vastaavat kielen rakenteita.

4 Rakenteisten editoreiden käyttökohteita

Rakenteisia editoreita on nykyään käytössä muun muassa erikoistarkoitukseen tehdyissä mallinnuskielissä (esim. LabVIEW, jolla ohjelmoidaan mm. LEGO–robotteja [28]), aloittelijoille suunnatuissa ohjelmointiympäristöissä sekä kielisuuntautuneessa ohjelmoinnissa.

4.1 Ohjelmoinnin oppimisympäristöt

Jo kuusikymmenluvulta lähtien on kehitetty lukuisia ohjelmointiympäristöjä tarkoituksena madaltaa kynnystä ohjelmoinnin opetteluun sekä tehdä ohjelmoinnista helpommin lähestyttävää suuremmalle joukolle ihmisiä [25]. Tässä tutkielmassa tämänkaltaisia ympäristöjä ovat Alice sekä Scratch.

Aloittelijoiden ympäristöissä syntaksivirheet voivat hankaloittaa ohjelmoinnin periaatteiden opettelemista suhteettoman paljon ja tähän mm. Alicen tekijät ovat kiinnittäneet huomiota [8]. Sekä Alicessa että Scratchissä ohjelmointi tapahtuu raahaamalla ohjelmointilausekkeita hiirellä editorissa, ilman että mitään välimerkkejä tarvitsee muistaa lisätä.

4.2 Kielisuuntautunut ohjelmointi

Kielisuuntautunut ohjelmointi (*language–oriented programming*) tarkoittaa ohjelmointiongelman ratkaisemista kehittämällä sovellusaluesuuntautunut, hyvin korkean tason kieli, jolla ohjelma toteutetaan [68]. Tällaisesta tietylle sovellusalueelle räätälöidystä kielestä käytetään nimitystä *sovellussuuntautunut kieli* (usein käytetään lyhennettä *DSL*, *domain specific language*) [64]. Tällaisilla kielillä on tarkoitus ilmaista ohjelman logiikka tiiviimmin sovellusalueen termeillä sekä mahdollistaa huonommin ohjelmointia osaavien, mutta sovellusalueesta paremmin perillä olevien, ihmisten kehittää ohjelmaa. Kielisuuntautunutta ohjelmointia tukevia työkaluja ovat mm. JetBrainsin *MPS* (Meta Programming System), Intentional Softwaren *Knowledge Workbench*, Microsoftin *Software Factories* sekä MetaCase–yhtiön *MetaEdit+*.

Martin Fowler kirjoittaa vuoden 2005 artikkelissaan [14] kielisuuntautuneeseen ohjelmointiin suunnatuista työkaluista, joista hän käyttää nimitystä *language work-*

bench. Viime vuosina language workbench –työkaluja on kehitetty aktiivisesti, mutta ne eivät ole vielä saavuttaneet valtavirran suosiota.

Fowler jaottelee sovellussuuntautuneet kielet *ulkoisiin* (*external*) eli erillisillä työkaluilla tehtyihin (mm. Lex [35] ja Yacc [22]) ja *sisäisiin* (*internal*) eli suoraan sovelluksen isäntäkielellä tehtyihin (malliesimerkki Lisp [40]). Kielisuuntautuneiden työkalujen yhteydessä Fowler tuo esille termit *konkreettinen syntaksi* ja *abstrakti syntaksi*. Konkreettisella tarkoitetaan kielen näkyvää esitystä, esimerkiksi alkaako koodilohko aaltosulkeella vai merkitäänkö lohkoa sisennyksellä. Abstraktilla taas kielen rakennetta, joka ei ota ota kantaa siihen miten se käyttäjälle näkyy. Esimerkiksi jos abstaktina esityksenä olisi lista puhelinnumeroista, sen konkreettisia muotoja voisivat olla vaikkapa XML [5] tai YAML [2].

Ulkoisten sovellussuuntautuneiden kielten haittapuolena on niiden huono integroituvuus isäntäkieleen (Fowler käyttää termiä *symbolic integration*). Tämä tarkoittaa mm. sitä, että isäntäkielessä tehdyt automaattiset refaktoroinnit eivät välity sovellussuuntautuneeseen kieleen. Edelleen haittapuolena on se, että mikään editori ei tue sovellussuuntautuneen kielen muokkaamista, kun nykyään IDE:iltä odotetaan paljonkin toiminnallisuuksia.

Sisäisillä kielellä näitä ongelmia ei ole (ainakaan yhtä merkittävästi), mutta niiden toivimuus riippuu siitä miten joustava isäntäkieli on. Fowlerin mukaan dynaamisilla kielillä (mm. Lisp, Smalltalk [15] sekä Ruby [61]) tämä on helpommin saavutettavissa kuin suosituilla staattisilla kielillä (Java [16], C++ [65], C# [20]). Toisaalta, vaikka IDE tukisi sisäisen sovellussuuntautuneen kielen käyttöä, se ei kuitenkaan tiedä miten sitä kuuluisi ja miten sitä ei kuuluisi käyttää. Lisäksi vaaditaan hyvää isäntäkielen tuntemusta, mikä on vastoin yhtä sovellussuuntautuneen kielen tavoitetta, että kieltä voisivat myös vähemmän ohjelmointia osaavat käyttää.

Fowlerin mukaan Modernit kielisuuntatuneet työkalut ovat rakenteisia editoreita, joskin myös monia tekstiin pohjautuvia työkaluja on olemassa [41]. Rakenteiset editorit muokkaavat kielen abstraktia syntaksia. Työkalu projisoi abstaktista rakenteesta esityksen editointia varten. Tämä esitys ei ole sama kuin kielen tallennusmuoto, joka on serialisointi abstraktista rakenteesta.

Uuden sovellussuuntautuneen kielen luomisessa on kolme vaihetta:

- abstraktin syntaksin (skeeman) määrittely
- editorin tekeminen
- generaattorin tekeminen (joka kääntää koodin ajettavaan muotoon)

Kielisuuntautuneet työkalut helpottavat sovellussuuntautuneiden kielten tekemistä mm. siten, että ei tarvitse kirjoittaa jäsennintä ja että eri kielten välillä on hyvä IDE–tuki. Riskinä mainitaan standardien puute ja siitä seuraava yhteen työkaluun sitoutuminen.

5 Historiallisia rakenteisia editoreita

Rakenteisia editoreita on kehitetty pian sen jälkeen, kun teknologia on sen mahdollistanut. Yksi ensimmäisia on Cornell Program Synthesizer, joka esitellään seuraavaksi. Hieman myöhemmin on kehitetty The Mjølner System, joka esitellään luvussa 6.1.

5.1 Cornell Program Synthesizer

Yksi ensimmäisiä rakenteisia ohjelmointiympäristöjä oli Cornellin yliopistossa kehitetty The Cornell Program Synthesizer, jota on kuvattu Teitelbaumin ja Repsin vuoden 1981 paperissa [67]. Synthesizerin periaatteena on, että ohjelmat eivät ole tekstiä vaan hierarkkisia rakennelmia, joten myös editorin tulisi tukea tätä näkemystä. Ympäristö tarjoaa nykyisistä IDE:istä tuttuja toimintoja, kuten debuggerin sekä virheellisten kohtien korostaminen ohjelmakoodissa. Sitä on käytetty ohjelmoinnin opetuksessa ja se tukee mm. erästä PL/I–kielen murretta.

Editoitavissa tiedostoissa on hierarkkinen rakenne, joka muodostuu kahdenlaisista elementeistä: malleista (*template*) sekä fraaseista (*phrase*). Malli on ennalta määritelty tekstilohko. Malleja ovat esimerkiksi aliohjelma tai if–lause. Mallissa on laajennuspaikkoja (*placeholder*), joihin voi kirjoittaa sisäkkäisiä malleja tai lausekkeita. Sisäkkäisten elementtien tulee olla oikean tyyppisiä, esimerkiksi if–lauseessa tulee olla ehtolauseke. Laajennuspaikkoja lukuunottamatta malleja ei voi muokata. Näin ohjelman rakenne säilyy aina eheänä.

Seuraavassa on esimerkki if–lauseen mallista, jossa ehtolausekkeen sekä else–lauseen laajennuspaikkoihin on lisätty koodia kun taas then–lauseen laajennuspaikka (*statement*) on jätetty tyhjäksi:

```
IF (k > 0)
   THEN statement
   ELSE PUT SKIP LIST ('not positive');
```

Fraasi on merkkijono, joka yleensä vastaa ohjelmointikielen lauseketta. Fraasia voi muokata vapaasti kuten tekstieditorissa, muuta fraasin tulee olla syntaksiltaan oikea, kun kursori siirtyy pois fraasista. Erottelulla malleihin sekä fraaseihin haetaan käytännöllistä tasapainoa rakenteisen ja tekstieditorin välille [67].

Mallien tekstiä ei kirjoiteta itse, vaan ne luodaan tekstimuotoisilla komennoilla. Editorin kursorin voi siirtää vain sellaisiin paikkoihin, joihin koodin kirjoittaminen on sallittua, siis malleissa laajennuspaikoista toiseen tai fraasissa vapaasti, kuten tekstieditorissa. Kun kursoria liikutetaan eteenpäin, se hyppää tarvittaessa seuraavan laajennuspaikan kohdalle.

Kun kursori on mallin kohdalla, editointikomennot muokkaavat mallia kokonaisuutena. Näin jotkin muokkaukset onnistuvat varsin tehokkaasti. Teitelbaum ja Reps kertovat [67], kuinka if–lauseen ympäröimisen while–silmukalla voi tehdä seitsemällä näppäimenpainalluksella:

- clip (if–lauseen poisto)
- `.dw` return (while–silmukan lisääminen)
- return (kursorin siirtäminen silmukan sisällä olevaan laajennuspaikkaan)
- insert (if–lauseen lisääminen takaisin)

Synthesizer tukee ohjelmien debuggausta, mm. askeltamalla ohjelman suoritusta malli kerrallaan. Ohjelman muokkausta ajon aikana tuetaan. Ohjelman suoritus keskeytetään, jos tullaan tyhjän laajennuspaikan kohdalle, ja suoritus jatkuu kun puuttuva kohta on täydennetty.

Teitelbaumin ja Repsin paperissa [67] todetaan, että mallien käytöllä saavutetaan monia etuja: mm. Ohjelma on aina syntaktisesti oikein, sisennys on automaattista, koodin kirjoittaminen on nopeaa ja ohjelmoija voi syntaksin yksityiskohtien sijaan työskennellä korkealla abstraktiotasolla. Toisaalta todetaan, että toisinaan Synthesizer on sekava ja epämukava. Tälläinen tilanne on esimerkiksi, jos halutaan siirtää aliohjelman lokaali muuttuja aliohjelman parametriksi. Tekstinä molemmat näyttävät samalta, mutta ohjelman jäsennyspuun kannalta parametri ja muuttuja ovat kaksi eri asiaa, joten parametrin siirtäminen aliohjelman muuttujaksi ei onnistu suoraviivaisella leikkaa–liitä –operaatiolla, kuten voisi äkkiseltään odottaa. Tällaiseen ehdotetaan ratkaisuksi *muunnoksia* mallien välille (template–to–template transformations), eli valmiita editointikomentoja, jotka tekevät useimmin suoritettavia koodin muokkauksia. Vastaava toiminnallisuus on muuten toteutettuna mm. JetBrains MPS:ssä (luku 6.6), jossa siitä käytetään nimeä *intention*.

6 Rakenteisia editoreita

Tarkastellaan joitakin olemassaolevia rakenteisia editoreita. Tarkasteltava joukko on ominaisuuksiltaan varsin kirjava. Editoreiden kohderyhmä vaihtelee laajasti. Niitä on kehitetty eri aikoina. Niitä yhdistää oikeastaan vain rakenteisuus.

Editorit voisi jakaa karkeasti kolmeen ryhmään. Ensimmäinen on *yleiskäyttöiset ohjelmointityökalut*, jotka vastaavat karkeasti ottaen nykyisiä sovelluskehittimiä. Tähän ryhmään kuuluvat The Mjølner System sekä Lava.

Toinen ryhmä on *aloittelijoille suunnatut editorit*, joiden ominaisuusvalikoima on suppea, mutta joissa painottuu helppokäyttöisyys. Tällaisia ovat Alice sekä Scratch.

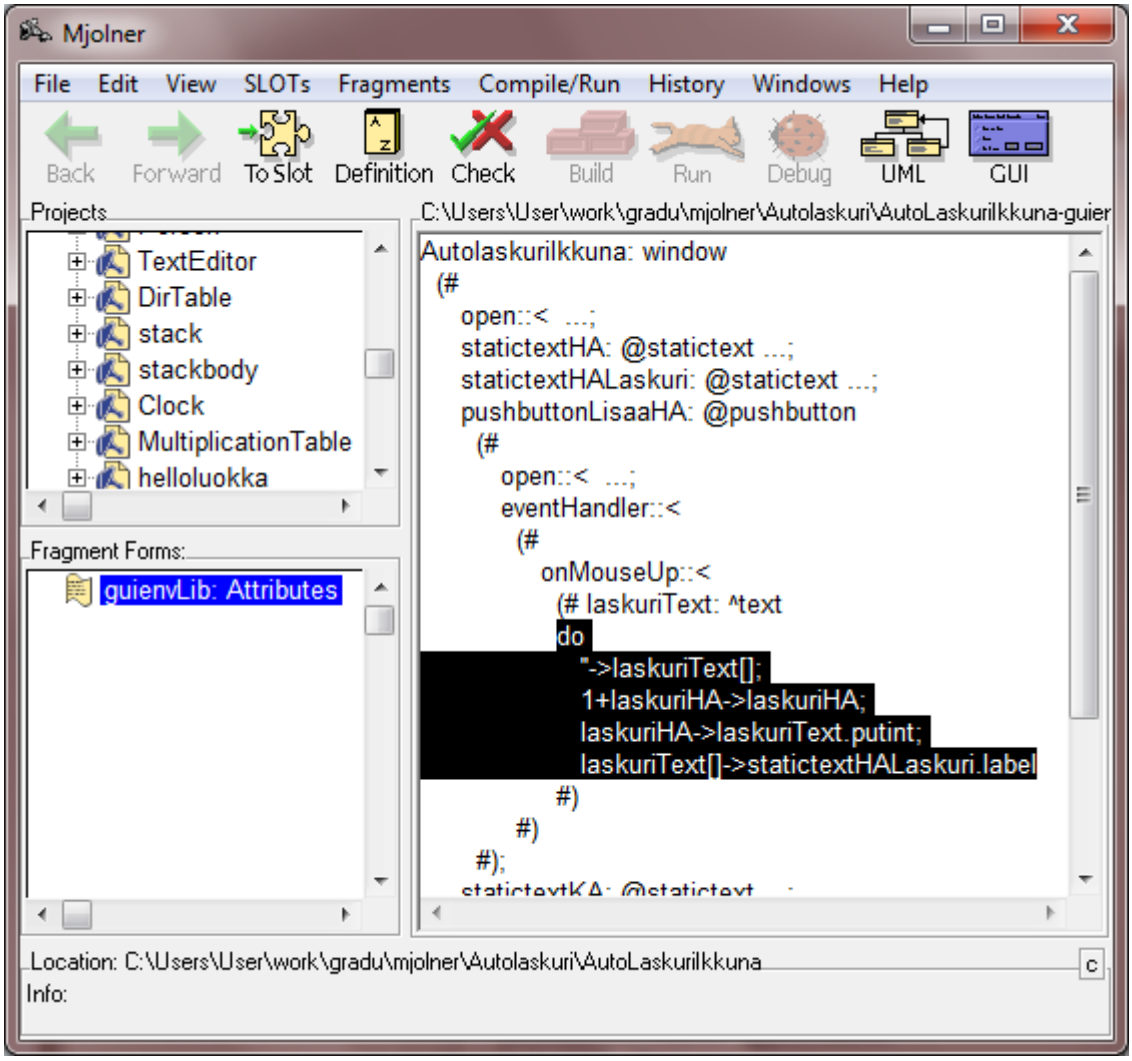
Kolmas ryhmä on *kielisuuntautuneen ohjelmoinnin työkalut*. Nämä on suunniteltu sovellussuuntautuneiden kielten kehittämiseen ja näissä rakenteisuus mahdollistaa usean kielen yhteistoiminnan. Tällaisista työkaluista mukana ovat JetBrains Meta Programming System sekä MetaEdit+.

Edellä mainittujen editoreiden lisäksi mukana on myös Subtext, jonka kaksiulotteinen ohjelmointimalli poikkeaa kaikista muista. Kyseessä on eräänlainen teknologiademo, joten sen käyttö rajoittuu yksinkertaisiin esimerkkiohjelmiin.

6.1 The Mjølner System

The Mjølner System –ohjelmointiympäristö syntyi yhteistyönä pohjoismaisten yliopistojen sekä yritysten kesken vuosina 1986–1991. Mjølner–projektin tavoitteena oli kasvattaa tuottavuutta laadukkaiden ohjelmistojen tuottamiseksi kehittämällä ympäristö, joka tukee suurten ohjelmien määritystä, toteutusta sekä ylläpitoa. Ympäristö on toteutettu seuraaville alustoille: Windows (95, 98, 2000, NT), Unix sekä OS X. [44] [27]

Ympäristö käyttää ohjelmointikieltä nimeltä BETA [53], joka on SIMULA–kieleen pohjautuva olio–ohjelmointikieli. Mjølnerin kehitys myös edisti BETA–kielen kehitystä. Kehitysprojekti johti yhtiön nimeltä Mjølner Informatics Ltd. perustamiseen vuonna 1988, joka kehitti ja myi järjestelmää monien vuosien ajan. Sen myynti ei kuitenkaan tuottanut paljon voittoa ja nykyään sen kehitys ei ole enää aktiivista. Tässä tutkielmassa testattu versio on GNU:n SDK:ta käyttävä 5.2.2 vuodelta 2002. [27]



Kuva 6.1: The Mjølner Systemin Sif–editorin pääikkuna

Ympäristöön kuuluu mm. seuraavat työkalut:

- Freja - Oliomallin suunnittelutyökalu
- Sif - Rakenteinen koodin editori ja selain (kuva 6.1)
- Frigg - Käyttöliittymäeditori
- Valhalla - Debuggeri
- Kääntäjä BETA–kielelle
- Kirjastot
- Yggdrasil - Metaohjelmointijärjestelmä

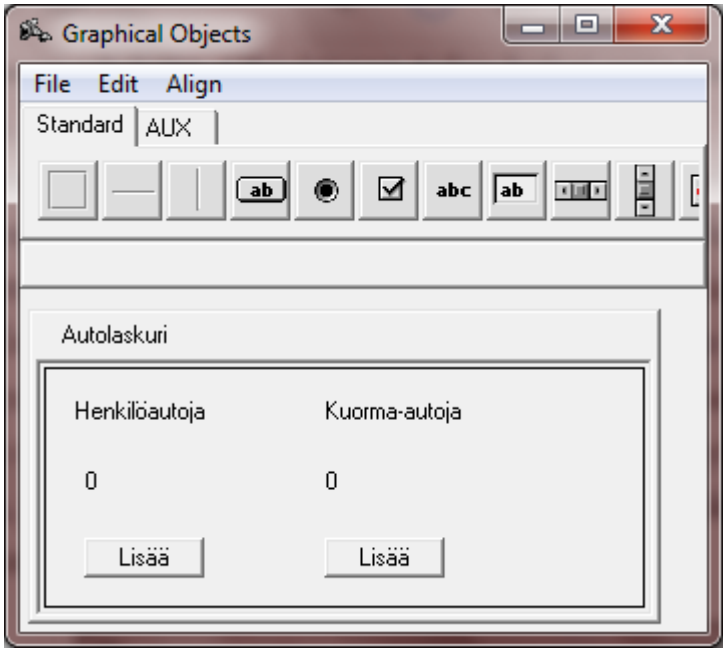
Metaohjelmointijärjestelmä tarjoaa yhtenäisen rajapinnan koodin ohjelmalliseen

muokkaamiseen. Eri työkalut käyttävät sitä tarjotakseen erilaisia näkymiä koodiin. Esimerkiksi Frejan UML–kaaviossa tehdyt muutokset näkyvät suoraan Sifin koodinäkymässä. Sif ei salli syntaksiltaan puuttellista koodia.

6.1.1 Esimerkki

Havainnollistetaan Mjølnerin käyttöä tekemällä Autolaskuri–ohjelma [30], jolla voi laskea ohi ajavien henkilö– ja kuorma–autojen määriä. Ohjelma näyttää lukumäärän sekä henkilö– että kuorma–autoille ja lisäksi painikkeet molempien lukemien kasvattamiseen yhdellä. Ohjelmaan tulee kaksi osaa (modulia): Ikkunaluokka sekä pääohjelma.

Ikkunaluokan tekeminen aloitetaan painamalla Mjølnerin työkalurivin *GUI*–painiketta. Avautuvasta dialogista valitaan tiedosto sekä luokan nimi ja sitten avautuu käyttöliittymäeditori, joka muistuttaa monia vastaavia graafisten käyttöliittymien työkaluja (esimerkiksi Borland Delphi [30]). Editorissa voi raahata käyttöliittymäelementtejä paikoilleen editoitavaan komponenttiin, tässä tapauksessa ikkunaan, sekä muokata niiden ominaisuuksia. Sitä mukaa kun käyttöliittymää muokataan editorin kautta, pääikkunan koodinäkymässä päivittyy ikkunaluokan koodi. Lopputulos näkyy kuvassa 6.2.



Kuva 6.2: Autolaskuriohjelman käyttöliittymän näkymä

Jotta autojen laskeminen onnistuisi, lisätään luokkaan koodieditorissa muuttuja

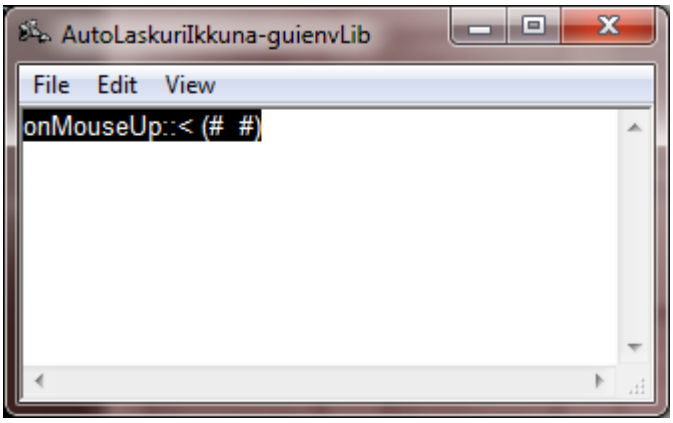
henkilöautojen määrälle. Valitaan koodieditorissa luokan viimeinen jäsen, *pushbuttonLisaaKA*–painike, ja painetaan enter. Tällöin koodiin tulee uusi elementti, johon voi kirjoittaa koodia. Siirrytään elementin kohdalla tekstin editointimoodiin yksinkertaisesti alkamalla kirjoittaa muuttujan määrittelyä (kuva 6.3).

```
    statictextKA: @statictext ...;
    statictextKALaskuri: @statictext ...;
    pushbuttonLisaaKA: @pushbutton ...;
    [[ laskuriHA: @integer ]]
#)
```

Kuva 6.3: Uuden muuttujan lisääminen ikkunaluokkaan

Näppäinyhdistelmä ctrl-välilyönti poistuu tekstineditointimoodista kokonaan rakenteiseen moodiin, mutta vain jos koodi on editoinnin jälkeen syntaksin mukainen. Samalla tavalla lisätään toinenkin muuttuja kuorma–autojen laskentaa varten.

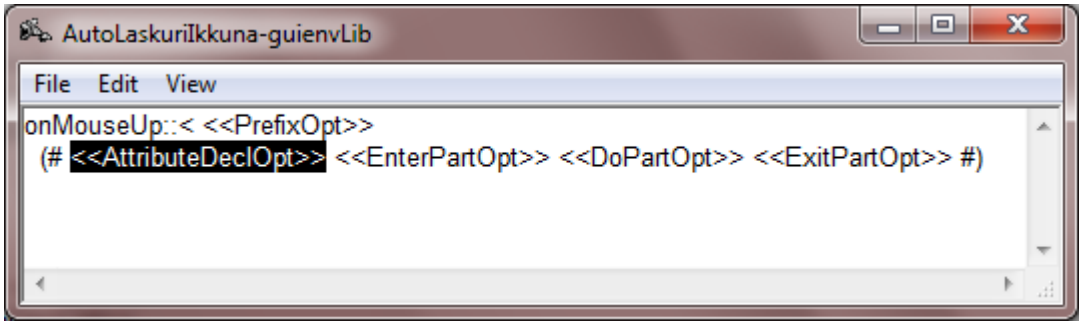
Lisätään seuraavaksi ohjelman varsinainen toiminnallisuus, eli laskurin arvon kasvattaminen sen alapuolella olevaa painiketta painamalla. Avataan jälleen käyttöliittymäeditori (ikkunaluokka tulee olla valittuna, jotta se tulee editoitavaksi) ja lisätään painikkeille *object inspector* –näkymästä tapahtumankäsittelijä tapahtumalle *onMouseUp*. Tapahtumankäsittelijän koodin kirjoittamista varten avautuu kuvan 6.4 pikkuikkuna, jossa näkyy ainoastaan tapahtumankäsittelijän koodi (koodia voi halutessaan muokata myös pääikkunan koodinäkymästä, pikkuikkuna on vain yksi vaihtoehtoinen näkymä koodin rakenteeseen).



Kuva 6.4: Pikkuikkuna tapahtumankäsittelijän kirjoittamiseksi

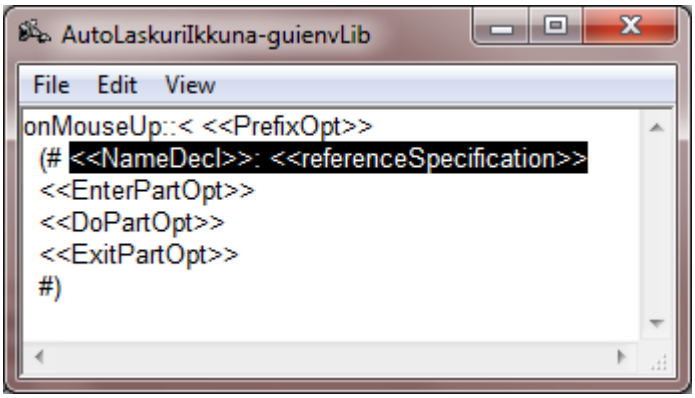
Koodin kirjoittamisen tapahtumankäsittelijään voisi tehdä seuraavasti. Poistetaan aluksi alussa näkyvä tyhjä koodilohko ja valitaan tilalle hiiren oikean painikkeen kontekstivalikosta vaihtoehto *ObjectDescriptor*, jolloin kyseinen rakenne auke-

aa koodiin (kuva 6.5). Rakenne sisältää *välikkeitä* (*nonterminal*), jotka koodissa esitetään kulmasuluilla (<<...>>). Välikkeet ovat BETA-kielen kieliopin rakenteita, joita täytyy vielä laajentaa eteenpäin, jotta saavutetaan kieliopin mukainen ohjelma. Sana *Opt* välikkeen perässä kertoo sen olevan *valinnainen*: sen voi jättää kokonaan poiskin.



Kuva 6.5: Tapahtumankäsittelijän keskeneräinen jäsennyspuu

Välikkeen laajentaminen tehdään klikkaamalla jälleen hiiren oikealla ja valitsemalla aukeavasta valikosta jokin kyseiseen paikkaan sopiva rakenne. Esimerkiksi välikkeen *AttributeDeclOpt* tilalle valitaan tässä esimerkissä *SimpleDecl*, joka on uuden muuttujan määrittelevä lauseke. Laajennettu koodi näkyy kuvassa 6.6: siinä muuttujan määrittely koostuu kahdesta osasta. Kirjoitetaan ensimmäiseen muuttujan nimi ja toiseen sen tyyppi (huomaa, että editori on rivittänyt koodin automaattisesti eri lailla koodin kasvaessa).

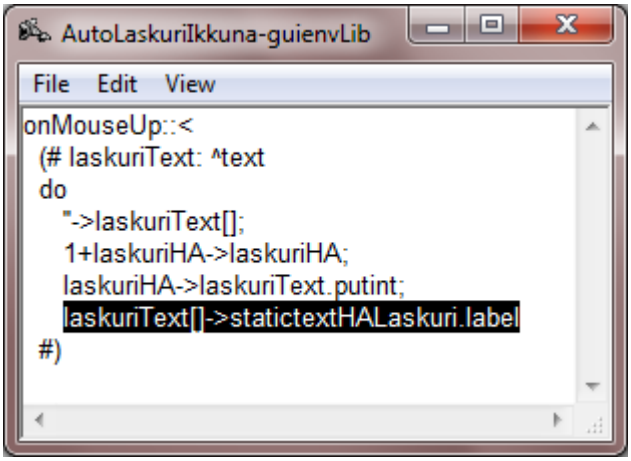


Kuva 6.6: Muuttujan määrittelylauseke (SimpleDecl) laajennettuna

Muuttujan määrittelyn olisi voinut tehdä samalla tavoin kuin tehtiin ikkunaluokan muuttujat autojen määrille: siirtymällä editointitilaan jo välikkeen *AttributeDeclOpt* kohdalla ja kirjoittamalla suoraan muuttujan nimen, kaksoispisteen sekä

tyypin. Muuttujan määrittelyn tapauksessa kumpikin tapa on helppo, mutta muuten lienee tapauskohtaista kuinka pitkälle välikkeitä kannattaa laajentaa.

Samalla tavoin jatkaen kirjoitetaan tapahtumankäsittelijän loppu koodi (muuttujan alustaminen, laskurin arvon kasvattaminen, arvon muuttaminen merkkijonoksi sekä merkkijonon asettaminen käyttöliittymän tekstikenttään). Kun tämä on tehty, poistetaan vielä jäljelle jääneet valinnaiset välikkeet valikon komentoa käyttämällä. Tapahtumankäsittelijän valmis koodi näkyy kuvassa 6.7. Samanlainen tapahtumankäsittelijä tehdään myös kuorma–autojen laskemiselle.



Kuva 6.7: Valmis tapahtumankäsittelijä

Lopuksi tehdään vielä pääohjelma. Luodaan Sif–editorin pääikkunasta uusi ohjelma, lisätään sille viittaus autolaskuri–ikkunaan ja kirjoitetaan lyhyt koodi, joka luo ja avaa ikkunan:

```
GuiEnv (# ikkuna: @AutoLaskuriIkkuna do ikkuna.open #)
```

Pääikkunasta käsin voidaankin sitten kääntää, ajaa tai debugata ohjelmaa.

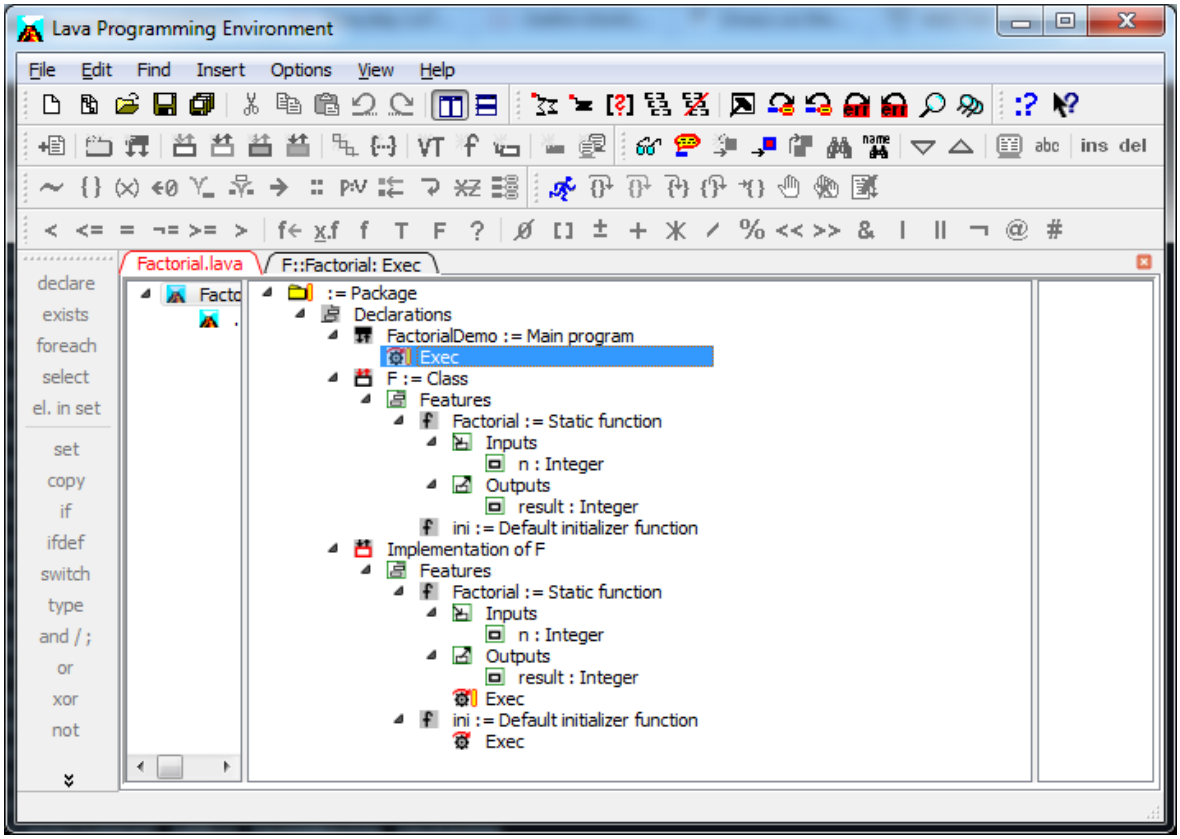
6.2 Lava

Lava [31] on kokeellinen olio–ohjelmointikieli, jota muokataan rakenteisesti. Sen ovat kehittäneet Klaus D. Günther sekä Irmtraut Günther. Lavan lähdekoodi on avoimesti saatavilla [33]. Ohjelmointi tapahtuu rakenteisessa ympäristössä nimeltä *LavaPE* (*Lava Programming Environment*), joka tukee mm. koodin muokkaamista sekä debuggaamista.

6.2.1 Ohjelmointi Lavalla

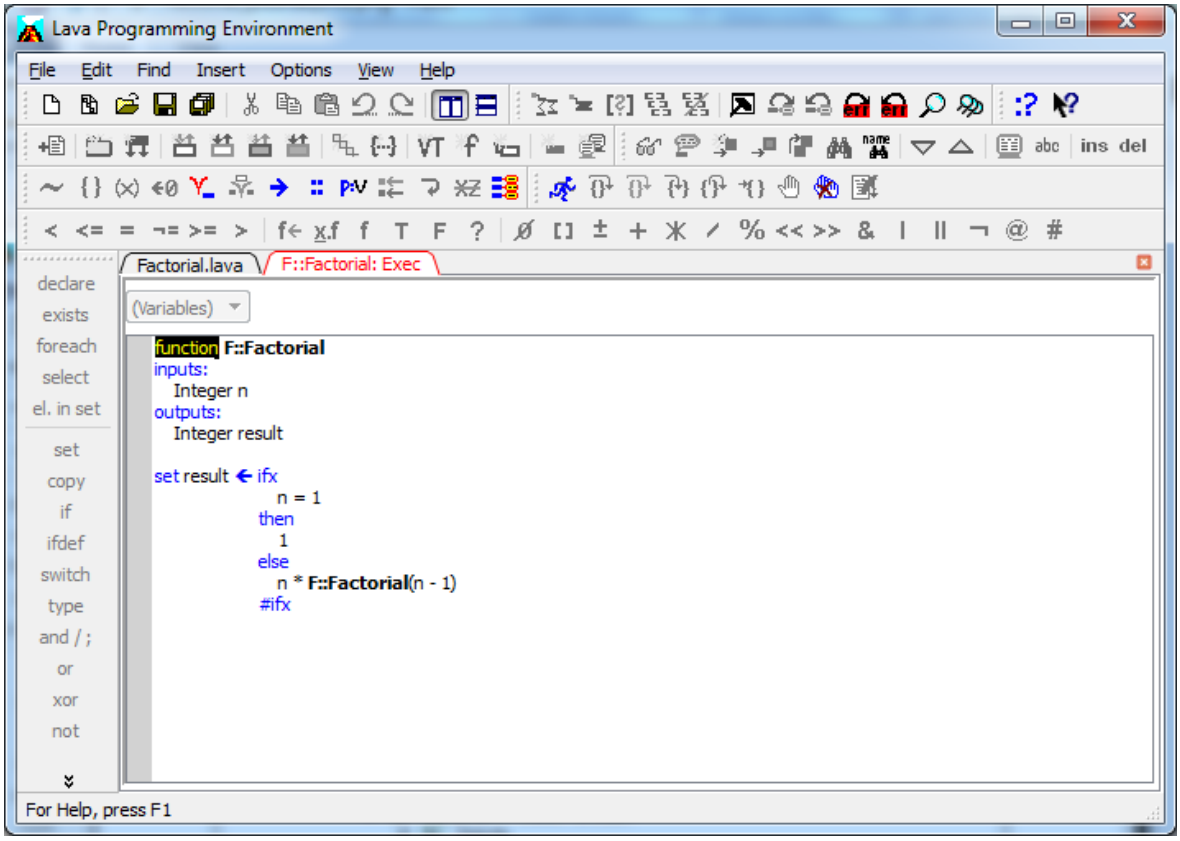
LavaPE:ssä koodin editointi on jaettu kahteen päänäkymään:

- *puunäkymä* (*declaration tree view*, kuva 6.8)
- *suoritusnäkymään* (*exec view*, kuva 6.9)



Kuva 6.8: Lavan puunäkymä

Puunäkymässä esitetään yksittäinen *paketti* (*package*), joka on `.lava`-päätteisessä tiedostossa. Puunäkymä esittää korkean tason näkymän paketin sisältöön: mm. luokat sekä niiden sisältämät metodit ja attribuutit.



Kuva 6.9: Lavan suoritusnäkymä

Puunäkymässä voi avata (expand) tai supistaa (collapse) puun solmuja, niin että vain halutut osat ovat näkyvissä. Puuhun voi lisätä uusia rakenteita työkalurivin painikkeista. Vain sellaiset painikkeet ovat aktiivisia, joiden rakenteet voidaan sijoittaa valittuun puun solmuun. Esimerkiksi uuden metodin voi luoda vain, jos kursori on luokkasolmun alla olevan “Features”–solmun kohdalla.

Suoritusnäkymässä sen sijaan näytetään yksittäinen metodi. Suoritusnäkymän koodi muistuttaa ulkoasultaan paljon perinteisiä ohjelmointikieliä (kuva 6.9) vaikka onkin rakenteinen. Kuten puunäkymässäkin, koodia voi muokata tai uusia rakenteita lisätä työkalurivin painikkeista. Lisäksi ikkunan vasemmassa reunassa on lista rakenteista, joita koodiin voi lisätä. Suoritusnäkymässä lisättäviä rakenteita ovat mm. if–lause sekä silmukat. Eri rakenteille on lisäksi pikanäppäimet.

Paikoissa, joissa koodiin voi asettaa muuttujan, tulee arvo valita koodinäkymän yläreunassa näkyvästä listasta. Nimeä ei voi suoraan kirjoittaa, mikä poistaa virheen mahdollisuuden.

6.2.2 Yhteenveto

Lava on esimerkki siitä, millainen rakenteinen ohjelmointiympäristö voisi olla. Sen yhteen kieleen keskittyvä, jossain määrin perinteisiä kieliä muistuttava, editori tekee ympäristöstä helposti lähestyttävän, mutta toteutus jättää paljon toivomisen varaa. Tällaisenaan Lavaa ei voi pitää vakavasti otettavana ohjelmointityökaluna ja tekijät toteavatkin, ettei se kahden hengen projektina voi kilpailla kaupallisia tuotteita vastaan [31].

6.3 Alice

Alice [1] on ilmainen ympäristö ohjelmoinnin opetteluun Carnegie Mellon –yliopistosta. Se tarjoaa 3D–maailman, jonka graafisia olioita ohjelmoija pääsee käskyttämään, luodakseen esim. pelin tai animaation. Tällä tavoin Alice opettaa olio–ohjelmoinnin perusteita. Alice luotiin ohjelmoinnin johdantokursseille auttamaan oppilaita, joilla on heikommat lähtötiedot. Alice auttaa ohjelmoinnin opettelussa mm. visualisoimalla ohjelman suoritusta. [7]

Alicen ensimmäinen versio [7] perustuu tekstimuotoiseen Python–ohjelmointikieleen [57]. Alice 2:ssa on rakenteinen editori, jossa ohjelmointilauseet ovat hiirellä raahattavia palikoita. Tällä vähennetään monimutkaisuutta, jotta oppilaan voisivat keskittyä syntaksin oikeinsaamisen sijasta mm. olio–ohjelmoinnin käsitteisiin [8].

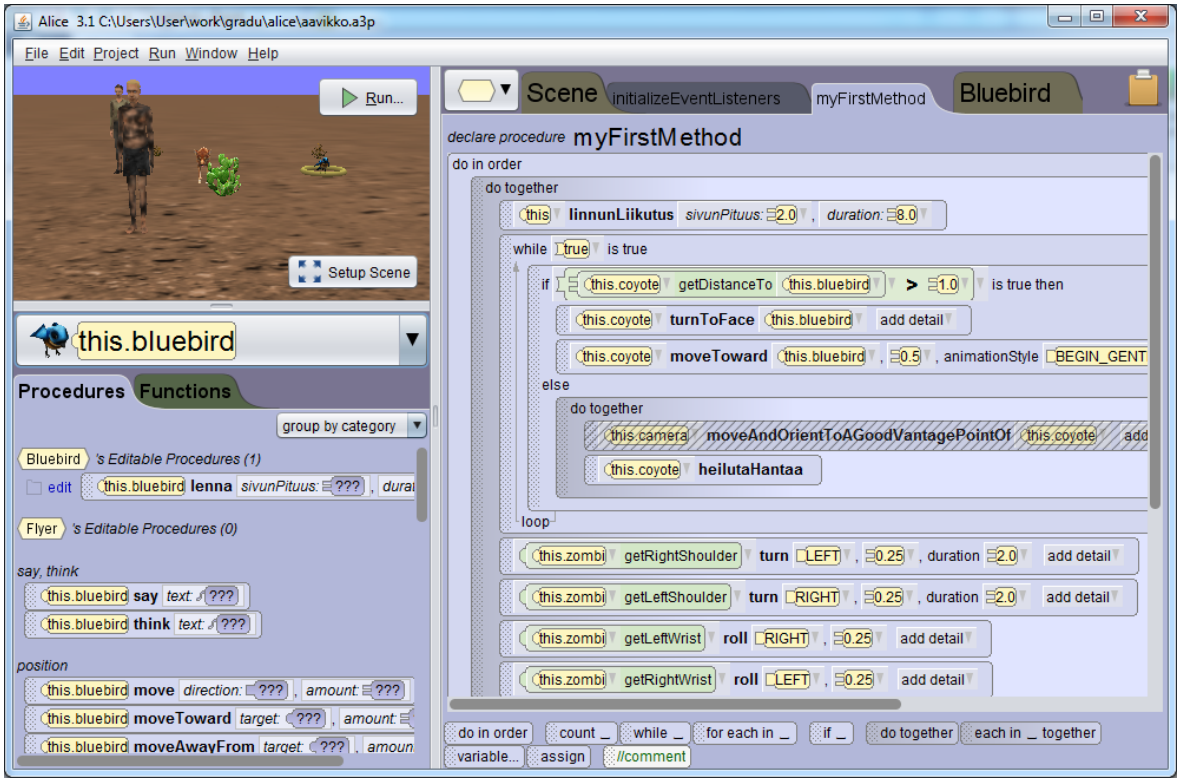
Rakenteisessa editorissa on ollut hankaluutena, että siirtyminen Alicesta "tavallisiin" kieliin (kuten Java tai C++) on ollut monille oppilaille hankalaa [56]. Lisäksi, koska oppilaitoksilla ei usein ole resursseja erillisen ohjelmoinnin johdantokurssin järjestämiseen, on toivottu oppimateriaaleja, jotka sulauttavat paremmin Alicen tavalliseen Java–kielellä pidettävään ohjelmointikurssiin [10]. Alice 3 on kehitetty näitä tarpeita silmällä pitäen. Siinä on moodi Alice–koodin näyttämiseen Java–syntaksin kanssa, joka on tarkempi kuin Alice 2:ssa. Lisäksi Alice–projektin voi viedä Java–IDE:hen, niin että samaa koodia pääsee muokkaamaan Java–kielellä. Tässä työssä kokeillaan versiota 3. [10]

6.3.1 Ohjelmointi Alicella

Kun ohjelma avataan, valitaan ensimmäisenä valmiista vaihtoehdoista maailma, johon tullaan sijoittamaan olioita. Tämä vaikuttaa vain 3D–näkymän ulkonäköön. Sen jälkeen tullaan editorinäkymään (kuva 6.10), jossa on seuraavat osat:

- 3D–maailman näkymä
- koodieditoripaneeli
- metodipaneeli
- kontrollirakenteet

Näiden lisäksi 3D–maailman näkymässä olevasta painikkeesta pääsee 3D–maailman muokkausnäkymään. Tässä voi lisätä uusia olioita sekä muuttaa niiden ominaisuuksia, kuten paikkaa, kokoa ja väriä.



Kuva 6.10: Alicen editorinäkymä

Oletuksena koodinäkymässä on auki tyhjä metodi nimeltään *myFirstMethod*, johon voi alkaa kirjoittamaan omaa koodia, toisin sanoen sitä mitä käyttäjä haluaa 3D–maailmassa tapahtuvan. “Koodaaminen” tapahtuu raahaamalla hiirellä Alice–kielen lauseita pääasiassa joko kontrollirakenteiden paneelista tai metodipaneelista. Raahattavat lauseet ovat Alice–kielen “atomeita”, jotka eivät koostu mistään merkeistä. Niinpä syntaksivirheitä ei voi tulla. Lauseet voivat sisältää toisia lauseita, esimerkiksi if–lause sisältää ehtolauseen.

Alicen ohjelmointikieli tukee proseduraalista ohjelmointia sekä olio–ohjelmointia (tietyin rajoituksin [56]). Kontrollirakenteet ovat enimmäkseen esim. Javasta tuttuja: mm. while–silmukka, if–lause sekä muuttujaan sijoittaminen. Lisäksi Alicelle ominaista on rakenne *do together*, joka suorittaa kaikki sen sisällä olevat lauseet yhtäaikaa. Jotkut lauseet ottavat parametreja, jotka ovat tiettyä tyyppiä. Tyyppejä ovat mm. kokonaisluku, desimaaliluku, boolean tai olion tyyppi. Olioille voi kirjoittaa omia metodeja tai funktioita (funktio palauttaa arvon, metodi ei). Omat metodit näkyvät käyttöliittymässä omina välilehtinään.

6.3.2 Alicen käyttö opetuksessa

Tietotekniikan opiskelijoiden laskevat määrät ovat olleet viime vuosina huolenaiheena alan opetuksessa. Schwartz, Stagner ja Morrison [62] esittävät kolme avainasiaa, joihin ohjelmoinnin oppimisympäristöjen tulisi vastata:

- Teknisten esteiden vähentäminen
- Kiinnostuksen tekniikkaa kohtaan hyödyntäminen
- Luovan itseilmaisun mahdollistaminen

Tämänkaltaisista lähtökohdista on kehitetty Alice. Teknisiä esteitä vähentää sen rakenteinen "drag and drop" -editori, joka poistaa syntaksin opettelun vaikeudet [8]. Se ei edes anna kirjoittaa koodia, jonka rakenteessa on puutteita. Esimerkiksi if–lauseeseen pitää syöttää boolean–tyyppinen ehtolauseke, ennen kuin ohjelmaa voi ajaa. Lisäksi ohjelman visualisoinnin on tarkoitus helpottaa kokonaisuuden hahmottamista [7].

Alice hyödyntää kiinnostuksen tekniikkaa kohtaan. Ala kiinnostaa oppilaita tavallisesti videopelien, animaatioiden ja multimediasovellusten kautta, joten tämänkaltaisten sovellusten luominen jo opintojen alkuvaiheessa vastaa oppilaiden odotuksiin [9]. Tarinankerronta mahdollistaa luonnollisesti luovan itseilmaisun.

Alicen käytöstä uusien tietotekniikan opiskelijoiden johdantokurssilla on saatu hyviä kokemuksia [48]. Riskiryhmään (vähäinen kokemus tietotekniikasta sekä matematiikasta) kuuluvien opiskelijoiden tulokset ohjelmointikursseilla olivat parempia, kiinnostus alaa kohtaan pysyi hyvänä ja he pysyivät paremmin opinto–ohjelmassa verrattuna verrokkiryhmään joka ei käyttänyt Alicea.

Powers, Ecott sekä Hirshfield [56] ovat havainneet Alicen käytössä paitsi hyviä, myös valitettavan paljon huonoja puolia. Toisaalta graafinen 3D–ympäristö tekee virheistä ennemminkin hauskoja kuin turhauttavia, toisaalta opiskelijoiden huomio saattaa karata animaatioiden hieromiseen ohjelmoinnin käsitteiden sijasta. Alice paransi heikompien oppilaiden itseluottamusta ohjelmointikykyihinsä, mutta itseluottamus ropisi sen jälkeen kun siirryttiin tekstipohjaisiin ohjelmointikieliin. Koska olioilla on graafinen esitys, on olio–ohjelmoinnin ymmärtäminen intuitiivista. Toisaalta, Alicen oliomallin puutteet antavat jopa harhaanjohtavat lähtötiedot tavanomaisia oliokieliä ajatellen.

6.4 Scratch

Scratch on Massachusetts Institute of Technology –yliopistossa (MIT) kehitetty ohjelmointikieli, jolla voi luoda interaktiivista taidetta, tarinoita, simulaatioita ja pelejä ja jakaa tuotoksensa muiden kanssa [63]. Scratchiä on käytetty paitsi nuorten ohjelmointikursseilla, myös yliopiston ohjelmoinnin johdantokursseilla [37]. Scratch ja Alice muistuttavat paljon toisiaan, joskin Scratchillä tehtävät animaatiot ovat kaksiulotteisia.

Scratchin tekijöiden tavoitteena on tehdä ohjelmoinnista vetoavaa kaikille ihmisille, myös heille, jotka eivät miellä itseään ohjelmoijiksi. Scratch pyrkii tähän madaltamalla ohjelmoinnin aloituskynnystä sekä mahdollistamalla monia erityyppisiä projekteja. Näiden tavoitteiden pohjalta on muodostunut kolme ydinperiaatetta: tehdä siitä värkkäiltävämpi (“tinkerable”), tarkoituksellisempi sekä sosiaalisempi kuin muista ohjelmointiympäristöistä. [59]

Scratch muistuttaa Alicea paljon siinä, miten ohjelmointi tapahtuu: Ohjelmointilausekkeet ovat hiirellä raahattavia palikoita (kuva 6.11). Palikat sopivat toisiinsa kuten Lego–palat, mikä mahdollistaa luovan kokeilemisen. Lisäksi palikoiden muodot sopivat toisiinsa niin, että siinä on syntaksin kannalta järkeä. Esimerkiksi kontrollirakenteet ovat C–muotoisia, mikä vihjaa, että niiden sisälle voi laittaa muita paloja. [59]

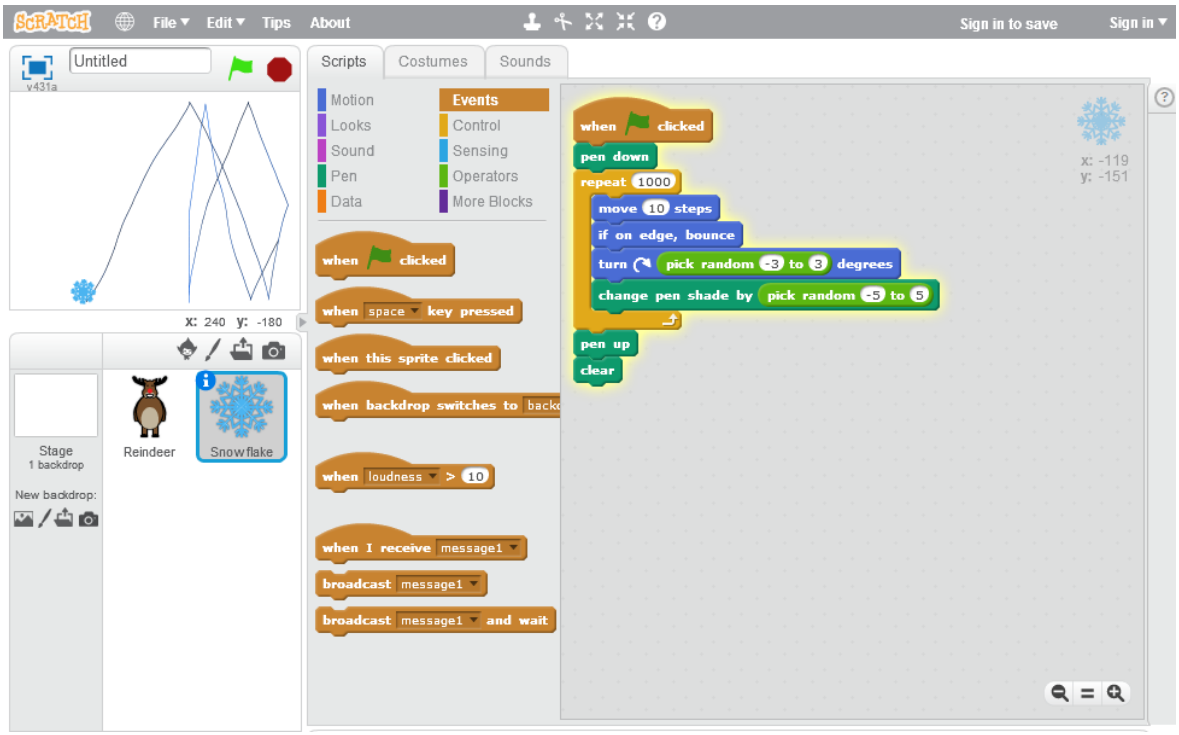
Kuva 6.11: Schratchin koodilohkoja

6.4.1 Yhteisöllisyys

Scratchin ympärille on kehittynyt luova yhteisö osoitteessa osoitteessa `http://scratch.mit.edu/` [43]. Sen jäsenet ovat enimmäkseen 8–16 -vuotiaita. Sieltä löytyy laaja kirjo projekteja, mm. pelejä, animaatioita, syntymäpäiväkortteja, simulaatioita, interaktiivisia oppaita ja niin edelleen. Mieleisten projektien kanssa tekeminen lisää motivaatiota oppia uutta. Scratchin tekijät ovat halunneet luoda Scratch–yhteisöön avoimen lähdekoodin kulttuurin, jossa tekijät jakavat tuotoksiaan keskenään ja voivat tehdä uusia versioita toisten ohjelmista. [59]

6.4.2 Editori

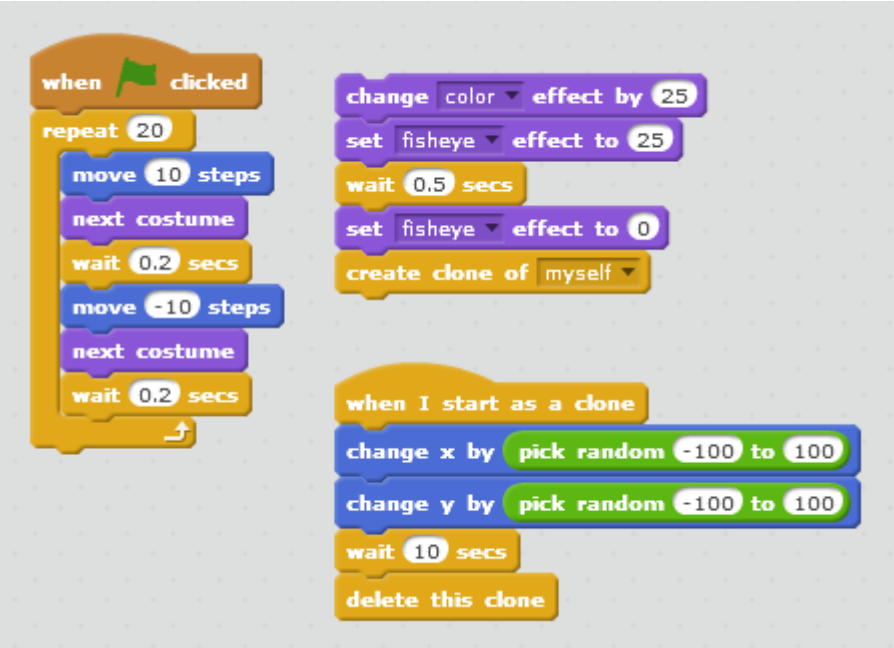
Scratchin nettisivun tarjoama editorinäkymä näkyy kuvassa 6.12. Kuten todettua, Scratchin editori muistuttaa paljon Alicen vastaavaa. Vasemmassa ylänurkassa on 2D–näkymä, johon voi sijoittaa graafisia olioita. Mukana olevat oliot näkyvät 2D–näkymän alapuolella. Keskellä sivua on valikko, jossa tarjolla olevat ohjelmointipalikat ovat ryhmiteltynä eri kategorioihin. Oikealla on koodinäkymä valitun olion koodilohkoille.



Kuva 6.12: Scratchin editorinäkymä

Palikoita yhteen liittämällä voi muodostaa koodilohkoja (Scratchissa käytetty

termi sille on *pino*). Palikat napsahtavat toisiinsa kiinni kun ne tuo vierekkäin, jos niiden muoto sopii yhteen. Kuvassa 6.13 on kolme koodilohkoa. Vasemmanpuoleisin suoritetaan käynnistettäessä ohjelma, alin kun olio luo itsestään kloonin. Oikealla ylhäällä oleva lohko on irrallinen, mutta sen voi suorittaa yksinkertaisesti klikkaamalla sitä. Ohjelman lopputulos, kun oikean yläkulman lohkoa on klikattu muutaman kerran, näkyy kuvassa 6.14.



Kuva 6.13: Monta koodilohkoa Scratchissä

6.4.3 Ohjelmointikieli

Scratch–kieltä voisi kuvailla proseduraaliseksi, tosin sillä lisäyksellä, että koodilohkot kuuluvat aina jollekin oliolle. Myös 2D–näkymän tausta on yksi olio. Oikean reunan koodinäkymässä näkyy vain valitun olion koodilohkot.

Olioille voi kirjoittaa proseduureja. Proseduurit voivat ottaa parametreja, mutta eivät voi palauttaa arvoja. Tämä on harmi, jos haluaisi tehdä vaikkapa oman operaattorin lukujen vertailua varten. Scratchin oma operaattorivalikoima kun on hiukan suppea, esimerkiksi *suurempi tai yhtäsuuri*– sekä *pienempi tai yhtäsuuri* –operaattoreita ei ole. Toki ohjelmoinnin ensiaskeleita ottava henkilö tuskin osaa kaivata funktioiden paluuarvoja.

Tapahtumat (*events*) ovat osa Scratchin työkalupakkia. Tapahtumia tulee mm. ohjelman käynnistyksestä, näppäimen painamisesta, jonkin olion klikkaamisesta tai tietyn ajan kulumisesta. Tapahtumat ovat tietyn muotoisia palikoita, joiden muoto

Kuva 6.14: Kuva Scratch–ohjelmasta

vihjaa, että niistä voi aloittaa uuden koodilohkon. Yksi tapahtuma on myös viestin vastaanottaminen. Oliot voivat lähettää nimettyjä viestejä ja vastaanottaa niitä.

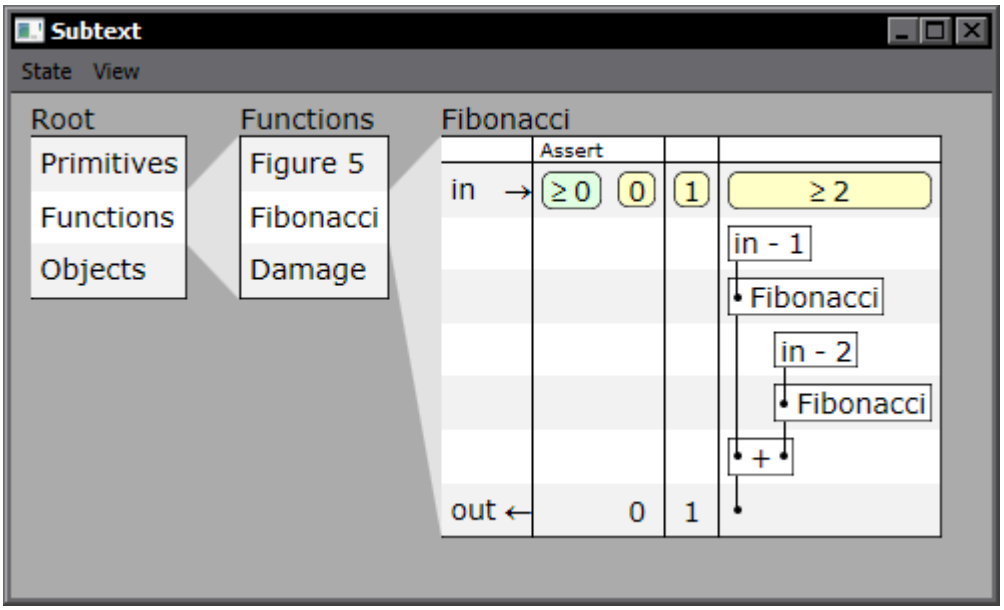
Scratch on suunniteltu mahdollisimman interaktiiviseksi. Jopa muutosten tekeminen koodilohkoon sen ollessa ajossa onnistuu, joten on helppoa kokeilla uusia ideoita inkrementaalisesti ja iteratiivisesti. [59]. Samoin rinnakkainen ohjelman suoritus on tehty helpoksi. Useamman lohkon voi käynnistää yhtäaikaa joko ohjelmallisesti tai lohkoja klikkaamalla. Kuvan 6.13 alimman lohkon, joka ohjaa klooni–olion eri paikkaan ja lopulta tuhoaa sen, ei perusidealtaan paljoa poikkea vaikkapa prosessin luomisesta!

6.5 Subtext

Subtext on Jonathan Edwardsin kehittämä rakenteinen ohjelmointiympäristö, jonka tavoitteena on tehdä ohjelmoinnista mahdollisimman yksinkertaista ja helppoa. Subtextista on useita eri versioita, jotka kukin esittelevät erilaisia ideoita. Versiot ovat luonteeltaan hyvin kokeellisia. Kotisivulla niistä on erilaisia esityksiä ja papereita, mutta ei latauslinkkiä. [66]

Tässä työssä tarkastellaan Subtextin versiota 2. Se on ladattavissa Windows–koneille osoitteesta `http://subtextual.org/subtext2.zip`.

Subtext 2:n erityispiirre on konditionaalien esittäminen kaksiulotteisessa taulussa ns. *skemaattisina tauluina* (*schematic tables*). Taulun riveillä näkyy ohjelman eteneminen ja sarakkeilla kontrollivuon vaihtoehtoiset reitit. Kuvassa 6.15 on Subtextilla tehty fibonacci–funktio. Funktiolla on parametri *in*, jonka perusteella valitaan ylhäältä alas kulkeva kontrollivuo. Funktion lopputulos näkyy alimmalla rivillä. Ohjelma pitää huolen siitä, että parametrin koko arvoalue tulee käsiteltyä (huomaa vihreällä näkyvä esiehto). Toisaalta myöskään parametrin arvoissa eri ehdoissa ei pääse tulemaan päällekkäisyyksiä, kuten vaikkapa if–lauseita käyttämällä saattaisi käydä.



Kuva 6.15: Fibonacci–funktio subtextissa

Skemaattisilla tauluilla saavutetaan Edwarsin mukaan seuraavia etuja verrattuna perinteisiin konditionaalirakenteisiin [13]:

- erilaisten konditionaalirakenteiden yhtenäistäminen
- ehtojen kattavuus konditionaaleissa
- päällekäisyyksien välttäminen ehdoissa
- laskenta (computation) ja logiikka pidetään erillään toisistaan
- selkeämmin hahmotettava logiikka

6.6 JetBrains Meta Programming System

JetBrains–yhtiön kehittämä *Meta Programming System* (*MPS*) [49] on ohjelmointiympäristö kielisuuntautuneeseen ohjelmointiin, jossa ohjelmoija voi kohtuullisella vaivalla luoda uusia sovellussuuntautuneita kieliä, jotka toimivat hyvin yhteen. Uutta kieltä ei tarvitse luoda tyhjästä, sillä ympäristön mukana tulee kaikki Java–kielen rakenteet sisältävä *BaseLanguage*. Uusi kieli voi olla tästä laajennos tai käyttää osia siitä.

MPS on rakenteinen, vaikkakin muistuttaa paljon perinteistä tekstieditoria. Toimintaperiaate on tekstieditoriin nähden erilainen, esimerkiksi avaavaa sulkua ei voi kirjoittaa ilman sulkevaa sulkua. Työkalussa on IDE:istä tuttuja täydennysominaisuuksia lähes joka paikassa, myöskin omissa kielissä. Näin koodin muokkaaminen on varsin tehokasta. Graafisia näkymiä koodiin ei toistaiseksi (versiossa 2.5) tueta.

Uuden sovellussuuntautuneen kielen luomisessa on karkeasti ottaen kolme osa–aluetta:

- kielen rakenteen määrittely
- editorin määrittely
- generaattorin määrittely

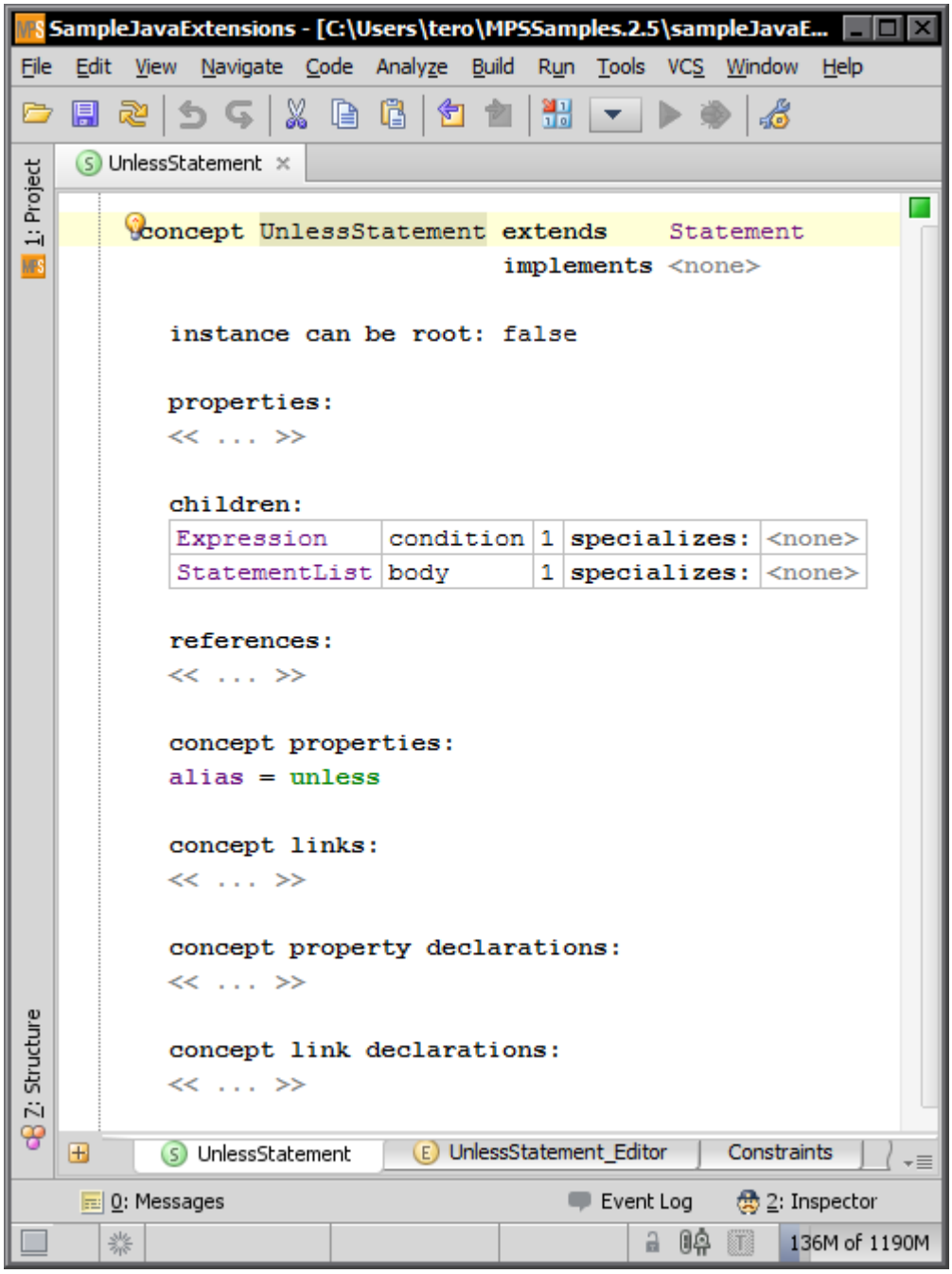
Näiden kaikkien määrittelyyn MPS:ssä on omat kielensä, jotka toimivat ihan samoilla periaatteilla kun muutkin kielet. Rakenne määritellään rakenteen kuvauskielellä, editori editorin kuvauskielellä ja generaattori generaattorin kuvauskielellä. Rakenne tarkoittaa kielen abstraktia syntaksia. Editori määrittää miten kieli esitetään käyttäjälle. Generaattori on muunnos kielestä jollekin kohdekielelle, esimerkiksi BaseLanguagelle, joka voidaan edelleen kääntää Javaksi.

6.6.1 Kielen laajentaminen

Rakenteisen editorin vahvuudet tulevat esille, kun olemassaolevaa kieltä laajennetaan tai useampi kieli halutaan saada pelaamaan yhteen. Siinä missä tekstipohjaisissa ohjelmointikielissä kääntäjän tekstistä parsima jäsennyspuu ei näy kääntäjästä ulospäin, rakenteisessa editorissa se on koko ajan näkyvillä.

MPS:n mukana tulee esimerkki, jossa MPS:n Java–kieleen on tehty laajennos unless–lauseelle (eli käänteinen if). Kuvassa 6.16 on MPS:n editorinäkymä unless–lauseen rakenteelle (tämä on siis ohjelman jäsennyspuun solmu). Huomattavaa on alussa oleva `extends Statement`, mikä tarkoittaa että unless on lause siinä missä

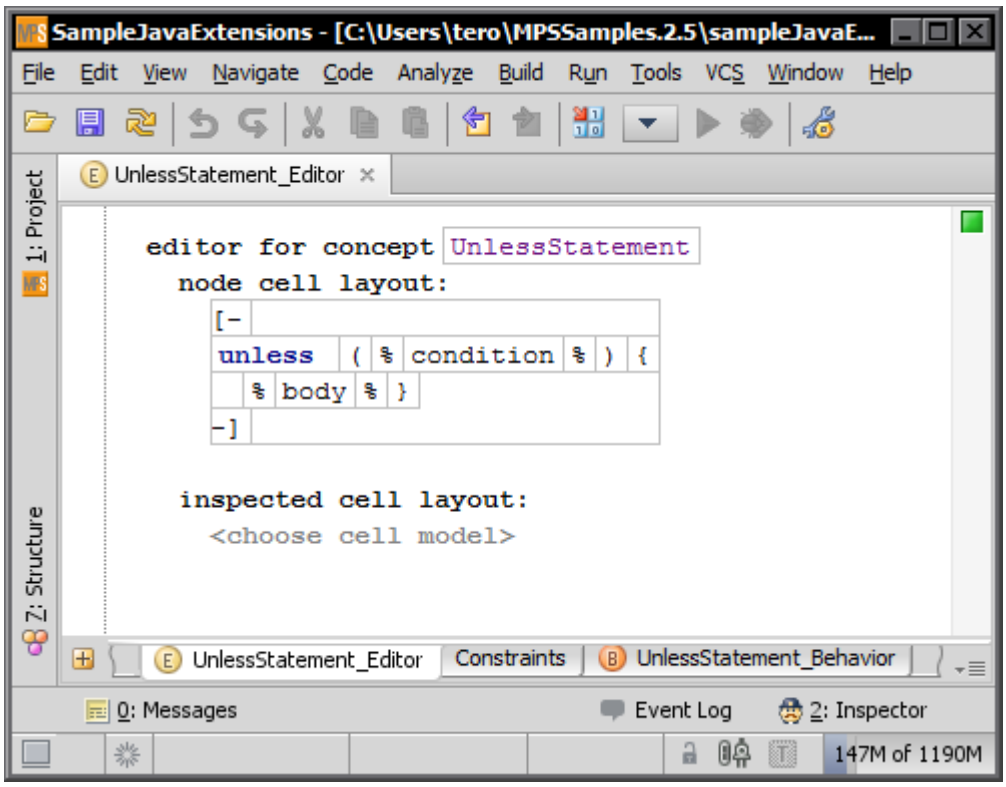
`if` tai `while`. `Children`-kohdassa määritellään, että unless–lause koostuu ehdosta (condition) sekä rungosta (body). Lisäksi pitää määritellä mm. että ehdon täytyy olla boolean–tyyppiä, mutta pääpiirteissään määrittely on tässä.



Kuva 6.16: Unless–lauseen määrittely MPS:ssä

Unless–lauseen editori on kuvassa 6.17. Tässä kuvataan, miten unless–lauseke näkyy käyttäjälle. Esityksestä nähdään mm. että lause alkaa sanalla `unless`, jonka

jälkeen tulee ehto suluissa ja lopuksi runko aaltosulkeiden välissä.



Kuva 6.17: Unless–lauseen editori MPS:ssä

Kuvassa 6.18 on unless–lauseen generaattorissa oleva sääntö, joka sanoo että `UnlessStatement`-solmu kuvataan if–lauseeksi, jonka ehdolle tehdään negaatio.

```
reduction rules:
  [concept    UnlessStatement]  --> <T if (!($COPY_SRC$[true])) {     T>
  [inheritors false                       $COPY_SRC$[<no statements>]
  [condition  <always>                  }
```

Kuva 6.18: Unless–lauseen generaattorisääntö MPS:ssä

6.6.2 Rakenteinen editori

MPS:n rakenteinen editori muistuttaa ulkonäöltään ja perustoiminnoiltaan tekstieditoria, vaikkakin on pohjimmiltaan erilainen. Liikkuminen solmujen välillä tai yksittäisen solmun tekstin sisällä tapahtuu nuolinäppäimillä. Tab ja shift–tab liikkuvat kokonaisten solmujen välillä. Enter asettaa nykyisen solmun perään uuden solmun

ja insert lisää uuden solmun ennen nykyistä solmua (usein nämä voidaan mieltää uuden rivin lisäämisenä).

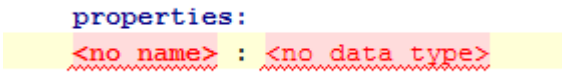
Koodi tarjoaa monista IDE:istä tuttuja toimintoja, kuten koodin täydentämisen, symbolin määrittelyyn hyppäämisen sekä symboleiden uudelleennimeämisen. Nämä toiminnot toimivat paitsi MPS:n mukana tuleville kielille, myös itse määritellyille kielille.

Vaikka editori muistuttaa ulkoisesti tekstieditoria, siinä muokataan kuitenkin ohjelman jäsennyspuuta eikä tekstitiedostoa. Niinpä ruudulla näkyvästä tekstistä ei voi valita mielivaltaista osaa kuten tekstieditorissa, vaan ainoastaan kokonaisia jäsennyspuun solmuja alisolmuineen. Näppäinkomennolla *ctrl+nuoli ylöspäin* voi nopeasti valita jäsennyspuusta yhtä tasoa ylempänä olevan kokonaisuuden. Esimerkiksi jos valittuna on silmukan sisällä oleva lause, tulee valituksi koko silmukka. Vastaavasti *ctrl+nuoli alaspäin* valitsee yhtä tasoa pienemmän kokonaisuuden.

Joissakin paikoissa, kielestä riippuen, näytetään harmaalla tekstillä paikat, joihin voi kirjoittaa uutta koodia. Painamalla tällaisessa paikassa enter tai insert editori muodostaa uuden siihen sopivan solmun. Kuvassa 6.19 on esimerkki tästä. Kun tekstin ”properties:” alla olevan solmun kohdalla painaa enter, tulee harmaalla olevan solmun tilalle yksittäisen property–tyyppisen solmun kentät, kuten on kuvassa 6.20.

```
concept IfStatement extends    AbstractCommand
                    implements <none>

  instance can be root: false
  alias: if
  short description: <no short description>

  properties:
  << ... >>

  children:
  condition   : LogicalExpression[1]
  trueBranch  : CommandList[1]
  falseBranch : CommandList[1]

  references:
  << ... >>
```

Kuva 6.19: Laajennettava solmu MPS:ssä



Kuva 6.20: Laajennettu solmu MPS:ssä

Joidenkin kielten yhteydessä kaikki kielen esittämä informaatio ei näy editorinäkymässä, sillä kaikkien optioiden näkyminen kerralla vaikeuttaisi olennaisten asioiden löytämistä. Tätä varten MPS:ssä on *inspector*–näkymä, jonka saa näkyviin kursorin alla olevalle objektille. Esimerkkikuvassa 6.21 on editorin määrittely erään sovellussuuntautuneen kielen if–lauseelle. Päänäkymässä (ikkunan ylempi puolisko) on varsinainen editointinäkymä ja inspectorissa (alempi puolisko) voi editoida yksityiskohtaisia tietoja. Näin päänäkymä pysyy tiiviinä ja selkeänä, mutta tarkemmat määrittelyt ovat helposti löydettävissä.

Joissain paikoin rakenteinen editori aiheuttaa hämmennystä tekstieditoriin tottuneelle. Esimerkiksi Java–kielen taulukon `String[] args` lisääminen metodin parametriksi ei onnistu kirjoittamalla ensin `String` ja sen jälkeen `[]`, vaan kirjoittamalla ensin `String args` ja lisäämällä hakasulut vasta sitten. Syynä tähän on, että metodin parametrien määrittelyssä pelkkä `String[]` ei ole järkevä rakenne, vaan parametrilistassa sallitaan ainoastaan parametrin määrittely, johon kuuluu sekä tyyppi että parametrin nimi [51].

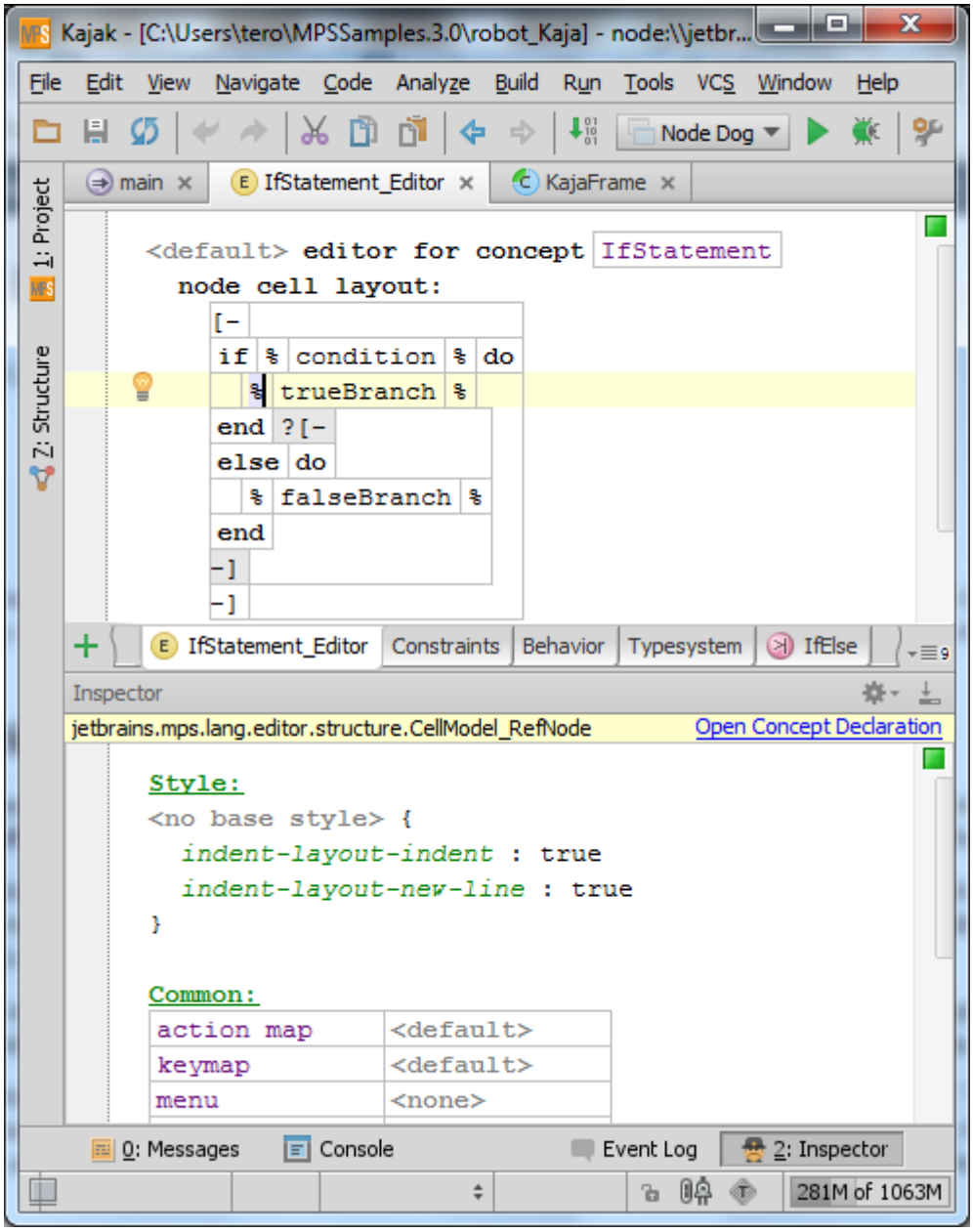
6.6.3 Yksikkötestit

MPS tukee yksikkötestaamista. Rakenteiseen ja kielisuuntautuneeseen editoriin tämä sopii mainiosti. MPS:n yksikkötestit pohjautuvat Java–kirjastoon JUnit [24], joten sitä käyttäneille testien kirjoittaminen on varsin tuttua.

Kuvassa 6.22 on yksinkertainen yksikkötesti, jossa testataan staattista metodia `max`. Kuten JUnitissa, testeille voi lisätä kaikille testeille yhteisiä jäsenmuuttujia, metodin joka ajetaan ennen jokaista testiä sekä metodin joka ajetaan jokaisen testin lopuksi. Kuvasta näkyy laajennuspaikat, joihin nämä voidaan lisätä (ennen testiä ajettava metodi on kuvassa laajennettu) ja lisääminen tapahtuu yksinkertaisesti painamalla enteriä kunkin paikan kohdalla — ei siis tarvitse muistaa oikeaa nimeämistä, annotaatiota tai muuta käytäntöä jolla metodit merkitään.

Itse testien lisääminen käy samaan tapaan kuin alustusten lisääminen. Kooditäydennys osaa ehdottaa erilaisia assert–lauseita. Testit voidaan ajaa MPS:stä käsin.

MPS tukee myös omien kielten eri elementtien testaamista niitä varten suunni-



Kuva 6.21: Inspector–näkymä MPS:ssä

telluilla kielillään [52].

```
test case NumbersTest extends <none> {
  <<members>>

  beforeTest {
    <no statements>
  }

  <<after test>>

  test max_noNumbersGiven_returnsIntMinValue {
    assert Integer.MIN_VALUE equals Numbers.max(new int[]{});
  }

  test max_oneNumberGiven_returnsGivenNumber {
    assert 4 equals Numbers.max(new int[]{4});
  }

  test max_twoNumbersGiven_returnsBiggerNumber {
    assert 1 equals Numbers.max(new int[]{0, 1});
  }
}
```

Kuva 6.22: Yksikkötesti MPS:ssä

6.7 MetaEdit+

MetaEdit+ on Metacase–yhtiön [42] ohjelmointiympäristö sovellussuuntautuneiden mallien luomiseen ja niillä ohjelmointiin. Ympäristöön kuuluu kaksi työkalua: *MetaEdit+ Workbench* mallinnuskielten luomiseen sekä *Metaedit+ Modeler* mallien käyttämiseen. Metacase kertoo, että yksinkertaisten mallien luominen onnistuu tunneissa ja että mallien käyttämisellä saavutetaan jopa kymmenkertainen nopeus ohjelmistonkehityksessä.

MetaEdit+:n mallinnuskielet ovat graafisia ja niiden luominen onnistuu kokonaan käyttöliittymästä käsin, ilman ohjelmointia. Mallin luomisessa on pääpiirteissään kolme vaihetta: sovellusalueen käsitteiden määrittely, sääntöjen määrittely sekä käsitteiden esittäminen graafisesti. Esittämisessä käytettävät graafiset elementit voivat olla esimerkiksi tekstilaatikkoja tai nuolia. Mallista saadaan ajettavaa koodia kirjoittamalla generaattori, joka luo mallin pohjalta koodia halutulle kohdekielelle (esimerkiksi C [26] tai Java).

6.7.1 Graafinen mallintaminen

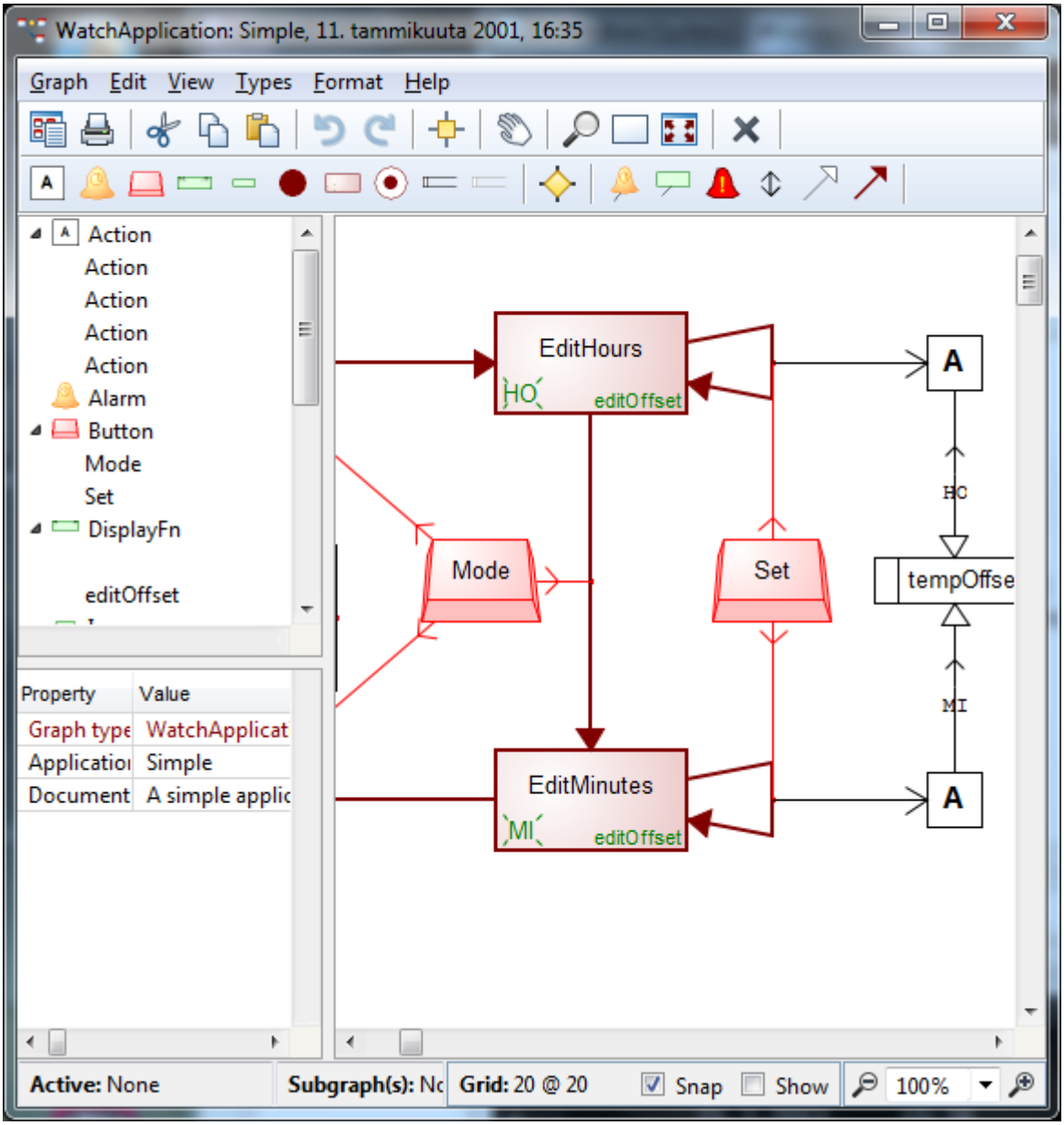
Mallinnuskielen käyttö muistuttaa vektorigrafiikkaohjelman käyttämistä. Käyttäjä luo sovellussuuntautuneen mallin asettamalla hiirellä paikalleen mallinnuskielen elementtejä, joita ovat mm. objektit sekä näiden väliset assosiaatiot. Assosiaatiot esitetään viivoina objektien välillä.

Yksi ohjelman mukana tulevista esimerkeistä esittelee rannekellojen ohjelmiston mallinnuskielen. Kuvassa 6.23 on mallinnusnäkymä. Siinä määritellään rannekellon toiminta tilakaaviona. Ikkunan yläreunassa, alemmalla työkalurivillä on valittavissa mallinnuskielen elementit, josta ne voidaan lisätä kaavioon. Vasemmalla näkyvät ylempänä kaaviossa olevat elementit sekä alempana valitun elementit ominaisuudet.

6.7.2 Mallinnuskielen luominen

Mallinnuskielen suunnittelija päättää sovellusalueen käsitteet, sekä sen, miten käsitteet esitetään graafisesti. Rannekelloesimerkissä käsitteitä ovat mm. tila sekä (fyysinen) painike. Suunnittelija on määritellyt näille graafisen esityksen. Tilasiirtymät esitetään nuolina tilaobjektien välillä. Assosiaatioihin voi liittyä enemmän kuin kaksi objektia. Esimerkiksi tilasiirtymiin voi liittyä myös painike, joka laukaisee tilasiirtymän. Painikeobjekti liittyy assosiaatioon erinäköisellä nuolella.

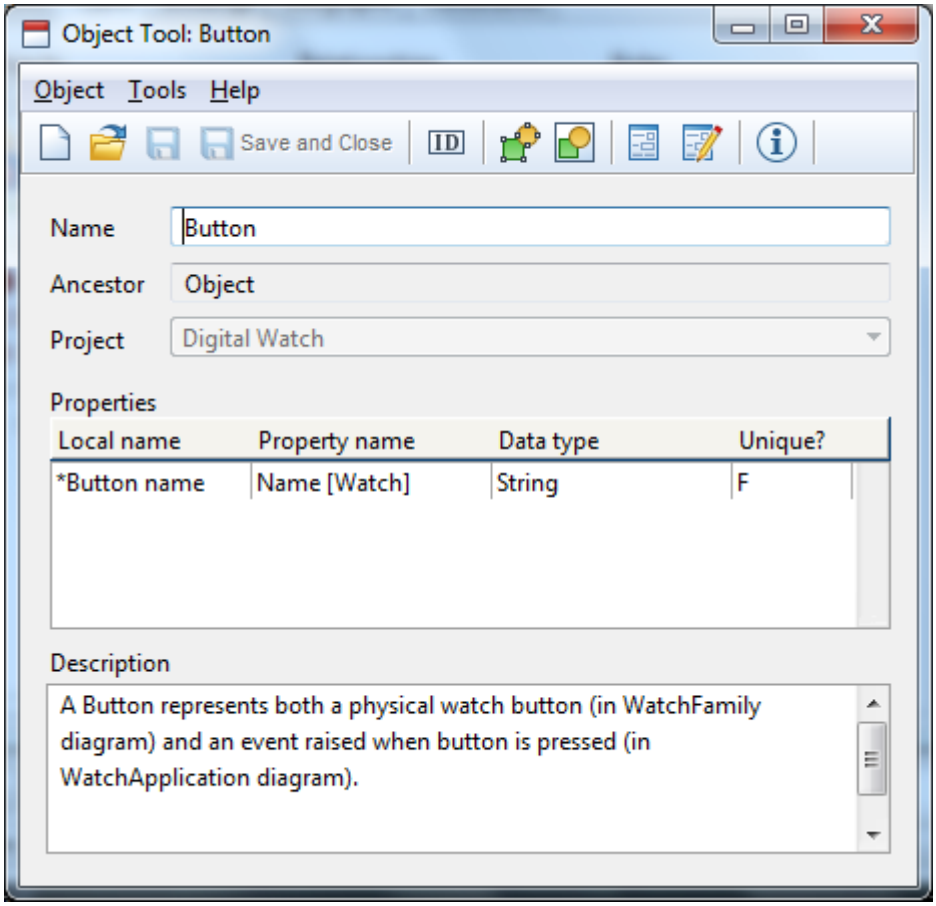
Mallinnuskielen elementit määritellään käyttöliittymän ikkunoiden kautta. Esi-



Kuva 6.23: MetaEdit+:n mallinnusnäkymä

merkiksi kuvassa 6.24 on painikeobjektin määrittely. Objektille voi määritellä ominaisuuksia, tässä nimi (”Button name”, joka näkyy otsikon Properties alla).

Painikkeelle on määritelty ulkoasu kuvan 6.25 symbolieditorissa. Kuvalle on määritelty, että siinä näkyy ominaisuuden ”Symbol name” teksti.



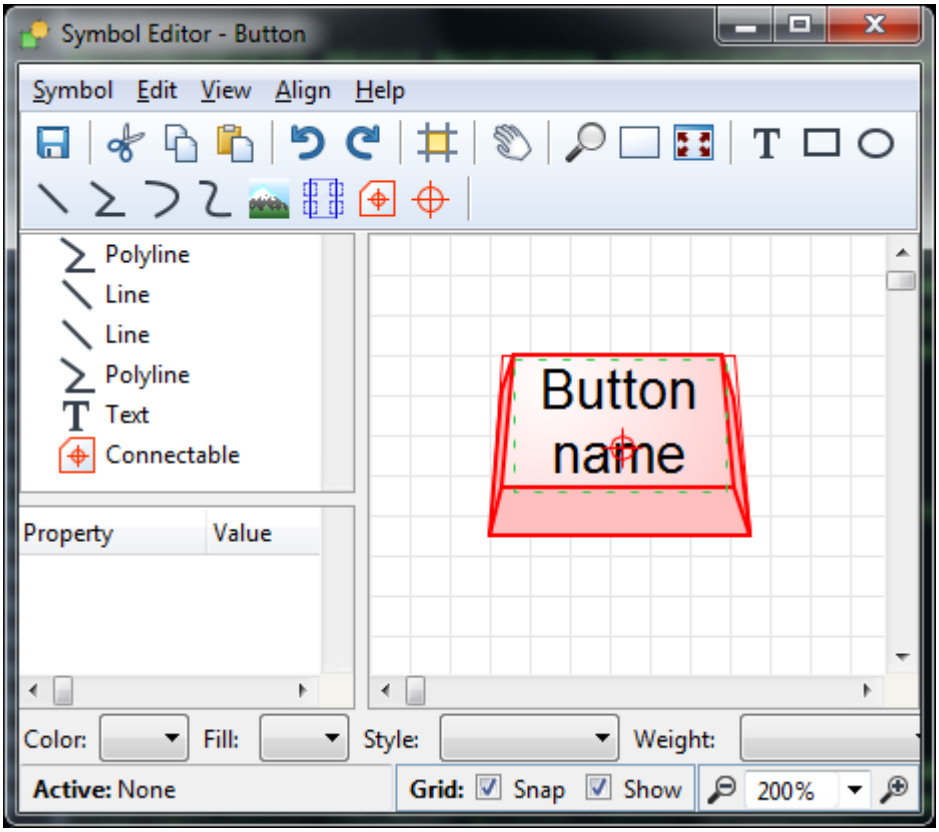
Kuva 6.24: MetaEdit+:n objektin määrittely

6.7.3 Koodin generointi malleista

Jotta malleilla voidaan tehdä jotakin hyödyllistä, täytyy kirjoittaa generaattori, joka muuttaa mallien sisältämän informaation halutulle kohdekielelle. Malleista saatava lopputulos voi olla esimerkiksi ajettava ohjelma tai dokumentti.

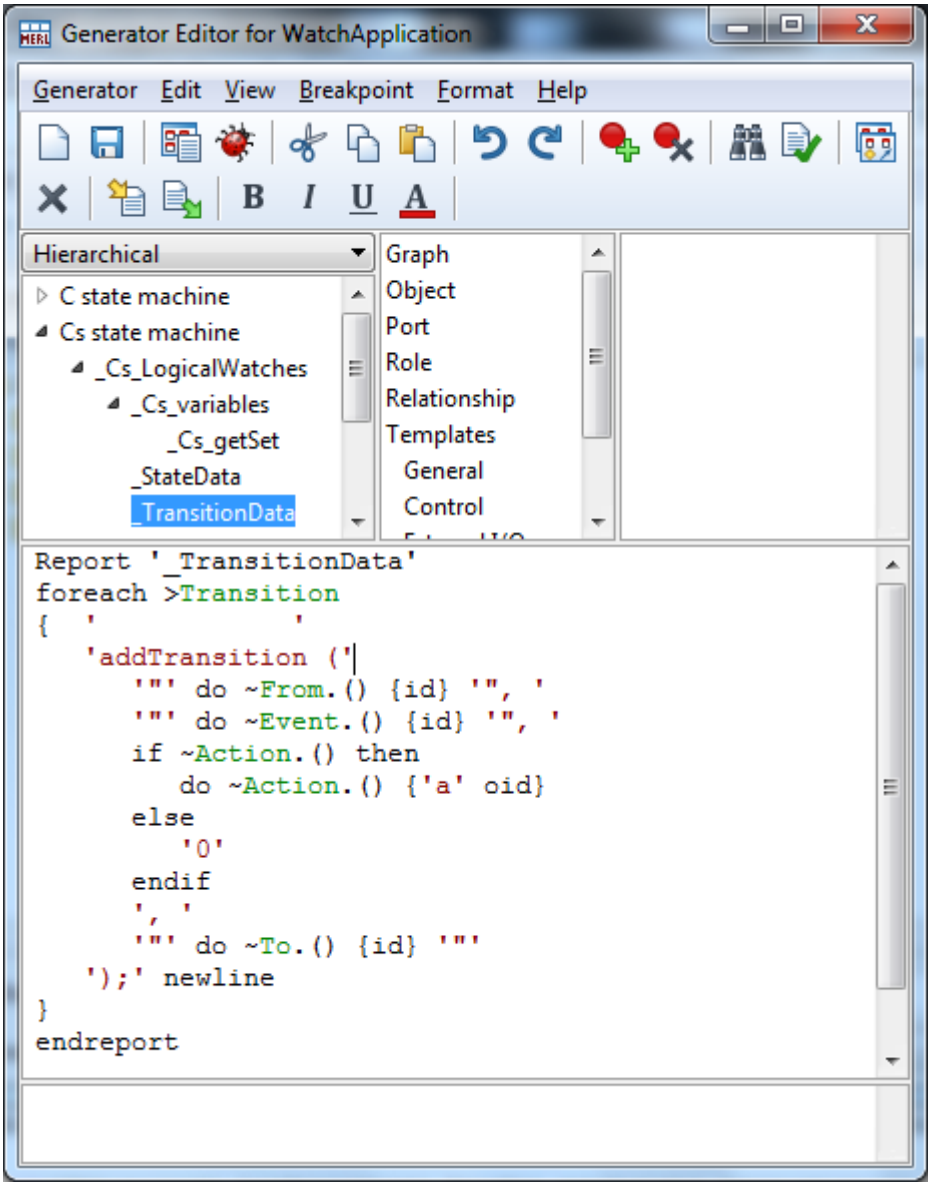
Generaattorit kirjoitetaan tarkoitusta varten luodulla kielellä nimeltään MERL. Kieli on tekstipohjainen. Se tarjoaa mm. lausekkeet mallin objektien läpikäymiseen (esimerkiksi kaikkien assosiaatioiden iterointi), muuttujat sekä ehto- ja silmukkalausekkeet. Generaattori voidaan jakaa pienempiin aligeneraattoreihin (*subreport*), joita voidaan kutsua generaattorista, samaan tapaan kuin aliohjelmia käytetään perinteisessä ohjelmoinnissa.

MERL tarjoaa varsin tiiviin syntaksin kaavioiden läpikäymiseen ja halutun tekstin tulostamiseen kohdetiedostoon. Kieli näyttää aluksi kryptiseltä, mutta ohjekirjan lukemisen jälkeen se on varsin selkeä ja johdonmukainen. Esimerkiksi objektien välisiä assosiaatioita merkitään aina >-merkillä. Niinpä rannekelloesimerkissä



Kuva 6.25: MetaEdit+:n symbolieditori

kaikkien Transition–tyyppisten assosiaatioiden läpikäyminen onnistuu syntaksilla `foreach >Transition`, kuten on kuvassa 6.26.



Kuva 6.26: MetaEdit+:n generaattorin editori

7 Tutkimus

Rakenteisen ja tekstimuotoisen ohjelmoinnin vertailemiseksi olisi hyödyllistä tehdä tutkimus koehenkilöillä. Näin on tehty esimerkiksi Miika Mäen pro gradu –tutkielmassa [47], jossa koehenkilöt vertailivat web–sovelluksen tekemistä sekä Spring Roo –nimisellä työkalulla että vapaavalintaisella metodilla. Ajan ja tutkimushenkilöiden puutteen takia vastaavaa vertailua ei tähän tutkielmaan tehdä, mutta esitetään kuitenkin hahmotelma siitä, millaisia tutkimuksia aiheesta voisi tehdä.

Jotta tulokset olisivat vertailukelpoisia, yhteen tutkimukseen valittaisiin editorit yhdestä samankaltaisten editorien ryhmästä. Ryhmät esiteltiin luvussa 6 ja ne ovat:

- yleiskäyttöiset ohjelmointityökalut
- aloittelijoille suunnatut editorit
- kielisuuntautuneen ohjelmoinnin työkalut

Yksittäisessä tutkimuksessa olisi vertailtavina sekä yksi rakenteinen editori, joka on sama kaikille koehenkilöille, että yksi tekstimuotoinen editori, joka on tutkimuksesta riippuen joko kaikille yhteinen tai jokaisen koehenkilön itse valitsema. Tällaisella tutkimusasetelmalla pyritään selvittämään miten tehokas rakenteinen editori on verrattuna ominaisuuksiltaan vastaavaan tekstimuotoiseen editoriin sekä se, miten helposti rakenteisen editorin käyttö on opittavissa.

Esitetään seuraavaksi hahmotelmat kolmesta erilaisesta tutkimuksesta, jotka perustuvat edellä esitettyihin ryhmiin.

7.1 Yleiskäyttöisten ohjelmointityökalujen tutkimus

Tutkimuksessa tutkitaan sellaista rakenteista editoria, joka ominaisuuksiensa puolesta muistuttaa perinteistä IDE:tä. Tutkimuksessa halutaan selvittää, onko rakenteiseen editointiin perustuva ohjelmointiympäristö tehokkaampi kuin tekstin muokkaukseen perustuva. Tehokkuudella tarkoitetaan seuraavia asioita: saadaanko annetut ominaisuudet tehtyä nopeammin, tehdäänkö vähemmän virheitä (bugeja) ja onko lopputulos ylläpidettävämpi.

Mukana on vähintään neljä kokenutta ohjelmoijaa, joille kaikille jokin perinteinen IDE on tuttu, mutta joilla ei ole välttämättä kokemusta rakenteisista editoreista. Tehtävänä on ohjelman tekeminen annetun vaatimusmäärittelyn pohjalta sekä annetulla rakenteisella editorilla että vapaavalintaisella tekstin editointiin perustuvalla työkalulla, joka on ennestään tuttu. Vapaavalintaisella työkalulla saavutetaan hyvä vertailukohta rakenteiselle editorille: jos rakenteisella editorilla ohjelma saadaan tuotettua tehokkaammin kun ohjelmoijalle ennestään tutulla editorilla, niin voitaneen sanoa että sillä saavutetaan todellista hyötyä.

Puolet koehenkilöista tekee ohjelman ensin rakenteisella editorilla ja puolet vapaavalintaisella työkalulla. Tällä pyritään vähentämään lopputuloksissa sitä vaikutusta, että sovellusalue on ensimmäisen ohjelman tekemisen jälkeen ohjelmoijalle jo tutumpi.

Vaatimusmäärittelyssä on vaatimuksia prioriteeteilla pakollinen sekä hyödyllinen. Ohjelmoijia neuvotaan kirjaamaan ylös kommentteja työn etenemisestä sekä kohdatuista hankaluuksista.

Lopputuloksia arvioidaan seuraavilla kriteereillä:

- Työhön kulunut aika
- Toteutetut pakolliset vaatimukset
- Toteutetut hyödylliset vaatimukset
- Löydettyjen virheiden määrä
- Ylläpidettävyys
- Omat kommentit

Ylläpidettävyyttä voisi arvioida siten, että kokeen järjestäjä toteuttaa jokaiseen toteutettuun ohjelmaan jonkin lisäominaisuuden ja mittaa siihen kuluvan ajan.

Koehenkilöiden kokemuksia arvioidaan vapaamuotoisten kommenttien lisäksi myös kyselylomakkeella, jossa on seuraavat sanalliset kysymykset:

- Kuinka monta päivää kesti, ennen kuin rakenteisen editorin käyttö oli sujuvaa? (jos rakenteinen editori ei ollut ennestään tuttu)
- Mitä hyötyjä rakenteisen editorin käytössä oli?
- Mitä haittoja rakenteisen editorin käytössä oli?
- Käyttäisitkö rakenteista editoria jatkossa?

Lisäksi osa kysymyksistä voisi olla numerovalintoja, joiden arvot ovat välillä 1–5.

Johtopäätöksiä tehdessä otetaan huomioon paitsi lukuina mitattavat tulokset, niin yhtä lailla sanalliset kommentit, jotka voivat paljastaa asioita jotka eivät käy luvuista ilmi.

7.2 Aloittelijoille suunnattujen editorien tutkimus

Aloittelijoille suunnattujen editorien kohdalla on hyödyllistä tutkia, kuinka paljon ne auttavat ohjelmoinnin oppimisessa, kun koehenkilöt eivät osaa ennestään ohjelmoida. Mielenkiintoista olisi nähdä myös, onko rakenteisesta editorista siirtyminen perinteiseen tekstipohjaiseen ohjelmointiin sujuvaa.

Koehenkilöiden tehtävänä on tehdä peli. Tämä siksi, että aloittelijoille suunnatut editorit painottuvat graafisten ohjelmien tekemiseen, mutta pelkän animaation tekemisessä ei tulisi kunnolla mukaan ohjelmoinnissa tarvittavaa päättelyä. Pelin ei tietenkään tulisi olla kovin monimutkainen, jotta sen saisi tehtyä muutamassa päivässä. Ulkoasun tulisi olla yksinkertainen, jotta koehenkilöiden aika ei menisi grafiikan hieromiseen.

Kokeessa jokainen tekee saman pelin kahdesti, sekä rakenteisella että tekstimuotoisella editorilla. Koehenkilöt jaetaan kahteen samankokoiseen ryhmään. Kuten yleiskäyttöisten editorien tutkimuksessa (7.1), toinen ryhmä aloittaa rakenteisella editorilla ja toinen tekstimuotoisella.

Koehenkilöitä tulisi olla vähintään neljä, mutta mitä useampi, sen parempi. Koehenkilöille pidetään perehdytystä ohjelmoinnin perusteisiin ja editorin käyttöön sekä yksinkertaisia harjoituksia. Lisäksi he tekevät peliä ohjaajien läsnäollessa ja ohjaajat pitävät yksinkertaista kirjanpitoa siitä, kuinka usein apua on tarvittu.

Arviointikriteerit voisivat olla samat kuin luvussa 7.1. Lisäksi otetaan huomioon kuinka usein koehenkilöt tarvitsivat ohjaajien apua päästäkseen eteenpäin. Lopputulosten pohjalta voisi vertailla eroja: missä asioissa oppiminen meni paremmin rakenteisella editorilla ja missä tekstieditorilla. Erityisen mielenkiintoista olisi vertailla rakenteisella editorilla aloittaneiden ryhmän tuloksia keskenään: jos jälkimmäinen tekstieditorilla tehty peli sujui huonommin kuin ensimmäisenä tehty, tai huonommin kuin verrokkiryhmällä joka aloitti tekstimuotoisella tavalla, niin on syytä pohtia josko rakenteiset editorit hidastavat tekstimuotoisen ohjelmoinnin omaksumista.

7.3 Kielisuuntautuneen ohjelmoinnin työkalujen tutkimus

Samoin kuin yleiskäyttöisten ohjelmointityökalujen tutkimuksessa, kielisuuntautuneiden työkalujen tutkimuksessa haluttaisiin selvittää onko rakenteinen lähestyminen tehokkaampi kuin tekstiin perustuva. Tutkimuksen *voisi* tehdä samaan tapaan kuin yleiskäyttöisten editorien tutkimuksen, jossa tekstiin pohjautuva kielisuuntautunut työkalu on koehenkilöille jo ennestään tuttu. Kuitenkin, koska

1. olisi hankala löytää koehenkilöitä, joille kielisuuntautunut ohjelmointi olisi tuttua
2. oppimiskynnys lienee merkittävä tekijä kielisuuntautuneiden työkalujen käyttöönotolle

on mielekkäämpää tutkia kielisuuntautuneen ohjelmoinnin oppimista samaan tapaan kuin aloittelijoille suunnattujen editorien tutkimuksessa.

Tutkimus olisi siis pitkälti samanlainen kuin ohjelmoinnin aloittelijoiden tutkimus, mutta ohjelmoinnin perusteiden sijaan opetellaankin kielisuuntautuneen ohjelmoinnin perusteita. Vertailukohtana kielisuuntautuneen ohjelmoinnin työkaluille olisi vapaavalintainen IDE. Ohjelmoijilla tulisi olla usean vuoden kokemus ohjelmointiprojekteista, jotta kielisuuntautuneen ohjelmoinnin oppiminen olisi realistista, mutta korkeintaan pintapuolinen tietämys kielisuuntautuneesta ohjelmoinnista.

Koska kielisuuntautuneen ohjelmoinnin työkaluja käytetään tyypillisesti erilaisiin liiketoimintajärjestelmiin (esimerkiksi JetBrainsin MPS:llä toteutettu ohjelmistovirheiden seurantajärjestelmä YouTrack [21]), on pelin sijaan tehtävänä webbisovellus. Sovelluksen tulisi olla tarpeeksi mutkikas, jotta sen tekeminen ilman sovellussuuntautuneita kieliä vaatisi paljon koodia. Koehenkilöille voidaan antaa vihjeitä siitä, minkälaisia sovellussuuntautuneita kieliä tehtävää varten kannattaa kehittää, mutta kannustetaan kuitenkin niiden luovaan käyttöön.

8 Vertailu

Vertaillaan seuraavaksi editoreita toisiinsa eri aspektien kannalta. Vertailu perustuu vain kirjoittajan omiin kokemuksiin editoreiden käytöstä. Olisi toki toivottavaa tehdä kunnollinen tutkimus aiheesta ja tätä onkin hahmoteltu luvussa 7. Yhteenvetona tästä luvusta voisi saada jonkinlaisia suuntaviivoja sille mitä rakenteisilta editoreilta voisi tulevaisuudessa odottaa jos eri editoreiden parhaat puolet yhdistettäisiin.

Vertaillaan editoreita seuraavien näkökulmien kannalta

- käytön tehokkuus
- näkymät
- ohjelmointikielten tuki

Käytön tehokkuudella tarkoitetaan tässä editorien ominaisuuksia, jotka tekevät sen käytöstä tehokasta.

Näkymillä tarkoitetaan koodin erilaisia esitystapoja. Rakenteiset editorit mahdollistavat parhaimmillaan useita erilaisia näkymiä samaan koodiin, joiden välillä voidaan vaihtaa aina tarpeen mukaan. Näkymiä arvoidaan sanallisesti sen mukaan, miten hyvin ne havainnollistavat koodia verrattuna puhtaasti tekstimuotoiseen esitystapaan. Useampi näkymä ei välttämättä ole parempi.

Ohjelmointikielten tuella tarkoitetaan tuettuen ohjelmointikielten kirjoa. Suppeimmillaan editori on sidottu yhteen ainoaan ohjelmointikieleen, mikä siten rajoittaa mitä editorilla on ylipäätään mahdollista tehdä. Se voi myös tukea useita kieliä tai se voi tarjota työkalut uusien (sovellussuuntautuneiden) kielten määrittelyyn, jolloin editori on paljon joustavampi työkalu kuin yhden kielen tapauksessa.

8.1 Näkymät

Rakenteisissa editoreissa on eroteltu ohjelmakoodin tietorakenteet sekä sen esittäminen toisistaan. Tämä mahdollistaa koodinäkymille mm.

- havainnollisen esitystavan
- tarjolla olevien toimintojen esittämisen kontekstin mukaan

- yksityiskohtien piilottamisen tarpeen mukaan

Toki nämä toiminnot ovat ainakin jossain määrin mahdollisia myös tekstipohjaisissa IDE:issä, mutta rakenteisiin editoreihin ne sopivat luontevammin. Tarkastellaan seuraavaksi, kuinka hyvin nämä toteutuvat vertailtavissa editoreissa.

Edelläolevaan listaan voisi lisätä myös sen, kuinka hyvin editorit mahdollistavat näkymän mukautuksen halutunlaiseksi. Koska rakenteinen esitystapa ei määrittele syntaksia, olisi esimerkiksi mahdollista, että käyttäjä voisi itse päättää kuinka paljon koodilohkoja sisennetään. Valitettavasti useimmissa tutkittavissa editoreissa, lukuunottamatta MPS:ää sekä Metaedit+:aa, mahdollisuudet tähän ovat kovin vähäisiä. Kahdessa edellä mainitussa taas käyttäjällä on varsin vapaat kädet tehdä sovellussuuntautuneille kielille halutunlainen ulkoasu.

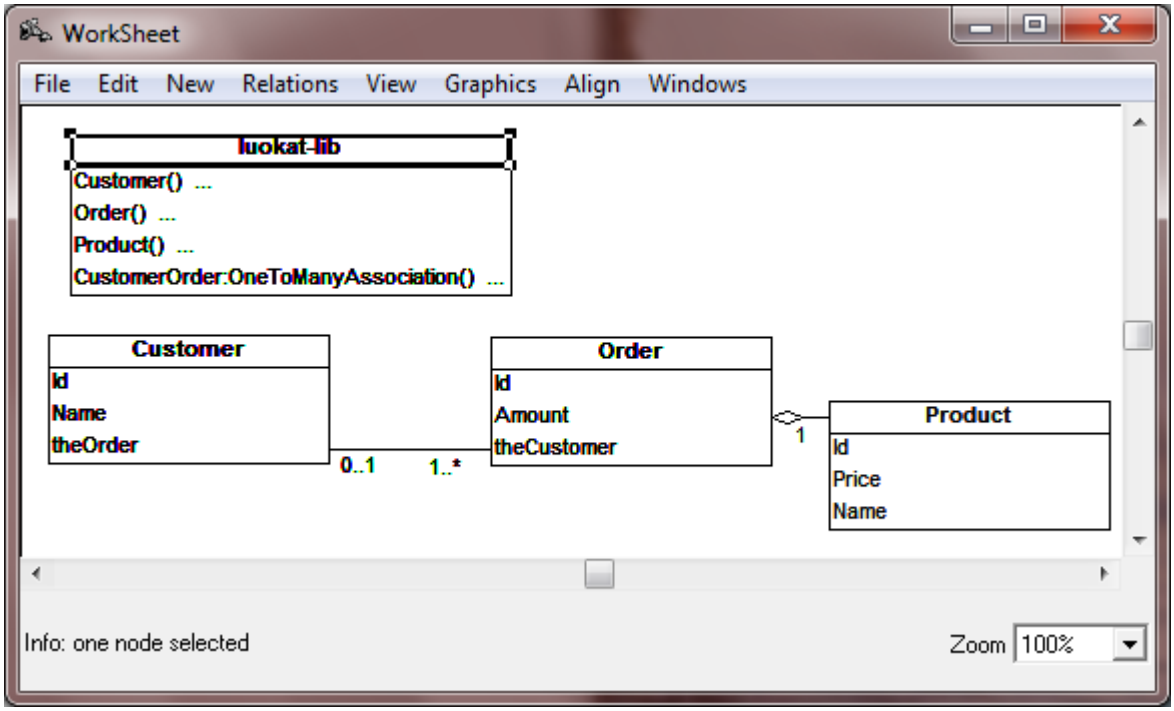
8.1.1 The Mjølner System

The Mjølner Systemissä toteutuu se, että rakenteisessa editorissa voi olla samaan koodiin useita näkymiä. Tarjolla on monta erilaista näkymää tarpeen mukaan: mm. koodi-, käyttöliittymä- sekä UML–näkymä (kuva 8.1). Muutokset yhteen näkymään näkyvät heti toisessa: jos koodinäkymässä lisää luokkaan uuden attribuutin, ilmaantuu se samantien näkyviin luokkakaavioon. Toisaalta, jos koodinäkymässä lisää uuden luokan, pitää se UML–näkymässä vielä itse raahata sopivaan paikkaan jos haluaa pitää UML–näkymän ajan tasalla.

Koodinäkymässä (kts. kuva 6.1) toteutuu myös yksityiskohtien piilottaminen: piiloonjäävä osa havainnollistetaan kolmella pisteellä. Valittu koodin osa (mukaanlukien kommentit) on helppo laajentaa tai supistaa yleiskuvan tai vaihtoehtoisesti yksityiskohtien näkemiseksi. Ohjelman tulostaminen sopivalla tarkkuustasolla tarjoaa hyvän lähtökohdan dokumentaatiolle [46].

Tarjolla olevien toimintojen esittäminen kontekstin mukaan toteutuu ohjelmointikielen rakenteille. Kuvassa 8.2 näkyy hiiren oikeasta painikkeesta aukeava kontekstivalikko, joka näyttää kaikki valitulle välikkeelle sopivat rakenteet. Valikon valinta laajentaa välikkeen tilalle valitun rakenteen, joka voi sisältää uusia välikkeitä, joita voi edelleen laajentaa eteenpäin. Tätä voi jatkaa kunnes joko tullaan kielen *päätesymboleihin* tai välikkeet ovat helppoja laajentaa suoraan tekstiä kirjoittamalla. Kontekstivalikko on hyödyllinen jos BETA–kieli on tuttu, mutta aloittelijalle se ei ole kovin havainnollinen.

Koodinäkymää ei voi kehua erityisen havainnolliseksi. Värejä ei ole käytetty lainkaan, joten koodin ulkoasu ei poikkea tavallisesta tekstieditorista.



Kuva 8.1: Mjølnerin UML–näkymä

```
pushbuttonLisaaHA: @pushbutton
 (#
   open::< (#  do (60,20)->size; (20,85)->position; INNER #);
   eventHandler::<
    (#
      onMouseUp::< <<PrefixOpt>>
       (# <<AttributeDeclO
       <<EnterPartOpt>>
       <<DoPartOpt>>
       <<ExitPartOpt>>
       #)
    #)
 #)
```

PatternDecl
SimpleDecl
RepetitionDecl
VirtualDecl
BindingDecl
FinalDecl
Empty

Kuva 8.2: Sif–editorin kontekstivalikko

8.1.2 Lava

Lavassa koodin muokkaus on jostain syystä jaettu kahteen näkymään: puunäkymään sekä suoritusnäkymään. Puunäkymässä on korkean tason näkymä yksittäiseen pakettiin ja suoritusnäkymässä esitetään yksittäinen funktio. Kuten Mjølne-

rissä, suoritusnäkymä muistuttaa kovasti perinteistä koodieditoria puolipisteineen, joskin tässä sentään avainsanat on väritetty. Merkkijonovakiota kirjoittaessa tulee merkkijonon ympärille kirjoittaa lainausmerkit. Tämä on hieman kummallista, sillä tekijät väittävät, ettei Lavalla ole syntaksia [34].

Yksityiskohtien piilottamista ei tueta suoritusnäkymässä, mutta puunäkymässä alemman tason solmut saa halutessaan piiloon.

Lavan työkalurivillä on näkyvissä suurin osa toiminnoista ja kulloinkin käytettävissä olevat toiminnot näkyvät harmautettuna. Tämä ei ole ehkä paras mahdollinen ratkaisu, sillä haluttu toiminto tulisi löytää kaikkien mahdollisten joukosta ja lisäksi käyttäjän tulee suunntata huomio hetkeksi pois siitä kohdasta koodia jota on katsomassa (ellei muista näppäinkomentoa, jos sellainen on). Minkäänlaista hiirellä klikattavaa kontekstivalikkoa ei ole.

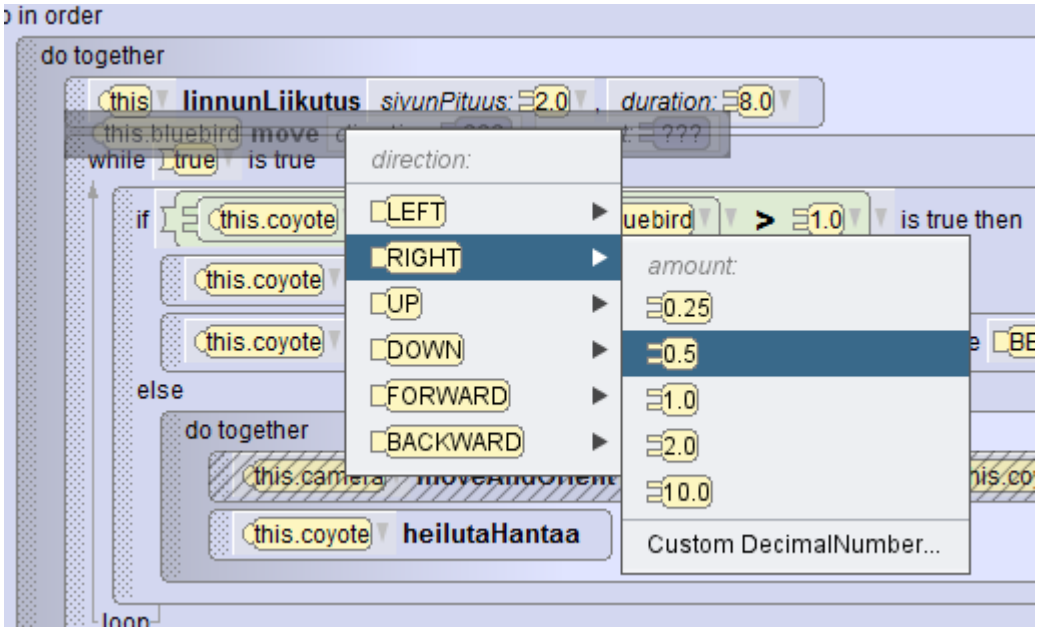
Hieman sekavaa on, että osa työkalurivien lisättävistä lausekkeista on yläreunan työkalurivissä, osa ikkunan vasemmassa reunassa. Ilmeisesti avainsanoilla esitettävät rakenteet ovat vasemmassa reunassa, kun taas symboleilla esitettävät operaattorit työkalurivissä.

8.1.3 Alice

Alicessa on luovuttu perinteisestä syntaksista. Sen sijaan kielen rakenteet ovat graafisia elementtejä, joita voi liikutella hiirellä paikasta toiseen ilman, että tarvitsee pelätä syntaksin menevän rikki. Tämä tekee editorista havainnollisen. Myös koodin "pois kommentointi" on toteutettu havainnollisesti: koodilohkolta voi ottaa pois *Is Enabled* -asetuksen, jolloin se näkyy harmautettuna (kts. kuva 6.10). Lisäksi olioiden esittäminen graafisesti tekee olio–ohjelmoinnista vähemmän abstraktia, joskin Alice ei toki yritäkään taipua abstraktimpien asioiden esittämiseen.

Erityisen hyvin Alicessa toimii vaihtoehtojen esittäminen kontekstin mukaan. Kun lauseita raahataan koodieditoriin, tulee kaikille pakollisille parametreille valita heti arvo aukeavasta valikosta (kuva 8.3). Jokin konkreettinen arvo tulee valita, vaikka tarkoitus olisikin käyttää esimerkiksi funktiosta saatua arvoa. Mielestäni tämä on pienen hämmennyksen jälkeen toimiva tapa, sillä konkreettisen arvon valitsemalla voi helposti testata siihenastista koodia ennen kuin tekee siitä yhtään mutkikkaampaa.

Esimerkiksi olion liikkumismetodin suunta–argumentille on kaikki vaihtoehdot (vasemmalle, eteen, jne.) nähtävissä kontekstivalikossa heti argumenttia klikkaamalla ja liikkumisen määrän numeroarvolle kontekstivalikossa on vaihtoehtoina mm. joitakin valmiita vakioarvoja, vapaavalintainen luku, satunnaisluvun gene-



Kuva 8.3: Parametrin valinta Alicessa

rointi tai matemaattinen lauseke.

Alicen käyttöliittymä tuntuu omien kokeilujen perusteella havainnolliselta. Koodin muokkaaminen hiirtä käyttämällä on helppoa ja intuitiivista. Graafinen ulkoasu antaa selkeitä vihjeitä siitä minkälainen lause sopii mihinkin kohtaan.

8.1.4 Scratch

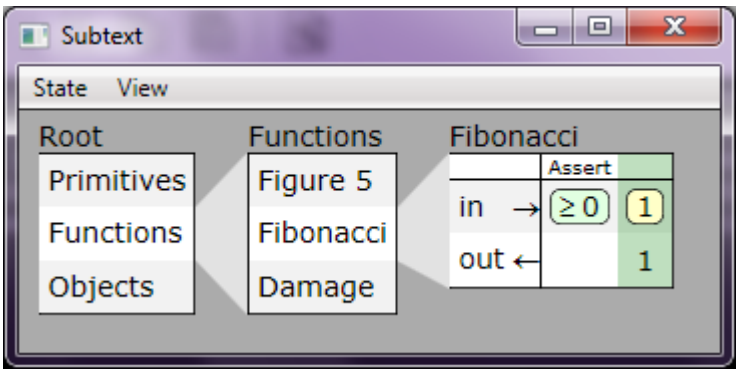
Samaan tapaan kuin Alicessa, Scratchin koodirakenteet ovat hiirellä raahattavia palikoita, joita voi helposti kokeilemalla liittää yhteen. Sekä Scratchissa että Alicessa aliohjelman parametrien muoto vihjaa, minkä tyyppisiä argumentteja aliohjelma ottaa vastaan. Vaikka tyypitys ei olisikaan kovin tuttu asia, niin lienee selvää, että kulmikkaaseen aukkoon (boolean–tyyppinen parametri) ei sovi ympyrän muotoinen funktio (lukuarvotyyppinen). Scratchissä on lisäksi palikoiden välillä pystysuunnassa pienet nystyrät, jotka vihjaavat niiden sopivan yhteen. Tosin on vaikea sanoa, onko tällä havainnollisuuden kannalta suurta merkitystä.

Vaihtoehtojen esittäminen kontekstin mukaan toimii myös Scratchissä. Kaikki tarjolla olevat proseduurit ovat näkyvillä, ryhmiteltynä eri sivuille aiheen mukaan. Proseduurien parametrit esitetään listana silloin, kun ne ovat luettelotyyppisiä, esimerkiksi näppäimistön näppäin. Numero- sekä merkkijonotyyppisille parametreille on yksinkertainen tekstikenttä, mikä vaikuttaa toimivalta ratkaisulta.

8.1.5 Subtext

Subtext 2:n taulukkomuotoinen ehtolausekenäkymä on radikaalisti erilainen kuin perinteisten ohjelmointikielten ehtolausekkeet. Kaksiulotteisuudella saavutetaan se, että kysymykseen "Millä ehdoilla funktion tulokseksi tulee 0" on helppo vastata katsomalla miltä sarakkeelta haluttu tulos löytyy. Tekeekö taulukkonäkymä ehtolauseista havainnollisia, siihen kysymykseen onkin paljon vaikeampi vastata ja se saattaa olla myös makuasia.

Subtextissa on mielenkiintoinen variaatio yksityiskohtien piilottamiseksi tarvittaessa: Ehtolauseissa voi saraketta tuplaklikkaamalla kohdistaa huomion vain tietyillä ehdoilla muodostuvaan suorituspolkuun jättäen näkyvistä ehdon poissulkemat suorituspolut. Esimerkiksi fibonacci–funktion (kuva 6.15) suorituspolku parametrin arvolla 1 näkyy kaikessa yksinkertaisuudessaan kuvassa 8.4.



Kuva 8.4: Fibonacci–funktio, josta suoritus ainoastaan parametrin arvolla 1 näkyvissä

8.1.6 JetBrains Meta Programming System

Meta Programming System mahdollistaa paitsi oman ohjelmointikielen, myös näkymän eli editorin luomisen sille. Niinpä näkymän saa muokattua aina juuri omaan sovellusalueeseen sopivaksi.

Näkymät koostuvat pääasiassa tekstistä, mutta ovat rakenteisia: käyttäjä voi editoria luodessaan määritellä esimerkiksi, että funktiot alkavat aina avainsanalla *function*. Kun luotua editoria sitten käytetään, ja käyttäjä valitsee editorin täydennyslistasta funktion, editori näyttää avainsanan *function* automaattisesti funktion alussa. Editorit tukevan koodin täydennystä omille sovellussuuntautuneille kielille.

Editorit mahdollistavat jossain määrin myös muitakin kuin tekstimuotoisen esitystavan. Esimerkiksi taulukkomuotoista esitystapaa on käytetty päätöstauluihin

mbeddr–projektissa [38].

8.1.7 MetaEdit+

MetaEdit+ tarjoaa graafisen näkymän mallinnuskielten käyttämiseen, samankaltaisen kuin monissa UML–editoreissa. Symbolit eivät ole ennalta määrättyjä, vaan mallinnuskielen kehittäjä voi tehdä niistä tarkoitukseen sopivia. Niinpä MetaEdit+ mahdollistaa juuri omaan sovellusalueeseen sopivan näkymän luomisen.

Näkymän luominen tapahtuu valitsemalla hiirellä dialogeista mallinnuskielen metamallin ominaisuudet. Tämä on varsin suoraviivainen ja toimiva tapa, kunhan metamallin käsitteet on ensin sisäistetty.

Sekä Meta Programming System että MetaEdit+ mahdollistavat omaan sovellusalueeseen sopivan näkymän luomisen. Siinä missä MPS:llä luodut editorit sopivat puumuotoisen, tekstinä esitettävän kielen esittämiseen, MetaEdit+:n näkymät sopivat graafisten mallinnuskielten luomiseen, jotka ovat graafimuotoisia.

8.2 Ohjelmointikielten tuki

Tarkastellaan mitä ohjelmointikieliä editorit tukevat. Tämä osaltaan määrittelee sen, kuinka käyttökelpoisia editorit ovat.

8.2.1 The Mjølner System

The Mjølner System käyttää BETA–kieltä, jonka kehitystyö alkoi vuonna 1976 SIMULA–kielen pohjalta. BETA on vahvasti tyypitetty. Olio–ohjelmointi on kielessä vahvasti tuettuna ja se on suunniteltu tukemaan oliomallinnusta. Kieli tukee monia abstrahointimekanismeja, kuten luokka, proseduuri, funktio, vuorottaisrutiini (coroutine), prosessi sekä poikkeus [53]. [27]

The Mjølner System tarjoaa useita kirjastoja sekä sovelluskehyksiä BETA–kielelle, mm. käyttöliittymiin, grafiikan piirtämiseen, verkossa hajautettuihin olioihin (distributed objects) sekä rinnakkaislaskentaan [44] [45].

Valitettavasti BETA–kielen kehitys ei ole enää aktivista [4] ja niinpä ympäristö on auttamatta ajastaan jäljessä.

8.2.2 Lava

Lava tukee omaa ohjelmointikieltään, joka on periaatteessa tehokas olio–ohjelmointikieli, mutta koska se ei ole laajasti käytössä, sitä ei voi sanoa kovin varmaksi va-

linnaksi mihinkään projektiin.

8.2.3 Alice

Alicessa on oma olio–ohjelmointikielensä, joskin ominaisuuksiltaan hyvin rajoittunut. Esimerkiksi olion luominen ohjelmallisesti ei onnistu. Alice onkin erikoistunut ainoastaan 3D–näkymän omaavien pelien ja animaatioiden kehittämiseen. Koska Alice on suunniteltu Java–kielen opettelua silmällä pitäen, Alicen semantiikka muistuttaa Javan käsitteitä. Tästä muistuttaa myös metodien nimen edessä toistuva `this`-avainsana (jonka tosin saa ohjelman asetuksista poistettua).

8.2.4 Scratch

Scratch tukee omaa ohjelmointikieltään, joka on suunniteltu 2D–multimediaesitysten ja pelien tekemiseen. Kieli on tarkoituksella ominaisuuksiltaan rajoittunut – esimerkiksi luokkia ei ole – mutta se muodostaa selkeän kokonaisuuden ja erilaisten koodinpätkien kokeileminen on nopeaa interaktiivisen mallin ansiosta. Uusien olioiden luominen ohjelmallisesti onnistuu, joten tässä asiassa Scratch on Alicea parempi. Toisaalta käyttäjän luomat proseduurit eivät voi palauttaa arvoa, toisin kuin Alicessa. Valinta Scratchin ja Alicen välillä onkin melko pitkälti makuasia ja myös siitä kiinni, haluaako luoda 2D– vai 3D–animaatioita.

8.2.5 Subtext

Subtextin tukee omaa ohjelmointikieltään, jonka erikoisuutena on kaksiulotteinen esitystapa. Kieli on hyvin kokeellinen, eikä sillä ole tehty kokonaisia sovelluksia.

8.2.6 JetBrains Meta Programming System

JetBrains Meta Programming System on kielisuuntautuneen ohjelmoinnin työkalu. Siinä käyttäjä miettii ensin miten tietyn ratkaisun voisi parhaiten kuvata ja toteuttaa sitten kielen, jolla kuvaus onnistuu. Rakenteisen editoinnin ansiosta eri kieliä voi yhdistellä keskenään. Tuettuja kohdekieliä on Javaa muistuttava BaseLanguage (joka lopulta käännetään Java–tavukoodiksi) sekä XML. Myös C–kielelle kohdekielenä on tehty toteutus [39]. Käyttäjä voi myös itse luoda kohdekielen, joskin “oikean” kielen toteuttaminen on toki työlästä [50].

8.2.7 MetaEdit+

MetaEdit+ mahdollistaa uusien sovellussuuntautuneiden mallinnuskielten luomisen, joten siinä mielessä se on hyvin monipuolinen työkalu. Mallinnuskielet ovat graafisia, mikä sopii joihinkin ongelmiin paremmin ja joihinkin huonommin. Kielen suunnittelijalla on vapaus tehdä kielestä haluamansa näköinen. Tehokkaan MERL–kielen ansiosta generaattorin voi kirjoittaa mille tahansa kohdekielelle. Toisin kuin MPS:ssä, mitään kohdekieltä ei suoraan tueta, joten generaattorin kirjoittajan täytyy pitää huoli kohdekielen syntaksin muotoilusta sekä jättää kohdekielien oikeellisuustarkistus kyseisen kielen kääntäjälle. Toisaalta, tämän voi nähdä myös niin, ettei MERL–kieli aseta mitään rajoitteita kohdekielen muodostamiselle.

8.3 Käytön tehokkuus

Arvioidaan lyhyesti, kuinka tehokkaita editorit ovat käyttää. Tehokkuudella tarkoitetaan tässä, että kuinka nopeasti käyttäjä saa haluamansa toiminnot tehtyä, kun editori on hänelle tuttu. Jälleen, koska editorit ovat kovin erilaisia, minkäänlaisen järjestelmällisen testin tekeminen olisi hyvin hankalaa, jos edes mielekästä. Painotettakoon vielä, että ohjelmoinnin opetteluun suunnitelluissa editoreissa tehokkuus ei liene edes tavoiteltava ominaisuus ja että varsinkin sovellussuuntautuneiden kielten alustoissa yksittäisen koodinpätkän kirjoittamiseen kuluvan ajan tuijottaminen hukkaa metsän puilta, sillä sopivan kielen kehittämällä voi välttyä monta kertaa pidemmän koodinpätkän kirjoittamiselta.

Yksi tehokkuuteen vaikuttava asia on näppäinkomennot. On havaittu, että näppäimistöllä suoritetut komennot ovat monissa tehtävissä hiirellä käytettyjä komentoja nopeampia käyttää [29]. Toinen asia on toiminnot, joita editori tarjoaa koodin muokkaamiseen, kuten vaikkapa koodin refaktorointi [54].

8.3.1 The Mjølner System

The Mjølner Systemin Sif–editorissa koodia muokataan rakenteisesti, mutta valitun kokonaisuuden voi valita muokattavaksi tekstimuotoisesti. Tekstimuokkauksen jälkeen editori tarkistaa, että muokattu osa on syntaksin mukainen ennen kuin antaa käyttäjän jatkaa, jotta rakenne pysyy eheänä. Sifin oppaan mukaan tekstieditointi on hyödyllinen muokattaessa yksityiskohtia [46]. Isommille kokonaisuuksille rakenteinen muokkaus on tehokasta. Esimerkiksi kokonaisen aliohjelman siirtäminen paikasta toiseen käy seuraavasti:

- valitaan siirrettävä aliohjelma
- tehdään toiminto *Cut* (*leikkaa*, pikanäppäin Ctrl–X)
- valitaan aliohjelma, jonka jälkeen siirrettävä aliohjelma tulee
- tehdään toiminto *Paste after* (*liitä jälkeen*, pikanäppäin Ctrl–Insert)

Editori antaa valita vain loogisesti kokonaisia osia ohjelmasta kerrallaan. Niinpä vaikkapa aliohjelman maalaaminen valituksi onnistuu yhdellä hiirenklikkauksella. Isojen kokonaisuuksien muokkaamista helpottaa myös, että symbolista (esimerkiksi luokan nimi) voi helposti navigoida sen määrittelyyn ja takaisin päin.

Navigointi koodissa nuolinäppäimillä tuntuu hieman epäselvältä eikä sitä selitetä ohjekirjassa. Editorissa on kuitenkin monia käyttöä tehostavia pikanäppäimiä ja näppäimistön ja hiiren yhteiskäytöllä se vaikuttaa tehokkaalta.

Refoktorointiominaisuuksia ei valitettavasti ole. Esimerkiksi muuttujan nimen muuttaminen ei päivitä viittauksia siihen, toisin kuin Lavassa. Toki muutettu nimi näkyy kuitenkin oikein eri näkymissä, kuten UML–näkymässä sekä koodinäkymässä.

8.3.2 Lava

Lavassa navigointi niin puunäkymässä kuin suoritusnäkymässä toimii näppäimistön nuolinäppäimillä pitkälti kuten voisi odottaakin – tosin suoritusnäkymässä nuolinäppäinten toiminta ei ole heti intuitiivista, sillä koodin näkymä ei vastaa jäsennyspuuta samaan tapaan kuin puunäkymässä. Suoritusnäkymässä on käteviä pikänäppäimiä, esimerkiksi i–kirjainta painamalla saa if–lauseen. Toisaalta läheskään kaikille valikoiden valinnoille ei ole edes alt–näppäimellä painettavaa pikanäppäintä, joten moni asia pitää klikata joko valikoista tai työkaluriviltä. Lisäksi kontekstivalikon puute tekee halutun toiminnon löytämisestä hidasta.

Tavallisimmat refaktorointi–toiminnot toimivat rakenteisen toteutuksen ansiosta [32]. Esimerkiksi metodin nimen muuttaminen tarvitsee tehdä vain yhteen paikkaan.

8.3.3 Alice ja Scratch

Sekä Alice että Scratch on suunniteltu ohjelmoinnin alkeiden opetteluun, joten tehokkuus ei liene niissä ensisijainen asia. Molemmat ovat suurimmaksi osaksi hiirellä käytettäviä. Mutta tehokkuus on kovin suhteellinen käsite. Molemmissa hii-

rikäyttöliittymä on hyvinkin sujuva ja halutut toiminnot saa nopeasti hiirellä raahattua oikeille paikoilleen. Erityisesti lausekkeiden siirtäminen aliohjelman sisällä paikasta toiseen käy hiirellä kätevästi. Molemmissa saa nopeasti näkyvää jälkeä aikaiseksi. Voitaneen sanoa, että Alice ja Scratch ovat matalaan oppimiskäyräänsä nähden tehokkaita.

8.3.4 Subtext

Subtext on luotu havainnollistamaan taulukkomuotoisia konditionaaleja ja sellaisenaan siinä ei ole juurikaan ominaisuuksia, joita voisi kuvailla tehokkaiksi. Kuitenkin, Subtext tarjoaa kontekstivalikon kutsuttavan funktion nimen täydentämiseen. Paitsi kontekstivalikkoa klikkaamalla, täydennettävän nimen voi valita myös klikkaamalla funktion nimeä jos se esiintyy muualla editorissa.

8.3.5 MetaEdit+

MetaEdit+ on pääasiassa hiirivetoinen. Tavalliset näppäinoikotiet toimivat. Mallinnuskielen luomisessa tai käyttämisessä yksittäisten komentojen nopea suorittaminen ei ole oleellista, sillä hyvin suunnitellulla mallinnuskielellä saadaan generoitua paljon koodia, jonka käsin kirjoittaminen saattaisi olla hyvinkin työlästä. Niinpä tehokkuuden kannalta käytettävyys korostuu enemmän, ja tässä suhteessa MetaEdit+ on tehokas käyttää.

8.3.6 JetBrains Meta Programming System

JetBrainsin kehitystyö IDE:iden parissa (mm. Java–IDE IntelliJ IDEA [19]) näkyy Meta Programming Systemissä. Siinä on mm. seuraavat käyttöä tehostavat ominaisuudet:

- Koodin täydennysvalikko
- Refaktorointi, mm. symbolin nimen muuttaminen
- *Intention*-ominaisuus, joka tarkoittaa automatisoituja koodin muunnoksia

Kaikkia näitä voi käyttää sujuvasti näppäimistöltä.

Nämä ominaisuudet toimivat juuri niin hyvin kuin mitä nykyaikaiselta IDE:ltä voi odottaa. MPS:n vahvuus on se, että edellä mainitut ominaisuudet toimivat myös omille kielille. Tosin omien kielten tehokas luonti tulee jyrkän oppimiskäyrän kanssa, ainakin mikäli kielisuuntautunut ohjelmointi ei ole ennestään tuttua. Jos uuden

ohjelmointiparadigman oppiakseen pitää “kiepauttaa aivot ympäri”, niin se todellakin pätee MPS:ään sekä kielisuuntautuneeseen ohjelmointiin.

8.4 Omat kokemukset

Lopuksi kerron omista kokemuksistani editorien parissa vastaamalla itse luvussa 7.1 esitettyihin kysymyksiin kootusti kaikkien editoreiden osalta. Valitettavasti en ole lainkaan kirjannut ylös editoreiden oppimiseen kulunutta aikaa, joten jätän vastaamatta kysymykseen siitä kuinka kauan oppiminen kesti.

Mitä hyötyjä rakenteisen editorin käytössä oli?

Opetelueditoreiden kanssa rakenteisuus teki ohjelmoinnista hauskaa: ohjelmat koostuvat palikoista, joita voi huoletta raahata hiirellä ilman, että syntaksi menisi rikki. Myös se, että kaikki mahdolliset palikat ovat valikoissa näkyvillä tekee aloituskynnyksestä matalan. Sama pätee muihinkin editoreihin: ei tarvinnutkaan enää olla tarkkana puuttuvien sulkumerkkien kanssa.

Yleiskäyttöiset editorit (The Mjølner System sekä Lava) eivät olleet tehokkaampia käyttää kuin nykyaikaiset IDE:t — joista minulle tutuin on Microsoft Visual Studio JetBrainsin ReSharper –laajennoksella. Tämä on ymmärrettävää, sillä testattuja rakenteisia editoreita ei ole kehitetty moniin vuosiin ja toisaalta IDE:t ovat kehittyneet koko ajan eteenpäin.

Rakenteisissa editoreissa on hyvää se, että ei tarvitse muistaa niin paljon mitä voi kirjoittaa mihinkin. Tämä tuli esille mm. MPS:n yksikkötesteissä: editori näyttää laajennuspaikat joihin esimerkiksi testien alustukset voi lisätä helposti enterin painalluksella.

Niin MetaEdit+ kuin Subtextkin näyttivät, että ohjelmointi voi olla paljon muutakin kuin mihin on totuttu, kun tekstiformaatin kahleista luovutaan. Metaedit+:ssa voi keskittyä ajattelemaan sovellusalueen käsitteillä ja miettimään niiden välisiä suhteita. Subtext, vaikka ei olekaan sellaisenaan hyödyllinen, välitti idean että graafisessa esityksessä ohjelman logiikan voi hahmottaa aivan eri lailla ja että tavallista ehtolausetta voisi katsella erilaisten “silmälasien” läpi aina tarpeen mukaan.

Mitä haittoja rakenteisen editorin käytössä oli?

Joissain tilanteissa MPS:n kanssa (katso luku 6.6.2) rakenteinen editori ei anna kirjoittaa suoraan lausetta, jonka kirjoittaminen tekstieditorilla olisi hyvin suoraviivaista. Mjølnerissä taas tällainen hoituu editointimoodia käyttämällä. Kuitenkin, tämä on mielestäni pieni puute ja jos jäisi murehtimaan yksittäisen lauseen kirjoittamista, niin silloin hukkuisi metsä puilta.

Käyttäisitkö rakenteista editoria jatkossa?

Tuskin olen hylkäämässä "perinteisiä" editoreita vielä pitkään aikaan, mutta ilman muuta voisin käyttää rakenteista vaihtoehtoa jos oikea työkalu kohtaa oikean projektin. Testatuista editoreista voisin käyttää jatkossa jo ihan oman mielenkiinnon vuoksi MPS:ää sekä Scratchiä (molemmat ovat vieläpä ilmaisia). Kielisuuntautunut ohjelmointi on kiehtova (joskin haastava) paradigma ja MPS:ssä on nykyaikaisista IDE:istä tutut hienoudet. Tosin minulla ei ole mielessä mitään projektia, johon paradigma erityisesti sopisi, mutta ehkä kokeilemalla selviää että miten se toimii eri tyyppisissä sovelluksissa.

Sekä Alicella että Scratchillä on hauskaa ohjelmoida pikku pelejä ja animaatioita omaksi huvikseen, ilman että niistä olisi kenellekään hyötyä. Näistä kahdesta Scratch on oma suosikkini karvan mitalla.

9 Yhteenveto

Tässä tutkielmassa kokeiltiin rakenteisia editoreita, joita käytetään ohjelmoinnissa. Rakenteinen editori eroaa tavanomaisesta tekstieditorista siinä, että ohjelmointikielen syntaksia ei ole: merkkijonojen sijaan editorissa muokataan loogisia kokonaisuuksia, kuten lausekkeita tai muuttujia.

Rakenteinen editori helpottaa ohjelmointia esimerkiksi poistamalla syntaksivirheiden mahdollisuuden sekä tarpeen koodin formatoinnille. Joissakin tapauksissa tekstieditorin käyttö voi olla kuitenkin helpompaa.

Käyttökohteina rakenteisille editoreille ovat muun muassa ohjelmoinnin opettelu sekä kielisuuntautunut ohjelmointi. Ohjelmoinnin opettelussa ohjelmakoodin visuaalisempi ulkoasu voi helpottaa koodin hahmottamista. Myöskin syntaksivirheiden mahdollisuuden puuttuminen on iso etu, joskin se voi haitata siirtymistä tavanomaisempiin ohjelmointikieliin. Kielisuuntatuneessa ohjelmoinnissa taas käytetään useaa, tiettyyn erikoistarkoitukseen suunniteltua, kieltä rinnakkain. Tämä olisi paljon monimutkaisempaa, jos kielet olisivat tekstimuotoisia.

Rakenteisia editoreita on kehitetty jo monen vuosikymmenen ajan. Toistaiseksi niiden käyttö ei ole kovin yleistä joitakin erikoistapauksia, kuten ohjelmoinnin oppimisympäristöjä, lukuunottamatta. “Tavanomaiseen” ohjelmointiin ei ole tarjolla varteenotettavaa vaihtoehtoa, ainakaan tässä tutkielmassa esitellyistä editoreista (toki joukko ei ole millään tapaa kattava). Kaikki ovat joko liian erikoistuneita, vanhentuneita tai soveltuvat vain yksinkertaisiin sovelluksiin. Aika näyttää, pitävätkö tekstimuotoiseen ohjelmointiin perustuvat IDE:t ja editorit edelleen pintansa vai lyökö rakenteinen editointi joskus kunnolla läpi.

Lähteet

[1] Alicen kotisivu. http://www.alice.org/. Viitattu 24.12.2012.

[2] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML) (tm) Version 1.2. Technical report, YAML.org, 9 2009.

[3] T. Berners-Lee and D. Connolly. Hypertext Markup Language - 2.0. RFC 1866 (Historic), November 1995. Obsoleted by RFC 2854.

[4] BETA–kielen kotisivu. http://www.cs.au.dk/~beta/. Viitattu 25.12.2015.

[5] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[6] Butler Lampsonin kotisivu. http://research.microsoft.com/en-us/um/people/blampson/Systems.html#qed. Viitattu 17.12.2015.

[7] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: A 3-D Tool for Introductory Programming Concepts. *J. Comput. Sci. Coll.*, 15(5):107–116, April 2000.

[8] Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching Objects-first in Introductory Computer Science. *SIGCSE Bull.*, 35(1):191–195, January 2003.

[9] Stephen Cooper, Wanda Dann, and Randy Pausch. Using Animated 3D Graphics To Prepare Novices for CS1. *Computer Science Education Journal*, 13:28–29, 2003.

[10] Wanda Dann, Dennis Cosgrove, Don Slater, Dave Culyba, and Steve Cooper. Mediated transfer: Alice 3 to Java. In Laurie A. Smith King, David R. Musicant, Tracy Camp, and Paul T. Tymann, editors, *SIGCSE*, pages 141–146. ACM, 2012.

[11] Darthmouth BASIC -käyttöopas. http://bitsavers.trailing-edge.com/pdf/dartmouth/BASIC_Oct64.pdf. Viitattu 9.4.2014.

[12] L. Peter Deutsch and Butler W. Lampson. An Online Editor. *Commun. ACM*, 10(12):793–799, 1967.

[13] Jonathan Edwards. No Ifs, Ands, or Buts: Uncovering the Simplicity of Conditionals. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 639–658, New York, NY, USA, 2007. ACM.

[14] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? `http://martinfowler.com/articles/languageWorkbench.html`, 2005. Viitattu 24.12.2012.

[15] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[17] Ian F. Darwin, Geoffrey Collyer. A History of UNIX before Berkeley: UNIX Evolution: 1975-1984. `http://www.darwinsys.com/history/hist.html`. Viitattu 17.12.2015.

[18] Integrated Development Environment (IDE). `http://www.techopedia.com/definition/26860/integrated-development-environment-ide`. Viitattu 22.6.2014.

[19] JetBrains IntelliJ IDEA:n kotisivu. `https://www.jetbrains.com/idea/`. Viitattu 26.9.2015.

[20] ECMA International. *Standard ECMA-334 - C# Language Specification*. 4 edition, June 2006.

[21] YouTrack-ohjelman kotisivu. `https://www.jetbrains.com/youtrack/`. Viitattu 13.11.2015.

[22] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, 1979.

[23] Jim Joyce. Interview with Bill Joy. `http://web.cecs.pdx.edu/~kirkenda/joy84.html`. Viitattu 17.12.2015.

[24] JUnit kotisivu. `http://junit.org/`. Viitattu 2.10.2015.

[25] Caitlin Kelleher and Randy Pausch. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.

[26] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[27] Bent Bruun Kristensen, Ole Lehrmann Madsen, and Birger Møller-Pedersen. The When, Why and Why Not of the BETA Programming Language. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 10–1–10–57, New York, NY, USA, 2007. ACM.

[28] National Instruments LabVIEW. `http://poweredby.labview.com/`. Viitattu 25.5.2014.

[29] David M. Lane, H. Albert Napier, S. Camille Peres, and Anikó Sándor. Hidden Costs of Graphical User Interfaces: Failure to Make the Transition from Menus and Icon Toolbars to Keyboard Shortcuts. *International Journal of Human-Computer Interaction*, 18(2):133 – 144, 2005.

[30] Vesa Lappalainen. Delphi-pikakurssi. `http://users.jyu.fi/~vesal/kurssit/winohj/delphi/moniste/m/d.html`, 1997. Viitattu 12.9.2015.

[31] Lavan kotisivu. `http://lavape.sourceforge.net/`. Viitattu 25.12.2014.

[32] Refaktorointi Lavassa. `http://lavape.sourceforge.net/doc/html/Refactoring.htm`. Viitattu 16.8.2015.

[33] Lavan lähdekoodi. `http://sourceforge.net/projects/lavape/`. Viitattu 25.12.2014.

[34] Lavan rakenteisen editoinnin edut. `http://lavape.sourceforge.net/doc/html/StructEdAdvant.htm`. Viitattu 25.12.2014.

[35] M. E. Lesk and E. Schmidt. UNIX Vol. II. chapter Lex – a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[36] Håkon Wium Lie and Bert Bos. Cascading Style Sheets, level 1. W3C recommendation, W3C, January 1999.

[37] David J. Malan and Henry H. Leitner. Scratch for Budding Computer Scientists. *SIGCSE Bull.*, 39(1):223–227, March 2007.

[38] mbeddr päätöstaulut. https://mbeddr.wordpress.com/2011/09/26/decision-tables/. Viitattu 12.7.2015.

[39] mbeddr kotisivu. http://mbeddr.com/. Viitattu 1.5.2015.

[40] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, August 1978.

[41] Bernhard Merkle. Textual Modeling Tools: Overview and Comparison of Language Workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 139–148, New York, NY, USA, 2010. ACM.

[42] MetaCase-yhtiön kotisivu. http://www.metacase.com/. Viitattu 24.12.2012.

[43] Scratchin kotisivu. http://scratch.mit.edu/. Viitattu 26.12.2014.

[44] The Mjølner System, kotisivu. http://www.daimi.au.dk/~beta/mjolner_system/. Viitattu 25.12.2015.

[45] The Mjølner System, peruskirjastot. http://www.cs.au.dk/~beta/mjolner_system/basiclib.html. Viitattu 25.12.2015.

[46] Sif-editorin ominaisuudet. http://www.cs.au.dk/~beta/Manuals/latest/mjolner-overview/mjolner-overview-4.html. Viitattu 11.9.2015.

[47] Miika Mäki. Tuottavuuden parantuminen Spring Roon avulla. Pro gradu – tutkielma, Jyväskylän yliopisto, 2012.

[48] Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the Effectiveness of a New Instructional Approach. *SIGCSE Bull.*, 36(1):75–79, March 2004.

[49] MPS:n kotisivu. http://www.jetbrains.com/mps/. Viitattu 24.12.2012.

[50] MPS-editorin usein kysytyt kysymykset. https://confluence.jetbrains.com/display/MPSD32/FAQ. Viitattu 1.5.2015.

[51] http://confluence.jetbrains.com/display/MPS/MPS+Frequently+Asked+Questions#MPSFrequentlyAskedQuestions-stringarrayentering. Viitattu 10.1.2014.

[52] Testaaminen MPS:ssä. https://confluence.jetbrains.com/display/MPSD32/Testing+languages. Viitattu 2.10.2015.

[53] Kristen Nygaard, Ole Lehrmann Madsen, Ole Lehrmann Madsen, and Birger Møller-pedersen. Object-oriented Programming in the BETA Programming Language, 1993.

[54] William F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[55] Kirill Osenkov. Designing, implementing and integrating a structured C# code editor. Master's thesis, Brandenburg University of Technology, 2007.

[56] Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. Through the Looking Glass: Teaching CS0 with Alice. *SIGCSE Bull.*, 39(1):213–217, March 2007.

[57] Python-kielen kotisivu. `https://www.python.org/`. Viitattu 30.9.2014.

[58] Trygve Reenskaug. Models - Views - Controllers. Technical report, Technical Note, Xerox Parc, 1979.

[59] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, November 2009.

[60] Dennis Ritchie. An incomplete history of the QED Text Editor. `https://www.bell-labs.com/usr/dmr/www/qed.html`. Viitattu 17.12.2015.

[61] Ruby-kielen kotisivu. `https://www.ruby-lang.org/en/`. Viitattu 11.5.2015.

[62] Jon Schwartz, Jonah Stagner, and Walt Morrison. Kid's Programming Language (KPL). In *ACM SIGGRAPH 2006 Educators Program*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[63] Scratched, Scratch-opetuksen yhteisö. `http://scratched.gse.harvard.edu/`. Viitattu 26.12.2014.

[64] Diomidis Spinellis. Notable design patterns for domain-specific languages, 2001.

[65] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[66] Subtextin kotisivu. `http://www.subtext-lang.org/`. Viitattu 14.7.2015.

[67] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981.

[68] Martin P. Ward. Language Oriented Programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.