Risto Peränen

Olio-ohjelmoinnin periaatteita ja Wolf-kielen hahmotelma

Tietotekniikan
pro gradu -tutkielma
28. marraskuuta 2007

JYVÄSKYLÄN YLIOPISTO
TIETOTEKNIIKAN LAITOS

Jyväskylä

Tekijä: Risto Peränen
Yhteystiedot: risto.peranen@jyu.fi
Työn nimi: Olio-ohjelmoinnin periaatteita ja Wolf-kielen hahmotelma
Title in English: Some Fundamental Principles of Object-Oriented Programming and Draft for Wolf-language
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 150
Tiivistelmä: Tutkimus käsittelee olio-ohjelmoinnin periaatteita ja hahmottaa Wolf-juontokielen, joka on suunniteltu erityisesti pelien kehittämiseen.
English abstract: Study focuses on some of the fundamental principles of the object-oriented programming and gives draft for Wolf-language. Wolf is a scripting language designed for game-development.
Avainsanat: Wolf-juontokieli, metaohjelmointi, olio-ohjelmointi, kääntäjä
Keywords: Wolf scripting language, metaprogramming, object-oriented programming,compilers

Sisältö

Kuvat

Listaukset

Taulukot

Esipuhe

Graduni tutkimusongelmana on suunnitella oliokieli, joka soveltuu pelien ohjaamiseen ja laajentamiseen. Tutkielma jakautuu osiin seuraavasti:

- Ensimmäinen osa kuvaa olio-ohjelmointia yleisellä tasolla. Mukana on myös hieman kritiikkiä olio-ohjelmoinnin nykykäytännöistä,
- Toinen osa keskittyy metaohjelmoinnin kuvaamiseen.
- Kolmas osa kuvaa Wolf-kielen kieliopin sekä luokan ja olion rakenteen,
- Kolmas osa hahmottaa toteutuksen kirjastolle, jolla voi Wolf-kieltä voi laajentaa C++-kielen avulla ja
- Neljäs osa antaa karkean kuvauksen Attribute-kirjaston toteutuksesta,
- Viides osa analysoi tuloksia ja kasaa loppumietteitä tutkielmasta.

Tämä tutkielma ei olisi valmistunut ilman apuvoimia. Jyväskylän yliopistosta haluan kiittää Antti-Juhania ja Jonne Itkosta, jotka tarjosivat alkuvuosina tukea kielen tutkimisessa ja kehittämisessä. Myös ohjaani Jarmo Ernvall ansaitsee kiitoksen nopeasta palautteesta, joka auttoi korjaamaan virheet tekstistä nopeammin. Työelämän puolelta haluan kiittää kollegoitani, jotka jaksoivat kuunnella pohdintojani kielen ominaisuuksista – erityisesti Erkki Häkkistä, Juha Perälää, Sampo Luukkosta, Kimmo Varista ja Jan Lapinkatajaa. Matti Katila löysi tekstistä kirjoitusvirheitä, joten kiitän myös häntä niiden löytämisestä. Matti Rintala antoi KC++-kirjaston tutkittavakseni ja ansaitsee lämpimät kiitokset ystävällisyydestään.

Suurin kiitos kuuluu kuitenkin Jenni Koskenkorvalle, joka jaksoi kannustaa myös vaikeina hetkinä.

Sanasto

JNI Java Native Interface. Rajapinta Java-kielen ja C++-kielen välillä.

Näkymä Katso kappale 12.5.

Näkökulma Katso kappale 12

Näkökulmaohjelmointi Ohjelmointiparadigma, joka keskittyy erityisesti näkökulmien hahmottamiseen.

Piirre Englanninkielisessä materiaalissa käytetään termia Trait. Katso luku 11. Piirteitä käytetään kuvaamaan myös luokan metodeja ja muuttujia, mutta tässä tekstissä piirteellä on oma erikoismerkityksensä.

Piirreohjelmointi Ohjelmointiparadigma, joka keskittyy erityisesti piirteiden hahmottamiseen.

self Wolf-kielen erikoismuuttuja, jolla viitataan muuttujaan itseensä. Semanttisesti samanlainen kuin C++- ja Java-kielen this-muuttuja.

super Wolf-kielen erikoismuuttuja, jolla viitataan muuttujan yläolioon. Semanttisesti samanlainen kuin Java-kielen super-muuttuja.

template C++-kielen metaluokka.

Metaluokka luokka, joka kuvaa luokan.

B <: A Luokka B, joka perii luokan A.

Tyyppilista Staattinen lista tyypeistä.

Tyyppivektori Staattinen lista vektoreista.

API Ohjelmointirajapinta. Käyttöliittymä, jolla kehittäjät voivat kontrolloida ja laajentaa ohjelmaa.

Sisältö

Osa I

Olio-ohjelmoinnin teoriaa

1 Tutkimusaiheen taustaa ja rajausta

Tämä kappale kertoo taustaa tutkimusongelmalle ja rajaa tutkimusaiheen.

1.1 Wolf-kielen taustaa

Ennen Wolf-kielen suunnittelun aloittamista, kirjoittaja kehitti sumean logiikan editoria, jolla pystyi käsittelemään sumeita joukkoja ja laskemaan vasteita syötteiden perusteella vuonna 2002. Projekti oli opettavainen, mutta käytti kovakoodausta. Kirjoittaja teki editoria varten yksinkertaisen kieliopin ja toteutuksen jäsentäjälle, joka luki sumean joukon kuvaukset tiedostosta ja liitti sen muuhun toimintaan.

Editori toimi kohtalaisen hyvin, mutta huonon suunnittelun ja vielä huonomman ylläpidon takia projekti ei edennyt. Editorin luominen opetti silti jäsentäjien tekemistä ja asetusten yhdistämistä tehokkaasti kontrolloivaan koodiin.

Ongelmaksi tuli, että asetusten lukeminen ei ollut tarpeeksi: itse logiikka kontrollereihin piti edelleen kirjoittaa itse. Eräs versio sisälsi jo mahdollisuuden luoda kontrollirakenteita ja luoda oikeita kontrolleja, mutta editori tuli liian vaikeaksi ylläpitää ja ymmärtää.

Editorin tekeminen antoi kuitenkin kipinän suunnitella omaa kieltä. Eräs suurimmista syistä oli kirjoittajan silloinen inho C++-kieleen ja sen kankeuteen. Kielen suunnittelu alkoi pikkuhiljaa, kun ideoita tarttui toisiinsa ja ne alkoivat muodostaa suurempaa kokonaisuutta. Peliprojekti IGIOS [19] ja työelämä vaikuttivat omalta osaltaan kielen suunnitteluun.

Wolf-kielen oli tarkoitus olla pieni kieli, joka toimisi yksinkertaisen virtuaalikoneen päällä. Kielen oli tarkoitus olla valmis parin vuoden kuluttua aloittamisesta – eli vuonna 2004. Ongelmaksi tuli kuitenkin, että ruokahalu alkoi kasvaa syödessä ja kieli alkoi laajenemaan liian geneeriseen suuntaan. Kiireet ja lukuisat epäonnistumiset lykkäsivät kielen valmistumista lisää. Kirjoittaja halusi tehdä kielestä korkeamman tason kielen kuin C++ ja vähemmän byrokraattisen kuin Java. Toisaalta, Python-ohjelmointikieli taas jätti useasti tarkastamatta asioita, joita kirjoittaja silloin piti tärkeänä tarkistaa.

1.2 Tutkimusaihe

Juontokieli on kieli, joka käännetään ja suoritetaan ajonaikana. Hyvä esimerkki juontokielestä on on Python-kieli [16], joka ohjaa korkealla tasolla C-kielisiä komponentteja. Wolf on juontokieli, joka on suunniteltu erityisesti pelien ohjaamiseen. Peleissä juontokielillä ohjataan tyypillisesti pelien käsikirjoitusta [7]. Täten alkuperäinen idea Wolf-kielelle lukea käyttäjän konfigurointitiedostoja ei ole muuttunut minnekään, vaikka kieli on monimutkaistunut ja monipuolistunut.

Pelien kääntäminen on sitä hitaampaa, mitä laajempi projekti on. Tämä aiheuttaa ongelmia pelinkehittäjille, koska parin vakion muuttaminen vaatii usean minuutin odotuksen, kun kääntäjä kääntää peliprojektin lähdekoodeja. Viitteen [7, sivu 251] mukaan peleissä juontokieliä käytetään, koska ne

- mahdollistavat helpon tavan alustaa ja käsitellä pelimaailman dataa,
- säästävät kehittäjien aikaa ja lisäävät tuottavuutta, koska juontokielten kääntäminen on nopeampaa
- kannustavat luovuuteen, koska juontokielet toimivat abstraktimmalla tasolla kuin C++, ja
- sallivat mahdollisuuden pelaajille laajentaa ja muokata peliä.

Wolf-kieli pyrkii auttamaan kehittäjiä näiden ongelmien kanssa. Se on tyypitetty kieli, mutta toimii silti kohtalaisen abstraktilla tasolla. Wolf-kielellä voi luoda käsikirjoituksen tai laajennoksia peleihin kohtalaisen helposti mutta turvallisesti. Alkuperäinen toiminta, jossa käyttäjän tekemät asetukset luettiin, on vain siirtynyt käyttämään myös kontrollilogiikkaa.

Tutkimusongelmana on hahmottaa Wolf-kieli, jolla voi ohjata pelejä. Tarkoituksena on vain ja ainoastaan hahmottaa kielen ominaisuuksia ja pohtia, miten kieltä voi laajentaa C++-kielellä tehdyillä komponenteilla. Valitettavasti tutkimuksen suppea laajuus estää määrittämästä standardikirjastoja tai edes pintapuolista hahmotusta, miten kääntäjä toteutetaan. Myöskään virtuaalikoneen rakenteeseen ei oteta kantaa tässä tutkielmassa.

2 Olio-ohjelmoinnin päämäärät ja periaatteet

Tämän luvun tarkoituksena on antaa hyvin suppea katsaus "oikeaan"olio-ohjelmointiin. Ohjenuorana on ollut Boochin [4, Consepts] neljä kultaista sääntöä:

- abstraktio,
- toteutuksen piilotus,
- modulaarisuus ja
- hierarkia

2.1 Luokka ja Olio

Logiikan peruskursseilla käytetään usein esimerkkiä:

> Kaikki ihmiset ovat kuolevaisia.
> Socrates on ihminen.
> Sockates on kuolevainen.

Esimerkkiä voi käyttää myös olio-ohjelmoinnin esittelyyn. Esimerkissä ihminen on luokka, jonka kaikki instanssit eli oliot ovat kuolevaisia. Luokalla ja oliolla voidaan ajatella olevan tietty yhteys joukko-oppiin matematiikassa. Matematiikassa voidaan määrätä joukko, jolla on joukko ominaisuuksia. Jokaisella joukon jäsenellä on ainakin ne ominaisuudet, jotka joukko määrää. Voidaankin luonnehtia karkea yleistys, että $socrates \in Human$. Korostettakoon, että tämä on pelkkä yleistys ja että oliot kuuluvat joukkoon vain matemaattisesti ajateltuna.

Ennen olio-ohjelmointikielten kehitystä oliota mallinnettiin yleisesti abstrakteilla tyypeillä[1]. Tyypissä kuvattiin matemaattisesti tilakone, jossa oli muuttujia eli dataa ja funktioita, joilla muutettiin tilaa. Kun oliokielet alkoivat yleistyä, nämä mallit pystyttiin kirjoittamaan suoraan koodina esimerkiksi Simula-ohjelmointikielellä [6, sivu 5].

[1] Englanninkielinen termi abstraktille datatyypille on Abstract Data Type (ADT)

2.2 Olio ja viestinvälitys

Nykyisin oliokielillä on melko helppo luoda uusia *luokkia*. Myös olio-suunnittelun työkalut keskittyvät useimmiten helpottamaan luokkien luomista. Olio-ohjelmoinnin kulmakivi on kuitenkin *olioiden* viestinvälitys oliolta toiselle. Alan Holubin mukaan hyvä olio-ohjelmisto keskittyy ennen kaikkea dynaamiseen toimintaan ja viestinvälitykseen luokkien välillä eli dynaamisen mallin miettimiseen [23]. Myös Timothy Budd painottaa viestinvälitystä olioiden välillä [9].

Luokat ilman olioita kuvaavat vain, miten ohjelmisto varaa muistia olioille ja mitä metodeja olioon kuuluu. Suunnitellessa tulisikin miettiä ensisijaisesti luokkien toiminnallisuutta ja vasta sitten datan säilytystä ja käsittelyä [23].

Lyhyesti sanoen, olio on olevainen, joka saa viestejä, käsittelee niitä ja lähettää viestejä takaisin kutsujalle tai toisille olioille.

2.2.1 Olio- ja luokkapohjaiset kielet

Miksi sitten tarvitaan luokkia, jos kommunikointi olioiden välillä on tärkeämpää kuin luokkien määrittely? Vastaus on hieman yllättäen, että luokkia ei *välttämättä* tarvita ollenkaan. Esimerkiksi Kevo-ohjelmointikieli [49] hylkäsi luokan käsitteen kokonaan ja käyttää vain olioita. Olio-kieliä, joissa ei ole olemassa luokkia, sanotaan oliopohjaisiksi kieliksi, ja vastaavasti kieliä, joissa oliot luodaan luokkien pohjalta, luokkapohjaisiksi kieliksi.

Oliopohjaisissa kielissä olioista pyydetään kopio, jonka toimintaa muutetaan halutunlaiseksi. Sen jälkeen kopioita käytetään alkuperäisen olion sijasta. Tämä oli hyvin oliomainen tapa käyttää olioita.

Kevon lisäksi muita esimerkkejä oliopohjaisista kielistä on hyvin vähän. Pelkästään olioihin perustuvat kielet eivät koskaan saavuttaneet kovin suurta suosiota [6]. Eräs syy oli, että hierarkian hahmotus oli huomattavan paljon vaikeampaa ja hyvien ohjelmien tekeminen vaati tiukempaa kurinalaisuutta. Muutokset yhteen olioon eivät muuta kaikkia olioita. Luokkia käyttävissä kielissä on hyvänä puolena, että luokkaa muuttamalla saa muokattua kaikkia sen instansseja kerralla. Toinen syy oli, että pelkästään olioita käyttävät kielet ovat yleisesti hitaampia, koska niiden toiminnan optimointi on vaikeaa kääntäjän kannalta. Luokkia käyttävien oliokielten kääntäjiä on helpompi tehdä tehokkaiksi, koska käännösaikana on enemmän tietoa luokasta ja siten myös sen luomista olioista. Kolmas syy oli, että olio-ohjelmoijat eivät muuta tapojaan kovin helposti. Useimmat olio-ohjelmoijat käyttivät prototyy-

pin määritteleviä olioita kuten luokkia. On kuitenkin huomattava, että sekä olio-pohjaisilla että luokka-pohjaisilla ohjelmointikielillä voi luoda miltei samanlaista toimintaa. Lopuksi todettakoon, että Wolf on kuitenkin perinteinen, luokkapohjainen kieli.

Tämä teos käsittelee tästä eteenpäin pelkästään luokka-pohjaista lähestymistapaa olio-ohjelmointiin, joten voimme tehdä seuraavan oletuksen:

> Jokainen olio on jonkin luokan instanssi [9].

2.3 Olio on toiminnallisuutta

Olio ohjelmointi on – vielä kerran – viestien välittämistä toisille olioille. Olio-ohjelmointi mallintaa usein arkielämää ja jäsentää ongelman erilaisiin kokonaisuuksiin. Koska luokat ovat ihanteellisessa maailmassa suljettuja ulkopuolisilta, on mahdollista tehdä mitä tahansa muutoksia luokan sisälle ilman, että muiden olioiden tarvitsee välittää muutoksista. Ainut, mistä tarvitsee välittää, on olioiden tarjoama toiminta ja että se pysyy samanlaisena.

On hyvin vaikea päättää, onko jokin ohjelmisto olio-orientoitunut vai ei, mutta artikkelissa [24] Holub antoi melko hyvän nyrkkisäännön:

1. Kaikki olion tiedot ovat yksityistä eikä muut oliot pääse muuttamaan sitä suoraan. (Tämä sääntö pätee myös toteutuksen yksityiskohtiin eikä pelkästään dataan)

2. Jäsenmetodit get ja set-metodit, joilla palautetaan luokan jäsenmuuttujia, ovat vain monimutkainen tapa julistaa luokan tietoja julkisiksi. Siten ne rikkovat olioajattelun perusideaa.

3. Olioilta ei pyydetä tietoa vaan palveluja.

4. On oltava mahdollista tehdä isojakin muutoksia olion toteutukseen ilman, että muutokset vaikuttavat vain luokkaan, joka määrittää olion.

5. Kaikien olioiden on tarjottava käyttäjille käyttöliittymä.

Holub korostaa, että ohjelma ei ole välttämättä huono, vaikka se ei täytetään edellä olevia ehtoja. Se ei vain noudata olio-ohjelmoinnin periaatteita orjallisesti.

Toisaalta, Stroustrup keskittyy abstraktioon kuin viestin välitykseen olio-ohjelmointia määritellessä [46]. Tämä johtuu enemmän näkemyseroista: Smalltalk ja muut

korkean tason kielet keskittyvät olioiden kommunikointiin mutta matalan tason kielet abstraktioon ja tyyppien hierarkiaan.

Wolf-kieli on korkean tason kieli, joten viestin välitys olioiden välillä on keskeinen osa kieltä. Wolf-kieltä ei ole tarkoitus käyttää systeemiohjelmointiin kuten C++-kieltä, joten rajaus on järkevä.

2.4 Esimerkki: kovalevyn tilaaminen

Ajatellaan esimerkkiä, jossa tilaan uuden kovalevyn tietokoneeseeni. Annan myyjälle merkin, mallin ja muut tiedot kovalevystä, jonka haluan ja hän laittaa levyn tilaukseen. Parin viikon päästä postista tulee ilmoitus, että voin hakea tilauksen postiennakolla.

Esimerkissä on olennaisinta, että minun ei tarvitse tietää, miten kauppias hankkii kovalevynsä. Hänellä voi olla varastollinen pelkkiä kovalevyjä, joista nappaa sen postiin tai hän voi tilata sen jostain muualta. Minun ei tarvitse tietää yksityiskohtia tilauksesta. Riittää, että kauppias tekee työnsä ja laittaa haluamani kovalevyn postiin. Yleensä minun ei kannata tuhlata aikaani sellaisten yksityiskohtien miettimiseen, jotka ovat kauppiaan vastuulla.

Edelleen, kauppias ei itse tuo kovalevyä minulle vaan laittaa sen postiin, joka vie paketin perille ja hoitaa laskutuksen minulta. Kauppiaan ei edelleenkään tarvitse itse tietää, miten posti kuljettaa paketin tai haaskata itse aikaa pakettien toimittamiseen tai logistiikan käsittelyyn.

Samoin postiennakko on tuttu ja turvallinen tapa asioida minulle, mikä vähentää epäselvyyksiä tilauksen yhteydessä.

Lisäksi, esimerkissämme on paljon epävarmoja tekijöitä, kuten arkielämässä yleensäkin: kovalevyä ei välttämättä saada ajoissa, sen hinta on kallistunut, posti hukkaa paketin, kovalevyssä on valmistusvika ja niin edelleen.

Perusideana on, että kommunikoin kauppiaalle. Minun vastuulla on antaa kovalevyn tekniset tiedot oikein. Kauppias ottaa tilauksen vastaan ja kommunikoi minulle, onko hän ymmärtänyt tilauksen oikein ja antaa muuta tietoa, jonka hän uskoo olevan tarpeellista tilauksen kannalta. Hänen vastuullaan on ymmärtää tilaus oikein, kysyä tarkennuksia, jos tilaus on epäselvä, ja huolehtia, että paketti lähtee postiin ajallaan. Postin tehtävänä on huolehtia paketti perille ja lähettää tieto tilauksen saapumisesta minulle.

2.5 Yhteenvetoa luvusta 2

- Jokainen olio on jonkin luokan instanssi luokkia käyttävissä kielissä.
- Olio-suunnittelu ei ole pelkästään luokkien ja rajapintojen eli staattisen mallin miettimistä.
- Olio-ohjelmoinnin kulmakivi on, että oliot kommunikoivat keskenään eivätkä muuta toistensa tilaa suoraan.

3 Tyypitys

Tämä luku kuvaa tyypityksen ja eroja tyypityksessä eri kielten välillä.

3.1 Heikko ja vahva tyypitys

Tyyppivirheellä tarkoitetaan tilannetta, jossa muistialuetta käsitellään väärän tyyppisenä – esimerkiksi kone käsittelee kokonaislukua tietueena. Kielessä on heikko tyypitys, jos kielellä voi luoda tyyppivirheen. Jos kielessä ei ole heikkoa tyypitystä, siinä on vahva tyypitys.

Kuvassa 3.1 esitetty C-kielinen ohjelma on selkeästi väärin mutta kääntäjä ei varoita virheestä millään tavalla.

Listaus 3.1: Tyyppivirhe C-kielessä

```
1
2  struct MyStruct {
3    int x;
4    char name[4]
5  };
6
7  void my_fun(param)
8    MyStruct *param;
9  {
10   param->x = 69;
11   strcpy(param->name,"test");
12 }
13
14 int main(void) {
15   /* assumes sizeof(int) = 2*sizeof(char) */
16   int first;
17   int second;
18   int third;
19
20   first = 1;
21   second = 2;
22   third = 3;
23
```

```
24    my_fun(&first);
25    /* further call are likely to fail since
26     * second and third are corrupted
27     */
28
29    return 0;
30  }
```

Ohjelmaa ajaessa aliohjelmalle my_fun annetaan parametriksi osoite kokonaislukuun. Aliohjelma kuitenkin olettaa parametrin olevan tyyppiä MyStruct ja käsittelee muistia sen mukaan. Kutsu aiheuttaa muuttujien second ja third kirjoittumisen yli, koska ne ovat pinossa first-muuttujan jälkeen. Kutsun jälkeen muisti on selkeästi korruptoitunut. Ohjelma ei valitettavasti kaadu ja luo core-tiedostoa vaan jatkaa toimintaansa. Seuraavat kutsut muihin aliohjelmiin, joissa luotetaan muuttujien second tai third arvoihin, epäonnistuvat melko varmasti tai korruptoivat muistia lisää.

Tyyppivirheet ovat virheistä ehkä vaikeimpia havaita, koska ne aiheuttavat yllättävää toimintaa epäsäännöllisesti ja korruptoivat tietoja. Kirjoittaja on kerran törmännyt ongelmaan, jossa ohjelma käyttäytyi omituisesti. Oliot eivät toimineet lainkaan säännönmukaisesti ja virhettä oli mahdoton toistaa säännönmukaisesti. Lopulta syyksi paljastui tyyppivirhe, joka korruptoi muistia mutta joka ei kaatanut ohjelmaa. Virheen jäljitys oli hyvin vaikeaa ja opetti kirjoittajaa olla optimoimatta kohdissa, joissa se voi olla vaarallista.

Esimerkkejä heikosti tyypitetyistä kielistä ovat C-kieli ja assembler. Uudemmissa kielistä on kaikissa enemmän tai vähemmän vahva tyypitys.

3.2 Staattinen tyypitys

Staattinen tyypitys tarkoittaa, että kääntäjä tarkistaa ohjelman tyyppivirheiden varalta käännösvaiheessa. Tarkistusten tehtävänä on estää, ettei esimerkin 3.1 kaltaisia tilanteita syntyisi. Esimerkkejä staattisesti tyypitetyistä kielistä ovat Pascal, C++, Eiffel ja Java – joskin Java-kielessä tyyppejä tarkistetaan paljon myös ajon aikana.

Esimerkissä 3.2 on tehty vain pieni muutos aliohjelman my_fun parametreihin. Koska first ei ole MyStruct-tyyppinen muuttuja ja koska sitä ei voi muuttaa suoraan MyStruct-tyyppiseksi, kääntäjä ilmoittaa käännösvirheestä. Kehittäjä huomaa tästä jo kehitysvaiheessa, että ohjelma on viallinen.

Listaus 3.2: C++-kielessä on vahva tyypitys

```
1  struct MyStruct {
2    int x;
3    char name[4]
4  };
5
6  void my_fun(struct MyStruct *param;)
7  {
8    param->x = 69;
9    strcpy(param->name,"test");
10 }
11
12 int main(void) {
13   /* assumes sizeof(int) = 2*sizeof(char) */
14   int first;
15   int second;
16   int third;
17
18   first = 1;
19   second = 2;
20   third = 3;
21
22   /* won't compile */
23   my_fun(&first);
24   return 0;
25 }
```

Staattinen tyypitys esti ohjelmoijien pahimpia ja klassisimpia virheitä tehokkaasti ja lisäsi siten ohjelmistojen luotettavuutta. Myös C-kielen nykyisissä versioissa on mahdollista kirjoittaa parametrien tyypit kuten C++-kielessä ja saada virheilmoituksia tyyppivirheistä. Aliohjelmia, joissa parametrien tyyppiä ei ole määrätty, ei juuri käytetä kuin harvinaisissa poikkeustilanteissa.

3.2.1 Vakaa staattinen tyypitys

Ihanteellisessa tilanteessa kääntäjä havaitsisi kaikki mahdolliset tyyppivirheet jo käännösaikana. Tälläisestä tyypityksestä käytetään nimitystä Sound type system englanninkielisessä materiaalissa. Kirjoittajan suomennos termille on vakaa staattinen tyypitys.

Valitettavasti tämä ei ole ollut esimerkiksi Javan kohdalla mahdollinen. Säiliöluokat ja esimerkiksi vertailuoperaattorit vaativat käytännössä tyypin muuttami-

sen toiseen, joka joskus aiheuttaa tyyppivirheen ajonaikana. C++:lla puolestaan on mahdollista tehdä heikosta tyypityksestä johtuvia virheitä, kuten C:ssäkin, mutta ne vaativat hieman enemmän työtä.

Eiffel-ohjelmointikieli tarjosi monia hyödyllisiä ratkaisuja tyypitykseen, mutta sekään ei pystynyt havaitsemaan kaikkia tyyppivirheitä jo käännösaikana [6, sivut 60-69].

Luvussa 4 perusteellaan, miksi Wolf-kieleen ei ole mahdollista luoda vakaata tyypitystä.

3.3 Dynaaminen tyypitys

Dynaaminen tyypitys tarkoittaa, että kääntäjä ei tarkista tyyppivirheitä käännöksen aikana vaan tarkastukset tehdään ohjelmaa suorittaessa. On huomattava, että määritelmän mukaan myös dynaamisesti tyypitetyt kielet ovat vahvasti tyypitettyjä, jos niillä ei voi korruptoida muistia tyyppivirheen takia.

Dynaamisesti tyypitetyissä kielissä ei muuttujia tai parametreja määrittäessä tarvitse välittää muuttujan tyypistä vaan tyyppi määräytyy vasta ajon aikana muuttujaa alustaessa. Olion metodia kutsuessa kutsutaan oikea metodi vasta ajon aikana. Tyyppivirheiden havaitseminen ei kuitenkaan tarkoita heikkoa tyypitystä, koska tyyppivirhe ei voi kaataa ohjelmaa suoraan tai korruptoida muistia.

Python kieli on hyvä esimerkki dynaamisesti tyypitetystä kielestä [16]. Se kuuluu juontokieliin eikä siinä ole tyyppitarkastuksia juuri lainkaan käännösaikana. Pythonissa on vahva tyypitys, mutta tyyppien tarkastamiset tehdään vasta ajonaikana. Kielessä ei ole mahdollista luoda esimerkin 3.1 kaltaista vaarallista tilannetta, jossa muisti olisi korruptoitunut, vaan Pythonin virtuaalikone havaitsee tyyppivirheen ja heittää poikkeuksen.

Jos ohjelmassa on virheellinen kutsu, joka rikkoo tyypityssääntöjä, Pythonin ajoympäristö generoi tyyppivirheen *ajon aikana*. Tätä toimintaa kuvataan esimerkissä 3.3. Kun my_fun-funktiota kutsutaan vääräntyyppisillä parametreilla, Pythonin virtuaalikone heittää poikkeuksen ajonaikana, jonka voi käsitellä poikkeuksena ajon aikana.

Listaus 3.3: Python on dynaamisesti tyypitetty kieli

```
1 class MyStruct:
2         a = 0
3         b = ""
```

```
4
5  def my_fun(myStruct):
6          myStruct.a = 69
7          myStruct.b = "test"
8
9  first = 1
10 second = 2
11 third = 3
12 my_fun(first) #type error
```

Hyvänä puolena dynaamisessa tyypityksessä on koodin yksinkertaistuminen. Dynaamisesti tyypetyissä kielissä ongelmaa voi käsitellä korkeammalla tasolla ja tehdä korkean tason optimointeja ohjelmaan kohtalaisen helposti, jotka staattisesti tyypitetyissä kielissä vaatisivat huomattavan määrän työtä. Toteuttamiseen käytettävästä ajasta suurempi osa menee itse ongelman ratkaisuun eikä lähdekoodin kirjoittamiseen, joten ohjelmiston saa tuotettua nopeammin kuin staattisesti tyypitetyillä kielillä.

Huonona puolena on, että dynaamisesti tyypitetyillä kielillä luotu ohjelma on yleensä hitaampi suorittaa. Kääntäjä ei voi optimoida koodia kovin paljoa, koska käännösvaiheessa ei ole tietoa oikeasta tyypistä. Koska haluttu funktio pitää etsiä ajon aikana, on ohjelman suorittaminen hitaampaa. Lisäksi metodi voi puuttua kokonaan, joka johtaa virheeseen ajon aikana.

Dynaamisesti tyypitettyjen kielten hitaus on kuitenkin pelkkä nyrkkisääntö ja perustuu nykyisiin toteutuksiin. Just In Time-käännöstekniikat optimoivat koodia ohjelman suorituksen aikana ja optimointeja voi tehdä korkeammalla tasolla. Tämä vaatii kuitenkin kontrolli- ja datavuon analysoimista ja jonkinlaista evalointia käännösaikana – kuten Lisp-ohjelmointikieli tekee.

3.4 Vahva tyypitys ja vahva testaus

Yleisesti staattisesti tyypitettyjä kieliä on pidetty luotettavampina kuin dynaamisesti tyypitettyjä kieliä. Brucen listasi kirjassaan muutamia syitä, miksi staattisesti tyypitetyt kielet ovat varmatoimisempia [6, sivut 8-9]. Hänen mukaansa staattisesti tyypitetyt kielet ovat suositeltavia seuraavista syistä:

1. Ne tuottavat aiemmin tarkempaa tietoa ohjelmoijan virheistä.

2. Ne pienentävät koodin kokoa, koska tyyppitarkastuksia ei tarvitse jättää lo-

pulliseen ohjelmaan.

3. Ne tarjoavat dokumentaatiota rajapinnoista, luokista ja komponenteista.

4. Kääntäjä on helpompi optimoida.

Jos asiaa katsotaan uuden, ketteriin menetelmiin keskittyneen ohjelmistoteollisuuden kannalta, staattiset kielet ovat siirtymässä puolustusasemiin. Pitkään staattisilla kielillä ohjelmoineet kehittäjät, kuten Robert C. Martin, puoltavat dynaamisesti tyypitettyjen kielten käyttöä [32].

Martin uskoo, että testilähtöinen kehitys ja ketterät kehitysmallit, joissa jokainen koodinpätkä testataan, ovat vieneet staattisesti tyypitettyjen kielten argumenteilta viime vuosina hieman pohjaa [32]. Ketterien menetelmien eräs kulmakiviä on testaaminen projektin jokaisessa vaiheessa. Staattinen tyypitys on vain yksi testi, joka auttaa luomaan turvallisia, luotettavia ohjelmia – mutta tyyppitarkastus on silti vain yksi tarkastus muiden testien joukossa. Martinin mukaan nyrkkisääntönä voidaan pitää, että mitä enemmän projekti käyttää ketteriä menetelmiä sitä vähemmän staattisesta tyypityksestä erikseen on hyötyä [33]. Staattisella tyypityksellä on kuitenkin aina korkea hinta, koska kehittäjät joutuvat käyttämään enemmän aikaa komponenttien luomiseen. Toisaalta on huomattava, että Brucen viitteessä [6] esittämillä tekniikoilla staattisesti tyypitettyihin kieliin voisi luoda joustavan tyypityksen. Martin kohdistaa kritiikkinsä nykyisiin tyypityksiin staattisesti tyypitetyissä kielissä, joita myös Bruce kritisoi [6, sivut 35-48].

Brucen kolmas argumentti, jonka mukaan staattisesti tyypitetyt kielet tarjoaisivat enemmän dokumentaatiota, ei ole täysin aukoton. Jos ohjelmistoa kehitetään ketterillä menetelmillä – kuten XP – koodin jokainen toiminnallisuus testataan ainakin yksikkötesteillä. Yksikkötestit tarjoavat jopa enemmän dokumentaatiota ja tietoa luokkien toiminnasta kuin pelkät rajapinnat. Testilähtöinen kehitys sulkee myös ensimmäisen argumentin pois, koska kehittäjän tekemät virheet havaitaan jo kehittäjän tekemissä testeissä tai viimeistään yksikkötesteissä.

Koodin käyttö dokumentaationa on kuitenkin hyödyllinen ominaisuus laajoja moduuleita suunniteltaessa ja laajentaessa. Useat dynaamiset kielet – esimerkiksi PHP ja Ruby – ovat lisänneet jonkinlaisen rajapinnan käsitteen juuri tämän takia. Rajapinnat tarjoavat helpon tavan kehittäjille työskennellä. Osa Python-ohjelmoijistakin kaipaisi selkeää rajapinnan määritystä, jotta ohjelmiston laajentaminen olisi helpompaa [15]. Tätä(kään) ongelmaa ei olisi, jos kehittäjät dokumentoisivat työnsä

huolellisimmin ja kertoisivat, mitä rajoituksia ja mahdollisuuksia heidän tekemällään moduulilla on.

Neljäs argumentti, jonka mukaan kääntäjän toimintaa on helpompi optimoida, on kuitenkin hyvin vahva puolustus staattisesti tyypitettyjen kielten puolesta peliympäristössä. Mitä enemmän tyyppejä ja koodin kontrollivuota on kääntäjän tiedossa jo käännöshetkellä, sitä helpompi on suunnitella optimointeja koodiin. Esimerkiksi Python voisi optimoida itseään tehokkaammaksi jo siinä vaiheessa, kun ohjelmaa käännetään tavukoodiksi.

Useimmissa tapauksissa dynaamisesti tyypitettyjen kielten suorituskyky on jo riittävä nykyisissä koneissa, mutta pelit vaativat hyvää suorituskykyä.

3.5 Tyyppipäättely

Tyyppipäättely tarkoittaa, että ohjelmassa on staattinen tyypitys – eli kääntäjä tarkistaa ohjelman tyyppivirheiden varalta – mutta tyyppejä ei tarvitse erikseen kirjoittaa näkyviin.

Ohjelmointikielistä Haskell ja ML käyttävä tyyppipäättelyä tyypityksen tarkastamiseen. Myös D-ohjelmointikielessä on mahdollista käyttää tyyppipäättelyä erityisellä avainsanalla auto, mutta D-kielen tyyppipäättely on alkeellinen [5]. Koska Lisp-ohjelmointikieli käyttää evalointia ohjelmaa suorittaessa, sen voi ajatella käyttävän tyyppipäättelyä ja JIT-käännöstä. Myöskään Lisp-kielessä kehittäjän ei tarvitse kirjoittaa tyyppiä näkyviin, vaan Lisp-tulkki evaluoi komentoja ja tarkistaa tyypin samalla.

Tyyppipäättely antaa hyvät puolet sekä staattisesta tyypityksestä että dynaamisesta tyypityksestä. Valitettavasti tyyppipäättelyä ei ole käytetty kovin laajasti oliokielissä ja voi aiheuttaa ikäviä yllätyksiä kielen kääntäjää toteuttaessa.

3.6 Tyypitys pelien juontokielissä

Nykyään pelien elinkaarta pidennetään usein antamalla pelaajille muokkaustyökalut, joilla kehittää ja muokata peliä haluamakseen. Tunnetuin esimerkki tämäntyyppisestä käyttäjien tekemästä muokkauksesta lienee Counter Strike [10].

Mitä enemmän pelin logiikkaa on skriptin varassa, sitä monipuolisempia muutoksia pelaajat voivat tehdä peliin mutta sitä huonompi suorituskyky voi olla. Pelit vaativat paljon suorituskykyä ja skriptienkin on suoriuduttava tehtävistään no-

peasti. Koska skriptinkin on oltava mahdollisimman optimaalista, useimmat monet peleihin tarkoitetut juontokielet – kuten UnrealScript [48] – ovat staattisesti tyypitettyjä.

Toisaalta, koska juontokielien tarkoitus on olla abstrakeja, Lua ja Python ovat kuitenkin alkaneet saada suosiota pelien ohjauskielinä, koska niillä saa tehtyä halutun toiminnan nopeasti kasaan ja kohtalaisen pienellä määrällä koodia ilman tyypityksen miettimistä.

Vaikka skriptiä ajetaan useimmiten jonkinlaisella virtuaalikoneella, staattinen tyypitys auttaa kääntäjää tekemään parempaa koodia niille. Java-kieltä ajetaan virtuaalikoneen päällä, mutta sen suorituskyky on kohtalaisen hyvä nykyisin – suurelta osin juuri staattisen tyypityksen ansiosta.

3.6.1 Wolfin tyypitys

Wolf-kielen tyypityksen valinta staattisen ja dynaamisen väliltä oli ehkä kielen vaikeimpia suunnittelupäätöksiä. Kirjoittajan pitkä tausta matematiikan parissa ja C++-kielen parissa puolsi staattisen tyypityksen käyttöä. Toisaalta, ohjelmointi Python-kielellä on selkeästi miellyttävämpää kuin Javalla tai C++:lla.

Kirjoittaja valitsi Wolf-kielen käyttämään staattista tyypitystä, koska se on kielen kehittäjän kannalta helpoin toteuttaa.

Toinen syy staattiseen tyypitykseen on, että aikataulu on melko tiukka. Kirjoittaja on hionut Wolf-kieltä suuntaan ja toiseen jo vuodesta 2002. Useimmat kohdat voi kuitenkin toteuttaa myöhemmin joustavammiksi, kun kielen pohja on valmiina.

Kolmas syy staattiseen tyypitykseen on, että kääntäjän toteuttaminen on huomattavasti helpompaa, kun kieli käyttää staattista tyypitystä. Tyypityksen hahmottaminen järkeväksi on siedettävämpää, kun tyypitys on pakollinen.

Paras vaihtoehto on käyttää tyyppipäättelyä, koska se antaa tyypitettyjen kielten "varmuuden"ja suorituskyvyn mutta säilyttää ohjelmoinnin helppouden. Lopullisissa versioissa kielestä Wolf-kieli käyttää tyyppipäättelyä, mutta kirjoittaja ei tunne aihetta tarpeeksi, jotta sen uskaltaisi ottaa mukaan kielen ensimmäisiin toteutuksiin.

3.7 Yhteenvetoa luvusta 3

Oleellisimmat asiat luvusta 3 ovat:

- Heikko tyypitys on erittäin huono ominaisuus,

- Dynaamisesti tyypitetyt kielet ovat vahvasti tyypitettyjä, helpompia käyttää ja suorituskyvyltään huonompia,
- Wolf-kieli käyttää staattista, explisiittistä tyypitystä ja
- lopullinen versio Wolf-kielestä käyttää tyyppipäättelyä.

4 Monimetodit

4.1 Mitä virtuaalisuus tarkoittaa?

Yleisesti olio-kielissä käytetään niin sanottua single-distpatching menetelmää olioiden luomisessa. Luokalle luodaan virtuaalitaulu, joka sisältää osoittimet metodeihin, joita halutaan luokassa käyttää. Kääntäjän tehtävänä on vain etsiä virtuaalitaulusta oikea indeksi metodille, jota käytetään kutsussa. Ajon aikana olion virtuaalitaulusta etsitään indeksin perusteella oikea metodi, jota kutsutaan.

Jokaisella oliolla, joka käyttää virtuaalista perintää, on oltava virtuaalitaulu [47]. Virtuaalitaulun perusteella haetaan oikea funktio, joka suoritetaan. Virtuaalitaulun osoitteet funktioihin voivat vaihtua lapsi- ja yläluokan välillä. Hyvänä puolena on, että käännösaikana indeksi voidaan kuitenkin laskea. Virtuaalisen metodin käyttäminen on siis vain muistiosoitteen hakeminen ja epäsuora metodikutsu, joka säilyttää suorituskyvyn hyvänä. Virtuaalitaulun käyttäminen on O(1) operaatio.

Huonona puolena on, että päättely oikeasta metodista tehdään vain yhden tyypin perusteella käännösaikana. Useissa kielissä – kuten Java – voidaan ylikuormittaa metodeja, mutta ylikuormitus on staattinen eikä dynaaminen. Metodien ylikuormitus aiheuttaa sekaannuksia, joita ei kovin helposti voi ratkaista single dispatch-tekniikalla [6].

4.2 Mitä monimetodit tarkoittavat?

Monimetodeissa oikea metodi päätellään jokaisen parametrin perusteella. Esimerkiksi eri kappaleiden törmäyksiä laskiessa voidaan luoda törmäysolio, joka tarkistaa kappaleiden tyypin, sijainnin ja läpimitan perusteella voivatko ne törmätä vai ei. Kehittäjä voi käyttää abstraktia tyyppiä muodoille ja vain metodit, joissa kappeleen tyypillä on väliä, tarkistaa oikean tyypin.

Tämä on kehittäjälle tehokkaampi työkalu kuin ylikuormitus, jossa lapsiluokassa voi kyllä ylikuormittaa metodeja, mutta yläluokka ei tiedä kuormituksesta mitään. Ajon aikana kutsutaan helposti väärää metodia tai sitten lapsiluokassa on tehtävä ajonaikaista tyyppipäättelyä – eli toteuttamaan monimetodit huonosti. Viittees-

sä [6, sivu 29] asiaa käsitellään syvällisemmin.

Wolf-kielessä monimetodeja voi pitää ylikuormituksena, joka tapahtuu dynaamisesti, koska kirjoittajan kokemuksien mukaan staattinen ylikuormitus ei ole riittävän tehokas. On hyvin tavallista, että isoissa projekteissa joutuu käyttämään erikseen Visitor-suunnittelumallia, joka vastaa oleellisilta osiltaan dynaamista ylikuormitusta. Viitteessä [6, sivut 98-108] käsitellään monimetodeja laajemmin. Oleellinen ero viitteen monimetodeihin ja Wolf-kielen metodeihin on, että Wolf on oliokieli ja viesteillä on vastaanottaja kuten normaaleilla olioillakin. Ainoastaan viestin kuormitus vaihtelee eri tyyppien välillä.

Huonona puolena on, että kääntäjä ei voi enää optimoida koodia niin tehokkaasti kuin single-dispatch tekniikoiden kanssa. Kirjoittaja valitsi kuitenkin monimetodit kieleen staattisen ylikuormituksen sijasta, koska käytännön elämässä parametrienkin tyyppi vaikuttaa valittavan metodin suoritukseen. Toinen syy on, että Wolf on tarkoitettu pelien juontokieleksi, jossa tulee usein vastaan tilanteita, joissa parametrien tyypeillä on väliä.

4.3 Käytännön esimerkki multimetodien käytöstä

Listauksessa 4.1 esitellään hieman monimetodien käyttöä käytännössä. Kielenä esittelyssä on käytetty C++, jossa ei ole monimetodeita. Käytännön elämässä Shape-luokkaan pitäisi soveltaa Visitor-suunnittelumallia ja lapsiluokat Circle ja Shape kertoisivat tyyppinsä Visitor-rajapinnalle. Kielissä, joissa ei ole käytössä kuin staattinen ylikuormitus, tämä on valitettavan yleinen, pakollinen käytäntö.

Oleellinen seikka esimerkissä on, että myShapes tallentaa muodot Shape-tyyppisenä. Ainoastaan itse grafiikan piirrossa, jossa konkreettinen tyyppi on tiedettävä, tarvitsee ottaa kantaa tyyppiin. Tämä auttaa luomaan abstraktiota ohjelmaan.

Listaus 4.1: Monimetodien käyttö

```
1
2 class Shape {
3 public:
4     /* common stuff here */
5     double location_x = 2.0;
6     double location_y = 5.0;
7 };
8
9 class Circle : public Shape {
```

```
10 public:
11    double radius = 2.5;
12 };
13
14 class Regtangle : public Shape {
15 public:
16    double side_x = 1.0;
17    double side_y = 2.0;
18 };
19
20 class FancyGraphics {
21 public:
22    /* general case */
23    void draw(Shape *c);
24    /* specialization for Circle */
25    void draw(Circle *c);
26    /* specialization for Regtangle */
27    void draw(Regtangle *c);
28 };
29
30 /* myShapes has circles and regtangles */
31 vector<Shape *>::iterator it;
32 FancyGraphics myFancy;
33 for(it = myShapes.begin();
34      it != myShapes.end();
35      ++it)
36 {
37    Shape *shape = *it;
38    myFancy.draw(shape);
39 }
```

4.4 Monimetodit ovat ongelmallisia tyypitykselle

Koska parametrit tulkitaan ajon aikana, ei tyypitys voi olla vakaa. Lisäksi on mietittävä, miten ikävät tyyppivirheet ajon aikana voisi vähentää minimiin.

Eräs vaihtoehto voisi olla, että alaluokkien määrittelemiä tyyppejä rajoitetaan reippaalla kädellä. Jos listauksen 4.1 FancyGraphics-luokalle tai sen lapsiluokalle luodaan uusi ylikuormitus draw-metodille, on parametrin oltava tyyppiä Shape. Tällöin kääntäjä voi estää pahimpia tyyppivirheitä jo käännösaikana. Lisäksi parametrien lukumäärä voi rajoittaa ja hahmottaa tyypitystä parempaan suuntaan. Toi-

saalta, viitteessä [6] esitetään melko kattava tyypitysmekanismi, jolla monimetodien hitaita etsimisiä voi jättää pois melko paljon.

4.5 Yhteenvetoa luvusta 4

Oleellisimmat asiat luvusta 4 ovat:

- Wolf-kieli käyttää dynaamista ylikuormitusta,
- monimetodit helpottavat kehittäjän työtä,
- monimetodit ovat vaikeita toteuttaa,
- monimetodien lopullinen toteutus ei vielä ole valmis.

5 Perinnän erityisominaisuudet Wolf-kielessä

Tämä luku kertaa perusasiat periytymisestä ja Wolf-kielen erityispiirteistä perinnälle.

5.1 Mitä perintä on?

Lyhyesti sanoen perintä on luokkien järjestämistä hierarkiaksi. Perinnän avulla voidaan luokkia järjestää järkeviin hierarkioihin. Esimerkiksi *Car*-luokka voi koota autojen ominaisuudet yhteen.

Jos luokka B perii luokan A, sanotaan luokkaa A luokan B alaluokaksi. Vastaavasti luokka B on luokan A yläluokka. Yläluokasta voidaan käyttää nimitystä kantaluokka, jos yläluokalla on useita alaluokkia.

Jos B on luokan A aliluokka, ilmaistaan tämä jatkossa merkinnällä:

$$B <: A$$

5.2 Moniperintä

Jotkin ohjelmointikielet sallivat luokalle useita yläluokkia. Tätä ominaisuutta kutsutaan moniperinnäksi. Moniperintä on mahdollista esimerkiksi Pythonissa, C++:ssa ja Eiffelissä. Jokaisessa kielessä on joitain ongelmia moniperinnän kanssa.

Yleisen käsityksen mukaan moniperintä aiheuttaa enemmän ongelmia kuin korjaa niitä [6]. Moniperintä on hieno ominaisuus, mutta sen toteuttaminen järkevästi ei ole triviaalia kielen toteuttajalle. Kirjoittajan oman näkemyksen mukaan Wolf-kieli ei tarvitse moniperintää, vaan metaohjelmointi korvaa yksiperinnän puutteet. Jokainen luokka voi siis periä vain yhden yläluokan kuten Java-ohjelmointikielessä.

Käytännössä yksiperintä selkeyttää virtuaalikoneen suunnittelua, helpottaa kääntäjän tekemistä ja mahdollistaa näppäriä optimointeja, joita käsitellään myöhemmin tutkielmassa.

Wolf-kielen rajapinnalla on seuraavat ominaisuudet:

- Rajapinta määrittää metodeja eli toimintaa.

- Rajapinta ei voi määritellä jäsenmuuttujia; Rajapinnalla ei siis ole erityistä tilaa.
- Rajapinta voi laajentaa toisia rajapintoja muttei periä luokkia.
- Rajapinnasta ei voi suoraan luoda olioita, vaan luokka toteuttaa rajapinnan, ja vasta luokasta voi luoda olion.
- Luokka voi toteuttaa useita rajapintoja.
- Luokka voi toteuttaa rajapinnan vain kerran.

Luokka määrittää sekä toteutuksen että toiminnallisuuden. Rajapinta määrittää vain toiminnallisuuden muttei toteutusta, joten rajapinnat ovat hieman joustavampia perinnän kannalta kuin luokat.

5.3 Yhteenvetoa luvusta 5

Perintä on käsitteenä laaja ja monimutkainen, mutta tämä tutkielma ei käsittele aihetta kuin hyvin pintapuolisesti. Oleelliset asiat tästä luvusta ovat

- Perintä on ihannetapauksessa hierarkian luomista,
- Moniperintää ei ole Wolf-ohjelmointikielessä,
- Rajapinnat ovat joustavampia hierarkian luomisessa kuin luokat ja
- Yksiperintä helpottaa kielen toteutusta.

6 Perinnän ongelmat

Perintä on oikein käytettynä erittäin tehokas työkalu, jolla voi luoda uutta toimintaa hyvin pienellä määrällä uutta koodia. Vaikka perintä on oleellinen ominaisuus nykyisissä oliokielissä, sillä on useita huonoja sivuvaikutuksia, joita käsitellään tässä luvussa. Luku ei ole kattava kuvaus ongelmista, joita perinnän liiallinen käyttö aiheuttaa vaan pelkästään perehdytys aiheeseen.

6.1 Vahva riippuvuus

Perittäessä alaluokan on tiedettävä, mistä luokasta se peritään[1]. Jo tämä vaatimus luo riippuvuuden ylä- ja alaluokan välille.

Holubin mukaan hyvänä nyrkkisääntönä ohjelmistokoodista yli 70 prosenttia käyttää rajapintoja viestien välitykseen oliolta toiselle [25], jotta riippuvuudet eri ohjelmistokomponenttien välillä ovat mahdollisimman vähäisiä. Rajapintoja käyttämällä abstraktion taso ohjelmassa on yleensä korkeampi ja tämä mahdollistaa helpomman ylläpidon – ja mahdollisesti komponentin käyttämisen uudelleen toisessa ohjelmassa. Kirjoittajan omien kokemuksien perusteella perintään perustuva uudelleenkäyttö ei toimi kuin triviaaleissa tapauksissa, jos tarkoituksena on tuottaa ylläpidettäviä ohjelmistoja.

6.2 Perintä toiminnallisuuden vuoksi

Edellä olevat esimerkit Sokrateksesta ja kuolevaisuudesta esitti perintää hierarkian kannalta: jos Sokrates on ihminen, hän on myös kuolevainen. Perintä hierarkian kannalta on kirjallisuudessa yleisesti hyvä, oikeaoppinen tapa periä. Jokainen perinnän taso vähentää abstraktiota mutta lisää ominaisuuksia ja jokaisella alaluokalla on myös yläluokan ominaisuudet. Suunnittelijan, toteuttajan ja ylläpitäjän on helppo käsittää perintäketju, joka perustuu hierarkiaan. Tämä on suurin puolustus hierarkiaan perustuville perintäketjuille.

[1] Osassa olio-kielistä on mahdollista periä dynaamisesti ja jopa vaihtaa kantaluokkaa ajon aikana. Tämä on kuitenkin niin harvinainen ominaisuus, ettei sitä käsitellä tässä tutkielmassa.

Toinen syy, miksi kehittäjät haluavat periä jonkin luokan, on käyttää *toteutusta* uudelleen alaluokissa. Uuden toiminnallisuuden toteutus on helppoa alaluokassa, joka käyttää perintää toteutuksen takia. Parilla rivillä voi yläluokan toiminnallisuutta käyttää uudelleen alaluokassa. Huonona puolena on riippuvuuden lisääntyminen ylä- ja alaluokan välille tarpeettoman paljon.

Perittäessä toteutusta alaluokka tekee enemmän tai vähemmän lopullisia oletuksia yläluokan toteutuksesta. Kun alaluokkia on tarpeeksi paljon, ei yläluokkaa enää voida optimoida tai muuttaa juuri mitenkään ilman riskiä, että jokin lapsiluokista hajoaa. Huonoimmillaan yläluokan virheitä ei välttämättä voi korjata, koska alaluokan toiminnallisuus hajoaisi, jos kantaluokan toiminta olisi täsmällistä ja tarkkaa; Useimmiten ohjelmiston toiminta on tarpeettoman hidasta, koska yläluokkia ei voi optimoida.

6.3 Muita ongelmia perinnässä

Särkyvän kantaluokan ongelma on tunnettu pitkään ja hyvä yhteenveto tuloksista vuodelta 1998 löytyy artikkelista [34]. Samaisessa lähteessä esitetään formaalia tekniikkaa, jolla rikkoontuvan kantaluokan voisi tunnistaa. Valitettavasti Wolf-kielen suunnittelussa ei ole resursseja toteuttaa automaattista tarkastusta kantaluokan rikkoontumiselle. Kieli vain varoittaa, jos perintäketju kasvaa tarpeettoman pitkäksi ja jos perintää käytetään ohjelmassa ylenpalttisen paljon.

Vaikka särkyvän kantaluokan ongelma aiheuttaa teknisiä ongelmia, siitä on haittaa myös kommunikaatiolle:

- Luokan alkuperäinen tarkoitus hämärtyy perintäketjun pidentyessä,
- Luokkiin ei uskalleta tehdä tarpeellisia muutoksia, koska ketju *voi* olla herkkä särkymään ja
- Koodi menettää osan suoraviivaisuudestaan.

6.3.1 Abstrakti luokka

Abstraktilla luokalla tarkoitetaan luokkaa, joka määrittää toteutusta, mutta jättää osan toiminnallisuudesta määrittelemättä. Useimmiten abstrakti luokka toteuttaa luokan miltei valmiiksi ja lapsiluokissa tarvitsee toteuttaa enää muutama rivi ohjelmakoodia. Käytännössä abstraktit luokat pitäisi suunnitella paljon paremmin kuin

nykyisellään ja perintä tapahtuu jo oletuksena toiminnallisuuden kierrätyksen takia.

Wolf-ohjelmointikielessä ei ole abstrakteja luokkia, mikä aiheuttanee närää osalle ohjelmoijista, jotka kokeilevat Wolf-ohjelmointikieltä. Abstraktit luokat ovat toki näppäriä jossain yhteyksissä, mutta useimmiten ne kasvavat liian suuriksi ja niistä peritään useita alaluokkia – mikä on niiden alkuperäinen tarkoituskin.

Kirjoittajan oman kokemuksen mukaan abstraktit luokat ovat yleensä vaikeita optimoida, ylläpitää ja periä oikein. Lisäksi kielissä, joissa on käytössä vain yksiperintä, abstraktit luokat sitovat tyyppihierarkian turhan tiukasti yhteen muottiin vain toiminnallisuuden eikä semantiikan takia. Ohjelmistojen elinkaaren aikana – kuten elämässä yleensäkin – hyvin harva asia on lopullista ja varmaa. Abstraktit luokat estävät väärin käytettyinä ohjelmistokoodin jatkuvan parantamisen ja pakottavat hyväksymään suunnitteluvirheitä, joita tehdään aina.

6.4 Yhteenvetoa luvusta 6

Oleelliset asiat tästä luvusta ovat:

- Perintä lisää vahvan riippuvuuden ala- ja yläluokan välille,
- Mitä pidempi perintäketju on ja mitä enemmmän peritään toiminnallisuuden takia, sitä varmemmin kantaluokkaa ei voi jatkossa muuttaa ja
- Wolf-kielessä ei ole abstraktin luokan käsitettä, koska sitä ei tarvita.

7 Virhetilanteiden hallinta

Tämä luku hahmottelee eri tapoja, miten virhetilanteita voi käsitellä poikkeuksien avulla. Aihe on yllättävän laaja, eikä tässä tutkielmassa paneuduta aiheeseen kovin syvällisesti. Tarkoituksena on vain antaa muutamia ideoita, miten poikkeukset voidaan käsitellä kielen toteutuksessa.

7.1 C-kielen virheenkäsittely

C-kieli luottaa miltei sokeasti kehittäjään, jonka vastuulla oli tarkistaa suoritetaanko operaatio oikein vai ei. Tämä vaatii kehittäjiltä sekä kommunikointia että kurinalaisuutta. Molemmat kyvyt ovat harvinaisia (kirjoittajan omien kokemuksien perusteella). Vähemmän yllättäen C-kielellä tehdyissä ohjelmistoissa on usein ollut pahoja ongelmia käsitellä virheitä. Tämä ei johtunut kehittäjien huonoudesta vaan ongelmien monimutkaisuudesta.

Yleisin ja edelleen käytetty tapa käsitella virheitä on palauttaa erillinen virhearvo aliohjelman suorituksesta. Esimerkiksi Unixin open-aliohjelma, joka avaa tiedoston, palauttaa tiedostokahvan numeron, jos suoritus onnistuu ja arvon -1 virhetilanteessa.

Virhearvo -1 ei silti vielä kerro paljoa, miksi tiedoston avaaminen ei onnistunut. Tätä varten open-aliohjelma muuttaa globaalia virhearvoa, jonka avulla päätellään, miksi tiedostoa ei voitu avata.

Tämäntyyppinen käsittely aiheuttaa kuitenkin melko pahoja ongelmia.

- Kehittäjä ei välttämättä tarkista virhearvoa, mikä johtaa ohjelman ennakoimattomaan kayttaytymiseen.
- Vaikka open-aliohjelmaa käyttävä aliohjelma tarkistaa virhearvon, se ei välttämättä välitä sitä eteenpäin. Virhettä ei näy käyttöliittymässä tai korkeammilla tasoilla lainkaan.
- Jos ohjelmaa ajetaan hajautetuissa tai säikeistetyssä ympäristössä, globaali virhearvo aiheuttaa ongelmia, koska alkuperäinen virhe voidaan ylikirjoittaa milloin tahansa uudemmalla virheellä.

Tätä ongelmaa helpottamaan kehitettiin poikkeukset, jotka ovat nykyään standardi osa olio-kieliä.

7.2 Mitä poikkeukset ovat?

Poikkeus on ennakoimaton virhe ohjelmaa suorittaessa. Esimerkiksi kahden kokonaisluvun jakaminen johtaa virheeseen, jos jakajana on nolla. Jos merkkijonoa käsiteltäessä yritetään muuttaa muistialuetta merkkijonon jälkeen, ohjelmaan voi tulla tietoturva-aukko buffer-overflow-virheen seurauksena. Java-ohjelmointikieli heittää poikkeuksen näissä tilanteissa eikä anna mahdollisuutta muistin korruptointiin.

Jos ohjelma joutuu tilanteeseen, josta normaalisti jatkamalla ohjelma voisi tehdä jotain vaarallista tai odottamatonta, kehittäjä voi estää suorituksen heittämällä poikkeuksen. Poikkeus on siis keskeytys normaaliin ohjelman suoritukseen ja se käsitellään erillään muusta ohjelmasta. Oleellista on, että ohjelma ei joudu vaaralliseen tilaan – kuten buffer-overflow voisi aiheuttaa.

Hyvänä puolena poikkeuksessa on ohjelman suoritusjärjestyksen muuttuminen: huolimaton kehittäjä ei voi enää ohittaa virhetilannetta olankohautuksella kuten C-kielessä. Huonona puolena on – ironista kyllä – suoritusjärjestyksen muuttuminen, koska huolimaton kehittäjä voi sokeasti luottaa siihen, ettei suoritusjärjestys muutu ja jättää resursseja vapauttamatta.

Poikkeuksen heittäminen on signaali ohjelmalle, että haluttua toimintaa ei voitu suorittaa loppuun. Poikkeus sisältää ihannetapauksissa tarpeelliset tiedot, jotta voidaan selvittää, mikä tarkalleen ottaen aiheutti virheen. Tämä mahdollistaa isojenkin kirjastojen tekemisen kohtalaisen miellyttäväksi käyttää, koska kirjaston kehittäjä voi ilmoittaa selkeästi mikä meni vikaan ohjelmaa suorittaessa. Käytännössä poikkeukset kyllä helpottavat ongelman syyn selvittämistä, mutta virheiden selvittäminen voi olla hyvin vaikeaa myös poikkeuksia käyttäessä.

Poikkeus on ohjelmoijan kannalta Wolf-kielessä normaali olio, joka voidaan luoda, käsitellä ja muokata kuten muitakin oliota. Ainut rajoitus on, että se periytyy erityisestä Exception-luokasta. Perinnän avulla poikkeukset muodostavat jonkin järkevän hierarkian.

7.3 Poikkeuskäsittelijä

Poikkeuskäsittelijäksi kutsutaan erityistä koodilohkoa, joka yrittää toipua virheestä.

Kehittäjän näkökulmasta poikkeuksen käsittely tapahtuu yleensä seuraavasti:

1. Kehittäjä merkitsee ohjelmalohkon, joka *voi* heittää poikkeuksen.
2. Merkitty ohjelmalohko suoritetaan.
3. Jos poikkeus heitettiin, suoritetaan sopiva poikkeuskäsittelijä. Jos sopivaa käsittelijää ei löydy, suoritetaan oletuskäsittelijä. Esimerkiksi C++-kielessä ohjelma keskeyttää toimintansa, jos virhettä ei käsitellä lainkaan.

Koska virhekäsittelijä suoritetaan vain, jos poikkeus tapahtuu, virheenkäsittelyyn liittyvä ohjelmakoodi pysyy erillään muusta, normaalista koodista, mikä helpottaa ylläpitoa ja siisteyttää lähdekoodia [40].

Dramaattinen sivuvaikutus on, että poikkeuksen käsittely ja heittäminen leikkaa koko ohjelman halki. Virheenkäsittely muodostaa oman näkökulmansa, jota on käsitelty tarkemmin luvussa 12.

Nykyisin ohjelmistot ovat jo todella laajoja ja rakentuvat entistä enemmän vanhoihin toteutuksiin. Poikkeus, joka pitää käsitellä, voi olla aiheutunut mitä erilaisimmista syistä. Muutama viattoman tuntuinen rivi voi kutsua useita muita osia itse ohjelmaa tai sen käyttämiä kirjastoja ja komponentteja. Poikkeus voi olla lähtöisin jopa eri koneelta, jos ohjelma on hajautettu eri koneiden välille. Ilman mitään tietoa poikkeuksesta poikkeuskäsittelijän pitäisi käydä jokainen mahdollinen virhetilanne läpi. Käytännössä tämä olisi mahdotonta toteuttaa järkevästi.

7.4 Poikkeusten ryhmittely

Kehittäjän ja ylläpitäjän onneksi käsittelijän voi määrätä hyväksymään vain tietyn tyyppisiä virheitä. Kappaleesta 2.3 muistamme, että jokainen olio on jonkin luokan instanssi, joten myös poikkeus on jonkin luokan instanssi. Koska poikkeukset ovat olioita, niitä voidaan myös periä ja muodostaa hierarkiaa. Tätä hierarkiaa voidaan hyödyntää poikkeuskäsittelijöitä suunniteltaessa.

Poikkeuskäsittelijää tehdessä voidaan määrätä, että se käsittelee vain tietyn tyyppisiä poikkeuksia. Esimerkiksi nollalla jakaminen on selkeä virhetilanne. Tällöin Wolf-kieli heittää poikkeuksen DivideByZero. Tämä poikkeus on peritty Wolf-kielessä MathException-luokasta – joka on myös poikkeus. Jos poikkeuskäsittelijä hyväksyy MathException-tyyppisiä poikkeuksia ja jos ohjelma heittää DivideByZero,

suoritetaan poikkeuskäsittelijä, koska DivideByZero on MathException-luokan alaluokka. Jos jokin muu poikkeus olisi heitetty, käsittelijää ei olisi suoritettu vaan sopivan käsittelijän etsintää olisi jatkettu.

Poikkeuksen ryhmittelyksi sanotaan hierarkiaa, jonka poikkeusluokat muodostavat. Poikkeuskäsittelijöissä ei tarvitse käydä erikseen läpi jokaista virhettä vaan poikkeuksia voi käsitellä abstraktimmalla tasolla.

7.5 Mitä poikkeuskäsittely vaatii

Kun poikkeus heitetään, ohjelman pitää tehdä seuraavat asiat:

1. Suorittaa mahdolliset siistijät paikallisessa ohjelmalohkossa, jotka on merkattu esimerkiksi Java-kielessä finally-avainsanalla. Myös C++:n paikalliset oliot pitää hävittää ennen poistumista ohjelmalohkosta, jossa poikkeus heitetään.

2. Etsiä sopiva käsittelijä poikkeukselle. Jos käsittelijä ei ole määritelty samassa lohkossa, jossa poikkeus heitetään, pitää pinoa kelata alaspäin, kunnes saavutaan lohkoon, jossa käsittelijä on määritelty.

3. Suorittaa poikkeuskäsittelijä ja

4. jatkaa toimintaa virheestä toipumisen jälkeen. On mahdollista, että poikkeuskäsittelijä heittää uuden poikkeuksen, jolloin algoritmia on sovellettava uudelleen.

Pinoa pitää siis kelata alaspäin, kunnes löydetään sopiva käsittelijä. Tätä varten pitää palauttaa koneen tila samanlaiseksi kuin ennen try-lohkoon menoa. Ongelmallisin osa on pinon kelaaminen alaspäin [13].

7.6 C++-ohjelmointikieli ja poikkeukset

Sekä Windowsin VC++ että GCC rekisteröivät try-lohkossa pinon tilan. Ohjelma jatkaa suoritustaan ja pino kasvaa. Kun poikkeus tapahtuu, pinoa kelataan alaspäin kohtaan, jossa try-lohko oli. Sen jälkeen kutsutaan poikkeuksen käsittelijää. Tämä on yleinen tapa käsitellä poikkeusta kääntäjän puolella.

Bjarne Stroustrup tiesi C++-ohjelmointikieltä suunnitellessa, että kieleen voisi tehdä poikkeuskäsittelyn, joka ei vaikuttaisi suoritusnopeuteen ellei poikkeusta varsinaisesti heitetä [47, Exception Handling].

Valitettavasti tämä olisi ollut hyvin vaikeaa toteuttaa niin, että uusi mekanismi olisi ollut yhteensopiva C-kielen, debuggereiden ja vanhojen kirjastojen kanssa. C++-kielessä jo pelkkä mekaniikka, jolla poikkeuksia käsitellään hidastaa ohjelman suoritusta, vaikka poikkeusta ei varsinaisesti heitettäisikään. C++-kääntäjät toimivat siten, että ennen poikkeuksia heittävään lohkoon siirtymistä merkataan lohko. Jos poikkeus tapahtuu käytetään merkkiä ja etsitään käsittelijä nopeasti.

Käytännössä osa kääntäjistä käyttää vieläkin kohtalaisen hitaita setjmp, getjmp-komentoja C-kielestä toteuttamaan poikkeusten käsittelyn. Setjmp-komento pitää suorittaa, vaikka poikkeusta ei heitettäisikään. Modernit kääntäjät eivät tuhlaa aikaa hitaisiin hyppykomentoihin vaan soveltavat enemmän tai vähemmän luovasti viitteessä [13] esitettyjä tekniikoita.

Yleensä hidastuminen poikkeusmekanismin takia ei ole ongelma, koska nykyiset kääntäjät osaavat optimoida lopullisen ohjelman tehokkaasti. Edelleen, poikkeusmekanismin tuoma hidastuminen on miltei aina hintansa väärti, koska virheiden käsittely on helpompaa ohjelmistokoodissa.

Valitettavasti ohjelmistot, joissa on hyvin korkeat suorituskykyvaatimukset, jättävät yleensä lopullisesta julkaisusta poikkeukset pois, jotta suorituskyky saadaan puristettua äärimmilleen. Pelit ovat suorituskyvyltään erittäin vaativia ohjelmia ja useimmissa nykyisissä peleissä viimeisimpiä optimointeja ennen pelin julkaisua on poikkeuksien poistaminen käännöksestä [51].

Kirjoittajan oman mielipiteen mukaan poikkeusten jättäminen pois suorituskyvyn vuoksi on nykyisillä toteutuksilla C++-kääntäjistä olla jo melko turhaa vaivannäköä, koska kääntäjät osaavat optimoida turhaa koodia pois hyvin tehokkaasti. Ainakin uusimmissa GCC-perheen C++-kääntäjissä poikkeukset eivät välttämättä aiheuta lainkaan nopeuseroa, jos poikkeusta ei heitetä. Matti Rintalan tekemissä testeissä C++-ohjelma, joka ei heittänyt poikkeuksia, oli yhtä nopea sekä poikkeusmekanismin kanssa että ilman poikkeusmekanismia [37, Performance Measurement].

7.7 Poikkeuksen käsittely säikeistetyissä ympäristöissä

Poikkeukset säikeistetyissä ympäristöissä ovat melko vaativa aihe. Wolf-kieli käyttää ensimmäisissä versioissa Matti Rintalan ideoita, joilla C++-kielessä voidaan käsitellä poikkeuksia monisäikeisessä ympäristössä [38]. On toki olemassa keinoja, joilla kieleen voisi lisätä suoraan tuen useammasta säikeestä tulevalle poikkeukselle [40]. Wolf-kieli kuitenkin pohjautuu C++-kieleen ja tästä syystä ainakin ensimmäiset to-

teutukset Wolf-kielestä käyttävät samoja rajoituksia ja mahdollisuuksia mitä C++-kielikin.

C++-ohjelmointikielessä ei suoraan ole tukea useamman yhtäaikaisen poikkeuksen käsittelylle. On vielä epäselvää pitäisikö Wolf-kielessä olla tuki useamman poikkeuksen yhtäaikaiselle käsittelylle.

Vaikka on olemassa erilaisia tapoja käsitellä poikkeusta uusissa kielissä, Wolf-kieli käyttää samoja rajoituksia ja mahdollisuuksia kuin C++-kieli, jotta yhteenliittäminen sujuisi ongelmatta. Apuna käytetään Matti Rintalan KC++-kirjastoa ja sen tarjoamia ideoita [39]. KC++:n ideoita voi lisäksi käyttää Attribute-kirjastossa, joka toteutetaan gradun aikana.

Toinen syy käyttää perinteisiä ratkaisuja ainakin ensimmäisissä versioissa on optimoinnin helpottaminen. Action Based exception handling, joka ottaa voimakkaasti kantaa poikkeusten rakenteeseen ja "hiekkalaatikkomaiseen"ajatteluun, olisi ehdottomasti selkeämpi [41]. Tavoitteena on kirjoittaa ensimmäinen toteutus Wolf-kielelle 2008 vuoden kesään mennessä eikä toteutukseen ehdi ottaa kaikki mahdollisia uusia innovaatioita. Lisäksi Wolf-kielessä on jo ennestään paljon uudistuksia verrattuna vanhoihin kieliin. Kaikkien uusien ideoiden lisääminen voi johtaa ylilyönteihin ja kääntäjän entistä vaikeampaan toteuttamiseen.

7.8 Zero-Overhead exception

On mahdollista luoda poikkeusmenettely, joka ei hidasta ohjelman suoritusta lainkaan, jos se ei heitä poikkeuksia [13]. Viitteessä esitetyn algoritmin perusidea oli määrittää lohkoja varten oma käsittelijä. Kääntäjä generoi taulukon, jossa oli lohkon aloitusosoite, -lopetusosoite ja käsittelijän osoite funktiopointterina. Kun poikkeus heitettiin, etsittiin oikea käsittelijä suorituksen mukaan. Tämä menetelmä voi kuitenkin olla tarpeettoman raskas Wolf-kielessä, joka toimii virtuaalikoneen päällä. Virtuaalikoneella ei välttämättä ole niin helppoa pääsyä laitteistorajapintaan. Lisäksi Wolf-kieli on säikeistetty ympäristö, joten algoritmin toimivuutta pitää miettiä vielä rinnakkaisuudenkin näkökulmasta.

7.8.1 Poikkeuksen käsittely metakielissä

Kielissä, joissa on tuettuna niin sanottu reflektio-ominaisuus, voidaan käyttää viitteessä [21] käytettyjä tekniikoita. Algoritmin perusidea oli seuraava:

1. Määritellään käsittelijä suoritettavan lohkon sisään erityisellä nimellä – kuten handleException.
2. Kun poikkeus tapahtuu, etsitään kyseisestä metodista handleException-metodia.
3. Jos käsittelijä löytyy, suoritetaan se ja jatketaan suoritusta poikkeuksen heittäneen kohdan jälkeen.
4. Jos käsittelijää ei löydy, poistutaan lohkosta, mennään edelliseen lohkoon ja siirrytään algoritmin edelliseen kohtaan.

Tämä menetelmä ei kuitenkaan toimi Wolf-kielessä niin hyvin kuin pitäisi, koska sen muuttaminen jälkeenpäin olisi vaikeampaa kuin KC++-tyylisen ratkaisun käyttäminen.

7.9 Yhteenvetoa luvusta 7

- Oleellisimmat erot poikkeuksissa virhearvoihin nähden ovat [47, Exception Handling]:
  - Kehittäjä ei voi jättää virhettä huomioimatta, koska poikkeus muuttaa ohjelman suoritusjärjestystä,
  - Koodi on yleensä siistimpää, koska virheenkäsittely ei enää ole normaalin koodin seassa ja
  - Laajojen kirjastojen tekeminen on helpompaa, koska poikkeuksia on helpompi ryhmitellä kokonaisuuksiin ja käsitellä niitä abstraktimmalla tasolla.
- Poikkeukset ovat yleisin tapa käsitellä yllättäviä tilanteita,
- Pinon hallinta voi olla melko monimutkaista poikkeuksenkäsittelyssä,
- Wolf-kielen tapa käyttää poikkeuksia ei vielä ole lyöty lukkoon ja
- Poikkeukset monisäikeisissä ympäristöissä ovat melko vaikea aihe.

8 Miksi OO on huono hopealuoti?

> Turhuuksien turhuus, sanoi Saarnaaja, turhuuksien turhuus, kaikki on turhuutta! (Saar. 1:2)

Tämä kappale sisältää hieman kritiikkiä olio-ohjelmoinnin nykytilanteesta. Mukana on hieman yleisiä ongelmia, mihin olio-paradigma ei ole pystynyt antamaan tyydyttävää vastausta. Suurin osa luvusta on kuitenkin käytännön ongelmia, jotka eivät niinkään johdu olio-paradigmasta itsestään vaan olio-kielten, -kehittäjien ja kommunikoinnin puutteesta.

Jotta luku ei olisi pelkästään katkerahkoa tarinaa, mukana on myös arviointia, miten Wolf-kieli pyrkii ratkaisemaan osan ongelmista.

8.1 Model View Controller

Model-View-Controller(MVC) on arkkitehtuurinen suunnittelumalli, jossa käyttöliittymään näkyvät oliot jaetaan kolmeen eri osa-alueeseen. Malli suorittaa loogisen laskennon. Näkymä eli View näyttää tiedon käyttäjälle. Kontrolliluokka eli Controller käsittelee käyttäjän syötteen ja ohjaa näkymää ja mallia. Teoriassa tämä toimii hyvin.

Ongelmana käytännön elämässä on, että muutoksia käyttöliittymässä näkyviin olioihin tapahtuu aina. Hyvin harva ohjelmisto pystyy sekä palvelemaan asiakkaan tarpeita että säilyttämään ohjelmiston luomisprojektin alussa luodun suunnitelman. Kun ohjelmisto kypsyy, voi olettaa, että sille tulee *aina* uusia vaatimuksia. Parannukset ohjelmistoon vaikuttavat aina jonkin verran ohjelmakoodiin. Useimmiten olioon pitää lisätä uusia kenttiä, näkymiä yhdistää ja muuta sellaista. Kun muutoksia käyttöliittymässä näytettäviin olioihin lisätään, pitää huonoimmillaan muuttaa kaikkia kolmea osaa kokonaisuudesta: mallia, johon muutos tulee, kontrolleria, joka näyttää tiedot, ja käyttöliittymää, jotta muutos näkyy käyttäjällekin asti.

Holub kritisoi voimakkaasti MVC-paradigmaa ja vaati oliomaisemman ratkaisun ottamista käyttöön [22, sivu 2]. Reenskaug kehitti MVC-mallin jo vuonna 1979 [30]. Holubin mukaan MVC on paradigmana vanhentunut, koska laitteisto on mennyt eteenpäin vuodesta 1979. Esimerkiksi käyttäjän syötettä ja tiedon näyttämistä

käyttäjälle voi vain harvoin erottaa toisistaan, kuten MVC Holubin mukaan olettaa.

Holubin kritiikki on hieman kohdistamatonta, koska MVC-mallin ideana oli erottaa malli, näkymä ja kontrolli kolmeen eri tasoon – ei luokkiin. Nykyiset toteutukset vain toteuttavat MVC-suunnittelumallin käyttämällä get/set-metodeja tai julkisia jäsenmuuttujia, mikä johtaa ongelmiin.

Huonoissa MVC-toteutuksissa malli-luokka käyttää naiivisti saantimetodia jokaiselle näytettävälle kentälle. View pyytää mallilta jokaisen kentän arvon erikseen ja näyttää ne sitten näkymässä. Kirjoittajan huonoin kokemus MVC:stä oli java-servlet alusta, joka käytti MVC:tä sisäisesti ja jsp-teknologiaa datan näyttämiseen käyttäjälle. Alla muutamia yleisiä virheitä, joita kirjoittaja havaitsi:

- Olioon, joka tarjosi tietoa käyttäjälle, tuli uusia käyttäjälle näytettäviä kenttiä säännöllisesti. Jokainen uusi kenttä vaikutti kontrolli-olioon. Jokainen uusi kenttä vaikutti näkymään. Jokainen uusi kenttä vaikutti JSP-sivuun.
- Kontrolliin liittyvää koodia alkoi valua näkymään ja toisinpäin.
- Mallista piti tarjota eri näkymiä. Tämä lisäsi sotkua entisestään, koska mallilla ei ollut varsinaisesti minkäänlaista rajapintaa eri näkymien tukemiseen.
- Yksikkötestaaminen oli mahdotonta.

MVC-malli on mukana tässä luvussa, koska siihen kiteytyy useimmat olio-ohjelmoinnin ongelmat:

- Se on hypetetty tekniikka, jota käytetään ylenmäärin ja väärissä paikoissa,
- Se on väärin ymmärretty, joten toteutus on väärin,
- Ylläpitoa ei yleensä mietitä mallia toteuttaessa, mikä johtaa huonoihin ratkaisuihin ylläpidossa ja
- Idea on hyvä, mutta käytännön sovellus johtaa huonoon lopputulokseen.

8.2 Olio-ohjelmat ovat hitaampia kuin perinteinen C-koodi

Tämän myytin alkuperä on jossain C++-kielen ensimmäisten kääntäjien paikkeilla. Nykyiset oliokielten toteutukset suoriutuvat kääntämisestä melko hyvin, jos kriteeteinä pidetään ohjelman suorituskykyä ja muistinkulutusta.

Myytti elää kuitenkin sitkeästi ja useimmat kehittäjät tekevät edelleen huonoa proseduraalista koodia oliokääntäjillä. Esimerkiksi suurin osa Java-koodista ei ole erityisen oliomaista [24].

Useimmiten huono oliokoodi johtaa suorituskyvyltään heikompiin ratkaisuihin koko ohjelmiston tasolla, koska muutoksia ja optimointeja ei voi tehdä ilman ohjelman rikkoontumista. Kirjoittaja on ylläpitänyt C++-kielellä toteutettua ohjelmaa, joka ei käyttänyt normaaleja string-oliota vaan C-kielen merkkijonoja, koska oliot ovat hitaampia. Ohjelma monimutkaistui kuitenkin kieltämättä nopeiden merkkijonojen takia niin paljon, että arkkitehtuuriset parannukset jäivät kehittäjältä tekemättä. Hyvin kapseloiduilla olioilla olisi alkuperäisen nopean merkkijonojen vertailun voinut korvata vielä nopeammalla kokonaislukujen vertailulla. Nopeat merkkijonot, jotka estivät arkkitehtuurin parantamisen, aiheuttivat myös ongelmia muistinkäsittelylle ja arvojen säilytykselle. Lopputuloksena optimoinneista oli ohjelma, joka oli painajainen ylläpitää ja joka toimi hitaammin kuin kunnollisesti suunniteltu olio-ohjelma.

Nyrkkisääntönä voidaan pitää, että mitä piilotetumpi toteutus on, sitä helpompi oliokielen kääntäjän on luoda tehokasta ohjelmakoodia. Nykyisistä C++-kääntäjistä esimerkiksi g++-kääntäjän versiot 4.0:sta eteenpäin ovat tukeneet metodien piilotusta kirjaston ulkopuolelta. Tämän ansiosta kääntäjä voi tehdä sitä enemmän optimointeja mitä paremmin toteutus olioista on piilotettu ja siten parantaa ohjelman suorituskykyä. Kirjoittajan omien kokemuksien mukaan ylläpidettävä, hyvä oliokoodi on suorituskyvyltään parempi kuin huonosti suunniteltu "optimoitu"ohjelmisto, mitä näkee valitettavan usein.

8.3 Wolf-kielen parannusehtotukset

Jotta riippuvuudet eri olioiden välillä ovat vähäisiä, jäsenmuuttujat olioista näkyvät vain oliolle itselleen. On olemassa muutamia melko yksinkertaisia kaavoja, joilla voidaan tarkistaa karkealla tasolla, onko olio-ohjelmisto laadukas vai ei. Koko ohjelmiston kompleksisuus lasketaan ja verrataan sitä tiettyihin raja-arvoihin [4] ja [42]. Jos esimerkiksi perintää on käytetty ylenpalttisesti, kääntäjä varoittaa. Ongelmana tähän parannusehdotuksessa on, että laatu on aina subjektiivinen käsite.

Huolimatta laadun subjektiivisuudesta, Wolf-kielisen ohjelman käännöksessä on lopuksi vielä kitinä-moduuli, joka varoittaa kehittäjää huonosti sisäistettävistä ratkaisuista. Jos yhdessä luokassa on suurin osa koko ohjelmiston toiminnallisuudesta, kääntäjä kehottaa jakamaan toiminnallisuutta useampaan luokkaan.

Lisäksi, Wolf-kieli tarkastaa jokaisen luokan, muuttujan ja metodin nimet. Jos nimet ovat kohtuuttoman lyhyitä tai tuntuvat sattumanvaraisesti valituilta, Wolf-kääntäjä antaa varoituksen käyttäjälle. Tämän säännön takana on eräs Kirjoittajan ylläpitämä ohjelma, jossa oli globaaleja muuttujia, joiden nimet olivat tasoa r,j ja tmp1…tmp11. Ylläpito oli vaikeaa jo pelkästään huonosti nimettyjen muuttujien takia.

C++-kielen esikääntäjät ovat mainio keksintö – jos niitä käytettäisiin oikein. Esikääntäjän makrot ovat oleellisesti metaohjelmointia, joka on tuettu Wolf-kielessä, mutta vähemmän vaarallisella tavalla. Jos metaohjelmoinnissa on bugi, se on helpommin jäljitettävissä kuin monimutkaisista makroista. Korostettakoon, että esikääntäjän makroissa sinänsä ei ole mitään vikaa. Niitä vain käytetään väärin.

Varoitusten antaminen kehittäjille on keppiä huolimattomuudesta. Wolf-kielen abstraktius ja suoraviivaisuus tarjoaa porkkanaa kehittäjille. Wolf-kielen syntaksi on melko suoraviivainen – kuten myöhemmin todetaan. Wolf-kielen syntaksi pitää suunnitella yksikäsitteiseksi ja niin helpoksi, että kehittäjän on helppo ilmaista, mitä tarkoittaa ohjelmaa tehdessään. Kirjoittajan haaveena on luoda hyvät kirjastot Wolf-kieleen, jotta samaa ratkaisua ei tarvitsisi keksiä uudelleen.

Kaikista varokeinoista huolimatta nämäkin keinot ovat vain laastaria ongelmaan. Kirjoittaja on varma jo ennen kielen suunnittelun lopettamista, että Wolf-kielellä voi tehdä huonoja ohjelmistoja. Wolf-kieli kuitenkin varoittaa, jos koodin laatu heikkenee. Kokemuksen perusteella huonoja ohjelmistoja ei koskaan tarkoituksella suunnitella ylläpidon painajaisiksi, vaan ne mätänevät kiireellinen pikakorjaus tai pieni oikaisu kerrallaan omaan monimutkaisuuteensa. Wolf-kieli varoittaa kehittäjiä ohjelmiston monimutkaisuuden kasvamisesta, mutta ei estä kääntämistä. Jos kehittäjä välttämättä haluaa ampua itseään jalkaan, kirjoittaja uskoo sen olevan hänen päätöksensä.

Vielä teknisiä ratkaisuja suurempi on ongelma on viestinnän epäonnistuminen miltei aina. Kommunikointi ja sen onnistuminen olisi hyvin tärkeää olio-ohjelmoinnissa. Luokkia tehdessä arkkitehti tai suunnittelija tekee aina oletuksia ongelma-alueesta. Jos viestintä asiakkaan kanssa epäonnistuu, oletukset saattavat olla virheellisiä. Jos viestintä kehittäjien ja arkkitehdin välillä epäonnistuu, ideat toteutetaan väärin. Jos kehittäjien viestintä ylläpitäjille epäonnistuu, ohjelmiston toimivat osat rikotaan helpolla korjauksella, joka rikkoo perusarkkitehtuurin oletuksia – joita harvemmin on edes dokumentoitu, koska nehän ovat itsestäänselviä.

Hyvän näkökulman kommunikoinnin vaikeuteen saa miettimällä parisuhdetta.

Vaikka kumppaninsa tuntisi miten hyvin tahansa ja vaikka väärinkäsityksen mahdollisuuden luulisi olevan melko pieni, välillä jokaisessa parisuhteessa tulee väärinkäsityksiä, jotka johtavat riitelyyn. Jos kommunikointi epäonnistuu läheisimmän ihmisen kanssa, miten se voisi onnistua paljon etäisempien ihmisten kanssa, joilla voi olla hyvinkin eri tavoite projektissa kuin itsellä?

8.4 Yhteenvetoa luvusta 8

- Huonoja olio-ohjelmistoja ei suunnitella erikseen huonoksi, ne vain taantuvat painajaismaiseksi sopaksi.
- Wolf-kieli tarkastaa ohjelman tilaa myös laadun näkökulmasta.
- Ohjelmistoissa on virheitä, jos ihminen liittyy toimintaan millään tavoin, koska ihminen on erehtyväinen.

9 Olio-ohjelmoinnin tulevaisuus

Tarve isoille, integroiduille ohjelmistoille on kasvanut koko ajan. Vaikka koneiden suorituskyky on kasvanut Mooren lain mukaisesti, ei ohjelmistojen laatu kasva yhtä nopeasti. Tämä tunnetaan myös ohjelmistotuotannon kriisinä. Sama ongelma on jatkunut läpi koko tietotekniikan historian. Tämä luku keskittyy uuteen, hypetettyyn tekniikkaan ja miten Wolf-kieli hyödyntää uusia ideoita.

9.1 Kehykset

Eräs tapa, jolla ongelmaa on hieman kierretty on kehykset[1]. Kehyksellä tarkoitetaan komponenttia, joka hoitaa yleiset rutiinitoimenpiteet ja jota voi laajentaa kohtalaisen helposti. Ongelma-alueen tyypillisimmät rutiinit voidaan siirtää kehyksen sisään ja uusien moduulien ei tarvitse toistaa samoja koodeja useaan kertaan.

9.1.1 Kehysten luokittelua

Alunpitäen kehys tarjosi vain muutamat laajennospaikat kehittäjille, jotka rakensivat oman toimintansa entisen päälle. Tätä rajoittunutta toimintaa kutsuttiin blackbox-kehykseksi. Paras suomenkielinen vastine tällä termille lienee mustalaatikko tai suljettu kehys. Esimerkkejä tämäntyyppisestä toiminnasta on esimerkiksi awt-kirjasto javan puolella.

Joskus toimintaa pitää muokata jossain kriittisessä kohdassa radikaalistikin ja koska ohjelmiston modulaarisuus on yleisestikin hyvä asia, on myöhemmin tullut niin sanottuja lasilaatikko-kehyksiä. Niissä toimintaa voi laajentaa enemmänkin ja kehyksen toiminta on helpompi ymmärtää. Esimerkkinä tämäntyyppisestä ajattelutavasta on javan Swing-käyttöliittymäkirjasto, jota on melko helppo muokata kuhunkin tilanteeseen sopivaksi.

Kirjoittaja itse pitää näitä seikkoja kuitenkin merkityksettöminä verrattuna siihen, että lasilaatikkokehyksen testaaminen on helpompaa. Hyvä testaus vähentää riskiä siihen, että jokin luokka- tai olio-rakenne on laajentaessa hajonnut ja kehyk-

[1] Kehyksen englanninkielinen termi on framework.

sen toimintaan voidaan luottaa enemmän.

9.1.2 Mitkä ovat kehysten suurimmat kompastuskivet?

Suurin ongelma kehyksissä on, että niitä ei käytetä. Ohjelmistotalojen ulkopuolisista kehyksistä ei tiedetä talon sisällä. On melko turhauttavaa etsiä valmiita komponentteja ja huomata, että mikään ehdokkaista ei täytä kaikkia vaatimuksia.

Edelleen, jokaisessa kehyksessä on omat rajoituksensa. Jos kehyksen toimintaa ja laajennusmahdollisuuksia ei ole kunnolla dokumentoitu ja testattu, sen käyttäminen on aina riski yritykselle. Kirjoittaja on itsekin käyttänyt lukuisia tunteja ihmetellessä, miten jokin tietty toiminta ei mene kuten maalaisjärki sanoisi ja syyksi on paljastunut jokin dokumentoimaton "ominaisuus", josta ei tietenkään ole kirjoitettu selkeää varoitusta dokumentaatioon.

9.1.3 Kehykset Wolfissa

Myöhemmin toteamme, että Wolf-kieli itsessään on melko suppea. Jotta kaikki tarpeellinen toiminta saataisiin tehdyksi, on standardikirjaston oltava melko laaja toiminnallisuudeltaan. Kirjoittaja ei kuitenkaan halua standardikirjastojen olevan suuria kooltaan, joten niissä on käytettävä melko aggressiivisesti kehyksiä.

Wolf-kieli on tarkoitettu pelien ohjelmointiin ja että pelimaailman objektit noudattava hyvin usein samantapaisia kaavoja mitä muutkin objektit, mutta niitä on vain pystyttävä muokkaamaan melko radikaalistikin. Pelien rakenne ja säännöt voivat muuttua nopeastikin kehityksen edetessä ja ilman kehyksiä jouduttaisiin muutokset tekemään useisiin eri luokkiin. Jos Wolf-kieli yleistyy ja sen ympärille kehittyy yhteisö, tarkoituksena on liittää hyväksi havaitut kehykset standardikirjastoon mukaan.

9.2 Hajautetut järjestelmät

Hajautetut palvelut ovat eräs kuumimmista aiheista ohjelmistotuotannossa – ainakin jos uskoo hypetystä, mikä asian ympärillä on tällä hetkellä.

9.2.1 Hajautusta tehokkuuden ja toimintavarmuuden takia

Kuten aiemmin todettiin, ohjelmistoteollisuus on ollut kriisissä jo alkuajoistaan asti. Osaan ongelmista ei vain ole olemassa järkevää algoritmia[2] ja on useimmiten halvempaa ostaa tehokkaampia alustoja ohjelmistoille kuin maksaa ohjelmoijille optimoinnista. Tähän ongelmaan on haettu ratkaisua hajautetuista palveluista. Käyttäjien tarvitseman palvelun vaatima kuorma sekä verkkoliikenteelle että laskentateholle jaetaan useamman koneen kesken. Yhdessä koneet muodostavat klusterin, jonka tehokkuus yhteensä on suurempi kuin erillisten koneiden.

Toinen syy palvelun hajauttamiseen on koneiden ja verkkoyhteyksien epävarmuus. On melko yleistä, että liiketoiminnan kannalta tärkeät toiminnat on varmennettu toisella palvelimella. Jos ensisijainen palvelin hajoaa tai yhteydet siihen ovat poikki, voidaan ottaa käyttöön toinen palvelinkone, joka palvelee asiakkaita.

9.2.2 Tarvetta OO-kommunikoinnille

Aiemmin klusterin eri koneiden – eli solmujen – yhteydenpito piti tehdä erikseen jokaiselle palvelulle tai kommunikointitapa oli tarkasti sidottu ongelma-alueeseen. Tämä nosti kehityskulut klusteroiduille ohjelmistoille korkeaksi. Olio-paradigman yleistyessä oli entistä enemmän tarvetta yleiselle tavalla määritellä olioita, jotka keskustelevat keskenään tietoverkon välityksellä.

Corba oli ensimmäinen laajalle levinnyt kommunikointiprotokolla, jolla fyysisesti eri koneilla olevat oliot pystyivät kommunikoimaan keskenään. Sen ongelmana oli monimutkaisuus ja protokollan paisuttaminen tarpeettomasti.

SOAP – eli Simple Object Access Protocol – oli Corbaa yksinkertaisempi ja käytti XML:ää kommunikointiin eri olioiden välillä. Hyvänä puolena XML:n käyttämisestä oli, että sitä oli helppo ihmisenkin ymmärtää. Huonona puolena oli vähemmän yllättäen järkyttävän raskas XML:n jäsentäminen.

Java kehitti oman protokollansa, jolla eri oliot pystyivät kommunikoimaan keskenään. Tätä lähestymistapaa käyttää myös Wolf-kieli. Voisi olla hyödyllistä kehittää kommunikointiprotokolla, joka sopisi erityisesti peliympäristöihin, mutta aihe on turhan vaativa tämän teoksen osana.

On huomattava, että peliympäristöissä hajautus on tarpeen eri syistä kuin normaalissa liiketoiminnassa[3]. Pelit eivät hajauta toimintaansa, jotta asiakkaan koneella

[2] Kauppamatkustajan ongelma on eräs tunnetuimpia esimerkkejä tästä

[3] Tähän voisi tietysti miettiä, mikä on normaalia liiketoimintaa ja mikä ei...

olisi vähemmän kuormaa tai tiedon hallinnoiminen olisi helpompaa. Pelit tarvitsevat kommunikointia eri koneiden välillä, jotta peli pyörisi samaan tahtiin muiden pelaajien kanssa. Jos pelihahmo kuolee pelaaja A:n koneella, sen on kuoltava myös pelaaja B:n koneella. Samoin tuulen henkeys, oudot valoilmiöt ynnä muut sellaiset on oltava samanlaisia jokaisen pelaajan koneella.

Käytännössä verkon latenssit estävät täysin reaaliaikaisen toiminnan toteuttamisen. Wolfin eräs suurista haasteista on saada peli näyttämään siltä, että se toimii reaaliaikaisesti. Tietysti voidaan pohtia, onko järkevää liittää skriptikieltä pelissä niin matalalle tasolle kuin verkkototeutus, mutta tämä teos ottaa kantaa myös verkon toimintaan ja miten etäolioita käytetään.

Valitettavasti tämä tutkielman laajuuteen ei voi ottaa mukaan hajautusta, vaikka aihe on mielenkiintoinen.

9.3 Palvelukeskeiset arkkitehtuurit

WWW-tekniikan yleistyttyä uusin villitys on kehittää palvelukeskeisiä arkkitehtuureja. Tämä tarkoittaa karkeasti maalaisjärkeä käyttäen sitä, että perinteisestä pääohjelmasta ja asiakas/palvelin-arkkitehtuurista on luovuttu. Esimerkkejä tämäntyyppisestä ajattelusta on Google-maps, joka tarjoaa pohjan, johon muut voivat lisätä omia toiminnallisuuksiaan. Tämä ajattelutapa on lisääntynyt nopeasti Ajax-teknologian kehityksen myötä. Ajax-teknologiat on myös hyvä esimerkki liikkuvasta (eli mobile) koodista: käyttäjä hakee omalle koneelleen osan toiminnasta ja osa toiminnasta on palvelimella, kuten perinteisissä ohjelmistoissa. Oleellista on, että asiakas hakee ohjelman itselleen ja ajaa sen omassa koneessaan. Samaa ideaa käytetään myös tekoälyn parissa agenttien toteuttamisessa.

Wolf ei ole kieli, jolla luotaisiin WWW-palveluja, mutta samantyyppistä ajattelua on myös Wolf-kielen ja -alustan toteutuksessa. Wolf toimii osana pelimoottoria eikä sillä ole varsinaisesti pääohjelmaa. Eri moduulit luovat oliota ja käyttävät niitä määräämättömän ajan. Osaa olioista käyttää myös muut moduulit. Palvelimella olevien toimintojen päälle voidaan rakentaa esimerkiksi tehokkaampi tekoäly, parempi säämalli, realistisempi ekonomia ja niin edelleen. Wolf-kieltä voidaan käyttää myös tämäntyyppiseen toimintaan tulevaisuudessa. Javan virtuaalikone on turhan raskas ja laaja, jotta sitä voisi käyttää mobiilikoodin toteuttamiseen pelialustoissa.

9.4 Yhteenvetoa luvusta 9

Vaikka useimmat tässä luvussa mainituista tekniikoista ovat vain hypeä, pyrin Wolfia suunnitellessa ottamaan huomioon myös tulevaisuuden suuntaukset. Palvelukeskeiset arkkitehtuurit yleistyvät melko varmasti vielä paljon nykyisestä ja peleissä on pakko ottaa huomioon verkkototeutus.

Osa II

Metaohjelmointi

10 Metaohjelmointi Wolfissa

De profundis clamavi ad te Domine – Psalm 130

Tämä kappale kertoo, mitä metaohjelmointi on, mitä sen avulla voi tehdä ja miten Wolf-kieli käyttää metaohjelmointia.

10.1 Mitä metaohjelmointi tarkoittaa?

Metaohjelmointi tarkoittaa lyhyesti ohjelmaa, joka generoi toisia ohjelmia. Kirjoittajan omien kokemusten perusteella metaohjelmointi myös pienentää kuilua ohjelman datan ja sitä käsittelevien aliohjelmien välillä: jos ohjelmalla voi ohjelmoida ohjelmia – ja oleellisesti itseään – voi ohjelmakoodia ajatella vain eräänä tiedon ilmenemismuotona.

Wolf-kielen tapauksessa metaohjelmointi tarkoittaa sitä, että Wolf-ohjelma voi muuttaa omaa koodiaan ja toimintaansa ajon aikana. Tämä on välttämätön vaatimus, koska peliympäristön on toimittava jatkuvasti eikä katkoksia pitäisi tapahtua, jos esimerkiksi pelien juonta tai tekoälyä muutetaan.

Toinen syy metaohjelmoinnin käyttöön on, että sillä voidaan tehdä monia asioita, jotka useimmissa kielissä ovat kielen ominaisuuksia. Eräs Wolf-kielen tavoitteista on olla pieni kieli, joka on nopea ymmärtää ja muokata. Wolf-kieli ei kuitenkaan voi olla suppea tai pieni toiminnallisuudeltaan, joten kirjastojen on oltava monipuolisia. Tämä johtaa helposti laajoihin kirjastoihin, joita on vaikea käyttää, opetella ja ylläpitää. Metaohjelmoinnilla saman toiminnallisuuden voi saavuttaa oleellisesti pienemmällä määrällä työtä.

Luvuissa 11 ja 12 esitellyt näkökulma- ja piirreohjelmointi ovat oleellisilta osiltaan metaohjelmointia. Geneeriset säiliöluokat ovat myös metaohjelmointia, mutta niistäkin on paikallaan kertoa omassa, erillisessä luvussa 13.

Reflektiivisyydellä tarkoitetaan muun muassa ohjelman kykyä kertoa jotain itsestään. Lisäksi muuttujia voi luoda ajon aikana merkkijonojen perusteella. Tätä aihetta ei valitettavasti ehdi käsittelemään tutkielmassa, koska aihe on jo ennestään liian massiivinen.

10.2 Wolf-kielen lähestymistapa metaohjelmointiin

Metaohjelmointi on ohjelmoinnin ohjelmointia ja auttaa monissa käytännön ongelmissa, kuten myöhemmin todetaan. Metaohjelmointi on Wolf-kielen tärkein ero nykyisiin kieliin ja suurin syy, miksi Wolf-kielen suunnittelu on jatkunut tekijän kiireistä huolimatta. Jos nykyisten kielten tuki metaohjelmoinnille olisi parempi, kirjoittaja olisi hylännyt oman kielen suunnittelun jo ajat sitten.

Metaohjelmointi on myös suurin syy, miksi kielen suunnittelu on vielä noin viisi vuotta myöhemminkin edelleen pahasti kesken. Vaikka metaohjelmointi on hyvä ominaisuus, on kääntäjän tekeminen vaikeampaa, jos mihinkään kohtaan kielessä ei voi luottaa. Metaohjelmointi voi ohittaa tai muuttaa miltei kaikkea kielestä, jos sille antaa valtaa tarpeeksi – kuten aluksi oli tarkoituksena Wolf-kielessä. On kielen suunnittelijan ja kehittäjän kannalta helpompi toteuttaa kieltä vähä kerrallaan kuin yrittää kerralla luoda tarpeeksi kattava tuki metaohjelmoinnille. Tästä syystä Wolf-kielessä on sekä piirteet että näkökulmaohjelmointi tuettuna mutta erillisinä kokonaisuuksina.

Päämääränä on kuitenkin pitää erilliset kokonaisuudet helposti hallittavina ja mahdollisimman yhtenäisinä. Vaikka metaohjelmoinnin toteuttaminen on vaikeaa, sen käyttämisen on oltava helppoa. Wolf-kieli pyrkii olemaan uskollinen palvelija kehittäjälle, joka varoittaa vaarallisista rakenteista, mutta metaohjelmoinnin kanssa yleisten varoitusten ja järkevyyden tarkastaminen on vaikeaa.

Stroustrup mainitsi, että C++-kielellä voi ampua koko jalkansa irti; Metaohjelmoinnilla sekä jalan että sen ampumisen irti voi määrittää uudelleen. Kirjoittaja yritti määrittää yleistä metakieltä, jossa ei olisi juuri mitään rajoituksia, useiden vuosien ajan turhaan. Jälkikäteen arvellen tätä voi pitää eräänä kielen suunnittelun suurimmista virheistä.

10.3 Yhteenvetoa luvusta 10

Tämän luvun tärkeimmät muistisäännöt ovat:

- Metaohjelmointi on vaikea tajuta kokonaisuudessaan. Tästä syystä Wolf-kieli ei yritä olla täydellinen metakieli vaan tarjota metaohjelmoinnin yleisimmät työkalut kehittäjien iloksi ja koodin analysoimiseksi.

- Metaohjelmoinnin määritteleminen yleisellä tasolla on vaikea tehtävä.

- Metaohjelmointia kuvaavat luvut on tässä tutkielmassa jaettu erillisiin lukuihin

11 Piirreohjelmointi ja uudelleenkäytettävät moduulit

Luvussa 6 käsiteltiin perinnän ongelmia. Perinnän käyttö toteutuksen käyttämiseksi uudelleen on aina kyseenalaista suunnittelua ja johtaa ennemmin tai myöhemmin ongelmiin ohjelmiston elinkaaren aikana. Piirreohjelmointi on kohtalaisen uusi, mielenkiintoinen tapa koostaa oliota. Piirteet tunnetaan englanninkielisessä kirjallisuudessa nimellä Trait – kirjoittaja joutui miettimään sopivan suomenkielisen vastineen termille. Aihealuetta käsitellään kohtalaisen laajasti, joten sille omistettiin kokonaan oma lukunsa.

11.1 Mitä piirteet ovat?

Piirre on uudelleenkäytettävä komponentti, joka tarjoaa toiminnallisuutta luokalle [44], joka sitä käyttää.

Piirteiden kolme perussääntöä ovat:

1. Luokassa määritellyt metodit ovat järjestyksessä korkeammalla kuin piirteessä määritellyt metodit. Toisin sanoen, jos luokka A määrittää metodin M_0, joka on määritelty myös jossain sitä käyttävässä piirteessä, käyttävät piirteet luokan A metodia oman määritelmänsä sijasta. Luonnollisesti myös luokan instanssia käyttävät oliot näkevät vain luokan itsensä määrittelemän metodin M_0.

2. Jos luokka käyttää piirteitä, on piirteissä määriteltyjen metodien toiminta oltava kuten luokka olisi toteuttanut metodit suoraan itse.

3. Piirteiden koostamisjärjestyksellä ei ole väliä. Yläluokassa käytetyt piirteet näkyvät samanlaisena lapsiluokassa.

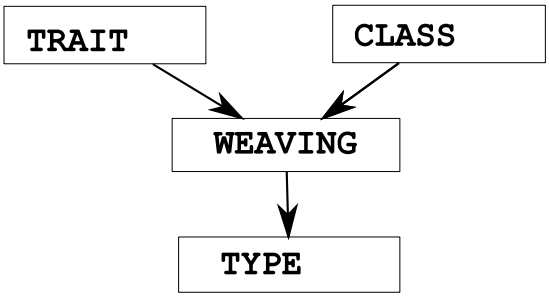
11.2 Piirteiden punominen

Piirteillä ei ole tyyppiä Wolf-kielessä, koska se saattaisi aiheuttaa ongelmia toteutukselle viitteiden [35, sivu 7] ja [14, sivut 49-50] mukaan. Piirteistä ei voi luoda instanssia. Niillä ei ole yläluokkaa, alaluokkaa tai mitään suoraa suhdetta olioon tai

luokkaan. Ne vain ovat kokoelma toiminnallisuutta, jonka voi liittää luokkaan. Piirteet voivat käyttää muita piirteitä ja saavat siten lisättyä toiminnallisuutta itseensä. Niillä ei kuitenkaan ole mitään hierarkiaa. Toteutuksen kannalta piirteet voivat olla kuten mikä tahansa abstrakti tyyppi, joten asia vaatii vielä hieman miettimistä toteutuksen kannalta. Ainakin tyyppitarkastuksissa piirteitä on käsiteltävä eri tavalla kuin normaaleja tyyppejä.

Piirteiden metodit liitetään luokan metodeihin erillisessä käännösvaiheessa, jota kutsutaan punomiseksi. Englanninkielisessä kirjallisuudessa käytetty termi punomiselle on weaving. Punominen on kuvaava ilmaisu, koska piirteiden tuomat metodit nivoutuvat luokan omiin metodeihin järjestelmällisesti ja muodostavat ehjän kokonaisuuden. Kuvassa 11.1 on annettu karkea kuvaus, miten punominen tapahtuu: luokka ja piirre liitetään yhteen erillisessä punomisvaiheessa, joka tuottaa uuden tyypin. Piirteiden punominen lisää metodeja luokkaan. Karkeasti yleistäen piirteitä voidaan ajatella leikkaa- ja liimaa-toimintona, jonka kääntäjän tekee käyttäjän puolesta.

On tärkeää huomata, että piirteet voivat täyttää toistensa vaatimuksia metodeista. Jos esimerkiksi piirre B vaatii metodin methodB ja jos piirre A määrittää metodin methodB, piirre A täyttää piirteen B vaatimukset. Tällöin piirteitä A ja B käyttävässä luokassa ei tarvitse määrittää metodia methodB.



Kuva 11.1: Piirteiden punominen Wolf-koodiksi

Koska piirteiden koostamisjärjestyksellä ei ole väliä, voidaan kaikki piirteet mieltää liitettävän haluttuun luokkaan kerralla. Jos kahdella piirteellä on metodeita, jotka ovat keskenään ristiriidassa, tilanne voidaan havaita [44].

11.3 Piirteet ovat koodimäärältään pieniä

Piirteissä on hyvin oleellista, että ne ovat pieniä ja kompakteja. Perusidea piirreohjelmoinnissa on, että palaset ovat helppoja ymmärtää ja että niitä voi käyttää helposti uudelleen. Jos piirteistä kirjoitetaan isoja, monoliittisia kokonaisuuksia, hyöty niiden käyttämisestä häviää olemattomaksi. Koska piirteet voivat käyttää toisia piirteitä oman toteutuksensa apuna, on massiivisten piirteiden tekeminen myös turhaa.

Isoissa luokissa on paljon ongelmia, jotka pätevät myös isoja piirteitä käyttäessä. Bob Koss listasi blogissaan muutamia isojen luokkien ongelmia [28]:

- Isoja luokkia on hitaampi sisäistää ja oppia.
- Isoja luokkia pitää muuttaa usein.
- Isoihin luokkiin pitää tehdä useita muutoksia ja tämä aiheuttaa ongelmia lähdekoodin hallinnan kanssa, koska muutosten yhdistämistä pitää tehdä enemmän.
- Isojen luokkien käyttäminen uudelleen on vaikeampaa, koska riippuvuuksia ympäristöön on yleensä enemmän.
- Dokumentointia pitää olla enemmän[1]
- Koodin laatu heikkenee, koska ongelmien korjaaminen on vaikeampaa ja epävarmempaa.
- Ylläpidon kustannukset nousevat, koska isot luokat ovat vaikeampia hallita ja kommunikointia esimerkiksi versioita yhdistäessä on oltava enemmän.
- Suorituskyvyn profilointi on vaikeampaa, koska luokkia ei voi testata erikseen.

Wolf-kielessä ei ole ainakaan nykyisessä toteutussuunnitelmassa tiukkoja rajoituksia luokkien koolle tai monimutkaisuudelle, mutta kielen kääntäjään on myöhemmissä versioissa tarkoitus varoittaa sotkuisesta koodista.

[1] Kossin mainitsi lähdekoodin kommentoinnin, mutta sama ongelma pätee myös muuhun dokumentaatioon.

11.4 Ero leikkaa-ja-liimaa-toimintoon

Piirteiden käyttämistä voi miettiä leikkaa- ja liimaa-komentoina, jonka tekee kääntäjä. Oleellista onkin, että kääntäjä tekee tämän operaation. Jos kehittäjä leikkaa- ja liimaa koodia, ei ole takeita konfliktien havaitsemisesta, lähdekoodia ei voi enää päivittää yhdestä paikasta ja kääntäjä ei voi optimoida toimintaa niin tehokkaasti kuin aiemmin.

11.5 Edut piirteiden käytöstä

Suurimmat edut piirteiden käytöstä ovat:

ristiriitojen havaitseminen Metodien väliset ristiriidat huomataan helposti piirteitä käyttäessä. Esimerkiksi perintää käyttäessä metodit voivat olla huolettomasti sotkussa keskenään eikä kehittäjä huomaa tätä – ennen paniikissa tapahtuu bugikorjausta, kun ohjelmisto leviää asiakkaan käytössä.

pääsy ylikirjoitettuihin ominaisuuksiin Koska piirteet instansoidaan aina per luokka, on särkyvän kantaluokan ongelma pienempi.

rajoittuneempi koostaminen Piirteiden avulla saadaan moniperinnän tuomat hyödyt miltei kokonaisuudessaan käyttöön ilman sen tuomia ongelmia. Piirteiden koostaminen on toimivampi menetelmä kuin perinnän käyttö toiminnallisuuden luomiseen.

järjestyksellä ei ole väliä Piirteet voidaan määrittää missä tahansa järjestyksessä. Tällöin on helpompi poistaa ja muuttaa piirteitä ilman konfliktia niiden välillä. Olisi periaatteessa mahdollista, että piirteiden välille muodostuisi riippuvuus huomaamatta ilman tätä vaatimusta.

punomisen tuottama koodi yhdessä paikassa Koodi, joka generoidaan punoessa, on luokan yhteydessä. Tämä helpottaa koodin ylläpitoa.

särkyvät hierarkiat Koska piirteillä ei ole hierarkiaa, niillä ei voi olla myöskään särkyvää hierarkiaa. Lisäksi kehittäjä voi aina kontrolloida, mitä piirteistä otetaan mukaan ja mitä ei.

11.6 Esimerkki piirteiden käytöstä

Vaikka Wolf-kielen kielioppi esitellään vasta luvussa 14, on tarpeen antaa yksinkertainen esimerkki piirteiden käytöstä jo tässä yhteydessä. Listauksessa 11.1 esitellään piirre CircleTrait, joka määrittää metodit circumference ja area. Luokka Circle käyttää piirrettä ja saa käyttöönsä sen metodit. Punomisen jälkeen luokkien Circle ja CircleEqual toiminnallisuus on samanlainen. Esimerkki on keinotekoinen mutta selventää ideaa piirteiden käytöstä. On huomattavaa, että Circle-luokan määritys on lyhyt verrattuna CircleEqual-luokkaan. Samaa piirrettä voi toki käyttää myös muissa luokissa, joissa halutaan käyttää samanlaista toiminnallisuutta. Isommissa projekteissa piirteitä käyttävä ohjelmisto olisi koodimäärältään selkeästi lyhyempi.

Listaus 11.1: Esimerkki piirteiden käytöstä

```
1
2  trait CircleTrait {
3     requires radius()->Double;
4
5     def circumference()->Double {
6         return 2*(Math Pi)*(self radius);
7     }
8
9     def area()->Double {
10        return (Math Pi)*(self radius)*(self radius);
11    }
12
13 }
14
15 class Circle {
16    var radius = 1.0;
17    uses CircleTrait { }
18 }
19
20 class CircleEqual {
21    var radius = 1.0;
22
23    def circumference()->Double {
24        return 2*(Math Pi)*(self radius);
25    }
26
27    def area()->Double {
28        return (Math Pi)*(self radius)*(self radius);
```

```
29      }
30
31  }
```

On huomattava, että piirteiden punonnassa ei välitetä normaaleista säännöistä muuttujien suojauksessa. Piirre CircleTrait vaatii radius-arvon ja punontavaiheessa luokan Circle muuttuja radius määritetään täyttämään piirteen CircleTrait vaatimus radiuksesta. Normaalisti luokan muuttujiin ei pääse ulkopuolelta käsiksi. Kirjoittajan näkemyksen mukaan punonta liittää piirteen osaksi luokkaa. Koska piirre on osa luokkaa, se pääsee käsittelemään luokan metodeja. Jos kehittäjä haluaa käyttää piirrettä, hän hyväksyy, että piirre pääsee käsiksi muuttujiin. Kirjoittaja katsoo, että piirteen käyttäminen on kehittäjän vastuulla.

11.7 Yhteenvetoa

Piirteet ovat siis oikein käytettynä hyvin voimakas työkalu kehittäjille. Missä tahansa kielessä on mahdollista luoda hirvittävän huonoa koodia, mutta piirteiden avulla Wolf-kielessä on hieman paremmat mahdollisuudet luoda hyvää ohjelmakoodia. Piirteiden avulla osaa perinnän ongelmista voidaan kiertää ja pitää ohjelmisto ylläpidettävänä hieman pidempään.

Piirteiden ottaminen mukaan kieleen on riski, koska käytännön esimerkit ovat Smalltalk-kielellä, joka on dynaamisesti tyypitetty oliokieli. Onneksi myös C#-kieleen on hahmoteltu kokeellinen laajennos, joka käyttää käyttää piirteitä, joten valmiita tuloksia tutkimuksesta on saatavilla [35]. Staattiset kielet eivät kuitenkaan aiheuta välttämättä kovin suuria ongelmia toteuttamiselle.

Piirreissä on kuitenkin vielä avoimia kysymyksiä jatkotutkimukselle, joita ei vielä ole ratkaistu. Eräs mielenkiintoisemmista ongelmista, on jäsenmuuttujien lisääminen piirteisiin[sivu 23] [44]. Tällä hetkellä piirteet eivät voi määrittää itselleen jäsenmuuttujia, mutta niiden lisääminen voisi olla mielenkiintoinen tutkimusaihe. Aihe on kuitenkin liian vaikea tutkielmassa pohdittavaksi.

Piirteiden avulla kehittäjä voi hajottaa luokan toimintaa useampaan paikkaan ja helpottaa ylläpitoa. Ironista kyllä, tämä on myös piirteiden heikko puoli. Piirteiden avulla toimintaa voi hajottaa liiankin helposti useampaan paikkaan. Jos piirre ei muodosta selkeää kokonaisuutta, se voi aiheuttaa edelleen ongelmia. Liiallinen pirstaloiminen on edelleen painajainen ohjelman ylläpidolle. Toiminnallisuuden hajoittamisessa piirteisiin on käytettä kohtuutta ja maalaisjärkeä.

Toisaalta, piirteiden avulla on mahdollista luoda vähällä vaivalla massiivinen luokka, jossa on paljon toiminnallisuutta. Luokkien vastuualueet on edelleen mietittävä tarkasti, vaikka piirteiden avulla voi luoda ison luokan vain käyttämällä piirteitä. Isot, kömpelöt luokat ovat isoja ja kömpelöitä, vaikka ne olisi toteutettu vähällä vaivalla käyttämällä piirteitä.

Vaikka piirteohjelmointi helpottaa oleellisesti kehittäjän ja ylläpitäjän työtä, se ei poista tarvetta luovalle ajattelulle. Muistutus lienee paikallaan, koska kirjoittajan kokemuksen mukaan uusia, lupaavia tekniikoita ylimainostetaan ja käytetään käytetään innokkaasti (myös) väärin.

12 Näkökulmaohjelmointi

Tämä luku antaa erittäin lyhyen yhteenvedon näkökulmaohjelmoinnin perusteista.

12.1 Mitä näkökulmaohjelmointi on?

Kirjoittaja oli mukana kurssilla Ohjelmointikielten periaatteet vuonna 2002. Seminaarityömme aiheena oli aspektien eli näkökulmien käyttö ohjelmia tehdessä. Päätösseminaaria varten tehdyssä artikkelissa totesimme, että "näkökulma on ohjelman läpitunkeva (cross-cutting) yhtenäinen ominaisuus, joka ei kuitenkaan ole toiminnallinen kokonaisuus eikä siten klassisten paradigmojen (FP, POP, OOP) menetelmin modularisoitavissa" [3]. Näkökulmaohjelmoinnilla ohjelmaa voi optimoida, siisteyttää ja pitää ohjelman eri osat toisistaan erillisenä. Tämä vähentää riippuvuutta eri osien välillä, joka puolestaan helpottaa komponenttien uudelleenkäyttöä muissa ohjelmissa [20].

Kuten piirteillä, myöskään näkökulmilla ei ole tyyppiä: niitä ei voi käyttää ohjelmassa tyyppitunnisteena, niillä ei voi alustaa muuttujia eikä niitä käsitellä tyyppitarkastuksessa. Näkökulmien määritelmät Wolf-kielessä ovat vielä hieman kesken. Esimerkiksi näkökulmat olisi järkevintä määrittää tyyppiparametreina, mutta niiden määrittäminen järkevästi on vaikeampaa.

12.2 Miksi näkökulmaohjelmointia?

Nykyisissä ohjelmissa on useita erilaisia näkökulmia. Esimerkiksi tyypillistä verkkosovellusta tehdessä pitää ottaa kantaa tietoturvaan, käyttäjien hallintaan, tiedon säilytykseen, käyttöliittymään ja moniin muihin asioihin. Kaikki nämä voidaan hoitaa perinteisellä olio-ohjelmoinnilla.

Huonona puolena eri näkökulmissa on, että ne menevät usein ristiin. Esimerkiksi tietoturvaa ei voi täysin erottaa muista osista ohjelmaa. Debug-tiedon kerääminen ja suorituksen analysointi vaikuttaa myös useimpiin luokkiin. Myös poikkeukset vaikuttavat ohjelman suoritukseen radikaalisti. Näkökulmaohjelmointi ratkaisee osan näistä ongelmista. Se pyrkii kapseloimaan nämä ei-toiminnalliset näkökulmat

yhteen ja pitää ne erillään muista näkökulmista. Näkökulmaohjelmoinnissa näkökulma on erillinen kokonaisuus, jonka tarkoituksena on mallintaa tiettyä näkökulmaa ohjelmaan.

Perinteisesti nämä globaalit näkökulmat monimutkaistavat ohjelman tekemistä ja niiden toteuttaminen synnyttää helposti niin kutsuttua spagettikoodia, jota on mahdotonta ylläpitää, käyttää uudelleen tai edes muuttaa, kun vaatimukset ohjelmistolle vaihtuvat ja lisääntyvät.

12.3 Näkökulmien käyttö ja toteutus

Kuten piirteet myös näkökulmat punotaan normaaleiden luokkien sekaan. Erona piirteiden punomiseen on, että näkökulmat eivät lisää metodeja vaan koodia olemassa oleviin metodeihin. Lisäyksillä on kuitenkin rajoituksensa: punominen voi lisätä suorittavaa koodia vain ennen, jälkeen tai sekä ennen että jälkeen metodin määritystä. Näkökulmat tulevat siis metodia kutsuvan ja kutsun käsittelevän olion väliin.

Koska näkökulmat voivat toimia tällä tavoin eräänlaisina portinvartijoina, niiden avulla on helpohko toteuttaa esi- ja jälkiehdot, joita käytetään ohjelman varmistuksiin – ilman muutoksia lähdekoodiin luokan puolella. Näkökulmilla voi myös lisätä virheenkäsittelyä ja lokitietojen keräämistä ohjelmasta kohtalaisen vaivattomasti. Wolf-kieli siis tukee jossain määrin sopimuspohjaista ohjelmointia. Rajapinnat ovat erinomainen kohde näkökulma-ohjelmoinnin käytölle, koska niillä voidaan lisätä esi- ja jälkiehtoja lähdekoodiin. Tällöin luokan toteutus voidaan varmentaa ja uusien luokkien virheet havaitaan nopeammin.

Toisaalta, näkökulmaohjelmoinnilla voi tehdä myös luovia, arveluttavia pikakorjauksia ongelmiin. Laiska ja kuriton kehittäjä voi luoda näkökulmien avulla jokaiselle mahdolliselle kutsulle erilaisen paluuarvon. Vaikka paras ratkaisu olisi korjata virheellisesti määritelty luokka, voi näkökulmien avulla lykätä ikävää ja työlästä korjaamista kunnes se siirtyy toisen kehittäjän vastuulle – jonka on sisäistettävä sekä luokka että siihen liittyvät näkökulmat ennen korjauksen aloittamista.

12.4 AspectJ ja näkökulmaohjelmoinnin hyödyt

Kurssilla käytetty kieli näkökulmaohjelmointiin oli AspectJ, joka on Java-kielen päälle tehty laajennos näkökulmaohjelmointia varten. Lyhyen yhteenvedon kielestä voi

lukea lähteestä [27]. AspectJ-kielessä normaaliin lähdekoodin sekaan lisätään erilaisia aspekteja, jotka ovat näkökulmia ohjelman toimintaan. Näkökulmien käyttäminen tapahtuu seuraavasti:

1. Näkökulmat kirjoitetaan erillisinä aspect-tyyppeinä koodin sekaan *erillään* normaalista tuotantokoodista.

2. Näkökulmat leikkaavat koko ohjelman halki. Nämä leikkauskohdat pitää määrittää erikseen. Oleellinen perusasia on, että leikkauskohtia ei tarvitse sotkea muun koodin sekaan.

3. AspectJ-kääntäjä kääntää AspectJ-lähdekoodin normaaliksi Java-kieliseksi ohjelmaksi.

4. Java-kielinen ohjelma käännetään kuten normaali Java-ohjelma.

AspectJ on siis kääntäjä. Automaattisesti generoitu koodi on yleensä aina rumaa eikä AspectJ tehnyt poikkeusta tähän nyrkkisääntöön ainakaan kurssia varten laaditun artikkelin aikaan. Hyvänä puolena oli, että AspectJ-kieli itsessään oli selkeää. Generoituun koodiin ei tarvitse koskea luomisen jälkeen. Lisäksi pikainen vilkaisu generoidusta koodista paljasti, että jo artikkelin kirjoittamisen aikaan AspectJ-kääntäjä teki käännöksen kohtalaisen hyvin.

Wolf-kielen eräs ensimmäisiä vaatimuksia oli näkökulmaohjelmoinnin tukeminen, koska sillä saavutetaan muun muassa seuraavia etuja:

- Ohjelmien ylläpito tulee helpommaksi, koska ei-toiminnalliset osat ohjelmasta voidaan erottaa muusta koodista.

- Erilaisia ei-toiminnallisia osia voidaan käyttää uudelleen. Tämä on erityisen hyödyllistä koodin analysoinnissa ja logitietojen keräämisessä. Myös virhekäsittelyyn liittyvää logiikkaa voidaan käyttää uudelleen.

- Ohjelman abstraktion taso nousee ja koodia on helpompi ymmärtää, joka on kirjoittajalle tärkein argumentti näkökulmaohjelmoinnin puolesta.

- Ohjelmaan voi tehdä siististi helppoja optimointeja korkealla tasolla.

- Ohjelmiston yhtäaikainen kehittäminen helpottuu, koska esimerkiksi tietoturvaan liittyviä osia voidaan tehdä samaan aikaan kuin esimerkiksi käyttöliittymää.

12.4.1 Näkökulmat Wolf-kielessä verrattuna AspectJ-kieleen

AspectJ-kielessä liitoskohdat kirjoitettiin säännöllisillä lausekkeilla. Tämä oli kirjoittajan oman näkemyksen mukaan hyvin huono tapa kirjoittaa koodia, koska kieli ei enää elänyt muiden tarpeiden mukana. Jos metodien tai muuttujien nimiä vaihdettiin, liitoskohdat lakkasivat toimimasta, mutta käännös meni läpi onnistuneesti.

Wolf-kielessä näkökulmat voidaan liittää normaalin koodin sekaan luokan ulkopuolelta, kuten AspectJ:ssä, mutta luokan ja muuttujien nimet pitää erikseen määrätä. Tässä lähestymistavassa on seuraavia etuja verrattuna säännöllisten lausekkeiden käyttöön:

- Jos metodien nimiä muutetaan ja liitoskohta käyttää tätä metodia, käännösvaiheessa huomataan virhe. Tämä vähentää virheen korjaamiseen käytettyä aikaa.
- Kääntäminen nopeutuu, kun ei enää tarvitse käyttää vertailuun regexp-lausetta.
- On helppoa generoida koodi siten, että liitoskohta muuttaa sitä käyttävien luokkien rakennetta. Tällä tavoin voidaan optimoida hieman näkökulmien generoiman lähdekoodin määrää. Tämä puolestaan vähentää muistinkulutusta lopullisesta ohjelmassa.

Oleellinen ero AspectJ-kielen ja Wolfin välillä on, että Wolf-kieli on suunniteltu alusta asti metakieleksi – joskin metaominaisuuksia toteutetaan vain vähän kerrallaan. AspectJ perustuu Java-kieleen, joka ei tue metaohjelmointia suoraan[1].

12.5 Näkymien määrittely

Kuten edellä todettiin, Wolf-kielessä näkökulmia ei liitetä muuhun ohjelmalogiikkaan säännöllisillä lausekkeilla. Jos jokainen näkökulma joutuu käyttämään luokan muuttujan tai metodin nimeä, on näkökulman toteuttaminen aivan liian työlästä, jotta kehittäjät ominaisuutta käyttäisivät. Lisäksi näkökulmia joutuisi muuttamaan joka kerta, kun luokkaan lisätään tai poistetaan metodi tai muuttuja.

Ratkaisuna on ryhmitellä muuttujat ja metodit jotenkin. Näkökulma ei käsittele yksittäisiä metodeja tai muuttujia vaan metodien ja muuttujien ryhmää, jolla on

[1] Reflektio ja 5. version geneeriset luokat ovat metaohjelmointia, joten Java-kielenkin tuki metaohjelmoinnille on parantunut

jokin tunniste. Tätä ryhmittelyä kutsutaan Wolf-kielessä näkymäksi. Luokan kehittäjä lisää tarpeelliseksi katsomansa metodit ja muuttujat näkymään. Näkökulmat lisäävät koodia näkymässä oleviin metodeihin ja muuttujiin. Tällöin näkökulmaa ei tarvitse muuttaa luokkaa muuttaessa.

Näkymän hyvänä puolena on sekin, että koska luokan ylläpitäjä itse voi huolehtia näkymän päivityksestä, on pienempi todennäköisyys versiohallinnan konflikteihin[2]

Kehittäjät voivat kielen lopullisessa versiossa määrittää näkymiä itse lisää. Wolf-kieli määrittää jokaiselle luokalle ainakin seuraavat näkymät:

allMethods Kaikki metodit tallennetaan tähän taulukkoon. Käyttäjä ei voi muuttaa tätä näkymää.

allVariables Tähän näkymään tallennetaan kaikki muuttujat. Tätä käytetään enimmäkseen muiden näkymien alustuksiin. Luokan määrittäjä ei voi muuttaa tätä näkymää.

notPersistant Tähän näkymään tallennetaan muuttujat, joita ei haluta tallentaa tietokantaan. Koska useimmiten kaikki muuttujat halutaan tallentaa tietokantaan, negaation käyttö vähentää pakollista kirjoittamista. Tähän näkymään ei voi tallentaa kuin luokan muuttujia.

userinterface Tähän tallentaan perusnäkymä käyttöliittymästä. Tähän näkymään ei voi tallentaa kuin muuttujia tai metodeja, joiden tyyppi on ()->Void. Tämä rajoitus on – jälleen kerran – tietoinen ratkaisu. Tätä näkymää tarvitaan käyttöliittymän luomiseen, josta kerrotaan tarkemmin luvussa 19.

shared Luokan kaikkien instanssien kesken jaetut metodit tai muuttujat. Jos metodi on määritelty näkymään shared, se ei voi viitata muuttujiin self ja super, koska ne on määritelty pelkästään oliossa.

unittests Tämä käsitellään tarkemmin kappaleessa 12.6. Tähän näkymään ei voi tallentaa kuin metodeja, joiden tyyppi on ()->Void. Myöskään näillä metodeilla ei ole pääsyä self ja super muuttujiin.

[2] Huoli versiohallinnan hajoamisesta ei tietenkään kuulu kielen suunnittelijan tehtävään, mutta Wolf-kieli pyrkii olemaan ystävällinen, uskollinen palvelija kehittäjälle kuten koirat isännälleen.

private Yksityiset metodit, joita ei haluta paljastaa julkiseen rajapintaan, pitää tallentaa tähän rajapintaan. Tämä on hyödyllinen määrittää ennen luokan julkaisua, koska mitä enemmän metodeja on määritelty julkiseen rajapintaan sitä helpompi luokkaa on käyttää väärin. Tähän näkymään ei voi tallentaa kuin metodeja. Muuttujat ovat aina yksityisiä Wolf-kielessä, joten niitä ei tarvitse erikseen määrittää yksityisiksi.

frozen Tähän näkymään määritellään metodit ja muuttujat, jotka halutaan määrittää lopullisiksi. Lapsiluokat eivät voi muuttaa tässä näkymässä olevien metodien toteutusta.

Näkymien määrityksen jälkeen näkökulma-ohjelmointi niputtaa lähdekoodin oikein. Kielessä on esimerkiksi sisäänrakennettuna näkökulma, joka tarkastaa, ettei frozen näkymässä olevia metodeja tai muuttujia muuteta luokissa. Kielessä itsessään ei esimerkiksi ole lainkaan private-, protected- tai public- määrittelyjä mutta vastaava toiminta voidaan määrittää näkökulmaohjelmoinnin puolelta. Näkymiä voi määrittää itse lisää ja käyttää niihin näkökulma-ohjelmointia.

Listauksessa 12.1 on esimerkki näkymien käytöstä. Luokan MyBase metodi myPrivate on määritelty näkymään private, joten sitä ei voi kutsua luokassa MyChild. Lisäksi myMethod on frozen, joten sitä ei voi muuttaa lapsiluokassa. Koska lapsiluokka yrittää rikkoa näitä sääntöjä, ohjelma ei käänny. Listauksessa käytetään Wolf-kielen kielioppia, joka esitellään tarkemmin luvussa 14.

Listaus 12.1: Esimerkki näkymien käytöstä

```
1
2  class MyBase {
3      view private := (myPrivate);
4      view frozen := (myFrozen);
5
6      def myFrozen(Int,Int)->Int {
7          param a;
8          param b;
9          return (self myPrivate : a b);
10     }
11
12     def myPrivate(Int,Int)->Int {
13         param a;
14         param b;
15         return a+b;
16     }
```

```
17 }
18 // invalid class
19 // since frozen override and access to private
20 class MyChild {
21    def myFrozen(Int,Int)->Int {
22       param a;
23       param b;
24       System out println "testing";
25       return (super myPrivate: a b);
26    }
27
28 }
```

12.6 Yksikkötestit luokkien sisälle

Yksikkötestauksen pitäisi kirjoittajan mielipiteen mukaan olla pakollinen osa luokan toimintaa. Ongelmana testien luomisessa on usein, että yksikkötestit vaativat olion tilan tarkastelua ja siten pääsyä myös luokan muuttujiin. Tämä taas rikkoo olio-ohjelmoinnin ideologiaa tiedon ja toteutuksen piilotuksesta. Ratkaisuna ongelmaan on määrittää metodit, jotka testaavat olion toimintaa, luokan sisälle. Testien kirjoittaminen ja luominen helpottuu tällä tavoin oleellisesti.

Liittäminen voitaisiin toki tehdä erillisellä avainsanalla test, jota seuraisi metodin määritys. Tämä kuitenkin paisuttaisi kielen kielioppia tarpeettomasti, joten ratkaisuna on ollut luoda näkymä unittests, johon listataan metodit, joita halutaan yksikkötesteihin mukaan. Luokka muuttuu siis itsessään yksikkötestiksi testauksen näkökulmasta.

Huonona puolena tässä on, että se tuntuisi paisuttavan luokan kokoa. Luokan koodirivien paisuminen on kuitenkin näennäistä, koska luokkaa on joka tapauksessa testattava ja erillinen yksikkötesti vaatisi *enemmän* koodirivejä kuin näkymän käyttäminen. Entäpä sitten muistinkulutus käännettävissä ohjelmissa? Tässäkään kohtaa luokkaa ei tarvitse paisuttaa tarpeettomasti, koska kääntäjä voi kääntää valmiista, tuotantokäyttöön menevästä luokasta testitapaukset pois.

Onko testauksen sotkeminen normaaleihin luokkiin sitten muuten paha? Tästä voitaisiin kiistellä loputtomiin, mutta kirjoittaja uskoo – kuten moni muukin ketterien menetelmien tunnustaja – että testauksen pitäisi olla erottamaton, kiinteä osa mitä tahansa ohjelmistoprojektia. Liika testien eristäminen vain johtaa testitapausten laiminlyöntiin ja niiden jättämiseen päivittämättä luokkaa muuttaessa.

Yksikkötesteiksi liitettävissä metodeissa on kuitenkin rajoituksia: Metodien tyypin on oltava ()->Void, koska yksikkötestit eivät saa parametreja mistään vaan suorittavat tietyn, rajoitetun ja suljetun testin. Metodeita ei voi kutsua normaalista ohjelmakoodista, koska yksikkötestit on tarkoitettu vain yksikkötestien käyttöön ja niiden kutsuminen muista luokista, jotka eivät kuulu testaukseen, rikkoisi olio-ohjelmoinnin perusperiaatteita kapseloinnista ja abstraktiosta.

Lopuksi on myös korostettava, että luokkien testaus yksitellen ei riitä vaan tarvitaan laajempaakin testausta. Jos Wolf-kielessä luokan perii luokasta Unittest, koko luokka hävitetään lopullisesta käännöksestä. Tällöin kehittäjät voivat määrittää testien apuluokat erillisiin luokkiin ja helpottaa testausta lisää.

12.7 Käytännön esimerkki näkökulmista

Tähän mennessä on käsitelty vain teoriaa näkökulmaohjelmoinnista. Listaus 12.2 on käytännön esimerkki näkökulman käytöstä. Listauksessa määritetään luokka MyClass ja näkökulma LogMessage, joka leikkaa luokkaa MyClass. Punomisen jälkeen luokan MyClass toiminta vastaa melko tarkkaan luokkaa MyEqual, joka ei käytä mitään näkökulmia. Haluttuja näkökulmia voi luoda itse lisää, mutta on hieman ongelmallista, missä järjestyksessä ne on liitettävä luokkaan.

Listaus 12.2: Esimerkki näkökulman käytöstä

```
1  aspect LogMessage {
2     around MyClass allMethods {
3          System log println "Before";
4          proceed();
5          System log println "after";
6     }
7  }
8
9  class MyClass {
10    def myMethod(Int,Int)->Int {
11       param a;
12       param b;
13       return a+b;
14    }
15 }
16
17 class MyEqual {
18    def myMethod(Int,Int)->Int {
```

```
19          param a;
20          param b;
21          //this is from aspect
22          System log println "Before";
23          //this is from original method
24          myMethod := a+b;
25          //this is from aspect
26          System log println "after";
27      }
28  }
```

12.8 Yhteys piirreohjelmointiin

Piirteet ovat selkeästi erittäin lähellä näkökulmaohjelmointia. Näitä kahta osa-aluetta ei vielä ole kunnolla tutkittu ja yhdistetty, mutta Wolf-kieltä toteuttaessa varmasti löytyy joitain yhteyksiä. Molemmat yhdistävät ominaisuuksia ja kutovat valmiin ohjelman pienistä palasista. Etenkin punomisessa on hyödyllistä tutkia, voiko sekä näkökulmien että piirteiden määrittelyä yhdistää yhdeksi kokonaisuudeksi. Eräs ongelma piirteiden punomisen jälkeen on päättää, missä kohtaa näkökulmat pitää määrittää luokkaan kiinni. Toisin sanoen liitospisteiden määrittely on vielä auki Wolf-kielen suunnittelussa.

AspectJ-kielellä on pystytty emuloimaan piirteitä jo pelkässä näkökulmaohjelmointikielessä [12]. Tutkimuksessa käytettiin näkökulmia mallintamaan piirreohjelmointia seuraavasti:

1. Määritetään rajapinta, joka mallintaa piirrettä.
2. Määritellään näkökulma (eli aspect AspectJ-kielessä), joka määrittää itse toteutuksen.
3. Liitetään näkökulman koodi rajapintaan liitospisteillä.
4. Luokka perii rajapinnan, joka mallintaa piirrettä.
5. Näkökulmien punonta liittää näkökulmassa olleen koodin automaattisesti mukaan luokkaan.
6. Luokkaan ei tarvitse tehdä itse koodia vaan pelkkä rajapinnan periminen riittää.

Tulosten analysoinnissa todettiin, että emulointi onnistuu näkökulma-ohjelmoinnin avulla, mutta kaikkia piirteiden vaatimuksia ei voitu toteuttaa kattavasti. Näkökulmaohjelmoinnissa on yhteinen piirre piirteiden kanssa: molempien tarkoituksena on liittää hajallaan oleva toiminnallisuus yhdeksi helposti hallittavaksi kokonaisuudeksi. Artikkelin tutkimustyö perustui tähän tietoon. Piirteillä ja näkökulmilla on kuitenkin erillisiä piirteitäkin: piirteet pystyvät havaitsemaan konfliktit ja punovat koodin yhteen luokassa. Niitä ei voi käyttää luokan ulkopuolella ja piirteet ovat toiminnallisuuden apuvälineitä. Näkökulmat taas keskittyvät ei-toiminnalliseen osaan ohjelmasta eikä niitä voi käyttää ilman leikkauspisteitä.

Oleellista on kuitenkin huomata, että sekä piirre- että näkökulmaohjelmointi on osa metaohjelmointia.

12.9 Yhteys poikkeuksiin

Poikkeusmenettely on eräs näkökulmaohjelmoinnin osa-alue. Sekä poikkeukset että näkökulmat leikkaavat halki ohjelman, eivät ole varsinaiseen toimintaan liittyviä ja näkökulmaohjelmoinnin eräs ensimmäisiä ja laajimmin käytetyistä ominaisuuksista on poikkeuksien käsittely. Eräs hypetyksen aihe näkökulmien pulpahtaessa pinnalle oli nimenomaan poikkeuskäsittelijöiden kierrättäminen ohjelmasta toiseen.

On siis ainakin jossain määrin järkevää tarkistaa, voiko näkökulmaohjelmoinnin ja poikkeuskäsittelyn toteutusta kierrättää ja käyttää toistensa hyväksi.

Wolf-kieli käyttää näkökulmaohjelmointia hyväkseen melko paljon ja etenkin C++-kielellä tehtyjen osien liittämiseksi Wolf-komponentteihin. Näkökulmat siisteyttävät oikein käytettynä koodia ja helpottavat usean kehittäjän yhteistyötä ilman koodin menemistä ristikkäin. Versiohallinta ei ole kielen suunnittelijan ongelma, mutta C++- ja Java-kielisissä ohjelmissa tulee luvattoman usein vastaan tilanteita, joissa kahden kehittäjän työn yhdistämistä samaan tiedostoon tarvitaan.

12.10 Yhteenvetoa luvusta 12

Wolf-kielessä ei vielä ole aivan tarkkaan tiedossa, miten näkökulmat liitetään muun koodin sekaan. Kieltä toteuttaessa piirreohjelmointi toteuttaneen ennen näkökulmien tukemista, koska ne ovat kirjoittajan omasta mielestä tärkeämpi ominaisuus kielelle. Tällä hetkellä varmaa on, että

- näkökulmat eli aspektit muuttavat sitä koskevaa luokkaa,

- ennen näkökulmien sitomista pitää varmistaa, että luokka on määritelty kunnolla toimivaksi ja
- näkökulmat on suunniteltava mahdollisimman uudelleenkäytettäväksi.

Tämä ei välttämättä ole aivan triviaali toimenpide. Piirteitä käyttäessä ne voidaan liittää suoraan olion määritelmään. Näkökulmat taas pitäisi pystyä lisäämään ja poistamaan myös luokan ulkopuolelta – esimerkiksi lokitiedon ja analyysin tekemistä varten, joka poistetaan varsinaisesta tuotantokäyttöön siirretystä ohjelmakoodista. Olisi melko kankeaa muuttaa itse luokan määritelmää tuon takia. Piirteitä kanssa ei tule vastaavaa ongelmaa vastaan, koska ne on aina määriteltävä luokan yhteyteen.

Näkökulmat ovat erittäin voimakas työkalu, joka voi aiheuttaa pahoja ongelmia, jos määritykset ja rajoitukset ovat liian löysiä. Kehittäjä haluaa varmistaa, että näkökulmat ovat järkevästi määritetty ennen julkaisua. Tarkastaminen vie aikaa, joten näkökulmien lopullinen toteutus voi vielä vaihtua.

Tämän luvun tärkeimmät muistisäännöt ovat:

- Näkökulmaohjelmointi siisteyttää koodia ja tekee useamman ohjelmoijan työn helpommaksi – kuten piirreohjelmointikin.
- Sekä piirre- että näkökulmaohjelmointien toteutus on kielen kääntäjän kannalta melko samantapainen. Punonnan toteutuksessa voitaneen käyttää yhteisiä komponentteja.
- Näkökulma ja piirreohjelmointi kuitenkin eroavat monessa suhteesta toisistaan. Tästä syystä toteutuksessa on oltava melko tarkkana.

13 Säiliö- ja geneeriset luokat

Tämä luku käsittelee lyhyesti säiliöluokkia ja niiden käyttämistä Wolf-kielessä. Luvussa on kohtalaisen vähän viitteitä, mutta säiliöluokkien käyttäminen on perustason ohjelmointia tietotekniikan alalla. Mukana on myös melko paljon kirjoittajan omia huomioita, joilla ei välttämättä ole kovin suurta akateemista arvoa.

Luku toimii kuitenkin hyvänä alustuksena säilyvyyteen, joka on hyvin laaja ja vaikea aihe Wolf-kielessä[1].

13.1 Geneeriset luokat

Geneeriset luokat ovat Wolf-kielessä metaluokkia, jotka saavat parametrina tyyppejä ja luovat niiden pohjalta uusia tyyppejä. Niiden avulla voidaan luoda hyvin helposti ja hallitusti uutta toimintaa. Suurin inspiraatio näille on ollut C++-kielen template-järjestelmä, jolla on toteutettu mainiota kirjastoja kuten C++-kielen std ja boost-kirjasto. Boost-kirjasto on nimenä hieman harhaanjohtava, koska kyseessä on oikeastaan nippu kirjastoja, joiden avulla C++-kehittäjän elämää helpotetaan. Lisää tietoa Boost-kirjastosta voi hankkia viitteestä [11].

Kirjoittaja ei kuitenkaan halua synnyttää metaohjelmoinnilla yhtä monimutkaista järjestelmää kuin C++-kielen templatet. Parhaimmat ja pahimmat templatet C++-kielen puolella ovat hyvin, hyvin vaikeita kääntää oikein. Templateiden erikoistaminen on tehtävä jotenkin järkevästi.

Itseasiassa C++-kielessä voi luoda luokkia, joiden käännös ei pitäisi pysähtyä koskaan kuten listauksessa 13.1 näytetään. Rekursiivia templateita kokonaisluvuille voi käyttää näppärästi laskemalla esimerkiksi fibonaccin lukuja kääntäjän puolella, jotta suoritus toimii nopeammin. Uusi standardi C++-kielestä korjasi tämän katkaisemalla rekursien syvyyttä, mutta ongelmaksi tuleekin määrittää, mikä on sopiva syvyys rekursiolle?

Wolf-kielessä metaluokat voivat käyttää vain luokkia, joten kokonaislukujen kanssa tehtävät rekursiot eivät onnistu. Lisäksi jokainen metaluokan instanssi luo uuden tyypin eikä tyyppi voi periä itsestään tai metaluokasta, joten ongelmaa on ratkais-

[1] Onneksi Pro gradu-tasossa osan vaikeudesta voi vielä ohittaa :)

tu määritelmien puolelta. Lisäksi myöhemmissä versioissa Wolf-kieleen on tarkoitus lisätä tuki ikuisen rekursion automaattiselle huomaamiselle myös normaaleissa kontrollirakenteissa.

Listaus 13.1: Päättymätön, päätön rekursiivinen template

```
1  template <int N>
2  class ShiverIn {
3  public:
4     static int EternalDarkness();
5
6  };
7
8  template <int N>
9  int ShiverIn<N>::EternalDarkness() {
10   return N*ShiverIn<N-1>::EternalDarkness();
11 }
12
13 /*
14 //uncomment this one and you have factorial
15 template <>
16 int ShiverIn<0>::EternalDarkness() {
17    return 1;
18 }
19 */
20
21 /* this wont' compile */
22 int main() { ShiverIn<5>::EternalDarkness(); return 0; }
```

13.2 Mitä säiliöluokat ovat?

Säiliöluokilla tarkoitetaan tässä yhteydessä yhteistä kantaluokkaa oliolle, jotka säilövät muita olioita ajonaikana. Olioiden varastointi ajon aikana on varmasti eräs yleisimpiä tehtäviä ohjelmoinnissa.

Useimmissa kielissä säiliöoliot ovat rakennettu kielen omilla rakenteilla. Esimerkiksi Java-kieli luo vektorit, listat yms rakenteet kielen sisällä. Wolf-kielelle riittää alustavasti, jos säiliöluokat tarjotaan natiivina rajapintana kuten Java-kielen JNI-luokat.

13.3 Erot C++- ja Java-kielen geneerisiin luokkiin

Suurin ero C++-kielen säiliöluokkiin on, että Wolf-kielessä säiliöluokilla on yhteinen kantaluokka Container. C++-kielessä yhteisestä kantaluokasta luovuttiin, koska se olisi kasvattanut luokan kokoa tarpeettomasti. Lisäksi C++-kielessä voi käyttää typedef-avainsanaa määrittämään käytetyn kokoelman tyyppi, joten yhteisen yliluokan puuttuminen ei ole niin suuri ongelma.

Java-kielen versiossa 1.5 oli mukana vihdoin yleiset, tyypitetyt säiliöluokat, joita kutsutaan kutsutaan nimellä Generics Java-kielessä. Säiliöluokkien käyttö on perusluonteeltaan melko samanlaista kuin Wolf-kielessä. Oleellinen ero on, että Wolf-kielessä säiliöluokat määritellään metaluokkina. Javan geneeristen tapauksessa luokilla vector<int> ja vector<double> on sama tyyppi. Täten luokan on mahdotonta toteuttaa jotain rajapintaa sekä int että double tyypeillä instantoituna. Wolf-kielessä säiliöluokasta erikoistetaan kokonaan uusi tyyppi, kun säiliöluokalle annetaan parametri. Siten luokan on mahdollista toteuttaa rajapinta jokin sekä int- että double-tyypeillä instantoituna, koska rajapinnat ovat erillisiä. Ero voi tuntua pilkunviilaamiselta, mutta Wolf-kieleen on tarkoitus luoda tyyppilistat, joiden avulla generoida rajapintoja.

Tyyppilistat oikein käytettynä säästävät huomattavia määriä työtä toteuttaessa Visitor-suunnittelumallia. C++-kielessä tyyppilistojen käsittely onnistuu myös ja tutkielman liitteenä on Wolf-kielen attribute-kirjasto, jolla kehittäjät voivat liittää omia komponenttejaan Wolf-kieleen kohtalaisen vaivattomasti. Valitettavasti väärin käytettyinä niillä saa tehtyä hyvin huonosti ylläpidettävää lähdekoodia, joten kirjoittajan on mietittävä, miten generoinnit tehdään. Tyyppilistojen miettiminen ei kuitenkaan onneksi ole kiireellisin asia kielessä.

13.4 Piirreohjelmoinnin käyttäminen

Säiliöluokat käyttävät Wolf-kielessä piirreohjelmointia mahdollisimman paljon. Lähteessä [44] piirreohjelmointia testattiin juuri säiliöluokkiin hyvillä tuloksilla. Säiliöluokissa on useita toimintoja, jotka voidaan määrittää järkevämmin piirreohjelmoinnin avulla.

Esimerkkinä hyvästä piirresuunnittelusta voisi olla esimerkiksi, että säiliöluokka määrittää metodin size, joka palauttaa säiliössä olevien alkioiden lukumäärän. Piirteitä voidaan käyttää esimerkiksi luodessa isEmpty-metodia, joka tarkistaa, onko

size-metodin paluuarvo nolla. Metodin isEmpty-kutsuminen on yleisesti suositeltavampaa kuin koon vertaaminen nollaan useissa paikoissa ohjelmakoodia, koska isEmpty on helpompi lukea ja ymmärtää kuin vertailu nollaan[2]

Valitettava tosiasia on kuitenkin, että ensimmäisiin versioihin kielestä ei kaikkea koodia voi siistiä tarpeeksi hyvin. Lisäksi toimiva esimerkki on Smalltalk-kielen variantista Squek-kielestä, joka on dynaamisesti tyypitetty kieli. Raportin [44] tuloksia ei voi suoraan hyödyntää Wolf-kielessä ennen tarkempaa selvitystä, koska Wolf on staattisesti tyypitetty kieli.

13.5 Säiliöluokan läpikäynti

Jotta säiliön sisältöä voi muuttaa, on tarjottava jokin mekanismi, jolla tietoa voi muuttaa. Yleisin, standardi tapa on käyttää iteraattoreita, joilla kehittäjä voi turvallisesti käsitellä säiliöolion sisältöä.

Iteraattoreita on kahta tyyppiä:

ulkoiset Ulkoisia iteraattoreita käytetään muun muassa Javassa ja C++-kielessä. Säiliöoliolta pyydetään iteraattori, joka pitää sisällään referenssin säilöttyyn olioon ja joka sisältää mekanismin siirtyä toisiin alkioihin säiliöoliossa. Toisin sanoen, iteraattori on säiliön ulkopuolella, mistä johtuu nimitys ulkoinen iteraattori.

sisäiset Säiliöolio osaa käydä alkionsa läpi itse ja tarvitsee vain funktio-olion, jota kutsua jokaiselle alkiolle. Sisäisesti säiliö käyttää jonkinlaista iteraattoria käydessään säilömänsä oliot läpi. Iteraattori on siis säiliön sisässä, mistä johtuu nimityksen sisäinen iteraattori.

Listaus 13.2: Iteraattorit Java-kielessä

```
1 List<MyObject> myList;
2 Iterator<MyObject> myIterator = myList.iterator();
3 while (myIterator.hasNext()) {
4   MyObject tmp = myIterator.next();
5   /* do something with tmp */
6   tmp.print();
7 }
```

[2]Ehkä hieman kaukaa haettu esimerkki?

Ruby-kieli käyttää sisäisiä iteraattoreita ja edellisen esimerkin toiminta saadaan huomattavan paljon pienemmällä vaivalla.

Listaus 13.3: Iteraattorit Ruby-kielessä

```
1 myList.each {
2   |tmp|
3   tmp.print()
4 }
```

Esimerkissä 13.2 on esimerkki ulkoisten iteraattorien käytöstä. Koska esimerkiksi Java- ja C++-kielissä ohjelmalohkot ja funktiot eivät ole natiiveja olioita, on käytännön syistä ollut pakko valita ulkoiset iteraattorit säiliöiden läpikäymiseen. Siksi oliolta on pyydettävä iteraattori, jonka avulla säiliötä voi käydä läpi. Koska iteraattori on pyydettävä luokan ulkopuolella, on yleensä luokan itsensä toiminta melko rajoittunutta – muttei rajattua. Jos esimerkiksi halutaan käsitellä vain alkioita, joille pätee jokin tietty ehto, on listaa iteroitava, verrattava ehtoa ja suoritettava toiminto, jos ehto täyttyy. Tämä johtaa usein hyvin samantapaiseen koodiin useissa kohtaa koodia. Tätä ongelmaa ei ole, jos säiliöluokat käyttävät sisäisiä iteraattoreita, koska sekä ehto että kutsuttava toiminto voidaan antaa parametrina säiliöoliolle, joka tekee rutiinikoodin käyttäjän puolesta – usein optimoidummalla ratkaisulla, mitä kehittäjillä on aikaa ongelman ratkaisuun käyttää. Kirjoittajan omien kokemuksien mukaan miltei joka kerta, kun ohjelmaa on alettu muokkaamaan ylläpidettävämmäksi, olisi sisäisillä iteraattoreilla parannukset voinut tehdä nopeammin ja ehkä optimaallisemmin.

Puhtaissa oliokielissä – kuten Ruby, Smalltalk ja Wolf – on kuitenkin mahdollista antaa funktio-olio parametrina säiliöoliolle ja käyttää sisäisiä iteraattoreita. Wolf-kieli käyttää sisäisiä iteraattoreita muun muassa seuraavista syistä:

- Säikeistetyissä ympäristöissä lukitusten ja synkronointien tekeminen on hieman helpompaa, koska olion voi lukita läpikäynnin ajaksi. Ulkoisia iteraattoreita käyttäessä on koko säieolio lukittava joka kerta erikseen, kun iteraattorilta haetaan alkion arvo,

- Wolf-kielen säiliöt on kirjoitettu C++-kielellä; ulkoisen iteraattorin käyttäminen vaatisi enemmän sisällön vaihtamista C++-osien ja Wolf-kielisten osien välillä. Optimointi *Wolf-kielen* tapauksessa on helpompaa sisäisillä iteraattoreilla.

- sisäiset iteraattorit käyttävät vain funktio-oliota olion läpikäyntiin. Säiliöluokan toiminta on helpompi saada vakaaksi nopeammin, koska toimintaa on vähemmän ja
- sisäiset iteraattorit ovat kirjoittajan itsensä mielipiteen mukaan oliomaisempia, koska niiden avulla säiliöluokan toteutus on helpompi piilottaa säiliöoliota käyttäviltä olioilta.

13.6 Yhteenvetoa luvusta 13

Oleellisimmat asiat Wolf-kielen säiliöluokista ovat:

- Wolf-kielessä suurin osa säiliöluokista tulee C++-komponenttien puolelta,
- Säiliöluokat käyttävät mahdollisimman paljon piirreohjelmointia ja
- Säiliöluokat käyttävät pääsääntöisesti sisäisiä iteraattoreita.

Osa III

Wolf-kielen kielioppi ja luokkien rakenne

14 Wolf-kielen peruskielioppi

Tämä luku käsittelee Wolf-kielen kielioppia. Vaikka nykyisillä integroiduilla kehitysympäristöillä kieliopin merkitys on vähentynyt koko ajan, on kielioppi edelleen merkittävä osa kielen suunnittelua.

14.1 Kieliopin taustaa ja pyrkimyksiä

Kaksi eniten Wolf-kieleen vaikuttanutta kieltä ovat Smalltalk ja Python. Molemmat ovat miellyttäviä korkean tason kieliä, joissa kehittäjä voi keskittyä itse ongelmaan teknisten yksityiskohtien sijasta. Molemmat kielet ovat myös antaneet osansa Wolf-kielen syntaksiin. Kirjoittaja on käyttänyt pitkään C++- ja Java-kieliä ja osa niidenkin kieliopista on tarttunut Wolf-kielen kielioppiin – merkittävimpänä aaltosulkujen käyttö ohjelmalohkojen merkitsemisessä.

Jäsentimen tekeminen ei enää ole vaikein osa kielen kääntäjän toteutusta Lex- ja Yacc-työkalujen ansiosta [1, sivu 140]. Kielioppi pitää vain määrittää Lex- ja Yacc-työkalujen omalla kuvauskielellä ja ne generoivat jäsentäjän vaatimat lähdekoodit automaattisesti. Työkalujen avulla on jo mahdollista tehdä kohtalaisen verbaali kieli, joka muistuttaisi yksinkertaistettua versiota luonnollisesta kielestä.

Kielen ei kuitenkaan kannata olla luonnollisen kielen tyylinen, koska ohjelmointikielen on oltava ilmaisuvoimainen. Luonnollinen kieli on liian väljää ja useimmat ihmiset puhuvat jopa *mielellään*. Ohjelmia tehdessä kommunikaatio koneen ja ihmisen välissä on pidettävä minimaallisena. Mitä laajempi ohjelma – eli mitä pidempään kehittäjä jaarittelee – sitä varmemmin ohjelmassa on virheitä. Normaali puheessa toistuu jatkuvasti fraaseja ja sanontoja; toistaminen ohjelmoinnissa on yleensä pahasta. Avainsanojen lisääntyminen kielioppiin on toinen ongelma. Näiden seikkojen nojalla, Wolf-kieli ei siis pyri olemaan lähellä ihmiskieltä.

Kieliopin on oltava yksinkertainen, jotta ohjelmaa on helppo analysoida. Bjarne Stroustrup loi C++:n kieleksi, jota voisi käyttää miltei minkä tahansa ongelman ratkaisuun. Valitettavasti C++-kieli on sekä semantiikaltaan että kieliopiltaan hyvin laaja. C++-kieltä pidetään hyvänä esimerkkinä vaikeasti jäsennettävästä kielestä ja pohjalla käytetty C-kielen syntaksi on venytetty äärirajoille [43].

Jos kieltä on vaikea jäsentää, sille tehdään yleensä vähemmän jäsentimiä ja analysaattoreita. Jos kielelle ei ole analysaattoreita, lähdekoodin laadun mittaaminen mekaanisesti on mahdotonta. Koodikatselmoinneissa joudutaan käyttämään aivan liian paljon vaivaa kielen yleisen tyylin ja kieliopin tarkasteluun, joka on mekaanista työtä ja jonka ohjelma voisi tehdä ihmisen puolesta. Wolf-kielen kielioppi on tarkoitettu yksinkertaiseksi, jotta tyylin ja yleisen laadun voi tarkistaa mekaanisesti ja jotta lähdekoodin tarkastaja voi käyttää aikansa luovaan työhön eli toiminnan tutkimiseen. Edelleen, mitä vaikeampi kieli on jäsentää, sitä vaikeampi kieltä on liittää kehitysympäristöihin. Ilman kunnollista integroitua kehitysympäristöä ohjelmiston kehittäminen ja muokkaaminen on hitaampaa, mikä vähentää tuottavuutta ja turhauttaa kehittäjiä. Kieltä on siis pystyttävä jäsentämään suoraviivaisesti.

Yleisesti Wolf-kieli pyrkii hahmottamaan jäsennystä ylhäältä alaspäin avainsanojen avulla, koska kirjoittaja piti Pascal-kielen tavasta määrittää luokat. Lisäksi kielioppiin on helpompi tehdä uusia sääntöjä myöhemmin. Tarkoituksena ei kuitenkaan ole tehdä top-down kieltä vaan etsiä sopiva kompromissi bottom-up ja top-down jäsennyksen välillä. Top-Down lähestymistapa ei ole kovin käytännöllinen ja se on ilmaisuvoimaltaan heikompi kuin bottom-up-kieliopit. Lisäksi, Yacc-työkalu, jolla Wolf-kielen ensimmäiset jäsentimet tehdään, käyttää bottom-up-jäsennystä ja siten myös Wolf-kieli käyttää bottom-up-jäsennystä.

Ensimmäinen versio Wolf-kielestä ei ole niin oliomainen kuin kirjoittaja haluaisi sen olevan. Lisäksi kieliopissa on aivan liikaa avainsanoja, joiden muistaminen on aina vaikeaa. Aikataulu on kuitenkin kireä, joten kieliopin parantaminen jää myöhemmäksi tehtäväksi. On joka tapauksessa hyvin todennäköistä, että kielen joutuu kirjoittamaan kokonaan uusiksi viisi tai kuusi kertaa, joten kieliopin kauneusvirheet eivät ole suurin ongelma kääntäjää suunnitellessa. Sen on vain toimittava riittävän hyvin, jotta kieltä voi testata.

Tässä luvussa käytetään kieliopin muotoiluun Backus-Naur-Form määritystä, josta käytetään myöhemmin termiä BNF-määrittely. Symbolia ϵ käytetään kuvaamaan tyhjää. Lisäksi useimmat produktiot ovat vasentekijöity, vaikka kieliopin kannalta se ei olisi aivan välttämätöntä.

14.2 Tyhjät rivit, välit ja kommentit

Kommentit, välimerkit ja rivinvaihdot poistetaan jo esikäännöksen aikana. Koska ne eivät päädy jäsentimelle asti, niitä ei ole listattu myöskään kieliopin määritelmissä,

koska se vain monimutkaistaisi listauksia tarpeettomasti.

Kommentit merkitään samalla tavalla kuin Java- ja C++-kielessä. Eräs syy tähän on, että Doxygen-työkalua ei tarvitse muokata kovin paljoa, jotta se osaa luoda API-dokumentaation myös Wolf-kielestä [50].

14.3 Muuttujien ja vakioiden määrittely

Wolf-kielessä muuttujia voi määrittää vain lohkojen alussa kuten C-kielessä. C++- ja Java-kielissä muuttujia voi määrittää vapaammin, mutta Wolf-kieli rajoitti tätä sääntöä. Koodin katselmointi ja tarkastaminen on helpompaa, jos muuttujat on aina määritelty lohkon alussa. Hyvin tehdyissä luokissa ei ole tuhansien rivien metodeja ja lyhyissä metodeissa muuttujat määritellään yleensä aina alkuun, joten rajoitus ei ole kovin merkittävä. Jos metodissa on tuhansia rivejä, Wolf-kielen kääntäjä alkaa antamaan varoituksia ylipitkistä metodeista.

Muuttujia määrittäessä pitää tietää sen tyyppi, jotta tyyppitarkastukset voi tehdä järkevästi. Kirjoittaja ei kuitenkaan pidä Java- ja C++-kielen tavasta määrittää tyyppi explisiittisesti; Myöhemmin kielen on tarkoitus tukea tyyppipäättelyä mahdollisimman joustavasti.

Muuttujat määritellään ja alustetaan lausekkeella. Olion muuttujat voi alustaa konstruktorin ulkopuolella. Idea on lähtöisin Java-kielestä, jossa muuttuja voi määrittää ja alustaa yhdellä kertaa. Alustuksessa käytetyn lausekkeen tyypin perusteella päätellään myös muuttujan tyyppi. Listauksessa 14.1 määritellään muutamia muuttujia, joiden tyyppi on kommentoituna.

Listaus 14.1: Esimerkki muuttujien määrittelystä

```
1 // Int-type
2 var myVar := 2+3;
3 //String
4 var kissa := "istuu puussa";
5 //Int
6 var lengthOfKissa := self kissa length;
```

Wolf-kielessä vakiot määritellään kuten muuttujatkin, mutta ne lisätään näkymään frozen. Frozen näkymässä olevia muuttujia tai metodeja ei voi muuttaa kuten luvussa 12 todettiin. Lauseke, joka alustaa vakion, ei saa käyttää olion muuttujia apunaan, koska se rikkoisi vakion määritelmää.

Jokaisessa luokassa on lisäksi aina määritelty erityismuuttujat self ja super, joista

self viittaa olioon itseensä ja super sen yläolioon[1]. Näiden muuttujien arvoa ei voi muuttaa sijoituslauseella.

Taulukossa 14.1 on annettu BNF-määritys muuttujien määrittelylle. Välike *identifier* määrittää muuttujan nimen, eikä se saa olla avainsana.

varDef	::=	**var** *varTail*
varTail	::=	*identifier assignOp expression stmtSep*
assignOp	::=	':='
stmptSep	::=	';'
identifier	::=	[a-z]*identifierTail*
identifierTail	::=	[a-z]*identifierTail*
	|	[A-Z]*identifierTail*
	|	[0-9]*identifierTail*
	|	ϵ

Taulukko 14.1: Muuttujien BNF-määrittely

14.4 Metodit

Metodit ovat ohjelmalohkoja, jotka voivat saada parametreja käsiteltäväkseen. Ohjelmalohkojen alkuun pitää määrittää parametrit, joita lohkossa on. Parametrit määritellään avainsanalla **param** ja määrittämällä muuttujan nimi. Arvoa ei luonnollisestikaan voida antaa, koska sille ei voi antaa mitään järkevää oletusarvoa käännösaikana (Wolf-kielessä ei ole käytössä oletusparametreja).

Lohkon suorituksen voi myös katkaista eri kohdista. Tämä aiheuttanee varmasti ongelmia tyyppipäättelylle: palasipa metodit mistä kohtaa tahansa, on paluutyypin oltava yhteensopiva muiden samasta lohkosta palautettavien arvojen kanssa. Lohkosta palaaminen tapahtuu avainsanalla **return**. Parametrien ja paluulauseen käytöstä on annettu esimerkki listauksessa 14.2.

Listaus 14.2: Esimerkki metodista Wolf-kielessä

```
1 def corruptedSum (Int,Int)->Int  {
2     param a;
3     param b;
4     var lastSum;
```

[1]Java- ja C++-kielessä self-muuttujaa vastaa muuttuja this

```
5     lastSum  := a+b;
6     return lastSum;
7  }
```

BNF-määrittely metodeille on annettu taulukossa 14.2. Kielioppi on tehty niin, että tyyppipäättelyn toteuttamisen jälkeen metodin tyypitys voidaan jättää pois. Helpoiten tämä onnistuu lisäämällä välikkeen *methodTypeDef* erääksi reduktioksi ϵ.

Tässä kohdassa ei myöskään ole vielä käsitelty tarkemmin välikettä *statement*, joka pitää sisällään lausekkeen, joka suoritetaan metodissa. Lausekkeet kuvataan vasta kohdassa 14.10. Välike *TypeIdentifier* määrittää tyypin, jossa kuvataan luokan nimi. Välike *TypeIdentifier* käsitellään tarkemmin kohdassa 14.5.

methodDef	::=	**def** *methodNameAndTypes blockDef*
methodNameAndTypes	::=	*identifier methodTypeDef*
methodTypeDef	::=	*methodParamsDef mapOperator TypeIdentifier*
methodParamsDef	::=	*listBegin TypeList listEnd*
listBegin	::=	'('
listEnd	::=	')'
listSep	::=	','
mapOperator	::=	'->'
TypeList	::=	*TypeIdentifier TypeIter*
	|	ϵ
TypeIter	::=	*listSep TypeIdentifier TypeIter*
	|	ϵ
blockDef	::=	*blockBegin paramDefs varDefs statements blockEnd*
blockBegin	::=	'{'
blockEnd	::=	'}'
paramDefs	::=	*paramDef paramDefs*
	|	ϵ
paramDef	::=	**param** *identifier stmtSep*
varDefs	::=	*varDef varDefs*
	|	ϵ
statements	::=	*statement stmtSep statements*
	|	ϵ

Taulukko 14.2: Metodien BNF-määrittely

14.5 Luokkien määrittely

Luokat määritellään joukkona määrityksiä rajapinnoista ja piirteistä, muuttujia sekä metodeja. Luokan määrityksessä on taas pyritty suoraviivaisuuteen.

Rajapinnat, joita luokka toteuttaa, pitää määritellä omille riveilleen avainsanalla **implements**. Yläluokka, josta luokka peritään, määritellään avainsanalla **extends**.

Piirteet otetaan käyttöön avainsanalla **uses**, jota seuraa määritykset käytölle. Piirteitä käyttäessä voi joskus joutua nimeämään metodeja uudelleen tai piilottamaan niiden näkyvyys luokasta. Nämä toimenpiteet on järkevintä tehdä luokan sisällä, koska kehittäjällä on – tai pitäisi olla – paras tuntemus luokkansa toiminnasta ja siten myös uudelleennimeämisen ja piilottamisen tarpeista. Piirteistä voi piilottaa tai uudelleennimetä metodeja – joskaan metodien piilottaminen ei ole erityisen hyvää oliokäytäntöä edes piirteitä käyttäessä.

Myös näkymät määritellään luokan sisällä, jotta näkökulmien punominen luokkiin on helpompaa. Metodien määritelmä on annettu jo edellä, joten sitä ei käsitellä enää tässä kohtaa.

Listaus 14.3: Esimerkki luokan määrittelystä

```
1
2  class MyClass {
3   extends MySuper;
4   implements MyInterfaceA;
5   implements MyInterfaceB;
6   implements MyInterfaceC;
7
8   uses ColorTrait {
9      rename equals colorEquals;
10  }
11  uses BlahTrait {}
12
13  view notPersistentView := (kissa);
14
15  var myTest := OtherClass null;
16  var kissa := "istuu puussa";
17
18
19  def mySum (Int,Int)->Int {
20     param first;
21     param second;
22     return first+second;
```

```
23    }
24
25    def testKissa()->Void {
26
27    }
28  }
```

Metaluokkien määrittely tapahtuu melko samalla tavalla kuin normaali luokkienkin määrittely. Listauksessa 14.4 annetaan esimerkki geneerisen luokan käytöstä. On huomattava, että Wolf-kielen kieliopissa ei vielä ole määritelty tarkkaan geneeristen luokkien erikoistamista, joka on varmasti hyödyllinen ominaisuus. Suurin ero verrattuna C++-kielen templateihin on, että geneeriset luokat sekä instanssoidaan että määritellään samalla syntaksilla – C++-kielessä joutuu käyttämään erillistä template-avainsanaa.

Listaus 14.4: Esimerkki geneerisen luokan määrittelystä

```
1  class MyTemplate(MyParamType) {
2
3      var myVar := MyParamType null;
4
5      def setVar(MyParamType)->Void {
6          param temp;
7          self myVar := temp;
8      }
9  }
```

Kuten taulukko 14.3 näyttää, Wolf-kielen kielioppi on hyvin rajoittunut luokan yhteydessä. Ainoastaan ohjeistuksen mukainen määritys luokan rakenteelle hyväksytään, mikä on tietoinen päätös. Kirjoittajan omien kokemusten perusteella mahdollisuus rikkoa kaikkia ohjeistuksia luokan rakenteesta tarkoittaa, että ohjeistuksia rikotaan myös käytännössä. Oikean rakenteen oppii kohtalaisen nopeasti, mutta huonosta tavasta on vaikeampi päästä eroon. Päätös ei varmasti ole kaikkien kehittäjien mieleen mutta sopii Wolf-kielen tarkoituksiin.

TypeIdentifier	::=	[A-Z]*identifierTail classParamsDef*
classParamsDef	::=	*methodParamsDef*
	|	ϵ
classDef	::=	**class** *TypeIdentifier classBlock*
classBlock	::=	*blockBegin bodyExtendTail blockEnd*
bodyExtendTail	::=	*extendDef bodyTailImplement*
extendDef	::=	**extends** *TypeIdentifier smtSep*
	|	ϵ
classBodyImplementsTail	::=	*implementsDefs usesDefs viewDefs varDefs methodDefs*
implementDefs	::=	*implementDef implementDefs*
	|	ϵ
implementDef	::=	**implements** *TypeIdentifier stmpSep*
usesDefs	::=	*usesDef usesDefs*
	|	ϵ
usesDef	::=	**uses** *TypeIdentifier traitModifier*
traitModifier	::=	*blockBegin traitModifyStmts blockEnd*
traitModifiers	::=	*traitModifier traitModifiers*
traitModifier	::=	*traitModifyRename*
	|	*traitModifyHide*
traitModifyRename	::=	**rename** *identifier identifier stmtSep*
traitModifyHide	::=	**hide** *identifier stmtSep*
viewDefs	::=	*viewDef viewDefs*
	|	ϵ
viewDef	::=	**view** *viewTail*
viewTail	::=	*identifier varList*
	|	*identifier*
varList	::=	*listBegin varListBody listEnd*
varListBody	::=	*identifier varListTail*
	|	ϵ
varListTail	::=	*listSep identifier varListTail*
methodDefs	::=	*methodDef methodDefs*
	|	ϵ

Taulukko 14.3: Luokkien BNF-määrittely

14.6 Rajapintojen määrittely

Listauksessa 14.5 annetaan esimerkki rajapinnan määrittelystä. Metodit, joita toteuttavan luokan pitää määrittää, annetaan avainsanalla **require**. Rajapinta, josta halutaan laajentaa, määritellään samaan tapaan kuin luokkien yhteydessä avainsanalla **extends**. Rajapinnan määritys on melko yksinkertainen.

Listaus 14.5: Esimerkki rajapinnan määrittelystä

```
1 interface MyInterface {
2    extends MyOtherInterface;
3    require myMethod::(Int,Int)->Int;
4    require otherMet\-hod::(Void)->Void;
5 }
```

BNF-määrittely rajapinnoille on annettu taulukossa 14.4. Suurin osa välikkeistä on peräisin luokan määrityksestä, joten määrityksissä ei ole mitään ihmeellistä.

interfaceDef	::=	**interface** *TypeIdentifier interfaceBlock*
interfaceBlock	::=	*blockBegin interfaceBody blockEnd*
interfaceBody	::=	*extendDef methodRequires*
methodRequires	::=	*methodRequire MethodRequires*
	|	ϵ
methodRequire	::=	**require** *methodNameAndTypes stmpSep*

Taulukko 14.4: Rajapintojen BNF-määrittely

14.7 Piirteiden määrittely

Esimerkki piirteiden määrityksessä on annettu jo kohdassa 11.1. On huomattava, että myös piirteisiin voidaan lisätä tyyppiparametreja. Kielioppi on käytännössä vain yhdistelmä luokan ja rajapinnan kieliopin välikkeistä, kuten taulukosta 14.5 havaitaan.

traitDef	::=	**trait** *TypeIdentifier traitBlock*
traitBlock	::=	*blockBegin traitBody blockEnd*
traitBody	::=	*methodRequires usesDefs methodDefs*

Taulukko 14.5: Piirteiden BNF-määrittely

14.8 Makrot ja niiden puute

Wolf-kielessä ei ole esikääntäjän makroja. Kirjoittaja ei erityisemmin pidä C++-kielen makroista, koska ne vaikeuttavat huomattavasti kääntäjän tekemistä, niiden tuottamat virheilmoitukset ovat harvoin hyödyllisiä ja IDE:t eivät useimmitenkaan osaa käsitellä niitä oikein. Wolf-kieli käyttää metaohjelmointia hyväkseen ja myöhemmissä versioissa kehittäjät voivat luoda omia ympäristöjä, joilla luoda ongelma-alueeseen soveltuva kieli.

Ympäristöjä ei toteuteta vielä ensimmäisessä versiossa, joten ne ohitetaan tässä tutkielmassa. Niiden ideana on kuitenkin tarjota hallittu tapa luoda makrojen tyyppistä toimintaa. Suurin inspiraatio näille on LaTeX-kielen ympäristöt, joihin kirjoittaja on mieltynyt.

14.9 Operaattorien määrittely

Wolf-kielen esikääntäjä on melko suppea tällä hetkellä. Kommenttien ja välimerkkien poistamisen lisäksi sen ainut tehtävä on muuttaa operaattorit funktiokutsuiksi. Tämä on pakollista, koska Wolf-kielessä ei ole erillistä operaattorien kuormitusta kuten C++-kielessä, mutta Wolf-kielessä on mahdollista ylikirjoittaa metodit, joita kutsutaan operaatioiden yhteydessä. Esikäännöksen tehtävänä on muuttaa operaattorit oikeiksi kutsuiksi metodeille.

Esimerkiksi a+b*c kääntyy muotoon a.plus(b.multiply(c)) ja -a muotoon (A null)-a.

Vaikka operaattoreita itsessään ei voi ylikuormittaa kuten C++-kielessä, kehittäjät voivat määrittää metodien toimintaa uudelleen. Tämän takia kirjoittaja ei usko, että ratkaisu on liian rajoittava kehittäjille. On myös huomattava, että esimerkiksi yhteen ja vähennyslaskuun voisi lisätä tarkastuksia näkökulmaohjelmoinnin avulla, jotta optimointi olisi tehokkaampaa. Kääntäjä voi tehdä reippaampia optimointeja, jos se voi käyttää hyväkseen algebraa ja korkean tason matematiikkaa. Huonona puolena on, että kielen vapautta on tässä tapauksessa rajoitettava. Tässä tutkielmassa aihetta ei käsitellä syvällisemmin.

14.10 Lausekkeet

Koska Wolf-kieli on puhdas oliokieli, ei erillisiä rakenteita ehtolauseille tai silmukoiden käsittelylle tarvita. Listauksessa 14.6 esitellään, miten perusrakenteet kuten while,if ja listan läpikäyminen toteutetaan metodeilla Wolf-kielessä. Esitykset ovat hieman kömpelöitä, koska kieliopissa ei vielä ole metodien automaattista tyyppipäättelyä. Kun tyyppipäättely on valmis, ei kankeaa (Bool)->Void määritystä tarvitse määrittää esimerkiksi ehtolauseisiin.

Tärkein seikka on, että esimerkiksi (2<4) on Bool-tyyppinen olio, jonka metodia ifTrue kutsutaan. Metodi saa parametrikseen lohkon, joka suoritetaan, jos ehto on tosi.

Listaus 14.6: Esimerkki kontrollirakenteista

```
1
2  (2 < 4) ifTrue : (Bool)->Void {
3      System out println : "never happen";
4  }
5
6  (i>5) whileTrue : (Bool)->Void {
7    param currentValue;
8    var myTemp;
9    i = i-1;
10   myTemp := i;
11   self callUnary;
12   System out println : "hello world";
13
14 }
15
16
17 myCollection foreach : (Int,MyCollectionType)->Void {
18     param index;
19     param value;
20
21     System out println: "index = "
22                     + index + ",value = "+value;
23 }
```

Lausekkeiden kielioppi on määritelty taulukossa 14.6. Mukana ei ole esimerkiksi kokonais- tai liukulukujen määritystä, koska ne määritellään samalla tavalla kuin muissa kielissä. Tässä kohtaa kielioppi ei enää voi olla top-down, koska määrityksiä ei saa järkevästi tehtyä ilman LR-parsinnan voimakasta ilmaisukykyä.

expression	::=	*identifier*
	|	*double*
	|	*int*
	|	*string*
	|	*array*
	|	*methodCall*
	|	*methodTypeDef blockDef*
array	::=	*listBegin arrayBody listEnd*
arrayBody	::=	*expression arrayBodyTail*
	|	ϵ
arrayBodyTail	::=	*arraySep expression arrayBodyTail*
	|	ϵ
statement	::=	*assingStmt*
	|	*methodCall*
	|	ϵ
assignStmt	::=	*identifier assignOp expression*
methodCall	::=	*unaryCall*
	|	*methodCallWithParam*
unaryCall	::=	*expression identifier*
methodCallWithParam	::=	*expression identifier msgSep expressions*
expressions	::=	*expression expressions*
msgSep	::=	':'

Taulukko 14.6: Lausekkeiden BNF-määrittely

14.11 Yhteenvetoa luvusta 14

Luku oli tarpeettoman pitkä verrattuna kieliopin monimutkaisuuteen. Oleellisimmat seikat kieliopista ovat:

- kielioppi on hyvin rajoitettu,
- metaluokkien erikoistamiseen ei vielä ole erillistä syntaksia tai semantiikkaa,
- metodien tyypitys voi myöhemmin siirtyä vapaaehtoiseksi tyyppipäättelyn myötä,
- kielioppi on melko lähellä top-down toteutusta ja
- selkeys oli eräs suunnittelun tavoite.

Kirjoittaja itse on jäävi arvioimaan, onko kielioppi selkeä vai ei. Tämä on kuitenkin lähellää kirjoittajan näkemystä hyvästä syntaksista – lisäksi sen integroiminen IDE-ympäristöihin pitäisi olla kohtalaisen suoraviivaista.

15 Olion ja luokan rakenne Wolf-kielessä

Tämä luku kuvaa olion ja luokan rakenteen Wolf-kielessä toteutuksen kannalta. Korostettakoon, että luku keskittyy kielen ajonaikaisiin rakenteisiin ja muistirakenteisiin. Tarkoituksena ei ole toistaa kaikkia hienouksia luokista ja olioista.

Eri kappaleet kertovat ongelmista, joita kielen kehittäjällä on tullut vastaan olion käsitettä määriteltäessä. Osa ongelmista on melko triviaaleja, osaa ongelmista ei voi ratkaista täydellisesti nykyisellä tietämyksellä ja suurinta osaa ongelmista kirjoittaja ei vielä tunne. Kuten kaikissa suurissa projekteissa, kaikkia ongelmia ei voi ratkaista etukäteen. Vaikka määritelmä on kehittynyt vuosien varrella ja saanut tiettyä jämäkkyyttä, on luokan ja olion lopullisia tietorakenteita varmasti vielä muokattava ennen ensimmäistä toteutusta kielestä.

15.1 Karkea yleiskuvaus

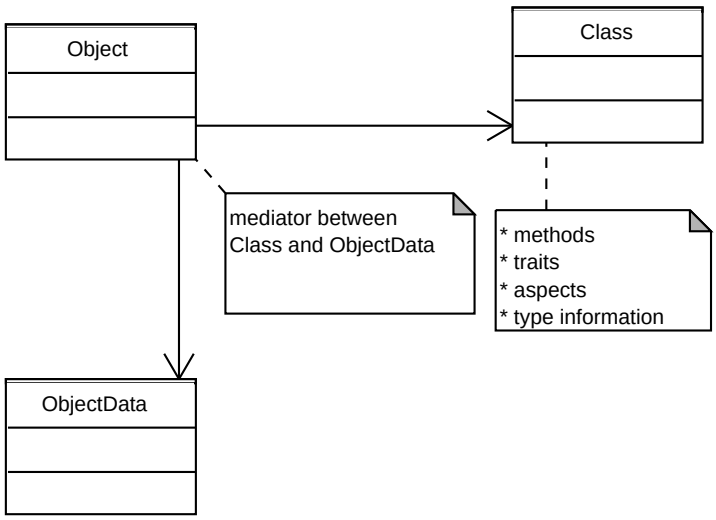
Oliolla on periaatteessa vain metodeja ja muuttujia. Muuttujat ovat jokaiselle oliolle yksilöllisiä, mutta metodit ovat samanlaisia jokaiselle oliolle. Ensimmäinen versio Wolf-kielestä tallensi myös metodit suoraan olioon. Tämä helpotti toteutusta, mutta kulutti muistia tarpeettomasti.

Isossa pelimaailmassa voi olla tuhansia tai kymmeniä tuhansia luokkia mutta miljoonia olioita. Wolf on oliopohjainen ohjelmointikieli, joten jokainen olio on jonkin luokan instanssi. Koska olioita on paljon suhteessa erilaisiin luokkiin, on järkevää sijoittaa mahdollisimman paljon ajonaikaista tietoa luokkaan.

Luokkaan kannattaa sijoittaa ainakin metodit. Luokka määrää niiden ominaisuudet täysin, joten ne voidaan jakaa luokan instanssien kesken. Wolf-kielen pitää pystyä kertomaan muuttujien, metodien ja luokkien nimet debuggerille ja muille halukkaille. Myös nimet kannattaa säilöä luokan sisään, koska ne ovat instansseille yhteiset.

Jokaisella oliolla on viite luokkaansa, jotta olion tyyppi voidaan päätellä nopeasti. Tyyppi pitää pystyä päättelemään nopeasti monimetodien suorittamisessa, reflektiivisyyden tukemisessa ja olioiden lataamisessa ja tallennuksessa. Myös rajapintojen käyttäminen vaatii tietoa tyypistä.

Kuvassa 15.1 on esitetty riippuvuus luokan ja olion välillä. Kaikilla saman luokan instansseilla on viite samaan luokkaan. Tämä säästää muistia ja helpottaa luokan – ja siten myös olioiden toiminnallisuuden – muuttamista jälkikäteen[1].



Kuva 15.1: Olion suhde luokkaan

15.2 Olioiden tunnistus ja yksilöllisyys

Eräs triviaalilta tuntuva ongelma on olioiden tunnistaminen eroon toisistaan. Wolf-kielessä oliolla on oltava yksilöllinen tunniste. C++-kielessä tunnisteena käytetään olion muistiosoitetta. Wolf-kielessä muistiosoitteen käyttäminen ei ole riittävä ratkaisu, koska

- kielen oliot ovat säilyviä ja niiden osoite muuttuu ladatessa ja tallentaessa,
- osa olioista on hajautettu eri koneiden välille ja muistiosoitetta kahden eri koneen välillä ei voi vertailla ja
- osoitteiden käyttäminen ei C++-kielessäkään ole riittävä tapa tunnistaa olioita erilleen moniperintää käyttäessä.

Miten eri olioiden tunnistus on sitten hoidettava? Ainakin Wolf-kielen ensimmäisissä toteutuksissa käytetään erillistä Identifier-tietorakennetta, joka tunnistaa oliot eroon toisistaan.

[1]Kuvassa mainittu ObjectData on vektori muuttujista, joita olio pitää sisällään. Tätä käsitellään myöhemmin tässä luvussa

Jotta tunniste on järkevä, sen on oltava kohtalaisen isokokoinen. Jos esimerkiksi tunnistetta varten varataan vain yksi tavu, voidaan sen avulla tunnistaa vain $2^8 = 256$ olioita toisistaan erilleen. Tämä on selkeästi liian vähän. Toinen ongelma on, että Wolf-kielessä oliot voivat olla säilyviä, joten erillisiä tunnisteita tarvitaan melkoisesti. Tunnistetta voi tietyssä mielessä verrata relaatiotietokannan primary-key kenttään, koska sen avulla tunnistetaan myös levyllä olevat oliot erilleen toisistaan. Tietokantaa on hyvin vaikea toteuttaa järkevästi, jos taulujen alkioiden tunnisteet vaihtuisivat lennosta koko ajan.

Mitä isompi tunniste on, sitä enemmän se vie muistia. Mitä enemmän tunniste vie muistia, sitä vähemmän muistiin mahtuu olioita. Mitä vähemmän muistiin mahtuu olioita, sitä enemmän niitä pitää tallentaa levylle ja tämä hidastaa pelin suorituskykyä. Pelin on toimittava sulavasti, joten ainakin lähiympäristössä olevat oliot ja entiteetit on oltava keskusmuistissa. Tämä johtaa siihen, että tunniste ei voi olla kovin hirvittävän iso.

Olion tunnisteen koko on alustavasti vain 8 tavua. Tämä on kompromissi ja melko satunnaisesti valittu, minkä takia tunnisteen kokoa on melko varmasti vaihdettava peliprojektista toiseen. Wolf-kielen toteutus ei saa riippua liiaksi tunnisteen koosta vaan sen on käytettävä joka vakioita tai esikääntäjän makroja arvon käyttämiseen.

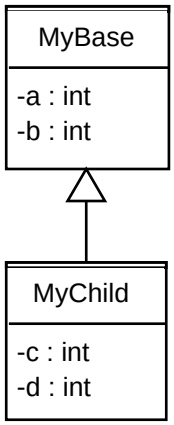
15.3 Ylä- ja alaluokka

Tämä kappale käsittelee ylä- ja alaluokan käsitettä kielen *toteuttajan* näkökulmasta. Tarkoituksena on hahmottaa erilaisia tapoja määrittää ylä- ja alaluokan suhdetta. Wolf-kielen on pakko joustaa puritaanisesta oliomaisuudesta, koska suorituskyvyn on oltava korkea.

Jatkossa käytetään kahta yksinkertaista luokkaa MyBase ja MyChild ylä- ja alaluokan hahmottamiseen. Niiden UML-notaatio on esitetty kuvassa 15.2.

15.3.1 Koostaminen

C++-kieli käyttää koostamista olioiden ylä- ja alaluokkien välillä. Listauksessa 15.1 on listattu tietueet, jotka kääntäjä generoisi MyBase ja MyChild luokista. Korostettakoon, että tämä on kääntäjän tekemää työtä. C++-kieltä käyttävältä kehittäjältä yksityiskohdat on piilotettu.



Kuva 15.2: UML-kaavio luokista MyBase ja MyChild

Listaus 15.1: Yläolio koostamalla

```
1  struct MyBase {
2    int a;
3    int b;
4  };
5
6  struct MyChild {
7    /* from MyBase */
8    int a;
9    int b;
10
11   /* from MyChild */
12   int c;
13   int d;
14 };
```

Tietueen MyChild alkuosa on samanlainen kuin MyBase-tietueessa. Kääntäjän ei tarvitse tällöin välittää ylä- ja alaluokan suhteesta niin paljoa, koska MyChild-oliossa muuttuja b sijaitsee samalla kohtaa kuin MyBase-oliossa. Tällöin metodille ei tarvitse generoida koodia erikseen sekä MyChild- että MyBase-luokille vaan voidaan käyttää samaa metodia molemmille. Tämä pienentää muistissa olevan ohjelman kokoa.

15.3.2 Viite yläluokkaan

Toinen tapa määritellä yläluokka alaluokkaan on käyttää referenssiä. Tämä on käytössä jossain puritaanisissa oliokielissä, joissa muuttujat ovat vain viitteitä varsinai-

siin olioihin, ja kielissä, jotka käyttävät moniperintää. Ainakin Smalltalk-kieli käyttää tätä tapaa ylä- ja alaolion määrittelyyn.

Listaus 15.2: Yläolio viitteenä

```
1  struct MyBase {
2    int a;
3    int b;
4  };
5
6  struct MyChild {
7    /* reference to MyBase */
8    MyBase *super;
9
10   /* from MyChild */
11   int c;
12   int d;
13 };
```

Tämä menetelmä on hieman joustavampi kuin koostaminen, koska yläluokkaa käsitellään vain ja ainoastaan oliona ja yläluokkana. Yläluokan määritystä voi muuttaa ilman, että se vaikuttaa niin radikaalisti lapsiluokkiin. Huonona puolena on hitaampi suorituskyky, koska yläluokan instassia joutuu kutsumaan, kuten muitakin olioita.

15.3.3 Yläluokan käsittely Wolf-kielessä

Wolf-kieli käyttää koostamista, koska

- suorituskyky on nopeampi,
- määritysten tekeminen on helpompaa ja
- C++-kielen liittäminen Wolf-kieleen on helpompaa.

Wolf-kieli voi käyttää koostamista turvallisesti, koska se tukee vain yksiperintää. Tämä helpottaa toteutusta oleellisti, koska alaluokka voi lisätä omat muuttujansa yläluokan muuttujien jälkeen. Tämä helpottaa määritelmien tekemistä, koska ylä- ja alaluokkaa voi molempia käsitellä vain joukkoina metodeja ja muuttujia.

Myöskään näkökulma- tai piirreohjelmointi ei aiheuta mitään sellaisia ongelmia yläluokan käsitteeseen, josta ei selviäisi yksiperintää käyttämällä. Ainut ongelmallinen osa-alue on C++-rajapintojen liittäminen kieleen. Tämä voi aiheuttaa isojakin ongelmia kielen toteutuksessa.

15.4 Olion ja luokan rakenne

Itse olio on melko yksinkertainen tietorakenne Wolf-kielessä, koska suurin osa monimutkaisista osista on siirretty luokan vastuulle. Luokkapohjaisuuden reippaalla käyttämisellä on helpompi käsitellä metaohjelmointia, koska luokan tilan muuttaminen on helpompaa kuin kaikkien luokan instanssien muuttaminen.

Esimerkkinä luokan muuttamisesta ajon aikana on debuggerien käyttäminen. Wolf-kielessä ei ole mahdollisuutta kääntää lähdekoodia debug-lipuilla kuten C++-kielessä, koska kirjoittajan omien kokemusten perusteella debuggaaminen on vain epäonnistumista yksikkötesteissä. Huolellinen yksikkötestaus vähentää debuggauksen määrää hyvin radikaalisti. Toisekseen, Wolf-kieli on tarkoitettu mahdollisimman yksiselitteiseksi kieleksi ja debug-lippujen kanssa tuotettu käännös eroaa optimoidusta liian paljon. Wolf-kielessä käännös erikseen debug-tietojen saamiseksi ei ole välttämätöntä, koska olion attribuutteja muutetaan luokan kautta. Siten on järkevämpää vain lisätä tarkkailija luokan muuttujaa käsittelevään olioon kuin tehdä käännös erikseen debug-tietoja varten. Tällä tavoin debug-tietoja voi muokata ohjelman suorituksen aikana.

Debug-tiedon lisääminen on vain eräs näkökulma ohjelmaan. Yleisemmin, Wolf-luokat tarvitsevat tietoa näkökulmista, joita siihen liittyy ja piirteistä, joita se käyttää. Lisäksi piirteitä varten tarvitaan lisää tietoa. Nämä kaikki esimerkiksi C++-kieleen nähden uudet ominaisuudet on sijoitettava luokkiin.

Jotta suorituskyky olisi optimaalinen on näkökulma- ja piirreohjelmoinnin tuomat lisäkilkkeet käännettävä nopeammaksi, suoraksi tavukoodiksi. Ajonaikana niitä ei siis enää välttämättä säilyttää muistissa toiminnallisuuden takia. Piirteitä ja näkökulmia ei voi kuitenkaan hävittää luokasta, vaikka tavukoodista se olisi optimoitu pois: Jos näin tehtäisiin, olisi mahdotonta sanoa olisiko metodi alunperin lähtöisin piirteestä vai luokan itsensä määrittämänä funktiona. Tämä puolestaan estäisi tehokkaasti reflektiivisyyden käytön ja johtaisi samantapaisiin ongelmiin kuin C++-kielessä, jossa ei ole tukea ajonaikaiselle tiedolle olioista kovin paljoa.

Vaikka Wolf-kieli käyttää koostamista olioiden rakenteessa, käyttää kieli viitteitä luokkien välillä ylä- ja alaluokan kuvaamiseen. Tämä helpottaa hieman käsittelyä eikä aiheuta kuitenkaan mittavaa ongelmaa suorituskyvylle.

Rajapinta on kielen toteuttajan näkökulmasta vain joukko metodeja, jotka luokan on toteutettava. Koska Wolf-kieli säilöö metodit vektorina, jotta niiden käyttö on nopeaa, on ainut ongelma löytää sopiva kuvaus rajapinnan metodeista luokan metodeihin. Tämäkään ei ole ongelma, koska voidaan luoda hyppytaulu olio meto-

dien ja rajapinnan metodien välillä.

15.5 Olion rakenne Wolf-kielessä

Primitiivit ovat olioita, joita ei voi koostaa muista olioista. Tämä kappale ei keskity niihin vaan niille on omistettu oma lukunsa 16. Tämä luku keskittyy pelkästään olioihin, jotka koostuvat primitiiveistä tai muista olioista. Aiemmat hahmotelmat eivät ottaneet eroja huomioon ja epäonnistuivat komeasti liialliseen abstraktioon.

Wolf-oliossa on edellisten kappaleiden tuloksena vain seuraavat tiedot:

- Viittaus tunnistetietoon,
- Viittaus luokkaan, jonka instanssi luokka on,
- Luokan omat, dynaamisesti ladattavat tiedot.

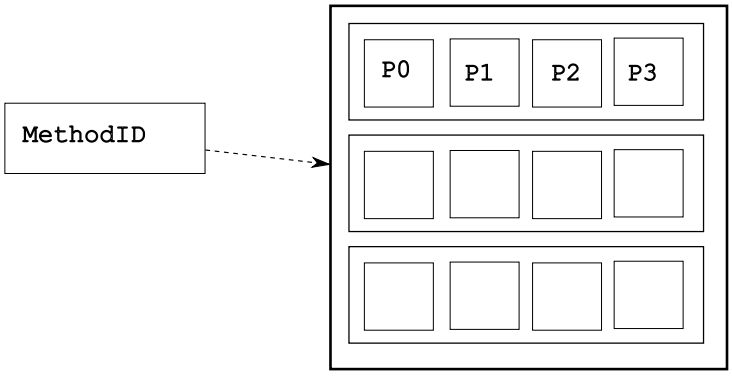
Ongelmaksi tulee – jälleen kerran – saumaton liittäminen C++-kieleen. Tämän ongelman voi ratkaista käyttämällä template-luokkia, jotka itse toteutus sitten perii. Koska C++-kieli sallii moniperinnän mutta Wolf-kieli ei, joutuu olion rakenteeseen tekemään melko isoja kompromisseja. Esimerkiksi niissä C++-luokissa, joita käytetään Wolf-kielenkin puolella, ei voi käyttää moniperintää. Tämä on isohko rajoitus, mutta kirjoittaja itse on selvinnyt vuosia ilman moniperintää.

15.6 Miten monimetodit toteutetaan Wolf-kielessä?

Wolf-kieli toteuttaa monimetodit toteuttamalla useampikertaisen virtuaalitaulun. Pohjalla on edelleen vanha tuttu virtuaalitaulu C++- ja Java-kielistä. Virtuaalitaulussa ei kuitenkaan ole osoittimia suoraan metodeihin vaan $(P+1) * N$-matriisiin, jotka on tarkoitettu metodille, missä P on metodissa olevien parametrien lukumäärä ja N metodin kuormitusten lukumäärä. Kuvassa 15.3 esitetään perusidea matriisille.

Matriisissa on riveittäin metodin tyypit ja metodin osoite[2]. Matriisia käydään läpi rivi riviltä ja etsitään metodia, joka ensimmäisenä sopii parametreihin. Lapsiluokissa virtuaalitaulua voidaan laajentaa, kun metodia kuormitetaan. Kutsun kompleksisuus on pahimmassa tapauksessa O(N)*O(P), missä N on erilaisten metodien lukumäärä ja P luokassa olevien parametrien lukumäärä, koska kaikki eri vaihtoehdot voi joutua huonoimmillaan käymään läpi.

[2]Täten matriisin rivin pituus on P+1.



Kuva 15.3: Metodeja varten tarvitaan matriisi

On huomattava, että Wolf-kieli tarkistaa ajon aikana mahdollisimman pitkälle, missä järjestyksessä metodit kannattaa vertailla keskenään. Nykyiset C++- ja Java-kääntäjät pyrkivät poistamaan virtuaaliset kutsut ohjelman tilan perusteella mahdollisimman pitkälle. Wolf-kieli pyrkii samaan, mutta ensimmäiset versiot kääntäjästä ovat varmasti melko tehottomia.

On korostettava, että monimetodien lopullinen määritys on vielä kesken. Alustava toteutus on mukana, jotta toteutusta voidaan testata ja jotta tekniikan voi todeta toimivaksi.

15.7 Yhteenvetoa luvusta 15

Oleellisimmat asiat luvusta ovat

- Wolf-kielen olio on melko suppea,
- Wolf-kielen luokka sisältää mahdollisimman paljon ajon aikana tarvittavaa tietoa,
- Rajapinnan toteutus luokassa on vain indeksien muuttamista toisiin,
- C++-rajapinnat aiheuttavat varmasti ongelmia toteutuksessa ja
- muistiosoitteiden käyttäminen ei riitä olioiden tunnistamiseen.

16 Primitiivit ja optimointi matalalla tasolla

Tämä luku käsittelee primitiivejä. Ne ovat olioita, joita ei voi luoda muista olioista tai muista primitiiveistä. Tälläisiä ovat esimerkiksi merkkijonot, kokonaisluvut ja liukuluvut. Jos oliokieltä ajattelee matematiikkana, niitä voi pitää alkeisfunktioina. Niiden käyttö poikkeaa jonkin verran normaalista oliokoodista *toteutukseltaan* – kehittäjille primitiivit näkyvät tavallisina olioina.

16.1 Kokonaisluvut

Tutkielmassa käytetään esimerkkinä vain kokonaislukuja, koska muut primitiivit käyttäytyvät melko samalla tavalla.

Python-kieli varaa kaikki oliot keosta. Tämä on hidasta eikä Wolf-kielellä ole varaa tuhlata muistia – ja aikaa – kokonaislukujen varaamiseen keosta. Ratkaisuna on, että kokonaisluku tallennetaan tilaan, jonka veisi normaalisti viite toiseen olioon. Ongelmallinen tämä on sen takia, että semantiikka vaihtuu kokonaislukujen ja normaalien olioiden välillä.

Jotta toiminta saataisiin oliomaiseksi, on kokonaisluvut määriteltävä siten, että Wolf-kielen kokonaisluvut näkyvät kehittäjälle *arvoina*, joita ei voi muuttaa. Kehittäjä voi ainoastaan alustaa kokonaislukumuuttujansa uudelleen, mutta ei muuttaa kokonaisluvun arvoa. Esimerkiksi peruslaskutoimitukset palauttavat aina uuden kokonaislukuarvon. Inspiraatio määritelmälle on Java-kielen String-luokka, joka on olio, mutta sitä ei voi muuttaa metodeilla. Sen semantiikka noudattaa muiden olioiden semantiikkaa tarkasti mutta sitä voidaan käsitellä erityisillä tavoilla virtuaalikoneen puolella.

Eräs syy kokonaislukujen muuttumattomuuteen on, että kokonaislukuja käytetään melko paljon ja niitä myös muutellaan kohtalaisen paljon [8]. Wolf-kielessä ei ole kuin referenssit eikä osoitteita käytössä. Tämä on iso ero verrattuna C++-kieleen, jossa voi määrätä, annetaanko parametrina viite vai arvo muuttujaan. Kirjoittaja ei halua Java-kielen tyylistä erottelua normaaleiden olioiden ja kokonaislukujen välille.

Toinen merkittävä syy kokonaislukujen rajoituksiin on, että Wolf-kielen oliota

pitää pystyä lataamaan kovalevyltä. Tarkempi kuvaus itse tallennusmenetelmästä annetaan luvussa 20, mutta olio ei tallenna oliosta kuin olion tunnisteen ja lataa sen perusteella oikean datan levyltä. Jos kokonaislukujen arvoja voisi muuttaa, pitäisi jokainen primitiivi hakea levyltä erikseen, koska muuten ohjelman logiikka ei toimisi samalla tavalla eri latauskertojen välillä, mikä ei ole haluttu toiminnallisuus. Lisäksi suorituskyky olisi melko huono, koska levyoperaatioita joutuisi tekemään huomattavan paljon enemmän ja logiikka kokonaislukujen käsittelyyn optimaalisesti olisi paljon monimutkaisempi.

Kokonaisluvuista ei myöskään voi periä eikä niiden määrittelyä voi muuttaa. Tarkasti määritellyille kokonaislukuluokille voidaan määrittää matemaattisesti optimointeja, joita kääntäjä voi käyttää reippaammin kuin muuten olisi mahdollista. Luokan määritelmä on lyötävä lukkoon, koska muutokset luokkaan tai sen periminen voisivat rikkoa algebraan perustuvat optimoinnit.

Muut primitiivit toimivat samoilla ideoilla kuin kokonaisluvut. Samat ideat toimivat myös totuusarvoille, liukuluvuille ja enum-muuttujille. Merkkijonot toteutetaan ainakin alustavasti normaaleina olioina, joiden toteutus vain käyttää hyväkseen C++-kielistä toteutusta.

16.2 Yhteenvetoa luvusta 16

Tärkeimmät asiat luvusta ovat:

- primitiiveillä voi optimoida kääntäjän toimintaa,
- primitiivit näkyvät kehittäjälle kuten normaalit oliot, mutta niiden määritelmä on estää muuttamisen,
- primitiivejä ei voi periä ja
- primitiivien arvoja ei voi muuttaa.

Osa IV

Attribute-kirjaston toteutus

17 Attribute-kirjaston keskeisimmät periaatteet

Tämä osa on käytännön harjoitustyö gradun osana. Se keskittyy kuvaamaan attribute-kirjaston käyttöä ja toteutusta.

Tämä luku sisältää keskeisimmät perusperiaatteet, joiden varassa Attribute-kirjastoa kehitetään. On varmaa, että kirjaston toteutus muuttuu ajan kuluessa, mutta periaatteet ja tavoitteet pysyvät melko samoina.

17.1 Vaatimusten hahmottelua

Wolf-kielessä on mahdollisuus tarkistella ympäristön olioita ajon aikana kuten Squeak-ohjelmointiympäristössä [17], joka on Smalltalk-kielen moderni toteutus. C++-kielisiä osia voi selailla kuten normaaleja, Wolf-kielen sisällä määriteltyjä olioita.

Wolf-kielessä on C++-kielisiä osia varten Attribute-kirjasto, joka on tarkoitettu C++-kielellä tehtyjen osien liittämiseen Wolf-kieleen. Pelimoottorin kehittäjien pitää määrittää luokistaan kuvaukset, joissa kuvataan eri attribuuttien ja metodien tyypit ja rekisteröidä luokkansa Wolf-kielen näkyville. Wolf-kieli huolehtii osien liittämisestä Wolf-kielen ohjelmointiympäristöön, josta pelien käsikirjoittajat voivat ohjata pelin tapahtumia. Wolf-kielen on saatava jotenkin tieto C++-kielisistä osista, jotta se voi käyttää valmista pelimoottoria pelin ohjaamiseen – Wolf-kieli on silti vain juontokieli.

Koska Attribute-kirjasto on tehty modulaarisesti ja liitosten määrä eri osien välillä on vähennetty minimiin, voidaan kirjastoa käyttää lisäksi apuna esimerkiksi olioiden tallentamiseen, käyttöliittymän generointiin ja olioiden synkronointiin eri koneiden välillä. Luvussa 18 kuvataan toteutus karkealla tasolla. Luvut 19 ja 20 kuvaavat kirjaston mahdollisia käyttötapauksia. Tutkielmassa ei kuitenkaan toteuteta kuin perusrunko, jonka päälle kehittäjät voivat lisätä toiminnallisuuttaan.

17.2 C++-kieli ja Wolf

Wolf-kieliset luokat eivät näy C++-kielisille osille ja toisinpäin. Attribute-kirjaston tehtävänä on välittää viestejä Wolfin ja C++:n välillä. Ongelman voi jakaa kahteen

osaan:

- laajentaminen, jolla tarkoitetaan C++-osien liittämistä Wolf-kieleen, ja
- sulauttaminen, jolla tarkoitetaan Wolf-kielisten osien liittämistä C++-kieleen.

Kielen laajentaminen on selkeästi helpompi tehtävä, koska Wolf-kieli tukee paljon ajonaikaista tulkkausta ja luokat ladataan vasta ohjelmaa ajaessa. Wolf-kieli on hyvin suppea kieli verrattuna C++-kieleen, mikä yksinkertaistaa huomattavasti rajapintojen ja kommunikaatioprotokollan luomista Wolf- ja C++-maailman välillä. Tärkein käyttötapaus on C++-kielen liittäminen Wolf-kieleen. Valitettavasti tämä on myös vaikein toteuttaa. C++-kieli ja Wolf-kieli elävät molemmat omissa maailmoissaan.

Sulauttaminen on vaikeampi tehtävä. C++ on matalantason kieli, joka operoi lähellä laitteistorajapintaa; Wolf-kieli on tarkoitettu korkean tason oliokieleksi, joka piilottaa yksityiskohdat käyttäjältä. Voidaan toki olla montaa mieltä, onko hyödyllistä tai järkevää välittää viestejä ja dataa Wolf-kielen puolelta C++-kieleen, koska kielet poikkeavat toisistaan jo perusideoiltaan ja -ideologioiltaan niin paljon. Esimerkiksi Python-kielen käyttäjät eivät mielellään sulauta Python-kieltä C-kielen sekaan, koska skriptikielen sulauttaminen matalantason kieleen aiheuttaa ongelmia[29]. Tässä tutkielmassa ei oteta kantaa tähän ongelmaan kovin syvällisesti vaan tarkoitus on vain saada Wolf-kieli käyttämään C++-kielisiä osia avuksi. Jos Wolf-kielellä voi ohjata C++-kieltä, on kieli jo käyttökelpoinen. Kielen toteutuksessa on lopullisessa versiossa on toki pystyttävä yhdistämään sujuvasti Wolf- ja C++-kieltä.

Mitä C++-kielisten osien liittäminen peliin sitten tarkoittaa Attribute-kirjaston yhteydessä? Ainakin kehittäjien on pystyttävä määrittämään

- perintäsuhteet luokilleen, koska ylä- ja alaluokkien suhteita on pystyttävä käsittelemään myös Wolfin puolella,
- luokkien toteuttamat rajapinnat, jotta moduulien toteutusta ei tarvitse paljastaa,
- luokkiensa jäsenmuuttujat ja
- luokkiensa metodit.

Lisäksi metodien ja jäsenmuuttujien on pystyttävä näyttämään metatietoa käyttäjälle ajon aikana. Metatietoja ovat esimerkiksi tyyppi, nimi, kehittäjän antama dokumentaatio, ynnä muut sellaiset.

17.2.1 Riippuvuus Wolf- ja C++-kielen välille

Tekipä liitoksen millä tahansa tavalla tulee eri kielisten osien välille riippuvuutta jonkin verran, joka vähentää kehittäjien mielenkiintoa käyttää Wolf-kieltä. Koska Wolf-kielen on pystyttävä tarjoamaan korkean tason palveluja kehittäjille, on helpointa tarjota miltei samat palvelut myös C++-kehittäjille.

Vaikka komponenteilla on riippuvuus Attribute-kirjastoon, se ei välttämättä ole iso menetys, koska Attribute-kirjasto itsessään on modulaarinen ja sen toimintaa voi muuttaa kohtalaisen helposti. Esimerkiksi olioiden tallennusta hoitava komponentti voidaan korvata erilaisilla toteutuksilla, jos pelituote niin vaatii. Lisäksi Attribute-kirjastoon voi liittää uusia toimintoja vanhojen rajapintojen päälle. Täten Attribute-kirjasto tarjoaa yleiskäyttöisen abstraktiokerroksen, joka helpottaa kehittäjien työtä.

Toisaalta, kirjaston on oltava helposti muokattava ja hyvin dokumentoitu, jotta se kannattaa ylipäätään ottaa peliprojektiin mukaan. Kehysten käyttäminen on aina riski, vaikka säästäisi aikaa. Mitä ystävällisempi kirjasto on kehittäjälle, sitä helpompi se on ottaa mukaan toteutukseen.

17.2.2 Siirrot ovat olioita

Ensimmäisissä hahmotelmissa Attribute-kirjastosta kentät piti asettaa yksitellen. Tämä osoittautui turhan hitaaksi ja monimutkaiseksi ratkaisuksi. Säikeistetyssä ympäristössä tämä tarkoittaisi myös jokaisen olion kentän lukitsemista ja vapauttamista erikseen.

Ongelman voi kiertää melko hyvin käyttämällä näkymiä, jotka esiteltiin luvussa 12. Kaikki näkymässä olevat kentät voidaan käsitellä kerralla. Toisin sanoen, tietoa haluava olio ja sitä tarjoava olio vaihtavat keskenään kaiken tarpeellisen datan yhdellä tapahtumalla[1]. On sekä teknisesti helpompaa että nopeampaa käsitellä kaikki näkymän kentät kerralla, koska lukituksia ei tarvitse tehdä kuin kerran.

Tämä on hieman ongelmallinen asia käyttöliittymää luotaessa, koska kenttiä voidaan niissä päivittää yksitellen, mutta ongelmaa käsitellään tarkemmin luvussa 19.

[1] Englanninkielisessä kirjallisuudessa tässä yhteydessä olevasta tapahtumasta käytetään nimitystä transaction

17.3 Tavoitteet

On makuasia, mikä käyttö mielletään helpoksi ja mikä ei. Attribute-kirjastolle on valittu seuraavat tavoitteet:

Luotettava toteutus Jos toteutuksessa on runsaasti bugeja, siihen ei voi luottaa. Tämä johtaa käytön vähenemiseen.

Selkeä API Attribute-kirjaston tarkoitus on yksinkertaistaa liitosta C++- ja Wolf-kielen välillä. Käyttäjiä ei pidä rasittaa liiallisella määrällä yksityiskohtia ja muistamista, koska se vie pohjaa koko kirjaston järkevyydeltä.

Modulaarinen toteutus Jotta toteutusta voidaan muuttaa, sen on oltava modulaarinen. Tämä tarkoittaa, että miltei minkä tahansa moduulin voi vaihtaa toiseen moduuliin muuttamatta koodia.

Siedettävä muistinkulutus Attribute-kirjaston on käytettävä kohtalaisen vähän muistia, koska peleissä tarvitaan muistia jo ennestään paljon. Muistin säästäminen on kuitenkin vähemmän tärkeä kriteeri kuin aiemmin mainitut tavoitteet.

Attribute-kirjasto toteutetaan testilähtöisellä kehitysmallilla, joka auttaa useimpien tavoitteiden saavuttamisessa: Jokainen ominaisuus on testattu ainakin yksikkötesteillä, mikä lisää luottamusta toimintaan. Kehittäjä joutuu itse käyttämään APIa yksikkötesteissä, joten ikävät piirteet API:sta karsiutuvat pois kehittäjän oman mukavuudenhalun takia. Ominaisuudet testataan erikseen, joten toteutuksesta miltei itsestään modulaarisempi, koska monoliittisia kokonaisuuksia on vaikeampi testata.

Muistinkulutukseen testilähtöisyys ei valitettavasti auta. Tämä ongelma on ratkaistava vain tekemällä erilaisia toteutushahmotelmia kirjastosta, joissa etsitään sopivaa kompromissia muistinkulutuksen, nopeuden ja lähdekoodin ylläpidettävyyden väliltä.

17.4 Yhteenvetoa luvusta 17

Tärkeimmät avainkohdat luvusta ovat, että

- Attribute-kirjaston tehtävänä on antaa tapa kommunikoida Wolf- ja C++-kielen välillä,

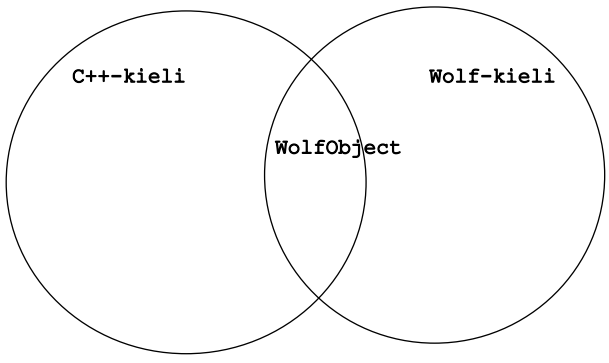
- Wolf-kielessä määritellään erilaisia näkymiä, joiden perusteella tietoa käsitellään,
- Attribute-kirjastossa näkymät päivitetään kerralla,
- Wolf-kielen sulauttaminen C++-kieleen on vaikeampaa kuin Wolf-kielen laajentaminen C++-kielisillä komponenteilla ja
- Attribute-kirjasto käyttää testilähtöistä kehitysmallia, koska se sopii parhaiten kirjaston tavoitteiden saavuttamiseen.

18 Attribute-kirjaston toteutus

Tämä luku kuvaa toteutusta attribute-kirjastolle. Tarkoituksena on antaa vain korkean tason ideat, joilla kirjasto toimii, eikä keskittyä yksityiskohtiin, joita tarkastellaan lähemmin itse toteutuksessa. On huomattava, että tässä luvussa puhutaan metodeista, jos ne ovat Wolf-kielen puolella, ja jäsenfunktioista, jos ne ovat C++-kielen puolella.

18.1 Abstrakti kantaluokka Object

Wolf-kielen virtuaalikone on toteutettu C++-kielellä. Tämä helpottaa hieman kehittämistä, koska sekä virtuaalikone että C++-kehittäjät voivat käyttää muutamia yhteisiä, sovittuja kantaluokkia. Wolf-kielen kehityskirjastossa on määritelty abstrakti kantaluokka Object, jonka avulla kehittäjät voivat laajentaa Wolf-kieltä enemmän tai vähemmän helposti C++-kielen puolelta. Abstrakti kantaluokka on wolf-nimiavaruudessa, joten siitä käytetään jatkossa merkintää wolf::Object. C++-kielessä nimiavaruus, johon luokka kuuluu, ilmaistaan samalla tavalla, joten merkintää käytetään myös tässä tutkielmassa korostamaan, että tarkoitetaan C++-kielen puolella määriteltyjä olioita. Kuvassa 18.1 on esitetty Object-luokan suhde C++- ja Wolf-maailmaan. Luokka Object kommunikoi sekä C++- että Wolf-kielen puolella.



Kuva 18.1: WolfObject näkyy sekä C++- että Wolf-kielessä

Wolf-kielen virtuaalikone käyttää wolf::Object-kantaluokkaa C++-kielisten osien komentamiseen Wolf-kielen puolelta. Wolf-kieli ei voi käyttää C++-kielen tyypi-

tystä, koska C++-kielen tyypitys on staattinen ja tapahtuu käännösaikana. Wolf-kieli puolestaan on dynaaminen kieli, joka tarkistaa osan tyypeistään ajon aikana. Kun käytetään wolf::Object-kantaluokkaa sekä C++- että Wolf-kielen puolella, C++-kielen laajuudesta ja monimutkaisuudesta ei tarvitse välittää kovin paljoa virtuaalikonetta toteuttaessa – Wolf-kielen kääntäjän ja virtuaalikoneen toteuttaminen on riittävän haastava tehtävä ilman lisäongelmia.

WolfObject-luokan instanssin on oltava tarpeeksi älykäs kertomaan sekä muuttujiensa että metodiensa nimet ja tyypit, jotta virtuaalikone voi kutsua metodeita ja käsitellä Wolf- ja C++-kielen olioita mahdollisimman saumattomasti yhteen. Muuttujien käsittelyä ei välttämättä tarvitsisi ottaa mukaan itse Wolf-kielen puolelle, mutta lisäksi luokan on pystyttävä kertomaan ylä- ja alaluokkansa, jotta Wolf-kieli voi tyypittää muuttujia oikein.

18.2 ValueDispatcher-kantaluokka

Muuttujien tyyppiä ei tiedetä Wolf-kielen puolella. Kuitenkin, jotta C++-kielisen luokan eri muuttujia voitaisiin käsitellä, pitää tietää jokaisen muuttujan tyyppi. Ongelmana on, että yleisen tavan kehittäminen eri tyyppisten muuttujien käsittelyyn on melko vaikea toteuttaa.

Wolf-kieli käyttää tähän tarkoitukseen wolf::ValueDispatcher-kantaluokkaa. Käsiteltävälle oliolle annetaan ValueDispatcher-luokan instanssi, johon olio kertoo muuttujiensa tyypit. ValueDispatcher-kantaluokka on siis oleellisesti Visitor-suunnittelumallin toteutus. Lisäksi ValueDispatcher-luokkaa käytetään muuttujien lukemiseen ja asettamiseen, koska kirjoittaja halusi tarjota yksinkertaisen tavan laajentaa ValueDispatcher-luokan toimintaa esimerkiksi käyttöliittymän ja tallennuksen kanssa. Ongelmana Visitor-suunnittelumallin kanssa on, että tyyppien lisääminen Visitor-kantaluokkaan on melko työlästä. Tämä ongelma ratkaistiin luomalla halutuista primitiivityypeistä tyyppilista ja rekursiivinen luokkamalli ValueDispatcherPart, jonka avulla rajapinnan voi generoida automaattisesti. ValueDispatcher-luokan määritys on esitelty listauksessa B.1. Idea rajapinnan ja toteutuksen generointiin tyyppilistojen avulla on lähtöisin artikkelista [45], mutta kirjoittaja muokkasi ideaa Wolf-kielen tarkoituksiin sopivaksi.

ValueDispatcher-luokan avulla oliosta voidaan kysyä kaikki virtuaalikoneen vaatimat tiedot, koska virtuaalikone ei voi käsitellä vähemmän abstraktia tyyppiä kuin wolf::Object, joka on myös mainittu primitiiveissä.

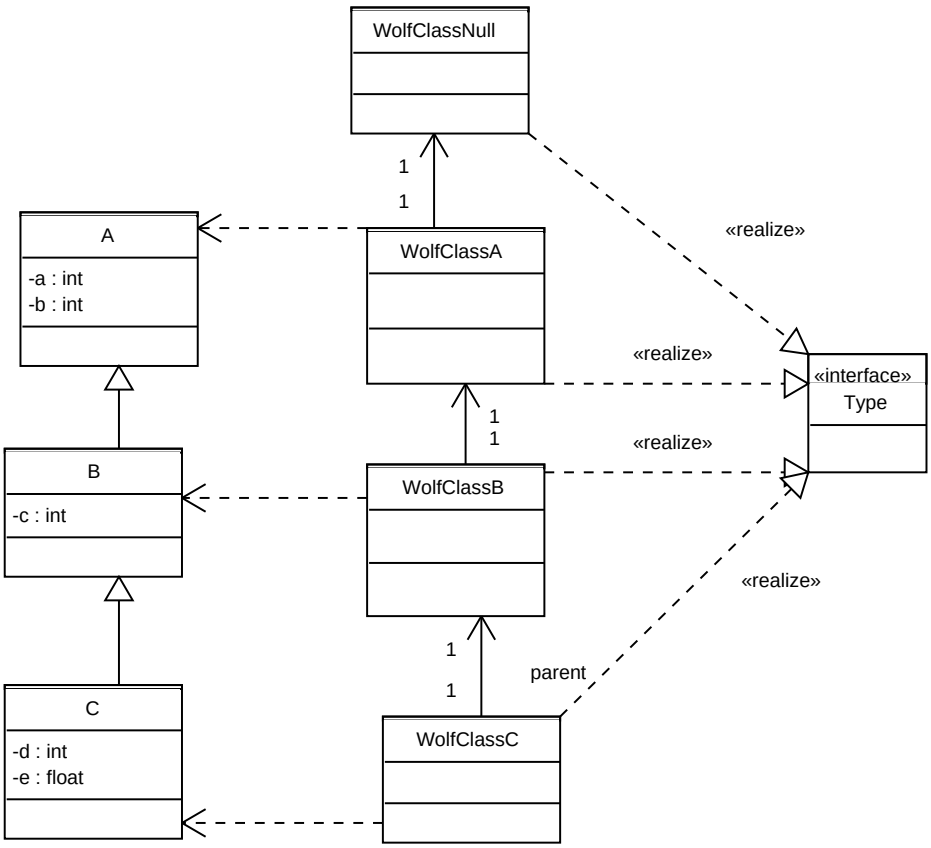
18.3 Hierarkian luominen laajennoksissa

Pelkkä muuttujien ja metodien määrittäminen ei valitettavasti riitä luomaan kunnollista abstraktiota. Tarvitaan myös hierarkian määrittely. Tämä käy luomalla – jälleen kerran – rekursiivinen luokkamalli, joka generoi hierarkian. Tässä vaiheessa ei vielä oteta kantaa itse wolf::Object-luokan instanssien luomiseen, vaan keskitytään luomaan wolf::extension::Class-luokat, joka sijaitsee – kuten nimestä voi päätellä – extension-nimiavaruudessa.

C++-kieltä käyttävän kehittäjän on määritettävä luokkiin, joilla haluaa Wolf-kieltä laajentaa, tietyt typedef-määrittelyt. Merkittävin näistä tyyppimäärittelyistä on yläluokan määritys, jotta luokkamalli wolf::extension::Class voidaan generoida oikein. Kun yläluokan tyyppi tiedetään, voi jokainen generoitava luokka generoida myös yläluokkansa. Näin jatketaan kunnes päästään perintähierarkian huipulle. On huomattava, että C++-kieltä käyttävän kehittäjän ei tarvitse kuin määrittää tyypit oikein, jotta luokkamalli wolf::extension::Class voi generoida hierarkian oikein. Tämä vähentää osaltaan riippuvuutta Wolf- ja C++-kielen välillä. Kuvassa 18.2 esitetään yksinkertainen perintäketju luokille A, B ja C sekä luokat WolfClassA, WolfClassB ja WolfClassC[1]. Kuvassa esitetty Type on Wolf-kielen abstrakti kantaluokka tyypin määritykselle.

WolfClassC-luokan instanssilla on viite luokkan WolfClassB instanssiin, jolla on edelleen viite luokkan WolfClassA instanssiin. WolfClassA luokan instanssilla on puolestaan viite WolfClassNull-luokan instanssiin, joka ei pidä sisällään mitään hyödyllistä. Se vain tarvitaan, jotta luokkamallit toimivat oikein. Luokkamalli wolf::extension::Class toteuttaa singleton-suunnittelumallin eli kustakin luokasta WolfClassA, WolfClassB ja WolfClassC ei voi olla kuin instanssi. Singleton-suunnittelumallia käytetään, jotta debuggerien lisääminen ja poistaminen tarvitsee tehdä vain yhteen paikkaan.

[1]Kirjoittaja käytti nimeä WolfClassA luokkamallin wolf::extension::Class instansoinnille tyyppiparametrilla A, koska WolfClassA on selkeämpi ja lyhyempi.



Kuva 18.2: Luokat muodostavat hierarkian

18.4 Accessorit

Tähän mennessä on vain käsitelty kahta erillistä luokkaa wolf::ValueDispatcher ja wolf::extension::Class. Tarvitaan lisäksi kolmas abstrakti kantaluokka: Accessor. Accessorit ovat yksinkertaisia luokkia, jotka saavat parametrikseen halutun luokan ja ValueDispatcher-luokan instanssin sekä indeksin, joka tarvitaan erilaisten näkymien takia.

Accessor tietää, miten luokan instanssin perusteella voi hakea halutun muuttujan ja lisäksi se tietää muuttujan tyypin. Sen ainut tehtävä onkin antaa haluttu muuttuja ValueDispatcher-luokan instanssin käsiteltäväksi. Accessor-kantaluokka on määritetty listauksessa B.2.

Esimerkiksi kuvan 18.2 luokka WolfClassA tallentaa accessorit muuttujille a ja b, WolfClassB tallentaa accessorin muuttujalle c ja WolfClassC tallentaa accessorin muuttujille d ja e. Oletetaan esimerkkinä, että halutaan käsitellä C-luokan instanssia. Tällöin WolfClassA hoitaisi instanssin muuttujien a ja b, WolfClassB muuttujan c ja WolfClassC muuttujien e ja d käsittelyn accessoriensa avulla.

Vaikka Wolf-kielen määritykset eivät (tietenkään) toimi aivan näin yksinkertaisesti, perusidea on sama myös Wolf-kielen luomien muuttujien parissa.

18.5 Metodit

Metodeja varten on Accessorien tapainen tietorakenne, jonka nimi on Method. Kun Wolf-kieli kutsuu metodia, se säilöö parametrit pinoonsa – kuten oikea konekin – ja kutsuu metodia. Method-luokan instanssi hakee parametrit virtuaalikoneen pinosta, tulkitsee ne ja kutsuu C++-kielistä metodia.

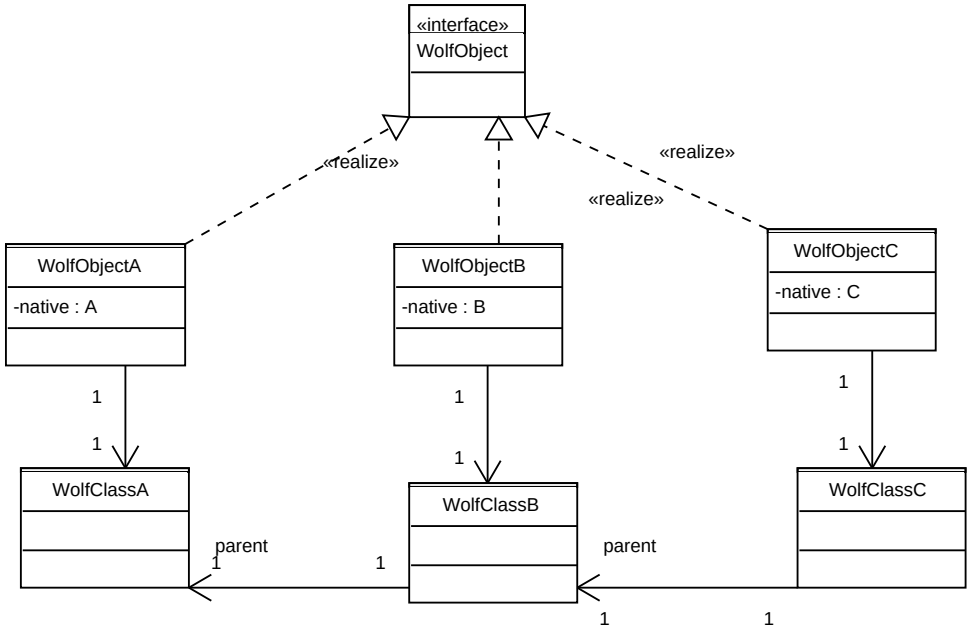
Wolf-kielen pitää saada tieto C++-kielellä kirjoitetuista metodeista, jotta se voi toimia oikein. Tämä tapahtuu lisäämällä ValueDispatcher-rajapintaan tuki myös metodien käsittelylle. Tämä ei vaadi kuin uuden tyypin lisäämisen tyyppilistaan. Method-luokka varten on oltava omat rajapintansa, joihin luokan instanssi voi kertoa parametriensa tyypit ja paluuarvonsa. Helpoiten tämä onnistui muokkaamalla Accessor-luokan yhteydessä käytettyjä ratkaisuja sopimaan Metodille.

Parametrien kutsuminen on melko suoraviivaista , kun Wolf-kieli on saanut tiedon parametrien tyypeistä. Wolf-kielen kääntäjä voi tehdä tarpeelliset tyyppitarkastukset ja itse Attribute-kirjaston tehtäväksi jää lähinnä hakea tiedot parametreista – tyyppitarkastuksia ei tarvitse enää tehdä, kun käännös on tehty.

Toteutuksessa metodit liitetään Wolf-kieleen luokkamallien avulla. Metodilla on tietty tyyppi ja tämän tyypin ansiosta voidaan generoida luokkamalli Method. C++-kehittäjän ei tarvitse nähdä vaivaa parametrien käsittelyssä, vaan Attribute-kirjaston rutiinit hoitavat ongelmat kehittäjän puolesta. Suuri osa ideoista, joilla Method-luokka toteutetaan, on lähtöisin boost-kirjastolta [11].

18.6 Olion määrittely laajentaessa

Laajennettavan luokan liittäminen Wolf-kieleen on helppoa, kun käytetään apuna luokkamallia wolf::extension::Class. Kuvassa 18.1 esitetty rakenne on melko yksinkertainen: esimerkiksi luokan WolfObjectC instanssilla on muuttuja native, joka on viite luokan C instanssiin. Lisäksi luokka toteuttaa abstraktin kantaluokan wolf::-Object, joten virtuaalikone voi käyttää sitä.



Kuva 18.3: Olion suhde luokkaan

Listauksessa B.3 esitellään yksinkertaistettuna, miten olion muuttujat käydään läpi. Listauksessa käsitellään vain C++-kielen puolella määriteltyjä muuttujia esimerkin selkeyttämiseksi. On oleellista havaita, että kutsussa yläluokalle (super-muuttuja listauksessa) ei tarvitse erillistä tietorakennetta vaan tyyppimuunnos hoidetaan jo Attribute-kirjaston puolella. Tämä helpottaa toteutusta omalta osaltaan.

Käytännössä esimerkiksi luokan WolfObjectC instanssi delegoi miltei kaikki kut-

sunsa luokalleen WolfClassC, joka käsittelee pyynnöt instanssin antamien jäsenmuuttujien pohjalta. Kun instanssi antaa muuttujansa native WolfClassC luokalle, sen tyyppi on C. WolfClassC kutsuu puolestaan WolfClassB:tä ja antaa saadun muuttujan native sillekin. Koska C perii luokan B, voi WolfClassB käyttää muuttujaa kuten luokan B instanssia. Sama idea jatkuu eteenpäin luokkaan WolfClassA. Tämän takia ei tarvitse luoda erikseen perintähierarkiaa WolfObjectC <: WolfObjectB <: WolfObjectA <: WolfObject. Perintähierarkian syvyys voidaan täten pitää matalana.

18.7 Käytännön esimerkki

Listauksessa 18.1 annetaan yksinkertainen esimerkki laajentamisesta. C++-kehittäjä on luonut luokan C, joka perii luokan luokan B. Lisäksi luokka toteuttaa abstraktin kantaluokan MyInterface.

Makrot WOLF_ACCESSORS_BEGIN ja WOLF_ACCESSORS_END määrittelevät staattisen jäsenfunktion, joka saa parametrikseen tyypin WolfClassC instanssin. Makro WOLF_ACCESSOR määrittää uuden tyypin, joka toteuttaa Accessorin halutulle tyypille ja joka käsittelee halutun muuttujan. Uuden tyypin määrittäminen vain muuttujan käsittelyyn on rumaa, mutta kehittäjän tutkimista vaihtoehdoista silti paras. Jokainen määritelty accessor lisätään parametrina saatuun luokkaan.

Vastaavasti makrot WOLF_METHODS_BEGIN ja WOLF_METHODS_END määrittelevät staattisen jäsenfunktion, joka myös saa parametrikseen tyypin WolfClassC instanssin. Makrot määrittelevät sopivan kääreen annetulle jäsenfunktiolle automaattisesti.

Makro WOLF_CLASS_CONSTRUCTOR, joka saa parametrikseen tyyppivektorin, määrittää konstruktorin vaatimat tyypit ja samalla myös parametrien lukumäärän.

Makro WOLF_CLASS_IMPLEMENTS määrittää abstraktit kantaluokat, jonka luokka toteuttaa. Luokka voi määrittää useamman abstraktin kantaluokan, mutta niissä ei saa olla muuttujia ja niiden kaikkien jäsenfunktioiden olisi syytä olla pure-virtual.

Listaus 18.1: Esimerkki laajentamisesta

```
1 class C : public B, public MyInterface {
2 public:
3   WOLF_CLASS(C);
4   WOLF_CLASS_EXTENDS(B);
```

```
5     WOLF_CLASS_IMPLEMENTS(<MyInterface>);
6     WOLF_CLASS_CONSTRUCTOR(<int>);
7
8     WOLF_ACCESSORS_BEGIN();
9     WOLF_ACCESSOR(e,"e_title","e_description");
10    WOLF_ACCESSOR(e,"e_title","e_description");
11    WOLF_ACCESSORS_END();
12
13    WOLF_METHODS_BEGIN();
14    WOLF_METHOD(doSomething,"doSomething","no_desc");
15    WOLF_METHOD(doOther,"doOther","");
16    WOLF_METHODS_END();
17
18    WOLF_CLASS_END();
19  public:
20    C(int i);
21    ~C();
22
23    void doSomething(int i,float b);
24    void doOther(MyInterface *myInterface);
25  private:
26    int e;
27    int d;
28  };
```

Makroissa on kohtalaisen paljon rajoituksia:

- käytettyjen tyyppien on oltava Wolf-kielen tiedossa,
- luokalla voi olla Wolf-kielen näkökannalta vain yksi yläluokka,
- abstrakteja kantaluokkia, joita luokka toteuttaa, voi olla korkeintaan yhdeksän[2] kappaletta ja
- metodeissa ei voi olla kuin yhdeksän parametria.

Lisäksi tutkielmaan toteutettua versiota koskee ainakin seuraavat rajoitukset:

- jäsenfunktioiden parametrien lukumäärää on kavennettu entisestään kolmeen,
- olioiden konstruktorit eivät vaadi parametreja,

[2]Yhdeksän on yläraja, jonka kirjoittaja katsoi sopivaksi parametreille ja abstraktien kantaluokkien määrälle. Yläraja on kyllä valittu melko sattumanvaraisesti.

- jäsenfunktiot voivat käyttää parametreina vain int tai float-tyyppisiä muuttuja,
- jäsenfunktiot voivat palauttaa vain void, int tai float-tyypin arvoja ja
- abstraktien kantaluokkien käsittelyä ei huomioida. Makro ei oleellisesti tee mitään hyödyllistä.

18.8 Yhteenvetoa luvusta 18

- Yksityiskohdat Attribute-kirjastolle näkyvät parhaiten toteutuksesta.
- C++-kielen perintähierarkia ei säily Wolf-kielen puolella sellaisenaan.
- Tutkielmaa varten tehty toteutus on attribute-kirjastosta on hyvin suppea.
- Laajennos tapahtuu luokkamalleilla, joiden käyttö voi hidastaa kääntämistä.
- Tutkielmaa varten tehty versio kirjastosta on todella suppea toiminnallisuudeltaan.

19 Käyttöliittymä ja PAC-suunnittelumalli

> *User Interaction: It Was Hard to Build, It Should Be Hard to Use* – Waterfall2006 [2]

Tämä kappale antaa hahmotelman, miten Wolf-kielessä luodaan käyttöliittymiä olioille. On korostettava, että koko käyttöliittymä on eräänlainen bonus, joka saadaan Attribute-kirjaston kautta. C++-kielen olioiden on joka tapauksessa pystyttävä kertomaan rakenteensa Wolf-kielelle, jotta Wolf-kieltä voi laajentaa Mikään ei estä tekemästä rakenteesta niin joustavaa, että samaa mekanismia voi käyttää käyttöliittymän luomiseen. Peliä kehittäessä voi olla kallisarvoinen tieto tarkistaa, miten eri oliot toimivat, muutella olioiden arvoja ja katsoa muutosten vaikutukset.

Kehittäjän kannalta voi olla hyödyllistä muuttaa säätilaa kontrolloivan olion arvoja pelin pyöriessä, mennä pelimaailmassa tien laidalle ja ihmetellä, miten nopeasti tullut talvi yllättää virtuaalimaailman autoilijat. Mikään ei tietenkään voita kunnon yksikkötestausta, mutta pelin hienosäädössä tämäntyyppinen toiminnallisuus on hyödyksi.

19.1 Kieli ja käyttöliittymä

Wolf-kieli ottaa kantaa käyttöliittymään, mutta vain kirjastojensa kautta. Yleensä kieli vanhenee hitaammin, mitä käyttöliittymät muuttuvat, joten ylläpidon kannalta olisi vaarallista määrittää käyttöliittymä osaksi kieltä kuten Cobol-kielessä.

Java-kielen Swing-kirjasto on abstraktoitu toteutus käyttöliittymästä ja se helpottaa omalta osaltaan kielen tekemistä alustariippumattomaksi [18]. Jos Swing-kirjastoa ei olisi, ohjelmien siirtäminen esimerkiksi Linux- ja Windows-ympäristöjen välillä olisi vaikeaa.

Pelisovelluksissa alustariippumattomuus on vielä tärkeämpää kuin perinteisissä työpöytäsovelluksissa, koska pelaajat kehittävät nykyään enemmän ja enemmän omia laajennoksia peleihin ja esimerkiksi OpenGL-kiihdytetyn käyttöliittymän päälle olisi tottumattoman melko vaikea tehdä omaa käyttöliittymää. Wolf-kielessä on perusrutiinit käyttöliittymän luomiseen ja toiminnallisuuteen, jotta peleihin olisi helpohko tehdä omia lisäosia – ja pidentää pelituotteen elinkaarta.

Wolf-kielen kirjastot ottavat hieman reippaammin kantaa käyttöliittymään kuin esimerkiksi Java-kieli: kieltä käyttävien kehittäjien ei tarvitse määrittää käyttöliittymää lainkaan, vaan kielen kirjastot tarjoavat mekanismit käyttöliittymän automaattiseen generointiin. Tämä on hyvin radikaali lähestymistapa, joka tuntuu ensivaikutelmalta liian suurelta vaatimukselta. Tämä kappale kertoo kuitenkin perusteet, miksi automaattinen käyttöliittymän luominen toimii – ainakin teoriassa – ja mitä etuja sillä saavutetaan.

19.2 Presentation-Abstraction-Control

Luvussa 8.1 kritisoitiin MVC-suunnittelumallin käyttämistä väärin useissa tilanteissa. Myöhemmin kehitetty Presentation-Abstraction-Control-malli kehitti alkuperäistä MVC-mallia eteenpäin hierarkioiden ja eri näkymien suuntaan – joka oli ideana jo MVC-mallissa.

PAC-mallissa on siis selkeästi erotettavissa kolme eri kerrosta, joista jokaisella on oma, tarkasti määritelty tehtävänsä:

Control Wolf-kielessä jokainen olio on käyttöliittymäsuunnittelun kannalta kontrollikerroksessa. Kontrollikerroksen tehtävänä on huolehtia käyttöliittymän sisällöstä mutta ei sen esityksestä.

Abstraction Abstraktiokerros toimii nimensä mukaisesti abstraktiona esityskerrokselle. Sen tehtävänä on eristää käyttöliittymän toteutus kontrolli-kerroksesta. Tämä kerros on Wolf-kielen vaativin osa saada toimimaan oikein. Sen on oltava tarpeeksi monipuolinen tarjoamaan hyvä käyttöliittymä, mutta tarpeeksi suppea, jotta se ei aiheuta ylläpidollista painajaista.

Presentation Esityskerros toteuttaa abstraktiokerroksen. Sen on huolehdittava käyttöliittymän alustuksista, vastattava käyttäjän vaatimuksiin ja kutsuttava olioilta käyttöliittymän päivitystä tai luomista.

On olennaista, että kontrollikerros ei välitä tippaakaan käyttöliittymän toteutuksesta. Varhaisissa testeissä oliot pystyivät luomaan käyttöliittymät sekä web- että Linuxin Gnome-ympäristöihin ilman muutoksia lähdekoodiin. Vaikka nämä testit ovat jo hävinneet inhimillisen virheen takia[1], kirjoittaja pystyi todistamaan, että käyttöliittymän toteutus voidaan erottaa muusta lähdekoodista.

[1] Real men don't take backups...

Samalla oliolla voi olla useita erilaisia näkymiä. Esimerkiksi opiskelijaa tarkasteltaessa opettaja ja sairaanhoitaja näkevät saman henkilön eri tavalla. Wolf-kielen tapa mallintaa erilaiset näkökulmat on luoda uusi näkymä eri näkökulmille. Näkymä kuvaa, mitä tietoja halutaan kussakin tilanteessa näyttää käyttäjälle. Pelimaailmassa on samasta oliosta eri näkymät pelin kehittäjiä ja pelaajia varten.

19.2.1 Taustaa päätökselle ottaa kantaa käyttöliittymään

Vuonna 2004 kirjoittaja oli luomassa Terveyspuntari-sovellusta [26], joka toteutettiin sovellusprojektina Jyväskylän yliopistolla. Terveyspuntari oli web-sovellus, jota käytetään edelleen Saarijärven kaupungilla. Sovellukseen pystyi luomaan päiväkirjoja, joihin käyttäjät pystyivät kirjoittamaan merkintöjä esimerkiksi painostaan ja veren sokeriarvoista. Sovellus antoi palautetta arvojen perusteella ja kehotti siirtymään lääkäriin, jos arvot olivat terveyden kannalta vaarallisia.

Sovelluksessa oli automaattisesti generoitavat käyttöliittymät. Jokaista päiväkirjaluokkaa kohti määriteltiin tietokannassa taulu, johon käyttäjien merkinnät tallennettiin, ja taulu, joka kuvasti luokan rakenteen. Luokan rakenteen perusteella voitiin generoida käyttöliittymä seuraavasti:

1. olion tyyppi-tunnisteen perusteella haettiin tietokannasta olion rakenne, jossa oli tiedot olion kentistä,

2. rakenteen perusteella generoitiin sopiva SQL-kysely tietokantaan, jolla muutettiin tai lisättiin tietoja päiväkirjaan,

3. rakenteen perusteella luotiin olion käyttöliittymä ja edelleen rakenteen perusteella kyselyn tulos liitettiin automaattisesti luotuun käyttöliittymään.

Samaa rajapintaa pystyi käyttämään erilaisten käyttöliittymien luomiseen eikä aikaa kulunut käyttöliittymän hiomiseen kuin projektin alussa ja lopussa, jolloin sitä muutettiin. Testaus nopeutui oliomaisen rakenteen ansiosta huomattavasti: jos käyttöliittymässä oli virheellinen arvo, se johtui todennäköisesti liiketoimintamallin virheistä, koska käyttöliittymän hoitavat luokat oli testattu huolellisesti jo aiemmin.

Vaikka Terveyspuntarissa oli pieniä puutteita arkkitehtuurin ja toteutuksen puolesta, käyttöliittymien muokkaaminen ja toiminnallisuuden lisääminen siihen oli helppoa. Kirjoittajan hyvät kokemukset PAC-mallista ja erittäin huonot kokemukset huonosti toteutetun MVC-mallin ylläpidosta johtivat päätökseen ottaa kantaa Wolf-

kielen käyttöliittymiin. PAC-mallin käyttö on Wolf-kielessä helpompaa kuin MVC-mallin käyttö, jotta kehittäjät eivät tekisi huonosti toteutettuja MVC-ratkaisuja.

19.3 Käyttöliittymät peleissä

Peleissä on eräs etu verrattuna perinteiseen työpöytä-ohjelmaan: ruutua pitää päivittää koko ajan. Tällöin ei tarvitse välttämättä käyttää Observer-suunnittelumallia ilmaisemaan käyttöliittymälle, milloin ruutua on päivitetty ja milloin ei. Tämä säästää vaivaa kehittäjältä.

Huonona puolena on, että käyttöliittymän tekeminen peliympäristöön on huomattavan paljon vaikeampaa kuin muihin graafisiin ohjelmiin: käyttöliittymän käsittelyyn vaadittavat luokat joutuu yleensä luomaan alusta lähtien itse. Erilaiset rajapinnat grafiikan piirtämiseen – tärkeimpinä OpenGL ja Direct3d – eivät ole yhteensopivia. Avoimen lähdekoodin projekti CEGUI helpottaa kuitenkin osaltaan käyttöliittymän rakentamista [36].

19.4 Käyttöliittymän interaktiivuus

Käyttöliittymä vaatii interaktiivisuuta, mikä eroaa useimmista Attribute-kirjaston käyttökohteista: useimmat tapahtumat vain siirtävät tai hakevat näkymässä määritellyt kentät oliosta ja unohtavat arvot sen jälkeen. Käyttäjän on kuitenkin pystyttävä vaihtamaan kenttiä yksitellen – kentän päivitys ei ole ongelma, koska ruutu päivitetään joka tapauksessa useita kertoja sekunnissa peleissä. Miten tälläinen toiminta voidaan saavuttaa?

Alkuperäinen idea oli luoda jokaista kenttää kohti olio, joka tarjosi interaktiivisuuden. Ajatus oli jopa niin houkutteleva, että kirjoittaja toteutti sen, koska se tuntui intuition perusteella oikealta. Huonona puolena oli, että viitteet olioon voivat vaihtua mallikerroksen oliossa, josta käyttöliittymän tiedot pyydetään. Tällöin käyttöliittymä näytti kenttänä eri oliota kuin itse oliossa oli. Tämä oli helppo ongelma korjata käyttämällä osoittimen osoitinta. Seuraava ongelma oli käyttöliittymän muistinkulutus. Jokaiselle kentälle jokaisessa oliossa piti uhrata muistia, vaikka suuri osa luokista on C++-maailmasta saatuja olioita, joissa on jo valmiina vaaditut tiedot.

Vaikka alkuperäinen Attribute-kirjasto pystyi generoimaan käyttöliittymiä ja toimi hyvin, kirjoittaja tiesi sen johtavan ongelmiin ylläpidossa. Valmis Attribute-kirjasto joutui uudelleenkirjoitettavaksi suunnitteluvirheen takia – mikä puolestaan hidasti

tämän tutkielman valmistumista.

Uusi versio korostaa enemmän käyttöliittymän tapahtumia eikä niinkään arvoja. Uusi protokolla malliolion ja käyttöliittymän välillä on karkeasti seuraava:

1. Käyttöliittymä pyytää mallioliolta tietojen näyttämistä varten lomakkeen.

2. Lomake pitää sisällään kopiot mallikerroksen olion kenttien nimistä ja arvoista. Täten lomake toimii myös proxynä[2].

3. Arvojen tyyppi tiedetään Attribute-kirjaston mekanismien kautta. Tyypin perusteella luodaan käyttöliittymän näyttämiseen tarvittavat oliot esimerkiksi CEGUI kirjastolla.

4. Jos arvo on tyypiltään olio, sitä varten luodaan oma alilomake, joka liitetään käyttöliittymään. Laajassa luokassa alilomakkeet ja niiden alilomakkeet tekisivät käyttöliittymästä sekavan, joten mukana on oltava jokin turvamekanismi alilomakkeiden syvyyden rajaamiselle. Turvamekanismit eivät ole vielä valmiina, koska kirjoittajan on mietittävä niitä useammasta näkökulmasta.

5. Jos mallikerroksen olio vaihtaa kentässä viitattua oliota johonkin toiseen olioon, käyttöliittymä havaitsee tämän, koska kentän nimi ja arvo pari ei vastaa enää.

6. Jos mallikerroksen olion muuttujan arvo on vaihtunut, tarkastetaan, onko uuden olion lomake samanlainen kuin vanhan olion lomake.

7. Jos lomake on samanlainen, käytetään edelleen vanhaa lomaketta. Muussa tapauksessa pyydetään uusi lomake alioliolle ja heitetään vanhat arvot pois.

Tämä tapa toimia on aavistuksen verran hitaampi kuin suorat viittaukset olioihin, mutta se toimii varmemmin. Hidastakin toimintaa voi optimoida melkoisesti, mutta virheellinen toiminta hidastaa Wolf-kielen valmistumista. Lisäksi protokolla sallii rutiinien siirtämisen Wolf-kielen valmiiden kehysten puolelle, jotta käyttöliittymän tekijöillä on vähemmän rutiineja toteutettavaksi itse. On korostettava, että lomakkeiden määritys ja luominen perustuu Attribute-kirjastoon ja että koko käyttöliittymä luodaan dynaamisesti.

[2]Koska kaikki muuttujat käsitellään viitteinä, proxyjen vaatima tila ei ole kovin suuri.

19.5 Yhteenvetoa luvusta 19

Tärkeimmät seikat luvusta 19 ovat:

- Wolf-kieli vähentää tarvetta kirjoittaa erillisiä käyttöliittymiä C++-kielisiin olioihin,
- Wolf-kielen käyttöliittymä on interaktiivinen,
- Käyttöliittymän päivitys ja tietojen muuttaminen perustuu tapahtumiin eikä yksittäisten arvojen muutoksiin ja
- Käyttöliittymän tekeminen peliympäristöön on vaikeaa.

20 Olioiden säilyvyys

Tämä luku käsittelee säilyvyyden ongelmia Wolf-kielessä, joka on kohtalaisen laaja aihe. Koska jokaisella pelillä on omat tarpeensa ja koska Wolf-kielen pitää sopeutua eri pelien tarpeisiin mahdollisimman hyvin, tämä tutkielma ei anna kovin kattavaa kuvausta olioiden tallentamisesta. Luvun mietteet perustuvat pääosin kirjoittajan kokemuksiin IGIOS-peliprojektin ajoilta [19]. Kuten luvussa 19 esitetty käyttöliittymä, myös säilyvyys on positiivinen yllätys, joka tulee Attribute-kirjaston kautta. Jos olion rakenne on jo tiedossa, miksei käyttäisi tietämystä myös olioiden lataamiseen ja tallentamiseen. Teknisesti tämä on vain yksi näkymä lisää olion muuttujiin.

Pelimaailma on pystyttävä tallentamaan levylle ja lataamaan sieltä uudelleen. Nykyisin pelialueiden koko on jatkuvasti kasvanut – esimerkiksi Second Lifen virtuaalitodellisuus laajenee koko ajan [31]. Koko pelialue ei mahdu koneen keskusmuistiin mutta pelialueen on silti tunnuttava olevan koko ajan olemassa. Olioita on pystyttävä tallentamaan ja hakemaan kovalevyltä isoissa peleissä.

20.1 Miten oliot tallennetaan levylle?

Olioiden tallentaminen serialisoimalla on melko triviaali tehtävä, joka vaatii vain olioiden kirjoittamisen johonkin tiedostovirtaan. Tallennus suoralla kirjoittamisella on myös riittämätön tapa hoitaa tallennus: olioiden referenssit toisiinsa eivät säily. Jos oliot eivät enää voi viitata toisiinsa kuten ennen tallennusta, tallennusmetodi on melko hyödytön.

Wolf-kielen metatiedot luokasta auttavat kuitenkin tallennusta: Luokka osaa kertoa olion rakenteen. Toisin sanoen, olion muuttujien tyypit tiedetään. Koska olion kaikkien tallennettavien muuttujien tyypit tiedetään, voidaan laskea olion levyllä vaatima tila. Edelleen, koska kaikkien Wolf-kielen tuntemien olioiden rakenne tunnetaan, voidaan oliotyyppiset muuttujat rakentaa uudelleen, jos tiedetään muuttujan paikka levyllä.

Näiden perustietojen pohjalle voidaan tallennus hoitaa seuraavasti:

1. Tarkastetaan olion tunnisteen perusteella, onko olio jo tallennettu kerran kovalevylle.

2. Jos olio on tallennettu kertaalleen kovalevylle, hypätään kohtaan 5
3. Jokainen luokka tallentaa omat instanssinsa tiettyyn tiedostoon. Etsitään tiedostosta sopiva paikka kirjoittamiselle.
4. Olion tyyppi, tunniste ja löydetty kohta tallennetaan globaaliin indeksiin, jossa on kaikkien olioiden tiedot.
5. Kirjoitetaan olion muuttujat levylle käyttämällä Attribute-kirjaston tarjoamaa rajapintaa.
6. Jos muuttujan tyyppi on olio, sovelletaan algoritmia niillekin.

Olioiden lataus tapahtuu jo helpommin:

1. Luetaan olion tunniste levyltä.
2. Etsitään tunnisteen perusteella globaalista indeksistä tyyppi ja paikka levyllä. Lisäksi tarkistetaan, onko olio jo muistissa.
3. Jos olio on jo muistissa, hypätään loppuun.
4. Jos olio ei ole muistissa, etsitään tyypin perusteella oikea tiedosto ja paikka tiedostosta.
5. Alustetaan muuttujat levyllä olevan tiedon perusteella käyttämällä Attribute-kirjastoa.
6. Jos muuttuja on olio, sovelletaan algoritmia siihenkin.

Tämä on yksinkertaistettu versio rakenteesta, jota Wolf-kieli käyttää olioiden tallennukseen, mutta antaa idean perusideasta.

20.2 Lohkotus ja super-Frustum

Käytännössä jokainen suurempi 3d-peli on jaettava jotenkin lohkoihin, jotka perustuvat geometriaan. Esimerkkejä lohkotukseista on esimerkiksi octree, BSB, quadtree ja niin edelleen. Jaotusta on pakko käyttää grafiikan piirtämisessä, koska niiden avulla voidaan nopeasti päätellä, voiko jokin olio näkyä pelaajalle ja pitääkö se piirtää näytölle. Pelaajan ympärille mielletään frustum, joka on tietyn muotoinen geometrinen kappale – yleensä kartio. Jos frustum osuu lohkoon, siinä voi olla oliota, joita pitää piirtää näytölle.

Samaa ideaa voi käyttää myös tallennukseen. Frustumin ympärille mielletään hieman laajempi kartio. Jos pelaajan ympärillä pysyvä super-Frustum osuu johonkin lohkoon, se pitää ladata muistiin. Suurin ongelma muistiin lataamisessa on miettiä, miten säilyvyyden saa toteutettua niin, ettei pelin suoritus lakkaa hetkeksikään. Tämä onnistuu helpoiten erillisessä säikeessä, joka pyörii taustalla.

Vastaavasti, kun super-frustum ei osu lohkoon, se pitää poistaa muistista. Ongelmallinen osa on, että olioissa voi olla toisiinsa referenssejä. Tähän ongelmaan on melko varmasti käytettävä mark-and-sweep-algoritmin sovellusta:

1. Roskienkeruu pyytää jokaista pelin moduulia[1] merkitsemään oliot, joita moduuli käyttää poistettavassa lohkossa.

2. Kun jokainen moduuli on merkannut olionsa, roskienkeruu tallentaa levylle ne oliot, jotka eivät ole merkattuina.

Mark and Sweep algoritmissa on huonona puolena, että se pysäyttää suorituksen. Onneksi yhdessä lohkossa ei ole kovin montaa oliota, joten tarkastus ei vie loputtoman kauan.

Ongelmallinen säilyvyyden käsittely muuttuu vielä monimutkaisemmaksi, kun otetaan huomioon, että kaikkia olioita ei voi päivittää kerralla, jos Wolf-kielellä luodaan uusi muuttuja johonkin kenttään. Pelimaailmassa voi olla satoja mega- tai jopa kymmeniä gigatavuja dataa. Jos pelimaailman kaikki oliot lisäisivät yhden uuden kentän olioihin, pelimaailmaa ei voitaisi päivittää pitkään aikaan. Tämä ei ole hyväksyttävä toimintamalli vaan olioita on päivitettävä laiskasti: sekä vanhojen että uusien versioiden on oltava kovalevyllä ja vanhoja versioita päivitettävä uuteen muotoon, kun vanhoja olioita käsitellään. Tässäkin kohtaa Wolf-kielen metatiedot luokasta auttavat: olioita voi luoda myös vanhentuneen datan perusteella. Koska lisäksi muuttujat pitää alustaa oliota luodessa, voi eri sukupolvien tukeminen samasta oliosta olla melko kivutonkin operaatio.

Ongelmana on, että säilyvyyden ongelmista ja ratkaisuista saisi kirjoitettu kirjan tai pari. Tässä tutkielmassa ei olekaan tarkoitus antaa kattavaa kuvaa tallennuksessa vaan tarjota vain ideoita tallennuksen mahdolliselle toteuttamiselle.

20.3 Yhteenvetoa luvusta 20

Tärkeimmät avainkohdat luvusta 20 ovat:

[1] Näihin moduuleihin kuuluu myös Wolf-kielen virtuaalikone.

- Säilyvyydessä kannattaa käyttää Attribute-kirjastoa,
- Säilyvyys on monimutkainen asia, jota ei pysty käsittelemään tämän tutkielman laajuudessa ja
- Pelimaailman on pyörittävä koko ajan, vaikka olioita tallennettaisiin ja hävitettäisiin.

Osa V

Tulosten analysointia ja loppumietteitä

21 Tulosten analysointia ja loppumietteitä

> Colonel Brandt: What will we do when we have lost the war?
> Captain Kiesel: Prepare for the next one.
> – Cross of Iron

Tässä osassa käsitellään ja analysoidaan tuloksia Wolf-kielestä. Tämä osa on toisaalta tieteellisin, koska mukana on tutkimustyön analysointia, toisaalta epätieteellisin, koska mukana on tulosten analysointia. Kirjoittaja on tehnyt kieltä kauan ja hartaasti, joten on vaikea välttyä henkilökohtaisen mielipiteen tuomisella mukaan tekstiin.

21.1 Metaohjelmointi pitäisi olla paremmin tuettuna

Päällimmäisenä tuntemuksena taustatutkimusta tehdessä jäi, että metaohjelmoinnin pitäisi olla seuraava askel oliokielten kehityksessä.

Näkökulma-ajattelu ja -ohjelmointi auttaa tekemään modulaarisempia ohjelmia. Osa kehittäjistä ajattelee näkökulmaohjelmoinnin tavoin, mutta työkalut näkökulmaohjelmointiin on melko alhaisella tasolla. Tilannetta voisi verrata olio-ohjelmointiin ennen Simula-kieltä: ajattelumalli on uuden ohjelmointiparadigman mukainen, mutta toteutukset joudutaan tekemään vanhojen paradigmojen työkaluilla. Kirjoittaja kehitti näkymän idean ja käsitteen Wolf-kieltä varten[1]. Näkymät ratkaisevat ainakin osan ikävistä ongelmista ohjelmiston ylläpidolle ja käännöksen suorituskyvylle verrattuna säännöllisten lausekkeiden käyttöön.

Rajapintojen metodeihin pitäisi pystyä määrittämään esi- ja jälkiehdot, jotta luokkien oletukset voitaisiin dokumentoida selkeämmin. Jos uusi luokka, joka toteuttaa rajapinnan ei mene läpi esi- ja jälkiehtojen tarkastuksista, se voi aiheuttaa yllätyksiä ohjelmiston kehittämiselle ja ylläpidolle. Tähänkin ongelmaan auttaisi näkökulmaohjelmoinnin käyttö. Kirjoittaja itse epäilee, että tämäkään ei ole riittävä suoja väärinkäsityksiä ja -käytöksiä vastaan. Mikään valmis työkalu ei estä kunnon yksikkötestausta, koska ehdotkin voidaan määrittää väärin tai liian löysäksi.

[1] Joskin kirjoittaja uskoo, että sama idea on keksitty myös jossain muualla itsenäisesti. Ratkaisumalli on liian yksinkertainen, jotta sitä ei oltaisi mietitty jo aiemmin.

Piirreohjelmointi säästäisi huomattavan paljon vaivaa ylläpitäjiltä ja kehittäjiltä, koska konfliktit havaitaan nopeammin kuin perintää käyttäen. Lisäksi perintää ei tarvitsisi enää käyttää vääräoppiseen toiminnan uudelleenkäyttöön, joka selkeyttäisi ohjelmien rakennetta. Kirjoittaja on joutunut ylläpitämään perintähierarkialtaan varsin arveluttavia järjestelmiä, jotka piirreohjelmointia käyttäen olisivat huomattavan paljon selkeämpiä käyttää ja laajentaa.

Valitettavasti metaohjelmoinnin toteuttaminen kieleen ei ole helppoa, jos suorituskyvyn on oltava korkea. Smalltalk- ja Lisp-kielet lienevät onnistuneimmin liittäneet metaohjelmoinnin mukaan kieleen.

21.2 Attribute-kirjaston mietteitä

Tutkielman käytännön osassa toteutetuun attribute-kirjastoon kirjoittaja on melko tyytyväinen. Kehitys alkoi jo vuonna 2005 kirjoittajan ollessa vielä mukana IGIOS-peliprojektissa [19] ja pitkä suunnitteluaika on auttanut karsimaan toiminnan mahdollisimman yksinkertaiseksi muttei yksinkertaisemmaksi.

Tämänhetkinen toteutus attribute-kirjastosta on melko nopea, mutta vaatii vielä paljon työtä, jotta se toimisi riittävän hyvin tuotantokäyttöön. Kirjaston toteutuksen voi katsoa kuitenkin onnistuneen hyvin tavoitteissaan, koska API on kohtalaisen selkeä, toteutus on modulaarinen ja muistinkulutus on siedettävä.

Kirjastoa toteuttaessa ei vielä ole mietitty sulauttamista lainkaan, mikä on iso puute. Ongelma ei kuitenkaan ole ajankohtainen vielä toviin, joten sitä ei mietitä vielä tässä vaiheessa kielen kehitystä.

21.3 Kielen suunnittelu on vaikeaa

Kirjoittajan alkuperäinen haave oli luoda parempi kieli kuin C++. Kirjoittaja ei voinut ymmärtää, miksi ohjelmointi C++-kielellä oli niin hirvittävän vaikeaa.

Kielen toteuttaminen osoittautui kuitenkin vaikeaksi. Vaikka useat kielet ovat vaikeita käyttää ja niiden kanssa joutuu tappelemaan turhien, ikävien yksityiskohtien kanssa, on paremman kielen tekeminen osoittautunut erittäin haastavaksi tehtäväksi – vaikka Wolf on tarkoitettu vain juontokieleksi. Vuonna 2002 loppupuolella kirjoittaja arveli projektin vievän pari vuotta. Nyt – 2007 vuoden loppupuolella – aikatauluarvio on edelleen pari vuotta.

Nykyään kirjoittaja ymmärtää ratkaisut, joita Bjarne Stroustrup teki C++-kieltä

luodessa. Lisäksi hän ymmärtää, miksi ratkaisuihin päädyttiin. Kirjoittaja on edelleen eri mieltä monesta ratkaisuista, joita kielessä on, mutta C++-kielen rajoitusten ja mahdollisuuksien parempi ymmärtäminen on auttanut luomaan parempia ohjelmia C++-kielellä. C++-kieli ei edelleenkään ole kirjoittajan suosikkikieli, mutta Wolf-kieltä ei enää kehitetä inhosta mitään kieltä kohtaan.

Kääntäjien parempi ymmärtäminen on auttanut luomaan tehokkaampia ohjelmia myös muilla kielillä, koska kirjoittajan oli syvennyttävä paremmin esimerkiksi Java-, Python- ja Smalltalk-kieliin Wolf-kieltä suunniteltaessa. Jokaisessa kielessä on rajoituksensa ja Wolf-kielenkin on hyväksyttävä tämä ikävä tosiasia. Kirjoittaja halusi Wolf-kielestä alunperin yksinkertaisen ja selkeän kielen. Ruokahalu kasvoi syödessä ja tällä hetkellä Wolf-kieli on jo matkalla kohti monimutkaista ja sekavaa kieltä – jollaista kirjoittaja halusi välttää kaikin tavoin.

Wolf-kielessä on paljon rajoituksia, mutta paljon kehitystä helpottavia työkaluja – kuten piirre- ja näkökulmaohjelmointi. On silti epäselvää, ovatko rajoitukset vain kehittäjien tiellä vai estävätkö ne todella tekemästä huonoja ratkaisuja. Wolf-kieli on kuitenkin suunniteltu kieleksi, jolla kirjoittaja itse haluaisi työskennellä. Suunnittelua helpotti oleellisesti, että kieltä ei ole tarkoitettu yleiskäyttöiseksi kieleksi, jolla voisi ratkaista kaikki kuviteltavissa olevat ongelmat. Peliympäristöt vaihtelevat, mutta pelit asettivat omat rajoituksensa kielen suunnittelulle. Toisaalta, käyttötarkoituksen rajaamisen ansiosta kieleen ei tarvinnut ottaa mukaan kaikkia mahdollisia ominaisuuksia, joita kirjoittaja on pohtinut.

21.4 Tutkielman kirjoittaminen on vaikeaa

Kirjoittaja on tehnyt kieltä liian pitkään. Tutkielman aiheena oman kielen hahmottaminen oli vaativa tehtävä. Tutkielma venyi sivumäärältään suureksi ja silti kirjoittajan oli jätettävä monta mieluista kohtaa pois tutkielmasta. Esimerkiksi kirjastoihin, säikeistykseen, hajautukseen, kääntämiseen ja moneen muuhun seikkaan liittyvät luvut piti jättää lopullisesta versiosta pois kokonaan, koska ne eivät olisi enää muodostaneet niin kiinteää kokonaisuutta muun osan kanssa.

Vaikka tutkielma on sivumäärältään pitkä, aivan liian moni luku on vain pieni raapaisu aiheeseen. Rasittavan moni luku oli katkottava torsoksi, koska esimerkiksi tyypityksestä on kirjoitettukin monta hyvää ja pitkää kirjaa. Turhan monen perustelun ja yksityiskohdan jäi pois, koska kirjoittaja halusi pitää sivumäärän edes alle 200:ssa.

Koska kirjoittaja oli tutkinut omaa kieltä harrastuksena ja kerryttänyt taustatietoa liian paljon. Taustatutkimus oli kuitenkin tehty liian yleisellä tasolla, jotta sitä olisi voinut suoraan soveltaa tutkielman kirjoittamiseen. Lisäksi, viitteiden hakeminen vuosien takaa luetuista materiaaleista asetti oman haasteensa työn etenemiselle.

Kieli on kyllä saanut uutta jämäkkyyttä tutkielman myötä. Koska ideoita oli pakko kirjoittaa ylös, osa suunnitteluvirheistä kielestä löytyi. Huonoja ideoita oli pakko karsia ja kielen määritykset selkeytyivät.

21.5 Tulevaisuuden ennustaminen on vaikeaa

Kirjoittaja jatkaa kielen tutkimista ja toteuttamista tutkielman jälkeenkin. Kirjoittaja uskoo, että virtuaalikoneet yleistyvät jatkossakin. Kun Wolf-kielen ensimmäinen versio on kasassa, kirjoittaja siirtyy tutkimaan siirtoa JVM:n tai .NET-alustan päälle. Eräs vaihtoehto on luoda kielestä perinteinen, käännettävä versio, joka toimii GCC-kääntäjäperheen kanssa.

On vaikea arvioida, milloin Wolf-kieli on valmis tuotantokäyttöön. Jos kielen vaatimukset eivät muutu, kieli saattaa olla valmis parin vuoden päästä. Viiden vuoden aikana kieli on toki kehittynyt eteenpäin, mutta ensimmäinenkään kääntäjä ei ole valmistunut. Kehityksen nopeuttamiseksi vaatimuksia on ollut pakko karsia ja rajoittaa. Kielen perusideat alkavat nyt olla kasassa, joten suunnittelu voi siirtyä konkreettisemmalle tasolle.

Varmaa on vain, että kielen kehitys jatkuu, jos ja vain jos se antaa mielihyvää. Tällä hetkellä vaikuttaa, että oman kielen suunnittelu on tarpeeksi antoisaa jatkossakin.

A Lähteet

[1] Alfred V. Aho, Monica S. Lam, Ravi Seth, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson, Addison Wesley, 74 Arlington Street, Suite 300, Boston, MA 02116, second edition, 2007.

[2] The Waterfall Alliance. Waterfall 2006. Saataville `http://www.waterfall2006.com/`. Viitattu 23. lokakuuta 2007, 2006.

[3] Karl-Mikael Björklid, Jonne Itkonen, and Risto Peränen. Näkökulmaohjelmointi – metaohjelmointia kuolevaisille. *Ohjelmointikielten periaatteet, päätösseminaari*, 2002.

[4] Grady Booch. *Object Oriented Design With Applications*. THe Benjamin/Cummings Publishing Company, Inc, 1991.

[5] Walter Bright. Declarations – d programming language. `http://www.digitalmars.com/d/declaration.html#AutoDeclaration`. Viitattu 22. toukokuuta 2007, 2007.

[6] Kim B. Bruce. *Foundation Of Object Oriented Languages – Types and Semantics*. The MIT Press, Cambridge, Massachusetts, 2002.

[7] Mat Buckland. *Programming Game AI by Example*. Wordware Publishing Inc., 2320 Los Rios Boulevard, Plano, Texas 75074, 2005.

[8] Timothy Budd. *A Little Smalltalk*. Addison Wesley, Reading, Massachusetts, 1987.

[9] Timothy Budd. *Introduction to Object-Oriented programming*. Addison Wesley, second edition, 1997.

[10] Valve Corporation. Counter-strike: Source. `http://www.steamgames.com/v/index.php?area=game&AppId=240`. Viitattu 22. toukokuuta 2007, 2004.

[11] Beman Dawes, David Abrahams, et al. Welcome to boost.org. Saatavilla `http://http://boost.org/index.htm`. Viitattu 09. lokakuuta 2007, 2007.

[12] Simon Denier. Traits programming with aspectj. *RSTI - L'objet*, 11(3):69–86, 2005.

[13] S. Drew, K. Gouph, and J. Ledermann. Implementing zero overhead exception handling, 1995.

[14] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.

[15] Phillip J. Eby. Java is not python, either... `http://dirtsimple.org/2004/12/java-is-not-python-either.html`. Viitattu 22. toukokuuta 2007, 2004.

[16] Python Software Foundation. Python programming language – official website. `http://www.python.org/`. Viitattu 21. toukokuuta 2007, 2007.

[17] The Squeak Foundation. Squeak. `http://www.squeak.org/`. Viitattu 15. elokuuta 2007, 2007.

[18] Amy Fowler. A swing architecture overview. *Sun Developer Network (SDN)*, 2003.

[19] Intelligent gamemaking and research in OS. Official homepage of the igios project, 2006. viitattu 14.06.2006.

[20] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[21] Markus Hof, Hanspeter Mössenböck, and Peter Pirkelbauer. Zero-overhead exception handling using metaprogramming. *Published in Proceedings SOFSEM'97*, 2002.

[22] Alan Holub. A scalable architecture for building object-oriented user interfaces. `http://www.javaworld.com/javaworld/jw-09-1999/jw-09-toolbox.html` Viitattu 28. toukokuuta 2007.

[23] Alan Holub. When it comes to good oo design, keep it simple. http://www.javaworld.com/javaworld/jw-01-2002/jw-0111-ootools.html?page=2 Viitattu 27. maaliskuuta 2007.

[24] Alan Holub. What is an object? the theory behind object-oriented user interfaces. *Java World (verkkojulkaisu http://www.javaworld.com/)*, 1999. Saatavilla http://www.javaworld.com/javaworld/jw-07-1999/jw-07-toolbox.html. Viitattu 27. maaliskuuta 2007.

[25] Alan Holub. Why extends is evil? improve your code by replacing concrete base classes with interfaces. Saatavilla http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html. Viitattu 25. kesäkuuta 2007, 2003.

[26] Timo Joensuu, Juha Lamminen, Risto Peränen, Jani Sillanpää, Marko Myllyaho, Jukka-Pekka Santanen, and Pasi Manninen. Oraakkeliprojekti. Saatavilla http://sovellusprojektit.it.jyu.fi/oraakkeli/. Viitattu 18. syyskuuta 2007, 2004.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[28] Bob Koss. Size matters. Saatavilla http://blog.objectmentor.com/articles/2006/12/21/size-matters. Viitattu 13. heinäkuuta 2007, 2006.

[29] Glyph Lefkowitz. Extending vs. embedding. Saatavilla http://www.twistedmatrix.com/users/glyph/rant/extendit.html. Viitattu 23. lokakuuta 2007, 2003.

[30] Inc. Linden Research˙ Reenskaug,trygve m. h. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. Viitattu 9. elokuuta 2007, 1979.

[31] Inc. Linden Research˙ Second life: Your world. your imagination. http://www.secondlife.com. Viitattu 9. elokuuta 2007, 2007.

[32] Robert C. (Uncle Bob) Martin. Are dynamic languages going to replace static languages? http://www.artima.com/weblogs/viewpost.jsp?thread=4639. Viitattu 22. toukokuuta 2007, 2003.

[33] Robert C. (Uncle Bob) Martin. Agile people still don't get it. http://www.butunclebob.com/ArticleS.UncleBob.AgilePeopleStillDontGetIt. Viitattu 22. toukokuuta 2007, 2006.

[34] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. *Turku Centre for Computer Science, McMaster University*, 1998. Saatavilla http://www.cas.mcmaster.ca/~emil/publications/fragile/. Viitattu 2. heinäkuuta 2007.

[35] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding Traits to (statically typed) languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, December 2005.

[36] James O'Sullivan, Tomas Lindquist Olsen, Paul D Turner, Patrick Kooman, Paul Schifferer, Olivier Delannoy, et al. Crazy eddies gui system. Saatavilla http://www.cegui.org.uk/wiki/index.php/Main_Page. Viitattu 07. lokakuuta 2007, 2007.

[37] Matti Rintala. Exceptions in remove procedure calls using c++ template metaprogramming. *Software – Practice and Experience*, 2006.

[38] Matti Rintala. Handling multiple concurrent exceptions in c++ using futures. *Advanced Topics in Exception Handling Techniques*, 2006.

[39] Matti Rintala. Kc++ thesis. *Advanced Topics in Exception Handling Techniques*, 2006.

[40] Alexander Romanovsky and Jörg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. *Advances in Exception Handling Techniques*, 2001.

[41] Alexander Romanovsky and Jörg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. *Lecture Notes in Computer Science*, Volume 2022/2001:147, 2001.

[42] Dr. Linda H. Rosenberg. Applying and interpreting object oriented metrics. Saatavilla http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html. Viitattu 11. marraskuuta 2007, 1998.

[43] Markku Sakkinen. The darker side of c++ revisited. *Structured Programming*, 13:4:155–177, 1992.

[44] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[45] Danil Shopyrin. Multimethods in c++: Finding a complete solution. Saatavilla `http://www.codeproject.com/cpp/mmcppfcs.asp`. Viitattu 07. lokakuuta 2007, 2006.

[46] Bjarne Stroustrup. Why c++ isn't just an object-oriented programming language. In *Addendum to OOPSLA'95 Proceedings. OOPS Messenger*, volume vol 6 no 4, pages 1–13, 1995.

[47] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Massachusetss, third edition, 1999.

[48] Tim Sweeney. Unrealscript language reference. `http://unreal.epicgames.com/UnrealScript.htm`. Viitattu 22. toukokuuta 2007, 1998.

[49] Antero Taivalsaari. Kevo programming language. `http://en.wikipedia.org/wiki/Kevo_(programming_language)`. Viitattu 17. elokuuta 2007, 2006.

[50] Dimitri van Heesch. Source code documentation generator tool. Saatavilla `http://www.stack.nl/~dimitri/doxygen/`. Viitattu 22. lokakuuta 2007, 2007.

[51] Kyle Wilson. Exceptions and error codes. Saatavilla `http://gamearchitect.net/Articles/ExceptionsAndErrorCodes.html`. Viitattu 9. heinäkuuta 2007, 2003.

B Oleellisia määrittelyjä

Listaus B.1: wolf::ValueDispatcher

```
1  /**
2   * General definition for typelist
3   *
4   * Basic concept is that types are linked list
5   * and one can define template that iter over each type
6   */
7  template <typename TYPE,typename NEXT=void>
8  struct TypeList {
9          /**
10          * Type defines type
11          */
12         typedef TYPE Type;
13         /**
14          * Next defines next item of the list
15          */
16         typedef NEXT Next;
17 };
18
19 /*!
20  * defines primitive types that ValueDispatcher
21  * interface requires. There will be more of these
22  */
23 typedef TypeList<Int,
24         TypeList<Real,
25         TypeList<Bool,
26         TypeList<String,
27         TypeList<Object*
28         > > > > > PrimitiveTypes;
29
30 class ValueDispatcher;
31 /*!
32 * \brief recursive template to generate
33 *  ValueDispatcher base class
34 */
35 template <typename ITEM>
```

```
36 class ValueDispatcherPart : public
37 ValueDispatcherPart<typename ITEM::Next>
38 {
39 public:
40    typedef typename ParamTrait<typename ITEM::Type>::Value
41            MyValue;
42    typedef ValueDispatcherPart<typename ITEM::Next>
43            NextPart;
44    ValueDispatcherPart() {}
45    virtual ~ValueDispatcherPart() {}
46    /*!
47     * this is used to initialize
48     *  or change information related to exporter
49     * \param index index for view
50     * \param info is metainfo relating this item
51     * \param defaultValue is default value for object
52     */
53   virtual void setInfo(int index,
54         const info::Info &info,MyValue defaultValue) = 0;
55    /*!
56     * set new value to dispatcher
57     * \param index index for view
58     * \param value is new value to Dispatcher
59     */
60    virtual void setValue(int index,MyValue value) = 0;
61    /*!
62     * get value from dispatcher
63     * \param index is index for attribute to query
64     * \param junk is needed because of C++-typing rules.
65     * \return required value from Proxy
66     * methods based on return-value)
67     */
68    virtual MyValue getValue(int index,MyValue junk) = 0;
69    using NextPart::setValue;
70    using NextPart::getValue;
71    using NextPart::importInfo;
72 };
73
74 /*!
75 * mandatory end-of-list definition to list
76 */
77 template <>
```

```
78  class ValueDispatcherPart<void> {
79  public:
80    ValueDispatcherPart() {}
81    virtual ~ValueDispatcherPart() {}
82    /*!
83     * needed by ValueDispatcherPart
84     */
85    void setValue() {}
86    /*!
87     * needed by ValueDispatcherPart
88     */
89    void getValue() {}
90    /*!
91     * needed by ValueDispatcherPart
92     */
93    void importInfo() {}
94  };
95
96  /*!
97  * \brief dispatcher is abstract interface to dispatch values
98  */
99  class ValueDispatcher : public
100 ValueDispatcherPart<PrimitiveTypes> {
101 public:
102   /*!
103    * start transaction
104    * \param self is object to be referred
105    * \param viewId is identifier for View
106    * \return ValueDispatcher that is spesified for viewId
107    */
108   virtual void beginTransaction(Object *self,int viewId) = 0;
109   virtual void endTransaction() = 0;
110   /*!
111    * start importation of metadata.
112    * Needed to set size of proxies
113    * etc ok
114    */
115   virtual void beginInfoTransaction(Object *self,
116                   int viewId,int size) = 0;
117   /*!
118    * end importation of metadata.
119    * This is needed to finalize transaction
```

```
120     */
121    virtual void endInfoTransaction() = 0;
122 };
```

Listaus B.2: wolf::Accessor

```
1  /*!
2   * \brief defintion of Accessor
3   *
4   * Accessor is used to fetch single field from
5   * C++-classes
6   * \param OBJ_TYPE extension class to Wolf
7   */
8  template <typename OBJ_TYPE>
9  class Accessor {
10 public:
11
12         Accessor() {}
13         virtual ~Accessor() {}
14
15         /*!
16          * called during initialization
17          */
18         virtual void initialize(OBJ_TYPE *self) = 0;
19
20         /*!
21          * set value from Object to ValueDispatcher
22          */
23         virtual void getValueFrom(int index,
24                     OBJ_TYPE *self,ValueDispatcher &disp) = 0;
25         /*!
26          * set value from ValueDispatcher to Object
27          */
28         virtual void setValueTo(int index,
29                     OBJ_TYPE *self,ValueDispatcher &disp) = 0;
30 };
```

Listaus B.3: "Perustelu"matalalle perintäketjulle

```
1  template <typename T>
2  class Class {
3  public:
4     void setValuesFrom(wolf::Object *self,
5                        T *native,
```

```
6                       void *instanceData,
7                       ValueDispatcher &dispatcher)
8       {
9           //call parents first
10          //since T <: (T::WolfNativeParent) this
11          //works without additional hierarchy on
12          //objects!!!
13          super->setValuesFrom(self,native,
14                               instanceData,dispatcher);
15
16          //get number of instances on parent-class
17          int index = super->numberOfVariables();
18
19          //iterate over own accessors
20          //...
21          for (it=accessors.begin();
22               it != accessors.end();
23               ++it)
24          {
25              //example simplified for
26              //native accessor only
27              Accessor<T> *accessor = *it;
28              accessor->setValue(native,dispatcher);
29          }
30      }
31
32      int numberOfVariables() {
33          return super->numberOfVariables()+accessors.size();
34      }
35
36  private:
37      Class<typename T::WolfNativeParent> *super;
38      std::vector<Accessor<T>* > accessors;
39  };
40
41  //NOTE: this is in extend-namespace
42  //so we are talking about
43  // wolf::extend::Object here!!
44  template <typename T>
45  class Object : public wolf::Object {
46  public:
47      void setValuesFrom(ValueDispatcher &dispatcher)
```

```
48      {
49          //delegate call to class
50          wolfClass->setValuesFrom(this,
51                                   native,instanceData,
52                                   dispatcher);
53
54      }
55  private:
56      //identifier of object
57      Identification *id;
58
59      //pointer to native object
60      //we want to use for extending
61      T *native;
62
63      //data required by instance-variables
64      //created by Wolf-language
65      //(managed by Class)
66      void *instanceData;
67
68      //class pointer
69      Class<T> *wolfClass;
70  };
```