Skriptikielten metapiirteet

Eero Vehmanen

Helsinki 14.1.2012

Pro gradu -seminaariesitelmän kirjallinen alustus

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

1 Johdanto

Useimpien käännettävien ohjelmointikielten rakenteet — kuten muuttujat, funktiot ja luokat — ovat konkreettisesti olemassa ainoastaan staattisesti menettäen merkityksensä ohjelman kääntämisen jälkeen. Ohjelmat käsittelevät syötteenä saamaansa dataa mutta eivät pysty vaikuttamaan itseensä, koska niillä ei ole itsestään informaatiota. Sen sijaan, jos suoritettavasta ohjelmasta on kuvaus kielen rakenteina ja ohjelmalla on siihen pääsy, ohjelma voi vaikuttaa sen omaan sisäiseen rakenteeseen ja mekanismeihin suoritusaikana. Tällöin kyse on *refleksiivisyydestä* (engl. reflection). Refleksiivisyyttä hyvin tukevissa kielissä, eli *refleksiivisissä kielissä*, saavutetaan merkittävää joustavuutta tavanomaisiin kieliin nähden. Ohjelmoija voi ohjelman kirjoittamisen ohella muokata kielen käyttäytymistä ja toteutusta, ja raja kielen suunnittelijan ja käyttäjän välillä hämärtyy [KRB91, s. 1]. Refleksiivisyys on ollut aina Brian Smithin [Smi82, Smi84] ja Pattie Maesin [Mae87a, Mae87b] merkittävästä 1980-luvulla tehdystä tutkimuksesta lähtien ohjelmointikielten tutkijoiden jatkuvan kiinnostuksen kohteena.

Refleksiivisyyttä tukevia ohjelmointikielten piirteitä kutsutaan *metapiirteiksi*. Perinteisesti eniten metapiirteitä ovat sisältäneet Lisp-perheen kielet (etenkin CLOS) ja Smalltalk. Nämä kielet eivät kuitenkaan ole saavuttaneet kovin merkittävää asemaa ohjelmistotuotannossa. Esimerkiksi suositussa Javassa metapiirteet ovat huomattavasti rajoittuneemmat; C++:ssa metapiirteet ovat vieläkin vähäisemmät. Refleksiivisten kielten asema ohjelmistotuotannossa on kuitenkin vahvistunut viime aikoina skriptikielten suosion kasvun myötä. Kaikki tärkeimmät skriptikielet sisältävät paljon metapiirteitä [Sco08, s.803].

Tämä kirjoitus perustuu tällä hetkellä keskeneräiseen pro gradu -tutkielmaan *Skriptikielten metapiirteet*, jossa perehdytään tarkemmin nimenomaan skriptikielten metapiirteisiin. Tutkielman tavoitteena on selvittää eri metapiirteiden hyötyjä ja ongelmia sekä tarjota ohjeita ja näkemyksiä niiden käyttötavoista. Refleksiivisyys on parhaimmillaan voimakas väline, joka tuo paljon mahdollisuuksia ohjelmoijalle. Samalla se tuo myös lisää vastuuta. Kun ohjelmoija saa kielen syvemmin hallintaansa, voi seurata ongelmia. Tutkielmassa rajoitutaan tarkastelemaan kolmea suosittua skriptikieltä: Pythonia, Rubya ja JavaScriptia.

Tässä kirjoituksessa ei kuitenkaan käsitellä skriptikieliä kuin hieman. Sen sijaan tässä keskitytään tutkielman alkuosaan käsittelemällä refleksiivisyyttä yleisesti. Erityisesti tarkastellaan refleksiivisyyttä oliokielissä. Olio-ohjelmointi edesauttaa pal-

jon refleksiivisyyden hyödyntämistä, ja oliokielet ovat olleet merkittävässä roolissa refleksiivisyyden kehityksessä. Oliot liittyvät olennaisesti myös skriptikielten metapiirteisiin: lähes kaikki merkittävimmät skriptikielet tukevat olio-ohjelmointia. Lopuksi esitetään lyhyt johdanto skriptikieliin.

2 Refleksiivisyys

Tässä luvussa perehdytään refleksiivisyyteen yleisellä tasolla. Selvitetään, mitä refleksiivisyys on, mitä käsitteitä siihen liittyy ja mitä hyötyä siitä on. Lisäksi selvitetään, millainen ohjelmointikielen arkkitehtuuri tukee refleksiivisyyttä hyvin ja kuinka sellaisen voi toteuttaa.

2.1 Keskeisiä käsitteitä ja termejä

Järjestelmän kykyä havainnoida ja käsitellä datana sen omaa suoritusaikaista tilaa kutsutaan *refleksiivisyydeksi*. Jotta järjestelmä voi olla refleksiivinen, sen tulee sisältää kuvaus itsestään eli *itserepresentaatio* (engl. self-representation) [Mae87b]. Järjestelmän ja sen itserepresentaation välillä on oltava *kausaalinen yhteys* (engl. causal connection): muutosten edellisessä on johdettava muutoksiin jälkimmäisessä, ja päinvastoin. Tällöin järjestelmän kuvaus itsestään on aina tarkka ja järjestelmä voi todella muokata itseään itserepresentaationsa avulla. Suoritusaikaisen tilan esittämistä datana kutsutaan *reifikaatioksi* (engl. reification) [BGW93, s. 24]. Vastakohtana reifikaatiolle on *reflektio* (engl. reflection)[1], jolla tarkoitetaan datan heijastumista järjestelmän tilaan [SoF96].

Tavallisesti tietokonejärjestelmät suorittavat laskentaa ratkaistakseen jotain kohdealueensa ongelmia. *Refleksiivisessä laskennassa* (engl. reflective computation) tehdään laskentaa järjestelmästä itsestään. Sen perimmäinen tarkoitus on jollain tavalla parantaa järjestelmän sisäistä toimintaa ja näin välillisesti edistää varsinaisen tavoitteen saavuttamista. Refleksiivistä laskentaa on esimerkiksi jonkin tilastotiedon kerääminen järjestelmän toiminnasta tai järjestelmän toiminnan optimointi itseään muokkaamalla.

Refleksiivisyydessä voidaan nähdä kaksi puolta. *Introspektio* (engl. introspection) on ohjelman tilan tarkkailemista; *intersessiolla* (engl. intercession) tarkoitetaan tilan tai sen tulkinnan tai semantiikan muokkaamista [BGW93, s. 24]. Introspektion avulla voidaan esimerkiksi oliokielissä selvittää olioiden ominaisuuksia (nimi, operaatiot, yliluokat jne.). Intersessiossa näitä ominaisuuksia muokataan, joten ohjelman käyttäytymistä voidaan muokata määrittelemällä olioiden operaatioita uusiksi tai lisäämällä uusia operaatioita.

[1] Termillä on sama englanninkielinen nimi kuin refleksiivisyydellä, mutta kyse on eri asiasta.

2.2 Refleksiivinen arkkitehtuuri

Ohjelmointikielellä on *refleksiivinen arkkitehtuuri* (engl. reflective architecture) [Mae87b], jos se on suunniteltu tukemaan refleksiivisyyttä. Tällaisen kielen tulkin täytyy antaa suoritettavalle ohjelmalle pääsy ohjelman itserepresentaatioon ja taata ohjelman ja itserepresentaation välinen kausaalinen yhteys. Reflektiivisessä arkkitehtuurissa ohjelma jakaantuu loogisesti vähintään kahteen eri tasoon. Ensimmäistä tasoa kutsutaan *perustasoksi* (engl. base-level). Sen tehtävä on suorittaa laskentaa kohdealueesta. Seuraava taso on *metataso* (engl. meta-level), joka kuvaa perustason laskentaa. Metatasolla voi edelleen olla oma metatasonsa, jolle se itse toimii perustasona. Päällekkäisistä tasoista muodostuu *refleksiivinen torni* (engl. reflective tower), joka voi olla näennäisesti ääretön. Tornin tasojen välillä on kausaalinen yhteys, joten kunkin metatasona toimivan tason tietorakenteet toimivat reifikaationa perustasonsa rakenteille ja toiminnalle, ja muutokset perustasossa reflektoituvat tälle reifikaatiolle.

Pattie Maesin [Mae87b] mukaan suurin osa refleksiivisen arkkitehtuurin sisältävistä kielistä perustuu *metasirkulaariseen tulkkiin* (engl. meta-circular interpreter). Tällainen tulkki on kirjoitettu samalla kielellä, jota se itse tulkkaa. Kielen tulkkaus muodostuu päällekkäisistä, toisiaan tulkkaavista metasirkulaarisista tulkeista. Äärettömän tornin ongelma ratkaistaan käytännössä siten, että jokin eri kielellä kirjoitettu tulkki tulkkaa metasirkulaarista tulkkia. Metasirkulaarisen tulkin avulla kausaalisen yhteyden vaatimus on helppo toteuttaa, koska järjestelmän itserepresentaatiota oikeasti käytetään järjestelmän toteuttamiseen. Metasirkulaarisen tulkin toteuttaminen vaatii, että ohjelmaa voidaan käsitellä kielen tietorakenteina.

Refleksiivistä laskentaa voidaan suorittaa myös ilman refleksiivistä arkkitehtuuria. Esimerkkinä on ohjelman jäljitysmekanismi, joka tulostaa viestin aina tietyn ehdon täyttyessä ohjelman suorituksessa. Ilman refleksiivistä arkkitehtuuria joudutaan kuitenkin joko muokkaamaan tulkkia tai vaihtoehtoisesti lisäämään ohjelmakoodin eri osiin ehdon tarkistuksia ja tulostuslauseita. Refleksiivisen arkkitehtuurin ansiosta refleksiivisestä laskennasta tulee modulaarisempaa, mikä tekee järjestelmän hallitsemisesta ja ymmärtämisestä helpompaa. Jäljitysmekanismi voidaan toteuttaa metatasolla, koskematta perustason koodiin lainkaan.

2.3 Metaohjelmointi

Refleksiivisyys liittyy laajempaan käsitteeseen *metaohjelmointi* (engl. metaprogramming). Sillä tarkoitetaan sellaisten ohjelmien kirjoittamista, jotka kuvaavat ja manipuloivat joko toisia ohjelmia tai itseään [CzE00, s. 398]. Jos kirjoituksen kohteena on ohjelma itse, kyse on refleksiivisyydestä. Refleksiivisyys on selvästi aina suoritusaikaista, mutta metaohjelmointi voi olla myös staattista. Esimerkiksi makroissa on kyse staattisesta metaohjelmoinnista, koska esikääntäjä generoi makroista koodia. Toinen esimerkki staattisesta metaohjelmoinnista on kääntäjä: se kirjoittaa ohjelmia kuvaamalla syöteohjelmat toiselle kielelle.

C++:ssa on hyvin vähän refleksiivisyyttä tukevia piirteitä, mutta se tukee parametroitujen mallien avulla tapahtuvaa metaohjelmointia [CzE00, s.397–501]. C++-ohjelma voi sisältää koodia, joka evaluoidaan jo käännösaikana. Tämä staattinen koodi voi manipuloida dynaamista koodia, mikä mahdollistaa esimerkiksi automaattisen koodin konfiguroinnin, geneerisen koodin kirjoittamisen ja koodin optimoinnin.

Termiä metaohjelmointi käytetään joskus rajatummassa merkityksessä tarkoittamaan juuri refleksiivisyyden hyödyntämistä. Näin tekee esimerkiksi Paolo Perrotta [Per10] käsitellessään Rubyn refleksiivisyyttä tukevia piirteitä eli metapiirteitä.

2.4 Refleksiivisyyden tyypit

Refleksiivisyys voidaan jakaa kahteen tyyppiin [Caz98, s. 2–3]. Tyyppi kertoo, mitä järjestelmän aspektia on mahdollista tarkkailla ja manipuloida. *Rakenteisella refleksiivisyydellä* (engl. structural reflection) tarkoitetaan ohjelman ja sen tietorakenteiden täydellistä reifikaatiota. Refleksiivisyyttä saadaan aikaan näitä rakenteita manipuloimalla.

Käyttäytymisrefleksiivisyydellä (engl. behavioral reflection) tarkoitetaan ohjelman suorituksen reifikaatiota. Sen tavoitteena on antaa ohjelmoijalle täydellinen kontrolli ohjelman suorittamaan laskentaan suoritusaikana [MDC96, s. 154]. Ohjelmoijan on siis päästävä käsiksi suorituksen kulkuun eikä pelkästään ohjelman rakenteisiin.

Ohjelmointikieli voi tukea molempia refleksiivisyyden tyyppejä, mutta käytännössä kielet eivät yleensä tarjoa molempia ainakaan kunnolla. Demersin ja Malenfantin [DeM95, s. 31] mukaan rakenteinen refleksiivisyys on helpompi toteuttaa tehokkaasti ja on siksi käytetympää.

2.5 Hyödyt

Refleksiivisyys tarjoaa eri tasojen myötä *haasteiden eriyttämistä* (engl. separation of concerns). Koska ohjelman tasot toimivat tietämättä ylemmistä tasoista mitään, refleksiivisyyteen sisältyy *läpinäkyvyyttä* (engl. transparency). Mahdollisuus lisätä ominaisuuksia perustasoon koskematta tuo *laajennettavuutta* (engl. extensibility). Nämä piirteet ovat refleksiivisyyden keskeisiä vahvuuksia [Caz08, s. i], ja niiden ansiosta on mahdollista luoda joustavia, helposti ylläpidettäviä ja muokattavia ohjelmia.

Refleksiivisyys tekee ohjelmointikielestä joustavamman ja dynaamisemman työkalun. Parhaimmillaan ohjelmoija voi muokata jopa kielen perusmekanismeja. Koska refleksiivisen kielen avulla päästään käsiksi ohjelman suoritusaikaiseen tilaan, erilaisten ohjelmointityökalujen, kuten virheenjäljittimen, toteuttaminen on helppoa.

3 Refleksiivisyys oliokielissä

Vaikka ensimmäiset oliokielet, kuten Simula ja Smalltalk-72, eivät sisältäneet juurikaan refleksiivistä laskentaa tukevia mekanismeja [Mae87b, s. 150], olio-ohjelmoinnilla on tärkeä merkitys refleksiivisyyden historiassa ja nykyhetkessä [DeM95]. Refleksiivisten kielten perustana voidaan pitää funktionaalista Lisp-perhettä. Vuonna 1958 ilmestynyt Lisp on alusta alkaen sallinut ohjelman osien manipuloinnin datana, mitä voidaan pitää alkeellisena refleksiivisyyden ilmenemisenä [DeM95, s. 29]. Varsinaisen refleksiivisyyden käsitteen ohjelmointikielten yhteyteen[2] toi Brian Smith [Smi82, Smi84] 1980-luvun alussa. Hän esitti refleksiivisen Lisp-murteen 3-Lisp, joka käytti refleksiivistä tornia.

Pattie Maes [Mae87a, Mae87b] antoi sysäyksen kiinnostukselle refleksiivisyyttä kohtaan osoittamalla refleksiivisyyden sopivan hyvin olio-ohjelmointiin. Hän esitteli 3-KRS:n — ensimmäisen olio-ohjelmointikielen, jossa oli varsinainen refleksiivinen arkkitehtuuri. Tätä ennen puhdas oliokieli Smalltalk-80 oli jo esitellyt metaluokat ja sisälsi paljon refleksiivisiä mekanismeja [Riv96]. Smalltalk on saanut Lispiltä vaikutteita ja on kirjoitettu enimmäkseen itsellään. Refleksiivisten kielten huipentumana on pidetty Common Lispin oliolaajennosta CLOS (Common Lisp Object System) ja sen *metaolioprotokollaa* (engl. metaobject protocol) [DeM95, s. 29]. Termi *metaolioprotokolla* selitetään luvussa 3.1. Gregor Kiczales, Jim des Rivieres ja Daniel G. Bobrow esittivät CLOS:n metaolioprotokollan vuonna 1991 ilmestyneessä kirjassaan *The Art of the Metaobject Protocol* [KRB91].

3.1 Refleksiivisyyden sopivuus oliokieliin

Olio-ohjelmoinnin tekniikat auttavat merkittävästi refleksiivisyyden hyödyntämistä [Mae87b, KRB91]. Oliot tarjoavat sopivan abstraktion refleksiivisen laskennan avuksi. Ne kapseloivat jonkin datan ja toiminnallisuuden ja voivat yhtä hyvin edustaa kohdealueen asioita tai toimia osana refleksiivistä laskentaa. Edellisessä roolissa toimivat oliot kuuluvat perustasolle; refleksiiviseen laskentaan osallistuvia olioita kutsutaan *metaolioiksi* (engl. metaobject). Metaoliot kuvaavat perustason olioita reifioiden niiden rakenteen ja käyttäytymisen. Kielen tarjoamaa rajapintaa metaolioihin kutsutaan metaolioprotokollaksi. Perustason ja metatason erottaminen toisistaan käy siis helposti olioiden avulla. Kapselointi piilottaa suurimman osan toteutuksesta, joten ohjelmoijan ei tarvitse tietää tarkkoja yksityiskohtia. Monimut-

[2] Aikaisemmin refleksiivisyyttä oli tietojenkäsittelytieteessä käsitelty tekoälyssä [DeM95, s. 29]

kaisten refleksiivisten mekanismien tapauksessa tämä on erityisen tärkeää. Lisäksi kapseloinnin ansiosta oliokielillä on kätevää luoda uudelleenkäytettäviä refleksiivisen laskennan kirjastoja.

Yksi tärkeä oliokielten piirre on periytymismekanismin tarjoama inkrementaalisuus. Luomalla erikoistuneita metaolioita periytymisen avulla on helppo laajentaa olemassa olevaa metatoiminnallisuutta samalla tavalla kuin perustasolla. Kielen inkrementaalinen laajentaminen voidaan yleensä tehdä rikkomatta olemassa olevaa järjestelmää, mikä lisää järjestelmän vakautta.

Olioiden avulla on mahdollista säädellä refleksiivisyyden paikallisuutta ja kestoa. Hetkellisen muutoksen olion metakäyttäytymisessä saa aikaan vaihtamalla olion metaolion toiseen[3]. Olennaista on, että perustason olioon ei tarvitse tehdä muutoksia. Paikallinen vaikutus on mahdollista saavuttaa kohdistamalla metalaskenta vain tiettyyn olioon tai oliojoukkoon.

Olio-ohjelmointi tarjoaa siis helppokäyttöiset ja joustavat puitteet refleksiivisyyden hyödyntämiselle. Toki oliokielissä on keskenään paljon eroja. Parhaiten refleksiivisyys sopii kieliin, joissa kaikki käsitteet on ilmaistavissa olioiden avulla. Tällöin kielen monia aspekteja on mahdollista käsitellä datana yhtenäisellä ja yksinkertaisella tavalla, mikä helpottaa refleksiivistä laskentaa. Vahvasti refleksiivisessä Smalltalkissa on pyritty tähän, mutta siinä luokkia ja olioita ei ole yhtenäistetty kunnolla, toisin kuin ObjVLisp-mallissa [Coi87]. Monet alkuaikojen oliokielistä — kuten Simula, C++ ja Eiffel — ovat oliomalleiltaan jäykkiä, eivätkä sovellu refleksiivisyyteen hyvin [DeM95].

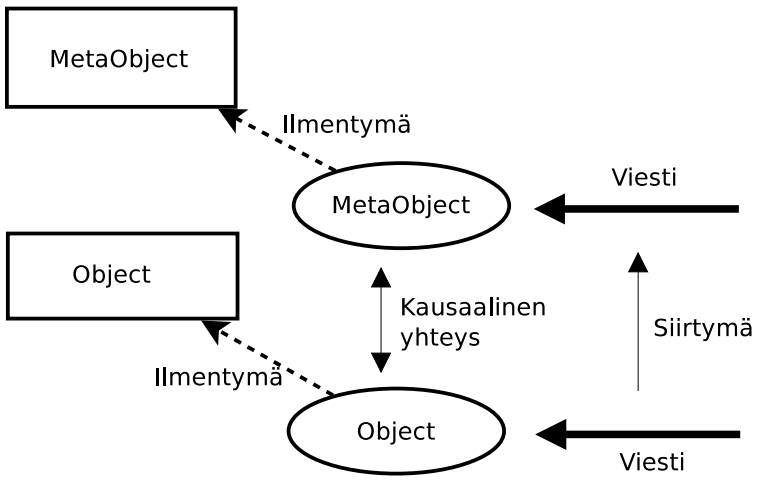
3.2 Refleksiiviset mallit

On olemassa erilaisia tapoja toteuttaa metaolioiden ja perustason olioiden muodostama järjestelmä. Walter Cazzola [Caz98] on analysoinut neljää erilaista mallia. Kaksi tärkeintä niistä ovat *metaoliomalli* (engl. metaobject model) ja *metaluokkamalli* (engl. metaclass model). Kaksi muuta mallia esitellään hyvin lyhyesti.

[3]Kaikissa refleksiivisissä kielissä metaolion vaihtaminen ei ole mahdollista. Esimerkiksi, jos metaoliona on olion luokka (ks. luku 3.2), sen vaihtaminen voi johtaa epäjohdonmukaisuuksiin.

Metaoliomalli

Metaoliomallissa [Caz98, s. 5–8] olioilla on erityisestä `MetaObject`-luokasta tai sen jälkeläisluokasta periytyvä metaolio (ks. kuva 3.1). Oliota, jolla on metaolio, kutsutaan metaolion *viiteolioksi* (engl. referent). Metaolio sieppaa viiteoliolle tarkoitetut viestit (eli metodikutsut), suorittaa tarvittaessa refleksiivistä laskentaa ja välittää viestit viiteoliolleen. Metaolio voi esimerkiksi ylläpitää tilastotietoa viiteolioon kohdistuvista kutsuista. Jokaista metodikutsua kohti se ensin kasvattaa laskuria yhdellä ja vasta sitten välittää kutsun viiteoliolleen. Refleksiivinen laskenta toteutetaan vain metaoliota muokkaamalla; viiteolio säilyy koskemattomana. Jokaisella metaoliolla voi olla myös oma metaolio, mikä johtaa virtuaalisesti äärettömään refleksiiviseen torniin. Siksi metaolio on luotava ”laiskasti” eli vasta kun se osoittautuu tarpeelliseksi [Fer89, s. 322].



Kuva 3.1: Metaoliomalli. Suorakulmiot kuvaavat luokkia, ellipsit olioita. ”Siirtymällä” (engl. shift-up action) tarkoitetaan oliolle tarkoitetun kutsun siirtymistä metaoliolle. Siirtymämekanismi voidaan toteuttaa siten, että tulkki (tai kääntäjä) tarkistaa, onko oliolla metaoliota, ja mahdollisesti kutsuu tätä olion sijaan.

Olio voi toimia usean viiteolion metaoliona. Linkki viiteoliosta metaolioon voidaan toteuttaa esimerkiksi ilmentymämuuttujan avulla. Tällöin metaolio voidaan helposti vaihtaa kyseisen muuttujan arvoa vaihtamalla. Tehokkuussyistä voi kuitenkin olla järkevää toteuttaa linkki eri tavalla ja sallia oliolle sen elinkaaren aikana korkeintaan yksi metaolio ja metaoliolle vain yksi viiteolio. Cazzola mainitsee tällaisen toteutuksen olevan kielissä CCEL, OpenC++, Iguana ja ABCL-R.

Metaoliomalli sopii hyvin käyttäytymisrefleksiivisyyteen. Metaolio sisältää vain viiteolionsa käyttäytymiseen liittyvää tietoa; viiteolion rakenne on kuvattuna sen luokassa. Oliokohtainen refleksiivinen laskenta voidaan toteuttaa luontevasti metaolioiden avulla, toisin kuin metaluokkamallissa. Luodaan vain `MetaObject`-luokan jälkeläisluokka ja asetetaan sen ilmentymä olion metaolioksi.

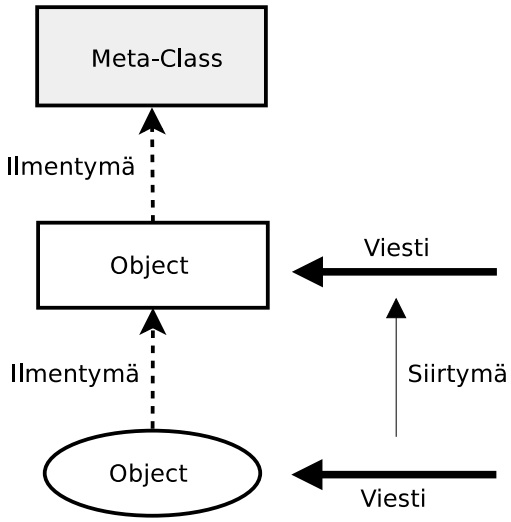
Metaoliomallin tarjoaman käyttäytymisrefleksiivisyyden myötä se sopii hyvin prototyyppikieliin. Prototyyppikielet eivät sovi kunnolla rakenteiseen refleksiivisyyteen: olion linkittäminen toiseen olioon, joka kuvaa sen rakenteen, on prototyyppiperustaisen ohjelmoinnin periaatteiden vastaista [MDC96, s. 155]. Sen sijaan oliolla voi olla metaolio, joka vastaanottaa sille tarkoitetut viestit.

Metaluokkamalli

Useimmissa oliokielissä oliolla on luokka, joka määrittelee sen. Sanotaan, että olio on luokkansa *ilmentymä* (engl. instance). Luvussa 3.1 todettiin kielen yhtenäisyyden kannalta olevan hyödyllistä, että kaikki kielen käsitteet on ilmaistavissa olioiden avulla — myös luokat. Kun luokka on olio, sekin on jonkun luokan ilmentymä. Tätä luokan luokkaa kutsutaan *metaluokaksi*[4] (engl. metaclass) [FoD99]. Edelleen metaluokalla on luokka (metametaluokka). Ilmentymien ketju jatkuu periaatteessa äärettömiin, paitsi jos siinä on sykli. Tyypillisesti metaluokkia tukevissa kielissä, jossa kaikki luokat ovat olioita, jokin luokka on itsensä ilmentymä. Näin on esimerkksi ObjVLispissä (luokka `Class`) [Coi87] ja Pythonissa (luokka `type`) [Lut09, s. 1059].

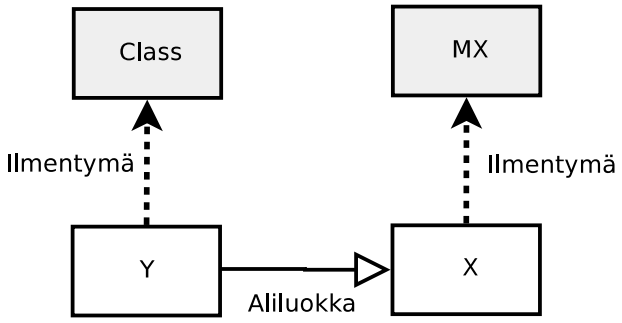
Metaluokkamallissa [Caz98, s. 4–7] perustason olion reifioiva metaolio on olion luokka ja refleksiivisen tornin muodostaa ilmentymien ketju (ks. kuva 3.2). Metaoliomallin linkkiä viiteolion ja metaolion välillä vastaa tässä mallissa olion ja sen luokan välinen linkki. Metaluokkamalli sopii rakenteelliseen refleksiivisyyteen, koska luokat määrittelevät olioiden rakenteen ja sopivat luonnostaan rakenteisen tiedon hallitsemiseen ja muokkaamiseen. Ongelmana käyttäytymisrefleksiivisyyden suhteen on yksittäisen ilmentymän metakäyttäytymisen kuvaaminen. Koska luokka toimii metaoliona, kaikilla luokan ilmentymillä on sama metakäyttäytyminen. Näin metaluokkamalli on metaoliomallia rajoittuneempi.

[4]Termiä metaluokka käytetään hieman epämääräisesti. Tarkkaan ottaen sillä tarkoitetaan olion luokan luokkaa: esimerkiksi oliolla `person` voi olla luokka `Person` ja metaluokka `MetaPerson`. Kuitenkin kun puhutaan luokan `Person` metaluokasta, tarkoitetaan usein (ja myös tässä tutkielmassa) luokkaa `MetaPerson`, vaikka tarkkaan ottaen metaluokka olisi luokan `MetaPerson` luokka (`MetaMetaPerson`).



Kuva 3.2: Metaluokkamalli. Suorakulmiot kuvaavat luokkia/metaluokkia, ellipsi oliota.

Periytyminen yhdessä erilaisten metaluokkien kanssa voi aiheuttaa *metaluokkien yhteensopivuusongelman* (engl. metaclass compatibility problem) [FoD99, s. 39–42]. Kuvassa 3.3 on esimerkki mahdollisesta yhteensopivuusongelmasta (Formanin ja Danforthin esimerkki [FoD99, s. 39] yksinkertaistettuna). Luokka `X` on metaluokan `MX` ilmentymä ja luokka `Y` metaluokan `Class` ilmentymä. Lisäksi `Y` on `X`:n aliluokka. Oletetaan, että `X`:n metodi `foo` kutsuu metaluokkansa `MX` metodia `bar`. Tämä tapahtuu `X`:stä `MX`:ään menevän linkin avulla (ei siis eksplisiittisesti `MX`:n nimellä). Luokka `Y` perii metaluokalta `X` metodin `foo`, mutta `Y`:n metaluokka `Class` ei sisällä metodia `bar`. Jos `Y`:n metodia `foo` kutsutaan, metodiin `bar` ei päästä käsiksi.



Kuva 3.3: Metaluokkien yhteensopivuusongelma

Kaksi kielen ominaisuutta aiheuttavat metaluokkien yhteensopivuusongelman: eksplisiittiset metaluokat ja moniperiytyminen. Eksplisiittisillä metaluokilla tarkoitetaan, että kielessä voi luokkamäärittelyn yhteydessä määritellä luokalle metaluokan.

Tällöin aliluokalle voi määritellä eri metaluokan kuin yliluokalle, ja edellisen esimerkin kaltainen tilanne on mahdollinen. Moniperiytymisessä luokan yliluokilla voi olla eri metaluokat, jolloin on epäselvää, mikä on ko. aliluokan metaluokka.

Metaluokkien yhteensopivuusongelma on ratkaistu metaluokkia tukevissa kielissä eri tavoin. Javassa on vain yksi metaluokka (`Class`), jota ei voi periä. Kaikki luokat ovat tämän ilmentymiä. Smalltalkissa ei ole eksplisiittisiä metaluokkia eikä moniperiytymistä. CLOS ja Python sallivat sekä eksplisiittiset metaluokat että moniperiytymisen. Ne havaitsevat yhteensopivuusongelman mahdollisuuden suoritusaikana ja heittävät poikkeuksen. Pythonissa ohjelmoija voi välttää poikkeuksen moniperiytymisen yhteydessä luomalla aliluokalle eksplisiittisesti metaluokan, joka perii kaikki yliluokkien metaluokat [MeS03b]. Forman ja Danforth [FoD99, s. 37–55] esittävät yhteensopivuusongelman ratkaisuksi mallia, jossa automaattisesti huolehditaan siitä, että luokan metaluokka on kaikkien luokan yliluokkien metaluokkien jälkeläinen.

Muut mallit

Viestireifikaatiomalli (engl. message reification model) ja *kanavareifikaatiomalli* (engl. channel reification model) perustuvat kommunikaation reifikaatioon. Ne sopivat yllä esitettyjä malleja paremmin hajautettuihin refleksiivisiin järjestelmiin, mutta ovat selvästi vähemmän käytettyjä. Viestireifikaatiomallissa jokaista metodikutsua kohti syntyy viestiolio, joka määrittää kuhunkin metodikutsuun liittyvän metalaskennan. Viestioliot voivat olla erilaisia, joten kullekin metodikutsulle voi asettaa omanlaisen käyttäytymisen. Viestiolio tuhoutuu metodikutsun jälkeen, joten metatasolla ei voi säilyttää informaatiota laskentojen välillä. Refleksiivinen torni muodostuu vain kahdesta tasosta.

Kanavareifikaatiomalli on viestireifikaatiomallin laajennos, jossa on pyritty vapautumaan sen rajoitteista. Tässä mallissa metodikutsujen ajatellaan kulkevan lähettäjä- ja vastaanottajaolion välisessä loogisessa *kanavassa* (engl. channel). Tämä looginen kanava reifioidaan kanavaolioksi, joka toimii olioiden välisenä välittäjänä lisäten metalaskentaa metodikutsujen ympärille. Samojen olioiden välille muodostetaan yhdenlainen kanava vain kerran: se säilyy, kunnes lähettäjä tai vastaanottaja tuhoutuu. Erilaisia kanavia on mahdollista luoda erilaisille samojen olioiden välisille metodikutsuille, mikä mahdollistaa refleksiivisen käyttäytymisen metodikohtaisen erikoistamisen. Kanavaolio voi säilyttää informaatiota eri kutsujen välillä, mikä on etu viestireifikaatiomalliin nähden.

3.3 Aspektiohjelmointi ja refleksiivisyys

Aspektiohjelmointi (engl. aspect-oriented programming) [Kic97] on 1990-luvun puolivälin jälkeen kehitetty ja suurta kiinnostusta herättänyt tekniikka, jolla pyritään parantamaan ohjelmien modulaarisuutta. Perinteisessä olio-ohjelmoinnissa ohjelma kuvataan luokkarakenteen avulla, joka jaottelee ohjelman olioihin jonkin tietyn näkökulman mukaisesti. Kuitenkin ohjelmat sisältävät tavallisesti paljon *läpileikkaavia ominaisuuksia* (engl. crosscutting concerns), joiden koodi jakautuu eri luokkiin. Tällaisia ominaisuuksia ovat esimerkiksi synkronointi ja lokin käsittely. Aspektiohjelmointi mahdollistaa ohjelman osittamisen useasta eri näkökulmasta [CCS04, s. 128] kapseloimalla ohjelman läpileikkaavat ominaisuudet erillisiksi modulaarisiksi *aspekteiksi* (engl. aspects). Ohjelman ylläpitäminen helpottuu ratkaisevasti, kun läpileikkaavan ominaisuuden muokkaaminen tapahtuu vain yhdessä moduulissa.

Aspektit kirjoitetaan aspekteja tukevalla kielellä, joka voi olla esimerkiksi jonkin oliokielen laajennos. Yksi käytetyimmistä on Javan laajennos AspectJ [Kic01]. Aspektit *punotaan* (engl. weave) yhteen luokkien kanssa metaohjelmoinnilla. Punominen voidaan toteuttaa eksplisiittisesti joko esikääntäjässä, kääntäjässä tai erillisellä muunnosohjelmalla [CzE00, s. 322]. Lopputuloksena on koodia (lähdekoodia tai esimerkiksi tavukoodia), jossa aspektit eivät enää ole näkyvissä erillisinä, vaan niiden koodi on sekoittunut luokkien koodiin. Osa koodingeneroinnista on mahdollista tehdä suoritusaikana sisällyttämällä kääntäjä mukaan suoritusaikaiseen järjestelmään.

Toinen vaihtoehto on toteuttaa punominen implisiittisesti ja dynaamisesti refleksiivisyyden avulla [CzE00, s. 322–323]. Tässä tavassa ei generoida koodia, mutta koska refleksiivisessä kielessä päästään suoritusaikana käsiksi kielen elementteihin ja niiden merkitystä voidaan muuttaa, tällä saavutetaan samat asiat kuin eksplisiittisellä koodin punomisella.

AspectJ:ssä (jossa punominen suoritetaan eksplisiittisesti) voidaan *neuvojen* (engl. advice) avulla määritellä koodia suoritettavaksi ohjelman *liitoskohdissa* (engl. join point). Tyypillinen liitoskohta on metodikutsu. Sen eteen ja jälkeen voidaan lisätä käyttäytymistä. Refleksiivisellä kielellä voidaan vastaava toteuttaa esimerkiksi korvaamalla halutut metodit dynaamisesti aspektikohtaista käyttäytymistä niiden ympärille lisäävillä versioilla [Sul01]. Refleksiivisyyden avulla toteutetussa aspektiohjelmoinnissa etuna on dynaamisuus ja joustavuus. Aspektikoodi voi tehdä päätöksiä suoritusaikaisten arvojen perusteella ja mukautua eri tilanteisiin. Tarvittaessa aspektikäyttäytymisen voi poistaa ohjelman suorituksen aikana. Lisäksi etuna on, että refleksiivisellä kielellä on mahdollista toteuttaa aspektiohjelmointia ilman kielen to-

teutuksen (tulkin tai kääntäjän) laajentamista. Refleksiivisyys on hyvin voimakas ja samalla myös vaarallinen väline, ja sen hallitseminen saattaa olla ohjelmoijalle rasite. Sen avulla voidaan kuitenkin tarjota ohjelmointirajapinta kehykseen, joka tarjoaa selkeämpää aspektiohjelmointia modulaaristen aspektimekanismien avulla. Esimerkiksi AspectS [Hir03] on tällainen refleksiivisyyteen perustuva aspektiohjelmointitoteutus Smalltalkilla.

Refleksiivisyydessä ongelmana on tehottomuus. AspectJ:n staattinen punominen tarjoaa AspectS:ää paremman suorituskyvyn. Lisäksi monen kielen metaolioprotokolla ei ole tarpeeksi rikas aspektiohjelmoinnin toteuttamista varten [Koj03, s. 3]. Javan refleksiivisyys ei yksin riitä AspectJ:n toteuttamiseen, koska Javassa ei ole mahdollista kääriä toiminnallisuutta metodikutsujen ympärille Smalltalkin tapaan.

Refleksiivisyyden ja aspektiohjelmoinnin suhde on hyvin läheinen. Ei ole sattumaa, että Gregor Kiczales — yksi aspektiohjelmoinnin käsitteen kehittäjistä — oli aikaisemmin kehittämässä CLOS-kielen metaolioprotokollaa. Aspektiohjelmointia voidaan pitää jopa eräänlaisena refleksiivisenä mekanismina [KoL04]. AspektJ:n aspektit käyttävät liitoskohtia reifioimaan ohjelman suoritusta ja neuvoja reflektoimaan ohjelman käyttäytymistä. Paitsi että refleksiivisyydellä voidaan emuloida aspektiohjelmointia (AspectS), aspektimekanismeilla voidaan — ainakin jossakin määrin — emuloida refleksiivisyyttä. Sergei Kojarski on kollegoineen [Koj03] toteuttanut suurimman osan Javan refleksiivisyydestä AspectJ:n avulla. Toki on huomioitava, että Javan refleksiivisyys on kaukana esimerkiksi Smalltalkin ja CLOS:n refleksiivisyydestä: se rajoittuu introspektioon [Sul01, s. 96].

Aspektiohjelmointi ja refleksiivisyys ovat saaneet ja tulevat saamaan paljon huomiota olio-ohjelmointiyhteisön keskuudessa [Caz08, s. i–ii]. Niiden tarjoamat hyödyt ovat tärkeässä roolissa ohjelmistojen evoluution kaikissa vaiheissa, suunnittelusta ja kehityksestä aina ylläpitoon asti.

4 Skriptikielten metapiirteitä

Varsinaisessa tutkielmassa tässä luvussa olisi tarkoitus perehtyä kolmen suositun skriptikielen — Pythonin, Rubyn ja JavaScriptin — keskeisimpiin metapiirteisiin sekä niiden tuomiin hyötyihin ja ongelmiin. Kunkin kielen kohdalla esiteltäisiin havainnollisia esimerkkejä metapiirteiden käytöstä. Tämä kirjoitus sisältää luvusta vain alkuosan, jossa käsitellään skriptikieliä yleisesti. Selvitetään, mitä skriptikielet ovat, miksi niiden suosio on kasvussa ja miksi ne sopivat hyvin refleksiivisyyteen.

4.1 Refleksiivisyys skriptikielissä

Tavanomaiset ohjelmointikielet (esimerkiksi C, C++ ja Java) on kehitetty itsenäisten, tehokkaiden ja turvallisten sovellusten luomista varten [Sco08, s. 649]. Tällaiset kielet pyrkivät tekemään paljon työtä käännösaikana, jotta itse ohjelman suoritus olisi tehokasta. Tehokkuuteen pyrkiminen näkyy myös matalan tason tyypeissä, jotka kuvataan tehokkaasti laitteistotasolle. Esimerkiksi kokonaislukujen, liukulukujen ja taulukkojen tehokkaaseen laskentaan kiinnitetään paljon huomiota kielten toteutuksessa. Jotta virheitä havaittaisiin mahdollisimman paljon jo käännösaikana, käytetään staattista tyypitystä. Skriptikielissä [Sco08, s. 649–718] sen sijaan kiinnitetään huomiota joustavuuteen, ilmaisuvoimaan ja sovellusten nopeaan kehitykseen.

Skriptikielten tyypillisiä piirteitä ja käyttökohteita

Skriptikielet ovat yleensä tulkattavia. Ne tosin tyypillisesti käännetään ensin jonkinlaiseen välimuotoon, kuten syntaksipuuhun tai tavukoodiesitykseen [Sco08, s. 653]. Lisäksi ne ovat usein dynaamisesti tyypitettyjä. Osa kielistä (Python, Ruby) on vahvasti tyypitettyjä, joten arvojen tyypit tarkastetaan juuri ennen niiden käyttöä; heikosti tyypitetyissä kielissä (JavaScript, Perl) arvon tyyppi muuttuu toiseksi kontekstin mukaan.

Skriptikielillä ohjelmoidaan verrattaen korkealla tasolla. Ohjelmoijan ei tarvitse huolehtia monista tehokkuuteen ja muistinhallintaan liittyvistä yksityiskohdista [Lou08, s. 23]. Korkean tason tietotyypit ovat tavallisia. Tietotyypeissä on ajateltu tehokkuuden sijaan helppokäyttöisyyyttä ja ilmaisuvoimaa: esimerkiksi hajautustaulut ovat tärkeässä asemassa monessa skriptikielessä [Ous98, s. 26]. Muuttujia ei tarvitse esitellä monissa kielissä. Pääsy käyttöjärjestelmän mekanismeihin, ja siten esimerkiksi tiedostojen käsittely, on tehty helpoksi. Lisäksi skriptikielet tukevat hyvin säännöl-

lisiä lausekkeita ja merkkijonojen käsittelyä. Ilmaisu on siis lyhyttä ja ytimekästä, mikä näkyy koodirivien pienenä määränä. Tämä mahdollistaa tavanomaisiin kieliin verrattuna merkittävästi nopeamman sovellusten kehityksen.

Ohjelmistotuotannossa skriptikieliä voidaan käyttää paitsi itsenäisinä sovelluskielinä, myös yhdistämään eri kielillä tehtyjä komponentteja. Tavanomaisilla kielillä voidaan luoda uudelleenkäytettäviä, tehokkuutta vaativia (mahdollisesti monimutkaisia algoritmeja sisältäviä) itsenäisiä komponentteja, jotka yhdistetään nopean kehityksen mahdollistavilla ja selkeillä skriptikielillä [Ous98]. Tämä on ollut joidenkin yleiskäyttöisten skriptikielten alkuperäinen tarkoitus; niitä kutsutaan joskus ”liimakieliksi” (eng. glue language) [Sco08, 650]. Lisäksi skriptikieliä voidaan käyttää laajentamaan toisella kielellä kirjoitettua sovellusta. Tällaisen sovelluksen tulee tarjota skriptikielen tulkki sekä ”koukkuja”, joiden avulla skripteistä päästään käsiksi sovellukseen [Sco08, s. 677]. Nykyään skriptikieliä käytetään paljon dynaamisten verkkosivujen luomiseen, sekä selain- että palvelinpuolella.

Historia ja tulevaisuus

Modernien skriptikielten esi-isiä ovat 1970-luvulla kehitetyt komentotulkit sekä tekstin prosessointiin ja raporttien generointiin tarkoitetut työkalut [Sco08, s. 650]. Näistä kehittyivät IBM:n kehittämä Rexx vuonna 1979 ja Larry Wallin kehittämä Perl vuonna 1987. 1990-luvun jälkimmäiseltä puoliskolta alkoi skriptikielten suosion voimakas nousu. Suuri syy tähän oli WWW:n kasvu. Perl saavutti suuren suosion palvelinohjelmoinnissa (myöhemmin PHP:stä tuli suositumpi); JavaScript yleistyi selaimissa. Skriptikielten merkitystä lisäsi ohjelmoijan tuottavuuden korostuminen [Lou08, s. 23]. Muun muassa tietokoneiden suorituskyvyn kasvun ja WWW:n tuoman tietoliikenteen myötä ohjelmien tehokkuus ei ollut enää pullonkaula. Sen sijaan vaadittiin ohjelmien nopeaa kehittämistä ja muokkausta muuttuvien vaatimusten mukaisiksi.

Skriptikielten kehitys on ollut nopeaa. Uuden skriptikielen kehittäminen vaatii selvästi lyhyemmän työmäärän kuin C#:n tai Javan kaltaisen käännettävän kielen kehittäminen [Sco08, s. 655]. WWW:n jatkuva kasvu ja avoimen lähdekoodin yhteisöjen dynaamisuus ovat myös edesauttaneet tätä kehitystä. Nopean kehityksen ansiosta skriptikielet ovat pystyneet hyödyntämään monia voimakkaita ja tyylikkäitä mekanismeja kuten ensimmäisen luokan funktioita, roskien keruuta, korkean tason tietotyyppejä ja iteraattoreita. Michael L. Scottin [Sco08, s. 717–718] mukaan skriptikielet saattavat hyvin tulla hallitsemaan tämän vuosisadan ohjelmointia syrjäyt-

täen tavanomaiset käännettävät kielet erikoistarkoituksiin käytettäviksi työkaluiksi. Yleiskäyttöisistä skriptikielistä varsinkin Pythonin ja Rubyn suosio on kasvussa.

Refleksiivisyys

Refleksiivisyys sopii luontevasti skriptikieliin [Sco08, s. 803–804]. Tämä on seurausta skriptikielten tulkattavuudesta ja dynaamisesta tyypityksestä. Suoritusaikaisten tyyppitarkistusten mahdollistaminen vaatii symbolitaulun yksityiskohtaisten tietojen säilyttämistä suoritusaikaan asti. Tulkattavissa toteutuksissa nämä ovat valmiiksi saatavilla. Javassa ja C#:ssa refleksiivisyyden toteutus on selvästi monimutkaisempaa. Refleksiivisyyden hyödyistä kertoo jotain se, että näiden kielten suunnittelijat ovat joka tapauksessa katsoneet tarpeelliseksi ottaa refleksiivisyyttä mukaan.

Kaikki tärkeimmät skriptikielet — Perl, PHP, Tcl, Python, Ruby ja JavaScript — sisältävät runsaasti metapiirteitä. Ne tukevat (Tcl:ää lukuunottamatta) olioparadigmaa [Sco08, s. 710], mikä on osaltaan hyödyksi refleksiivisyydessä. Python ja Ruby ovat eksplisiittisesti olioperustaisia, ja niissä luokat ovat olioita. JavaScript ei sisällä luokkia, vaan käyttää periytymiseen prototyyppejä.

Lähteet

BGW93 Bobrow, D. G., Gabriel, R. G. ja White, J. L., CLOS in Context – The Shape of the Design Space. Teoksessa *Object-oriented programming*, Paepcke, A., toimittaja, MIT Press, Cambridge, 1993, sivut 29–61.

Caz08 Cazzola, W. et al., *Proceedings of the 5th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'08)*. Fakultät für Informatik, Otto-von-Guericke-Universität, Magdeburg, 2008.

Caz98 Cazzola, W., Evaluation of object-oriented reflective models. *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, Brussels, Belgium, 1998.

CCS04 Cazzola, W., Chiba, S. ja Saake, G., Software evolution: A trip through reflective, aspect, and meta-data oriented techniques. Teoksessa *Object-Oriented Technology. ECOOP 2004 Workshop Reader*, Malenfant, J. ja Ostvold, B., toimittajat, osa 3344 sarjasta *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2005, sivut 118–132.

CzE00 Czarnecki, K. ja Eisenecker, U. W., *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, USA, 2000.

Coi87 Cointe, P., Metaclasses are first class: The objvlisp model. *ACM SIGPLAN Notices*, 22,12(1987), sivut 156–162.

DeM95 Demers, F.-N. ja Malenfant, J., Reflection in logic, functional and object-oriented programming: a short comparative study. *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 1995, sivut 29–38.

FoD99 Forman, I. R. ja Danforth, S. H., *Putting metaclasses to work*. Addison-Wesley, USA, 1999.

Fer89 Ferber, J., Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices - Special issue: Proceedings of the 1989 ACM OOPSLA conference on object-oriented programming*, 24,10(1989), sivut 317–326.

Hir03 Hirschfeld, R., AspectS - aspect-oriented programming with squeak. *Revised Papers from the International Conference NetObjectDays on*

Objects, Components, Architectures, Services, and Applications for a Networked World, NODe '02, London, UK, UK, 2003, Springer-Verlag, sivut 216–232.

Kic97 Kiczales, G. et al., Aspect-oriented programming. Teoksessa *ECOOP'97 – Object-Oriented Programming*, osa 1241 sarjasta *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1997, sivut 220–242.

Kic01 Kiczales, G. et al., An overview of AspectJ. *ECOOP '01 Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001, sivut 327–353.

KRB91 Kiczales, G., des Rivieres, J. ja Bobrow, D. G., *The Art of the Metaobject Protocol*. MIT Press, Cambridge, 1991.

KoL04 Kojarski, S. ja Lorenz, D. H., AOP as a first class reflective mechanism. *OOPSLA '04 Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, sivut 216–217.

Koj03 Kojarski, S., Lieberherr, K., Lorenz, D. H. ja Hirschfeld, R., Aspectual reflection. *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

Lou08 Loui, R. P., In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41,7(2008), sivut 22–26.

Lut09 Lutz, M., *Learning Python*. O'Reilly, USA, neljäs painos, 2009.

Mae87a Maes, P., *Computational Reflection*. Väitöskirja, Vrije Universiteit Brussel, 1987.

Mae87b Maes, P., Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22,12(1987), sivut 147–155.

MDC96 Malenfant, J., Dony, C. ja Cointe, P., A semantics of introspection in a reflective prototype-based language. *LISP and Symbolic Computation*, 9,2(1996), sivut 153–179.

MeS03b Mertz, D. ja Simionato, M., Metaclass programming in Python, part II, 2003. `http://www.ibm.com/developerworks/linux/library/l-pymeta2/`. [13.4.2011]

Ous98 Ousterhout, J., Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31,3(1998), sivut 23–30.

Per10 Perrotta, P., *Metaprogramming Ruby: Program Like the Ruby Pros.* Pragmatic Bookshelf, 2010.

Riv96 Rivard, F., Smalltalk: a reflective language, 1996. `http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/rivard/rivard.html`. [18.4.2011]

Sco08 Scott, M. L., *Programming Language Pragmatics.* Morgan Kaufmann, USA, kolmas painos, 2008.

SoF96 Sobel, J. M. ja Friedman, D. P., An introduction to reflection-oriented programming, 1996. `http://www.cs.indiana.edu/~jsobel/rop.html`. [13.4.2011]

Smi82 Smith, B., *Reflection and Semantics in a Procedural Language.* Väitöskirja, Massachusetts Institute of Technology, 1982.

Smi84 Smith, B., Reflection and semantics in lisp. *POPL '84 Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1984, sivut 23–35.

Sul01 Sullivan, G. T., Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44,10(2001), sivut 95–97.