View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Helsingin yliopiston digitaalinen arkisto

Hienojakoiset staattiset pääsyrajoitteet Java-ohjelmien osien eristyneisyyden takaajina

Petri Vuorio

Pro gradu -tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 9.5.2016

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author

Petri Vuorio

Työn nimi — Arbetets titel — Title

Hienojakoiset staattiset pääsyrajoitteet Java-ohjelmien osien eristyneisyyden takaajina

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	9.5.2016	92 sivua + 6 liitesivua

Tiivistelmä — Referat — Abstract

Java-kielessä ohjelmaelementti voi olla saatavissa kaikkialla, vain aliluokissa sekä esittelevässä pakkauksessa, vain esittelevässä pakkauksessa tai vain esittelevässä luokassa. Saatavuustasojen jaottelua voidaan pitää karkeana. On olemassa tapauksia, joissa ohjelmaelementin saatavuuden täytyy olla laajempi kuin saatavuuden todellinen tarve vaatisi. Seurauksena abstraktioiden sisäiset toteutusyksityiskohdat ovat näkyvissä laajemmalle kuin on tarpeen, mikä yleensä johtaa ohjelman sisäisten riippuvuuksien lisääntymiseen heikentäen sen modulaarisuutta.

Optimaalisen saatavuuden mahdollistamiseksi esitän ratkaisuksi kytkettävää pääsynvalvontaa. Se soveltaa kytkettävien tyyppijärjestelmien ideaa pääsynvalvontaan. Kytkettävä tyyppijärjestelmä on ohjelmointikielestä erillinen staattinen tyyppijärjestelmä, jonka toteutus on kytkettävissä tarpeen mukaan ohjelmointikieleen valinnaisena laajennoksena.

Tutkielmassa esitän suunnitelmat kytkettävästä pääsynvalvontajärjestelmästä pääsymääreineen sekä kytkettävän pääsynvalvontakehyksen toteutuksesta Javalle. Osoitan kytkettävän pääsynvalvonnan mahdollistavan optimaalisen saatavuuden soveltamalla suunnittelemaani pääsynvalvontajärjestelmää tapauksiin, joissa Javan pääsynvalvontajärjestelmä osoittautuu karkeaksi. Kytkettävän pääsynvalvontakehyksen suunnitelman pohjalta rakennettu prototyyppi osoittaa ratkaisun olevan käytännössä toteutettavissa.

Suunnitellut pääsymääreet hyödyntävät monipuolisesti erilaisia saatavuusperusteita. Yksinkertaisen perusarkkitehtuurin ansiosta eri pääsymääreiden yhteiskäyttö on ongelmatonta. Kytkettävä pääsynvalvonta osoittautuu ilmaisuvoimaiseksi ja valinnaisuutensa ansiosta joustavaksi välineeksi täydentämään ohjelmointikielen omaa pääsynvalvontaa.

ACM Computing Classification System (CCS):

Software and its engineering~Constraints
Software and its engineering~Object oriented languages

Avainsanat — Nyckelord — Keywords

kytkettävä pääsynvalvonta, Java, tiedon piilottaminen, olio-ohjelmointikielet

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Ohjelmointikielten rakennuspalikoita **4**

2.1 Ohjelman käännösprosessi 4

2.2 Tyyppijärjestelmät 10

2.3 Javan keskeistä tyyppikäsitteistöä 12

3 Tiedon piilottamisen mekanismit **14**

3.1 Tiedon piilottaminen 14

3.2 Staattisten oliokielten pääsynvalvontamekanismeja 16

3.3 Pääsynvalvontamekanismien eri tyypit 25

4 Valinnaiset tyyppijärjestelmät **32**

4.1 Staattisten tyyppijärjestelmien vahvuuksia ja ongelmia 32

4.2 Ratkaisumalleja tyyppijärjestelmien ongelmiin 34

4.3 Javan valmiudet kytkettävän tyyppijärjestelmän toteuttamiseen 36

4.4 Kytkettävä tyyppijärjestelmäkehys Checker Framework 40

5 Kytkettävä pääsynvalvonta Javaan **43**

5.1 Javan pääsynvalvonnan karkeus 43

5.2 Kytkettävä pääsynvalvontajärjestelmä 47

5.3 Pääsymääreet 51

5.4 Kytkettävä pääsynvalvontakehys Javaan 56

5.5 Optimaalinen pääsynvalvonta 64

5.6 Kokemuksia ja havaintoja 75

6 Yhteenveto **82**

Lähteet **85**

Liitteet

1 Kytkettävän pääsynvalvontajärjestelmän yksityiskohtia

2 Esimerkkejä kytkettävän pääsynvalvontajärjestelmän käytöstä

1 Johdanto

Ohjelmointikielten voidaan yleisesti katsoa koostuvan kolmesta osasta: syntaksi, semantiikka ja toteutus. Kielen syntaksi määrittelee ohjelman muodollisen oikeellisuuden. Syntaksisäännöt esimerkiksi määräävät, millaisen merkin tai sanan tulee seurata toista. Kielen semantiikka määrittelee ohjelman merkityksen. Semanttiset säännöt määräävät esimerkiksi, mihin jokin tunnus viittaa, miten eri tyyppisiä arvoja käsitellään, tai missä ohjelman osassa toinen ohjelman elementti on saatavissa. Kielen toteutukseen kuuluvat kielestä riippuen kääntäjä ja suoritusympäristö. Kielen toteutus tarkastaa ohjelman syntaktisen ja semanttisen oikeellisuuden ja muuntaa syntaktisesti ja semanttisesti virheettömän ohjelman suoritettavaan muotoon.

Ohjelmointikielet ovat karkeasti jaoteltavissa staattisiin ja dynaamisiin kieliin. Staattisten kielten ohjelmat käännetään laitteiston ymmärtämäksi konekieleksi ennen ohjelman suoritusta. Staattisten kielten syntaksi ja semantiikka ovat tarkastettavissa kokonaan käännösaikaisesti. Dynaamiset kielet tulkataan vasta suoritusaikaisesti konekielelle, minkä yhteydessä suoritetaan syntaktiset ja semanttiset tarkastukset. Käytännössä jako staattisiin ja dynaamisiin kieliin staattisten ei ole yksikäsitteinen, vaan kielissä ja niiden toteutuksissa on lähes aina sekä staattisia että dynaamisia piirteitä. Esimerkiksi alla mainitut pääsynvalvonta ja tyyppijärjestelmä saattavat olla toteutettu kielen kääntäjään, mutta myös suoritusympäristöön.

Varhaisissa ohjelmointikielissä mikä tahansa ohjelman osa oli kutsuttavissa ja kaikki ohjelman data oli käsiteltävissä missä tahansa ohjelman sisällä. Nykyisissä ohjelmointikielissä on tyypillisesti jonkinlainen pääsynvalvontajärjestelmä, mikä mahdollistaa ohjelman modularisoinnin ja sen osien abstrahoinnin, kun ohjelman osien yksityiskohdat ovat piilotettavissa muilta osilta. Tyypillisesti ohjelmaelementit voivat olla julkisia, eli saatavissa kaikkialla ohjelmassa, tai yksityisiä, eli saatavissa ainoastaan rajoitetun ohjelmayksikön sisällä. Näiden lisäksi kielet saattavat mahdollistaa pääsyn rajoittamisen esimerkiksi moduulin tai muun rakenteen sisään tai olio-ohjelmointikielissä olion perintähierarkian mukaan.

Yleisesti ohjelmointikielet ovat tyypitettyjä: jokainen arvo ja tietorakenne on jonkin tyyppinen. Kielen tyyppijärjestelmä määrittelee tyyppien suhteet toisiinsa ja pyrkii estämään tyyppivirheet ohjelmissa, esimerkiksi tyypille sopimattoman operaation suorituksen. Kieleen sidotussa staattisessa tyyppijärjestelmissä on paljon etuja, mutta myös ongelmiakin. Yhtenä ratkaisuna ongelmiin on ehdotettu valinnaisia tyyppijärjestelmiä. Valinnaiset tyyppijärjestelmät mahdollistavat staattisen tyyppijärjestelmän toteuttamisen dynaamisiin kieliin, mutta myös staattisesti tyypitettyjen kielten tyyppijärjestelmän rikastamisen ilman tarvetta muuttaa kielen määrittelyä. Valinnaiset tyyppijärjestelmät voivat olla kytkettäviä, jolloin niitä voidaan sovelluskohtaisesti liittää kieleen tarpeen mukaan jopa useita.

Tutkielmassa perehdytään staattisten olio-ohjelmointikielten ja erityisesti Java-kielen pääsynvalvontaan. Java jaottelee saatavuuden neljään tasoon, joiden mukaan ohjelmaelementti

voi olla saatavissa kaikkialla, vain aliluokissa sekä esittelevässä pakkauksessa, vain esittelevässä pakkauksessa tai vain esittelevässä luokassa. Monessa muussa ohjelmointikielessä jaottelu on samantapainen. Väitän jaottelua karkeaksi. Karkeuden vuoksi ohjelmoija törmää tilanteisiin, joissa kohteen saatavuus on asetettava laajemmaksi kuin mitä todellinen saatavuuden tarve olisi. Hienojakoisempi pääsynvalvonta mahdollistaisi optimaalisen saatavuuden ja ohjelman osien eristyneisyyden. Esitän tutkielmassa väitettä tukevia tapauksia.

Ratkaisuksi hienojakoisemman pääsynvalvonnan saavuttamiseksi esitän kytkettävää pääsynvalvontaa toteutettavaksi Javaan. Sen ajatuksena on soveltaa kytkettävien tyyppijärjestelmien ideaa pääsynvalvontaan. Tutkielman keskeinen tavoite on osoittaa, että kytkettävä pääsynvalvonta on toteutettavissa Javaan, ja että se pystyy ratkomaan ne Javan pääsynvalvonnan karkeuteen liittyvät ongelmat, jotka edellä mainituissa esimerkkitapauksissa on esitetty.

Tavoitteen saavuttamiseksi olen laatinut suunnitelmat kytkettävästä pääsynvalvontajärjestelmästä pääsymääreineen sekä kytkettävän pääsynvalvontakehyksen toteutuksesta. Pääsynvalvontajärjestelmän suunnittelun taustaksi olen kartoittanut tunnettujen käännettävien olio-ohjelmointikielten pääsynvalvontajärjestelmiä, ja analysoinut niiden yhteisiä ja erottavia piirteitä. Osoitan kytkettävän pääsynvalvonnan mahdollistavan optimaalisen saatavuuden ja ohjelman osien eristämisen toisistaan soveltamalla suunnittelemaani pääsynvalvontajärjestelmää esittelemissäni tapauksissa, joissa Javan pääsynvalvontajärjestelmä osoittautuu karkeaksi.

Suunnittelemastani kytkettävästä pääsynvalvontakehyksestä olen toteuttanut prototyypin Javan versiolle 8. Se kykenee suorittamaan tutkielmassa määriteltyjen pääsymääreiden mukaiset semanttiset tarkastukset. Prototyypissä olen hyödyntänyt kytkettävän tyyppijärjestelmäkehyksen Checker Frameworkin tarjoamia työkaluja. Se integroituu Checker Frameworkin tapaan Java-kääntäjään ilman tarvetta muunnellulle kääntäjälle tai erikseen suoritettaville työkaluille. Prototyyppi osoittaa, että kytkettävän pääsynvalvonnan toteuttaminen Javaan on mahdollista.

Seuraavassa luvussa 2 perehdytään yleisellä tasolla ohjelmointikielen kolmeen peruskomponenttiin: syntaksiin, semantiikkaan ja toteutukseen ohjelman käännösprosessin näkökulmasta. Ohjelmointikielen semanttisista ominaisuuksista tutustutaan erityisesti tyyppeihin ja tyyppijärjestelmiin. Luvun päätteeksi kerrataan tutkielman tarkastelun kohteena olevan Javan keskeistä tyyppikäsitteistöä.

Luvussa 3 tutustutaan tiedon piilottamiseen ja ohjelmointikielten pääsynvalvontaan. Luvussa kartoitetaan tunnetuissa ohjelmointikielissä käytettyjä pääsynvalvontamekanismeja, ja analysoidaan, mitä yhteisiä ja eroavia piirteitä eri ohjelmointikielten pääsynvalvontamekanismeissa on.

Tutkielman luvussa 4 esitellään valinnaiset tyyppijärjestelmät. Luvun aluksi kartoitetaan ohjelmointikielten tyyppijärjestelmien vahvuuksia ja ongelmakohtia eritoten ohjelmointi-

kieliin pakollisena kuuluvien staattisten tyyppijärjestelmien osalta. Seuraavaksi esitellään valinnaisten tyyppijärjestelmien idea, ja pohditaan, kuinka valinnaiset ja kytkettävät tyyppijärjestelmät pyrkivät vastaamaan pakollisten tyyppijärjestelmien ongelmiin. Luvun loppupuoliskolla esitellään Javan annotaatiot ja annotaatioiden käännösaikainen prosessointimekanismi, jotka tekevät kytkettävän tyyppijärjestelmän toteuttamisen Javaan mahdolliseksi. Lopuksi tarkastellaan, kuinka Checker Framework-tyyppijärjestelmäkehys toteuttaa Javaan kytkettävän tyyppijärjestelmän.

Luku 5 esittelee kytkettävän pääsynvalvonnan kokonaisuudessaan. Aluksi kartoitetaan Javan pääsynvalvonnan karkeaa jaottelua eri saatavuustasoihin, ja esitellään tapauksia, joissa tiedon piilottaminen ei toteudu optimaalisesti. Seuraavaksi esitellään kytkettävän pääsynvalvonnan idea sekä suunniteltu kytkettävä pääsynvalvontajärjestelmä pääsymääreineen. Kytkettävälle pääsynvalvontajärjestelmälle esitellään suunnitelma sen toteutuksesta, kytkettävästä pääsynvalvontakehyksestä, sekä kuvaillaan suunnitelman pohjalta toteutettua prototyyppiä. Luvun loppupuolella sovelletaan suunniteltua kytkettävää pääsynvalvontajärjestelmää luvun alussa esitettyihin tapauksiin, ja osoitetaan, että kytkettävä pääsynvalvonta mahdollistaa optimaalisen tiedon piilottamisen kyseisissä tapauksissa. Päätteeksi pohdiskellaan tutkielmaprosessin aikana syntyneitä kokemuksia ja havaintoja kytkettävästä pääsynvalvonnasta sekä kartoitetaan aiheeseen liittyviä jatkokehityskohteita.

2 Ohjelmointikielten rakennuspalikoita

Ohjelmointikielten käsitetään tavallisesti koostuvan kolmesta peruselementistä: syntaksista (syntax), semantiikasta (semantics) ja toteutuksesta (implementation) [Fer14, s. 5–6]. Ohjelmointikielen syntaksi määrittelee, kuinka ohjelman eri elementit, kuten esittelyt ja lausekkeet, tulee esittää muodostaakseen muodollisesti oikeellisen ohjelman. Kielen semantiikka määrittelee ohjelmien merkityksen. Semanttiset säännöt asettavat rajoituksia, millaisessa yhteydessä erilaiset kielen syntaksin sallimat konstruktiot ovat mahdollisia. Kielen toteutus käsittää tyypillisesti tyypillisesti kääntäjän ja suoritusympäristön.

Tässä luvussa tutustutaan ohjelmointikielten rakennuspalikoihin ohjelman käännösprosessin näkökulmasta. Lisäksi perehdytään tyyppeihin ja tyyppijärjestelmiin, sekä kerrataan lopuksi Java-kielen keskeistä tyyppikäsitteistöä.

2.1 Ohjelman käännösprosessi

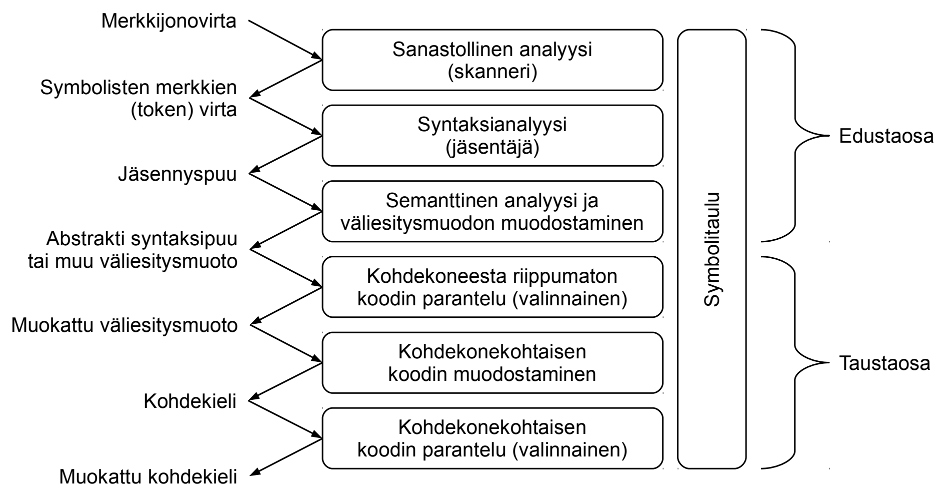
Ohjelmointikielen kääntäjät tyypillisesti jakautuvat toiminnallisesti edusta- ja taustaosaan, joista kumpikin sisältää useampia eri vaiheita. Seuraavaksi tarkastellaan käännöksen vaiheita kokonaisuutena ja perehdytään hieman tarkemmin kääntäjän edustaosan toimintaan.

Käännöksen vaiheet

Ohjelmointikielen kääntäjän toiminta jakautuu useaan erilaiseen vaiheeseen. Kuva 2.1 havainnollistaa jäljempänä kuvattuja kääntäjään tyypillisesti kuuluvaa neljää perusvaihetta: sanastollinen analyysi (lexical analysis), syntaksianalyysi (syntax analysis), semanttinen analyysi (semantic analysis) ja kohdekonekohtaisen koodin muodostaminen [Sco09, s. 25–35][Fer14, s. 7–9]. Kuvassa esitetään lisäksi kaksi valinnaista koodin optimointivaihetta. Vaiheet on listattu kuvan keskellä, ja kunkin vaiheen syöte ja tuloste on listattu kuvan vasemmassa laidassa.

Kääntäjän toiminta jaetaan tyypillisesti kahteen osaan: edustaosa (front end) ja taustaosa (back end). Symbolitaulu (symbol table) pitää kirjaa ohjelmassa käytetyistä tunnisteista koko käännösprosessin ajan. Kuvassa esitetyt käännöksen kolme ensimmäistä vaihetta muodostavat edustaosan. Kahdessa ensimmäisessä vaiheessa, sanastollinen analyysi ja syntaksianalyysi, kääntäjä analysoi käännettävän ohjelman syntaktisesti, eli tarkastaa, että ohjelma on muodollisesti oikein. Kolmannessa vaiheessa, semanttinen analyysi, kääntäjä analysoi ohjelmaelementtien merkityksen tarkastaen, että ohjelma noudattaa kielen semanttisia sääntöjä.

Kuvan 2.1 viimeiset kolme vaihetta muodostavat kääntäjän taustaosan. Sen pääasiallinen tarkoitus on kohdekonekohtaisen koodin muodostaminen edustaosan tuottamasta väliesitysmuodosta. Sitä ennen taustaosa voi valinnaisesti parannella kohdekoneesta riippumatonta



Kuva 2.1: Ohjelmointikielen kääntäjän käännösprosessin perusvaiheet. Mukailtu lähteestä [Sco09, s. 26].

väliesitysmuotoa ja kohdekoodin muodostamisen jälkeen parannella kohdekonekohtaista koodia.

Käytännössä ohjelmointikielen kääntäjä saattaa jakaa käännösprosessin paljon edellä mainittua useampaan eri vaiheeseen. Vaiheiden määrä riippuu käännettävästä kielestä sekä kääntäjän toteutuksesta. Esimerkiksi Scala-kääntäjän version 2.11.8 käännösprosessi sisältää 25 erilaista vaihetta[1]. Kaikki vaiheet ovat kuitenkin tyypillisesti luokiteltavissa kuuluvaksi johonkin edellä mainituista neljästä perusvaiheesta ja kahdesta optimointivaiheesta.

Ohjelman syntaksin analysointi

Käännösprosessin aluksi skanneriksi (scanner) kutsuttu komponentti lukee lähdekoodin merkkijonovirtana (character stream) tehden sille sanastollisen analyysin [Sco09, s. 41–42]. Skanneri on teoreettisesti deterministinen äärellinen automaatti (deterministic finite automaton, DFA), joka ryhmittelee lähdekoodin merkkijonovirran symbolisiksi merkeiksi (token) säännöllisinä lausekkeina (regular expression) esitettyjen sääntöjen mukaisesti. Kukin symbolinen merkki vastaa jotakin ohjelmointikielen sanaa (word) [Sco09, s. 43–44]. Sana voi olla esimerkiksi avainsana (keyword), operaattori (operator), literaali (literal) tai tunniste (identifier). Sanastollisen analyysin lopputuloksena syntyy symbolisten merkkien virta (token stream).

Jäsentäjä (parser) suorittaa skannerin tuottamalle symbolisten merkkien virralle syntaksianalyysin [Sco09, s. 67–70]. Analyysin aikana se järjestää symboliset merkit yhteydettömän

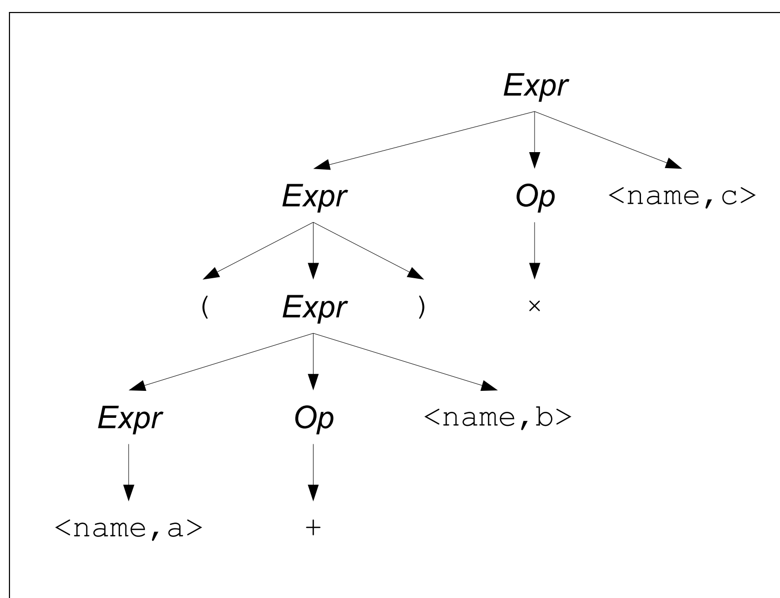
[1] Scala-kääntäjä listaa käännösprosessin vaiheet komennolla `scalac -Xshow-phases`. Komennossa annettu optio on dokumentoitu kääntäjän ohjeessa, joka sisältyy Scalan jakelupakettiin.

kieliopin (context-free grammar, CFG) mukaan jäsennyspuuksi (parse tree). Yhteydetön kielioppi on joukko rekursiivisia sääntöjä, jotka määrittävät, kuinka jäsennyspuu muodostetaan.

Listauksessa 2.1 on esimerkki yhteydettömästä kieliopista, joka mahdollistaa muuttujien väliset yhteen-, vähennys-, kerto- ja jakolaskut sekä osalaskutoimitusten ryhmittelyn sulkuja käyttäen. Kielioppi on esitetty niin kutsutussa Backus-Naur-muodossa, joka on peräisin Algol 60-kielen [BBG+63] määrittelystä. Kielen jokaista sääntöä sanotaan tuotokseksi (production) [Sco09, s. 46–48]. Kussakin säännössä nuolen vasemmalla puolella on muuttuja (variable, nonterminal) ja oikealla puolella joukko muotoja, jonka muuttuja voi saada. Vaihtoehtoiset muodot on erotettu toisistaan pystyviivalla. Sulkumerkit on kyseisessä esimerkissä alleviivattu, mikä tarkoittaa, että sulkumerkit tulkitaan sellaisenaan eikä muodon sisäiseksi ryhmittelyksi. Kuvassa 2.2 on esimerkki jäsennyspuusta, joka on muodostettu lausekkeesta `(a+b)×c` listauksen 2.1 kieliopin mukaisesti. Siitä on nähtävissä, kuinka jokainen kieliopin vasemman puolen muuttuja laajentuu oikean puolen muodoksi.

Listaus 2.1: Esimerkki yksinkertaisesta muuttujien välisen laskutoimituksen mahdollistavan kielen yhteydettömästä kieliopista. Lähde: [CT12, s. 89]

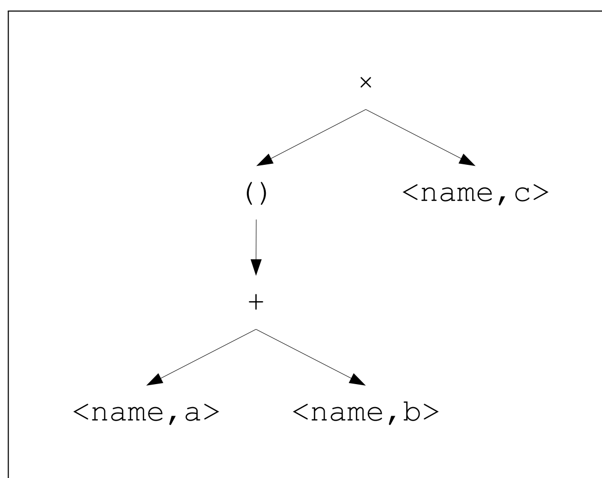
```
1 Expr → ( Expr )
2      | Expr Op name
3      | name
4 Op   → | + | - | × | ÷
```



Kuva 2.2: Lausekkeen `(a+b)×c` jäsennyspuu muodostettuna listauksen 2.1 kieliopin mukaisesti. Mukailtu lähteestä [CT12, s. 89].

Abstrakti syntaksipuu ja vierailija-suunnittelumalli

Jäsennyspuuta kutsutaan toisinaan myös konkreettiseksi syntaksipuuksi (concrete syntax tree), koska se kuvaa täydellisesti konkreettisesti kuinka symbolisten merkkien virta on jäsennetty kieliopin sääntöjen mukaisesti [Sco09, s. 32]. Käännöksen semanttisessa analyysivaiheessa kääntäjä tuottaa abstraktin syntaksipuun (abstract syntax tree, AST), tai lyhyemmin ilmaistuna syntaksipuun, poistamalla jäsennyspuusta ylimääräiset solmut, jotka tyypillisesti ovat yhteydettömän kieliopin muuttujasolmuja. Näin ollen kuvan 2.2 jäsennyspuusta rakennettu syntaksipuu voisi näyttää kuten kuvassa 2.3.

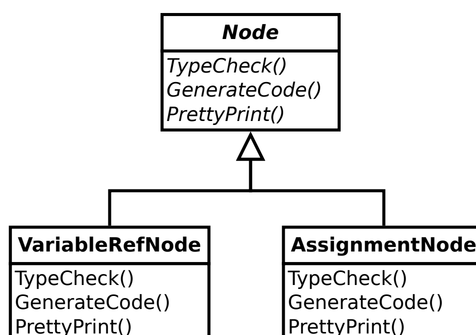


Kuva 2.3: Kuvan 2.2 jäsennyspuu muokattuna abstraktiksi syntaksipuuksi.

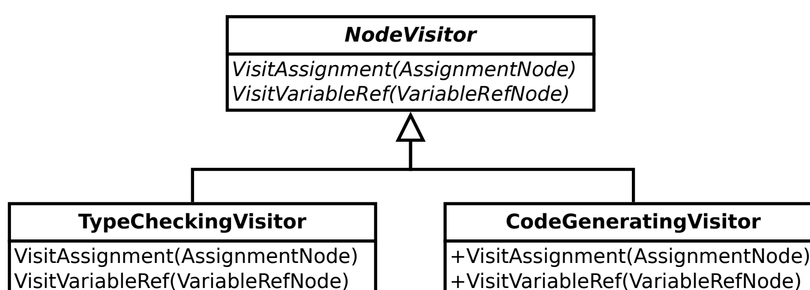
Vierailija-suunnittelumalli (visitor pattern) on suosittu tapa syntaksipuun läpikäyntiin. Esimerkiksi OpenJDK:n Java-kääntäjän syntaksipuurajapinta perustuu kyseiseen suunnittelumalliin [Jav16a]. Vierailija-suunnittelumallin tarkoituksena on mahdollistaa erilaisten operaatioiden toteuttaminen oliorakenteen kaikille elementeille ilman tarvetta muuttaa kyseisiä elementtejä uuden operaation lisäämiseksi [GHJV95].

Kuvan 2.4 UML-kaavio havainnollistaa tilannetta ilman vierailija-mallia. Se esittää osan abstraktin `Node`-luokan luokkahierarkiasta. Sen jokaisen aliluokan on toteutettava yliluokkansa esittelemät operaatiot. Jos `Node`-luokkaan halutaan lisätä uusi operaatio, on kaikkiin sen aliluokkiin lisättävä operaation toteutus.

Vierailija-suunnittelumalli erottaa operaation toteutuksen oliorakenteen elementtien toteutuksista [GHJV95]. Kukin operaatioista toteutetaan omaan vierailijaluokkaan. Vierailijaluokka toteuttaa vierailijarajapinnan, joka määrittelee operaation jokaiselle oliorakenteen elementin tyypille. Kuvassa 2.5 on esitetty vierailijaluokan abstrakti yliluokka `NodeVisitor` ja sen perivät operaatioluokat `TypeCheckingVisitor`, joka toteuttaa kuvan 2.4 `TypeCheck`-operaation jokaiselle elementtityypille, sekä `CodeGeneratingVisitor`, joka toteuttaa `GenerateCode`-operaation. Kullekin vierailijaluokan metodeista annetaan



Kuva 2.4: Kaikki operaatiot määritellään rajapinnassa ja toteutusten on toteutettava ne. Lähde: [GHJV95]



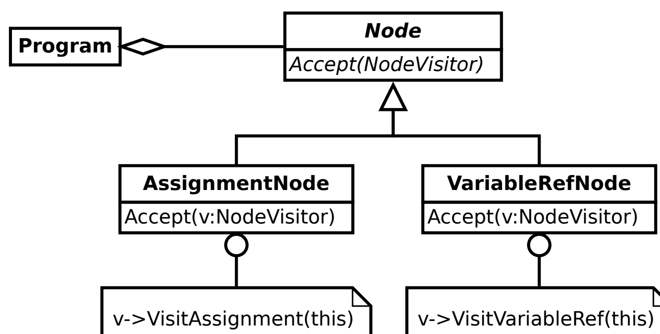
Kuva 2.5: Vierailijan esittely ja sen toteuttavat operaatioluokat. Lähde: [GHJV95]

parametrina elementtiluokan ilmentymä, jolle operaatio suoritetaan. Uusi operaatio lisätään tekemällä uusi `NodeVisitor`-luokan aliluokka.

Vierailijaluokan operaation kutsua varten tulee tietää elementin todellinen tyyppi, jotta kutsu voidaan kohdistaa oikeaan operaatiometodiin. Yksinkertaisinta on antaa elementin itse kertoa tyyppinsä. Kuvassa 2.6 on vierailijamallin mukaiset elementtityypit. Abstrakti `Node`-luokka esittelee ainoastaan `Accept`-metodin, joka ottaa parametrinaan vierailijaluokan. Jokainen elementtiluokka toteuttaa metodin kutsumalla tyyppinsä mukaista vierailijaluokan operaatiometodia antaen itsensä parametrina.

Jos uusia elementtien tyyppejä esitellään usein, uuden elementtityypin lisääminen vierailijarajapintaan ja jokaiseen vierailijatoteutukseen on työläs operaatio [GHJV95]. Vierailijamalli toimii parhaiten silloin, kun elementtityypit pysyvät suhteellisen vakioina, mutta operaatioita halutaan usein lisätä.

Abstraktin syntaksipuun läpikäynti voidaan toteuttaa esimerkiksi siten, että vierailijaluokka kutsuu käsittelemänsä solmun lapsisolmujen `Accept`-metodia antaen itsensä parametrina [GHJV95]. Jos läpikäynnistä tehdään abstrakti luokka, riittää sen perivän luokan toteuttaa ainoastaan suoritettavat operaatiot ja antaa yliluokan huolehtia puun läpikäynnistä. Tällainen abstrakti yliluokka on toteutettu esimerkiksi OpenJDK:n syntaksipuurajapintaan [Jav16a].



Kuva 2.6: Vierailija-suunnittelumallin mukaan toimivat oliorakenteen elementit. Lähde: [GHJV95]

Semanttiset tarkastukset

Semanttisen analyysin vaiheessa kääntäjä rakentaa symbolitaulun ja suorittaa lukuisia tarkastuksia varmistaakseen, että ohjelma noudattaa semanttisesti kielen määrittelyä.

Symbolitaulu rakennetaan tyypillisesti semanttisen analyysin aluksi, jolloin kääntäjä käy läpi ohjelman syntaksipuun ja tallentaa symbolitauluun jokaisen ohjelmassa esiintyvän nimen [Sco09, s. 29]. Sen lisäksi kääntäjä tallentaa symbolitauluun muuta nimen merkitykseen liittyvää tietoa, kuten esimerkiksi nimeen liittyvän tyypin, mahdollisesti viitteen nimen esittelyyn sekä lohkon, jossa nimi on olemassa.

Sen jälkeen kääntäjä käy tavallisesti useaan otteeseen symbolitaulun läpi ja suorittaa monenlaisia tarkastuksia staattisilla analyysialgoritmeilla. Staattiseksi analyysiksi (static analysis) kutsutaan yleisesti algoritmeja, jotka ennustavat ohjelman suoritusaikaista toimintaa [Sco09, s. 179–180]. Jos ohjelman voidaan päätellä aina noudattavan sääntöjä, analyysi on täsmällinen (precise).

Suoritettavat tarkastukset riippuvat käännettävästä kielestä ja sen säännöistä. Kääntäjä voi esimerkiksi tarkastaa, että jokaista ohjelmassa käytettyä nimeä käytetään sen voimassaoloalueella ja että se viittaa johonkin olemassa olevaan esittelyyn. Tarkastukset voivat olla asiayhteyssidonnaisia, esimerkiksi että `return`-lauseen jälkeen samassa lohkossa ei saa esiintyä muita lauseita. Myös tyyppitarkastukset ja pääsynvalvonta eli pääsymääreiden asettamien pääsyrajoitteiden noudattamisen valvonta kuuluvat tyypillisesti semanttiseen analyysiin.

Kielestä riippuen kääntäjän suorittamat staattiset analyysit eivät ole aina täsmällisiä [Sco09, s. 179–180]. Tällöin kääntäjä generoi ohjelmaan suoritusaikana tarkastuksen suorittavan koodin.

2.2 Tyyppijärjestelmät

Ohjelmointikielissä jokaisella arvolla on tyyppi. Sellaisissakin kielissä, jotka eivät yksikäsitteisesti määrittele tyypin käsitettä, voidaan ajatella olevan yksi universaali tyyppi, joka käsittää kaikki kielessä käsiteltävät arvot [Car97]. Tällainen kieli on esimerkiksi symbolinen konekieli (assembly), joka on tekstimuotoinen vastine laitteiston ymmärtämälle konekielelle [Sco09, s. 5–6]. Kun kieli määrittelee tyypin käsitteen, se tarvitsee myös säännöt, jotka määrittelevät tyyppien ominaisuudet ja esimerkiksi eri tyyppien yhteensopivuuden. Tätä säännöstöä ja sitä valvovaa toteutusta kutsutaan tyyppijärjestelmäksi. Tässä luvussa perehdytään tyyppien ja tyyppijärjestelmien käsitteisiin.

Tyyppi

Tietokonelaitteisto käsittelee sen muistissa olevia bittejä eri tavoin: käskyinä, osoitteina, merkkeinä, kokonais- ja liukulukuina [Sco09, s. 290]. Bitit itsessään eivät sisällä tietoa, kuinka niitä tulee käsitellä, ne eivät eroa toisistaan olemalla eri tyyppisiä. Tyypittömyys näkyy vastaavasti symbolisessa konekielessä. Ohjelmaan on mahdollista kirjoitta millaisia operaatioita tahansa suoritettavaksi muistissa sijaitseville biteille niiden muistipaikasta riippumatta. Sen sijaan useimmissa korkeamman tason kielissä kukin arvo sidotaan johonkin tyyppiin. Sanotaan, että arvolla on tyyppi.

Tyyppi on teoreettisesti tarkasteltuna rajoite, joka määrittelee niiden kelvollisten arvojen joukon, jotka ovat tyypin mukaisia [Tra09]. Esimerkiksi tyyppi `int` edustaa monissa ohjelmointikielissä kokonaislukua. Ohjelman muuttujan, jonka tyyppi on `int`, voidaan olettaa saavansa arvokseen ainoastaan kokonaislukuja.

Ohjelmointikielissä tyyppejä käytetään arvojen luokittelemiseksi ja määrittelemään arvoille kelvolliset operaatiot [Tra09]. Tyypin `int` arvoille kelvollisia operaatioita ovat esimerkiksi yhteen- ja vähennyslasku. Tyypillisesti ohjelmointikielet määrittelevät pienen joukon perustyyppejä sekä antavat ohjelmoijan määritellä omia tyyppejään.

Vaikka teoriassa tyyppi määrittelisi rajoittamattoman arvojen joukon, käytännössä ohjelmointikielet rajoittavat usein arvojen joukkoa [Tra09]. Esimerkiksi monet kielet rajoittavat `int`-tyypin kuvaamaan kokonaislukujen osajoukkoa, johon kuuluvat arvot voidaan esittää tietyllä määrällä bittejä. Rajoittamalla arvojen joukkoa tyyppi on mahdollista sitoa alla olevaan laitteistotason esitysmuotoon, mikä yksinkertaistaa kielen toteutusta. Esimerkiksi Javan `int`-tyyppi on määritelty kokonaisluvuksi suljetulta väliltä −2 147 483 648 ja 2 147 483 647, joka voidaan esittää 32 bitillä [GJS+15, s. 43]. C++ lähestyy asiaa eri tavoin. Se määrittelee `int`-tyypin kokonaisluvuksi, mutta jättää sen lukualueen kielen toteutuksen määriteltäväksi [Str13, s. 148–150]. Siten kokonaisluvun lukualue saattaa vaihdella kääntäjän ja kohdelaitteiston mukaan.

Ohjelmointikieliä, joiden muuttujille ja arvoille voidaan määrittää tyyppi, kutsutaan tyypitetyiksi kieliksi [Car97]. Tyypittämättömiksi kieliksi kutsutaan kieliä, joiden muuttujien arvoja ei rajoiteta tyypein. Voidaan kuitenkin ajatella, että tyypittämättömissä kielissä on yksi universaali tyyppi, johon kaikki arvot kuuluvat [CW85]. Aiemmin mainittu symbolinen konekieli on yksi tällainen kieli: kaikki arvot muistissa ovat pelkkiä bittijonoja. Toinen esimerkki on puhdas LISP, jossa kaikkien arvojen voidaan katsoa olevan niin kutsuttuja S-lausekkeita. Myös lambdakalkyyliä voidaan pitää tyypittämättömänä, koska sen kaikki arvot ovat funktiota: myös numerot ja muut tietorakenteet ovat teoriassa funktioita.

Tyyppijärjestelmä

Tyyppijärjestelmä on ohjelmointikielen komponentti, joka pitää kirjaa tyypeistä ja muuttujista ja käytännössä ohjelman kaikkien lausekkeiden ja arvojen tyypeistä [Car97]. Tyyppijärjestelmän perimmäinen tarkoitus on ehkäistä suoritusvirheitä ohjelman suoritusaikana.

Tyyppijärjestelmän voi ajatella koostuvan kahdesta pääkomponentista: mekanismista määritellä tyypit sekä luoda sidos niiden ja kielen rakenteiden välillä ja joukosta sääntöjä, jotka määrittelevät tyyppien yhtäläisyyden, yhteensopivuuden sekä tyyppipäättelyn [Sco09, s. 290]. Tyyppien yhtäläisyyssäännöt määrittelevät, milloin kahden eri arvon tyypit ovat yhtäläiset. Tyyppien yhteensopivuussäännöt määrittelevät, missä kontekstissa minkäkin tyyppistä arvoa on mahdollista käyttää. Tyyppipäättelysäännöt määrittävät lausekkeen tyypin sen osien tyyppien ja mahdollisesti myös kontekstin perusteella. Erityisesti polymorfismia tukevissa kielissä on tärkeää kyetä erottamaan toisistaan muuttujan tai viitteen tyyppi sekä olion tyyppi, johon muuttuja tai viite osoittaa.

Kielen toteutus suorittaa kielen tyyppijärjestelmän mukaiset tyyppitarkastukset käännösaikaisesti ja/tai suoritusaikaisesti. Tyyppijärjestelmän mukaan kelvottomien operaatioiden, esimerkiksi kokonaisluvun ja merkkijonon yhteenlasku, suoritus estetään käännösvirheenä tai suoritusaikaisen virheenhallintamekanismin aktivointina. Kielestä riippuen kääntäjä tai suoritusympäristö saattaa pyrkiä välttämään virhetilanteen tekemällä mahdollisuuksien mukaan implisiittisen tyyppimuunnoksen.

2.3 Javan keskeistä tyyppikäsitteistöä

Vaikka tutkielman lukijan oletetaan tuntevan olio-ohjelmoinnin käsitteet ja Java-kielen pääpiirteissään, kerrataan tässä luvussa lyhyesti tutkielman kannalta keskeisimmät Javan käsitteet liittyen tyyppeihin ja olioihin.

Tyypit ja oliot

Java on staattisesti ja vahvasti tyypitetty kieli. Javan tyypit (type) on jaettu kahteen ryhmään: primitiivityypit (primitive type) ja viitetyypit (reference type) [GJS+15, s. 41–43]. Primitiivityypit toteuttavat niin kutsuttua arvomallia (value model) [Sco09, s. 225–228]. Arvomallissa tyypin mukaiset arvot välitetään ja niitä käytetään arvoina. Esimerkiksi muuttujaan tallennetaan tyypin mukainen arvo. Kun muuttujan arvo sijoitetaan toiseen muuttujaan, arvo kopioidaan muuttujasta toiseen. Javan primitiivityyppejä ovat esimerkiksi `boolean` ja `int` [GJS+15, s. 41–43]. Ne ovat kaikki kieleen sisäänrakennettuja tyyppejä.

Viitetyypit toteuttavat niin kutsuttua viitemallia (reference model) [Sco09, s. 225–228]. Viitetyyppien ilmentymät välitetään ja niitä käytetään aina viitteinä ilmentymään. Esimerkiksi viitetyypin muuttuja sisältää ainoastaan muistiviitteen olioon. Kun muuttujan arvo sijoitetaan toiseen muuttujaan, kopioidaan viite muuttujasta toiseen, mutta viitteen kohteena oleva olio pysyy samana. Viitetyyppejä on neljä erilaista: luokka (class), rajapintaluokka (interface), tyyppimuuttuja (type variable) ja taulukko (array) [GJS+15, s. 52]. Lisäksi on olemassa erikoinen `null`-tyyppi, joka edustaa mitä tahansa viitetyyppiä.

Olio (object) on jonkin luokan ilmentymä (instance) tai taulukko [GJS+15, s. 53]. `null`-tyypistä ei ole koskaan olemassa ilmentymää. Null-viite on muistiviitteen erikoistapaus, joka osoita mihinkään olioon.

Luokka on kuvaus oliosta, jonka mukaan luokan ilmentymät rakennetaan. Luokan esittely (declaration) määrittelee uuden nimetyn viitetyypin [GJS+15, s. 193]. Luokan rungossa (class body) esitellään sen jäsenet määritellään sen jäsenet (member) [GJS+15, s. 207–208], joita voivat olla kentät (field), metodit (method) sekä sisäkkäiset luokat ja rajapinnat. Luokan runkoon voivat sisältyä myös luokan muodostimet (constructor), ilmentymän alustaja (instance initializer) ja luokan staattinen alustaja (static initializer).

Luettelotyyppi (enum, enumeration type) on tyyppi, jonka arvojoukko on sen esittelyssä lueteltu joukko nimiä eli lueteltuja vakioita [Sco09, s. 297–298]. Luettelotyyppi on tyyppiturvallinen vastine luettelolle vakioita, joiden arvojen tarkoitus on ainoastaan erottaa vakiot toisistaan. Javan luettelotyyppi on luokan erikoistapaus [GJS+15, s. 269–278]. Sen luetellut vakiot ovat luettelotyypin ilmentymiä. Lueteltujen vakioiden lisäksi se voi sisältää kenttiä ja metodeita. Luettelotyypille ei voi määrätä yliluokkaa eikä luettelotyyppi voi olla toisen luokan yliluokka. Se voi kuitenkin toteuttaa rajapintoja.

Rajapintaluokka (interface) on abstrakti tyyppi, jonka jäseniä voivat olla vakiot, abstraktit metodit sekä sisäkkäiset luokat ja rajapintaluokat [GJS+15, s. 279]. Luokka voi toteuttaa (implement) rajapinnan, jolloin luokka toimii rajapintaluokan edustajana. Annotaatiotyypit (annotation type) ovat rajapintaluokkien erikoistapaus, jotka kuvaillaan luvussa 4.3.

Sisäkkäiset tyypit

Luokat on jaettavissa päätason luokkiin (top-level class) ja sisäkkäisiin luokkiin (nested class) [GJS+15, s. 191–192]. Päätason luokka on luokka, joka ei ole sisäkkäinen luokka. Sisäkkäinen luokka on luokka, jonka esittely sijaitsee toisen luokan sisällä. Sisäkkäisiä luokkia ovat jäsenluokat (member class), paikalliset luokat (local class) ja anonyymit luokat (anonymous class).

Jäsentyyppi (member type) on tyyppi, jonka esittely sijaitsee välittömästi toisen tyyppiesittelyn sisällä [GJS+15, s. 256]. Jäsentyyppi voi olla luokka, jolloin sitä kutsutaan jäsenluokaksi, tai rajapintaluokka, jolloin sitä kutsutaan jäsenrajapinnaksi (member interface). Koska teknisesti luettelotyypit ovat luokkia ja annotaatiotyypit rajapintaluokkia, myös ne voivat olla jäsentyyppejä. Luokat, rajapintaluokat, luettelotyypit ja annotaatiotyypit voivat kaikki sisältää jäsentyyppejä [GJS+15, s. 256, 293–294].

Jäsenluokat voivat olla staattisia, jolloin ne käyttäytyvät kuten päätason luokatkin [GJS+15, s. 256–257]. Luettelotyypit, rajapintaluokat ja annotaatiotyypit ovat aina implisiittisesti staattisia [GJS+15, s. 199, 269]. Jos jäsenluokka ei ole staattinen, sitä kutsutaan sisäluokaksi (inner class) [GJS+15, s. 199–202]. Sisäluokat ovat sidoksissa aina sitä ulomman luokan ilmentymään. Toisin sanoen sisäluokan ilmentymää ei voi olla ilman sitä ulomman luokan ilmentymää. Sisäluokalla on pääsy sen ulomman luokan ilmentymän yksityisiin jäseniin, jonka kontekstissa se on luotu. Paikalliset ja anonyymit luokat ovat jollekin suoritettavalle lohkolle paikallisia sisäluokkia. Paikalliset luokat ovat nimettyjä, ja niiden esittely ja ilmentymien luonti ovat toisistaan erillisiä [GJS+15, s. 413–414]. Anonyymeillä luokilla ei ole nimeä, ja niiden ilmentymä luodaan esittelyn yhteydessä [GJS+15, s. 491–492].

3 Tiedon piilottamisen mekanismit

Tiedon piilottamisen käsitteen myötä pääsynvalvonnasta on tullut ohjelmointikielten keskeinen ohjelmien modularisoinnin väline. Tässä luvussa tutustutaan tiedon piilottamisen käsitteeseen sekä esitellään tunnettujen staattisten olio-ohjelmointikielten pääsynvalvontajärjestelmiä. Luvun lopuksi tehdään yhteenveto esitellyissä kielissä esiintyneistä saatavuusperusteista.

3.1 Tiedon piilottaminen

Olio-ohjelmoinnin perustavanlaatuisten konseptien, kuten kapseloinnin (encapsulation), perinnän (inheritance) ja metodien dynaamisen sidonnan (dynamic method binding) juuret ovat 1960-luvun puolivälissä kehitetyssä Algol 60-kieleen perustuvassa Simula-kielessä [Dah04][Sco09, s. 450]. Nykykieliin verrattuna Simulan kapselointi oli heikkoa tiedon piilottamisen osalta [Sco09, s. 450]. Merkittäviä parannuksia siihen esittelivät muun muassa CLU [LSAS77], Modula [Wir80], Euclid [LHL+77] ja niiden sukulaiskielet. Seuraavaksi tutustutaan tiedon piilottamisen käsitteeseen sekä esitellään Modula-kielen pääsynvalvontajärjestelmä esimerkkinä moduuliperusteisesta tiedon piilottamisesta.

Moduuleihin perustuva tiedon piilottaminen

Tiedon piilottamisen (data hiding) käsitteen esitteli David Parnas vuonna 1972 [Par72]. Tiedon piilottamisella hän ei tarkoittanut pelkän datan piilottamista, vaan väitti, että järjestelmän modularisoinnin ensisijaisena tarkoituksena tulisi olla kriittisten suunnitteluperusteiden piilottaminen. Siihen aikaan yleisesti käytetystä vuokaaviosta modularisoinnin perusteena olisi syytä luopua. Vuokaavion käyttö modularisoinnin perusteena johtaa esimerkiksi siihen, että keskeiset tietorakenteet ovat paljaita kaikille tietorakennetta käyttäville moduuleille. Muutos tietorakenteen muodossa tai käyttötavassa johtaa siihen, että kaikkia tietorakennetta käyttäviä moduuleja on muutettava. Jos sen sijaan tietorakenne on moduulin sisäinen, piilotettu toteutusyksityiskohta, jonka sisältämiin tietoihin muut moduulit pääsevät käsiksi ainoastaan moduulista ulos näkyvien operaatioiden kautta, tietorakenteen muutokset heijastuvat ainoastaan tietorakenteen sisältävään moduuliin.

Kenties merkittävin hyöty tiedon piilottamisesta on, että sen ansiosta järjestelmän moduulit ovat toisistaan mahdollisimman riippumattomia (decoupled) [Blo08, s. 67–70]. Sen seurauksena niiden kehitys, testaaminen, optimointi, käyttö, semantiikka ja muokkaaminen on mahdollista muista moduuleista riippumatta. Se parantaa ohjelmien luotettavuutta ja ylläpidettävyyttä, kun mahdolliset virheet on helpompi paikallistaa tietyn moduulin sisään. Siksi voidaan pitää suositeltavana, että kukin ohjelmaelementti on saatavissa (accessible) niin rajoitetusti kuin mahdollista ja ohjelmakomponenttien julkiset rajapinnat ovat mahdollisimman minimaalisia. Kutsuttakoon tätä minimaalisen saatavuuden periaatteeksi.

Saatavuuden käsitteeseen liittyy läheisesti näkyvyys (visibility). Tyypillisesti sanotaan, että jokin tyyppi on näkyvissä ja tyypin jäsenet saatavissa. Tässä tutkielmassa ei tehdä erottelua näkyvyyden ja saatavuuden välillä, vaan yksinkertaisuuden vuoksi puhutaan aina saatavuudesta.

Modula — moduulirakenne ohjelmointikielessä

Esimerkkinä moduuleihin perustuvasta tiedon piilottamisesta olkoon Modula-kieli, jonka Pascal-kielen kehittäjä Niklaus Wirth kehitti 1970-luvun lopulla Pascalin pohjalta. Alkuperäistä Modula-kieltä [Wir76] seurasi pian Modula-2 [Wir80], johon tavallisesti viitataan puhuttaessa Modulasta [Wir02]. Kielen nimi juontaa moduulien käsitteestä, joka oli kielen keskeinen uusi piirre.

Sanalla moduuli (module) tarkoitetaan yleistäen itsenäistä yksikköä, joita yhdistelemällä saadaan aikaiseksi suurempi kokonaisuus [Mer]. Modula-kielessä moduuli tarkoittaa kokonaisuutta, joka sisältää joukon esittelyitä sekä sarjan lauseita [Wir80]. Moduulit tekivät mahdolliseksi sisäisten toteutusyksityiskohtien piilottamisen Modula-ohjelmissa. Se on eräänlainen suojamuuri sen sisällä esitellyille ohjelman osille: moduulin sisällä esitellyt komponentit ovat saatavissa moduulin sisällä, mutta mikään moduulin ulkopuolella ei oletuksena näy moduulin sisälle eikä mikään moduulin sisältä näy sen ulkopuolelle.

Moduulin suhteet sen ulkopuoliseen maailmaan määritellään moduulin esittelyssä [Wir80]. Ne moduulin esittelemät nimet, jotka halutaan julkistaa moduulin ulkopuolella käytettäväksi, tulee listata moduulin esittelyyn kuuluvassa export-lauseessa. Vastaavasti moduulin ulkopuolella esitellyt nimet, joita halutaan käyttää moduulin sisällä, on listattava moduulin esittelyyn kuuluvassa import-lausessa.

Lohkorakenteeseen perustuva tiedon piilottaminen

Vuonna 1973 Wulf ja Shaw esittivät, että muuttujien tulisi olla paikallisia: niiden tulisi olla olemassa ainoastaan esittely-ympäristössään [WS73]. Abstrakteille tietotyypeille muuttujien paikallisuus on keskeinen vaatimus [Sha84]. Ohjelman lohkorakenteeseen perustuvat säännöt muuttujien olemassaolosta ja saatavuudesta vain esittely-ympäristössään mahdollistivat muuttujien paikallisuuden toteutumisen.

Edellä esitelty Modula-kieli on yksi esimerkki ohjelmointikielestä, jossa ohjelman lohkorakenne rajoittaa esiteltyjen kohteiden saatavuutta: ne ovat saatavissa vain sen lohkon sisällä, jossa ne on esitelty [Wir80]. Esimerkiksi proseduurin sisällä esitelty muuttuja on saatavissa vain kyseisen proseduurin sisällä.

Lohkorakenteeseen perustuva tiedon piilottaminen tapahtuu implisiittisesti lohkorakenteen mukaan, eikä ohjelmoijan tarvitse, eikä saa, erikseen merkitä ohjelmaelementtejä

lohkorakenteen sisäisiksi. Tämän tutkielman näkökulmasta oleellinen kysymys tiedon piilottamisessa on, kuinka ohjelmoija voi vaikuttaa ohjelmaelementin näkyvyyteen. Koska lohkorakenteeseen perustuva tiedon piilottaminen on implisiittistä, sitä ei jatkossa enää käsitellä puhuttaessa tiedon piilottamisesta.

3.2 Staattisten oliokielten pääsynvalvontamekanismeja

Tässä luvussa esitellään kahdeksan tunnetun staattisen olio-ohjelmointikielen pääsynvalvontajärjestelmät. Kielet on valittu niiden historiallisen merkittävyyden ja yleisyyden sekä pääsynvalvontajärjestelmien poikkeavuuden perusteella. Vertailtavuuden vuoksi kaikki esiteltävät kielet ovat staattisia oliokieliä. Näin ollen pääsynvalvonta on kaikissa kielissä staattinen ominaisuus. Siitä seuraa, että ohjelmaelementtien saatavuus on pääteltävissä käännösaikaisesti.

Kielet esitellään syntyhetken mukaan aikajärjestyksessä. Kunkin kielen kohdalla käytetään pääsääntöisesti kielen omaa termistöä, poikkeuksena näkyvyys, joka muun tutkielman tavoin sisällytetään saatavuuden käsitteeseen. Yksinkertaisuuden vuoksi luokan muodostin käsitetään myös sen jäseneksi, ellei erikseen toisin mainita.

CLU

1970-luvulla Simula 67-kielestä vaikutteensa saanut CLU [LSAS77] oli ensimmäinen toteutettu ohjelmointikieli, johon oli rakennettu mekanismi datan abstrahoimiseksi [Lis96]. CLUssa luokan käsitettä vastaa klusteri (cluster) [LSAS77]. Klusteri esittelee määriteltävän tyypin olioiden esitysmuodon, olioille suoritettavat operaatiot sekä niiden toteutukset. Operaatioiden toteutukset ovat CLUssa proseduureja.

Kieleen rakennetun datan abstrahoinnin myötä tiedon piilottaminen tuli mahdolliseksi. Klusterissa esiteltyyn olion esitysmuotoon pääsee käsiksi vain klusterin proseduureista [LSAS77]. Siten olion tilaa on mahdollista muuttaa ainoastaan tyypin operaatioita kutsumalla. Klusteriin on myös mahdollista sisällyttää proseduureja, joita ei ole esitelty tyypin operaatioiksi. Sellaiset proseduurit ovat kutsuttavissa ainoastaan saman tyypin proseduureista [Lis96]. Proseduureissa käytetyt muuttujat eivät ole olioita, vaan osoittimia olioihin [LSAS77]. Olioiden jakamiseksi ei siten ole tarpeellista jakaa muuttujia proseduurin ulkopuolelle, joten muuttujat ovat näkyviä ainoastaan proseduurin sisällä.

Smalltalk

Smalltalk on Simulasta vaikutteensa saanut kieli, jonka sanotaan vakiinnuttaneen olio-ohjelmointi -termin [Kay93]. Se kehitettiin Xerox Palo Alto Research Centerissä 1970-luvulla pitkälti Alan Kayn innovoimana, ja vuonna 1980 se julkaistiin nimellä Smalltalk-80. Kaikki

Smalltalkissa perustuu olioihin sekä niiden väliseen kommunikointiin. Oliot kommunikoivat keskenään lähettämällä toisilleen viestejä (messages), jotka ovat pyyntöjä oliolle suorittaa sen operaatioita [GR83, s. 6].

Jokainen olio on Smalltalkissa jonkin luokan ilmentymä, ja jokaisella saman luokan ilmentymällä on luokan määrittelemät ominaisuudet (properties) [GR83, s. 8–9, 40]. Muissa oliokielissä Smalltalkin luokan ominaisuus-käsite vastaa luokan jäsentä. Ominaisuudet jakautuvat julkisiin ja yksityisiin. Julkisia ominaisuuksia ovat viestit, ja ne muodostavat olion julkisen rajapinnan. Ilmentymämuuttujat ja metodit ovat olion yksityisiä ominaisuuksia. Ilmentymämuuttujat ylläpitävät olion sisäistä tilaa. Metodit kuvaavat ne operaatiot, jotka suoritetaan olion vastaanottaessa operaatiota vastaavan viestin. Vaikka metodit käsitetään olioiden yksityisiksi ominaisuuksiksi, Smalltalk ei varsinaisesti estä millään tavoin niiden kutsumista viestein [Lew95, s. 56].

Muuttujat ovat Smalltalkissa yksityisiä tai jaettuja [GR83, s. 21–22]. Muuttujien saatavuustaso määräytyy muuttujanimen ensimmäisen kirjaimen mukaan: pienellä kirjaimella alkavat muuttujat ovat yksityisiä ja isolla kirjaimella alkavat jaettuja. Yksityisiä muuttujia ovat väliaikaiset muuttujat ja ilmentymämuuttujat [GR83, s. 44–47]. Väliaikaiset muuttujat ovat olemassa ja käytettävissä ainoastaan metodien sisällä. Ilmentymämuuttujat ovat ilmentymäkohtaisia ja saatavissa ainoastaan saman olion sisällä. Ilmentymämuuttujat sekä metodit periytyvät aliluokille ja ovat myös niiden ilmentymien käytettävissä [GR83, s. 58].

Jaetut muuttujat ovat saatavissa myös muista olioista niin kutsuttujen altaiden (pool) välityksellä [GR83, s. 47–48]. Jokaisella luokalla on oma altaansa, jonka muuttujiin pääsee käsiksi ainoastaan saman luokan ja sen metaluokan [GR83, s. 84] sisällä. Kaikkien luokkien kesken on jaettu `Smalltalk`-niminen muuttuja-allas [GR83, s. 47–48]. Sen sisältämiä muuttujia kutsutaan globaaleiksi muuttujiksi. Esimerkiksi luokkien nimet tallennetaan aina tähän globaaliin muuttuja-altaaseen, joten luokat ovat saatavissa kaikkialla [GR83, s. 58]. Edellä mainittujen allastyyppien lisäksi ohjelmoija voi esitellä omia muuttuja-altaita, joiden sisältämät muuttujat jaetaan niiden luokkien kesken, jotka määrittelevät altaan käyttöönsä. Luokan aliluokka perii käyttöönsä yliluokkansa jaetut muuttujat, mutta aliluokan esittelemiin uusiin jaettuihin muuttujiin ei ole pääsyä yliluokasta. [GR83, s. 58].

C++

Tanskalainen tietojenkäsittelytieteilijä Bjarne Stroustrup alkoi vuonna 1979 kehittää uutta ohjelmointikieltä C with Classes, joka vuonna 1983 sai nimekseen C++ [Str96, s. 700–702, 716]. Merkittävä vaikutin kielen syntyyn oli Simula-kieli. Kielen toteutus oli alkujaan C-kieltä laajentava esiprosessori. Kielen kehittymisen myötä sille on kehitetty lukuisia itsenäisiä kääntäjiä [Str96, s. 739–740]. Nykyistä C++-versiota Stroustrup kuvailee ”yleiskäyttöiseksi ohjelmointikieleksi, joka tarjoaa suoran ja tehokkaan laitteistomallinnuksen yhdistettynä kevyiden abstraktiomäärittelyiden helppouteen” [Str13, s. 9]. Kielen ensim-

mäinen ISO-standardi vahvistettiin vuonna 1998 [Str13, s. 25]. Vuonna 2015 C++ oli joidenkin arvioiden [Cas15] mukaan kolmanneksi suosituin ohjelmointikieli.

C++:ssa luokan jäsenillä voi olla kolme erilaista saatavuustasoa: yksityinen (private), suojattu (protected) ja julkinen (public) [Str13, s. 453, 469, 600]. Yksityiset jäsenet ovat saatavissa ainoastaan saman luokan jäsenfunktioista sekä jäsenluokkien funktioista. Suojatut jäsenet ovat saatavissa edellisten lisäksi niiden luokkien jäsenfunktioista, jotka perivät suojatun jäsenen määrittelevän luokan. Julkiset jäsenet ovat saatavissa kaikkialla. Saatavuustaso asetetaan määreillä `private`, `protected` ja `public`. Oletuksena luokan jäsenet ovat yksityisiä.

Perintäsaatavuuden eli saatavuuden rajoittaminen kantaluokkaa (base class) perittäessä on myös mahdollista [Str13, s. 605]. C++:ssa kantaluokaksi kutsutaan perittyä luokkaa. Oletuksena kantaluokka peritään yksityisenä, mutta edellä mainittuja määreitä käyttäen perinnän voi määritellä tapahtumaan myös suojattuna tai julkisena. Erilaiset perintäsaatavuudet palvelevat erilaisia käyttötarkoituksia [Str13, s. 605], ja kielen tuki moniperinnälle [Str13, s. 583] mahdollistaa niiden monipuolisen hyödyntämisen.

Esimerkiksi kun luokka C perii kantaluokan B yksityisenä, ainoastaan C:n jäsenet pystyvät aksessoimaan B:n jäseniä C:n jäseninä ja tekemään tyyppimuunnoksen C:stä B:ksi [Str13, s. 605]. Yksityinen perintä on tarkoitettu tilanteeseen, jossa kantaluokkaa käytetään osana luokan sisäistä toteutusta eikä se ole osa luokan julkista rajapintaa. Jos C perii B:n suojattuna, myös C:n perivät luokat pystyvät aksessoimaan B:n jäseniä C:n jäseninä ja tekemään tyyppimuunnoksen C:stä B:ksi. Suojattu perintä on hyödyllinen yksityistä perintää vastaavissa tilanteissa, mutta joissa luokka on suunniteltu edelleen perittäväksi. Julkinen perintä on sen sijaan tarkoitettu tilanteisiin, jossa luokka toimii kantaluokkansa alityyppinä eli erikoistettuna kantaluokan edustajana. Tällöin kantaluokasta tulee osa luokan julkista rajapintaa. C++:n kehittäjän mukaan tämä on yleisin perinnän muoto [Str13, s. 605].

Perintäsaatavuutta määrittäessään perivä luokka pystyy rajoittamaan kantaluokkansa jäsenten näkyvyyttä, muttei lisäämään sitä. Jos perivä luokka on rajoittanut kantaluokkansa jäsenien näkyvyyttä, se pystyy palauttamaan valitsemiensa jäsenten näkyvyyden alkuperäiseksi luettelemalla kyseiset jäsenet `using`-avainsanan kanssa [Str13, s. 606–607].

Edellä mainittujen saatavuustasojen lisäksi luokan yksityiset osat on mahdollista paljastaa määrätyille funktioille tai luokille esittelemällä ne luokan tovereina (friend) [Str13, s. 571–573]. Esimerkiksi luokka A esitellessään funktion f toverifunktiokseen antaa tälle pääsyn kaikkiin yksityisiin, myös yksityisinä perimiin jäseniinsä. Funktio f sijaitsee luokkamäärittelyn ulkopuolella eikä ole osa A:n ilmentymää. Luokka A esitellessään luokan B toveriluokakseen tekee kaikista luokan B funktiosta luokan A toverifunktioita.

Toveruus ei ole transitiivinen ominaisuus [Str13, s. 683]. Esimerkiksi jos luokka C on B:n toveri ja B on A:n toveri, C ei ole A:n toveri. Toveruus ei myöskään ole periytyvä ominaisuus. Siten jos luokka B on A:n toveri ja C perii B:n, C ei ole A:n toveri.

Toverifunktiot on tarkoitettu käytettäväksi silloin, kun luokan toteutusta tarvitsevan funktion on tarkoitus yksinkertaistaa luokan käyttöä tai funktio tarvitsee pääsyn kahden luokan toteutukseen [Str13, s. 576]. Tietorakenteen läpikäyvä iteraattoriluokka on tyypillinen esimerkki toveriluokasta, joskin se on toteutettavissa myös jäsenluokkana. Kahden eri luokan välinen operaattorifunktio, esimerkiksi matriisi- ja vektori-luokkien välinen kertolasku, on yksittäisen toverifunktion tyypillinen käyttötapaus. Pääsääntöisesti funktiot, jotka tarvitsevat pääsyn luokan toteutukseen, tulisi toteuttaa luokan jäseninä.

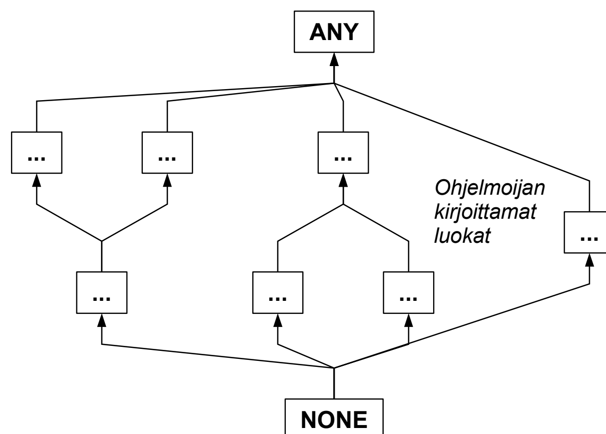
Eiffel

Eiffel on Bertrand Meyerin 1980-luvun puolivälissä kehittämä Simula 67-kielestä vaikutteensa saanut moniperintää tukeva olio-ohjelmointikieli [Mey85, s. 1–2]. Tavoitteena oli luoda tutkimuskielen sijaan ohjelmistojen tuottamiseen suunnattu kieli, jonka keskeisiä suunnitteluperiaatteita olivat tehokkuus, luotettavuus, uudelleenkäytettävyys, laajennettavuus ja yhteensopivuus pitkällä aikavälillä, modulaarisuus ja siirrettävyys. Kielen tuli pystyä myös käsittelemään monimutkaisia ja hyvin dynaamisia tietorakenteita huolehtien automaattisesti muistinhallinnasta. Kielen ensimmäinen ECMA-standardi [M+05] julkaistiin vuonna 2005.

Eiffel-kielen pääsynhallinta perustuu kutsusääntöön ja valikoivaan saatavuuteen. Kutsusäännön mukaan luokassa C esiintyvä kutsu `x.f(...)` on kelvollinen silloin, kun f on kutsun kohdeolion x:n tyypin mukaisen luokan esittelemä tai perimä ominaisuus (feature), johon pääsy on sallittu C:stä [Mey01, s. 31]. Ominaisuuden sanotaan tällöin olevan kutsuttavissa (available for call) tai saatavissa (available) [M+06, s. 52]. Eiffelissä luokan jäseniä kutsutaan ominaisuuksiksi ja suoraan tai välillisesti perittyjä luokkia kantaluokiksi (base class) [M+06, s. 35–36, 73–74]. Ominaisuudet ovat attribuutteja (attribute) tai rutiineja (routine). Attribuutit ovat joko vakioita (constant) tai muuttujia (variable). Rutiinit ovat proseduureja (procedure), jos ne eivät palauta arvoa, tai funktioita (function), jos ne palauttavat arvon.

Luokan ominaisuuden valikoiva saatavuus tarkoittaa, että jonkin luokan X ominaisuus f on saatavissa f:n määrittelyn yhteydessä luetelluista luokista ja niiden perivistä luokista [M+06, s. 52–53]. Näitä luokkia kutsutaan asiakkaiksi. Tämä on oleellinen piirre X:n uudelleenkäytön kannalta, sillä sen ansiosta on koska tahansa jälkikäteen mahdollista kirjoittaa uusi luokka, joka perii ennestään olemassa olevan asiakkaan ja jolla on siten pääsy f:ään.

Eiffelin luokkahierarkialla on tärkeä rooli valikoivan saatavuuden käytettävyydessä. Luokkahierarkian juurena on luokka `ANY` ja pohjana `NONE` [Mey01, s. 15]. Toisin sanoen `ANY` on jokaisen luokan kantaluokka, ja jokainen luokka on luokan `NONE` kantaluokka. Ohjelmoijan kirjoittamat luokat asettuvat hierarkiassa edellä mainittujen luokkien väliin. Kuva 3.1 havainnollistaa hierarkiaa. Kuvan laatikot esittävät luokkia ja nuolet perinnän suuntaa.



Kuva 3.1: Eiffelin luokkahierarkia [Mey01, s. 15].

Luokkahierarkian mukaan, jos luokan ominaisuus on saatavissa luokasta `ANY`, se on saatavissa kaikkialla [M+06, s. 52]. Ominaisuuden sanotaan tällöin olevan yleisesti saatavissa. Jos luokan ominaisuus on saatavissa ainoastaan luokasta `NONE`, ominaisuus on saatavissa ainoastaan saman olion sisällä. Tällöin ominaisuuden sanotaan olevan salaisuus (secret). Muulloin ominaisuuden sanotaan olevan valikoiden saatavissa. Jos ominaisuuden esittelyn yhteydessä ei ole annettu asiakaslistaa, ominaisuus on tällöin yleisesti saatavissa.

Eiffelin luokan esittelyssä eritellään sen ilmentymän alustukseen mahdollisesti käytettävät proseduurit [M+06, s. 107–110]. Samassa yhteydessä luetellaan ne asiakkaat, joille luokan ilmentymän luonti on sallittu. Luokan sanotaan olevan näille asiakkaille luotavissa (available for creation). Jos asiakkaita ei erikseen luetella, ilmentyvä on luotavissa kaikkialla. Ilmentymän luotavuus on tällä mekanismilla eriytetty ilmentymän alustukseen käytettävien proseduurien saatavuudesta. Siten proseduuri voi olla luokan salaisuus, vaikka sitä on mahdollista käyttää luokan ilmentymän luontiin.

Kuten C++, myös Eiffel tukee moniperintää ja perittyjen ominaisuuksien saatavuuden rajoittamista. Oletuksena jokaisen perityn ominaisuuden saatavuus säilyy alkuperäisenä, ellei sitä erikseen muuteta perinnän yhteydessä [Mey01, s. 71–72]. Esimerkiksi luokan B ollessa luokan C kantaluokka C voi määrittää valituille tai kaikille B:n ominaisuuksille asiakasjoukot, joille kyseiset ominaisuudet ovat saatavissa C:n ominaisuuksina.

Java

Javan pääsynvalvontajärjestelmä mahdollistaa pääsyn rajoittamisen luokkiin ja niiden jäseniin. Tässä yhteydessä rajapintaluokat, luettelotyypit sekä annotaatiotyypit käsitetään myös luokiksi ja muodostimet luokan jäseniksi, ellei erikseen toisin mainita. Luokan ja sen jäsenten saatavuus määräytyy esittelyn yhteydessä olevien pääsymääreiden perusteel-

la. Oletussaatavuus, kun esittely ei sisällä pääsymääreitä, riippuu siitä, onko kyseessä konkreettinen luokka, rajapintaluokka, luettelotyyppi vai rajapinta.

Mahdolliset pääsymääreet ovat `public`, `protected` ja `private`, joilla määritellään kohteelle jokin neljästä saatavuustasosta [GJS+15, s. 131–132, 164–168]. Määreitä vastaavat saatavuustasot ovat ovat julkinen (public), suojattu (protected) ja yksityinen (private). Jos luokan tai sen jäsenen esittelystä jätetään pääsymääre pois, useimmiten sen saatavuustaso on tällöin pakkauksen sisäinen (package access tai package-private). Poikkeukset tähän sääntöön mainitaan jäljempänä.

Julkinen luokka tai luokan jäsen on saatavissa kaikkialla [GJS+15, s. 164–168]. Luokan yksityinen jäsen on saatavissa kaikkialla sen esittelevän päätason luokan sisällä. Pakkauksen sisäiset luokat ja jäsenet ovat saatavissa kaikista samaan pakkaukseen kuuluvista luokista.

Suojatut jäsenet ovat saatavissa aina saman pakkauksen sisällä ja seuraavien ehtojen täyttyessä myös niistä jäsenen esittelevän luokan aliluokista, jotka sijaitsevat jäsenen esittelevän luokan pakkauksen ulkopuolella [GJS+15, s. 164–170]. Olkoon C luokka, jolla on suojattu kenttä tai metodi x, ja B eri pakkauksessa sijaitseva C:n aliluokka. Pääsy B:stä x:ään on sallittu ainoastaan viitattaessa siihen B:n tai sen aliluokan jäsenenä, muttei C:n jäsenenä. Esimerkiksi jos luokassa B on esitelty muuttujat `C c; B b;`, joihin on sijoitettu sama B:n ilmentymä `c = b = new B();`, kutsu `b.x` on sallittu, mutta `c.x` ei. Toinen ehto rajoittaa suojatun muodostimen saatavuutta. C:n muodostimeen on pääsy B:stä ainoastaan muodostimesta ja anonyymiluokan luontilauseesta `super`-viittausta käyttäen. Sen sijaan C:n ilmentymän luonti suojattua muodostinta käyttäen ei ole sallittu C:n pakkauksen ulkopuolella.

Päätason luokat voivat olla julkisia tai pakkauksen sisäisiä. Luokkien jäsenten saatavuustaso voi pääsääntöisesti olla mikä tahansa. Kuitenkin rajapintaluokkien jäsenet ovat aina julkisia [GJS+15, s. 285, 288, 293–294]. Annotaatiotyypit samaistetaan tässä rajapintaluokkiin. Luettelotyyppien luettelovakiot ovat aina julkisia, ja niiden muodostimet ovat aina yksityisiä [GJS+15, s. 271–273]. Luettelotyyppien muiden jäsenten mahdollisille saatavuustasoille ei ole rajoituksia.

C#

Microsoft tarvitsi 1990-luvun loppupuolella uudelle .NET-alustalleen yleiskäyttöisen ohjelmointikielen [Ham08]. Tähän tarkoitukseen Microsoft julkaisi C#-kielen vuonna 2000 [Zan07]. Kielen juuret ovat C-kieliperheen kielissä ja se on suunniteltu olemaan yksinkertainen, moderni ja yleiskäyttöinen olio-ohjelmointikieli [Inf06, s. xvi]. Kielen yksinkertaisuudesta on tosin esitetty myös päinvastaisia näkemyksiä [Ste14, s. xxxiii]. Kielestä on olemassa sekä ECMA- [C#06] että ISO-standardi [Inf06].

C#-kielessä tyypit on jaettu kahteen pääkategoriaan: arvotyypit ja viitetyypit [C#12, s. 77–79]. Arvotyypit jakautuvat edelleen kahteen kategoriaan: tietueet (struct) ja luettelotyypit (enum). Viitetyyppejä ovat luokat (class), rajapintaluokat (interface), taulukot (array) ja delegaatit (delegate) [C#12, s. 83–84].

Tietueet ja luokat ovat monilta ominaisuuksiltaan samankaltaisia tyyppejä. Keskeisin ero niiden välillä on se, että tietueet ovat arvotyyppejä ja luokat viitetyyppejä [C#12, s. 367]. Toinen keskeinen ero on, että luokat voivat periä toisia luokkia, mutta tietueet eivät tue perintää, joskin ne voivat toteuttaa rajapintoja [C#12, s. 276–278, 369]. Tietueet ja luokat ovat myös ainoita tyyppejä, joiden jäsenten saatavuutta on mahdollista rajoittaa [C#12, s. 60–61]. Kaikkien muiden tyyppien jäsenet ovat aina julkisia. Yksinkertaisuuden vuoksi, mitä seuraavassa luokista sanotaan, koskee myös tietueita, ellei toisin mainita.

C# määrittelee viisi saatavuustasoa, jotka määrittävät luokkien ja niiden jäsenten saatavuuden: julkinen (public), suojattu (protected), sisäinen (internal), suojattu sisäinen (protected internal) sekä yksityinen (private) [C#12, s. 60–61]. Saatavuustasoja vastaavat pääsymääreet ovat `public`, `protected`, `internal`, `protected internal` ja `private`. Päätason tyyppien saatavuus voi olla ainoastaan julkinen tai sisäinen. Jos päätason tyypin saatavuutta ei aseteta pääsymääreellä, sen saatavuus on oletuksena sisäinen. Luokkien jäsenten saatavuus voi olla mikä tahansa. Suojattu ja suojattu sisäinen saatavuus eivät kuitenkaan ole tietueiden jäsenille mahdollisia, koska tietueet eivät tue perintää. Luokan jäsenen saatavuustaso on oletuksena yksityinen, ellei sitä pääsymääreellä muuksi aseteta.

Luokan yksityinen jäsen on saatavissa ainoastaan jäsenen määrittelevästä luokasta ja sen sisäkkäisistä luokista [C#12, s. 60–63]. Sisäisen saatavuustason luokka tai luokan jäsen on saatavissa sen määrittelevän jakeluyksikön (assembly) sisällä. Jakeluyksikön käsite kuvataan jäljempänä. Suojattu luokan jäsen on saatavissa jäsenen määrittelevän luokan sisällä sekä sen perivien luokkien ilmentymissä. Suojattu sisäinen luokan jäsen on saatavissa kuten suojattu jäsen, mutta myös saman jakeluyksikön sisällä. Julkinen luokka tai luokan jäsen on saatavissa kaikkialla.

Tietyt C#-kielen rakenteet vaativat, että tyypin saatavuus on vähintään yhtä laaja kuin toisen luokan tai sen jäsenen saatavuus. Yksi tällainen vaatimus on, että luokan saatavuus ei saa olla kantaluokkansa saatavuutta laajempi [C#12, s. 64–65]. Siten esimerkiksi jos kantaluokan B saatavuustaso on sisäinen, tulee B:n perivän luokan C saatavuustason olla sisäinen tai yksityinen, muttei esimerkiksi julkinen. Kuitenkin C:n jäsenet saavat olla julkisia, jolloin ne voivat olla saatavissa jonkin C:n toteuttaman rajapinnan kautta myös jakeluyksikön ulkopuolella. Toinen vaatimus on, että luokan jäsenen tyypin tai jäsenen paluuarvon ja parametrien tyyppien saatavuuden on oltava vähintään yhtä laaja kuin itse jäsenen saatavuuden. Siten esimerkiksi jos luokan C jäsenen x tyyppi on T, ja T:n saatavuustaso on sisäinen, tulee x:n saatavuustason olla sisäinen tai yksityinen, muttei esimerkiksi julkinen.

Jakeluyksikkö on .NET-ympäristöön tarkoitetun ohjelman yksikkö, joka kokoaa ohjelmiston tai sen osan käännetyt ohjelmatiedostot yhdeksi kokonaisuudeksi [Ass]. Jakeluyksikkö on joko suoritettava .exe-päätteinen tiedosto tai .dll-päätteinen dynaamisesti linkitettävä kirjastotiedosto.

Jakeluyksikkö voi määritellä yhden tai useamman jakeluyksikön toverikseen (friend). Tällöin jakeluyksikön sisäisen saatavuustason luokat ja jäsenet ovat saatavissa myös toveriyksiköistä [Fri]. Jakeluyksiköiden välinen toveruus on tarkoitettu käytettäväksi esimerkiksi yksikkötestauksen yhteydessä, kun yksikkötestit ja testattava koodi sijaitsevat eri jakeluyksiköissä.

Scala

Scala on olio- ja funktionaalisen ohjelmointiparadigman yhdistävä kieli [OR14]. Sen kehitys alkoi akateemisena tutkimusprojektina vuonna 2001 ja ensimmäinen julkinen versio julkaistiin vuonna 2004. Kielen kehitys on edelleenkin jatkunut aktiivisena. Scala on kerännyt ympärilleen laajan kehittäjäyhteisön ja kasvattanut suosiotaan myös ohjelmistoteollisuudessa. Sen suunnitteluperiaatteina on ollut muun muassa käytännönläheisyys ja ohjelmoijan tuottavuuden parantaminen, hyvä yhteentoimivuus Java-kielen kanssa sekä tehokas suorituskyky Java-virtuaalikoneessa.

Scala-kielessä käännösyksikön sisällä lohkorakenteen samalla tasolla on mahdollista esitellä samannimiset luokka ja ainokaisolio (singleton) [OAC⁺, luku 5.5]. Ainokaisolio on olio, josta ohjelman suoritusaikana on olemassa ainoastaan yksi ilmentymä. Scalassa luokan kanssa saman nimistä ainokaisoliota kutsutaan kumppanimoduuliksi (companion module) tai kumppaniolioksi (companion object), ja kumppaniolion kanssa saman nimistä luokkaa kutsutaan kumppaniluokaksi (companion class). Kumppaniolio vastaa toiminnallisuudeltaan Java-luokan staattisia osia.

Oletuksena Scala ei rajoita pääsyä luokkiin ja niiden jäseniin, vaan kaikki on julkista. Ohjelmoija voi rajoittaa pääsyä jäseniin tehden niistä yksityisiä (private) tai suojattuja (protected) käyttäen `private`- tai `protected`-määreitä [OAC⁺, luku 5.2]. Yksityiset päätason esittelyt ovat saatavissa ainoastaan saman pakkauksen sisällä [OAC⁺, luku 9.2]. Yksityiset jäsenet ovat saatavissa ainoastaan sen esittelevän luokan tai ainokaisolion (singleton object) sekä sen kumppanin sisällä [OAC⁺, luku 5.2]. Suojatut jäsenet ovat saatavissa kuten yksityiset jäsenet, mutta lisäksi myös niistä luokista, jotka perivät rajoitetun jäsenen esittelevän luokan, sekä niiden kumppanimoduuleista. Pääsymääre luokan esittelyn yhteydessä rajoittaa luokan oletusmuodostimen saatavuutta [OAC⁺, luku 5.3].

Sekä `private`- että `protected`-määreille on mahdollista antaa lisämääre muodossa *määre*`[`C`]` tai *määre*`[this]` [OAC⁺, luku 5.2]. Näistä ensimmäinen laajentaa saatavuutta sallimalla pääsyn myös kaikkialta C:n sisällä. C voi olla sama tai ulompi luokka tai

pakkaus, jonka sisällä rajoitettavan jäsenen esittely sijaitsee. Scalassa pakkaukset muodostavat hierarkian, ja siten C:n ollessa pakkaus pääsy sallitaan kaikista C:n sisältämistä pakkauksista. Lisämääreen ollessa `this` pääsy rajoitetaan luokan sijasta sen ilmentymään, eli estetään ilmentymän jäseneen pääsy määreen salliman luokan toisesta ilmentymästä. Tällaista yksityisyyttä nimitetään oliotason yksityisyydeksi (object-level privacy) tai olioyksityisyydeksi (object-privacy).

Muodostimien osalta saatavuuden määrittely on osoittautunut Scalassa vajavaiseksi. Lisämääreellä rajoitetun muodostimen semantiikka ei ole itsestään selvä, koska muodostimen aksessointi, rinnakkaisen muodostimen kutsu tai uuden ilmentymän luonti, on hyvin eri tyyppinen toimenpide kuin olion jäsenen aksessointi. Esimerkiksi miten olioyksityinen muodostin on käytännössä aksessoitavissa? Scalan spesifikaatio [OAC^{+}] ei ota erikseen kantaa lisämääreellä rajoitettujen muodostimien semantiikkaan. Se on johtanut tilanteeseen, että kääntäjä toimii niiden osalta epämääräisesti [Phi12]. Toistaiseksi ei ole varmuutta, missä vaiheessa tilanne korjaantuu.

Ceylon

Ceylon on Javasta inspiraationsa saanut saanut [s. vii][Ceyc], vuonna 2011 julkaistu [Kin11] ohjelmointikieli. Kielen kehityksen keskeisiä tavoitteita ovat olleet muun muassa soveltuvuus laajamittaiseen kehitystyöhön, kieleen sisäänrakennettu modulaarisuus, koodin luettavuus, muttei yhtä monisanainen kuin Java, sekä käännösaikainen tyyppiturvallisuus [s. vii][Ceyc]. Ceylon-kääntäjän kohdekielinä ovat Java-tavukoodi ja Javascript. Kielen kehitys tapahtuu Gavin Kingin vetämänä avoimena yhteisöprojektina, jota sponsoroi Red Hat [Ceyb].

Kielen pääsynvalvonta perustuu kahteen saatavuustasoon: jaettu (shared) ja jakamaton (unshared). Kaikki ohjelmaelementit ovat oletuksena jakamattomia [Ceyc, s. 75–76]. Kehittäjän tulee erikseen annotoida jaetuksi tarkoitetut elementit `shared`-annotaatiolla. Kuitenkaan tyyppiparametrit ja lohkon sisäiset esittelyt eivät voi olla jaettuja. Tyyppiparametrit ovat saatavissa ainoastaan sen esittelyn sisällä, johon ne kuuluvat. Lohkon sisäiset esittelyt ovat saatavissa ainoastaan kyseisen lohkon sisällä.

Kunkin jaettavissa olevan ohjelmaelementin saatavuus riippuu edellisen lisäksi myös sen esittelykontekstista [Ceyc, s. 75–76]. Oletuksena luokan tai rajapinnan jäsen on saatavissa vain sen määrittelevän luokan tai rajapinnan sisällä. Jaettu jäsen on sen sijaan saatavissa kaikkialla, missä sen määrittelevä luokka tai rajapintakin on saatavissa. Päätason esittely, esimerkiksi päätason luokka, on oletuksena saatavissa vain sen pakkauksen sisällä, jossa esittelyn käännösyksikkö sijaitsee. Päätason esittelyn ollessa jaettu se on saatavissa kaikkialla, missä esittelyn käännösyksikön sisältävä pakkauksin on saatavissa. Pakkaukset ovat oletuksena saatavissa vain saman moduulin sisällä. Jaetut pakkaukset ovat saatavissa myös niistä moduuleista, jotka tuovat (import) kyseisen pakkauksen käyttöönsä [Ceyc, s. 152–153].

Ceylonin moduuli on joukko pakkauksia, joiden nimillä on yhteinen alkuosa, koottuna yhdeksi arkistoksi [Ceyc, s. 146–153]. Moduuli toimii Ceylon-ohjelman jakeluyksikkönä. Siihen kuuluu kuvailutiedosto, jonka `import`-lauseissa luetellaan ne moduulin ulkopuoliset pakkaukset, jotka halutaan tuotavaksi moduulin sisällä käytettäväksi. Jos tuonti on jaettu eli annotoitu `shared`-annotaatiolla, tulee tuonnista transitiivinen. Siten esimerkiksi jos moduuli $m1$ tuo jaetusti pakkauksen p, ja moduuli $m2$ tuo moduulin $m1$, tuo $m2$ implisiittisesti myös p:n.

Yhteenvetona voidaan todeta, että Ceylonissa on neljä eri saatavuustasoa: yksityinen, pakkauksen sisäinen, moduulin sisäinen sekä julkinen [Ceya]. Kunkin ohjelmaelementin saatavuustaso määräytyy kolmen seikan perusteella: millainen elementti on kyseessä, onko elementti jaettu ja ovatko elementin ympäröivät elementit jaettuja.

Jaettujen esittelyiden tulee täyttää lisäksi seuraavat ehdot ollakseen kelvollisia [Ceyc, s. 76]. Elementin tyypin tulee olla saatavissa siellä, missä elementtikin on saatavissa, ja toteutetun rajapinnan tai perityn luokan tulee olla saatavissa siellä, missä toteuttava/perivä luokkakin. Lisäksi tyyppi, josta on tehty alias, on oltava saatavissa siellä, missä aliaskin on saatavissa.

Ceylonin pääsynvalvontaan saattaa olla odotettavissa vielä muutoksia. Kielen määrittelydokumentti esittää pohdinnan, tulisiko `shared`-annotaatioon lisätä määre, kuinka laajalle jakamisen vaikutus ulottuu [Ceyc, s. 76].

3.3 Pääsynvalvontamekanismien eri tyypit

Edellisessä luvussa tarkasteltiin eri ohjelmointikielten pääsynvalvontamekanismeja ja -perusteita. Näiden ohjelmointikielten saatavuusperusteet ja kunkin saatavuusperusteen mukaan rajoitettavissa olevat elementit on koottu taulukoihin 3.1 ja 3.2. Taulukoista ensimmäinen vertailee kielten julkisuuden ja yksityisyyden käsitteitä, jälkimmäinen edellisten välimaastoon asettuvia muun tyyppisiä saatavuusperusteita.

Taulukoiden riveillä ovat vertaillut kielet ja sarakkeissa rajaus, jonka sisälle saatavuus on rajoitettavissa. Taulukoiden soluissa ilmaistaan ne ohjelmaelementit, joille kyseinen saatavuusrajoite on mahdollista asettaa yksikäsitteisesti pääsymäärein elementin esittelyn yhteydessä. Sulkuihin on merkitty ne ohjelmaelementit, joiden saatavuus voi määrätyissä olosuhteissa olla sarakkeen mukainen, mutta saatavuus ei määräydy yksistään elementin esittelyssä olevien pääsymääreiden perusteella. Taulukoissa käytetyt lyhenteet selityksineen on koottu taulukkoon 3.3.

Taulukoissa ei oteta kantaa tilanteeseen, jossa ulompi yksityisempi elementti rajoittaa sisemmän julkisemman elementin saatavuuden ulomman elementin saatavuuden mukaiseksi. Esimerkiksi pakkauksen sisäisessä luokassa sijaitsevan julkisen metodin saatavuus rajoittuu todellisuudessa luokan mukaan pakkauksen sisäiseksi. Taulukoissa esimerkin mukainen metodi käsitetään julkiseksi.

Taulukko 3.1: Luvussa 3.2 vertailtujen kielten ohjelmaelementtien mahdollinen julkisuus ja yksityisyys. Taulukossa käytettyjen lyhenteiden selitykset ovat taulukossa 3.3.

Kieli	**Rajoittamaton**	**Luokka**	**Olio**
CLU	PL Mu Me	Ke	-
Smalltalk	PL Mu Me	Ke	Ke
C++	PL JL Ke Mu Me	JL Ke Mu Me	-
Eiffel	Ke Mu Me	-	Ke Mu Me
Java	PL JL Ke Mu Me	JL Ke Mu Me	-
C#	PL JL Ke Mu Me	JL Ke Mu Me	-
Scala	PL JL Ke Mu Me	Ke Mu Me	Ke Mu Me
Ceylon	Mo (Pa PL JL Ke Mu Me)	JL KE Mu Me	-

Taulukko 3.2: Luvussa 3.2 vertailtujen kielten ohjelmaelementtien muu mahdollinen saatavuus kuin julkinen ja yksityinen. Taulukossa käytettyjen lyhenteiden selitykset ovat taulukossa 3.3.

Kieli	**Mod.tyyppi**	**Moduuli**	**Aliluokat**	**Muuta**
CLU	-	-	-	-
Smalltalk	allas	Ke	-	-
C++	-	-	JL Ke Mu Me	PS To
Eiffel	-	-	Ke Mu Me	TV
Java	pakkaus	PL JL Ke Mu Me	JL Ke Mu Me	-
C#	jakeluyksikkö	PL JL Ke Mu Me	JL Ke Mu Me	To
Scala	pakkaus	Ke Mu Me	Ke Mu Me	PH
Ceylon	pakkaus	PL (JL KE Mu Me)	-	EY
	moduuli	Pa (PL JL Ke Mu Me)		

Seuraavissa aliluvuissa tehdään vertailua eri kielten pääsynvalvontamekanismien välillä ja tulkitaan tarkemmin edellä mainittujen taulukoiden sisältöä. Pääsynvalvontamekanismit on luokiteltu kuuden eri perusteen mukaan: yksityisyys ja julkisuus, moduulit, luokkahierarkia, tyyppihierarkia, toveruus sekä esittelykonteksti.

Yksityisyys ja julkisuus

Kaikki luvussa 3.2 esitellyt kielet tekevät selkeän rajauksen rajoittamattoman julkisen saatavuuden ja yksityisyyden välillä. Yksityisyyden käsite vaihtelee hieman kielestä riippuen: yksityisyys saattaa tarkoittaa luokan tai olion sisäistä yksityisyyttä tai molempia.

Taulukko 3.3: Taulukoissa 3.1 ja 3.2 käytetyt lyhenteet aakkosjärjestyksessä ja niiden selitykset.

Lyhenne	**Selite**
-	saatavuusperuste ei sovellettavissa
EY	esittely-ympäristöperusteinen saatavuus
JL	jäsenluokka
Ke	kenttä
Me	metodi
Mo	moduuli
Mu	muodostin
Pa	pakkaus
PH	pakkaushierarkiaan perustuva saatavuus
PL	päätason luokka tai luokka, jos kieli ei tue sisäkkäisiä luokkia
PS	perintäsaatavuus, rajoittaminen perinnän yhteydessä
To	toverisaatavuus
TV	tyyppiperusteinen valikoiva saatavuus

Taulukko 3.1 kuvaa kyseisten kielten yksityisyys- ja julkisuuskäsitteitä. Sarakkeessa Rajoittamaton luetellaan elementtityypit, joiden saatavuus on asetettavissa rajoittamattomaksi eli julkiseksi. Yksityisyys on taulukossa jaettu kahteen sarakkeeseen: luokkayksityisyys (Luokka) ja olioyksityisyys (Olio).

Kaikki taulukon ohjelmointikielet mahdollistavat kaikkien elementtiensä julkisuuden. Kielistä kaikki paitsi Eiffel mahdollistavat ainakin joidenkin ohjelmaelementtien saatavuuden rajoittamisen luokan sisäiseksi. Olioyksityisyyden mahdollistavat ainoastaan Smalltalk, Eiffel ja Scala. Kaikki sallivat yksityiset kentät. Näistä CLUta ja Smalltalkia lukuun ottamatta kaikki sallivat myös yksityiset muodostimet ja metodit. Jäsenluokkien yksityisyyden mahdollistavat kaikki niitä tukevat kielet Scalaa lukuun ottamatta.

Moduuliperusteinen saatavuus

Jotta voidaan puhua moduuliperusteisesta saatavuudesta, on ensiksi määriteltävä moduulin käsite tässä yhteydessä. Sanalla moduuli tarkoitetaan yleistäen itsenäistä yksikköä, joita yhdistelemällä saadaan aikaiseksi suurempi kokonaisuus [Mer]. Tämän määritelmän mukaan esimerkiksi luokka on moduuli. Jos ohjelmointikieli sallii luokan jäsenen saatavuuden rajoittamisen luokan sisäiseksi, on kyse moduuliperusteisesta saatavuudesta.

Käytännössä luokkia ei kutsuta ohjelmointikielissä moduuleiksi, vaan kielet tyypillisesti määrittelevät luokkaa laajempia yksiköitä, jotka täyttävät moduulin määritelmän. Ohjelmointikielestä riippuen niitä kutsutaan esimerkiksi pakkauksiksi, jakeluyksiköiksi tai

moduuleiksi. Taulukossa 3.2 kaikkia tällaisia komponentteja kutsutaan yleisnimellä moduuli. Taulukon Moduuli-sarakkeeseen on koottu ne ohjelmaelementit, joiden saatavuus on mahdollista rajoittaa moduulin sisäiseksi. Mod.tyyppi-sarakkeessa ilmaistaan, minkä tyyppisiä luokkaa laajempia moduuleja kieli tukee.

Vertailluista kielistä CLU, C++ ja Eiffel eivät tue moduuliperusteista, eli yksityisyyttä laajemman ohjelmayksikön sisäistä saatavuutta laisinkaan. Smalltalkissa on kentille oma moduulimainen saatavuutta rajoittava yksikkö. Muut kielet sallivat kaikkien ohjelmaelementtien saatavuuden rajoittamisen moduulin sisään, poikkeuksena Scala, jonka luokkien saatavuutta ei pysty rajoittamaan moduulin sisäiseksi.

Javan, Scalan ja Ceylonin moduulit ovat pakkauksia. Kielten pakkauskäsitteen semantiikassa on eroja. Javassa pakkaukset ovat itsenäisiä ilman yhteyttä toisiin pakkauksiin. Javassa ei esimerkiksi ole pakkaushierarkian käsitettä, vaikka pakkaukset nimetäänkin hierarkkisesti [GJS+15, s. 176]. Sen sijaan Scalassa pakkaukset voivat olla sisäkkäisiä ja muodostaa hierarkkisen rakenteen [OAC+], kuten taulukon Muuta-sarakkeessa on mainittu. Se vaikuttaa myös saatavuuteen: pakkauksen sisällä saatavissa olevat ohjelmaelementit ovat saatavissa myös alipakkauksista. Ceylon määrittelee pakkaukset Javan tapaan [Ceyc].

Ceylon on vertailluista kielistä on ainoa, jossa on pakkausten lisäksi erillinen moduulien käsite. Moduulit kokoavat sisään tietyn nimihierarkian mukaiset pakkaukset. Kaikkien ohjelmaelementtien saatavuus on mahdollista rajoittaa moduulin sisään, myös pakkausten. C#:n moduulit ovat jakeluyksiköitä, jotka ovat semanttisesti lähimpänä Javan jar-tiedostoja ja Ceylonin moduuleja.

Smalltalkin muuttuja-altaat on taulukossa rinnastettu moduuleiksi, vaikka ne eivät olekaan edellä esitetyn määritelmän mukaan moduuleja. Sen sijaan muuttuja-allas on mekanismi ryhmitellä muuttujia ryhmiksi, joiden sisältämät muuttujat ovat saatavissa kaikissa kyseistä allasta käyttävissä olioissa. Altaiden rinnastaminen taulukossa moduuleihin on tehty tämän ryhmittelyominaisuuden perusteella, koska moduuleillakin on sama ominaisuus.

Perintäperusteinen saatavuus

Luokkaperintä on yksi olio-ohjelmoinnin keskeisimmistä käsitteistä. Vertailluista kielistä suurin osa mahdollistaa perintähierarkiaan perustuvan pääsynvalvonnan. Taulukon 3.2 Aliluokat-sarakkeessa esitetään ohjelmaelementit, joiden saatavuutta on mahdollista rajoittaa perintähierarkian mukaan. Luokkien perintähierarkiaan perustuvaa saatavuutta nimitetään tutkielmassa perintäperusteiseksi saatavuudeksi.

Taulukon kielistä CLU, Smalltalk ja Ceylon eivät salli perintäperusteista pääsyn rajoittamista. CLU ei mahdollista perintää laisinkaan. Smalltalkissa ainoastaan kenttien saatavuutta on mahdollista rajoittaa, mikä tapahtuu muuttuja-altaisiin perustuen. Ceylonissa on tehty tietoinen valinta olla toteuttamatta perintäperusteista saatavuutta [Ceyc, s. 6].

Kaikki muut taulukon kielet mahdollistavat kaikkien luokan jäsenten saatavuuden rajoittamisen luokkahierarkian mukaan esittelevän luokan aliluokille. Jäsenluokkia tukevat kielet C++, Java ja C# mahdollistavat jäsenluokkienkin saatavuuden rajoittamisen. Scalassa luokat, myös jäsenluokat ovat aina julkisia.

Javassa perintäperusteisesti saatavissa oleva elementti on aina implisiittisesti saatavissa myös saman moduulin, Javan käsittein pakkauksen, sisällä. Muissa kielissä tällaista sidosta eri saatavuusperusteiden välillä ei ole.

Eiffelissä ei ole erillistä määrettä tai konstruktiota ilmaisemaan saatavuutta esittelevän luokan periville luokille. Sen sijaan ominaisuus on osa Eiffelin tyyppiperusteista valikoivaa saatavuutta, jota tarkastellaan seuraavassa aliluvussa.

Perintäperusteiseen saatavuuteen liittyy olennaisesti myös perintäsaatavuus, eli saatavuuden rajoittaminen perinnän yhteydessä. Vertailluista kielistä C++ mahdollistaa ainoana perintäsaatavuuden.

Tyyppiperusteinen valikoiva saatavuus

Muista luvussa 3.2 esitellyistä kielistä poiketen Eiffelin pääsynvalvonta perustuu tyyppiperusteiseen valikoivaan saatavuuteen, mikä on ilmaistu taulukon 3.2 Muuta-sarakkeessa. Sen sijaan, että saatavuuden rajoittaminen tapahtuisi ohjelman modulaarisen rakenteen mukaan, rajoittaminen tapahtuu rajoitettavan elementin valitsemien tyyppien ja niiden perintähierarkian perusteella. Tätä kutsutaan valikoivaksi saatavuudeksi, koska rajoitettava elementti valitsee ohjelman osat, niin kutsutut asiakkaansa, joista se on saatavissa. Koska valinnan perusteena on asiakkaan tyyppi, puhutaan tyyppiperusteisesta valikoivasta saatavuudesta.

Oletuksena Eiffelin luokkien jäsenet ovat saatavissa ainoastaan saman olion sisällä. Saatavuus on laajennettavissa niin, että määrätyt jäsenet ovat saatavissa jäsenen esittelyn yhteydessä asiakkaiksi määritellyistä luokista. Eiffelissä tyyppiperusteinen valikoiva saatavuus on myös perintäperusteinen, koska pääsy luokan jäseneen sallitaan myös sen asiakkaiden perivistä luokista.

Julkinen, rajoittamaton saatavuus saavutetaan tyyppiperusteisen valikoivan saatavuuden avulla määrittelemällä asiakasjoukoksi kaikkien luokkien kantaluokan. Perintäperusteinen saatavuus jäsenen esittelevän luokan perivistä luokista saavutetaan määrittelemällä esittelevä luokka jäsenen asiakkaaksi. Yksityisyys saavutetaan, kun asiakasjoukoksi määritellään tyhjä joukko. Tyhjää joukkoa voidaan kuitenkin pitää epäintuitiivisena [Mey97, s. 192–193]. Ratkaisuna on mahdollista käyttää tyhjän joukon sijaan tarkoitukseen määriteltyä luokkaa, joka ei sisällä erityistä toteutusta, ja jota ei voi periä. Eiffelissä tällainen luokka on `NONE`.

Toveruus

Ohjelmayksikkö, esimerkiksi luokka, esitellessään toisen ohjelmayksikön toverikseen sallii tälle pääsyn omiin rajoitettuihin, esimerkiksi yksityisiin, elementteihinsä. Taulukossa 3.2 vertailluista kielistä C++ ja C#, tarkemmin sen suoritusympäristö .NET, mahdollistavat toveruuden. Näiden toveruuskäsite tosin on aivan eri tasolla: C++:ssa toveriksi määritellään luokka tai funktio, jolle annetaan pääsy esittelevän luokan kaikkiin yksityisiin jäseniin, kun taas .NET-ympäristössä toveriksi määritellään jakeluyksikkö, jolle annetaan pääsy esittelevän jakeluyksikön sisäisen näkyvyyden ohjelmaelementteihin.

Toveruus muistuttaa valikoivaa saatavuutta. Molemmille yhteistä on se, että esittelevä ohjelmayksikkö antaa valikoiden pääsyn toiselle ohjelmayksikölle. Toveruus on näin määriteltävissä valikoivan saatavuuden yhdeksi muodoksi. Merkittävin ero edellä mainittujen kielten toveruuskäsitteessä ja aiemmin käsitellyn tyyppiperusteisen valikoivan saatavuuden välillä lienee se, että jälkimmäinen sallii pääsyn myös sallitut luokat perivistä tyypeistä, kun taas toveruus ei periydy.

Myös sisäkkäisillä luokilla on toveruuden kanssa yhteisiä piirteitä. Molemmista on pääsy ulomman tai toveriksi esitelleen luokan yksityisiin jäseniin. Merkittävin ero toveruuden ja sisäkkäisten luokkien välillä on rakenteessa: sisäkkäinen luokka on nimensä mukaisesti esiteltävä ulomman luokkansa sisällä ja voi siten olla vain yhden päätason luokan sisällä. Sen sijaan taas toveri on toveriksi esittelevästä luokasta erillinen ohjelmayksikkö ja voi olla useankin luokan toveri.

Esittely-ympäristösidonnainen moduuliperusteinen saatavuus

Kaikki luvussa 3.2 vertaillut kielet yhtä lukuun ottamatta määrittelevät pääsymääreillään yksikäsitteisesti, millaisen ohjelmayksikön, kuten luokka tai pakkaus, sisällä rajoitettava elementti on saatavissa. Poikkeuksen sääntöön tekee Ceylon, mikä käy ilmi taulukoista 3.1 ja 3.2. Niissä on merkitty sulkuihin sellaiset ohjelmaelementit, joiden saatavuus voi määrätyissä olosuhteissa olla sarakkeen mukainen, mutta saatavuus ei määräydy yksistään elementin esittelyssä olevien pääsymääreiden perusteella.

Ceylonissa kukin elementti esitellään yhdellä pääsymääreellä joko jaetuksi tai jakamattomaksi. Jakamaton elementti on saatavissa ainoastaan esittely-ympäristössään, jaettu elementti kaikkialla missä esittely-ympäristönsäkin on. Elementin konkreettinen saatavuus riippuu siten esittely-ympäristön saatavuudesta, mistä voidaan käyttää nimitystä esittely-ympäristösidonnainen saatavuus.

Vaikka konkreettista saatavuuden laajuutta ei ilmaista pääsymääreessä, modulaarisuuteen perustuvat saatavuustasot yksityinen, moduulin sisäinen ja julkinen ovat mahdollisia saavuttaa myös esittely-ympäristöstä riippuvaisesti. Luokan jäsen on luokan sisäisesti yksityinen sen ollessa jakamaton. Jäsenen saatavuus on moduulin sisäinen, kun se on

jaettu, mutta sen määrittelevä luokka ei ole. Jäsenen saatavuus on julkinen, kun jäsen, sen määrittelevä luokka ja kaikki ulommat luokkaa laajemmat moduulit ovat jaettuja.

Esittely-ympäristöön perustuva saatavuuden voidaan edellä esitettyyn perustuen todeta olevan käsite, joka ei määrittele, mihin ohjelmointikielen piirteeseen perustuen ohjelmaelementti voi olla saatavissa. Sen sijaan käsite määrittelee, minkä eri tekijöiden yhteisvaikutuksesta ohjelmaelementin konkreettinen saatavuus määräytyy. Ceylonin tapauksessa voidaan puhua esittely-ympäristösidonnaisesta moduuliperusteisesta saatavuudesta, koska saatavuus perustuu ohjelman modulaariseen rakenteeseen.

4 Valinnaiset tyyppijärjestelmät

Perinteisesti tyyppijärjestelmät ovat olleet keskeinen osa ohjelmointikieliä. Voidaan sanoa, että ne ovat pakollinen osa kieltä. Tässä luvussa tarkastellaan, millaisia vahvuuksia tyyppijärjestelmillä on ja millaisia ongelmia tyyppijärjestelmien pakollisuus tuo tullessaan. Valinnaisia ja kytkettäviä tyyppijärjestelmiä on ehdotettu ratkaisuna pakollisten tyyppijärjestelmien ongelmiin. Luvussa kartoitetaan, millaiset valmiudet Javassa on kytkettävän tyyppijärjestelmän toteuttamiseksi ja tutustutaan kytkettävään tyyppijärjestelmäkehykseen Checker Frameworkiin.

4.1 Staattisten tyyppijärjestelmien vahvuuksia ja ongelmia

Staattisten tyyppijärjestelmien edut tunnetaan laajalti, mutta niiden ongelmat jäävät monesti pimentoon. Tässä luvussa tarkastellaan niiden vahvuuksia ja ongelmakohtia.

Staattisen tyypityksen vahvuuksia

Staattisella tyyppijärjestelmällä on useita tunnettuja hyviä puolia. Staattinen tyypitys takaa, että tyyppijärjestelmän vastaiset epäjohdonmukaisuudet ohjelmassa havaitaan jo käännösaikana. Yleisesti virheiden havaitsemista ajoissa pidetään hyvänä asiana, koska virheiden korjaaminen on silloin yleensä yksinkertaisempaa. Staattisen tyyppijärjestelmän onkin havaittu ainakin Javan kaltaisissa kielissä helpottavan tyyppivirheiden korjaamista [HKR+13]. Sen sijaan semanttisten virheiden ehkäisemisessä ja korjaamisessa staattisesta tyyppijärjestelmästä ei ole havaittu olevan apua.

Kun ohjelma on tyyppijärjestelmän kannalta todettu käännösvaiheessa oikeelliseksi, suoritusaikaiset tyyppitarkastukset on mahdollista jättää pois, mikä parantaa ohjelman suorituskykyä. Kaikkia suoritusaikaisia tarkastuksia ei kuitenkaan ole tavallisesti mahdollista jättää pois [Car97]. Esimerkiksi taulukoiden indeksointia ei pysty käännösaikana päättelemään, vaan se on tarkastettava suoritusaikana.

Ohjelman staattinen tyypitys muodostaa implisiittisesti koneellisesti tarkastettavan dokumentaation ohjelmasta, minkä on havaittu edesauttavan oikean tyypin paikantamista lähdekoodissa [HKR+13]. Tyypitys muodostaa ohjelmoijalle konseptuaalisen kehyksen, joka edesauttaa ohjelman suunnittelua, ylläpitoa ja ohjelman ymmärtämistä [Bra04].

Staattisen tyypityksen ongelmia

Tunnetuista eduistaan huolimatta staattiset tyyppijärjestelmät eivät ole ongelmattomia. Staattisten ja dynaamisten tyyppijärjestelmien eduista, ongelmista ja paremmuudesta on väitelty jo kauan [Car97], eikä lopullista ratkaisua ole edelleenkään löydetty [HKR+13].

Staattisten tyyppijärjestelmien ongelmakohdaksi on muodostunut tyyppijärjestelmän ilmaisuvoiman riittämättömyys. Joissakin tilanteissa tyypit saattavat olla liian sallivia, toisissa taas liian rajoittavia [Tra09]. Esimerkkinä tyyppijärjestelmän liiallisesta sallivuudesta Javan kääntäjä ei pysty staattisesti estämään jakoa int-tyypin arvolla nolla, vaikka staattisen tyyppijärjestelmän näkökulmasta jakolasku on oikein [Tra09]. Staattisen tyyppijärjestelmän ilmaisukyvyn rajoittuneisuus puolestaan johtaa esimerkiksi yksikäsitteisten tyyppimuunnosten (explicit type cast) tarpeeseen. Yksinkertaisissa ohjelmointitehtävissä tyyppimuunnosten tarpeen on havaittu hidastavan ohjelman kehitystä verrattuna dynaamiseen tyyppijärjestelmään, jossa yksikäsitteisiä tyyppimuunnoksia ei tarvita [SH11]. Staattisen tyyppijärjestelmän ilmaisuvoiman kasvattamisen esteeksi muodostuu tyyppijärjestelmien monimutkaisuus. Staattiset tyyppijärjestelmät ovat jo ennestään varsin monimutkaisia, ja käytännössä ilmaisuvoiman pienikin kasvattaminen kasvattaa merkittävästi tyyppijärjestelmän monimutkaisuutta [Mac93][MD04].

Staattisen tyypityksen etuna pidetyllä vaatimuksella ohjelman oikeellisuudesta tyyppijärjestelmän näkökulmasta on ongelmalliset puolensakin. Lähes kaikki ohjelmistojärjestelmät muuttuvat usein jatkuvasti, ja harvoin suunnitellusti ja kurinalaisesti alkuperäisen kehitystyön jälkeen. Siten järjestelmän implisiittisenä vaatimuksena voi pitää, että ohjelmisto mahdollistaa sen muuttumisen. Siitä seuraa, että käytetyn ohjelmointikielen tulee mahdollistaa muutokset. Monesti on toivottavaa tehdä muutokset ohjelmistoon pienissä osissa, jotta mahdollisesti syntyneet virheet on helppo yhdistää tehtyyn muutokseen. On tyypillistä voida olettaa, että muutoksen keskellä oleva ohjelma ei toimi kokonaisuutena oikein. Staattiset tyyppijärjestelmät vaativat ohjelmiston olevan aina tyypeiltään oikeellinen ja estävät siten tämän tyyppisen kehityksen, koska tyyppijärjestelmää ei pysty väliaikaisesti kytkemään pois. Staattisen tyyppijärjestelmän voi väittää johtavan järjestelmän ennenaikaiseen luutumiseen (ossification), koska muutosten tekeminen on hankalampaa [Tra09].

Staattisten tyyppijärjestelmien oikeellisuus ja monimutkaisuus

Tyyppijärjestelmä ja sen merkitys ohjelmointikielessä on hyvin keskeinen. Kun kielessä on staattinen tyyppijärjestelmä, sen antamiin suoritusaikaisiin tyyppiturvallisuustakuisiin on helppo luottaa. Hyvä tyyppijärjestelmä on formaalisti todistettu vankaksi ja täydelliseksi. Käytännössä todelliset järjestelmät ovat liian monimutkaisia formalisoitavaksi [Bra04]. Siksi formalisointi on tyypillisesti tehty tyyppijärjestelmän osajoukolle tai tyyppijärjestelmästä on tehty sitä yksinkertaistavia oletuksia [DE97][NvO98][DEK99][CGLO06]. Jos yksinkertaistus onkin virheellinen, ei sitä käyttävä todistus tyyppijärjestelmän oikeellisuudesta ole pätevä.

Staattiset tyyppijärjestelmät ovat monimutkaisia, usein jopa kielen määrittelyn monimutkaisin osa [Tra09]. Jotta tyyppijärjestelmään voisi täysin luottaa, ei riitä, että tyyppijärjestelmän määrittely on oikeellinen, vaan myös sen toteutuksen tulee olla virheetön. Monimutkaisuudesta johtuen ei ole mahdottomuus, että ne sisältävät virheitä. Tyyppijärjes-

telmässä oleva virhe saattaa johtaa ohjelman mahdottomaksi oletettuun suoritusaikaiseen toimintaan [Car97].

Yksi tunnettu esimerkki tyyppijärjestelmässä olleesta virheestä on Eiffel-kieli [Mey85]. Siinä on sallittua, että metodi, joka korvaa perityn luokan metodin, määrittelee parametreikseen korvatun metodin parametrien alityyppejä. Esimerkiksi oletetaan, että luokalla `BaseClass` on metodi `method`, jonka ainoan parametrin tyyppi on `ParamType`. Luokka `InheritingClass` perii `BaseClass`:n ja korvaa metodin `method`. Kielen sääntöjen mukaan korvaavan metodin on sallittua määrittää sen parametrin tyypiksi `ParamSubtype`, kun luokka `ParamSubtype` perii `ParamType`-luokan. Kutsuttaessa `InheritingClass`-luokan `method`-metodia luokan `BaseClass` edustajana, on tyyppijärjestelmän mukaan sallittua antaa metodin parametrina `ParamType`-luokan ilmentymä. Koska korvaava metodi on kuitenkin määritellyt parametrin tyypin uudelleen `ParamSubtype`ksi, kutsu johtaa virheeseen [Coo89].

Tyyppijärjestelmässä havaitut virheet ovat erityisen ongelmallisia, koska niiden korjaaminen on usein mahdotonta säilyttäen yhteensopivuuden ennestään olemassa oleviin ohjelmiin [Tra09]. Siksi edellä mainittu Eiffelin tyyppijärjestelmän virhe on edelleen kielessä, vaikka sen merkitystä on jossain määrin pystytty vähentämään.

Yleisesti tyyppijärjestelmien ajatellaan edesauttavan luotettavuutta ja tietoturvaa todistamalla mekaanisesti ohjelman ominaisuuksia. Toisaalta tyyppijärjestelmien on väitetty vahingoittavan luotettavuutta ja tietoturvaa tekemällä asioista monimutkaisia ja hauraita [Bra03]. Kielen turvallisuus on käytännössä yhtä vahva kuin sen heikoin lenkki. Jos tyyppijärjestelmä tai sen toteutus pettää, seuraukset ovat määrittelemättömät [Bra04].

4.2 Ratkaisumalleja tyyppijärjestelmien ongelmiin

Staattisen tyypityksen ongelmien ratkaisemiseksi on muutettava ajattelutapaa. Sen sijaan, että asetettaisiin staattinen ja dynaaminen tyypitys vastakkain, tulisi tyyppijärjestelmät jakaa pakollisiin ja valinnaisiin [Bra04]. Staattisten tyyppijärjestelmien ongelmat johtuvat pääosin niiden tiukasta sidonnaisuudesta ohjelmointikieleen.

Ratkaisuksi pakollisten tyyppijärjestelmien ongelmiin on esitetty valinnaista tyypitystä. Se mahdollistaa staattisen ja dynaamisen tyypityksen parhaiden puolien yhdistämisen [Bra04]. Tässä luvussa tarkastellaan valinnaisen tyypityksen ominaisuuksia.

Valinnainen tyypitys

Ratkaisuksi pakollisten tyyppijärjestelmien ongelmiin on esitetty valinnaista tyypitystä [Bra04]. Valinnaisen tyyppijärjestelmän määritelmä ei vielä ole vakiintunut. Yksinkertainen tapa määritellä valinnainen tyyppijärjestelmä kuvailla se mahdollisuudeksi lisätä

ohjelmoijan tai kääntäjän tyyppipäättelijän toimesta ohjelman valikoituihin osiin, staattisia tyyppejä, jotka kääntäjä tarkastaa käännösaikaisesti [Tra09]. Määrittely ei ota kantaa siihen, tulisiko valinnaisesti käyttöön otettavan tyyppijärjestelmän vaatia koko ohjelman olevan staattisesti tyypitetty [THF08], vai onko staattisen ja dynaamisen tyypityksen suhde vapaasti valittavissa [SV08]. Määrittely jättää myös avoimeksi, onko ohjelman tyyppirikkomuksia käsiteltävä ehdottomina virheitä, kuten staattisesti tyypitetyissä kielissä, vai antavatko ne aihetta korkeintaan varoittaa ohjelmoijaa [Tra09]. Yhdenmukaista käsitystä ei ole syntynyt siitäkään, saavatko valinnaisen tyyppijärjestelmän staattiset tyypit vaikuttaa ohjelman suoritusaikaiseen semantiikkaan. Esimerkiksi ohjelmakoodin lisäoptimointi olisi mahdollista staattisesti tyypitetyille osille, jos erilainen suoritusaikainen semantiikka sallitaan.

Valinnaisen tyypityksen määrittelyksi on myös esitetty, että valinnaisen tyyppijärjestelmän ei tule vaikuttaa kielen suoritusaikaiseen semantiikkaan eikä vaatia tyyppiannotaatioita syntaksissaan [Bra04]. Näistä kahdesta vaatimuksesta ensimmäistä voidaan pitää merkittävimpänä, mutta myös ankarampana vaatimuksena. Sen seurauksena tyyppijärjestelmä ei esimerkiksi voi mahdollistaa metodien kuormittamista tyyppien perusteella, koska ilman tyyppitietoa on mahdotonta valita oikea suoritettava metodi. Toisaalta tiukat vaatimukset mahdollistavat sen, että ohjelmassa on mahdollista käyttää useita eri tyyppijärjestelmiä.

Perinteisesti ohjelmointikieli ja sen pakollinen tyyppijärjestelmä ovat kaksisuuntaisesti toisistaan riippuvaisia: ohjelmointikieli riippuu tyyppijärjestelmästä ja tyyppijärjestelmä ohjelmointikielestä. Jos tyyppijärjestelmästä tehdään valinnainen, kieli ei ole enää riippuvainen tyyppijärjestelmästä, ja riippuvuudesta tulee yhdensuuntainen. Tällöin, tyyppijärjestelmän on mahdollista kehittyä nopeammin kuin kielen [Bra04]. Siten aiemmin tyypittämättömästä ohjelmasta on myöhemmin mahdollista tehdä tyyppitarkastettu. Silti ohjelman toiminnan voidaan taata olevan sama kuin aiemmin, edellyttäen, että tyyppijärjestelmä ei vaikuta ohjelman suoritusaikaiseen semantiikkaan.

Kytkettävä tyypitys ja tyyppipäättely

Kun tyyppijärjestelmästä on tehty valinnainen, eikä suoritusympäristö ole riippuvainen siitä, tyyppijärjestelmää on mahdollista käsitellä kielen liitännäisen (plug-in) tavoin [Bra04]. Kieltä voi tällöin käyttää sellaisenaan tai siihen voi kytkeä yhden tai useamman tarkoitukseen sopivan tyyppijärjestelmän.

Kytkettävän tyyppijärjestelmän toteuttaminen ohjelmointikieleen asettaa kielelle tiettyjä vaatimuksia. Kielen rakenteisiin tulee pystyä liittämään tyyppitiedon sisältävää metadataa, jonka tulee olla käsiteltävissä abstraktiin syntaksipuun kaikissa solmuissa (node) [Bra04]. Annotaatiot ovat tähän yksi mahdollinen ratkaisu, mikäli kieli tukee niitä riittävän laajalti. Muita kytkettävien tyyppijärjestelmien toistaiseksi ratkomattomia kysymyksiä on, kuinka ratkotaan tyyppijärjestelmän nimiavaruuksien ja eri versioiden hallinta [Bra04]. Lisäksi

lukuisten annotaatioiden aiheuttama syntaktinen sotku voi muodostua käytännön esteeksi kytkettävän tyyppijärjestelmän käyttöönotolle.

Annotointiin liittyviä ongelmia on mahdollista pyrkiä vähentämään tyyppipäättelyn avulla. Tyyppipäättelyn tulisi kuitenkin olla valinnainen siinä missä tyyppijärjestelmänkin, eikä tyyppijärjestelmän tulisi riippua tyyppipäättelystä [Bra04], koska tyyppijärjestelmään kuuluva pakollinen tyyppipäättely lisää tyyppijärjestelmän monimutkaisuutta. Jos tyyppipäättelykin on kytkettävä, tyyppipäättelijöitä voi samanaikaisesti olla käytössä useita erilaisia, joista kukin voi toteuttaa vain yhden menetelmän suorittaa tyyppipäättely.

Ideaalitilanteessa kytkettävän tyyppijärjestelmän ja tyyppipäättelyn toteutukset integroituvat kehittäjän jo käyttämiin työkaluihin, esimerkiksi sovelluskehittimeen tai kielen kääntäjän, eikä erillisiä työkaluja tai muunneltua kääntäjä tarvita lainkaan.

4.3 Javan valmiudet kytkettävän tyyppijärjestelmän toteuttamiseen

Jotta ohjelmointikieleen olisi mahdollista toteuttaa kytkettävä tyyppijärjestelmä, kielen rakenteisiin tulee pystyä liittämään tyyppitiedon sisältävää metadataa, jonka tulee olla käsiteltävissä abstraktiin syntaksipuun kaikissa solmuissa. Lisäksi on eduksi, jos tyyppitarkastukset ovat integroitavissa kielen perustyökaluihin, tyypillisesti kääntäjään.

Javan annotaatiot mahdollistavat metadatan lisäämisen ohjelmaelementteihin, jonka Java-kääntäjä tallentaa abstraktiin syntaksipuuhun. Metadatan käännösaikainen analysointi tapahtuu Java-kääntäjään kytkettäviä annotaatioprosessoreita käyttäen. Tässä luvussa perehdymme näihin Javan välineisiin.

Annotaatiot

Metadatalla tarkoitetaan tietoa joka kuvaa tietoa. Javan annotaatiot ovat metadataa, jotka ilmaisevat tietoa ohjelmasta, mutta eivät ole osa ohjelmaa itseään [Jav15]. Koska annotaatiot ovat vain metadataa, ei niillä sellaisenaan ole vaikutusta ohjelman suoritukseen. Sen sijaan annotaatioiden tarkoitus on toimia datana erilaisille työkaluille, jotka esimerkiksi käännösaikaisesti tai käännettyjä luokkatiedostoja lukemalla analysoivat ohjelmaa tai generoivat luokka- tai datatiedostoja annotaatioiden perusteella. Erilaiset sovelluskehykset tai ohjelma itse voivat suoritusaikaisesti lukea annotaatioita Javan reflektiorajapintaa käyttäen.

Annotaatiot tulivat osaksi Java-kieltä sen versiossa 5 [Jav16c]. Annotaatioita oli mahdollista liittää kaikkiin esittelyihin: luokan, rajapintaluokan, annotaation, luettelotyypin, kentän, metodin, muodollisten parametrien, muodostimen sekä paikallisen muuttujan esittelyyn [GJS+15, s. 304]. Lisäksi annotaatioita pystyi liittämään pakkausten kuvaustiedostossa package-info.java sijaitsevaan pakkauksen esittelyyn. Kytkettävän tyyppijärjestelmän

kannalta lista on puutteellinen, sillä tyyppejä käytetään muuallakin kuin esittelyjen yhteydessä. Javan versiossa 8 kieleen lisättiin mahdollisuus liittää annotaatioita edellisten lisäksi myös tyyppiparametrien yhteyteen sekä kaikkiin muihin yhteyksiin, missä tyyppeihin viitataan [Jav15]. Nämä niin kutsutut tyyppiannotaatiot mahdollistavat kytkettävän tyyppijärjestelmän toteuttamisen Javaan käyttäen ainoastaan kielen omia konstruktioita.

Annotaatiotyyppi on tyyppimäärittely, millaista dataa annotaatio liittää annotoituun ohjelmaelementtiin. Tyypillisimmin ohjelmoija käyttää kieleen sisäänrakennettuja tai ulkoisten kirjastojen määrittelemiä annotaatiotyyppejä. Esimerkkinä Javan sisäänrakennetuista annotaatiotyypeistä mainittakoon `@SuppressWarnings`. Kääntäjät ja siihen mahdollisesti liitetyt työkalut tekevät monenlaisia tarkastuksia käännettävälle ohjelmalle havaitakseen virheitä mahdollisimman aikaisessa vaiheessa. Toisinaan jotkut niiden tuottamista varoituksista osoittautuvat aiheettomiksi. `@SuppressWarnings` on tarkoitettu vaientamaan aiheettomat varoitukset [GJS+15, s. 307–308]. Kääntäjä tai muu annotaatiota tukeva työkalu jättää varoittamatta annotoidun elementin sisällä havaitsemistaan ongelmista. Käännösvirheiden ohittamista annotaatio ei kuitenkaan mahdollista.

Annotaatiot sisältävät muutakin dataa niiden tyypin lisäksi, mikä annetaan annotaation nimettyinä parametreina. Esimerkiksi `@SuppressWarnings`-annotaatiotyyppi esittelee parametrin `value`. Sen arvoksi annetaan kääntäjän tai työkalun määrittelemä merkkijono, joka kertoo, minkä tyyppiset annotoidun elementin sisällä havaitut varoitukset tulisi jättää ilmoittamatta [GJS+15, s. 307–308]. Esimerkiksi `@SuppressWarnings(value = "deprecation")` kertoo Java-kääntäjälle, että sen tulisi jättää varoittamatta `@Deprecated`-annotaatiolla merkittyihin ohjelmaelementteihin kohdistuvista kutsuista. Annotaation eri parametrit erotetaan toisistaan pilkulla samalla tavoin kuin esimerkiksi metodikutsun muodolliset parametrit.

Annotaation voi tietyissä tilanteessa esittää edellistä yksinkertaistetummin. Jos annotaatiolle annetaan vain yksi `value`-niminen parametri, ei parametria tarvitse erikseen nimetä [GJS+15, s. 314–315]. Esimerkiksi `@SuppressWarnings("deprecation")` vastaa täysin yllä esitettyä annotaatiota. Muun nimisiä parametreja ei saa jättää annotaatiossa nimeämättä. Jos annotaatiolle ei anneta laisinkaan parametreja, sulkumerkit saa jättää annotaation perästä pois [GJS+15, s. 313–314]. Siten esimerkiksi `@Deprecated()` on ilmaistavissa: `@Deprecated`. Annotaation parametrin tyypin ollessa taulukko sen alkioiden arvot ilmaistaan aaltosulkeiden sisällä pilkulla toisistaan erotettuna. Jos taulukossa on vain yksi arvo, aaltosulkeet on mahdollista jättää pois. Esimerkiksi `@Target({ElementType.TYPE})` on ilmaistavissa: `@Target(ElementType.TYPE)`.

Annotaatiotyyppien esittely ja meta-annotaatiot

Annotaatiotyypit ovat teknisesti Javan rajapintaluokkien erikoistapaus [GJS+15, s. 294–297]. Annotaatiotyypin esittelyssä esitellään metodit, jotka määrittelevät, millaista dataa annotaation parametreissa tallennetaan. Metodin nimi määrittelee annotaation parametrin nimen. Metodin paluuarvon tyyppi määrittelee annotaation parametrin tyypin, jonka tulee olla jokin seuraavista: Javan primitiivityyppi, `String`- tai `Class`-luokan ilmentymä, luettelotyyppi, annotaatio tai yksiulotteinen taulukko, jonka elementtien tyyppi on jokin edellä luetelluista.

Annotaatiotyypin metodiesittelyn yhteydessä on mahdollista määritellä sille oletusarvo, jolloin annotaatiosta on mahdollista jättää pois metodin nimen mukainen parametri [GJS+15, s. 299–300]. Java-kääntäjä ei tallenna oletusarvoa annotaatioon, vaan oletusarvo luetaan annotaatiotyypin määrittelystä parametrin lukuhetkellä. Tällä on vaikutusta silloin, jos oletusarvo muuttuu annotoidun luokan kääntämisen jälkeen.

Listauksessa 4.1 on esimerkki `@MyAnnotation`-annotaatiotyypin esittelystä. Rivillä 8 ennen `interface`-avainsanaa esiintyvä `@`-merkki kertoo, että kyseessä on annotaation esittely. Esittelyyn sisältyy kaksi määrittelyä annotaation parametreiksi: kokonaisluku `value` ja merkkijonotaulukko `strings`, jonka oletusarvo on tyhjä taulukko.

Listaus 4.1: Esimerkki annotaatiotyypin esittelystä.

```
1  import java.lang.annotation.*;
2
3  @Target({ElementType.TYPE, ElementType.METHOD})
4  @Retention(RetentionPolicy.CLASS)
5  @Documented
6  @Inherited
7  @Repeatable(MyAnnotations.class)
8  public @interface MyAnnotation {
9    int value();
10   String[] strings() default {};
11 }
```

Esimerkin annotaatiotyyppi on annotoitu viidellä meta-annotaatiolla. Meta-annotaatioiksi kutsutaan annotaatiotyyppiin liitettyjä annotaatioita, joilla määritellään annotaatiotyypin ominaisuuksia. Esimerkin meta-annotaatiot kuuluvat `java.lang.annotation`-pakkaukseen [Jav16b]. `@Target`-meta-annotaation parametreina luetellaan ne elementtityypit, johon annotaatio on liitettävissä. `@Retention`-meta-annotaation parametri määrää, onko annotaatio olemassa ainoastaan käännösaikaisesti vai tallennetaanko se käännettyyn luokkatiedostoon ja onko luokkatiedostoon tallennettu annotaatio käytettävissä suoritusaikaisesti. `@Documented`-meta-annotaation seurauksena Javadoc-työkalu lisää esimerkin tapauksessa `@MyAnnotation`-annotaation osaksi sillä annotoidun elementin dokumentaatiota. `@Inherited` saa annotaation periytymään annotoidun luokan aliluokkiin, kun annotoitu luokka on konkreettinen luokka eikä rajapintaluokka. `@Repeatable`-meta-annotaatio tekee mahdolliseksi, että kohde-elementin voi annotoida usealla saman annotaatiotyypin

annotaatiolla [GJS+15, s. 300-304]. Käytännössä se toimii syntaktisena yksinkertaistuksena annotaatiolle `@MyAnnotations({@MyAnnotation(0), @MyAnnotation(1)})`. Esimerkissä `@MyAnnotations` on `@Repeatable`-annotaatiolle parametrina annettu itse määritelty annotaatiotyyppi, joka toimi säiliönä (container) toistuville annotaatioille.

Annotaatioprosessorit

Javan versiossa 6 Java-kääntäjään lisättiin tuki kytkettäville annotaatioiden käsittelijöille, annotaatioprosessoreille [JSR06]. Annotaatioprosessorit ovat Java-luokkia, jotka toteuttavat `javax.annotation.processing.Processor`-rajapinnan. Ne sijoitetaan joko käännösaikaiseen luokkapolkuun tai erikseen kääntäjälle kerrottavaan annotaatioprosessoreiden polkuun [Jav14]. Kunkin annotaatioprosessorin käyttöönotto käännöksen yhteydessä voi tapahtua automaattisesti luokkapolun konfiguraatiotiedostojen perusteella tai yksikäsitteisesti kääntäjän komentorivioption perusteella.

Annotaatioiden prosessointi tapahtuu yhdessä tai useammassa kierroksessa (round) [Jav14]. Kääntäjän käytyä läpi käännöskomennossa annetut lähdekoodi- ja luokkatiedostot, se poimii niistä löytyneet annotaatiot ja antaa ne annotaatioprosessoreille käsiteltäväksi. Jos jokin prosessoreista tuotti uusia lähdekooditiedostoja, kääntäjä aloittaa uuden kierroksen. Jokaisella uudella kierroksella kääntäjä antaa annotaatioprosessorille prosessoitavaksi joukon edellisen kierroksen tuottamista lähdekoodi- ja luokkatiedostoista löytyneitä annotaatioita, kunnes uusia annotaatioita ei enää ole käsiteltäväksi.

Annotaatioprosessorit lukevat ohjelman elementtejä `javax.lang.model`-pakkauksessa ja sen alipakkauksissa määritellyn mallinnusrajapinnan kautta [Jav16b, javax.lang.model]. Rajapinta perustuu peiliperustaiseen (mirror-based) mallinnukseen [BU04], ja sen keskeisinä käsitteinä ovat elementit ja tyypit. Elementit edustavat staattisia kielen konstruktioita, kuten esimerkiksi luokkien, muuttujien ja metodien esittelyitä. Tyyppien mallinnus erottelee esimerkiksi geneeriset ja niin kutsutut raakatyypit (raw type) toisistaan. Ohjelman elementtien läpikäynti tapahtuu vierailija-suunnittelumallin mukaisesti. Rajapinta on abstraktia syntaksipuuta paljon korkeatasoisempi, koska se ei tunne esimerkiksi lausekkeen tai lohkon käsitettä. Se mahdollistaa ainoastaan ohjelman lukemisen, muttei sen muokkaamista.

4.4 Kytkettävä tyyppijärjestelmäkehys Checker Framework

Kytkettävän tyyppijärjestelmäkehyksen voidaan ajatella palvelevan kahta pääasiallista asiakasjoukkoa: ohjelmoijia ja tyyppijärjestelmien suunnittelijoita [PAC+08]. Ohjelmoijat voivat kirjoittaa tyyppimääreitä ohjelmaansa ja suorittaa tyyppitarkastuksen varmistaakseen, että ohjelma noudattaa tyyppijärjestelmää. Tyyppijärjestelmien suunnittelijat voivat määritellä tyyppimääreitä (type qualifier) ja niiden semantiikan sekä luoda tyyppitarkastajan ohjelmoijan käytettäväksi.

Javalle on toteutettu joitakin kytkettäviä tyyppijärjestelmäkehyksiä: JavaCOP [ANMM06], JQual [GF07] ja Checker Framework [PAC+08]. Näistä Checker Framework on tiettävästi ainoa, jonka on osoitettu skaalautuvan todellisten ohjelmistojen tarpeisiin [DDE+11]. Kehystä kehitetään edelleen jatkuvasti ja siitä julkaistaan säännöllisesti uusia versioita. Tässä luvussa tarkastellaan Checker Frameworkin peruspiirteitä.

Kytkettävältä tyyppijärjestelmäkehykseltä vaadittavia ominaisuuksia

Jotta kytkettävä tyyppijärjestelmäkehys palvelisi ohjelmoijaa, sen on oltava integroitavissa käytössä olevaan ohjelmointikieleen, suoritusaikaiseen järjestelmään sekä työkaluihin [PAC+08]. Erillisen työkalun käyttö on aina hankalampaa pelkän kääntäjän käyttöön verrattuna. Toisaalta muunneltu kääntäjä on myös ongelmallinen: kääntäjän uusiin versioihin tulleet muutokset edellyttävät muutosten toteuttamista myös muunneltuun kääntäjään. Lisäksi jos tarvittavat välineet jakautuvat usean muunnellun kääntäjän kesken, niiden samanaikainen käyttö tuskin tulee kyseeseen. Siksi tyyppijärjestelmän täyttävät tarkastukset tulee pystyä suorittamaan alkuperäisellä muuntelemattomalla kääntäjällä.

Kehyksen lisäksi itse tyyppimääreiden toteutustavan on oltava osa ohjelmointikieltä, jota kaikki työkalut tukevat. Esimerkiksi kommentteihin kätkettyjen tyyppimääreiden ongelmana on, etteivät kehitysympäristöjen refaktorointitoiminnot tunnista niitä luotettavasti. Tyyppimääreiden on myös säilytettävä käännetyissä ohjelmatiedostoissa vastaavasti kuten kielen omat tyyppitiedot, jotta tyyppitieto on hyödynnettävissä silloinkin, kun käännetään ohjelmaa, joka käyttää jo aiemmin käännettyjä luokkia.

Tyyppijärjestelmien kehittäjien tavoitteena on luoda tyyppijärjestelmä, joka on hyödyksi ohjelmoijille. Jotta tyyppijärjestelmäkehys tukisi tätä tavoitetta, sen on mahdollistettava yksinkertaisten tyyppisääntöjen määrittely helposti ja deklaratiivisesti. Monimutkaisempia sääntöjä tulee pystyä määrittelemään proseduraalisesti. Tyyppijärjestelmän kehittäjä tarvitsee tyyppikehykseltä ilmaisuvoimaisuutta ja skaalautuvuutta.

Javan tyyppiannotaatiot

Javan annotaatiot täyttävät edellä mainitut tyyppimääreiden toteutustavan edellytykset: ne ovat osa ohjelmointikieltä ja ne tallennetaan käännettyihin luokkatiedostoihin. Annotaatiot eivät myöskään vaikuta ohjelman suoritusaikaiseen toimintaan, joten ne täyttävät myös aiemmin kerrotun kytkettävän tyyppijärjestelmän edellytyksen.

Vuonna 2006 julkaistu Java 6 oli kielen uusin versio Checker Frameworkin syntyaikoihin. Tuolloin Javassa ei ollut tyyppiannotaatioita. Annotaatioita oli mahdollista liittää ainoastaan luokkien, muuttujien ja muiden esittelyihin, muttei esimerkiksi tyyppimuunnosten tai geneeristen tyyppien tyyppiparametrien yhteyteen. Siksi Checker Frameworkia varten suunniteltiin laajennos Javan annotaatiojärjestelmään [PAC+08]. Laajennoksesta tehtiin virallinen Java-kielen määrittelypyyntö (Java Specification Request) JSR-308 [JSR14] nimellä Annotations on Java® Types, joka liitettiin osaksi vuonna 2015 julkaistua Java-kielen versiota 8 [Jav16c]. Checker Frameworkin käyttö Javan versiossa 8 ei siten vaadi erillisiä työkaluja eikä muutoksia Java-kääntäjään.

Tyyppijärjestelmät ja niiden tarkastajat

Checker Frameworkissa tyyppijärjestelmän toteutus, jota kehyksessä kutsutaan tarkastajaksi, koostuu neljästä osasta: tyyppimääreet ja niiden hierarkia, implisiittiset esittelysäännöt, tyyppisäännöt sekä kääntäjärajapinta. Tarkastajat ovat Javan annotaatioprosessoreita, jotka perivät kehyksen abstraktin tarkastajatoteutuksen [Che16, s. 177–178].

Tyyppimääreet ovat Javan annotaatioita [PAC+08], ja niitä käytetään Javan käytetään tyyppinimien käytön yhteydessä. Jokainen tyyppimääre asettaa rajat tyypin edustamille arvoille rajoittaen Javan tyypin sallimien arvojen joukkoa. Esimerkiksi tyyppi `@NonNull String` sallii ainoastaan sellaiset merkkijonot, joiden arvo ei ole `null`. Aivan kuten Javan luokat muodostavat luokkahierarkian, kytkettävään tyyppijärjestelmään kuuluvat tyyppimääreet muodostavat tyyppihierarkian, joka ilmaisee muodollisten tyyppien keskinäiset alityyppisuhteet. Hierarkian määrittely on mahdollista tehdä joko deklaratiivisesti tai proseduraalisesti.

Joitakin tyyppejä ja lausekkeita tulee käsitellä niin kuin niillä olisi tietty tyyppimääre, vaikka sitä ei yksikäsitteisesti olekaan asetettu. Esimerkiksi jokainen literaali `null`-arvoa lukuun ottamatta on implisiittisesti `@NonNull` [PAC+08]. Tyyppijärjestelmä voi esitellä tyyppimääreitä implisiittisiksi määräämillään ehdoilla. Ehtojen määrittely on mahdollista tehdä joko deklaratiivisesti tai proseduraalisesti.

Tyyppijärjestelmän oleellinen osa on tyyppisäännöt. Kehykseen on sisäänrakennettuna tyyppihierarkiasäännön tarkastus [PAC+08]. Sen mukaan esimerkiksi sijoituslauseen vasemman puolen tyypin on oltava oikean puolen tyypin ylityyppi tai tyyppien on oltava samat.

Tyyppijärjestelmän semantiikan vastaiset rikkeet ovat tyyppivirheitä, jotka kääntäjä ilmoittaa käännösvirheinä. Jos tarkastaja tekee virheellisen päättelyn lauseen tyypeistä, on väärät virheilmoitukset mahdollista vaientaa Javan `@SuppressWarnings`-annotaatiolla. Vaihtoehtoisesti tietyn tyyppiset virheet on mahdollista vaientaa koko käännöksessä kääntäjälle annettavalla komentorivioptiolla.

Kuten luvussa 4.3 kerrottiin, Javan annotaatioprosessorit lukevat ohjelmakoodia abstraktia syntaksipuuta korkeatasoisempaa ohjelmaelementteihin perustuvaa rajapintaa käyttäen. Rajapinta on kuitenkin ohjelman tyyppitarkastuksia varten aivan liian korkeatasoinen. Siksi Checker Framework suorittaa tarkastukset abstraktin syntaksipuun tasolla Oraclen OpenJDK:n kääntäjäpuurajapintaa (Compiler Tree API) käyttäen [Che16, s. 171, 177–178]. OpenJDK-sidoksesta seuraa, että Checker Framework ei toimi muiden, esimerkiksi Eclipsen kääntäjän kanssa. Sidokseen on syynä, etteivät työkalut tue mitään yhteistä standardia syntaksipuurajapintaa.

Jos oman tyyppijärjestelmään liittyy muitakin tyyppisääntöjä kuin aiempana mainittu typpihierarkian tarkastus, niiden tarkastamiseksi on toteutetaan oma abstraktin syntaksipuun vierailija-suunnittelumallin mukainen vierailijaluokka, joka suorittaa tarvittavat tarkastukset määrätyille syntaksipuun solmuille [Che16, s. 171].

Kääntäjärajapinnan välityksellä tarkastaja kertoo Java-kääntäjälle, mitkä annotaatiot kuuluvat tarkastajan tyyppijärjestelmään ja mitä komentoriviparametreja kääntäjän tulisi välittää tyyppijärjestelmän tarkastajalle [PAC+08].

Alkujaan Checker Frameworkissa oli sisäänrakennettuna viisi eri tyyppijärjestelmää: yksinkertaisten tyyppijärjestelmien perustarkastaja, null-osoitinvirheitä havaitseva Nullness Checker, yhtäläisyysvertailuvirheitä havaitseva Interning Checker sekä muuttuvuusvirheitä havaitsevat Javari Checker ja IGJ Checker [PAC+08]. Huhtikuussa 2016 julkaistun version 1.9.13 manuaali luettelee 21 eri sisäänrakennettua tyyppijärjestelmää, sekä listaa muutamia kolmansien osapuolten kehittämiä tyyppijärjestelmiä [Che16, s. 11].

5 Kytkettävä pääsynvalvonta Javaan

Tutkielmassa on tähän asti tarkasteltu ohjelmointikielten semanttisia piirteitä erityisesti pääsynvalvonnan ja tyyppijärjestelmien osalta sekä perehdytty kytkettävien tyyppijärjestelmien periaatteeseen. Tässä luvussa esitellään kytkettävä pääsynvalvonta Javaan soveltaen kytkettävien tyyppijärjestelmien ideaa pääsynvalvontaan. Sen tavoitteena on mahdollistaa optimaalinen tiedon piilottaminen silloinkin, kun ohjelmointikielen oma pääsynvalvontajärjestelmä ei mahdollista riittävän hienojakoista saatavuuden rajoittamista. Aluksi kartoitetaan tilanteita, joissa Javan saatavuusrajoitteet osoittautuvat tarvetta karkeammiksi.

5.1 Javan pääsynvalvonnan karkeus

Java-kieli jaottelee saatavuuden neljään tasoon: julkinen, suojattu, pakkauksen sisäinen ja yksityinen. Tällaista jaottelua voi pitää karkeana: käytännössä tulee vastaan tilanteita, joissa ohjelmaelementin saatavuustaso on asetettavaksi laajemmaksi, kuin mitä todellinen saatavuuden tarve on, koska suppeampi saatavuustaso olisi tarpeeseen nähden jo liian suppea. Sen seurauksena abstraktioiden sisäiset toteutusyksityiskohdat ovat näkyvissä laajemmalle kuin olisi tarpeen, mikä johtaa ohjelman sisäisten riippuvuuksien lisääntymiseen heikentäen sen modulaarisuutta [Par72].

Alla on kuvailtu neljä tilannetta, joissa optimaalinen saatavuus ei ole saavutettavissa Javan pääsymääreillä: asiakkaan, rajapinnan ja toteutuksen välinen saatavuus, testaamista varten avattu kapselointi, oliotason yksityisyys sekä pääsy aliluokista. Tilanteet on pyritty valitsemaan siten, että ne edustaisivat kattavasti eri tyyppisiä Javan pääsyrajoitteiden ongelmakohtia.

Asiakkaan, rajapinnan ja toteutuksen välinen saatavuus

Tiedon piilottamisen keskeinen ajatus on, että ohjelmakomponentti tarjoaa vakaan julkisen rajapinnan, jonka välityksellä ohjelmakomponenttia käyttävä asiakas kommunikoi kyseisen komponentin kanssa. Komponentin rajapinnan toteutukseen liittyvät yksityiskohdat, jotka saattavat muuttua, pidetään piilossa muilta ohjelmakomponenteilta.

Javassa tämä voi toteutua modularisoinnin eri tasoilla. Luokka voi määritellä julkisen rajapinnan esittelemällä rajapinnan muodostavat jäsenensä luokan ulkopuolelta saataviksi. Samalla luokka pitää toteutuksen yksityiskohdat piilossa esittelemällä ne luokan yksityisinä jäseninä. Laajemmassa mittakaavassa yksittäinen luokka voidaan nähdä toteutuksen yksityiskohtana. Tällöin on mahdollista esitellä rajapinnaksi julkisia rajapintaluokkia ja esitellä konkreettiset toteutusluokat pakkauksen sisäisiksi.

Laajennetaan mittakaavaa edelleen siten, että rajapinta koostuu useista eri pakkauksissa sijaitsevista rajapintaluokista toteutuksen jakautuessa myös eri pakkauksiin. Tyypillisesti eri pakkauksissa sijaitsevien toteutusluokkien välillä on keskinäisiä riippuvuuksia, jolloin ainoa mahdollisuus on esitellä riippuvuudet julkisina. Tässä vaiheessa sisäiset toteutusyksityiskohdat eivät olekaan enää piilossa, vaan julkisia, ja siten myös sovellusrajapintaa käyttävän asiakkaan saatavissa.

Yksi keino rajoittaa saatavuutta on erottaa asiakas, rajapinta ja toteutus eriaikaisesti käännettäviin käännösprojekteihin. Jos toteutusprojekti ei ole asiakasprojektin kanssa samassa käännösaikaisessa luokkapolussa, ei asiakas pysty luomaan riippuvuutta toteutukseen. Toteutus on tällöin asiakkaalta piilossa. Ohjelmiston jakaminen useampaan käännösprojektiin kuitenkin monimutkaistaa ohjelman käännösprosessia. Toisaalta suurissa ohjelmistoissa eri käännösprojekteiksi jakaminen yksinkertaistaa osakokonaisuuksien kehittämistä ja on siten tavoiteltu rakenne. Kuitenkin pienten ohjelmien jakaminen eri käännösprojekteihin tekee käännösprosessista tarpeettoman monimutkaisen.

Testaamista varten avattu kapselointi

Yksikkötestien tarkoitus on antaa takuu, että ohjelma täyttää testien sille asettamat vaatimukset. Mitä kattavammin testit testaavat ohjelman ominaisuuksia, sitä laajempi on takuu ohjelman oikeellisuudesta sillä edellytyksellä, että testit testaavat oikeita asioita.

Yleisesti ajatellaan, että luokkien ja olioiden yksityisiä jäseniä ei tulisi testata [LF03, luku 8.2]. Yksityisten jäsenten testaaminen on monella tapaa ongelmallista. Koska yksityiset jäsenet ovat toteutuksen yksityiskohtia, niiden voidaan olettaa muuttuvan. Jos luokan yksityinen jäsen muuttuu, on myös sitä testaavaa koodia muutettava, jotta testit testaisivat oikeita asioita. Sen sijaan julkiseen rajapintaan kohdistuvat testit testaavat, että rajapinnan mukaiset toiminnot ja oletukset toteutuvat. Yksityisten jäsenten voidaan tyypillisesti olettaa tulevan testatuksi välillisesti julkisen rajapinnan kautta. Kuitenkin jos sisäinen toteutus muuttuu, ei testejä tarvitse sen vuoksi muuttaa.

Tietyissä tilanteissa yksityisen jäsenen testaaminen on perusteltua. Joskus jonkin ominaisuuden testaaminen julkisen rajapinnan kautta saattaa osoittautua tarpeettoman monimutkaiseksi, jolloin suoraan yksityisen jäsenen testaaminen tekee testistä paljon yksinkertaisemman. Esimerkiksi julkinen rajapinta saattaa vaatia laajan tai työläästi muodostettavan syötteen, josta varsinainen testattava ominaisuus tarvitsee vain pienen osan. Olion saattaminen testiä varten haluttuun tilaan saattaa julkisen rajapinnan kautta olla hyvin monimutkaista, jolloin olion tilan asettaminen suoralla kutsulla olisi paljon helpompaa.

Jos yksityisen jäsenen testaaminen on todettu perustelluksi, on vielä pystyttävä ohittamaan yksityisyyden suojamuuri. Javassa on mahdollista käyttää sen reflektiorajapintaa yksityisten jäsenten aksessoimiseksi. Rajapinnan monimutkaisuus verrattuna suoriin kutsuihin johtaa kuitenkin testikoodin monimutkaisuuteen, mikä puolestaan altistaa testikoodin virheille.

Yksinkertaisempi mahdollisuus on avata kapselointia laajentamalla saatavuutta, mutta sen seurauksena toteutuksen yksityiskohdat paljastuvat muillekin kuin testeille.

Googlen kehittämä Guava-kirjasto[1] on pyrkinyt ratkomaan ongelmaa esittelemällä `@VisibleForTesting`-annotaatiotyypin. Sen on tarkoitus kertoa, että yksityisen jäsenen saatavuutta on laajennettu ainoastaan testikäyttöä varten, eikä sitä tulisi aksessoida testien ulkopuolelta. Ratkaisumalli on saanut suosiota, sillä GitHub-koodinjakopalvelusta on löydettävissä muitakin projekteja, joihin on toteutettu saman niminen annotaatiotyyppi. Mutta koska annotaatiot ovat ainoastaan metadataa, myös `@VisibleForTesting` toimii ainoastaan dokumentaationa, eikä se itsessään estä pääsyä luokan jäseneen myös testikoodin ulkopuolelta.

Oliotason yksityisyys

Eri olio-ohjelmointikielten yksityisyyskäsite vaihtelee olion ja luokan sisäisen yksityisyyden välillä. Näistä oliotason yksityisyys on yksityisyyskäsitteistä tiukin: ainoastaan olio itse pääsee käsiksi yksityisiin jäseniinsä. Edes saman luokan toisella ilmentymällä ei ole pääsyä eri ilmentymän yksityisiin jäseniin. Luokkatason yksityisyys sen sijaan mahdollistaa luokan ilmentymälle pääsyn toisen saman luokan ilmentymän yksityisiin jäseniin.

Minimaalisen saatavuuden periaatteen mukaan ohjelmaelementtien tulisi olla saatavissa niin rajoitetusti kuin mahdollista. Sen perusteella oliotason yksityisyyttä voisi pitää normina. Kuitenkin luvussa 3.2 vertailluista kielistä ainaostaan Smalltalk, Eiffel ja Scala mahdollistavat olion sisäisen yksityisyyden. Muissa kielissä, kuten Javassa, yksityisyys on aina luokan sisäistä. Kielistä Scala on ainoa, joka yhdistää molemmat yksityisyyskäsitteet. Kielten erilaiset yksityisyyskäsitteet saavat aikaan hämmennystä erityisesti aloittelevien sekä uuteen kieleen tutustuvien ohjelmoijien keskuudessa[2] ja pohdintaa, olisiko olioyksityisyys mahdollista esimerkiksi Javassa[3].

Voigt, Iwrin ja Churcher ovat analysoineet joukon ohjelmia, niissä käytettyjä pääsymääreitä ja rajoitettujen elementtien todellista käyttöä [VIC10]. Heidän käyttämässään aineistossa vain 3% yksityisiin jäseniin kohdistuvista viittauksista tapahtui saman luokan toisesta ilmentymästä. Sen perusteella tarve luokan sisäiselle yksityisyydelle vaikuttaa pieneltä. Miksi sitten olioyksityisyys on kielissä niin harvinainen? Kyseinen tutkimus ei kerro, kuinka oleellisia mainitut 3% viittauksista on.

Olioyksityisyys on ongelmallinen ainakin vertailtaessa saman luokan ilmentymiä keskenään. Jos Javan yksityisyys olisikin olion sisäistä, tällöin esimerkiksi `Object.equals()`-metodi ei pystyisi vertailemaan niitä olioiden jäseniä, jotka ovat olioyksityisiä. Vertailun

[1] `https://github.com/google/guava`

[2] `http://programmers.stackexchange.com/q/258046`, `http://stackoverflow.com/q/11354649`, `http://stackoverflow.com/q/17027139`, `http://stackoverflow.com/q/16586809`, `http://stackoverflow.com/q/1581478`

[3] `http://stackoverflow.com/q/18062545`, `http://stackoverflow.com/q/33884964`

mahdollistamiseksi vertailtavat jäsenet eivät voisi enää olla yksityisiä. Samalla tavoin `Comparable.compareTo()`-metodi tarvitsee pääsyn myös toisen olion jäseniin. Mutta jos ainoastaan yllä mainitut vertailumetodit ovat ongelmallisia olioyksityisyyden näkökulmasta, käytännöllisen olioyksityisyyden voisi saavuttaa yhdistämällä olio- ja luokkayksityisyys niin, että olioyksityisiin jäseniin olisi pääsy edellä mainituista vertailumetodeista.

Pääsy aliluokista

Javan `protected`-pääsymääre rajoittaa pääsyn rajoitettavan elementin sisältävän luokan aliluokkiin sekä samassa pakkauksessa sijaitseviin luokkiin. Perintään ja moduulimaiseen rakenteeseen perustuvan saatavuuden yhdistämistä on paitsi pidetty tiedon piilottamista ja eheyttä uhkaavana tekijänä [ABV00], on se myös osoittautunut tietyissä käytännön tilanteissa ongelmalliseksi [Sch04]. Saatavuus ainoastaan aliluokista ilman pakkaustason saatavuutta on nähty toivottavaksi [MPH99]. Silti Javassa ei edelleenkään ole sitä mahdollistavaa pääsymäärettä [GJS+15].

Listaus 5.1 havainnollistaa yhtä eri saatavuustyyppien yhdistämiseen liittyvää ongelmaa. Koska samaan listaukseen on koottu useita käännösyksiköitä, kukin niistä on erotettu rivinumeroinnissa omalla kirjaimellaan. Listaus esittelee pakkauksessa `pkg1` sijaitsevan luokan `Superclass`, jolla on suojatun saatavuustason metodi `protectedMethod`. Luokan aliluokista `Subclass` ja `OverridingSubclass`, jotka sijaitsevat pakkauksessa `pkg2`, jälkimmäinen korvaa yliluokkansa metodin. `Superclass`-luokan kanssa samassa pakkauksessa sijaitseva luokka `Accessor` havainnollistaa pääsyä kyseiseen metodiin. Pääsy `Superclass`- ja `Subclass`-luokkien `protectedMethod`-metodiin on sallittu. Mutta koska `OverridingSubclass` korvaa kyseisen metodin, korvaavan metodin ollessa suojattu, pääsyä ei sallitakaan. Pääsy kuitenkin sallitaan, jos metodia aksessoidaan `Superclass`-luokan edustajana. Metodin saatavuus riippuu siis siitä, onko aliluokka korvannut sen.

Voigtin, Iwrinin ja Churcherin analysoimassa aineistossa suojatun saatavuustason elementteihin kohdistuneista aksessoinneista 27,9% tapahtui aliluokista ja samasta pakkauksesta vain 5,6%, muiden aksessointien ollessa luokan sisäisiä [VIC10]. Tulosten perusteella Javan `protected`-määreeseen kuuluva pakkauksen sisäinen saatavuus ei näyttäydy erityisen hyödyllisenä piirteenä. Päin vastoin, määreen semantiikka aiheutti sekaannuksia tutkimukseen osallistuneiden opiskelijoiden keskuudessa.

Kuten luvussa 3.2 kävi ilmi, C++:ssa, C#:ssa ja Scalassa `protected`-määre lähtökohtaisesti sallii pääsyn ainoastaan aliluokista. C#:ssa ja Scalassa saatavuutta on mahdollista laajentaa rakenteeseen perustuen. Toisaalta Ceylonissa ei ole perintäperustaista saatavuutta lainkaan, vaan saatavuus perustuu aina rakenteeseen ja määrittely-ympäristöön.

Listaus 5.1: Javan `protected`-pääsymääreen ongelma: metodin pakkauksen sisäinen saatavuus riippuu siitä, onko aliluokka korvannut sen.

```
A.1  package problematicprotected.pkg1;
A.2  public class Superclass {
A.3    protected void protectedMethod() {}
A.4  }
A.5
B.1  package problematicprotected.pkg2;
B.2  import problematicprotected.pkg1.Superclass;
B.3  public class Subclass extends Superclass {}
B.4
C.1  package problematicprotected.pkg2;
C.2  import problematicprotected.pkg1.Superclass;
C.3  public class OverridingSubclass extends Superclass {
C.4    @Override
C.5    protected void protectedMethod() {}
C.6  }
C.7
D.1  package problematicprotected.pkg1;
D.2  import problematicprotected.pkg2.*;
D.3  public class Accessor {
D.4    public void access() {
D.5      new Superclass().protectedMethod(); // OK
D.6      new Subclass().protectedMethod(); // OK
D.7      new OverridingSubclass().protectedMethod(); // NO ACCESS!
D.8      ((Superclass) new OverridingSubclass()).protectedMethod(); // OK
D.9    }
D.10 }
```

5.2 Kytkettävä pääsynvalvontajärjestelmä

Edellisessä luvussa esitellyissä tilanteissa ohjelmoijalla olisi tarve hienojakoisemmalle pääsynvalvonnalle. Tässä luvussa esitellään, kuinka se on toteutettavissa kytkettävän pääsynvalvonnan avulla. Kytkettävä pääsynvalvonta soveltaa kytkettävien tyyppijärjestelmien ideaa ohjelmointikielen pääsynvalvontaan: se on valinnainen tarpeen mukaan liitettävä kielen laajennos, joka on mukautettavissa sovelluskohtaisiin tarpeisiin. Tässä luvussa tutustutaan kytkettävän pääsynvalvonnan perusperiaatteisiin.

Peruskäsitteet

Kytkettävän pääsynvalvonnan tavoite on täydentää kielen omaa pääsynvalvontaa. Se esittelee uusia pääsymääreitä, joiden avulla on mahdollista saada aikaan optimaalinen ohjelmaelementtien saatavuus: ohjelmaelementit ovat saatavissa ohjelman osista, joista niihin tulee olla pääsy, muttei sitä laajemmin.

Kytkettävä pääsynvalvonta on ohjelmointikielen pääsynvalvontaan viety sovellus kytkettävien tyyppijärjestelmien perusajatuksista: se on valinnainen ja kytkettävissä ohjelmointikieleen tai irrotettavissa siitä tarpeen mukaan. Kehittäjä pystyy näin itse valitsemaan, millaista pääsynvalvontaa hän haluaa ohjelmassaan käyttää ja missä vaiheessa ohjelman

kehitystyötä. Hän voi ottaa pääsynvalvontajärjestelmän käyttöön kokonaisuudessaan, osan siitä tai jättää kokonaan käyttämättä. Näin esimerkiksi ohjelmoija voi väliaikaisesti ottaa pääsynvalvontajärjestelmän pois käytöstä kokeillakseen nopeasti jotakin ideaa, ja tämän jälkeen kytkeä järjestelmä takaisin käyttöön toteuttaakseen idean kunnollisesti.

Kytkettävä pääsynvalvontajärjestelmä on järjestelmä, joka toteuttaa kytkettävää pääsynvalvontaa. Järjestelmä koostuu yhdestä tai useasta pääsymääreestä sekä niiden tarkastajista. Kytkettäväksi pääsynvalvontajärjestelmäksi voidaan myös kutsua pääsynvalvontajärjestelmää, joka on kokoelma pienempiä osajärjestelmiä.

Kytkettävän pääsynvalvontajärjestelmän pääsymääreitä kutsutaan kytkettäviksi pääsymääreiksi, tai pelkästään pääsymääreiksi, kun ei ole vaaraa sekoittaa käsitettä ohjelmointikielen sisäänrakennettuihin pääsymääreisiin. Kytkettävä pääsymääre on määre, jolle on määritelty nimi, esittelysäännöt ja mahdolliset muodolliset parametrit sekä pääsyrajoitteen semantiikka, muttei välttämättä kaikkia tarkastamiseen liittyviä käytännön yksityiskohtia.

Määreen esittelysäännöt määräävät, millaisen ohjelmaelementin yhteydessä määrettä on sallittua käyttää. Säännöt voivat perustua esimerkiksi elementin lajiin, kuten luokan tai metodin esittely, elementin muihin pääsyrajoitteisiin, elementin sijaintiin tai mihin tahansa muuhun tarkastettavissa olevaan perusteeseen. Pääsymääre voi määritellä myös muodollisia parametreja, joilla määreen rajoitetta on mahdollista tarkentaa rajoitekohtaisesti. Määreen esittelysäännöt määrittelevät, millaiset muodollisten parametrien arvot ovat sallittuja ja millaisissa yhteyksissä.

Jotta pääsymääreen noudattamista voidaan valvoa, tarvitaan kullekin pääsymääreelle asianmukainen tarkastaja. Pääsymääretarkastaja on kääntäjään kytkettävä valvonnan suorittava toteutus, joka määrittelee tarkastuksen yksityiskohdat. Sen tehtävänä on tarkastaa, että pääsymääreitä käytetään niiden esittelysääntöjen mukaisesti, ja että pääsymääreellä rajoitettuihin elementtejä aksessoidaan määreen asettamien rajoitusten puitteissa. Pääsymääreen esittelysääntöjen tai rajoitteiden rikkomisen tarkastaja raportoi käännösvirheenä.

Pääsymääreiden rajoitesemantiikka

Pääsymääreiden rajoitesemantiikka asettaa lisärajoitteita ohjelmointikielen sisäänrakennettujen pääsymääreiden lisäksi. Ohjelmaelementin efektiivinen sallittujen aksessoijien joukko on kielen pääsymääreiden ja kytkettävien pääsymääreiden sallimien aksessoijajoukkojen leikkaus. Kytkettävät pääsymääreet eivät periydy luokkahierarkiassa, vaan aliluokkien on yksikäsitteisesti määriteltävä saatavuutensa. Jotta rajoitettu elementti olisi saatavissa sen esittely-ympäristönsä ulkopuolelta, täytyy elementin lisäksi sen esittely-ympäristön olla saatavissa.

Kytkettävät pääsymääreet ovat itsenäisiä ja toisistaan riippumattomia. Kukin pääsymääre saa vapaasti määritellä saatavuusperusteensa ja vaikutusalueensa. Jos elementille ei ole

asetettu pääsymäärettä, ei sen saatavuutta rajoiteta pääsynvalvontajärjestelmän toimesta lainkaan. Yksi pääsymääre määrittelee yksikäsitteisesti sallitun aksessoijajoukon eli niiden ohjelman osien joukon, joista pääsy rajoitettuun elementtiin on sallittu.

Silloin kun rajoitettavalle elementille asetetaan useampi kuin yksi pääsymääre, kuinka sallittujen aksessoijien joukko tulisi tällöin muodostaa? Jos sallittujen aksessoijien joukko muodostettaisiin yksittäisten määreiden sallimien aksessoijien leikkauksena, lopputuloksena syntyisi tilanteita, joissa rajoitettu ohjelmaelementti ei ole saatavissa missään. Esimerkiksi luvussa 5.3 määritelty määre `@AccessFromClass` rajoittaa pääsyn valikoivasti aksessoivan luokan perusteella, ja `@SelfAccess` rajoittaa pääsyn saman olion sisään. Oletetaan, että luokan C jäsenen x saatavuus rajoitetaan molemmilla määreillä siten, että `@AccessFromClass` sallii pääsyn ainoastaan luokasta B. Jos x:n sallittujen aksessoijien joukko muodostetaan kummankin määreen sallimien aksessoijajoukkojen yhdisteenä, on tuloksena tyhjä joukko. Toisin sanoen x ei ole saatavissa missään. Vastaavia tilanteita syntyisi erityisen helposti luvussa 5.4 esiteltyjen määrealiasten käytön yhteydessä.

Sen sijaan, että määreiden sallima aksessoijajoukko muodostettaisiin yksittäisten määreiden sallimien aksessoijajoukkojen leikkauksena, on se muodostettava yhdisteenä. Tällöin määreiden semanttisen mallin tulee olla sellainen, että lähtötilanteessa pääsyä ei ole sallittu missään, ja määreillä annetaan pääsy määreen määrittelemistä ohjelman osista. Vaikka pääsymääreet yhdistettynä laajentavat elementin saatavuutta, niistä puhutaan edelleen pääsyrajoitteina, koska lopputuloksena elementin saatavuutta on rajoitettu verrattuna tilanteeseen ilman pääsymääreitä.

Lähestymistapa mahdollistaa pääsymääreiden yksinkertaisuuden: `@AccessFromClass` on mahdollista määritellä sallimaan pääsy luokan jäseneen ainoastaan tietyistä luokista ilman tarvetta ratkoa tilannetta, että luokan sisäinen pääsy jäseneen tulisi turvata jotenkin. Sen sijaan luokan sisäinen saatavuus on mahdollista asettaa erikseen ohjelmoijan valitsemalla toisella pääsymääreellä.

Pääsymääreiden yksinkertaisuus puolestaan parantaa pääsynvalvontajärjestelmän laajennettavuutta: kun yksittäinen pääsymääre määrittelee mahdollisimman suppean vastuualueen, myös riski eri määreiden yhteisvaikutusten aiheuttamista ongelmista on mahdollisimman pieni. Tämä on syytä muistaa pääsymääreitä suunniteltaessa. Yksinkertaisuuden ansiosta ohjelmoija pystyy halutessaan entistä tarkemmin määrittelemään kunkin ohjelmaelementin saatavuuden yhdistelemällä eri pääsymääreitä.

Kytkettävä pääsynvalvonta on käännösaikaista

Kytkettävien tyyppien tavoin myös kytkettävä pääsynvalvonta tulee määritellä staattiseksi. Toisin sanoen sen pääsyrajoitteet ovat voimassa ainoastaan käännösaikaisesti. Se ei muuta ohjelman suoritusaikaista semantiikkaa millään tavoin. Näin ollen kytkettävät pääsymääreet eivät rajoita saatavuutta sellaisesta ohjelmakoodista, joka ei käytä kyseistä

pääsynvalvontajärjestelmää. Pääsynvalvonnan staattisuuden seuraukset ovat silti erilaisia kuin tyyppijärjestelmien.

Kytkettävä tyyppijärjestelmä takaa, että ohjelman tyypitetty osa ei sisällä tyyppivirheitä. Kytkettävän pääsynvalvontajärjestelmän tarkoitus taata, että käännettävä ohjelma ei sisällä laittomia viittauksia rajoitettuihin elementteihin. Merkittävä ero on siinä, että tyyppitarkastukset suoritetaan tyypitetyn elementin perusteella, kun taas saatavuustarkastukset suoritetaan ohjelmaelementin viittauksen kohteena olevan elementin perusteella. Näin ollen mahdolliset tyyppivirheet rajoittuvat käännettävän ohjelman sisälle ja ovat siten staattisesti tarkastettavissa, kun taas pääsyrajoitteita rikkova viittaus saattaa tapahtua käännösprojektin ulkopuolella toisessa käännösprojektissa. Jos käännösprojektin A rajoitettuun elementtiin viittaavassa käännösprojektissa B on rajoitteiden vastaisia viittauksia, niitä ei havaita, ellei kyseisessäkin projektissa ole samanlainen pääsynvalvonta kytkettynä kääntäjään. Näin ollen kytkettävä pääsynvalvontajärjestelmä pystyy takaamaan pääsyrajoitteiden toteutumisen ainoastaan käännösprojektin sisällä. Takuun erikoistapaus on yksityisiin ohjelmaelementteihin kohdistuneet kytkettävät pääsyrajoitteet silloin, kun suoritusympäristö estää niihin kohdistuvat viittaukset luokan ulkopuolelta. Siten kytkettävän pääsynvalvontajärjestelmän voidaan sanoa takaavan aina luokan yksityisiin jäseniin asetettujen kytkettävien pääsyrajoitteiden toteutuminen.

Entä jos kytkettävä pääsynvalvonta saisikin vaikuttaa suoritusaikaiseen toimintaan eikä olisi ainoastaan staattinen ominaisuus? Suoritusaikaisesti suoritettavilla tarkastuksilla olisi mahdollista taata pääsyrajoitteiden voimassaolo silloinkin, kun rajoitettuun elementtiin viittaavaa ohjelmaa ei ole käännetty pääsynvalvontajärjestelmän kanssa. Se johtaisi kuitenkin odottamattomiin suoritusaikaisiin virheisiin, koska kääntäjä ei ilman pääsynvalvontajärjestelmää havaitse laitonta viittausta. Tämä on erityisen vaarallista, koska ohjelmoija luottaa siihen, että kääntäjä havaitsee laittomat viittaukset.

Suoritusaikaisuuden toteuttaminen ei myöskään ole ongelmatonta. Suoritusaikaiset tarkastukset tulisi olla kytkettävissä suoritusympäristöön, koska kääntäjä ei pysty generoimaan tarkastuskoodia esimerkiksi rajoitettuihin kenttiin tai luokkaesittelyihin. Käytännön edellytys suoritusympäristöön kytkettävyydelle on, että suoritettava ohjelma pystyy tekemään kytkennän. Jos kytkentä jää ohjelman käyttäjän vastuulle, on todennäköistä, että kytkentä jää tekemättä, jolloin pääsyrajoitteet eivät ole voimassa. Ainakaan esimerkiksi Javan virtuaalikoneessa ei tällaista laajennosmahdollisuutta ole [LYBB15]. Lisäksi luonnollinen seuraus suoritusaikaisista saatavuustarkastuksista on niiden negatiivinen vaikutus suorituskykyyn, jonka merkitys on kuitenkin sovelluskohtaista.

Näiden seikkojen perusteella voidaan todeta kytkettävän tyyppijärjestelmän ehdon suoritusaikaisen semantiikan säilymisestä olevan pätevä myös kytkettävän pääsynvalvontajärjestelmän kohdalla, ja kytkettävän pääsynvalvontajärjestelmän staattisuuden olevan perusteltua.

5.3 Pääsymääreet

Tässä luvussa esitellään joukko kytkettäviä pääsymääreitä, jotka on suunniteltu täyttämään luvun 5.1 tapauksissa esiin tulleet pääsynvalvonnan tarpeet. Kullekin määreelle määritellään sen semantiikka: mahdolliset rajoitettavat elementtityypit, esittelyehdot, saatavuuden perusteet sekä määreen mahdolliset parametrit. Suunnittelun kantavana ajatuksena on ollut yksinkertaisuus: määreiden semantiikka on selkeästi rajattu, yksikäsitteinen ja suoraviivainen. Kaikki esitellyt määreet ovat itsenäisiä ja toisistaan riippumattomia.

@AccessFromClass — luokkaperusteinen valikoiva saatavuus

`@AccessFromClass`-pääsymääre rajoittaa elementtiin pääsyn määrätyistä luokista. Rajoitettava elementti voi olla pakkaus, luokka, kenttä, muodostin tai metodi.

Pääsymääre ottaa ainoana parametrinaan *value* joukon luokkia, joista on pääsy rajoitteen kohteeseen. Pääsy sallitaan myös lueteltujen luokkien aliluokista sekä kaikkien edellä mainittujen sisäkkäisistä luokista. Kaikkialta muualta on pääsy estetty. Jos luokkajoukko on `{Object}`, määre ei rajoita lainkaan pääsyä kohteeseen. Luokkajoukon ollessa `{NoClass}` määre ei salli lainkaan pääsyä kohteeseen. Luokkajoukko on pakollinen parametri, eikä se saa olla tyhjä.

Määreen toiminta vastaa pääosin Eiffel-kielen [M$^+$06] valikoivaa saatavuutta. Mutta toisin kuin Eiffelissä, jossa jäsenen jakaminen valikoidusti olion ulkopuolelle ei vaikuta jäsenen saatavuuteen olion sisällä, `@AccessFromClass`-pääsymääre ei määrittele minkäänlaista esittelykontekstin sisäistä, esimerkiksi `private`-tason, saatavuutta. Sen sijaan sisäinen saatavuus tulee määritellä erikseen esimerkiksi `@AccessWithin`- tai `@SelfAccess`-määreillä.

@AccessFromInherited — perintäperusteinen saatavuus

`@AccessFromInherited`-pääsymääre rajoittaa elementtiin pääsyn elementin esittelevään luokkaan, sen aliluokkiin sekä kaikkien edellä mainittujen sisäkkäisiin luokkiin. Rajoitettava elementti voi olla jäsenluokka, kenttä, muodostin tai metodi. Pääsymääre on parametriton.

Määre sallii pääsyn aliluokasta, kun aksessointi täyttää Javan spesifikaatiossa määritellyt pakkauksen ulkopuolelta tapahtuvaa `protected`-määrellä rajoitettun elementin aksessointia koskevat ehdot [GJS$^+$15, §6.6.2]. Kyseiset ehdot ovat tiivistettävissä epämuodollisesti seuraaviin sääntöihin. Olkoon C luokka, jolla on suojattu kenttä tai metodi x, ja B C:n aliluokka. Pääsy x:ään on sallittu ainoastaan viitattaessa siihen B:n tai sen aliluokan jäsenenä, muttei C:n jäsenenä. C:n muodostimeen on pääsy `super`-viittausta käyttäen B:n muodostimesta sekä anonyymiluokan luontilauseesta.

@AccessFromAnnotated — annotointiperusteinen valikoiva saatavuus

`@AccessFromAnnotated`-pääsymääre rajoittaa elementtiin pääsyn ainoastaan määrätyillä annotaatiotyypeillä annotoiduista elementeistä. Rajoitettava elementti voi olla pakkaus, luokka, kenttä, muodostin tai metodi.

Pääsymääre ottaa ainoana parametrinaan *value* joukon annotaatiotyyppejä. Määre sallii pääsyn rajoitettuun elementtiin ainoastaan lueteltujen tyyppisillä annotaatioilla annotoiduista elementeistä. Kun annotoituun elementtiin sisältyy lohko, pääsy sallitaan kaikkialta lohkon sisällä. Annotaation mahdollinen periytyvyys huomioidaan aksessoivan elementin annotaatioita selvitettäessä. Kaikkialta muualta on pääsy estetty. Annotaatiotyyppijoukon ollessa `{NoneAnnotated}` määre ei salli lainkaan pääsyä kohteeseen. Annotaatiotyyppijoukko on pakollinen parametri eikä se saa olla tyhjä.

@AccessFromMatching — hahmonsovitukseen perustuva valikoiva saatavuus

`@AccessFromMatching`-pääsymääre rajoittaa elementtiin pääsyn ainoastaan sellaisiin ohjelman osiin, jotka sopivat parametrina annettuihin merkkijonohahmoihin (pattern). Aksessoivan ohjelman osaan viittaavaa merkkijonoa, jota kutsuttakoon aksessoijaviitteeksi, sovitetaan hahmoihin. Pääsy rajoitettuun elementtiin sallitaan, jos ja vain jos aksessoijaviite täsmää vähintään yhteen annetuista hahmoista. Rajoitettava elementti voi olla pakkaus, luokka, kenttä, muodostin tai metodi.

Aksessoijaviite muodostetaan aksessoivan luokan binäärinimestä sekä aksessoivan jäsenen esittelyn osasta. Luokan binäärinimi on Javan spesifikaatiossa määritelty käännettyjen luokkatiedostojen käyttämä muoto luokan nimestä [GJS+15, §13.1]. Aksessoijaviitteen täsmälliset muodostamissäännöt ovat liitteessä 1. Listauksessa 5.2 on esimerkkejä aksessoijaviitteistä. Niistä ensimmäinen viittaa `my.pkg`-pakkauksessa sijaitsevan `MyClass`-luokan parametrittomaan muodostimeen. Toinen viittaa edellä mainitun luokan sisäkkäisen luokan `MyInner` metodiin `myMethod`, joka ottaa parametrinaan merkkijonotaulukon `java.lang.String[]` sekä `MyInner`-luokan sisäkkäisen luokan `EvenDeeper`-ilmentymän.

Listaus 5.2: Esimerkkejä aksessoijaviitteistä.

```
1 my.pkg.MyClass::new()
2 my.pkg.MyClass$MyInner::myMethod(java.lang.String[],my.pkg.MyClass$MyInner$EvenDeeper)
```

Jos aksessointi tapahtuu paikallisessa tai anonyymissä luokassa, eikä siitä muodostettu aksessoijaviite täsmää mihinkään hahmoista, luodaan luokan määrittely-ympäristöstä, esimerkiksi metodista, toinen aksessoijaviite, jota sovitetaan hahmoihin. Jos määrittelyympäristökin sijaitsee paikallisessa tai anonyymissä luokassa, sovelletaan tätä sääntöä rekursiivisesti. Kyseisellä säännöllä taataan pääsy rajoitettuun kohteeseen tilanteessa, jossa

paikallinen tai anonyymi luokka toimii määrittely-ympäristönsä sisäisenä toteutusyksityiskohtana, määrittely-ympäristönsä edustajana.

Hahmot, joihin aksessoijaviitettä sovitetaan, on mahdollista antaa pääsymääreelle kahdessa eri muodossa: säännöllisenä lausekkeena (regular expression) tai möykkyhahmona (glob). Säännöllinen lauseke annetaan pääsymääreen *regex*-parametrina. Vain yksi säännöllinen lauseke on sallittu. Säännöllisen lausekkeen syntaksi on sama, kuin mitä Java-kirjaston `java.util.regex.Pattern`-luokan dokumentaatiossa on määritelty. Parametrin oletusarvo on tyhjä merkkijono, joka tulkitaan siten, ettei säännöllistä lauseketta sovelleta pääsyn rajoittamisessa.

Möykkyhahmot ovat säännöllisiä lausekkeita yksinkertaisempia hahmoja. Mistä tahansa möykkyhahmosta on johdettavissa sellainen säännöllinen lauseke, joka tunnistaa täsmälleen saman joukon merkkijonoja. Yksi tai useampi möykkyhahmo annetaan pääsymääreen *glob*-parametrina. Hahmon syntaksi on sovellettu versio Java-kirjaston `java.nio.file.FileSystem`-luokan `getPathMatcher`-metodin dokumentaatiossa määritellystä. Täsmällinen kuvaus möykkyhahmojen syntaksista on liitteessä 1. Parametrin oletusarvo on tyhjä joukko, joka tulkitaan siten, ettei möykkyhahmoja sovelleta pääsyn rajoittamisessa.

Listauksessa 5.3 on esimerkkejä möykkyhahmoista. Näistä ensimmäinen täsmää mihin tahansa `my.pkg`-pakkauksessa sijaitsevaan luokkaan. Toinen täsmää missä tahansa pakkauksessa, myös nimettömässä oletuspakkauksessa, sijaitsevaan `MyClass`-nimiseen luokkaan. Kolmas esimerkki täsmää minkä tahansa luokan yksiparametriseen `myMethod`-nimiseen metodiin.

Listaus 5.3: Esimerkkejä möykkyhahmoista.

```
1 my.pkg.*:***
2 {,**.}MyClass::***
3 **::myMethod(?**)
```

Kumpikin *regex*- ja *glob*-parametreista on itsessään valinnainen, mutta vähintään toisella parametreista tulee olla jokin muu kuin oletusarvo. Jos kummankin parametrin yhdistetty hahmojoukko on `@AccessFromAnnotated`-määreen määrittelemän vakion `NEVER_MATCHING` arvo, määre ei salli lainkaan pääsyä kohteeseen.

Hahmojen sovitus tapahtuu merkistöherkästi. Toisin sanoen isot ja pienet kirjaimet tulkitaan eri merkeiksi. Hahmot on lisäksi suunniteltava sellaisiksi, että ne täsmäävät kokonaisuudessaan haluttuihin aksessoijaviitteisiin, koska aksessoijaviite sovitetaan hahmoihin aina kokonaisuudessaan. Pääsyn sallimiseksi ei siten riitä, että aksessoijaviite sisältää hahmon mukaisen merkkijonon eli täsmää ainoastaan aksessoijaviitteen osaan.

Hahmoissa on mahdollista käyttää ennalta määriteltyjä muuttujia, jotka hahmoa sovitettaessa korvataan muuttujan arvolla. Muuttuja voi edustaa esimerkiksi rajoitettavan

elementin pakkauksen, luokan tai metodin nimeä. Muuttujien ansiosta täsmälleen samaa hahmoa voi käyttää eri elementtien rajoittamisessa. Muuttujat ilmaistaan hahmoissa `${` ja `}` -merkkien välissä. Listauksessa 5.4 on esimerkki möykkyhahmosta, joka täsmää kaikkiin rajoitettavan elementin kanssa samassa pakkauksessa sijaitseviin luokkiin. Hahmossa on käytetty muuttujaa *PackageNamePrefix*, joka korvataan sovituksen yhteydessä rajoitetun elementin pakkauksen nimellä mukaan lukien sitä seuraava piste. Kaikki käytettävissä olevat muuttujat on kuvattu liitteessä 1.

Listaus 5.4: Esimerkki möykkyhahmosta, joka täsmää kaikkiin samassa pakkauksessa oleviin luokkiin sisäluokat mukaan luettuna.

```
1 ${PackageNamePrefix}*{,$**}::***
```

@SelfAccess — oliotason yksityisyys

`@SelfAccess`-pääsymääre rajoittaa elementtiin pääsyn ainoastaan sen esittelevän luokan ilmentymän sisään. Toisin sanoen saman luokan toisesta ilmentymästä ei ole pääsyä rajoitettuun elementtiin. Rajoitettava elementti voi olla olion kenttä tai metodi. Luokan staattinen jäsen ei voi olla rajoituksen kohteena. Pääsymääre ei rajoita pääsyä olion yliluokilta perimiinsä jäseniin, vaan rajoitukset perintähierarkiassa on toteutettavissa Javan sisäänrakennetuilla pääsymääreillä ja sisäkkäisten luokkien tapauksessa `@SelfAccess`-määreen parametrin avulla.

Määre sallii pääsyn rajoitettuun jäseneen ainoastaan, kun siihen viitataan pelkällä jäsenen tunnisteella tai `this`- tai `super`-avainsanojen kanssa. Kyseisten avainsanojen edessä saa Javan syntaksin mukaisesti käyttää luokan nimeä tarkentamaan, minkä luokan jäseneen viitataan. Liitteessä 1 on esitetty määreen sallimien aksessointilausekkeiden muodollinen syntaksi.

`@SelfAccess` ottaa ainoana parametrinaan *value* luettelotyypin `SelfAccessScope`, joka määrää rajoitettavan jäsenen yksityisyyden rajauksen. Rajaus voi olla `SELF`, jolloin jäsen on saatavissa ainoastaan sen esitelleen olion sisällä, tai `INNER`, jolloin jäsen on saatavissa edellisen lisäksi siihen sidoksissa olevien sisäluokkien ilmentymistä sekä sisäkkäisistä aliluokista. Jos rajausta ei erikseen anneta, oletusrajaus on `INNER`.

`@SelfAccess`-määreen rajoitetta ei määritelty muodostimille, koska olion sisäisen muodostimen semantiikka ei ole itsestään selvä. Kuten luvussa 3.2 todettiin, Scalassa oliolle yksityiset ja muut lisämäärein rajoitetut muodostimet ovat osoittautuneet semanttisesti ongelmalliseksi. Niistä ei edelleenkään ole täsmällistä määrittelyä, vaikka kieli salliikin kyseiset rajoitteet. Scalan lisäksi muita luvussa 3.2 käsiteltyjä olion sisäisen saatavuuden sallivia kieliä ovat Smalltalk ja Eiffel. Smalltalkissa ongelmaa ei synny, koska ainoastaan olion kentät ovat olion sisäisiä. Eiffelissä luokan ilmentymän luotavuus on erotettu alustusproseduurin saatavuudesta, minkä seurauksena saatavuuden ja luotavuuden semantiikka

on yksikäsitteinen. Kytkettävän pääsynvalvontajärjestelmän suunnitteluprosessin aikana ei tullut esiin sellaisia käyttötapauksia, jotka olisivat edellyttäneet olion sisäisiä muodostimia. Näiden seikkojen valossa on perusteltua rajoittaa `@SelfAccess`-määre rajoittamaan ainoastaan kenttiä ja metodeja.

@AccessFromTest — saatavuus testikoodista

`@AccessFromTest`-pääsymääre rajoittaa elementtiin pääsyn ainoastaan testikoodista. Rajoitettava elementti voi olla luokka, kenttä, muodostin tai metodi. Rajoitettu elementti ei saa olla yksityinen. Pääsymääre on parametriton.

Testikoodi on koodia, joka suoritetaan ohjelmiston testaamiseksi tyypillisesti käännöksen yhteydessä tai erillisessä testiajossa. Testikoodia ei tarvita ohjelman suorituksen aikana, eikä testikoodin tule sijaita ohjelmiston suoritusaikaisessa luokkapolussa. `@AccessFromTest`-määre ei ota kantaa menetelmiin, kuinka koodin luokitellaan olevan testikoodia, josta pääsy sallitaan. Määreen tarkastajatoteutukset saavat vapaasti määritellä testikoodin luokitteluperiaatteensa.

@AccessWithin — rakenteen mukaan rajoitettu saatavuus

`@AccessWithin`-pääsymääre rajoittaa elementtiin pääsyn sitä ympäröivän pakkaus- ja luokkarakenteen mukaan. Sillä säädellään saatavuutta sisäkkäisissä luokissa ja pakkauksen sisällä. Rajoitettava elementti voi olla jäsenluokka, kenttä, muodostin tai metodi sekä tietyillä parametreilla myös päätason luokka.

Pääsymääre ottaa ainoana parametrinaan *value* luettelotyypin `AccessWithinScope`, joka esittää saatavuustason, jonka mukaan saatavuus rajataan. Saatavuustaso on pakollinen parametri. Mahdolliset saatavuustasot kuvauksineen on lueteltu taulukossa 5.1. Päätason luokkien rajoittamiseen sallittuja saatavuustasoja ovat ainoastaan taulukon P-sarakkeessa tähdellä (*) merkityt.

Taulukko 5.1: `@AccessWithin`-pääsymääreen saatavuustasot ja niiden kuvaukset.

Saatavuustaso	P	Kuvaus
`NOTHING`	*	ei saatavissa lainkaan
`CLASS`		saatavissa ainoastaan elementin esittelevässä luokassa
`NESTED`		saatavissa elementin esittelevässä sekä kaikissa sen sisäkkäisissä luokissa
`OUTER`		saatavissa elementin esittelevässä sekä kaikista sitä ulomman tason luokissa
`NESTED_AND_OUTER`		saatavissa elementin esittelevässä luokassa, sen sisäkkäisissä sekä ulomman tason luokissa
`TOP_LEVEL`		saatavissa päätason luokassa ja kaikkialla sen sisäkkäisissä luokissa
`PACKAGE`	*	saatavissa saman pakkauksen sisällä
`ALL`	*	saatavissa kaikkialla

5.4 Kytkettävä pääsynvalvontakehys Javaan

Kytkettävä pääsynvalvontakehys muodostaa perustan kytkettävän pääsynvalvontajärjestelmän toteuttamiseksi Javaan. Sen tarkoituksena on yksinkertaistaa ja minimoida pääsymääreiden ja niiden tarkastajien toteuttamiseksi tarvittavaa työtä. Tässä luvussa kuvaillaan suunnitelma kytkettävästä pääsynvalvontakehyksestä sekä esitellään prototyyppi, joka toteuttaa pääsynvalvontakehyksen suunnitelluista ominaisuuksista keskeisimmät.

Pääsynvalvontakehyksen toimintaperiaate

Luvussa 4.3 todettiin Javassa olevan valmiudet kytkettävän tyyppijärjestelmän toteuttamiseksi. Kytkettävän pääsynvalvontajärjestelmän vaatimukset ohjelmointikielelle ovat hyvin samankaltaiset: tarvitaan metadatakonstruktio pääsymääreiden ilmaisemiseksi ja kääntäjään integroitavat metadatan käsittelijä. Javan annotaatiot soveltuvat pääsymääreiden ilmaisemiseksi ja annotaatioprosessorit pääsymääretarkastajien kytkemiseksi Javakääntäjään. Kyseiset tekniikat ovat olleet Java-kielessä jo versiosta 6 lähtien. Pääsymääreiden ilmaisemiseksi ei tarvita Java 8:n uusia tyyppiannotaatioita, vaan esittelyjen yhteyteen liitettävät annotaatiot riittävät. Annotaatioprosessorin tulee kyetä lukemaan ohjelman abstraktia syntaksipuuta, jotta määretarkastaja pystyy havaitsemaan viittaukset rajoitettuihin elementteihin ja tarkastamaan niiden kelvollisuuden. Kuten luvussa 4.4 esitelty Checker Framework on osoittanut, syntaksipuun lukeminen on mahdollista. Syntaksipuun muuntelua ei tarvita, koska pääsynvalvontajärjestelmä ei aiheuta mitään muutoksia ohjelman suoritusaikaiseen toimintaan.

Pääsynvalvontakehyksen keskeinen tehtävä on poimia ohjelmasta viittaukset rajoitettuihin elementteihin ja antaa viittaukset kullekin pääsymääretarkastajalle tarkastettavaksi. Kehys poimii myös rajoitettujen elementtien esittelyt ja antaa pääsymääretarkastajan tehtäväksi tarkastaa esittelyn oikeellisuus. Kaiken tämän kehys toteuttaa kulkemalla käännettävän ohjelman abstraktin syntaksipuun läpi poimien poimia sieltä kaikki viittaukset potentiaalisesti rajoitettuihin ohjelmaelementteihin sekä pääsymäärein rajoitettujen ohjelmaelementtien esittelyt.

Kukin pääsymääretarkastaja on tyypillisesti toteutettu tarkastamaan vain tietyn määreen tai samanlaisen saatavuusperusteen mukaisten pääsymääreiden asettamat rajoitteet. Pääsymääretarkastajan tehtäviin kuuluu myös tarkastaa, että pääsymäärettä käytetään siten kuin sen määrittely sallii. Tyypillisesti määreannotaatiotyypin esittelyssä asetetaan rajoitteet, kuinka määreannotaatiota on mahdollista käyttää. Annotaatio voi esimerkiksi olla sallittu ainoastaan luokan jäsenille. Java-kääntäjä tarkastaa automaattisesti kyseisten rajoitteiden toteutumisen. Annotaatiotyypin esittelyssä ei kuitenkaan ole mahdollista ilmaista esimerkiksi, että annotaatio on sallittu ainoastaan luokan ei-staattisille jäsenille. Tämän tyyppisten rajoitteiden tarkastaminen kuuluu pääsymääretarkastajille.

Pääsynvalvontakehys poimii käännettävästä koodista kytkettävillä pääsymääreillä rajoitetut elementit, ja valitsee kutakin pääsymäärettä tukevan tarkastajan, jolle kehys antaa tehtäväksi tarkastaa määreen oikeanlainen käyttö. Pääsynvalvontakehys poimii myös viittaukset potentiaalisesti rajoitettaviin kohteisiin, ja selvittää kohteiden saatavuuden määräävät pääsymääreet sekä niiden asettamien rajoitteiden tarkastajat. Kehys antaa tarkoituksenmukaisille tarkastajille tehtäväksi niiden tukemien viittausten laillisuuden tarkastamisen. Pakkausten mahdolliset kytkettävien pääsymääreiden asettamat rajoitteet tarkastetaan aksessoitaessa pakkaukseen kuuluvaa luokkaa pakkauksen ulkopuolelta. Rajoitetta ei tarkasteta aksessoitaessa pakkauksen sisällä samaan pakkaukseen kuuluvaa luokkaa.

Javan sisäänrakennettu pääsynvalvonta tarkastaa varsinaisen aksessoinnin lisäksi luokan alussa esiteltyjen `import`-lauseiden tyyppien saatavuuden [GJS+15, s. 164]. Kytkettävässä pääsynvalvontakehyksessä `import`-lauseet on Javasta poiketen jätettävä tarkastamatta, koska esimerkiksi metodiperusteinen pääsynvalvonta voi rajoittaa pääsyn olemaan sallittu ainoastaan luokan tietyistä metodeista. Vaikka tällä tavoin rajoitettua käännösyksikön nimiavaruuteen tuotua luokkaa käytettäisiin ainoastaan sallittujen metodien sisällä, `import`-lause tuottaa virheen, koska se ei ole sallitun metodin sisällä. Lauseen tarkistamatta jättäminen ei aiheuta merkittävää haittaa, vaikka nimiavaruuteen tuotuun luokkaan ei olisi pääsyä, sillä jokainen aksessointi kyseiseen luokkaan tarkastetaan.

Pääsymääreiden aliakset

On tilanteita, joissa yhtä tai useampaa pääsymäärettä käytetään toistuvasti enimmäkseen samoilla parametreilla. Ohjelmoijan kannalta olisi käytännöllistä, jos kyseisen määreyhdistelmän aikaansaamiseksi tarvittaisiin ainoastaan yksi, kenties jopa täysin parametriton pääsymääre. Toisaalta ohjelmoija saattaa haluta käyttää pääsymääreitä omilla, sovelluskohtaista semantiikkaa kuvaavilla annotaatioilla, joihin liittyy pääsyrajoitteita. Tällaisia tilanteita varten kytkettävään pääsynvalvontakehykseen on suunniteltu tuki pääsymäärealiaksille, lyhyemmin ilmaistuna aliaksille.

Aliakset ovat pääsymääreen kaltaisia Javan annotaatioita, jotka edustavat yhtä tai useampaa todellista kytkettävää pääsymäärettä. Pääsynvalvontakehys muuntaa kuhunkin ohjelmaelementtiin liitetyt aliakset tarkastusprosessin aikana aliasten edustamiksi todellisiksi pääsymääreiksi. Siten edellä kuvatuissa tilanteissa pääsymääreiden räätälöintiä varten ei tarvitse toteuttaa uusia pääsymääretarkastajia. Sen sijaan on yksinkertaisempaa tehdä yhdestä tai useammasta pääsymääreestä alias.

Alias edustaa kutakin edustamaansa pääsymäärettä joko pakollisena tai valinnaisena. Aliaksen pakollisena edustamat pääsymääreet tulkitaan aina voimassa oleviksi. Valinnaiset pääsymääreet tulkitaan voimassa oleviksi ainoastaan, jos aliaksella annotoitua elementtiä ei ole annotoitu yksikäsitteisesti kyseisellä pääsymääreellä eikä toisella aliaksella, joka edustaa kyseistä pääsymäärettä pakollisena määreenä. Toisin sanoen ohjelmaelementtiin yksikäsitteisesti määritelty pääsymääre tai aliaksen pakollinen pääsymääre korvaavat valinnaisen pääsymääreen. Tällä on merkitystä silloin, kun valinnaisen ja korvaavan pääsymääreen parametrit eroavat toisistaan.

Tulkkaus aliaksesta pääsymääreeksi

Kytkettävä pääsynvalvontakehys muuntaa aliakset pääsymääreiksi niin kutsuttuja aliastulkkeja käyttäen. Aliastulkit ovat kehykseen kytkettäviä komponentteja, joiden tehtävä on muuntaa kunkin aliastulkin tukemat aliakset pääsymääreiksi. Tarkastaessaan ohjelmaelementin pääsymääreitä pääsynvalvontakehys ohjaa sellaiset elementin annotaatiot, jotka eivät ole pääsymääreitä, aliastulkeille tulkattavaksi. Lopputuloksena saadut pääsymääreet annetaan pääsymääretarkastajille tarkastettavaksi.

Ohjelmoija voi vapaasti rakentaa omia aliastulkkejaan ja kytkeä niitä vapaasti pääsynvalvontakehykseen. Seuraavaksi esitellään korkean tason suunnitelmat kahdesta yleiskäyttöisestä aliastulkista osaksi kytkettävää pääsynvalvontakehystä.

Ensimmäinen aliastulkeista mahdollistaa uusien aliasten määrittelyn deklaratiivisesti. Ohjelmoijan riittää esitellä uusi annotaatiotyyppi, joka on meta-annotaatioin merkitty valittujen pääsymääreiden aliakseksi. Meta-annotaatioilla määritellään, mitä pääsymääreitä alias edustaa, mitkä edustetuista määreistä ovat valinnaisia ja mitkä pakollisia, edustettujen

määreiden parametrien vakioarvot, sekä sidotaan alias-annotaation mahdolliset parametrit edustettujen pääsymääreiden parametreihin.

Joskus annotaation tarkoitus on dokumentoida koodiin, että annotoituun elementtiin liittyy semanttisesti pääsyrajoitteita. Yksistään annotaatio ei kuitenkaan pysty pääsyrajoitetta valvomaan, vaan rajoitteen noudattaminen jää tavallisesti ohjelmoijan vastuulle. Luvussa 5.1 mainittu `@VisibleForTesting` on yksi esimerkki tällaisesta annotaatiotyypistä. Kytkettävän pääsynvalvonnan avulla annotaation kuvaamaa rajoitetta on mahdollista valvoa automaattisesti. Olisi kuitenkin kömpelöä, jos ohjelmoija joutuisi jokaisen rajoitteesta kertovan annotaation rinnalle lisäämään myös varsinaisen pääsymääreannotaation. Sen sijaan on käytännöllisempää tehdä rajoitteesta kertovasta annotaatiosta alias. Tällä tavoin annotaatioon kytketään sen semantiikkaa valvova toteutus.

Toinen yllä mainituista aliastulkeista on tarkoitettu edellä kuvattuun tilanteeseen. Se tunnistaa aliasannotaatiotyypin vertaamalla annotaation tyyppiä ennalta määriteltyihin annotaatiotyyppeihin tai annotaation tyypin nimeä ennalta määriteltyyn hahmoon. Vertailun täsmätessä se tulkitsee annotaation edustamikseen pääsymääreiksi. Käytännössä kyseinen aliastulkki on abstrakti toteutus, joka tulee periyttää. Periytettyyn aliastulkkiin määritellään annotaatioilla, millaisiin annotaatiotyyppeihin tai annotaatiotyyppien nimien hahmoon sen tulee täsmätä, ja edellisen aliastulkin yhteydessä mainituilla meta-annotaatioilla, mitä pääsymääreitä täsmäävät aliakset edustavat. Jos vertailu tehdään annotaation tyypeillä, tulee kyseisten annotaatioiden olla käännösaikaisessa luokkapolussa. Hahmoina on mahdollista käyttää säännöllistä lauseketta tai möykkyhahmoa samoin kuin luvussa 5.3 kuvatun `@AccessFromMatching`-pääsymääreen yhteydessä.

Oletusmääreet

Kytkettävät pääsymääreet ilmaistaan Javan annotaatioina. Koska kukin määreistä määrittelee vain yhden saatavuusperusteen, eri määreitä joutuu usein käyttämään yhdessä. Tästä seuraa, että saavuttaakseen optimaalisen saatavuuden annotaatioiden määrä lisääntyy sietämättömästi, jos jokainen luokka ja sen jäsen täytyy annotoida usealla kytkettävällä pääsymääreellä. Edellä esiteltyjen aliasten avulla samanaikaisesti käytettävien pääsymääreiden määrää on mahdollista vähentää, mutta edelleen jokainen rajoitettava elementti on annotoitava erikseen.

Tehdään oletus, että tyypillisesti ohjelmoija haluaa saman tyyppisille ohjelmaelementeille samanlaisen saatavuuden. Esimerkiksi ohjelmoija haluaa kaikkien yksityisten jäsenten olevan olioyksityisiä, tai kaikkien pakkauksen sisäisten kenttien olevan saatavissa ainoastaan testikoodista. Tällaisessa tilanteessa annotoinnin tarvetta vähentää merkittävästi, jos pakkaus- tai luokkakohtaisesti on mahdollista määritellä ohjelmaelementtien oletussaatavuus.

Oletusmääreet tekevät oletussaatavuuden asettamisen mahdolliseksi. Ne ovat pakkauksiin ja luokkiin liitettäviä annotaatiotyyppejä, jotka on meta-annotaatiolla merkitty oletus-

määreiksi. Oletusmääreen parametrina annetaan ne ohjelmaelementit, esimerkiksi kenttä tai metodi, sekä niiden Javan sisäänrakennettu saatavuustaso, esimerkiksi yksityinen tai julkinen, joiden oletussaatavuus halutaan asettaa. Lisäksi parametrina annetaan kytkettävä pääsymääreannotaatio tai aliasannotaatio, joka määrittelee edellä mainitut ehdot täyttävien ohjelmaelementtien oletussaatavuuden.

Oletusmääre on voimassa sillä annotoidun pakkauksen tai luokan sisällä. Sisäkkäiselle luokalle on mahdollista asettaa uusi oletusmääre, joka sen sisällä korvaa ulomman luokan tai pakkauksen oletusmääreen. Oletusmääreitä on mahdollista asettaa useita, jolloin on mahdollista asettaa esimerkiksi eri elementtityypeille erilaiset määreet, tai samoille elementeille useita eri pääsymääreitä.

Kun ohjelmaelementille asetetaan kytkettävä pääsymääre, se korvaa oletusmääreeksi asetetun samantyyppisen ja vain samantyyppisen määreen. Esimerkiksi jos oletusmääreeksi on asetettu `@SelfAccess @AccessFromAnnotated(MyAnnotation)` ja ohjelmaelementille `@AccessFromAnnotated(OtherAnnotation)`, on ohjelmaelementin todellinen saatavuus `@SelfAccess @AccessFromAnnotated(OtherAnnotation)`. Korvaaminen suoritetaan sen jälkeen, kun kaikki aliakset on tulkattu pääsymääreiksi.

Koska Javassa annotaation parametriksi annettava annotaation tyyppi on aina esiteltävä yksikäsitteisesti, jokaista oletusmääreenä käytettävää kytkettävää pääsymääre- ja aliasannotaatiotyyppiä kohti on toteutettava oletusmääreannotaatiotyyppi. Nimeämiskäytännön mukaisesti oletusmääre nimetään lisäämällä pääsymääreen nimen perään `ByDefault`, esimerkiksi `@SelfAccessByDefault`. Pääsynvalvontakehys ei tätä kuitenkaan pakota. Jotta oletusmääreitä olisi mahdollista asettaa useita, on oletusmääreannotaatiotyypin oltava toistuva.

Omien pääsynvalvontajärjestelmien toteuttaminen

Yksinkertaisimmillaan pääsynvalvontajärjestelmä koostuu yhdestä kytkettävästä pääsymääreestä sekä sen tarkastajasta. Sen enempää ei vaadita uuden pääsynvalvontajärjestelmän toteuttamiseksi kytkettävään pääsynvalvontakehykseen. Käytännössä, jos pääsymäärettä halutaan käyttää deklaratiivisesti määritellyn määrealiaksen osana tai oletusmääreenä, on näitä varten esiteltävä vielä omat annotaatiotyypit.

Kytkettävä pääsymääre esitellään pääsynvalvontakehykseen Javan annotaatiotyyppinä, joka on annotoitu meta-annotaatiolla `@AccessModifier`. Sen lisäksi pääsymääreannotaation tulee tallentua luokkatiedostoihin sekä tulla dokumentoiduksi Javadoc-rajapintadokumentaatioon. Lisäksi määreannotaatio saa olla liitettävissä ainoastaan ohjelmaelementtien esittelyihin. Toisin sanoen määreannotaatiotyypeillä tulee olla edellisen lisäksi meta-annotaatiot `@Retention(RetentionPolicy.CLASS)`, `@Documented` ja `@Target(...)`. Määreannotaatioiden ei ole kiellettyä olla olemassa myös suoritusaikaisesti (`RetentionPolicy.RUNTIME`), mutta tyypillisesti sille ei ole tarvetta, joten sitä ei suositella.

`@Target`-meta-annotaation parametreina ei saa olla sellaisia elementtityyppejä, jotka eivät määreen määrittelyn mukaan ole rajoitettavissa. Määrettä ei esimerkiksi saa esitellä tyyppiannotaationa.

Pääsymääretarkastaja esitellään luokkana, joka toteuttaa `ModifierChecker`-rajapinnan. Pääsynvalvontakehys tarjoaa abstraktin toteutusrangan uuden tarkastaja toteuttamisen yksinkertaistamiseksi. Yksinkertaisimmillaan tarkastajan tulee ainoastaan kertoa, minkä pääsymääreen aksessointeja se tarkastaa, ja toteuttaa kyseinen tarkastus. Jos pääsymääreen esittelysäännöt eivät ole toteutettavissa yksistään määreannotaatiotyypin kohdeelementtien rajaamisella, on tarkastajan toteutettava myös esittelysääntöjen tarkastus.

Ennen kuin pääsynvalvontakehys antaa pääsymääretarkastajalle tehtäväksi tarkastaa aksessoinnin laillisuus, se on selvittänyt, missä kohdassa abstraktia syntaksipuuta aksessointi tapahtuu, mikä on aksessoinnin kohde-elementti, ja mikä on se kohde-elementin pääsymääre ja sen parametrit, jonka tarkastaja tarkastaa. Kehys on tässä vaiheessa tulkannut myös aliakset todellisiksi pääsymääreiksi. Tarkastajan on näiden tietojen pohjalta varsin suoraviivaista suorittaa tarkastus. Vastaavasti pääsymääreen esittelysääntöjen tarkastamista varten pääsynvalvontakehys on selvittänyt rajoitettavan elementin sekä tarkastajan tukeman pääsymääreen parametreineen.

Pääsynvalvontajärjestelmä kytketään pääsynvalvontakehykseen antamalla kehykselle pääsymääretarkastajan täydellinen nimi (fully qualified name, FQN) kääntäjän komentorivioptiona. Toinen vaihtoehto pääsymääretarkastajan kytkemiseksi kehykseen on periyttää kehyksen `AccessChecker`-annotaatioprosessori ja lisätä uusi tarkastaja sen oletuksena käyttämien tarkastajien listaan. Sen jälkeen käytetään alkuperäisen sijaan periytettyä annotaatioprosessoria käännöksen yhteydessä.

Testi- ja tuotantokoodista tapahtuvan aksessoinnin havaitseminen

Luvussa 5.3 esitelty `@AccessFromTest`-määre ei määrittele, kuinka pääsymääretarkastajan tulisi päätellä, tapahtuuko aksessointi testikoodista vai tuotantokoodista. Tässä luvussa esitellään kolme tapaa, kuinka päättely on mahdollista toteuttaa.

Javan yleisimmät yksikkötestikehykset JUnit 4[4] ja TestNG[5] tunnistavat testiluokkien testimetodit `@Test`-annotaatiosta. Myös kehitteillä oleva JUnit 5[6] toimii samoin. Sen sijaan tuotantokoodissa tuskin käytetään `@Test`-nimisiä annotaatioita. Niinpä yleispätevä mekanismi testikoodin tunnistamiseksi on etsiä luokasta `@Test`-nimistä annotaatiota sen tyypin pakkauksesta riippumatta. Yhdenkin annotaation löytyessä luokka voidaan luokitella testikoodiksi ja sallia pääsy `@AccessFromTest`-määreellä rajoitettuun elementtiin.

[4] JUnit 4: `http://junit.org/junit4/`
[5] TestNG: `http://testng.org/`
[6] `JUnit~5:http://junit.org/junit5/`

JUnitin vanhempi versio 3[7] on yhä laajalti käytössä vanhoissa ohjelmistoissa. Se ei tue annotaatioita, joten testiluokkien tunnistamiseen on käytettävä muuta tapaa. Kaikki JUnit 3:n testiluokat perivät jonkin JUnitin abstrakteista luokista. Näin ollen luokat, jotka perivät jonkin `junit.framework`-pakkauksessa sijaitsevan luokan, voidaan päätellä testiluokiksi.

Kolmas tapa toteuttaa testikoodista tapahtuvan aksessoinnin päättely on tehdä se kääntäjän komentorivioption perusteella. Tyypillisesti tuotantokoodiluokat ja testiluokat käännetään erillään toisistaan. Käännökseen voi tällöin lisätä komentorivioption, joka kertoo pääsymääretarkastajalle, käännetäänkö tuotantokoodia vai testikoodia. Tuotantokoodia käännettäessä pääsyä ei sallita koskaan, testikoodia käännettäessä pääsy sallitaan aina. Tämä on varmin tapa päätellä, mistä aksessointi tapahtuu, mutta vaatii kehittäjältä vaivaa asettaa eri optiot tuotanto- ja testikäännöksiin.

Pääsynvalvontakehyksen prototyyppi

Tässä luvussa esitellystä pääsynvalvontakehyksestä on toteutettu prototyyppi osoittamaan, että kytkettävä pääsynvalvontajärjestelmä on mahdollista toteuttaa Javaan. Prototyyppi käyttää hyväkseen luvussa 4.4 esiteltyä Checker Frameworkia perustoimintoihinsa. Riippuvuus Checker Frameworkista on keskeinen syy, että prototyyppi vaatii toimiakseen Javan version 8, vaikka pääsynvalvontakehyksen toteutukseen tarvittava tekniikka on ollut olemassa Javan versiosta 6 lähtien.

Prototyypin keskeinen komponentti on annotaatioprosessori `AccessChecker`. Annotaatioprosessorin alustusvaiheessa se päättelee käyttöön otettavat pääsymääretarkastajat sisäänrakennettujen asetusten sekä komentoriviparametrien perusteella ja alustaa valitut pääsymääretarkastajat sekä niiden ohjaimen. `AccessChecker` periytyy Checker Frameworkin `SourceChecker`-luokasta, joka ohjaa annotaatioiden prosessoinnin suorituksen abstraktin syntaksipuun vierailijaluokalle. `AccessChecker`in rinnalle toteutettu vierailijaluokka poimii ohjelmasta kaikki esittelyt ja aksessoinnit ja ohjaa niiden käsittelyn tarkastajien ohjaimelle.

Pääsymääretarkastajien ohjaimen tehtävä on ohjata pääsymääreiden tarkastukset asianmukaiselle pääsymääretarkastajalle. Ohjain pitää yllä tietoa, mikä tarkastaja suorittaa minkäkin määreen tarkastukset. Se selvittää, mitä määreitä kussakin esittelyssä tai aksessoinnin kohteessa on käytetty, ja kutsuu sen mukaan sopivia tarkastajia. Se huolehtii myös pakkausten pääsymääreiden tarkastamisesta käyttäen asianmukaista pääsymääretarkastajaa, kun luokan nimeä aksessoidaan esittelevän pakkauksen ulkopuolelta. Ohjain välittää mahdolliset tarkastuksissa ilmenneet virheet annotaatioprosessorille raportoitavaksi edelleen käyttäjälle.

[7] JUnit 3: `http://junit.sourceforge.net/junit3.8.1/javadoc/index.html`

Loppuosa prototyypistä koostuu pääsymääretarkastajista sekä automatisoiduista testeistä, jotka todentavat, että kehys toimii oikein testitapausten määrittelemällä tavalla. Kehyksen luokkien testit ovat tavallisia yksikötestejä, mutta kaikki määretarkastajien testit perustuvat Checker Frameworkin CheckerFrameworkTest-testiluokkaan, joka suorittaa määrättyjen testitapausluokkien käännöksen käyttäen määrättyjä tarkastajia. Kussakin testitapausluokassa virheelliset aksessoinnit on merkitty virhetilannetta vastaavalla kommentilla. Testi tarkastaa, että kaikki merkityt virhetilanteet toteutuivat käännöksen aikana, ja että odottamattomia virhetilanteita ei syntynyt. Vastaava testi suoritetaan myös koko prototyypin lähdekoodille, jonka odotusarvona on, ettei virhetilanteita synny lainkaan.

Pienenä yksityiskohtana ja esimerkkinä prototyypin käytöstä mainittakoon, että sen omien pakkausten saatavuus on rajoitettu ainoastaan sellaisiin ohjelmaelementteihin, jotka on annotoitu kehykseen liittyviksi. Kehyksen omat pakkaukset on esimerkiksi annotoitu näin. Koska pääsynvalvontakehys on tavallisesti käännettävän ohjelman käännösaikaisessa luokkapolussa, muttei suoritusaikaisessa, on tärkeää, ettei käännettävästä koodista pysty vahingossa luomaan riippuvuutta kehykseen. Edellä mainittu rajoitus estää riippuvuuden syntymisen. Kuitenkin kehystä laajentava kehyksen ulkopuolinen komponentti voi annotoida itsensä kehykseen liittyväksi, ja näin luvallisesti aksessoida kehystä.

Pääsynvalvontakehyksen prototyypin toteutusaste

Edellä esitelty pääsynvalvontakehyksen prototyyppi ei ole kokonainen toteutus, vaan siihen on sisällytetty ainoastaan kehyksen keskeisimmät perusosat: toimintaa ohjaavan peruskehyksen, joka toteuttaa kytkettävyyden mahdollistamalla käytettävien pääsymääretarkastajien valitsemisen, sekä luvussa 5.3 määriteltyjen pääsymääreiden tarkastajat pääpiirteissään. Alla on selostettu prototyypin puutteet suhteessa tässä luvussa määriteltyihin pääsynvalvontakehyksen ominaisuuksiin.

Prototyypin `@AccessFromMatching`-määreen tarkastaja muodostaa aksessoijaviitteen ainoastaan metodeista ja muodostimista tapahtuvasta aksessoinnista. Aksessoijaviite muodostetaan täsmällisen kutsuympäristön mukaisesti. Jos pääsy ei sen perusteella ole sallittu, prototyyppi ei yritä muodostaa mahdollisen ulomman kutsuympäristön mukaista aksessoijaviitettä. Hahmoista prototyypin toteutus tukee ainoastaan säännöllisiä lausekkeita. Möykkyhahmojen perusteella tapahtuvaa hahmonsovitusta ei ole toteutettu, kuten ei myöskään ennalta määriteltyjen muuttujien korvaamista niiden arvoilla.

`@AccessFromInherited`-määreen tarkastajan toteutus ei tarkasta Java-kielen mukaisia lisärajoitteita koskien `protected`-määreellä rajoitettujen elementtien aksessointia [GJS+15, §6.6.2]. Kaikkien muiden pääsymääreiden tarkastajat on toteutettu määrittelyn mukaisesti. `@AccessFromTest`-määreen tarkastaja toteuttaa testikoodista tapahtuvan aksessoinnin päättelyn edellisessä luvussa kuvatuista tavoista `@Test`-annotaation tunnistukseen perustuen.

Oletusmääreiden asettamista ja pääsymääreiden aliasointimekanismia ei toteutettu prototyyppiin. Niiden tarkoitus on yksinkertaistaa ja helpottaa pääsynvalvontakehyksen käyttöä, mutta niiden olemassaolo ei ole välttämätöntä määriteltyjen pääsymääreiden mukaisen pääsynvalvonnan toteutumiseksi.

Kytkettävän pääsynvalvontakehyksen prototyyppi toteuttaa siis pääsynvalvonnan keskeiset toiminnot. Puutteet ovat pieniä yksityiskohtia tai käyttömukavuutta parantavia ominaisuuksia. Näin ollen toteutetun prototyypin voidaan todeta osoittavan, että kytkettävä pääsynvalvonta on mahdollista toteuttaa Javaan, ja että suunnitellut pääsymääreet toimivat käytännössäkin suunnitellusti.

5.5 Optimaalinen pääsynvalvonta

Luvussa 5.1 esiteltiin neljä tilannetta, joissa optimaalinen saatavuus on mahdotonta saavuttaa Javan sisäänrakennetuilla pääsymääreillä, ja rajoitettavien elementtien saatavuus jää laajemmaksi kuin olisi tarve. Tässä luvussa osoitetaan, kuinka kussakin edellä mainituista tilanteista on mahdollista saavuttaa optimaalinen saatavuus kytkettävän pääsynvalvontajärjestelmän avulla. Kustakin tilanteesta esitetään ohjelmaesimerkki sekä kytkettävän pääsynvalvontakehyksen prototyypin kanssa suoritetun käännöksen tulokset.

Asiakkaan, rajapinnan ja toteutuksen välinen saatavuus

Tiedon piilottamisen periaate on pitää yksityiset toteutusyksityiskohdat piilossa. Luvussa 5.1 esiteltiin tilanne, jossa toteutus jakautuu useaan pakkaukseen ja sisältää pakkausten välisiä keskinäisiä riippuvuuksia. Toteutusluokkien on oltava silloin julkisia. Seuraavaksi tarkastellaan kolmea eri ratkaisumallia, kuinka luvussa 5.3 esiteltyjä pääsymääreitä käyttäen toteutus on mahdollista piilottaa asiakkaalta. Kaikki niistä perustuvat valikoivaan saatavuuteen, mutta saatavuuden peruste vaihtelee eri ratkaisumalleissa.

Toteutus on mahdollista piilottaa tyyppiperusteisesti, jos kaikki rajapinnan luokat toteuttavat jonkin yhteisen rajapinnan. Tällöin pääsy toteutusluokkiin sallitaan ainoastaan edellä mainitun yhteisen rajapintaluokan toteuttavista luokista. Koska toteutusluokat toteuttavat myös rajapinnan, toteutuksen eri luokkien väliset riippuvuudet ovat sallittuja. Käytännössä näin ei välttämättä aina ole, joten toteutusluokille voi tarvittaessa esitellä oman yhteisen rajapinnan, jonka kaikki toteutusluokat perivät.

Listauksessa 5.5 on esimerkki toteutuksen tyyppiperusteisesta piilottamisesta. Siinä rajapintaluokat `Api1` ja `Api2` sekä tehdasluokka `Factory` perivät kaikki `Api`-rajapintaluokan. Kaksi toteutusluokkaa `Impl1` ja `Impl2` ovat kumpikin eri pakkauksissa, ja ne rajoittavat `@AccessFromClass`-määreellä pääsyn itseensä ainoastaan `Api`-tyypin luokista. `Client`-luokka aksessoi toteutusluokkia sallitusti tehdasluokan kautta mutta myös luoden suoraan `new`-lausekkeella uuden ilmentymän. Jälkimmäinen johtaa käännösvirheeseen. Selkeyden

vuoksi tässä ja seuraavissa listauksissa `import`-lauseet on jätetty pois paitsi `Client`-luokan yhteydessä, jotta käännöksen tuloksessa esiintyvät rivinumerot täsmäisivät listaukseen.

Toinen vaihtoehto piilottaa toteutus on käyttää annotaatioperusteista pääsynvalvontaa. Rajapinta ja toteutus merkitään omilla annotaatioillaan ja pääsy toteutukseen sallitaan rajapinnan ja toteutuksen annotaatioilla merkityistä luokista.

Listauksessa 5.6 on edellisestä listauksesta sovitettu esimerkki. Siinä esitellään annotaatiotyyppi `@Api`, jolla on merkitty rajapintaluokat `Api1` ja `Api2` sekä tehdasluokka `Factory`. Sitä seuraa annotaatiotyypin `@Impl` esittely, jolla on merkitty eri pakkauksissa sijaitsevat toteutusluokat `Impl1` ja `Impl2`. Kumpaankin on pääsy rajattu `@AccessFromAnnotated`-määreellä ainoastaan `@Api`- tai `@Impl`-annotaatioilla merkitystä koodista. `Client`-luokka aksessoi toteutusluokkia kuten edellisessä esimerkissä.

Kolmas vaihtoehto toteutuksen piilottamiseksi on rajoittaa pääsy hahmonsovitusperusteisesti pakkausnimen mukaan. Pakkauksia on tällöin käytännöllistä käsitellä pakkausnimen mukaan hierarkiana. Pääsy toteutukseen sallitaan tällöin niistä pakkauksista, jotka pakkausnimen mukaan kuuluvat määrättyyn pakkaushierarkiaan.

Listauksessa 5.7 on edellisistä listauksista sovitettu esimerkki. Siinä esitellään rajapintaluokat `Api1` ja `Api2` sekä tehdasluokka `Factory` pakkauksiin `hiddenimpl.pkgs.api` ja `hiddenimpl.pkgs.api.pkg`. Toteutusluokat `Impl1` ja `Impl2` sijaitsevat pakkauksissa `hiddenimpl.pkgs.impl` ja `hiddenimpl.pkgs.impl.pkg`. Toteutusluokkiin pääsy on rajattu `@AccessFromMatching`-määreellä pakkaushierarkioihin `hiddenimpl.pkgs.api` ja `hiddenimpl.pkgs.impl`. Omassa pakkauksessaan sijaitseva `Client`-luokka aksessoi toteutusluokkia kuten edellisissä esimerkeissä.

Listauksissa 5.8, 5.9 ja 5.10 ovat listausten 5.5, 5.6 ja 5.7 käännösten tulokset käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä. Käännösten tuloksista nähdään, että suora viittaus toteutusluokkaan on kussakin tapauksessa johtanut käännösvirheeseen. Viittaukset toteutusluokkien välillä ja rajapinnasta toteutusluokkiin sen sijaan ovat olleet sallittuja.

Kaikissa edellä esitetyissä esimerkeissä pääsymääreet sekä rajapintaa ja toteutusta merkitsevät annotaatiot oli yksinkertaisuuden vuoksi sijoitettu suoraan luokan yhteyteen. Kun luokkia on useita, on käytännöllisempää sijoittaa kyseiset annotaatiot package-info.java-tiedostoon pakkauksen esittelyyn, jolloin pääsyrajoitteet ja merkitsevät annotaatiot koskevat automaattisesti jokaista pakkauksen luokkaa, eikä niitä tarvitse annotoida yksitellen.

Sovelluksesta riippuu, millainen ratkaisumalli toteutuksen yksityiskohtien piilottamiseksi toimii parhaiten. Edellä on esitetyt esimerkit osoittavat, että luvussa 5.3 esiteltyjen pääsymääreiden avulla toteutuksen piilottaminen asiakkaalta on mahdollista myös silloin, kun toteutusluokat jakautuvat eri pakkauksiin, ilman tarvetta jakaa ohjelmaa eri käännösprojekteihin.

Listaus 5.5: Esimerkki tyyppiperusteisesta toteutuksen piilottamisesta.

```
A.1  package hiddenimpl.types.api;
A.2  public interface Api {}
A.3
B.1  package hiddenimpl.types.api;
B.2  public interface Api1 extends Api {
B.3    void method();
B.4  }
B.5
C.1  package hiddenimpl.types.api.pkg;
C.2  public interface Api2 extends Api {
C.3    void method();
C.4  }
C.5
D.1  package hiddenimpl.types.api;
D.2  public class Factory implements Api {
D.3    public static Api1 api1() { return new Impl1(); }
D.4    public static Api2 api2() { return new Impl2(); }
D.5  }
D.6
E.1  package hiddenimpl.types.impl;
E.2  @AccessFromClass(Api.class)
E.3  public class Impl1 implements Api1 {
E.4    @Override
E.5    public void method() { new Impl2(); }
E.6  }
E.7
F.1  package hiddenimpl.types.impl.pkg;
F.2  @AccessFromClass(Api.class)
F.3  public class Impl2 implements Api2 {
F.4    @Override
F.5    public void method() { new Impl1(); }
F.6  }
F.7
G.1  package hiddenimpl.types.client;
G.2  import hiddenimpl.types.api.Factory;
G.3  import hiddenimpl.types.impl.Impl1;
G.4  import hiddenimpl.types.impl.pkg.Impl2;
G.5  public class Client {
G.6    public void access() {
G.7      Factory.api1().method();  // OK
G.8      Factory.api2().method();  // OK
G.9      new Impl1().method();     // Error!
G.10     new Impl2().method();     // Error!
G.11   }
G.12 }
```

Listaus 5.6: Esimerkki annotaatioperusteisesta toteutuksen piilottamisesta.

```
A.1  package hiddenimpl.annos.api;
A.2  public @interface Api {}
A.3
B.1  package hiddenimpl.annos.api;
B.2  @Api
B.3  public interface Api1 {
B.4    void method();
B.5  }
B.6
C.1  package hiddenimpl.annos.api.pkg;
C.2  @Api
C.3  public interface Api2 {
C.4    void method();
C.5  }
C.6
D.1  package hiddenimpl.annos.api;
D.2  @Api
D.3  public class Factory {
D.4    public static Api1 api1() { return new Impl1(); }
D.5    public static Api2 api2() { return new Impl2(); }
D.6  }
D.7
E.1  package hiddenimpl.annos.impl;
E.2  public @interface Impl {}
E.3
F.1  package hiddenimpl.annos.impl;
F.2  @Impl
F.3  @AccessFromAnnotated({Api.class, Impl.class})
F.4  public class Impl1 implements Api1 {
F.5    @Override
F.6    public void method() { new Impl2(); }
F.7  }
F.8
G.1  package hiddenimpl.annos.impl.pkg;
G.2  @Impl
G.3  @AccessFromAnnotated({Api.class, Impl.class})
G.4  public class Impl2 implements Api2 {
G.5    @Override
G.6    public void method() { new Impl1(); }
G.7  }
G.8
H.1  package hiddenimpl.annos.client;
H.2  import hiddenimpl.annos.api.Factory;
H.3  import hiddenimpl.annos.impl.Impl1;
H.4  import hiddenimpl.annos.impl.pkg.Impl2;
H.5  public class Client {
H.6    public void access() {
H.7      Factory.api1().method();  // OK
H.8      Factory.api2().method();  // OK
H.9      new Impl1().method();     // Error!
H.10     new Impl2().method();     // Error!
H.11   }
H.12 }
```

Listaus 5.7: Esimerkki pakkausnimen mukaan hahmonsovitusperusteisesta toteutuksen piilottamisesta.

```
A.1  package hiddenimpl.pkgs.api;
A.2  public interface Api1 {
A.3    void method();
A.4  }
A.5
B.1  package hiddenimpl.pkgs.api.pkg;
B.2  public interface Api2 {
B.3    void method();
B.4  }
B.5
C.1  package hiddenimpl.pkgs.api;
C.2  public class Factory {
C.3    public static Api1 api1() { return new Impl1(); }
C.4    public static Api2 api2() { return new Impl2(); }
C.5  }
C.6
D.1  package hiddenimpl.pkgs.impl;
D.2  @AccessFromMatching(regex = "hiddenimpl\\.pkgs\\.(api|impl)\\..*")
D.3  public class Impl1 implements Api1 {
D.4    @Override
D.5    public void method() { new Impl2(); }
D.6  }
D.7
E.1  package hiddenimpl.pkgs.impl.pkg;
E.2  @AccessFromMatching(regex = "hiddenimpl\\.pkgs\\.(api|impl)\\..*")
E.3  public class Impl2 implements Api2 {
E.4    @Override
E.5    public void method() { new Impl1(); }
E.6  }
E.7
F.1  package hiddenimpl.pkgs.client;
F.2  import hiddenimpl.pkgs.api.Factory;
F.3  import hiddenimpl.pkgs.impl.Impl1;
F.4  import hiddenimpl.pkgs.impl.pkg.Impl2;
F.5  public class Client {
F.6    public void access() {
F.7      Factory.api1().method();  // OK
F.8      Factory.api2().method();  // OK
F.9      new Impl1().method();     // Error!
F.10     new Impl2().method();     // Error!
F.11   }
F.12 }
```

Listaus 5.8: Listauksen 5.5 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
hiddenimpl/types/client/Client.java:9: error: [access.accessfromclass.illegal] illegal
    access from disallowed type.
    new Impl1().method();
        ^
  found    : hiddenimpl.types.client.Client
  required : [hiddenimpl.types.api.Api]
hiddenimpl/types/client/Client.java:10: error: [access.accessfromclass.illegal] illegal
    access from disallowed type.
    new Impl2().method();
        ^
  found    : hiddenimpl.types.client.Client
  required : [hiddenimpl.types.api.Api]
2 errors
```

Listaus 5.9: Listauksen 5.6 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
hiddenimpl/annos/client/Client.java:9: error: [access.accessfromannotated.illegal] illegal
     access from code not annotated with one of required annotations.
    new Impl1().method();
        ^
  found    : []
  required : [hiddenimpl.annos.api.Api, hiddenimpl.annos.impl.Impl]
hiddenimpl/annos/client/Client.java:10: error: [access.accessfromannotated.illegal]
    illegal access from code not annotated with one of required annotations.
    new Impl2().method();
        ^
  found    : []
  required : [hiddenimpl.annos.api.Api, hiddenimpl.annos.impl.Impl]
2 errors
```

Listaus 5.10: Listauksen 5.7 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
hiddenimpl/pkgs/client/Client.java:9: error: [access.accessfrommatching.illegal] illegal
    access from non-matching context.
    new Impl1().method();
        ^
  found    : hiddenimpl.pkgs.client.Client::access()
  required : hiddenimpl\.pkgs\.(api|impl)\..*
hiddenimpl/pkgs/client/Client.java:10: error: [access.accessfrommatching.illegal] illegal
    access from non-matching context.
    new Impl2().method();
        ^
  found    : hiddenimpl.pkgs.client.Client::access()
  required : hiddenimpl\.pkgs\.(api|impl)\..*
2 errors
```

Kapseloinnin raottaminen testeille

Luvussa 5.1 kerrottiin `@VisibleForTesting`-annotaatiosta, jonka tarkoituksena on kertoa, että ohjelmaelementin saatavuutta on laajennettu ainoastaan, jotta siihen olisi pääsy testikoodista. Annotaatio itsessään ei rajoita pääsyä. Sen sijaan luvussa 5.3 esitellyistä pääsymääreistä `@AccessFromTest` nimenomaisesti rajoittaa elementin saatavuuden ainoastaan testiluokkiin. Määreen toiminta perustuu valikoivaan saatavuuteen, missä valikoivuuden perusteena on, onko kohteena oleva koodi testi- vai muuta koodia.

Listauksessa 5.11 on esimerkki saatavuuden rajoittamisesta testiluokkiin. Siinä luokan `Restricted` pakkauksen sisäisesti saatavissa oleva kenttä `state` on annotoitu `@VisibleForTesting`-annotaatiolla. Sille on myös asetettu pääsymääre `@AccessFromTest`. Luokat `TestClass` ja `ProductionClass` luovat `Restricted`-luokan ilmentymän ja asettavat sen `state`-kentän arvon. Ensimmäinen luokka on testiluokka, ja sen metodi on annotoitu `@Test`-annotaatiolla. Jälkimmäinen luokka on tuotantokoodia, jonka käytettäväksi `state`-kenttä ei ole tarkoitettu. Listauksessa 5.12 on käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä: `state`-kentän asettaminen tuotantokoodissa johtaa käännösvirheeseen.

Listaus 5.11: Esimerkki testisaatavuudesta.

```
A.1 public class Restricted {
A.2   @VisibleForTesting
A.3   @AccessFromTest
A.4   String state = "";
A.5 }
A.6
B.1 public class TestClass {
B.2   @Test
B.3   public void testRestricted() {
B.4     new Restricted().state = "testing";
B.5   }
B.6 }
B.7
C.1 public class ProductionClass {
C.2   public void accessRestricted() {
C.3     new Restricted().state = "production";
C.4   }
C.5 }
```

Listaus 5.12: Listauksen 5.11 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
testaccess/ProductionClass.java:3: error: [access.accessfromtest.illegal] illegal access
    from non-test class.
    new Restricted().state = "production";
                    ^
1 error
```

Esimerkissä `@AccessFromTest`-määrettä käytettiin yksikäsitteisesti `@VisibleForTesting`-annotaation rinnalla. Pääsynvalvontakehyksen aliasointimekanismin avulla on

`@VisibleForTesting`-annotaatiosta mahdollista tehdä alias `@AccessFromTest`-määreelle. Sen seurauksena `@VisibleForTesting`-annotaation semantiikan toteutuminen tarkastetaan automaattisesti käännösaikaisesti.

Oliotason yksityisyys

Javan `private`-määre rajoittaa elementin saatavuuden päätason luokan sisäiseksi, eikä oliotason yksityisyyttä ole mahdollista saavuttaa. Kytkettävän pääsynvalvontajärjestelmän `@SelfAccess`-määre määrittelee olioyksityisyyden Javaan. Kuten luvussa 5.1 todettiin, olioyksityisyys on varsin vahva tiedon piilottamisen taso, mutta esimerkiksi olioiden vertailuissa ongelmallinen.

Luvussa 5.2 esitelty kytkettävä pääsynvalvontajärjestelmä mahdollistaa saatavuuden laajentamisen luvun 5.3 pääsymääreillä sen verran, että olioiden vertailu on mahdollista, mutta muutoin olioyksityisyys säilyy. Listauksessa 5.13 on tästä esimerkki, joka esittelee `Comparable`-rajapinnan toteuttavan luokan `SelfAccessExample`. Luokalla on kolme metodia `equals`, `compareTo` ja `valueOf` sekä yksityinen kenttä `value`. Kentästä on tehty `@SelfAccess`-määreellä olioyksityinen, mutta saatavuutta on laajennettu `@AccessFromMatching`-määreellä sallien pääsy kyseisen luokan `equals`- ja `compareTo`-vertailumetodeista. Sen sijaan `valueOf`-metodia koskee `value`-kentän olioyksityisyysrajoitus. Pääsynvalvontakehyksen prototyypin kanssa tehdyn käännöksen tulosteesta listauksessa 5.14 on nähtävissä, että `value`-kentän aksessointi `valueOf`-metodissa johtaa käännösvirheeseen.

Listaus 5.13: Esimerkki käytännöllisestä olioyksityisestä saatavuudesta.

```
1  package selfaccess;
2  public class SelfAccessExample implements Comparable<SelfAccessExample> {
3    @SelfAccess
4    @AccessFromMatching(
5        regex = "selfaccess\\.SelfAccessExample::(equals|compareTo)\\([^,]*\\)")
6    private String value;
7
8    @Override
9    public boolean equals(final Object obj) {
10     return this.value.equals(((SelfAccessExample) obj).value); // OK
11   }
12
13   @Override
14   public int compareTo(final SelfAccessExample other) {
15     return this.value.compareTo(other.value); // OK
16   }
17
18   public String valueOf(final SelfAccessExample instance) {
19     return instance.value; // Error!
20   }
21 }
```

Listaus 5.14: Listauksen 5.13 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
selfaccess/SelfAccessExample.java:20: error: [access.selfaccess.illegal] illegal access to
     INNER-scoped object-private member.
    return instance.value;
                   ^
  target : instance.value
1 error
```

Usean pääsymääreen toistaminen jokaisen rajoitettavan elementin kohdalla on käytännössä kömpelöä, joten menetelmän yksinkertaistamiseksi on syytä määritellä esimerkin pääsymääreille alias, nimeltään esimerkiksi `@ComparableSelfAccess`:

```
@ComparableSelfAccess(SelfAccessScope scope):
   @SelfAccess(scope)
   @AccessFromMatching(glob="${TypeBinaryName}::{equals,compareTo}(?**)")
```

Määritelty alias edustaa pakollisena `@SelfAccess`- ja `@AccessFromMatching`-määreitä. Aliaksen *scope*-parametri välitetään `@SelfAccess`-määreelle. `AccessFromMatching`-määreen hahmossa on käytetty *TypeBinaryName*-parametria, joka korvataan rajoitettavan elementin luokan nimellä. Näin ollen määritelty alias sallii pääsyn elementtiin saman olion sisällä sekä saman luokan yksiparametrisista `equals`- ja `compareTo`-metodeista.

Pääsy aliluokista ilman pääsyä saman pakkauksen luokista

Javan `protected`-määre todettiin ongelmalliseksi luvussa 5.1, koska se sallii aliluokkien lisäksi pääsyn esittelevän luokan pakkauksen sisällä. Mahdollisuus rajoittaa saatavuus ainoastaan aliluokkiin todettiin hyödylliseksi.

Kytkettävään pääsynvalvontajärjestelmään määritellyn `@AccessFromInherited`-pääsymääreen asettama pääsyrajoite on kuten Javan `protected`-pääsymääreen sillä erolla, että `@AccessFromInherited` ei salli pääsyä saman pakkauksen sisällä. Se on samankaltainen `@AccessFromClass({`C`})`-määreen kanssa, kun C on luokka, jolle kyseinen määre on asetettu. Merkittävin ero on viimeksi mainittujen välillä on, että `@AccessFromInherited`-määre asettaa lisärajoitteita Javan `protected`-määreen sääntöjen mukaisesti. Lisäksi parametrittomuus yksinkertaistaa perintäperusteisen saatavuuden hyödyntämistä esimerkiksi määrealiaksissa, sekä siten, ettei ohjelmoijan ole mahdollista antaa parametrina väärää luokkaa.

Koska `@AccessFromInherited`-määre ei salli pääsyä saman pakkauksen sisällä, sen asettaman pääsyrajoitteen avulla voidaan kiertää luvussa 5.1 mainittu ongelma koskien metodin saatavuuden riippuvuutta metodin korvaamisesta aliluokassa. Listaus 5.15 havainnollistaa määreen toimintaa. Se sisältää listausta 5.1 vastaavat luokat, mutta kaikki suojatut metodit on annotoitu `@AccessFromInherited`-määreellä. Lisäksi listauksessa 5.1 virheen

aiheuttanut rivi on kommentoitu pois listauksessa 5.15, koska se ei ole enää oleellinen, mutta aiheuttaa edelleen virheen. Listauksesta 5.16 on nähtävissä prototyyppitarkastajalla suoritettu käännöksen tulos: rajoitettuun metodiin ei ole enää pääsyä saman pakkauksen sisällä, vaan kaikki kolme metodikutsua aiheuttavat käännösvirheen.

Listaus 5.15: Listauksen 5.1 listaus täydennettynä `@AccessFromInherited`-pääsymääreillä.

```
A.1   package protectedwithoutpkgaccess.pkg1;
A.2   public class Superclass {
A.3     @AccessFromInherited
A.4     protected void protectedMethod() {}
A.5   }
A.6
B.1   package protectedwithoutpkgaccess.pkg2;
B.2   import protectedwithoutpkgaccess.pkg1.Superclass;
B.3   public class Subclass extends Superclass {}
B.4
C.1   package protectedwithoutpkgaccess.pkg2;
C.2   public class OverridingSubclass extends Superclass {
C.3     @Override
C.4     @AccessFromInherited
C.5     protected void protectedMethod() {}
C.6   }
C.7
D.1   package protectedwithoutpkgaccess.pkg1;
D.2   import protectedwithoutpkgaccess.pkg2.*;
D.3   public class Accessor {
D.4     public void access() {
D.5       new Superclass().protectedMethod();
D.6       new Subclass().protectedMethod();
D.7       // new OverridingSubclass().protectedMethod();
D.8       ((Superclass) new OverridingSubclass()).protectedMethod();
D.9     }
D.10  }
```

`@AccessFromInherited` ei poista kyseistä `protected`-määreen ongelmaa, mutta rajoittaa sen vaikutusta, kun luokan jäsenen voi yksikäsitteisesti määritellä saatavaksi esittelevän luokkansa lisäksi ainoastaan sen aliluokissa, muttei samassa pakkauksessa. Siten määreellä rajoitetun elementin saatavuus ei riipu siitä, korvaako toisessa pakkauksessa sijaitseva luokka rajoitetun metodin vai ei.

Javan pääsyrajoitteiden mallintaminen

Luvussa 5.2 tuli esille, että pääsymääreiden tulee olla vastuualueeltaan tarkkaan rajautuneita, mikä mahdollistaa eri määreiden yhteentoimivuuden ja pääsynvalvontajärjestelmän laajentamisen. Sen seurauksena esimerkiksi asetettaessa luokan A jäsenelle `x` pääsymääre `@AccessFromClass(B)`, x on saatavissa luokasta B, muttei enää luokasta A. Siksi luokan A pääsy x:ään on sallittava toisella pääsymääreellä.

Koska lähtökohtana on Javan sisäänrakennettu pääsynvalvontajärjestelmä, jota kytkettävä pääsynvalvontajärjestelmä täydentää, on tärkeää, että kytkettävä pääsynvalvontajärjestel-

Listaus 5.16: Listauksen 5.15 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
protectedwithoutpkgaccess/pkg1/Accessor.java:5: error: [access.accessfrominherited.illegal
    ] access is permitted only from inheriting class.
    new Superclass().protectedMethod();
                    ^
  accessing class      : protectedwithoutpkgaccess.pkg1.Accessor
  required subclass of : protectedwithoutpkgaccess.pkg1.Superclass
protectedwithoutpkgaccess/pkg1/Accessor.java:6: error: [access.accessfrominherited.illegal
    ] access is permitted only from inheriting class.
    new Subclass().protectedMethod();
                  ^
  accessing class      : protectedwithoutpkgaccess.pkg1.Accessor
  required subclass of : protectedwithoutpkgaccess.pkg1.Superclass
protectedwithoutpkgaccess/pkg1/Accessor.java:8: error: [access.accessfrominherited.illegal
    ] access is permitted only from inheriting class.
    ((Superclass) new OverridingSubclass()).protectedMethod();
                                           ^
  accessing class      : protectedwithoutpkgaccess.pkg1.Accessor
  required subclass of : protectedwithoutpkgaccess.pkg1.Superclass
3 errors
```

mä kykenee mallintamaan myös Javan sisäänrakennetun pääsynvalvontajärjestelmän. Näin edellä esitetyssä esimerkissä pystytään sallimaan luokan A jäsenelle x myös esimerkiksi luokan sisäinen saatavuus.

Javan pääsynvalvontajärjestelmä pystytään mallintamaan luvussa 5.3 määritellyillä kytkettävillä pääsymääreillä `@AccessWithin` ja `@AccessFromInherited`. Näistä `@AccessWithin` sallii pääsyn rajoitettuun elementtiin ohjelman rakenteen mukaan ja `@AccessFromInherited` luokkien perintähierarkian mukaan. Taulukossa 5.2 esitetään edellä mainittujen kytkettävien pääsymääreiden vastaavuudet Javan sisäänrakennettuihin pääsymääreisiin. Sulkuihin merkitty Java-määre package-private tarkoittaa Javan oletussaatavuutta, jolle ei ole olemassa omaa pääsymäärettä.

Taulukko 5.2: Javan sisäänrakennetut pääsymääreet ja niitä vastaavat kytkettävät pääsymääreet.

Java-määre	**Kytkettävät määreet**
`public`	`@AccessWithin(ALL)`
`protected`	`@AccessFromInherited @AccessWithin(PACKAGE)`
(*package-private*)	`@AccessWithin(PACKAGE)`
`private`	`@AccessWithin(TOP_LEVEL)`

Taulukon perusteella Javan sisäänrakennettuja pääsymääreitä vastaavat aliakset on mahdollista määritellä seuraavasti:

@Public:
`@AccessWithin(ALL)`

@Protected:
`@AccessFromInherited`
`@AccessWithin(PACKAGE)`

@PackagePrivate:
`@AccessWithin(PACKAGE)`

@Private:
`@AccessWithin(TOP_LEVEL)`

Liitteessä 2 on esimerkkilistaukset Javan sisäänrakennettujen ja niitä vastaavien kytkettävien pääsymääreiden käytöstä sekä listausten käännöksessä havaituista virheistä. Niistä on havaittavissa, että kytkettävillä pääsymääreillä on esimerkin tilanteissa täysin mallinnettavissa Javan omien pääsymääreiden toiminta.

5.6 Kokemuksia ja havaintoja

Suunnittelu-, toteutus- ja arviointiprosessin aikana kytkettävän tyyppijärjestelmän havaittiin soveltuvan hyvin sen kohteena olleeseen ongelmakenttään, mutta samalla nousi esiin joitakin siinä piileviä ongelmiakin. Tässä luvussa esitellään havaittuja ongelmakohtia, tarkastellaan muita aiheeseen liittyviä ratkaisuja sekä esitetään jatkokehitys- ja tutkimuskohteita kytkettävän pääsynvalvontajärjestelmän ja pääsynvalvontakehyksen kehittämiseksi.

Hahmonsovitukseen perustuvan valikoivan saatavuuden piirteitä

Hahmonsovitukseen perustuva `@AccessFromMatching`-pääsymääre on erittäin ilmaisuvoimainen — sillä on koko säännöllisten lausekkeiden ilmaisuvoima käytössään. Säännölliset lausekkeet [MY60] ovat merkkijonojen tunnistamisessa yleisesti käytetty tehokas menetelmä. `@AccessFromMatching`-määreen yhteydessä tunnistamisen kohteena ovat aksessoijaviitteet.

Toisin kuin luokkiin, Javassa pakkauksiin ja metodeihin ei pysty luomaan itsenäisiä viitteitä. Vaikka Javan versio 8 esitteli uutena piirteenä metodiviitteet, ne eivät ole arvoja, joita voisi sijoittaa muuttujiin [GJS$^+$15, s. 536–539]. Viitattaessa pakkauksiin ja metodeihin on pääsymääreissä siten käytettävä merkkijonoesitystä. Tästä johtuen hahmonsovitukseen perustuva `@AccessFromMatching`-määre soveltuu hyvin juuri pakkausten ja metodien perusteella tapahtuvan valikoivan saatavuuden toteuttamiseksi.

Aksessoijaviitteeseen sisältyy aksessoivan tyypin koko binäärimuotoinen nimi. Sen seurauksena `@AccessFromMatching`-määrettä on mahdollista käyttää tyyppiperusteisen valikoivan saatavuuden toteuttamiseksi. Koska merkkijonoesityksessä ei pysty huomioimaan luokkien perintähierarkiaa eikä toteutettuja rajapintoja, määreen soveltuvuus mainittuun tarkoitukseen on kuitenkin erittäin rajallinen. Määre kuitenkin mahdollistaa pääsyn sallimisen tietyn käytännön mukaan nimetyistä luokista, esimerkiksi kaikista `Test`-päätteisistä luokista. Nimeämiskäytäntöjä ei tosin tämän kaltaisessa yhteydessä pidetä suositeltavana [Blo08, s. 169–175]. Sen sijaan tulisi suosia annotointiperusteista valikoivaa saatavuutta.

Hahmonsovituksen ilmaisuvoimasta kertoo myös se, että varomaton ohjelmoija pystyy sen avulla luomaan riippuvuuksia esimerkiksi vieraan luokan sisäisiin toteutusyksityiskohtiin. Mikään ei estä luomasta hahmoa, joka täsmää tarkoituksellisesti jonkin toisen luokan yksityiseen jäseneen tai paikalliseen tai anonyymiin luokkaan, josta `@AccessFromMatching`-määreellä rajoitettavan elementin ei pitäisi olla lainkaan tietoinen. Näin määre luo toiseen suuntaan ongelman, jota se toiseen suuntaan ratkoo: aksessoijasta rajoitettuun elementtiin määre vahvistaa tiedon piilottamista murentaen sen rajoitetusta elementistä aksessoijaan. Tätä voidaan pitää pääsymääreen merkittävänä heikkoutena, jonka ratkaisemiseksi tarvitaan jatkotutkimuksia.

Hahmonsovitukseen perustuvalla saatavuudella on muitakin käytännön ongelmia. Hahmoista saattaa helposti tulla pitkiä, monimutkaisia ja vaikeaselkoisia lukea ja ymmärtää. Tämä koskee erityisestä säännöllisiä lausekkeita, mutta vaikka möykkyhahmojen tarkoitus on yksinkertaistaa hahmon syntaksia, nekin voivat muodostua monimutkaisiksi. Monimutkaisuuden seurauksena virheiden mahdollisuus kasvaa samalla kuin testaaminen ja ylläpidettävyys vaikeutuvat.

Työkalujen, kuten integroitujen ohjelmointiympäristöjen (integrated development environment, IDE), tuki hahmojen ylläpitoon jää usein olemattomaksi. Jotkut ohjelmointiympäristöt saattavat esimerkiksi jonkin nimen muuttamisen yhteydessä tunnistaa yksinkertaisesta hahmosta kyseisen nimen ja muuttaa sen muiden automaattisten koodimuutosten yhteydessä [Int16], mutta se ei ole itsestään selvää. Hahmojen monimutkaistuessa niihin tarvittavien muutosten huomaaminen ja toteuttaminen jää aina todennäköisemmin ohjelmoijan vastuulle. Automaation puute kasvattaa riskiä, että joitakin tarvittavia muutoksia jää huomaamatta, ja ohjelmaan jää virheitä.

Jos samaa hahmoa on käytetty useaan kertaan, on ohjelmoijan tehtävä tarvittavat muutokset jokaiseen hahmon esiintymään erikseen. Koska Javan annotaatioissa on mahdollista käyttää merkkijonoliteraalin sijaan viittausta merkkijonovakioon [GJS+15, s. 312], on muutosten tekemistä edellisessä tapauksessa mahdollista helpottaa käyttämällä merkkijonovakiota, jolloin riittää tehdä tarvittavat muutokset ainoastaan kyseiseen vakioon.

Mainittujen ongelmien valossa vaikuttaa siltä, että ensisijaisesti tulisi suunnitella erilliset hahmonsovitusperusteista pääsynvalvontaa välttävät pääsymääreet pakkaus- ja metodiperusteisen valikoivan saatavuuden toteuttamiseksi ja luopua kokonaan

`@AccessFromMatching`-määreestä. Ennen kuin korvaavat pääsymääreet ovat olemassa, on syytä suosia muita pääsynvalvontatapoja hahmonsovitukseen perustuvan valikoivan pääsynvalvonnan sijaan. Silloin, kun hahmonsovitukseen perustuva valikoiva pääsynvalvonta on tarpeellista, olisi sen vaikutusalue syytä rajata mahdollisimman pieneksi, jotta mahdolliset ongelmatkin rajautuisivat mahdollisimman pienelle alueelle.

Metodiperusteisen valikoivan saatavuuden ongelma

Saatavuus määrätyistä metodeista eli metodiperusteinen valikoiva saatavuus on osoittautunut ongelmalliseksi. Luvussa 5.3 esitetyistä kytkettävistä pääsymääreistä sen mahdollistavia ovat `@AccessFromAnnotated`, `@AccessFromMatching` sekä tarkastajan toteutustavasta riippuen voisi olla myös `@AccessFromTest`. Vaikka pääsyn salliminen kohteeseen tietyt kriteerit täyttävästä luokan ulkopuolisesta metodista olisi semanttisesti oikein, aksessoinnin tarve saattaa olla toisessa metodissa, josta pääsy ei ole sallittu. Tällainen tilanne syntyy esimerkiksi silloin, kun metodi on jaettu kutsuiksi yhteen tai useampaan yksityiseen alimetodiin, joista yhdessä tai useammassa on tarve aksessoida rajoitettua elementtiä. Tiedon piilottamisen kannalta on oikein, että rajoitettu elementti ei pysty sallimaan pääsyä toisen luokan yksityisestä metodista, koska elementti ei voi olla tietoinen yksityisen metodin olemassaolosta.

Listaus 5.17 havainnollistaa ongelmaa. Siinä esitellään luokat `Restricted` ja `Accessor`. Pääsy `Restricted`-luokan `action`-metodiin on rajoitettu luvussa 5.3 esitellyllä pääsymääreellä `@AccessFromMatching` minkä tahansa luokan parametrittomiin `accessPermitted`- ja `accessDenied`-nimisiin metodeihin. Luokka `Accessor` sisältää molemmat kyseisistä metodeista, ja molempien kutsun seurauksena kutsutaan `action`-metodia. Mutta koska `accessDenied`-metodi ei kutsu suoraan `action`-metodia vaan `internalUnprivileged`-metodia, joka kutsuu `action`-metodia, käännös epäonnistuu, koska `internalUnprivileged`-metodilla ei ole pääsyä `action`-metodiin. Tiedon piilottaminen ei kuitenkaan vaarantuisi pääsyn sallimisesta, koska `internalUnprivileged`-metodin voidaan katsoa olevan luokan sisäinen piilotettava toteutusyksityiskohta, eikä mikään epäkelvollinen metodi kutsu `internalUnprivileged`-metodia.

Ongelman yksinkertaisella kiertämisellä on merkittäviä haittavaikutuksia. Jos ongelman yrittää kiertää sisällyttämällä listauksen `internalUnprivileged`-metodin koodin `accessDenied`-metodiin, seurauksena luokan sisäinen abstrahointi hajoaa heikentäen luokan sisäistä rakennetta, luettavuutta ja ylläpidettävyyttä. Lisäksi, jos `internalUnprivileged`-metodia kutsuttaisiin useasta kelvollisesta metodista, tulee koodi sisällyttää eli monistaa jokaiseen kutsuvista metodeista, mikä merkittävästi heikentää luokan ylläpidettävyyttä. Toinen tapa kiertää ongelma olisi tehdä lisätä `internalUnprivileged`-metodi `Restricted`-luokan sallittujen metodien listaan. Tämä kuitenkin rikkoisi `Accessor`-luokan abstrahoinnin ja loisi riippuvuuden kyseisen luokan toteutusyksityiskohtaan.

Listaus 5.17: Esimerkki metodiperusteisen valikoivan saatavuuden ongelmasta: `action`-metodin kutsu luokan sisäisestä `internalUnprivileged`-metodista on laiton, vaikka se ei vaaranna tiedon piilotusta.

```
A.1 public class Restricted {
A.2   @AccessFromMatching(regex = ".*::access(Permitted|Denied)\\(\\)")
A.3   public static void action() { }
A.4 }
A.5
B.1 public class Accessor {
B.2   public void accessPermitted() { Restricted.action(); }
B.3   public void accessDenied() { internalUnprivileged(); }
B.4   private void internalUnprivileged() { Restricted.action(); }
B.5 }
```

Ratkaisuna metodiperusteisen valikoivan saatavuuden ongelmaan ehdotan kytkettävään pääsynvalvontakehykseen lisättäväksi mahdollisuuden löyhentää metodiperusteisten valikoivan saatavuuden määreiden vaikutusta. Staattista analyysiä hyödyntäen on mahdollista sallia pääsy rajoitettuun elementtiin myös sellaisille metodeille, joilla ei pääsymääreen perusteella ole pääsyä rajoitettuun elementtiin, mutta jotka täyttävät seuraavat kaksi ehtoa. Metodia saa kutsua suoraan tai välillisesti ainoastaan sellaisista metodeista, joille pääsy on sallittu. Välillisen kutsun kaikkien kutsuketjun metodien on täytettävä sama ehto. Lisäksi sallittavan metodin saatavuuden tulee olla sellainen, että käännösaikaisesti pystytään varmistumaan, ettei sallittavaa metodia ole mahdollista kutsua kääntämisen jälkeen sellaisesta metodista, jolla ei ole pääsyä rajoitettuun elementtiin. Esimerkiksi yksityiset metodit täyttävät tämän ehdon.

Ratkaisumalli ei johda ongelman yllä mainittujen kiertotapojen kaltaiseen koodin laadun heikkenemiseen, koska ratkaisu ei vaadi ohjelman rakenteen muuttamista. Ensimmäinen, metodin kutsumista koskeva ehto ei muodostu rajoitteeksi, koska sellaisella metodilla, jolla ei ole pääsyä rajoitettuun elementtiin, ei pitäisi olla tarvettakaan aksessoida rajoitettua elementtiä suoraan tai välillisesti toisen metodin kautta. Toinen, metodin saatavuutta koskeva ehto ei myöskään muodostu rajoitteeksi, koska rajoitetun elementin aksessoinnin voidaan ajatella semanttisesti tapahtuvan määreen sallimassa metodissa. Metodin toiminnallisuuden hajauttaminen alimetodeihin on luokan sisäinen toteutusyksityiskohta, jonka ei tiedon piilottamisen periaatteen mukaan tule näkyä luokan ulkopuolelle muutoinkaan. Ratkaisumallin toteuttaminen kytkettäväksi minkä tahansa pääsymääreen tarkastajan yhteyteen vaatii vielä yksityiskohtaisempaa teknistä suunnittelua.

Muita aiheeseen liittyviä ratkaisuja

Asiakkaan, rajapinnan ja toteutuksen välinen saatavuuden ongelmaa on mahdollista ratkoa pääsynvalvonnan lisäksi moduulijärjestelmien avulla. Tässä tapauksessa moduuleilla tarkoitetaan Javan pakkauksia laajempaa ohjelmakomponenttia. OSGi [OSG14] on dynaaminen modularisointijärjestelmä, jonka ensimmäinen määrittely julkaistiin vuonna

2000 [OSGa]. Siitä on useita avoimen lähdekoodin toteutuksia [OSGb], esimerkiksi Eclipse-kehitysympäristön käyttämä Equinox yhtenä tunnetuimmista. OSGi kokoaa joukon luokkia ja pakkauksia kimpuksi (bundle) [OSG14]. Kimppu määrittelee riippuvuutensa toisiin kimppuihin sekä kimpun ulkopuolelle paljastettavan rajanpinnan. OSGi sisältää myös kimppujen versioiden hallinnan ja ajon aikaisen elinkaaren hallinnan. Kimppuja on mahdollista ottaa käyttöön ja poistaa käytöstä ajon aikaisesti, ja samanaikaisesti voi olla käytössä saman kimpun kaksi eri versiota.

Javan versioon 9 on suunnitteilla tuki moduuleille [Pro16]. Sen toteutuessa Javaan tulisi moduulien esittelykonstruktiot, joilla määritellään moduulin nimi, sen riippuvuudet toisiin moduuleihin sekä moduulin ulkopuolelle paljastettavat pakkaukset [Rei16]. Moduulijärjestelmä ei sisällä versiointia, vaan siitä huolehtiminen jää käännöstyökalujen ja sovelluskehysten vastuulle. Moduulien asettamien rajoitteiden toteutumista valvotaan sekä käännösaikaisesti kääntäjän toimesta että suoritusaikaisesti Java-virtuaalikoneen toimesta.

OSGI ja Java 9:n moduulit lähestyvät modularisointia eri lähtökohdista. Niiden ei silti tarvitse olla keskenään vaihtoehtoisia modularisointimekanismeja, vaan ne voivat jopa täydentää toisiaan [Bar15]. Toistaiseksi niiden yhteiselon saavuttamiseksi on vielä monia kysymyksiä ratkottavana.

Modularisointi moduulijärjestelmien tai kytkettävän pääsynvalvonnan avulla ovat eri kokoluokan ratkaisuja. Tyypillisesti yksi moduuli vastaa kokoluokaltaan yhtä käännösprojektia. Kytkettävä pääsynvalvonta pystyy sen sijaan eristämään ohjelman osia toisistaan yhden käännösprojektin sisällä. Molemmilla tekniikoilla on siten oma käyttötarkoituksensa ja niiden voidaan katsoa täydentävän toisiaan.

Testausta varten avatun kapseloinnin valvontaan on olemassa joitakin koodin laadunvalvontatyökaluihin tehtyjä lisäosia. Staattisen analyysin avulla ohjelmointivirheitä etsivään FindBugs-työkaluun [HP04] on kehitetty eri tahojen toimesta `@VisibleForTesting`-annotaation väärinkäytöksiä etsiviä tarkastajia[8]. Myös SonarQube-laadunhallinta-alustaan[9] on olemassa ulkopuolisen tahon kehittämä `@VisibleForTesting`-annotaation väärinkäytöksiä etsivä lisäosa[10]. Sen lisäksi, että nämä työkalujen laajennokset tarkastavat, että `@VisibleForTesting`-annotoitua elementtiä ei aksessoida testikoodin ulkopuolelta, osa niistä tarkastaa esimerkiksi, että annotoidun elementin saatavuus on pakkauksen sisäinen.

Viimeksi mainittu ehto on tutkielmassa esitellyn kytkettävän tyyppijärjestelmän `@AccessFromTest`-määreen rajoitetta tiukempi: jälkimmäinen vaatii ainoastaan, ettei

[8] FindBugs-työkaluun kehitettyjä `@VisibleForTesting`-annotaation käytön tarkastajia:

- `https://writeoncereadmany.wordpress.com/2016/04/08/how-to-find-bugs-part-3-visiblefortesting/`
- `https://github.com/arxes-tolina/findbugs-plugin`
- `https://github.com/Monits/findbugs-plugin`

[9] SonarQube: `http://www.sonarqube.org/`

[10] `@VisibleForTesting`-annotaation käytön tarkastaja SonarQube-alustalle: `https://github.com/arxes-tolina/sonar-plugins/`

rajoitettava elementti ole yksityinen. Periaatteessa `@AccessFromTest`-määreeseen olisi ollut mahdollista määritellä esittelyehdoksi, että rajoitettavan elementin tulee olla pakkauksen sisäinen. Mutta pääsymääreen ja `@VisibleForTesting`-annotaatiotyypin välillä on pieni semanttinen ero: `@VisibleForTesting` tarkoittaa, että elementin tulisi olla yksityinen, mutta sen saatavuutta on laajennettu, jotta sitä on mahdollista aksessoida testeistä. `@AccessFromTest`-määreen semantiikka on, että rajoitettuun elementtiin tulee olla pääsy testikoodista. Tiukempien ehtojen saavuttamiseksi olisi kuitenkin mahdollistaa kytkettävää pääsynvalvontakehystä mahdollistamaan esittelyehdot pääsymääreiden aliaksille. Jos tällöin `@VisibleForTesting` määriteltäisiin `@AccessFromTest`-määreen aliakseksi, pystyisi aliakselle määrittelemään esittelyehdoksi aliaksen semantiikan mukaisesti vaatimuksen pakkauksen sisäisestä saatavuudesta.

Pääsynvalvontajärjestelmän jatkokehityskohteita

Kytkettävän pääsynvalvontajärjestelmän suunnittelu- ja kehitys- ja arviointiprosessin aikana suunnitelmista löytyi monia asioita, jotka kaipaisivat lisähuomiota, suunnittelua ja kehittämistä. Tässä luvussa esitellään niistä keskeisimmät.

Kytkettävän pääsynvalvontajärjestelmän suunnitelmassa ei ollut mukana lainkaan elementin ja sen edustaman tyypin välisiä saatavuusehtoja, jollaiset on määritelty esimerkiksi luvussa 3.2 esitellyissä C#:ssa ja Ceylonissa. Ehtojen tarkoituksena on varmistaa, että kaikki tarvittavat ohjelmaelementit ovat todellisuudessa saatavissa niitä aksessoivassa ympäristössä ja estää virheitä ohjelmassa. Yleistäen ehdot ovat:

1. Kentän tyypin tulee olla saatavissa siellä missä kenttäkin on saatavissa. Toisin sanoen kentän saatavuus ei saa olla sen tyypin saatavuutta laajempi.
2. Metodin paluuarvon tyypin tulee olla saatavissa siellä missä metodikin on saatavissa. Toisin sanoen metodin saatavuus ei saa olla sen paluuarvon tyypin saatavuutta laajempi.
3. Luokan toteuttaman rajapinnan tulee olla saatavissa siellä missä toteuttava luokkakin on saatavissa. Toisin sanoen luokan saatavuus ei saa olla laajempi kuin sen toteuttavan rajanpinnan saatavuus.
4. Luokan yliluokan tulee olla saatavissa siellä missä luokkakin on saatavissa. Toisin sanoen luokan saatavuus ei saa olla laajempi kuin sen yliluokan saatavuus.

Ehdotan edellä esiteltyjen ehtojen toteuttamista kytkettävään pääsynvalvontajärjestelmään siten, että kukin ehdoista on valinnaisesti toisistaan riippumatta otettavissa käyttöön. Kyseisten ehtojen valvonta tulisi toteuttaa erikseen koskemaan Javan sisäänrakennettua pääsynvalvontaa sekä kytkettävää pääsynvalvontaa.

Edellisten lisäksi kytkettävästä pääsynvalvontajärjestelmästä puuttuu Javan ehto, että että perityn metodin korvaavan metodin saatavuuden on oltava vähintään yhtä laaja kuin perityn metodin [GJS+15, s. 249]. Myös tämän ehto olisi syytä suunnitella osaksi kytkettävää pääsynvalvontajärjestelmää.

Jo aiemmassa luvussa ehdotin ongelmallisesta `@AccessFromMatching`-määreestä luopumista. Sen tilalle tulisi suunnitella kaksi uutta pääsymäärettä pakkaus- ja metodiperusteisen pääsynvalvonnan toteuttamiseksi, jotka välttäisivät hahmontunnistukseen perustuvaa pääsynvalvontaa ja pyrkisivät sen sijaan hyödyntämään kielen omia konstruktiota mahdollisimman paljon.

Mainittakoon tässä yhteydessä kertaalleen aiemmassa luvussa mainittu ratkaisumalli metodiperusteisen pääsynvalvonnan ongelmaan, jota ehdotan toteutettavaksi: sallitun metodin lisäksi pääsy tulisi sallia metodin toteutusyksityiskohtana pidettävistä sisäisistä metodikutsuketjuista silloin, kun pystytään staattisen analyysin avulla varmistumaan, ettei kutsuketjuun päädytä koskaan kielletystä metodista.

Kytkettävän pääsynvalvontajärjestelmän lisäksi myös pääsynvalvontakehyksen suunnitelmaan mahtuu jatkokehityskohteita. Tämänhetkisessä pääsynvalvontakehyksessä valinnaisuus toteutuu mahdollisuudella kytkeä ja poistaa käytöstä yksittäin pääsymääretarkastajia. Valitut tarkastajat suorittavat tarkastukset kaikille käännettävissä luokissa tapahtuville aksessoinneille. Mutta valinnaisuus pitäisi voida käsittää myös asteittaisuutena: ohjelmoijan tulisi pystyä valitsemaan ohjelman osa esimerkiksi pakkauksittain, missä osissa käännettävän ohjelman aksessoinnti tarkastetaan. Valinta pitäisi pystyä tekemään myös käänteisesti: mihin ohjelman osaan kohdistuvat aksessoinnit tarkastetaan.

Uusien pääsynvalvontajärjestelmien luontia pääsynvalvontakehykseen olisi mahdollista hieman helpottaa. Jos pääsymäärettä halutaan käyttää deklaratiivisesti määritellyn määrealiaksen osana tai oletusmääreenä, on näitä varten esiteltävä vielä omat annotaatiotyypit. Niiden luonti on hyvin mekaaninen prosessi, joten se automatisoitavissa toteuttamalla tarkoitukseen sopiva annotaatioprosessori.

Kehityskohteista voitaneen todeta, että kehitettävää riittää. Ja se on kytkettävän pääsynvalvonnan tarkoitus: olla helposti kehitettävissä ja laajennettavissa aina tarpeen mukaan.

6 Yhteenveto

Tutkielmassa tarkasteltiin ohjelmointikielten semanttisista rajoitteista erityisesti tiedon piilottamisen mekanismeja eli pääsynvalvontaa ja tyyppijärjestelmiä. Pääsynvalvonnan osalta vertaillut staattiset olio-ohjelmointikielet tekevät jaon ohjelmaelementtien saatavuudesta yksityiseksi ja julkiseksi. Yksityisyys on kielissä yleisemmin luokka- kuin oliokohtaista. Kahden ääripään lisäksi ohjelmaelementtien saatavuutta pystyy kielestä riippuen rajoittamaan ohjelman modulaarisen rakenteen tai olioiden perintähierarkian perusteella tai aksessoivan tyypin perusteella valikoivasti. Saatavuuden taso määrätään tyypillisesti yksikäsitteisillä pääsymääreillä, mutta yhdessä vertailluista kielistä myös esittely-ympäristö vaikuttaa saatavuuden määräytymiseen.

Staattiset tyyppijärjestelmät mahdollistavat muun muassa tyyppivirheiden havaitsemisen jo käännösaikana, mutta ne ovat usein varsin monimutkaisia ja tiukasti kieleen sidottuja. Valinnaiset tyyppijärjestelmät mahdollistavat staattisen tyypityksen lisäämisen dynaamiseen kieleen tai kielen oman staattisen tyyppijärjestelmän rikastamisen. Valinnaisina ne voivat olla yksinkertaisempia ja kehittyä kieltä nopeammin. Kytkettävät tyyppijärjestelmät vievät valinnaisuuden ajatusta vielä pidemmälle: ne ovat staattisia tyyppijärjestelmiä, jotka eivät vaikuta ohjelman suoritusaikaiseen semantiikkaan. Se mahdollistaa, että kieleen on mahdollista liittää erilaisia tyyppijärjestelmiä sovelluskohtaisen tarpeen mukaan.

Javan tapa jaotella saatavuus neljään tasoon, jotka ovat julkinen, suojattu, pakkauksen sisäinen ja yksityinen, on karkea. Niillä ole mahdollista saavuttaa optimaalista saatavuutta. Esimerkiksi pakkausten välillä toisistaan riippuvaiset luokat ovat aina julkisia, yksityisiä jäseniä ei pysty paljastamaan ainoastaan aliluokille tai testiluokille, ja olioyksityisyys on Javassa tuntematon käsite. Ongelman ratkaisemiseksi esittelin idean kytkettävästä pääsynvalvonnasta, joka soveltaa kytkettävien tyyppijärjestelmien ideaa pääsynvalvontaan. Tutkielman keskeinen tavoite oli osoittaa, että kytkettävä pääsynvalvonta mahdollistaa optimaalisen saatavuuden edellä mainituissa tapauksissa, ja että se on toteutettavissa Javaan.

Suunnittelemani kytkettävä pääsynvalvontajärjestelmä pääsymääreineen osoittautui toimivaksi ratkaisuksi esitettyyn tarpeeseen. Sen avulla optimaalinen saatavuus oli mahdollista saavuttaa kaikissa edellä mainituissa tapauksissa. Lisäksi se kykenee mallintamaan Javan sisäänrakennetun pääsynvalvontajärjestelmän. Suunnitelmaan kuului myös Javaan kytkettävä pääsynvalvontakehys pääsynvalvontajärjestelmän toteutukseksi, jonka pohjalta rakennettu prototyyppi osoitti ratkaisun olevan konkreettisesti toteutettavissa.

Suunnitelluista pääsymääreistä `@AccessFromMatching`, jonka rajoitusperuste pohjautuu merkkijonohahmon tunnistukseen, vaikutti suunnitteluvaiheessa vahvalta ja ilmaisuvoimaiselta. Siihen liittyi useita eri käsitteitä, minkä vuoksi sen määrittelystä tuli muihin verrattuna kaikista laajin. Määreen tavoitteena oli mahdollistaa erityisesti pakkauksiin ja metodeihin perustuva valikoiva saatavuus. Määreen merkkijonoesitykseen pohjautuva vali-

koivuus osoittautui kuitenkin niin ongelmalliseksi, että päädyin suosittelemaan määreestä luopumista kokonaan ja uusien merkkijonoesitystä välttävien pakkaus- ja metodiperusteiseen saatavuuteen perustuvien pääsymääreiden suunnittelua tilalle. Muiden suunniteltujen pääsymääreiden kohdalla ei tullut esiin erityisiä heikkouksia tutkielmaprosessin aikana, vaan ne osoittautuivat arvioinnissa ongelmitta tarkoituksensa täyttäviksi.

Suunnitellut kytkettävät pääsymääreet hyödyntävät monipuolisesti erilaisia saatavuusperusteita, ja ne määrittelevät saatavuuden yksikäsitteisesti. Yksikään pääsymääreistä ei määrittele saatavuutta sen esittely-ympäristöstä riippuvaisesti, koska se ei ollut edellytys optimaalisen saatavuuden mahdollistamiseksi. Tämä kielten pääsynvalvontamekanismien vertailussa esiin tullut saatavuuden määräytymisperuste on käytännölliseltä vaikuttava poikkeus saatavuuden määräytymisperusteiden joukossa, minkä vuoksi sen hyötyjä ja sovellettavuutta kytkettävään pääsynvalvontaan olisi jatkossa hyödyllistä tutkia.

Kytkettävä pääsynvalvontakehys on pohja pääsynvalvontajärjestelmien toteutukselle. Sen ansiosta uusia pääsymääreitä ja niiden tarkastajia on mahdollista kehittää helposti ja ottaa käyttöön tarpeen mukaan. Kun yksittäinen pääsymääreen rajoitesemantiikka on määritelty selkeästi rajatun perusteen mukaan, on määreiden yhteiskäyttö ongelmatonta. Eri pääsymääreiden yhdistelymahdollisuuden ansiosta kytkettävät pääsynvalvontajärjestelmät ovat erittäin ilmaisuvoimaisia.

Kytkettävän pääsynvalvonnan ilmaisuvoimasta ei silti ole syytä sokeutua, vaan on muistettava, että se on suunniteltu täydentämään kielen omaa pääsynvalvontaa. Kielen oman pääsynvalvontajärjestelmän tulee edelleen olla ensisijainen saatavuuden rajoittamismekanismi, jonka tarkoitus on asettaa reunaehdot ohjelmaelementtien saatavuudelle. Kytkettävän pääsynvalvonnan tarkoitus on asetettujen reunaehtojen puitteissa tehdä ainoastaan tarkentava rajaus ohjelmaelementtien optimaalisen saatavuuden aikaansaamiseksi. Kytkettävän pääsynvalvontajärjestelmän mukaisten rajoitteiden tarkastaminen tapahtuu ainoastaan käännösaikaisesti, ja sekin on valinnaista. Näin ollen kytkettävä pääsynvalvontajärjestelmä ei pysty antamaan takuita pääsyrajoitteiden noudattamisesta ohjelman suoritusaikana eikä käännösprojektin ulkopuolelta tulevista aksessoinneista. Ainostaan kielen oma pääsynvalvonta on voimassa suoritusaikaisesti ja käännösprojekteista riippumattomasti.

Kytkettävän pääsynvalvonnan merkityksellisyys käytännössä riippuu siitä, mitä asioita ohjelmoijat arvostavat. Internetistä on löydettävissä keskusteluja[1], joiden mukaan tiedon piilottaminen on yliarvostettua, ja luokan yksityiset jäsenet jopa haitallisia. Toisaalla taas kysellään, kuten luvussa 5.1 viitattiin, miksi Java ei mahdollista olioyksityisyyttä, ja miksi suojatut jäsenet ovat saatavissa myös saman pakkauksen sisällä. Provokatiivisesti voisi kysyä, että jos kerran tiedon piilottaminen on tarpeetonta, miksi moderneihinkin kieliin on määritelty pääsynvalvonta? Kuten luvussa 3.1 viitattiin, tutkijat ja asiantuntijat ovat vakuuttuneita tiedon piilottamisen hyödyistä. Kytkettävän pääsynvalvonnan valttina on

[1] `http://c2.com/cgi/wiki?MethodsShouldBePublic`,
`http://ginstrom.com/scribbles/2007/11/12/three-reasons-to-avoid-private-class-members/`

sen valinnaisuus — ohjelmoija voi mieltymystensä mukaan ottaa siitä käyttöön tapauskohtaisesti tarpeelliseksi katsomansa osan, joka auttaa häntä tuottamaan modulaarisempia, testattavampia ja ylläpidettävämpiä ohjelmia.

Seuraava luonnollinen askel olisi toteuttaa kytkettävään pääsynvalvontajärjestelmään tutkielman luvussa 5.6 esitetyt kehitysideat sekä rakentaa kytkettävän pääsynvalvontakehyksen prototyypistä kokonainen toteutus. Kun pääsynvalvontakehyksestä on olemassa kokonainen toteutus, kytkettävän pääsynvalvonnan arvoa on mahdollista arvioida empiirisin menetelmin.

Lähteet

[ABV00] Alexander, R.T., Bieman, J.M. ja Viega, J.: *Coping with Java programming stress.* Computer, 33(4):30–38, huhtikuu 2000, ISSN 0018-9162. http://dx.doi.org/10.1109/2.839318.

[ANMM06] Andreae, Chris, Noble, James, Markstrum, Shane ja Millstein, Todd: *A Framework for Implementing Pluggable Type Systems.* Teoksessa *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, sivut 57–74, New York, NY, USA, 2006. ACM, ISBN 1-59593-348-4. http://doi.acm.org/10.1145/1167473.1167479.

[Ass] *Assemblies and the Global Assembly Cache (C# and Visual Basic).* https://msdn.microsoft.com/en-us/library/ms173099%28v=vs.140%29.aspx [15.3.2016].

[Bar15] Bartlett, Neil: *OSGi and Java 9 Modules Working Together*, marraskuu 2015. http://njbartlett.name/2015/11/13/osgi-jigsaw.html [18.4.2016].

[BBG+63] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., Wijngaarden, A. van ja Woodger, M.: *Revised Report on the Algorithm Language ALGOL 60.* Commun. ACM, 6(1):1–17, tammikuu 1963, ISSN 0001-0782. http://doi.acm.org/10.1145/366193.366201.

[Blo08] Bloch, Joshua: *Effective Java.* Prentice Hall PTR, 2 painos, 2008, ISBN 978-0-321-35668-0.

[Bra03] Bracha, Gilad: *Pluggable types.* Colloquium at Aarhus University, maaliskuu 2003. http://bracha.org/pluggable-types.pdf [12.10.2015].

[Bra04] Bracha, Gilad: *Pluggable type systems.* OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.175.1460.

[BU04] Bracha, Gilad ja Ungar, David: *Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages.* Teoksessa *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, sivut 331–344, New York, NY, USA, 2004. ACM, ISBN 1-58113-831-8. http://doi.acm.org/10.1145/1028976.1029004.

[C#06] *C# Language Specification.* Numero ECMA-334 sarjassa *Ecma Standards.* Ecma International, 4. painos, kesäkuu 2006. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[C#12] *C♯ Language Specification Version 4.0.* Microsoft Corporation, 2012. https://msdn.microsoft.com/en-us/library/ms228593.aspx [12.3.2016].

[Car97] Cardelli, Luca: *Type Systems.* Teoksessa *The Computer Science and Engineering Handbook*, luku 103, sivut 2208–2236. CRC Press, tammikuu 1997.

[Cas15] Cass, Stephen: *The 2015 Top Ten Programming Languages.* IEEE Spectrum, heinäkuu 2015. http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages [9.3.2016].

[Ceya] *Ceylon: FAQ about language design.* http://ceylon-lang.org/documentation/1.2/faq/language-design/ [24.4.2016].

[Ceyb] *Ceylon: Frequently Asked Questions.* http://ceylon-lang.org/documentation/1.2/faq/ [27.3.2016].

[Ceyc] *Ceylon Language Specification, Version 1.2.* http://ceylon-lang.org/documentation/1.2/spec/pdf/ceylon-language-specification.pdf [26.3.2016].

[CGLO06] Cremet, Vincent, Garillot, François, Lenglet, Sergueï ja Odersky, Martin: *Mathematical Foundations of Computer Science 2006: 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006. Proceedings*, luku A Core Calculus for Scala Type Checking, sivut 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, ISBN 978-3-540-37793-1. http://dx.doi.org/10.1007/11821069_1.

[Che16] *The Checker Framework Manual: Custom pluggable types for Java*, huhtikuu 2016. http://types.cs.washington.edu/checker-framework/releases/1.9.13/checker-framework-manual.pdf [10.4.2016].

[Coo89] Cook, W. R.: *A Proposal for Making Eiffel Type-safe.* The Computer Journal, 32(4):305–311, 1989. http://dx.doi.org/10.1093/comjnl/32.4.305.

[CT12] Cooper, Keith Daniel ja Torczon, Linda: *Engineering a Compiler.* Morgan Kaufmann, Amsterdam, 2. painos, 2012, ISBN 978-0-12-088478-0.

[CW85] Cardelli, Luca ja Wegner, Peter: *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Comput. Surv., 17(4):471–523, joulukuu 1985, ISSN 0360-0300. http://doi.acm.org/10.1145/6041.6042.

[Dah04] Dahl, Ole Johan: *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, luku The Birth of Object Orientation: the Simula Languages, sivut 15–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, ISBN 978-3-540-39993-3. http://dx.doi.org/10.1007/978-3-540-39993-3_3.

[DDE+11] Dietl, Werner, Dietzel, Stephanie, Ernst, Michael D., Muşlu, Kivanç ja Schiller, Todd W.: *Building and Using Pluggable Type-checkers.* Teoksessa *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, sivut 681–690, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0445-0. `http://doi.acm.org/10.1145/1985793.1985889`.

[DE97] Drossopoulou, Sophia ja Eisenbach, Susan: *Java is type safe — Probably.* Teoksessa Akşit, Mehmet ja Matsuoka, Satoshi (toimittajat): *ECOOP'97 — Object-Oriented Programming*, nide 1241 sarjassa *Lecture Notes in Computer Science*, sivut 389–418. Springer Berlin Heidelberg, 1997, ISBN 978-3-540-63089-0. `http://dx.doi.org/10.1007/BFb0053388`.

[DEK99] Drossopoulou, Sophia, Eisenbach, Susan ja Khurshid, Sarfraz: *Is the Java Type System Sound?* Theor. Pract. Object Syst., 5(1):3–24, tammikuu 1999, ISSN 1074-3227. `http://dx.doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAPO2>3.0.CO;2-T`.

[Fer14] Fernández, Maribel: *Programming Languages and Operational Semantics.* Springer-Verlag London, 2014, ISBN 978-1-4471-6367-1.

[Fri] *Friend Assemblies (C# and Visual Basic).* `https://msdn.microsoft.com/en-us/library/0tke9fxk%28v=vs.140%29.aspx` [15.3.2016].

[GF07] Greenfieldboyce, David ja Foster, Jeffrey S.: *Type Qualifier Inference for Java.* Teoksessa *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, sivut 321–336, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-786-5. `http://doi.acm.org/10.1145/1297027.1297051`.

[GHJV95] Gamma, Erich, Helm, Richard, Johnson, Ralph ja Vlissides, John: *Design patterns: elements of reusable object-oriented software*, luku Visitor, sivut 331–344. Addison-Wesley, 1995.

[GJS+15] Gosling, James, Joy, Bill, Steele, Guy, Bracha, Gilad ja Buckley, Alex: *The Java® Language Specification, Java SE 8 Edition.* Oracle Corporation, helmikuu 2015. `http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.

[GR83] Goldberg, Adele ja Robson, David: *Smalltalk-80 The Language and its Implementation.* Addison-Wesley, 1983, ISBN 0-201-11371-6. `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

[Ham08] Hamilton, Naomi: *The A-Z of Programming Languages: C#*, lokakuu 2008. `http://www.computerworld.com.au/article/261958/a-z_programming_languages_c_/` [12.3.2016].

[HKR+13] Hanenberg, Stefan, Kleinschmager, Sebastian, Robbes, Romain, Tanter, Éric ja Stefik, Andreas: *An empirical study on the impact of static typing on software maintainability.* Empirical Software Engineering, 19(5):1335–1382, 2013, ISSN 1573-7616. `http://dx.doi.org/10.1007/s10664-013-9289-1`.

[HP04] Hovemeyer, David ja Pugh, William: *Finding Bugs is Easy.* SIGPLAN Not., 39(12):92–106, joulukuu 2004, ISSN 0362-1340. `http://doi.acm.org/10.1145/1052883.1052895`.

[Inf06] *Information technology — Programming languages — C#.* Numero ISO/IEC 23270:2006(E) sarjassa *International standard*. ISO, 2. painos, syyskuu 2006. `http://standards.iso.org/ittf/PubliclyAvailableStandards/c042926_ISO_IEC_23270_2006%28E%29.zip`.

[Int16] *IntelliJ IDEA 2016.1 Help*, luku Rename Dialog for a Class or an Interface. maaliskuu 2016. `https://www.jetbrains.com/help/idea/2016.1/rename-dialog-for-a-class-or-an-interface.html` [15.4.2016].

[Jav14] *Java Platform, Standard Edition Tools Reference, Release 8 for Oracle JDK on Solaris, Linux, and OS X*, luku javac. Oracle, elokuu 2014. `http://docs.oracle.com/javase/8/docs/technotes/tools/unix/javac.html` [27.4.2016].

[Jav15] *The Java™ Tutorials*, luku Annotations. Oracle, heinäkuu 2015. `http://docs.oracle.com/javase/tutorial/java/annotations/index.html` [21.4.2016].

[Jav16a] *Java Compiler Tree API*, 2016. `http://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html` [5.5.2016].

[Jav16b] *Java™ Platform, Standard Edition 8 API Specification*, 2016. `http://docs.oracle.com/javase/8/docs/api/index.html` [21.4.2016].

[Jav16c] *Java Programming Language Enhancements*, 2016. `http://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html` [21.4.2016].

[JSR06] *JSR 270: Java™ SE 6 Release Contents*, joulukuu 2006. `https://jcp.org/en/jsr/detail?id=270`.

[JSR14] *JSR 308: Annotations on Java® Types*, helmikuu 2014. `https://jcp.org/en/jsr/detail?id=308`.

[Kay93] Kay, Alan C.: *The Early History of Smalltalk.* Teoksessa *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, sivut 69–95, New York, NY, USA, 1993. ACM, ISBN 0-89791-570-4. `http://doi.acm.org/10.1145/154766.155364`.

[Kin11] King, Gavin: *First official release of Ceylon*, joulukuu 2011. http://ceylon-lang.org/blog/2011/12/20/ceylon-m1-newton/ [27.3.2016].

[Lew95] Lewis, Simon: *The Art and Science of Smalltalk*. Prentice Hall International (UK), 1995, ISBN 0-13-371345-8. http://sdmeta.gforge.inria.fr/FreeBooks/Art/artAdded174186187Final.pdf.

[LF03] Link, Johannes ja Frlich, Peter: *Unit Testing in Java : How Tests Drive the Code*. Morgan Kaufmann, 2003, ISBN 978-155-860868-9.

[LHL+77] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G. ja Popek, G. J.: *Report on the Programming Language Euclid*. SIGPLAN Not., 12(2):1–79, helmikuu 1977, ISSN 0362-1340. http://doi.acm.org/10.1145/954666.971189.

[Lis96] Liskov, Barbara: *History of Programming languages—II*. luku A History of CLU, sivut 471–510. ACM, New York, NY, USA, 1996, ISBN 0-201-89502-1. http://doi.acm.org/10.1145/234286.1057826.

[LSAS77] Liskov, Barbara, Snyder, Alan, Atkinson, Russell ja Schaffert, Craig: *Abstraction Mechanisms in CLU*. Commun. ACM, 20(8):564–576, elokuu 1977, ISSN 0001-0782. http://doi.acm.org/10.1145/359763.359789.

[LYBB15] Lindholm, Tim, Yellin, Frank, Bracha, Gilad ja Buckley, Alex: *The Java® Virtual Machine Specification, Java SE 8 Edition*. Oracle Corporation, helmikuu 2015. http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[M+05] Meyer, Bertrand *et al.*: *Eiffel: Analysis, Design and Programming Language*. Numero ECMA-367 sarjassa *Ecma Standards*. Ecma International, 1. painos, kesäkuu 2005. http://www.ecma-international.org/publications/standards/Ecma-367-arch.htm.

[M+06] Meyer, Bertrand *et al.*: *Eiffel: Analysis, Design and Programming Language*. Numero ECMA-367 sarjassa *Ecma Standards*. Ecma International, 2. painos, kesäkuu 2006. http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[Mac93] MacQueen, David B.: *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991–1992 McMaster University, Hamilton, Ontario, Canada*, luku Reflections on standard ML, sivut 32–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, ISBN 978-3-540-47776-1. http://dx.doi.org/10.1007/3-540-56883-2_2.

[MD04] Meijer, Erik ja Drayton, Peter: *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*. OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.394.3818.

[Mer] *Module.* http://www.merriam-webster.com/dictionary/module [25.4.2016].

[Mey85] Meyer, Bertrand: *Eiffel: A Language for Software Engineering.* tekninen raportti, University of California, Department of Computer Science, joulukuu 1985. http://se.ethz.ch/~meyer/publications/eiffel/eiffel_report.pdf.

[Mey97] Meyer, Bertrand: *Object-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle River, N.J, 2. painos, 1997, ISBN 0-13-629155-4.

[Mey01] Meyer, Bertrand: *An Eiffel Tutorial.* tekninen raportti, Interactive Software Engineering Inc. (ISE), heinäkuu 2001. http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial.pdf [9.3.2016].

[MPH99] Müller, Peter ja Poetzsch-Heffter, Arnd: *JIT'98 Java-Informations-Tage 1998: Frankfurt/Main, 12./13. November 1998*, luku Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur, sivut 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, ISBN 978-3-642-59984-2. http://dx.doi.org/10.1007/978-3-642-59984-2_1.

[MY60] McNaughton, R. ja Yamada, H.: *Regular Expressions and State Graphs for Automata.* IRE Transactions on Electronic Computers, EC-9(1):39–47, March 1960, ISSN 0367-9950. http://dx.doi.org/10.1109/TEC.1960.5221603.

[NvO98] Nipkow, Tobias ja Oheimb, David von: *Javalight is Type-safe — Definitely.* Teoksessa *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, sivut 161–170, New York, NY, USA, 1998. ACM, ISBN 0-89791-979-3. http://doi.acm.org/10.1145/268946.268960.

[OAC+] Odersky, Martin, Altherr, Philippe, Cremet, Vincent, Dubochet, Gilles, Emir, Burak, Haller, Philipp, Micheloud, Stéphane, Mihaylov, Nikolay, Moors, Adriaan, Rytz, Lukas, Schinz, Michel, Stenman, Erik ja Zenger, Matthias: *Scala Language Specification, Version 2.11.* http://www.scala-lang.org/files/archive/spec/2.11/ [04.03.2016].

[OR14] Odersky, Martin ja Rompf, Tiark: *Unifying Functional and Object-oriented Programming with Scala.* Commun. ACM, 57(4):76–86, huhtikuu 2014, ISSN 0001-0782. http://doi.acm.org/10.1145/2591013.

[OSGa] *OSGi Alliance — Specifications.* https://www.osgi.org/developer/specifications/ [18.4.2016].

[OSGb] *OSGi Alliance — Where to Start.* https://www.osgi.org/developer/where-to-start/ [18.4.2016].

[OSG14] *OSGi Core Release 6 Specification*. OSGi Alliance, kesäkuu 2014. `https://www.osgi.org/developer/downloads/release-6/` [18.4.2016].

[PAC+08] Papi, Matthew M., Ali, Mahmood, Correa, Jr., Telmo Luis, Perkins, Jeff H. ja Ernst, Michael D.: *Practical Pluggable Types for Java*. Teoksessa *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, sivut 201–212, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-050-0. `http://doi.acm.org/10.1145/1390630.1390656`.

[Par72] Parnas, D. L.: *On the Criteria to Be Used in Decomposing Systems into Modules*. Commun. ACM, 15(12):1053–1058, joulukuu 1972, ISSN 0001-0782. `http://doi.acm.org/10.1145/361598.361623`.

[Phi12] Phillips, Paul: *Can we please reach some kind of decision on access*, joulukuu 2012. `https://issues.scala-lang.org/browse/SI-6794` [22.4.2016].

[Pro16] *Project Jigsaw*, maaliskuu 2016. `http://openjdk.java.net/projects/jigsaw/` [18.4.2016].

[Rei16] Reinhold, Mark: *The State of the Module System*, maaliskuu 2016. `http://openjdk.java.net/projects/jigsaw/spec/sotms/2016-03-08` [18.4.2016].

[Sch04] Schirmer, Norbert: *Analysing the Java package/access concepts in Isabelle/-HOL*. Concurrency and Computation: Practice and Experience, 16(7):689–706, 2004, ISSN 1532-0634. `http://dx.doi.org/10.1002/cpe.800`.

[Sco09] Scott, Michael Lee: *Programming Language Pragmatics*. Morgan Kaufmann, Amsterdam, 3. painos, 2009, ISBN 978-0-12-374514-9.

[SH11] Stuchlik, Andreas ja Hanenberg, Stefan: *Static vs. Dynamic Type Systems: An Empirical Study About the Relationship Between Type Casts and Development Time*. Teoksessa *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, sivut 97–106, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0939-4. `http://doi.acm.org/10.1145/2047849.2047861`.

[Sha84] Shaw, M.: *Abstraction Techniques in Modern Programming Languages*. IEEE Software, 1(4):10–26, lokakuu 1984, ISSN 0740-7459. `http://dx.doi.org/10.1109/MS.1984.229453`.

[Ste14] Stephens, Rod: *C# 5.0 Programmer's Reference*. Wiley, 2014, ISBN 978-1-118-84697-1.

[Str96] Stroustrup, Bjarne: *History of Programming languages—II*. luku A History of C++: 1979–1991, sivut 699–769. ACM, New York, NY, USA, 1996, ISBN 0-201-89502-1. `http://doi.acm.org/10.1145/234286.1057836`.

[Str13] Stroustrup, Bjarne: *The C++ programming language*. Addison-Wesley, Upper Saddle River (N.J.), 4. painos, 2013, ISBN 978-0-321-56384-2.

[SV08] Siek, Jeremy G. ja Vachharajani, Manish: *Gradual Typing with Unification-based Inference*. Teoksessa *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, sivut 7:1–7:12, New York, NY, USA, 2008. ACM, ISBN 978-1-60558-270-2. `http://doi.acm.org/10.1145/1408681.1408688`.

[THF08] Tobin-Hochstadt, Sam ja Felleisen, Matthias: *The Design and Implementation of Typed Scheme*. Teoksessa *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, sivut 395–406, New York, NY, USA, 2008. ACM, ISBN 978-1-59593-689-9. `http://doi.acm.org/10.1145/1328438.1328486`.

[Tra09] Tratt, Laurence: *Dynamically Typed Languages*. Nide 77 sarjassa *Advances in Computers*, luku 5, sivut 149–184. Elsevier, 2009. `http://dx.doi.org/10.1016/S0065-2458(09)01205-4`.

[VIC10] Voigt, Janina, Irwin, Warwick ja Churcher, Neville: *Class Encapsulation and Object Encapsulation: An Empirical Study*. 2010. `http://hdl.handle.net/10092/5583`.

[Wir76] Wirth, Niklaus: *MODULA*. tekninen raportti, Eidgenössische Technische Hochschule Zürich, 1976. `http://dx.doi.org/10.3929/ethz-a-000199440`.

[Wir80] Wirth, Niklaus: *MODULA-2*. tekninen raportti, Eidgenössische Technische Hochschule Zürich, 1980. `http://dx.doi.org/10.3929/ethz-a-000189918`.

[Wir02] Wirth, Niklaus: *Software Pioneers*. luku Pascal and Its Successors, sivut 108–119. Springer-Verlag New York, Inc., New York, NY, USA, 2002, ISBN 3-540-43081-4. `http://dx.doi.org/10.1007/978-3-642-59412-0_8`.

[WS73] Wulf, W. ja Shaw, Mary: *Global Variable Considered Harmful*. SIGPLAN Not., 8(2):28–34, helmikuu 1973, ISSN 0362-1340. `http://doi.acm.org/10.1145/953353.953355`.

[Zan07] Zander, Jason: *Couple of Historical Facts*, marraskuu 2007. `https://blogs.msdn.microsoft.com/jasonz/2007/11/22/couple-of-historical-facts/` [13.3.2016].

Liite 1. Kytkettävän pääsynvalvontajärjestelmän yksityiskohtia

Liitteeseen on koottu yksityiskohtaisia teknisluontoisia määritelmiä kytkettävään pääsynvalvontajärjestelmään liittyen. Liitteessä esiintyvät lähdeviittaukset viittaavat tutkielman lähdeluetteloon.

@AccessFromMatching

`@AccessFromMatching`-pääsymääre rajoittaa elementtiin pääsyn ainoastaan sellaisiin ohjelman osiin, jotka sopivat parametrina annettuihin merkkijonohahmoihin (pattern). Aksessoivan ohjelman osaan viittaavaa merkkijonoa, jota kutsuttakoon aksessoijaviitteeksi, sovitetaan hahmoihin. Pääsy rajoitettuun elementtiin sallitaan, jos, ja vain jos, aksessoijaviite täsmää vähintään yhteen annetuista hahmoista. Rajoitettava elementti voi olla pakkaus, luokka, kenttä, muodostin tai metodi. Hahmot, joihin aksessoijaviitettä sovitetaan, on mahdollista antaa pääsymääreelle kahdessa eri muodossa: säännöllisenä lausekkeena (regular expression) tai möykkyhahmona (glob).

Aksessoijaviitteen syntaksi

Aksessoijaviite *AccessorReference* muodostetaan aksessoivan luokan binäärinimestä sekä aksessoivan jäsenen esittelyn osasta. Luokan binäärinimi on Javan spesifikaatiossa määritelty käännettyjen luokkatiedostojen käyttämä muoto luokan nimestä [GJS+15, §13.1]. Annotoinnin yhteydessä aksessointi annotaation tyyppiin tai annotaation parametreista käsitellään niin kuin aksessointi tapahtuisi annotoitavasta elementistä. Säännöissä käytetään samaa merkintätapaa kuin Java-kielen spesifikaatiossa [GJS+15, §2.4].

AccessorReference:
    TypeBinaryName `::` *MemberSignature*
    PackageName
        kun aksessoija on pakkaukseen liitetty annotaatio

TypeBinaryName:
    PackageNamePrefix *OuterClassNamesPrefix* *TypeSimpleName*
        tyypin binäärimuotoinen nimi siten kuin Javan spesifikaatiossa [GJS+15, §13.1] on määritelty

PackageNamePrefix:
    [*PackageName* `.`]

PackageName:
    pakkauksen nimi siten kuin Javan spesifikaatiossa [GJS+15, §6.5.3] on määritelty

OuterClassNamesPrefix:
[*OuterClassNames* `$`]

OuterClassNames:
TypeSimpleName {`$` *TypeSimpleName*}

TypeSimpleName:
Identifier

Identifier:
Java-kielen tunnus siten kuin Javan spesifikaatiossa [GJS+15, §3.8] on määritelty

MemberSignature:
MemberName `(` *ParameterTypes* `)`
kun aksessoija on metodi
MemberName
kun aksessoija on staattisen luokkamuuttujan tai ilmentymämuuttujan tyyppiesittely
`new` `(` *ParameterTypes* `)`
kun aksessoija on muodostin
`new`
kun aksessoija on olion alustuslohko tai ilmentymämuuttujan alustuslauseke
`static`
kun aksessoija on luokan staattinen alustuslohko tai staattisen luokkamuuttujan alustuslauseke

MemberName:
Identifier

ParameterTypes:
[*TypeBinaryName* {`,` *TypeBinaryName*}]

Möykkyhahmot (glob)

Möykkyhahmot ovat säännöllisiä lausekkeita yksinkertaisempia hahmoja. Mistä tahansa möykkyhahmosta on johdettavissa sellainen säännöllinen lauseke, joka tunnistaa täsmälleen saman joukon merkkijonoja. Hahmon syntaksi on sovellettu versio Java-kirjaston `java.nio.file.FileSystem`-luokan `getPathMatcher`-metodin dokumentaatiossa määritellystä. Taulukossa 1 on kuvaus möykkyhahmossa käytettävistä ilmauksista. Kaikki muut ilmaukset, mitä taulukossa ei ole lueteltu, sovitetaan kirjaimellisesti merkkijonon osaksi.

Taulukko 1: `@AccessFromMatching`-pääsymääreen möykkyhahmoissa käytettävät erikoisilmaukset ja niiden kuvaukset.

Ilmaus	Kuvaus
*	Täsmää nollaan, yhteen tai useampaan määrään nimen osan merkkejä, muttei erotinmerkkeihin `$ . : ( ) ,`.
**	Täsmää nollaan, yhteen tai useampaan määrään nimen osien merkkejä erotinmerkit mukaan lukien, muttei erotinmerkkeihin `: ( ) ,`.
***	Täsmää nollaan, yhteen tai useampaan määrään merkkejä kaikki erotinmerkit mukaan lukien.
?	Täsmää yhteen nimen osan merkkiin.
$[X]$	Täsmää yhteen mihin tahansa merkkiin, joka kuuluu hakasulkeiden sisällä lueteltuun joukkoon X. Esimerkiksi `[abc]` täsmäävät merkkeihin `a`, `b` ja `c`. Joukon tai sen osan voi esittää myös merkkivälinä. esimerkiksi `[a-e]` ja `[abc-e]` täsmäävät merkkeihin `a`, `b`, `c`, `d` ja `e`. Joukon ensimmäisen merkin ollessa huutomerkki (`!`) joukko tulkitaan käänteisenä. Esimerkiksi `[!abc]` täsmää kaikkiin muihin merkkeihin paitsi a, b ja c. Hakasulkeiden välin ollessa tyhjä ilmaus `[]` tulkitaan kirjaimellisesti.
$\{X\}$	Täsmää osahahmoon, joka kuuluu aaltosulkeiden sisällä lueteltuun joukkoon X. Osahahmot erotetaan toisistaan pilkulla (`,`). Esimerkiksi `{ab,c*}` täsmää merkkeihin `ab` tai nimen osan sisällä mihin tahansa `c`-kirjaimella alkavaan. Osahahmot voivat sisältää erikoisilmauksia, myös sisäkkäisiä $\{X\}$-ilmaisuja. Aaltosulkeiden välin ollessa tyhjä ilmaus `{}` tulkitaan kirjaimellisesti.
\	Suojausmerkki (escape character), joka estää seuraavan merkin tulkitsemisen erikoisilmaukseksi. Esimerkiksi \`,` tulkitaan $\{X\}$-ilmaisun osahahmon sisällä pilkuksi eikä osahahmojen erotinmerkiksi. Ilmaus \\ tulkitaan kenoviivaksi.

Ennalta määritellyt muuttujat

Sekä säännöllisissä lausekkeissa että möykkyhahmoissa on mahdollista käyttää ennalta määriteltyjä muuttujia, jotka hahmoa sovitettaessa korvataan muuttujan arvolla. Muuttuja voi edustaa esimerkiksi rajoitettavan elementin pakkauksen, luokan tai metodin nimeä. Muuttujien ansiosta täsmälleen samaa hahmoa voi käyttää eri elementtien rajoittamisessa. Muuttujat ilmaistaan hahmoissa `${` ja `}` merkkien välissä.

Muuttujat muodostetaan rajoitetusta elementistä samalla tavoin kuin aksessoijaviite aksessoivasta ohjelman osasta noudattaen aksessoijaviitteen syntaksia. Mahdolliset muuttujat ovat samat kuin aksessoijaviitteen syntaksissa on määritelty, paitsi *AccessorReference* ja *Identifier*, joita ei käytetä muuttujina. Vaikka muuttujaa *TypeSimpleName* käytetään syntaksin määrittelyssä myös rajoitettua luokkaa ulompien luokkien yhteydessä, muuttujana se korvataan rajoitetun elementin sisältävän luokan yksinkertaisella nimellä.

@SelfAccess

`@SelfAccess`-pääsymääre rajoittaa elementtiin pääsyn ainoastaan sen esittelevän luokan ilmentymän sisään. Toisin sanoen saman luokan toisesta ilmentymästä ei ole pääsyä rajoitettuun elementtiin. Rajoitettava elementti voi olla olion kenttä tai metodi. Luokan staattinen jäsen ei voi olla rajoituksen kohteena. Pääsymääre ei rajoita pääsyä olion yliluokilta perimiinsä jäseniin, vaan rajoitukset perintähierarkiassa on toteutettavissa Javan sisäänrakennetuilla pääsymääreillä ja sisäkkäisten luokkien tapauksessa `@SelfAccess`-määreen parametrin avulla.

Sallitut aksessointilausekkeet

`@SelfAccess`-määre sallii pääsyn rajoitettuun jäseneen ainoastaan, kun siihen viitataan pelkällä jäsenen tunnisteella tai `this`- tai `super`-avainsanojen kanssa. Kyseisten avainsanojen edessä saa Javan syntaksin mukaisesti käyttää luokan nimeä tarkentamaan, minkä luokan jäseneen viitataan. Alla on esitetty määreen sallimien aksessointilausekkeiden (*SelfAccessExpression*) muodollinen syntaksi. Merkintätapa on sama kuin Java-kielen spesifikaatiossa [GJS+15, §2.4].

SelfAccessExpression:
  SelfFieldAccess
  SelfMethodInvocation
  `(` *SelfAccessExpression* `)`

SelfFieldAccess:
  Identifier
  SelfReference `.` *Identifier*

Identifier:
  Java-kielen tunnus siten kuin Javan spesifikaatiossa [GJS+15, §3.8] on määritelty

SelfReference:
  SelfPrimary
  `super`
  TypeName `.` `super`

SelfPrimary:
  `this`
  TypeName `.` `this`

TypeName:
  tyyppinimi siten kuin Javan spesifikaatiossa [GJS+15, §6.5] on määritelty

SelfMethodInvocation:

MethodName `(` *[ArgumentList]* `)`

SelfReference `.` *[TypeArguments] Identifier* `(` *[ArgumentList]* `)`

ArgumentList:

metodin parametrilista kuten Javan spesifikaatiossa [GJS+15, §15.12] on määritelty

TypeArguments:

tyyppiparametrit kuten Javan spesifikaatiossa [GJS+15, §4.5.1] on määritelty

Liite 2. Esimerkkejä kytkettävän pääsynvalvontajärjestelmän käytöstä

Liitteeseen on koottu joitakin esimerkkilistauksia kytkettävän tyyppijärjestelmän käytöstä.

Javan pääsyrajoitteiden mallintaminen

Listaus 1 havainnollistaa Javan sisäänrakennettujen pääsyrajoitteiden toimintaa. Luokka `Restricted` esittelee neljä metodia: `publicMethod`, `protectedMethod`, `packagePrivateMethod` ja `privateMethod`. Metodien saatavuus on järjestyksessä julkinen, suojattu, pakkauksen sisäinen ja yksityinen. Luokan metodi `access` havainnollistaa pääsyä edellä esiteltyihin metodeihin. Kaikkiin näistä on pääsy luokan sisällä.

Samassa pakkauksessa sijaitseva luokka `Accessor` havainnollistaa `Restricted`-luokan metodien saatavuutta saman pakkauksen sisällä. Alipakkauksessa `pkg` sijaitseva luokka `Accessor` havainnollistaa `Restricted`-luokan metodien saatavuutta pakkauksen ulkopuolella. Viimeisenä luokka `SubclassAccessor` havainnollistaa `Restricted`-luokan metodien saatavuutta pakkauksen ulkopuolisessa aliluokassa. Listauksessa sallitut kutsut on merkitty kommentilla ”OK” ja virheelliset kutsut kommentilla ”Error!”. Listauksessa 2 on kääntäjän tuloste, josta virheelliset kutsut on myös havaittavissa.

Listaus 3 esittää saman kuin listaus 1, mutta kaikki metodit on asetettu Javan `public`-määreellä julkisiksi. Metodien saatavuus on rajoitettu kytkettäviä pääsymääreitä `@AccessWithin` ja `@AccessFromInherited` yhdistelemällä siten, että saatavuus vastaa Javan sisäänrakennettujen pääsymääreiden semantiikkaa. Vastaavuudet on esitetty tutkielman luvun 5.5 taulukossa 5.2. Listauksesta 4 on nähtävissä, että vaikka virheilmoitukset ovat erilaiset, virheet syntyvät samoista kutsuista kuin listauksessa 1.

Listaus 1: Esimerkki Javan pääsyrajoitteiden vaikutuksesta.

```
A.1  package javaaccess.javanative;
A.2  public class Restricted {
A.3    public void publicMethod() {}
A.4    protected void protectedMethod() {}
A.5    void packagePrivateMethod() {}
A.6    private void privateMethod() {}
A.7
A.8    public void access() {
A.9      publicMethod();         // OK
A.10     protectedMethod();      // OK
A.11     packagePrivateMethod(); // OK
A.12     privateMethod();        // OK
A.13   }
A.14 }
A.15
B.1  package javaaccess.javanative;
B.2  public class Accessor {
B.3    public void access() {
B.4      final Restricted restricted = new Restricted();
B.5      restricted.publicMethod();          // OK
B.6      restricted.protectedMethod();       // OK
B.7      restricted.packagePrivateMethod();  // OK
B.8      restricted.privateMethod();         // Error!
B.9    }
B.10 }
B.11
C.1  package javaaccess.javanative.pkg;
C.2  import javaaccess.javanative.Restricted;
C.3  public class Accessor {
C.4    public void access() {
C.5      final Restricted restricted = new Restricted();
C.6      restricted.publicMethod();          // OK
C.7      restricted.protectedMethod();       // Error!
C.8      restricted.packagePrivateMethod();  // Error!
C.9      restricted.privateMethod();         // Error!
C.10   }
C.11 }
C.12
D.1  package javaaccess.javanative.pkg;
D.2  import javaaccess.javanative.Restricted;
D.3  public class SubclassAccessor extends Restricted {
D.4    public void access() {
D.5      publicMethod();         // OK
D.6      protectedMethod();      // OK
D.7      packagePrivateMethod(); // Error!
D.8      privateMethod();        // Error!
D.9    }
D.10 }
```

Listaus 2: Listauksen 1 käännöksen tulos.

```
javaaccess/javanative/Accessor.java:8: error: privateMethod() has private access in
    Restricted
    restricted.privateMethod();         // Error!
              ^
javaaccess/javanative/pkg/Accessor.java:7: error: protectedMethod() has protected access
    in Restricted
    restricted.protectedMethod();       // Error!
              ^
javaaccess/javanative/pkg/Accessor.java:8: error: packagePrivateMethod() is not public in
    Restricted; cannot be accessed from outside package
    restricted.packagePrivateMethod();  // Error!
              ^
javaaccess/javanative/pkg/Accessor.java:9: error: privateMethod() has private access in
    Restricted
    restricted.privateMethod();         // Error!
              ^
javaaccess/javanative/pkg/SubclassAccessor.java:7: error: cannot find symbol
    packagePrivateMethod(); // Error!
    ^
  symbol:   method packagePrivateMethod()
  location: class SubclassAccessor
javaaccess/javanative/pkg/SubclassAccessor.java:8: error: cannot find symbol
    privateMethod();        // Error!
    ^
  symbol:   method privateMethod()
  location: class SubclassAccessor
6 errors
```

Listaus 3: Javan pääsyrajoitteiden mallintaminen kytkettävällä tyyppijärjestelmällä.

```
A.1  package javaaccess.pluggable;
A.2  public class Restricted {
A.3    @AccessWithin(AccessWithinScope.ALL)
A.4    public void publicMethod() {}
A.5
A.6    @AccessFromInherited
A.7    @AccessWithin(AccessWithinScope.PACKAGE)
A.8    public void protectedMethod() {}
A.9
A.10   @AccessWithin(AccessWithinScope.PACKAGE)
A.11   public void packagePrivateMethod() {}
A.12
A.13   @AccessWithin(AccessWithinScope.TOP_LEVEL)
A.14   public void privateMethod() {}
A.15
A.16   public void access() {
A.17     publicMethod();         // OK
A.18     protectedMethod();      // OK
A.19     packagePrivateMethod(); // OK
A.20     privateMethod();        // OK
A.21   }
A.22 }
A.23
B.1  package javaaccess.pluggable;
B.2  public class Accessor {
B.3    public void access() {
B.4      final Restricted restricted = new Restricted();
B.5      restricted.publicMethod();          // OK
B.6      restricted.protectedMethod();       // OK
B.7      restricted.packagePrivateMethod();  // OK
B.8      restricted.privateMethod();         // Error!
B.9    }
B.10 }
B.11
C.1  package javaaccess.pluggable.pkg;
C.2  import javaaccess.pluggable.Restricted;
C.3  public class Accessor {
C.4    public void access() {
C.5      final Restricted restricted = new Restricted();
C.6      restricted.publicMethod();          // OK
C.7      restricted.protectedMethod();       // Error!
C.8      restricted.packagePrivateMethod();  // Error!
C.9      restricted.privateMethod();         // Error!
C.10   }
C.11 }
C.12
D.1  package javaaccess.pluggable.pkg;
D.2  import javaaccess.pluggable.Restricted;
D.3  public class SubclassAccessor extends Restricted {
D.4    public void access() {
D.5      publicMethod();         // OK
D.6      protectedMethod();      // OK
D.7      packagePrivateMethod(); // Error!
D.8      privateMethod();        // Error!
D.9    }
D.10 }
```

Listaus 4: Listauksen 3 käännöksen tulos käytettäessä kytkettävän pääsynvalvontakehyksen prototyyppiä.

```
javaaccess/pluggable/Accessor.java:8: error: [access.accesswithin.illegal] illegal access
    from outside of permitted scope.
    restricted.privateMethod();         // Error!
              ^
  required : TOP_LEVEL
javaaccess/pluggable/pkg/Accessor.java:7: error: [access.accessfrominherited.illegal]
    access is permitted only from inheriting class.
    restricted.protectedMethod();       // Error!
              ^
  accessing class      : javaaccess.pluggable.pkg.Accessor
  required subclass of : javaaccess.pluggable.Restricted
javaaccess/pluggable/pkg/Accessor.java:8: error: [access.accesswithin.illegal] illegal
    access from outside of permitted scope.
    restricted.packagePrivateMethod();  // Error!
              ^
  required : PACKAGE
javaaccess/pluggable/pkg/Accessor.java:9: error: [access.accesswithin.illegal] illegal
    access from outside of permitted scope.
    restricted.privateMethod();         // Error!
              ^
  required : TOP_LEVEL
javaaccess/pluggable/pkg/SubclassAccessor.java:7: error: [access.accesswithin.illegal]
    illegal access from outside of permitted scope.
    packagePrivateMethod(); // Error!
    ^
  required : PACKAGE
javaaccess/pluggable/pkg/SubclassAccessor.java:8: error: [access.accesswithin.illegal]
    illegal access from outside of permitted scope.
    privateMethod();        // Error!
    ^
  required : TOP_LEVEL
6 errors
```