Skaalautuvien ja uudelleenkäytettävien ohjelmistokomponenttien muodostus Scalalla

TURUN YLIOPISTO
Informaatioteknologian laitos
Tietojenkäsittelytieteet
Henrik Huttunen
2007

TURUN YLIOPISTO
Informaatioteknologian laitos

HUTTUNEN, HENRIK: Skaalautuvien ja uudelleenkäytettävien ohjelmistokomponenttien muodostus Scalalla

LuK-tutkielma, 29 sivua
Tietojenkäsittelytieteet
Huhtikuu 2007

Uudelleenkäytettävien ohjelmistojen rakennusyksikön, ohjelmistokomponentin, kehittämisessä käytetään usein joko funktionaalista osiin jakamista tai oliomallinnusta. Scala-ohjelmointikieli tukee molempia lähestymistapoja vahvasti staattisesti tyypitetyssä ympäristössä.

Scala pohjautuu vahvasti tyyppiteoriaan, johon sisältyy mm. polkusidonnainen tyyppi, singleton-tyyppi, abstrakti jäsentyyppi ja yhdistelmätyyppi. Komponetteja yhdistetään käyttämällä sekoitekoostamista piirreluokkien, tyyppijäsenten ja itsetyyppien avulla. Moniperinnän pääongelmat vältetään luokkalinearisaatiota käyttämällä. Komponenttien yleistystä tukee yhtenäinen tyyppijärjestelmä: kaikki arvot ovat olioita. Lisäksi myös funktiot ovat olioita, jolloin on mahdollista abstrahoida toistuvaa toiminnallisuutta. Oliohierarkioiden läpikäymistä varten Scalassa on olioiden hahmonsovitus, jota voidaan käyttää sekä tapausluokkien (engl. case class) että erottimien (engl. extractor) avulla. Sen vähentämiseksi, että ohjelmoijan tarvitsisi kirjoittaa tyyppiesittelyjä, Scalassa on paikallinen tyypinpäättelymekanismi.

Tässä tutkielmassa esitetään, kuinka Scala hyödyntää funktionaalisen ja olioparadigman käsitteitä. Erityisesti keskitytään niihin abstraktioihin, joilla saadaan skaalautuvia, uudelleenkäytettäviä komponentteja ytimekkäästi ilmaistuksi. Lisäksi vertaillaan joitakin yksittäisiä tekniikoita muissa kielissä oleviin ratkaisuihin. Lähteinä on käytetty Scalan tutkimukseen liittyviä julkaisuja ja yleistä ohjelmistokirjallisuutta.

Avainsanat:

Scala, uudelleenkäytettävä, komponentti, sekoitekoostaminen, piirreluokka, itsetyyppi, abstrakti tyyppijäsen

Sisältö

1 Johdanto

Keskeisin ohjelmistoarkkitehtuurin käsite on o*hjelmistokomponentti*: se on pienin itsenäinen, rajapintojen kautta palvelujaan tarjoava ohjelmistoyksikkö, josta muodostetaan kokonaisuuksia. Komponentti voi olla kooltaan pieni funktio tai täysimittainen sovellus. [1] Kuitenkin ohjelmointikielien tuki uudelleenkäytettävien komponettien muodostamisessa vaihtelee suuresti. [2]

Tässä tutkielmassa esittelen kielen, joka tukee vahvasti komponenttipohjaista ohjelmointia. [2] Esittelyä varten määrittelen funktionaalisen ohjelmoinnin tarkoittavan sitä ohjelmointitapaa, missä funktiot ovat ykkösluokan kansalaisia: niitä voidaan sijoittaa muuttujiin, antaa toisille funktioille argumentteina ja palauttaa funktion arvona. [3] Oliosuuntautuneella ohjelmoinnilla eli olio-ohjelmoinnilla puolestaan tarkoitan ohjelmointitapaa, jossa ongelman mallinnus ja toteutus pohjautuvat vahvasti luokkahierarkioiden käyttöön ja suoritus perustuu olioiden keskenäiseen vuorovaikutukseen.

Funktionaalisen ja oliosuuntautuneen ohjelmoinnin yhdistävä *Scala*-ohjelmointikieli antaa ohjelmoijalle lukuisia toisiinsa liittyviä ohjelmointiabstraktioita, joiden avulla komponentteja voidaan muodostaa ytimekkäästi. Scala on vahvasti staattisesti tyypitetty, ja se pohjautuu hyvin muodostettuun tyyppiteoriaan, *vObj*-kalkyyliin. [4] Scalan taustan teoreettisuus johtuu siitä, että sitä kehitetään sveitsiläisessä teknillisen korkeakoulun (EPFL) tutkimusyksikössä. Useat kyseisen tutkimusyksikön julkaisut ovatkin keskeisiä tämän tutkielman lähteitä.

Tarkastelen tässä tutkielmassa ensin Scalan abstraktioita ja annan havainnollistavia esimerkkejä. Luvussa 3.1 selvitän funktionaalisen ohjelmoinnin etuja yksityiskohtaisesti, kun taas yleisimmät olio-ohjelmoinnin edut käsittelen lyhyesti. Sen jälkeen osoitan, miten Scalalle ominaisia oliosuuntautuneita abstraktioita yhdistämällä voidaan tukea ytimekästä skaalautuvien ja uudeelleenkäytettävien komponenttien muodostamista. Vertailen myös Scalan tekniikoita muiden kielien vastaaviin, jotta nähdään, miksi valitut menetelmät ovat tarpeellisia.

2 Scalan ohjelmointiabstraktiot

2.1 Perusrakenteet

2.1.1 Yhtenäinen oliomalli

Scala pohjautuu yhtenäiseen tyyppijärjestelmään, jossa kaikki arvot ovat olioita. Jokainen olio on puolestaan jonkin luokan ilmentymä. Lisäksi on olemassa kaikkien luokkien yläluokka, juuriluokka `Any`, sekä kaikkien luokkien alaluokka `Nothing`. [5] Yhtenäisyydellä on se etu, että vältytään poikkeustapauksilta. Koska perustyypitkin (esim. kokonaislukuja mallintava `Int`) esitetään luokkien avulla, voidaan niitä laajentaa perinnällä tai käyttää geneerisyyden tukena. Lisäksi täysin yhtenäisen tyyppijärjestelmän sisällyttäminen kieleen vaatii usean olio-ohjelmointitekniikan käyttöä, esimerkiksi infix-operaatioita ja tyyppirajattu geneerisyyttä. [6]

Olioihin viitataan jollakin tunnisteella (engl. identifier). Tunniste on muuttuja, ja sen viittausta voidaan muuttaa ainoastaan, jos sen esittelyssä käytetään `var`-määrettä. Muutoin tunniste on vakio, ja sen viittaus on muuttumaton. Silloin käytetään `val`-määrettä tai poikkeustapauksissa sitä ei tarvitse erikseen kirjoittaa näkyviin. On kuitenkin huomattava, että vaikka viittaus itse ei voi muuttua, viittauksen kohteena oleva olio mahdollisesti voi.

2.1.2 Metodit ja funktiot

Jokaisella operaatiolla on Scalassa palautusarvo. Proseduuri-käsitettä vastaavat ne operaatiot, joiden palautusarvona on `Unit`-luokan ainoa instanssi `()`. Operaatiota, joka on luokan jäsen, sanotaan *metodiksi*. Lisäksi jostakin esimääritetystä funktioluokasta peritty luokka ja sen apply-metodi muodostavat yhdessä operaation, jota kutsutaan *funktioksi*.

Metodit esitellään ja määritetään käyttämällä `def`-määrettä, tyyppiparametreja, arvoparametreja tyyppeineen, palautustyyppiä ja suoritusrunkoa. Metodin rungon eri suorituspolkujen tulee päättyä arvolausekkeeseen, jossa saa halutessa käyttää imperatiivisista kielistä tuttua `return`-sanaa. [5]

Metodin formaalit parametrit voivat olla joko viitearvoja tai ns. nimeltä kutsuttavia (nimikutsu, engl. call-by name). Nimeltä kutsuttavat parametrit evaluoivat arvonsa vasta, kun niitä todella käytetään suorituspolulla. Ne toimivat siis *laiskasti* (engl. lazy). Niiden esitysmuoto on `nimi: => A`. Viitearvoparametrit puolestaan evaluoidaan aina, ja ne kirjoitetaan muodossa `nimi: A`. [5]

Koodi 1 määrittelee metodin `korotaToiseen`, joka palauttaa argumenttina annetun `x`:n korotettuna toiseen potenssiin. Huomautettakoon, että palautustyypin voi jättää pois ei-rekursiivisista metodeista kääntäjän tyypinpäättelyn ansiosta (ks. luku 2.1.3).

Koodi 1

```
1: def korotaToiseen(x: Int): Int = x*x
```

Koodi 2 demonstroi, kuinka `x` parametrin sisältämää arvoa ei koskaan evaluoida. Jos `x`:n tyypiksi vaihdettaisiin `Int`, tulostuisi konsoliin `et näe tätä`.

Koodi 2

```
1: def eiEvaluoidu = Console.println(”et näe tätä”); 10
2: def testaa(x: => Int) = if(false) x else 0
3: testaa(eiEvaluoidu)
```

Scalassa funktiot ovat olioita. Se tarkoittaa, että niitä voidaan antaa toisille funktioille argumentteina, palauttaa funktion arvona ja sijoittaa muuttujaan sekä vakioon. Funktioille, joilla on eri ariteetti, on oma luokkansa `Function0`, `Function1` jne. Esimerkiksi kaksiparametrisen funktion määrittely on muotoa `Function2[-a1, -a2, +b]{...}`, missä `a1, a2` ja `b` ovat varianssimerkittyjä tyyppiparametreja (ks. luku 2.1.3). Sille on olemassa lyhennysmerkintä `(a1, a2) => b`. [5]

Funktioilla on metodi $\texttt{apply(p}_1\texttt{: a}_1\texttt{, ... p}_n\texttt{: a}_n\texttt{): b}$, jota voidaan kutsua syntaktisen sokerin ansiosta muodossa $\texttt{funktio(arg}_1\texttt{,...,arg}_n\texttt{)}$, missä funktio on tunniste funktioluokasta luotuun olioon. Luokkiin liittyvät metodit eivät ole funktioita, mutta niitä voidaan ottaa käyttöön funktioina `&`-erikoismerkinnän avulla. [5]

Koodi 3 havainnollistaa korkeamman asteen funktioiden käyttöä.

Koodi 3

```
1: def muotoile(muotoilija: String => String, sanat: List[String]) =
2:    for(val sana <- sanat) Console.print(muotoilija(sana))
3: def huuda(x: String) = x.toUpperCase
4: val lista = List("apua, ", "koski")
5: muotoile(huuda, lista)
```

`muotoile`-metodi käy parametrinaan saamansa sanalistan läpi. Se muotoilee jokaisen sanan yksitellen toisena parametrina saadulla funktiolla `muotoilija` ja tulostaa muotoillun sanan konsoliin. `muotoilija`n tyyppi on `String => String`. Ts. se on funktio, jolla on yksi `String`-tyyppinen parametri ja jonka palautusarvo on myös `String`. `huuda` on `muotoilija`ksi `sopiva metodi`, joka muuntaa annetun merkkijonon kirjaimet isoiksi. Viimeisellä rivillä `muotoile`-metodia kutsutaan argumenteilla `huuda` ja `lista`, jonka alkioina on `apua` ja `koski`. Konsoliin tulostuu `APUA, KOSKI`.

Funktiot ja metodit voivat sisältää toisia funktioita ja metodeita. Lisäksi voidaan luoda anonyymeja funktioita. Tällöin funktion runko annetaan sellaisenaan kohdassa, missä vaaditaan funktiotyyppinen arvo. Koska metodit ovat funktioita yleisessä mielessä, käytän vastedes pääsääntöisesti molemmista nimitystä funktio.

Koodi 4 luo anonyymin funktion. Se palauttaa uuden sanan annetusta sanasta siten, että o-kirjaimet on korvattu u:lla. Anonyymi funktio ja sanalista annetaan `muotoile`-metodin argumentiksi ja konsoliin tulostuu `apua, kuski`.

Koodi 4

```
1: muotoile({x => x.replace('o', 'u')}, lista)
```

Joskus halutaan, että funktiolle voidaan antaa vain osa parametreista ja että silloin saadaan uusi, parametrilistamäärältään lyhyempi funktio. Tätä kutsutaan currying-menetelmäksi, ja se saadaan aikaiseksi määrittelemällä funktiolle useita parametrilistoja: $(p^1_1, \ldots, p^1_n) \ldots (p^m_1, \ldots, p^m_n)$. Jos kutsu on vaillinainen parametreiltaan, täytyy käyttää `&`-merkkiä parametreiltaan vaillinaisen funktiokutsun edessä, ellei kutsu ole koko funktion arvo ja funktion palautustyyppi on erikseen asetettu funktiotyypiksi. [7]

Koodissa 5 määritetään metodi `g`, jolla on kaksi parametrilistaa. Uusi metodi `f` muodostetaan `g`:stä siten, että `g`:n ensimmäinen argumentti annetaan. Testikutsu `f(2)` vastaa siis `g`:n kutsua `g(5)(2)`.

Koodi 5

```
def g(x: Int)(y: Int) = x*y; def f = &g(5); f(2)
```

2.1.3 Tyypinpäättely

Scala on staattisesti tyypitetty. Kääntäjän täytyy siis tietää lausekkeiden tyyppi käännösaikana. Jotta ohjelmoijan ei tarvitsisi nähdä paljon kirjoitusvaivaa tyyppimäärittelyjen takia, kääntäjässä on Hindley–Milner-tyylinen *tyypinpäättely* (engl. type inference). [7] Sen ansiosta ohjelmista voidaan jättää pois mm. muuttujien, vakioiden tai ei-rekursiivisten funktioiden (palautus)tyyppi sekä funktio- tai muodostinkutsun tyyppiparametrit, jos ne voidaan päätellä ympäristöstä.

2.2 Luokkakonstruktiot

2.2.1 Luokka

Luokan määritys alkaa `class`-määreellä. Sen jälkeen tulee luokan nimi ja ensisijainen luokkamuodostin. Ensisijaisella luokkamuodostimella on mahdollisesti tyyppiparametreja, arvoparametreja tyyppeineen ja luokkarunko. [5] Kun olio luodaan, vastaavan luokan ensisijainen muodostin suoritetaan melkein, kuin se olisi metodi. Tosin se on kompleksisempaa toissijaisten muodostimien sekä ala- ja yläluokkien takia.

Luokkarungossa voidaan määrittää metodeita, muuttuja- tai vakioattribuutteja, abstrakteja tyyppijäseniä, sisäluokkia sekä toissijaisia muodostimia. Niiden näkyvyyttä voidaan hallita `private`- ja `protected`-määreillä. Oletuksena ne ovat julkisia, pois lukien muodostimen parametrit. Muodostimen parametreille, joiden edessä on var- tai val-määre, generoidaan lisäksi julkinen saantifunktio (engl. getter) sekä var-määreisille parametreille asetusfunktio (engl. setter). [5]

Luokan yliluokka on oletuksena `AnyRef`, ellei erikseen `extends`-määreellä peritä jotain muuta luokkaa. [5] Luvussa 2.2.5 nähdään, miten perintämekanismi, tai oikeammin koostaminen, toimii.

Kutsun vastedes ensisijaista muodostinta pelkästään muodostimeksi, jos toisarvoisia muodostajia ei ole.

Koodi 6 esittelee, kuinka uusi luokka voidaan määritellä ja luoda siitä olio.

Koodi 6

```
1: class EkaLuokka(x: Int){
2:     val y = x*2
3: }
4: val eka = new EkaLuokka(5)
5: Console.println(eka.y)
```

Koodi määrittää luokan `EkaLuokka`. Sen muodostin saa parametrinaan `Int`-tyyppisen argumentin `x`, joka kerrottuna kahdella sidotaan vakiotunnukseen `y`. Rivillä neljä luodaan uusi olio `EkaLuokasta` ja rivillä viisi käytetään uuden olion julkista vakioattribuuttia tulostamisessa.

2.2.2 Objekti-singleton-luokka

`Object`-sanalla määritetyt luokat ovat niin sanottuja singleton-suunnittelumalleja. [8] Objekteista ei voi luoda useampia esiintymiä, ja niitä käytetään olioiden tapaan. Scalan erikoispiirre on, että samanniminen luokka ja objekti nähdään toisiaan tukevana. Luokkaan liittyvää objektia kutsutaan kumppaniksi (engl. companion object). Objektia käytetään mm. tehdas-suunnittelumallin [8] apuna sekä sellaisten metodien säilytyspaikkana, jotka olisi määritelty Javassa staattiseksi. [9]

Koodi 7 määrittelee `EkaLuokka`-luokan kumppanin. Kumppanilla on `apply`-metodi, joka luo uuden `EkaLuokan` oletusarvolla. Viidennellä rivillä kutsutaan kumppanin `applya` käyttäen syntaktista sokeria, ja tulostetaan luodun olion `y`-vakioon sidottu arvo.

Koodi 7

```
1:  object EkaLuokka{
2:     private val oletus = 5
3:     def apply() = new EkaLuokka(oletus)
4:  }
5:  Console.println(EkaLuokka().y)
```

2.2.3 Abstraktit luokat

Abstraktit luokat määritellään `abstract class` -määreellä. Ne voivat esitellä metodeja, muuttujia ja vakioita ilman määrittelyosaa. Niistä ei voida luoda olioita ilman, että vaillinaisesti toteutetut metodit täydennetään aliluokassa tai koostamisella. [5]

Esimerkki: `abstract class A{ def x: Int }` määrää `A`:n, jolla on toteuttamaton metodi `x`.

Piirreluokka (piirre, engl. trait) on abstraktin luokan erikoistus: se määritetään `trait`-määreellä eikä sen muodostimelle voi antaa arvoparametreja. Javan *interface*-käsitteeseen nähden sillä on suuri ero: piirteellä voi olla toteutettuja jäseniä, erityisesti metodeita. [5] Erikoisluonteensa ansiosta ainoastaan piirreluokkia voidaan käyttää sekoitekoostamisessa yhden yläluokan lisänä. (ks. luku 2.2.5). Ideana on määrittää jokin yleiskäyttöinen ominaisuus, toteuttaa se yhdessä paikassa ja käyttää sitä useassa eri yhteydessä mahdollisesti muiden piirteiden kanssa. Piirreluokkia sekoitetaan `with`-määreellä, ja ne voivat periä toisia piirreluokkia.

Esimerkki: `trait A{...}; trait B{...}; class C extends A with B`

2.2.4 Tyyppikäsitteitä

Abstraktit *tyyppijäsenet* (engl. abstract type member) mahdollistavat oliotyylisen tavan abstrahoida komponenttien vaaditut palvelut. [2] Tyyppi ilmaistaan `type`-määritteellä, ja sille voidaan antaa tyyppirajoitteita, myös aliluokissa. Tyyppiä voidaan rajoittaa sekä ylhäältä merkillä `:>` että alhaalta merkillä `<:`. [5] Viimeistään olionluonnin yhteydessä pitää kääntäjän tietää, mihin konkreettiseen tyyppiin abstrakti tyyppi sidotaan.

Koodissa 8 ensimmäisellä rivillä abstraktilla luokalla `A` on tyyppijäsenenä `a`, jolle annetaan rajoitus, että `a`:n täytyy olla `AnyRef`-luokan alityyppiä. Toisella rivillä asetetaan `a`-tyypin aliakseksi `String` kaikille `B`:stä luotaville esiintymille.

Koodi 8

```
1: abstract class A{ type a <: AnyRef }
2: class B extends A{ type a = String }
```

Abstrakteihin tyyppeihin liittyy läheisesti kolme käsitettä. Ensimmäinen niistä on polkusidonnainen tyyppi (engl. path-dependent type), mikä tarkoittaa, että tyyppi määräytyy muuttumattoman valintapolun mukaan. Esimerkiksi, jos luokassa A on jäsentyyppi `T` ja olio `o` on luokan `A` ilmentymä, niin `o.T` olisi polkusidonnainen tyyppi. Yleisesti se on muotoa $x_0 . . . x_n.t$, missä `n >= 0`, x_0 on vakioarvo, ja jokainen x_i tarkoittaa polun etuliitteen $x_0 . . . x_{i-1}$ vakioattribuuttia. Lisäksi `t` on polun $x_0 . . . x_n$ tyyppijäsen. [2]

Toinen käsite, tyyppivalinta (engl. type selection), tarkoittaa luokan sisäluokan valintaa käyttämällä `#`-merkintää muodossa `UlompiLuokka#SisempiLuokka`. Kolmas käsite puolestaan on singleton-tyyppi, millä tarkoitetaan tietyn *olion* ainutlaatuista tyyppiä – ei siis luokan tyyppiä. Näiden kolmen välillä on suora yhteys: polkusidonnainen tyyppi `p.t` on lyhenne lausekkeesta `p.type#t`, missä `p.type` tarkoittaa `p`:n singleton-tyyppiä. [2]

Polkusidonnaisten tyyppien ansiosta voidaan mm. käyttää abstraktin tyypin sisältävää oliota, vaikkei konkreettista tyyppiä tiedettäisi. Koodi 9 on siitä esimerkki.

Koodi 9

```
1: abstract class A{
2:   type T
3:   val arvo: T
4:   def tulosta(x: T): Unit
5: }
6: def f(x: A) = x.tulosta(x.a)
```

Koodi määrittää abstraktin luokan `A`, sille jäsentyypin `T` ja kaksi metodia, jotka perustuvat `T`:n käyttöön. `f`:ssä `x.tulosta`-metodikutsulle voidaan antaa argumenttina `x`.a. Todellista tyyppiä ei tunneta, mutta jäsentyyppien vastaavuus riittää. Se pitää paikkansa, koska niillä on sama polkusidonnainen tyyppi `x.T`. [2]

Singleton-tyyppi helpottaa puolestaan kutsuketjujen muodostamista. [2] Koodi 10 demonstroi sitä.

Koodi 10

```
01: class Kuorrutin(init: String){
02:     def arvo = teksti
03:     protected var teksti = init
04:     def eteen(x: String): this.type
05:         = { teksti = x + teksti; this }
06:     }
07: class LaajennettuKuorrutin(init: String)
08:       extends Kuorrutin(init){
09:     def taakse(x: String): this.type
10:         = { teksti = teksti + x; this }
11: }
12: val kuorrutin = new LaajennettuKuorrutin("mies")
13: val tulos = kuorrutin.eteen("Ihme").taakse(" MacGyver").arvo
14: Console.println(tulos)
```

Koodi 10 määrittää `eteen` ja `taakse` -metodien palautustyypiksi `this.typen`, joka on siis oliokohtainen tyyppi. Sen ansiosta `kuorrutin.eteen`-kutsun tiedetään palauttavan tyypin, jolla on `taakse`-metodi. Ilman singleton-tyyppiä vastaava ei onnistuisi. Rivillä 12 muodostetaan kutsujen ketjutuksella merkkijono, joka tulostetaan lopuksi: `Ihmemies MacGyver`.

Joustavaa tyyppien käyttöä varten on yhdistelmätyyppi (engl. compound type). Se muodostetaan useasta eri tyypistä listaamalla muodostettavan tyypin osatyypit `with`-määreellä eroteltuna.

Esimerkki. `def x(y: A with B).` [5]

Luokan itseviitteen eli sisäisen `this`-tunnisteen tyyppi, *itsetyyppi*, on useimmissa oliokielissä luokka itse. Scalassa se voidaan kuitenkin halutessa määrittää lähes miksi tahansa. Jos luokalla on `requires`-määrite, sen jäljessä oleva (yhdistelmä)tyyppi asettuu luokan itsetyypiksi. Tyyppijärjestelmän eheyden takia vaaditaan kaksi ehtoa:

1. Luokan `A` itsetyyppi täytyy olla `A`:n yläluokkien itsetyyppien alityyppi.
2. Luokan itsetyyppi on yläluokka sen olion tyypille, joka luodaan.

[10]

Geneerisyys eli eksplisiittinen parametrinen polymorfismi voidaan mallintaa abstraktien tyyppien avulla ilman ongelmia. [2] Kuitenkin funktionaalisille kielille tyypillisen

geneerisyyden käyttö on tarpeellista sen ytimekkäämmän esitystavan vuoksi, erityisesti polymorfisten funktioiden määrittelyissä.
Koodi 11 määrittää polymorfisen metodin `max`. Sitä voidaan soveltaa kaikkiin alkioihin, joiden ylätyyppi on `Ordered`. Ylärajoite tarvitaan takaamaan vertailuoperaation `<` olemassaolo.

Koodi 11

```
1: def max[a <: Ordered[a]](x: a, y: a) = if(y < x) x else y
```

Koodista nähdään, että tyyppiparametri voi itse olla rajoitteen osana. Scala tukee siis F-rajoitettua polymorfismia. [2]

2.2.5 Sekoitekoostaminen

Tähän mennessä on esitelty yksittäisiä tekniikoita, joilla voidaan säädellä luokkakonstruktioihin liittyviä yksityiskohtia. Tarvitaan kuitenkin vielä näitä yhdistävä mekanismi, jotta voidaan rakentaa eri palasista kokonaisuuksia. Scalassa sellaista menetelmää kutsutaan *sekoitekoostamiseksi* (engl. mixin composition). Se mahdollistaa kontrolloidun tavan mallintaa moniperintää edellä mainittujen tekniikoiden tukemana.

Sekoitekoostamiseen liittyy *luokkalinearisaatio* (engl. class linearization), joka määrää täsmällisesti luokkahierarkiaan kuuluvien luokkien järjestyksen ja sitä kautta luokkaan sisältyvien jäsenien korvautuvuuden. On siis yksikäsitteistä, mistä luokasta kukin jäsen on periytynyt.

Määritelmä:

Oletetaan seuraava määrittely luokalle `A`:

$$\texttt{class A with } A_1 \texttt{ with } \ldots \texttt{ with } A_n \texttt{ \{ \ldots \}}$$

Silloin luokan `A` linearisaatio `L(A)` määritellään seuraavasti:

$$\texttt{L(A) = A, L(}A_n\texttt{) +> \ldots +> L(}A_1\texttt{)}$$

missä `+>` tarkoittaa yhdistämistä, jossa `+>`-operaation oikealla puolella olevat elementit korvaavat vasemmalla puolella olevat identtiset elementit.

```
{a, A} +> B = { a, A +> B, jos a ei kuulu B:hen
              { A +> B, jos a kuuluu B:hen
```

[5]

Seuraava esimerkki havainnollistaa, kuinka linearisaatio muodostuu vaiheittain.

Esimerkki

```
trait A extends AnyRef
trait B extends AnyRef
class C extends B with A
```

```
Silloin:
L(C) = C +> L(A) +> L(B) = C, A, AnyRef +> B, AnyRef
     = C, A, B, AnyRef
```

Luokkalinearisaatio jalostaa perimisrelaatiota: Jos `C` on `D`:n aliluokka, niin silloin `C` edeltää `D`:tä missä tahansa linearisaatiossa, jossa molemmat `C` ja `D` esiintyvät. Lisäksi luokan `C` linearisaation loppuosa on aina `C`:n lähimmän yläluokan luokkalinearisaatio. [5]

Jäsenten korvautuvuus on hyvin määritelty luokkalinearisaation ansiosta:

```
1. Konkreettinen eli toteutettu jäsen korvaa aina
signatuuriltaan yhtäläisen abstraktin jäsenen.

2. Jos molemmat jäsenet ovat konkreettisia tai abstrakteja,
valitaan sen luokan jäsen, joka on muodostettavan luokan
luokkalinearisaatiossa järjestyksessä aiemmin.
```

[2]

Seuraavassa staattinen yläluokka tarkoittaa perittyä yläluokkaa eikä luokkalinearisaation mukaista edeltäjää.

Määritelmä: luokassa olevan yläluokan kutsun (`super.M`) täytyy toteuttaa seuraava:

```
Ol. on olemassa luokat C ja D. Lisäksi D perii C:n.
Silloin super.M kutsu C:ssä on tyyppikorrekti, jos se viittaa
M-nimiseen jäseneen C:n staattisessa yläluokassa.
D:ssä puolestaan super.M viittaa jäseneen M', joka on
signatuuriltaan sama kuin M ja sisältyy ensimmäiseen
mahdolliseen C:tä seuraavaan luokkaan, D:n
luokkalinearisaatiossa.
```

Lisäksi on mahdollista kutsua staattisen yläluokan abstraktia jäsentä, jos kutsumetodi on merkitty `abstract override` -määreellä. Todellinen kutsun kohde määräytyy luokkalinearisaation mukaan. `super`-kutsut ovat poikkeuksellisen hyödyllisiä Scalassa, koska niitä voidaan ketjuttaa yksiselitteisen luokkalinearisaation avulla monimutkaisen luokkahierarkian läpi. [2]

Sekoitekoostaminen pohjautuu vahvasti piirreluokkien käyttöön: sekoittamalla niitä luokkaan saadaan lisättyä toiminnallisuutta. Sekoitusjärjestystä vaihtamalla voi säätää jäsenten korvautuvuutta, ja täten saadaan aikaiseksi erilaista toimintaa. Lisäksi silloin, kun ei haluta etukäteen sitoutua johonkin tiettyyn (piirre)luokkaan, voidaan käyttää abstrakteja jäseniä tai geneerisyyttä. On myös tärkeää, että kääntäjä pystyy takaamaan yhdistelmien tyyppioikeellisuuden käännösaikana.

Koodi 12 havainnollistaa sekoitekoostamista.

Koodi 12

```
1: abstract class Teksti{ def getText(x: String): String  }
2: class Huutaja extends Teksti{
3:     def getText(x: String) = x.toUpperCase }
4: trait Tulostaja extends Teksti{
5:     abstract override def getText(x: String) =
6:        Console.println(x); super.getText(x)
7: }
8: val x = new Huutaja with Tulostaja
9: val y = x.getText("ok"); Console.println(y)
```

Ensimmäisellä rivillä määritetään abstrakti luokka, jolla on metodi `getText` ilman toteutusta. Luokka `Huutaja` toteuttaa sen muuntamalla annetun merkkijonon isokirjaimiseksi. Piirre `Tulostaja` toteuteuttaa myös `getText`in, siten että ensin tulostetaan merkkijono ja sitten kutsutaan `super`in `getText`iä. Vaikka `Tulostaja`n yläluokka ei toteuta `getText`iä, tämä määritelmä on mahdollista `abstract override` -määreen ja luokkalinearisaation ansiosta. Rivillä kahdeksan muodostetaan yhdistelmätyyppi. `x`:ään sidotun olion tyypin linearisaatiossa luokka `Huutaja` on ennen `Tulostaja`a, joten Tulostajan `super.getText`-kutsu voidaan ohjata sen näennäisyläluokalle. Viimeinen rivi saa aikaan tulostukset `ok` ja `OK`.

2.3 Hahmonsovitus

2.3.1 Yleistä

Hahmonsovittaminen (engl. pattern matching) on tietyllä tavalla käänteinen operaatio luokan konstruoinnille: tutkitaan, miten oliot on muodostettu. Hahmonsovitus tehdään sovittamalla (engl. match) jotain oliota eri tapauksia (engl. case) vasten, ja olion rakenteen ja arvojen mukaan haarautetaan suoritusta. Scalassa siihen on kaksi seuraavaksi esiteltävää tapaa.

2.3.2 Tapauskohtainen luokka

Tapauskohtainen luokka (engl. case class) – lyhyemmin *tapausluokka* – on luokka, jota käytetään olioiden hahmonsovittamisessa. Tapausluokat mahdollistavat, että ainoastaan välttämätön informaatio paljastetaan ja pidetään modulaarisuusperiaatteesta mahdollisimman paljon kiinni. Tapausluokille on lisätty syntaktista sokeria normaaliin luokkaan nähden: Ne voidaan luoda ilman `new`-määrettä. Lisäksi arvoparametreille luodaan julkinen saantimetodi automaattisesti. Tapausluokalla voi olla myös runko ja piilotettuja jäseniä tarvittaessa. [7]

Standardikirjaston geneerinen `Option[T]`-tyyppi on funktionaalinen, turvallinen tapa mallintaa arvoa, jota ei välttämättä ole asetettu (vertaa `null`). [11] `Option[T]`:n perivät tapaukset `None` ja `Some(x)`, siten että `None` vastaa asettamatonta tilaa ja `Some(x)` asetettua, missä `x` on varsinainen arvo.

Koodi 13 sovittaa `x`:ää `Some`a ja `None`a vasten ja palauttaa neliöidyn luvun siinä tapauksessa, että varsinainen arvo oli määrätty.

Koodi 13

```
1: def nelioi(x: Option[Int]) = x match {
2:     case Some(arvo) => Some(arvo * arvo)
3:     case None => None
4: }
```

2.3.3 Erotin

Erotin (engl. extractor) mahdollistaa hahmonsovituksen, jossa olioiden informaatio voidaan piilottaa täysin. [11] Erotinluokka tai -objekti saadaan erottimeksi määrittelemällä `unapply`-metodi, joka saa parametrikseen tarkasteltavan olion. Jos olio on halutun hahmon mukainen, palautetaan joko `true` tai `Some(x)`, missä `x` vastaa oliosta erotettua tietoa. Muutoin palautetaan `false` tai `None` vastaavasti. [5] Muut vaihtoehdot sivuutetaan.

Usein `unapply`lla on käänteismetodi `apply`, jonka avulla muodostetaan tietyn hahmon mukainen olio. `apply`lla ei kuitenkaan ole mitään rajoituksia.

Koodi 14 luo `Positiiviset`-erottimen, jonka avulla saadaan konstruoitua sovitettavasta listasta uusi, sovitetun listan alkioiden neliöjuuret sisältävä lista.

Koodi 14

```
1: object Positiiviset{
2:   def unapply(x: List[Int]) = {
3:       if(x.forall(x => x > 0))
4:           Some(x.map(x => scala.Math.sqrt(x)))
5:       else None
6: } }
7: List(4, 16, 9) match {
8:    case Positiiviset(neliojuuret) => Console.println(neliojuuret)
9: }
```

2.4 Implisiittiset parametrit ja muunnokset

Funktiokutsun puuttuvat argumentit voidaan täydentää automaattisesti, jos parametrilista on merkitty *implisiittiseksi*. Merkintä tehdään lauseella `implicit` $p_1,...,p_n$ joka asettaa parametrit $p_1,...,p_n$ implisiittisiksi. Funktiolla tai muodostimella voi olla vain yksi implisiittinen parametrilista, ja sen täytyy olla viimeisenä. Eräät mahdolliset täydennyksessä käytettävät argumentit ovat ne, joiden tunnisteeseen päästään kiinni ilman määrittelyaluetta (engl. scope) laajentavan etuliitteen käyttöä. Lisäksi tunniste täytyy olla määritelty implisiittiseksi. Muut vaihtoehdot sivuutetaan. [5]

Implisiittiset metodit ja parametrit voivat määritellä implisiittisiä muunnoksia, joita kutsutaan *näkymiksi*. Näkymä `S`-tyypistä `T`:hen määräytyy funktiotyypistä `S => T` tai `(=>S)=>T` tai metodista, joka on muunnettavissa jompaan kumpaan edellä mainituista tyypeistä. Näkymää käytetään kahdessa tilanteessa (1 tai 2):

```
Oletetaan ensin, että lauseke e on tyyppiä T.
1. Jos e ei täytä odotetun tyypin T’ vaatimuksia. Tällöin
etsitään implisiittistä muunnosta T => T’.
2. Jos valinnassa e.m oleva jäsen m ei ole T:ssä. Tällöin
etsitään sellaista e:hen soveltuvaa näkymää, jota
käytettäessä saadaan m-niminen jäsen.[5]
```

Koodi 15 luo näkymän `Int`-tyypistä `Double`en. Rivillä kaksi tapahtuu käännösaikainen muunnos `5 -> intToDouble(5)`, jotta saadaan `x`:n vaatima tyyppi.

Koodi 15

```
1: implicit def intToDouble(x: Int): Double = x.toDouble
2: val x: Double = 5
```

Standardikirjaston Predef-luokan toteutuksesta löytyy ensimmäistä riviä vastaava näkymämäärittely. Scalassa numeeriset tyypit eivät siis ole periytymissuhteessa, vaan niiden välinen vuorovaikutus automatisoidaan implisiittisillä muunnoksilla.

2.5 Varianssi

Tyyppiä A sanotaan *kovariantiksi*, jos se säilyttää alityypityssuhteen järjestyksen, ts. seuraava ehto pätee: `S <: T => A[S] <: A[T]`. Jos järjestys kääntyy toisinpäin, on kyse *kontravariantista* tyypistä, ja jos kumpikaan ei päde, sanotaan tyyppiä *invariantiksi*. Scalan tyypit ovat oletuksena invarianttina. Tyypin määrittelyssä voidaan kuitenkin merkitä tyypin käyttö a) kovariantiksi +:lla b) kontravariantiksi –:lla. Varianssin oikeellisesta käytöstä huolehtii kääntäjä. Ohjelmoijan on silti oleellista tietää, miten varianssi suhtautuu tyypin käyttösijainnin mukaan. Seuraavat lienevät välttämättömimmät:

```
1) Muuttujan tyyppi on invariantissa paikassa
2) Funktion palautustyyppi ja luokan vakioviittausjäsen on
kovariantissa paikassa
3) Funktion arvoparametrin tyyppi on kontravariantissa
paikassa
```

[5]

Paikkoihin sitominen takaa tyyppien varianssikäyttäytymisen oikeellisuuden käännösaikana. Tätä voi verrata Javaan, jossa luokan käyttäjän kannalta joustavampi, *wildcardseja* käyttävä varianssijärjestelmä pakottaa tarkistamaan tyyppieheyden suorituksen aikana. [12]

Esimerkki

`class List[+a]` tarkoittaa, että `List` käsittelee tyyppiparametria `a` kovarianttina. Nyt voidaan luoda singleton-objekti `Nil` vastaamaan kaikkia nolla-alkioisia erityyppisiä `List`-luokkia: `object Nil extends List[Nothing]`. Tämä on mahdollista, koska kaikille tyypeille `S` on voimassa `Nothing <: S` ja koska `List` on kovariantti. Edellisistä seuraa: `List[Nothing] <: List[S]`. Siispä `Nil <: List[S]`.

3 Scalan ohjelmointiabstraktioiden hyödyntäminen

3.1 Funktionaalisen lähestymistavan edut

Ohjelmoinnissa on oleellista nähdä keskeiset abstraktiot, rakentaa uusia ohjelmanosia niiden varaan ja yleistää näitä edelleen. Tärkeää erityisesti on pystyä ajattelemaan ongelmaa abstraktioiden kautta. Korkeamman asteen funktioiden merkitys onkin siinä, että niitä voidaan käsitellä samalla tavalla kuin muita ohjelman elementtejä. [3] Niinpä silloin, kun määritetään uusi datatyyppi, pitäisi sen käsittelyä varten luoda korkeamman asteen funktio. Sen ansiosta datatyypin käsittely on helppoa ja yksityiskohtien tieto rajoittuu yhteen paikkaan. [13] Edellä mainitut asiat antavat kuitenkin vain vaillinaisen kuvan funktionaalisesta paradigmasta. Muita tärkeitä, osin päällekkäisiä piirteitä ovat mm. puhtaat funktiot, laiskat parametrit, currying, (häntä)rekursio ja hahmonsovitus, joiden hyödyllisyyttä käsitellään seuraavaksi.

Funktio on *puhdas*, jos funktion osien suoritusjärjestys ei muuta palautettavaa arvoa eikä suoritus aiheuta sivuvaikutuksia, ts. muuta tilaa. Puhtaat funktiot mahdollistavat ohjelman matemaattisen analysoinnin ja vähentävät virheiden määrää [13]. Lisäksi niistä koostetun ohjelman oikeellisuuden tarkistus on kompleksisuudeltaan osiensa summa eikä tulo, kuten olisi tilaa muuttavan ohjelman osalta. [14] Scalassa ei voi pakottaa puhtautta, mutta ainakin puhtaiden funktioiden määrittely on kokemukseni perusteella syntaktisesti yhtä vaivatonta kuin epäpuhtaiden.

Puhtaiden funktioiden yhteydessä käytetään usein laiskoja parametreja. Niiden tehokkainta muotoa, tarvekutsua (engl. call-by-need), ei ole tuettuna Scalassa, mutta sitä voidaan jäljitellä helposti nimikutsuparametrien avulla. Laiskat parametrit mahdollistavat mm. omien kontrolliabstraktioiden suoraviivaisen määrittelyn currying-menetelmän tukemana.

Koodi 15 näyttää, kuinka while-silmukka voitaisiin määritellä käyttämällä nimikutsuparametreja. `suoritettava` evaluoituu vain, jos `ehto` on `true`.

Koodi 15

```
1: def While (ehto: => boolean)(suoritettava: => Unit): Unit
2:     = if (ehto) { suoritettava ; While(ehto)(suoritettava) }
```

[7]

`While`a voitaisiin käyttää esimerkiksi seuraavasti:

```
1: var x = 5; While(x > 2){x = x - 1; Console.println(x)}
```

Lisäksi eräs hyvin oleellinen funktionaalisen ohjelmoinnin käsite on r*ekursio.* Rekursiivinen funktio on funktio, joka määritetään käyttämällä viittausta itseensä. *Häntärekursiivinen* funktio puolestaan on rekursiivinen funktio, jossa itseensä viittaaminen tapahtuu viimeisenä suorituspoluilla. Häntärekursion käyttö on käytännössä välttämätöntä, koska yleinen rekursio on funktiokutsujen tilavaatimukselta lineaarinen. Se aiheuttaa pinon täyttymisen kohtuullisen pienillä kutsumäärillä. Häntärekursiossa rekursiiviset funktiokutsut voidaan sijoittaa kunkin edeltävän kutsun tilalle, jolloin tilavaatimus on vakio. [14] Rekursio on voimakas abstraktio, koska hyvin moni matemaattinen tehtävä voidaan ilmaista rekursiivisesti. Niinpä funktionaalisessa ohjelmoinnissa on kätevää usein ratkaista ongelma määrittämällä ensin rekursiivinen datatyyppi. Tyypin käsittelyä varten tarvitaan tapa haarautua tyypin mukaan: siihen kelpaa hahmonsovitus varsin hyvin, koska se on ytimekästä ja selkeää. Sen ansiosta rekursiivisen käsittelyfunktion muodostaminen on suoraviivaista: se noudattaa käsiteltävän tyypin rakennetta. [14] Rekursio on siis lyhyt ja selkeä esitystapa muodostaa käyttökelpoisia ohjelmia, mistä havainnollisena esimerkkinä käy koodi 16.

Koodi 16

```
1: abstract class Lauseke
2: case class Summa(x: Lauseke, y: Lauseke) extends Lauseke
3: case class Arvo(x: Int) extends Lauseke
4: def evaluoi(x: Lauseke): Int = x match {
5:    case Summa(a, b) => evaluoi(a) + evaluoi(b)
6:    case Arvo(a) => a
7: }
8: val lauseke = Summa(Arvo(3), Summa(Arvo(4), Arvo(2)))
9: Console.println(evaluoi(lauseke))
```

Koodi määrittää abstraktin `Lausekeen` ja periyttää siitä kaksi eri tapausluokkaa, `Summa`n ja `Arvo`n. `Summa` koostuu kahdesta lausekkeesta kun taas `Arvo` kokonaisluvusta. Rivillä 4 `evaluoi` sovittaa `Lauseke`-tyyppistä oliota `x` tapauksia vasten ja hajoittaa tuloksenlaskennan rekursiivisesti. Lopuksi evaluoidaan esimerkkilauseke.

3.2 Oliosuuntautuneen lähestymistavan edut

Vaikka funktionaaliset piirteet ovat Scalassa tärkeitä, ne ovat tuttuja useista kielistä (esim. Lisp, Haskell) jo useilta vuosikymmeniltä. Vähän käytettyjä, uusiakin, oliotekniikoita sen sijaan on useita, ja erityisesti niiden yhdistäminen uniikilla tavalla on Scalassa mielenkiintoista tutkimuksen kannalta. Ne käsitellään seuraavassa, 3.3-luvussa. Sitä ennen esitän muutaman yleisen huomion olionäkökulmasta.

Olioperusteisuus on ihmiselle luontainen tapa hahmottaa ympäristöä ja mahdollistaa ohjelmistokehityksen, jossa ohjelmistosta saadaan samarakenteinen käsitemaailman kanssa. Etuna on mm. systemaattinen ohjelmiston rakennus. [15] Muita oleellisia syitä käyttää oliosuuntautunutta ohjelmistonkehitystä ovat laajennettavuus, uudelleenkäytettävyys ja yhteensopivuus. [6]

Olio-ohjelmoinnin tärkeimpiä käsitteitä ovat muuttuva tila, polymorfismi ja perintä. Niiden oikeellinen käyttö on kuitenkin vain osittain ymmärretty. [14] Niinpä olio-ohjelmoinnin voi käsittää usealla tavalla. Esimerkiksi vahvasti olioideologiaa noudattavassa Smalltalkissa tyypeillä ei sinänsä ole suurta merkitystä; sen sijaan olennaista on olioiden välinen kommunikointi dynaamisten viestien välityksellä. [16] Tehokkuusnäkökulmasta tehdyn moniparadigmaisen C++:n onnistuneisuutta voi puolestaan kuvata Alan Kayn sanoin (alkup. englannink.): ”Keksin termin olio-orientoitunut ja voin kertoa, että C++ ei ollut mitä ajattelin.” [16] Vaikka Scala pohjautuu useista paradigmoista otettuihin piirteisiin, sen tyyppiteoriaa noudattava toteutus ja sekoitekoostamisen yksiselitteisyys tukevat oliomallinnusta, pääosin luokkien konstruointia. Toisaalta, vaikka piirreluokkien käyttö antaa joustavuutta ja hienojakoistaa uudelleenkäyttöä, se hajoittaa rakennetta ja heikentää ongelma-alueen ja ohjelman samarakenteisuutta. [15]

3.3 Komponenttiperustainen lähestymistapa

3.3.1 Yleistä

Molemman edellä esitellyn paradigman tavoitteena on parantaa uudelleenkäytettävyyttä. Pyritään siis rakentamaan uusia ohjelmia aiemmin tehdyistä komponenteista. Sitä kuitenkin rajoittavat useimpien ohjelmointikielten puutteet, erityisesti kyvyttömyys

käsitellä vaadittuja rajapintoja korkealla tasolla . Scalan kolme keskeistä käsitettä, abstraktit tyyppijäsenet, itsetyyppimerkinnät ja modulaarinen sekoitekoostaminen, mahdollistavat komponettiperustaisen ohjelmoinnin. [2]

3.3.2 Abstraktioiden käyttö

Edellä esitellyt menetelmät muodostavat palvelurajapintoihin pohjautuvan komponenttimallin perustan, missä komponenteilla on hyvin määritellyt vaaditut ja tarjotut palvelut. Scalan mallissa luokat vastaavat komponentteja, konkreettiset luokkien jäsenet tarjottuja palveluja sekä abstraktit jäsenet vaadittuja palveluja. Komponenttien muodostaminen perustuu sekoitekoostamiseen, joka mahdollistaa suurempien osien koonnin pienemmistä. Palvelujen koostaminen pohjautuu niiden samannimisyyteeen. Niinpä abstrakti jäsen M voidaan toteuttaa yksinkertaisesti sekoittamalla mukaan piirreluokka, jossa on toteutettu M-niminen jäsen. Tämä menetelmä mahdollistaa rekursiivisesti kytkettävät komponentit ilman, että niiden välisiä kytköksiä tarvitsisi määrittää yksitellen parametrein. Se skaalautuu myös hyvin, vaikka komponenttien välillä olisi lukuisia vaadittuja ja tarjottuja rajapintoja, sillä kääntäjä päättelee yhdistykset automaattisesti. Tärkein etu kuitenkin on, että komponentit ovat laajennettavissa perimällä ja jäsenten korvauksilla. Lisäksi voidaan jopa lisätä uusia palveluita suljettuihin olemassa oleviin komponentteihin implisiittisten muunnosten ja parametrien avulla. Olemassa olevat palvelut voidaan myös päivittää. [2]

Abstraktit tyypit soveltuvat erityisen hyvin tilanteisiin, joissa usea tyyppi muuntuu yhdessä kovariantisti. Tätä on kutsuttu perhepolymorfismiksi (engl. family polymorphism). Seuraava Oderskyn ja Zengerin esittelemä, itseni suomentama, esimerkki (koodi 17) demonstroi, kuinka Scalassa voidaan käyttää Tarkkailija-suunnittelumallia (engl. Observer [8]) perhepolymorfisesti. [2]

Koodi 17

```
01: abstract class KohdeTarkkailija{
02:   type K <: Kohde
03:   type T <: Tarkkailija
```

Aluksi määritellään abstrakti luokka, jolla niputetaan `Kohde` (engl. Subject) ja `Tarkkailija` yhteen. Sitten määritetään abstraktit tyyppijäsenet `K` ja `T`, jotta ei sitouduta tiettyyn konkreettiseen tyyppiin.

```
04:   abstract class Kohde requires K {
```

Abstraktin `Kohde`-luokan itsetyypiksi sidotaan tyyppi `K`. Sen ansiosta voimme antaa `this`-viitteen sellaiselle metodille, joka ottaa parametrinaan `K`:n eikä `Kohde`:tta.

```
05:     private var tarkkailijat: List[T] = List()
06:     def rekisteroi(tarkkailija: T) =
07:         tarkkailijat = tarkkailija :: tarkkailijat
```

Määritetään `tarkkailijat`-lista, johon voidaan lisätä uusi tarkkailija `rekisteroi`-metodilla. Jälleen, `rekisteroi` ei sitoudu mihinkään konkreettiseen tyyppiin vaan käyttää abstraktia `T`:tä.

```
08:     def julkaise =
09:         for(val t <- tarkkailijat) t.ilmoita(this)
```

`Julkaise` käy läpi tarkkailijat ja kertoo niille, että muutos on tapahtunut antamalla muutoksen kohteen parametrina. Huomaa: tässä käytetään vaatimusta, että `this` on tyyppiä `K`.

```
10:   }
11:   abstract class Tarkkailija {
12:     def ilmoita(kohde: K): Unit
13:   } }
```

Tarkkailijalle esitellään vain metodi muutoksien ilmoittamista varten.

```
14: object SensoriLukija extends KohdeTarkkailija {
15:   type K = Sensori
16:   type T = Monitori
```

Luodaan konkreetti Kohde–Tarkkailija-pari, `SensoriLukija`, jossa monitoreilla tarkkaillaan sensorin tilaa. `KohdeTarkkailijassa` olevat abstraktit tyypit `K` ja `T` sidotaan `SensoriLukijassa` konkreettisiin, sensoria ja monitoria vastaaviin luokkiin.

```
17:   abstract class Sensori extends Kohde{
18:     val nimi: String
19:     var arvo = 0.0
20:     def muutaArvo(uusi: Double) = {
21:       arvo = uusi
22:       julkaise
23:   } }
```

`Sensori` on `Kohde`-luokan aliluokka. Sille määritetään nimi ja muuttuva arvo sekä metodi, joka muutoksen tapahtuessa julkaisee tiedon tarkkailijoille.

```
24:   class Monitori extends Tarkkailija{
25:     def ilmoita(s: Sensori) =
26:         Console.println(s.nimi + ": " + s.arvo)
27:   } }
```

`Monitori` puolestaan tulostaa konsoliin nimen ja arvon ilmoitetun muutoskohteen perusteella. [2]

Seuraavassa luodaan testitapaus, missä on kaksi sensoria ja kaksi monitoria. Molemmat monitorit rekisteröidään ensimmäiseen sensoriin, mutta vain ykkösmonitori toiseen sensoriin. Lopuksi muutetaan sensoreiden arvoa, jolloin tarkkailijat saavat viestin ja näytölle tulostuu sensoreiden nimet ja arvot.

```
28: object Testi{
29:   import SensoriLukija._
30:   val s1 = new Sensori{ val nimi = "sensori1" }
31:   val s2 = new Sensori{ val nimi = "sensori2" }
32:   val m1 = new Monitori; val m2 = new Monitori
33:   s1.rekisteroi(m1); s1.rekisteroi(m2); s2.rekisteroi(m1)
34:   s1.muutaArvo(5.3); s2.muutaArvo(2.6)
35: }
```

`SensoriLukija` voisi myös olla abstrakti, jolloin esimerkiksi abstrakteille tyypeille `K` ja `T` annettaisiin ylätyyppirajoite. Silloin `SensoriLukija`n perijät voisivat käyttää joko `Sensori`a tai sille määriteltyä alaluokkaa – vastaavasti `Monitori`lle. [2]

Toinen konkreettinen esimerkki esiteltyjen abstraktioiden hyödyllisyydestä on *scalac*, Scala-kielen kääntäjä, jonka toteutus on kirjoitettu Scalalla. Sen rakenne koostuu useista moduuleista, mm. symbolinimistä, symboleista, tyypeistä ja määrittelyistä. Lisäksi niiden välillä on lukuisia rekursiivisia viittauksia. Jos viittaukset toisiin moduuleihin määritettäisiin muuttuvilla, staattisilla viittauksilla, kääntäjää ei voisi käsitellä laajennettavana komponenttina. Kääntäjästä ei voisi myöskään luoda useaa samanaikaisesti toimivaa instanssia yhden virtuaalikoneen sisälle. Vastaava ongelma on ratkaistu Javan kääntäjässä konteksti-suunnittelumallilla, missä kääntäjä parametrisoidaan kuvauksella komponenttitunnisteista komponenttitoteutuksiin. Haittapuolena on joko käännösaikaisesti tyyppiturvaton ratkaisu tai tyyppiturvallinen versio, jossa ylimääräistä koodia on paljon ja joka on modulaarisuusperiaatteen vastainen. Lisäksi molemmat ovat ohjelmointimalleja, joiden oikeinkäyttöä ei voida taata kääntäjällä. [2]

Kokonaisuuden kannalta parempi ratkaisu on käyttää ohjelmointikieltä, joka tukee komponenttipohjaista ohjelmointia. Kuitenkin useimmissa kielissä on vähintään yksi puute, joka estää staattisesti kovakoodaamattoman ja laajennettavan komponentin muodostamisen. Scalan abstraktioilla ongelma kuitenkin voidaan ratkaista.

Ratkaisu saataisiin sisällyttämällä eri moduulit yhden luokan sisään, mutta tällöin kääntäjästä tulisi monoliittinen ja vaikeasti ylläpidettävä. Järkevä ratkaisu saadaan sekoitekoostamisen ja itsetyyppien avulla: ne mahdollistavat luokkien hajautuksen omiin tiedostoihinsa. Koodi 18 esittää tällaisen ratkaisumallin. [2]

Koodi 18

```
1: abstract class Tyypit requires (Tyypit with Nimet
2:                                 with Symbolit with Maaritelmat){
3:          class Tyyppi { ... }
4:  // Tyyppi-luokan alaluokat ja tyyppiin liittyvät operaatiot
5: }
```

Vastaavanlainen abstrakti luokka määritetään symboleille, määritelmille ja nimille.

```
6: class SymboliTaulu extends Nimet with Tyypit
7:                            with Symbolit with Maaritelmat
8: class ScalaKaantaja extends SymboliTaulu with Puut with ...
```

[2]

`SymboliTaulu` yhdistää kaikki ylläolevat luokat. `ScalaKaantaja` voi nyt periä `SymboliTaulu`n ja sen lisäksi muita osia, jotka eivät riipu rekursiivisesti toisistaan.

Itsetyyppimerkintä ja nimipohjainen sekoitekoostaminen mahdollistavat sellaisten osien ytimekkään muodostuksen, jotka riippuvat toisistaan rekursiivisesti. Ohjelmoijan ei tarvitse erikseen välittää parametrien avulla eri luokkien instansseja muodostusta varten, ja silti kääntäjä pystyy tarkistamaan komponentin konstruoinnin tyyppioikeellisuuden. Etuna on myös laajennettavuus: Oletetaan tilanne, jossa haluttaisiin muodostaa valmiin kääntäjän avulla uusi kääntäjä. Uusi kääntäjä toimisi muuten aivan kuin valmis kääntäjä, mutta se kirjoittaisi jokaisen uuden symbolin muodostuksen konsoliin. Koodi 19 osoittaa, kuinka se tehdään. [2]

Koodi 19

```
1: abstract class TulostettavatSymbolit extends Symbolit {
2:   override def uusiTermiSymboli(nimi: Nimi): TermiSymboli {
3:      val x = super.uusiTermiSymboli(nimi)
4:      Console.println(”Luodaan uusi symboli: ” + x)
5:      x
6:   }
7:   // korvataan metodi samoin muille symbolien luomisille
8: }
9: class TulostavaKaantaja extends ScalaKaantaja
                           with TulostettavatSymbolit
```

Symboleiden luontimetodit korvataan uudella metodilla, joka kutsuu `super`in vastaavaa metodia, ottaa arvon talteen, tulostaa ensin ja palauttaa sitten arvon. Uuden

käyttäytymisen lisääminen onnistuu suoraviivaisella sekoitekoostamisella, koska `ScalaKaantaja` on määritelty yhtenä luokkana (rivi 9). Huomaa, että ei tarvittu muuttaa ollenkaan valmiiksi muodostettua kääntäjää. Staattisten komponenttien tai kovakoodattujen viittausten olemassaolo olisi estänyt tämän. Toinen vaihtoehto olisi käyttää aspektipohjaista ohjelmointia (engl. AOP, Aspect-Oriented Programming), mikä perustuu koodin uudelleenkirjoitukseen. [2]

Scalassa voidaan perinnän lisäksi laajentaa olemassaolevia, suljettuja komponentteja. Se onnistuu implisiittisen parametrien ja muunnosten avulla. Jos haluttaisiin esim. että lauseke `3 * ”Moi”` antaisi arvon `MoiMoiMoi`, niin normaalisti täytyisi määritellä `*`-metodi `Int`-luokalle, siten että se ottaa parametrina `String`-tyypin. `Int` on kuitenkin suljettu; siksi joudutaan käyttämään näkymää. Toimiva esimerkki siitä on koodi 20:

Koodi 20

```
1: class Monistus(kertaa: Int){
2:     private def monista(x: Int, y: String): String =
3:         if(x > 0) y + monista(x - 1, y) else ""
4:     def *(x: String) = monista(kertaa, x)
5: }
6: implicit def intToMonistus(x: Int): Monistus = new Monistus(x)
7: Console.println(3 * "Moi")
```

Rivillä kuusi määritetään implisiittinen muunnos `Int`-tyypistä `Monistus`-tyypiksi, jossa puolestaan on tarvittava `*`-metodi. Varsinaista merkkijonomanipulaatiota varten määritetään apumetodi `monista`. Rivin seitsemän `3 * ”Moi”` -lauseke evaluoidaan seuraavasti:

`3 * ”Moi” -> intToMonistus(3) * ”Moi” -> new Monistus(3) * ”Moi” ->* ”Moi” + monista(2, ”Moi”) ->* ”MoiMoiMoi”.`

3.4 Vertailua muissa kielissä oleviin abstraktioihin

3.4.1 Sekoitekoostaminen vs. klassinen moniperintä

Usean luokan piirteiden perimistä yhteen luokkaan, eli *moniperintää*, pidetään usein tarpeettomana ja ongelmiensa takia ei-toivottuna kielen mekanismina. Kuitenkin tässäkin tutkielmassa on esitetty tilanteita, missä sitä tarvitaan. Seuraavaksi osoitan, kuinka perinteiset moniperintään liittyvät ongelmat on ratkaistu sekoitekoostamisessa.

Koskimiehen mukaan suurin ongelma on luokkien periytymissuhteen radikaali monimutkaistuminen: periytymissuhteita ei voi kuvata selkällä hierarkisella puurakenteella vaan tarvitaan verkkorakenne. [15] Vaikka Scalassa yhdistelmätyyppien takia rakentuu luokkien välille verkkomuoto, luokkien *käyttäytyminen* määräytyy lineaarisesta mallista: luokkalinearisaatio antaa yksikäsitteisen järjestyksen luokille tietyn luokan suhteen. Samalla se ratkaisee toisen ongelman, nimittäin nimikonfliktit: kun samasignatuurinen jäsen periytyy useammasta luokasta, ongelmana on päättää, mikä niistä valitaan perivälle luokalle. [15] Tähän esitettiin oletuskorvautuvuus luvussa 2.2.5. Joskus nimeen perustava automaattinen korvautuvuus ei ole kuitenkaan toivottua. Tällöin perivässä luokassa olevassa metodissa `f` on mahdollista viitata useassa eri yläluokassa esiintyvään samannimiseen `f`-metodiin antamalla kutsussa yläluokan nimi muodossa `super[YlaLuokka].f`. Lisäksi, koska piirreluokkien muodostimille ei voida antaa arvoparametreja, uusien piirteiden lisääminen luokkaan on suoraviivaista. Virheellisiä korvauksia ei ole myöskään helppoa tehdä pakollisen `override`-määreen ansiosta.

3.4.2 Hahmontunnistus vs. Vierailija-suunnittelumalli ja instanceof-tyyppitestaus

Useimmissa oliokielissä oliohierarkian läpikäyminen ulkopuolelta on hankalaa: joudutaan joko käyttämään dynaamista tyyppitestiä tai Vierailija-suunnittelumallia (engl. Visitor [8]). Scalassa tätä ongelmaa varten on olemassa hahmontunnistus joko tapausluokkien tai erottimien avulla. Emir, Odersky ja Williams ovat tutkineet, miten nämä eri tapaukset eroavat seuraavien kriteereiden osalta: koodin ytimekkyys, tietojen paljastus, luokkahierarkian ja hahmojen laajennettavuus, tehokkuus ja skaalautuvuus. [11] Seuraavassa muutamia huomioita tuloksista.

1) Vierailijalla on suhteellisen korkea merkinnällinen lisätaakka. Lisäksi se riippuu tiedon esitysmuodosta eikä ole helposti laajennettava.
2) Tyyppitestaukset ovat nopeita, mutta niiden käyttö vaatii runsaasti koodia. Lisäksi ne paljastavat täysin esitysmuodon.
3) Tapausluokat eivät vaadi luokkahierarkian muodostusta varten ylimääräistä työtä. Myös tunnistus on ytimekästä. Uusien hahmontunnistusmetodien lisäys on helppoa, mutta uusien tapausten ei, sillä hahmot ovat tapausluokkien kanssa yksi–yhteen-

suhteessa. Erottimien ja tapausluokkien yhteiskäyttö kuitenkin ratkaisee ongelman. Lisäksi Javan virtuaalikoneen HotSpot optimoi tapausluokkien käytön lähes yhtä tehokkaaksi kuin tyyppitestauksien.

4) Erottimilla voidaan piilottaa täysin tiedon esitysmuoto. Niiden muodostaminen ei ole kuitenkaan ytimekästä, sillä ne määritetään yhdessä luokan kanssa. Hahmontunnistus on puolestaan yhtä suoraviivaista kuin tapausluokkien. Erottimia on myös helppo laajentaa. Joustavuuden hintana on suorituskyvyn puute: ne ovat noin kaksi kertaa muita tehottomampia. [11]

4 Pohdinta

Tässä tutkielmassa olen selvittänyt, kuinka ohjelmointikielten puutteet rajoittavat tehokasta komponenttipohjaista ohjelmointia. Itsenäisen, helposti laajennettavan komponentin rakentamisen estävät erityisesti klassisen moniperinnän ongelmat tai sen täydellinen puuttuminen sekä vaadittujen rajapintojen abstrahoinnin mahdottomuus. Kolme esiteltyä lähestymistapaa, funktionaalinen, olio-orientoitunut ja komponenttipohjainen, ovat merkittäviä ongelman kannalta. Koska Scala tukee niistä kaikkia, voidaan ongelman vaatimusten mukaan valita sopivat menetelmät ja rakentaa uudelleenkäytettäviä ohjelmia, vieläpä ytimekkäästi. Sen osoittamisessa olen luvussa 3.3.2 käyttänyt apuna esimerkkejä, jotka ovat viimeistä lukuun ottamatta alun perin Oderskyn ja Zengerin esittelemiä.

Keskeiset Scalan abstraktiot ovat sekoitekoostaminen, itsetyyppimerkinnät ja abstraktit tyyppijäsenet. Tyyppijäsenien avulla voidaan käsitellä komponentin vaadittuja rajapintoja abstraktisti; itsetyyppimerkintä mahdollistaa ytimekkään, toisiinsa rekursiivisia viittauksia sisältävien moduulien yhdistämisen sekoitekoostamisen tukemana. Luokkalinearisaatio tekee sekoitekoostamisesta hyvin määritellyn moniperinnän muodon, minkä ansiosta komponenttien laajentaminen on suoraviivaista.

Lisäksi implisiittiset parametrit ja muunnokset antavat joustavuutta sekä mahdollistavat suljettujen komponenttien laajentamisen. Myös varianssi ja tyyppirajoitteet edesauttavat luokkakonstruktioiden hienojakoistamista. Hahmonsovitus ja korkeamman asteen funktiot ovat puolestaan luonnollinen tapa ratkaista rekursiivisia ongelmia.

Scalan tyyppikäsitteitä olisi hedelmällistä käsitellä tarkemmin. Scalan sähköpostilistalla käydyistä keskusteluista on myös ilmennyt, että tutkimuksen kohteena on yhä korkeamman abstraktiotason konstruktioiden ja refleksiivisyyden lisääminen kieleen.

Viitteet

[1] Koskimies, K. & Mikkonen, T. (2005). *Ohjelmistoarkkitehtuurit*. Talentum Media Oy. Helsinki.

[2] Odersky, M. & Zenger, M. (2005). Scalable Component Abstractions. Proceedings of OOPSLA 2005, San Diego. http://lamp.epfl.ch/~odersky/papers/ScalableComponent.pdf (tarkistettu 31.3.2007).

[3] Abelsson, H. & Sussman, G.J. (1985). *Structure and Interpretation of Computer Programs*. The Massachusetts Institute of Technology.

[4] Odersky M., Cremet, V., Röckl, C. & Zenger, M. (2003). A Nominal Theory of Objects with Dependent Types. Proceedings of ECOOP 2003. http://lamp.epfl.ch/~odersky/papers/ecoop03.pdf (tarkistettu 31.3.2007).

[5] Odersky, M. The Scala Language Specification, version 2.4. Programming Methods Laboratory, EPFL, Switzerland. http://www.scala-lang.org/docu/files/ScalaReference.pdf (tarkistettu 31.3.2007).

[6] Meyer, B. (1997). *Object-Oriented Software Construction*. Second Edition. Prentice Hall, New Jersey.

[7] Odersky, M. Scala by Example. Programming Methods Laboratory, EPFL, Switzerland. http://www.scala-lang.org/docu/files/ScalaByExample.pdf (tarkistettu 31.3.2007).

[8] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2001). *Olio-ohjelmointi – Suunnittelumallit*. Oy Edita Ab. IT Press. Helsinki.

[9] Schinz, M. (2007). A Scala Tutorial for Java programmers. Programming Methods Laboratory, EPFL. http://www.scala-lang.org/docu/files/ScalaTutorial.pdf (tarkistettu 30.3.2007).

[10] Odersky & al. (2006). An Overview of the Scala Programming Language. Second edition [Tekninen raportti]. Programming Methods Laboratory, EPFL, Switzerland. http://www.scala-lang.org/docu/files/ScalaOverview.pdf (tarkistettu 31.3.2007).

[11] Emir B., Odersky, M. & Williams, J. (2007). Matching Objects with Patterns [Tekninen raportti]. Programming Methods Laboratory, EPFL, Switzerland. http://www.scala-lang.org/docu/files/MatchingObjectsWithPatterns-TR.pdf (tarkistettu 31.3.2007).

[12] Bracha, G. (2004). Generics in the Java Programming Language. Sun Microsystems. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf (tarkistettu 30.3.2007).

[13] Hughes, J. (1989). Why Functional Programming Matters. *Computer Journal* vol. 32, no 2: pages 98–107.

[14] vanRoy, P.V. & Haridi, S. (2004) – *Concepts, Techniques and Models of Computer Programming*. The MIT Press. Cambridge, Massachusetts.

[15] Koskimies, K. (2000). *Oliokirja*. Satku – Kauppakaari. Helsinki.

[16] Kay, A. (1997) The Computer Revolution hasn't happened yet [Konferenssipuhe]. Keynote OOPSLA 1997.