Harri Linna

Suunnittelumallien vertailu

Tietotekniikan kandidaatintutkielma

22. joulukuuta 2018

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Harri Linna

Yhteystiedot: harri.s.linna@student.jyu.fi

Ohjaaja: Antti-Juhani Kaijanaho

Työn nimi: Suunnittelumallien vertailu

Title in English: Comparison of design patterns

Työ: Kandidaatintutkielma

Sivumäärä: 27+4

Tiivistelmä: Suunnittelumallien vertailulla on tarkoitus tutkia ratkaisevatko suunnittelumallit samankaltaisen suunnitteluongelman kontekstista riippumatta. Kirjallisuuskatsaus perustuu pääosin Erich Gamman (1995) mallikokoelmaan (Elements of reusable object-oriented software) ja Tommi Mikkosen (1998) artikkeliin (Formalizing design patterns), jossa sovelletaan DisCoa suunnittelumallien formalisointiin. Vertailemalla kahta eri konstruktiota päätteltiin, että jotkin suunnittelumallit ratkaisevat samankaltaisen suunnitteluongelman.

Avainsanat: suunnittelumalli, oliosuunnittelu, formalisointi, DisCo

Abstract: Comparison of design patterns is meant to argue about whether design patterns resolve a similar design problem irrespective of context. Literature review for the most part is based on Erich Gamma's (1995) range of patterns (Elements of reusable object-oriented software) and Tommi Mikkonen's (1998) article (Formalizing design patterns) which applies the DisCo languange to formalization of design patterns. By comparing two separate constructs, was reasoned that some design patterns resolve a similar desing problem.

Keywords: design pattern, object-oriented design, formalization, DisCo

Kuviot

Taulukot

Sisältö

1 Johdanto

Suunnittelumallit ovat ratkaisumalleja, jotka ratkaisevat tietyn suunnitteluongelman (Michelucci 2013, s. 279–280). Käsitteen vakiinnutti arkkitehti Christopher Alexander kirjassaan The Timeless Way of Building, jossa mallit esittävät kolmiosaisena sääntönä kontekstin, ongelman ja ratkaisun välistä yhteyttä (Buschmann ym. 1996).

Suunnittelumallien ryhmittelyyn on useita luokittelutapoja, joista Erich Gamman, Richard Helmin, Ralph Johnsonin ja John Vlissidesin esitys vuodelta 1993 on eräs tunnetuimmista ja vaikutusvaltaisimmista. Siinä suunnittelumallit jaetaan rakennemalleihin ja käytösmalleihin, mitkä yleisesti pelkistyvät arkkitehtonisen rakenteen tasolle taikka abstrahoituvat algoritmisen käyttäytymisen tasolle (Gamma ym. 1993). Arkkitehtuuri kuvaa ohjelmiston komponenttien väliset rakenteet, kun vuorostaan algoritmi on abstraktio aliohjelmasta, joka toteuttaa tietyn ohjelmointitehtävän. Suunnittelumallit sijoittuvat abstraktiotasolla algoritmien ja arkkitehtuurien väliin. (Michelucci 2013; Buschmann ym. 1996)

Kirjallisuuskatsauksena tämän tutkielman tavoitteena on vastata seuraavaan tutkimusongelmaan: Ratkaisevatko suunnittellumallit samankaltaisen suunnitteluongelman kontekstista riippumatta? Toisin sanoen vaikuttaako suunnittelumallin valintaan ratkaisevasti se, missä kontekstissa ratkaistava ongelma esiintyy. Tutkimusongelman kannalta olennaisinta on vertailla malliluokkia sekä yksittäisiä suunnittelumalleja toisiinsa, mikä tehdään luvussa 2. Hannu-Matti Järvisen ja Reino Kurki-Suonion kehittämä, ja sittemmin Tommi Mikkosen soveltama DisCo-spesifikaatiokieli esitellään puolestaan luvussa 3 suunnittelumallien formalisointimenetelmänä. Formalisoituja suunnittelumalleja vertaillaan vuorostaan luvussa 4.

2 Suunnittelumallien vertailua

Suunnittelumalleja vertaillaan ensin luokittain, joista edetään luokkien väliseen vertailuun. Gamma (1995) jakaa suunnittelumallit kahteen luokkaan: Staattisia luokkahierarkioita strukturoiviin rakennemalleihin ja dynaamisia ajoaikaista käyttäytymistä kuvaaviin käytösmalleihin.

Rakennemalliluokan suunnittelumallit perustuvat ajatukseen, että mallien kuvaamat luokkahierarkiat erotetaan rajapintojen taakse, jotka määrittävät luokkahierarkioiden välisen kommunikaation. Tästä syystä ajoaikaisen käyttäytymisen, kuvaamiseksi tarvitaan rakennekuvaus taustalle, jotta luokkahierarkian oliot voisivat kommunikoida keskenään. Näin ollen rakenteellista kuvausta käytöksellä laajentava käytösmalli esittelee poikkeuksetta yhden tai useamman rakennemallin.

Luokkajaosta seuraa edellisen esimerkin kaltaisia suunnittelumalleja yhdistäviä ja erottavia ominaisuuksia. Toisaalta luokittelutavasta riippuvaiset ominaisuudet menettävät validiutensa muutettaessa luokittelutapaa, mikä tarkoittaa ettei esimerkin väittämä välttämättä yleisty muille suunnittelumalleille kuin Gamman (1995) esittämille malleille.

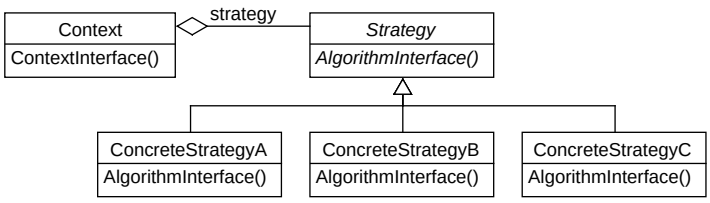
2.1 Rakennemallit

Suunnittelumallien nimeäminen rakennemalleiksi palvelee ennen kaikkea mallien dokumentointia ja myöhemmin mallien löytämistä hakemistolistauksista. Keskeistä ei ole niinkään määritellä tarkasti luokkaa itseään, vaikka sekin hyödyttäisi edellä mainittua tavoitetta. Rakennemalleissa on kuitenkin havaittavissa tiettyjä yhdistäviä ominaisuuksia, jotka ovat johtaneet kyseiseen luokitteluun. Esimerkiksi Gamma ja Toivonen (2001, s. 137) toteavat ”[rakennemallit] eivät yhdistele rajapintoja tai toteutuksia, vaan luovat uutta toiminnallisuutta olioita koostamalla.”

Yhtäältä rakennemallit laajentavat toiminnallisuuttaan ulkoistamalla sen toteutuksen koosteoliolle. Tätä kutsutaan delegoinniksi. Rakennemallien yhteydessä delegointi tapahtuu osasuhteena koosteessa, jota kuvataan Unified Modeling Language (UML) -notaatiossa avoi-

mella salmiakki-kuviolla (◇). Esimerkiksi aliohjelmakutsun vastaanottaja siirtää implementoinnin vastuun jäsenmuuttujalleen. Toiminnallisuus perustuu siten koosteisuuteen.

Toisaalta ulkoistetun toiminnallisuuden toteuttajan elinkaari on riippuvainen koostajastaan, joten hieman yllättäen rajapintoja ja toteutuksia yhdistellään. Olkoot perusteina kirjallisuuden luokittelutavat, joista Forsell (1998) mainitsee Gamman (1995) luokittelun lisäksi Zimmerin (1995) ja Buschmannin (1996) luokittelut. Näissä strategia [1] ja silta [2] luokitellaan samoin siitä huolimatta, että strategia kuuluu Gamman (1995) mukaan käytösmallien luokkaan.

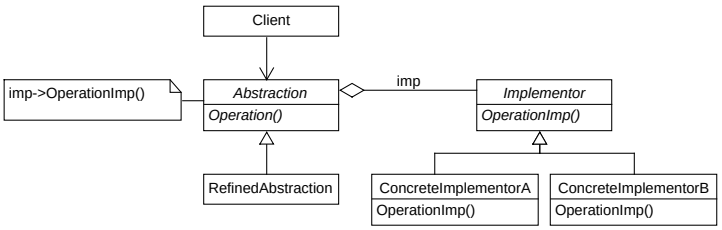


Kuvio 1. Strategia (Gamma 1995, s. 316)

Gamma (1995) luokittelee strategian käytösmalliksi, koska siinä esiintyy rajapintojen yhdistelyä. Toisaalta siihen ei liity ajoaikaista käyttäytymistä, mikä on käytösmalleissa tyypillistä. Nimittäin Gamman (1995, s. 139–218, s. 315–324) esityksestä päätellen rakennemallien kuvaamiseksi riittäisi staattinen luokkakaavio. Myös kuvio 1 riittää kuvata luokkakaaviolla, mutta siitä puuttuu vaadittua dynaamisuutta kuuluakseen käytösmalleihin. Vaikuttaisikin siltä, että strategia voi rajapintojen yhdistelystä huolimatta olla rakennemalli.

Strategian tehtävä on ulkoistaa algoritmin toteutus koosteoliolle, jolloin algoritmia voidaan vaihtaa ajoaikana. Kapseloinnin kannalta ei ole juuri merkitystä koostetaanko algoritmin toteutus rajapinnassa (Strategy) vaiko sen aliluokassa (ConcreteStrategy), koska hyvän ohjelmointitavan mukaisesti jäsenmuuttujat (strategy) julistetaan yksityisiksi. Tällöin rajapinnan (ContextInterface) käyttäjä (Context) ei ole tietoinen koosteoliosta (strategy). Siten kuviot 1 ja 2 eroavat lähinnä kontekstiltaan (Context/Client), sillä rakenteellinen delegointi vastaa algoritmillista kapselointia.

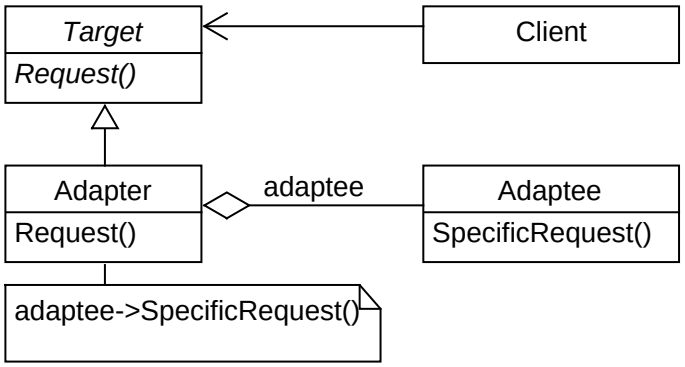
Gamman (1995, s. 20) mukaan kannattaa suosia olioiden koostamista periyttämisen sijas-



Kuvio 2. Silta (Gamma 1995, s. 153)

ta. Koosteisuus on eräänlaista dynaamisuutta, mikä mahdollistaa toimintojen delegoinnin, kapseloinnin sekä varioinnin ajoaikana. Toisin sanoen koosteisuus mahdollistaa varioitavan toiminnallisuuden kapseloinnin. Kapseloinnissa olion tilaa voi muuttaa vain olio itse. Siinä missä strategia edustaa algoritmia, silta edustaa komponenttia.

Puhdas rajapinta koostuu vain funktioista, mistä seuraa että rajapinnat eivät voi suoraan liittyä toisiinsa koosteella. Näin ollen rajapinta (Abstraction) esittää abstraktia luokkaa, eikä siitä voida luoda ilmentymää. Sen sijaan perintä mahdollistaa rajapinnan koosteen siirtämisen aliluokkiin, joten koosteen (imp) määrittäminen rajapintaan (Abstraction) vastaa koosteen (imp) määrittämisestä luokkaan (RefinedAbstraction). Näin ollen kooste (imp) voisi yhtäpitävästi liittyä abstraktin luokan (Abstraction) asemesta konkreettiseen luokkaan (RefinedAbstraction). Tämän jälkeen silta vastaa kuvion 3 sovitinta.



Kuvio 3. Sovitin (Gamma 1995, s. 141)

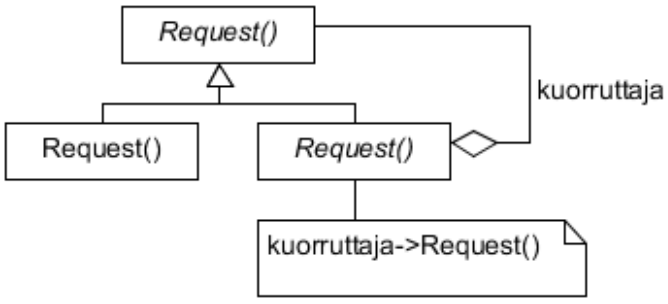
Sovitinta käytetään Gamman (1995, s. 161) mukaan yleensä sen jälkeen, kun järjestelmä on jo suunniteltu. Siltaa sen sijaan käytetään heti suunnittelun alkuvaiheessa, jolloin kehittäjällä

on suurempi kontrolli rajapintojen määrittelyyn. Toisin sanoen siltaa käytettäessä rajapinnat liitetään toisiinsa jo ohjelmiston suunnitteluvaiheessa.

Vastaavasti sovittimessa yhdistetään rajapinta (Target) olemassa olevaan komponenttiin (Adaptee) sovittimen (Adapter) välityksellä. Sovittimessa asiakas (Client) on tietämätön rajapinnan (Target) toteuttavan sovittimen (Adapter) yksityisestä jäsenmuuttujasta (adaptee), joka sovittaa pyynnöt (Request) luokan (Adaptee) mukaisiksi pyynnöiksi (SpecificRequest).

Kuviot 2 ja 3 ovat siten melko samanlaisia. Näistä kannattaisi käyttää ennemmin siltaa, koska oliosuunnittelussa suositaan periaatetta ”Ohjelmoi rajapintaa vasten, älä toteutusta vasten” (Gamma 1995, s. 18). Sovittimen käyttö on kuitenkin perusteltua, jos liitetään yhteen jo olemassa olevia luokkia, joiden rajapinnat ovat yhteensopimattomat. Tällöin rajapintaan tehtävät muutokset vaikuttavat ainoastaan sovittimeen. Sillan käyttö aiheuttaisi muutoksia rajapintoihin sillan molemmin puolin.

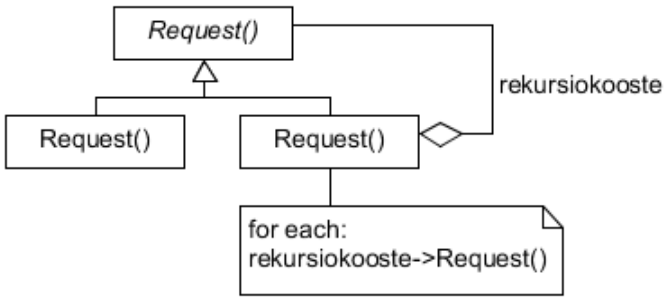
Silta ei ole suinkaan ainut suunnittelumalli, joka muistuttaa sovitinta. Kuorruttaja [4] on erikoistapaus sovittimesta, jossa sovittimen (Adapter) jäsenmuuttuja (adaptee) osoittaa sovittimen (Adapter) itsensä rajapintaan (Target) muodostaen rekursiivisen rakenteen. Kuorruttaja on kuvattu kuviossa 4.



Kuvio 4. Kuorruttaja (Gamma 1995, s. 177)

Kuorruttaja vastaa pitkälti rekursiokoostetta, mutta koosteen kardinaalisuhteeseen liittyy vain yksi olioviite. Rekursiokoosteen nimi viittaa oliorakenteen rekursiiviseen määrittelyyn, jonka tarkoituksena on koostaa joukko rajapinnan toteuttavia luokkia. Tästä johtuu kuvion 5 pyynnön (Request) silmukkarakenne, joka kuvaa rajapinnan toteuttavien luokkien joukon

(rekursiokooste) läpikäymistä iteratiivisesti. Mainittakoon että kyseinen silmukkarakenne (for each) on toteutusriippuvainen yksityiskohta, jonka käyttöä ei mallin sovellukselta edellytetä.



Kuvio 5. Rekursiokooste (Gamma 1995, s. 164)

Rekursiokoosteen yhteydessä esiintyy vääjäämättä perintää, joka staattisen luonteensa vuoksi rikkoo kapseloinnin (Snyder 1986, s. 38–45). Perinnän yhteydessä toiminnon toteuttavat aliluokat kiinnitetään jo käännösaikana, jolloin muutokset yliluokkiin vaikuttavat kaikkiin aliluokkiin. Siispä perintä kannattaa tehdä siten, että konkreettiset luokat periytetään abstrakteista kantaluokista. Tällöin luokkahierarkia säilyy matalana ja hallittavana. Kuorruttaja ja rekursiokooste tekevät juuri tämän.

Edellä kuvatut rakennemallit käyttäytyvät ajoaikana oleellisesti samalla tavalla ja ovat varioitavissa toisistaan vain pienin muutoksin. Variointipisteet ovat merkittäviä jatkotutkimuskohteita, koska niillä erotetaan rakenteellisesti yhtenevät, mutta tarkoitukseltaan eroavat rakennemallit toisistaan.

2.2 Käytösmallit

Gamman ja Toivosen (2001, s. 221) mukaan käytösmallit kuvaavat suunnitteluratkaisuja olioiden väliseen vuorovaikutukseen liittyen. Suunnittelussa kontrollin kulun hallintaa tärkeämpää olisi keskittyä siihen, miten oliot liitetään toisiinsa. Käytösmallit edustavatkin eräänlaista rajapintasuunnittelua, jonka kontekstissa esiintyvät suunnitteluongelmat liittynevät olioiden välisiin käyttösuhteisiin.

Ensimmäiseksi käytösmallit sisältävät kuvauksen paitsi vuorovaikutuksesta myös olioita ja luokkia koskevista malleista. Tällä ei suoraan tarkoiteta, että käytösmalli aina sisältäisi rakennemallin, mutta vuorovaikutuksen taustalla on aina vuorovaikuttavia olioita ja luokkia. Toisin sanoen käytösmalleihin voidaan olettaa liittyvän jonkin asteinen rakennekuvaus.

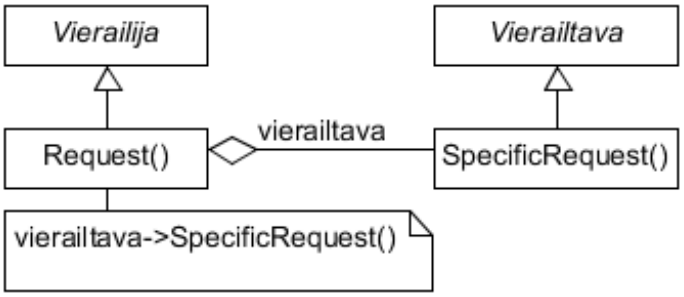
Toiseksi käytösmallit kuvaavat vuorovaikutuksen lisäksi tavan, jolla rakennekuvauksen oliot ja luokat liitetään toisiinsa. Tässä liittäminen tapahtuu ajoaikana eli käytösmallien kuvauksessa tulee huomioida alustustoimenpiteet osallistuvien luokka-instanssien osalta.

Kolmanneksi alustustoimenpiteiden jälkeen ei tule keskittyä kontrollinkulun hallintaan, josta vastaa itseasiassa taustalla oleva rakennemalli. Voidaan olettaa että käytösmallien osaksi kuuluu rakennekuvauksen alustaminen, joka luonnollisesti suoritetaan aivan aluksi. Käytösmallit voidaan siten ajatella koostuvan kahdesta osasta: Ensinnä usein yhden tai useamman rakennemallin sisältävän rakennekuvauksen alustamisesta. Toisekseen instantioitujen luokkahierarkioiden välisen kommunikaation kuvaamisesta.

Siinä missä rakennemallit käyttävät kontrollinohjauksessa osasuhdetta, käytösmallit hyödyntävät käyttösuhteita. Toisin sanoen rajapinnat tarjoavat keinot kontrollin ohjaukseen, joka toteutuu esimerkikisi parametrin välityksessä. Keskeinen ero on siinä, että toistensa kanssa keskenään vuorovaikutuksessa olevilla olioilla ei välttämättä ole toisistaan riippuvat elinkaaret, kuten osasuhteessa. Käyttösuhteessa koosteisuutta ei edellytetä, joten olioiden välinen assosiaatio on löyhempi.

Käytösmalleja yhdistää se, että pyynnöt esitetään kohteilleen epäsuorasti. Mallien käyttöön liittyy alustustoimenpiteitä, mikä johtuu siitä, että käytösmallit ovat luonteeltaan dynaamisia. Rakenteiden väliset käyttösuhteet luodaan vasta ajoaikana, mutta alustustoimenpiteiden jälkeen, lopputulos muistuttaa erehdyttävästi rakennemallien toimintaa. Käytösmallien voidaan tulkita olevan rakennemallien alustustoimenpiteitä ajoaikana, eikä tällainen tulkinta ole kovin kaukana koko totuudesta, mikä on perusteltavissa, sillä että käytösmallien luokkakaaviot näyttävät olevan rakennekaavioita. Varsinainen tapahtumasekvenssikaavioilla kuvattava vuorovaikutus, liittyy alustustoimenpiteisiin.

Vierailijan dynaamisuus perustuu parametrin välitykseen. Vierailtavalle oliolle (Visitable) välitetään parametrina vierailija (Visitor), joka kohdistaa operaationsa vierailtavaan (Visi-



Kuvio 6. Vierailija

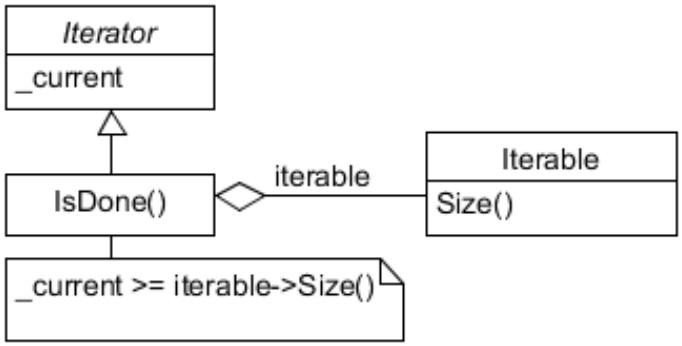
table). Liikkuvia osia ovat niin sanotut vieras (Visitor) ja isäntä (Visitable), jotka yhdessä määrittävät toiminnallisuuden. Vieras kiinnitetään parametrin välityksen yhteydessä ja se pyytää isäntää (SpecificRequest) käsittelemään pyynnön (Request), mikä määräytyy isännän tyypin perusteella. Lopullinen suorituspolku tiedetään, kun kontrolli etenee rajapinnasta vierailijaan (Visitor). Tässä oletetaan, että kieli on vahvasti tyypitetty, jolloin metodien kuormittaminen tyypin perusteella on mahdollista.

Niin ikään vierailijan kiinnitys tapahtuu mallissa parametrin välityksellä, koska vierailtavan ei tarvitse eikä pidä olla tietoinen mahdollisista vierailijoistaan. Vierailtava olio voitaisiin kuitenkin sisällyttää vierailijaan, eikä ohjelman toiminta oleellisesti muuttuisi. Syynä on se, että vierailija kiinnitetään rajapinnan näkökulmasta ensiksi, koska vierailija on oltava tiedossa ennen kuin metodikutsu voidaan kohdistaa vierailtavaan. Vierailija on kiinnitetty, joten vierailija voidaan alustaa vierailtavalla, jolloin saadaan kuvion 6 mukainen luokkakaavio.

Gamman (1995, s. 335) esityksessä rajapinnan (Interface) näkökulmasta metodin (Accept) kutsuminen vastaa pyyntöä (Request) vierailijalle (Visitor), koska metodin (Accept) toteutus välittää vierailtavan olion vierailijalle pyynnön parametrissa (Visitor.Request(Visitable)), mikä voidaan kaaviossa tulkita vierailijan koosteolion alustamiseksi vierailtavalla oliolla. Lopputuloksena saadaan sovitinta [3] vastaava luokkakaavio.

Suunnittelumallin todellinen kontribuutio on siinä, että koosteen sijaan käytettiin parametrivälityksessä viitettä olioon itseensä, joka on vierailijan näkökulmasta todellinen varioinnin kohde. Kontrollin ohjaus on varsin kekseliäs, mutta lopulta kyseessä on tekniikka, jossa

pyynnöt esitetään kohteilleen epäsuorasti. Vierailijan tapauksessa pyynnöt esitettiin epäsuorasti vierailijaa käyttäen oliolle itselleen. Kyseessä on selkeästi erikoistapaus tavallisemmasta tilanteesta, jossa pyynnön toteuttaminen ulkoistetaan toiselle oliolle. Koosteolion sijaan käytettiin hyödyksi parametrin välitystä.



Kuvio 7. Iteraattori

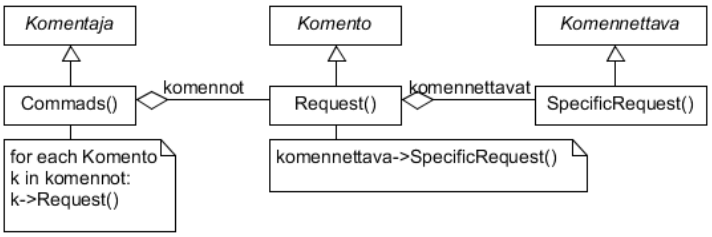
Suunnitteluratkaisun päätarkoitus on tarjota yhtenäinen rajapinta oliorakenteen läpikäyntiin. Varsinaisten olioiden tyypillä ei siten ole merkitystä rakenteen läpikäynnin kannalta. Iteraattori on riippumaton tyypeistä, mihin perustuu sen käytettävyys. Näin ollen rakenteet ja iteraattorit voidaan julistaa niin sanotusti geneerisiksi tyyppiluokiksi.

Iteraattorin toteutus riippuu oliorakenteesta. Tästä syystä iteroitava rakenne kutsuu eksplisiittisesti iteraattorin konstruktoria. Täten kaikki mallin palaset ovat tiedossa, kunhan iteroitava rakenne on selvillä, eli käytännössä silloin kun metodia (Create) kutsutaan. Metodin (Create) tarkoitus on suorittaa iteraattorin valinta rajapintaa käyttävän olion puolesta. Tämä voidaan tehdä, koska iteraattori on rakennekohtainen. Samalla menetetään kuitenkin mahdollisuus erilaisen iteraattorin luomiseksi – ainakin jos hyvän ohjelmointitavan mukaisesti halutaan ohjelmoida rajapintaa vasten.

Toisin sanoen kunhan tiedetään iteroitava rakenne, niin tiedetään myös varsinainen iteraattori. Ohjelman toimintaa muuttamatta voitaisiin kutsua iteroitavan rakenteen metodin (Create) asemesta kuvitteellista iteraattorin kuormitettua, staattista metodia Iterator::Create(Iterable), joka palauttaisi iteroitavaa rakennetta vastaavan iteraattorin. Kuvitteellinen metodi toimisi aivan kuten alkuperäinen metodi (Create), mutta kutsu kohdistetaankin iteraattorille (Itera-

tor). Itseasiassa ratkaisu voisi olla edellä mainittua käytännöllisempi, sillä tarve iteraattorille saatetaan huomata vasta oliorakenteen määrittelyn jälkeen. Tällöin rajapintaa jouduttaisiin muuttamaan jälkikäteen, mistä aiheuttuu yhteensopivuusongelmia. Huomaa, että metodin (Create) poistaminen johtaa rajapinnan poistamiseen, sillä metodi (Create) saattaa olla ainoa eri tietorakenteita yhdistävä metodi. Lopputuloksena saadaan kuvion 7 mukainen luokkakaavio.

Saavutettiin esitystapa, joka muistuttaa staattiselta rakenteeltaan sovitinta [3]. Ero johtuu siitä, että toisin kuin sovitin, iteraattori sisältää ajoaikaista logiikkaa, joka on nähtävissä toteutuksessa. Sovitin on kuitenkin mallin taustalla, sillä käsitteellisessä mielessä iteraattori tavallaan sovittaa läpikäynti-algoritmin tietorakenteeseen. Tyypillisesti algoritmit kuvataan strategiana [1]. Tässä tapauksessa varioinnin kohteena on kuitenkin iteraattori eikä tietorakenne, joten koostesuhde on strategian suhteen käänteinen.



Kuvio 8. Komento

Komennossa niin sanotusti liikkuvia osia ovat komento ja komennon suorittaja. Aikaisemmista malleista poiketen komennon lopullinen ajoaikainen suorittaja irrotetaan komennon rekisteröivästä rajapinnasta, mikä on hyvin tyypillistä etenkin tapahtumankäsittelyssä.

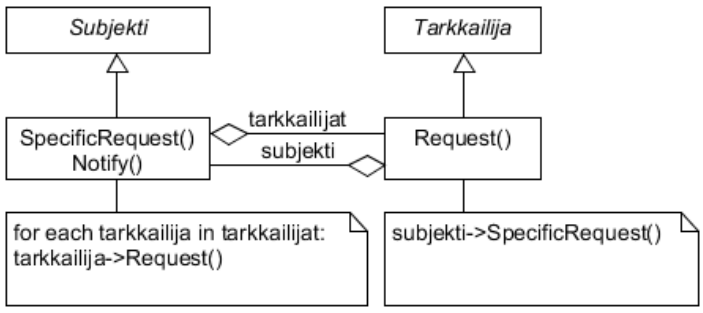
Tapahtumasekvenssikaavio paljastaa, että komennon luontihetkellä on tiedossa paitsi komento ja komennettava myös komentaja. Komento määritetään eksplisiittisesti kutsumalla tietyn komennon rakentajaa, joka kapseloinnin periaatteiden mukaisesti alustaa koosteolionsa kyseisellä parametrilla. Komento liittyy vain parametrin tyypin mukaisiin kohteisiin, joten molemmat ovat luontihetkellä tiedossa.

Samoin tiedetään komentaja, jolle komentoa ollaan rekisteröimässä. Rekisteröinti voidaan toteuttaa esimerkiksi linkitetyllä listalla, jonka kaikille alkioille komento suoritetaan. Suori-

tettava metodi on määrätty komennon rajapinnassa. Tavoitteena on suorittaa komento aina, kun komentajan tietyt ehdot astuvat voimaan. Tästä seuraa, että suoritettavan komento pitää kiinnittää viimeistään komentoa tallennettaessa komennon suorittajaan, eli komento kiinnitetään ajoaikana ensiksi.

Tästä syystä ei ole oikeastaan merkitystä suorittaako komennon komentaja vaiko rajapinta. Vaihtoehtoisesti rajapinta voisi kuunnella komentajan tapahtumia ja suorittaa komennon itse, jolloin alustustoimenpiteiden jälkeen toiminta säilyisi oleellisin osin ennallaan. Silloin Rajapinta lähettäisi pyynnön, eikä komentaja. Ratkaisun kannalta oleellista ei ole niinkään komennon suorittaja, vaan mallin komponenttien väliset riippumattomuudet siltä osin, kun rajapintariippuvuudet jätetään huomiotta.

Muodostetaan alustustoimenpiteiden jälkeistä tilaa kuvaava luokkakaavio 8. Huomataan, ettei luokkakaavio eroa juuri edellisistä esityksistä. Luokkakaavion pohjalta on mahdollista luoda ajoaikainen komentorakenne, kunhan rakenne alustetaan asianmukaisesti ajoaikana. Tällöin rajapinta lisää uuden komennon komennot listaan. Dynaamisuus seuraa esimerkiksi komentoja sisältävän linkitetyn listan läpikäynnistä.



Kuvio 9. Tarkkailija

Tarkkailija koostuu tarkkailijasta ja tarkkailtavasta subjektista. Esi- ja jälkiehtojen rikkominen aiheuttaa usein ajoaikaisen virheen, jota ei havaita käännösaikana. Tästä syystä esi- ja jälkiehdot ovat merkittäviä virheenkäsittelystä huolimatta, sillä niillä taataan ohjelman oikeellisuus vaikkakaan ei virheettömyys. Esi- ja jälkiehtojen tarkoitus on asettaa ohjelman ja muuttujien tilaa rajoittavia sääntöjä ohjelman ajoaikaiselle toiminnalle. Ohjelman oikeellisuus voidaan tällöin osoittaa määrättyjen ehtojen puitteissa.

Tarkkailijan pitää tietää subjektin luokka, jotta tietää mihin arvoon rajapintaan kuulumaton, tarkkailtu ominaisuus vaihtui. Subjekti ei tiedä tarkkailijan tyyppiä, ja on ainoastaan vastuussa muutoksien ilmoittamisesta rajapinnan kautta. Tarkkailijan näkökulmasta subjektin kiinnitetään ensin, koska subjektin luokalla on merkitystä tarkkailijan kannalta. Subjektin alustukseen kuuluu lisäksi tarkkailijan lisääminen listalle, jonka alkioille lähetetään pyyntö muutoksen tapahtuessa. Rajapinnan tavoin tarkkailijat ovat riippumattomia subjektin sisäisestä tilasta ja niiden näkökulmasta subjekti on viimekädessä vastuussa muutoksien ilmoittamisesta tarkkailijoille. Tällöin muutoksen jälkeen voidaan kutsua suoraan tarkkailijaa.

2.3 Yhteenveto suunnittelumallien vertailusta

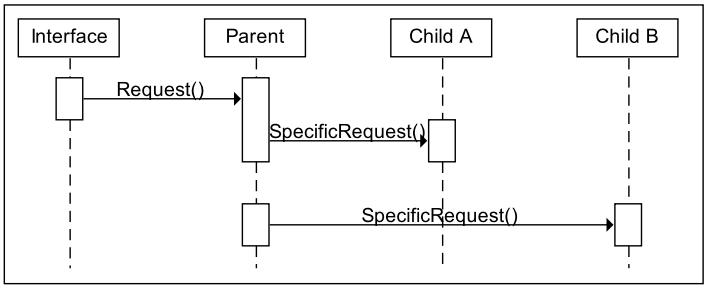
Suunnittelumallien vertailun yhteydessä havaittiin muutamia Gamman (1995) luokittelua noudattavia rakenne- ja käytösmallien piirteitä. Kerrataan seuraavaksi yhteenvetona näistä tärkeimmät, jotka liittyvät keskeisesti luvun 3 suunnittelumallien formalisointiin ja edelleen niiden vertailuun luvussa 4.

Rakennemallien kontrollinohjaus perustuu olioiden ja luokkien koostamiseen, joka tarjoaa keinot toimintojen rakenteelliseen delegointiin. Esimerkiksi metodikutsun toteuttaja on osasuhteessa kutsun delegoineen osapuolen kanssa. Vastaavasti käytösmallien kontrollinohjauksessa esiintyy rajapintojen koostamista, jolla saavutetaan olioiden ja luokkien välille toivottua dynaamisuutta. Esimerkiksi ohjelmistoon voidaan lisätä rajapinnan toteuttavia luokkia ilman, että olemassa olevien luokkien lähdekoodia tarvitsisi uudelleen kääntää, koska lisätyn luokan ilmentymiltä edellytetään vain sovitun rajapinnan toteuttamista. Sitä vastoin luokkahierarkiaan kohdistuvat muutokset vaikuttavat suoraan hierarkian muihin luokkiin, mikä hankaloittaa merkittävästi ohjelmiston ylläpitoa. Staattinen perintä voidaan siten tulkita dynaamisten rajapintojen vastakohtana, jotka tarjoavat ratkaisun analogisesti yhtenevään ongelmaan: luokkien välisen assosiation määrittämiseen osa- vai käyttösuhteena. Rakennemallit tarjoavat ratkaisun edelliseen ja käytösmallit jälkimmäiseen valintaan.

Seuraavaksi tutkitaan kuinka staattiset rakennemallit olisivat kuvattavissa dynaamisina rakenteina, mikä on perusteltua, koska kaikki edellä luetellut rakennemallit ovat toisiinsa nähden ekvivalentit ajoaikana. Rekursiokooste- ja kuorruttaja-mallit nimittäin laajentavat silta-,

sovitin- ja strategiamalleja siten, että metodikutsu välitetään koosteoliolle joko ennen pyynnön käsittelyä tai sen jälkeen. Pyynnön toteuttaja voi olla, toimintoa pyytäneen olion näkökulmasta, eri kuin alkuperäisen pyynnön vastaanottaja. Edellä kuvatun toiminnallisuuden mahdollistavat olioluokkien monimuotoisuus eli polymorfismi. Toisin sanoen toimintojen pyytäminen epäsuorasti rajapintojen tai abstraktien kantaluokkien kautta mahdollistavat toiminnon toteutuksen muuttamisen ajoaikana vaihtamalla delegoinnin kohteena olevan luokkailmentymän toisen olioluokan ilmentymäksi. Lopputuloksena syntyy dynaaminen rakenne, joka toteuttaa alkuperäisen staattisen vastineensa.

Edellä johdettua tulosta voisi kuvata siten, että dynaamiset mallit ovat ilmaisuvoimaisempia kuin staattiset mallit. Staattiset luokkakaaviot voidaan siten muuntaa dynaamisiksi tapahtumasekvenssikaavioiksi menettämättä kontrollinohjaukseen liittyvää informaatiota matkalla. Tällöin rakennemallit vaikuttaisivatkin olevan käytösmallien osajoukko. Jos ohjelmarakenteita verrattaisiin kielioppeihin, näyttäisi olevan perusteltua väittää, että tapahtumasekvenssikaaviolla voidaan kuvata samat ja lisäksi laajempi joukko ohjelmarakenteita kuin luokkakaaviolla – ainakin mitä tulee ajoaikaisuuteen. Käännösaikaiset rajapintojen määritykset eli luokkahierarkiat kannattaa silti edelleen tehdä luokkakaaviolla. Tulos on merkittävä siksi, että yleisyytensä vuoksi luokkakaavioita käytetään tilanteissa, joihin ne eivät sovellu. Esimerkiksi ajoaikaisen kontrollin kulun kuvaamiseen soveltuisi paremmin kuvion 10 kaltainen tapahtumasekvenssikaavio.



Kuvio 10. Dynaaminen rakennekuvaus

Vaikka dynaamisia malleja voitaisiin soveltaa staattisten mallien sijaan, useimmiten olemassa olevat rakenteet kuitenkin sanelevat, mitä ratkaisumallia kannattaa soveltaa missäkin tilanteessa. Suunnittelumallithan liittyvät rajattuun ongelmaan tietyssä kontekstissa. Raken-

nemallien eduksi voidaankin katsoa niiden implisiittinen alustaminen heti käännösaikana, jolloin niitä ei tarvitse erikseen alustaa ajoaikana. Tässä mielessä vierailijakin [6] muistuttaa rakennemalleja ilman koosteisuutta.

Rakennemallit eroavat toisistaan vain hyvin vähän. Vertailtavia rakennemalleja ovat sovitin, silta, strategia, rekursiokooste ja kuorruttaja. Mallien rakennekuvaukset, UML-luokkakaaviot, muistuttavat toisiaan siksi, että ne on esitetty samassa lähteessä. Siksi koetaan tarpeelliseksi arvioida rakennekuvausten lisäksi mallien käsitteellistä merkitystä ja käyttötarkoitusta. Myös mallien alkuperäinen luokittelutapa saattoi osaltaan vaikuttaa tiettyjen mallien dokumentointiin tai ”löytämiseen”. Esimerkiksi tarvetta strategiamallille ei olisi välttämättä ollut, jos alun perin olisi hyväksytty rajapinnat rakennemalleihin käytösmallien sijaan (Gamma 1995). Täten mallien luokittelutapa saattaa aiheuttaa erilaisten variaatioiden dokumentoinnin kokonaan uusina suunnittelumalleina.

Käytösmallien kuvaaminen luokkakaavioilla on hyvin tavallista, sillä kyseisten mallien rakenteet seuraavatkin usein suoraan ohjelmakooditoteutuksesta, josta mallit tunnistettiin. Tämän luvun tarkoitus oli kuvata prosessia dynaamisen rakenteen konstruointiin tapahtumasekvenssikaavion [10] avulla perehtymättä itse ohjelmakooditoteutukseen. Menettely eroaa perinteisestä ”reverse engineering” -lähestymistavasta, jossa mallinnetaan ohjelmakoodia. Lisäksi esitettiin hieman Gamman (1995) esityksestä poikkeava tapa luokitella suunnittelumalleja perustuen niiden dynaamisten elementtien olemassaololle.

3 Suunnittelumallien formalisointi

Suunnittelumallien vertailua rajoittavat niiden dokumentointitavasta johtuvat informalismit. Sanalliset kuvaukset, kaaviot tai koodiesimerkit ovat riittämättömät mallien vertailuun; tarvitaan yhteinen esitysmuoto, jolla esitetään vain mallien karakteristiset eli niitä määrittävät ominaisuudet.

Distributed Cooperation -spesifikaatio (DisCo) kehitettiin alun perin reaktiivisten järjestelmien kuvaamiseen, mutta se soveltuu myös suunnittelumallien formalisointiin (Järvinen ja Kurki-Suonio 1990; Mikkonen 1998). DisCossa suunnittelumallien osa- ja käyttösuhteet kuvataan relaatioina, joihin liittyy sekä esi- että jälkiehtoja. Nämä muodostavat yhdessä taustalla olevan suunnittelumallin mukaisen käyttäytymisen siten, että formalisoinnin jälkeen sen karakteristiset ominaisuudet toteutuvat kaikkina ajan hetkinä. Valintatilanne kahden tai useamman funktion suoritusjärjestysten välillä ratkaistaan epädeterministisesti (Järvinen ja Kurki-Suonio 1990). DisCo ei pakota mitään tiettyä funktioiden suoritusjärjestystä, kunhan huolehditaan esi- ja jälkiehtojen toteutumisesta.

Formaalin suunnittelumallin funktioita suoritetaan vain sille asetettujen esiehtojen toteutuessa. Esiehtojen toteutuminen vuorostaan määräytyy voimassa olevista relaatioista. Vastaavasti funktioiden suorittamisen jälkeen sen jälkiehtojen mukaiset relaatiot astuvat voimaan toteuttaen funktion jälkiehdot (Järvinen ja Kurki-Suonio 1990; Mikkonen 1998). Tällöin suunnittelumallien sisäinen tila pysyy muuttumattomana funktiosuoritusten välissä ja niissäkin tilamuutokset kohdistuvat vain ja ainoastaan vuorovaikuttaviin olioihin. Tästä seuraa funktioiden sivuvaikutuksettomuus.

Mikkosen (1998) mukaan suunnittelumallit määrittelevät ohjelmiston alijärjestelmien tai komponenttien välisiä käyttö- ja osasuhteita, ja sen lisäksi formalisoidut suunnittelumallit ovat toteutusriippumattomia (Mikkonen 1998, s. 115). Siispä formalisoinnin ansiosta kolmiosaisen mallin kontekstin vaikutus kumoutuu. Seuraavassa luvussa siirrytään vertailemaan nyt ongelmaa ja sen ratkaisua, mitkä määrittyvät mallin karakteristisista ominaisuuksista.

4 Formalisoitujen suunnittelumallien vertailua

Mikkonen (1998) formalisoi artikkelissaan kaksi suunnittelumallia: tarkkailijan [9] ja välittäjän. Moniperintää hyödyntäen hän johti niistä Gamman (1995, s. 299–300) esittämän mallien yhdistelmän, tarkkailija-välittäjän (Change Manager). Formalisoitu tarkkailija-välittäjä ei kuitenkaan sellaisenaan soveltunut tämän tutkielman tarpeisiin, koska se määriteltiin moniperinnästä johtuen tarkkailijan ja välittäjän avulla.

Tämän luvun tarkoituksena onkin aluksi konstruoida tarkkailijan ja välittäjän formaalista esityksestä riippumaton esitystapa luvun 2.3 havaintoihin nojautuen. Konstruktion seurauksena syntynyttä formalisoitua mallia vertaillaan Mikkosen (1998, s. 120–122) alkuperäiseen esitykeen. Toisin sanoen vahvistetaan mallien yhtäpitävyys, mikä toimii konstruktiivisena todistuksena sille, että mallin karakteristiset ominaisuudet säilyvät muunnoksessa. Lopuksi konstruoitua mallia vertaillaan Gamman (1995, s. 299–300) alkuperäiseen tarkkailija-välittäjään (Change Manager).

Konstruktio säilyttää Mikkosen (1998, s. 120) mukaan mallin karakteristiset ominaisuudet, joten voidaan todeta mallien samankaltaisuus. Tällöin ainakin nämä kaksi suunnittelumallia ratkaisevat samankaltaisen suunnitteluongelman kontekstista riippumatta, mikä vastaisi myöntävästi johdannossa mainittuun tutkimusongelmaan. Mallien vertailtavuudelta edellytetään kuitenkin karakteristisen ominaisuuksien vertailtavuus. Toisin sanoen esi- ja jälkiehtojen määrittämien relaatioiden täytyy vastata toisiaan myös merkityksiltään. Ongelma liittyy siihen, kuinka karakteristiset ominaisuudet tulisi määritellä siten, että ne kuvaavat yhtäpitävästi relaatiot, joihin esi- ja jälkiehdot liittyvät. Relaatioiden yhtäpitävyys voidaan tässä yhteydessä olettaa, sillä Mikkonen (Mikkonen 1998, s. 118) perusti tarkkailija-välittäjänsä (Managed Observer) Gamman (1995, s. 299–300) vastaavaan malliin (Change Manager). Tosin Mikkonen konstruoi kyseisen mallin tarkkailijan ja välittäjän avulla, mutta nekin perustuvat Gamman (1995) esitykseen. Mallien voidaan siten olettaa toteuttavan samankaltaiset karakteristiset ominaisuudet, kuten Mikkonen (1998) artikkelissaan perusteli.

Tarkkailija-välittäjämallin konstruktio ilman moniperintää on kuvattu liitteessä A. Välittäjän relaatio Connected asetetaan funktiossa Connect. Vastaavasti tarkkailijan relaatio Attached

asetetaan funktiossa Attach. Tarkkailija-välittäjä asettaa vastaavat relaatiot MgdConnected ja Registered funktioissa MgdConnect ja Register taulukon 1 mukaisesti. Alustusten jälkeen kuvion 10 metodien Request ja SpecificRequest vastaavuudet esiintyvät funktioissa Put ja Get sekä funktioissa Notify ja Update. Kutsujärjestyksen voi päätellä paitsi esi- ja jälkiehdoista myös dynaamisten rakenteiden reaktiivisesta käyttäytymisestä. Kuvion 10 rajapinnasta (Interface) saapuvien pyyntöjen (Request) vastaanottaja (Parent) ohjaa pyynnöt (Request) niitä vastaavien toimintojen (SpecificRequest) toteuttajille (Child).

Taulukko 1. Vastaavuustaulukko

Managed Observer	Mediator	Observer
MgdConnect	Connect	—
MgdDisconnect	Disconnect	—
Register	—	Attach
Release	—	Detach
Modify	Put	Notify
Dispatch	Get	Update

Taulukossa 1 on esitetty mallien tarkkailija, välittäjä ja tarkkailija-välittäjä vastaavat funktiot. Funktioiden muodostamat relaatiot eivät vastaa toisiaan merkityksiltään. Sen sijaan relaatiot muodostetaan samalla tavalla.

Taulukon 1 relaatiot ovat käyttö- ja osasuhteita eli viimekädessä olioviitteitä. Formaaleissa määrittelyissä usein virheellisesti oletetaan, että mallin karakteriset ominaisuudet olisivat kaikki relaatioita (Mikkonen 1998; Taibi 2003). Esimerkiksi Mikkosen (1998) tarkkailijamallissa relaatio Updated kuvaa sitä, että tarkkailija on ajantasalla subjektin kanssa. Näin ei kuitenkaan tarvitse olla, sillä kyseessä on toteutusriippuvainen yksityiskohta. Mallin käyttötarkoitus on kuvata tarkkailijan reagoimista muutoksiin subjektissa. Eikä niinkään todeta, että tietyt arvot olisivat samat. Gamma (1995) esitti arvon päivityksen relaatiolla Updated selkeyttääkseen mallin toimintaperiaatetta, mutta varsinainen toteutus on mallin kannalta triviaali.

Perusteluna olkoon se, että tarkkailija voi muuttaa arvoaan subjektista riippumatta. Formalisoidussa mallissa tämä aiheutuisi tahattoman relaation Updated purkautumisen, sillä sub-

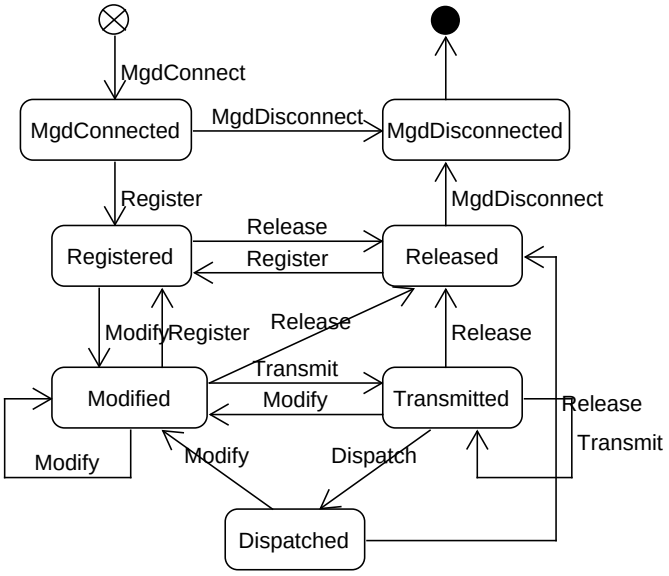
jektin ja tarkkailijan arvot olisivat erisuuret, vaikka oliot olisivat edelleen yhteydessä toisiinsa. Todisteena olkoon sekin, että relaatio Detach purkaa myös relaation Updated, koska nyt tarkkailijalla ei ole keinoa kohdistaa arvon noutavaa funktiota, koska subjekti ei enää viittaa tarkkailijaan [9]. Detach ei varsinaisesti muuta synkronoituja arvoja vaan poistaa osasuhteen, mikä vaikuttaa tarkkailijan käyttösuhteeseen. Tästä syystä subjektin ja tarkkailijan arvojen yhtäpitävyys on toissijaista osasuhteen eli olioviitteen olemassolon rinnalla. Relaatioiden merkityksen määrittäminen riippuu valitusta näkökulmasta ja lähestymistavasta, jolla mallin toimintaa pyritään kuvaamaan. Tärkeintä lienee kuitenkin valinnan johdonmukaisuus.

Relaatioiden tulee olla vain ja ainostaan osa- tai käyttösuhteita siten, että osasuhteet olettavat viiteolion alustetuksi, kun käyttösuhteet olettavat parametrin välitystä tai olioviitteen selvittämistä muuten. Tällöin voidaan esi- ja jälkiehdot yhdistää funktioiden välillä, sillä sivuvaikutuksettomat funktiot vain oleellisesti ryhmittelevät ja nimeävät tiettyjä koodisekvenssejä, mihin perustuu tilojen yhdistäminen kuviossa 11. Tällaisessa formaalissa esityksessä esi- ja jälkiehdot kuvaavat täsmällisesti algoritmeja, kuten esimerkiksi listan järjestämistä. Abstraktit toimenpiteet, kuten liitoksen muodostaminen sen sijaan vaativat mallin käyttötarkoituksen tuntemista. Voidaan helposti keksiä tapauksia, jotka toteuttavat esi- ja jälkiehdot poiketen mallin käyttötarkoituksesta (Haikala ja Märijärvi 2004, s. 109). Formalisointi on kuitenkin riittävän tarkka kuvaamaan eri mallien rakenteelliset yhtäläisyydet.

Esimerkiksi liitteen A relaatio Attached ilmenee ohjelmakoodissa yksinkertaisena olioviitteenä, eli Attach-metodi oikeastaan alustaa olioviitteen, jota pitkin subjekti ilmoittaa muutoksista tarkkailijalle. Relaatiot Connected ja Updated esitetään yhteisellä notaatiolla, vaikka niiden toteuttamiseksi vaaditut ohjelmarakenteet eroavat toisistaan. Connected merkitsee osasuhteen olemassaoloa, mutta Updated liittyy päivitettävien olioiden jäsenmuuttujien arvojen yhtäpitävyyteen. Ongelma koskee relaatiosuhteen määrittelyä, joka näyttäisi puuttuvan tekstuaalisesta notaatiosta. Relaatiot voidaan tässä mielessä nähdä joukkona matalamman abstraktiotason ominaisuuksia, jotka riippuvat relaation osapuolten luokkamäärittelystä. Toisaalta käytösmallien yhtäpitävyyden tarkastelussa keskitytään ennen kaikkea kontrollin kulkuun, jolloin toissijaiset relaatioiden toteutukset voidaan sivuuttaa. Näin ollen riittäisi verrata konstruktioita, kun verrataan käytösmallien ajoaikaista yhtäpitävyyttä.

DisCo-menetelmässä toiminnot ovat pienin suorituksen yksikkö, jotka assosioivat luokkien

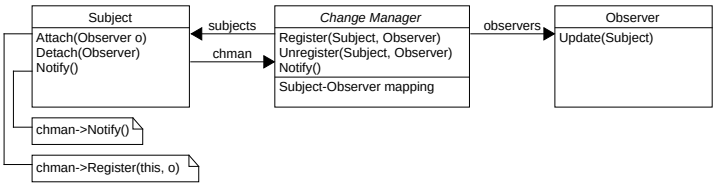
määrittämiä olioita toisiinsa relaatioilla (Mikkonen 1998, s. 116). Toisin sanoen ohjelma voi vaihtaa sisäistä tilaansa ainoastaan suorittamalla jonkin ennalta määrätyn toiminnon. Lisäksi suoritettavilta toiminnoilta vaaditaan karakteristisen ominaisuuksien voimassaoloa, mikä edellyttää esi- ja jälkiehtojen toteutumista toimintojen suoritusten välissä.



Kuvio 11. Tarkkailija-välittäjä

Tilakaavio havainnollistaa kuinka tilasiirtymät ovat pitkälti ennaltamäärättyjä kustakin tilasta. Aiemmin esitettiin, että käytösmallit sisältävät alustettavat rakenteet ja siihen kohdistuvat alustustoimenpiteet. Visuaalinen tilakaavio havainnollistaa tilannetta: Aluksi lähettäjä (MgdSubject) tulee yhdistää (MgdConnect), ja vastaanottaja (MgdObserver) rekisteröidä (Register) välittäjään (Manager). Lopuksi tarkkailija (MgdObserver) saa tiedon subjektin (MgdSubject) tilamuutoksista. Näin ollen tarkkailija-välittäjän käyttö sovelluksessa on kaksivaiheinen prosessi, johon kuuluu alustukset ja kontrollinkulku. Yhdistetään tilat MgdConnected, Registered ja Dispatched sekä tilat Modified ja Transmitted. Alkutilaksi valitaan alustusten jälkeinen Disconnected, joka on samalla hyväksyvä lopetustila. Vertaa tarkkailija-välittäjää [11] tarkkailijaan [9] yhdenmukaisuuden toteamiseksi.

Yhdennäköisyys lopputulemassa ei ole yllättävää, sillä Mikkonen (1998) määritteli tarkkailija-välittäjän periytymään tarkkailijasta ja välittäjästä. Niin ikään tarkkailija ja välittäjä käyttäytyvät samoin, vaikka ovat itsenäisiä suunnittelumalleja ja omaavat erilaiset rakennekuvauk-



Kuvio 12. Muutosmanageri (Gamma 1995, s. 300)

set. Syynä on se, että DisCo vain huolehtii karakteristisiten ominaisuuksien säilymisestä, eikä ota kantaa toteutusyksityiskohtiin.

Näiden esimerkkien valossa jotkin suunnittelumallit, ainakin tarkkailija-välittäjä ja muutosmanageri, ratkaisevat samankaltaisen suunnitteluongelman kontekstista riippumatta. Toisin sanoen tarkkailija-välittäjän tilalle voidaan vaihtaa muutosmanageri, sillä konteksti ei näyttäisi vaikuttavan ratkaisevasti siihen kumpi valitaan. Suunnitteluratkaisut ovat johdettavissa samoista olio-ohjelmoinnin suunnitteluperiaatteista, joita ovat esimerkiksi delegointi, kapselointi, perintä ja rajapinnat. Suunnittelumallit tarjoavat ratkaisumalleja aivan syystäkin, sillä ne edustavat hyviä olio-ohjelmoinnin suunnitteluperiaatteita.

5 Yhteenveto

Vertailussa huomattiin, että suunnittelumallit eroavat toisistaan loppujen lopuksi aika vähän. Voidaan kysyä, että onko mallien kirjaaminen ja dokumentointi kovinkaan hyödyllistä, koska kehitystyö näyttäisi joka tapauksessa johtavan yhdenmukaisiin rakenteisiin. Mallien tarkoitus ei alun perinkään ole esittää ennen näkemättömiä suunnitteluratkaisuja, mutta tarvitaanko montaa mallia kuvaamaan oleellisesti samaa asiaa. Vaikka mallit vaikuttavat olevan toistensa variaatioita, ne ovat edelleen yhtä käyttökelpoisia ohjelmistokehityksessä kuin ennenkin. Ainakaan edelliset löydökset eivät heikennä olemassa olevien mallien ilmaisuvoimaa.

Uusien sovellusaluekohtaisten mallien soveltamisien julkaisua tulisi toisaalta jarrutella, koska suurin osa yleisimmistä malleista lienee jo kuvattu. Ohjelmistorakenteiden äärellisten kombinaatioiden seurauksena mallejakaan ei ole aivan loputtomasti. Olettaen, että käytetään Michelucci (2013) määritelmää, jossa suunnittelumallit sijoitettiin abstraktiotasolla algoritmien ja arkkitehtuurin väliin. Tällöin rajattaisiin pois niin sanotut arkkitehtuurimallit. Esimerkiksi Mikkonen (1998) esitti tarkkailija-välittäjä -mallin (Managed Observer), joka on tarkkailija- ja välittäjä-mallien yhdistelmä. Voidaan argumentoida puolesta ja vastaan ovatko suunnittelumallien yhdistelmät, sovellukset tai variaatiot itsenäisiä malleja.

Mallien formaali esittäminen voisi sellaisenaan tuoda arvokasta tietoa siitä kuinka suunnittelumallit ymmärretään. Tutkimuksessa tulisi jatkossa keskittää voimavarat mallien analysointiin, analyysityökalujen kehittämiseen sekä yhdenmukaisten mallikokoelmien dokumentointiin ohjelmistokehittäjien avuksi.

Lähteet

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad ja Michael Stal. 1996. “A system of patterns: Pattern-oriented software architecture”.

Forsell, Marko. 1998. “Suunnittelumallien käyttö ohjelmistotuotannossa” [kielellä fin]. Pro gradu -työ : Jyväskylän yliopisto, tietojenkäsittelytieteiden laitos, tietojärjestelmätiede. Tohtorinväitöskirja.

Gamma, Erich, toimittanut. 1995. *Design patterns : elements of reusable object-oriented software* [kielellä eng]. Addison-Wesley professional computing series. Lisäpainokset: 5th pr. 1995 - 8th pr. 1996 - 11th, 13th pr. 1997 - 19th pr. - 20th pr. 2000. - 22nd pr. 2001. - 24th pr. 2002. - 43rd pr. 2015. Reading (MA): Addison-Wesley.

Gamma, Erich, Richard Helm, Ralph Johnson ja John Vlissides. 1993. “Design patterns: Abstraction and reuse of object-oriented design”. Teoksessa *European Conference on Object-Oriented Programming,* 406–431. Springer. doi:10.1007/3-540-47910-4.

Gamma, Erich, ja Anita Toivonen, toimittaneet. 2001. *Olio-ohjelmointi : suunnittelumallit : design patterns* [kielellä fin]. IT Press professional. Helsinki: Edita, IT Press.

Haikala, Ilkka, ja Jukka Märijärvi. 2004. *Ohjelmistotuotanto* [kielellä fin]. 10. uud. p. Lisäpainokset: 11. p. 2006. Helsinki: Talentum. ISBN: 952-14-0850-2 nidottu.

Järvinen, Hannu-Matti, ja Reino Kurki-Suonio. 1990. *The DisCo language* [kielellä eng]. Tampere: Tampere University of Technology. ISBN: 951-721-496-0 nidottu.

Michelucci, Pietro, toimittanut. 2013. *Handbook of Human Computation* [kielellä eng]. New York, NY: Springer New York. doi:10.1007/978-1-4614-8806-4.

Mikkonen, T. 1998. “Formalizing design patterns” [kielellä eng], 115–124. doi:10.1109/ICSE.1998.671108.

Snyder, Alan. 1986. “Encapsulation and inheritance in object-oriented programming languages” [kielellä eng]. *ACM SIGPLAN Notices* 21 (11): 38–45. doi:10.1145/960112.28702.

Taibi, Toufik. 2003. “Formal Specification of Design Patterns - A Balanced Approach.” [kielellä eng]. *The Journal of Object Technology* 2 (4): 127. doi:10.5381/jot.2003.2.4.a4.

Zimmer, Walter. 1995. “Relationships between design patterns”. *Pattern languages of program design* 57.

Liitteet

A Tarkkailija-välittäjä (Mikkonen 1998, s. 118–120)

$$
\begin{aligned}
&\mathit{MgdConnect}(\text{mgr:Manager; mb:Mbox; ms:MgdSubject}):\\
&\quad \neg mgr \cdot Connected \cdot ms,\\
&\quad \wedge \neg mb \cdot Sender \cdot \text{class MgdSubject},\\
&\quad \wedge mb \in mgr.Boxes,\\
&\quad \rightarrow mgr \cdot Connected' \cdot ms,\\
&\quad \wedge mb \cdot Sender' \cdot ms
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{MgdDisconnect}(\text{mgr:Manager; mb:Mbox; ms:MgdSubject}):\\
&\quad mgr \cdot Connected \cdot ms,\\
&\quad \wedge mb \cdot Sender \cdot ms,\\
&\quad \wedge mb \in mgr.Boxes,\\
&\quad \wedge \neg ms \cdot Attached \cdot \text{class MgdObserver},\\
&\quad \rightarrow \neg mgr \cdot Connected' \cdot ms,\\
&\quad \wedge \neg mb \cdot Sender' \cdot ms,\\
&\quad \wedge \neg \text{class Mbox} \cdot Receiver' \cdot ms
\end{aligned}
$$

Register(ms:MgdSubject; mo:MgdObserver; mb:Mbox; mgr:Manager):

$$
\begin{aligned}
& \neg ms \cdot Attached \cdot mo, \\
\wedge\ & \neg mgr \cdot Connected \cdot mo, \\
\wedge\ & \neg mb \cdot Sender \cdot \text{class MgdObserver}, \\
\wedge\ & mb \in mgr.Boxes, \\
\wedge\ & mgr \cdot Connected \cdot ms, \\
\rightarrow\ & ms \cdot Attached' \cdot mo, \\
\wedge\ & mgr \cdot Connected' \cdot mo, \\
\wedge\ & mb \cdot Sender' \cdot mo
\end{aligned}
$$

Release(ms:MgdSubject; mo:MgdObserver; mb:Mbox; mgr:Manager):

$$
\begin{aligned}
& ms \cdot Attached \cdot mo, \\
\wedge\ & mgr \cdot Connected \cdot mo, \\
\wedge\ & mb \cdot Sender \cdot mo, \\
\wedge\ & mb \in mgr.Boxes, \\
\rightarrow\ & \neg ms \cdot Attached' \cdot mo, \\
\wedge\ & \neg ms \cdot Updated' \cdot mo, \\
\wedge\ & \neg mgr \cdot Connected' \cdot mo, \\
\wedge\ & \neg mb \cdot Sender' \cdot mo, \\
\wedge\ & \neg \text{class Mbox} \cdot Receiver' \cdot mo
\end{aligned}
$$

$Modify$(ms:MgdSubject; mb:Mbox; mgr:Manager; d):

$$
\begin{aligned}
& mgr \cdot Connected \cdot ms, \\
\wedge\ & mgr \cdot Connected \cdot \{\}, \\
\wedge\ & mb \cdot Sender \cdot ms, \\
\wedge\ & mb \in mgr.Boxes, \\
\rightarrow\ & \neg ms \cdot Updated' \cdot \text{class MgdObserver}, \\
\wedge\ & ms.Data' = d, \\
\wedge\ & mb \cdot Reveiver' \cdot \{\}, \\
\wedge\ & \neg mb \cdot Receiver' \cdot (\text{class MgdObserver - } \{\}), \\
\wedge\ & mb.Message' = d
\end{aligned}
$$

$Transmit$(ms:MgdSubject; {mo}*:MgdObserver; mb:Mbox; mgr:Manager; d):

$$
\begin{aligned}
& mgr \cdot Connected \cdot ms, \\
\wedge\ & mgr \cdot Connected \cdot mo, \\
\wedge\ & mb \cdot Sender \cdot ms, \\
\wedge\ & mb \in mgr.Boxes, \\
\wedge\ & d = ms.Data, \\
\wedge\ & ms \cdot Attached \cdot mo, \\
\wedge\ & \neg ms \cdot Updated \cdot mo, \\
\rightarrow\ & mb \cdot Receiver' \cdot mo, \\
\wedge\ & \neg mb \cdot Receiver' \cdot (\text{class MgdObserver - mo}), \\
\wedge\ & mb.Message' = d
\end{aligned}
$$

Dispatch(ms:MgdSubject; mo*:MgdObserver; mb:Mbox; mgr:Manager; d):

$$
\begin{aligned}
& ms \cdot Attached \cdot mo, \\
\wedge\ & \neg ms \cdot Updated \cdot mo, \\
\wedge\ & mgr \cdot Connected \cdot mo, \\
\wedge\ & mb \cdot Receiver \cdot mo, \\
\wedge\ & mb \in mgr.Boxes, \\
\wedge\ & mb.Message = d, \\
\wedge\ & mb \cdot Sender \cdot ms, \\
\rightarrow\ & ms \cdot Updated' \cdot mo, \\
\wedge\ & mo.Data' = d, \\
\wedge\ & \neg mb \cdot Receiver' \cdot mo
\end{aligned}
$$