TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MARKO TUOMINEN

# KOMPONENTTIEN RAJAPINNAT AKTIOPOHJAISESSA OHJELMOINTIKIELESSÄ

Diplomityö

Tarkastaja: professori Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
28. helmikuuta 2018

# TIIVISTELMÄ

**Marko Tuominen**: Komponenttien rajapinnat aktiopohjaisessa ohjelmointikielessä
Tampereen teknillinen yliopisto
Diplomityö, 47 sivua
Helmikuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: aktioparadigma, komponentti, olio, aktio, subjekti, rinnakkainen ohjelmointi, rajapinta

Aktioparadigma on 80-luvulla kehitetty ohjelmointiparadigma, jossa ohjelman rakenne toteutetaan ohjelmantilaa varastoivien olioiden ja ohjelman suoritusta edistävien aktioiden avulla. Tämän paradigman päälle on lähdetty suunnittelemaan uutta aktiojärjestelmää, jonka keskeisenä ideana on esitellä uudenlainen aktioiden ympärille rakennettu rinnakkaisten ohjelmien suunnittelu ja toteutustapa, jossa siirretään mahdollisimman paljon rinnakkaisuuden vaatimista erikoisvaatimuksista laitteiston vastuulle huolehdittavaksi. Aktiojärjestelmällä suoritettavia ohjelmia tehdään uudella vielä suunnitteluvaiheessa olevalla aktio-ohjelmointikielellä, jota kutsutaan tässä diplomityössä lyhyesti aktiokieleksi. Aktiokielellä tehtävää aktio-ohjelmointia on suunniteltu käytettävän reaktiivisissa sulautetuissa järjestelmissä.

Diplomityössä tehtiin alustavaa teoreettista selvitystyötä, kuinka aktiokielellä toteutettujen komponenttien välinen rajapintatoiminta kannattaisi toteuttaa. Komponentilla tarkoitetaan kokonaisuutta, jolla on muusta ohjelmasta erotettu sisäinen toteutus, jonka kanssa ulkomaailma kommunikoi käyttörajapinnan avulla. Aktiokieleen lisättiin komponenttien määrittelemiseksi kolme uutta rakennetta: käyttörajapinta, komponentti ja komponenttiolio. Näiden rakenteiden päälle suunniteltiin erilaisia tapoja toteuttaa komponenttien välinen rajapintatoiminta, jotka kuvattiin rajapintamalleina. Rajapintamallit suunniteltiin pääasiassa käyttämällä aktiokielen perusrakenteita, koska niitä käyttämällä minimoidaan muutokset aktiokieleen ja saadaan hyödynnettyä mahdollisimman paljon näiden rakenteiden jo olemassa olevia ominaisuuksia. Rajapintamalleja suunniteltiin yhteensä kuusi erilaista.

Rajapintamalleissa esitettyjen ideoiden pohjalta aktiokieleen suunniteltiin rajapintakokonaisuus, jonka tarkoituksena on yhdistää rajapintamalleissa esitetyt toimivimmat ratkaisut esitykseksi aktiokielessä käytettäväksi standardirajapintarakenteeksi. Rajapintakokonaisuudessa päädyttiin tukemaan aktioita rajapintatoiminnan päärakenteena, koska niitä käyttämällä saadaan toteutettua helposti omaksuttava ja intuitiivinen rakenne käyttää. Aktiota käyttämällä rajapintatoimintaan saadaan selkeä työnjako, joka samalla tukee aktioiden roolia aktiokielen toiminnallisina rakenteina.

# ABSTRACT

**Marko Tuominen**: Component interfaces in action-based programming language
Tampere University of Technology
Master of Science Thesis, 47 pages
February 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen

Keywords: action paradigm, component, object, action, subject, parallel programming, interface

Action paradigm is a programming paradigm developed in the 1980's, where the program structure is implemented using objects that store the program state and actions that advance the execution of the program. Based on this paradigm the design of a new action system is created whose main idea is to present a new kind of design and implementation of parallel programs built around the actions, moving as much as possible from the special requirements required for parallelism to be handled by the hardware. The programs running on the action system are being carried out with a new action-programming language at the design stage, which is briefly referred to as the action language in this thesis. Action-based programming in action language is designed to be used in reactive embedded systems.

The thesis is preliminary theoretical work on how to implement interfaces between the components implemented in the action language. Component is an entity that has an internal implementation with which the outside world communicates through the interface. Three new structures were added to the action language to define the components: an interface, a component and a component object. Different ways to implement interfaces that are described as interface templates were designed based on these structures. The interface templates were mainly designed using low-level structures of the action language because they minimize changes to the action language and maximize the existing capabilities of these structures. A total of six different interface templates were designed.

Based on the ideas presented in the interface templates, an interface was designed to integrate the most effective solutions presented in the interface templates into a standard interface architecture for use in an action language. In the interface, it was decided to support the action as the main structure of the interface, since by using them, an easy-to-understand and intuitive structure can be achieved for the interface. By using actions, a clear division of labor is realized that at the same time supports the role of actions as the functional structures of an action language.

# ALKUSANAT

Tämä diplomityö on tehty Tampereen teknillisen yliopiston Tietotekniikan laboratoriolle 2018.

Haluan kiittää diplomityön ohjaajaa Hannu-Matti Järvistä työssä käsiteltävän aiheen esittelemisestä minulle, ja hänen näkemyksistään ja mielipiteistään koskien työtä. Haluan kiittää myös Santtu Tikkaa hänen ajatuksistaan ja avusta koskien työtä.

Tampereella, 11.02.2018

Marko Tuominen

# SISÄLLYSLUETTELO

# 1. JOHDANTO

Nykyinen ohjelmointi pohjautuu vahvasti prosesseihin. Prosessit ovat tietokoneohjelmien instansseja, joilla on tiedossa suoritettavan ohjelman ohjelmakoodi ja tieto ohjelman nykyisestä tilasta. Ensimmäiset ohjelmat sisälsivät ainoastaan prosessorilla suoritettavat konekäskyt, minkä takia ainoastaan yhtä ohjelmaa pystyttiin ajaa kerrallaan. Prosessit ratkaisivat tämän ongelman ja ovat vakiinnuttaneet paikkansa ohjelmoinnin yhtenä perusyksikkönä.

Prosessien suorittamiseen erikoistettujen prosessoreiden ympärille on kehittynyt laaja liiketoiminta, jonka tavoitteena on kehittää toinen toistaan tehokkaampia prosessoreita markkinoille. Prosessoreiden suorituskyky kasvoi merkittävästi, kun yritykset alkoivat nostaa prosessoreissa käytettävää kellotaajuutta, mikä määrää pääasiassa sen, kuinka monta konekäskyä prosessori voi suorittaa sekunnissa. Kellotaajuuden nostaminen ei kuitenkaan ole nykyään yleistä, vaan suorituskykyä kasvatetaan pääasiassa lisäämällä ohjelmakoodia suorittavien suoritinytimien määrää. Tämän takia olisi tärkeää, että ohjelmat pystyisivät hyödyntämään useasta ytimestä saatavan suoritustehon.

Rinnakkaisuutta ei kuitenkaan voida noin vain lisätä ohjelmaan, vaan monen suoritinytimen hyödyntäminen vaatii ohjelmakoodin rakentamista alusta alkaen toimimaan rinnakkain. Ohjelmakoodin suorittaminen rinnakkain tuo mukanaan useita ongelmia, joihin ohjelmoijan on otettava kantaa, kuten poissulkeminen, lukkiutumiset, synkronointi, ja nälkiintyminen. Nämä ongelmat moninkertaistuvat ohjelmakoodin määrän kasvaessa, mikä tekee ohjelman oikean toiminnallisuuden varmistamisesta lähes mahdotonta kokeneellekin ohjelmoijalle. Tämän takia ohjelmia ei monesti toteuteta toimimaan rinnakkain ja prosessoreiden monesta suoritinytimestä saatava laskentateho jää merkittävästi hyödyntämättä.

Rinnakkainen ohjelmointi on tällä hetkellä hyvin samankaltaisessa tilanteessa kuin muistinhallinta oli 80-luvulla. Tuohon aikaan ohjelmoijan vastuulla oli määrittää ja valvoa mihin ohjelma sijoittui muistissa, mikä suuressa osassa ohjelmista lähinnä hidasti ja vaikeutti turhaan ohjelmoijien työtä. Tilanteen ratkaisuksi kehitettiin virtuaalimuisti. Virtuaalimuistin ideana on siirtää muistinhallinta prosessorin vastuulle, jotta ohjelmoija voi keskittyä varsinaisen ohjelman toimintalogiikan suunnittelemiseen. Aktiopohjainen järjestelmä tarjoaa virtuaalimuistin kaltaisen muutoksen rinnakkaiseen ohjelmointiin.

Aktiojärjestelmä on rakennettu 80-luvulla kehitetyn aktioparadigman päälle [1]. Tämän järjestelmän ydinideana on korvata nykyisin suosittu ohjelmoinnin yksikkö, prosessi, uudella yksiköllä, aktio, joka on suunniteltu matalalta tasolta asti rinnakkaistumaan. Ak-

tiojärjestelmässä aktioiden suoritus tapahtuu automaattisesti rinnakkain, jolloin monesta suoritinytimestä saatava suoritusteho saadaan hyödynnettyä paremmin. Aktiojärjestelmän ideana on siis siirtää mahdollisimman suuri osa rinnakkaisuuden hallitsemisesta laitteiston vastuulle. Aktioita tukevaa laitteistoa ei ole vielä toteutettu käytännössä, mutta aktiojärjestelmästä on olemassa C-ohjelmointikielellä toteutettu simulaatio-ohjelma [2] ja Raspberry Pi 2:lle toteutettu testijärjestelmä [3]. Aktiojärjestelmää on suunniteltu käytettävän reaktiivisissa sulautetuissa järjestelmissä.

Aktiojärjestelmällä suoritettavien ohjelmien tekemiseen tarvitaan myös uusi aktioihin perustuva ohjelmointikieli, jota kutsutaan jatkossa lyhyesti aktiokieleksi. Tämän diplomityön tarkoituksena on tehdä alustavaa teoreettista selvitystyötä, miten aktiokielellä toteutettujen komponenttien välinen rajapintatoiminta kannattaisi toteuttaa ja arvioida saavutettujen ratkaisujen soveltuvuutta ja käytettävyyttä aktiokielen osana. Komponentilla tarkoitetaan ohjelman sisäistä kokonaisuutta, jolla on selkeästi määritelty vastuualue ja muusta ohjelmasta erotettu sisäinen toteutus, jonka kanssa komponentin ulkomaailma kommunikoi käyttämällä komponentin määrittelemää käyttörajapintaa. Aktiokielen taustateoria [2] pohjautuu Joint Actions:hin [4], TLA:han (Temporal Logic of Actions) [5] ja ohjelmointikieliin DisCo [6][7] ja Unity [8].

Diplomityön rakenne on seuraava. Luvussa 2 käydään läpi aktiokielen rakenteet: olio, aktio ja subjekti, joiden käyttöä esitellään lyhyillä malliohjelmakoodeilla. Luvun lopussa käydään myös lyhyesti läpi aktiojärjestelmän rakennetta. Luvussa 3 esitellään aktiokieleen lisättävät rakenteet: käyttörajapinta, komponentti ja komponenttiolio, joita käytetään komponenttien määrittelemiseen. Tämän lisäksi luvussa käydään läpi tässä työssä suunniteltuja rajapintamalleja, joissa kuvataan erilaisia ratkaisuja komponenttien rajapintarakenteiden toteuttamiseksi. Luvussa 4 tehdään rajapintamallien kriittistä arviointia ja toisiinsa vertailua, jonka pohjalta suunnitellaan esitys aktiokielessä käytettäväksi standardirajapintarakenteeksi rajapintamalleissa esitettyjen lupaavimpien ideoiden pohjalta. Lopuksi luvussa 5 kootaan päätulokset yhteen.

# 2. AKTIO-OHJELMOINTI

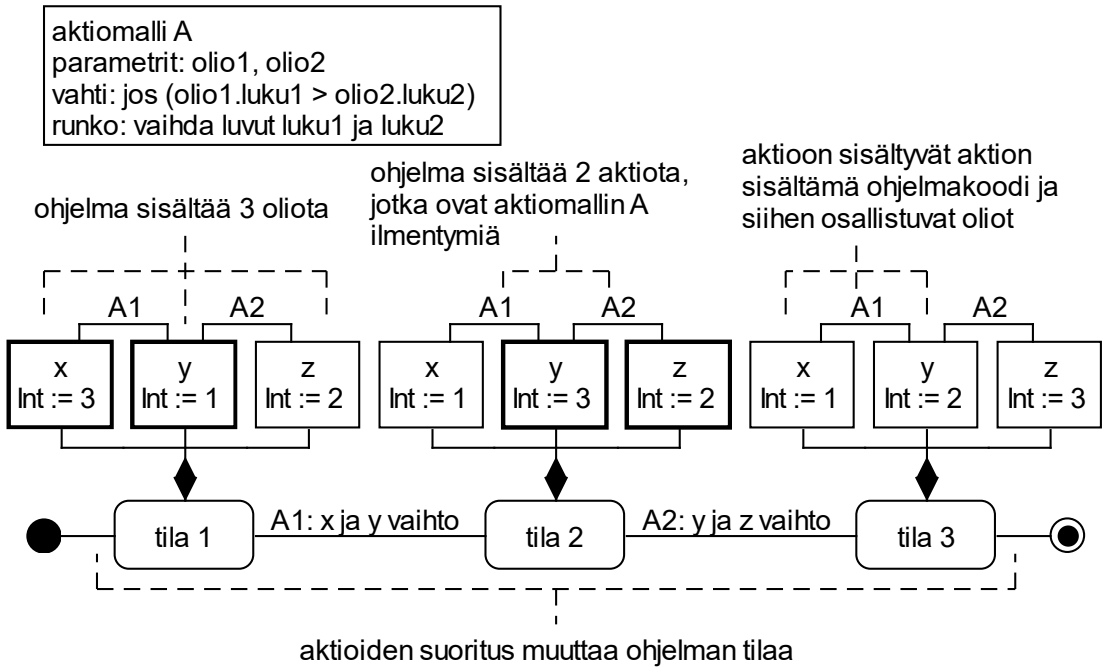
Aktiokielellä tehtävässä aktio-ohjelmoinnissa ohjelmakoodit rakennetaan koostumaan automaattisesti rinnakkain suoritettavista aktioista. Aktio-ohjelmoinnin keskeinen ero moniin nykyisin suosittuihin ohjelmointikieliin on se, että aktiot eivät tue viestinvälitystä tai aliohjelmia. Näiden sijasta ohjelman eteneminen rakentuu aktioille määritellyistä ehdoista, joiden täyttyessä kyseisen aktion ohjelmakoodi voidaan suorittaa. Aktio-ohjelman normaalissa suorituksessa luodaan joukko aktioita, joita suoritetaan rinnakkain sitä mukaa, kun niille määritetyt ehdot täyttyvät ja jokin käytössä olevista suoritinytimistä vapautuu käytettäväksi. Aktiot hyödyntävät siis automaattisesti kaikkia järjestelmässä olevia suoritinytimiä. Luvussa käsiteltävän aktiokielen ja aktiojärjestelmän teoriat pohjautuvat julkaisuihin [9][10].

Aktio-ohjelmoinnin perustana toimii ohjelman tila (*program state*), joka koostuu kaikista ohjelman muuttujien arvoista valitulla hetkellä. Ohjelman muuttujat ovat varastoitu luokista luotuihin olioihin, joiden muuttujia aktiot valvovat ja muokkaavat. Aktiot toimivat ohjelman tilasta toiseen siirtävinä rakenteina ja täten mahdollistavat ohjelman etenemisen. Oliot voidaan ajatella siis ohjelman passiivisiksi tietovarastoiksi ja aktiot aktiiviseksi ohjelman toimintalogiikaksi.

Kuvassa 1 havainnollistetaan korkean tason esimerkki aktioilla ja olioilla toteutetulle lukujen järjestämisalgoritmille. Algoritmissa järjestettävät luvut varastoidaan suorakaiteilla kuvattuihin olioihin *x*, *y* ja *z*, joiden välille luodaan aktiot *A1* ja *A2* aktiomallista *A*. Aktiomalli määrittelee aktion rakenteen samalla tavalla kuin luokka määrittelee olion rakenteen (aktiomallista tarkemmin alaluvussa 2.3). Algoritmissa aktiot ottavat vastuun niille parametreina annettujen olioiden lukujen järjestämisestä sijoittamalla pienemmän luvuista kuvasta katsottuna vasempaan olioon ja suuremman oikeaan. Nämä oliot ovat jaettu aktioille niin, että aktioilla on yksi yhteinen olio, jolloin aktion tekemä muutos olion muuttujaan voi aiheuttaa toisen aktion vahdin ehdon täyttymisen, minkä seurauksena kyseinen aktio suorittaa lukujen uudelleenjärjestämisen. Tällä tavalla aktiot automaattisesti järjestävät luvut suuruusjärjestykseen kuvasta vasemmalta oikealle luettuna aktioiden määrästä riippumatta.

Kuvassa 1 esitetyn ohjelman suorituspolku on aina sama (vasemmalta oikealle), mutta jos luvut jaettaisiin olioille erilailla (kuten 3, 2, 1), olisi mahdollisia suorituspolkuja useita (ensin luvut 3 ja 2 tai 2 ja 1 vaihtavat paikkaa). Merkittävää on myös huomioida ohjelman mahdollisten tilojen lukumäärä. Yleisessä tapauksessa ohjelman mahdollisten tilojen lukumäärä on muuttujien mahdollisten arvojen karteesisen tulon mahtavuus. Vaikka toteutuksessa käsitellään kokonaislukuarvoisia muuttujia, joiden arvoalue on

laaja, on esimerkissä valituilla arvoilla ohjelmassa vain 3 laillista tilaa. Ohjelman mahdollisten tilojen kannalta on siis merkittävämpää huomioida ohjelman toimintalogiikka, kuin teoreettisten tilojen määrä. Kuvassa 1 esitetty algoritmi muutetaan aktiokielellä toteutetuksi ohjelmakoodiksi tulevissa aktiokielen teoriaa käsittelevissä alaluvuissa.




***Kuva 1.*** *Lukujen järjestysalgoritmi toteutettuna aktioilla ja olioilla.*

Tässä luvussa tarkastellaan aktio-ohjelmoinnin teoriaa, ensin esittelemällä aktiokielen perusrakenteet, jota seuraa aktioiden suorittamiseen erikoistuneen aktiojärjestelmän lyhyt esittely. Teoriaa lukiessa on kuitenkin tärkeää huomioida, että aktiojärjestelmä ja aktiokieli ovat molemmat vielä toteutusvaiheessa, minkä takia esiteltävä teoria saattaa muuttua vielä. Tässä työssä sivuutetaan ohjelmien mahdolliset virhetilanteet ja niistä palautuminen, koska niihin liittyvät määritelmät eivät ole vielä täysin valmiita.

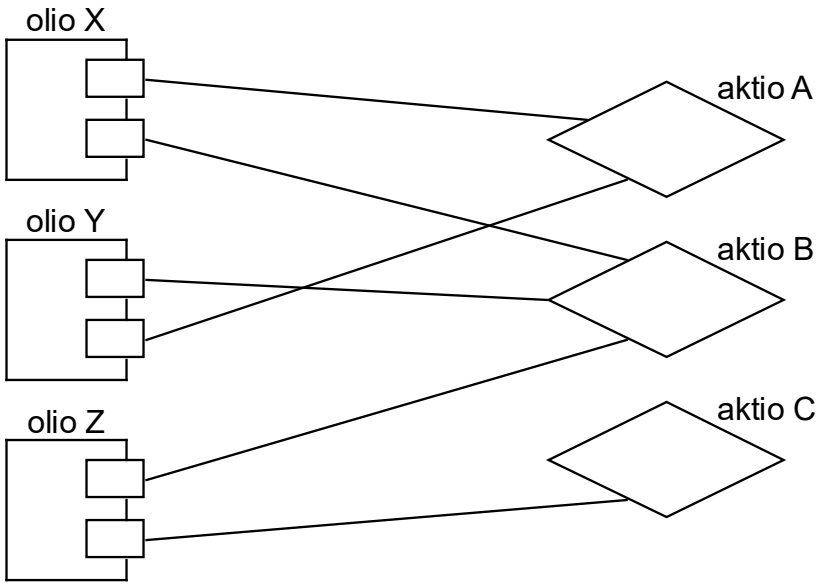
## 2.1 Aktiokieli

Aktiokieli on suunnitteluvaiheessa oleva matalan tason aktio-ohjelmointikieli, jota käytetään aktiojärjestelmän päällä suoritettavien ohjelmien tekemiseen. Aktiokielen perusrakenteita ovat oliot, aktiot ja subjektit. Oliot ja aktiot ovat kielen matalan tason rakenteet, jotka yhdessä muodostavat ohjelmointikielen perustan. Ohjelmoija voi kyseisten rakenteiden avulla luoda niitä vastaavia rakenteita aktiojärjestelmätasolle. Aktiot ja oliot ovat riittävät rakenteet kielellä ohjelmoimiseen, mutta ohjelmakoodimäärän kasvaessa pelkästään niillä ohjelmoiminen voi olla työlästä ohjelmoijalle. Tämän takia subjektirakenne on lisätty aktiokieleen. Subjektit ovat olioista ja aktioista koostuvia aktiokielen

korkean tason rakenteita, joiden päätehtävänä on helpottaa ohjelmoijan työtä. Subjektit sivuutetaan toistaiseksi ja niihin palataan tarkemmin alaluvussa 2.5.

Aktiokielellä toteutetun ohjelman suoritus alkaa luomalla automaattisesti ohjelmakoodissa määritetty alustusaktio (*initially*-niminen aktio). Alustusaktiota voidaan käyttää ohjelman varsinaisen toiminnallisuuden muodostavien olioiden, aktioiden ja subjektien luomiseen. Alustusaktion suorittamisen jälkeen aktioita suoritetaan määräämättömässä järjestyksessä rinnakkain sitä mukaa, kun niiden määrittämät ehdot toteutuvat. Ohjelman suoritus loppuu automaattisesti, kun mitään aktioista ei voida enää suorittaa.

Kuvassa 2 kuvataan olioiden ja aktioiden esittämiseen käytettäviä piirrossymboleja, joita voidaan käyttää aktio-ohjelman ohjelmistoarkkitehtuurin suunnittelemiseen [9][10]. Kuvassa vasemmalla on kuvattuna olioita (luokkia voidaan kuvata samalla symbolilla), joiden aktioille tarjoamat muuttujat esitetään vaaka-asennossa olevilla suorakaiteilla. Kuvan oikeassa laidassa on kuvattuna aktioita, joiden symbolina toimii vaaka-asennossa oleva salmiakki. Aktioon kuuluvat muuttujat yhdistetään aktioon viivalla.




***Kuva 2.*** *Olioiden ja aktioiden kuvaamiseen käytettäviä piirrossymboleja.*

Seuraavissa viidessä alaluvussa käydään läpi aktiokielen perusrakenteiden määrittäminen, joiden käyttöä demonstroidaan lyhyillä ohjelmakoodiesimerkeillä. Näissä ohjelmissa käytetään julkaisuissa [9][10] esitettyä pseudokoodisyntaksia. Yleisesti aktiokielen rakenteet aloitetaan määrittämällä halutun rakenteen tyyppi avainsanalla (esimerkiksi *class*, *action* tai *subject*), jota seuraa rakenteen sisällön määrittäminen, mikä lopetetaan *end*-avainsanaan. Rakenteiden esittelyssä käytetään sanaa ohjelmakoodilohko, jolla tarkoitetaan kahden avainsanan väliin jäävää ohjelmakoodille määritettyä osaa rakenteessa.

## 2.2 Luokat ja oliot

Luokat ovat yksi aktiokielen perusrakenteista, jotka toimivat rakennekuvauksina ohjelman aikana niistä luoduille olioille. Oliot ovat aktiokielen tietovarastorakenteita, joiden sisältämästä datasta muodostuu ohjelman tila. Olioihin voidaan varastoida kaikkia ohjelmointikielen tukemia tietotyyppejä ja tietorakenteita. Oliot eivät sisällä metodeja, joten ne eivät voi yksinään muuttaa ohjelman tilaa.

Ohjelmassa 1 kuvataan luokan määrittely. Luokka rakentuu yhdestä ohjelmakoodilohkosta, joka alkaa *class*-avainsanalla ja päättyy *end*-avainsanaan. Lohkon sisällä määritellään luokan sisältämät tietotyypit. Ohjelmassa 2 esitetään toteutus kuvassa 1 kuvatun ohjelman lukuja varastoivalle luokalle. Ohjelmassa alustusaktio luo kolme oliota *x*, *y* ja *z*, joille asetetaan arvot 3, 1 ja 2.

```
1  // Yleinen luokan rakenne
2  class Example is
3     data 1;
4     data 2;
5     ...
6     data k;
7  end;
```

***Ohjelma 1.*** *Luokan määritteleminen.*

```
1  // Kuvan 1 luokan esittely ja olioiden alustaminen
2  class Node is
3     value: integer;
4  end;
5
6  initially
7     x, y, z: Node;
8     x.value, y.value, z.value := 3, 1, 2;
9  end;
```

***Ohjelma 2.*** *Esimerkkiohjelmakoodi: lukuja varastoivat oliot.*

## 2.3 Aktiot

Aktiot ovat aktio-ohjelmoinnin nimikkorakenne, joiden tehokkaan suorittamisen ympärille aktiojärjestelmä on suunniteltu. Aktiot toimivat aktio-ohjelmoinnin toiminnallisina perusrakenteina, joiden suoritus siirtää ohjelman tilasta toiseen ja täten mahdollistaa ohjelman etenemisen. Aktiojärjestelmä suorittaa aktioita automaattisesti rinnakkain, minkä ansiosta aktioista rakentuvat ohjelmat rinnakkaistuvat automaattisesti muiden aktio-ohjelmien kanssa ja keskenään, mikä on aktio-ohjelmoinnin merkittävin etu suhteessa nykyisin suosittuihin ohjelmointiparadigmoihin.

Aktio-termillä voidaan tarkoittaa kahta eri asiaa riippuen kontekstista. Aktiolla voidaan tarkoittaa järjestelmätasolla suoritettavaa perusyksikköä tai aktio-ohjelmointikielellä

määriteltyä ohjelmakoodia sisältävää rakennetta. Jotta nämä kaksi määritelmää eivät sekoittuisi, niin selkeyden vuoksi tässä diplomityössä aktiokielellä määritellystä aktiorakenteesta käytetään nimitystä aktiomalli, josta voidaan luoda aktiojärjestelmässä suoritettavia aktioita. Tämän lisäksi aktioita käsitellään tekstissä pääasiassa ohjelmoijan näkökulmasta, jolla tarkoitetaan sitä, että jos aktion jokin osallistujista (*participant*) vaihtuu tai samasta aktiomallista luotu aktion instanssi suoritetaan useaan kertaan ohjelman aikana, niin näissä tapauksissa kyseisen aktion ajatellaan olevan sama aktio, vaikka teknisesti kyseessä onkin aina uusi aktio.

Aktiot rakentuvat kahdesta osasta: vahdista (*guard*) ja rungosta (*body*). Vahti määrittää aktiolle ehtolauseen, jonka arvon perusteella se voidaan aktivoida (*enable*). Aktiivinen tila tarkoittaa sitä, että aktio antaa luvan järjestelmälle vuorontaa sen suoritettavaksi vapaalle suoritinytimelle. Aktion vahti toimii siis esiehtona aktion suoritukselle. Aktion runko sisältää aktion toiminnallisuuden määrittelevän ohjelmakoodin, jossa voidaan muokata aktioon osallistuvien olioiden dataa. Ennen rungon ohjelmakoodin suoritusta, aktiojärjestelmän vuorontaja (*scheduler*) lukitsee kaikki suoritettavan aktion osallistujat, mikä varmistaa kriittisen alueen poissulkemisen. Näin ollen aktioita voidaan suorittaa huoletta rinnakkain ennalta määräämättömässä järjestyksessä. Jos aktiivisen aktion jokin osallistujista on jo lukitussa tilassa, niin silloin aktio jää aktiiviseen tilaan odottamaan vuoroaan vuorontajalta.

Aktioiden vahti jaetaan kahteen osaan, lokaaliksi (*local guard*) ja yleiseksi vahdiksi (*common guard*). Lokaali vahti viittaa aktion yhteen tiettyyn osallistujaan ja yleinen vahti voi viitata useampaan osallistujaan. Vahdin jako kahteen erilliseen osaan on tehty tehokkuussyistä [9]. Lokaali ja yleinen vahti muodostavat yhdessä ehtolauseen, jonka tuottama arvo voi olla tosi tai epätosi, mikä määrittää milloin aktio voidaan laittaa aktiiviseen tilaan. Ehtolauseen eri osat voivat saada myös määrittelemättömän (*undefined*) arvon, mutta tässä tapauksessa koko ehtolauseen arvo tulkitaan kuitenkin joko todeksi tai epätodeksi tilanteesta riippuen. Epätoden tapauksessa aktio laitetaan epäaktiiviseen tilaan odottamaan muutoksia aktion osallistujiin. Kun toinen aktio tekee muutoksia epäaktiivisen aktion ehtolauseeseen kuuluviin muuttujiin, niin silloin muutokset tehnyt aktio automaattisesti laajennetaan kattamaan epäaktiivisen aktion vahdin arvon uudelleen laskeminen.

Ohjelmassa 3 kuvataan aktiomallin määrittely. Aktiomalli rakentuu kahdesta ohjelmakoodilohkosta, vahdista ja rungosta, jotka erotetaan toisistaan *body*-avainsanalla. Vahtilohkossa määritetään aktion osallistujat ja aktion aktivoiva ehtolause, ja runkolohkossa määritetään aktion varsinainen toiminnallisuus. Ohjelmassa 4 esitetään toteutus kuvassa 1 kuvatun ohjelman lukuja järjestävälle aktiolle. Ohjelmassa kaksi *swapNumbers*-aktiota ottavat parametreina osallistujiksi kaksi *Node*-tyyppistä oliota, joihin varastoidut luvut aktiot järjestävät suuruusjärjestykseen sijoittaen *left*-olioon pienemmän ja *right*-olioon suuremman luvuista. Näiden aktioiden tekemät muutokset olioihin aiheuttavat samoja olioita vahtivien aktioiden virittymisen, joiden suoritus loppuu, kun kaikki luvut

ovat järjestetty suuruusjärjestykseen. Samaa periaatetta voidaan käyttää esimerkiksi järjestämään halutun tietorakenteen alkiot automaattisesti jokaisen muutoksen jälkeen.

```
1  // Yleinen aktiomallin rakenne
2  action example is
3     participants 1 ... k;
4     <<guard>>
5  body
6     <<action logic>>
7  end;
```

***Ohjelma 3.*** *Aktiomallin määritteleminen.*

```
1  // Kuvan 1 aktion määrittely ja aktioiden alustaminen
2  action swapNumbers is
3     participant Node as left;               // Viite ensimmäiseen olioon
4     participant Node as right;              // Viite toiseen olioon
5     common guard left.value > right.value;  // Määritetään ehtolause
6  body
7     left.value, right.value := right.value, left.value;
8  end;
9
10 initially
11    // Käytetään ohjelmassa 2 alustettuja olioita
12    create swapNumbers(x, y);
13    create swapNumbers(y, z);
14 end;
```

***Ohjelma 4.*** *Esimerkkiohjelmakoodi: lukuja järjestäviä aktioita.*

## 2.4 Aktioilla ohjelmointi

Aktioilla ohjelmoiminen vaatii ohjelmoijalta erilaisen lähestymisnäkökulman lukea ja kirjoittaa ohjelmakoodia kuin monet tällä hetkellä suositut ohjelmointikielet, joissa yleensä suositaan imperatiivista ohjelmointiparadigmaa. Imperatiivisessa ohjelmoinnissa ohjelmakoodin suoritus etenee käsky kerrallaan rivi riviltä, johon ohjelmoija voi vaikuttaa erilaisilla lauseilla (*statement*) kuten ehtolauseilla tai hyppykäskyillä. Aktio-ohjelmoinnissa ohjelmakoodin suoritusjärjestyksestä vastaavat aktioiden vahdit, joiden rungossa oleva ohjelmakoodi suoritetaan imperatiivisesti. Ohjelmoijan vastuulla on määrittää aktioiden vahdit niin, että aktiot suoritetaan ohjelman kannalta mielekkäässä järjestyksessä. Monen yhtäaikaisen aktiivisen aktion tapauksessa ohjelmointiympäristö vastaa aktioiden suoritusjärjestyksestä, minkä ei pitäisi haitata järkevästi toteutetun ohjelman lopputulosta.

Aktio-ohjelmoinnin havainnollistamiseksi ohjelmassa 5 kuvataan miltä Fibonaccin lukuja laskeva ohjelmakoodi voisi näyttää aktioilla ja olioilla toteutettuna. Ohjelmassa oliot *n0* ja *n1* sisältävät yhteenlaskettavat luvut, joiden välille määritetty *addNumbers*-aktio iteroi Fibonaccin lukuja kunnes määritelty iteraatiomäärä on suoritettu. Iteraatios-

sa yhteenlaskettu luku varastoidaan oliolle *n1*, jonka edellinen arvo siirretään talteen oliolle *n0*. Lopuksi ohjelma tulostaa lopputuloksen käyttäjälle.

```
1  class Number is
2     value: integer;
3  end;
4
5  action addNumbers is
6     Number as n0;
7     Number as n1;
8     Number as iteration;
9     Number as end;
10    common guard is iteration.value != end.value;
11 body
12    n0.value, n1.value := n1.value, (n0.value + n1.value);
13    ++iteration.value;
14 end;
15
16 action result is
17    Number as iteration;
18    Number as end;
19    common guard is iteration.value == end.value;
20 body
21   print result;
22 end;
23
24 initially
25   n0, n1, iteration, end: Number;
26   n0.value, n1.value := 0, 1;
27   iteration.value, end.value := 0, random;
28
29   create addNumbers(n0, n1, iteration, end);
30   create result(iteration, end);
31 end;
```

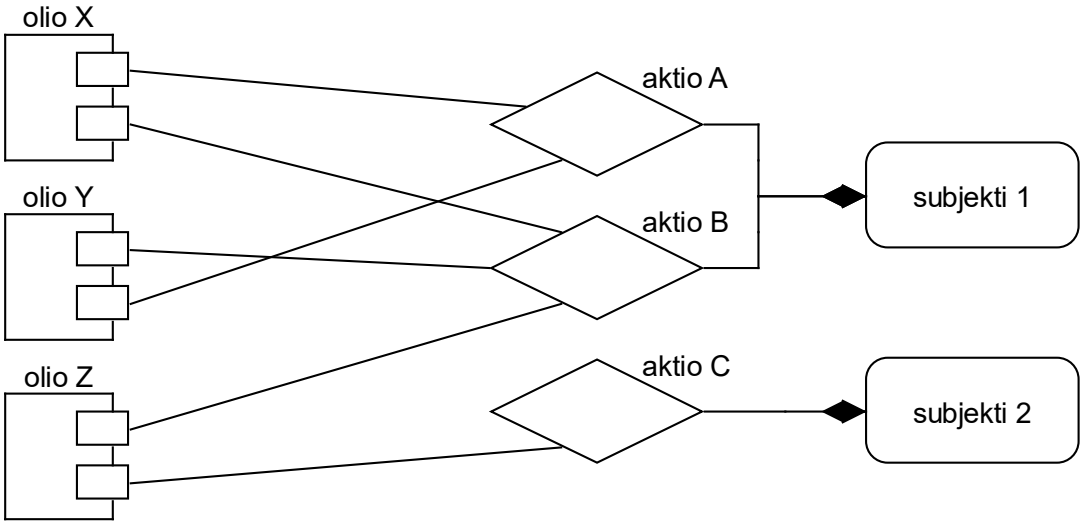
***Ohjelma 5.*** *Fibonaccin lukuja laskeva ohjelmakoodi aktioilla ja olioilla toteutettuna.*

Korkean tason yleistyksenä aktioilla toteutetun ohjelman voidaan ajatella rakentuvan globaaleista ehtolauseista (aktioiden vahdit), joiden täyttyessä aktiot muokkaavat globaaleja muuttujia (olioita). Näin ajateltuna paljastuu yksi aktioilla ohjelmoimisen heikkouksista, nimittäin koodin paikoitellen vaikea seurattavuus. Aktiot eivät muodosta rivi riviltä etenevää ohjelmakoodia, minkä takia ohjelman oikean toiminnallisuuden todentaminen voi tuottaa ongelmia koodirivimäärän kasvaessa. Tämä on huolestuttava merkki aktiokielen käytettävyyden kannalta. Aktiokielellä on pystyttävä tuottamaan helposti kirjoitettavaa ja luettavaa koodia, jotta ohjelmointikieli voitaisiin ottaa käyttöön laajamittaisesti. Tämän ongelman ratkaisuksi aktiokieleen on määritelty olioiden erikoistapaus, subjektirakenne.

## 2.5 Subjektit

Subjekti on ohjelmoijalle tarkoitettu aktiokielen korkean tason perusrakenne, jonka ideana on muuttaa ohjelman aktioajattelumalli muistuttamaan enemmän imperatiivista

ohjelmointia. Subjektit esittävät ohjelmoijalle kuvitelman rivi riviltä suoritettavasta ohjelmakoodista, joka todellisuudessa rakentuu aktioista ja olioista niin kuin aikaisemmin esitetyt esimerkit. Aktio-ohjelman käännösvaiheessa kääntäjä ensin purkaa subjektit olioiksi ja aktioiksi, minkä jälkeen ohjelmasta tuotetaan suoritettava binääri. Kuvassa 3 on esitetty olioiden, aktioiden ja subjektien muodostama rakenne.




***Kuva 3.*** *Subjektit koostuvat olioista ja aktioista.*

Subjektit ovat käytännössä komponentteja, joiden toiminta perustuu implisiittiseen oliorajapintaan. Subjekti rakentuu yhdestä ohjelmakoodilohkosta, niin kuin luokka, jonka sisällä määritetään subjektin ohjelmakoodi. Subjektin sisällä voidaan määritellä pysyviä muuttujia ohjelmaan (koska ne ovat todellisuudessa olioiden muuttujia), toistorakenteita ja odotuslausekkeita, jotka pakottavat subjektin odottamaan vaaditun odotusehdon ennen kuin subjektin ohjelmakoodin suoritusta voidaan edistää. Odotuslausekkeiden ja toistorakenteiden avulla ohjelmoija voi varmistua määrättyjen ehtojen toteutumisesta tietyssä vaiheessa subjektin ohjelmakoodia. Ohjelmassa 6 kuvataan subjektin määrittely.

```
1  // Yleinen subjektin rakenne
2  subject Example(parameter 1, parameter 2, ... parameter k) is
3  begin
4     <<subject logic>>
5
6     // Odotusehdon määritteleminen
7     wait until <<conditional expression>> then
8        ...
9     end wait;
10
11    // Subjektin logiikkaa voidaan ohjata toistorakenteilla
12    forever do ... end do;
13    while ... end loop;
14    for ... end loop;
15 end Example;
```

***Ohjelma 6.*** *Subjektin määritteleminen.*

Ohjelmassa 7 esitetään subjektilla tehty toteutus kuvassa 1 kuvatulle algoritmille. Ohjelmassa oliot ja aktiot ovat korvattu yhdellä subjektilla, joka käsittelee silmukassa sille annettuja lukuja. Silmukassa määritettyjen ehtojen täyttyessä suoritetaan kyseistä ehtoa vastaava valintalausekkeen (*select*) haara, jossa vaihdetaan kyseiseen ehtoon liittyvät luvut oikeaan järjestykseen. Subjektin suoritus loppuu, kun luvut ovat suuruusjärjestyksessä.

```
1  // Lukujen järjestysalgoritmi
2  subject SortNumbers(x, y, z: integer) is
3  begin
4     while x < y or y < z loop
5        select
6           when x < y do
7              x, y := y, x;
8           end;
9        or
10          when y < z do
11             y, z := z, y;
12          end;
13       end select;
14    end loop;
15 end SortNumbers;
16
17 initially
18    create SortNumbers(3, 1, 2);
19 end;
```

***Ohjelma 7.*** *Esimerkkiohjelmakoodi: lukuja järjestävä subjekti.*

Ohjelmassa 7 esitetyn subjektin avulla numeroita järjestävä algoritmi pystyttiin määrittelemään yhden rakenteen avulla, mikä helpottaa ohjelman luettavuutta ja ohjelman suorituksen etenemisen seuraamista merkittävästi verrattuna olioilla ja aktioilla toteutettuun versioon kyseisestä algoritmista. Subjektien avulla ohjelmoijan ei tarvitse enää tulkita aktiomallien vahteja, vaan hän voi suoraan lukea odotuslausekkeiden ja toistorakenteiden ehtoja ja päätellä niistä missä vaiheessa ohjelman eteneminen on halutulla hetkellä. Subjektit helpottavat ja selkeyttävät siis aktiokielen käytettävyyttä huomattavasti. Sub-

jekteja käytettäessä tulee kuitenkin kiinnittää erityistä huomiota lopputuloksen rinnakkaistumiseen, koska esimerkiksi ohjelman 7 subjekti ei enää suorita alkuperäistä algoritmia rinnakkain.

## 2.6 Subjekteilla ohjelmointi

Ohjelma 8 toteuttaa saman toiminnallisuuden kuin aikaisemmin esitelty Fibonaccin lukuja järjestävä ohjelma 5 käyttämällä subjektia olioiden ja aktioiden sijasta. Ohjelma rakentuu Fibonacci-subjektista, jonka sisältämä silmukka laskee halutun määrän Fibonaccin lukuja ja tulostaa lopullisen tuloksen käyttäjälle. Ohjelman suorituspolku etenee imperatiivisesti, mikä tekee ohjelman suorituksen seuraamisesta vaivatonta.

```
1  subject Fibonacci(iteration_end: integer) is
2  begin
3     iteration: integer := 0;
4     n0, n1: integer := 0, 1;
5
6     while iteration != iteration_end loop
7        n0, n1 := n1, (n0 + n1);
8     end loop;
9
10    print result;
11 end Fibonacci;
12
13 initially
14   create Fibonacci(x);
15 end;
```

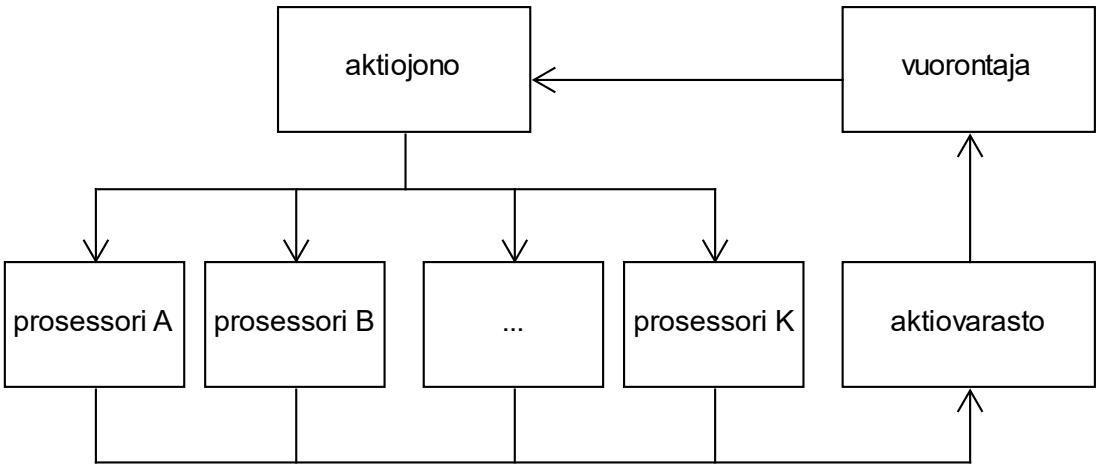
***Ohjelma 8.*** *Fibonaccin lukuja laskeva ohjelmakoodi subjektilla toteutettuna.*

Ohjelmat 5 ja 8 ovat molemmat valideja aktiokielellä toteuttavissa olevia ohjelmia, joissa kummassakin on omat hyvät puolensa. Subjektin avulla ohjelmakoodista saatiin lyhyempi ja ohjelman rakenne selkeytyi helpommin luettavaksi kokonaisuudeksi. Ohjelman yksinkertaisuuden takia koodimäärän väheneminen ei ole olennainen etu kyseisen toteutuksen tapauksessa, mutta laajempaa ohjelmaa ajateltaessa se auttaa ohjelmoijien tuottavuudessa. Ohjelman rakenteen selkeyttäminen on toisaalta merkittävä etu. Subjektin avulla ohjelmoija pystyy saamaan kuvan ohjelman suorituspolusta yksinkertaisesti lukemalla ohjelmakoodia rivi riviltä. Aktioiden tapauksessa ei yleisesti ole olemassa selkeää paikkaa, mihin ohjelma seuraavaksi etenee, vaan ohjelmoijan on pystyttävä tunnistamaan seuraavaksi aktivoituva aktio aktioille määritellyistä ehdoista. Tilannetta monimutkaistaa myös se, että aktiivisia aktioita on monesti useita yhtä aikaa. Toisaalta aktioiden avulla ohjelmoija saa tarkalleen määritettyä mitä ohjelma tulee tekemään, jolloin taitava ohjelmoija pystyy saamaan kaiken hyödyn irti ohjelmointikielestä. Subjekteilla ohjelmoitaessa kääntäjä huolehtii olioiden ja aktioiden luomisesta, jolloin ohjelman todellinen matalan tason toteutus piiloutuu ohjelmoijalta.

## 2.7 Aktiojärjestelmä

Aktio-ohjelmoinnissa laitteistolla on merkittävä rooli aktioista saatavien etujen realisoinnissa. Rinnakkaisen ohjelmoinnin vaatima synkronisointimekanismi (*synchronization*) toteutuu automaattisesti aktioiden vahtien avulla ja aktioiden suoritusjärjestyksestä vastaava vuorontaja ottaa vastuun ohjelman muuttujien poissulkemisen (*mutual exclusion*) toteuttamisesta. Ohjelmoijan ei tarvitse myöskään huolehtia viestienvälityksestä (*message passing*), koska aktiojärjestelmässä ei ole olemassa prosesseja.

Aktioita käsittelevä testijärjestelmä on jo toteutettu ja sen rakenne on kuvattu kuvassa 4 [9]. Kyseinen järjestelmä koostuu neljästä eri toiminnallisesta yksiköstä: vuorontajasta, aktiojonosta, prosessoreista ja aktiovarastosta. Vuorontajan tehtävänä on vuorontaa aktioita perustuen aktioiden tilaan. Vuoronnetut aktiot sijoitetaan aktiojonoon, josta vapaana olevat prosessorit ottavat niitä suoritettavaksi. Suoritettavilla aktioilla on atomisuus-ominaisuus, eli ne suoritetaan kokonaan tai kaikki niiden muutokset hylätään. Aktion onnistuneen suorituksen jälkeen sen tekemät muutokset kopioidaan prosessorin välimuistista ohjelman tilan muutokseksi, jonka jälkeen aktio palautetaan aktiovarastoon. Aktiovarastossa olevat aktiot odottavat muutoksia niiden vahtien muuttujiin, jonka perusteella ne virittyvät ja vuoronnetaan uudelleen suoritukseen. Kuvassa 4 on esitetty testiaktiojärjestelmä kaaviona.




***Kuva 4.*** *Vuorontaja, aktiojono, prosessori(t) ja aktiovarasto muodostavat aktiojärjestelmän.*

Kuvasta 4 on nähtävissä yksi aktiojärjestelmän merkittävimmistä eduista, skaalautuvuus. Kyseiseen järjestelmään voidaan lisätä tai poistaa prosessoreita mielivaltaisesti ja niistä saatava suoritusteho realisoituu suoraan aktio-ohjelmien suoritusnopeudeksi. Aktio-ohjelman näkökulmasta on samantekevää, kuinka monta suoritinydintä järjestelmässä on, koska aktiot ovat suunniteltu matalalta tasolta asti suoritettavaksi rinnakkain. Teoriassa tämä tarkoittaa sitä, että järjestelmään voidaan lisätä suoritinytimiä esimerkik-

si ohjelman suorituksen aikana, jolloin niistä saatava suoritusteho otetaan automaattisesti käyttöön ohjelman suorituksessa. Samalla käyttämättömät prosessorit voidaan laittaa horrostilaan, jolloin saadaan säästöjä myös sähkönkulutuksessa [10]. Teoreettisesti kuvan 4 rakenne mahdollistaa ohjelman suorituksen jakamisen suoraan myös verkon yli, mutta tämä vaatii vielä lisää tutkimusta.

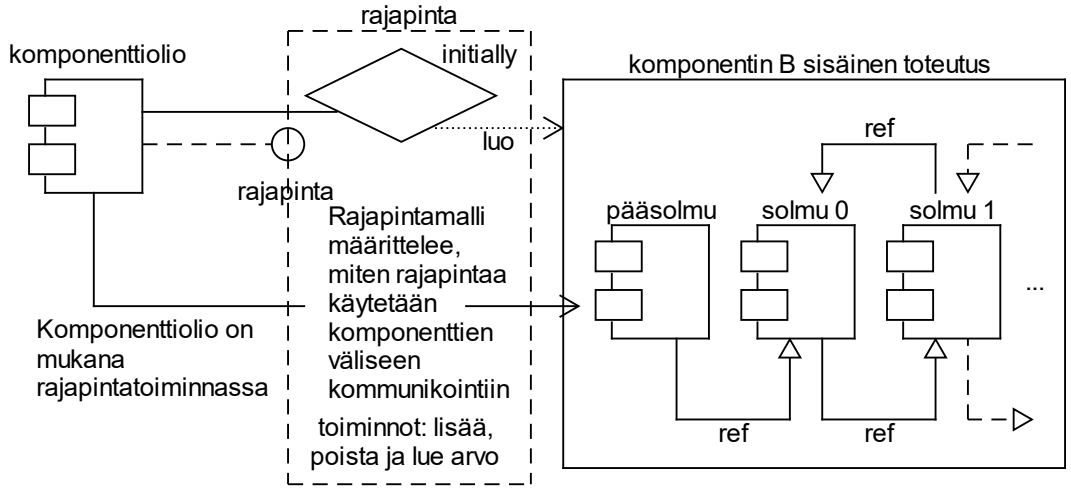
# 3. KOMPONENTTIEN RAJAPINNAT

Oliot, aktiot ja subjektit muodostavat aktiokielen perusrakenteet, joiden avulla voidaan toteuttaa aktio-ohjelman toimintalogiikka. Niillä pystytään rakentamaan järkevästi pienen skaalan ohjelmia ja algoritmeja, kuten edeltävässä luvussa esitetyt esimerkkiohjelmat, mutta ne eivät ole yksinään riittäviä ohjelman skaalan kasvaessa laajemmaksi. Perusrakenteilla ohjelmoitaessa ongelmaksi muodostuu ohjelmakoodin hajanaisuus, koska pelkästään perusrakenteita hyödyntämällä ohjelmaan on haastavaa kuvata helposti ymmärrettäviä isoja kokonaisuuksia, komponentteja. Tässä diplomityössä tällaisten kokonaisuuksien toteuttamiseksi aktiokieleen määritellään kolme uutta rakennetta, komponentti (*component*), komponenttiolio (*component object*) ja käyttörajapinta (*interface*), jotka esitellään tarkemmin alaluvussa 3.1.

Komponenttien välinen rajapintatoiminta kuvataan tässä diplomityössä rajapintamallien muodossa. Rajapintamallien suunnittelussa lähdetään tavoitteesta rakentaa malleissa esitetyt toimintaperiaatteet pohjautumaan mahdollisimman paljon aktiokielen matalan tason rakenteisiin, olioihin ja aktioihin. Näin minimoidaan aktiokieleen tarvittavien muutosten tekeminen ja saadaan hyödynnettyä mahdollisimman paljon näiden rakenteiden jo olemassa olevia ominaisuuksia. Näihin rakenteisiin pohjautuminen vähentää myös ohjelmoijien tarvetta opetella uusia rakenteita heidän hyödyntäessään rajapintamalleissa esitettyjä ideoita. Rajapintamalleissa ei siis käytetä subjekteja, mutta monet rajapintamallien ideat voidaan muuttaa suoraan subjekteiksi ohjelmoijan niin halutessaan. Olioita ja aktioita täydennetään rajapintamalleissa uusilla aktiokieleen lisättävillä rakenteilla tarpeen mukaan.

Komponenttien välisen rajapintatoiminnan tarkka rakenne tulee selkeytymään, kun aktiokieltä lähdetään konkreettisesti toteuttamaan, minkä takia tässä vaiheessa aktiokielen suunnittelua rajapintarakennetta ei haluta rajoittaa pelkästään yksittäiseen rajapintamalliin. Täydellisen yksittäisen ratkaisun etsimisen sijasta rajapintamallien suunnittelun tavoitteena on esittää mahdollisimman monta erilaista näkökulmaa rajapintarakenteen toteutukselle ja arvioiden niiden soveltuvuutta osaksi aktiokieltä. Tällä tavalla saadaan mahdollisimman laaja näkemys käsiteltävään aiheeseen, mitä voidaan hyödyntää myöhemmin aktiokielen toteutuksen yhteydessä. Vaikka jotkin rajapintamalleista osoittautuisivatkin käyttökelvottomiksi jo tässä diplomityössä, niin niistä voi silti löytyä joitakin toimivia osia, joita voidaan mahdollisesti hyödyntää jollain muulla tavalla osana aktiokieltä. Käyttökelpoisen rajapintamallin tulisi tukea aktio-ohjelmoinnin erilaista luonnetta ja samalla tarjota helposti laajennettava pohja mahdollisille tulevaisuuden muutoksille ohjelmointikieleen.

Rajapintamallien soveltuvuutta komponenttien välisen kommunikaation toteuttamiseen havainnollistetaan suunnittelemalla niillä esimerkkitoteutus taulukkotietorakenteelle. Taulukkoa käytetään esimerkkirakenteena, koska se sisältää selkeän sisäisen toteutuksen, johon voidaan kohdistaa monenlaisia pyyntöjä muilta komponenteilta, joita muiden komponenttien tulee pystyä tekemään sujuvasti käyttörajapinnan avulla. Ohjelmaesimerkeissä taulukon oletetaan toteuttavan toiminnot *lisää*, *poista* ja *lue arvo*, joista keskitytään lukemisoperaation toteutukseen. Lukemisoperaation esitteleminen ohjelmakoodeissa on käytännössä riittävä, koska siinä yhdistyvät tiedon välittäminen taulukolle (luettava indeksi) ja tuloksen palauttaminen rajapinnan käyttäjälle (arvo halutussa kohdassa taulukkoa). Taulukon oletetaan koostuvan solmuolioista, joihin on varastoitu kokonaislukuarvoinen muuttuja kuvaamaan taulukon sisältöä, referenssit (*reference*) seuraavaan ja edelliseen solmuun taulukossa ja tieto siitä, kuinka mones alkio solmu on taulukossa. Referenssien avulla muut rakenteet pääsevät käsiksi viitattuun rakenteeseen, mutta estävät kyseisen rakenteen muokkaamisen suoraan sen läpi. Taulukon solmuihin pääsee käsiksi pääsolmun avulla, johon taulukon instanssilla (komponenttioliolla, tarkemmin alaluvussa 3.1) on referenssi. Kuvassa 5 on esitetty esimerkkitaulukon rakenne. Jatkossa, kun tekstissä mainitaan esimerkkitaulukko, niin silloin viitataan tässä kappaleessa esiteltyyn taulukkorakenteeseen tai variaatioon siitä.




***Kuva 5.*** *Esimerkkitaulukon rakenne.*

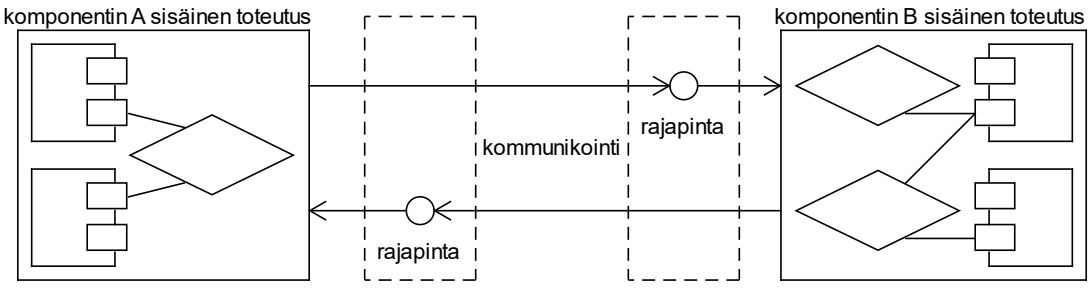
Tässä luvussa esitellään yhteensä kuusi erilaista rajapintamallia alaluvuissa 3.2 – 3.7. Rajapintamallit esitellään järjestyksessä suhteessa siihen, mitä aktiokielen perusrakennetta on hyödynnetty niiden toteutuksen päärakenteena. Ensimmäiset kaksi mallia ovat oliomalleja, joita seuraa kaksi aktiomallia ja lopuksi esitellään kaksi olioiden ja aktioiden yhteistoiminnan ympärille rakennettua yhdistelmämallia. Rajapintamallien esittelyissä keskitytään pääasiassa käsittelemään mallien rakenteita, joita tutkitaan kriittisesti ja toisiinsa vertaillen luvussa 4.

## 3.1 Komponentti, komponenttiolio ja käyttörajapinta

Ennen rajapintamallien suunnittelemista aktiokieleen tarvitaan rakenteet toteuttamaan komponentteja osaksi aktio-ohjelmaa. Tämän takia tässä työssä määritellään kolme uutta rakennetta aktiokieleen: komponentti, komponenttiolio ja käyttörajapinta, joiden tarkoituksena on luoda selkeä pohja aktiokielellä määriteltävien kokonaisuuksien toteuttamiseen. Näiden rakenteiden päälle voidaan suunnitella tässä työssä käsiteltävät rajapintamallit. Näissä rakenteissa on tarkoituksenmukaisesti tavoiteltu minimalistista toteutusta, jotta niiden toteutukset eivät vaikuttaisi ja ohjaisi merkittävästi rajapintamallien suunnittelua.

Komponentti on yhdestä ohjelmakoodilohkosta rakentuva perusrakenne, jota voidaan käyttää kokoamaan tiettyyn toiminnalliseen kokonaisuuteen kuuluvat rakenteet yhden yhteisen nimen alle. Ohjelmakoodi on mielekästä jakaa komponentteihin, koska niiden avulla ohjelma saadaan ositettua selkeisiin itsenäisiin kokonaisuuksiin, jotka voidaan suunnitella ja toteuttaa toisistaan erillään. Komponentit auttavat ohjelmakoodin hajanaisuuden vähentämisessä, koska ohjelmoijan on helppo hahmottaa tiettyyn toiminnallisuuteen liittyvät rakenteet lukemalla komponentin määritelmä, minkä lisäksi komponenteille voidaan määritellä omat selkeät vastuualueensa, joiden järkevästä toteutuksesta ne ottavat vastuun. Tämä helpottaa ohjelman luettavuutta merkittävästi ja komponenttien avulla saadaan myös helpommin toteutettua uudelleen käytettävää ohjelmakoodia. Komponentin instantioiminen luo komponentin luojalle komponenttiolion, joka edustaa kyseistä komponentin ilmentymää. Komponenttioliota käytetään komponentin rajapinnassa määriteltyjen rakenteiden luomiseen.

Komponenttien välille muodostuvien suorien riippuvuussuhteiden vähentämiseksi aktiokieleen lisätään myös uusi käyttörajapintarakenne, jonka avulla komponenttien väliset suorat riippuvuussuhteet voidaan siirtää rajapintoihin. Käyttörajapinnat määrittelevät komponentin ulkomaailmalle tarjoamat toiminnallisuudet, joiden varsinaiset toteutukset määritellään komponenttirakenteessa. Tällä tavalla komponentti voi piilottaa sisäisen toteutuksensa rajapinnan käyttäjältä, mikä mahdollistaa myös komponentin sisäisen toteutuksen helpon vaihtamisen tarvittaessa. Käyttörajapinnat ovat yleisesti ohjelmoinnissa tuettu ratkaisu ohjelman sisäisten kokonaisuuksien välisen informaation jakamiseen, minkä takia aktiokielessä ei todennäköisesti ole järkevää lähteä kilpailemaan ohjelmoijien jo oppimaa toimintaperiaatetta vastaan. Kuvassa 6 on esitetty rajapintarakenteen idea. Komponenttien välinen kommunikointi määritellään rajapintamalleissa.




***Kuva 6.*** *Komponentit kuvaavat ohjelman sisäisiä kokonaisuuksia, joiden välisiä suoria riippuvuussuhteita voidaan vähentää rajapinnoilla.*

Ohjelmassa 9 esitellään käyttörajapinnan määrittely. Käyttörajapinta koostuu yhdestä ohjelmakoodilohkosta, joka sisältää komponentin ulkomaailmalle esiteltävät rakenteet, joiden varsinainen toteutus kuvataan komponenttirakenteessa. Ohjelmassa 10 esitellään rajapinnan toteuttavan komponentin määrittely. Komponentti koostuu yhdestä ohjelmakoodilohkosta, jonka sisällä määritellään perusrakenteiden lisäksi komponentin alustusaktio ja komponenttiluokka (*component class*), joka määrittelee komponentin luomisen yhteydessä luotavan komponenttiolion rakenteen.

```
1  // Yleinen käyttörajapinnan rakenne
2  interface Example is
3     <<Class declarations>>
4     <<Action declarations>>
5     <<Subject declarations>>
6  end Example;
```

***Ohjelma 9.*** *Käyttörajapinnan määritteleminen.*

```
1  // Yleinen komponentin rakenne
2  component Example is
3     <<Class declarations>>
4     <<Action declarations>>
5     <<Subject declarations>>
6
7     component class is ... end;
8     initially is ... end;
9  end Example;
```

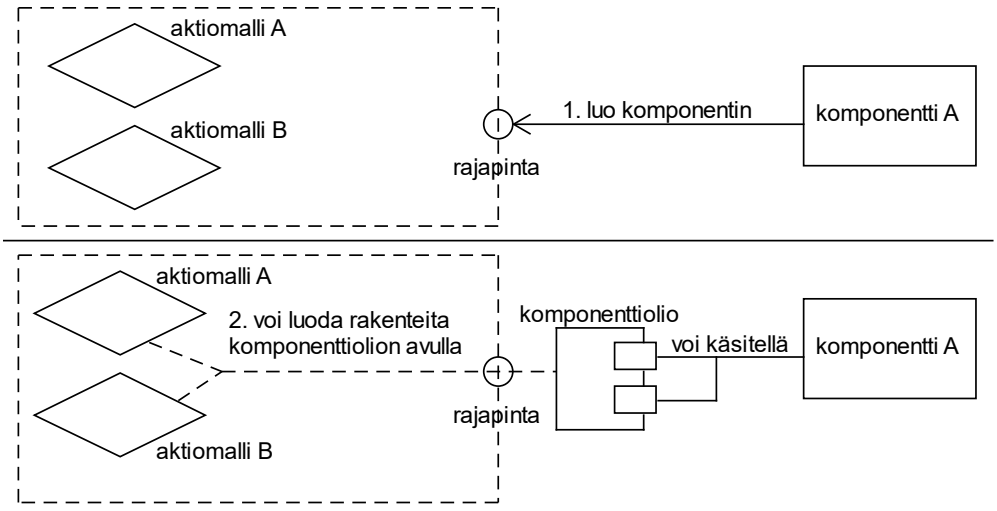
***Ohjelma 10.*** *Komponentin määritteleminen.*

Ohjelmassa 11 on kuvattu esimerkkitaulukon sisäiset luokat. Ohjelmassa *MainNode*-luokka edustaa taulukon pääsolmua ja *Node*-luokka tietoa varastoivia solmuja. Myöhemmin käsiteltävissä esimerkkitaulukkoon liittyvissä ohjelmaesimerkeissä jätetään ohjelmassa 11 esitetyt luokat esittelemättä, koska ne pysyvät samoina kaikissa esimerkeissä.

```
1  // Esimerkkitaulukon sisältämät luokat
2  component Array is
3     class MainNode is
4        firstNode: &Node := NULL;
5     end;
6
7     class Node is
8        value, index: integer;
9        before, next: &Node := NULL;
10    end;
11
12    component class is
13       mNode: &MainNode := NULL;
14    end;
15
16    initially
17       create MainNode as head; mNode->head;
18    end;
19 end Array;
```

***Ohjelma 11.*** *Esimerkkitaulukon rakenne koostuu tietoa varastoivista solmuista.*

Komponenttia edustava komponenttiolio on normaali aktiokielen olio, jolle on määritelty ylimääräiseksi tehtäväksi toimia komponentin instanssin edustajana. Komponentin instantioiminen luo luomiskomennossa määritellyn nimen mukaisen komponenttiolion ja rajapinnan toteuttajan puolella määritetyn alustusaktion, jonka osallistujaksi komponenttiolio asetetaan automaattisesti. Komponenttiolio luodaan ennen komponentin alustusaktion suorittamista, jolloin alustusaktiossa voidaan muokata komponenttiolion muuttujia suoraan. Alustusaktion suorittamisen jälkeen muut komponentit voivat luoda komponentin rajapinnassa määriteltyjä rakenteita määrittelemällä komponenttiolion nimen rakenteen luontikomennossa. Kuvassa 7 on esitetty komponenttiolion toimintaperiaate.




***Kuva 7.*** *Komponenttiolio edustaa komponentin instanssia.*

Komponenttiolio vastaa komponentin elinkaaresta poistamalla kaikki komponenttiin liittyvät rakenteet samalla, kun komponenttiin liittyvä komponenttiolio poistetaan. Tämän lisäksi komponenttiolio ottaa vastuun tiedostaa, milloin komponentin alustusaktio on suoritettu onnistuneesti, jota ennen tapahtuvat rajapintatoiminnot se estää. Komponenttioliolla voidaan myös hallita komponentin saavutettavuutta luomalla kopioita siitä tai jakamalla muille komponenteille referenssejä siihen. Komponenttiolion monistaminen luo siitä ja alustusaktiosta kopion, joka virittyy automaattisesti kopioimisen seurauksena ja suorittaa uuden komponentin alustuksen.

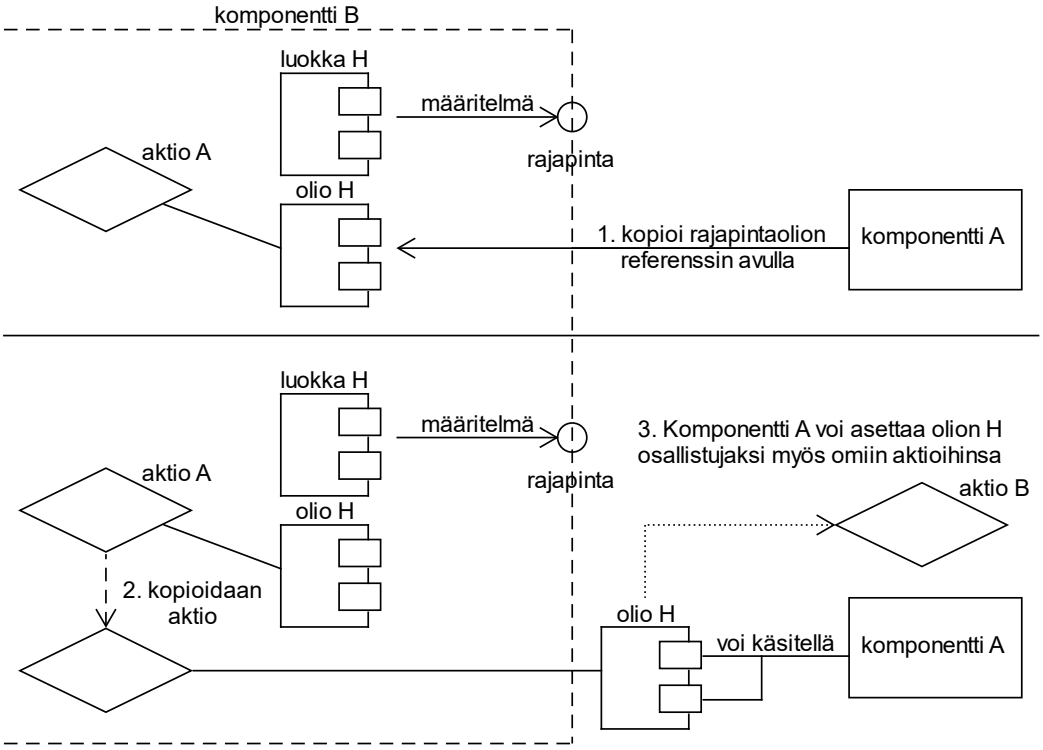
Komponenttioliota voidaan käyttää toteuttamaan myös monia komponentille hyödyllisiä toiminnallisuuksia. Komponenttiolio on mukana komponentin rajapinnassa esiteltyjen rakenteiden luonnissa, jolloin luotavissa rakenteissa voidaan esimerkiksi kopioida tietoja komponenttioliosta ja alustaa muuttujia komponenttiolioon varastoitujen tietojen perusteella. Komponenttioliota voidaan siis esimerkiksi käyttää komponentin yksityisten (*private*) muuttujien tietovarastona. Komponenttiolioon varastoitua dataa voidaan muuttaa ainoastaan komponentin sisällä määriteltyjen rakenteiden avulla. Komponenttiolio kokonaisuutena tarjoaa siis helposti laajennettavan pohjan komponentin instanssin esittämiselle aktiokielessä. Komponentin, komponenttiolion ja käyttörajapintojen ympärille suunnitellaan seuraavissa alaluvuissa esiteltävät rajapintamallit.

## 3.2 Rajapintaolion kopioiminen

”Rajapintaolion kopioiminen” -rajapintamallissa komponenttien välinen toiminta rakennetaan toimimaan komponenttien välille luotavan yhteisen resurssin avulla, jota edustavat rajapintaoliot. Rajapintaolion tehtävänä on toimia komponenttien välillä yhteisenä tietorakenteena, johon molemmat osapuolet voivat tehdä muutoksia ja johon tehtyihin muutoksiin molemmat osapuolet voivat reagoida omalla tavallaan. Rajapinnan toteuttava komponentti ottaa vastuun rajapintaolioiden alustamisesta ennakkoon, jonka jälkeen rajapinnan käyttäjän tulee kopioida (*duplicate*) itselleen kopio tällaisesta oliosta. Kun oliosta luodaan kopio, niin samalla luodaan automaattisesti kopiot myös kaikista aktioista, joissa kyseinen olio on osallistujana [2]. Tämän ominaisuuden ansiosta komponentti voi asettaa rajapintaolion osallistujaksi rajapintapalvelut toteuttaviin aktioihin ennakkoon, jolloin rajapinnan käyttäjä pääsee käsiksi komponentin tarjoamiin toimintoihin suoraan olion kopioimisen jälkeen. Rajapinnan käyttäjä voi kopioimisen jälkeen tehdä muutoksia rajapintaolion muuttujiin, joihin rajapinnan toteuttava komponentti reagoi sen määrittelemien aktioiden avulla, jotka voivat sijoittaa tuotetut paluuarvot tähän samaan olioon.

Kuvassa 8 havainnollistetaan rajapintaolion kopioimisessa tapahtuva toiminta. Kuvassa komponentti *A* kopioi rajapinnassa esitellyn luokan *H* tyyppisen rajapintaolion, minkä seurauksena kyseiseen olioon liitetty aktio kopioituu myös. Tämän jälkeen komponentti *A* voi muokata kopion muuttujia, mikä virittää aktion toiminnallisuudet rajapinnan taka-

na. Olion kopioimisen yhteydessä kopioituvat aktiot eivät näy rajapinnan käyttäjälle millään tavalla.




***Kuva 8.*** *Rajapintaolion kopioimisen toimintaperiaate.*

Rajapintaolion kopioiminen on aktiokielen kannalta suoraviivainen tapa toteuttaa rajapintarakenne, koska se hyödyntää aktiokielen jo olemassa olevia toiminnallisuuksia. Rajapintamallia voidaan siis hyödyntää aktiokielellä ohjelmoitaessa, vaikka kielessä suosittaisiinkin jotain muuta rajapintamallia. Ratkaisu skaalautuu myös suoraan useille yhtäaikaisille käyttörajapinnoille, missä tapauksessa eri rajapintaoliot voivat edustaa omia rajapintojaan. Rajapintaolioiden kopioimisessa tulee kuitenkin olla tarkkana, koska olion kopioimisen yhteydessä saatetaan joutua suorittamaan usean aktion kopioiminen. Tämä voi käydä raskaaksi rakennetta käytettäessä osana laajempaa kokonaisuutta, jossa rajapintaolioita kopioidaan yhtä aikaa monesta eri paikasta. Rajapinnan käyttäjä saattaa myös joutua luomaan kopioita aktioista, jotka eivät ole oleellisia käyttäjälle. Aktioiden kopioimisen määrään voidaan kuitenkin vaikuttaa järkevällä ohjelmoinnilla, esimerkiksi määrittelemällä omat rajapintaoliot vastaamaan eri komponentin toimintoja.

Ohjelmassa 12 on määritetty esimerkkitaulukon käyttörajapinta, jossa määritetään kolme rajapintaluokkaa, *Message_add*, *Message_remove* ja *Message_get*, jotka vastaavat toimintoja *lisää*, *poista* ja *lue arvo*. Rajapinnan käyttäjä kopioi itselleen haluamaansa toimintaa vastaavan rajapintaolion komponenttiolioon varastoitujen referenssien avulla, ja täyttää olion muuttujiin haluamaansa toimintaa vastaavat tiedot. Rajapinnan takana

määritetyt aktiot virittyvät kopioituun olioon tehtyjen muutosten seurauksena ja toteuttavat rajapinnan käyttäjän haluaman toiminnan.

```
interface Array is
   // State kertoo mitä rajapintaoliolla tehdään sillä hetkellä
   // init = aloitustila, ready = valmis aktioiden virittämistä varten
   // working = tulosta työstetään, done = tulos on luttavissa oliossa
   type State is (init, ready, working, done);

   // Rajapintaluokat
   class Message_add;
   class Message_remove;
   class Message_get;
end Array;

-----------------------------------------------------------------------

component Array is
   initially
      // 1. Luodaan rajapintaoliot a, r, g (add, remove, get)
      // 2. Luodaan aktiot ja asetetaan oliot osallistujiksi niihin
      // 3. Alustetaan komponenttiolion referenssit olioihin a, r, g
   end;

   class Message_add is ... end;
   class Message_remove is ... end;

   class Message_get is
      state: State := init;
      index: integer;
      returnValue: integer;
   end;

   // Aktioiden toteutukset, joihin Message-oliot ovat osallistujina.
   Action Array_add is ... end;
   Action Array_remove is ... end;
   Action Array_get is ... end;
end Array;
```

***Ohjelma 12.*** *Komponentti alustaa käyttörajapinnassa esitettyjen luokkien pohjalta rajapintaoliot, joista luotujen kopioiden avulla rajapinnan käyttäjä kommunikoi komponentin kanssa.*

Toinen tapa toteuttaa ohjelmassa 12 esitetty rakenne on määrittää rajapinnan taakse rajapintaolioiden käsittelemiseen tarkoitettu tulkkiaktio, jonka vastuulla on muuttaa rajapinnan käyttäjän rajapintaolioon tekemät muutokset komponentin toiminnallisuudeksi. Tällainen rakenne luo rajapintaolion kopioimisen yhteydessä ainoastaan kopion tulkista ja komponentin tarvitsee esitellä vain yksi rajapintaolio kaikkia rajapintatoimintoja varten. Tällä tavalla minimoidaan rajapintaolion kopioimisen yhteydessä tehtävä aktioiden kopiointi ja samaa olion kopiota voidaan käyttää kaikkien viestien välittämiseen komponentille.

Rajapintaolion jakaminen komponenttiolion referenssien avulla on yksi tapa toteuttaa rajapintaolion jakaminen, mutta aktiokielen kannalta olisi mielekkäämpää, jos rajapin-

taoliot pystyttäisiin suoraan sitomaan komponentin rajapintaan. Yksi mahdollinen tapa toteuttaa rajapintaolion sitominen komponentin rajapintaan on lisätä aktiokieleen uusi komponentin rakenteiden jakamiseen tarkoitettu rakenne, välittäjä (*mediator*). Välittäjän ideana on toimia oikopolkuna komponentin haluamiin rakenteisiin esittelemällä rajapinnassa rakenne, johon komponentti voi sitoa haluamansa rakenteen rajapinnan takana. Käytännössä välittäjä on siis rajapinnassa esiteltävä referenssi. Ohjelmassa 13 on kuvattu tulkkiaktiolla toteutettu esimerkkitaulukon rakenne, jossa hyödynnetään välittäjärakennetta.

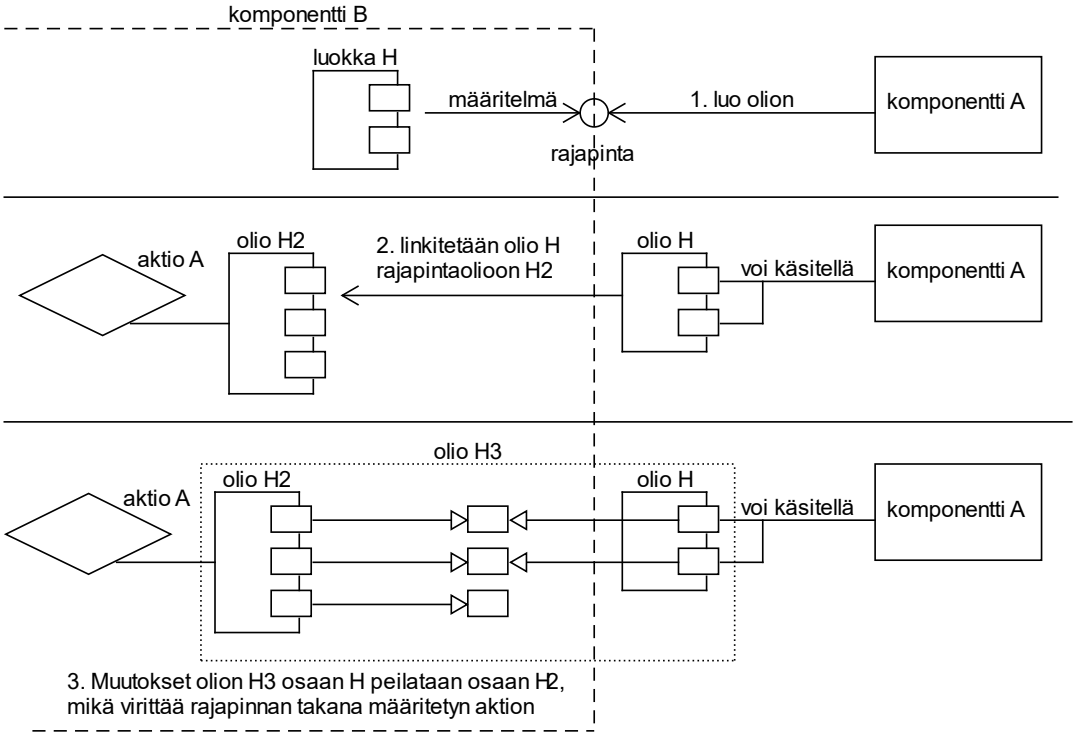
```
1  interface Array is
2     type State is (init, ready, working, done);
3     type Action is (add, remove, get);
4
5     class Message is
6        state: State := init;
7        action: Action;
8        valueOrIndex: integer;
9        returnValue: integer;
10    end;
11
12    interfaceObject: mediator Message; // välittäjä
13 end Array;
14
15 ------------------------------------------------------------------------
16
17 component Array is
18    initially
19       create Message as m;
20       create Interpreter(m) as i;
21       interfaceObject->m;
22    end;
23
24    Action Interpreter is
25       participant Message as info;
26       common guard info.state == ready;
27    begin
28       // Tulkitaan viestiluokasta tieto ja luodaan sitä vastaava aktio.
29    end;
30
31    Action Array_push() is ... end;
32    Action Array_remove() is ... end;
33    Action Array_get() is ... end;
34 end Array;
```

***Ohjelma 13.*** *Rajapintaolioiden kopioimiseen perustuva rajapintarakenne tulkkiaktiolla ja välittäjällä toteutettuna.*

Välittäjärakenteen avulla rajapintaolioiden jakaminen saadaan sidottua komponentin käyttörajapintaan, mikä parantaa rajapintamallin käytettävyyttä merkittävästi. Tämän lisäksi komponentin ei tarvitse enää esitellä komponenttiolion rakennetta rajapinnan käyttäjälle, koska käyttäjä saa kaikki kommunikointiin tarvittavat tiedot suoraan rajapintamäärittelystä. Välittäjärakennetta voidaan myös hyödyntää muunkin tyyppisten rakenteiden jakamiseen, minkä ansiosta rakenteesta saadaan yleiskäyttöinen ja joustava.

## 3.3 Linkitettävät rajapintaoliot

”Linkitettävät rajapintaoliot” -rajapintamallissa komponenttien välinen toiminta rakennetaan rajapintaolioiden avulla, samaan tapaan kuin edellisessä alaluvussa esitellyssä rajapintamallissa, mutta olioiden kopioimisen sijasta komponenttien välinen rajapintatoiminta aloitetaan käyttämällä aktiokieleen lisättävää uutta linkittämiskomentoa (*link*). Linkittämiskomento yhdistää määritellyt oliot yhdeksi olioksi ja muuttaa niiden yhteensopivat muuttujat kyseisten olioiden yhteisiksi muuttujiksi, mikä tarkoittaa sitä, että jatkossa näihin muuttujiin tehdyt muutokset peilautuvat kaikkiin linkitettyjen olioiden vastaaviin muuttujiin. Linkittämiskomennon avulla rajapinnan käyttäjä voi linkittää luomansa yhteensopivan olion komponentin linkitystä odottaviin rajapintaolioihin, jotka komponentti on voinut ennakkoon asettaa osallistujiksi rajapinnan takana määriteltyihin aktioihin. Linkitetyt oliot eivät kuitenkaan menetä vanhaa rakennettaan linkittymisen seurauksena, vaan kyseiset oliot muodostavat uuden olion, joka sisältää kaikkien linkitettyjen olioiden rakenteet, jolloin kaikki osapuolet voivat käsitellä omaa osaansa oliosta. Linkittymisen jälkeen rajapinnan käyttäjä voi tehdä muutoksia linkittämänsä olion muuttujiin, mitkä peilautuvat komponentin olioon ja virittävät halutut toiminnallisuudet paljastamatta komponentin aktiota rajapinnan käyttäjälle. Kuvassa 9 on kuvattu rajapintamallin rakennetta korkealla tasolla.




***Kuva 9.*** *Rajapintaolioiden linkittämisen periaate. Linkitettävillä olioilta ei tarvitse olla identtistä rakennetta.*

Linkittämiskomento ei vaadi olioilta identtistä rakennetta, vaan oliot voivat linkittyä toisiinsa myös osittain. Linkittämiskomennon suorittaminen liittää yhteen annettujen olioiden muuttujat, joilla on yhteinen nimi ja joille on määritetty aktiokieleen lisättävä uusi *shared*-lisätyyppi, jonka ideana on estää olioiden välillä tapahtuvat ei-toivotut linkittymiskomennot. Käytännössä komponentit voivat periyttää oman luokkansa rajapinnassa esitellystä luokasta ja laajentaa yhteen linkitettävät oliot kattamaan ylimääräisiä muuttujia toiselta osapuolelta piilossa. Tällä tavalla osapuolten ei tarvitse paljastaa toisilleen kommunikoimiseen käytettävän olion koko rakennetta, mikä helpottaa aktioiden liittämistä olioon, jotka eivät suoraan liity kyseiseen toimintaan.

Ohjelmassa 14 on esitetty linkitettävien rajapintaolioiden avulla toteutettu rakenne esimerkkitaulukolle. Kommunikoidakseen taulukon kanssa rajapinnan käyttäjä joko määrittelee itse yhteensopivan olion, luo rajapinnassa esitellyn viestiolion tai periyttää itselleen käyttörajapinnassa esitellystä viestiluokasta olion, minkä jälkeen hän voi linkittää olionsa rajapinnassa esiteltyyn viestiluokasta periytettyyn *MessageExtended*-luokkaan. Rajapinnan takana instantioitu olio on komponentin alustuksen yhteydessä asetettu rajapinnan takana määritellyn tulkkiaktion osallistujaksi, jolloin linkityksen avulla tehdyt muutokset olioon aloittavat komponentin toiminnallisuuden rajapinnan takana. Ohjelmassa hyödynnetään edeltävässä luvussa esiteltyä välittäjärakennetta.

```
1  interface Array is
2     class Message is // State ja Action ovat samat kuin ohjelmassa 13
3        state: shared State := init;
4        action: shared Action;
5        valueOrIndex: shared integer;
6        returnValue: shared integer;
7     end;
8
9     interfaceObject: mediator Message;
10 end Array;
11
12 ------------------------------------------------------------------------
13
14 component Array is
15    initially
16       create MessageExtended as ml;
17       create Interpreter(ml);
18       interfaceObject->ml;
19    end;
20
21    class MessageExtended : Message is ... end;
22
23    Action Interpreter(info: MessageExtended) is
24       participant MessageExtended as info;
25       common guard info.state == ready;
26    begin ... end;
27 end Array;
```

***Ohjelma 14.*** *Rajapinnan käyttäjä kommunikoi komponentin kanssa luomalla itselleen yhteensopivan olion ja linkittämällä sen komponentin olioon välittäjän avulla.*

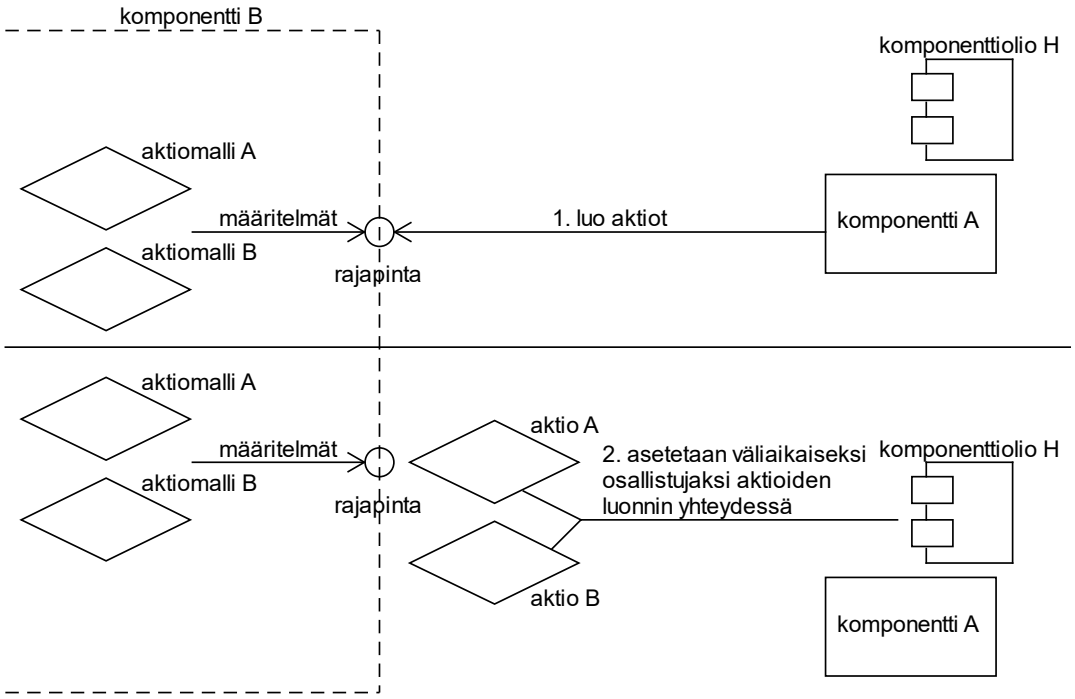
Rajapintaolioiden määrästä ei saa kuitenkaan muodostua pullonkaula mallin kannalta tilanteissa, joissa ei tiedetä ennakkoon komponenttia käyttävien tahojen määrää. Tämä ongelma voidaan kuitenkin helposti korjata muuttamalla linkittämiskomento sellaiseksi, että se luo aluksi kopion linkityksen kohteena olevista oliosta, minkä jälkeen linkittyminen suoritetaan loppuun. Tällä tavalla yhtä rajapintaoliota voidaan käyttää useiden linkittymiskomentojen toteuttamiseen.

Rajapintaolion kopioimiseen ja linkittämiseen perustuvissa rajapintamalleissa on yksi yhteinen ongelma, joka on komponenttien kanssa kommunikoimisen aloittamisen hitaus. Molemmissa malleissa rajapinnan käyttäjän tulee aluksi suorittaa alustava toimenpide, olion kopioiminen tai linkittäminen, minkä jälkeen hän pystyy vasta aloittamaan varsinaisen toiminnan komponentin kanssa. Optimaalisessa tilanteessa rajapintatoiminta voitaisiin suorittaa yhdellä toimenpiteellä, mikä tosin onnistuu rajapintaoliomalleillakin, kunhan komponentti on päässyt käsittelemään rajapintaoliota. Rajapintatoiminnan aloittamisen hitauden merkitys kuitenkin vähentyy merkittävästi, kunhan komponentit käyttävät samaa oliota jatkuvien rajapintatoimintojen suorittamiseen.

## 3.4 Aktiorajapinta

”Aktiorajapinta” -rajapintamallissa komponenttien välinen toiminta rakennetaan aktioiden avulla. Rajapintamallissa komponentti esittelee käyttörajapinnassaan joukon aktiomalleja vastaamaan erilaisia komponentin ulkomaailmalle tarjoamia palveluita, joita muut komponentit voivat suorittaa luomalla toiminnallisuutta edustavan aktion. Näitä aktioita kutsutaan rajapinta-aktioiksi, jotka yleensä luodaan palvelun tarpeen seurauksena, viritetään aktion luonnin yhteydessä ja poistetaan aktion suorituksen jälkeen. Rajapinta-aktiot toimivat siis yleensä kertaluontoisina komponenttien välisinä palvelupyyntöinä. Tämä rajapinnan toimintaperiaate muistuttaa paljon nykyisin suosituissa ohjelmointikielissä käytettyjä komponenttien jäsenfunktiokutsuja, mikä on merkittävä etu rajapintamallin omaksuttavuuden ja käytettävyyden kannalta. Funktioiden suorittamisen sijasta ohjelmakoodi on vain sijoitettu suoritettaviin aktioihin.

Komponenttiolio asetetaan väliaikaiseksi osallistujaksi rajapinta-aktioihin automaattisesti niiden luonnin yhteydessä. Väliaikainen aktioon osallistuminen tarkoittaa sitä, että komponenttiolion osallistuminen aktioon puretaan aktion alustustoimenpiteiden jälkeen. Tämän ominaisuuden avulla rajapinta-aktioiden alustuksessa voidaan aina olettaa pääsy komponenttiolioon varastoituihin tietoihin, mikä helpottaa näiden rakenteiden sitomista rajapinnan takana luotuihin muihin rakenteisiin. Tämä tekee uudelleenkäytettävien aktiomallien toteuttamisesta myös helpompaa, koska aktion rungossa voidaan keskittyä itse toiminnallisuuden määrittelemiseen. Komponenttiolioon voidaan myös varastoida esimerkiksi komponentilta usein kysyttyjä tietoja, jolloin joissain tapauksessa saatetaan säästyä koko aktion sisältämän logiikan suorittamiselta. Kuvassa 10 kuvataan rajapinta-aktioiden toimintaperiaate.




***Kuva 10.*** *Komponenttiolio asetetaan väliaikaiseksi osallistujaksi rajapinta-aktioihin automaattisesti niiden luontihetkellä.*

Rajapinta-aktioiden paluuarvojen palauttaminen toteutetaan aktiorajapinnassa antamalla komponentin aktiolle parametreinä halutun muuttujan nimi ja referenssi haluttuun olioon, jotta aktio voi suorituksensa lopuksi luoda paluuarvon sijoittavan aktion huolehtimaan paluuarvon palauttamisesta. Tällainen toimenpide on todennäköisesti hyvin yleistä aktiokielessä, minkä takia aktiokieleen kannattaa määritellä valmis paluuarvoaktiomalli toiminnan suorittamiseksi. Ohjelmassa 15 esitellään mahdollinen toteutus tämänlaiselle aktiolle, joka käyttää kolmea parametria arvon palauttamiseen: referenssi haluttuun olioon, muuttujan nimi kyseisessä oliossa ja paluuarvomuuttuja.

Ohjelmassa 15 on esitetty esimerkkitaulukon rakenne toteutettuna rajapinta-aktioilla ja hyödyntäen paluuarvoaktiota. Ohjelmassa *Main*-subjekti hakee taulukosta arvon *value*-muuttujaan ja tulostaa sen arvon käyttäjälle. Aktiosta saatava paluuarvo voidaan sijoittaa suoraan subjektissa määritettyyn muuttujaan, koska subjekti on tilallinen rakenne. Aktiomallin *Array_get* lopussa käytetään uutta aktiokielen komentoa ”*remove this*”, joka määrittää aktion poistamaan itsensä sen onnistuneen suorituksen jälkeen. Tällainen komento on mielekäs lisäys aktiokieleen aktiorajapintojen kannalta, koska aktiorajapinnassa aktion onnistunut suoritus yleensä vastaa palvelun loppuun suorittamista, minkä takia aktioita ei ole mielekästä palauttaa aktiovarastoon odottamaan uudelleen vuorontamista. Aktiomallissa *Array_get* käytetään uutta *&object*-parametrityyppiä, joka tunnistaa annetun olion tyypin ja luo siihen referenssin.

```
1  subject Main is
2  begin
3     create Array as a;
4     value: integer;
5     create a.Array_get(0, value, this);
6     wait until value != NULL then
7        print (value);
8     end wait;
9  end Main;
10
11 interface Array is
12    Action Array_get(index, rValue: integer, rObject: &object);
13    Action Array_add(...);
14    Action Array_remove(...);
15 end Array;
16
17 ------------------------------------------------------------------------
18
19 component Array is
20    Action Array_get(index, rValue: integer, rObject: &object)
21       common guard == true;
22    begin
23       ...
24       // Luodaan aktiokielen paluuarvoaktio
25       create return(rObject, rObject.rValue, <<found value>>);
26       remove this;
27    end;
28
29    Action Array_add(...) is ... end;
30    Action Array_remove(...) is ... end;
31 end Array;
```

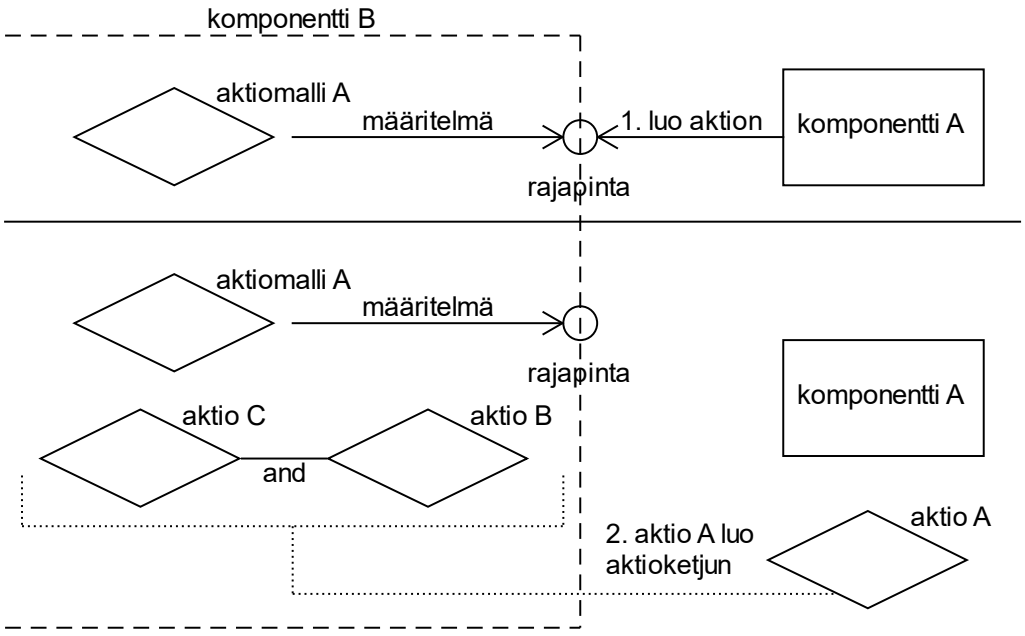
***Ohjelma 15.*** *Aktion paluuarvomuuttuja palautetaan paluuarvoaktion avulla.*

Rajapinta-aktioiden ympärille rakennetussa rajapintamallissa merkittävänä etuna on sen toimintamallin selkeys ja yksinkertaisuus. Aktiot ovat tavallisesti aktiokielessä ohjelman suoritusta edistäviä perusrakenteita, minkä takia ohjelmoijan on helppo omaksua ne myös rajapintapalveluiden toteutettaviksi rakenteiksi. Aktioita käyttämällä pystytään rakentamaan selkeä työnjako komponenttien välille, minkä ymmärtäminen vaatii ohjelmoijalta vain aktioiden perusperiaatteen tuntemisen.

## 3.5 Aktioketjut

”Aktioketjut” -rajapintamallissa komponenttien välinen toiminta rakennetaan käyttämällä rajapinta-aktioita samaan tapaan kuin edellisessä alaluvussa esitellyssä rajapintamallissa, mutta tavanomaisten aktioiden sijasta komponenttien väliseen kommunikointiin käytetään useasta aktiosta muodostuvia aktiota. Rajapintamallin ideana on rakentaa rajapinta-aktiot koostumaan selvästi toisistaan eroteltavista osista, osa-aktioista, joiden yhteistoiminta tuottaa komponentin tarjoaman toiminnallisuuden. Osa-aktiot ovat tavallisia itsenäisestikin toimivia aktioita, jotka ovat yhdistetty yhdeksi aktioksi loogisella operaattorilla ”ja” (*and*). Osa-aktioista koostuvia aktioita kutsutaan aktioketjuiksi (*actionchain*).

Aktioketjut ovat perusperiaatteeltaan aktiorajapinnan erikoistapaus, missä rajapinta-aktiot toimivat aktioketjujen rakentajina. Aktioketjuksi yhteenliitetyt osa-aktiot suoritetaan aktiorunko kerrallaan määritellyssä järjestyksessä vasemmalta oikealle, niin kuin ne olisivat yksi aktio. Käytännössä rajapinnan toteuttava komponentti esittelee käyttörajapinnassa aktion, johon kohdistuva luomiskomento luo rajapinnan takana määritellyn aktioiden sarjan. Aktioketjun suorituksen aikana ohjelman tilaa ei tallenneta osa-aktioiden välissä, mutta aktiot käsitellään silti omina kokonaisuuksinaan estäen niitä viittaamasta suoraan toistensa väliaikaisiin muuttujiin. Aktioketjuun kuuluvien kaikkien osa-aktioiden tulee olla olemassa ennen kuin aktioketjun suoritus voidaan aloittaa. Kuvassa 11 on esitetty aktioketjun toimintaperiaate.




***Kuva 11.*** *Komponentin käyttörajapinnassa esitelty aktiomalli toimii rajapinnan takana määritetyn aktioketjun rakentajana.*

Aktioketjut jakavat rajapinta-aktiot selkeisiin toisistaan eroteltaviin itsenäisiin osiin mahdollistaen niiden suunnittelemisen ja toteuttamisen erillään toisistaan. Tämä on merkittävä etu rajapintamallissa, koska nykyisin ohjelmointia tehdään yleensä ryhmissä. Aktioketjut helpottavat ohjelmointitiimin sisällä tehtävää työnjakoa ja niitä voidaan hyödyntää paikallistamaan mahdollisia ohjelman aikana tapahtuvia virhetilanteita. Aktioketjujen avulla voidaan myös toteuttaa helpommin uudelleenkäytettävää ohjelmakoodia sijoittamalla usein käytetyt algoritmit omiin osa-aktioihinsa, jolloin algoritmi voidaan tuoda kaikkien aktioiden käytettäväksi helposti liittämällä se tarvitseviin aktioihin. Esimerkiksi esimerkkitaulukon tapauksessa yksi aktio voidaan määritellä ottamaan vastuu halutun solmun etsimisestä taulukosta, kun toinen aktio huolehtii taulukon muokkaamisesta, ja kolmas aktio suorittaa palvelun suorittamiseen liittyvät lopetustoimet kuten paluuarvon palauttamisen. Edellä kuvatunlainen työnjako aktioille onnistuu

myös ilman aktioketjuja, mutta silloin ongelmia voi aiheuttaa aktioiden sattumanvarainen suoritusjärjestys, jos esimerkiksi jokin ulkopuolinen aktio käy muokkaamassa taulukkoa aktioiden suorituksen välissä. Aktioketjuja käyttämällä komponentti voi varmistua toiminnallisuuksien tapahtuvan saman aktion aikana. Aktioketjujen käyttäminen selkeyttää myös ohjelman luettavuutta tilanteissa, joissa komponentin palveluun liittyy useita aktioita. Aktioketjun määritelmä kokoaa yhteen paikkaan tiettyyn palveluun liittyvät aktiot, jolloin ohjelmoijan on helppo selvittää tiettyyn komponentin palveluun liittyvät osat.

Ohjelmassa 16 on toteutettu edellisessä kappaleessa esitetty rakenne esimerkkitaulukolle. Ohjelmassa komponentti esittelee käyttörajapinnassa *Array_get*-aktion, jonka suorittaminen luo kahdesta osa-aktiosta, *find_position* ja *return_value,* koostuvan aktioketjun. Osa-aktio *find_position* vastaa käyttäjän parametrien vastaanottamisesta ja halutun solmun etsimisestä taulukosta, minkä jälkeen osa-aktio *return_value* palauttaa löydetyn arvon rajapinnan käyttäjälle. Näin tehdyllä vastuunjaolla saadaan hakualgoritmi toteutettua omaksi kokonaisuudekseen, jota voidaan hyödyntää myös solmuja muokkaavien aktioketjujen toteutuksissa. Ohjelmassa esitetty rakenne on joustava ja mahdollistaa myöhemmin tehtävät muokkaukset toiminnallisuuksien eri osiin.

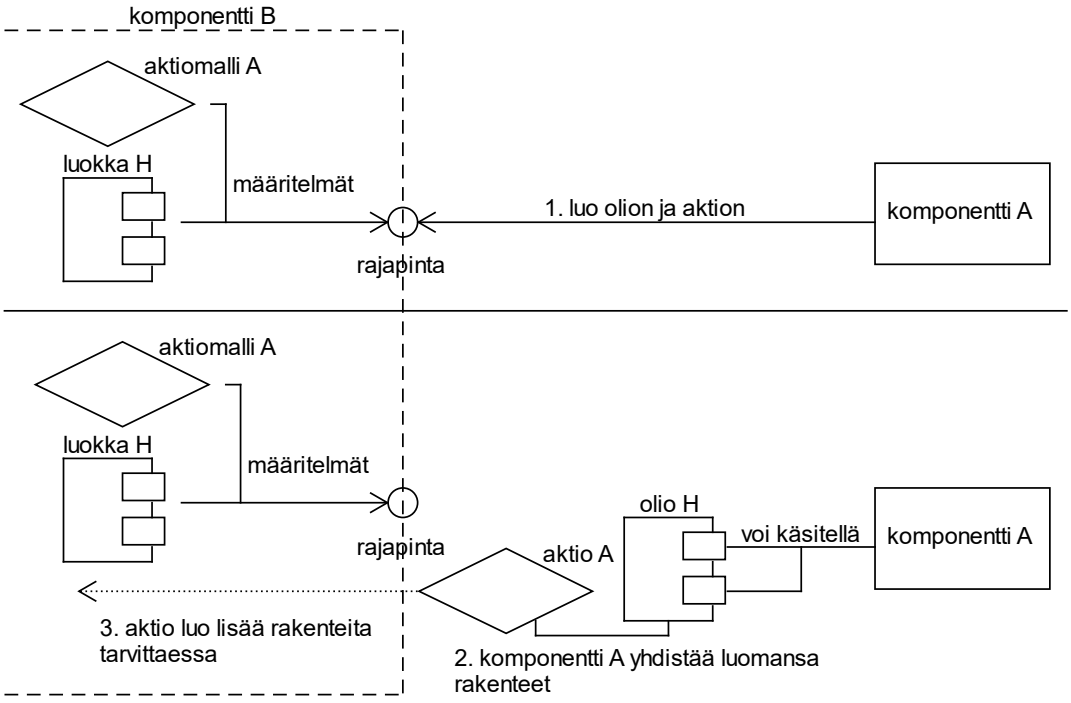
```
1  component Array is
2     action Array_get(index, rValue : integer, rObject: &object) is
3        create TempValues as tv;
4        create find_position(tv, index)
5               AND return_value(tv, rValue, rObject);
6     remove this; end;
7
8     actionchain Array_add() is ... end; // find & add
9     actionchain Array_remove() is ... end; // find & remove
10
11    class TempValues is temp: integer; end;
12
13    Action find_position(index: integer) is
14       participant TempValues as tv;
15       common guard == true;
16    begin
17       // hakualgoritmi hakee halutun solmun taulukosta
18       tv.temp := <<found value>>;
19    remove this; end;
20
21    Action return_value(rValue: integer, rObject: &object) is
22       participant TempValues as tv;
23       common guard == true;
24    begin
25       create return(rObject, rObject.rValue, tv.temp);
26       remove tv;
27    remove this; end;
28 end Array;
```

***Ohjelma 16.*** *Rajapinnassa esitetty aktio Array_get rakentaa osa-aktioista find_position ja return_value koostuvan aktioketjun.*

Aktioketjujen käyttäminen rajapintatoiminnan toteuttamiseen ei vaadi aktiokielen muokkaamista tai uusien rakenteiden lisäämistä aktiokieleen, mikä on merkittävä etu rajapintamallin kannalta. Tämän lisäksi aktioketjujen käyttömahdollisuudet eivät rajoitu pelkästään rajapintarakenteen määrittelemiseen, sillä ne ovat erittäin hyvä rakenne myös komponenttien laajentamiseen. Ohjelmoija voi halutessaan määritellä esimerkiksi komponentin aktion kattamaan hänelle tarpeelliset toiminnallisuudet sisällyttämällä siihen oman osa-aktionsa. Tällä tavalla komponentin normaali toiminnallisuus pysyy ehjänä ja periytetty komponentti saa alustettua sille tarpeelliset oliot.

## 3.6 Täydentävät rajapintarakenteet

”Täydentävät rajapintarakenteet” -rajapintamallissa komponenttien välinen toiminta rakennetaan toimimaan olioista ja aktioista koostuvien yhdistelmärakenteiden avulla. Rajapintamallissa komponentti esittelee käyttörajapinnassa luokista ja aktiomalleista koostuvan joukon rakenteita, joista luodut rakenteet eivät yksinään tee mitään, mutta yhdistettynä toisiinsa muodostavat yhdistelmärakenteen, joka toteuttaa halutun toiminnallisuuden. Aktiot toimivat rajapintamallissa komponentin palveluiden toteuttajina, joita aktivoidaan rajapintaluokista luotavilla olioilla. Kuvassa 12 on kuvattu rajapintamallin toimintaperiaate.




***Kuva 12.*** *Rajapinnan käyttäjä luo olion ja aktion, joiden yhteenliittäminen aloittaa komponentin tarjoaman toiminnallisuuden.*

Rajapintamallin ideana on siis määritellä komponentin käyttörajapinta koostumaan yhdisteltävistä rajapintarakenteista, joiden avulla rajapinnan käyttäjä voi rakentaa itselleen sopivimman kokoonpanon rajapinnan tarjoamista toiminnallisuuksista. Rajapintamallia sovellettaessa ohjelmoijan on kuitenkin suunniteltava komponentin rajapintarakenne huolella, koska täydentävien rajapintarakenteiden avulla pystytään luomaan helposti myös vaikeasti lähestyttäviä ja käytettäviä rajapintoja. Rajapinnan käyttäjän tulisi pystyä hahmottamaan nopeasti mitä toiminnallisuuksia komponentti tarjoaa ja kuinka hän pystyy suorittamaan niitä, mikä voi olla haastavaa, jos rajapinnassa esitellään iso joukko näennäisesti toisiinsa liittymättömiä luokkia ja aktiomalleja. Tämä käytettävyysongelma voidaan korjata esittelemällä rajapintarakenteita ”yhden suhde moneen” -periaatteella. Käytännössä komponentin käyttörajapinnassa kannattaa siis esitellä joko yksi luokka tai aktiomalli, josta luotu rakenne yhdistetään moneen vastakkaiseen rakenteeseen. Tällaista rakennetta hyödyntämällä rajapinnan käyttäjän tarvitsee vain tunnistaa, kumpia rakenteita käyttörajapinnassa esitellään enemmän, minkä jälkeen hän saa alustavan kuvan siitä, miten rajapintaa on tarkoitus käyttää.

Ohjelmassa 17 on toteutettu esimerkkitaulukon rakenne käyttämällä täydentäviä rajapintarakenteita. Ohjelmassa esimerkkitaulukko esittelee käyttörajapinnassa viestiluokan ja kolme keskeneräistä aktiomallia, jotka edustavat komponentin toiminnallisuuksia *lisää*, *poista* ja *lue arvo*. Näiden rakenteiden avulla rajapinnan käyttäjä voi rakentaa tälle tarpeelliset toiminnallisuudet toteuttavan yhdistelmärakenteen, johon kuuluvan olion avulla rajapinnan käyttäjä voi suorittaa aktioiden toiminnallisuuksia.

```
1  interface Array is
2     // State, Action ja Message ovat samat kuin ohjelmassa 13
3     class Message is ... end;
4
5     // common guard info.action == get AND info.state == ready;
6     Action Array_get(...);
7     Action Array_add(...);
8     Action Array_remove(...);
9  end Array;
10
11 ------------------------------------------------------------------------
12
13 component Array is
14    Action Array_get(...)
15       participant Message as info;
16       common guard info.action == get AND info.state == ready;
17    begin
18       ...
19    end;
20
21    Action Array_add(...);
22    Action Array_remove(...);
23 end Array;
```

***Ohjelma 17.*** *Komponentin käyttörajapinnassa esitellään joukko rajapintarakenteita, joista rajapinnan käyttäjä voi rakentaa sopivimman kokonaisuuden.*

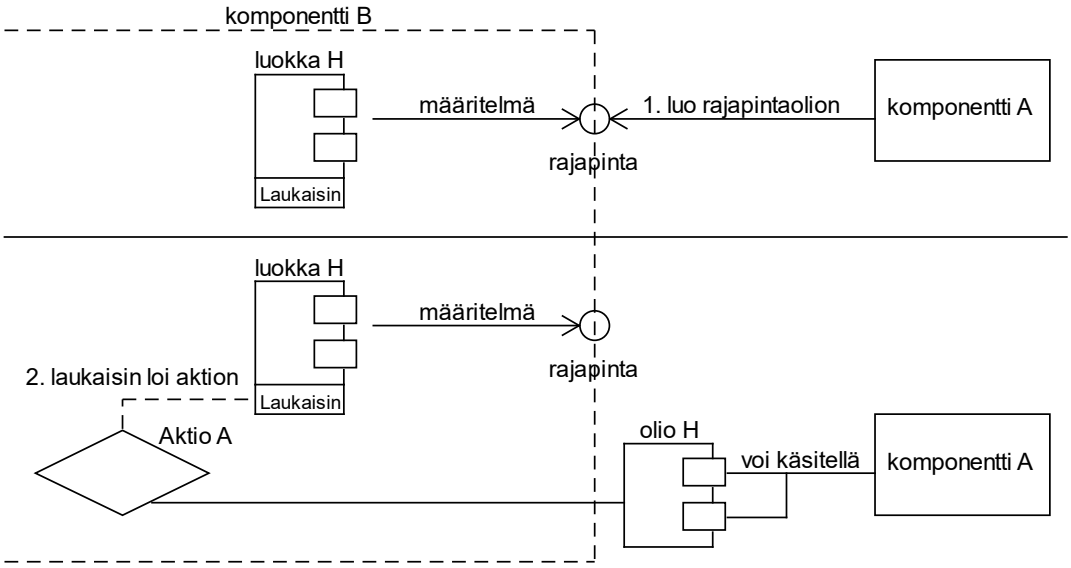
Yhdistelmärakenteiden ideaa voidaan laajentaa esittelemällä aktiokieleen keskeneräiset aktiot. Keskeneräisellä aktiomallilla tarkoitetaan rakennetta, josta luotavan keskeneräisen aktion virittymisehto viittaa rakenteeseen, jota ei välttämättä ole olemassa aktion luomisen hetkellä, jolloin aktion virittyminen on mahdotonta ilman ulkoista apua. Tällaisessa rakenteessa aktio korvaa puuttuvat osallistujat malliosallistujilla (*dummy participant*), jotka rajapinnan käyttäjän tulee korvata haluamillaan oliolla aktion virittämiseksi. Tällainen rakenne soveltuu erityisen hyvin tilanteisiin, joissa komponentti haluaa tehdä yleislähetyksiä (*broadcasting*) omasta toiminnastaan kaikille kiinnostuneille komponenteille. Keskeneräisiä aktioita käyttämällä voidaan myös toteuttaa rajapintarakenne, missä komponentti esittelee rajapinnassa tarjoamansa palvelut rajapinnan käyttäjän rakenteisiin lisättävinä lisäosa-aktioina, joiden virittämiseen käytettävän olion määritteleminen jätetään rajapinnan käyttäjän vastuulle. Tämän rakenteen toimintaperiaate on identtinen yhdistelmärakenteen kanssa, mutta lisäosa-aktioita käyttämällä rajapinnan käyttäjän tarvitsee vain luoda haluttu aktio ja liittää se omaan toteutukseensa. Tällä tavalla rajapinnan käyttäjän ei tarvitse huolehtia aktioita virittävistä oliosta, vaan tämä voi määrittää lisäosa-aktion virittämisen oman toteutuksensa huolehdittavaksi.

## 3.7 Rajapintalaukaisimet

”Rajapintalaukaisimet” -rajapintamallissa komponenttien välinen toiminta rakennetaan toimimaan yhdistelmärakenteiden avulla, mutta toisin kuin edeltävässä alaluvussa rajapinnan käyttäjä ei ole enää vastuussa rakenteen kokoamisesta, vaan sen suorittaa aktiokieleen lisättävä uusi laukaisinrakenne (*trigger*). Laukaisin on aktiokielen rakenteisiin lisättävä ylimääräinen ohjelmakoodilohko, joka koostuu laukaisimen suorittamisen aloittavasta ehdosta ja laukaisimen virittymisen yhteydessä suorittavasta ohjelmakoodista. Laukaisimen ideana on tunnistaa laukaisimella laajennettuun rakenteeseen kohdistuva laukaisimessa määritelty toiminta, jonka tapahtuessa kyseinen ohjelmakoodi suoritetaan osana kyseistä toimintaa. Laukaisimet voivat tunnistaa olioiden ja aktioiden luomisen, kopioimisen ja poistamisen. Laukaisimen ohjelmakoodi suoritetaan osana toiminnan toteuttavaa funktiota aina kun sen vahtima toiminta tapahtuu. Laukaisimen ohjelmakoodissa sallitaan aktiokielen yleisten komentojen suorittaminen (rakenteiden luominen, poistaminen, kopioiminen jne.) ja kaikki samat toiminnallisuudet, mitä laukaisimella laajennettu toiminnallisuus pystyy tavallisestikin tekemään.

Laukaisinten pääasiallisena tarkoituksena on automatisoida ohjelman aikana useasti toistuvia toimintoja ja tätä kautta tehdä ohjelman kulku sujuvammaksi. Laukaisinten ideana ei siis ole syrjäyttää aktioita ohjelmakoodin edistävänä rakenteena, vaan päinvastoin tukea sitä. Laukaisimet tarjoavat rakenteen, joka huolehtii ohjelman aikana useasti tapahtuvien yksinkertaisten toiminnallisuuksien toteuttamisesta, joita varten ei ole mielekästä toteuttaa aktiota. Rajapintatoteutuksen kannalta laukaisimia voidaan käyttää esimerkiksi olion luonnin yhteydessä luomaan halutut aktiot, joihin kyseinen olio voidaan samalla asettaa osallistujaksi. Kuvassa 13 on esitetty edellä kuvatunlainen kom-

ponentin rajapintatoteutus. Kyseistä rakennetta voidaan käyttää myös toiseen suuntaa luomalla tarvittavat oliot aktion luonnin yhteydessä.




***Kuva 13.*** *Luokkaan liitetty laukaisin luo rajapinnan takana määritellyn aktion olion alustuksen yhteydessä.*

Laukaisimilla voidaan varmistaa rajapinnan käyttäjälle yksinkertainen komponentin kanssa toiminnan aloittaminen, vaikka toiminnallisuuden toteuttamiseen tarvittaisiinkin useita rakenteita. Käytännössä rajapintalaukaisimien vastuulle voidaan laittaa kaikki rajapintatoiminnan toteuttamiseen vaadittavat alustustoiminnot, jolloin voidaan varmistaa, että toiminnan aloittaminen komponentin kanssa vaatii vain yhden toiminnon rajapinnan käyttäjältä. Tämä vähentää erilaisten rakenteiden esittelemisen tarvetta yhdistelmärakenteita käyttävien komponenttien käyttörajapinnassa, mikä selkeyttää merkittävästi käyttörajapintojen luettavuutta ja käytettävyyttä.

Laukaisimet ovat hyvin yleiskäyttöinen rakenne, jota voidaan esimerkiksi hyödyntää tässä diplomityössä esitettyjen oliorajapintamallien optimoimiseen. Laukaisimia voidaan esimerkiksi käyttää rajapintaolioiden kopioimisen yhteydessä luomaan kaikki siihen tarpeelliset aktiot, jolloin niitä ei tarvitse luoda ja alustaa valmiiksi odottamaan rajapinnan käyttäjän kopioimista. Jos ratkaisua halutaan viedä vielä pidemmälle, niin laukaisimilla voidaan poistaa kokonaan kopioimisen tarve, luomalla kaikki aktiot rajapintaolion luonnin yhteydessä. Samalla tavalla linkityksen kohteena oleva olio voi kopioimisen yhteydessä luoda kaikki tarpeelliset aktiot rajapinnan taakse.

Laukaisimet toimivat erityisen hyvin olioiden kanssa, koska ne antavat ohjelmoijalle mahdollisuuden lisätä rajallisen määrän logiikkaa näihin rakenteisiin. Tämä tuo paljon uusia mahdollisuuksia ohjelmoijalle, minkä takia laukaisimet voisivat olla hyvä lisäys aktiokieleen. Laukaisimia voidaan käyttää myös aktioiden kanssa, koska niiden toteut-

taminen aktioille ei poikkea merkittävästi niiden toteuttamisesta olioille, mutta aktioiden tapauksessa ne tarjoavat enemmänkin ylimääräisen kerroksen, johon ohjelmakoodia voidaan sijoittaa. Laukaisimia voidaan käyttää esimerkiksi edeltävässä luvussa esiteltyjen keskeneräisten aktioiden tapauksessa korvaamaan malliosallistujansa alustuksen yhteydessä luotavilla olioilla.

Ohjelmassa 18 on toteutettu esimerkkitaulukon rakenne hyödyntämällä laukaisimia. Ohjelmassa viestiluokkaan liitetty laukaisin luo rajapinnan takana määritellyn tulkkiaktion, joka tulkitsee rajapinnan käyttäjän tekemät muutokset olion muuttujiin ja luo tarvittavat aktiot halutun toiminnan suorittamiseksi. Tällä tavalla rajapinnan käyttäjälle tarvitsee esitellä käyttörajapinnassa vain viestiluokka, eikä rajapinnan toteuttavan komponentin tarvitse turhaan luoda rakenteita odottamaan mahdollista rajapintatoimintaa varten.

```
1  interface Array is
2     type State is (init, ready, working, done);
3     type Action is (add, remove, get);
4
5     class Message is
6        trigger: on creation // Laukaisin reagoi olion luontiin
7           create Interpreter(this); // Välittää itsensä parametrina
8        end trigger;
9
10       state: State := init;
11       action: Action;
12       valueOrindex: integer;
13       returnValue: integer;
14    end;
15 end Array;
16
17 ------------------------------------------------------
18
19 component Array is
20    Action Interpreter is
21       participant Message as info;
22       common guard info.state == ready;
23    begin
24       switch(info.action)
25          case: put ... end;
26          case: remove ... end;
27          case: get ... end;
28       end switch;
29    end;
30 end Array;
```

***Ohjelma 18.*** *Rajapintaluokan instantiointi luo rajapinnan takana määritellyn tulkkiaktion.*

Laukaisimia voidaan käyttää myös komponentin rakenteen toteutuksessa. Komponentin alustusaktio voidaan määritellä komponenttiolion laukaisimen luomaksi aktioksi ja komponentin purkaminen voidaan määritellä komponenttiolion poistamisen yhteydessä suoritettavalla laukaisimella. Laukaisimien avulla komponentin rakenteesta saadaan selkeä kokonaisuus. Laukaisin on hyvin monikäyttöinen rakenne, jota voidaan tulevaisuu-

dessa mahdollisesti laajentaa vahtimaan myös muita erilaisia toiminnallisuuksia, minkä takia se voisi olla mielekäs lisäys aktiokieleen.

# 4. RAJAPINTAMALLIEN ARVIOINTI

Rajapintamallien ideana on esitellä erilaisia vaihtoehtoisia näkökulmia ja ratkaisuja aktiokielen komponenttien välisen rajapintatoiminnan toteuttamiseksi. Tähän asti näitä malleja on ollut järkevää tutkia pääasiassa toisistaan erillään, jotta niiden vaikutukset toisiinsa voitaisiin minimoida ja tätä kautta laajempi näkemys komponenttien rajapintojen mahdollisesta rakenteesta voitaisiin saavuttaa. Kokonaisvaltaisen rajapintaratkaisun suunnittelemiseksi on kuitenkin oleellista myös verrata näissä malleissa esitettyjä ratkaisuja keskenään, ja tätä kautta vahvistaa tai hylätä niissä esitettyjä ideoita. Tässä luvussa tehdään rajapintamallien kriittistä tutkimista ja toisiinsa vertailua, minkä pohjalta suunnitellaan esitys aktiokielessä käytettäväksi standardirakenteeksi.

Rajapintamalleja tutkittaessa tulee ottaa huomioon myös niiden suunnittelun pohjaksi valitut rakenteet: komponentti, komponenttiolio ja käyttörajapinta, jotka ovat omalla tavallaan ohjanneet rajapintamallien ratkaisuja toimimaan näiden rakenteiden ympärillä. Nämä rakenteet lisättiin aktiokieleen määrittelemään tapa kuvata komponentteja aktiokielellä ja toimimaan selkeänä pohjana jonka päälle rajapintamallit voidaan suunnitella. Tämän diplomityön puitteissa nämä rakenteet toteuttivat niille määritellyt tehtävänsä, mutta luonnollisesti aktiokielessä voidaan valita jokin erilainen tapa toteuttaa komponentit osaksi aktiokieltä. Rajapintamalleissa esitetyt ideat pysyvät kuitenkin pääkohdiltaan pätevinä, vaikka pohjarakenteet korvattaisiinkin jollain muilla vastaavilla ratkaisuilla. Pohjarakenteiden lisäksi rajapintamalleja tutkittaessa tulee muistaa niiden perustuminen täysin aktiokielen teoriaan, minkä takia niissä esitettyjen ratkaisujen testaaminen käytännössä ennen aktiokieleen lisäämistä on suositeltavaa.

Lopputuloksena rajapintamalleja suunniteltiin yhteensä kuusi erilaista, joissa on edustettuina olioilla, aktioilla ja yhdistelmärakenteilla toteutettuja rajapintaratkaisuja. Näitä kolmea erilaista tapaa toteuttaa rajapintarakenne tutkitaan tarkemmin niille omistetuissa alaluvuissa 4.1 – 4.3, joissa kiinnitetään huomiota erityisesti malleissa esitettyjen ideoiden selkeyteen ohjelmoijan näkökulmasta. Alaluvussa 4.4 suunnitellaan esitys aktiokielessä käytettäväksi standardirajapintarakenteeksi, jossa yhdistetään rajapintamalleissa esitetyt lupaavimmat ideat yhdeksi kokonaisuudeksi. Standardirakenteen ideana on toimia pohjaoletuksena komponenttien väliselle rajapintatoiminnalle, joka vähitellen korjaantuisi kohti parasta mahdollista rakennetta ohjelmoijien käyttäessä aktiokieltä.

## 4.1 Oliorajapintamallit

Oliorajapintamallien ”rajapintaolion kopioiminen” ja ”linkitettävät rajapintaoliot” toimintaperiaate perustuu olioiden käyttämiseen komponentteja yhdistävänä rakenteena.

Näissä rajapintamalleissa lähtökohtana on luoda komponenttien välille yhteinen resurssi, rajapinnan toteuttavan komponentin ennalta alustama olio, jota komponentit käyttävät niiden väliseen kommunikointiin reagoimalla olion muuttujiin tehtyihin muutoksiin. Komponenttien väliseen kommunikointiin käytettäviä olioita kutsutaan rajapintaolioksi, joiden jakaminen rajapinnan käyttäjälle toteutettiin rajapintamalleissa kopioimalla ja linkittämällä. Oliorajapintamalleissa rajapinnan käyttäjän tavoitteena on päästä käsiksi rajapintaolioon ja rajapinnan toteuttava komponentti ottaa vastuun lupaamiensa toiminnallisuuksien yhdistämisestä kyseiseen olioon.

Rajapintaolioiden käyttäminen komponenttien rajapintatoiminnan päärakenteena on monella tapaa haastava lähtökohta rajapintarakenteen toteuttamiselle. Ensimmäinen haaste oliorajapintamalleissa on saada oliot toteuttamaan komponentin ulkomaailmalle tarjoamat toiminnallisuudet. Aktiokielessä oliot ovat suunniteltu varastoimaan ohjelman sisältämiä muuttujia eikä suorittamaan ohjelman toimintalogiikkaa, minkä takia rajapintarakenteen toteutuksessa tarvitaan myös aktioita. Nämä aktiot voidaan kuitenkin piilottaa rajapinnan käyttäjältä asettamalla rajapintaolio osallistujaksi haluttuihin aktioihin rajapinnan takana, jolloin rajapinnan käyttäjän tekemät muutokset rajapintaolion muuttujiin voidaan muuttaa suoraan komponentin toiminnallisuudeksi. Tällä tavalla saavutetaan tilanne, jossa rajapinnan käyttäjän näkökulmasta komponentin rajapinta näyttää koostuvan vain rajapintaolioista.

Toinen haaste oliorajapintamalleissa on saada ohjelmoijat tukemaan rajapintaoliota konseptina. Yleensä ohjelmointikielissä tuetaan komponenttien rajapintarakenteiden kutsumista tai luomista, minkä takia tällainen olion muuttujien muokkaaminen toiminnallisuuksien suorittamiseksi ei välttämättä tunnu ohjelmoijasta intuitiiviselta käyttää. Tämä johtuu siitä, että rajapinnan käyttäjä ei ole enää suoraan vastuussa haluamansa toiminnan aloittamisesta, vaan toiminta aktivoi itse itsensä käyttäjän tekemän muutoksen seurauksena. Lopputulos näissä tavoissa aloittaa toiminta on identtinen, mutta se ei kuitenkaan välttämättä tunnu ohjelmoijasta täysin samalta. Tämän lisäksi rajapintaolio ja komponenttiolio toimivat hyvin samanlaisessa roolissa komponentin edustajana, mikä voi myös aiheuttaa sekaannusta ohjelmoijalle. Näiden ongelmien vaikutukset kuitenkin luonnollisesti vähenevät ajan kuluessa ohjelmoijien tottuessa rakenteen käyttämiseen.

Kolmas haaste oliorajapintamalleissa on kehittää järkevä tapa toteuttaa rajapintaolion jakaminen rajapinnan käyttäjälle, jotta tämä pystyy suorittamaan olion kopioimisen tai linkittämisen komponentin kanssa kommunikoinnin aloittamiseksi. Rajapintaolion jakaminen malleissa on ongelmallista, koska sen sitominen suoraan käyttörajapintaan on rajapintarakenteen periaatteen vastaista, ja rajapintaolioiden jakaminen komponenttiolion osoittimilla ei ole vakuuttava ratkaisu yleiseksi toimintaperiaatteeksi. Tämän takia välittäjärakenne suunniteltiin aktiokieleen. Välittäjärakenteen avulla saavutetaan optimaalinen tilanne komponentin kanssa toiminnan aloittamisessa, missä rajapinnan käyttäjä pääsee käsiksi rajapintaolioon suoraan käyttörajapinnassa esiteltävän referenssin avulla.

Rajapintaoliot ovat erityisen hyvä rakenne tilanteisiin, joissa komponenttien tulee pystyä jakamaan tietoa toisilleen palvelun suorituksen aikana. Tällaisessa tilanteessa palvelun toteuttava aktio voidaan jakaa osa-aktioihin, joiden virittymisehdot voidaan sitoa rajapintaolion muuttujiin, joita rajapinnan käyttäjä voi muokata. Tällä tavalla komponentin palvelu etenee osa kerrallaan ja pystyy reagoimaan käyttäjän tekemiin muutoksiin. Rajapintaoliossa on myös se merkittävä etu, että se säilyy ohjelman muistissa rajapintatoiminnallisuuden toteuttamisen jälkeenkin, mikä mahdollistaa saman toiminnallisuuden pyytämisen komponentilta useasti saman rakenteen avulla. Rajapinnan käyttäjä voi myös luoda omalle puolelleen aktiot reagoimaan rajapintaolioon tehtäviin muutoksiin, jolloin rajapintaolion idea saadaan täysin hyödynnettyä.

Molemmissa oliorajapintamalleissa on omat etunsa ja erikoistapauksensa, joissa ne ovat parempia toteutuksia suhteessa toisiinsa, mutta loppujen lopuksi ne ovat toimintaperiaatteeltaan hyvin samantyyppisiä malleja. Tämän takia rajapintaolioiden kopioiminen voidaan ajatella näistä malleista paremmaksi ratkaisuksi aktiokielessä käytettäväksi rajapintamalliksi, koska linkittämisen hyödyntäminen vaatii linkittämiskomennon määrittelemistä aktiokieleen. Vaikka linkittymiseen perustuvaa rajapintamallia ei käytettäisikään aktiokielen rajapintatoteutuksena, on mallissa esitelty ajatus yhdistettävistä olioista potentiaalisesti mielenkiintoinen rakenne hyödyntää jossakin muussa muodossa osana aktiokieltä.

## 4.2 Aktiorajapintamallit

Aktiorajapintamallien ”aktiorajapinta” ja ”aktioketjut” toimintaperiaate perustuu aktioiden käyttämiseen komponentteja yhdistävänä rakenteena. Näissä rajapintamalleissa lähtökohtana on rakentaa komponenttien välinen toiminta koostumaan komponenttien välillä suoritettavista palvelupyynnöistä, joita edustavat rajapinta-aktiot. Rajapinta-aktio on komponentin käyttörajapinnassa esitellystä aktiomallista luotava aktio, joka toteuttaa komponentin lupaaman toiminnallisuuden, viritetään yleensä aktion luonnin yhteydessä ja usein poistetaan aktion suorittamisen jälkeen. Kommunikointiin käytettävät rajapinta-aktiot toteutettiin rajapintamalleissa yksittäisinä ja osa-aktioista koostuvina aktioina.

Rajapinta-aktioita käyttämällä komponenttien välille muodostuu selkeä työnjako, missä rajapinnan toteuttava komponentti esittelee ulkomaailmalle tarjoamansa toiminnallisuudet aktiomalleina, joista rajapinnan käyttäjä tunnistaa toimintaansa edistävän aktion ja luo sen palvelun toteuttamiseksi. Tätä työnjakoa tukee myös aktioiden normaali rooli aktiokielen toiminnallisina perusrakenteina, minkä takia ohjelmoijan on helppo samaistua ajatukseen rajapinta-aktioiden luomisesta komponenttien palveluiden pyytämiseksi. Rajapinta-aktioissa esitetty konsepti muistuttaa paljon tällä hetkellä suosituissa ohjelmointikielissä tuettuja jäsenfunktiokutsuja, joissa käytännössä ainoina merkittävinä eroina ovat funktion sisältämän ohjelmakoodin välitön suorittaminen funktiokutsun yhteydessä ja ominaisuus palata funktiokutsun suorituspaikkaan funktion sisältämän oh-

jelmakoodin suorittamisen jälkeen. Rajapinta-aktioiden samankaltaisuus jäsenfunktioiden kanssa on merkittävä etu rajapintarakenteessa.

Rajapinta-aktioiden käytössä on kuitenkin myös ongelmansa, nimittäin aktiorajapintamalleissa aktiota käytetään monella tavalla aktioiden normaalin toimintaperiaatteen vastaisesti. Aktio on pohjimmiltaan suunniteltu reaktiiviseksi ohjelman aikana monta kertaa suoritettavaksi rakenteeksi, joka valvoo ohjelman tilaa ja jonka sisältämä ohjelmakoodi suoritetaan seurauksena määriteltyyn ohjelman tilaan. Aktiorajapintamalleissa aktioita käytetään enemmän aktiivisena rakenteena, jotka viritetään pyynnöstä ja monesti poistetaan toiminnan suorituksen jälkeen, minkä takia aktion virittymisehdon määrittelemisestä ja sen arvon evaluoinnista tulee lähinnä hidaste rajapinta-aktioiden kannalta.

Rajapinta-aktioiden käyttäminen voi aiheuttaa myös teknisiä haasteita sulautetulle järjestelmälle. Aktiorajapintamalleissa aktioita saatetaan luoda ja poistaa merkittäviä määriä ohjelman aikana, mikä voi aiheuttaa huomioon otettavan määrän työtä järjestelmälle riippuen aktioiden luomiseen ja poistamiseen tarvittavien toimenpiteiden vaativuudesta. Tämä tilanne on erittäin huono esimerkiksi verrattuna oliorajapintamalleihin, joissa aktiota tarvitsee luoda yleensä vain kerran rajapinnan käyttäjää kohden, minkä takia aktiorajapintamallien resursseja hukkaava toimintaperiaate vaatii todennäköisesti optimointia jollakin tavalla. Tämän ongelman vaikutuksia voidaan vähentää lisäämällä järjestelmätasolle logiikkaa vastaamaan aktioiden uudelleen käytettävyydestä ja ottamalla ongelma huomioon kääntäjän optimoidessa ohjelmakoodia, mutta tämä vaatii ylimääräistä logiikkaa näihin rakenteisiin ja ei kunnolla korjaa tilannetta.

Aktiorajapintamallien merkittävin etu on niiden yksinkertainen mutta samaan aikaan kattava rakenne. Näissä malleissa selkeästi määritellään aktioiden rooli komponenttien toiminnallisuuksien toteuttajana, jolloin rajapinnan käyttäjän tarvitsee vain selvittää mikä aktio toteuttaa tämän haluaman toiminnallisuuden ja miten kyseisen aktion parametrit vaikuttavat siihen. Aktiorajapintamallien käytettävyys on siis erinomaista ja omaksuttavuus helppoa. Tämän lisäksi aktiorajapintamalleissa hyödynnetään pelkästään aktiokielen jo olemassa olevia toiminnallisuuksia, mikä mahdollistaa aktiorajapintamallien hyödyntämisen suoraan osana aktiokieltä. Aktiot toimivat jo entuudestaan olioita yhdistävänä rakenteena, mikä voi edesauttaa ohjelmoijien samaistumista ajatukseen aktioista myös komponentteja yhdistävänä rakenteena.

## 4.3 Yhdistelmärajapintamallit

Yhdistelmärajapintamallien ”täydentävät rajapintarakenteet” ja ”rajapintalaukaisimet” toimintaperiaate perustuu olioista ja aktioista koostuvien yhdistelmärakenteiden käyttämiseen komponentteja yhdistävänä rakenteena. Näissä rajapintamalleissa lähtökohtana on käyttää olioita ja aktioita tasavertaisina rajapintarakenteina, jotka yksinään eivät tee mitään, mutta yhdistettynä toisiinsa tai rajapinnan käyttäjän rakenteisiin toteuttavat komponentin lupaaman toiminnallisuuden. Rajapintamalleissa komponentit esittelevät

käyttörajapinnassa aktiomalleista ja/tai luokista koostuvan joukon, josta kootaan komponentin kanssa kommunikointiin käytettävä yhdistelmärakenne rajapinnan käyttäjän manuaalisella työllä tai laukaisimien mahdollistaman automaation avulla.

Yhdistelmärajapintamallien isoin ongelma on niiden tulkinnanvaraa jättävä komponenttien välisen toimintamallin määrittely. Oliorajapintamalleissa rajapinnan toteuttava komponentti ottaa vastuun rajapintaolioiden alustamisesta, jolloin rajapinnan käyttäjän tarvitsee vain selvittää, mikä rajapintaolio toteuttaa tämän haluaman palvelun ja miten rajapintaolion eri muuttujat vaikuttavat kyseiseen toimintaan. Aktiorajapintamalleissa rajapinta-aktiot toimivat aina samalla periaatteella, joten rajapinnan käyttäjän tarvitsee selvittää vain mikä aktio edustaa mitäkin komponentin palvelua ja miten parametrit vaikuttavat aktion suoritukseen. Olio- ja aktiorajapintamalleissa rajapinnan käyttäjä siis tietää ennakkoon, miten tämä aloittaa kommunikoimisen rajapinnan toteuttavan komponentin kanssa, mikä tekee komponenttien välisestä kommunikoinnista suoraviivaista ja selkeää. Yhdistelmärajapintamallit määrittelevät komponenttien välisen kommunikoinnin alkavan yhdistelmärakenteen luomisella, mutta ne eivät ota tarkemmin kantaa kyseisen rakenteen toteutukseen. Yhdistelmärakenteen vapaamman rakenteen takia rajapinnan käyttäjän tulee selvittää tapauskohtaisesti, kuinka komponentin yhdistelmärakenne kootaan ja miten sitä käytetään kommunikoimiseen komponentin kanssa. Tämä tuo paljon vastuuta komponenteille huolehtia niiden rajapintarakenteen loogisesta rakenteesta ja selkeästä käytettävyydestä. Ongelman vaikutuksia voidaan vähentää kattavalla dokumentaatiolla, mutta tämä luonnollisesti aiheuttaa ylimääräistä työtä molemmille osapuolille, eikä sekään varmista rajapinnan selkeää käytettävyyttä. Tämän takia yhdistelmärajapintamalleissa kannattaa yleensä suosia joko olioita tai aktioita komponentin käyttörajapinnan päärakenteena, jota täydennetään toisella rakenteella, jolloin rajapinnan käyttäjä voi tehdä oletuksia rajapinnan tarkoituksenmukaisesta käytöstä jo pelkästään rajapinnan koostumuksen perusteella.

Yhdistelmärajapintamallien vapaampi ja sovellettavampi rakenne voi olla toisaalta myös näiden mallien merkittävin etu, koska se mahdollistaa helpommin uusien innovaatioiden syntymisen ja se sallii aktiokielen luontaisen kehittymisen kohti ohjelmoijaenemmistön mielestä parasta ratkaisua. Tämä näkökulma on sikäli järkevä aktiokielen kannalta, koska ohjelmoijat tulevat kuitenkin päättämään loppujen lopuksi minkälainen komponenttien rajapintarakenne tuntuu loogiselta käyttää ja osoittaa parhaiten toimivuutensa käytännön ohjelmoinnissa. Tästä huolimatta aktiokieleen on kuitenkin mielekästä määritellä jonkinlainen standardirakenne toimimaan mallina ensimmäisille toteutuksille, jota voidaan käyttää aktiokielen esittelemiseen ja uusien ohjelmoijien ohjaamiseen. Tähän tarkoitukseen yhdistelmärajapintamallien vapaampi rakenne ei sovellu erityisen hyvin ja lähinnä monimutkaistaa komponenttien välistä toimintaa.

Laukaisimia ei voida suoraan käyttää aktiokielellä ohjelmoitaessa, mutta niiden toteuttaminen osaksi aktiokieltä ei ole iso haaste, koska ne vain laajentavat olioiden ja aktioiden olemassa olevia toiminnallisuuksia. Laukaisimet eivät siis sinänsä lisää aktiokieleen

uutta toiminnallisuutta, vaan ne vain antavat ohjelmoijalle mahdollisuuden lisätä ohjelmakoodia toiminnallisuuksiin, jotka eivät ennen olleet heidän käytettävissään. Laukaisimet mahdollistavat uudenlaisia tapoja ohjelmoida aktiokielellä ja lisäävät suuren määrän uusia interaktioita rakenteiden välille, minkä takia niiden toteuttamiskelpoisuutta ja toimivuutta osana aktiokieltä on erittäin haastavaa arvioida. Laukaisimet lisäävät paljon piilotettua tietoa ja toiminnallisuutta aktio-ohjelmaan, jossa esimerkiksi rakenteen luominen voi aloittaa pitkän rakenteiden luomisen ketjun rakenteen luojan tietämättä, mikä johtaa siihen, että ohjelmoija ei voi laukaisimien olemassaolon takia olla enää varma siitä, mitä kaikkea mikäkin toiminnallisuus todellisuudessa tulee tekemään. Tämä voi tehdä aktio-ohjelman suorituksen seuraamisesta vaikeaa ja hankaloittaa ohjelmakoodin luettavuutta. Laukaisimia käytettäessä tulee siis kiinnittää paljon huomiota ohjelmakoodin luettavuuteen ja tiedostaa niiden mahdolliset sivuvaikutukset. Vaikka laukaisimia ei lähdettäisikään tukemaan aktiokielen rajapintarakenteena, niin kannattaa niitä silti tutkia yleisesti aktiokieleen lisättävänä rakenteena.

## 4.4 Rajapintakokonaisuus

Olio-, aktio- ja yhdistelmärajapintamalleissa esiteltiin erilaisia toimintamalleja ja rakenteita komponenttien välisen rajapintatoiminnan toteuttamiseksi. Aktiokieleen ei kuitenkaan luonnollisesti tarvita montaa erilaista rajapintamallia ohjaamaan komponenttien välistä toimintaa, minkä takia tässä alaluvussa suunnitellaan rajapintamalleissa esitettyjen ideoiden pohjalta yksittäinen esitys aktiokielessä käytettäväksi rajapintakokonaisuudeksi. Tämän kokonaisuuden tavoitteena ei kuitenkaan ole mitätöidä rajapintamalleissa esitettyjä ideoita, vaan koota mallien pohjalta yksittäinen toimiva kokonaisuus, joka voidaan ajatella tämän diplomityön esitykseksi aktiokielessä käytettäväksi standardirajapintarakenteeksi. On kuitenkin tärkeää muistaa, että kaikissa tässä työssä esitetyissä toteutuksissa ja ratkaisuissa on omat etunsa ja ongelmansa, minkä takia niihin kaikkiin perehtyminen on suositeltavaa. Rajapintakokonaisuus on siis enemmänkin tämän diplomityön kirjoittajan henkilökohtainen näkemys ja mielipide tässä työssä tutkittuun aiheeseen, ja se on vain yksi mahdollinen ratkaisu toteuttaa rajapintarakenne.

Rajapintakokonaisuuden päärakenteeksi valitaan aktiot, koska ne toimivat muutenkin aktiokielen toiminnallisina rakenteita, jolloin niiden käyttäminen rajapintarakenteena tukee aktiokielen jo olemassa olevaa toimintamallia ja helpottaa ohjelmoijien samaistumista niiden käyttämiseen. Aktiot ovat yksinkertainen mutta samaan aikaan kattava rakenne, joiden käyttäminen rajapintatoiminnan toteuttamiseen ei vaadi aktiokielen muuttamista, mikä tekee aktioista ideaalisen rakenteen komponenttien välisen toiminnan toteuttamiseksi. Rajapinta-aktiot ovat selkeä rakenne toteuttaa komponenteille tehtävät yksittäiset palvelupyynnöt, mutta yleisesti aktiot kannattaa pyrkiä suunnittelemaan niin, että samaa aktion instanssia voidaan hyödyntää useasti ohjelman aikana, jolloin vältytään järjestelmän turhalta rasittamiselta. Aktioiden käyttäminen lisäosina rajapinnan käyttäjän rakenteisiin on myös järkevä tapa soveltaa aktiota rajapintatoimintaan.

Komponentit voivat myös tarpeen mukaan määritellä rajapintatoimintansa käyttämällä rajapintaolioita ja yhdistelmärakenteita, joita tuetaan välittäjärakenteella ja malliosallistujilla. Laukaisimiin suhtaudutaan rajapintakokonaisuudessa varauksella tiedostamalla niiden mahdolliset sivuvaikutukset, minkä takia niitä ei lähdetä lisäämään aktiokieleen ohjelmoijan vapaaseen käyttöön ilman niiden vaikutusten tarkempaa tutkimista. Laukaisimet nähdään kuitenkin potentiaalisesti erittäin hyödyllisinä rakenteina lisätä aktiokieleen. Aktiokieleen lisätyt rakenteet: komponentti, komponenttiolio ja käyttörajapinta nähdään onnistuneiksi lisäyksiksi aktiokieleen, joiden lopullinen toteutus tulee tarkentumaan, kun aktiokieltä aletaan käytännössä toteuttamaan.

# 5. YHTEENVETO

Aktiokieli on suunnitteluvaiheessa oleva aktio-ohjelmointiin tarkoitettu ohjelmointikieli, jolla toteutettavien komponenttirakenteiden rajapintatoimintaa tutkittiin ja suunniteltiin tässä diplomityössä. Aktiokieli koostuu kolmesta perusrakenteesta: olioista, aktioista ja subjekteista, joista ensimmäiset kaksi vastaavat suoraan aktiojärjestelmän vastaavia rakenteita ja subjektit ovat ohjelmoijalle tarkoitettuja korkean tason rakenteita, jotka kääntäjä purkaa olioiksi ja aktioiksi. Nämä rakenteet eivät kuitenkaan olleet riittäviä toteuttamaan komponentteja mielekkäästi osaksi aktiokieltä, minkä takia ennen rajapintaratkaisujen suunnittelemista aktiokieleen määriteltiin kolme uutta rakennetta tukemaan komponenttien määrittelyä. Nämä rakenteet ovat: komponentti, komponenttiolio ja käyttörajapinta, jotka määrittelevät rajapinnan toteuttavan komponentin rakenteen, komponentin instanssin ja komponentin ulkomaailmalle tarjottavan rajapinnan. Näiden rakenteiden avulla saatiin toteutettua selkeä pohja komponenttien määrittelemiselle, jonka päälle lähdettiin suunnittelemaan erilaisia rajapintaratkaisuja.

Rajapintaratkaisujen suunnittelussa otettiin tavoitteeksi suunnitella ne käyttämällä mahdollisimman paljon hyödyksi aktiokielen perusrakenteita, jotta minimoitaisiin tarvittavat muutokset aktiokieleen ja saataisiin hyödynnettyä mahdollisimman paljon näiden rakenteiden olemassa olevia ominaisuuksia. Rajapintaratkaisut kuvattiin työssä rajapintamalleina, joita suunniteltiin yhteensä kuusi erilaista. Nämä rajapintamallit voidaan jakaa kolmeen kategoriaan, olio-, aktio- ja yhdistelmärajapintamalleihin, sen perusteella, mitä aktiokielen perusrakenteita käytettiin niiden toteutuksen päärakenteina. Subjektit jätettiin rajapintamallien toteutusten ulkopuolelle, koska rajapintamallien suunnittelussa haluttiin lähestyä rajapintaratkaisua mahdollisimman matalalta tasolta, mitä subjektit eivät edusta.

Oliorajapintamalleissa, joita ovat ”rajapintaolioiden kopioiminen” ja ”linkitettävät rajapintaoliot”, rajapintatoiminta rakennetaan komponenttien välille luotavan yhteisen resurssin ympärille, jota edustaa rajapinnan toteuttavan komponentin ennalta alustama olio. Tällaista oliota kutsutaan rajapintaolioksi, jonka rajapinnan toteuttava komponentti asettaa osallistujaksi tarjoamiensa palveluiden toteuttaviin aktioihin, joita rajapinnan käyttäjä voi virittää muokkaamalla rajapintaolion muuttujia. Rajapintaolion jakaminen rajapinnan käyttäjälle toteutettiin rajapintamalleissa kopioimalla ja linkittämällä. Kopioimisessa rajapinnan käyttäjä luo itselleen kopion rajapintaoliosta, mikä luo samalla kopiot myös kaikista aktioista, joihin kyseinen olio on osallistujana, minkä jälkeen hän voi virittää komponentin palveluja muokkaamalla olion muuttujia. Samaa periaatetta käytettiin linkittämisessä, mikä on uusi komento aktiokieleen. Linkittymisessä rajapinnan käyttäjä linkittää oman olionsa komponentin rajapintaolioon, mikä tekee määritellyistä

olioista yhden olion säilyttäen niiden entisen rakenteen ja linkittää yhteensopivat muuttujat niiden toisiinsa, jolloin osapuolten tekemät muutokset näihin muuttujiin peilautuvat myös muihin vastaaviin muuttujiin oliossa. Näiden mallien isoimmaksi ongelmaksi muodostui niiden erilainen tapa toteuttaa rajapintatoiminta. Ohjelmoijat ovat hyvin tottuneita rakenteiden luomiseen ja kutsumiseen, minkä takia oliomallien tapa suorittaa toiminnallisuuksia reaktiona rajapinnan käyttäjän muutoksiin vaatii ohjelmoijilta totuttelua.

Aktiorajapintamalleissa, joita ovat ”aktiorajapinta” ja ”aktioketjut”, komponenttien välinen toiminta rakennetaan komponenttien välisten palvelupyyntöjen avulla, joita edustavat komponentin käyttörajapinnassa esiteltävät aktiot. Tällaisia aktioita kutsutaan rajapinta-aktioksi, joiden luominen toimii viestinä rajapinnan toteuttavalle komponentille suorittaa kyseisen aktion edustama palvelu samaan tapaan kuin esimerkiksi olio-ohjelmoinnissa jäsenfunktiot edustavat olion palveluita. Rajapinta-aktioita käytettiin rajapintamalleissa siis komponenttien välisinä kertaluontoisina palvelupyyntöinä, jotka toteutettiin rajapintamalleissa yksittäisinä aktioina ja osa-aktioista koostuvina aktioketjuina. Näiden mallien isoimmaksi ongelmaksi muodostui aktioiden tuhlaava käyttömalli, missä aktioita luodaan ja poistetaan jatkuvasti ohjelman aikana, mikä on vastakohta aktioiden tarkoituksenmukaiselle käyttämiselle. Tämän ongelman vaikutukset riippuvat luonnollisesti aktioiden luomisen ja poistamiseen liittyvien operaatioiden vaativuudesta, johon ohjelmoija voi myös vaikuttaa määrittelemällä uudelleenkäytettäviä aktioita silloin kun ne sopivat tilanteeseen.

Yhdistelmärajapintamalleissa, joita ovat ”täydentävät rajapintarakenteet” ja ”rajapintalaukaisimet”, komponenttien välinen toiminta rakennetaan olioista ja aktioista koostuvien yhdistelmärakenteiden avulla. Näissä rajapintamalleissa rajapinnan toteuttava komponentti esittelee käyttörajapinnassa luokista ja/tai aktiomalleista koostuvan joukon, joista joko rajapinnan käyttäjä tai aktiokieleen lisättävä uusi laukaisinrakenne rakentaa komponentin kanssa kommunikointiin käytettävän yhdistelmärakenteen. Rajapintamallit eivät määrittele yhdistelmärakenteelle tarkkaa rakennetta, mikä tuo paljon vastuuta ja vapauksia rajapinnan toteuttavalle komponentille. Tämä voi aiheuttaa merkittäviä käytettävyysongelmia komponenttien väliseen rajapintatoimintaan, koska laukaisimet lisäävät paljon piilotettua tietoa aktiokieleen ja rajapinnan käyttäjän tulee selvittää erikseen tapauskohtaisesti, kuinka toimia jokaisen komponentin kanssa. Toisaalta vapaampi rakenne voi myös mahdollistaa innovaatioiden syntymisen ja sallia komponenttien välisen toimintamallin kehittymisen kohti ohjelmoijaenemistön mielestä parasta ratkaisua.

Lopuksi rajapintamalleissa esitettyjen ratkaisujen pohjalta suunniteltiin yksittäinen esitys aktiokielessä käytettäväksi rajapintakokonaisuudeksi, jonka tarkoituksena on toimia työn esityksenä komponenttien välisen rajapintatoiminnan standardirakenteeksi. Rajapintakokonaisuuden päärakenteeksi valittiin aktio, koska aktioita käyttämällä komponenttien välille saadaan selkeä työnjako, missä rajapinnan toteuttava komponentti esitte-

lee rajapinnassaan tarjoamansa palvelut aktiomalleina, joista rajapinnan käyttäjä tunnistaa toimintaansa edistävän palvelun ja pyytä sitä luomalla palvelua vastaavan aktion. Aktioiden käyttäminen rajapintatoiminnan toteuttamiseen tukee myös aktioiden normaalia roolia aktiokielen aktiivisina rakenteina, mikä tekee niistä helposti omaksuttavan ja intuitiivisen rakenteen käyttää. Komponentit voivat toteuttaa palvelunsa käyttämällä aktiorajapintamalleissa esitettyjä rajapinta-aktioita, mutta yleisesti komponenttien kannattaa pyrkiä toteuttamaan aktionsa niin, että samaa aktion instanssia voidaan käyttää rajapinnan käyttäjän tekemien toistuvien toimintojen suorittamiseen. Komponentit voivat myös halutessaan käyttää rajapintaolioita ja yhdistelmärakenteita palvelujensa toteuttamiseen, joita tuetaan välittäjärakenteella ja malliosallistujilla. Laukaisimet nähdään potentiaalisesti hyvinä rakenteina aktiokieleen, mutta niitä ei lähdetä toteuttamaan ennen kuin niitä on testattu kattavasti. Komponenttien määrittelemiseen käytetyt rakenteet, komponentti, komponenttiolio ja käyttörajapinta, nähdään onnistuneina lisäyksinä aktiokieleen.

# LÄHTEET

[1] Aktiopohjaisen ohjelmoinnin tukiryhmä, Tampere, 2016, Saatavissa (viitattu 12.12.2017): http://apotr.github.io/

[2] Actions Home Page, Tampere, 2014, Saatavissa (viitattu 5.4.2017): http://www.cs.tut.fi/~hmj/Actions/

[3] A. Mäkinen, Aktiojärjestelmä moniydinsuorittimelle, diplomityö, Tampereen teknillinen yliopisto, 2015

[4] R. J. Back, R. Kurki-Suonia, Distributed cooperation with action systems, ACM Transactions on Programming Languages and Systems (TOPLAS), 1988, s.513–554

[5] L. Lamport, The Temporal Logic of Actions, ACM Transactions on Programming Languages and Systems (TOPLAS), 1994. s.872–923

[6] The DisCo Home Page, Tampere, 1992, Saatavissa (viitattu 7.2.2018): http://disco.cs.tut.fi/index.html

[7] H.-M. Järvinen, The Design of a Specification Language of Reactive Systems, väitöskirja, Tampereen teknillinen korkeakoulu, 1992

[8] K. M. Chandy, J. Mistra, Parallel Program Design: A Foundation, Massachusetts, 1988

[9] H.-M. Järvinen, Actions, Objects and Subjects, The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications, 2013, s.285–291

[10] H.-M. Järvinen, Action-based Computing, seminaari, 2014, Tampereen teknillisessä yliopistossa