Jouni-Olavi Hätinen

Havainnoiva ohjelmointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Ohjelmistotekniikka

20.5.2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Jouni-Olavi Hätinen Havainnoiva ohjelmointi 51 sivua 20.5.2015
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaajat	Lehtori Simo Silander Lehtori Vesa Ollikainen

Ohjelmointiparadigma on tapa kuvata maailmaa ohjelmoinnin menetelmin. Jokaisella ohjelmointiparadigmalla täytyy tämän vuoksi olla jokin filosofinen pohja, joka vastaa niihin kysymyksiin, jotka määrittelevät pohjan koko maailmankuvallemme. Tämän insinöörityön tavoite oli verrata tunnetuimpia ohjelmointiparadigmoja tunnetuimpiin metafyysisiin teorioihin ja löytää yhteys filosofian ja ohjelmoinnin välillä.

Työssä tutustutaan funktionaaliseen, proseduraaliseen ja olio-ohjelmointiin ja jaetaan ne subjektiivisen, dualistisen ja objektiivisen käsitteen alle sillä perusteella, miten ne käsittelevät datan ja funktioiden välistä suhdetta. Työ käy läpi niin René Descartesin kuin Platoninkin teorioita vertailemalla niitä eri ohjelmointiparadigmoihin.

Tämän jälkeen työ esittelee samaa logiikkaa käyttäen Immanuel Kantin transsendentaaliseen idealismiin pohjautuvan havainnoivan ohjelmoinnin ohjelmointiparadigman. Työ käy läpi transsendentaalisen idealismin perusajatuksia ja niiden yhteyttä ohjelmointiin. Tämän jälkeen työ kuvaa havainnoivan ohjelman perusrakenteen ja esittelee lopuksi, miten havainnoivaa ohjelmointia on mahdollista toteuttaa käytännössä nykyisillä ohjelmointikielillä.

Havainnoivan ohjelmoinnin toteutuksen toimivuus osoittaa, että filosofisia teorioita on mahdollista hyödyntää ohjelmointiparadigmojen analysoinnissa ja kehityksessä. Tämä avaa uusia mahdollisuuksia ohjelmoinnin alalla, sillä nuori, jatkuvassa muutoksessa oleva ala saa filosofian kautta tuhansia vuosia pitkät juuret.

Avainsanat	ohjelmointiparadigma, filosofia, Immanuel Kant

Author Title Number of Pages Date	Jouni-Olavi Hätinen Sensible Programming 51 pages 20 May 2015
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructors	Simo Silander, Senior Lecturer Vesa Ollikainen, Senior Lecturer

A programming paradigm is a way to describe the world by means of programming. Therefore, every programming paradigm must have a philosophical foundation that answers the questions defining the base of our conception of the world. The goal of this thesis was to compare the most renowned programming paradigms to the most renowned metaphysical theories and find the link between philosophy and programming.

The thesis examines functional, procedural and object-oriented programming and categorises them under subjective, dualistic and objective concepts on the basis of their way to handle the relationship between data and functions. It examines several philosophers, including René Descartes and Plato by comparing their theories to various programming paradigms.

After this the thesis implements the link between philosophy and programming by introducing a new programming paradigm called ”sensible programming”. Sensible programming is based on Immanuel Kant's transcendental idealism. The thesis examines the basic ideas of transcendental idealism and their link to programming. After this the thesis describes the basic structure of a program using sensible programming paradigm and demonstrates how it is possible to implement sensible programming with the programming languages we use today.

The functionality of the sensible programming paradigm proves that it is possible to use philosophical theories to analyse and develop programming paradigms. This opens new possibilities in the field of programming, because through philosophy this young, constantly evolving field obtains roots that are thousands of years old.

Keywords	programming paradigm, philosophy, Immanuel Kant

Metropolia

Sisällys

1 Johdanto 1

2 Taustaa 2

2.1 Ohjelmointiparadigman tehtävät 2
2.2 Ohjelmointiparadigmojen erottelu subjektiivisiin, objektiivisiin ja dualistisiin 3
2.3 Subjektiivinen ohjelmointi 5
2.3.1 Periaatteet ja toteutus 5
2.3.2 Vahvuudet ja heikkoudet 6
2.3.3 Subjektiivinen vai subjektisuuntautunut ohjelmointi 7
2.4 Objektiivinen ohjelmointi 8
2.4.1 Periaatteet ja toteutus 8
2.4.2 Vahvuudet ja heikkoudet 9
2.5 Dualistinen ohjelmointi 10
2.5.1 Periaatteet ja toteutus 10
2.5.2 Vahvuudet ja heikkoudet 11

3 Havainnoiva ohjelmointi 12

3.1 Ajatuksia proseduraalisesta ja olio-ohjelmoinnista 12
3.1.1 Havainnoivan ohjelmoinnin tavoitteet 15
3.2 Transsendentaalinen idealismi 17
3.2.1 Ilmiöt 18
3.2.2 Aistit 20
3.2.3 Ymmärrys 21
3.2.4 Arvostelmakyky ja skematismi 23
3.2.5 Transsendentaalisen idealismin yhteys havainnoivaan ohjelmointiin 24

4 Havainnoivan ohjelman rakenne 26

4.1 Pääohjelma 26
4.2 Käyttäjä 28
4.2.1 Syötteenhallinta 30
4.2.2 Tunnistusmoduulit 30
4.2.3 Toteutusfunktiot 32
4.3 Tietopohja 32
4.3.1 Näkymät 33
4.3.2 Objektit 34

4.4 Ohjelman suoritus 36

5 Havainnoivan ohjelman toteutus 38

5.1 Ohjelmointikielen vaatimukset 38
5.2 Toteutusesimerkkejä 39
5.3 Taso 1: Muuttujat ja lauseet 40
5.4 Taso 2: Tietueet ja toteutusfunktiot 41
5.5 Taso 3: Objektit ja tunnistusmoduulit 42
5.6 Taso 4: Näkymät ja syötteenhallinta 44
5.7 Taso 5: Tietopohja ja käyttäjä 46
5.8 Havainnoiva ohjelmointi opetuskäytössä 48

6 Lopuksi 50

Lähteet 51

Sanastoa

Reaalimaailma	**Ohjelmointi**
a posteriori	ajonaikainen
a priori	käännöksenaikainen
aika	ohjelman suorituksen kulku
aisti	rajapinta
analyyttinen	final
arvostelma	lause tunnistusmoduulissa
avaruus	muistiavaruus
dualistinen	funktiot ja tietorakenteet erotteleva
hallusinaatio	paikallinen muuttuja
havainnoiva/ havaintopohjainen	funktiot ja tietorakenteet erotteleva, mutta rajapintojen taakse piilottava
havainto	rajapinnan kutsu ja rajapinnan parametrien tunnistus
havaitsija	käyttäjä
ilmiö	objekti
intuitio	parametri
kohde	objekti parametrina
käsite	toteutusfunktio
maailma	tietopohja
objektiivinen	funktiot tietorakenteiden sisällä määrittelevä
olio	tuntematon
ominaisuus	muuttuja
päätelmä	lause toteutusfunktiossa
skeema	tunnistusmoduuli
subjektiivinen	tietorakenteet funktioiden sisällä määrittelevä
synteettinen	ei-final
tahto	syötteenhallinta
tavoitteet	syötteet
tieto	data
todellisuus	pääohjelma
tunnistus	tunnistusmoduuli
tyyppi	tietue
välittäjä	kontrolleri
ymmärrys	kyky toteuttaa funktioita lauseiden avulla
ympäristö	näkymä

1 Johdanto

Ohjelmointiparadigma on tapa kuvata maailmaa ohjelmoinnin menetelmin. Se on perinteisesti ollut työkalu erityisesti teknisten ja matemaattisten ongelmien ratkaisuun, ja sen rakenteet vaikuttavat vahvasti siihen, miten paradigmaa käyttävät ohjelmointikielet sisäisesti toteutetaan. Sovellusalueiden tullessa lähemmäs arki- ja työelämäämme siitä on kuitenkin tullut myös tapa, jolla kuvataan ympäröivää maailmaa sekä omaa asemaamme maailmassa. Se on näkökulma asioihin, joiden kanssa olemme tekemisissä päivittäin. Tämän näkökulman tarkentamiseksi sen tulee vastata kysymyksiin, jotka määrittelevät perustan koko maailmankuvallemme: mitä on olemassa, miten se on olemassa ja mitä me voimme tietää siitä, mikä on olemassa.

Yhden tunnetuimman vastauksen näihin kysymyksiin esitti Immanuel Kant teoksessaan ”Puhtaan järjen kritiikki” (Kant 2013). Teos kuvaa maailman aistien, ymmärryksen ja ilmiöiden vuorovaikutukseksi, jossa puhtaille totuuksille ja pysyville käsitteille jää hyvin vähän tilaa. Sen lisäksi, että emme Kantin mukaan koekaan maailmaa sellaisena kuin se on vaan aistiemme ja ymmärryksemme muokkaamina, ymmärryksemmekin muuttuu jatkuvasti uusien kokemusten myötä. Monimutkaisimpien ohjelmoinnin ongelmien edessä meidän tuleekin kysyä, voimmeko enää kuvata ohjelmistojemme rakenteita paradigmoilla, jotka pyrkivät yksinkertaistamaan maailman filosofian näkökulmasta?

Tämä insinöörityö pyrkii vastaamaan tuohon kysymykseen paneutumalla eri ohjelmointiparadigmojen filosofiseen pohjaan ja esittelemällä Kantin transsendentaaliseen idealismiin pohjautuvan havainnoivan ohjelmoinnin teorian. Työssä kehitetty teoria esittää ohjelmoinnin rakenteita ohjelman käyttäjän huomioon ottavasta näkökulmasta ja nostaa esille uusia tapoja käsitellä ohjelman tietosisältöä. Lopuksi työ esittelee erilaisia tapoja, joilla havainnoivaa ohjelmointia voi toteuttaa käytännössä nykyisillä ohjelmointikielillä.

2 Taustaa

2.1 Ohjelmointiparadigman tehtävät

Ohjelmointiparadigma on tapa kuvata maailmaa ohjelmoinnin menetelmin, mutta se on myös tapa ratkaista maailmassa esiintyviä ongelmia. Sen tarkoitus on tarjota työkaluja, joilla ohjelmoija pystyy havainnollistamaan ja ratkaisemaan työ- ja arkielämään liittyviä ongelmia ohjelmoinnin avulla. Ongelmien ratkaisutavat liittyvät usein tiedonkäsittelyn automatisointiin tai tiedon havainnollistamiseen käyttäjälle.

Jotta ohjelmointiparadigma pystyisi ratkaisemaan sille asetetut ongelmat, sen tulee pystyä kuvaamaan ne siten, että ongelmien kuvaukset ja kuvausten avulla luodut ratkaisut ovat aina ohjelman ulkopuolelta tarkasteltuna ennustettavia ja selkeitä. Ennustettavuudella tarkoitetaan tässä sitä, että samoilla käyttäjän antamilla ennakkoehdoilla suoritettuna ohjelma käyttäytyy aina tietyllä ymmärrettävällä tavalla, ja selkeydellä tarkoitetaan sitä, että nimenomaan käyttäjä pystyy ymmärtämään, millä tavalla ohjelma seuraavaksi käyttäytyy.

Sekä ohjelman ennustettavuus että selkeys riippuvat vahvasti käytetyn ohjelmointiparadigman sisäisestä rakenteesta. Ohjelma, joka koostuu monista erilaista, heterogeenisistä tietorakenteista mutta samankaltaisista, homogeenisistä suoritusrakenteista, hyötyy ennustettavuuden kannalta paradigmasta, joka määrittelee ensin tietorakenteet ja kunkin tietorakenteen yhteyteen omat käsittelyfunktionsa (kuten esimerkiksi olio-ohjelmoinnissa). Tällainen ohjelma voisi olla esimerkiksi relaatiotietokantaan pohjautuva asiakastietoja käsittelevä sovellus.

Toisaalta ohjelma, joka koostuu hyvin samankaltaisista tietorakenteista mutta monista erityyppisistä suoritusrakenteista toimii ennustettavammin paradigmalla, joka toimii juuri päinvastoin, eli määrittelee ensin ohjelman suoritukseen vaadittavat funktiot ja jokaisen funktion yhteyteen omat tietorakenteensa (kuten esimerkiksi funktionaalisessa ohjelmoinnissa). Tällainen ohjelma voisi olla esimerkiksi tiedonlouhintasovellus, joka käsittelee CSV-tiedostoista luettua tietoa ennalta määriteltyjen operaattoreiden avulla.

Näillä molemmilla lähestymistavoilla on omat käyttötarkoituksensa riippuen siitä, halutaanko ohjelmassa korostaa tietorakenteita vai suoritusrakenteita, mutta

kumpikaan niistä ei vastaa suoraan kysymykseen, miten kuvata ohjelmia, jotka koostuvat sekä lukuisista erilaisista tietorakenteista että lukuisista erilaisista suoritusrakenteista.

Tässä työssä pyritään löytämään vastaus tuohon kysymykseen paneutumalla syvällisesti ohjelmointiparadigmojen filosofiseen pohjaan ja kehittämällä sitä kautta uusi tapa kuvata ohjelman tieto- ja suoritusrakenteita. Vaikka filosofia on yhdistetty ohjelmointiparadigmoihin silloin tällöin (mm. Harrison), laajasti sovellettavissa oleva selkeä yhteys ohjelmien rakenteiden ja filosofisten teorioiden välillä on toistaiseksi jäänyt pimentoon. Tämän yhteyden löytäminen olisi kuitenkin hyvin merkittävää, sillä mikäli filosofia olisi mahdollista yhdistää ohjelmointiin, voisi ohjelmointiparadigmoja tulevaisuudessa arvioida, soveltaa ja kehittää filosofisia teorioita hyödyntäen.

Seuraavissa luvuissa esitellään uusi tapa, joka yhdistää ohjelmointiparadigmoja tunnettuihin filosofisiin teorioihin. Tapa jakaa ohjelmointiparadigmat subjektiivisiin, objektiivisiin ja dualistisiin riippuen niiden tavasta käsitellä datan ja funktioiden välistä suhdetta.

2.2 Ohjelmointiparadigmojen erottelu subjektiivisiin, objektiivisiin ja dualistisiin

Vaikka ohjelmointia on perinteisesti ajateltu teknisten ja matemaattisten ongelmien ratkaisutapana, kaikilla ohjelmointiparadigmoilla on kuitenkin joko tietoisesti tai tiedostamatta luotu filosofinen pohja. Pohja vaikuttaa yleisellä tasolla ohjelman rakenteeseen siten, rakentuuko ohjelma tietorakenteen vai suoritusrakenteen ympärille. Tällä tavalla paradigmat on mahdollista jakaa subjektiivisiin (suoritusrakenteen ympärille rakentuviin) ja objektiivisiin (tietorakenteen ympärille rakentuviin) paradigmoihin. Lähemmin tarkasteltuna tämä jaottelu vaikuttaa ohjelman tiedonkulkuun, testaukseen, uudelleenkäytettävyyteen ja vastuuseen datasta.

Tässä työssä käsitellään kolmea suosittua ohjelmointiparadigmaa: funktionaalista ohjelmointia, proseduraalista ohjelmointia ja olio-ohjelmointia. Olio-ohjelmointiin suunniteltuja kieliä ovat mm. C++ ja Java. Proseduraaliseen ohjelmointiin suunniteltuja kieliä ovat mm. C ja PHP. Funktionaaliseen ohjelmointiin suunniteltuja kieliä ovat mm. Erlang ja Haskell. Monet kielet tukevat useita eri ohjelmointiparadigmoja, mutta kielet on usein alun perin suunniteltu nimenomaan tiettyä paradigmaa ajatellen. (Comparison of programming languages 2015.)

Yhteistä näille kaikille ohjelmointiparadigmoille on ohjelman rakenteen jako dataan sekä dataa käsitteleviin funktioihin. Data koostuu muistiin tallennetuista ohjelman muuttujista, jotka on usein koottu yhteen tietorakenteiksi, joita kutsutaan paradigmasta riippuen joko tietueiksi tai objekteiksi. Funktiot taas koostuvat ohjelmariveistä, jotka suorittavat käskyjä muuttujille. Funktioiden tehtävä on käsitellä tietoa. Mikäli pohdimme funktioita filosofisesta näkökulmasta, ne muistuttavat paljon ihmisen ymmärrystä, eli sitä, miten teemme päätelmiä tiedosta, jota keräämme maailmasta. Näin ajateltuna funktiot vastaavat siis kirjaimellisesti mielemme funktioita, ja funktioiden ulkopuoliset muuttujat ovat meille tietoa ulkoisesta maailmasta.

Mikäli funktiot määrittelevät ja käsittelevät kaikki tietorakenteet itsenäisesti, jolloin tietorakenteet ovat riippumattomia funktion ulkopuolisesta ohjelmasta, voidaan sanoa, että funktiot toimivat subjektiivisesti. Tällöin mikään funktioiden ulkopuolinen osa ei määrittele funktioiden käsittelemän datan muotoa, eikä funktioiden välillä ole jaettuja tietorakenteita tai jaettua tietosisältöä. Toisaalta mikäli ohjelma määrittelee kaikki tietorakenteet funktioiden ulkopuolella ja sallii vain ennalta määrättyjen funktioiden pääsyn kuhunkin tietorakenteeseen, jolloin niitä voidaan käsitellä vain yhdellä, ennalta määritellyllä tavalla, voidaan sanoa, että funktiot toimivat objektiivisesti. Tämä saa aikaan ohjelmointiparadigmoissa jaottelun, joka määrää sen, onko paradigma subjektiivinen vai objektiivinen (kuva 1).

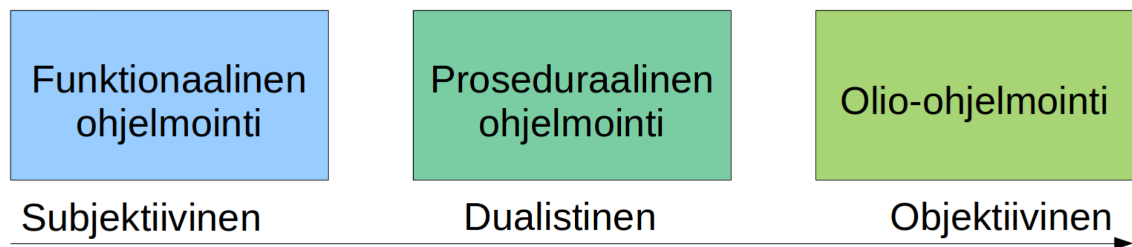
Paradigma on subjektiivinen silloin, kun se ei salli jaettua dataa funktioiden välillä. Subjektiivisessa ohjelmoinnissa tiettyä muistissa olevaa dataa voidaan siis käsitellä vain yhdessä, ennalta määritellyssä funktiossa. Funktionaalinen ohjelmointi, jossa kaikki tietorakenteet määritellään funktioiden sisällä eikä funktioiden välillä ole jaettua dataa (Functional programming 2014: 1), on siis subjektiivista.

Paradigma on objektiivinen silloin, kun se määrittelee tietorakenteet pysyvästi etukäteen ja sallii vain ennalta määrättyjen funktioiden pääsyn kuhunkin tietorakenteeseen. Objektiivisessa paradigmassa tietorakenteita voidaan siis käsitellä vain yhdellä, ennalta määritellyllä tavalla, ja myös tietorakenteiden muoto on ennalta määrätty. Olio-ohjelmointi, jossa tietorakenteet ja niitä käsittelevät funktiot, eli metodit, kapseloidaan samaan luokkaan, on siis objektiivista.

Yksinkertaistetusti voidaan sanoa, että ohjelmointiparadigma on objektiivinen silloin, kun data määrittelee funktiot ja subjektiivinen silloin, kun funktio määrittelee datan.

Objektiivisen mallin mukaan meillä ei voi olla omia käsityksiä maailmasta, vaan se esiintyy meille objektiivisesti tietynlaisena. Subjektiivisen mallin mukaan taas me emme jaa samaa maailmaa toisten havaitsijoiden kanssa, vaan kokemamme maailma on yksilöllisesti ainoastaan mielemme sisällä ja riippumaton muista havaitsijoista.

Proseduraalinen ohjelmointi sijoittuu subjektiivisen ja objektiivisen ajattelutavan välimaastoon. Siinä data ja funktiot on erotettu toisistaan, mikä tekee paradigmasta dualistisen. Toisaalta proseduraalinen ohjelmointi sallii objektiivisen jaetun datan funktioiden välillä, mutta toisaalta se myös sallii subjektiivisen datan vapaan käsittelyn. Dualistisessa ohjelmoinnissa tietorakenteet ovat siis objektiivisia, mutta suoritusrakenteet ovat subjektiivisia.

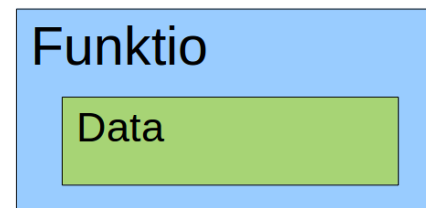


Kuva 1: Ohjelmointiparadigmojen jako subjektiivisiin, dualistisiin ja objektiivisiin.

2.3 Subjektiivinen ohjelmointi

2.3.1 Periaatteet ja toteutus

Subjektiivisen ohjelmoinnin periaate on ohjelman suorituksen ja toimintojen korostaminen tietorakenteiden sijaan (kuva 2). Siinä missä subjektiivinen filosofia korostaa ulkoisen maailman riippuvuutta yksilön kokemuksesta (Subjectivism 2014), subjektiivinen ohjelmointi korostaa tietorakenteiden riippuvuutta yksittäisistä funktioista. Puhdas subjektiivinen näkökulma vie periaatteen niin pitkälle, että tietorakenteita ei ole lainkaan funktioiden ulkopuolella, ja täten ulkoista maailmaakaan ei ole yksilön kokemuksesta riippumatta.



Kuva 2: Subjektiivisessa ohjelmoinnissa data sisältyy funktioihin.

Subjektiivinen ohjelmointi perustuu filosofiseen solipsismiin, jonka suurimpia vaikuttajia olivat kreikkalainen Gorgias ja ranskalainen René Descartes (Higgins: 2a; Thornton: 1). Gorgias esitti teoksessaan ”Melissoksesta, Ksenofaneesta ja Gorgiaasta” väitteen, jonka mukaan mitään ei ole olemassa, ja mikäli jotain on olemassa, siitä ei voida tietää

mitään, ja mikäli jotain voidaan tietää, sitä tietoa ei voida kommunikoida muille. (Higgins: 2a.) Tätä väittämää voidaan soveltaa suoraan subjektiivisen ohjelmoinnin periaatteeseen, jossa funktioiden ulkopuolisia muuttujia ei ole, ja jos näin onkin, niitä ei ole mahdollista jakaa funktioiden välillä.

Mikäli kuvaamme funktioita kykynä ajatella, René Descartesin tunnetuinta lausetta: ”Ajattelen, siis olen” (Descartes 2005: 2), voisi havainnollistaa yksinkertaisen tulostuskomentotestin avulla. Eräs tapa testata, suoritetaanko tietty kohta ohjelmasta, on lisätä tulostuskomento kyseiseen kohtaan. Mikäli ruudulle ilmestyy tulostuskomennon mukainen teksti, kyseinen kohta ohjelmasta suoritetaan varmasti, vaikka mistään muusta ohjelman testaaja ei voisikaan olla varma. Aivan kuten Descartes päätteli: koska minä ajattelen, minun täytyy myös olla olemassa (kuva 3).

```
printf(”Cogito, ergo sum.”);
```

Kuva 3: "Ajattelen, siis olen." - René Descartesin tapa testata ohjelman toimintaa.

Tunnetuin subjektiivisen ohjelmoinnin paradigma on funktionaalinen ohjelmointi. Funktionaalisen ohjelman rakenne koostuu funktioista, jotka koostuvat lausekkeista (expressions). Funktiot eivät jaa dataa keskenään, vaan kaikki toiminta tapahtuu arvoparametrien ja paluuarvojen avulla. Tällaisia funktioita kutsutaan puhtaiksi funktioiksi (pure functions). Puhtaan funktion määritelmä on se, ettei sillä saa olla sivuvaikutuksia, eli se ei saa vaikuttaa muualla ohjelmassa käytettyjen muuttujien arvoihin. (Functional programming 2014: 1-2.)

2.3.2 Vahvuudet ja heikkoudet

Subjektiivisen ohjelmoinnin eräs vahvuus on funktioiden testattavuus. Koska subjektiivisen ohjelmoinnin funktioiden käyttäytyminen ei riipu mistään funktion ulkopuolisesta datasta, funktiot on mahdollista testata luotettavasti muuttamalla ainoastaan funktion saamien parametrien arvoja.

Toinen vahvuus subjektiivisella ohjelmoinnilla on mahdollisuus käyttää yksittäisiä funktioita uudelleen toisissa ohjelmissa. Testattavuuden lisäksi funktioiden riippumattomuus datasta edesauttaa sitä, että funktioita on helppo käyttää uudelleen,

koska funktion käyttäjän ei tarvitse tuntea funktion toteutusta vaan ainoastaan sen parametrit ja paluuarvo.

Kolmas subjektiivisen ohjelmoinnin vahvuus on funktioiden rinnakkainajo. Koska funktiot eivät jaa tietoa keskenään, niillä ei ole sivuvaikutuksia toisiinsa, ja ne voidaan suorittaa hyvin ennalta määrittelemättömässä järjestyksessä, jopa rinnakkain.

Siinä missä subjektiivisen ohjelmoinnin vahvuus on funktioiden riippumattomuus tietorakenteista, sen heikkous on tietorakenteiden riippuvuus funktioista. Mikäli ohjelmassa ilmeneekin tarve käyttää yhteisiä tietorakenteita eri funktioille, muutokset näihin tietorakenteisiin vaikuttavat ennalta-arvaamattomasti moniin eri funktioihin. Koska kaikki funktioiden käyttämä ulkopuolinen data täytyy syöttää funktioille parametreina, jokainen muutos tietorakenteisiin saattaa aiheuttaa muutoksia funktiokutsuihin läpi ohjelman.

Tämä ei ole ongelma ainoastaan silloin, kun tietorakenteet muuttuvat ajassa, esimerkiksi ohjelman uudelleenorganisoinnin myötä. Se on erityisen suuri ongelma silloin, kun ne muuttuvat ohjelman suorituksen aikana, eli silloin, kun samoja toimintoja suoritetaan useille erilaisille tietorakenteille. Juuri tämän vuoksi subjektiivinen ohjelmointi soveltuu hyvin ennen kaikkea sellaisiin ohjelmiin, joiden tietorakenne on yksinkertainen ja pieni, mutta huonosti sellaisiin ohjelmiin, joiden tietorakenne on monimutkainen ja suuri.

2.3.3 Subjektiivinen vai subjektisuuntautunut ohjelmointi

Subjektin huomioon ottavaa näkökulmaa ohjelmiston rakenteessa käsittelee myös William Harrisonin ja Harold Ossherin 90-luvun alkupuolella kehittämä subjektisuuntautunut ohjelmointi (subject-oriented programming, Harrison 1993). Subjektisuuntautunut ohjelmointi on ohjelmointiparadigma, joka keskittyy yhden objektiivisen näkemyksen sijaan yksittäisten subjektien näkemykseen ohjelman toiminnasta. Paradigman mukaan suurten ohjelmien toteutus helpottuu, mikäli suunnittelijoiden ei tarvitse suunnitella objektiivista tietorakennetta, joka kuvaa maailman todellisia piirteitä ja on samanlainen kaikkialla ohjelmassa vaan ainoastaan subjektiivisia tietorakenteita, jotka kuvaavat yksittäisten subjektien käsitystä maailmasta. (Harrison 1993: 1.)

Subjekti reaalimaailman analogiana voisi olla esimerkiksi metsurin näkemys puusta. Se sisältää muuttujat, jotka kuvaavat puun lajia, pituutta, paksuutta ja sitä, onko puu kaadettu vai ei. Lisäksi subjekti sisältää funktiot, jotka käsittelevät näitä muuttujia. Toinen subjekti voisi olla maanomistajan näkemys puusta. Se sisältää esimerkiksi puun arvoa kuvaavan muuttujan sekä sitä käsittelevän funktion.

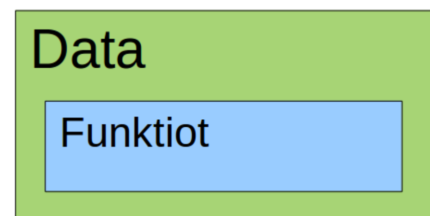
Käytännössä subjektisuuntautunut ohjelmointi toteutetaan olio-ohjelmointia mukaillen siten, että ohjelma koostuu luokista, jotka koostuvat muuttujista ja muuttujia käsittelevistä funktioista. Objektien sijaan luokat kuitenkin kuvaavat subjekteja ja subjektien näkemystä maailmasta. (Harrison 1993: 3.2.)

Subjektisuuntautunut ohjelmointi ei nimestään huolimatta ole subjektiivista siten kuin se tässä työssä on esitetty. Tämä johtuu siitä, että subjektisuuntautuneessa ohjelmoinnissa funktiot eivät määrittele dataa vaan päinvastoin. Mitä tahansa subjektiivista tietoa ohjelma siis käsitteleekin, se käsittelee sitä vain yhdellä, ennalta määrätyllä tietorakenteella ja yhdellä, ennalta määrätyllä tavalla. Subjektisuuntautunut ohjelmointi on siis pohjimmiltaan objektiivista ohjelmointia, jonka periaatteet ja rakenne esitellään seuraavassa luvussa.

2.4 Objektiivinen ohjelmointi

2.4.1 Periaatteet ja toteutus

Objektiivisen ohjelmoinnin periaate on korostaa tietorakenteiden merkitystä liittämällä tietorakenteita käsittelevät funktiot suoraan niiden yhteyteen (kuva 4). Siinä missä objektiivinen filosofia korostaa sitä, että maailma koostuu universaaleista totuuksista, joihin ei voi olla yksilöllisiä näkemyseroja (Mulder: 1), objektiivinen ohjelmointi korostaa sitä, että sekä tietorakenteet että tietorakenteita käsittelevät funktiot on määritelty ennalta, eikä voi olla useita erilaisia tapoja jäsentää tietoa tai käsitellä tiettyä tietorakennetta käyttäen samaa rajapintaa. Objektiivinen ohjelmointi pyrkii kuvaamaan maailman sellaisena, kuin se on, tai ainakin sellaisena, kuin se meille kaikille objektiivisesti näyttäytyy.



Kuva 4: Objektiivisessa ohjelmoinnissa funktiot sisältyvät dataan.

Tunnetuin objektiivisen ohjelmoinnin paradigma on olio-ohjelmointi. Olio-ohjelmoinnin funktiot on yhdistetty käsiteltävien muuttujien kanssa luokkiin, ja muuttujat on piilotettu

(ts. kapseloitu) luokan sisälle niin, että ulkopuolinen tarkkailija ei pääse niitä näkemään. Ohjelman toimintaa tarkkaillaan ulkopuolisena kolmannen persoonan näkökulmasta.

Eräs tunnettu erityisesti olio-ohjelmointia kuvaava filosofinen teoria on platoninen realismi. Platoninen realismi on Platonin kehittämä metafyysinen oppi, joka jakaa todellisuuden aistimaailmamme ulkopuolisiin universaaleihin sekä aistiemme havaitsemiin partikulaareihin. Oppi kutsuu universaaleja ”ideoiksi” ja partikulaareja ”olioiksi”.

Opin mukaan ideat ovat olemassa omassa maailmassaan, joka on havaittavissa vain järjen avulla, kun taas oliot ovat epätäydellisiä jäljitelmiä näistä ideoista. Meillä voi olla esimerkiksi mielessämme idea puusta, jonka jäljitelmän havaitsemme oliona katsoessamme ulos ikkunasta. Mikään olio ei voi kuvata ideaa täydellisesti, vaan kaikki oliot ovat vain varjoja varsinaisista ideoista. Ideat ovat objektiivisia siinä mielessä, että ne ovat universaaleja ja ennalta määriteltyjä sekä riippumattomia aistimaailmassa tapahtuvista muutoksista. Oliot taas ovat objektiivisia siinä mielessä, että ne ovat riippumattomia havaitsijasta. (Balaguer 2009: 1.)

Olio-ohjelmointi ottaa erityisesti vahvoja vaikutteita juuri platonisesta realismista. Olio-ohjelma koostuu luokista (ideoista), joita emme kuitenkaan käsittele suoraan vaan olioiden (partikulaarien) kautta. Oliot eivät koskaan kuvaa luokkaansa käyttäjälleen täydellisesti vaan rajapintojen (varjojen) avulla. Luokat ovat muuttumattomia ja riippumattomia ohjelman suorituksesta. Oliot taas ovat riippumattomia käyttäjästä, joka kutsuu sen toteuttamia rajapintoja. Olio-ohjelmointi on siis objektiivista, eli subjektista riippumatonta ohjelmointia.

2.4.2 Vahvuudet ja heikkoudet

Objektiivisen ohjelmoinnin vahvuus on funktiokutsujen riippumattomuus tietorakenteista. Koska funktiot ovat suoraan tietorakenteen määrittelemiä, niillä on vapaa pääsy tietorakenteeseen eikä funktion kutsujan tarvitse tuntea tietorakennetta lainkaan. Tämä on erityisen tärkeää silloin, kun käsiteltävät tietorakenteet ovat monimutkaisia ja saattavat muuttua ohjelman kehityksen aikana tai ohjelman sisällä. Objektiivinen ohjelmointi sietää siis hyvin tietorakenteiden muutoksia, mikä on etenkin muuttuvissa ohjelmistoprojekteissa todella hyödyllinen ominaisuus.

Objektiivisen ohjelmoinnin eräitä heikkouksia ovat funktioiden uudelleenkäytettävyys ja testattavuus. Koska funktiot ovat tiukasti kiinni tietorakenteessaan, niistä on lähes mahdoton tehdä yleiskäyttöisiä. Myös funktioiden testaaminen on vaikeaa, koska ulkopuolisen on mahdoton tietää varmasti, mitä tietorakenteen muuttujia funktio todellisuudessa käyttää.

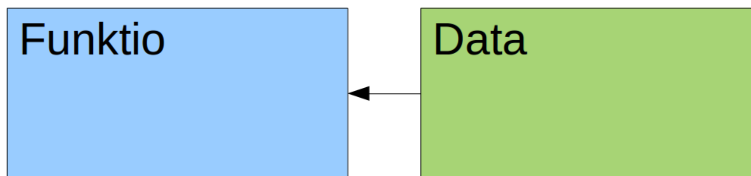
Mahdollisesti suurin heikkous objektiivisessa ohjelmoinnissa on kuitenkin tietorakenteiden välinen kommunikointi. Täydellisesti objektiivisessa ohjelmoinnissa kaikki toiminta tapahtuu kapseloidusti tietorakenteiden sisällä. Monesti toteutusten täytyy kuitenkin sisältää toimintoja, joissa useat tietorakenteet toimivat keskenään. Koska kaikki funktiot on toteutettu tietorakenteiden yhteydessä, tämä aiheuttaa sen, että tietorakenteet tulevat välttämättä riippuvaisiksi toisistaan.

Joe Armstrong, Erlang-ohjelmointikielen kehittäjä kuvasi tätä riippuvaisuutta erityisesti olio-ohjelmoinnin osalta seuraavasti: ”Halusit banaanin, mutta sait banaania pitelevän gorillan ja koko viidakon.” Armstrongin mukaan funktioiden uudelleenkäytettävyys on erityisesti olio-ohjelmoinnin ongelma siksi, koska olio-ohjelmoinnin luokat kuuluvat riippuvuuksiensa vuoksi johonkin ympäristöön, jota ne kantavat välttämättä mukanaan. (Seibel 2009: 221.)

2.5 Dualistinen ohjelmointi

2.5.1 Periaatteet ja toteutus

Dualistinen ohjelmointi ottaa subjektiivisesta ja objektiivisesta ohjelmoinnista eroavan näkökulman erottamalla tietorakenteet ja funktiot toisistaan. Näkökulma pohjautuu filosofiaan, jonka mukaan meillä on olemassa aineellisesta maailmasta erotettavissa oleva mieli tai sielu sekä ulkoinen, aineellinen maailma, jota mielen avulla käsittelemme (Robinson 2011: 1). Tämä nykyaikainen käsitys dualismista (nk. kartesiolainen dualismi) on alun perin peräisin ranskalaiselta René Descartesilta (Robinson 2011: 1.2). Dualistisessa ohjelmoinnissa data ja sitä käsittelevät funktiot erotetaan toisistaan niin, että kaikilla ohjelman funktioilla on sisäisten muuttujiensa lisäksi yhteinen, jaettu tietorakenne (kuva 5).



Kuva 5: Dualistisessa ohjelmoinnissa funktiot ja data erotetaan toisistaan.

Tunnetuin dualistisen ohjelmoinnin paradigma on proseduraalinen ohjelmointi. Proseduraalisessa ohjelmoinnissa dataa käsitellään funktioilla, jotka voivat saada käsiteltävän datan joko parametreina tai globaaleina muuttujina funktioiden ulkopuolelta. Samoja muuttujia voidaan jakaa funktioiden kesken, ja funktiot voivat muuttaa muualla ohjelmassa käytettyjen muuttujien arvoja suoraan.

2.5.2 Vahvuudet ja heikkoudet

Dualistisen ohjelmoinnin suurin vahvuus on sen joustavuus tietorakenteiden ja suoritusrakenteiden suunnittelussa ja toteutuksessa. Koska tietorakenteet ja funktiot eivät ole suoraan riippuvaisia toisistaan, niiden avulla on mahdollista kuvata monimutkaisia rakenteita, kuten esimerkiksi funktioita, joissa käsitellään eri objekteihin liittyviä tietorakenteita tai funktioita, joiden toteutus saattaa vaihtua näkökulmasta riippuen.

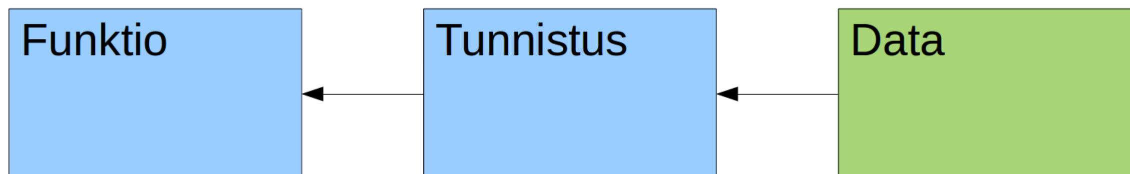
Dualistisen ohjelmoinnin suurin vahvuus on myös sen suurin heikkous. Koska funktiot ja tietorakenteet on erotettu toisistaan, muutokset tietorakenteissa saattavat aiheuttaa ennustamattomia sivuvaikutuksia ohjelman suorituksessa kuten subjektiivisessa ohjelmoinnissa. Toteutustavasta riippuen dualistinen ohjelmointi kantaa mukanaan myös objektiivisen ohjelmoinnin heikkoudet. Jos ohjelmassa on käytetty kaikkialla ohjelmassa näkyviä globaaleja muuttujia, kaikkien ohjelman funktioiden testattavuus ja uudelleenkäyttö kärsii. Dualistinen ohjelmointitapa tarjoaa siis vapauksia, mutta vaatii sen, että ohjelman kirjoittaja tuntee ohjelman toiminnan täysin, mikä on suuremmissa projekteissa lähes mahdoton vaatimus.

Dualistiseen ohjelmointiin osittain perustuva havainnoivan ohjelmoinnin paradigma pyrkii korjaamaan nämä ongelmat esittelemällä erillisen rajapinnan funktioiden ja tietorakenteiden väliin. Näin tietorakenteiden toteutus on mahdollista piilottaa funktioilta, eikä rajapinnan kutsujan tarvitse tietää tietorakenteiden toteutusta.

3 Havainnoiva ohjelmointi

Tässä työssä kehitetty havainnoiva ohjelmointi on ohjelmointiparadigma, jonka tarkoitus on kehittää dualistista ohjelmointia rakenteellisemmaksi, jotta sillä saavutettaisiin sekä subjektiivisen että objektiivisen ohjelmoinnin edut ilman niiden haittoja. Tähän se pyrkii lisäämällä funktioiden ja käsiteltävien tietorakenteiden väliin tunnistusmoduulin, joka määrittelee, mitä funktioita missäkin tilanteessa kutsutaan ja mitä muuttujia funktioille annetaan parametreiksi (kuva 6). Nimensä havainnoiva ohjelmointi saa pyrkimyksestään kuvata sitä, miten saamme tietoa maailmasta havaintojen avulla. Havainnoilla tarkoitetaan tässä tiedon vastaanottamista ja tunnistamista ennen tiedon käsittelyä.

Periaatteellisena pohjana datan ja funktioiden erotteluun havainnoiva ohjelmointi käyttää proseduraalista ohjelmointia, mutta paradigman toteutus tapahtuu olio-ohjelmointikielellä, koska olio-ohjelmointi tarjoaa proseduraalisia ohjelmointikieliä paremmat valmiudet ohjelmiston rakenteellisuuden toteuttamiseen. Havainnoivassa ohjelmoinnissa funktiot saavat käsittelemänsä datan ainoastaan parametrien kautta, joten vaikka funktiot eivät olekaan puhtaita, vaikutteita on otettu myös funktionaalisesta ohjelmoinnista.



Kuva 6: Havainnoivassa ohjelmoinnissa funktioilla ei ole suoraa pääsyä tietorakenteisiin, vaan käsiteltävä data annetaan funktioille tunnistusmoduulien kautta.

3.1 Ajatuksia proseduraalisesta ja olio-ohjelmoinnista

Proseduraalinen ohjelmointi ja olio-ohjelmointi ovat tämän päivän kaksi suosituinta ohjelmointiparadigmaa. Koska havainnoiva ohjelmointi käyttää pohjanaan hieman näitä molempia, on syytä peilata havainnoivaa ohjelmointia nimenomaan niihin. Huomattavaa on, että vaikka proseduraalinen ohjelmointi mahdollistaa ohjelmien toteutuksen monilla eri tavoilla, yleisin toteutustapa on lähempänä subjektiivista, eli funktionaalista ohjelmointia kuin objektiivista, eli olio-ohjelmointia. Erityisesti sen vuoksi tämä vertailu on erityisen mielekäs.

Sekä proseduraalisessa että olio-ohjelmoinnissa kaikki ohjelman toiminnot tapahtuvat funktioiden sisällä pääasiassa funktioiden ulkopuolisen datan avulla. Funktioilla on kaksi tapaa ottaa vastaan käsiteltävä ulkopuolinen data: parametreilla (subjektiivinen tapa) tai funktion ulkopuolisilla julkisilla muuttujilla (objektiivinen tapa). Julkisilla muuttujilla tarkoitetaan tässä tapauksessa funktion ulkopuolisia muuttujia, joita ei anneta funktioille parametreina, mutta jotka silti näkyvät ja ovat käsiteltävissä funktioiden sisällä. Olio-ohjelman luokan sisäiset muuttujat ovat siis myös julkisia funktioiden näkökulmasta katsottuna. (Tässä käytetään termiä funktio metodin sijaan korostamaan sitä, että proseduraalisen ohjelmoinnin funktio ja olio-ohjelmoinnin metodi eivät tältä osin eroa toisistaan merkittävällä tavalla.)

Ainoastaan parametreja funktioissaan käyttävän ohjelman etu on se, että funktioiden toiminta on täysin testattavissa funktioiden ulkopuolelta. Tämä johtuu siitä, että funktion ulkoisen rajapinnan kautta on mahdollista antaa kaikki se data, jota funktio toteutuksessaan käyttää. Mikäli ohjelmassa käytetään julkisia muuttujia, jokaisen funktion testattavuus kärsii riippumatta siitä, käyttääkö funktio niitä vai ei. Syy tähän on se, että funktion testaaja ei voi tietää, käyttääkö funktio julkisia muuttujia katsomalla vain sen rajapintaa.

Toinen etu parametreja käyttävässä funktiossa on uudelleenkäytettävyys. Ainoastaan parametreista riippuva funktio on täysin kopioitavissa eri ohjelmien välillä ilman lisätoimenpiteitä tai ymmärrystä siitä, miten funktio toimii. Julkisia muuttujia käyttävää funktiota ei voi huoletta kopioida toisiin ohjelmiin, koska funktion riippuvuus julkisista muuttujista on mahdollista selvittää vain tarkastelemalla funktion toteutusta. Yksittäisen funktion uudelleenkäyttö on siis mahdollista vain, mikäli ohjelmoija tietää, miten funktio toimii.

Pääasiassa julkisia muuttujia funktioissaan käyttävän ohjelman etu on se, että funktioiden rajapinnat säilyvät samoina tietorakenteiden muutoksista huolimatta. Tämä on etu etenkin suurissa ohjelmistoissa, joissa tietorakenteet eivät muutu ainoastaan ajan mukana vaan myös ohjelman sisällä, jolloin samaa rajapintaa voidaan käyttää käsittelemään useita eri tietorakenteita. Parametreja käyttävän funktion kutsun tulee taas olla erilainen jokaiselle eri tietorakenteelle, mikä voi aiheuttaa paljon ehtolauseita ohjelman sisällä ja muutoksia tietorakenteen muuttuessa. Jokainen tietorakenteen muutosta koskeva funktiokutsu täytyy muuttaa käsin, mikä saattaa aiheuttaa

huomattavaa lisätyötä, mikäli tietorakenteet muuttuvat etenkin ohjelmistokehityksen loppuvaiheessa. Jos esimerkiksi asiakastietoja tulostavaan funktioon ”tulosta(nimi, osoite)” halutaan lisätä puhelinnumero, täytyy jokainen funktiokutsu muokata tähän muotoon. Ongelman voi kiertää käyttämällä parametrina asiakastietuetta, mutta silloin funktion testattavuus ja uudelleenkäytettävyys kärsivät, koska asiakastietue sisältää todennäköisesti paljon muuttujia, joita funktio ei tarvitse lainkaan.

Proseduraalinen ja olio-ohjelmointi ottavat kaksi eri näkökulmaa parametrien ja julkisten muuttujien käyttöön. Puhtaasti proseduraalisessa ohjelmoinnissa pyritään suosimaan parametrillisia funktioita, koska julkisten muuttujien vaikutus saattaa ulottua laajalle, ja samalla funktiot säilyvät uudelleenkäytettävinä ja testattavina. Julkisten muuttujien käyttö proseduraalisessa ohjelmassa on ongelmallista, sillä muuttujien julkisuus käsittää koko ohjelman, jolloin jo muutama julkinen muuttuja tekee minkä tahansa ohjelman osan täydellisestä testaamisesta käytännössä mahdotonta.

Mikäli ohjelma esimerkiksi sisältää käyttäjää koskevat julkiset muuttujat *userid*, *role* ja *log,* testaajan täytyy olettaa, että näitä muuttujia käytetään jokaisessa ohjelman funktiossa, vaikka näin ei välttämättä ole. Vaikka julkisia muuttujia pyritäänkin välttämään, silloin tällöin muuttujia määritetään julkisiksi siksi, koska niitä käytetäänkin monissa eri paikoissa ja julkisina niitä on vaivattomampi käyttää, sillä niitä ei tarvitse syöttää aina parametreina. Yleensä tällaiset muuttujat liittyvät esimerkiksi lokitietojen kirjaukseen tai käyttöoikeuksien tarkistukseen. Olio-ohjelmointia kehittävässä aspektisuuntautuneessa ohjelmoinnissa tällaisia toimintoja kutsutaan poikkileikkauksellisiksi toiminnoiksi (cross-cutting concerns) (Sannikka 2005: 8). Julkisten muuttujien käyttö proseduraalisessa ohjelmoinnissa osoittaa, että aspektisuuntautuneen ohjelmoinnin ratkaisemat olio-ohjelmoinnin ongelmat koskettavat myös proseduraalista ohjelmointia.

Olio-ohjelmoinnissa lähes kaikki funktioiden käyttämä data on julkista kaikille samaa tietorakennetta käsitteleville funktioille. Olio-ohjelmoinnin ajatus on jakaa ohjelma tietorakenteensa mukaan pieniin luokkiin, jotka sisältävät sekä muuttujat että muuttujia käsittelevät funktiot. Näin funktioiden rajapinnat säilyvät muuttumattomina tietorakenteiden muutoksista huolimatta. Muuttujien julkisen luonteen vuoksi olio-ohjelman funktioita ei kuitenkaan ole mahdollista testata luotettavasti eikä käyttää uudelleen muissa ohjelmissa ilman muutoksia.

Toinen testattavuutta haittaava olio-ohjelmoinnin periaate on se, että vaikka muuttujat ovatkin julkisia niitä käsitteleville funktioille, ne usein piilotetaan luokan ulkopuolisilta tarkkailijoilta. Testaaja ei siis suoraan pääse kokeilemaan erilaisia muuttujien arvoja, vaan testauksen on tapahduttava luokan ulkopuolelle näkyvien rajapintojen kautta.

Ilkeämielinen ihminen voisi sanoa, että olio-ohjelmat ovat kasa pieniä huonosti suunniteltuja proseduraalisia ohjelmia, eikä hän olisikaan kovinkaan kaukana totuudesta (vrt. kuva 7). Käytännössä luokat nimittäin ovat pieniä proseduraalisia ohjelmia, joissa data on julkista, eli funktioiden ulkopuolella, mutta kaikkien funktioiden saatavilla. Olio-ohjelmoinnin tarkoitus on kuitenkin pitää luokat niin pieninä, etteivät testattavuuden ja uudelleenkäytettävyyden ongelmat muodostu liian suuriksi. Toisin kuin proseduraalisessa ohjelmoinnissa, olio-ohjelmoinnissa funktion ei ole tarkoituskaan olla itsenäinen yksikkö, vaan itsenäinen yksikkö on luokka.

```
global date;
global time;

function checkTime();
```

```
class DateTime
{
  private date;
  private time;

  public function checkTime();
}
```

Kuva 7: Huonosti suunniteltu proseduraalinen ohjelma ja hyvin suunniteltu olio-ohjelman luokka ovat hyvin samannäköisiä.

Sekä proseduraalisessa että olio-ohjelmoinnissa on siis omat hyvät puolensa. Siinä missä proseduraalinen ohjelmointi korostaa funktioiden itsenäisyyttä, testattavuutta ja uudelleenkäytettävyyttä, olio-ohjelmoinnin tarkoitus on antaa selkeät rajapinnat ohjelman tietorakenteen käytölle. Proseduraalisen ohjelmoinnin suurin puute onkin juuri se, että vaikka se mahdollistaa paljon vapauksia sekä tietorakenteen että suoritusrakenteen suunnittelussa, se ei ota lainkaan kantaa siihen, mikä on loppujen lopuksi funktioiden ja käsiteltävän datan välinen rajapinta.

3.1.1 Havainnoivan ohjelmoinnin tavoitteet

Havainnoivan ohjelmoinnin teoria pyrkii kuvaamaan ohjelmointiparadigman, joka kehittää proseduraalisen ohjelmoinnin näkökulmaa eteenpäin lisäämällä siihen funktioiden ja datan välisen rajapinnan. Paradigman tavoite on esitellä ohjelmiston

rakenne, jonka kaikki osat ovat testattavia ja uudelleenkäytettäviä, mutta jolla on silti selkeät ja monikäyttöiset rajapinnat.

Siinä missä proseduraalisen ohjelmoinnin toiminnallinen yksikkö on funktio ja olio-ohjelmoinnin toiminnallinen yksikkö on luokka, havainnoivan ohjelmoinnin toiminnallisia yksiköitä ovat parametrilliset funktiot sekä julkisia rajapintoja toteuttavat tunnistusmoduulit, jotka hallitsevat tietorakenteiden ja parametrillisten funktiokutsujen välistä kommunikaatiota. Moduulit tuntevat tietorakenteet ja funktioiden kutsut, mutta eivät itse toteuta funktioita tai muokkaa tietorakenteiden sisältöä.

Filosofisena pohjana havainnoiva ohjelmointi käyttää Immanuel Kantin kehittämää transsendentaalisen idealismin teoriaa. Transsendentaalinen idealismi on mahdollisesti nykyajan tunnetuin metafysiikkaa käsittelevä teoria, joka aloitti uuden aikakauden filosofian alalla löytämällä tuoreen näkökulman siihen, miten koemme ja ymmärrämme maailman. Tuo näkökulma ei ole puhtaasti objektiivinen eikä subjektiivinen vaan perustuu siihen, miten hankimme tietoa maailmasta aistiemme avulla ja käsittelemme sen ymmärryksessämme.

Havainnoiva ohjelmointi keskittyy Kantin ajatuksia mukaillen siihen, että funktioiden toteutukset eivät pääse tietorakenteiden sisältöön suoraan vaan rajapintojen kautta. Tällä tavalla saavutetaan ohjelmiston rakenne, jossa tietorakenteet ja funktiot ovat itsenäisiä ja riippumattomia toisistaan. Osat ovat täten vaihdettavissa toisiin, uudelleenkäytettävissä muissa projekteissa sekä täysin testattavissa ulkopuolelta käsin säilyttäen samalla selkeät rajapinnat.

Tämän ohjelmointiparadigman kaikki osat on jatkossa perusteltu tarkasti, mutta monet näistä osista ovat tuttuja jo muista paradigmoista. Ne erot, jotka havainnoivassa ohjelmoinnissa ovat proseduraaliseen ja olio-ohjelmointiin verrattuna, eivät ole suuria, mutta ne ovat sitäkin merkittävämpiä. Edes Kant ei lähtenyt kehittämään uutta teoriaansa puhtaalta pöydältä vaan toi uuden näkökulman jo olemassa oleviin metafyysisiin teorioihin. Samalla tavalla havainnoiva ohjelmointi pohjautuu myös vanhoihin ajatuksiin ja käytäntöihin tuoden uuden näkökulman siihen, miten olemme aiemmin tottuneet ohjelmia tekemään. Uutta paradigmaa esiteltäessä on kuitenkin syytä olla perusteellinen, sillä muutoksia on mahdollista perustella vain perustelemalla myös kaikki se, mikä ei muutu.

Havainnoivan ohjelman ydinrakenne jakautuu neljään osaan, joita kaikkia vastaa jokin transsendentaalisen idealismin käsite: rajapintoihin (aistit), funktioihin (ymmärrys), tunnistusmoduuleihin (skeemat) sekä muuttujista ja tietueista koostuviin tietorakenteisiin, eli objekteihin (ilmiöt). Jokainen näistä osista on esitelty seuraavaksi omissa luvuissaan.

3.2 Transsendentaalinen idealismi

Transsendentaalinen idealismi on Immanuel Kantin kehittämä metafysiikkaa käsittelevä teoria (van Inwagen 2014: 1), jonka hän esitteli ensimmäistä kertaa teoksessaan ”Puhtaan järjen kritiikki”. Teoria yhdistää subjektiivisen ja objektiivisen näkemyksen erottamalla maailman oliot ihmismielen käsityskyvyn ulkopuolelle kieltämättä kuitenkaan suoraan niiden olemassaoloa. Teoria jakaa maailman käsityskykymme ulkopuolisiin olioihin (noumena) sekä aistimaailmassamme havaittavissa oleviin ilmiöihin (phenomena) (Kant 2013: B295-B315).

Transsendentaalisen idealismin mukaan oliot eivät esiinny meille suoraan vaan ilmiöiden kautta, emmekä pysty koskaan tietämään millaisia oliot todellisuudessa ovat, jos niitä ylipäätään on olemassa. Ajatus vastaa siltä osin platonista realismia, että emme koe universaaleja ideoita puhtaina vaan epätäydellisten olioiden kautta. Platonisen realismin oliot ovat kuitenkin objektiivisesti todellisia, ja koemme ne juuri sellaisina kuin ne maailmassa esiintyvät. Transsendentaalisen idealismin ilmiöt ovat sen sijaan vain oman aistimellisuutemme rajaamia havaintoja varsinaisista olioista, joita emme voi koskaan aistia suoraan. Voimme esimerkiksi nähdä ikkunasta puun aivan eri tavalla kuin kärpänen sen näkee. Se, mitä näemme, ei ole puu itsessään vaan pelkästään ilmiö siitä. Platonisen realismin mukaan ihminen ja kärpänen näkevät puun samalla tavalla sellaisena kuin se maailmassa esiintyy, mutta kärpänen ei ymmärrä näkevänsä puuta puhtaan idean merkityksessä. Platoninen realismi sanoo, että ihminen voi päästä universaaleihin ideoihin järjen avulla, mutta transsendentaalisen idealismin mukaan tämä ei ole mahdollista (mistä tuleekin teoksen nimi ”Puhtaan järjen kritiikki”).

Toinen transsendentaalisen idealismin ajatus on kykymme käsitellä ilmiöistä saamaamme tietoa havaintoa edeltävän synteettisen tiedon avulla. Kantin mukaan tietomme havainnoista voi olla analyyttista tai synteettistä. Analyyttinen tieto tarkoittaa tietoa, joka on johdettavissa väittämästä itsestään (Kant 2013: B11). Esimerkiksi lause

”kuutiolla on kuusi sivua” on analyyttinen, koska sivujen määrä sisältyy valmiiksi kuution määritelmään. Tällaista havaintoa edeltävää tietoa kutsutaan analyyttiseksi a priori -tiedoksi.

Synteettinen tieto tarkoittaa tietoa, joka on aiempaa tietoa lisäävää. Esimerkiksi lause ”kappale on kuutio” antaa lisätietoa kappaleesta, mutta vaatii, että tarkastelemme kappaletta, jotta saamme tietää, onko lause totta. Tällaista havaintoa edellyttävää tietoa kutsutaan synteettiseksi a posteriori -tiedoksi. A priori tarkoittaa siis ”ennen kokemusta” ja a posteriori ”kokemuksen jälkeen”. (Kant 2013: B11.)

Kant oli ensimmäinen filosofi, joka väitti, että meillä voi olla synteettistä a priori -tietoa havainnoista. Kantin mukaan esimerkiksi matemaattinen yhtälö 7 + 5 = 12 vaatii synteettistä a priori -tietoa, koska luku 12 ei sisälly minkään sitä edeltävän käsitteen määritelmään (Kant 2013: B16). Synteettinen a priori -tieto on siis tietoa lisäävää mutta havainnosta riippumatonta (ts. havaintoa edeltävää) tietoa.

Puhtaan järjen kritiikki esittelee edellä mainittujen lisäksi useita erilaisia käsitteitä ja ajatuksia metafysiikasta, mutta merkittävimpiä niistä erityisesti ohjelmoinnin kannalta ovat ilmiöt, aistit, ymmärrys ja skematismi. Seuraavissa luvuissa esitellään nämä käsitteet yksitellen, sekä miten ne ovat yhteydessä ohjelmointiin.

3.2.1 Ilmiöt

Kant jakaa todellisuuden aistimaailmamme ja käsityskykymme saavutettavissa oleviin ilmiöihin (phenomena) sekä meille saavuttamattomissa oleviin olioihin sinänsä (noumena) (Kant 2013: B295-B315). Jako ilmiöihin ja olioihin ei sinänsä ole ainutlaatuista, mutta Kantin mukaan meillä ei voi olla minkäänlaista tietoa siitä, millaisia oliot sinänsä ovat, kun taas esimerkiksi platoninen realismin mukaan oliot sinänsä (ideat) ovat kohteita, jotka ovat saavutettavissa järjen avulla. Transsendentaalisen idealismin mukaan me tunnemme vain sen, miten oliot ilmenevät meille ilmiöiden kautta, mutta olioiden todellinen luonne jää meille tuntemattomaksi. Ilmiöt eivät välttämättä esiinny meille kaikille samanlaisina, vaan ilmiöiden luonne on riippuvainen sekä olioista sinänsä että havaitsijan kyvystä aistia ja ymmärtää ilmiöitä.

Havainnoivassa ohjelmoinnissa ilmiöitä kuvaavat objektit: tietorakenteet, jotka koostuvat tietueista, eli joukoista muuttujia. Muuttujat kuvaavat ilmiöiden

ominaisuuksia, joista teemme päätelmiä ymmärryksessämme. Ominaisuuksia voisivat olla esimerkiksi väri tai paino. Ne kuvaavat ohjelman objekteja niille annettujen arvojen avulla.

Tietueet kuvaavat ilmiöiden ominaisuusjoukkoja, jotka esiintyvät usein yhdessä tiettyjen havaintojen yhteydessä. Esimerkiksi väri ja kirkkaus ovat ominaisuuksia, jotka esiintyvät usein näköhavainnon yhteydessä. Muoto ja kovuus taas esiintyvät usein tuntohavainnon yhteydessä. Tietueet ovat siis havainnoivan ohjelmoinnin tapa koota yhdessä esiintyviä objektien ominaisuuksia yhteen.

Sen sijaan, että objektit kuvaisivat maailman olioita suoraan, ne pyrkivät transsendentaalisen idealismin tavoin kuvaamaan ilmiöitä niin kuin ne meille esiintyvät. Tässä lähestymistavassa on se ongelma, että ilmiöt eivät välttämättä esiinny meille kaikille samanlaisina. Havainnoivan ohjelmoinnin erityinen ehto onkin se, että vaikka eri havaitsijoilla saakin olla erilaisia käsityksiä ilmiöistä, ilmiöiden itsensä täytyy olla kaikille havaitsijoille samat. Ohjelmassa ei siis saa olla objekteja tai muuttujia, jotka esittävät samoja asioita kahdella eri tavalla tai joiden arvot vaikuttavat toistensa arvoihin.

Esimerkkinä tästä voisi olla suorakulmion pinta-ala. Joissain ohjelmissa suorakulmion pinta-ala saatetaan tallentaa tietokantaan sellaisenaan, kun taas toisissa ohjelmissa se saatetaan laskea suorakulmion sivujen pituuksien avulla. Mikäli ohjelma sisältää muuttujina sekä suorakulmion pinta-alan että sivujen pituudet, ne vaikuttavat suoraan toistensa arvoihin. Ohjelmassa saa kuitenkin olla vain yksi tapa kuvata mitä tahansa ilmiötä tai ominaisuutta, koska muussa tapauksessa ohjelman objektit saattavat joutua tilaan, jossa ne sisältävät ristiriitaista tietoa (esim. suorakulmio, jonka pinta-ala on 10 m^2 ja sivujen pituudet 5 m ja 7 m). Ohjelman suunnittelijan on tässä tapauksessa valittava suorakulmion ominaisuuksien esitykseen joko pinta-ala tai sivujen pituudet mutta ei molempia. Tällä rajoituksella pyritään välttämään ominaisuuksien keskinäistä riippuvuutta ja ristiriitaisen tiedon esiintymistä ohjelmassa.

Niin ikään Kantin transsendentaaliseen idealismiin perustuva, luvussa 2.3.3 käsitelty subjektisuuntautunut ohjelmointi ottaa erilaisen näkökulman, jonka mukaan jokaisen subjektin havaitsemat ilmiöt kuvataan omissa luokissaan erillään muiden subjektien havaitsemista ilmiöistä (Harrison 1993: 3). Tällöin ilmiöt ovat välttämättä erilaiset jokaiselle subjektille. Tällä pyritään lisäämään joustavuutta ohjelmistokehityksessä,

koska ohjelmistosuunnittelijoiden ei tarvitse päättää yhdestä objektiivisesta tietorakenteesta, joka olisi kaikille ohjelman subjekteille sama (Harrison 1993: 2-3).

Subjektisuuntautuneen lähestymistavan suurin ongelma on juuri se, että koska ilmiöluokkien muuttujat eivät kuvaa ilmiöitä sellaisina kuin ne esiintyvät kaikille vaan ainoastaan yhden subjektin näkökulmasta, eri ilmiöiden muuttujat saattavat vaikuttaa toistensa arvoihin, jolloin subjektien välille syntyy riippuvuuksien verkko. Esimerkiksi kun metsuri kaataa puun, tieto puun kaatumisesta täytyy välittää kaikille eri subjekteille, joita se erityisesti koskettaa (Harrison 1993: 4). Koska riippuvuuksien hallintaa ei ole mahdollista automatisoida, ilmiöiden välisten riippuvuuksien määrä ja monimutkaisuus vaikuttavat suoraan riippuvuuksien verkon monimutkaisuuteen. Tätä ongelmaa ei ole havainnoivassa ohjelmoinnissa, koska se sisältää ehdon, että ilmiöiden täytyy olla kaikille havaitsijoille samat.

3.2.2 Aistit

Aistit ovat kykymme vastaanottaa ilmiöiden meille välittämää tietoa. Transsendentaalisen idealismin mukaan havaintomme ilmiöistä riippuvat siitä, miten pystymme aistimaan niitä, eli vastaanottamaan ilmiöiden lähettämiä intuitioita. Tätä varten ilmiöllä tulee olla affekti, eli kyky lähettää intuitioita, ja meillä tulee olla tuota affektia vastaava aistimellisuus, eli kyky vastaanottaa intuitioita. (Kant 2013: B33.)

Aistimellisuus on siis tapamme saada tietoa ulkoisesta maailmasta, mutta se on myös tapamme saada tietoa itsestämme. Aistimme itseämme ilmiöiden kautta samalla tavalla kuin aistimme ulkoista maailmaa, ja tietoisuutemme omasta tilastamme on aistiemme rajoittama. Emme siis ole itsetietoisia, vaan saamme tietoa itsestämme intuitioiden kautta samalla tavalla kuin saamme tietoa ulkoisesta maailmasta.

Intuitiot ovat ilmiöiden representaatioita, jotka ovat olemassa vain aistimellisuudessamme. Mikäli poistamme aistimellisuutemme tai muutamme sen luonnetta, sitä vastaavat intuitiot lakkaavat olemasta. Voimme saada ilmiöistä useita erilaisia intuitioita, mutta ilmiöiden todellinen luonne (olio sinänsä) jää meille aina tuntemattomaksi. (Kant 2013: A42.)

Tämä tarkoittaa sitä, että intuitiot eivät ole olion tai ilmiön ominaisuuksia, vaan ne ovat täysin havainnoijasta riippuvaisia. Voimme esimerkiksi saada silmillämme valon

välityksellä intuition puusta, mutta jos suljemme silmämme, intuitiota ei enää ole. Mikäli käsitämme intuitiot funktioiden parametreina, tämä tarkoittaa sitä, että funktioiden parametrit ovat nimenomaan objekteista riippumattomien rajapintojen eivätkä objektien määrittämiä, vaikka parametrien arvot tulevatkin objekteilta.

Aistimme perustuvat kahteen intuitioon, jotka ovat olemassa a priori, eli ennen kaikkea kokemusta. Nämä ovat aika ja avaruus. Kantin mukaan aika ja avaruus eivät ole ulkoisia olioiden ominaisuuksia vaan aistimellisuutemme muoto ja edellytys sille, että voimme ylipäätään aistia mitään. (Kant 2013: A33-A36.) Meillä on olemassa vain yksi avaruus ja yksi aika, ja kaikki pienemmät avaruudet ja lyhyemmät ajat ovat vain osia niistä (Kant 2013: A25, A189).

Havainnoivassa ohjelmoinnissa aikaa kuvaa ohjelman suorituksen kulku ja avaruutta muistiavaruus. Samoin kuin aika ja avaruus ovat Kantin mukaan ennakkoehtoja sille, että voimme ylipäätään aistia mitään, ohjelman suorituksen kulku ja muistiavaruus ovat ennakkoehtoja sille, että ohjelma voi ylipäätään tehdä mitään.

Aisteja vastaavat havainnoivassa ohjelmoinnissa funktioiden ja tietorakenteiden väliset rajapinnat. Proseduraalisessa ohjelmoinnissa nämä rajapinnat ovat samat kuin funktioiden rajapinnat, mikä on proseduraalisen ohjelmoinnin ehkä suurin kompastuskivi. Tämä tarkoittaa nimittäin sitä, että yhdellä toiminnon tyypillä (esim. piirtäminen) ei ole yhteistä rajapintaa useille eri tietorakenteille, vaan rajapinta riippuu sekä toiminnon tyypistä että tietorakenteen tyypistä. Funktion kutsujan täytyy siis tietää sekä haluamansa toiminnon että tietorakenteen tyyppi.

Rajapinnat kuvaavat käytännössä kykyjä havainnoida sovelluksen objekteja. Esimerkiksi rajapinta ”piirra(Kissa)” kertoo, että ohjelmassa voidaan piirtää Kissa-luokan objekti, mutta se ei kerro sen enempää, miten piirto käytännössä tehdään. Rajapinnoilla ei itsellään ole pääsyä objektien tietorakenteeseen, eivätkä ne myöskään tunne funktioiden toteutuksia. Ne tietävät ainoastaan sen, mitä toimintoja ohjelmassa on ja minkälaisille objekteille nämä toiminnot voidaan suorittaa.

3.2.3 Ymmärrys

Siinä missä aistit ovat kykymme vastaanottaa intuitioita ilmiöistä, ymmärrys on kykymme ajatella ilmiöitä. Yksinään sekä aistit että ymmärrys ovat merkityksettömiä,

mutta yhdessä nämä kyvyt muodostavat kognition (ja myös subjektin siinä mielessä, kun käsitämme subjektin aistivana ja ajattelevana oliona). (Kant 2013: B75-B76.)

Ajattelu tapahtuu ymmärryksessämme käsitteiden avulla. Muodostamme käsitteitä jatkuvasti aistiemme kautta saamistamme intuitioista ja pyrimme soveltamaan aiemmin muodostamiamme käsitteitä uusiin intuitioihin. Käsitteet ovat aina yleisiä ja niillä voidaan viitata useisiin eri kohteisiin.

Samalla tavalla kuin aistimme perustuivat aikaan ja avaruuteen, eli a priori -intuitioihin, myös ymmärryksemme perustuu käsitteisiin, jotka ovat meillä a priori, eli ennen mitään kokemusta. Näitä käsitteitä Kant kutsuu puhtaiksi ymmärryksen käsitteiksi, eli kategorioiksi (kuva 8). Me emme sovella kategorioita suoraan ilmiöihin, vaan aivan kuten aika ja avaruus ovat ennakkoehtoja aisteillemme, kategoriat ovat ennakkoehtoja sille, että voimme ylipäätään ymmärtää mitään. (Kant 2013: A96.)

Kategorioita on neljä: kvantiteetti, kvaliteetti, relaatio ja modaalisuus. Jokaisella kategorialla on myös kolme johdettua käsitettä. Kategorioilla on vahva suhde puhtaisiin intuitioihin, eli aikaan ja avaruuteen. (Kant 2013: B104-B109.)

	Kvantiteetti	
	Ykseys	
	Moneus	
	Kaikkeus	
Kvaliteetti		**Relaatio**
Reaalisuus		Inherenssi ja subsistenssi
Negaatio		Kausaliteetti ja riippuvuus
Limitaatio		Yhteisyys
	Modaalisuus	
	Mahdollisuus	
	Olemassaolo	
	Välttämättömyys	

Kuva 8: Kategorioiden taulukko (Kant 2013: B104-B109).

Havainnoivassa ohjelmoinnissa ymmärrystä vastaa kyky toteuttaa funktioita. Se on ohjelman kyky käsitellä muuttujia lauseiden avulla. Mikäli kuvaamme aistin rajapintana, eli kykynä määritellä funktiokutsun muoto ja sen tarjoamat parametrit, ymmärrys voidaan kuvata rajapinnan toteutuksena, eli kykynä muodostaa lauseita parametreina saaduille muuttujille. Havainnoivassa ohjelmoinnissa näitä toteutuksia kutsutaan toteutusfunktioiksi.

Mikäli tarkastelemme Kantin kategorioiden taulukkoa, huomaamme siellä selkeitä yhteyksiä ohjelmointiin. Kantin tekemä kategorioiden jako määrään, laatuun, suhteeseen ja todennäköisyyteen voidaan käsittää ohjelmoinnin termein lukuihin, muuttujien tyyppeihin, muuttujien keskinäisiin suhteisiin sekä ehtolauseisiin. Samalla tavalla kuin me muodostamme Kantin mukaan päätelmiä kategorioiden avulla, myös ohjelmat muodostavat lauseita näitä kategorioita mukaillen.

Erityinen piirre ohjelmoinnissa on kuitenkin se, että kategorioiden lisäksi myös funktiot ovat olemassa a priori, eli ennen ohjelman suoritusta. Tämä tarkoittaa sitä, että aivan kuten emme sovella kategorioita suoraan ilmiöihin, me emme sovella toteutusfunktioitakaan suoraan objekteihin, vaan teemme sen nk. skematismin, eli tunnistusmoduulien avulla.

3.2.4 Arvostelmakyky ja skematismi

Me emme käsittele intuitioita suoraan ymmärryksessämme, vaan meillä tulee olla kyky yhdistää ymmärryksessä käytetyt käsitteet intuitioihin. Tätä kykyä Kant kutsuu arvostelmakyvyksi. (Kant 2013: B172-B173.) Kun näemme puun, aistimme sen tiettynä joukkona valon aallonpituuksia, jonka arvostelmakykymme yhdistää käsitteeseen ”puu”. Käytämme tätä käsitettä ymmärryksessämme, jolla on kyky tehdä päätelmiä tuon käsitteen avulla. Ymmärrys on siis kykymme tehdä päätelmiä, mutta vain arvostelmakyvyn avulla voimme kohdistaa päätelmät johonkin aistien havaitsemaan kohteeseen. Arvostelmakyky on siis tapa tunnistaa ilmiöitä ennen niiden käsittelyä ymmärryksessä.

Arvostelmakykymme siis kertoo ymmärryksellemme sen, miten intuitiot jakautuvat käsitteiden alle. Kokemustemme kautta opittuja a posteriori -käsitteitä voimme Kantin mukaan soveltaa aistiemme kohteisiin suoraan, mutta puhtaat a priori -käsitteet ovat liian erilaisia aistihavaintoihin nähden, jotta voisimme soveltaa niitä suoraan aistien

kohteisiin. Sen vuoksi meillä tulee olla sääntöjä, joiden avulla yhdistämme a priori -käsitteet aistihavaintoihin. Näitä sääntöjä Kant kutsuu skeemoiksi. (Kant 2013: B177.)

Havainnoivassa ohjelmoinnissa skeemoja vastaavat tunnistusmoduulit, jotka ovat ohjelman rajapintojen ja toteutusfunktioiden välissä. Toteutusfunktiot vastaavat siis a priori -käsitteitä, eli sellaista tietoa ohjelman tietosisällöstä, joka on olemassa ennen ohjelman suoritusta. Tunnistusmoduulit toteuttavat ohjelman aistin, eli rajapinnan, kutsumalla toteutusfunktioita. Aivan kuten skeematkaan eivät omista käsitteitä, moduulit eivät myöskään itse suoraan toteuta rajapintojen määrittämiä toimintoja vaan ainoastaan kutsuvat varsinaisia toteutusfunktioita. Rajapinnat siis toimivat tiedon vastaanottajina ja tunnistusmoduulit tiedon välittäjinä.

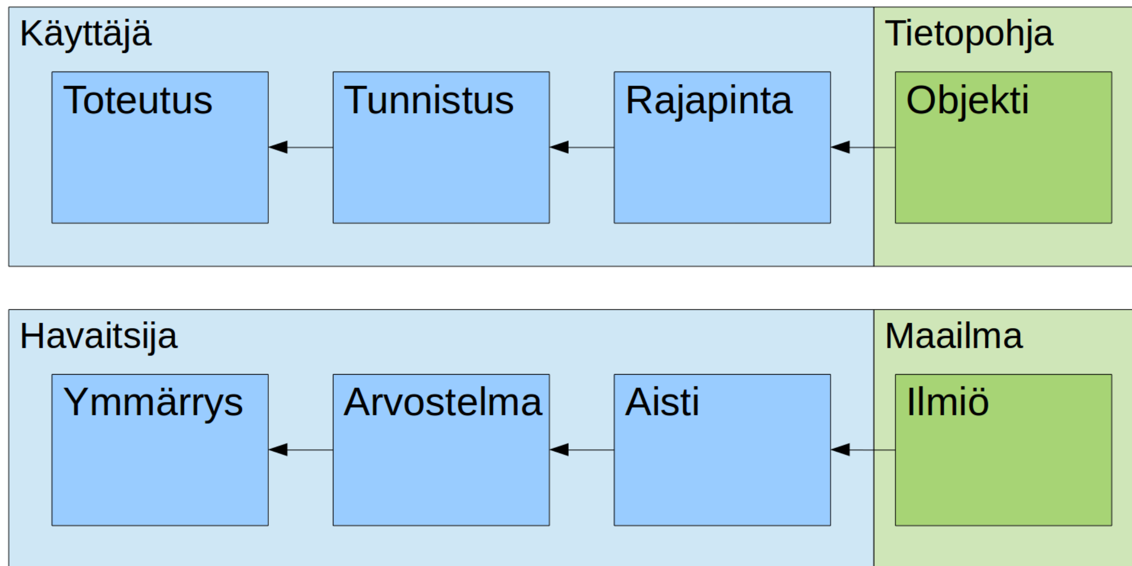
Tunnistusmoduulit ovat keskeisin osa havainnoivaa ohjelmointia, mutta yksikään ohjelmointikieli ei pysty toteuttamaan niitä täydellisesti. Ideaalissa tilanteessa moduuli pystyy vapaasti välittämään objektien ominaisuuksia eteenpäin, mutta ei pääse muokkaamaan niitä itse. Esimerkiksi C++:lla toteutettuna moduuliluokalle on mahdollista antaa friend class -ominaisuuden avulla pääsy sekä tietorakenteisiin että funktioihin. Moduulia ei kuitenkaan pysty estämään käsittelemästä dataa, vaan toiminta perustuu luottamukseen siitä, että data kulkee moduulin läpi käsittelemättömänä. Käsittelemättömyydellä tarkoitetaan tässä sitä, ettei dataa muokata, tulosteta tai tallenneta pysyvästi millään tavalla.

3.2.5 Transsendentaalisen idealismin yhteys havainnoivaan ohjelmointiin

Vaikka rajapinnat, toteutusfunktiot ja tunnistusmoduulit ovatkin toisistaan erillisiä osia, ne on mahdollista käsittää yhdessä osina ohjelman käyttäjää. Käyttäjällä pyritään kuvaamaan kirjaimellisesti ohjelman loppukäyttäjää, joka on ohjelman tietosisällön havaitsija. Objektit taas voidaan käsittää osina ohjelman tietopohjaa. Kuva 9 havainnollistaa sitä, miten transsendentaalisen idealismin käsitteet ja havainnoivan ohjelmoinnin osat vastaavat toisiaan.

Käyttäjä kuvaa transsendentaalisen idealismin subjektia, eli havaitsijaa, joka kokee maailman aistien ja ymmärryksen kautta. Tietopohja taas kuvaa transsendentaalisen idealismin maailmaa, joka koostuu havaitsijan aistien kohteina olevista ilmiöistä. Yhdessä havaitsija ja maailma muodostavat todellisuuden sellaisena, kuin se meille

näyttäytyy. Niin ikään yhdessä käyttäjä ja tietopohja muodostavat ohjelman sellaisena, kuin se meille näyttäytyy.



Kuva 9: Transsendentaalisen idealismin vastaavuus havainnoivan ohjelmoinnin kanssa.

4 Havainnoivan ohjelman rakenne

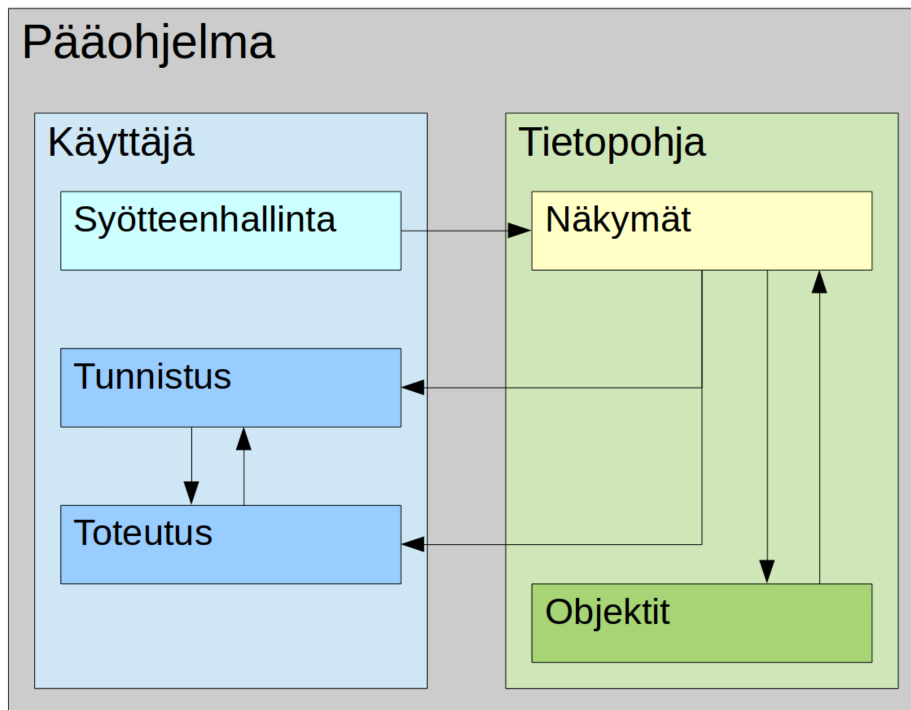
Objektit, rajapinnat, funktiot ja moduulit kuvaavat havainnoivan ohjelmoinnin ydinrakenteen, joka määrittelee sen, miten tietoa käsitellään. Se onkin ohjelmointiparadigman päätehtävä. Tiedonkäsittely on kuitenkin vain yksi osa toimivaa ohjelmaa, ja se vaatii lisäksi ympärilleen malleja, jotka määrittelevät ohjelman muut osat, kuten syötteenkäsittelyn, tietokantayhteyden ja näkymien hallinnan.

Seuraavissa luvuissa on esitelty esimerkki havainnoivaa ohjelmointia hyödyntävän ohjelman rakenteesta kokonaisuudessaan. Rakenne kuvataan ensin yleisesti pääohjelmasta käsin, minkä jälkeen jokainen ohjelman osa käydään läpi tarkemmin ohjelman rakenteen mukaisessa järjestyksessä. Transsendentaalinen idealismi ei ota kantaa kaikkiin näiden osien käsittelemiin asioihin, joten osat on pyritty havainnollistamaan ohjelmoinnin käsitteiden lisäksi käsitteillä, jotka kuvaavat reaalimaailmaa kokemallamme tavalla.

4.1 Pääohjelma

Pääohjelma on nimensä mukaisesti se osa ohjelmaa, joka määrittelee yleisimmällä tasolla koko ohjelman rakenteen. Kuvassa 10 esitetty pääohjelma koostuu kuvan 9 tavoin käyttäjästä sekä tietopohjasta. Yksinkertaistetusti voidaan sanoa, että käyttäjä määrittelee ohjelman suoritusrakenteen ja tietopohja tietorakenteen. Pääohjelmassa voi olla useita käyttäjiä ja tietopohjia, mutta tämän luvun esimerkeissä niitä molempia on vain yksi.

Pääohjelman tehtävä on luoda käyttäjä ja tietopohja ja aloittaa ohjelman suoritus kutsumalla käyttäjän määrittelemän syötteenhallinnan suoritusfunktiota. Pääohjelman täytyy myös tarjota käyttäjälle jonkinlainen rajapinta, jonka avulla käyttäjän on mahdollista valita oikea näkymä annettujen syötteiden perusteella.

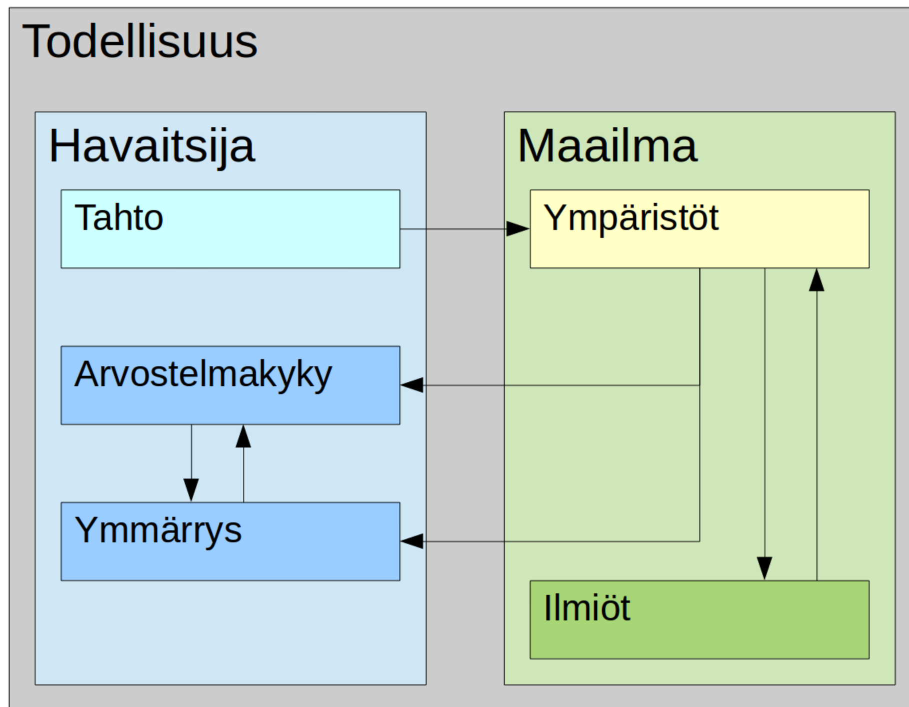


Kuva 10: Pääohjelma koostuu käyttäjästä ja tietopohjasta, joista käyttäjä määrittelee ohjelman syötteenhallinnan ja toimintojen toteutukset ja tietopohja ohjelman tietorakenteet ja näkymät.

Käyttäjä sisältää aiemmin esitettyjen tunnistusmoduulien ja toteutusfunktioiden lisäksi ohjelman syötteenhallinnan. (Rajapinnat on tässä esityksessä toteutettu suoraan tunnistusmoduuleissa rakenteen yksinkertaistamiseksi.) Ohjelman suoritus alkaa aina syötteenhallinnasta, jonka tehtävä on vastaanottaa syötteet, käsitellä ne muun ohjelman ymmärtämään muotoon ja välittää ne eteenpäin näkymille.

Tietopohja sisältää ohjelman objektien ja muuttujien lisäksi kaikki ohjelman näkymät. Näkymien tehtävä on määritellä näkymässä suoritettavat toiminnot sekä välittää objektit ja muuttujat käyttäjän tunnistusmoduuleille ja toteutusfunktioille.

Jos pääohjelmaa pitäisi kuvata yhdellä reaalimaailmaa kuvaavalla käsitteellä, se olisi todellisuus (kuva 11). Todellisuus sisältää kaiken sen, mikä on meidän aistiemme ja ymmärryksemme kohteena. Se pyrkii kuvaamaan maailmaa sellaisena kuin me sen koemme ottamatta kantaa siihen, millainen maailma todellisuudessa on. Todellisuutta voidaan kutsua myös tarkemmin havaitsijan todellisuudeksi, mikäli halutaan korostaa havaitsijan merkitystä todellisuuden määrittelijänä.



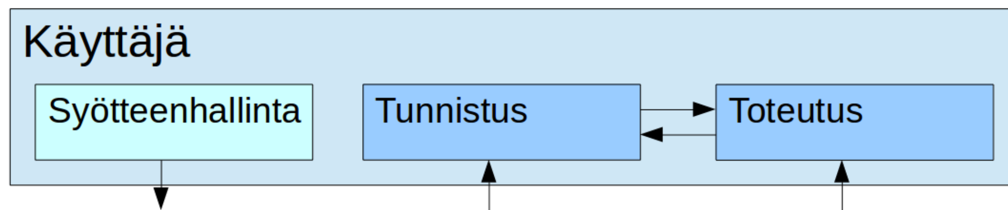
Kuva 11: Todellisuus koostuu havaitsijasta ja havaitsijan ulkopuolisesta maailmasta.

Havaitsijan todellisuus sisältää havaitsijan sekä maailman. Havaitsija sisältää nk. vapaan tahdon, eli tavoitteellisuuden sekä kyvyn tunnistaa ja ymmärtää maailman ilmiöitä. Maailma taas sisältää kaikki havaitsijan havaittavissa olevat ilmiöt sekä ympäristöt, jotka välittävät ilmiöt havaitsijalle.

Havaitsijan todellisuus ei ole sama asia kuin olemassa oleva todellisuus, koska se ei sisällä kirjaimellisesti kaikkea mahdollista olemassa olevaa vaan ainoastaan aistien ja ymmärryksen kohteita. Se ei myöskään ole sama asia kuin havaittavissa oleva todellisuus, koska se ei sisällä kaikkea mahdollista havaittavissa olevaa vaan ainoastaan havaitsijan aistien kohteita. Se on siis havaitsijan kokema todellisuus.

4.2 Käyttäjä

Käyttäjä on ohjelman päätoimija, joka määrittelee kuvan 12 mukaisesti ohjelman syötteenhallinnan, rajapinnat, tunnistusmoduulit ja toteutusfunktiot. Syötteenhallinta määrittelee, miten syötteet vastaanotetaan ja käsitellään. Rajapinnat määrittelevät, mitä toimintoja ohjelmassa voi suorittaa. Tunnistusmoduulit määrittelevät, miten objektit tunnistetaan eri toimintojen yhteydessä ja toteutusfunktiot määrittelevät, miten nämä toiminnot toteutetaan.

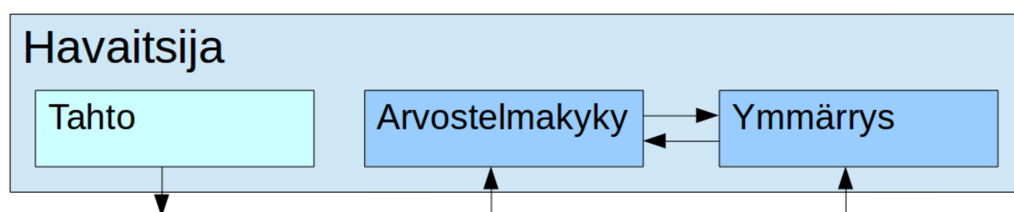


Kuva 12: Käyttäjän vastuulla on ohjelman syötteenhallinta sekä toimintojen tunnistus-moduulit ja toteutukset.

Käyttäjä pyrkii kuvaamaan kirjaimellisesti ohjelman loppukäyttäjää. Käyttäjä onkin ohjelman keskeisin osa, sillä ilman sitä ohjelmaa ei ole lainkaan. Yksinkertaisimmat ohjelmat saattavat toimia ilman erillistä tietorakennetta, mutta ilman käyttäjän tarjoamia funktioita ohjelma ei periaatteessa voi tehdä mitään. Vain kaikista yksinkertaisimmat ohjelmat voivat suorittaa kaikki toimintonsa suoraan pääohjelmassa, mutta pääohjelmassakin suoritetut toiminnot kuvaavat periaatteellisella tasolla käyttäjää silloinkin, kun erillistä luokkaa käyttäjälle ei ole lainkaan.

Syötteenhallinnan lisäksi käyttäjä siis määrittelee myös sen, miten objekteja tunnistetaan ja miten funktiot toteutetaan. Pääohjelma voi vaihtaa käyttäjän toiseen, jolloin ohjelman suoritus voi muuttua täysin. Tällä tavalla on mahdollista hallita esimerkiksi erilaisia käyttöliittymiä mobiililaitteille ja tietokoneille tai täysin erilaisia toimintokokonaisuuksia eri työntekijäryhmille. Myös yksittäisiä toteutuksia on mahdollista vaihtaa saman käyttäjän sisällä jopa suorituksen aikana, mikä mahdollistaa hyvin joustavan, jopa oppivan suoritusrakenteen ohjelman sisällä.

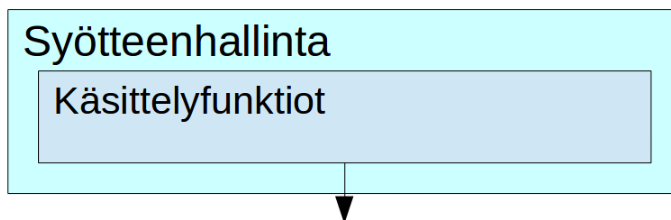
Reaalimaailman käsitteillä käyttäjää vastaa havaitsija, joka on tahdon, aistien ja ymmärryksen käyttäjä (kuva 13). Havaitsija määrittelee tavoitteet aisteilleen tahtonsa avulla, minkä jälkeen hän tunnistaa aisteilla saadut havainnot arvostelmakyvyllään ja käsittelee ne ymmärryksessään.



Kuva 13: Havaitsija on tahdon, aistien ja ymmärryksen käyttäjä.

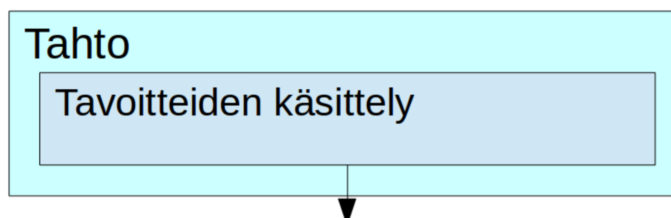
4.2.1 Syötteenhallinta

Ohjelman syötteenhallinta määrittelee sen, miten ohjelman ulkopuolelta tulevat syötteet vastaanotetaan ja esikäsitellään. Pääohjelma aloittaa ohjelman suorituksen aina syötteenhallinnasta. Syötteenhallinnan tehtävä on vastaanottaa ohjelman käyttäjän tai toisten ulkoisten ohjelmien antamat syötteet, käsitellä ne ja välittää ne eteenpäin halutulle näkymälle. Syötteenhallinta ei suorita toimintoja syötteiden avulla, mutta se sisältää kaikki tarvittavat käsittelyfunktiot, jotka muokkaavat syötteet muun ohjelman ymmärtämään muotoon (kuva 14).



Kuva 14: Syötteenhallinta koostuu ainoastaan käsittely-funktioista.

Reaalimaailman käsitteillä syötteenhallintaa vastaa havaitsijan tahto ja syötteitä havaitsijan tavoitteet (kuva 15). Tahto antaa meille mahdollisuuden asettaa tavoitteita, jotka vaikuttavat siihen, mihin ympäristöön haluamme seuraavaksi mennä ja mitä haluamme siellä ympäristössä tehdä. Ympäristöjä voivat olla esimerkiksi koti, työpaikka tai puisto. Ympäristöt tarjoavat meille erilaisia mahdollisuuksia, joita haluamme tavoitella sekä kohteita, jotka tunnistamme ja joita osaamme hyödyntää tavoitteidemme saavuttamiseksi.

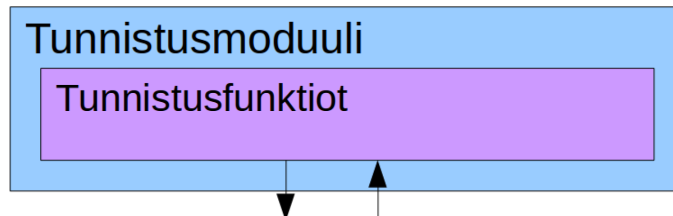


Kuva 15: Tahto käsittelee havaitsijan tavoitteita.

4.2.2 Tunnistusmoduulit

Tunnistusmoduulit ovat ohjelman tapa tunnistaa ja erotella ohjelman tietorakenteita eri toimintojen yhteydessä. Tunnistusmoduulit ovat luokkia, joiden vastuulla on oikeiden toteutusten kutsu kunkin toiminnon yhteydessä. Moduulit koostuvat ainoastaan

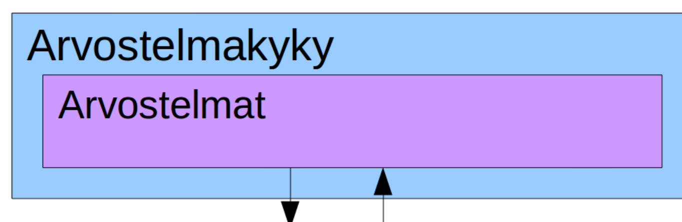
tunnistusfunktioista (kuva 16), mutta ohjelman toteutuksesta riippuen myös toteutusfunktioita sisältävät luokat on mahdollista sisällyttää tunnistusmoduuleihin. Tunnistusmoduuleilla on pääsy sekä objekteihin että toteutusfunktioihin, minkä avulla moduulit voivat hakea tarvittavat muuttujat ja tietueet objekteista ja välittää ne objekteihin ja toimintoon soveltuvalle toteutusfunktiolle.



Kuva 16: Tunnistusmoduulit koostuvat ainoastaan julkisista tunnistusfunktioista.

Tunnistusmoduulien funktiot eivät saa itse käsitellä saamiaan objekteja tai muuttujia, vaan niiden tarkoitus on toimia ainoastaan tiedon välittäjinä. Niitä voisi verrata olio-ohjelmoinnin gettereihin: ne ovat tapa päästä objektien ominaisuuksiin jonkin rajapinnan kautta. Toisin kuin getterit, tunnistusmoduulien funktiot liittyvät aina johonkin tiettyyn toimintoon, esimerkiksi tulostamiseen tai muuttujien muokkaukseen.

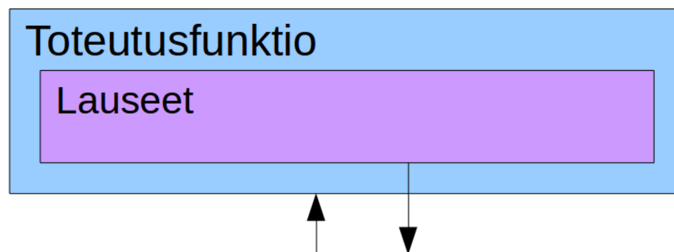
Reaalimaailmassa tunnistusmoduuleita vastaa havaitsijan arvostelmakyky, eli se osa havaitsijan mieltä, joka tunnistaa ilmiöitä ja erottelee ne toisistaan (kuva 17). Ennen kuin havaitsija voi tehdä päätelmiä aistimistaan ilmiöistä, hänen täytyy tunnistaa ilmiöt ja luokitella ne. Vasta luokittelun jälkeen mielekkäiden päätelmien teko on mahdollista. Jos esimerkiksi näemme autoa ajaessamme kaukana tiellä jotain, mutta emme ole varmoja, onko se eläin, jätesäkki vai kuoppa tiessä, emme pysty tekemään tarvittavia päätelmiä, jotta tietäisimme, mitä tehdä seuraavaksi.



Kuva 17: Havaitsijan mielessä ilmiöiden tunnistus tapahtuu arvostelmien avulla.

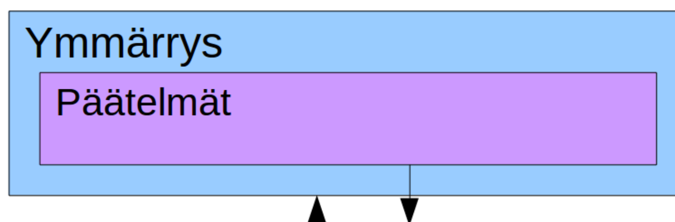
4.2.3 Toteutusfunktiot

Toteutusfunktiot ovat nimensä mukaisesti funktioita, joiden tarkoitus on toteuttaa näkymän tarjoamia ja käyttäjän valitsemia toimintoja lauseiden avulla (kuva 18). Toteutusfunktioilla ei ole pääsyä objektien sisäisiin muuttujiin tai tietueisiin, vaan funktiot saavat tarvitsemansa muuttujat ja tietueet tunnistusmoduulien kautta. Mikäli funktion saamissa muuttujissa on objekteja, funktio voi kutsua niille toimintoja tunnistusmoduulien kautta.



Kuva 18: Toteutusfunktiot koostuvat lauseista, jotka käsittelevät muuttujia.

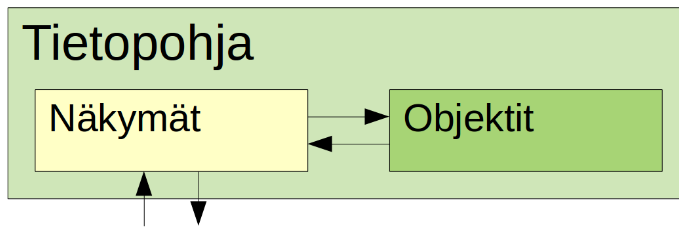
Reaalimaailmassa toteutusfunktioita vastaa havaitsijan ymmärrys, eli se osa havaitsijan mieltä, joka tekee päätelmiä havaitsemistaan ilmiöistä (kuva 19). Päätelmä voisi olla esimerkiksi se, että tiessä oleva kuoppa on syvä ja sitä pitää väistää, tai että autosta kuuluva ääni kuulostaa siltä, että auto tulee viedä huoltoon.



Kuva 19: Ymmärrys tekee päätelmiä ilmiöiden ominaisuuksista.

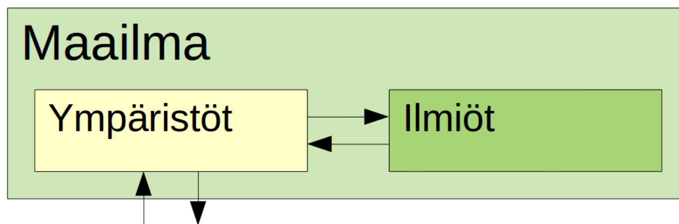
4.3 Tietopohja

Tietopohja sisältää kaikki ohjelman objektit ja näkymät. Objektit ja muuttujat ovat tietopohjan sisällä joko suoraan tai tietokantayhteyden kautta. Näkymien kautta käyttäjä voi suorittaa toimintoja objekteille (kuva 20).



Kuva 20: Tietopohja tarjoaa objektit ja ohjelman toiminnot käyttäjälle näkymien avulla.

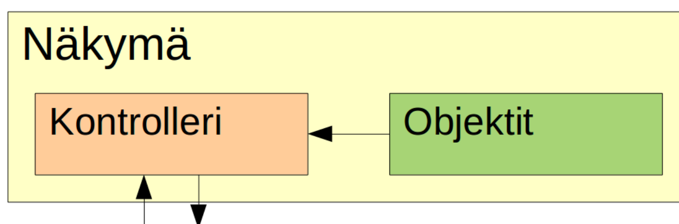
Reaalimaailmassa tietopohjaa vastaa ulkoinen maailma (kuva 21). Maailma kuvaa kaikkea sitä, mikä on olemassa havaitsijan ulkopuolella ja havaitsijan aistien ja ymmärryksen kohteena. Se sisältää kaikki ilmiöt, jotka voimme havaita sekä ilmiöiden yhteyden olioihin sinänsä, joita emme pysty aistein havaitsemaan tai ymmärtämään. Lisäksi maailma sisältää ympäristöt, jotka tarjoavat ilmiöt aistiemme havaittaviksi.



Kuva 21: Maailma tarjoaa ilmiöt havaitsijan aisteille ympä-ristöjen välityksellä.

4.3.1 Näkymät

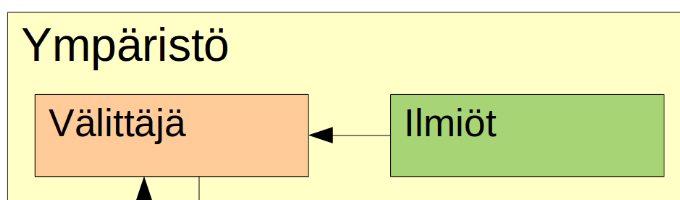
Näkymät ovat ohjelman tiloja, jotka sisältävät tilaan liittyviä objekteja sekä kontrollerifunktion, joka sisältää objektien käsittelyyn vaadittavien toimintojen kutsut (kuva 22). Näkymät kuvaavat kirjaimellisesti niitä näkymiä, jotka käyttäjä tiettynä aikana näkee. Esimerkkeinä näkymistä voisivat olla esimerkiksi web-sivuston etusivu, tilaussivu, pelin päävalikko tai pelinäkymä. Näkymä saa käsittelemänsä objektit tietopohjan kautta.



Kuva 22: Näkymät kutsuvat objekteihin kohdistuvia toimin-toja kontrollerifunktion sisällä.

Kontrolleri on funktio, jonka tehtävänä on määritellä, mitä toimintoja näkymässä voi suorittaa, mille objekteille niitä suoritetaan ja mitä ulkopuolelta tulevilla syötteillä tehdään. Kontrollerilla on pääsy kaikkiin näkymän objekteihin, ja tarvittaessa se voi hakea lisää objekteja tietopohjan rajapintojen kautta. Kontrolleri käyttää toimintojen suoritukseen käyttäjän tunnistusmoduuleita ja toteutusfunktioita.

Reaalimaailmassa näkymiä kuvaavat ympäristöt, jotka ovat ulkoisesta maailmasta koostuvia alueita, joiden sisällä toimimme. Ympäristöjä voisi olla esimerkiksi koti, työpaikka tai puisto. Ympäristöt koostuvat maailman ilmiöistä sekä välittäjästä, joka välittää ilmiöt aistiemme havaittaviksi (kuva 23).



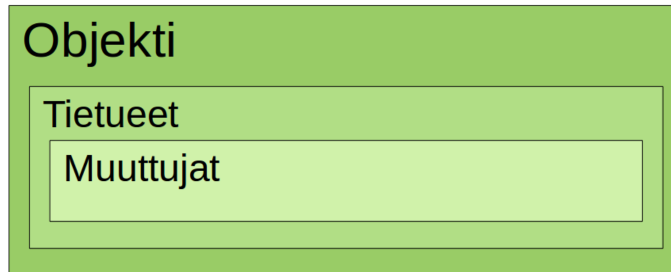
Kuva 23: Ympäristöt antavat ilmiöt havaitsijan aisteille välittäjän kautta.

Välittäjä määrittelee sen, mitä voimme ympäristössä tehdä ja mitä ilmiöitä havaita. Yksinkertainen esimerkki välittäjästä on valo. Vaikka meillä olisikin kyky nähdä, ilman valoa emme voi nähdä mitään. Valo siis välittää ilmiöt meille näköaistimme kautta. Välittäjä voi kuitenkin kuvata myös abstrakteja käsitteitä, kuten sosiaalisia suhteita ihmisten välillä. Sosiaaliset suhteet tarjoavat ihmisille mahdollisuuden toimia kunkin sosiaalisen suhteen määrittelemissä rajoissa. Esimerkiksi perhesuhteet mahdollistavat erilaisia kanssakäymisiä kuin ammatilliset suhteet.

4.3.2 Objektit

Objektit kuvaavat ohjelman tietosisältöä yhteen kokoavan tietorakenteen avulla. Niiden avulla on mahdollista koota yhteen toisiinsa liittyviä tietueita ja muuttujia. Objektit kuvaavat tietosisältöä aina suoraan tietueiden ja muuttujien avulla eivätkä sisällä lainkaan johdettua tietoa, eli funktioita (kuva 24). Objektien pääasiallinen tehtävä on tiedon jäsentäminen ohjelman toiminnan ja ymmärrettävyyden kannalta järkeviin tietorakenteisiin. Niiden ei kuitenkaan tarvitse mukailla ohjelman toimintaa orjallisesti, eikä tämä aina olekaan mahdollista, sillä ohjelma saattaa sisältää toiminnallisuuksia, jotka eivät vastaa yksittäisen tietorakenteen sisältöä. Esimerkiksi piirtofunktio saattaa

tarvita muuttujia piirrettävästä objektista, piirtävästä kamerasta ja ohjelman grafiikkamoottorista.



Kuva 24: Objektit koostuvat muuttujia sisältävistä tietueista.

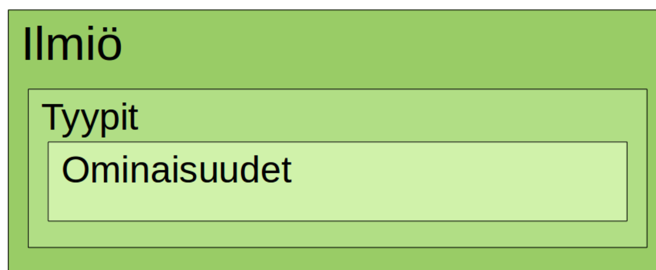
Objektit tulee määritellä siten, että ne muistuttavat ilmiöitä, jotka ohjelman käyttäjä tunnistaa. Tietueet taas tulee määritellä siten, että ne mukailevat ohjelmassa suoritettavia toimintoja. Tämä tarkoittaa sitä, että usein yhdessä käytettävät muuttujat yhdistetään samaan tietueeseen. Esimerkiksi koordinaatit x ja y voitaisiin yhdistää samaan sijaintitietueeseen. Tähän sijantitietueeseen ei kuitenkaan tule yhdistää koordinaatteihin liittyvää tekstitietoa, sillä vaikka nämä ominaisuudet liittyvätkin toisiinsa, niitä ei välttämättä käytetä samanaikaisesti.

Objektit muistuttavat tietorakenteeltaan hyvin paljon olio-ohjelmoinnin olioita, eli ne voivat periä toisia objekteja ja niiden ominaisuuksia. Objektit eivät kuitenkaan toteuta tai määrittele yhtäkään funktiota, vaan ne ovat ainoastaan tietorakenteita, joita objektien ulkopuoliset funktiot muokkaavat. Objektien on tarkoitus näyttää rakenteensa vain tunnistusmoduuleille eikä lainkaan toteutusfunktioille, joilla tulee olla pääsy objektien muuttujiin ainoastaan tunnistusmoduulien välityksellä.

Objektiivisen ohjelmoinnin, kuten esimerkiksi olio-ohjelmoinnin, periaatteen mukaan kaikki oliot ovat itse tietoisia itsestään. Havainnoivan ohjelmoinnin objektit eivät kuitenkaan ole itsetietosia. Todellisen maailman esimerkkejä tästä ovat elottomat kappaleet, kuten kivi tai tuoli. Nämä oliot eivät pysty havainnoimaan itseään millään tavalla. Kuitenkin myös elolliset oliot, kuten ihminen, ovat rajoittuneita minänsä havainnoinnista. Ihmiset tuntevat oman toimintansa pelkästään aistiensa ja ymmärryksensä kautta. Ihmisellä on esimerkiksi elintoimintoja, joiden sisäistä toimintaa hän ei pysty tuntemaan. Tällä tavalla elollisetkin oliot ovat jakautuneet objektiin ja subjektiin ja tarkkailevat itseään vain subjektin näkökulmasta. Havainnoiva ohjelmointi käyttää tätä jaottelua rakenteessaan, koska edellä mainittuun päätelmään vedoten

siihen perustuva tapa kuvata maailmaa on lähempänä todellisuutta kuin itsetietoinen tapa, ja pystyy siksi myös tarkemmin kuvaamaan maailmassa esiintyviä ongelmia.

Objektit vastaavat reaalimaailman ilmiöitä (kuva 25). Ilmiöt kuvaavat asioita, jotka havaitsemme ja tunnistamme aistiemme ja ymmärryksemme avulla. Ilmiöt sisältävät tyyppejä, jotka kuvastavat tietueiden tapaan ominaisuuksia ryhmiteltynä siten kuin ne esiintyvät yhdessä. Esimerkiksi äänen korkeus ja voimakkuus ovat ominaisuuksia, jotka molemmat esiintyvät aina kuulohavainnon yhteydessä.



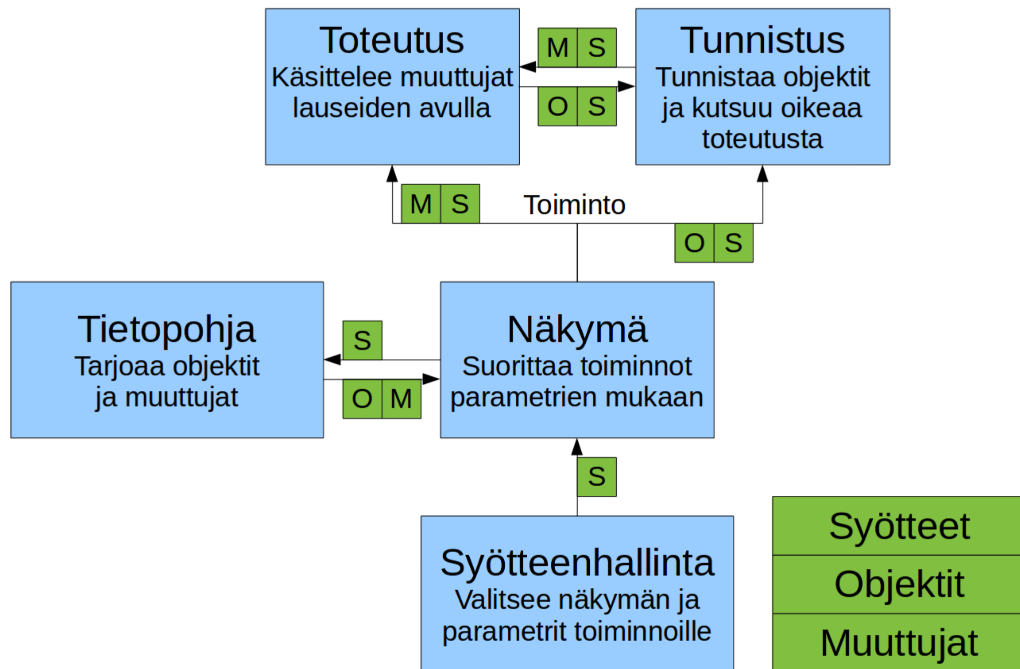
Kuva 25: Ilmiöt koostuvat ominaisuuksia sisältävistä tyypeistä.

4.4 Ohjelman suoritus

Ohjelman suoritus on kuvattu kokonaisuudessaan kuvassa 26. Suoritus alkaa aina syötteenhallinnasta. Syötteenhallinnan tehtävä on kerätä kaikki käyttäjän antamat syötteet, valita syötteiden perusteella haluttu näkymä ja välittää näkymään liittyvät lisäsyötteet näkymän kontrollerille.

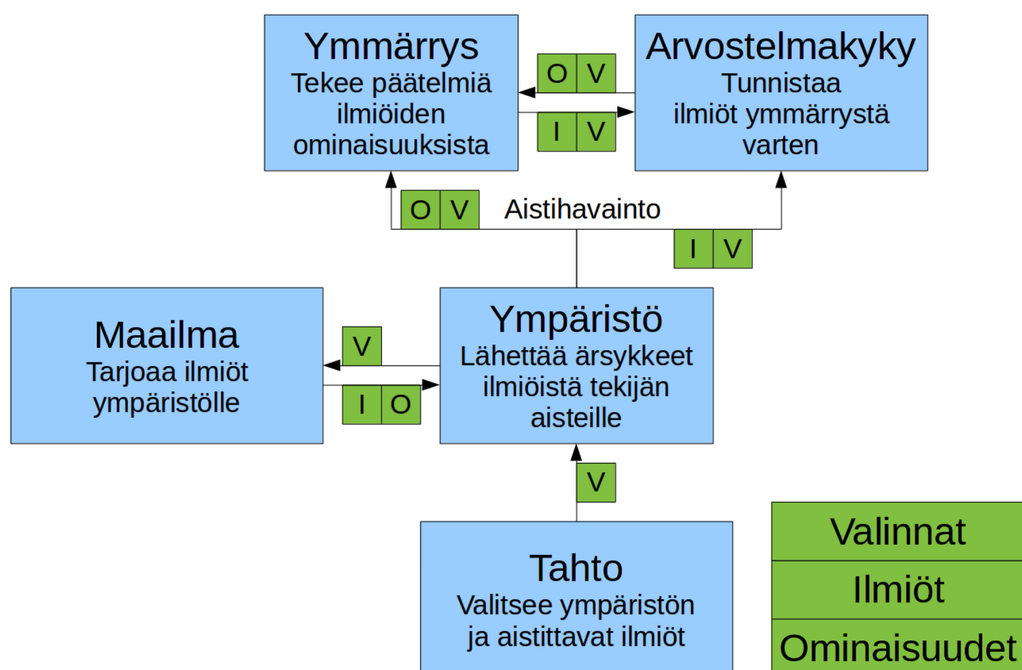
Näkymän kontrolleri valitsee saamiensa syötteiden perusteella, mitä toimintoja näkymässä suoritetaan, pyytää toimintojen vaatimat objektit ja muuttujat tietopohjan tarjoaman rajapinnan kautta ja suorittaa toiminnot tunnistusmoduulien ja toteutusfunktioiden avulla. Objektit täytyy tunnistaa joka toiminnon yhteydessä, mutta mikäli toiminnot käyttävät vain yksittäisiä, mihinkään objektiin kuulumattomia muuttujia, kontrolleri voi kutsua suoraan toteutusfunktioita.

Tunnistusmoduuleilla on pääsy objektien muuttujiin, joten se hakee objektista toteutusfunktion vaatimat muuttujat ja välittää ne toteutusfunktiolle. Toteutusfunktio käsittelee muuttujat, ja mikäli muuttujien joukossa on uusia objekteja, funktio voi kutsua toisia tunnistusmoduulien rajapintoja objekteille.



Kuva 26: Ohjelman suorituksen kulku alkaa syötteenhallinnasta.

Reaalimaailmassa ohjelman suoritusta vastaa havaintokokemuksen kulku ajassa (kuva 27). Havaitsija valitsee tahtonsa avulla ympäristön ja ympäristössä tehtävät toiminnot, minkä jälkeen ympäristö välittää ilmiöt havaitsijan tunnistettavaksi ja ymmärrettäväksi.



Kuva 27: Havaintokokemuksen kulku ajassa.

5 Havainnoivan ohjelman toteutus

Yhtäkään käytössä olevaa ohjelmointikieltä ei ole suunniteltu havainnoivaa ohjelmointia ajatellen, mikä asettaa haasteita paradigman toteutukseen käytännössä. Paradigma on kuitenkin mahdollista toteuttaa nykyisillä ohjelmointikielillä vähintäänkin tyydyttävästi.

Osa ohjelmointikielistä soveltuu paremmin havainnoivaan paradigmaan kuin toiset. Tässä luvussa käydään läpi, mitä ominaisuuksia ohjelmointikieleltä vaaditaan ja mitkä ominaisuudet ovat toivottuja mutta eivät välttämättömiä. Lisäksi osiossa käydään läpi muutamia esimerkkejä C++:lla kirjoitettuna.

5.1 Ohjelmointikielen vaatimukset

Havainnoiva ohjelmointi vaatii ohjelmointikieleltä tiettyjä ominaisuuksia, jotta sen pystyy toteuttamaan mahdollisimman hyvin. Näitä ovat erityisesti funktioiden ylikuormitus (function overloading) ja ystäväluokat (friend classes). Paradigman toteutus on mahdollista myös ilman näitä ominaisuuksia, mutta se vaatii kompromisseja joko datan turvallisuudessa tai koodin luettavuudessa.

Funktioiden ylikuormitusta tarvitaan erityisesti tunnistusmoduulien toteutuksessa. Sen avulla erilaisia tietorakenteita on mahdollista tunnistaa erityyppisten parametrien kautta. Jos ohjelmassa on esimerkiksi mahdollista piirtää kissa ja hiiri, tunnistusmoduuli voi sisältää funktiot ”piirra(Kissa)” ja ”piirra(Hiiri)”. Mikäli funktioiden ylikuormitusta ei ole, objektin tyyppi täytyy tarkistaa jollakin muulla tavalla, esimerkiksi ehtolauseella, joka tarkistaa objektin luokan.

Myös ystäväluokkia tarvitaan nimenomaan tunnistusmoduuleissa. Sen avulla sekä toteutusfunktiot että objektien muuttujat on mahdollista piilottaa kaikilta muilta paitsi halutuilta tunnistusmoduuleilta. Mikäli ystäväluokkia ei ole, objektien muuttujien piilotuksen voi toteuttaa esimerkiksi niin, että objektien tunnistus on toteutettu objektiluokassa itsessään erillisten moduulien sijaan.

Tämän toteutuksen huono puoli on se, että tunnistus on silloin rajattu yhteen objektiin kerrallaan, ja sellaisia funktioita, jotka käyttävät useamman kuin yhden objektien muuttujia toteutuksessaan, täytyy kutsua ketjuttamalla objektien tunnistusfunktioita.

Tällainen funktio voisi olla esimerkiksi piirtofunktio, joka tarvitsee toteutuksessaan piirrettävän objektin sekä piirtävän kameran muuttujia. Tunnistus, joka voitaisiin muuten tehdä yhdellä kutsulla ”piirra(Kamera, Kissa)”, täytyy siis ketjuttaa esimerkiksi seuraavasti: ”Kissa->piirra(Kamera)”, jonka sisällä kutsutaan ”Kamera->piirra(this->kuva, this->sijainti)”.

5.2 Toteutusesimerkkejä

Havainnoivan ohjelmoinnin ajatus, että maailma jakautuu havaitsijan ulkopuoliseen ilmiöistä koostuvaan todellisuuteen ja havaitsijan sisäisiin ilmiöitä käsitteleviin aisteihin ja ymmärrykseen, näkyy vahvasti sen toteutuksessa. Havaitsijan ulkopuolinen todellisuus ja havaitsija koostuvat toisiaan vastaavasta hierarkkisesta rakenteesta. Jokaista todellisuuden fyysistä tasoa vastaa looginen taso (taulukko 1).

Todellisuus	
Fyysinen	**Looginen**
Maailma	Havaitsija
Ympäristö	Tavoitteet
Ilmiö	Arvostelmakyky
Tyyppi	Ymmärrys
Ominaisuus	Päätelmä

Taulukko 1: Todellisuuden jakautuminen fyysiseen ja loogiseen todellisuuteen.

Ohjelman rakenteessa jako muodostuu tietorakenteen ja suoritusrakenteen välille (taulukko 2). Tietorakenne sisältää muistissa olevaa dataa (mm. muuttujia ja objekteja) ja suoritusrakenne dataa käsitteleviä rakenteita (mm. lauseita ja funktioita).

Ohjelma		
Tietorakenne	**Suoritusrakenne**	**Taso**
Tietopohja	Käyttäjä	5
Näkymä	Syötteenhallinta	4
Objekti	Tunnistus	3
Tietue	Toteutus	2
Muuttuja	Lause	1

Taulukko 2: Ohjelman jakautuminen tietorakenteeseen ja suoritusrakenteeseen.

Rakenteet toimivat pareittain: lauseet käsittelevät muuttujia, toteutusfunktiot tietueita, tunnistusmoduulit objekteja ja niin edelleen. Ohjelman ei tarvitse käyttää kaikkia näitä rakenteita, vaan niistä on tarpeen mukaan mahdollista valita ne, jotka sopivat ohjelman käyttötarkoitukseen ja kokoon parhaiten. Esimerkiksi hyvin pienessä opetukseen tarkoitetussa esimerkkiohjelmassa voidaan käyttää pelkästään muuttujia ja lauseita ilman ainoatakaan tietuetta tai toteutusfunktiota. Useimmissa ohjelmissa rakenteiden käyttö on kuitenkin suotavaa.

Ohjelmat voidaan jakaa viiteen tasoon sen mukaan, mitä rakenteita ohjelmassa käytetään. Seuraavaksi esitellyissä esimerkeissä tasot on otettu käyttöön suoraan hierarkkisessa järjestyksessä, mutta kehittäjän on mahdollista valita haluamansa rakenteet myös jossakin muussa järjestyksessä, mikäli se on ohjelman rakenteen kannalta järkevää.

5.3 Taso 1: Muuttujat ja lauseet

Tason 1 rakenteet sisältävät ainoastaan muuttujia ja lauseita. Tällainen ohjelma soveltuu hyvin pieniin esimerkkeihin ja ohjelmoinnin alkeita käsittelevään opetukseen (esimerkkikoodi 1). Esimerkkinä voisi olla ohjelma, joka tulostaa käyttäjän syötteen (nk. ”Hello World”), tai ohjelma, joka laskee yhden yksinkertaisen laskun, kuten karkauspäivän tai painoindeksin.

Tason 1 ohjelma saa sisältää vain muuttujia, joita ei käytetä ohjelman sisällä yhdessä muuten kuin satunnaisesti (ts. joiden yhdistäminen tietueiksi ei toisi rakenteen kannalta merkittävää etua). Ohjelma voi sisältää apufunktioita, joilla on mahdollista vähentää koodin toistoa tai selventää ohjelman rakennetta.

Rakenne	**Sisältää**	**Vastuut**
Muuttuja	Dataa	Datan formaatti ja sisältö.
Lause	Komennon	Datan käsittely.

```
int main()
{
  // Muuttujat
  std::string teksti;
  // Lauseet
  getline(std::cin, teksti);
  std::cout << teksti;
}
```

Esimerkkikoodi 1: Hello World -ohjelma ei vaadi ympärilleen monimutkaisia rakenteita.

5.4 Taso 2: Tietueet ja toteutusfunktiot

Tason 2 rakenteissa muuttujat tai osa muuttujista on jaettu tietueisiin siten, että usein yhdessä käytettävät muuttujat yhdistetään toisiinsa tietueiden avulla. Tietueet tulee muodostaa siten, että ne sisältävät ainoastaan funktioiden kannalta toisissaan tiukasti kiinni olevia muuttujia. On oletettava, että kun funktio saa parametrinaan tietueen, tuon tietueen jokaista tai lähes jokaista muuttujaa käsitellään funktiossa aina (esimerkkikoodi 2).

Esimerkkinä tietueesta on sijainti. Sijainti sisältää muuttujina koordinaatit x ja y. Aina kun funktio saa parametrinaan sijainnin, tulee olettaa, että funktio käyttää tietueesta sekä x- että y-koordinaattia. Tietueeseen ei tule lisätä esimerkiksi tuohon x- ja y-koordinaattiin liittyvää tekstimuuttujaa, koska vaikka nämä muuttujat liittyvätkin toisiinsa, niitä ei välttämättä käsitellä samoissa funktioissa. Tietueet pyrkivät siis kuvaamaan ilmiöiden yksittäisiä toiminnallisia kokonaisuuksia, mutta niiden tarkoitus ei ole sisältää ilmiöiden kaikkia ominaisuuksia.

Kun tietue on kerran otettu käyttöön tuotannossa, sen muuttujia ei saa sen jälkeen enää poistaa tai muuttaa. Mikäli myöhemmin esiintyy tarvetta samankaltaiselle mutta hieman erilaiselle tietueelle, se täytyy tehdä alkuperäisen rinnalle. Alkuperäiseen tietueeseen voi lisätä muuttujia sillä oletuksella, että lähes kaikkien muuttujien tulee edelleen olla tarpeellisia kaikissa tietuetta käsittelevissä funktioissa.

Kun puhutaan ”lähes kaikista muuttujista”, kyseessä on lopulta ohjelmiston kirjoittajan valinta, mutta hyvän tavan mukaisesti vähintään 4/5 tietueen muuttujista tulisi olla käytössä, mikäli tietue annetaan kokonaisuudessaan funktiolle parametrina. Muussa tapauksessa on suositeltavaa muokata funktiota niin, että muuttujat annetaan yksitellen tai harkita tietueen uudelleensuunnittelua, jos sitä ei ole otettu vielä käyttöön.

Rakenne	**Sisältää**	**Vastuut**
Tietue	Muuttujia	Muuttujien määritys ja oletusarvojen asetus.
Funktio	Lauseita	Tietueiden ja ominaisuuksien käsittely.

```
//
// Tietue
struct sijainti
{
  // Muuttujat
  int x;
  int y;
};


//
// Toteutusfunktio
void siirra(sijainti* _sijainti, int _dx, int _dy)
{
  // Lauseet
  _sijainti->x += _dx;
  _sijainti->y += _dy;
}
```

Esimerkkikoodi 2: Sijaintitietue annetaan siirtofunktiolle parametrina kokonaisuudessaan, koska funktio käyttää tietueen jokaista muuttujaa. Tämä lyhentää parametrilistoja.

5.5 Taso 3: Objektit ja tunnistusmoduulit

Tason 3 rakenteet sisältävät tunnistusmoduuleita sekä objekteja, jotka on muodostettu erilaisista tietueista (esimerkkikoodi 3). Siinä missä tietueet pyrkivät kuvaamaan ilmiöitä toiminnallisuuden kannalta (mitä tämä tekee), objektit pyrkivät kuvaamaan ilmiöitä siten kuin me tunnistamme ne ja erotamme ne toisista ilmiöistä (mikä tämä on). Taso 3 on ensimmäinen taso, jossa käytetään havainnoivan ohjelmoinnin periaatetta datan ja funktioiden välisestä rajapinnasta.

Esimerkkinä objektista on piste. Piste sisältää tason 2 esimerkissä esitetyn sijainti-nimisen tietueen ja sen lisäksi toisen tietueen, joka sisältää pisteeseen liittyvää tekstitietoa. Objekti ei koskaan sisällä muuttujia suoraan vaan aina tietueiden kautta.

Objekteja käsitellään tunnistusmoduuleissa, joille annetaan friend class -ominaisuuden avulla pääsy objektien sisäisiin tietueisiin ja tietueiden muuttujiin. Moduulien ainoa tarkoitus on tunnistaa objektit ja kutsua objekteihin ja toimintoihin sopivia toteutusfunktiota. Tunnistusmoduuli ei saa itse käsitellä muuttujia, vaan se ainoastaan välittää ne toteutusfunktiolle. Tunnistusmoduuli voi tarkistaa muuttujien arvoja ja luoda väliaikaisia listoja objektien tietueista, mikäli toteutus parametreinaan sellaisia vaatii. Se ei kuitenkaan saa muokata muuttujien arvoja, tallentaa niitä pysyvästi tai tulostaa mitään tietoa käyttäjälle esimerkiksi näytön, lokitiedostojen tai äänilaitteiden kautta.

Käytettäessä tunnistusmoduuleita toteutusfunktiot on mahdollista yhdistää luokiksi. Tällöin funktiot voi asettaa yksityisiksi luokkiensa sisällä, jolloin on mahdollista rajoittaa tunnistusmoduulien pääsyä funktioihin friend class -ominaisuuden avulla. Tämä on erityisen hyvä käytäntö varsinkin suurissa ohjelmistoissa, sillä tunnistusmoduulilla täytyy siinä tapauksessa olla oikeus sekä objektin tietosisältöön että toteuttavan funktion suoritukseen, mikä lisää turvallisuutta.

Toteutusluokat voivat sisältää myös tunnistusmoduuleita, joita voidaan käyttää parametreina saatujen objektien käsittelyyn. Toteutusten sisällä olevia tunnistusmoduuleita ei kuitenkaan saa koskaan luoda toteutusluokkien sisällä, koska se tulisi aiheuttamaan ennemmin tai myöhemmin loputtoman alustusten kierteen toteutusten ja tunnistusten välillä. Tunnistusmoduulit täytyy siis tuoda aina valmiiksi luotuna toteutusten ulkopuolelta. Tämä onnistuu helpoiten käyttäjien (kts. luku 5.7) avulla.

Rakenne	**Sisältää**	**Vastuut**
Objekti	Tietueita	Tietueiden luonti ja oletus-arvojen asetus.
Tunnistusmoduuli	Toteutusfunktioiden luokkia	Objektien tunnistus ja oikeiden toteutusfunktioiden kutsu.

```
//
// Objekti
class Piste
{
  // Annetaan oikeus tunnistusmoduulille
  friend class Muokkausmoduuli;

  private:
  // Tietueet
  sijainti m_sijainti;
  tieto m_tieto;
};


//
// Tunnistusmoduuli
class Muokkausmoduuli
{
  // Toteutusfunktiot voidaan jakaa luokkiin käytön rajoittamiseksi
  Siirra* m_siirra;
```

```
  public:
  // Konstruktori
  Muokkausmoduuli(Siirra* _siirra)
  {
    m_siirra = _siirra;
  }

  // Tunnistusfunktio
  void siirra(Piste* _piste, int _dx, int _dy)
  {
    m_siirra->siirra(&_piste->m_sijainti, _dx, _dy);
  }
};


//
// Toteutusluokka
class Siirra
{
  // Annetaan oikeus tunnistusluokalle
  friend class Muokkausmoduuli;

  private:
  // Toteutusfunktio
  void siirra(sijainti* _sijainti, int _dx, int _dy)
  {
    // Lauseet
    _sijainti->x += _dx;
    _sijainti->y += _dy;
  }
};
```

Esimerkkikoodi 3: Piste-luokka sisältää piilotettuja tietueita, jotka näytetään ainoastaan Muokkausmoduulille. Muokkausmoduuli ei kuitenkaan käsittele tietueita itse, vaan se ainoastaan välittää ne eteenpäin toteutusfunktioille.

5.6 Taso 4: Näkymät ja syötteenhallinta

Tason 4 rakenteiden tarkoitus on yhdistää yhdessä käytettyjä objekteja näkymiksi ja tarjota näkymiin liittyvä syötteenhallinta (esimerkkikoodi 4). Näkymissä tehdään näyttöön tulostuksen lisäksi kaikki muukin toiminta, joka näkymään liittyy. Näkymien tehtävä onkin määritellä, mitä kaikkia toimintoja kussakin käyttäjälle näkyvässä näkymässä suoritetaan.

Näkymät koostuvat objekteista ja tunnistusmoduuleista sekä kontrollerifunktiosta, joka suorittaa näkymän toiminnot tunnistusmoduulien ja objektien avulla. Sekä objektit että moduulit tuodaan näkymällä ulkopuolelta, ja vain kontrollerifunktio luodaan näkymän sisällä.

Syötteenhallinnan tehtävä on hallita ohjelman ulkopuolelta ohjelmaan tulevia syötteitä. Syötteiden antaja voi olla käyttäjä mutta myös esimerkiksi toinen ohjelma. Syötteenhallinta vastaanottaa syötteet, käsittelee ne muun ohjelman ymmärtämään muotoon ja välittää ne näkymälle.

Rakenne	**Sisältää**	**Vastuut**
Näkymä	Näkymään liittyvät objektit ja objektien käsittelyyn tarvittavat moduulit.	Näkymään liittyvien toimintojen suoritus moduulien avulla.
Syötteenhallinta	Syötteenhallintaan liittyvät toiminnot.	Syötteiden vastaanotto ja välittäminen näkymälle.

```
//
// Syötteenhallinta
class Syotteenhallinta
{
  public:
  // Suorita
  void suorita(Listanakyma* _lista, Muokkausnakyma* _muokkaus)
  {
    int nakyma = 1;

    while(nakyma > 0)
      {
        // Hae syötteet
        std::cin >> nakyma;

        // Valitse näkymä
        if(nakyma == 1)
          {
            _lista->suorita();
          }
        else if(nakyma == 2)
          {
            _muokkaus->suorita();
          }
      }
  }
};


//
// Muokkausnäkymä
class Muokkausnakymä : public Nakyma
{
  private:
  // Objektit
  Piste* m_piste;

  // Moduulit
  Muokkausmoduuli* m_muokkausmoduuli;
```

```
  public:
  // Konstruktori
  Muokkausnakyma(Piste* _piste)
  {
    m_piste = _piste;
  }

  // Suorita
  void suorita()
  {
    m_muokkausmoduuli->siirra(m_piste, 10, 10);
  }

  // Aseta moduuli
  void aseta_moduuli(Muokkausmoduuli* _muokkausmoduuli)
  {
    m_muokkausmoduuli = _muokkausmoduuli;
  }
};
```

Esimerkkikoodi 4: Ohjelman suoritus alkaa syötteenhallinnasta, joka ottaa vastaan käyttäjän syötteet ja kutsuu valitun näkymän suoritusfunktiota. Näkymät sisältävät kaikki näkymässä tarvittavat objektit ja toimintojen vaatimat moduulit sekä suoritusfunktion, joka määrittelee sen, mitä näkymässä tehdään (tässä esimerkissä piste siirretään, mutta ruudulle ei tulosteta mitään).

5.7 Taso 5: Tietopohja ja käyttäjä

Tason 5 rakenteiden tarkoitus on tarjota yhtenäinen pohja sekä tietorakenteelle että suoritusrakenteelle. Taso koostuu tietopohjasta ja käyttäjästä (esimerkkikoodi 5).

Tietopohja sisältää kaikki ohjelman jaetut objektit ja ohjelman näkymät. Tietopohjan vastuulla on objektien haku tietokannasta tai muusta tietolähteestä, objektien luonti sekä ohjelman eri näkymien määritys.

Käyttäjä sisältää kaikki ohjelman tunnistusmoduulit ja toteutusfunktiot sekä syötteenhallinnan, joka voi sisältää esimerkiksi tiedon tämänhetkisestä valitusta näkymästä. Käyttäjän tehtävä on luoda syötteenhallintayksikkö ja tunnistusmoduulit.

Rakenne	**Sisältää**	**Vastuut**
Tietopohja	Kaikki ohjelman jaetut objektit ja ohjelman näkymät.	Tietokantayhteys, objektien haku ja luonti, ohjelman eri näkymien määritys.
Käyttäjä	Tunnistusmoduulit, toteutusfunktiot ja syötteenhallinta.	Syötteidenhallinnan ja moduulien määritys.

```
//
// Tietopohja
class Tietopohja
{
  private:
  // Objektit
  Piste* m_piste;

  public:
  // Näkymät
  Listanakyma* m_listanakyma;
  Muokkausnakyma* m_muokkausnakyma;


  // Konstruktori
  Tietopohja()
  {
    m_piste = new Piste();

    m_listanakyma = new Listanakyma(m_piste);
    m_muokkausnakyma = new Muokkausnakyma(m_piste);
  }

  // Aseta moduulit
  void aseta_moduulit(Tulostusmoduuli* _tulostus, Muokkausmoduuli*
  _muokkaus)
  {
    m_listanakyma->aseta_moduuli(_tulostus);
    m_muokkausnakyma->aseta_moduuli(_muokkaus);
  }
}


//
// Käyttäjä
class Kayttaja
{
  private:
  // Syötteenhallinta
  Syotteenhallinta* m_syotteenhallinta;

  // Tunnistusmoduulit
  Tulostusmoduuli* m_tulostusmoduuli;
  Muokkausmoduuli* m_muokkausmoduuli;


  public:
  // Konstruktori
  Kayttaja(Tietopohja* _tietopohja)
  {
    Tulosta* tulosta = new Tulosta();
    Siirra* siirra = new Siirra();

    m_tulostusmoduuli = new Tulostusmoduuli(tulosta);
    m_muokkausmoduuli = new Muokkausmoduuli(siirra);

    _tietopohja->aseta_moduulit(m_tulostusmoduuli, m_muokkausmoduuli);
  }
```

```
  // Suorita
  void suorita(Tietopohja* _tietopohja)
  {
    m_syotteenhallinta->suorita(_tietopohja->m_listanakyma,
    _tietopohja->m_muokkausnakyma);
  }
}
```

Esimerkkikoodi 5: Tietopohjan tehtävä on sisältää kaikki ohjelman objektit ja muuttujat joko suoraan tai esimerkiksi tietokantayhteyden kautta. Käyttäjän tehtävä on sisältää ohjelman syötteenhallinta sekä kaikki ohjelman tunnistusmoduulit ja toteutusfunktiot.

5.8 Havainnoiva ohjelmointi opetuskäytössä

Havainnoiva ohjelmointi soveltuu erityisesti opetuskäyttöön todella hyvin. Koska paradigman avulla on mahdollista toteuttaa ohjelmia eritasoisina ohjelman koosta ja monimutkaisuudesta riippuen, sitä voidaan soveltaa opetusympäristössä eritasoisille oppilaille ja tehtäville.

Esimerkiksi yksinkertainen Hello World -ohjelma tai karkausvuosilaskuri voidaan toteuttaa tason 1 ohjelmana ja hieman monimutkaisempi nelikulmion geometrisia ominaisuuksia laskeva ohjelma tason 2 ohjelmana. Erityisen hyvä ominaisuus eritasoisissa ohjelmissa on se, että tasojen rakenteet eivät muutu oleellisesti korkeammalle tasolle mentäessä, vaan tasolla 2 toteutetut funktiot ovat täysin käyttökelpoisia myös tasoilla 4 ja 5. Tämän vuoksi paradigman avulla on mahdollista suunnitella opiskeluaikana toteutettavia monimutkaisia ohjelmia, joiden yksinkertaisimpien funktioiden toteutuksia voi alkaa kirjoittaa jo hyvin varhaisessa vaiheessa.

Oppilaiden kirjoittamia funktioiden toteutuksia on myös mahdollista jakaa ja käyttää uudelleen muiden oppilaiden projekteissa. Tämä mahdollistaa aivan uuden, sosiaalisen ohjelmointiopetuksen. Jakaminen onnistuu esimerkiksi jaetulla tiedostopalvelimella, johon yksittäiset funktiot tallennetaan tiedostoina. Proseduraalinen ohjelmointi on tällaiseen jakamiseen liian arvaamaton, koska se ei sisällä yhtenäistä rajapintaa, joka mahdollistaisi erilaisten (eri oppilaiden kirjoittamien) funktioiden käytön samalla rajapinnalla. Olio-ohjelmoinnin luokat taas ovat usein liian suuria kokonaisuuksia ollakseen riippumattomia ohjelman muusta rakenteesta ja täten jaettavissa itsenäisesti. Havainnoivan ohjelmoinnin toteutusfunktiot ovat kuitenkin olio-ohjelmoinnin luokkia

huomattavasti pienempiä, mutta niitä käytetään siitä huolimatta rajapintojen kautta, joten ne soveltuvat erinomaisesti jaettaviksi ja uudelleenkäytettäviksi.

Havainnoivan ohjelmoinnin heikko puoli opetuksessa on se, että sitä ei ole vielä mahdollista toteuttaa täydellisesti kovinkaan monella ohjelmointikielellä. Sitä on sen vuoksi vaikea perustella pääasiallisena ohjelmointiparadigmana opetuksessa. Mikäli oppilaitos kuitenkin käyttää pääasiallisena opetuskielenään C++:aa tai jotain muuta havainnoivaa ohjelmointia täysin tukevaa kieltä, on se erittäin harkitsemisen arvoinen vaihtoehto muille ohjelmointiparadigmoille.

6 Lopuksi

Tässä työssä esiteltiin, miten tunnetut filosofiset teoriat on mahdollista yhdistää loogisesti tunnettuihin ohjelmointiparadigmoihin. Lisäksi työssä osoitettiin filosofisten teorioiden ja ohjelmointiparadigmojen yhdistämisen toimivuus soveltamalla sitä Immanuel Kantin transsendentaaliseen idealismiin ja esittelemällä siihen perustuva havainnoivan ohjelmoinnin teoria. Tämän jälkeen teorian toimivuus osoitettiin esittelemällä tapa, jolla paradigma on toteutettavissa nykyisillä ohjelmointikielillä.

Työn lähtökohta oli rohkea ja kunnianhimoinen, mutta tulokset olivat sitäkin hedelmällisemmät. Yhteys filosofisten teorioiden ja ohjelmointiparadigmojen välillä avasi tulevaisuutta varten lukuisia mahdollisuuksia myös muiden filosofisten teorioiden hyödyntämiseen ohjelmointiparadigmojen analysoinnissa ja kehityksessä. Ohjelmointi on alana edelleen nuori, mutta se sai työn myötä tuhansia vuosia pitkät juuret. Lisäksi Kantin transsendentaaliseen idealismiin pohjautuva havainnoiva ohjelmointi toi mukanaan uuden tavan luoda etenkin monimutkaisia ohjelmia, joita funktionaalinen, proseduraalinen tai olio-ohjelmointi eivät välttämättä pysty täydellisesti kuvaamaan.

Aiheen haastavuuden ja teoreettisuuden vuoksi työn aikatauluttaminen oli erittäin vaikeaa, ja työn valmistuminen kestikin toivottua pidempään. Erityisesti käsitteiden määritteleminen ja niistä selkeiden kokonaisuuksien muodostaminen osoittautui todella aikaavieväksi ja hankalaksi. Aihepiiri laajeni myös huomattavasti alkuperäistä suuremmaksi pelkästä ohjelmointiparadigman määrittelystä kokonaisen ohjelman toteutuksen ja filosofisen pohjan kuvaukseen, mikä oli lopputuloksen kannalta positiivinen asia, mutta vaikutti työn valmistumisaikaan merkittävästi. Lisäksi, vaikka havainnoivan ohjelmoinnin toteutus onkin käytännössä toimiva, se jäi edelleen kaipaamaan yksinkertaistamista ja selostuksia, jotta sen ymmärtäminen ja käyttöönotto olisi helpompaa. Kokonaisuutta ajatellen nämä ovat kuitenkin vain haasteita, jotka odottavat ratkaisuaan tulevaisuudessa.

Lähteet

1 Balaguer, Mark. 2009. Platonism in Metaphysics. Verkkodokumentti. Stanford Encyclopedia of Philosophy. <http://plato.stanford.edu/entries/platonism>. Päivitetty 7.4.2009. Luettu 20.4.2015.

2 Comparison of programming languages. 2015. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Comparison_of_programming_languages>. Päivitetty 17.4.2015. Luettu 30.4.2015.

3 Descartes, René. 2005. Metodin esitys. Project Gutenberg. <https://www.gutenberg.org/ebooks/15085>. Luettu 20.4.2015.

4 Functional programming. 2014. Verkkodokumentti. Haskell Wiki. <https://wiki.haskell.org/Functional_programming>. Päivitetty 24.12.2014. Luettu 27.4.2015.

5 Harrison, William. 1993. Subject-Oriented Programming. New York: IBM Thomas J. Watson Research Center.

6 Higgins, Francis. Gorgias (483—375 B.C.E.). Verkkodokumentti. Internet Encyclopedia of Philosophy. <http://www.iep.utm.edu/gorgias>. Luettu 20.4.2015.

7 van Inwagen, Peter. 2014. Metaphysics. Verkkodokumentti. Stanford Encyclopedia of Philosophy. <http://plato.stanford.edu/entries/metaphysics>. Päivitetty 31.10.2014. Luettu 29.4.2015.

8 Kant, Immanuel. 2013. Puhtaan järjen kritiikki. Helsinki: Gaudeamus Oy.

9 Mulder, Dwayne. Objectivity. Verkkodokumentti. Internet Encyclopedia of Philosophy. <http://www.iep.utm.edu/objectiv>. Luettu 27.4.2015.

10 Robinson, Howard. 2011. Dualism. Verkkodokumentti. Stanford Encyclopedia of Philosophy. <http://plato.stanford.edu/entries/dualism>. Päivitetty 3.11.2011. Luettu 27.4.2015.

11 Sannikka, Juhani. 2005. Aspektisuuntautunut ohjelmointi ja ohjelmiston modularisointi. Erikoistyö. Kuopion yliopisto.

12 Seibel, Peter. 2009. Coders at Work. New York: Springer Science+Business Media.

13 Subjectivism. 2014. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/wiki/Subjectivism>. Päivitetty 5.11.2014. Luettu 27.4.2015.

14 Thornton, Stephen. Solipsism and the Problem of Other Minds. Verkkodokumentti. Internet Encyclopedia of Philosophy. <http://www.iep.utm.edu/solipsis>. Luettu 27.4.2015.